The Design and Implementation of a 3D Graphics Pipeline for the Raw Reconfigurable Architecture

by

Kenneth William Taylor

B.S., Computer and Systems Engineering and Computer Science
Rensselaer Polytechnic Institute, 2002

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2004

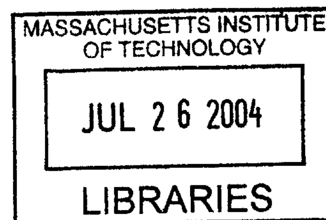
© Kenneth William Taylor, MMIV. All rights reserved.

The author hereby grants to MIT permission to reproduce and distribute publicly paper and electronic copies of this thesis document in whole or in part.

Author ....
Department of Electrical Engineering and Computer Science
May 20, 2004

Certified by ....
Anant Agarwal
Associate Professor, CSAIL
Thesis Supervisor

Accepted by ....
Arthur C. Smith
Chairman, Department Committee on Graduate Students


MASSACHUSETTS INSTITUTE OF TECHNOLOGY
JUL 26 2004
LIBRARIES

BARKER

The Design and Implementation of a 3D Graphics Pipeline for the Raw Reconfigurable Architecture

by
Kenneth William Taylor

Submitted to the Department of Electrical Engineering and Computer Science
on May 20, 2004, in partial fulfillment of the
requirements for the degree of
Master of Science in Electrical Engineering and Computer Science

Abstract

This thesis presents the design and implementation of a 3D graphics pipeline, built on top of the "Raw" processor developed at MIT. The Raw processor consists of a tiled array of CPUs, caches, and routing processors connected by several high-speed networks, and can be treated as a coarse-grained reconfigurable architecture. The graphics pipeline has four stages, and four-way parallelism in each stage, and is mapped on to a 16-tile Raw array. It supports basic rendering functions such as hardware transform and lighting, perspective correct texture mapping, and depth buffering, and is intended to be used as a slave processor receiving rendering commands from a host system. The design process is described in detail, along with difficulties encountered along the way, and a comprehensive performance evaluation is carried out. The paper concludes with many suggestions for architectural and performance improvements to be made over the initial design.

Thesis Supervisor: Anant Agarwal
Title: Associate Professor, CSAIL

Acknowledgments

I would like to thank my thesis advisor, Anant Agarwal, for first introducing me to the Raw architecture and all the interesting thesis ideas revolving around that rich project. I also thank Anant for being so patient with me in the hours nearing the submission deadline. I would also like to thank the members of the Computer Architecture Group, who either directly or indirectly assisted me greatly on this project: from the innumerable maintainers of the starsearch testing suite and all its amazingly useful examples, to the maintainers of the BTL simulator (notably, Michael Taylor, who personally fixed one bug just for me), to others such as Paul Johnson who set me up with the CAG computer system, on which I did a large amount of the work for this project.

I also tip my hat to Kurt Akeley and Pat Hanrahan, whose online lectures for their Stanford Real-Time Graphics course originally inspired me to do a graphics architecture project, even though I knew very little about the subject to begin with.

Finally, I would like to send many, many thanks to April for her support and sympathy through the last few weeks of this project, and also for spending a large amount of her time helping me proofread.

Contents

1 Introduction and Motivation 15
1.1 Goals . . . 16
1.2 Methodology . . . 16
1.3 Outline of this Thesis . . . 16

2 3D Rendering Algorithms 19
2.1 Introduction to 3D Rendering in General . . . 19
2.1.1 Defining a Scene . . . 20
2.1.2 Geometry Transformation . . . 20
2.1.3 Rasterization and Interpolation . . . 23
2.1.4 Rendering to the Framebuffer . . . 26
2.2 3D Rendering Implementation of this Project . . . 27
2.2.1 Geometry Transformation . . . 27
2.2.2 Rasterization and Interpolation . . . 29
2.2.3 Texture Mapping and Blending . . . 32
2.2.4 Rendering to the Framebuffer . . . 33
2.3 Summary . . . 33

3 Designing the Parallel Rendering Architecture 35
3.1 Design Goals . . . 35
3.2 Basic Parallelization . . . 36
3.2.1 Graphics Pipelines . . . 36
3.2.2 Mapping Parallel Pipelines to the Raw Architecture . . . 39
3.3 Difficult Design Problems . . . 39
3.3.1 Distributing Primitives Across the Pipelines . . . 41
3.3.1.1 Communication in the Raw Architecture . . . 41
3.3.1.2 Choosing a Network for Commands . . . 42
3.3.1.3 Techniques for Using the Static Network Dynamically . . . 42
3.3.1.4 Developing the Distribution Algorithm on the Switch . . . 43
3.3.1.5 Improving on Round Robin Distribution . . . 45
3.3.2 Maintaining Sequential Correctness . . . 46
3.3.2.1 Dealing With Sequence Number Rollover . . . 48
3.3.3 Synchronization in the Compositing Stage . . . 50
3.3.3.1 Attempts at Cache Coherence . . . 50
3.3.3.2 A Simpler Approach . . . 51
3.3.4 Other Difficult Issues . . . 52
3.3.4.1 Asynchronously Reseting the Processor . . . 52

3.3.4.2 Switching Modes and Flushing the Pipeline 52
3.4 Summary 53

4 Implementation of the 3D Processor **55**
4.1 Overview of Architectural Organization 55
4.2 Detailed Description of Implementation 58
4.2.1 The Boot Sequence 58
4.2.2 Command Mode 58
4.2.2.1 Render State Commands 62
4.2.2.2 Texture Management Commands 62
4.2.2.3 Depth and Framebuffer Commands 63
4.2.2.4 Miscellaneous Commands 63
4.2.3 Scene Streaming Mode 63
4.2.3.1 Distributing Commands Among the Pipelines 64
4.2.3.2 Stage 1 65
4.2.3.3 Stage 2 68
4.2.3.4 Stage 3 69
4.2.3.5 Stage 4 69
4.3 Interfacing the Graphics Architecture in a System 70
4.3.1 The Framebuffer Side 70
4.3.2 The Host Side 72
4.4 Summary 72

5 Testing, Validation and Performance Results **73**
5.1 The Testing Framework 73
5.1.1 The BTL Simulation Environment 73
5.1.2 Framebuffer Controller 74
5.1.3 Render Host Interface 76
5.1.3.1 External Program Control 76
5.1.3.2 Performance Profiling 77
5.2 Feature Validation 78
5.3 Performance Results 79
5.3.1 Performance Model 80
5.3.2 Empirical Performance Results 82
5.4 Summary 87

6 Improvements and Suggestions for Future Work **89**
6.1 Improving Parallelization Efficiency 89
6.1.1 Reducing Pipelining Overhead 89
6.1.2 Reducing Parallelization Overhead 90
6.1.3 Improving Load Balancing 91
6.1.4 Improving Parallelization Bottlenecks 92
6.2 Improving Raw Performance 93
6.3 Scaling the Graphics Architecture 94
6.4 Ways to Extend This Thesis 94
6.5 Summary 95

7 Conclusion **97**

A Additional Rendered Images 99

B Single-Tile Code Listing 103

B.1 SingleTile.c 103

C Full Implementation Code Listing 151

C.1 Common-sw.h 151
C.2 Common-sw.S 152
C.3 render_datatypes.h 153
C.4 Stage1-datatypes.h 160
C.5 ZBuf_datatypes.h 164
C.6 render_cmds.h 165
C.7 Stage1-Main.c 169
C.8 Stage1-Main-sw.S 189
C.9 Stage1-Aux.c 190
C.10 Stage1-Common.c 192
C.11 Stage1-sw.S 210
C.12 Stage1-sw-0.S 212
C.13 Stage1-sw-1.S 217
C.14 Stage1-sw-2.S 222
C.15 Stage1-sw-3.S 227
C.16 Stage2-Common.c 230
C.17 Stage2-sw.S 241
C.18 Stage3-Common.c 242
C.19 Stage3-sw.S 255
C.20 Stage4-Common.c 256
C.21 Stage4-sw.S 267

D Verification Framework Code Listing 271

D.1 RenderInterface.bc 271
D.2 render_framebuffer.bc 272
D.3 render_host.bc 284
D.4 render_framebuffer.h 294
D.5 render_client.h 298
D.6 triangletest.c 318
D.7 cubetest.c 327
D.8 texturetest.c 335
D.9 ordertest.c 339

List of Figures

2-1 Orthographic vs. Perspective Projection 21
2-2 Interpolation in Screen Space 24
2-3 Primitives that Intersect the Clipping Plane 28
2-4 Transformation of Normals 29
2-5 Triangle Rasterization to Pixels 30

3-1 Basic Concepts of Pipelining 37
3-2 A Simple 3D Graphics Pipeline 38
3-3 Basic Pipeline Layout on Raw 40
3-4 Problems with Simple Sequencing 47

4-1 Overview of Implementation 57
4-2 State Transition Diagram for a Typical Stage 1 Switch Processor 67
4-3 The Processor in a System Context 71

5-1 Framebuffer Message Header 75
5-2 Sample Screenshots 79
5-3 Triangle Performance Test Pattern 80
5-4 Performance: Speedup, Utilization, and Estimated Overhead 83
5-5 Performance: Utilization per Stage 85
5-6 Performance: Active Cycles per Stage 86
5-7 Performance: Triangles and Pixels per Second 88

List of Tables

2.1 Coordinate Mapping in Different Texture Modes 32

3.1 Sequence Numbers Assigned to a Stream of Primitives 48

4.1 Render State Variables Stored in the First Pipeline Stage 59
4.2 Rendering Processor Commands 61
4.3 Full State Transition Table for Stage 1's Switch Code 66

5.1 Summary of Meta Commands for the RenderHost Interface 77

Chapter 1

Introduction and Motivation

The "Raw" Architecture [21] is a tiled microprocessor array with tightly coupled interprocessor networks, programmable static routing, and direct software access to pin I/O. The architecture can be implemented on a single chip, and supports a wide variety of parallel processing paradigms, such as shared memory, message-passing, and systolic array processing. A Raw Processor is, at its basic level, an array of microprocessors; however, it can also be configured and treated as a deeply pipelined stream processor [8], a wide-issue processor for instruction-level parallelism [22], or a coarse-grained reconfigurable architecture.

This latter use is of particular interest in this paper, and was one of the original motivations for the Raw project. The idea was to create, with Raw, an architecture that could be fine-tuned to different tasks, but which was coarser-grained than FPGAs and therefore on which datapath and other higher-level structures were easier to implement, and perhaps more efficient in terms of performance and space cost, as well.

In this vein, the paper presents an implementation of an architecture for real-time 3D graphics on the Raw processor, hoping that, while not being as efficient as a full-custom design, the processor will attain a large performance improvement over a software-only implementation on a general-purpose CPU. Also, it is hoped that implementing the processor on Raw will prove a much easier engineering task than a full-custom design might be.

A real-time 3D graphics processor is, in general, a co- or slave processor which receives rendering commands from a host processor and outputs a rendered image to a memory known as a "framebuffer." The framebuffer may also be connected to a Video DAC (digital to analog converter) for display on a screen, but this is not strictly necessary. The goal is to offload the chore of rendering 3D images from the main processor so that it may continue to do other useful work, and also to provide a highly optimized datapath for that 3D rendering to attain performance that a general processor would not be able to attain. 3D rendering hardware is used in a wide range of applications, from high-end workstations for scientific and medical imaging to consumer devices such as video game consoles. 3D graphics accelerators have become increasingly popular as consumer-level add-ons to computers, especially for use with video gaming. In this market, the engineering goals include not only high performance, but also low cost and moderate power usage.

The actual process of performing 3D rendering (explained in more detail in Chapter 2) is highly parallelizable due to the typical lack of much dependence between separate parts of a rendered image. Modern graphics processors generally gain large amounts of performance through a combination of pipelining the computation, and running several computations in

parallel. Such a structure might map nicely to a two-dimensional mesh like that of the Raw processor. Additionally, the Raw processor's many modes of communication — streaming messages, dynamic messages, shared-memory — and large amount of I/O bandwidth would prove very useful in a complex multi-featured device such as a 3D rendering processor. For more detail regarding the parallelization of 3D algorithms and how it was applied to the Raw see Chapters 3 and 4.

1.1 Goals

The major goal of this thesis was to design and verify a 3D rendering architecture on a Raw chip from scratch in order to gain insight into the major performance and implementation issues that would come up, especially those specific to the Raw architecture itself. The 3D architecture itself does not need to support state-of-the-art features, but should implement some relatively interesting (if well-known) techniques such as texture mapping, alpha blending, directional lights, and hardware transform and lighting operations. The performance should be reasonable — as in, it should be feasible that the architecture could operate interactively, even if that would require very low resolution or simple scenes. Detailed performance measurements should be possible, to allow for a good deal of insight into the performance issues that come up. It was hoped that by the end of the project, much more would be known about the bottlenecks and performance pitfalls about implementing such an architecture on Raw, and that this knowledge would be useful to advise future endeavors in the area.

1.2 Methodology

The project described in this thesis was implemented solely as a simulation, although it could conceivably be implemented on existing hardware. A framework was developed around the simulation to be able to verify its correct functionality and to measure detailed information about its performance. Several simple tests were run to verify functionality — not a full test suite by far but enough for reasonable confidence that the device was functioning correctly for the most part. One such test was modified to allow it to run over many different combinations of rendering modes and parameters, and output large batches of performance results to the disk to be analyzed later by scripts. These results were distilled down, and are presented in this thesis along with an intuitive performance model to try to pinpoint where the inefficiencies stemmed from. This information is then used to motivate suggestions for future improvement in similar lines of inquiry.

For more details about the testing methods used in this thesis and their results, see Chapter 5.

1.3 Outline of this Thesis

This thesis begins with a general introduction to 3D rendering algorithms in Chapter 2, both to serve as background information for those not familiar with the topic, and to describe the algorithms used in this project so that the rest of the paper can concentrate on the architectural features of the design. Chapter 3 talks about the design process of trying to parallelize those rendering algorithms on to the Raw, including several hurdles and abandoned ideas along the way. The final design of the rendering architecture as presented

in this thesis is described in detail in Chapter 4. Chapter 5 introduces the verification and performance framework used with the architecture, and presents the performance results along with some discussion. The majority of the discussion, however, is left for Chapter 6, which analyses many ways in which this architecture could be improved upon, and its performance made more competitive with today's technology. Chapter 7 quickly summarizes the thesis.

Additionally, there are several appendices attached to this paper. Appendix A presents some additional rendered images to ease the image clutter in Chapter 5. Appendices B through D present the actual code used to program the Raw, and to build the verification and testing framework. Finally, Appendix E lists the raw data that were used to generate the performance results seen in Chapter 5.

Chapter 2

3D Rendering Algorithms

This chapter serves to introduce the 3D rendering algorithms used by the architecture described in this paper. How these algorithms are parallelized across the Raw chip is described in Chapter 4. The purpose of this chapter is twofold: firstly, as background information on how real-time 3D rendering is generally done, for those readers who may not be very familiar with the subject, and secondly, to describe the specific algorithms used in this project, as a reference for the rest of the paper which assumes this information is known.

As this thesis is intended to study architecture and not algorithms, the actual rendering methods used are relatively standard and simple. Enough rendering features are supported to allow exploration of some non-trivial aspects of the architecture, but many advanced and complex features (which most modern rendering processors support) were not implemented due to the limited time available and scope of the thesis. Namely, features such as bump mapping, environment mapping, programmable shaders, mip-mapping, trilinear filtering, anisotropic filtering, anti-aliasing, and multi-texturing, which have become common on modern graphics cards, were not implemented in this project. However, in the interest of presenting a reasonably usable and interesting rendering pipeline, features such as hardware geometry transform, hardware lighting (supporting one directional hardware light and one ambient), perspective-correct texture mapping (in fact, perspective-correct interpolation of all values), smooth or "Gouraud" shading, hardware z-buffer support, and transparency were implemented, and the algorithms used for these features, among others, are presented here. A good comprehensive source on many aspects of rendering is [11]

Several sources were consulted in the process of writing the rendering algorithms used in this project, primarily the lecture notes available at [3]. and [1].

2.1 Introduction to 3D Rendering in General

3D Rendering is the process of presenting a three-dimensional model as a two-dimensional image, with correct perspective such that its three-dimensionality is recognizable. More precisely, 3D Rendering is a *projection* of the three-dimensional model to a two-dimensional plane, and the subsequent *rasterization* of that projection to a digital image, which is usually displayed on a computer monitor. The projection to a two-dimensional plane may be a *perspective* projection, in which objects further from the virtual viewer appear smaller, or it may be an *orthographic* projection, in which objects appear proportional to their original size no matter what distance they are from the viewer.

Real-time 3D Rendering requires that the digital images of a scene be produced very quickly — ideally at least 30 frames per second and typically 60 frames per second and beyond — in order to fool the eye into believing that the scene is in constant motion, and also to allow instantaneous feedback to user controls or other inputs, such as in a video game. Generally, certain methods of 3D Rendering are suitable for real-time rendering while others are not, though the latter may create images of improved quality or realism. For example, ray-tracing [13] is almost exclusively used for non-real-time rendering.

2.1.1 Defining a Scene

When rendering a 3D image, the geometry of the world is generally broken down into simpler shapes, known as *primitives*. With real-time rendering, these primitives are almost exclusively flat triangles, although quadrilaterals are sometimes used as well. Some advantages of triangles are that three points uniquely describe a triangle, and that all triangles are guaranteed to fall completely in a plane (which is defined by the same three points as the triangle). Flat surfaces can be easily broken down into patterns of these primitives, and curved surfaces must be approximated as a mesh of primitives. From here on in this paper, it will be assumed that all primitives in the scene are triangles, and the terms "triangle" and "primitive" will be used interchangeably.

Each triangle may have other information associated with it, such as color, texture, light reflectance, transparency, *normals* (vectors parallel to the original surface of the object which are used for smoother lighting), and blending mode, to be used in drawing the triangle. More advanced renderers may even support *shaders*, which are custom programs that determine how to draw each triangle. Additionally, the scene has information associated with it, such as ambient light and location of lights, position of the camera (which defines the view to be rendered), and so on.

2.1.2 Geometry Transformation

The first step in drawing a scene is to transform the coordinates of the vertices of the triangles to coordinates in the two-dimensional image that is to result. In a 3D transformation, straight lines in the scene will always map to straight lines in the final image, and therefore triangles will map to triangles (or polygons to polygons, etc). Actually, this transformation is usually a larger set of transformations each applied consecutively. For example, a scene may contain a model of a desk. That desk model, for the convenience of whomever created the scene, may have its own local coordinate system in which all its primitive vertices are defined. The desk may then be scaled, rotated, translated, and skewed (though desks are rarely askew) to its final position in *world coordinates*. In fact, it may go even further than this, as there may be a pencil sharpener model on the desk itself, with its own local coordinate system. In this case, the pencil sharpener could to be transformed into the desk's coordinate system, and then the desk (plus sharpener) could be transformed into world coordinates. This structure of local coordinate systems for objects, built into compound objects, and finally built into a scene is known as a *scene hierarchy*

Once all primitives are defined in world coordinates, they are then transformed into the local coordinate system of the viewer, also known as *eye space* or *camera space*. In eye space, one axis is generally aligned in the direction that the eye is looking, and the other two specify up-down and left-right directions. For example, the z coordinate may specify distance from the camera, while the y and x coordinates specify up-down and left-right

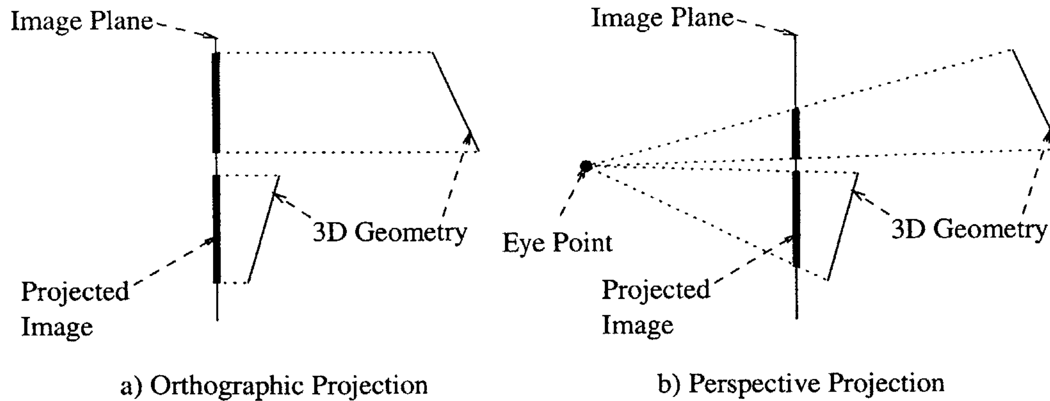


Figure 2-1: Orthographic vs. Perspective Projection
Figure 2-1a shows an orthographic projection, where the depth coordinate is truncated and the geometry appears as a direct projection from the scene to the final image plane. Figure 2-1b shows the same world geometry with a perspective projection, which causes more distant objects to appear smaller on the image plane. With perspective projection, there is a virtual "eye point" where all the lines of projection converge.

coordinates respectively.

Next, these three-dimensional coordinates must be projected into two-dimensional coordinates for use in the final image. There are several ways to perform this projection. Orthographic projection would simply discard the z coordinate, resulting in a direct projection to the 2D plane. Perspective projection, however, would divide the x and y coordinates by the distance from the viewing plane (which is related to, but not generally the same as z). This makes distant objects look smaller, creating an image that looks like a realistic two-dimensional representation of a three-dimensional scene. Figure 2-1 illustrates how the scene geometry is projected to the final image plane using orthographic and perspective projection.

Other transformations usually occur, as well. For example, often times the geometry is transformed into a set of coordinates that specifies the visible parts of the scene within a normalized range (such as 0 to 1 or -1 to 1), and any geometry that has coordinates outside that range is partially or fully off the screen (or to close to or too far away from the viewer). These are known as *normalized device coordinates*, and the range of visible coordinates defines what is known as the *viewing frustum*. Discarding objects outside of that frustum before further processing is known as *culling*. Sometimes, objects appear partially within the frustum and must be split up, so that the part outside of the frustum can be culled; this process is called *clipping*. Generally, culling and clipping is done for performance reasons, although some geometry must be culled for correct operation — most notably, any geometry that has a vertex on the eye point must be culled to avoid a divide by zero error when doing perspective projection, and any geometry behind the image plane usually must be culled to avoid rendering geometry that appears behind the virtual viewer.

Finally, the coordinate system is transformed to the actual coordinate system of the resulting digital image, where each pixel is addressed by an integer coordinate.

To specify all these transformations, matrices are used, and coordinates expressed as vectors multiplied with matrices result in the new, transformed coordinates. In fact, if several multiplications must happen sequentially, all the matrices for those transformations

may be multiplied together to produce one matrix that performs all the transformations in one step. This can result in a large performance gain over doing every transformation individually for every single vertex in the system.

In general, a three-dimensional coordinate can be represented by a column 3-vector, and transformed with a 3x3 matrix, like so:

$$\begin{pmatrix} x' \\ y' \\ z' \end{pmatrix} = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix}$$

Here are some examples of matrices used for various transforms (adapted from [5]):

$$\begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta \\ 0 & \sin\theta & \cos\theta \end{bmatrix} \quad \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & s_z \end{bmatrix}$$

A rotation by θ about the z axis. A rotation by θ about the x axis. Scaling by s_x, s_y, and s_z.

However, this simple matrix multiplication cannot represent all the transformations one would be interested in performing. For example, translations are not representable, and neither are perspective projections. To represent all these transformations, 4x4 matrices are used, along with four-dimensional vectors for the coordinates:

$$\begin{pmatrix} x' \\ y' \\ z' \\ w' \end{pmatrix} = \begin{bmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ m & n & o & p \end{bmatrix} \begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix}$$

Three-dimensional coordinates are represented in the four-dimensional vector by treating w specially — the true values of x, y, and z are obtained by dividing them all by the value of w. Coordinates represented in this way are known as *homogeneous coordinates*. Most of the time, $w = 1$, and the rest of the coordinates can be treated normally. Homogeneous coordinates are used because now translations and perspective projections of three-dimensional coordinates can be represented in a 4x4 matrix:

$$\begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

A translation by t_x, t_y, and t_z (from [5]). A simple perspective projection, where x and y coordinates are divided by the distance z (from [2]).

Generally, if w is nonzero, then the homogeneous coordinates specify a point in space that can be translated, scaled, rotated, etc. However, if w is zero, then the coordinates specify a direction (towards a point at infinity). Directions can be scaled and rotated, but cannot be translated. Normals to surfaces and light directions are two values for which representation as a direction would be appropriate[1].

[1]In the actual implementation of the 3D architecture described in this paper, normals are represented

As stated earlier, a large number of transformations can be combined together into one matrix, to speed computation. Generally, geometry is transformed from whatever model space it is in to normalized device coordinates. Then culling occurs, and *perspective divide* (where the three coordinates are divided by the value of w). Finally, the resulting coordinates are transformed into pixel addresses for display on the screen (though this can be done by a simple scaling, and does not require a full matrix multiplication). One exception is that normals to surfaces are transformed to world coordinates, rather than directly to normalized device coordinates, for use in lighting calculations (as lights are generally also defined in world coordinates).

2.1.3 Rasterization and Interpolation

Once the image pixel coordinates that correspond to triangle vertices are calculated, then the triangles can be "filled in" with pixels, and this process is called *rasterization*. One method of performing rasterization is to trace one edge of a triangle, and fill in the rest horizontally from that edge. Another method is to calculate three inequalities, one for each edge, and use them to determine whether a pixel is inside the triangle. Then, for all pixels in a bounding box around the triangle, if it is inside the triangle, draw it.

Along with simply drawing the correct pixels, values from the vertices are *interpolated* across the triangle. These values can include color, texture coordinates, light intensity, and depth (depth for each pixel is used for *depth buffering*, described later). Alternatively, each triangle may use the same value for color and light intensity across the entire shape, instead of smoothly interpolating values.

One way to linearly interpolate the values from the three vertices of the triangle is to solve a plane equation of the form $v = Ax + By + C$, which takes the location on the screen (x, y) and returns an interpolated value at that point, v. The values for A, B and C can be solved for using routine linear algebra using the known solutions at the three vertices of the triangle. Also, to speed things up, instead of calculating the entire formula for every pixel (which requires two multiplies and two adds), one can calculate an initial value for each row of pixels, and an incremental value for the row, which is how much the equation changes for each pixel ($\frac{\partial v}{\partial x}\Delta x$, where Δx is the spacing between pixels, generally 1). For each pixel in the row, this incremental value is added to the current total, which only requires one addition per pixel.

However, unless the primitive is aligned parallel to the image plane, simply interpolating the parameters of the vertices linearly across the screen will not produce accurate results. The reason for this is illustrated in Figure 2-2. Equally spaced points on the screen are spaced further apart on a primitive as it becomes more distant from the viewer, and likewise equally spaced points on the primitive are spaced more closely together on the screen as the primitive becomes more distant. The result is that the parameters, which are assumed to be distributed linearly across the primitive's surface, are distributed in a non-linear fashion on the screen. Interpolating in screen space is usually acceptable for smooth parameters such as color and light intensity, since the inaccuracy will not be easily noticeable. However, with texture coordinates, the resulting textures can easily look incorrect with simple screen-space interpolation, and so-called *perspective correct* interpolation is generally a must.

To determine how to interpolate variables correctly, recall that in a perspective projection, the coordinates of a vertex in screen space are divided by the distance from the viewer

---

with a nonzero w. This was simply an oversight, and special logic was needed to make sure normals were not affected by translations.

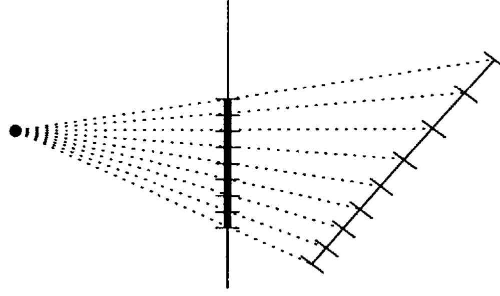


Figure 2-2: Interpolation in Screen Space
Equally spaced points across the screen result in nonlinearly spaced points across the surface of the primitive, and vice versa. Therefore, interpolating a parameter linearly across the screen will not result in the same values for the pixels as interpolating linearly across the primitive. Interpolating correctly in these situations is called *perspective correct interpolation*.

(non-linear interpolation is only required in perspective projections). Before this perspective division, equally spaced coordinates on a triangle's surface can be linearly interpolated across the triangle from its vertices due to the planar nature of a triangle. After the perspective divide, however, each coordinate on the surface of a triangle must be divided by the distance from the viewer, z, to produce the equivalent screen coordinate. If x_1 and x_2 are the x coordinates of one edge of a triangle *before* perspective divide, and z_1 and z_2 are the depths of each of those points respectively, and s is a parameter that is stepped in equally spaced increments from 0 to 1, then a set of undivided x coordinates that are evenly interpolated across the primitive is given by $x_1 + s(x_2 - x_1)$, and a set of screen-space coordinates that correspond to these undivided x coordinates on the triangle's edge is given by (from [4]):

$$\frac{x_1 + s(x_2 - x_1)}{z_1 + s(z_2 - z_1)}$$

However, polygon filling algorithms work the other way: they step linearly across the screen, and need to determine how far along the primitive's surface they have gone for each position on the screen. That is, if x'_1 and x'_2 are (post-perspective-divide) coordinates of the edge of a triangle in screen-space, which correspond to points x_1 and x_2 with depths z_1 and z_2 before perspective divide, and t is a parameter that is stepped in equally spaced increments from 0 to 1 as the rasterizing algorithm is moving across the primitive, then the current screen coordinate that corresponds to t is given by (from [4]):

$$x'_1 + t(x'_2 - x'_1) = \frac{x_1}{z_1} + t\left(\frac{x_2}{z_2} - \frac{x_1}{z_1}\right)$$

The challenge is to determine a value for s, and hence the distance along the surface of the triangle, for every value of t, which represents the distance along the projected image of the triangle on the image plane. To do this, we set screen coordinates equal to screen coordinates:

$$\frac{x_1}{z_1} + t\left(\frac{x_2}{z_2} - \frac{x_1}{z_1}\right) = \frac{x_1 + s(x_2 - x_1)}{z_1 + s(z_2 - z_1)}$$

and solve for s in terms of t (also from [4]):

$$s = \frac{tz_1}{z_2 + t(z_1 - z_2)}$$

Now as t is stepped linearly in screen space during rasterization, the corresponding distance along a primitive's surface can be found with the above formula, and the value of s used to interpolate any parameter from one vertex to the other. The resulting interpolated values of the parameter will be perspective correct. This method can be easily extended to two dimensions.

In practice, the above method is rarely used directly, as calculating the formula to map t to s for every pixel is relatively expensive — tz_1 must be interpolated, which at best requires an addition per pixel for incremental interpolation; then s must be calculated which requires an add and a divide (assuming that the z value in the denominator is incrementally interpolated); and finally the actual parameter must be calculated using $v_1+s(v_2-v_1)$, which requires a multiply and two adds. Instead, parameters at each vertex are pre-multiplied with $\frac{1}{z}$ as part of the perspective divide step, then are interpolated linearly in screen space, and finally are divided by the screen-space interpolated value of $\frac{1}{z}$ for that point to produce the interpolated parameter. Assuming incremental interpolations, this method requires two adds and a divide per pixel, in contrast to the 4 adds, one multiply, and one divide required for the straightforward method. This method can be shown to be correct by substituting for s in $v_1 + s(v_2 - v_1)$, to get:

$$v_1 + \frac{tz_1}{z_2 + t(z_1 - z_2)}(v_2 - v_1)$$

Manipulating this formula further results in:

$$v_1 + \frac{t\frac{1}{z_2}}{\frac{1}{z_1} + t\left(\frac{1}{z_2} - \frac{1}{z_1}\right)}(v_2 - v_1)$$

$$\frac{1}{\frac{1}{z_1} + t\left(\frac{1}{z_2} - \frac{1}{z_1}\right)} \left\{v_1 \left[\frac{1}{z_1} + t\left(\frac{1}{z_2} - \frac{1}{z_1}\right)\right] + t\frac{1}{z_2}v_2 - t\frac{1}{z_2}v_1\right\}$$

$$\frac{1}{\frac{1}{z_1} + t\left(\frac{1}{z_2} - \frac{1}{z_1}\right)} \left(\frac{1}{z_1}v_1 - t\frac{1}{z_1}v_1 + t\frac{1}{z_2}v_2\right)$$

$$\frac{\frac{1}{z_1}v_1 + t\left(\frac{1}{z_2}v_2 - \frac{1}{z_1}v_1\right)}{\frac{1}{z_1} + t\left(\frac{1}{z_2} - \frac{1}{z_1}\right)}$$

This is exactly what we are looking for: the value, v, pre-multiplied by $\frac{1}{z}$, linearly interpolated in screen space (due to the parameter t), divided by $\frac{1}{z}$ linearly interpolated in screen space. The value v can represent any parameter that is defined at the vertices of the triangle that must be interpolated in a perspective-correct manner, such as texture coordinates, color value, light intensity, and so on.

The resulting pixel objects, which contain x and y screen coordinates along with a depth value and interpolated (or perhaps flat-shaded) values of all its parameters, are called *fragments*. Fragments go through an optional texture lookup, then all their values are blended together to get the final value to be sent to the resulting image, which for the

purposes of this paper is stored in special memory known as the *framebuffer*.

2.1.4 Rendering to the Framebuffer

When fragments from primitives are drawn to the final buffer that stores the resulting digital image, care must be taken to make sure that closer objects are drawn on top of more distant objects. One method to ensure this is known as the *painter's algorithm*, in which primitives are first sorted from back-to-front, and then drawn in this order to ensure that closer fragments are overwriting those that are further away, and not vice-versa. However, the painter's algorithm has a hard time dealing with intersecting primitives, and is not very efficient for parallelized rendering schemes, and so a different method is used. This method, known as *z-buffering* or *depth buffering* , uses another buffer, separate from the framebuffer, where the current depths of the pixels stored in the framebuffer are kept. When a fragment is to be drawn, first its depth is compared with the pixel already in the framebuffer, by looking up the latter's depth in the z-buffer. If it is in front of that pixel; it is rendered, otherwise it is discarded. Note that even with a z-buffer, translucent primitives must be drawn back-to-front to ensure that they are correctly blending with the polygons behind them, but that opaque polygons can be drawn in any order.

The value stored in the z-buffer is typically not the actual z coordinate of the pixels, but rather a value 1) monotonically related to the depth, so that fragments maintain the correct depth ordering relative to one another, and 2) that when linearly interpolated in screen space provide the same depth ordering as perspective-correct interpolated z values, so that the overhead of perspective-correct interpolation can be avoided in the rasterization stage (see Section 2.1.3). A typical value that could be used in the z-buffer is

$$\frac{(far + near) - 2near \cdot far\frac{1}{z}}{far - near}$$

where $near$ and far are the distances of the near and far clipping planes of the viewing frustum, respectively (see Section 2.1.2). The projection matrix can be arranged to automatically produce this value for the transformed z' coordinate, after perspective divide.

Using these nonlinearly distributed values instead of z directly also causes the precision of the z-buffer to be much higher closer to the viewer than in the distance, which is usually desirable, depending on the application. When z-buffer precision is not high enough to accurately discern between two very close fragments, then unwanted visual effects can occur. To reduce the likelihood of these effects, the values are stored in the z-buffer with as much numerical precision as possible, usually using 16 or 32-bit fixed point numbers. It is also recommended that the programmer place the $near$ clipping plane as far away from the viewer as possible, and the far clipping plane as close as possible, to make best use of available precision.

This section has introduced the basics on which the 3D rendering algorithms used in this project are based, intended for those without much background on the subject. The next section details the actual implementation of these algorithms used in this project, for reference use with the rest of this paper, which expands on the parallelization of these algorithms, and assumes that the basic implementation is understood.

2.2 3D Rendering Implementation of this Project

A single-processor version of the 3D Rendering code used in the processor described in this paper is available in Appendix B. This code is basically a stripped-down version of the fully parallelized code (Appendix C), with code used for parallel synchronization removed and code spread across multiple processors placed into one thread. However, it still retains some elements of the structure of the parallelized code, as it was not written from scratch. The parallelized code and the rendering architecture in general is described in more detail in Chapters 3 and 4. This section serves to describe the actual 3D rendering algorithms used in the project — to this end, the actual structure of the code and some implementation details are not considered to be very important.

2.2.1 Geometry Transformation

The processor receives from an external source input that is primarily organized by primitive, consisting of the coordinates of three vertices to define a triangle, and the texture coordinates for those vertices. The rest of the info that is needed for each vertex (color, normal, texture mode, lighting mode, transparency, etc) is taken from the current render state. The external source can send commands at any time to change the render state, and the new render state will take effect for all subsequent vertices and primitives.

The render state also specifies the transformation matrices that are currently in use — the external controlling program provides a matrix that transforms from the current model coordinates to world coordinates, ModelToWorld, and a matrix that transforms from world coordinates to normalized device coordinates, as well as performing any perspective projection desired, WorldToView. Internally, two more matrices are kept: ModelToView, which is the matrix multiplication of WorldToView and ModelToWorld, and is used to transform incoming geometry directly to normalized device coordinates in one step; and NormalToWorld, which is used specifically to transform vertex normals from model-space to world-space for lighting calculations, and is discussed further below. The WorldToView matrix, after the transformed vector is normalized by dividing all the coordinates by the resulting w value (the fourth coordinate), causes all the x, y, and z coordinates within the viewing frustum to lie in the range -1 to 1. x specifies the horizontal screen coordinate and increases from left to right. y specifies the vertical screen coordinate and increases from top to bottom. z specifies the depth value to be stored in the z-buffer, and therefore should be linearly interpretable as described in Section 2.1.4, and increases from near to far coordinates.

The model-space x, y, z vector is multiplied by the ModelToView matrix, to result in pre-perspective-divide coordinates (this is sometimes known as *clipping space*). Simple clipping is performed here to remove any geometry that is closer than the near clipping plane, which both removes geometry behind the viewer and vertices that are too close to the eye point. This must occur before perspective divide, as a point too close to the eye point could result in $w \approx 0$, and a possible divide by zero. Since the perspective divide has not occurred yet, z is compared against $\pm w$ instead of ± 1. Also, the case of w being close to zero is checked for explicitly, in case the transformation matrix is badly formed, or just doing something unexpected.

If a primitive intersects the near clipping plane (rather than being fully on one side or the other), it is simply dropped. Tests against the other planes in the frustum were not implemented due to limited time, although a full implementation would ideally perform

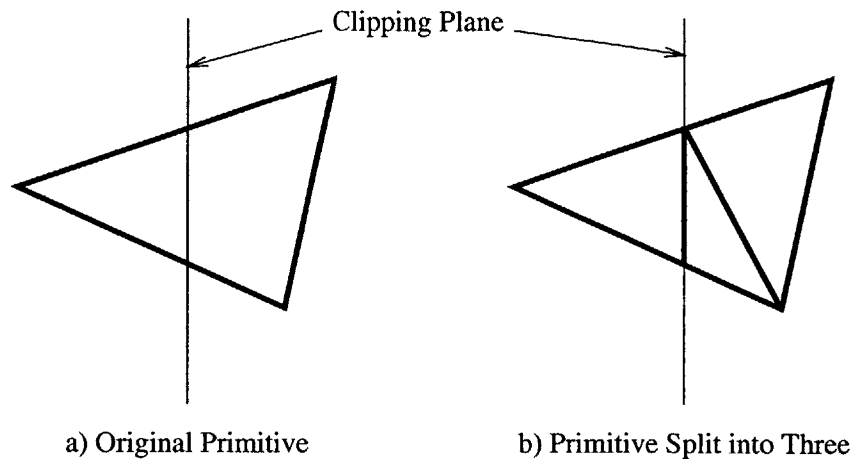


Figure 2-3: Primitives that Intersect the Clipping Plane
Ideally, a primitive that partially intersected a clipping plane would be split into smaller primitives that are either entirely without or outside of the viewing frustum. However, the project described in this paper does not implement this algorithm due to time constraints.

these. Also, a full implementation, instead of dropping a primitive that intersected one of the clipping planes, would instead break the primitive up into smaller primitives, so that each would be fully in or out of the frustum, resulting in a better quality rendering (Figure 2-3). This feature was also not implemented due to time constraints.

At this point, if the primitive is still visible, a quick backface culling check is performed[2]. *Backface culling* removes any polygons that are facing away from the viewer — where in this case having vertices specified in a clockwise order is defined as facing the viewer. Backface culling is done for performance (as most objects are solid, and the sides' backfaces are interior to the object and cannot be seen), and also because the rasterization algorithm used (Section 2.2.2) requires primitives to be oriented in this fashion for it to work correctly. A programmer who desires a double-sided primitive can either render two primitives back to back, or re-arrange vertices in software as necessary.

If the primitive turns out to not be visible at this point, it is discarded and the code moves on to the next primitive. If it is visible, then next up comes the directional and ambient lighting calculations, if lighting is enabled for this primitive. Ambient light's intensity is modulated with the primitive's reflectivity, using a fixed point representation which uses 0–255 to represent 0.0–1.0. The code also uses one global directional light, and the intensity of the light is calculated at every vertex by the negative dot product of the global light direction vector and the "normal" of the vertex. The normal defines a direction perpendicular to the surface of the object at the vertex, and if light is shining directly along the normal it should be brightest, while if it is shining parallel to the surface (at a 90° angle to the normal), it should be darkest. Light shining from the other side of the surface should have no effect (i.e., negative results from the dot product are treated as zero).

The normals for the vertices need to be transformed to world coordinates before they are compared with the light vector, which is also stored in world coordinates. However, there are

[2]Actually, it was discovered late in the process that this is implemented incorrectly — backface culling should be performed *after* perspective division, as the division can re-arrange the vertices and change the primitive's orientation with respect to the viewer. However, this was discovered too late to be able to fix, test, and re-run simulations with the new code, so the older version remains in this paper.

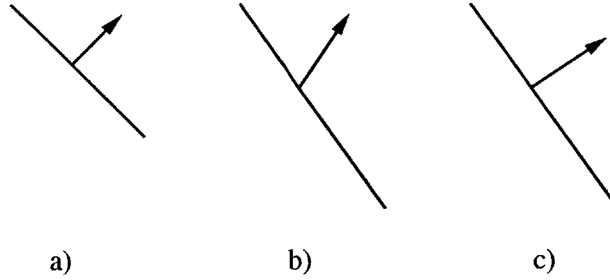


Figure 2-4: Transformation of Normals
Figure 2-4a shows a surface and its normal before translation. Figure 2-4b shows the normal transformed using the same matrix as the geometry, which results in a skewed image. Figure 2-4c shows a correctly transformed normal under the same geometry transformation.

two things one must be careful of when transforming normals: 1) that normals should not be translated, and 2) that under certain transformations, the normal cannot be transformed to world coordinates by the same matrix that is used for the geometry vertices. The code creates a special transformation matrix to be used on normals called NormalToWorld, using the function MatrixInvTrans. This function ensures the former consideration by only using the upper 3x3 part of the matrix, which results in no translation. The latter problem is illustrated in Figure 2-4. To correctly transform a normal under all circumstances, one must use the transpose of the inverse of the geometry-to-world transformation matrix. However, the scaling of the normal is not important, as it is normalized after it is transformed; therefore, only the adjoint needs to be calculated instead of the full inverse (which is the adjoint divided by the determinant). The code to do this calculation is from [11]. Given the input matrix M, the transposed adjoint is:

$$\begin{bmatrix} m_{11}m_{22} - m_{12}m_{21} & m_{12}m_{20} - m_{10}m_{22} & m_{10}m_{21} - m_{11}m_{20} & 0 \\ m_{21}m_{02} - m_{22}m_{01} & m_{22}m_{00} - m_{20}m_{02} & m_{20}m_{01} - m_{21}m_{00} & 0 \\ m_{01}m_{12} - m_{02}m_{11} & m_{02}m_{10} - m_{00}m_{12} & m_{00}m_{11} - m_{01}m_{10} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Finally, the vertices' new coordinates x', y', and z', the color and alpha values r, g, b, and a, the texture coordinates u and v, and the light intensity *intensity* are all multiplied by $\frac{1}{z}$, if applicable. $\frac{1}{z}$ is also stored as w_1, and all these values, along with mode information, are sent to Stage 2, which performs rasterization and interpolation.

2.2.2 Rasterization and Interpolation

To rasterize the triangle, the code first calculates a bounding box around the triangle using the min and max coordinates of its vertices. Then it sets up the incremental interpolation for each parameter that is to be interpolated, scans across the entire bounding box region, and for every pixel within the polygon, sends an "Untextured Fragment" to the next stage, along with its interpreted parameters.

Before the triangle is rasterized, however, the final transformation step is performed: transforming from normalized coordinates to screen pixel coordinates, where the pixel coordinates run from 0 to VWIDTH and VHEIGHT, which are defined at compile time. Vertex coordinates are defined as floating point numbers, where 0 is the left or top screen edge,

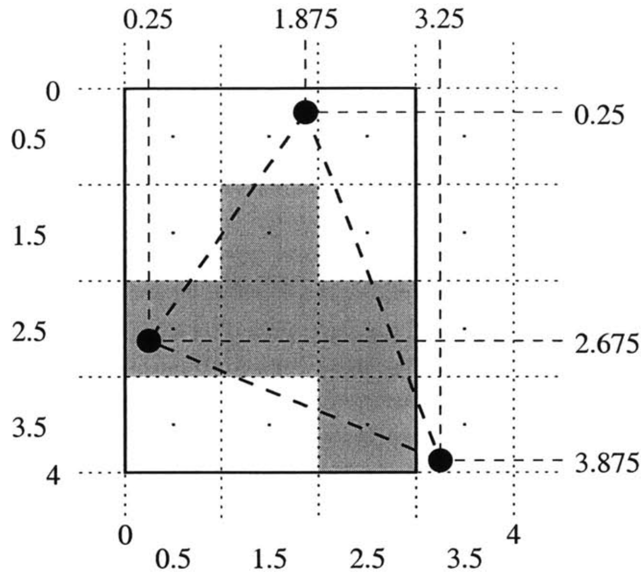


Figure 2-5: Triangle Rasterization to Pixels
The dotted grid represents the actual screen pixels, with the black dots in the center of each square representing the center of the pixel. The coordinates to the left and bottom are screen-space coordinates. A triangle to be rendered has heavy dashed lines, and dots for its vertices, with its vertex coordinates given above and to the right of the image. The black rectangle represents the initial bounding box used for rasterization of the triangle, and the shaded pixels are those which are actually generated once rasterization is complete.

and VWIDTH + 1 and VHEIGHT + 1 are the right and bottom edges, respectively. Pixels are centered on .5 values — 0.5, 1.5, 2.5, etc., up to VWIDTH + 0.5 or VHEIGHT + 0.5. In this system, a floating point value can simply be truncated to give you its nearest screen pixel neighbor. When rasterizing, a pixel is only filled in if its center point is within the primitive — therefore, the actual bounding box need only include those pixels whose centers are within the primitive's general (floating point) bounding box. An example triangle on a small grid of pixels is shown in Figure 2-5, with the bounding box shown, and pixels within the primitive shaded grey.

Next, the stage sets up the equations for each line of the primitive, as well as each variable that is to be interpolated (see Section 2.1.3). To determine whether a pixel is within the triangle, three plane equations are set up, one for each line, that evaluate to less than or equal to zero if the pixel is on the "inside" side of the line, or on the line, and greater than zero if the pixel is on the "outside" side of the line. The sides are determined by assuming that the triangle vertices are given in a clockwise ordering. If any pixel evaluates all three line equations to less than or equal to zero, then it is inside the triangle and a fragment is generated. Given line vertices (x_1, y_1) and (x_2, y_2), the equation for each line is:

$$l = ax + by + c$$

$$a = y_2 - y_1$$
$$b = x_1 - x_2$$

$$c = y_1 x_2 - y_2 x_1$$

where $l \leq 0$ means the pixel (x, y) is on the right-hand side of the line going from (x_1, y_1) to (x_2, y_2), which is the inside of the triangle made up of three of such lines defined in a clockwise order.

The set-up of equations to interpolate the vertex parameters is the same for each parameter to be interpolated (color r, g, b, alpha a, texture coordinates u and v, depth z', light intensity i, and perspective correction factor $\frac{1}{z}$). The generic parameter p, along with the parameter's values at each of the vertices, p_0, p_1, and p_2, where the vertices themselves have coordinates (x_0, y_0), (x_1, y_1), and (x_2, y_2), will be used in place of the above variables to demonstrate the form of the set-up equations:

$$p = p_a x + p_b y + p_c$$

$$p_a = \frac{1}{\det M}(p_0(y_1 - y_2) - y_0(p_1 - p_2) + b_{1e})$$

$$p_b = \frac{1}{\det M}(x_0(p_1 - p_2) - p_0(x_1 - x_2) + b_{2e})$$

$$p_c = \frac{1}{\det M}(p_0 m_e - x_0 b_{1e} - y_0 b_{2e})$$

$$\frac{1}{\det M} = \frac{1}{x_0(y_1 - y_2) - y_0(x_1 - x_2) + m_e}$$

$$m_e = x_1 y_2 - x_2 y_1$$

$$b_{1e} = p_1 y_2 - p_2 y_1$$

$$b_{2e} = x_1 p_2 - x_2 p_1$$

The $\frac{1}{\det M}$, m_e, b_{1e}, and b_{2e} variables are calculated separately as they are used in more than one location, and calculating them once saves set-up time[3].

When stepping through the bounding box to perform the rasterization, the equations for the triangle's lines and for each of the parameters are evaluated incrementally. The initial value for each row is calculated, as well as an incremental amount that is added to the value for each pixel traversed. Since all the equations are linear, this generates a very accurate solution very quickly (one add per pixel). In the case of generic parameter p, the incremental value is $\frac{\partial p}{\partial x}\Delta x$, where Δx is the spacing between pixels, and with a pixel spacing of 1, this ends up being simply p_a (which would be called "pdx" in the code). The starting value of each row is also generated incrementally from the starting value of the previous row, by adding $\frac{\partial p}{\partial y}\Delta y$, which is simply p_b, or "pdy" in the code. One final optimization performed is that when pixels within the triangle are found in a row, the loop ends as soon as another pixel outside of the triangle is found, instead of always going to the end of the bounding box.

One exception is that the z values are calculated from the full equations each time instead of via incremental evaluation. They are also stored in a fixed point representation from this point forward. Both of these are to help increase the dynamic range of the z-buffer — however, the current implementation of this code does not take full advantage of

[3]Note that there may be vastly more efficient ways to perform these same calculations, and all the calculations in this project. This project was mostly concerned about generating a correct and reasonably fast implementation, so not as much time was spent on trying to optimize it as much as possible, nor research the best possible algorithms for the job

	Texture Coordinates																		
Original	-1.4	-1.2	-1.0	-0.8	-0.6	-0.4	-0.2	0.0	0.2	0.4	0.6	0.8	1.0	1.2	1.4	1.6	1.8	2.0	2.2
Repeat	0.6	0.8	0.0	0.2	0.4	0.6	0.8	0.0	0.2	0.4	0.6	0.8	0.0	0.2	0.4	0.6	0.8	0.0	0.2
Mirror	0.6	0.8	1.0	0.8	0.6	0.4	0.2	0.0	0.2	0.4	0.6	0.8	1.0	0.8	0.6	0.4	0.2	0.0	0.2
Clamp	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.2	0.4	0.6	0.8	1.0	1.0	1.0	1.0	1.0	1.0	1.0

Table 2.1: Coordinate Mapping in Different Texture Modes
This table shows how the original texture coordinates for each fragment get mapped into a 0–1 range for texture lookup. Repeat repeats the same texture over and over in either direction; Mirror does the same but flips every other instance of the texture; and Clamp sets the coordinate to the closest edge of the 0–1 range (actually, Clamp uses the color of the last texel on either edge instead of 0.0 and 1.0 exactly, which have different interpretations when used under bilinear filtering).

the range available with the fixed point notation, as the calculation still occurs with 32-bit floating point numbers.

Each untextured fragment generated by this algorithm is passed then to Stage3, where texturing and blending occur.

2.2.3 Texture Mapping and Blending

At this point, the fragment's texture value is looked up if necessary, and the final color value and alpha of the fragment is determined from the texture value, the light intensity, the color value, and its blending modes.

Texture coordinates are defined as floating point values from 0 to 1, where 0 is the left or top edge of the texture, and 1 is the bottom or right edge. Just like with screen coordinates, the texture pixels' (or texels') centers fall on 0.5, 1.5, 2.5, etc. Values less than 0 and greater than 1 are allowed, and they are interpreted according to the primitive's *wrapping mode*: none, repeat, mirror, and clamp. A wrapping mode of "none" means that texels outside of the 0–1 range should be left black. "Repeat" means that the texture should be treated as an infinite tiled array, such that a coordinate of 1.3 should be treated as 0.3, 4.6 should be treated as 0.6 and -3.9 should be treated as 0.1. "Mirror" is similar to repeat, except that alternating tiles are flipped, so that adjacent tiles are mirrors of one another. "Clamp" simply takes the texel value of the closest actual texture edge (0 or 1) and uses that value forever in either direction. Table 2.1 demonstrates the mapping of coordinates in different texture modes.

The normalized texture coordinates are then multiplied by the width or height of the texture to generate texel coordinates — these coordinates are simply truncated in nearest-neighbor mode, which produces a blocky-looking texture but renders more quickly. Also available is bilinear filtering, which interpolates linearly between texel colors to create a smoother appearance (it is called *bi*linear because the interpolation occurs on a two-dimensional texture). For bilinear filtering, the colors of the four texels that surround the given point are retrieved from memory, and the final color is a weighted average of the texels according to the given point's distance from their texture centers. To make this easier, the code shifts everything by 0.5, so that pixel centers are aligned with integers, and then the actual texel coordinates are compared with the truncated coordinates, and the truncated coordinates plus one. Special care must be taken so that the Repeat, Mirror, and Clamping modes work correctly under bilinear filtering, by re-wrapping after the shift occurs, and

fetching the correct four texels for blending even when on a wrapping edge.

After the texture lookup, the code then blends the texture, color, and ambient and directional light together to create the final fragment color and alpha. First, if a hard alpha threshold is specified for the texture or color values, then that is applied (a hard alpha threshold specifies a value, below which the color is completely transparent and above which it is completely opaque). A slight hack is applied at this point, which allows a hard alpha texture to be blended with colors without resulting is 50% transparency in the transparent areas of the hard-alpha texture — this is reasonable, as not only would such partial transparency look wrong, but it would also slow down the rendering considerably. Next, the color and texture values are combined according to the texture mode: none, in which no color is applied to this fragment at all; color only, in which just the color is passed through; texture only, in which just the texture is passed through; col/tex blend, where the result is a weighted average of the color and texture; tex-on-color decal, where the texture is primarily shown, and the color values are shown only where the texture is transparent; color-on-tex decal, where the color is primarily shown, and the texture is shown only where the color is transparent; and color/texture modulation, where the result is the normalized modulation of the color and texture values (treating 255 as 1.0).

Finally, if the primitive is lit, then the color of each fragment (but not the alpha value, of course) is modulated with the directional light and ambient light separately. These two results are then added together (using saturating addition, which maxes out at 255 for each of red, green and blue) to create realistic lighting. If the primitive is unlit, then the color is sent through as-is.

2.2.4 Rendering to the Framebuffer

The final color is a 32-bit value, 8 bits each for red, green, blue, and the alpha value (to blend with what is already on the screen). The framebuffer maintains two "pages" for the screen — one is the "front" buffer, which is currently displayed, and the other is the "back" buffer, which is generally where the next frame is being drawn. When the frame is finished, the pages will generally be flipped, so that the process of drawing the scene is hidden from the viewer. However, the external program sending commands to the processor has full control over which buffer to draw to.

First, if necessary, the code checks the z-buffer to see if the fragment should be drawn to the screen (the z-buffer check can be disabled by the user). Then, if the fragment is being drawn, and it is transparent, the pixel value already on the screen at that point (in the appropriate buffer) is retrieved, and blended with the current color value. Then the new pixel color is sent out (again, to the appropriate buffer), and the z-buffer is updated with the new value of z' (again, if necessary, as the z-buffer write can also be disabled).

This process repeats for every fragment in the primitive. When the primitive is finished, the process starts over by reading in the next command.

2.3 Summary

In this chapter, some basic theoretical groundwork for the process of 3D rendering was introduced. Also, this chapter described the actual rendering algorithms that were used in the processor described by this paper. This information is provided as background information to the parallelization of the rendering architecture, and its performance analysis, which was the focus of this project and of the remaining chapters of this thesis.

Chapter 3

Designing the Parallel Rendering Architecture

This chapter describes the design process for the architecture presented in this paper. The initial goals and starting point are outlined, and several interesting problems and the development of their solutions are discussed. For a full description of the final design of the processor see Chapter 4.

3.1 Design Goals

The first design goal was to implement this architecture on Raw in such a way that it could conceivably be run on currently available hardware. At the time, this meant using a 4x4 Raw tile configuration, along with some glue logic for specialized I/O. Also, the design was to be implementable mostly in C, rather than direct assembly, for ease of development and debugging, except for static switch routing code and low-level cache manipulation and interrupt control.

The Raw processor was to be used as a slave, or co-processor, which accepted external commands and outputted rendered images through a relatively simple interface: a render host interface which took commands from the host system and sent them with minimal translation and flow control into the Raw processor, and a framebuffer controller which interpreted messages coming from Raw as pixels sets and reads in some generically defined framebuffer memory, that could conceivably be connected to a DAC for video display. A major design goal of both the host and framebuffer interfaces was to be able to treat the Raw processor as a black box of sorts: the external logic should require a minimum of knowledge about what exactly is going on inside the Raw processor, how many tiles it has, what its parallelization algorithms are, etc.

Internal to the processor, on major goal was balanced parallelization across all 16 tiles, with tiles remaining active as much as possible. However, the resulting image should be exactly equivalent to the image produced on a single-tile implementation of the algorithm (See Chapter 2). Finally, though it was not expected that performance be very good on the first pass of this architecture, a reasonable performance on 320x240 images was expected (on the order of 30 frames per second at least)[1].

[1]Note that this is an incredibly meek goal compared to modern graphics accelerators, which are pushing 1600x1200 frames, with multiple passes for anti-aliasing and such, at frame rates above 60Hz. Analysis and recommendations on the speed issue are given in Sections 5.3.2 and 6.1.4.

3.2 Basic Parallelization

The design of this project began blindly in a way, with no idea what the bottlenecks would be, how different computations would balance against each other, or how efficiently different parallelization techniques would use the processing power available. The project was treated as a sort of exploration, in the hopes that the end result would offer a good deal of insight as to how the architecture could be more efficiently laid out. In this sense, the results were successful, as many insights into improvements where noted in the process of doing this project — see Chapter 6 for a detailed rundown of many of these suggestions and insights.

However, this meant in order to begin the parallelization, a starting point had to be determined in the design stage without much guidance in terms of hard performance data for different structures. This starting point began by using a result well-known in computer graphics — that rendering is well-suited to deep pipelining and massive parallelization, due to few dependencies between different primitives. In fact, a "graphics pipeline" is a construct so common that it even influences the models of high-level graphics APIs such as OpenGL [16].

3.2.1 Graphics Pipelines

The concept of a graphics, or rendering pipeline is similar to that of pipelines in digital architecture in general: split a computation up into several stages, and advance the data items to be computed along one stage at a time, so that every stage is doing work simultaneously, much like an assembly line for an automobile. In hardware, pipelining is a huge win, since oftentimes all the hardware to do each computation is already there, and simply sits idle waiting for the rest of the computation to complete. Pipelining usually requires two additions to a non-pipelined setup: adding a small to moderate amount of state storage between stages, and making sure that cases where one computation depends on another and handled correctly (i.e., one computation may need to wait for another one to complete before it can continue). However, the speed improvements can be substantial — instead of each piece of data taking n cycles, multiple pieces of data can be processed in that same n cycles, improving the *throughput* of the computation. One downside is that it may take longer for each individual piece to go through the computation — an increase in *latency* — but when there are not many dependencies between computations, latency is not a huge problem. Figure 3-1 illustrates some basic concepts of pipelining.

A graphics algorithm, such as the one described in Chapter 2, can be easily pipelined — one primitive may be going through perspective transformation while another is being rasterized, while yet another's pixels are going through z-buffer lookup. In fact, a full rendering pipeline may look like Figure 3-2, with an Application that generates geometry and textures to render, interpretation of the rendering Commands, Geometry transform (which may include sub-stages such as Transforming to eye coordinates, Clipping, and doing Projection), Rasterization into fragments, Texture mapping, combining Fragments into final colors, and outputting to the Display. In a full system, parts of this pipeline may be implemented in software and other parts in hardware.

In fact, often a rendering algorithm can be pipelined arbitrarily deeply, as the only dependence between simple primitives is their rendering order, which is preserved with a simple pipeline. More complex dependencies can occur, however, such as primitives that blend with the image in the framebuffer — these primitives must make sure not to read the framebuffer until the primitives before it have written. This could be solved via interlocking

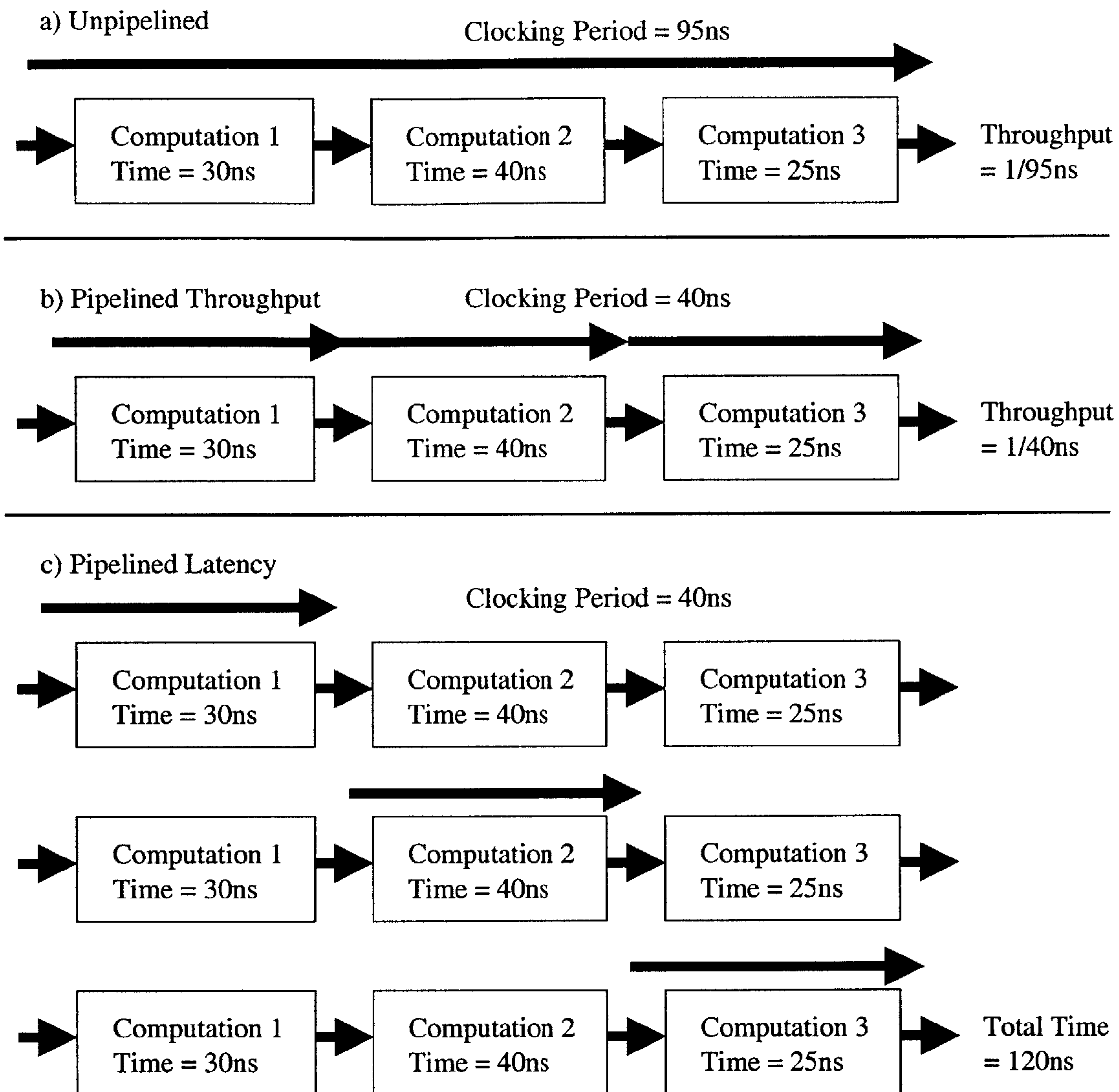


Figure 3-1: Basic Concepts of Pipelining
Figure 3-1a pictures a three-step computation, which has a total running time of 95ns. If data is clocked through this pipeline once every 95ns, then the computation would have a throughput of 1/95ns. Figure 3-1b shows the same computation, but pipelined, so that three sets of data can be going through it at once, one per step of the computation. On each clock each piece of data moves ahead one step. The whole pipeline must be clocked at the speed of the slowest computation, 40ns, and so on average data comes out of the pipeline every 40ns, and the throughput would be 1/40ns. However, as shown in Figure 3-1c, the time it takes one individual piece of data to get through the pipeline is 120ns — this is the "latency" of the pipeline.

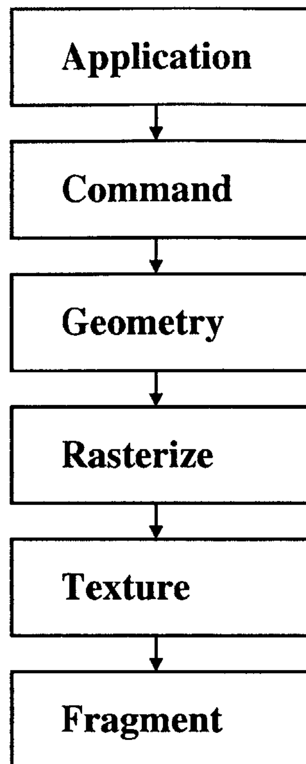


Figure 3-2: A Simple 3D Graphics Pipeline

(stalling the primitive) or forwarding the data needed before writing it to the framebuffer. Generalized shaders for primitives can cause more complex dependencies to occur.

It is also possible to run several graphics pipelines in parallel with each other; although now more effort must be taken to ensure that ordering and dependencies are handled correctly. Due to the nature of graphics data, this is generally a much easier and more efficient effort than with general-purpose CPUs. One possible classification of graphics parallelization is described in [12], where the location in the pipeline where dependencies are handled and correct render order is determined is of key importance. A so-called "sort-first" architecture separates primitives according to where they appear on the screen, with one parallel pipeline dedicated to each region or set of regions in screen space. Since there is no overlap between regions, there will be no dependencies between the parallel pipelines for basic fragment operations. A "sort-middle" architecture is similar to sort-first, except that the full geometry processing and transformation occurs before sorting into different pipelines according to screen region. In a "sort-last" architecture, all primitives are transformed and rasterized in any distribution across the parallel pipelines, and the final image is *composed* together from the separate fragments, often using a depth buffer to ensure correct ordering between dependent fragments (i.e., those on top of each other). However, primitives which do not use the frame buffer (such as writes directly to the framebuffer) and transparent primitives that depend on the values of what was drawn before them cannot be ordered correctly simply through the use of a depth buffer. Solving the problems with these kinds of dependencies is discussed more in Section 3.3.2.

3.2.2 Mapping Parallel Pipelines to the Raw Architecture

It was decided to use a sort-last organization for this project, primarily due to load balancing concerns. Sort-first and middle architectures separate computation based on screen regions, and it is difficult to balance this load across pipelines without complex regioning algorithms. Sort-last algorithms also allow primitives to be distributed in any arbitrary fashion across the parallel pipelines, and require minimal communication between the pipelines until the final compositing step, and so give the designer a large amount of flexibility in deciding how to implement the rest of the pipeline.

The 4x4 configuration of the Raw processor suggested a very intuitive pipeline configuration: four pipeline stages, with four-way parallelism in each stage. Deciding upon exactly how to split up a graphics pipeline into four balanced pipeline stages was a bit of a shot in the dark at this point, but a reasonable starting point was assumed: That the first stage could do command interpretation and geometry transforms, the second would do rasterization, the third would do texture lookup, and the fourth would do the final compositing. This seemed reasonable because the balance between geometry and rasterization depends on the primitive size (larger primitives spend more time rasterizing, while smaller primitives spend more time in geometry transformation), and texture mapping and final compositing both are on par with rasterization in terms of computation cost[2]. Each stage would consist of four Raw tiles, implementing the same processing step with four-way parallelism. The data to be rendered would be somehow distributed across the pipelines at the top stage, then would proceed straight down each pipeline until the last stage, at which point writes to (and possible reads from) the Framebuffer would be serialized. Each pipeline stage has a bank of RAM off to the side of the Raw chip, where shared data such as the render state, texture memory and the depth buffer can be stored. A simple diagram of this setup, with basic data flow indicated, is shown in Figure 3-3.

Finally, this discussion is assuming that all the tiles are being used to actively process the 3D data, and that none are being used as control or communication tiles.

Given these starting points, constraints, and goals, the design task moved on to developing exactly how each point of the architecture was to be implemented. Some parts were relatively straightforward, but others presented very tricky problems. These problems are the focus of the rest of this chapter.

3.3 Difficult Design Problems

In moving from a basic picture of the architecture to a fully operable implementation, several interesting and difficult problems in parallelization, communication and synchronization were encountered, and the design process in tackling these problems is described here. The first major challenge was developing a method to distribute incoming primitives evenly across the top four tiles, while meeting the goals of a black-box external interface and not using a specialized control tile. Next, the problem of sequential correctness in the face of transparent primitives or primitives that do not fully use the z-buffer, in light of the fact that primitives can go out-of-order in parallel pipelines, is described. Finally, the issue of synchronizing the bottom four tiles, which perform the compositing of the fragments to the framebuffer, so that only one can access the z-buffer and frame buffer at a time, is tackled. For a description of the final design of the rendering architecture, see Chapter 4.

[2]Whether these initial assumptions were accurate or reasonable can be determined by looking at the actual performance data in Section 5.3.2.

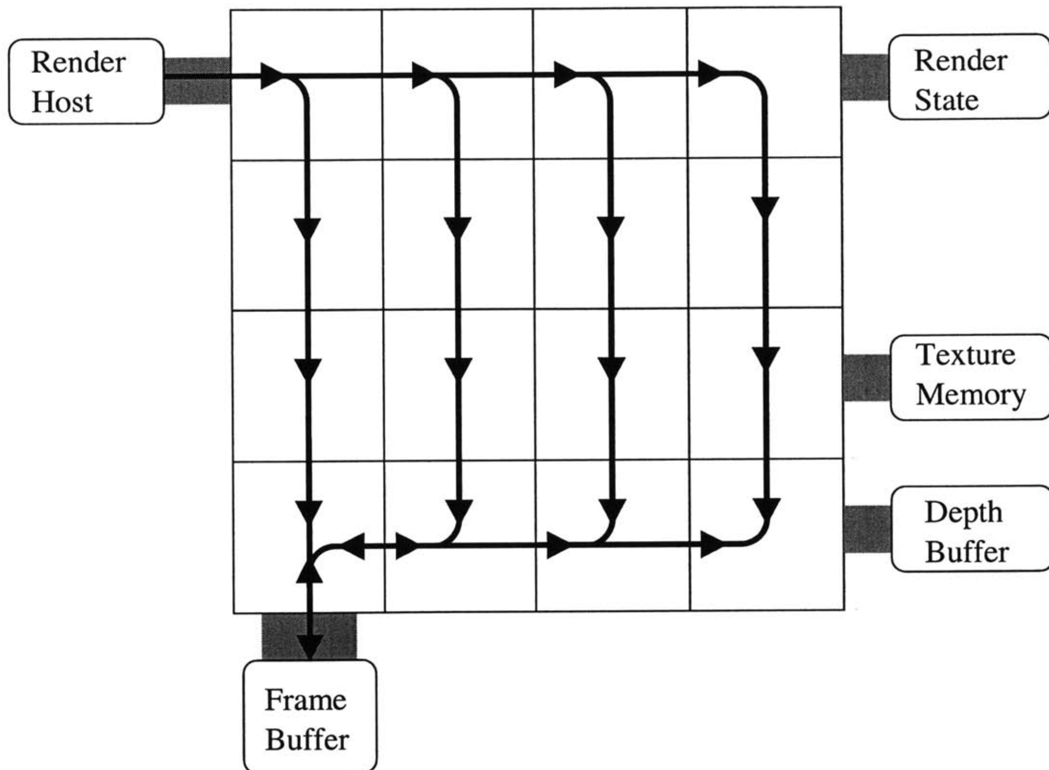


Figure 3-3: Basic Pipeline Layout on Raw
The data comes in to the Raw processor from an external Render Host interface on the top left. It is then split among the four parallel pipelines, and send down through the four pipeline stages as it is processed. Finally, it is recombined in the last stage where reads to, and writes from, the framebuffer are serialized. Each pipeline stage also has shared RAM memory, which is shown to the right of the Raw processor.

3.3.1 Distributing Primitives Across the Pipelines

Basically, the processor takes rendering commands from an external source, and somehow splits these commands into blocks to be (more or less) evenly distributed across the parallel pipelines. It was a goal to be able to do this with reasonable load-balancing, without the external source needing any special knowledge about the internal setup of the processor, and without needing to dedicate a tile to command interpretation and distribution (or at best hamper a tile's performance by making it perform both the command distribution and calculations for the first pipeline). To lay the groundwork for determining the best way to communicate with the external device, and internally among the tiles, all the major modes of communication on the Raw chip are summarized in the next section.

3.3.1.1 Communication in the Raw Architecture

On the Raw processor, there are several methods available to communicate between tiles and with external (I/O) devices [21]. There are two *dynamic networks*, which accept messages with a header that specifies the destination tile or I/O port for the message and its length, and then automatically route these messages to their appropriate destination. One dynamic network — the Memory Dynamic Network or MDN — is intended for use only with memory system messages and other low-level communication that is guaranteed not to deadlock. The other — the General Dynamic Network or GDN — is for use by general user program communication[3]. However, both the MDN and GDN are functionally identical in terms of the network behavior and hardware (they are implemented as two-dimensional wormhole-routed [17] bidirectional mesh networks between the tiles with no end-around connections).

There are also two *static networks* that connect all the tiles on the Raw chip in a two-dimensional mesh without end-around connections. However, there is no message routing on the static network, and instead individual words are routed according to a statically defined program on each tile. The static network program can be different for each tile, and is usually defined by the programmer of the application that is doing the communication. The program is run on a "static switch processor" on each tile, and can simultaneously route any number of non-conflicting combinations between the network directions North, East, South, and West, a small static switch register file, the other static network, and the inputs and outputs of the main processor on the tile (though there are some other restrictions, such as the fact that the register file is one-ported, meaning only one register can be written to and one read from at a time, though that read value can go to many destinations). The switch processor can also perform basic decision making (using comparisons to zero), decrement registers, and perform branches, subroutines calls and loops. The static network is very versatile if communication patterns are known beforehand, and has been used successfully as a network for Instruction Level Parallelism [22] and Stream Processing [8].

Finally, tiles can communicate through shared memory. A tile is allocated a private slice of all the memory available (in RAM modules external to the chip), but can access the entire range of memory if needed. There is a data cache on each tile, and in case of a cache miss a MDN message is sent to the appropriate RAM module to retrieve the data. The caches are write-back, and there is no hardware cache coherence mechanism, though software can specifically flush cache values out to memory or invalidate cache locations to

[3]If communication on the GDN deadlocks, then a complex deadlock recovery scheme is implemented using the MDN to communicate. This is why the MDN must be guaranteed not to deadlock. However, the specific details of these algorithms is beyond the scope of this paper. For more info, see [20].

load the newest value, allowing some limited use of shared memory communication.

3.3.1.2 Choosing a Network for Commands

One major drawback, for this application, of using the dynamic network to inject commands into the processor from the external source is the fact that the dynamic network requires the sender to know exactly which tile to send the message to. If the external source sent the messages directly to the tiles that were to process them, then the external source would have to know the internal configuration of the rendering processor, and also would be in control of the load balancing algorithm for the whole architecture! This did not seem to be a viable situation, given the goals above.

This could be ameliorated by having the external source always send commands to a certain tile — say $(0,0)$, the upper-left corner tile — and have that tile then redistribute the commands to the other pipelines according to some load-balancing algorithm. However, this solution does not go along with the goal of having every tile being involved in processing the data, with no special control tiles.

Therefore, it was decided to use the static network, at least for the external interface. With the static network, external I/O devices can simply "push" bytes into the Raw processor, without caring about where they end up being routed. The static network would also be used for inter-tile communication in the top stage if possible.

However, the static network has its own drawback — it is designed for times when the routing patterns can be determined at compile time, or by very simple decision making at runtime. Dynamic events are difficult to handle with static routing; dynamic events such as when commands come in from the external source, and when the tiles are finished with their geometry processing and can accept new data cannot be predicted easily by static network code.

Additionally, if the static switch is blocked waiting for a value to appear at the source of a specified route, then the entire switch blocks and no other simultaneous routes proceed until the value arrives. This method of flow control guarantees correct route ordering to the programmer, but can also limit the use of the switch in dynamic situations. For example if the switch is waiting for a command to input from the west, it cannot also be routing the output of the main tile processor south to the next pipeline stage. Any waiting means that all the routes for that switch processor will be suspended (this also applies to blocking because an outgoing buffer is full).

Obviously, these limitations of the switch processor must be overcome, or worked around, in order to program the static network to work with the dynamic events of the first stage of a parallel rendering architecture.

3.3.1.3 Techniques for Using the Static Network Dynamically

If the programmer wants the static network to be able to respond to dynamic events in some way, there are basically two choices available. The first is to use the available decision making and looping commands, which can decrement registers and make conditional branches based on comparisons of registers to zero (including greater than or less than comparisons). For example, the end of a series of commands intended for one processor could be marked with a zero, or a certain number of zeros in a row. The switch could then start passing commands on to the next processor. Alternatively, a count of the number of words in this

command group could be sent to the switch, and it could count down from that number until zero, and then take some action.

The other technique is to have the main tile processor forcefully change the switch's current program counter, and cause it to execute some other program. One has to be careful, though, to make sure there are no routes in-flight at this time, or to make sure that any current routes are maintained in the switch-over. This can be done to respond to a dynamic event known to the tile processor — for example, that it has done processing its current set and can take more commands now.

A more difficult problem is using the static network to perform two independent communications, such as communicating incoming data across the top row of tiles, and sending transformed primitives down each pipeline. The problem, as mentioned in the previous section, is that while the switch is waiting on one stream of data, it cannot simultaneously be sending another stream — all waits have to be synchronized. So if no commands are coming in from the external host, the switch physically cannot route data coming out of the processor. One possible solution to this that was considered was to have a constant stream of zeros in place of any dynamic stream, so that the switch never had to block on anything. Data could be marked by a nonzero count value, and then that many words could be sent.

Finally, some esoteric solutions are possible. For example, the switch processor supports a jump-to-register command to support returning from procedure calls. However, it could also be used to jump to an address sent to the tile over the static network, allowing the switch in effect to make more complex decisions on incoming data than simply comparison to zero. However, using this technique can be tricky and should not be necessary.

3.3.1.4 Developing the Distribution Algorithm on the Switch

It was decided at the outset that data should be distributed by primitive, where each subsequent primitive would be processed by a different tile in the top row. Therefore, the switch would have to recognize end-primitive markers in the incoming stream to know when to start sending data to a different processor. Using a zero word or series of a certain number of zero words to represent the end of a primitive was considered, but this leads to the problem of dealing with zeros or series of zeros that normally appear in the data stream (for example, in sending a transformation matrix in to the processor). They could be somehow escaped, but a simpler method was desired. Another idea was to have the switch stop after every word or every n words and wait for the tile processor to tell it if the primitive has ended yet or not (the tile processor can do more complex decision making on the incoming data, and send a 1 or 0 to the switch). However, this seemed inefficient, and other tiles also may need to know when the primitive ends to rearrange their routing patterns, requiring all the switches to periodically stop and wait for messages from the switch for the currently active tile, resulting in even more inefficiency and complex communication. Finally, a solution was decided upon which seemed to work well: first, the command would send a byte giving a count of the number of words left to route, and the switch would subsequently route that many words. After the block of data had been sent, the next word would again be a count of the number of words to follow. Each block could be an individual command, for example, or perhaps several commands grouped together. The switch would continue reading the count, and routing the words, until it came across a count of zero. This would signify the end of a primitive, and the switch (and all other switches which were routing that stream) could act accordantly.

The next problem was to decide exactly how to distribute the primitives to different

tiles. The easiest method would be a purely round-robin approach, where each primitive goes to the next tile in the row, and then wraps around at the end back to the first tile. To implement this, each switch processor would have three states: active, where it is currently routing commands to its own tile processor; passing, where it is currently passing commands to its east to send them to the active tile down the way; and idle, where the active tile is to its west and the switch is just waiting for a message to arrive so it can become the next active tile. A tile in the active state would route commands to its processor until it encountered an end-primitive marker as described above. Then it would switch to passing mode (or directly to idle if it is the last processor in the row), and begin to count the number of end-primitives that go by until the active processor wraps around back to the beginning, and it no longer has to pass commands and can switch to idle mode. Note that the first (westernmost) processor in the row would switch directly from passing to active mode again on wrap-around.

In the single processor version (Section 2.2), each command that changes the renderstate has an immediate effect on subsequent vertices and primitives. This means that the updated renderstate must somehow be communicated to the other tiles that receive commands after the active tile. One way to do this is to use a shared memory location in the off-chip RAM, and have each tile invalidate its renderstate before starting the computation, and flush any changes from its cache before the next tile starts up (this can be made more efficient by keeping a "changed" variable, which each processor will look at before deciding whether to invalidate all of its cached renderstate). However, a tile must make sure that it has finished flushing its updated renderstate before it allows a new tile to begin processing. This can be accomplished by having the switch, after reaching and end prim, wait for the processor to send it a word before passing the commands to the next tile in the chain. Note that this ok-to-go word must be passed along to other tiles as well, so they can wait before rearranging their routing patterns to the new tile.

What if the next active processor is not yet ready to receive another primitive? In this case, the switch for that processor would block trying to send data to it, causing it to stop reading the stream coming from the west. This stream would back up, filling up buffers and causing switch processors that were sending the stream to also block. Eventually, the stopped stream would back up until it reached the I/O interface where the external source of commands was connected. The static network interface has flow control signals that would tell the external source that it has to block and wait for there to be room for new commands to be sent, and the external source would have to follow these flow control rules. Therefore, the static network's own flow control capability should handle limiting the rate of incoming commands.

Now when a processor has finished transforming its primitive, it will want to send the result south to the next stage in the pipeline. Originally, the plan was to use the static network for this as well. However, if the static switch is currently blocking on the command stream, then it will not be able to route data coming out of the processor (note that having two static networks does not help this situation, as they are both routed by the same switch and if one is blocking, the other blocks as well). The solution described in the previous section of having a constant stream of zeros moving through the static network was considered, but one problem with that method was how to generate the stream of zeros during idle times between messages. It would be easy enough to modify the external source glue logic to send zeros into the Raw chip when no commands were available, but zeros would also have to be sent south from each processor along the paths to the next pipeline stages as well, and the processor is busy doing transformations, and does not have time to

send constant zeros to the switch! Additionally, this solution only works for the case where the switch would be blocking because there is no data coming in — the situation where the switch blocks because the stream is backed up is not addressed at all. Therefore, this idea was discarded.

Eventually it was decided that the transformed primitives sent from the first stage to the next would be send as messages over the GDN. This does not result in too much of a loss of efficiency, and allows the static network program to concentrate wholly on distributing the primitives among the top four tiles.

3.3.1.5 Improving on Round-Robin Distribution

The above scheme would work as a final solution. However, pure round-robin distribution as described above has one drawback: it will wait for the next tile in line to accept the current primitive, even if there are other tiles ready for a new primitive that are now waiting idle. This can result in poor utilization of the parallel pipelines and uneven load balancing. An ideal solution would send the commands to the next available tile, not simply the next tile in the round robin sequence.

In order to do this, the static network programs would have to somehow know when each tile has finished its computation and is ready to accept new commands. This is a purely dynamic event, and can happen at any time after the tile finishes getting its primitive.

One way to deal with this, as described earlier, would be to have the processor actually change the program counter of the switch processor when the event occurs. If the switch processor is set up like a state machine, then the main processor can change its program counter to a new state based on its current state. One thing to be careful about using this method is to make sure that any routes in flight are still routed after the switch — this can be accomplished by making sure that every program line in every state in the switch code that can be forcefully interrupted by the processor like this has an equivalent line in the resulting state, with the same routes to perform. The main processor must make sure to change the switch to the correct state.

The switch code will now have at least five states: Active, as before, passing-busy where it is passing commands to the east and its own processor is still busy, passing-ready where it is passing commands to the east and its own processor is ready for new commands, idle-busy where it is waiting for input from the west and its processor is busy, and idle-ready where it is waiting for input from the west and its processor is ready to accept new commands. When the switch leaves the active state, it immediately goes to a -busy state. When the processor is finished, it forcefully changes the switch state from its -busy state to the corresponding -ready state. A switch will only enter the active state and start routing commands to the processor if it is in a -ready state.

Realizing a full-utilization round robin scheme actually requires quite a bit more states and communication between the switch processors, the nitty gritty details of which are described in Section 4.2.3.1. But generally, when one processor finishes, and the static network starts redirecting data to the next processor in the chain; if that processor is not ready for data the switches must pass it and redirect to the processor after that. In fact, if all the processors are busy, the static network would spin around the chain of processors waiting for one to become ready. During this time, the commands coming from the external source must be blocked so they do not become lost in the static network. Each switch must communicate with all the others about whether its processor is going to take its turn or pass it on (for example, by sending a 1 or 0 value west), so that the other switches can

change their routing state accordantly. Once all the details were worked out, however, the final solution worked just as intended, and no processor was left idle for more than a few cycles as its turn came around.

3.3.2 Maintaining Sequential Correctness

One goal of this project is that the parallelized version of the pipeline should produce the exact same results as the sequential single-tile version described in Section 2.2. And as mentioned in Section 3.2.1, one issue that affects parallel pipelines is dealing with dependencies between primitives processed in parallel. Basically, a dependency exists between primitives if they overlap on screen (or more specifically, dependencies exist between the overlapping fragments of those primitives). A depth buffer can solve many of these dependencies, by making sure only the frontmost of the fragments is rendered, no matter in which order they are actually processed in the pipelines. However, if a primitive does not use the z-buffer (which can be specified by the programmer), or if a primitive is transparent (in which case all the geometry behind it must be rendered before it is), then the z-buffer alone will not ensure a sequentially correct result; some other method must be used.

Fine grained interlocking between dependent fragments would conceivably yield the best results, but is difficult to implement without dedicated hardware, as it would require a large amount of inter-tile communication on an architecture like Raw. Therefore, a more coarse grained solution was devised: make sure that primitives that are either transparent or do not use the z-buffer, known from here on as "in-order" primitives, must be rendered in the same order with respect to other primitives as they arrived in the graphics processor. Other primitives, known from here on as "out-of-order" primitives, can execute in any order relative to each other, but must still execute in the same order with respect to any in-order primitives that may be rendering alongside them.

This scheme is rather conservative — not all in-order primitives would depend on one another and on out-of-order primitives, only those that overlapped each other. Also, sometimes out-of-order and in-order primitives that overlap can still be rendered in any order, such as when a transparent primitive is behind an opaque primitive. But for simplicity, this project implemented this strict ordering policy whenever in-order primitives are involved. Suggestions for improvements over this policy appear in Section 6.1.2.

The next step, after determining the desired ordered rendering policy, is determining how to implemented. A simple starting point is to give every in-order primitive a sequence number, and have another sequence number stored in the last stage of the pipeline. An in-order primitive can only go forward to the framebuffer when its sequence number is the same as the current sequence number in the last stage, and will increment the last stage's sequence number when it is completed. This solves the issue of in-order primitives being ordered relative to each other.

The problem now is getting out-of-order primitives to be rendered in the correct relation to the in-order primitives, but still allow them to render in any order amongst themselves. First, out-of-order primitives must not be allowed to render in the middle of a group of in-order primitives. This can be accomplished by resetting the sequence number to zero after the group of in-orders has completed, and only allowing out-of-orders to proceed in the last stage if the sequence number is zero. However, two problems can still occur in this situation: out-of-order primitives that should have gone before the group of in-order primitives could be delayed in the pipeline and end up rendering after the in-orders, and out of order primitives that should go after the in-orders could end up rendering before

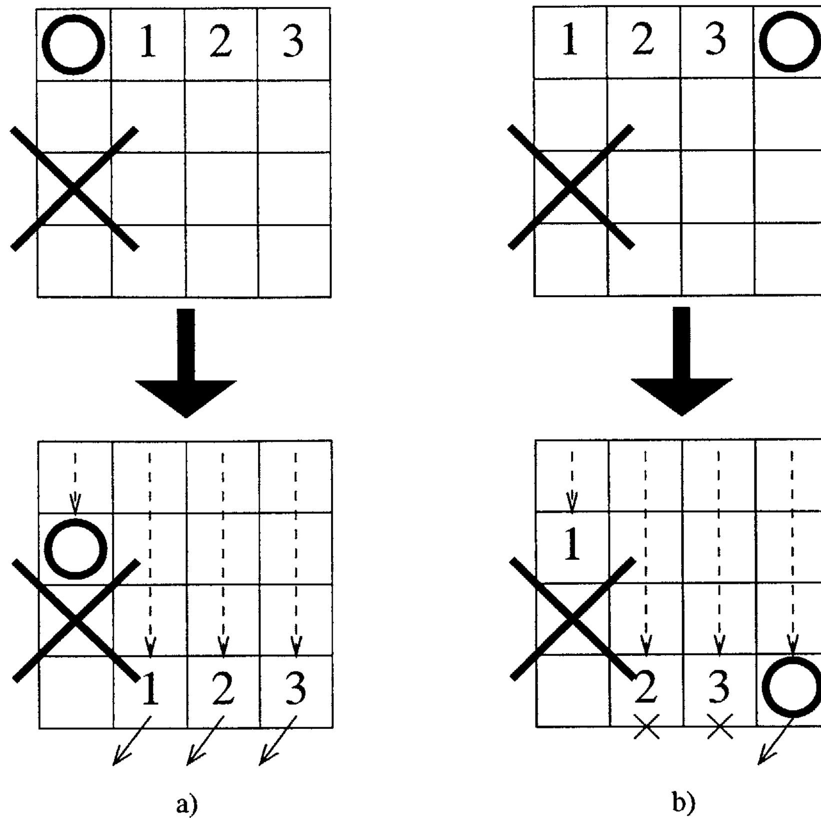


Figure 3-4: Problems with Simple Sequencing
This figure illustrates the simple sequencing algorithm described in the text where in-order primitives receive sequence numbers, and out-of-order primitives receive zeros. The large "X" in each figures represents some sort of blocking or long delay in the pipeline. In Figure 3-4a, the out-of-order primitive, represented by "O", should go before the group of in-order primitives, but ends up being blocked and going after them. In Figure 3-4b, the out-of-order primitive should go after the group of in-order primitives, but the first in-order primitive is blocked, and the other two are waiting for the sequence number to increment, so the out-of-order primitive ends up going before them.

them (for example, if the in-orders are delayed somehow). See Figure 3-4 for illustrations of some of these situations.

A solution was developed for the latter problem, but it assumed that the incoming primitives were distributed in a purely round-robin fashion, and could not "skip" over a processor if it was busy. The solution involved sequencing a certain number of out-of-order primitives after the end of an in-order primitive group, to make sure that these out-of-orders will not render before the in-order group goes. The number that needed to be sequenced was no greater than the number of pipeline stages minus one, as the round-robin distribution guaranteed that only this many primitives could ever skip ahead of a primitive that arrived before them. Using the improved round-robin algorithm, however, the problem becomes much more difficult, as an unbounded number of primitives could skip ahead of any particular primitive (this is also illustrated by Figure 3-4; if the blockages never clear, innumerable other primitives could pass by the ones stuck in the blocked pipelines). No good solution was discovered for the former problem, either, since there was no way to tell

Primitive Type:	O	O	O	I	I	I	O	O	O	O	I	I	O	I	O	I	O	O	O	I
Sequence Number:	1	1	1	4	5	6	7	7	7	7	11	12	13	14	15	16	17	17	17	20
Completed Number:	3	1	2	4	5	6	8	9	7	10	11	12	13	14	15	16	17	19	18	20

Table 3.1: Sequence Numbers Assigned to a Stream of Primitives
This table demonstrates how sequence number assignment works in the sequential consistency algorithm used in this project. A primitive marked I is considered in-order, and one marked O is considered out-of-order. Out-of-order primitives get the sequence number of the last in-order primitive plus one, while in-order primitives get a number corresponding to their position in the total stream of primitives. "Completed Number" shows a possible final ordering of this stream of primitives, in terms of the order that they complete in (as out-of-order primitives can be rendering simultaneously).

an in-order primitive to wait for out-of-order primitives before it that had not gone yet.

To solve all these problems, it was decided to simply give every primitive entering the processor a sequence number. To allow out-of-order primitives to render out-of-order, they were given the sequence number of the last in-order primitive to go plus one, and are allowed to go whenever the sequence number in the last stage is equal to or greater than their own sequence number. This makes sure that out-of-order primitives will not skip ahead of groups of in-order primitives. In-order primitives are given a sequence number that has a value as if every out-of-order primitive was given its own sequence number, and out-of-order primitives increment the last-stage sequence number whenever they complete their rendering. In this way, in-order-primitives will wait for all previous out-of-order primitives to complete, in whichever order they choose, before being able to proceed. This algorithm achieves the ordering goal stated at the beginning of this section, and is implemented as described in Section 4.2.3.2. Table 3.1 demonstrates the sequence numbers that a sample stream of in-order and out-of-order primitives might generate.

3.3.2.1 Dealing With Sequence Number Rollover

This algorithm works with no problems but assumes that the sequence number is unbounded. In a real implementation, the sequence number would be represented in a format with a finite upper bound. For example, if the sequence number is represented as a 32 bit unsigned integer, it will "roll over" if it reaches $2^{32} - 1$, or 4,294,967,295. Rolling over means that incrementing this value will result in 0, instead of the value plus one (basically, the addition is *modulo* 2^{32}). The roll over is actually not a problem for in-order primitives, as they simply check to see if their number has arrived yet, and will handle the rollover case just as well as a normal increment. The problem is with out-of-order primitives, which will go as long as the sequence number in the last stage is equal to *or greater than* their own sequence number. When the last-stage sequence number wraps around to zero, suddenly primitives which believe they should be able will no longer do so, since their sequence number is greater than zero. As a result, the whole pipeline could lock up.

It may seem like 4 billion is a large enough number to not have to worry about the rollover case, but consider a high performance 3D application with 100,000 visible primitives running at 60 frames per second[4]. Assuming that the sequence numbers are not reset between frames (they are not, in the final version of this architecture — this simplifies the

[4]These numbers are just an example. For actual performance data on the 3D processor designed in this thesis, see Section 5.3.2.

design of the last stage). Every second, the sequence number would increase by 6,000,000. It would take merely 715.8 seconds, or 11.9 minutes, for the sequence number to reach 2^{32} and roll over — certainly a short enough time that it cannot be ignored!

One early attempt at a solution to the roll over problem was, instead of giving the out-of-order primitives the sequence number of the last in-order primitive, to make sure that the out-of-order primitives' sequence numbers were never more than a fixed amount below the current sequence number (defined as the number that would be given to an in-order primitive at this point). This number would be far enough behind the current sequence number to guarantee that the out-of-order primitive receiving it would not end up waiting for its turn when it actually should have been able to go. This would occur if the out-of-order primitive got to the last stage before the last stage's sequence number had incremented to the level of the primitive, and can be guaranteed not to happen by realizing that when a sequence number is given to an in-order primitive, the sequence number in the last stage at that exact point in time cannot be more than some number t greater than the number given to the in-order primitive, where t is the number of pipeline stages multiplied by the number of parallel pipelines. That is, in the worst case, if every tile is processing a different primitive, then the current first stage in-order sequence number minus the last stage sequence number at the same time cannot be greater than the number of tiles in the system.

If an out-of-order primitive, therefore, is given a sequence number that is the in-order number minus the number of tiles in the system, then it can never get to the last stage before its sequence number has come up if it should be able to go. If it is not supposed to be rendered yet — for example, if several in-order primitives have gone before it but got held up, and it has reached the last stage before all of them — then the last stage's sequence number can never be more than t less than the out-of-order primitive's sequence number[5]. However, rollover will cause the last stage's number to be less than this value, as it will be 0, but any out-of-order primitives in the last stage at that point will have sequence numbers up near the top of the range (due to the algorithm described in the previous paragraph). Therefore, a primitive can detect a roll-over and continue to operate normally.

Anyway, this method works for the case of a simple rollover where the last-stage sequence number suddenly becomes zero. However, there is also the case of "double-rollover" where the last stage sequence number increases until it re-enters the range where the out-of-order primitives think it is not their turn yet (t below the primitives' sequence numbers) and no longer realize that rollover has occurred. A good amount of time was spent trying to solve this double-rollover problem, but the schemes got more and more complex and exceptions and cases where they did not work correctly were numerous.

In the end, a much simpler solution was found to handle the roll-over case: When the sequence number that would be given to an in-order primitive rolls back over to zero in the first stage, freeze the first stage and let the entire pipeline flush out before sending any primitives with the rolled over sequence number down the pipe (See Section 3.3.4.2 for more info on flushing the pipeline). This guarantees sequential consistency at the cost of a little performance every 4 billion triangles. The performance cost is so low given the number of primitives between rollovers that the solution is acceptable for now, though not theoretically ideal.

[5]Actually, t, which is the number of tiles in the system, is a conservative number. The optimal number could be even smaller than it, but it is not important to have the closest possible bound in this algorithm. A conservative bound works fine.

3.3.3 Synchronization in the Compositing Stage

In the final stage of the pipeline, fragments from several different primitives must be combined correctly into the framebuffer to generate the final image, with possibly several fragments ready to be composed at any particular time. To update a typical fragment the z-buffer must be read, then written, then the framebuffer must be written. In the case of a transparent primitive, the framebuffer must be read as well. The final stage of each pipeline must be synchronized in such a way that the actions of reading the z-buffer, writing the z-buffer, and reading/writing the depth buffer occur as one *atomic* action. That is, no two fragments that have any dependency between themselves should be able to overlap accesses to the z-buffer or framebuffer, or else the final image may not be the correct result.

For example, consider two fragments on top of each other, near and far. The correct final result is that near's color ends up in the frame buffer, and its depth ends up in the z-buffer. However, consider the case where far reads the z-buffer first and sees that it can be drawn, but there is no lock to prevent another primitive from accessing the z-buffer before far is done with it, and near also reads the z-buffer and sees that it can be drawn. Assume that near is faster, and is able to update the z-buffer and draw to the framebuffer during this time as well. Then far continues, writes the z-buffer and updates the framebuffer. Now far's values are stored in the framebuffer and depth buffer, exactly the opposite of the correct result!

If, however, the frame and depth buffers were correctly locked, z would have gotten access to the z-buffer, seen that it can be drawn, updated the z-buffer and updated the depth buffer. Near would not be able to go until far had completed its actions. When near went, it would update the z-buffer and framebuffer with the correct values. Several other situations can be easily described where overlapping accesses to the buffers can result in the wrong final image.

This is basically describing the common *mutual exclusion* problem in computer science [10]: several concurrent processes (the tiles in the last stage of the pipelines) want to access shared resources (the depth and frame buffers), but only one can access them at a time. We need a*locking mechanism* to ensure that while one can access the resources, the others are forced to wait until the one with the access finishes what it is doing. Technically, this lock only needs to occur between dependent primitives, but for simplicity this design locks any attempts to access the depth or frame buffers (this of course results in less performance than may be possible otherwise. See Section 6.1.4 for suggestions on improving this performance).

First, several shared-memory algorithms for mutual exclusion were examined, such as the Bakery algorithm [9] and Peterson's algorithm for mutual exclusion [15]. However, both of these algorithms require a cache coherence mechanism to operate properly.

3.3.3.1 Attempts at Cache Coherence

The Raw processor affords no hardware cache coherence mechanism for shared memory, though it provides software the ability to flush and invalidate cache entries. One simple way to ensure cache coherence is to simply not cache any shared variables — follow every write to the variable by a flush, and precede every read by an invalidate. This, however, would generate a large amount of memory network traffic, especially with several processors simultaneously spinning on a shared variable, waiting for it to change.

An attempt was made at a simpler invalidate-based cache coherence protocol. Invalidate protocols work by caching shared data, and when a processor writes to the shared location,

sending a message that invalidates any other cached copies of the shared data. The raw tiles support what is called an "external interrupt" — a specially formed header with no payload sent on the MDN, which causes an interrupt trap in the processor. It is generally intended for external devices to send to raw tiles (hence the name), but it can also be sent from one tile to another. The idea was to use these interrupts as the invalidate messages, and have the interrupt service routine perform the invalidation of all the shared memory locations.

From the programmer's perspective, this should be transparent, as the interrupt will invalidate the cache, and then the next read of the location will read in the new value. On the tile that is writing to the shared memory, the algorithm is obviously not transparent, as it will have to explicitly flush the value and then send interrupts to the other tiles.

However, this algorithm did not initially work. It turns out that the reason it did not work is that the writing tile did not wait for acknowledgments from all the other tiles after sending invalidate messages before continuing. But to send back acknowledgments, the other tiles must know which tile is performing the interrupt — information that is not readable by software from the interrupt message alone. This information had to be sent via some other route, though sending standard messages between tiles in the MDN can mess up the memory management algorithms of the Raw, and using the GDN to send these messages seemed to have some problems as well (such as the lack of ordering between MDN and GDN messages, and the risk of GDN deadlock considering that the GDN was also being used for other things, such as communication with the framebuffer).

Finally, even if it could have been made to work, these shared memory mutual exclusion algorithms required a large number of cycles and overhead to access a lock once it was unlocked, but it turns out that the performance of the compositing stage is critical (See Section 5.3.2). Obviously, a simpler method for ensuring mutual exclusion was required.

3.3.3.2 A Simpler Approach

Instead of using a shared-memory algorithm for mutual exclusion, a simple message-passing algorithm was attempted. In this algorithm, a token is sent in round-robin fashion between the processors in the last stage. If a processor needs to access the framebuffer or depth buffer, it has to wait for the token to arrive to do so. Once it has the token, then it will not pass it on until it is finished with the shared resource, and another processor can access it. If any processor does not need to access the shared resource, then it can simply pass the token on to the next.

Using the GDN to send the token around, along with the ability for incoming GDN messages to trigger an interrupt, turns out to be a pretty efficient method for mutual exclusion on the Raw processor. The value of the token could also be set to the final stage's current sequence number (See Section 3.3.2), removing one more shared memory lookup. Finally, a tile can buffer and partially process fragments while waiting for the token, and then quickly read and update the depth and frame buffers when it finally gets the token, for better processor utilization and optimized pixel writing (the original version of this algorithm performed the lock for one fragment at a time as they came out of the previous pipeline stage, causing most of the tiles in previous stages to be waiting for the last stage tile to get its turn, only to have one fragment drawn and then have to wait again. Other techniques were experimented with, such as drawing more than one fragment per turn if they were available, but this buffering technique was the best performing).

As well as this method works, however, it still does not bring about an optimal pixel

rate for the graphics processor, as many cycles are still spent between pixel writes to the framebuffer, performing z-buffer access and looping and control code. For some suggestions on improving the pixel rate further, see Section 6.1.4.

3.3.4 Other Difficult Issues

This section describes two miscellaneous design problems that did not fit into the three major categories above, but are interesting nonetheless.

3.3.4.1 Asynchronously Reseting the Processor

One design idea near the beginning of this project was to have an asynchronous software reset capability — where the controller could send a signal at any point in the operation of the processor to reset its state. This would be useful, for example, for an operating system if a rendering process is killed, or for when the computer itself is soft-reset, or for recovery if the processor locks up (though it would be nice if the processor never locked up on its own!) The original plan was to have the external interface send an interrupt to tile $(0, 0)$, and then have that tile propagate the interrupt to all other tiles in the processor. All the tiles would then work together to reset the processor. The hard part would be to remove all the in-flight messages from the static and dynamic networks before performing the reset. It was determined that this feature was not incredibly important for this thesis and would require too much time to implement, and so it was skipped. It would be a useful feature in a finished product, however.

3.3.4.2 Switching Modes and Flushing the Pipeline

It was decided later on in the design that the processor would operate in two modes: command mode and scenestream (see more about these two modes in Chapter 4). Scenestream mode is when primitives are being pushed through the processor to produce a final image, and is the main focus of the design decisions in this chapter. Command mode is for more complex interactions that did not map well to scenestream's architecture, such as texture management and direct framebuffer access. Before switching from scenestream to command mode, the graphics pipeline had to be *flushed* of all in-flight primitives, to make sure that all the commands before the invocation of command mode had been fully processed. Another time when flushing the pipeline is necessary is when the sequence numbers for primitive ordering roll over, as described in Section 3.3.2.

A pipeline flush is basically a barrier-type synchronization which makes sure that all the pipelines have finished their current computations before continuing any processing. A simple way to implement such a flush is to send marker primitives down each pipeline which cause messages to be sent back up when the markers hit the bottom, and the bottom tiles finish all their work. When all the marker responses are received, then it is known that the pipeline has been flushed. One way to get all the markers to be sent is to send GDN messages to all the tiles in the top row, letting them know that it is time to flush the pipeline. A GDN message can trigger an interrupt, so a tile can service it even while waiting on something else (for example, waiting for new commands to come in from the static network). The flushing tile can refuse to send the message to the static network that allows it to start sending commands to the next tile in the chain, which is usually used to make sure that shared memory is flushed before another tile tries to read it, to make sure that computation is halted until the flush is completed. When the markers reach the

bottom, each tile in the last stage can send a GDN message back up to the originating tile (whose address is stored in the marker), who will then either continue the computation or send another message to the top row tiles informing them that scenestream has ended and to start command mode.

3.4 Summary

This chapter presented the initial goals for the design of the processor architecture presented in this paper, as well as a rationale for the design decisions which provided a starting point for the rest of the design. Then, several difficult problems encountered in this project and the processes by which they were solved were presented. Chapter 4 will describe the final architecture produced by this design process, and Chapter 5 presents the performance results of the implementation of this architecture. Chapter 6 suggests several improvements to be made to the architecture, based on what was learned in implementing this project.

Chapter 4

Implementation of the 3D Processor

This chapter describes the actual implementation of the parallel rendering pipeline that resulted from the design process described in Chapter 3. The rendering algorithms are the same as described in Section 2.2, so this chapter will focus specifically on architectural and parallelization issues instead of the 3D algorithms. For reference the source code for the final version (as of the writing of this thesis) of the processor is available in Appendix C.

4.1 Overview of Architectural Organization

As described in Section 3.2.2, a 16-tile Raw processor is divided into four pipeline stages with four-way parallelism each. Rendering commands arrive from the west side of the chip into the upper left tile[1], and are distributed in modified round-robin fashion among the top four tiles on the chip, so that the incoming command stream will not block as long as at least one tile is ready to receive the commands. Each parallel pipeline then acts independently of the others, with data passing vertically down from one tile to the next, until the fourth and last stage in the chip, in which the tiles must take turns updating the depth and frame buffers with the correct image data, taking care not to violate sequential consistency where necessary (Section 3.3.2).

The rendering steps in a typical graphics pipeline were distributed among the four available stages, in a fashion that *a priori* was hoped to produce reasonable load balancing. The first stage handled command interpretation, geometry transformation and lighting, visibility culling and perspective projection. It would send transformed primitives (in normalized device coordinates, and with perspective-divided parameters for each vertex) to the second stage. Stage 2 would perform rasterization and parameter interpolation across the primitive, resulting in a stream of fragments with interpolated values for the parameters being sent to the next stage. Stage 3 would blend together the different parameters to create the final color value for each fragment depending on the current primitive's blending modes — this also includes doing texture lookups, and lighting modulation. The textured fragments would be then sent to Stage 4, where their depths would be compared with the current

[1]These commands would supposedly come from some glue logic, which then connected to an I/O interface of a controlling computer. The glue logic would implement the flow control algorithms described in later sections, as well as the static network interface itself, but would otherwise pass the command data through unaltered.

values in the z-buffer, and where their colors would possibly be sent out to the frame buffer, which is attached to the south side of the bottom left tile.

This implementation uses a variety of different methods to communicate between tiles. Command data being sent from the external source arrives on the static network (See Section 3.2.2 for a discussion of the available communication networks on Raw), and is distributed to the next available processor also via the static network. The first Stage of processors holds the current render state (which can be altered by commands) in a memory bank connected to the easternmost tile in the row[2]. The processors will invalidate their cached copies of render state data when a change has occurred, and will flush out their caches when they cause a change themselves. Finally, Stage 1 sends its results to Stage 2 via a series of messages over the General Dynamic Network (the static switch program that distributes the incoming commands is too busy and complex to support yet another stream, so the static network is bypassed altogether).

Stage 2 receives the commands from the north, rasterizes, and sends its commands out via the static network on to Stage 3. Stage 2 does not need to store any state in memory. Stage 3, however, stores the texture maps (if used) in the RAM block to the east, as well as any indexing information for the texture maps. When finished, Stage 3 streams textured primitives down to the final stage over the static network.

Only one tile at a time can access the z-buffer and framebuffer in Stage 4, which preserves sequential consistency for most primitives but is a pretty conservative approach. To ensure the mutual exclusion between tiles, a "token" is passed among the tiles in a round-robin fashion using the General Dynamic Network (GDN). Only when a tile is in possession of this token can it access the depth and frame buffers, and when it does not need it anymore, it passes it to the next tile. In order to accommodate the fact that the tiles will often be blocked waiting for data to arrive over the static network, the incoming GDN messages can produce an interrupt, and much of the token-passing functionality can be handled in the interrupt service routine. The z-buffer is stored in a RAM chip off the east side of the Raw, and is accessed the same way as the other external memory. The framebuffer is accessed by sending GDN messages to the framebuffer controller, which is south of the bottom leftmost tile in the Raw chip. The framebuffer controller may also send GDN messages back to the tiles in the case of reads from the framebuffer for blending.

The graphics pipeline normally operates in what is called "scenestream" mode while rendering a scene. It is in this mode that primitives and fragments are being streamed through the processor and rendered to the display. However, there is another mode known as "command" mode, in which only one tile is active (the others are idling waiting for scenestream to begin again), and that tile handles all commands coming in from the external interface. Command mode is useful for commands such as texture memory management, direct framebuffer access, and other commands that do not map well to the communication structure of scenestreaming mode.

Figure 4-1 illustrates the basic layout of this implementation on Raw. The remainder of this chapter will describe the implementation of each part of the 3D processor in much more detail.

[2]Having memory banks along the east edge of the processor is actually the default configuration to use in the simulation environment for Raw. It turns out that it is also convenient for this particular design.

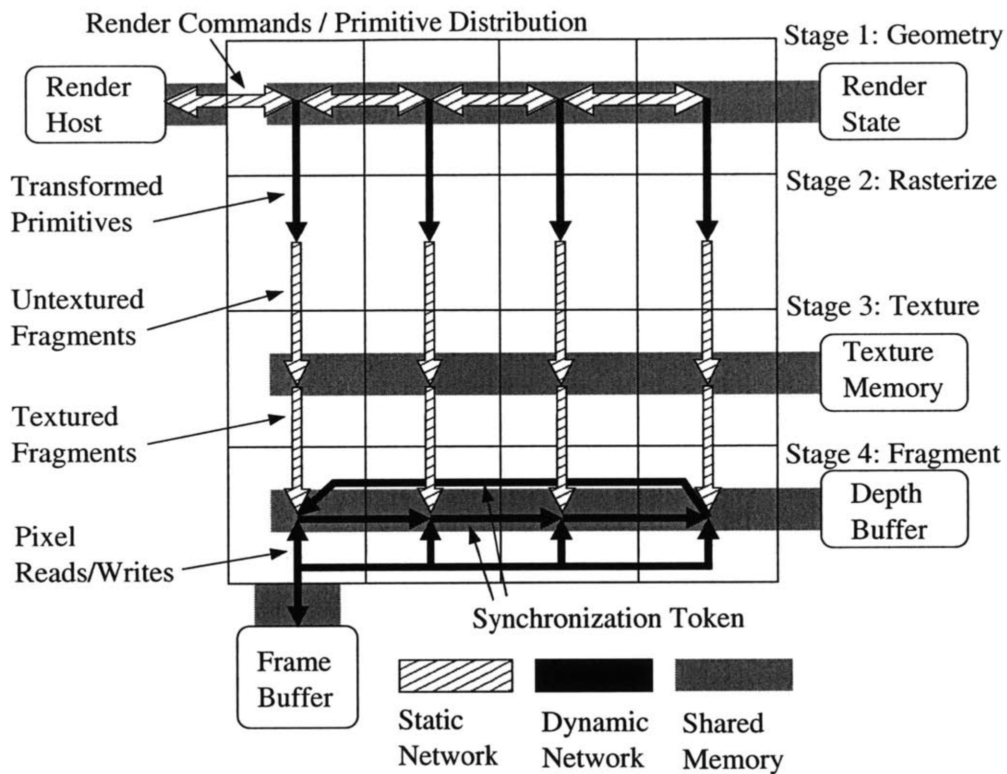


Figure 4-1: Overview of Implementation
This figure presents an overview of the implementation on Raw, the name of each stage in the pipeline, what data is communicated between tiles, and the different means of communication used.

4.2 Detailed Description of Implementation

4.2.1 The Boot Sequence

When the 3D processor is first powered on, the upper left tile in the Raw array, tile $(0,0)$, takes control. All the other tiles are either waiting for messages from this tile or need no initialization and are simply waiting for scenestream mode to begin.

Tile $(0,0)$ first initializes its own interrupts and certain global variables (actually, most tiles do this right on startup). Then it allocates render state information, which due to the way memory is allocated on Raw will be stored in the RAM module to the east of the first stage, and sends a pointer to the render state structure via the GDN to every other tile in the first stage. The values stored in the render state and their meanings are given in Table 4.1 for reference. The tile waits for acknowledgments back from all the tiles in the first row before continuing.

Next, the control tile sends a message to the westernmost tile in Stage 3 to request a pointer to the texture memory control structure, TexManager. Stage 3 had allocated TexManager, as well as texture memory itself, right after boot-up, causing it to be stored in the RAM to the east of Stage 3's tiles. When the westernmost tile of Stage 3 receives the GDN message from the control tile, it sends a pointer to the TexManager structure to all the other tiles in Stage 3, and waits for responses back, before finally sending the pointer to TexManager back to the control tile. This sequence makes it possible to sort out the different GDN purely by the order in which they arrive, without having to tag them with extra identifying data.

After receiving the response from Stage 3, the control tile finally sends a message to the westernmost tile in Stage 4, requesting a pointer to the Depth Buffer. This tile, upon boot-up, allocated the Depth Buffer structure (which also contains a shared variable specifying which page in the framebuffer to render to) and initialized the depth buffer to 0x7FFFFFFF, the furthest number in its fixed point range. When it receives the GDN message from the control tile, it sends a pointer to the depth buffer to all the other tiles in the row, waits for acknowledgments, and then sends the pointer to the control tile. After finishing with the startup GDN messages, the Stage 4 tiles turn on the GDN_AVAIL interrupt, which triggers an interrupt whenever a GDN message arrives, and is used for its synchronization algorithm. (See Section 4.2.3.5).

After all these initializations have completed, all the tiles in the processor except the upper-leftmost tile enter "scenestreaming" mode (See Section 4.2.3), including setting up their static networks to route in this mode, and wait for scene data to be passed down the pipeline. The upper-leftmost tile, or the control tile, sets up its static network for and then enters "command" mode, which is the subject of the next section.

4.2.2 Command Mode

In Command Mode, the external host computer can send instructions to the graphics processor (via the static network through the western port of the top left tile) to update the render state, read and write the frame buffers and depth buffers directly, flip the framebuffer's active page, and upload to and manage texture memory. Basically, it can do everything except send geometry through the pipeline, which requires the processor to be in a different mode called "scene streaming." Of course, the external host can change the mode to scene streaming from command mode, as well.

Variable:	Description:
Updated	Bit vector for whether the renderstate has been updated since the last time each processor had seen it (and therefore must be invalidated); one bit per processor. must be invalidated)
ModelToWorld	Matrix to transform from model to world coordinates.
WorldToView	Matrix to transform from world to homogeneous view coordinates.
ModelToView	WorldToView * ModelToWorld, maps from world coordinates to homogeneous view coordinates.
NormalToWorld	Matrix to transform the normal from model space to world space; equals the transpose of the inverse of ModelToWorld.
nx, ny, nz	Current vertex normal.
rgba	Current vertex color.
pInfo.p.Mode.	Mode bits, which pretty much all the stages need, are contained together in this substructure.
...draw	Whether or not to draw the primitive at all.
...lit	Whether to perform lighting modulation on the primitive (1), or leave it fully bright (0).
...useamb	Whether to use ambient light in the lighting calculations.
...usedir	Whether to use directional light in the lighting calculations.
...texmode	The color/texture blending mode. One of: 0, none; 1, color only; 2, texture only; 3, color/texture blend; 4, decal texture over color; 5, decal color over texture, or 6 color/texture modulated.
...texalpha	0 = none, 1 = soft, 2 = hard (alpha is clamped to 100% or 0%).
...colalpha	0 = none, 1 = soft, 2 = hard (alpha is clamped to 100% or 0%).
...colinterp	Whether to linearly interpret color, or just flat shade it.
...litinterp	Whether to linearly interpret lighting, or just flat shade it.
...texinterp	0 = nearest-neighbor mode; 1 = bilinear filtering mode
...outoforder	Whether to force a primitive to be considered out of order, no matter what its parameters are.
...textile	Texture tiling mode: 0 = none; 1 = repeat; 2 = mirrored; 3 = clamp.
...nousez	Do not check the z-buffer when rendering, just write to the screen.
...nowritez	Do not update the z-buffer when writing to the screen.
pInfo.p.SeqNum	The number of primitives that have gone, and the sequence number to give the next ordered primitive
pInfo.TextureID	Current Texture ID number.
pInfo.alphaThresh	Threshold to clamp alpha values in "hard alpha" modes.
pInfo.ambColor	Color and intensity of ambient light, stored as eight-bit fields.
pInfo.dirColor	Color and intensity of directional light, stored as eight-bit fields.
ldx, ldy, ldz	Directional vector for use with directional lighting.
ambreflect	Reflectance of the ambient light by the current primitive.
dirreflect	Reflectance of the directional light by the current primitive.
dirdefined	Whether the data for directional light has been defined yet by the controlling program
LaggedSeqNum	The sequence number given to out-of-order primitives, equal to one plus the last sequence number of an in-order primitive.

Table 4.1: Render State Variables Stored in the First Pipeline Stage

Because commands in command mode can return information to the external host after certain commands (such as reading the framebuffer or checking the amount of texture memory free), and these return values are also sent over the static network, there must be a kind of dynamic flow control in the static network's switch program (Recall from Section 3.3.1.1 that all routes that are simultaneously specified on one line of the switch program must go together, or they all will block, even if some are ready to go. If one is trying to route commands from the west to a processor, and from the processor back west with one line of switch code, then there must be data moving in both directions simultaneously for it to work. In order to take turns sending and receiving data, the switch processor must have some way of knowing exactly how many words are going to be transferred in each direction). Therefore, the switch program acts as follows: the first word from the external command source is routed to the processor, so the processor knows what command it is dealing with. Then, the processor sends to the switch the number of additional words to read in from the external source. The switch processor stores this number in a register, and then routes that many words from the west to the processor, decrementing the register until it reaches zero in order to do the counting. Next, the processor sends the switch the number of words that are to be returned from the processor out to the external host (this number, or the previous one, can be zero). The switch routes these words, and when finished, returns to the beginning of its cycle waiting for the first word of a command to route to the processor. All the commands understood by the processor, which modes they can be used in, and how many words of data they contain and how many are sent in return are listed in Table 4.2.

Command	CM	SS	Words	Returned
RENDER_BEGINSCENE	X		0	*
RENDER_ENDSCENE		X	0	*
RENDER_VERTEX		X	5	0
RENDER_COLOR	X	X	1	0
RENDER_MODELMATRIX	X	X	16	0
RENDER_VIEWMATRIX	X	X	16	0
RENDER_NORMAL	X	X	3	0
RENDER_SET_LIT	X	X	1	0
RENDER_SET_USEAMB	X	X	1	0
RENDER_SET_USEDIR	X	X	1	0
RENDER_SET_TEXMODE	X	X	1	0
RENDER_SET_TEXALPHA	X	X	1	0
RENDER_SET_COLALPHA	X	X	1	0
RENDER_SET_COLINTERP	X	X	1	0
RENDER_SET_LITINTERP	X	X	1	0
RENDER_SET_TEXINTERP	X	X	1	0
RENDER_SET_OUTOFORDER	X	X	1	0
RENDER_SET_TEXTILE	X	X	1	0
RENDER_SET_NOUSEZ	X	X	1	0
RENDER_SET_NOWRITEZ	X	X	1	0
RENDER_SET_TEXTUREID	X	X	1	0

Table 4.2: Rendering Processor Commands (continued...)

Table 4.2: (continued...)

Command	CM	SS	Words	Returned
RENDER_COLTEXBALANCE	X	X	1	0
RENDER_ALPHATHRESH	X	X	1	0
RENDER_AMBCOLOR	X	X	1	0
RENDER_DIRCOLOR	X	X	1	0
RENDER_DIRLIGHT	X	X	3	0
RENDER_AMBREFLECT	X	X	1	0
RENDER_DIRREFLECT	X	X	1	0
RENDER_CLEARFB	X		2	0
RENDER_CLEARZ	X		0	0
RENDER_SETPAGE	X		1	0
RENDER_FLIPPAGE	X		1	0
RENDER_ALLOCATE_TEXTURE	X		2	1
RENDER_DEALLOC_TEXTURE	X		1	0
RENDER_UPLOAD_TEXTURE	X		2+	0
RENDER_TEXMEM_AVAIL	X		0	1
RENDER_COMPACT_TEXMEM	X		0	0
RENDER_WRITE_FB	X		4	0
RENDER_WRITE_FB_BLOCK	X		4+	0
RENDER_READ_FB	X		3	1
RENDER_READ_FB_BLOCK	X		4	0+
RENDER_WRITE_Z	X		3	0
RENDER_WRITE_Z_BLOCK	X		3+	0
RENDER_READ_Z	X		2	1
RENDER_READ_Z_BLOCK	X		3	0+
RENDER_RESET	X		0	0
RENDER_HALT	X		0	0

Table 4.2: Rendering Processor Commands
This table lists the mnemonics for the rendering commands understood by the processor, whether they can be run during Command Mode and Scene Streaming, the number of words of data sent with the command, and the number of words the command expects in return. A + signifies a variable number of words, with the minimum listed. The * signifies that the ENDSCENE command does not require a return value, but that the command interface controller will pause after sending this command, waiting for the processor to send a word to let it know that it is okay to continue.

Commands with variable data sizes must be read in two passes. The first pass reads the minimum set of data, including the parameter that specifies the length of the rest of the data. Next, if there is additional data, the control tile reads the next word in (which the switch automatically routed to it), then tells the switch to route the rest of the data. The only commands with special flow control requirements are RENDER_END and BEGINSCENE, which can only occur during scene streaming, and that will halt and not send any subsequent commands until the processor has left or entered scenestreaming mode, signaled by a word being sent in response to the ENDSCENE/BEGINSCENE command.

The commands executable in command mode fall into three general categories: render state commands, texture management commands, and depth/frame buffer manipulation commands. RENDER_BEGINSCENE, which causes the processor to enter scene streaming mode, is in a category of its own, as are RENDER_RESET and RENDER_HALT.

4.2.2.1 Render State Commands

The commands from RENDER_COLOR to RENDER_DIRREFLECT in Table 4.2 simply modify the corresponding value in the render state (See Table 4.1. These commands can be used in either command mode or scenestreaming mode, and it is up to the programmer whether they find the ability to do state updates outside of scenestreaming mode useful. A changed state value affects all subsequent primitives and vertices until the value is changed again or the program sends a RENDER_RESET command. Note that the code for this project implements the render state commands separately for command mode and scenestreaming mode; this is due to the differing flow control algorithms used for the two modes.

4.2.2.2 Texture Management Commands

Textures are handled by the commands RENDER_ALLOCATE_TEXTURE through RENDER_COMPACT_TEXMEM. To add a new texture to the processor's memory, first the command RENDER_ALLOCATE_TEXTURE is called with a width and a height as parameters. The processor allocates that space in texture memory if possible, and returns a *texture ID* for use in subsequent references to that texture. If space cannot be allocated, it returns -1. Next, the texture data for a specific texture ID can be uploaded using RENDER_UPLOAD_TEXTURE. This can also be used to overwrite textures already in texture memory with new images. RENDER_DEALLOC_TEXTURE deallocates the memory for a particular texture ID to free up more space. RENDER_TEXMEM_AVAIL returns the total amount of texture memory free, and RENDER_COMPACT_TEXMEM attempts to compress fragmented free space in texture memory into a contiguous block[3].

Textures are stored internally as contiguous chunks of data in the texture memory space, with no alignment or size restrictions (beyond the 32-bit boundaries imposed by the Raw processor). The data are indexed in two ways: a linked list of texture allocations which specify the beginning and end addresses for all textures stored in memory, and an array of texture IDs with pointers to the beginning of each texture's data. The linked list is used to search for available space when allocating textures, and the array is used for quick texture lookup from an ID when rendering a scene. When a texture is deallocated, a space is left in texture memory where it used to be. Another texture may fit in that space, but over time, the free space can become fragmented and no longer usable (as textures need to be stored contiguously). The program controlling the 3D processor may decide to compact texture memory to make use of this extra free space, if necessary.

If more textures to render a frame than there is room for in texture memory, the scene rendering may be suspended by leaving scenestream, deallocating textures and uploading the new ones, and then resuming scenestream. Obviously this would have a negative impact on performance. It is expected that texture memory management will be handled on a lower level than the application software — perhaps in the API or driver for the 3D processor (See Section 4.3).

[3]Texture memory compaction is not implemented in this version of the 3D processor code.

4.2.2.3 Depth and Framebuffer Commands

The 3D processor also provides commands to allow the application to directly modify the framebuffer or depth buffer. These are not implemented in scene streaming mode because they complicate sequential consistency, and because only one tile can access the frame and depth buffers at a time and synchronizing the controlling tile with the tiles in Stage 4 would have been difficult.

RENDER_CLEARFB and RENDER_CLEARZ can be used to blank out the buffers before every frame. RENDER_SETPAGE and RENDER_FLIPPAGE can be used to manipulate the paging mechanism of the framebuffer, allowing drawing to go on in a "back buffer" while a steady image is sent to the screen. The FLIPPAGE command can also be set to wait for a VBLANK signal before flipping the page — the VBLANK signal[4] specifies that the beam of a CRT monitor has gone off the bottom edge of the screen and has not started over at the top yet. Flipping the page during VBLANK generates a better image than flipping while the beam is in the middle of the CRT, which can create a sort of "shearing" or "tearing" effect in the image.

The remaining commands allow direct reads and writes to the framebuffer (specifying a page, as well) and the depth buffer, either one pixel at a time or in a block of pixels.

4.2.2.4 Miscellaneous Commands

There are three other commands, which do not fit into the above categories. RENDER_RESET performs a synchronous reset of the processor, clearing texture memory and resetting the render state to the default. Asynchronous reset (i.e., resetting at any time no matter what else is going on in the processor) is not implemented, for reasons described in Section 3.3.4.1. RENDER_HALT simply halts the 3D Processor permanently (perhaps in later versions, it could enter some power saving mode). And RENDER_BEGINSCENE enters scene streaming mode, which is the topic of the next section.

4.2.3 Scene Streaming Mode

Scene streaming mode accepts vertex commands, and renders geometry using the fully parallelized pipeline configuration. Communication is very one-way in this mode: commands enter the processor with no return values, and are pushed down the pipes until the final image is rendered to the framebuffer at the bottom — this is why it is called the scene *streaming* mode.

When entering scenestream, most tiles do not have to do anything special, as they are basically always in scene streaming mode, but are simply waiting for incoming data to arrive. The upper left tile, however, must transition from command mode to scene streaming. It does this by changing its static switch program to the scenestreaming program, sending some initialization data to the switch, enabling GDN_AVAIL interrupts (which are used for part of the scenestream algorithm in the first stage), then sending a word out over the static network to tell the external command source that the BEGINSCENE command is finished, and it can start sending data again. The tile then jumps to the same code as is used in all the other first stage tiles during scene streaming.

[4] As of the writing of this thesis, the testing framework does not yet simulate a VBLANK signal, and this feature has not been tested.

4.2.3.1 Distributing Commands Among the Pipelines

The static network in the first stage (top four tiles) has the job of taking commands into the chip and distributing them among the tiles such that no tile is ever left idle if a new primitive is ready. The development of the algorithms to do so is described in Section 3.3.1.

In order to do this, the static network must be able to detect the transitions between primitives (which are in a way synchronization boundaries). To accomplish this, in scenestreaming mode the external command stream must be broken up into blocks. Each block contains one word which specifies the size of the rest of the block, and then a series of data words which make up the normal instruction stream. Block boundaries can occur at any place in the stream, and do not necessarily have to line up with commands. One exception is that a block of zero length must occur between command boundaries. This is because a block of zero length is a special marker which signifies the end of a primitive, and that the next primitive should be forwarded to a different processor.

The distribution algorithm in the static network delivers primitives in a relatively fair ordering: the scheme is generally round-robin but "skips" over processors that are currently busy to forward primitives to those that can process them. However, the static network is subject to some strict communication and synchronization constraints, as described in Section 3.3.1.2, and so a rather complex scheme is employed to produce the desired result.

Each switch processor has four major states: Idle, Active, Passing, and Counting (actually there are several more states, which are described shortly). Several of the states have two sub-states: one for when the tile processor is Busy, and cannot accept new commands, and one for when the tile processor is Ready for another primitive. Each state is located in a separate section in the switch processor's code, produces a certain routing pattern, and responds to events by switching to other states, again depending on the current state.

In the Active state, for example, the switch routes all incoming data from the west to the processor, while simultaneously monitoring it for an end-of-primitive marker. In the Idle state, the processor would do nothing, as all the current routing would be going on to its west, and would simply be waiting for a word to appear from the west signifying that something had happened. The processor would react differently depending on whether it was in the Idle-Ready or Idle-Busy state; for example, if the incoming event was a notice that the routing pattern had gone around and now it was that tile's turn to get a primitive, the switch would accept the turn if it was in Idle-Ready and head to the Active state. However, if it were in Idle-Busy, it would reject the turn and, depending on its location, might go to the Counting state. The Counting state is used to determine the next tile to route commands to. Basically, every tile sends words west specifying whether they have taken or passed their turn when it comes to them. The Counting state counts the number of passed turns so the tile knows when its own turn comes up, and it can take it or pass accordingly (Counting modes also have Busy and Ready sub-states). If a tile to its east ends up taking its turn, then the current tile has to start routing commands west — this is the Passing state. In this state, the switch is also monitoring the data for an end primitive command, so that it knows when the routing pattern has to be rearranged, by returning to the Counting state.

The Counting and Passing states only occur when the active tile, or tile whose turn it is, are to the current tile's east. When the active tile is to the west, the processor remains in an Idle state, simply waiting for a message to arrive from the west. Note that the westernmost tile in the top row will never be in an Idle state, as it is always either the active tile or Passing/Counting due to an active tile to its east. Its tile code will have to jump directly

from the highest count in the Counting state to deciding whether to take the turn or not, as the turn transfers from the last tile in the row to the first. Likewise, the last or easternmost tile in the row has no Counting or Passing states, and only alternates between Active and Idle. The middle tiles remain Idle until they get a message from the west, then it is their turn, then after their turn they Count or Pass for a certain number of tiles to their east before returning to Idle mode.

In other words, tile 0 goes from Active/PassTurn to Counting/Passing to Counting/Passing to Counting/Passing and back to Active, where each transition happens either on an endprimitive, or on the signal that a tile down the line passed its turn. Tile 1 goes from Idle to Active/PassTurn to Counting/Passing to Counting/Passing and back to Idle. Tile 2 goes from Idle to Active/PassTurn to Counting/Passing and back to Idle. Tile 3 simply goes from Idle to Active/PassTurn and back. There is, of course, additional state needed: all of these modes except for Active need to have sub states for their tile processor being Ready and Busy. Also, the Counting and Passing states need some way to count, and this is accomplished by the use of sub states as well. Finally, transitional states such as TakeTurn, PassTurn and DoneActive make the state machine more manageable.

One additional detail is that, after an end primitive, the just recently active tile must wait for the tile processor to signal that it is okay for another tile to start (since shared memory must be flushed, etc.). It uses this signal (just a word sent from the processor) to send to the next tile to the east to wake it from Idle mode. An exception is the last tile in the row, which must send a signal to the first tile in the row when this occurs, and the first tile must not take its turn until it receives that signal. Therefore, when all the tiles have detected the endprimitive while Passing commands to the last tile in the row, they can then route the okay-to-go signal east to west from the last tile back to the first before heading into Idle state. A state transition diagram for a typical middle tile's switch program is given in Figure 4-2. The first and last tiles' programs would be modified as described in this and the previous paragraphs. Notice that this code can be extended to any number of processors by increasing the number of Counting and Passing states that each tile goes through. One could conceivably even write a script which automatically generated such code for any processor width desired.

Finally, to signal to the switch that the tile processor is now ready to accept new commands, the tile processor forcefully changes the switch's program counter to point to the new state. The code to do this has to be careful, though, to move the program counter to a location that is performing the same routing as the original. Therefore every state that can be interrupted from Busy to Ready must have a line-for-line equivalent state to be transitioned to. Table 4.3 lists all the states in the final implementation of this code and some of their nitty gritty details.

When scenestream ends, the switch programs must return to their starting points. Scenestream ending is discussed in more detail in the next section.

4.2.3.2 Stage 1

After commands start arriving from the static network, the active processor first invalidates the Update word in the render state and checks one bit which corresponds to its processor number. If the bit is 1, that means the render state has been changed and it must invalidate its entire cached copy. If it is zero, then the tile can continue to use the copy of the render state in its cache, though the Sequence Number values (SeqNum and LaggedSeqNum) need to be invalidated each time. The tile then sets its bit in the Updated word to zero, clears

State Name	Description
Idle_Ready	Route once E→W (or, in the last tile, Processor→W). (this route is skipped on startup, and at certain other times). Wait for token from W. On token, go to Take_Turn.
Idle_Busy	Route once E→W (or, in the last tile, Processor→W). (this route is skipped on startup, and at certain other times). Wait for token from W. On token, go to Pass_Turn. On Proc_Ready, go to Idle_Ready.
Active	Route data W→Processor, until End_Primitive. On End_Primitive, go to Done_Active (unless it is the last tile, then go to Idle_Busy)
Done_Active	Wait for word from processor. On data, route the word→E, and go to Counting_1_Busy. On Proc_Ready, go to Done_Active_Ready.
Done_Active_Ready	Wait for word from processor. On data, route the word→E, and go to Counting_1_Ready.
Passing_N_Ready	Route data W→E, until End_Primitive. On End_Primitive, if this is the max N for the tile, go to Idle_Ready (or if it is tile 0, Take_Turn). If it is not the max N, go to Counting_N+1_Ready.
Passing_N_Busy	Route data W→E, until End_Primitive. On End_Primitive, if this is the max N for the tile, go to Idle_Busy (or if it is tile 0, Pass_Turn). If it is not the max N, go to Counting_N+1_Ready. On Proc_Ready, go to Passing_N_Ready.
Counting_N_Ready	Wait for data from the east. On data, route it west. If the data was non-zero, go to Passing_N_Ready. If it was zero, and this is the max N for the tile, then go to Idle_Ready, skipping the first E→W route (or if it is tile 0, Take_Turn, skipping the wait for token). If this is not the max N for the tile, go to Counting_N+1_Ready.
Counting_N_Busy	Wait for data from the east. On data, route it west. If the data was non-zero, go to Passing_N_Busy. If it was zero, and this is the max N for the tile, then go to Idle_Busy, skipping the first E→W route (or if it is tile 0, Pass_Turn, skipping the wait for token). If this is not the max N for the tile, go to Counting_N+1_Busy. On Proc_Ready, go to Counting_N_Ready.
Take_Turn	If this is tile 0, wait for a token from the east. If this is not tile zero, send a non-zero number west. Finally, go to Active.
Pass_Turn	If this is tile 0, wait for a token from the east. Send a token east. Next, if this is not tile 0, send a zero west. Finally, go to Counting_1_Busy (unless it is the last tile, then go to Idle_Busy, skipping the first route). On Proc_Ready, go to Take_Turn.

Table 4.3: Full State Transition Table for Stage 1's Switch Code
This table summarizes the complicated state transitions involved in the first stage routine. Proc_Ready occurs when the tile processor signals the switch that it can accept new data.

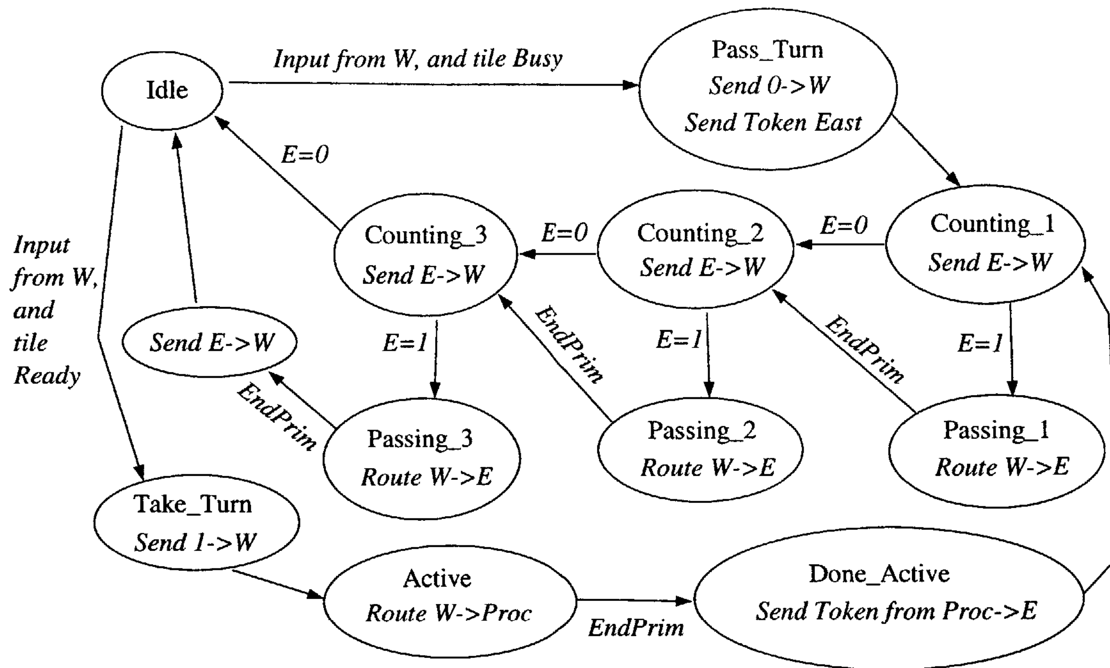


Figure 4-2: State Transition Diagram for a Typical Stage 1 Switch Processor
See text for complete discussion.

some variables, and starts to read and execute commands from the static network (including the block word counts in the stream), flushing any state variables that are changed from the cache. When it arrives at an endprimitive command, it stops reading commands, and processes the primitive (or perhaps end scene command) it received.

First, if the command was an endscene command, it initiates a flush of the pipeline. The pipeline flush starts by sending a GDN message to all the other Stage 1 tiles, containing the flusher's tile number in the message. These GDN messages will trigger interrupts on the receiving tiles, which will set a flag saying that a flush request was received. After the tiles are done processing their current primitive (or immediately, if they are currently idle), they will check the flush request flag and send a special pipeline flushing primitive, which includes the tile number of the tile that initiated the flush, to the next stage. This pipeline-flushing primitive will travel down to the last stage, at which point the last stage will send a response back to the initiator of the flush via the GDN. When the initiator has received all four responses (it also sends its own pipeline-flushing primitive down the pipe), then it knows that the flush is complete, and there are no more in-flight commands in the pipeline.

After flushing out the pipeline, it sends another GDN message to every Stage 1 tile with 0xFFFFFFFF as the data, which signifies an end-of-scenestream event. When the tiles receive this message, they reset the static networks and leave the "DoSceneStream" function — which does nothing but cause it to be called again in a loop on all the tiles except for the first (westernmost) tile, which returns to command mode. The endscene command itself causes the external command interface controller to wait for acknowledgment from the processor before sending any more commands in, guaranteeing that the static network is empty when it is reset to command mode.

Note that an endscene command still needs to be followed by an end primitive marker,

so that the tile will process it and not be stuck in command-reading mode.

If the command was not an endscene, then it must have been a command to render a triangle, or primitive. Processing only occurs if at least 3 vertices were sent (any less would be a programmer error on the side of the external application, and cause the tile to discard this primitive). If there were at least three vertices, then the code calculates whether the primitive should be considered "unordered" or "ordered" (see Section 3.3.2). Unordered primitives can be rendered in any order among themselves, because they use the z-buffer and have no partially transparent bits. Ordered primitives must be rendered in the exact order they are sent in as commands among themselves, and have to be in the same relative position versus any unordered primitives. The code uses a conservative heuristic to determine the ordering status of the primitive — if its blending mode and the alpha values of its vertices could ever possibly lead to a situation with partially transparent fragments (fully opaque and fully clear are okay), then it is marked as ordered. The sequence number and "lagged sequence number" (the number of the last ordered primitive plus one, given to all subsequent unordered primitives) are updated, and flushed from the cache. Finally, in the case that this primitive is going to be sent while SeqNum is rolled over (detected when SeqNum = 0, as on boot-up SeqNum starts at 1), then the tile flushes the pipeline clear before continuing (using the same method as the pipeline flush on endprimitive described above). After all this, then the tile sends a word to the static network letting it know that the next tile can begin processing commands. (How the sequence numbers are used to ensure correct ordering in the last stage of the pipeline is described in Section 4.2.3.5).

Finally, the stage performs projection, visibility culling, perspective division, and streams out the new primitive south. If the primitive turns out to be not visible or culled, dummy primitive info is streamed south anyway. This info has the ModeBit "draw" set to zero, and further stages do not bother doing any computation on it; they simply pass it south. The last stage uses it to increment the sequence number, but nothing else. When Stage 1 passes its primitive info south, it sends it over the GDN, in messages no greater than 31 words (a restriction imposed by Raw), because the static network is too busy with the command distribution algorithm to be able to easily handle the primitive streaming as well.

4.2.3.3 Stage 2

Stage 2 needs to perform very little above its basic rendering algorithm in the parallel pipeline. It basically reads in the primitive info from Stage 1, sends some of that primitive information (such as rendering modes) south over the static network to Stage 3, then proceeds to rasterize the primitive, interpolating whichever variables are necessary according to the primitive's render mode, and sending the fragments south, also over the static network (For more info on rasterization and variable interpolation, see Section 2.2.2). Before each fragment, it sends a marker value specifying that a fragment is coming up next. This is to differentiate it from the case where the primitive has finished, and a new batch of primitive info is coming up next, which uses a different marker. Also, the each primitive is preceded by a marker specifying whether it's actually a primitive, or a one-word-long pipeline flush command (which simply gets forwarded south).

4.2.3.4 Stage 3

Stage 3 also does not need to do much beyond its basic rendering algorithms (which are described in Section 2.2.3, and include looking up texture values and blending together the final color for the fragment from its interpolated color, texture and light values). One exception is ensuring that shared texture memory is cache coherent, while still taking advantage of the cache for multiple lookups of the same texture (which is common in 3D scenes, especially with tiling and bilinear filtering. — for more discussion of the benefits of texture caching, see [7].) It does this via two mechanisms: first, whenever there is a pipeline flush, it invalidates the texture ID array, just in case the processor had gone into command mode and added or removed textures. Secondly, there is a word in each texture ID entry that specifies if it has been updated yet since the last invalidate, with one bit per processor in Stage 3. When a processor looks up the texture, it invalidates it if this bit has been set, and then clears the bit. This way, each processor in the row gets a chance to load the new texture data into its cache if there is any change.

This stage receives primitive info and fragments from the prior stage over the static network, and sends some of that primitive info and textured/blended fragments south to the next stage, also via the static network. It uses the same system of markers before each data structure (i.e., primitive info vs. pipeline flush, new fragment vs. end-of-primitive). However, because it is asking the static network to do both sending and receiving, it must use some kind of flow control. It ends up doing something similar to the control tile in command mode, where the switch sends the first word it receives from the north, then waits for a count of subsequent words to send from the north. After sending those words to the processor, the switch waits for a count of words to send south. The processor sends those words out, and the cycles repeats.

4.2.3.5 Stage 4

Stage 4 takes the textured fragments sent from Stage 3 and combines them into the depth and frame buffers. However, only one tile in stage 4 can access the depth and frame buffers at any time. To solve this mutual exclusion problem, a one-word "token" is passed around the tiles in a ring. Whichever tile has the token can access the shared resources, and sends the token to the next tile when it is done with them. The token passing is generally handled by a GDN_AVAIL interrupt routine, which, upon receiving the GDN token, checks a global variable ("taketurn") to see if the normal tile code wants to take its turn yet. If so, then it puts the token into another global variable (as its value is used as described below), and resets the taketurn flag, which the tile code would typically be spinning on. If not, then it simply sends the token to the next processor in the chain (note that if the tile code successfully acquires the token, then it is its responsibility to send the token along when finished).

The value of the token is also the current sequence number for Stage 4. This ensures sequential consistency by only allowing "ordered" primitives to be rendered when the sequence number is exactly equal to the sequence number stored in the primitive, and only allowing "unordered" primitives to be rendered if the sequence number is equal to or greater than the primitive's number, for reasons described in Section 3.3.2. When a primitive is finished rendering, it increments the sequence number (unordered primitives may give up the token before they are finished rendering, in which case they leave the sequence number unchanged) and send it to the next tile.

The code in the last stage was heavily tweaked to try to keep the token for as short a time as possible, but do as much work as possible during that time. Also, it tries to keep the previous stages from stalling unless absolutely necessary. When rendering unordered primitives, while a tile does not have the token yet (or its sequence number has not arrived yet), the tile buffers as many fragments from Stage 3 as possible, while simultaneously waiting for its turn to come around (as long as it has at least one fragment buffered up). It accomplishes this by busy-waiting on multiple variables (namely, the taketurn flag and the number of entries in the static network's input buffer). As soon as it gets its turn, it runs through all the fragments on its buffer as quickly as possible, invalidating and checking the z-buffer values, updating and flushing with new z-buffer values, reading from the frame buffer, and blending and writing to the frame buffer all as necessary. Several attempts were made to restructure the code to try to make it run a bit faster, such as copying the value of a "volatile" variable to a non-volatile one when it was known that the value would not be changed, and explicitly pulling out commonly used expressions that the compiler optimizations did not catch. If time constraints had allowed it, a hand-coded assembly version of the inner loop would have been useful — in fact, hand-coding all the inner loops in the project might have been a good idea, see Chapter 6.

Ordered primitives do not do this buffering, but rather render the fragments as soon as they come out of Stage 3, as only one primitive will be rendering at a time[5].

When interacting with the framebuffer, the tile has to be careful to disable the GDN_AVAIL interrupt before sending a read to the framebuffer, as the framebuffer responds via the GDN.

Finally, when Stage 4 gets a pipeline-flushing marker, it sends a GDN message to the originator of the pipeline flush, as described in Section 4.2.3.2.

4.3 Interfacing the Graphics Architecture in a System

This whole paper so far has made allusions to the frame buffer and the command generating interface, but not much has been spoken about how these interfaces would couple the 3D processor into an entire system. That is the purpose of this section. An illustration of all the components of a full system is given in Figure 4-3.

4.3.1 The Framebuffer Side

The "framebuffer" interface on the south of the Raw chip is actually a simplified interface, tailored to Raw and implemented on some glue logic outside of the Chip[6]. The framebuffer controller would handle GDN messages and other aspects of the "framebuffer" protocol, and would translate these into actual reads and writes to an actual framebuffer — implemented as high-speed VRAM connected to a Video DAC. The Video DAC would continuously scan through VRAM (VRAMs are usually dual-ported) while the Raw was also updating its image, and convert the stored colors to an analog signal to be sent to a display or monitor. The VRAM would be dual-paged, and the controller could select that the DAC display one page or the other. A complete implementation would also allow the video to change

[5]Although buffering could have been used for in-order primitives as well, letting them queue up values to render while waiting for their turn.

[6]The original plan for this thesis included implementing this glue logic in an FPGA, but there was not enough time to actually do so, and it was a very non-critical part of the project.

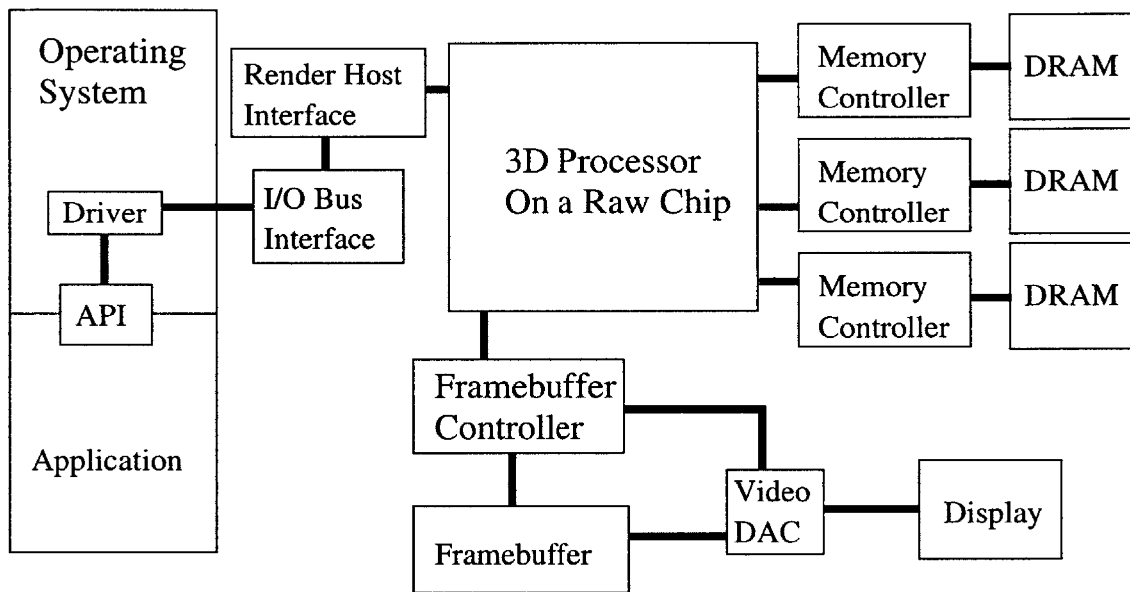


Figure 4-3: The Processor in a System Context
The Raw Processor, being used as a 3D graphics processor as described in this paper, is shown here integrated into a host system. Its host interface communicates with an I/O bus interface, which in turn communicates over the bus to the host processor. That host processor may be running an operating system, which includes a driver for the 3D processor. The driver is used by software applications through an API. The 3D Processor is also connected to the framebuffer controller, which in turn drives the framebuffer and a video DAC (Digital to Analog Converter) to display the image on a screen.

resolutions and modes, but this implementation remains fixed with a 32-bit color mode with compile-time video resolution.

4.3.2 The Host Side

This processor receives its rendering commands from an external host, through the render host interface attached to the north-westernmost tile. This interface is also connected to glue logic, which takes the raw rendering commands and both performs static network flow control as well as flow control and blocking required by the higher level 3D processor input algorithms. The interface would receive commands from some host-specific bus or I/O channel, such as PCI or AGP, and could implement things like DMA based texture transfer.

On the other end of the I/O channel would be a controlling host processor. Basically, this processor is running a program that is generating rendering commands for the 3D processor. However, the software-side approach is typically layered. There is an operating system, which controls I/O accesses in general, a driver, which controls the I/O interfacing with this 3D rendering device in particular, an Application Programming Interface (API) which presents a higher level view of 3D rendering to user applications (e.g., one that did automatic texture management or had higher level primitive calls), and finally the application itself, which determines what all the other components are going to be doing.

The point in looking at the whole system's approach is that, the abilities and performance of this device cannot be truly evaluated unless studied as part of an entire system. For example, it is quite possible for drivers, for instance, to implement optimizations that the hardware does not implement (such as selectively setting primitives to be outoforder if they do not overlap), or for the VRAM memory technology to determine the maximum pixel fill rate. However, there was not enough time in this project to simulate such a full system's approach, so (as seen in the next chapter), simply the two immediate interfaces to the Raw chip are simulated (along with a simplistic VRAM model).

4.4 Summary

This chapter has described in more detail the implementation of the various parallelization, synchronization and communication techniques used in this project, and how they fit together to make the final 3D Graphics processor. Also described was how such a processor would fit into a total system from the application end to the video display. The next two chapters analyze the actual performance of the implementation of this chip, suggest some ways that it can be improved, and generally talk about the lessons learned in implementing this architecture.

Chapter 5

Testing, Validation and Performance Results

This chapter describes the testing and verification methods used with the project described in the last several chapters. The actual code for much of the testing framework is attached in Appendix D. Verification and performance results are then presented, and the performance results are analyzed. Additional images from verification are available in Appendix A. Chapter 6 expands on the results in this chapter to describe some suggestions for improvement over this design and possible future work.

5.1 The Testing Framework

This section describes the framework used to test this architecture, which was used for development, debugging, feature validation and performance measurement. Although the Raw hardware that this processor targeted is available, the project itself was never implemented on actual hardware, and instead was completed and tested solely in simulation, using a simulator known as BTL [18].

5.1.1 The BTL Simulation Environment

"BTL" is a cycle-accurate simulator/debugger for the Raw architecture. It is highly extensible [19] through a programming language known as "bC." Pretty much any aspect of the simulation can be changed through bC code, however it is most useful for simulating external I/O devices that interact with the Raw processor, or linking them to other processes running on the computer, such as a Verilog simulation. The bC code can also examine any aspect of the running processor's state and generate detailed profiling information.

The BTL simulator was used through a build environment known as "starsearch." Starsearch is primarily aimed at keeping an array of regressive tests to be used to verify the build environment, including the BTL simulator itself, but it also provides a very simple interface to build and get Raw programs up and running quickly. What would otherwise be a rather complex build process — involving compiling and linking several programs, packing them together into a boot module, along with the boot program and other helper code that needs to be compiled for the particular Raw geometry in use, and starting up BTL to simulate the program (or downloading it to an actual Raw testboard for real-world verification) — is presented as a simple makefile interface with commands such as "make

run" and "make debug," with easily customizable makefiles for each project.

To add devices to a BTL simulation, one must specify a "machine file," which is simply a bC code file which is executed at the beginning of simulation, and which is supposed to set up all the devices that are attached to the Raw chip. A default machine file is provided with BTL, which has the capability to produce memory modules, a "print service" for quick debug output, a "host interface" for allowing the Raw chip to make system calls on the host computer (not to be confused with the render host interface described in this thesis, which supplies commands to the 3D Processor), a PCI interface, and several other possible devices. A "device" in bC code is really a subroutine which is cooperatively multitasked with the rest of the devices in the simulation, where each device is run until it yields, then the simulation is advanced one clock, then the devices are run again. Each device can access any point of the Raw simulation, and send and receive data on whichever Raw I/O ports it wishes, though it will generally stick to the point to which it was assigned, as they are generally made to simulate real devices.

The verification of the 3D processor built on top of Raw uses two major bC devices — one to simulate the frame buffer controller, and one to simulate the render host interface that streams commands into the Raw chip. (Actually, there are two more bC devices used in the framework: the framebuffer itself is simulated as a separate device from the framebuffer controller, and another device exists solely to gather performance data from the running simulation.)

5.1.2 Framebuffer Controller

The framebuffer controller accepts commands from the GDN for reading and writing its memory, as well as commands that flip the active buffer and back buffer. It interfaces with a separate device which implements a very simple RAM-like interface, and stores the actual values of the framebuffer for retrieval. The reason for this split was to make it easier to record the signals coming in and out of the framebuffer controller itself for later verification tests against a Verilog model, but this idea never came to fruition.

The framebuffer's default size is 640x480, though it can be changed via command-line arguments to the BTL simulator (practically all of this project used 320x240 for a working resolution). Also, the framebuffer controller can optionally (again, specified by command line arguments) display on the screen an image of what is currently in the framebuffer's memory. This image can also be updated in realtime, and the switch to do so can be toggled on and off by the user of BTL while in an interactive debugging environment using render_realtimeupdate_on() and render_realtimeupdate_off() (since any BTL function can be called from the command line in that environment). Also, due to the slowness of the rendering method in X window, the entire framebuffer image is not refreshed if it is covered up by another window and then uncovered — the refreshing has to be done manually by the user using render_refresh_display().

GDN messages sent to the framebuffer controller consist of a standard GDN header with no extra information in it, which is followed by a special framebuffer header which contains a 3-bit command, a 2-bit page specifier, and a 19 bit offset address into video memory, which can support just above 640x480x32 bit resolution, which is enough for this project (though a more robust product would support higher resolutions). The layout of this header word, along with the meanings of the different bit combinations, is given in Figure 5-1. The message may contain subsequent words depending on the particular command: read single, page flip, and reset commands have no further data; write single carries a second word:

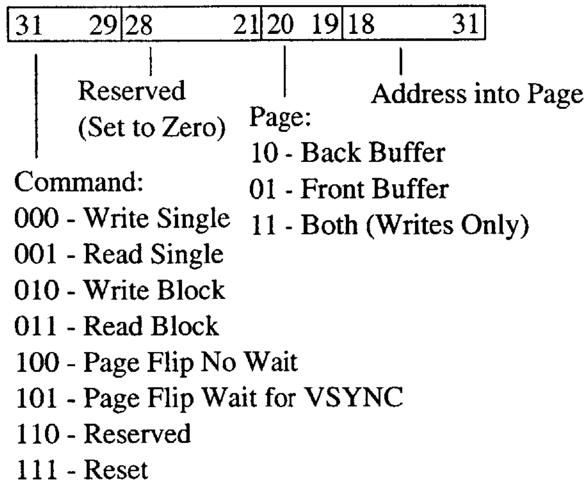


Figure 5-1: Framebuffer Message Header

the color data to write; read block's second word is the length of the block to read; finally, write block sends a word for length, and then the data to write and the message may go all the way up to the raw-imposed limit of 31 words in the message[1]. Additionally, for the commands read single, read block and page flip+wait-for-vsync, the framebuffer controller will construct a GDN message and send it back to the original requesting tile (as garnered from the GDN header in the original message). After the requisite GDN header, the read single command sends back the read word of data. The read block command simply sends back multiple read words of data, depending on the length specified in the original command. The flip+wait-for-vsync command waits for VSYNC, then flips the page and sends a one-word GDN message back to the requesting tile (the word is incidentally the same framebuffer header that was the original page flip request).

Internally, the framebuffer controller then updates or reads to/from the framebuffer itself using a simplified RAM interface with 5 control bits, a data bus D and an address bus A. The control bit PAGE specifies which page to display on the monitor: 1 or 0. nCS0 and nCS1 are low-active chip select signals, which select the bank for page 0 or 1 for reading and writing (though only one bank should be selected for reading at a time, as they share address and data busses). nWE specifies a low-active write enable, and nOE specifies a low-active output enable. Again, only one should be active at any time as both reads and writes are performed on the same data bus. The VRAM would supposedly have an independent data bus that attaches to a Video DAC for output to a monitor. In the simulations, the RAM is synchronous, but it does not have to be — this is simply a side-effect of the fact that the BTL simulation itself is synchronous.

[1]Note that this 31-word limit only requires 5 bits to store the length. Therefore, sending a whole 32-bit word for the length is overkill, and that value would fit much more snugly into some of the 9 bits that are not used in the framebuffer's header word. Block read/write mode is not used in the current implementation of the 3D Processor, but it is a viable performance optimization, and removing one more word of overhead would make it that much more efficient.

5.1.3 Render Host Interface

The render host interface, in a real-world implementation of this architecture, would map between some sort of host-specific communication, such as an AGP or PCI bus, and the static network stream of commands that the 3D Processor expects. In the simulation framework, the host interface simply takes a pre-defined stream of render commands, and sends them into the static network. The host interface also responds to certain "meta" commands, which instruct it to do certain things other than streaming into the processor, such as waiting for the processor to send a word back before continuing, printing out debugging information or halting the simulation. In a sense, then, the simulated render host interface is much less intelligent than a real host interface controller would have to be — practically all of the flow control details of the static network interface to the 3D Processor (such as waiting for responses after certain commands, and sending commands during scene streaming in "blocks") are coded into a pre-defined input sequence rather than generated by the interface logic itself.

5.1.3.1 External Program Control

Originally, the test input sequences would be taken directly from files. However, this would have been tedious, inflexible and hard to debug. Many 3D rendering tasks lend themselves to programmable control — for example, rendering a scene with multiple instances of the same object can be done by making a subroutine that draws the object, then calling that subroutine multiple times with different ModelToWorld matrices set. Describing a scene often benefits from looping and modularity that cannot be exploited by using a raw stream of the rendering commands. Additionally, commands that return data to the controlling program can be tested more accurately by having a program that actively makes a decision on the returned data. Finally, for something like performance measurement, where a large set of different combinations of render modes must be tested, being able to programmatically cycle through all the combinations is a must — a static command file could become untenably huge to maintain in such situations.

One solution is to write programs to automatically generate a command file, and then use those files as inputs. But for maximum flexibility, it was decided to simply spawn a controlling process at the same time as the BTL simulation was running, and use pipes[2] to connect the output of the process to the input of the render host interface, and vice versa. In this way, a program can communicate completely interactively with a simulated version of the 3D processor. To further the abstraction, a header file was written (render_client.h), which wrapped the low level pipe communications up into a series of procedure calls resembling a reasonable 3D rendering API. Test programs could be easily developed in this API, making validation and performance testing much more efficient.

The actual data transferred over the pipe is formatted as a series of 11 byte ASCII strings. By default, the string was a hexadecimal number representing the word to send over the static network next, such as "0x1234ABCD" (the 11th byte allows for a newline after the string). The interface is not very lenient or resilient to errors, so the 11 byte alignment must be strictly kept. Along with hexadecimal values, several "meta" commands are recognized by the render host interface, which instruct it to act in special ways. These are summarized in Table 5.1.

[2]A "pipe" in Unix is simply a channel through which two separate processes can communicate. What one puts in one end the other will read out the other end.

Meta Command:	Description
wait******	Causes the interface to wait for a word from the 3D processor before continuing.
halt******	Causes the interface to interrupt the simulation.
read******	Immediately after, program will send a hex number specifying number of words to read. Then interface sends that many words from static network to the external program.
debug*****	Next 11 characters after this command are printed to the simulator's output to be read by the operator or logged.
pstart****	Starts recording performance profiling data (see next section).
pstop*****	Stops recording performance profiling data
penterss**	Marks the beginning of scene stream (for performance prof.)
pexitss***	Marks the end of scene stream (for performance prof.)
pframe****	Marks the beginning of a new frame (for performance prof.)
pprim*****	Marks the beginning of a new primitive (for performance prof.)
preport***	Prints out a report of gathered performance prof. statistics, to be read by operator or logged.
preset****	Resets all performance counters and state.

Table 5.1: Summary of Meta Commands for the RenderHost Interface
These commands instruct the renderhost bC device to carry out special actions relevant to debugging and simulating the hardware. They are inserted in the stream of hexadecimal numeric values which specify actual words to send over the network.

All the commands that begin with "p" are used for performance profiling, which is the subject of the next section.

5.1.3.2 Performance Profiling

The render host interface code, in addition to providing communication between the simulated 3D Processor and an external controlling processor, also serves to monitor the performance of the processor, under the direction of certain meta commands, in order to report on the results of performance tests. It collects data for the performance comparison in three ways:

- Directly recording number of cycles passed, number of cycles in scenestreaming mode, and number of primitives and frames, all from its own observations or the meta commands that specify these things. Also, code in the render framebuffer bC device keeps track of the number of pixels rendered, and shares it with the render host's data collection.

- The render host spawns a new device whose only purpose is to monitor all the tiles in the processor to see which are actively running commands and which are currently blocking on something. This is part of the way the profiling code records active cycles in the code (vs. total cycles), from which the processor utilization is estimated

- "magc" instructions are Raw assembly instructions that cause the simulator to hook into bC code. Some magic instructions are used to count internal event occurrences, such as the number of drawn primitives (those that are not discarded after the first

stage), the number of fragments generated by the second stage, the number of texels drawn by the third stage, as well as the number of textured fragments sent to the fourth stage (which may be less than the number of untextured fragments in case the blending rendered it 100% invisible. Also, generating active cycles by looking at whether the processor is blocked or not is not completely adequate, as the processors in this design spend a good amount of time busy waiting, which shows up as active when it should be waiting. To solve this, two more magc commands are used to signify the beginning and end of a busy waiting period that should not be counted towards active cycles.

In the end, the render host reports the total active cycles for each stage in the pipeline. Dividing by four gives the average active cycles per processor in each stage, and dividing by the total cycles that have passed gives the average processor utilization for each stage.

5.2 Feature Validation

Feature correctness validation, for the most part, was a simple matter of writing programs in the above framework to test various features, and check to see if the visual output was correct. This was a very informal process; however, it was decided that an incredibly rigorous feature verification scheme was not necessary for this project. The point of this project was to examine one possible way of parallelizing rendering computations on the Raw architecture, and to look at the performance and design tradeoffs. Of course, the various rendering features had to be pretty much correct in order to make sure that no "cheating" was going on, increasing performance at the price of correct operation, and also to make sure that all the details necessary to make a 3D processor with this feature set were fully considered. In other words, if a certain performance claim was made, good or bad, about a feature that actually did not work correctly at all, then that performance claim is basically made bogus and incomparable to other systems. Therefore, general feature correctness was a goal, although complete elimination of bugs was not attempted[3].

Near the beginning of the development and debugging cycle, while the code was first being implemented, the biggest verification challenge was to get the processor to run through all the commands without locking up due to a synchronization error. During this time, the basic triangletest.c code was developed, which featured several triangles of varying sizes and colors rotating around the screen (the triangletest code eventually evolved into code to perform the complete performance analysis of the processor, as described in Section 5.3).

Once basic issues of synchronization and lockups were addressed, general feature correctness tests were performed, such as displaying triangles with various texture modes with texturetest.c, opaque and transparent triangles rendering in the correct order relative to each other with ordertest.c, and finally full 3D scenes with depth and lighting and various transformations, such as cubetest.c. A few screenshots of these tests are shown in Figure 5-2; several more appear in Appendix A. The correct performance of these tests, plus the fact that the performance test could run for days and did not lock up (and seemed to be producing the correct images) was enough to satisfy the lax requirements for correct operation of this project, and most likely weeded out the larger show-stopper type bugs completely.

[3]One good (or bad, depending on your perspective) thing about using a reconfigurable architecture like the Raw or an FPGA is that bugs can be fixed "in the field" without having to replace hardware, making the pressure to produce a perfect first iteration much lower.

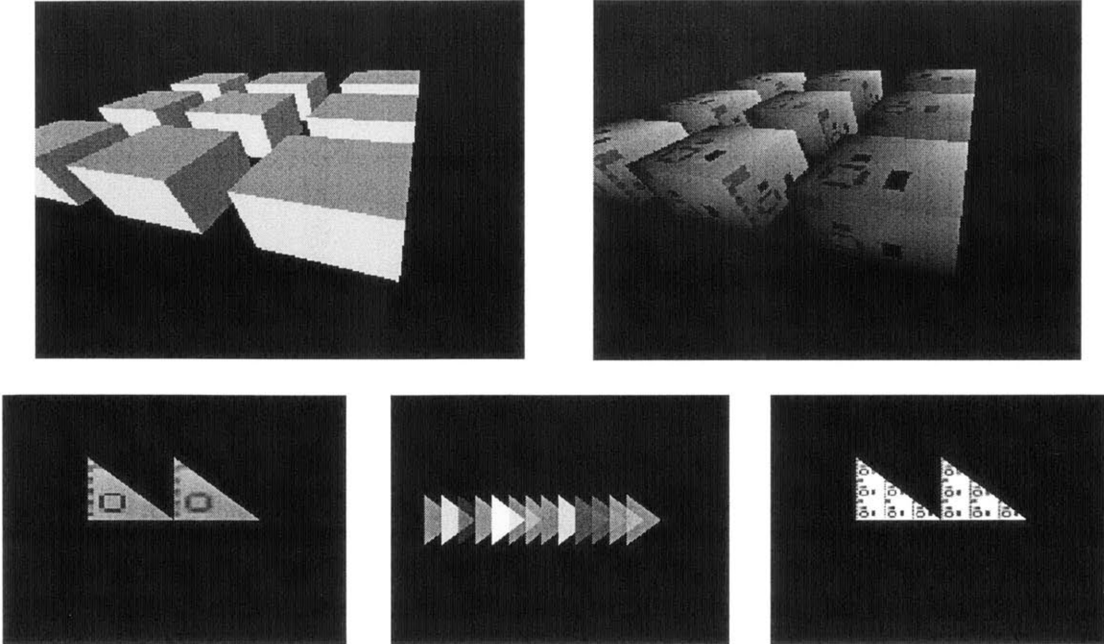


Figure 5-2: Sample Screenshots
Top row: Untextured, unlit, flat shaded cubes, and then fully textured and lit cubes.
Bottom row: Texture test showing bilinear filtering, a test for correct ordering of transparent primitives, and another texture test showing tiling.

5.3 Performance Results

As mentioned previously, the program used to generate performance data descended from a simple rotating triangle test program. In fact, it uses the same triangle pattern to do the test (Figure 5-3), except it draws the pattern four times (for a total of 42 primitives), and does not rotate it. It does, however, scale the pattern in order to get data for the effects of different primitive sizes on the screen, using sizes 1.0, 0.5, 0.3, 0.2, 0.1, 0.05, and 0.01 times the original size. The largest primitive in the largest size takes up roughly one eight of the screen, while taking up only one pixel in the smallest scaling (all the other primitives in the test pattern generate no pixels at at that scaling). The test script runs through all sorts of combinations of render modes (though not every possible combination, as that would create an untenably huge amount of data): texture mode is varies between COLOR, TEXTURE and BLEND (other modes are approximately the same computation-wise as blend); texinterp, which chooses between nearest neighbor and bilinear filtering; whether the primitives should be lit (with both ambient and directional light); colinterp, and litinterp, which choose between flat shading and interpolated shading; whether the primitives should be transparent (they are either all transparent or all opaque in the test); nousez and nowritez, which determine the usage of the depth buffer; and finally unordered, which forces normally in-order primitives to render out-of-order anyway, and only has an effect if the primitive is transparent or one of nousez and nowritez are on (also, it does not have an effect on the single tile implementation). Every combination of these modes is simulated with every primitive scaling factor listed above. In addition, the entire test is run

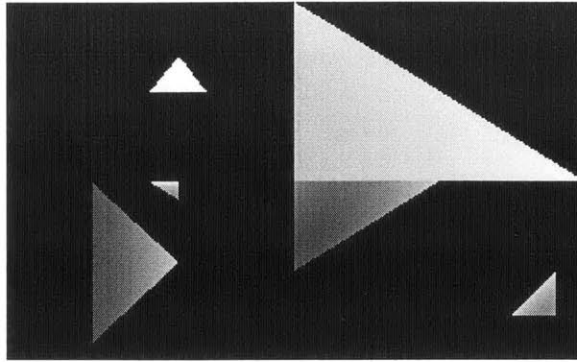


Figure 5-3: Triangle Performance Test Pattern

both on the full implementation of the processor and on the single-tile implementation[4]. This all generates a large amount of data, for which a Perl script was written to process and generate tables of the information in various relationships.

After as much performance data about various modes had been recorded, the next task was to analyze the data for certain meaningful relationships and performance results. One important metric is overall speedup with the 16-tile parallel configuration versus the single-tile implementation of the same algorithms, which will tell us generally how good the parallelization was. Ideally, the speedup will equal the number of processors times the processor utilization — this should be checked against the actual numbers and any differences analyzed. The processor utilization per stage can also be used to determine where the bottlenecks are, and why the system achieves less-than-ideal speedup. The speedup and utilization is expected to change according to the test parameters, as different stages will have differing loads. The direct effect of the test parameters on the load of the different stages can be estimated by looking at the active cycles of each stage, although this ignores aspects of the basic rendering algorithm that require waiting, such as accessing memory or reading from the framebuffer. The analysis of the different load balancing and parallelization efficiency will lead to some key insights for improvements described in Chapter 6.

In addition to the performance metrics above, certain absolute measures, such as triangles per second and pixels per second are of interest. These metrics will assume a nominal Raw processor frequency of 300MHz, though the equivalent metrics of cycles-per-triangle and cycles-per-pixel can be used as well. These metrics are commonly used to classify modern graphics processors, as they describe the basic limitations of the processor: pixel rate limits the maximum frequency a certain resolution can be drawn at (which is worse if there is *overdraw*, or multiple overlapping primitives drawn over each other), and the triangle rate limits the maximum frequency for a scene of certain geometric complexity. Analyzing how to improve on these two important metrics will also be an important aspect of Chapter 6.

5.3.1 Performance Model

The parallelized version of the pipeline should achieve speedup over the single-tile implementation in two ways: pipelined execution, and parallel pipelines. Whereas the execution

[4]Originally, the tests were also to be run with and without texture caching enabled, to analyze the effects of having cached textures on the performance of the pipeline. However, there was an error in the code that simulated disabling the cache, which produced incorrect results that were not discovered until it was too late to re-run the simulations for inclusion in this thesis.

time of the single tile processor will equal the total sum of all the execution times for each individual stage's work done, the execution time of a pipelined implementation will equal the maximum of all the execution times for each individual stage's work done. However, the individual execution times may be significantly longer due to parallelization overhead. Also, the maximum speedup is only achieved if all the stages in the pipeline are balanced — otherwise, stages which have less work to do will be sitting idle, wasting potential cycles, while the other stages finish what they are doing.

The measure of active versus total cycles is an approximation of the cycles wasted by each tile; however, it is not completely accurate, as even the single tile case will not achieve 100% utilization — certain occurrences, such as cache misses, data hazards, and waiting for framebuffer reads can result in a blocked processor. Therefore, it is difficult to determine exactly how many wasted cycles are added by parallelization overhead and balancing problems, and how many are inherent to the algorithm's structure. Also, as mentioned above, the parallelized tiles may have more work to do per tile, which does not come across in a pure utilization measurement.

Ideally, the speedup for multiple pipelines in parallel will be the speedup of one pipeline times the number of pipelines. However, there is some added overhead to the work that must be done by each stage due to parallelization: this includes overhead to distribute the primitives among tiles at the top, plus overhead to run the mutual exclusion algorithm at the bottom, along with extra wait times due to flushing, invalidating, and handling cache misses when accessing shared memory between tiles. For example, a single pipeline implementation could cache all renderstate and z-buffer accesses, leading to significantly less work to be done by the first and last stages (the single-tile implementation would also have such an advantage). Also, if the final compositing stage cannot keep up with the rate of fragments to render, the pipelines will end up stalling waiting for its last stage processor to gain access to the framebuffer and depth buffers to be able to process its buffered fragments. Note that effectively only one final stage tile will be active at any time, and therefore the maximum throughput of the final compositing stage is that of one tile in the compositing stage (this, of course, is complicated by having finite buffering — though if the buffers fill this generally means that the throughput of the final stage is not adequate for the amount of data coming out of Stage 3). The stages can of course do some of their work while it is not their turn, such as reading in and buffering the fragments to be drawn — but generally, the utilization of the final stage is expected to remain around 25%. For reasonable parallelization performance, each tile in the final stage must be capable of handling the total fragment bandwidth of all the previous pipeline stages. Having in-order primitives can make the performance much worse, as only one primitive is being rendered at a time, causing the final stage to be slowed down to the rate of only one pipeline at a time, while it usually is fast enough to handle multiple pipelines simultaneously.

The relationship between pipeline overhead, parallelization overhead, pipeline load balancing, the throughput of the final stage, buffering and other factors, and their effects on total work to be performed, utilization, and speedup is complicated. Attempts were made to generate an analytical model to relate all these parameters, but they came out confusing, with many variables that could not be verified empirically and many relationships left unsolved (such as the relationship between a pipeline having to wait on the last stage to output its fragments, the utilization of processors in that stage, and the amount of buffering between the stages). Therefore, no analytical model for performance is presented, and rather the empirical results from the next section are compared against the intuitive understanding of what impacts performance that has been developed in this section.

5.3.2 Empirical Performance Results

Most of the measurements of performance in this section look at how the variables change across a set of different typical render modes: starting from a baseline of flat colored only, this is modified by individually adding interpolated color, nearest neighbor texture, bilinear texture, flat colored lit, interpolated lit, transparent, and unordered transparent. Also, for comparison, a mode with interpolated color, texture, and lighting turned on is included, run as opaque, transparent, and unordered transparent. Finally, flat colored with no z access (but still unordered) is measured for the best case scenario. This set of data allows a feel for the individual contributions of different features, how the contributions combine together, and the full performance range from worst to best case. Each chart is displayed versus scaled primitive size on the horizontal axis, and the variable of interest on the vertical axis.

While preparing preliminary versions of these graphs, an anomaly was noticed when the triangle scaling was very small, especially at 0.01, both Stage 1 and Stage 4 would be at around 25% utilization. This was unexpected, as Stage 1 should have had more utilization (since the triangles were so small, the computation was geometry-limited and stage 1 did not have to wait for subsequent stages to finish rasterization), and Stage 4 should have had less (as there were only 4 pixels drawn per test, and it should have been waiting idly for pixels to arrive the rest of the time). However, the data indicated that only one tile of stage 1 was active at a time, and stage 4 was running at almost full speed! Two problems were discovered that lead to these anomalous results: an earlier bug fix in Stage 1 caused it to incorrectly wait until *after* geometry transform calculations before it sent the token into the network allowing the switches to start routing data to the next tile. Therefore, each tile would only start up after the previous had finished completely. In Stage 4, most of the cycles were used up passing the token around while waiting for input from the static network, and these were counted as active cycles by the profiling code. Stage 1 was fixed by rearranging the algorithm to send the token as soon as possible (and the efficiency regarding how render data was flushed from the cache was improved), and Stage 2 was fixed by labeling the automatic token-passing code in the interrupt handler as a busy-wait section. There was not enough time to re-run all the performance tests (the full suite takes over a week to simulate, even when spread across a dozen or so machines), but the scalings of 0.01 and 0.05, which are believed to be the most affected, were re-run and inserted into the results from before. Higher scalings cause the processor to be rasterization-limited, so the first stage is spending most of its time waiting for the latter stages anyway (and therefore parallelization of the first stage is not as big an issue), and the last stage spends most of its time drawing pixels and less processing the token-passing interrupts.

Figure 5-4 compares the speed-up over a single-tile implementation, the average processor utilization in the parallelized architecture, the expected speedup based on that utilization, and the estimated overhead based on the difference between that expected speedup and the actual speedup. The maximum speedup this architecture can achieve over the single-tile implementation, it appears, is a little less than 5. And that is for flat-shaded untextured unlit primitives that do not access the z-buffer at all. The speedups for various combinations of rendering modes fall within a 3 to 4.5 times improvement, with the speedup peaking somewhere at moderate triangle sizes. Transparent primitives cause the speedup to drop to less than 3, even when allowed to render out-of-order, and to fall to less than 2 when strict ordering is maintained. Very small primitives also cause the speedup to drop to less than 2. For a 16-tile processor, speedups in this range mean that there is either a lot of overhead, or that a lot of potential processing power is being wasted.

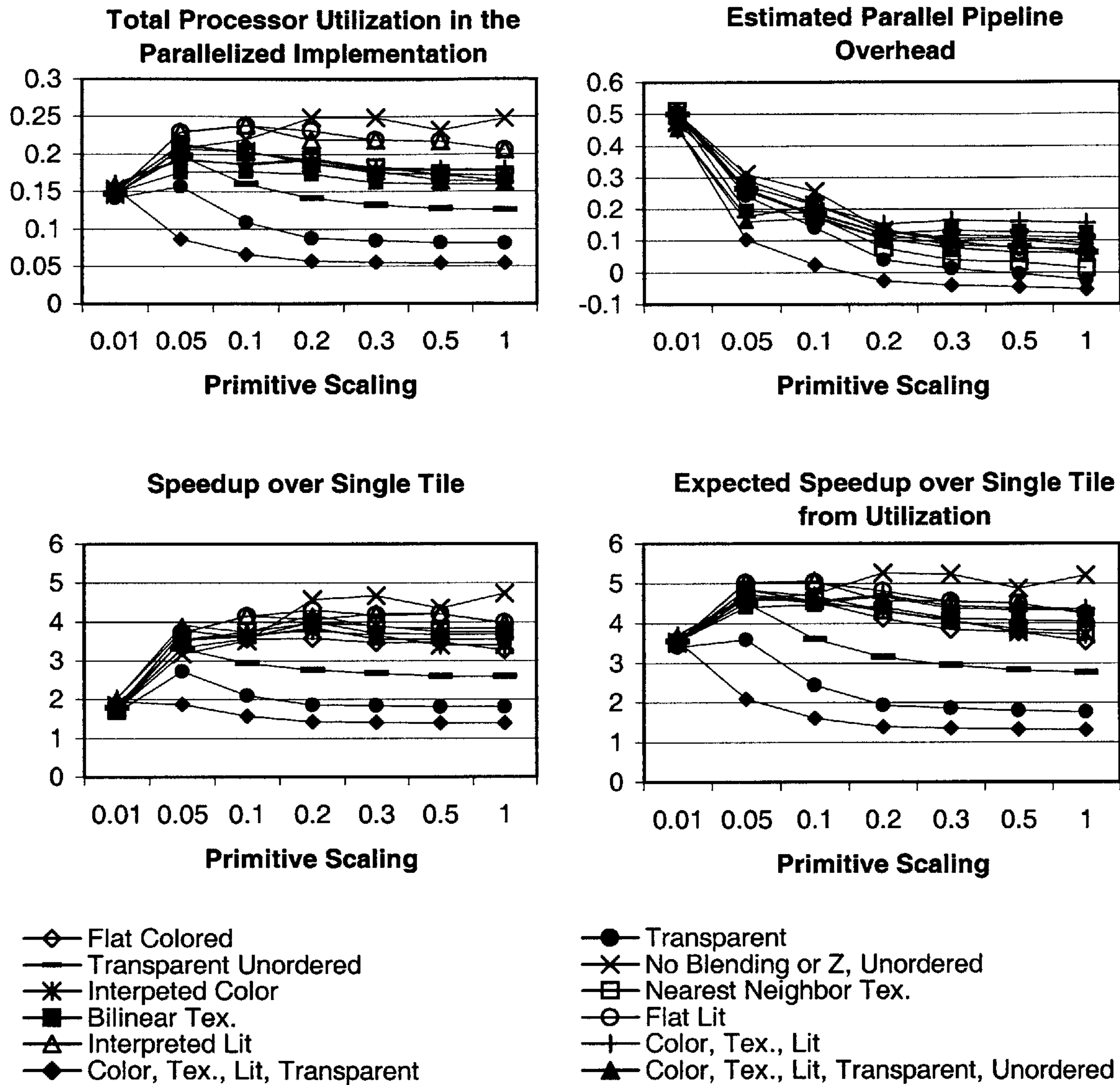


Figure 5-4: Performance: Speedup, Utilization, and Estimated Overhead

To determine where the problem lies, the speed-up is compared to the total processor utilization of the Raw chip. The processor utilization is used to calculate an "expected speedup," which assumes that there is no parallelization overhead. This expected speedup is equal to processor utilization on the Raw chip divided by the utilization on a single tile (not shown, though it ranged from about 60% to 80%), times the number of tiles (sixteen). When the actual speedup does not match the expected speedup, the discrepancy is assumed to be due to pipeline overhead (note that negative pipeline/parallelism overhead is possible — in these cases, the parallelized pipeline algorithm performs less overall work than the single tile algorithm)[5]. Based on this analysis, the overhead varies from around 50% with small primitives to less than 15%, down to a negligible amount of overhead for large primitives. This does not account for the poor efficiency of this parallel design, which varies from around 28% to around 9.4%. A large part of the inefficiency must be due to poor processor utilization — and indeed, the utilization on the raw never seems to go above 25% or so — and so the causes of this poor utilization must be explored.

Figure 5-5 breaks down the average processor utilization by stage, and Figure 5-6 plots the total active cycles for each stage. (The large discrepancy between primitive scalings of 0.05 and 0.1 in the total active cycles for stage 1 is caused by the fix mentioned earlier that was only re-run for these two data sets. It turns out that the fix also made Stage 1 have to do less work, and so its active cycles decreased.) Obviously the amount of utilization between the stages varies widely, indicating a problem with load balancing: Stage 1 only achieves decent utilization with very small primitives, and its utilization drops off rapidly as the primitives get larger; Stage 2's utilization is the best for primitives which require very little processing other than rasterization, but drops off progressively more with blended fragments, texture mapped fragments, transparent and unordered fragments in different combinations; Stage 3's utilization, on the other hand, is best when a lot of blending and texture mapping is taking place, and worst when transparency and out-of-order fragments are passing through — the completely unblended primitive lands right in the middle; Finally, Stage 4's utilization is best for the unblended fragments (probably because they can be rasterized quickly enough to saturate the 4th Stage's bandwidth), worst for the out of order primitives, and about the same for everything else. Another way to view the load balancing problem is to look at the total active cycles per stage — when a stage has a lot of work to do, it typically has a better utilization, at the expense of other stages who have to slow down their own computations to wait for the busiest stage.

In addition to load balancing concerns, however, is the fact that the processor utilization still does not go about 50% for any stage at any time, which implies that some other inefficiencies are involved — recall that the single tile implementation achieved around 60-80% utilization. Perhaps stages are still waiting on one another even when they are relatively unloaded due to inherent pipeline overhead, or unavoidable bottlenecking at the compositing stage. Also, perhaps some parallelization overhead is involved — this is certainly the case with Stage 4, whose 25% utilization cap stems from the fact that its synchronization algorithm only allows one tile to be doing most of the work at any time. Stage 1 may also stuffer from parallelization overhead, though not as severe. Finally, the abysmal performance of in-order primitives is due to the bottlenecking that occurs given the current over-conservative and inefficient sequential consistency algorithm for such in-order primitives.

Figure 5-7 displays the triangles per second and pixels per second achieved by the single-

[5]Overhead could also be estimated by looking at the total number of active cycles computed by the Raw processors versus the number computed in the single tile. The answers come out identical.

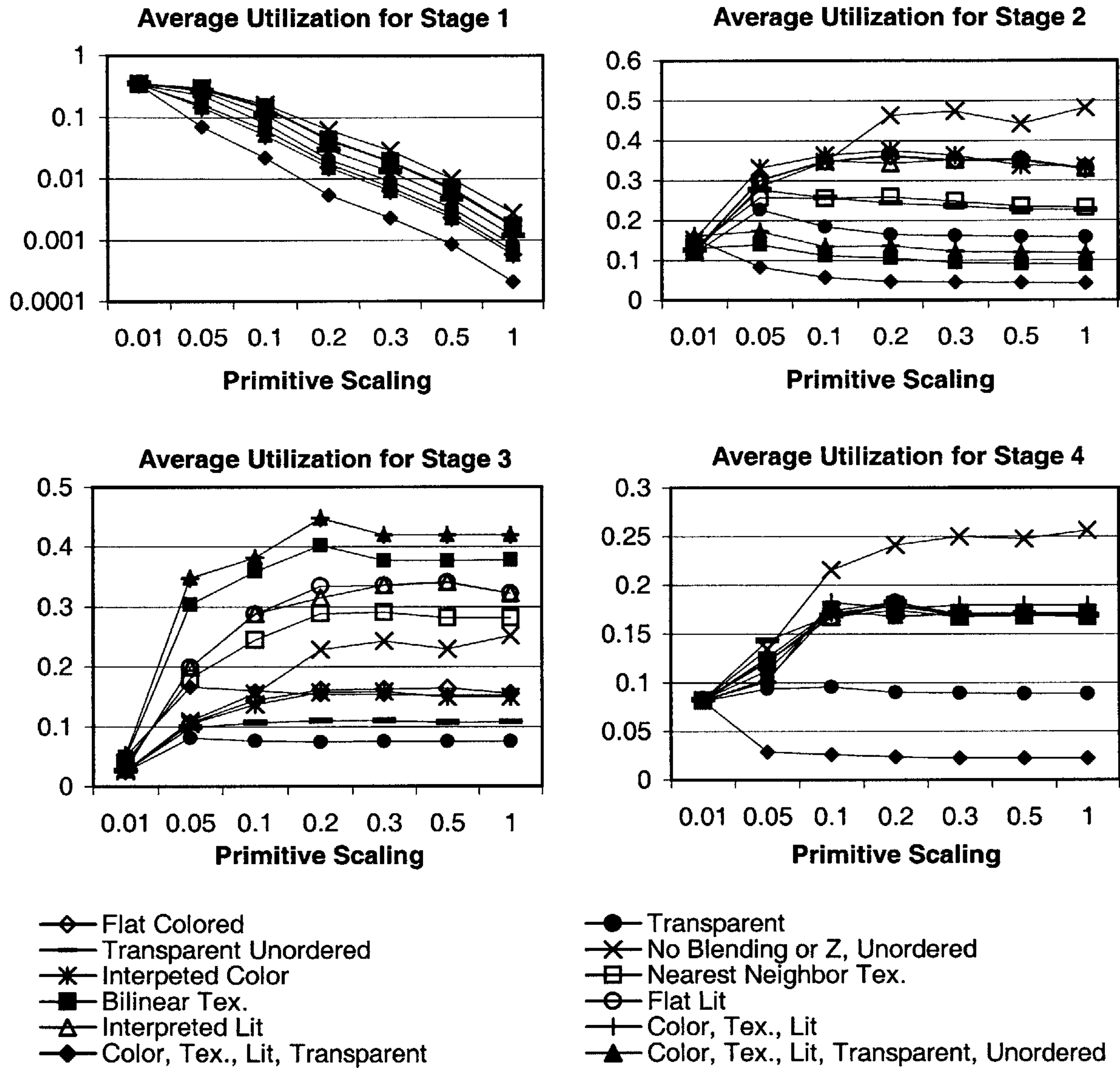


Figure 5-5: Performance: Utilization per Stage

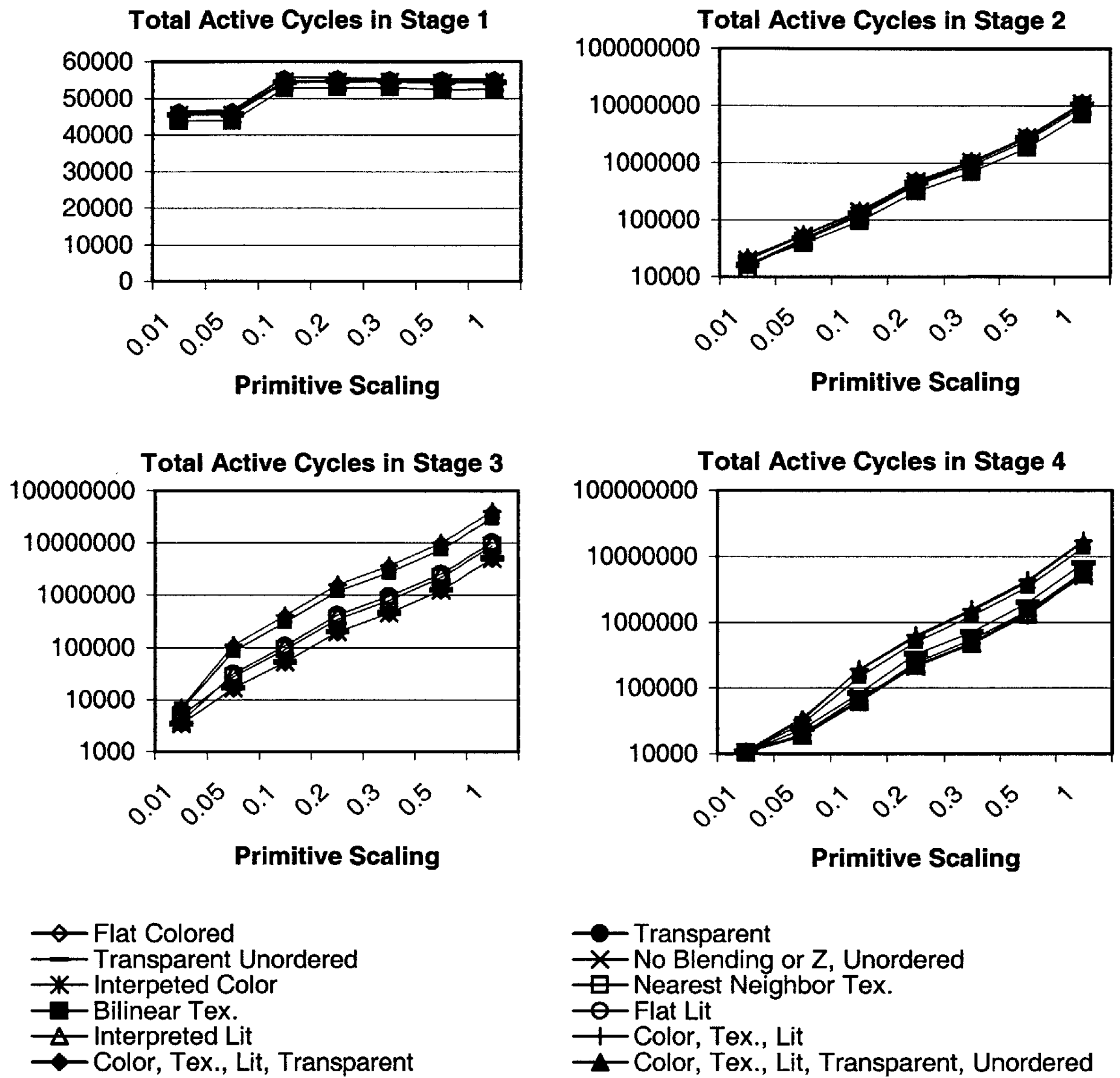


Figure 5-6: Performance: Active Cycles per Stage

tile and parallelized implementations on a 300MHz Raw chip. The peak pixel rate achieved looks to be just below 4 million pixels per second, compared with 800,000 pixels per second for a single tile, and the peak triangle rate is around 200,000 triangles per second, compared with around 100,000 for a single tile. For a screen resolution of 320x240, this pixel rate can achieve around 50 frames per second (less for more complicated primitives), which could not have any more than around 4,000 triangles so to sustain that frame rate. A 640x480 image would run (at peak) at around 13 frames per second with a maximum of 15,400 triangles, though this frame rate is not very ideal for interactive use, especially since most scenes use more complicated operations such as texturing that would slow it down even further.

For comparison, commercial graphics processors of a few years ago could readily push 300 million pixels per second, and transform 6 million vertices per second; And modern graphics processors claim numbers of up to 6 billion pixels per second and 300 million vertices per second! Although these numbers represent peak rates, rather than achieved rates, and are no doubt inflated by marketing, they still demonstrate that even at its best, this architecture has a huge amount of room for improvement compared to the current state of the art in consumer graphics accelerators, or even the state of the art several years ago. While the architecture described in this paper is only comfortable at 320 by 240 resolution, modern graphics accelerators perform quite well at resolutions closer to 1600 by 1200, handle multiple textures per primitive, and perform techniques such as anti-aliasing and anisotropic filtering that require huge amounts of pixel throughput. Also, in terms of triangle and geometry throughput, though modern CPUs can only source around 2 million triangles per second to the graphics card, modern graphics cards can procedurally generate many more primitives than that through techniques such as "vertex shaders," and have been able to achieve triangle rates into the hundreds of millions.

All is not lost, however, as there are many ways the architecture described in this paper can be improved immensely, and they are the subject of the next Chapter. In particular, the Raw architecture can be scaled by adding more processor tiles to the chip, and the ability for this 3D architecture to scale along with it is analyzed in Section 6.3.

5.4 Summary

This section presented the framework used for verification and performance testing of the architecture described in this paper, and then presented the verification and performance results garnered from that framework. The speedup from parallelization was not ideal, due partially to overhead from parallelization and communication that was not present in the single-tile version, but mostly due to an imbalance in the workloads performed between the different stages of the pipeline. However, in this kind of architecture, such imbalance is hard to avoid because of the differing requirements of different images to render.

The absolute performance of the architecture implemented on a 300MHz Raw chip is far below that of most commercial 3D Graphics implementations (though it is adequate to render images in 320x240 resolution in realtime). This can be somewhat improved by scaling the architecture horizontally — that is, adding more parallel pipelines — but eventually the speed will be limited by the bandwidth of the last stage, where only one tile can be rendering at a time. Suggestions on modifying this architecture in various ways to improve its performance and efficiency are the topic of the next chapter.

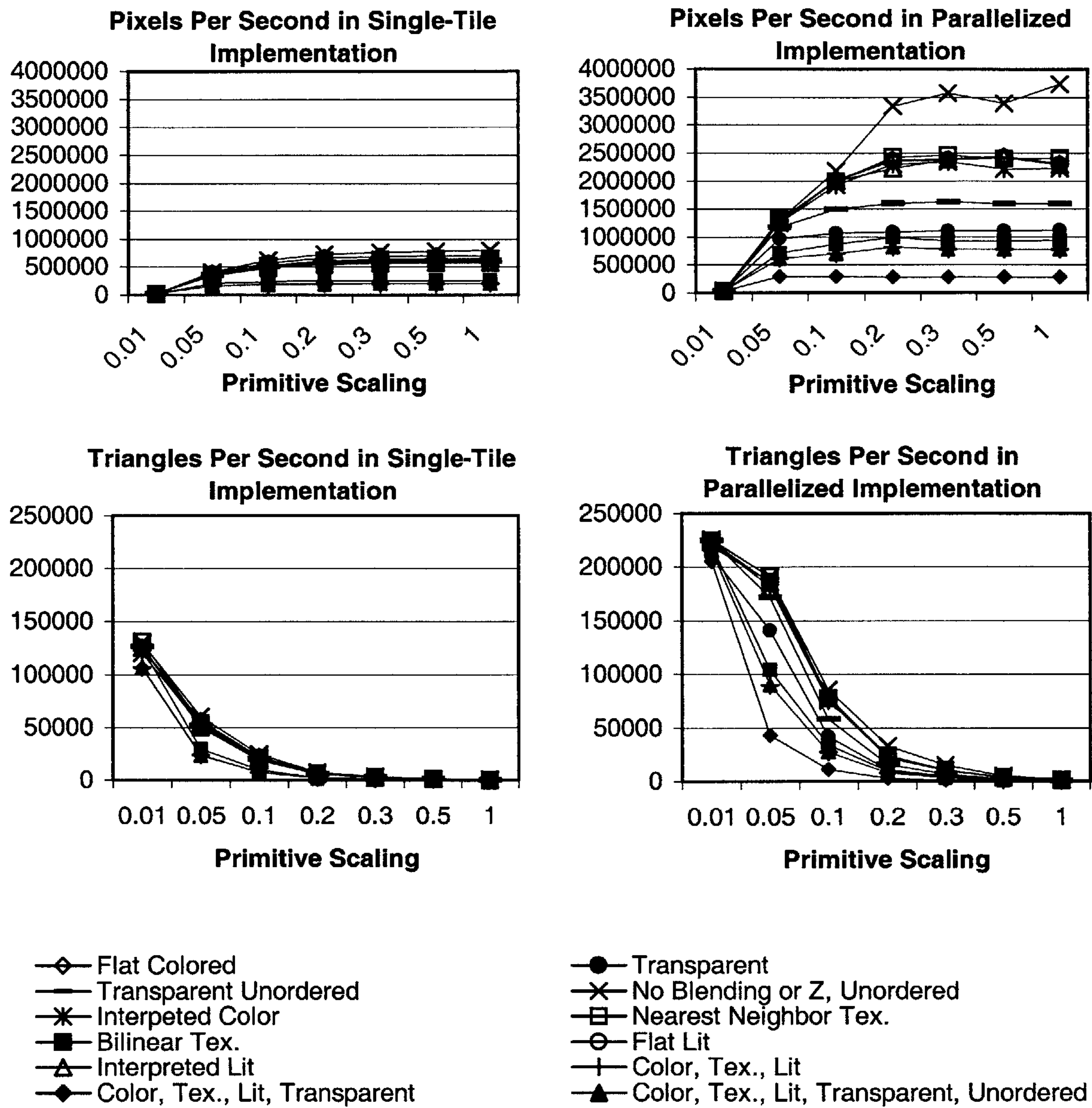


Figure 5-7: Performance: Triangles and Pixels per Second

Chapter 6

Improvements and Suggestions for Future Work

This chapter will describe some ideas for improvements over the current rendering architecture, and suggestions for future work in similar projects. From the results of the previous chapter, it is apparent that this rendering architecture has a way to go before realizing the full benefit of parallelizing the rendering algorithms, and before coming within the same region of performance as commercial offerings of even several years ago. However, the results from the last chapter allow us to analyze the bottlenecks and imbalances in the pipeline and make some suggestions based on what was learned from this project. Next, the issue of scaling the architecture to larger Raw arrays is discussed, including why the current architecture does not lend itself very well to scalability and what can be done about it. Finally, other ways in which this project could have been extended or improved, but was not due to time constraints, are discussed.

6.1 Improving Parallelization Efficiency

There are basically four ways in which the parallelization of this architecture is made less efficient: overhead due to pipelining, overhead due to parallel pipelines, load imbalance in the pipelines, and the serialization bottleneck at the end of the pipelines.

6.1.1 Reducing Pipelining Overhead

Pipelining overhead is defined here as operations that the processor would not have to perform if all the code was in one tile, but that it does because of the pipelining. The major culprit is communication overhead between the tiles. There is no pipeline interlocking or any further synchronization required, so these are not a problem. Note that pipelining overhead does not include extra work required due to the existence of multiple parallel pipelines — that is discussed in the next section. It also does not include processors stalling for extra cycles due to imbalance in the pipeline — that is discussed in Section 6.1.3.

As mentioned, the major overhead to pipelining the graphics algorithms is the need to package up, send and receive all the data from one stage to the next, whereas in a single processor implementation one could just leave all relevant variables in memory. In a way, this is unavoidable (and attempting to use shared memory would be even more inefficient), but techniques could possibly be devised to reduce the amount of data that

needs to be transferred, and the overheads for sending and receiving that data. The full set of fragment parameter information does not have to be sent for a fragment that does not have all the modes turned on (for example, it makes no sense to send "intensity" down the pipe if it is a non-lit primitive). The same argument holds for primitive info, which can be likewise reduced to the bare minimum necessary. The trick is to be able to do this without introducing too much additional overhead in decision making logic to reduce and expand the information being passed.

One may try more advanced schemes such as compression, and even temporal compression (where the data for the current primitive or fragment is related to previous ones), though the cost of such techniques would be an issue. However, there is one case where such a temporal scheme might make sense: untextured fragments, textured fragments, and pixels all are rasterized row by row, but their full x and y coordinates are sent with every fragment. With hundreds of thousands of fragments on the screen, this can be a significant amount of additional transfer. A simple improvement would assume that each subsequent fragment has the coordinates $(x+1, y)$ relative to previous fragments in the same primitive, unless a special marker came down the pipe setting up a new x or y coordinate (for example, starting a new row or skipping over 100% opaque fragments). This optimization could be combined with the "block" writing mode for the framebuffer for even more optimal pixel performance.

Also, flow control overhead could be reduced, such as in Stage 3, where it must send flow control words to the static network (See Section 4.2.3.4). Using the GDN in such cases would avoid such flow control overhead but add the overhead of GDN headers and the message length, so may or may not afford an improvement, depending on the circumstances.

6.1.2 Reducing Parallelization Overhead

Parallelization overhead is defined as the additional work required by each of the tiles in a pipeline when there are multiple parallel pipelines among which the rendering data is split. This additional work may include non-active cycles due to cache misses and the like, but in general extra waiting due to load imbalancing or bottlenecks are not included in this overhead — they are discussed in the next two sections.

Most of the overhead from parallelization occurs in two areas: communication and synchronization. In the architecture described in this paper, synchronization overhead only occurs in the first and final stages. The first stage's synchronization overhead is minimal: a handful of extra cycles are required by the static network to determine to which processor to send new primitives, and a few extra cycles are required by the tile code to begin and end every primitive transaction. The final stage is a bit tricker to analyze, because only one tile is truly active at any time, leading to some tricky performance analysis, but it also usually has low synchronization overhead, as handling the token only requires a handful of cycles per pass. However, when ordered primitives are rendered, the sequential synchronization of the last stage makes it so that only one pipeline is rendering for a long chunk of time, making it impossible to overlap rendering and take full advantage of the speed of the last stage.

One possible solution to the ordered primitive problem is to somehow split ordered primitives across several rasterizing pipelines, and let the pieces all render simultaneously just like unordered primitives do. Another solution is to implement less conservative sequential locking, and only lock out an ordered primitive if another primitive that overlaps it or may overlap it is present in the final stage. The locking could even be done on a

fragment-by-fragment basis; however, such dynamic fine grained locking becomes difficult to do efficiently on a coarse-grained machine like the Raw, though this solution can be used to great effect on fully custom hardware.

Communication overhead occurs whenever parallel tiles need to communicate some state information with each other. This occurs primarily in this architecture through use of shared memory structures. The problem with shared memory structures is that flushes, invalidates, and cache misses are an inherent part in keeping data consistent. It would be much more efficient if instead of having to flush data all the way back to main memory, then have the tile right next to you load that data again from main memory, one could send the data updates directly between tiles, via GDN or static messages (though static messages would be difficult given that the static network is already being used to route the primitive and fragment data).

The two major uses of shared memory communication in the pipeline are the render state in the first stage, and the z-buffer in the last stage. Both use round-robin type synchronization (though the first stage is more complex, and done transparently through the static network), so perhaps sending render state updates and z-buffer changes round-robin would also work. Such a technique would have to be careful about synchronization and deadlocking, however, and would probably require a lot of work to get working efficiently. Still, the benefits could be pretty substantial.

One improvement to z-buffer handling that was discovered too late to add to the implementation described in this paper is that the tile currently invalidates before and flushes the z-buffer after every single pixel write, even if the one tile is doing several writes in a row. Invalidating the z-buffer before all the buffered writes, and flushing afterwards may result in more efficient cache usage. Also, the current methods for cache flushing and invalidating are not optimal — see Section 6.2 for more information.

An inefficiency that is related to communication overhead and shared memory is the fact that each tile in the texture-mapping stage has its own cache, so each would have to load the texture from main memory individually, even if the same texture was used for all primitives. Though a solution to this problem is not known, perhaps some sort of direct inter-tile communication could again alleviate it a bit.

6.1.3 Improving Load Balancing

The graphics pipeline gets the best performance improvement over an unpipelined implementation if none of the stages need to wait idly on previous (or subsequent) stages to finish up. The best way to achieve this is to give all stages approximately the same amount of work to do. This is difficult with a 3D rendering processor, however, because different data sets will stress the stages differently — for example, lots of small primitives will spend more time doing geometry transformations, while a few very large primitives will spend much more time during rasterization, and untextured primitives will spend considerably less time than textured primitives doing texture lookup and combination.

For the most part, this seems like an unavoidable problem. However, it is not inconceivable to somehow develop a dynamically load-balanced system, where more resources are given to the parts of the pipeline that need it the most. The pipeline could be made incredibly small-grained and modular, and could reconfigure itself for whichever kind of primitives are being rendered. Untextured primitives would leave the texture mapping stage right out, lots of small primitives would provide more resources for geometry transformations, few very large primitives may cause the geometry transformation to occur on the same tile

as the rasterization. The amount of communication and synchronization required to do such realtime reconfiguration seems mind-boggling, though perhaps it could be implemented on a lesser scale — such as having a handful of pre-defined pipeline configurations to choose from based on the kinds of primitives that are being rendered (e.g., a for a textured primitive, Stage 2 may not only do rasterization but also perform texture coordinate mapping and maybe even load the texture color from memory, while for an untextured primitive Stage 2 would only perform half of the interpolation steps — with these kinds of variations hard-coded into Stage 2 itself based on the kind of primitive). Such a technique would still face some communication and caching difficulties, but could produce much better load balancing over the entire processor.

A dynamic, modular pipeline structure could also scale much more easily to larger Raw fabrics; See Section 6.3.

6.1.4 Improving Parallelization Bottlenecks

A parallelization bottleneck is a slowdown when too many parallel events have to be serialized to some extent, and therefore can be data-dependent (as opposed to overhead, which happens generally due to the structure of the pipeline). Bottlenecks can also limit the performance of the system when scaled; See Section 6.3.

The major bottleneck in this architecture is the fact that pixel writes have to be serialized. This is generally not a problem, as the pixel writing stage can usually process more fragments than the previous stages (rasterization and texture/blending) can generate. However, the performance of the previous stages can be improved by adding more parallel pipelines, while the performance of the last stage remains the same bottleneck. As noted in the previous chapter, the pixel rate for the current implementation of this 3D architecture on Raw is pretty slow, and the maximum pixel rate that the final stage can perform at is not much better, and so improving the performance of this last stage is critical in increasing the overall performance of the architecture.

The bottleneck of the last stage can be improved in two ways: by keeping the serialization point and improving its maximum throughput, or by attempting to remove the serialization point and letting the architecture scale to improve performance.

One interesting method to improving the maximum throughput of the final stage is, instead of using four processors in parallel to render the pixels, use four processors in another pipeline configuration. Each processor could perform one step of the pixel rendering (z buffer read, z buffer write, frame buffer read + blending, frame buffer write), to hopefully increase the throughput of pixel writing and reduce the number of cycles per pixel. Stage 3 would have to somehow serialize its outputs to arrive at the first tile in this sub-pipeline, though some method using the GDN or even interestingly programmed static network messages may be workable. The stages of the pipeline would also have to worry about interlocking when there are z-buffer or frame-buffer dependencies among the multiple fragments that would be in-flight, which may require some tricky communication on the Raw architecture. However, such a setup may gain the benefits of fine-grained synchronization between primitives that were described in Section 6.1.2. Additionally, if this pipeline was implemented in a modular way, as described in the previous section, it could be scalable when there are more tiles available in larger Raw processors.

However, even with the fastest, most efficient pixel pushing pipeline in the world, the processor could never go above one pixel per cycle simply due to the communication speed of the Raw networks. But Raw has several networks in parallel, which add up to even more

pixel bandwidth! If somehow primitives could be sent to the framebuffer in parallel, instead of in a serialized fashion, large performance gains could be realized as the architecture scales up. The framebuffer system may have to be modified to increase its bandwidth, as well, and many modern graphics cards have huge memory bandwidth available just for this purpose. Modern consumer-level graphics processors and memory systems can sometimes output up to eight pixels per cycle, and even more! However, one must devise a method to ensure sequential consistency among the parallel primitives without compromising the parallel performance benefits, and also one must up the bandwidth of the depth buffer to accommodate the scaling of the architecture, as well.

It might also be possible to combine these techniques, such as having a couple short pipelines in parallel. Such a compromise solution may end up giving the best overall results, but it is uncertain at this point.

Finally, accesses to shared memory such as the z-buffer and texture memory are subject to parallelization bottlenecks as well. Perhaps spreading the memories across several chips, and taking fuller advantage of the Raw processor's massive I/O bandwidth would lead to some substantial gains.

6.2 Improving Raw Performance

All the previous sections concentrate on reorganizing the way the rendering algorithms are parallelized and pipelined, in order to squeeze more utilization and performance out of them. However, that is not to say the original rendering algorithms are optimal! The performance of the 3D processor can probably be vastly improved by spending time to optimize the critical parts of the 3D rendering algorithms themselves.

Firstly, the inner loops for each stage, especially the rasterizing, texture blending, and pixel pushing loops, could be hand-coded in assembly to squeeze every possible cycle of performance out of them. The current C code is primarily aimed at correctness, not optimal performance, and there is probably lots of room for improvement (although some attempts were made to make Stage 4's pixel pushing loops a bit more efficient). There are also some sections of the code that are quite un-optimal in their current form — especially flushes and invalidates for shared memory blocks. These will loop over every word in the data structure and attempt to flush it or invalidate it from the cache, without regard to block size or (for very large structures) whether the entire structure can even fit in the cache. The result is that much of the time these cache management loops spend more time than the need to spinning on the data. A smarter algorithm could save a lot of cycles, especially with things like render state and texture memory.

In addition to more efficiently coding the existing algorithms, those algorithms could be modified to increase their performance. Math could be migrated from floating point to fixed point in many cases. The triangle rasterization algorithm could be improved to waste less cycles between rows. There may be more efficient ways to calculate bilinear filtering, etc.

More significant changes and additions could also be made to the algorithms used. Fast z clear, hierarchical z buffers, or even z compression could be used to reduce the delay in checking the z buffer for every pixel. The changes could even occur with external hardware support — for example, the framebuffer controller could be modified to be able to do blends into the framebuffer for you, significantly reducing the latency when trying to render a transparent primitive.

Another example of an algorithmic improvement would be to accept mesh structures instead of only individual primitives. A mesh structure is a set of primitives who share certain vertices, and therefore less vertex calculations would have to be performed (along with less I/O bandwidth needed to send all the vertices).

Basically, having been written for correctness and under a strong time constraint, the code for the underlying algorithms has a good amount of room for improvement in terms of its basic performance. A release version of such a 3D processor should obviously be optimized as fully as possible.

6.3 Scaling the Graphics Architecture

Several previous sections have mentioned how some improvements can affect the scalability of the processor (intuitively defined as the ability for the parallel application to take advantage of increased parallel resources). This is an important concern on Raw, as a major premise of the Raw architecture is that, as technology improves, more and more tiles can fit on a single chip, resulting in improved performance. So the question is, can this rendering pipeline be easily modified to attain improved performance on a larger Raw chip with more processing tiles?

As for being easily scaled, the architecture can readily be extended horizontally, adding more rendering pipelines in parallel with each other. Some code will have to be rewritten, and constants changed, but the basic structure of the code will remain the same (it may even be possible to automate the generation of scaled code). However, scaling in the vertical direction, adding more pipeline stages per pipeline, would require radical rewrites of all the tile code to accommodate the new pipeline structure. As mentioned previously in Section 6.1.3, having a modular, more finely-grained pipeline structure may make scalability much more attainable. A dynamically rebalanced pipeline may also be able to take fuller advantage of more pipeline resources.

However, there is still an upper limit to the scalability of any pipeline, where the computation is divided so finely that the overhead of pipelining dominates each tile's activity. To avoid this, some of the extra "vertical" processors could also be used for parallelizing the pipelines (that is, for "horizontal" scaling). This would require a more complicated placement of the tiles and of the routing between them, but may result in improved utilization of the extra processor resources.

Parallel scaling may still run up against performance bottlenecks, however, especially with the final stage serializing accesses to the framebuffer, and with simultaneous shared memory accesses. As suggested in Section 6.1.4, the I/O itself can be scaled so that several accesses to the framebuffer and the RAM can be processed at any time. However, synchronization between such accesses (to resolve dependencies, for instance) will become more complicated as the number of simultaneous accesses increases. In the end, the scalability of this architecture can be improved greatly over the current implementation, but it will run up against some very challenging limits as the number of tiles available continues to increase.

6.4 Ways to Extend This Thesis

Besides performance improvements and general suggestions for future work, there are many other ways that the project developed in this thesis can be extended. In the vein of im-

proving the processor itself, for example, the available feature set could be made more rich: more blending modes, mip-mapping and multi-texturing, anti-aliasing, additional hardware lights, programmable shaders, line drawing routines and innumerable other features could be added to the processor. The processor could also be modified to support multiple resolutions and image modes, standard 2D PC graphics, and so on.

Another possibility would be to write an OpenGL interface for the graphics AI, to allow a much larger set of programs to be run and tested on the architecture. The processor could also be implemented in actual hardware, perhaps even with a low-level driver to interface the hardware through an API layer with a 3D application.

A final possibility — though there surely are many more — is to port the architecture to something like StreamIt [6], a streaming programming language for Raw, which may allow more flexible, dynamic pipeline designs. Using stream processing for polygon rendering has shown to produce good results [14].

6.5 Summary

This chapter took the performance results from the previous chapter, and the experience from implementing this 3D architecture in general, and provided some avenues for improvement over the current implementation, and suggestions for future work in the area. The suggestions mentioned various ideas on reducing pipelining and parallelization overhead, improving load balancing, reducing the impact of key bottlenecks, and improving the scalability of the architecture.

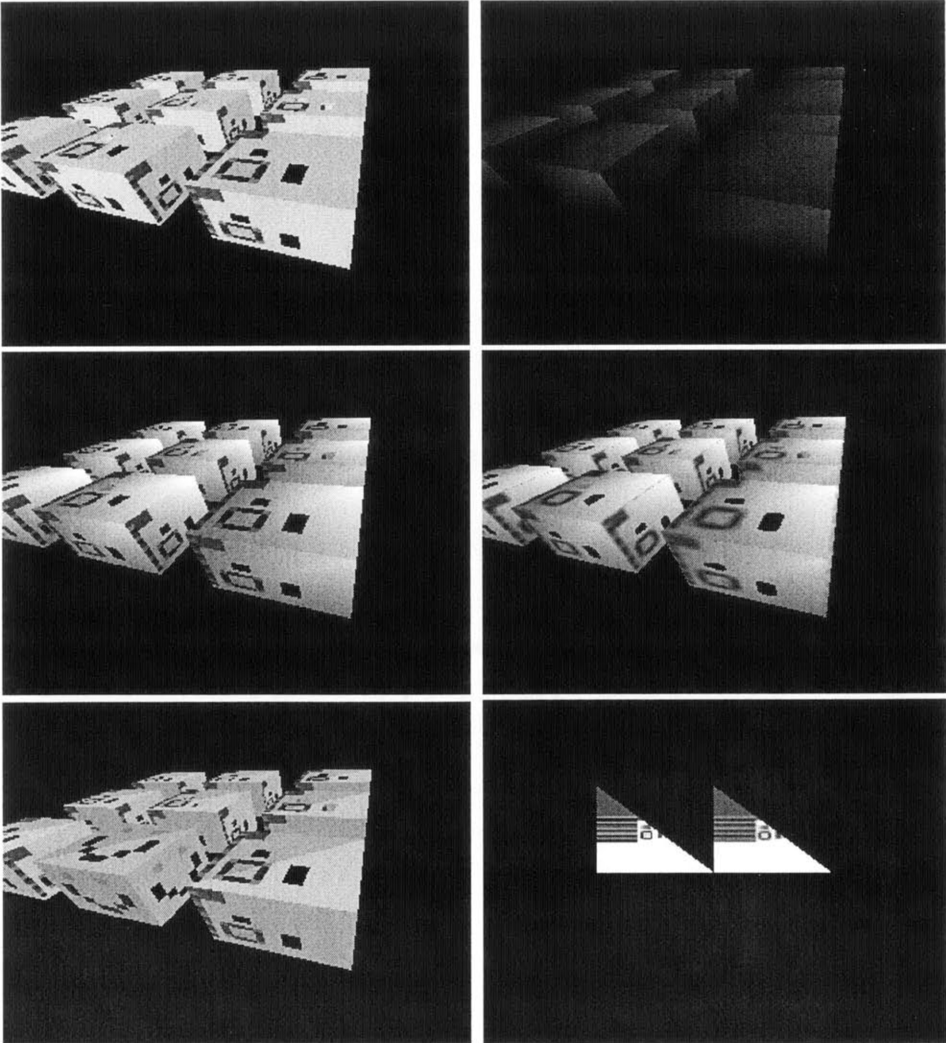
Chapter 7

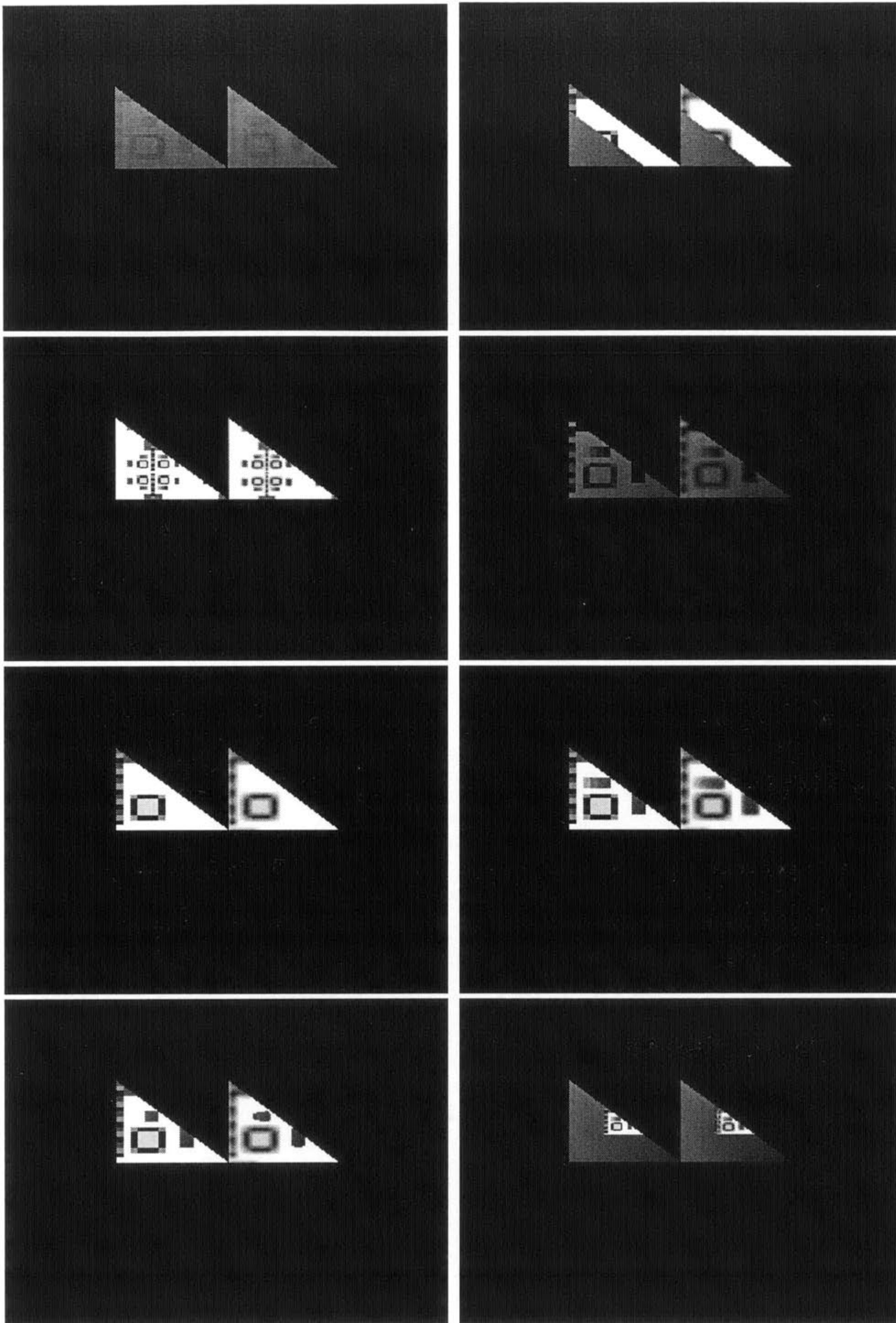
Conclusion

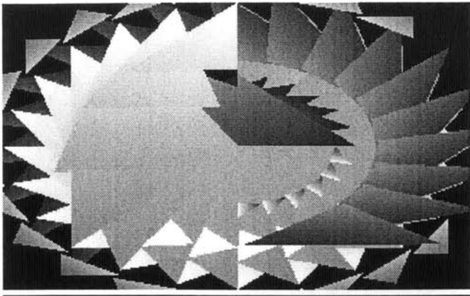
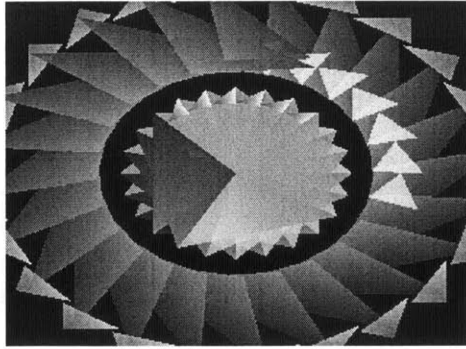
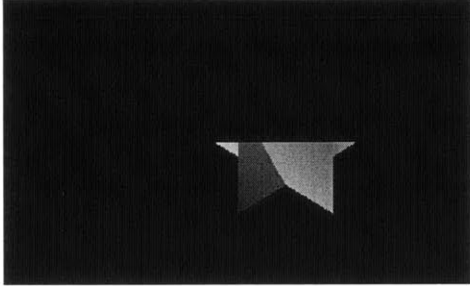
This paper presented the background, design, implementation and analysis of a complete 3D graphics processor built on the Raw reconfigurable architecture. Starting from the basics of the 3D rendering algorithms themselves, the paper worked through the development of pipelining and parallelizing those algorithms on the Raw architecture. A fully-working rendering architecture was developed, with several of the difficult problems involved detailed in this report, and implemented on a simulated Raw processor. Exhaustive performance measurements were taken, and it was found that the performance of this from-scratch 3D architecture was well below that of any modern 3D processor. The major culprits for this were discussed, and improvements and directions in which to head were suggested. In order for a 3D processor on Raw to be successful, it is critical that it make the most use of available processors as possible, and waste as few cycles as possible. Perhaps the solution would be a more efficient, less parallelized but deeper pipeline — or perhaps the other direction would be more fruitful. Whatever the case, the fact remains that this is a very challenging application, that will provide much fruitful and interesting work to anyone who decides to pursue it further.

Appendix A

Additional Rendered Images





Appendix B

Single-Tile Code Listing

Note: the single-tile code uses headers from the complete implementation in Appendix C, and uses the same testing framework that is in Appendix D.

B.1 SingleTile.c

```
// SingleTile.c
//
//  Ken Taylor, MIT Master's Thesis 2004
//
// Last Updated: 4/18/2004
//
// the whole renderer implemented as a single tile for speedup comparison

#include "module_test.h" // includes raw.h
#include "raw_compiler_defs.h" // for PASS (testing)                              10
#include "../implement/Stage1-datatypes.h" // shared datatypes
#include "../implement/ZBuf_datatypes.h" // for z-buffer/stage-4 interaction
#include "../implement/render_framebuffer.h"
#include "../implement/render_cmds.h" // command defines
#include "../implement/Common-sw.h"

// Start up the switch (code in assembly)
void setup_switch_main(void);
void setup_switch_scenestream(void);
                                                                                  20
RenderState *prs;
TexManager *ptm;
ZBufData *pzbd;

// matrix multiply! r = M*e
void MatrixMult(float *r0, float *r1, float *r2, float *r3,
                TransformMatrix M,
                float e0, float e1, float e2, float e3)
{
```

```
  *r0 = M[0][0]*e0 + M[0][1]*e1 + M[0][2]*e2 + M[0][3]*e3;                30
  *r1 = M[1][0]*e0 + M[1][1]*e1 + M[1][2]*e2 + M[1][3]*e3;
  *r2 = M[2][0]*e0 + M[2][1]*e1 + M[2][2]*e2 + M[2][3]*e3;
  *r3 = M[3][0]*e0 + M[3][1]*e1 + M[3][2]*e2 + M[3][3]*e3;
}

// matrix to matrix multiply! X = M*Y
void MatrixMatrixMult(TransformMatrix X, TransformMatrix M,
                      TransformMatrix Y)
{
  int i,j,k;                                                                40
  for(i = 0; i<4; i++)
    for(j = 0; j<4; j++)
      {
        X[i][j] = 0;
        for(k = 0; k < 4; k++)
          {
            X[i][j] += M[i][k] * Y[k][j];
          }
      }
}                                                                           50

// invert and transpose a matrix! X = (Y^-1)^T
// note: used only for normals, so:
// - only computing the adjoint, not the inverse, and,
// - only computing it for the upper left 3x3 (no translation on normals)
// code from http://www.gignews.com/realtime020100.htm, accessed 4/20/04
void MatrixInvTrans(TransformMatrix X, TransformMatrix Y)
{
  X[0][0] = Y[1][1] * Y[2][2] - Y[1][2] * Y[2][1];
  X[0][1] = Y[1][2] * Y[2][0] - Y[1][0] * Y[2][2];                          60
  X[0][2] = Y[1][0] * Y[2][1] - Y[1][1] * Y[2][0];
  X[1][0] = Y[2][1] * Y[0][2] - Y[2][2] * Y[0][1];
  X[1][1] = Y[2][2] * Y[0][0] - Y[2][0] * Y[0][2];
  X[1][2] = Y[2][0] * Y[0][1] - Y[2][1] * Y[0][0];
  X[2][0] = Y[0][1] * Y[1][2] - Y[0][2] * Y[1][1];
  X[2][1] = Y[0][2] * Y[1][0] - Y[0][0] * Y[1][2];
  X[2][2] = Y[0][0] * Y[1][1] - Y[0][1] * Y[1][0];
  X[0][3] = X[1][3] = X[2][3] = 0;
  X[3][3] = 1;
}                                                                           70


// clears the renderstate (prs) to its initial values
void ClearRenderState()
{
  int i,j;
```

```
prs->Updated = 0xE;
for(i = 0; i < 4; i++)  // init all matrices to the identity matrix
  for(j = 0; j < 4; j++)                                                    80
    {
      prs->ModelToWorld[i][j] = prs->WorldToView[i][j] =
        prs->ModelToView[i][j] = prs->NormalToWorld[i][j] =
        (i == j) ? 1.0f : 0.0f;
    }
// temp projection matrix for testing
// near = -3, far = -1
// left = -1, right = 1
// top = -1, bot = 1
//                                                                          90
// -3  0  0  0
//  0 -3  0  0
//  0  0 -2 -3
//  0  0  1  0
/*prs->WorldToView[0][0] = prs->ModelToView[0][0] = -3;
prs->WorldToView[1][1] = prs->ModelToView[1][1] = -3;
prs->WorldToView[2][2] = prs->ModelToView[2][2] = -2;
prs->WorldToView[3][3] = prs->ModelToView[3][3] = 0;
prs->WorldToView[2][3] = prs->ModelToView[2][3] = -3;
prs->WorldToView[3][2] = prs->ModelToView[3][2] = 1;*/                      100

prs->nx = 1.0f;
prs->ny = 0.0f;
prs->nz = 0.0f;
prs->rgba = 0x000000FF;
prs->pInfo.p.Mode.lit = 0;
prs->pInfo.p.Mode.useamb = 0;
prs->pInfo.p.Mode.usedir = 0;
prs->pInfo.p.Mode.texmode = 1;
prs->pInfo.p.Mode.texalpha = 1;                                             110
prs->pInfo.p.Mode.colalpha = 1;
prs->pInfo.p.Mode.colinterp = 1;
prs->pInfo.p.Mode.litinterp = 0;
prs->pInfo.p.Mode.texinterp = 0;
prs->pInfo.p.Mode.outoforder = 0;
prs->pInfo.p.Mode.textile = 1;
prs->pInfo.p.Mode.nousez = 0;
prs->pInfo.p.Mode.nowritez = 0;
prs->pInfo.p.SeqNum = 1;
prs->pInfo.TextureID = 0;                                                   120
prs->pInfo.ColTexBalance = 0.5f;
prs->pInfo.alphaThresh = 128;
prs->pInfo.ambColor = 0;
prs->pInfo.dirColor = 0;
prs->ldx = 1.0f;
```

```
  prs->ldy = 0.0f;
  prs->ldz = 0.0f;
  prs->ambreflect = 255;
  prs->dirreflect = 255;
  prs->dirdefined = 0;                                              130
  prs->LaggedSeqNum = 1;
}



// takes cmd, and executes it.
// returns true if going into scenestream, false if not.
// potentially uses prs, ptm, pzbd.
unsigned doCommand(unsigned cmd)
{                                                                   140
  int i,j;

  switch(cmd)
    {
    case RENDER_BEGINSCENE:
      static_send(0);
      static_send(0);
      return 1;
      break;
                                                                    150
    case RENDER_COLOR:
      static_send(1);
      prs->rgba = static_receive();
      static_send(0);
      prs->Updated = 0x0E;
      break;

    case RENDER_MODELMATRIX:
      static_send(16);
      for(i = 0; i < 4; i++)                                        160
        for(j = 0; j < 4; j++)
          {
            prs->ModelToWorld[i][j] = static_receive_f();
          }

      static_send(0);

      MatrixMatrixMult(prs->ModelToView, prs->WorldToView,
                       prs->ModelToWorld);
                                                                    170
      MatrixInvTrans(prs->NormalToWorld, prs->ModelToWorld);

      prs->Updated = 0x0E;
```

```
  break;

case RENDER_VIEWMATRIX:
  static_send(16);
  for(i = 0; i < 4; i++)
    for(j = 0; j < 4; j++)                                          180
      {
        prs->WorldToView[i][j] = static_receive_f();
      }

  static_send(0);

  MatrixMatrixMult(prs->ModelToView, prs->WorldToView,
                   prs->ModelToWorld);

  prs->Updated = 0x0E;                                              190

  break;

case RENDER_NORMAL:
  static_send(3);
  prs->nx = static_receive_f();
  prs->ny = static_receive_f();
  prs->nz = static_receive_f();

  static_send(0);                                                   200

  prs->Updated = 0x0E;
  break;

case RENDER_SET_LIT:
  static_send(1);
  prs->pInfo.p.Mode.lit = static_receive();
  static_send(0);
  prs->Updated = 0x0E;
  break;                                                            210

case RENDER_SET_USEAMB:
  static_send(1);
  prs->pInfo.p.Mode.useamb = static_receive();
  static_send(0);
  prs->Updated = 0x0E;
  break;

case RENDER_SET_USEDIR:
  static_send(1);                                                   220
  prs->pInfo.p.Mode.usedir = static_receive();
```

```
    static_send(0);
    prs->Updated = 0x0E;
    break;

  case RENDER_SET_TEXMODE:
    static_send(1);
    prs->pInfo.p.Mode.texmode = static_receive();
    static_send(0);
    prs->Updated = 0x0E;                                  230
    break;

  case RENDER_SET_TEXALPHA:
    static_send(1);
    prs->pInfo.p.Mode.texalpha = static_receive();
    static_send(0);
    prs->Updated = 0x0E;
    break;

  case RENDER_SET_COLALPHA:                               240
    static_send(1);
    prs->pInfo.p.Mode.colalpha = static_receive();
    static_send(0);
    prs->Updated = 0x0E;
    break;

  case RENDER_SET_COLINTERP:
    static_send(1);
    prs->pInfo.p.Mode.colinterp = static_receive();
    static_send(0);                                       250
    prs->Updated = 0x0E;
    break;

  case RENDER_SET_LITINTERP:
    static_send(1);
    prs->pInfo.p.Mode.litinterp = static_receive();
    static_send(0);
    prs->Updated = 0x0E;
    break;
                                                          260
  case RENDER_SET_TEXINTERP:
    static_send(1);
    prs->pInfo.p.Mode.texinterp = static_receive();
    static_send(0);
    prs->Updated = 0x0E;
    break;

  case RENDER_SET_OUTOFORDER:
    static_send(1);
```

```
  prs->pInfo.p.Mode.outoforder = static_receive();                    270
  static_send(0);
  prs->Updated = 0x0E;
  break;

case RENDER_SET_TEXTILE:
  static_send(1);
  prs->pInfo.p.Mode.textile = static_receive();
  static_send(0);
  prs->Updated = 0x0E;
  break;                                                              280

case RENDER_SET_NOUSEZ:
  static_send(1);
  prs->pInfo.p.Mode.nousez = static_receive();
  static_send(0);
  prs->Updated = 0x0E;
  break;

case RENDER_SET_NOWRITEZ:
  static_send(1);                                                     290
  prs->pInfo.p.Mode.nowritez = static_receive();
  static_send(0);
  prs->Updated = 0x0E;
  break;

case RENDER_SET_TEXTUREID:
  static_send(1);
  prs->pInfo.TextureID = static_receive();
  static_send(0);
  prs->Updated = 0x0E;                                                300
  break;

case RENDER_COLTEXBALANCE:
  static_send(1);
  prs->pInfo.ColTexBalance = static_receive_f();
  static_send(0);
  prs->Updated = 0x0E;
  break;

case RENDER_ALPHATHRESH:                                              310
  static_send(1);
  prs->pInfo.alphaThresh = static_receive();
  static_send(0);
  prs->Updated = 0x0E;
  break;

case RENDER_AMBCOLOR:
```

```
  static_send(1);
  prs->pInfo.ambColor = static_receive();
  static_send(0);                                                    320
  prs->Updated = 0x0E;
  break;

case RENDER_DIRCOLOR:
  static_send(1);
  prs->pInfo.dirColor = static_receive();
  static_send(0);
  prs->Updated = 0x0E;
  break;
                                                                     330
case RENDER_DIRLIGHT:
  static_send(3);
  prs->ldx = static_receive_f();
  prs->ldy = static_receive_f();
  prs->ldz = static_receive_f();
  static_send(0);
  prs->dirdefined = 1;
  prs->Updated = 0x0E;
  break;
                                                                     340
case RENDER_AMBREFLECT:
  static_send(1);
  prs->ambreflect = static_receive();
  static_send(0);
  prs->Updated = 0x0E;
  break;

case RENDER_DIRREFLECT:
  static_send(1);
  prs->dirreflect = static_receive();                                350
  static_send(0);
  prs->Updated = 0x0E;
  break;

case RENDER_CLEARFB:
  {
    int i;
    unsigned page, rgbx;

    static_send(2);                                                  360
    page = static_receive(); // page (like z->fbmode)
    rgbx = static_receive(); // rgbx
    static_send(0);

    for(i = 0; i < VWIDTH * VHEIGHT; i++)
```

```
      {
        fb_set_pixel_rawaddr(i, rgbx, page & FBMODE_BACK,
                             (page & FBMODE_FRONT)>>1);
      }
    }                                                                370
    break;

  case RENDER_CLEARZ:
    {
      int i;
      static_send(0);
      static_send(0);

      for(i = 0; i < VWIDTH*VHEIGHT; i++)
        {                                                            380
          pzbd->buf[i] = 0x7FFFFFFF;
        }

      // flush zbd
      flush_variable(pzbd, sizeof(ZBufData));
    }

    break;

  case RENDER_SETPAGE:                                               390
    static_send(1);
    // page .. FBMODE_NONE, BACK, FRONT, BOTH
    pzbd->fbmode = static_receive();
    flush_word(&pzbd->fbmode);
    static_send(0);
    break;

  case RENDER_FLIPPAGE:
    static_send(1);
    if(static_receive())// wait for vsync?                           400
      fb_flip_page_vsync();
    else
      fb_flip_page();
    static_send(0);
    break;

  case RENDER_ALLOCATE_TEXTURE:
    {
      unsigned sizex, sizey;
      unsigned totalsize;                                            410
      signed nexttexID, texID = -1;
      TexAllocation *pAlloc = 0;
```

```
// get next texture id to use
for(nexttexID = 0; ptm->pTexEntryTable[nexttexID].valid && nexttexID
      < ptm->MaxTextures; nexttexID++);

static_send(2);
sizex = static_receive();
sizey = static_receive();                                                    420

if(nexttexID < ptm->MaxTextures)
  {
    totalsize = sizex*sizey;

    if(ptm->pAllocHead == 0)
      {
        // base case, all memory is clear
        if(totalsize*sizeof(unsigned) <= ptm->TexMemorySize)
          {                                                                  430
            pAlloc = (TexAllocation*)malloc(sizeof(TexAllocation));
            if(pAlloc)
              {
                ptm->pAllocHead = ptm->pAllocTail = pAlloc;
                pAlloc->ID = texID = nexttexID;
                pAlloc->pBegin = ptm->pTexMemory;
                pAlloc->pEnd = ptm->pTexMemory + totalsize;
                pAlloc->pNext = 0;
                pAlloc->pPrev = 0;
                ptm->TexMemoryFree -= totalsize*sizeof(unsigned);            440
              }
          }
      }
    else
      {
        // search for a block large enough to hold the texture
        if(totalsize*4 <= ptm->TexMemoryFree)
          {
            TexAllocation * pta;
            unsigned * lastbegin = ptm->pTexMemory +                         450
              ptm->TexMemorySize/sizeof(unsigned);
            for(pta = ptm->pAllocTail; pta != 0;
                lastbegin = pta->pBegin, pta = pta->pPrev)
              {
                if(lastbegin - pta->pEnd >= totalsize)
                  {
                    // here's a spot that will work, allocate it right after
                    // pta->pEnd, and allocate new texallocation at pAlloc.
                    pAlloc = (TexAllocation *) malloc(sizeof(TexAllocation));
                    if(pAlloc)                                               460
                      {
```

```
                    if(pta->pNext)
                      pta->pNext->pPrev = pAlloc;
                    else
                      ptm->pAllocTail = pAlloc;

                    pAlloc->pNext = pta->pNext;
                    pAlloc->pPrev = pta;
                    pta->pNext = pAlloc;
                    pAlloc->ID = texID = nexttexID;                          470
                    pAlloc->pBegin = pta->pEnd;
                    pAlloc->pEnd = pAlloc->pBegin + totalsize;
                    ptm->TexMemoryFree -= totalsize*sizeof(unsigned);
                  }
              }
          }

          if(pta == 0)
            { // went all the way to the beginning, do final check
              // if there's room at beginning, allocate new texalloc at pAlloc.   480

              if(lastbegin - ptm->pTexMemory >= totalsize)
                {
                  pAlloc = (TexAllocation*)malloc(sizeof(TexAllocation));
                  if(pAlloc)
                    {
                      ptm->pAllocHead->pPrev = pAlloc;
                      pAlloc->pNext = ptm->pAllocHead;
                      pAlloc->pPrev = 0;
                      ptm->pAllocHead = pAlloc;                              490
                      pAlloc->ID = texID = nexttexID;
                      pAlloc->pBegin = ptm->pTexMemory;
                      pAlloc->pEnd = pAlloc->pBegin + totalsize;
                      ptm->TexMemoryFree -= totalsize*sizeof(unsigned);\
                                                                    }
                }
            } // if(pta == 0)
        } // if(totalsize*4 <= ptm->TexMemoryFree)
      } // else ( if(ptm->pAllocHead == 0))
  } // if(nexttexID < ptm->MaxTextures)                                      500

if(pAlloc && texID != -1)
  {
    // allocated the texture correctly, now add the texture entry
    ptm->pTexEntryTable[texID].valid = 1;
    ptm->pTexEntryTable[texID].updated = 0x0F;
    ptm->pTexEntryTable[texID].Width = sizex;
    ptm->pTexEntryTable[texID].Height = sizey;
    ptm->pTexEntryTable[texID].pAlloc = pAlloc;
```

```
        ptm->pTexEntryTable[texID].pBegin = pAlloc->pBegin;                510

        ptm->NumTextures++;

        // and flush texentry out
        flush_variable(&ptm->pTexEntryTable[texID], sizeof(TexEntry));

        // don't need to flush TexAllocation, as only the main
        // processor (this one) ever uses that data.
        // likewise, don't need to flush TexManager out.
                                                                           520
      }

    static_send(1);
    static_send(texID);

  }
  break;

case RENDER_DEALLOC_TEXTURE:
  {                                                                        530
    signed id;
    TexAllocation *pAlloc;
    static_send(1);
    id = static_receive();

    if(id >= 0 && id < ptm->MaxTextures)
      {
        pAlloc = ptm->pTexEntryTable[id].pAlloc;
        if(ptm->pTexEntryTable[id].valid && pAlloc)
          {                                                                540
            if(pAlloc->pNext)
              pAlloc->pNext->pPrev = pAlloc->pPrev;
            else
              ptm->pAllocTail = pAlloc->pPrev;

            if(pAlloc->pPrev)
              pAlloc->pPrev->pNext = pAlloc->pNext;
            else
              ptm->pAllocHead = pAlloc->pNext;
                                                                           550
            ptm->TexMemoryFree +=
              (pAlloc->pEnd - pAlloc->pBegin)*sizeof(unsigned);
            ptm->NumTextures--;
            free(pAlloc);
            ptm->pTexEntryTable[id].pAlloc = 0;
            ptm->pTexEntryTable[id].valid = 0;
            ptm->pTexEntryTable[id].updated = 0xF;
```

```
            flush_variable(&ptm->pTexEntryTable[id], sizeof(TexEntry));
                                                                              560
            // don't need to flush TexAllocation, as only the main
            // processor (this one) ever uses that data.
            // likewise, don't need to flush TexManager out.
          }
      }

    static_send(0);
  }
  break;
                                                                              570
case RENDER_UPLOAD_TEXTURE:
  {
    unsigned size, temp;
    signed id;
    unsigned *texbegin = 0, *texend = 0, *texiter;

    static_send(2);
    id = static_receive();
    size = static_receive();
    static_send(0); // reading back 0                                         580

    // hack: at this point, static network sends us
    //        next word unconditionally

    if(size > 0)
      {
        if(id >= 0 && id < ptm->MaxTextures)
          if(ptm->pTexEntryTable[id].valid && ptm->pTexEntryTable[id].pAlloc)
            {
              texiter = texbegin = ptm->pTexEntryTable[id].pAlloc->pBegin;    590
              texend = ptm->pTexEntryTable[id].pAlloc->pEnd;
            }

        // get first word
        temp = static_receive();
        size--;

        if(texiter != texend)
          *texiter = temp;
                                                                              600
        texiter++;

        static_send(size); // get rest of words
        for( ;texiter < texend && size > 0; size--, texiter++)
          {
```

```
        *texiter = static_receive();
      }

      // program screwed up, but try not to lock up...
      for( ; size > 0; size-- )                                          610
        static_receive();

      // flush out what we uploaded
      if(texbegin != texend)
        flush_variable(texbegin, (texend - texbegin)*sizeof(unsigned));

      static_send(0); // reading back 0

    }
                                                                         620
  }
  break;

case RENDER_TEXMEM_AVAIL:
  static_send(0);
  static_send(1);
  static_send(ptm->TexMemoryFree); // texmem avail
  break;

case RENDER_COMPACT_TEXMEM:                                              630
  // TODO
  static_send(0);
  static_send(0);
  break;

case RENDER_WRITE_FB:
  {
    int x,y;
    unsigned page,rgbx;
    static_send(4);                                                      640
    x = static_receive(); // x
    y = static_receive(); // y
    page = static_receive(); // page
    rgbx = static_receive(); // rgbx
    fb_set_pixel_rgbx(x, y, rgbx,
                      (page & FBMODE_BACK) != 0,
                      (page & FBMODE_FRONT) != 0);

    static_send(0);
  }                                                                      650
  break;

case RENDER_WRITE_FB_BLOCK:
```

```
  {
    unsigned length;
    // TODO
    static_send(4);
    static_receive(); // x
    static_receive(); // y
    static_receive(); // page                                  660
    length = static_receive(); // length
    static_send(0);

    if(length > 0)
      {
        // first rgbx in block
        static_receive();
        length--;
        static_send(length);
        for( ; length > 0 ; length--)                           670
          static_receive(); // rest of rgbx's
        static_send(0);
      }

  }
  break;

case RENDER_READ_FB:
  {
    int x,y;                                                    680
    unsigned page;
    static_send(3);
    x = static_receive(); // x
    y = static_receive(); // y
    page = static_receive(); // page - 0 = back, 1 = front
    static_send(1);
    static_send(fb_read_pixel(x, y, page));
  }
  break;
                                                                690
case RENDER_READ_FB_BLOCK:
  {
    unsigned length;
    // TODO
    static_send(4);
    static_receive(); // x
    static_receive(); // y
    static_receive(); // page
    length = static_receive(); // length
    static_send(length);                                        700
    for( ; length > 0; length--)
```

```
      static_send(0); // read rgbx
    }

    break;

  case RENDER_WRITE_Z:
    {
      unsigned x,y;
      signed val;                                                710
      static_send(3);
      x = static_receive(); // x
      y = static_receive(); // y
      val = static_receive(); // val
      pzbd->buf[x+VWIDTH*y] = val;
      flush_word(&pzbd->buf[x+VWIDTH*y]);
      static_send(0);
    }
    break;
                                                                 720
  case RENDER_WRITE_Z_BLOCK:
    {
      unsigned length;
      // TODO
      static_send(3);
      static_receive(); // x
      static_receive(); // y
      length = static_receive(); // length
      static_send(0);
                                                                 730
      if(length > 0)
        {
          // first val in block
          static_receive();
          length--;
          static_send(length);
          for( ; length > 0 ; length--)
            static_receive(); // rest of val's
          static_send(0);
        }                                                        740

    }
    break;

  case RENDER_READ_Z:
    {
      unsigned x,y;
      static_send(2);
      x = static_receive(); //x
```

```
    y = static_receive(); //y                                          750
    static_send(1);
    invalidate_word(&pzbd->buf[x+VWIDTH*y]);
    static_send(pzbd->buf[x+VWIDTH*y]); // val
  }
  break;

case RENDER_READ_Z_BLOCK:
  {
    unsigned length;
    // TODO                                                            760
    static_send(3);
    static_receive(); // x
    static_receive(); // y
    length = static_receive(); // length
    static_send(length);
    for( ; length > 0; length--)
      static_send(0); // read val
  }
  break;
                                                                       770
case RENDER_RESET:
  {
    int i;
    static_send(0);
    static_send(0);

    // clear out render state
    ClearRenderState();

    // clear out texture memory                                        780

    // (assuming that everything is set up correctly!)
    for(i = 0; i < ptm->MaxTextures; i++)
      {
        if(ptm->pTexEntryTable[i].valid)
          {
            ptm->pTexEntryTable[i].valid = 0;
            free(ptm->pTexEntryTable[i].pAlloc);
          }
      }                                                                790

    ptm->TexMemoryFree = ptm->TexMemorySize;
    ptm->NumTextures = 0;
    ptm->pAllocHead = 0;
    ptm->pAllocTail = 0;
```

```
        // clear out z buffer
        pzbd->fbmode = FBMODE_BACK;
                                                                         800
#ifdef INIT_ZBUF
        for(i = 0; i < VWIDTH*VHEIGHT; i++)
          {
            pzbd->buf[i] = 0x7FFFFFFF;
          }

#endif

        // tell fb to reset
        fb_reset();                                                      810
      }
      break;

    case RENDER_HALT:
      static_send(0);
      static_send(0);
      // halting!
      while(1);
      break;
                                                                         820
    default:
      static_send(0);
      static_send(0);
    }

  return 0;
}

void DoSceneStream(void);
                                                                         830
void begin(void)
{
  int i;
  // things we need to do on initial bootup:

  // for framebuffer code
  //      funny desty (sender x,y = 0)
  fb_init_fbhdr(0,0);

  //// allocate shared memory for stage1.                                840
  prs = (RenderState*)malloc(sizeof(RenderState));

  //// set renderstate to startup defaults
  ClearRenderState();
```

```
  //// allocate tex mem, and tex control structures.
  ptm = (TexManager*)malloc(sizeof(TexManager));
  ptm->pTexMemory = (unsigned*)malloc(TEXMEMSIZE*sizeof(unsigned));
  ptm->TexMemorySize = ptm->TexMemoryFree= TEXMEMSIZE;
  ptm->pTexEntryTable = (TexEntry*)malloc(TEXENTRIES*sizeof(TexEntry));          850
  ptm->MaxTextures = TEXENTRIES;
  for(i = 0; i < TEXENTRIES; i++)
    {
      ptm->pTexEntryTable[i].valid = 0;
      ptm->pTexEntryTable[i].updated = 0;
    }
  ptm->NumTextures = 0;
  ptm->pAllocHead = 0; // the funny thing, is that the allocation list will be in
  ptm->pAllocTail = 0; // stage 1's memory, which is OK since we don't really need it.
                                                                                  860
  pzbd = (ZBufData*)malloc(sizeof(ZBufData));

  //initialize zbd to zero
  pzbd->fbmode = FBMODE_BACK;

#ifdef INIT_ZBUF
  for(i = 0; i < VWIDTH*VHEIGHT; i++)
    {
      pzbd->buf[i] = 0x7FFFFFFF;
    }                                                                             870
#endif

  //// initialize framebuffers to all black (optional step with compiler def?)
#ifdef INIT_FB_BLACK
  // TODO
#endif


  while(1)
    {                                                                             880
      // going into command mode:
      //// set up the static network
      setup_switch_main();

      //// send acknowledgement out to renderhost, telling it we're booted
      //// and ready for commands.
      static_send(1);

      // command mode:
      //// loop reading command, performing action, sending back                  890
      ////  responses if necessary. Everything can be done with memory
      ////  accesses (make sure to flush changed memory) and gdn messages
      ////  (for writing to framebuffer) except scenestream mode.
```

```
      while(!doCommand(static_receive()));

      // going into scenestream:

      setup_switch_scenestream();

      //// send one word out of static network to tell it we're ready to      900
      ////  accept messages/
      static_send(1);

      // scenestream mode:
      DoSceneStream();

    }

}
                                                                            910
// a helper function - reads from the sn,
// and  decrements the block variable. if it's
// zero, reads a new one from the static network.
// assumes that *b is currently >0!
static inline unsigned block_receive(unsigned *b)
{
  unsigned temp = static_receive();
  if(--(*b) <= 0)
    *b = static_receive();
  return temp;                                                              920
}
static inline float block_receive_f(unsigned *b)
{
  float temp;
  temp = static_receive_f();
  if(--(*b) <= 0)
    *b = static_receive();
  return temp;
}
                                                                            930
void Stage2(TransPrim *tp);

void DoSceneStream()
{
  while(1)
    {
      int i,j;
      unsigned blockLength;
      unsigned numVerts = 0; // number of vertices we've got (don't render
      //                        without 3.                                  940
      InputPrim ip; // data stored on input
```

```
TransPrim tp; // object to output

float tnx,tny,tnz; // temp normal in world coords, for lighting
float tempz; // temp z before converting to fixed point.

unsigned unordered = 0; // is it an unordered prim, or ordered?
unsigned isendscene = 0;

unsigned visible = 0; // is visible or was clipped?                    950

// clear input prim + trans prim
for(i = 0; i<3; i++)
  {
    ip.v[i].x = 0.0f;
    ip.v[i].y = 0.0f;
    ip.v[i].z = 0.0f;
    ip.v[i].nx = 1.0f;
    ip.v[i].ny = 0.0f;
    ip.v[i].nz = 0.0f;                                                 960
    ip.v[i].u = 0.0f;
    ip.v[i].v = 0.0f;
    ip.v[i].rgba = 0;

    tp.v[i].x = 0.0f;
    tp.v[i].y = 0.0f;
    tp.v[i].z = 0;
    tp.v[i].w1 = 1.0f;
    tp.v[i].u = 0.0f;
    tp.v[i].v = 0.0f;                                                  970
    tp.v[i].r = 0.0f;
    tp.v[i].g = 0.0f;
    tp.v[i].b = 0.0f;
    tp.v[i].a = 0.0f;
  }


// read blockLength from static network
blockLength = static_receive();
                                                                       980

// now, loop while blockLength!=0:
while(blockLength > 0)
  {
    unsigned cmd;
    //// read next command. decrement blockLength. if blockLength = 0 now,
    ////  read in blockLength.
    cmd = block_receive(&blockLength);
```

```
//// switch on command:                                                        990
////  read in command's data. update:
////   ip, *rs, numVerts.
////    when making a new vertex, get data from *rs.
////    when changing normals or whatnot, update *rs.
////   keep track of blockLength, if it reaches 0,
////    read in blockLength again. if read blockLength = 0, break.
////  if command is endscene.... set isendscene = 1

switch(cmd)
  {                                                                          1000
  case RENDER_ENDSCENE:
    isendscene = 1;
    break;

  case RENDER_VERTEX:
    if(blockLength > 0)
      {
        ip.v[numVerts].x = block_receive_f(&blockLength);
        ip.v[numVerts].y = block_receive_f(&blockLength);
        ip.v[numVerts].z = block_receive_f(&blockLength);                    1010
        ip.v[numVerts].u = block_receive_f(&blockLength);
        ip.v[numVerts].v = block_receive_f(&blockLength);
        ip.v[numVerts].nx = prs->nx;
        ip.v[numVerts].ny = prs->ny;
        ip.v[numVerts].nz = prs->nz;

        ip.v[numVerts].rgba = prs->rgba;

        numVerts++;
      }                                                                      1020

    break;

  case RENDER_COLOR:
    if(blockLength > 0)
      {
        prs->rgba = block_receive(&blockLength);
      }
    break;
                                                                             1030
  case RENDER_MODELMATRIX:
    if(blockLength > 0)
      {
        for(i = 0; i < 4; i++)
          for(j = 0; j < 4; j++)
            {
              prs->ModelToWorld[i][j] = block_receive_f(&blockLength);
```

```
        }

      MatrixMatrixMult(prs->ModelToView, prs->WorldToView,                1040
                       prs->ModelToWorld);

      MatrixInvTrans(prs->NormalToWorld, prs->ModelToWorld);
    }
  break;

case RENDER_VIEWMATRIX:
  if(blockLength > 0)
    {
      for(i = 0; i < 4; i++)                                               1050
        for(j = 0; j < 4; j++)
          {
            prs->WorldToView[i][j] = block_receive_f(&blockLength);
          }

      MatrixMatrixMult(prs->ModelToView, prs->WorldToView,
                       prs->ModelToWorld);
    }
  break;
                                                                           1060
case RENDER_NORMAL:
  if(blockLength > 0)
    {
      prs->nx = block_receive_f(&blockLength);
      prs->ny = block_receive_f(&blockLength);
      prs->nz = block_receive_f(&blockLength);
    }
  break;

case RENDER_SET_LIT:                                                       1070
  if(blockLength > 0)
    {
      prs->pInfo.p.Mode.lit = block_receive(&blockLength);
    }
  break;

case RENDER_SET_USEAMB:
  if(blockLength > 0)
    {
      prs->pInfo.p.Mode.useamb = block_receive(&blockLength);             1080
    }
  break;

case RENDER_SET_USEDIR:
  if(blockLength > 0)
```

```
      {
        prs->pInfo.p.Mode.usedir = block_receive(&blockLength);
      }
    break;
                                                                      1090
  case RENDER_SET_TEXMODE:
    if(blockLength > 0)
      {
        prs->pInfo.p.Mode.texmode = block_receive(&blockLength);
      }
    break;

  case RENDER_SET_TEXALPHA:
    if(blockLength > 0)
      {                                                               1100
        prs->pInfo.p.Mode.texalpha = block_receive(&blockLength);
      }
    break;

  case RENDER_SET_COLALPHA:
    if(blockLength > 0)
      {
        prs->pInfo.p.Mode.colalpha = block_receive(&blockLength);
      }
    break;                                                            1110

  case RENDER_SET_COLINTERP:
    if(blockLength > 0)
      {
        prs->pInfo.p.Mode.colinterp = block_receive(&blockLength);
      }
    break;

  case RENDER_SET_LITINTERP:
    if(blockLength > 0)                                               1120
      {
        prs->pInfo.p.Mode.litinterp = block_receive(&blockLength);
      }
    break;

  case RENDER_SET_TEXINTERP:
    if(blockLength > 0)
      {
        prs->pInfo.p.Mode.texinterp= block_receive(&blockLength);
      }                                                               1130
    break;

  case RENDER_SET_OUTOFORDER:
```

```
  if(blockLength > 0)
    {
      prs->pInfo.p.Mode.outoforder = block_receive(&blockLength);
    }
  break;

case RENDER_SET_TEXTILE:                                                    1140
  if(blockLength > 0)
    {
      prs->pInfo.p.Mode.textile = block_receive(&blockLength);
    }
  break;

case RENDER_SET_NOUSEZ:
  if(blockLength > 0)
    {
      prs->pInfo.p.Mode.nousez = block_receive(&blockLength);              1150
    }
  break;

case RENDER_SET_NOWRITEZ:
  if(blockLength > 0)
    {
      prs->pInfo.p.Mode.nowritez = block_receive(&blockLength);
    }
  break;
                                                                            1160
case RENDER_SET_TEXTUREID:
  if(blockLength > 0)
    {
      prs->pInfo.TextureID = block_receive(&blockLength);
    }
  break;

case RENDER_COLTEXBALANCE:
  if(blockLength > 0)
    {                                                                       1170
      prs->pInfo.ColTexBalance = block_receive_f(&blockLength);
    }
  break;

case RENDER_ALPHATHRESH:
  if(blockLength > 0)
    {
      prs->pInfo.alphaThresh = block_receive(&blockLength);
    }
  break;                                                                    1180
```

```
      case RENDER_AMBCOLOR:
        if(blockLength > 0)
          {
            prs->pInfo.ambColor = block_receive(&blockLength);
          }
        break;

      case RENDER_DIRCOLOR:
        if(blockLength > 0)                                              1190
          {
            prs->pInfo.dirColor = block_receive(&blockLength);
          }
        break;

      case RENDER_DIRLIGHT:
        if(blockLength > 0)
          {
            prs->ldx = block_receive_f(&blockLength);
            prs->ldy = block_receive_f(&blockLength);                    1200
            prs->ldz = block_receive_f(&blockLength);

            prs->dirdefined = 1;
          }
        break;

      case RENDER_AMBREFLECT:
        if(blockLength > 0)
          {
            prs->ambreflect = block_receive(&blockLength);               1210
          }
        break;

      case RENDER_DIRREFLECT:
        if(blockLength > 0)
          {
            prs->dirreflect = block_receive(&blockLength);
          }
        break;
                                                                         1220
      default:

      }
  }


if( numVerts >= 3 )
  {
```

```
// do clipping first - we don't want to update the sequence number                1230
// if the primitive is clipped!

// do screenspace transform
for(i = 0; i < 3; i++)
  {
    MatrixMult(&tp.v[i].x, &tp.v[i].y, &tp.v[i].z, &tp.v[i].w1,
               prs->ModelToView,
               ip.v[i].x, ip.v[i].y, ip.v[i].z, 1.0f);

    // todo: optimize for when we don't use z coordinate?                          1240
  }

// clipping. necessary:
//   - backface culling
//   - at least dropping polys behind near plane
// would be nice TODO:
//   - full frustum culling
//   - clipping to near plane, including splitting triangles, regen vertices+values
//      (this can create more than one prim - complexifying this code!)
                                                                                   1250
visible = 1;

// near plane and singularity dropping
if( tp.v[0].w1>0 && (tp.v[0].z <= -tp.v[0].w1) ||
    tp.v[0].w1<0 && (tp.v[0].z >= -tp.v[0].w1) ||
    tp.v[1].w1>0 && (tp.v[1].z <= -tp.v[1].w1) ||
    tp.v[1].w1<0 && (tp.v[1].z >= -tp.v[1].w1) ||
    tp.v[2].w1>0 && (tp.v[2].z <= -tp.v[2].w1) ||
    tp.v[2].w1<0 && (tp.v[2].z >= -tp.v[2].w1)
    || abs(tp.v[0].w1) <= 1e-100                                                   1260
    || abs(tp.v[1].w1) <= 1e-100
    || abs(tp.v[2].w1) <= 1e-100)
  visible = 0;

// backface culling - vertices are defined clockwise when prim facing the screen
if( visible )
  {
    //if point 2 is on rhs of point 0->1 vector, keep...
    // ax + by + c > 0
    // a = y1 - y0, b = x0 - x1, c = y0x1-y1x0                                     1270

    if((tp.v[1].y - tp.v[0].y)*tp.v[2].x + (tp.v[0].x - tp.v[1].x)*tp.v[2].y
       + tp.v[0].y * tp.v[1].x - tp.v[1].y*tp.v[0].x >= 0 )
      visible = 0;
  }

}
```

```
if(isendscene)                                                              1280
  {
    return;
  }
else
  {

    if(visible)
      {
        // copy info over to transprim
        tp.pInfo = prs->pInfo;                                              1290

        // ambient light modulation
        if(tp.pInfo.p.Mode.lit && tp.pInfo.p.Mode.useamb)
          { // modulate amb intensity with reflectivity
            unsigned intens = prs->pInfo.ambColor & 0x0FF;
            prs->pInfo.ambColor &= 0xFFFFFF00;
            // max of each is 255 - treated as 1.0 (fixed point modulation)
            prs->pInfo.ambColor |= ((prs->ambreflect * intens) + 255 ) >> 8;
          }
                                                                            1300
        if(!prs->dirdefined)
          tp.pInfo.p.Mode.usedir = 0;

        // directed light modulation
        if (tp.pInfo.p.Mode.lit && tp.pInfo.p.Mode.usedir)
          for(i = 0; i < 3; i++)
            {
              float w;
              float prod;
              //// transform normal into world coordinates tnx tny tnz      1310
              MatrixMult(&tnx, &tny, &tnz, &w,
                         prs->NormalToWorld,
                         ip.v[i].nx, ip.v[i].ny, ip.v[i].nz, 1.0f);

              // normalize normal
              w = 1.0f/sqrtf(tnx*tnx+tny*tny+tnz*tnz);
              tnx *= w;
              tny *= w;
              tnz *= w;
                                                                            1320
              //// dot product tn<xyz> with rs->ld<xyz> (be careful of sign!)
              prod = - tnx*prs->ldx - tny*prs->ldy - tnz*prs->ldz;

              tp.v[i].intensity = (prod <= 0) ? 0
                : ( prod * ((float)prs->dirreflect)/255.0f
```

```
                * ((float)(prs->pInfo.dirColor & 0x0FF))/255.0f );
            }


        // perspective division and streaming!                                    1330
        for(i = 0; i < 3; i++)
          {
            tp.v[i].w1 = 1/tp.v[i].w1;
            tp.v[i].x = tp.v[i].w1 * tp.v[i].x;
            tp.v[i].y = tp.v[i].w1 * tp.v[i].y;
            if (!(tp.pInfo.p.Mode.nowritez && tp.pInfo.p.Mode.nousez))
              tp.v[i].z = tp.v[i].w1 * tp.v[i].z;
            if (tp.pInfo.p.Mode.texmode != 0 && tp.pInfo.p.Mode.texmode != 2)
              {
                tp.v[i].r = tp.v[i].w1 * (ip.v[i].rgba >> 24);                       1340
                tp.v[i].g = tp.v[i].w1 * (( ip.v[i].rgba << 8 ) >> 24);
                tp.v[i].b = tp.v[i].w1 * (( ip.v[i].rgba << 16 ) >> 24);
              }
            if (tp.pInfo.p.Mode.colalpha != 0)
              tp.v[i].a = tp.v[i].w1 * (ip.v[i].rgba & 0x0FF);
            if (tp.pInfo.p.Mode.texmode > 1)
              {
                tp.v[i].u = tp.v[i].w1 * ip.v[i].u;
                tp.v[i].v = tp.v[i].w1 * ip.v[i].v;
              }                                                                      1350
            if(tp.pInfo.p.Mode.lit && tp.pInfo.p.Mode.usedir)
              tp.v[i].intensity = tp.v[i].w1 * tp.v[i].intensity;
          }


        magic_perf_drawnprim();

        Stage2(&tp);

                                                                                     1360
       }

     }
   }
}

void Stage3(PrimInfo * pi, UntexFragment * utf);

void Stage2(TransPrim *tp)
{                                                                                    1370
  int ii,j;
  UntexFragment utf; // fragmetns we output
  float ulx;
```

```
float uly;
float lrx;
float lry;

unsigned correctinterp; // are we doing any perspective-correct interpolation?

float w10,w11,w12;                                                          1380

// line equations: L - line value for top left corner axul+byul+c
// (incremental    tL - temp value for each line
//  model for      ttL - temp value for each pixel
//  better         adx - x increment for L
//  performance)   bdy - y increment for L
float L0, tL0, adx0, bdy0; // v0->v1
float L1, tL1, adx1, bdy1; // v1->v2
float L2, tL2, adx2, bdy2; // v2->v0
// clockwise, so rhs is < 0:                                                1390
//  a = y2-y1, b=x1-x2, c=y1x2-y2x1

// plane equations, x - top left corner value
//                  tx - temp for each line
//                  ttx - temp for each pixel
//                  xdx - x increment for x
//                  xdy - y increment for x
// don't do incremental model for z, do calc for
// each point (better accuracy?)
float za, zb, zc;                                                           1400
float r, tr, rdx, rdy;
float g, tg, gdx, gdy;
float b, tb, bdx, bdy;
float a, ta, adx, ady;
float u, tu, udx, udy;
float v, tv, vdx, vdy;
float i, ti, idx, idy;
float w1, tw1, w1dx, w1dy;
float me, b1e, b2e, detM1;
// precalc me = x2y3-x3y2                                                   1410
//         b1e = z2y3-z3y2
//         b2e = x2z3-x3z2
// det M  = x1(y2-y3)-y1(x2-x3)+me
// det B1 = z1(y2-y3)-y1(z2-z3)+b1e
// det B2 = x1(z2-z3)-z1(x2-x3)+b2e
// det B3 = -x1(b1e)-y1(b2e)+z1(me)
// A = detB1/detM, B=detB2/detM, C=detB3/detM
// Ax+By+C = z

// when w1 is interped, invert and multiply each                            1420
// other interpreted value by it, for perspect correct.
```

```
//---this is where code actually begins---

unsigned nousez, nowritez, texmode, colinterp, colalpha;
unsigned lit, usedir, litinterp;

// unpack bitfields for better performance
nousez = tp->pInfo.p.Mode.nousez;
nowritez = tp->pInfo.p.Mode.nowritez;                                    1430
texmode = tp->pInfo.p.Mode.texmode;
colinterp = tp->pInfo.p.Mode.colinterp;
colalpha = tp->pInfo.p.Mode.colalpha;
lit = tp->pInfo.p.Mode.lit;
usedir = tp->pInfo.p.Mode.usedir;
litinterp = tp->pInfo.p.Mode.litinterp;

////// get inverses of (w^-1) values, to multiply with
//////  parameters to get their real values.
w10 = 1/tp->v[0].w1;                                                     1440
w11 = 1/tp->v[1].w1;
w12 = 1/tp->v[2].w1;

////// scale prim's x and y values to screen space, using compiler defs.
////// VWIDTH, VHEIGHT. scale so -1 -> 0, and 1 -> VWIDTH/HEIGHT
////// (pixels are centered on .5 steps - pixel 0 is at 0.5, pixel 1 is at 1.5, etc.
tp->v[0].x = tp->v[0].x*(VWIDTH/2.0f) + VWIDTH/2.0f;
tp->v[0].y = tp->v[0].y*(VHEIGHT/2.0f) + VHEIGHT/2.0f;
tp->v[1].x = tp->v[1].x*(VWIDTH/2.0f) + VWIDTH/2.0f;
tp->v[1].y = tp->v[1].y*(VHEIGHT/2.0f) + VHEIGHT/2.0f;                   1450
tp->v[2].x = tp->v[2].x*(VWIDTH/2.0f) + VWIDTH/2.0f;
tp->v[2].y = tp->v[2].y*(VHEIGHT/2.0f) + VHEIGHT/2.0f;

// for bounding box
//  minimum is lowest n such that n+0.5 is greater than or equal to lowest pixel coord.
//  maximum is highest n such that n+0.5 is less than or equal to highest pixel coord.
//  we want the bounding box to be the lowest and highest n+0.5 that's within
//   the prim.
//  3.1 ... min at 3, max at 2 ... 3.6 min at 4, max at 3
                                                                         1460
ulx = (signed)(tp->v[0].x+0.5f);
lrx = (signed)(tp->v[0].x-0.5f);
uly = (signed)(tp->v[0].y+0.5f);
lry = (signed)(tp->v[0].y-0.5f);

for(ii = 1; ii < 3; ii++)
  {
    ulx = (ulx <= (signed)(tp->v[ii].x + 0.5f)) ? ulx : (signed)(tp->v[ii].x + 0.5f);
    uly = (uly <= (signed)(tp->v[ii].y + 0.5f)) ? uly : (signed)(tp->v[ii].y + 0.5f);
```

```
    lrx = (lrx >= (signed)(tp->v[ii].x - 0.5f)) ? lrx : (signed)(tp->v[ii].x - 0.5f);      1470
    lry = (lry >= (signed)(tp->v[ii].y - 0.5f)) ? lry : (signed)(tp->v[ii].y - 0.5f);
  }

  if(ulx < 0) ulx = 0;
  if(ulx >= VWIDTH) ulx = VWIDTH - 1;
  if(lrx < 0) ulx = 0;
  if(lrx >= VWIDTH) lrx = VWIDTH - 1;
  if(uly < 0) uly = 0;
  if(uly >= VHEIGHT) uly = VHEIGHT - 1;
  if(lry < 0) uly = 0;                                                                     1480
  if(lry >= VHEIGHT) lry = VHEIGHT - 1;

  // move to center of pixels
  ulx+=0.5;
  uly+=0.5;
  lrx+=0.5;
  lry+=0.5;

  ////// set up plane equations for each line, z, rgba, uv, intensity:
  //  a = y2-y1, b=x1-x2, c=y1x2-y2x1 // rhs is inside // clockwise faces front            1490
  adx0 = tp->v[1].y - tp->v[0].y;
  bdy0 = tp->v[0].x - tp->v[1].x;
  L0 = adx0*ulx + bdy0*uly + tp->v[0].y*tp->v[1].x - tp->v[1].y*tp->v[0].x;
  adx1 = tp->v[2].y - tp->v[1].y;
  bdy1 = tp->v[1].x - tp->v[2].x;
  L1 = adx1*ulx + bdy1*uly + tp->v[1].y*tp->v[2].x - tp->v[2].y*tp->v[1].x;
  adx2 = tp->v[0].y - tp->v[2].y;
  bdy2 = tp->v[2].x - tp->v[0].x;
  L2 = adx2*ulx + bdy2*uly + tp->v[2].y*tp->v[0].x - tp->v[0].y*tp->v[2].x;
                                                                                           1500
  correctinterp = 0;

  // these values are the same for all parameters for x,y
  me  = tp->v[1].x*tp->v[2].y - tp->v[2].x*tp->v[1].y;
  detM1 = 1/(tp->v[0].x*
           (tp->v[1].y-tp->v[2].y)-tp->v[0].y*(tp->v[1].x-tp->v[2].x)+me);


  if(!nousez || !nowritez)
    {                                                                                      1510
      // set up z interp:
      b1e = tp->v[1].z*tp->v[2].y - tp->v[2].z*tp->v[1].y;
      b2e = tp->v[1].x*tp->v[2].z - tp->v[2].x*tp->v[1].z;
      za = detM1*(tp->v[0].z*(tp->v[1].y-tp->v[2].y)
                -tp->v[0].y*(tp->v[1].z-tp->v[2].z)+b1e);
      zb = detM1*(tp->v[0].x*(tp->v[1].z-tp->v[2].z)
                -tp->v[0].z*(tp->v[1].x-tp->v[2].x)+b2e);
```

```
      zc = detM1*(tp->v[0].z*me - tp->v[0].x*b1e - tp->v[0].y*b2e);
    }
                                                                              1520
  if (texmode != 0 && texmode != 2)
    {
      if(colinterp)
        {
          //set up r,g,b interp
          correctinterp = 1;
          b1e = tp->v[1].r*tp->v[2].y - tp->v[2].r*tp->v[1].y;
          b2e = tp->v[1].x*tp->v[2].r - tp->v[2].x*tp->v[1].r;
          rdx = detM1*(tp->v[0].r*(tp->v[1].y-tp->v[2].y)
                    -tp->v[0].y*(tp->v[1].r-tp->v[2].r)+b1e);                1530
          rdy = detM1*(tp->v[0].x*(tp->v[1].r-tp->v[2].r)
                    -tp->v[0].r*(tp->v[1].x-tp->v[2].x)+b2e);
          r = rdx*ulx + rdy*uly +
            detM1*(tp->v[0].r*me - tp->v[0].x*b1e - tp->v[0].y*b2e);

          b1e = tp->v[1].g*tp->v[2].y - tp->v[2].g*tp->v[1].y;
          b2e = tp->v[1].x*tp->v[2].g - tp->v[2].x*tp->v[1].g;
          gdx = detM1*(tp->v[0].g*(tp->v[1].y-tp->v[2].y)
                    -tp->v[0].y*(tp->v[1].g-tp->v[2].g)+b1e);
          gdy = detM1*(tp->v[0].x*(tp->v[1].g-tp->v[2].g)                    1540
                    -tp->v[0].g*(tp->v[1].x-tp->v[2].x)+b2e);
          g = gdx*ulx + gdy*uly +
            detM1*(tp->v[0].g*me - tp->v[0].x*b1e - tp->v[0].y*b2e);

          b1e = tp->v[1].b*tp->v[2].y - tp->v[2].b*tp->v[1].y;
          b2e = tp->v[1].x*tp->v[2].b - tp->v[2].x*tp->v[1].b;
          bdx = detM1*(tp->v[0].b*(tp->v[1].y-tp->v[2].y)
                    -tp->v[0].y*(tp->v[1].b-tp->v[2].b)+b1e);
          bdy = detM1*(tp->v[0].x*(tp->v[1].b-tp->v[2].b)
                    -tp->v[0].b*(tp->v[1].x-tp->v[2].x)+b2e);                1550
          b = bdx*ulx + bdy*uly +
            detM1*(tp->v[0].b*me - tp->v[0].x*b1e - tp->v[0].y*b2e);
        }
      else
        {
          // color is average of vertices
          r = (tp->v[0].r*w10 + tp->v[1].r*w11 + tp->v[2].r*w12)/3;
          g = (tp->v[0].g*w10 + tp->v[1].g*w11 + tp->v[2].g*w12)/3;
          b = (tp->v[0].b*w10 + tp->v[1].b*w11 + tp->v[2].b*w12)/3;
        }                                                                    1560
      if (colalpha != 0)
        {
          if(colinterp)
            {
              // set up a interp
```

```
            correctinterp = 1;

            b1e = tp->v[1].a*tp->v[2].y - tp->v[2].a*tp->v[1].y;
            b2e = tp->v[1].x*tp->v[2].a - tp->v[2].x*tp->v[1].a;
            adx = detM1*(tp->v[0].a*(tp->v[1].y-tp->v[2].y)                    1570
                        -tp->v[0].y*(tp->v[1].a-tp->v[2].a)+b1e);
            ady = detM1*(tp->v[0].x*(tp->v[1].a-tp->v[2].a)
                        -tp->v[0].a*(tp->v[1].x-tp->v[2].x)+b2e);
            a = adx*ulx + ady*uly
              + detM1*(tp->v[0].a*me - tp->v[0].x*b1e - tp->v[0].y*b2e);
          }
        else
          {
            // alpha is average of vertices
            a = (tp->v[0].a*w10 + tp->v[1].a*w11 + tp->v[2].a*w12)/3;          1580
          }
      }
  }

if (texmode > 1)
  {
    //set up u,v interp
    correctinterp = 1;

    b1e = tp->v[1].u*tp->v[2].y - tp->v[2].u*tp->v[1].y;                        1590
    b2e = tp->v[1].x*tp->v[2].u - tp->v[2].x*tp->v[1].u;
    udx = detM1*(tp->v[0].u*(tp->v[1].y-tp->v[2].y)
                -tp->v[0].y*(tp->v[1].u-tp->v[2].u)+b1e);
    udy = detM1*(tp->v[0].x*(tp->v[1].u-tp->v[2].u)
                -tp->v[0].u*(tp->v[1].x-tp->v[2].x)+b2e);
    u = udx*ulx + udy*uly +
      detM1*(tp->v[0].u*me - tp->v[0].x*b1e - tp->v[0].y*b2e);

    b1e = tp->v[1].v*tp->v[2].y - tp->v[2].v*tp->v[1].y;
    b2e = tp->v[1].x*tp->v[2].v - tp->v[2].x*tp->v[1].v;                        1600
    vdx = detM1*(tp->v[0].v*(tp->v[1].y-tp->v[2].y)
                -tp->v[0].y*(tp->v[1].v-tp->v[2].v)+b1e);
    vdy = detM1*(tp->v[0].x*(tp->v[1].v-tp->v[2].v)
                -tp->v[0].v*(tp->v[1].x-tp->v[2].x)+b2e);
    v = vdx*ulx + vdy*uly +
      detM1*(tp->v[0].v*me - tp->v[0].x*b1e - tp->v[0].y*b2e);
  }

if (lit && usedir)
  {                                                                             1610
    if(litinterp)
      {
        // set up intensity interp
```

```
      correctinterp = 1;

      b1e = tp->v[1].intensity*tp->v[2].y - tp->v[2].intensity*tp->v[1].y;
      b2e = tp->v[1].x*tp->v[2].intensity - tp->v[2].x*tp->v[1].intensity;
      idx = detM1*(tp->v[0].intensity*(tp->v[1].y-tp->v[2].y)
                -tp->v[0].y*(tp->v[1].intensity-tp->v[2].intensity)+b1e);
      idy = detM1*(tp->v[0].x*(tp->v[1].intensity-tp->v[2].intensity)          1620
                -tp->v[0].intensity*(tp->v[1].x-tp->v[2].x)+b2e);
      i = idx*ulx + idy*uly +
        detM1*(tp->v[0].intensity*me - tp->v[0].x*b1e - tp->v[0].y*b2e);
    }
  else
    {
      // intens is average of vertices
      i = (tp->v[0].intensity*w10 + tp->v[1].intensity*w11
          + tp->v[2].intensity*w12)/3;
    }                                                                          1630
  }

if(correctinterp)
  {
    // set up w1 interp

    b1e = tp->v[1].w1*tp->v[2].y - tp->v[2].w1*tp->v[1].y;
    b2e = tp->v[1].x*tp->v[2].w1 - tp->v[2].x*tp->v[1].w1;
    w1dx = detM1*(tp->v[0].w1*(tp->v[1].y-tp->v[2].y)
                -tp->v[0].y*(tp->v[1].w1-tp->v[2].w1)+b1e);                    1640
    w1dy = detM1*(tp->v[0].x*(tp->v[1].w1-tp->v[2].w1)
                -tp->v[0].w1*(tp->v[1].x-tp->v[2].x)+b2e);
    w1 = w1dx*ulx + w1dy*uly
      + detM1*(tp->v[0].w1*me - tp->v[0].x*b1e - tp->v[0].y*b2e);


  }

////// from uly to lry
for(ii = (int)uly; ii <= (int)lry; ii++)                                       1650
  {
    unsigned gotrow = 0;
    tL0 = L0; tL1 = L1; tL2 = L2;
    L0+=bdy0; L1+= bdy1; L2+=bdy2;

    // also initialize incremental interp for r, g, b, a, u, v, i, and w1 for row
    // (surrounding everything by if clause adds too much overhead
    if(colinterp)
      {
        tr = r; tg = g; tb = b;                                                1660
        r+=rdy; g+=gdy; b+=bdy;
```

```
      ta = a;
      a+=ady;
    }
  tu = u; tv = v;
  u+=udy; v+=vdy;
  if(litinterp)
    {
      ti = i;
      i+=idy;                                                    1670
    }
  tw1 = w1;
  w1+=w1dy;

  // from ulx to lrx
  for(j = (int)ulx; j <= (int)lrx; j++)
    {
      if( tL0 < 0 && tL1 < 0 && tL2 < 0)
        {
          float tempw;                                           1680

          gotrow = 1;

          // fill in utf with x,y.
          utf.x = j;
          utf.y = ii;

          // fill in as necessary: z (scaled), u/w1, v/w1,
          // rgba (packed+scaled+/w1), intensity/w1
          if(!nousez || !nowritez)                               1690
            {
              // note: this loses precision. TODO: take full advantage of signed
              // fixed point precision somehow? (use software double-sized ints?)
              float tempz;

              tempz = za * ((float)j + 0.5f) + zb * ((float)ii + 0.5f) + zc;
              utf.z = tempz*(signed)(0x7FFFFFFF);
            }

          if(correctinterp)                                      1700
            {
              tempw = 1/tw1;
            }

          if (texmode != 0 && texmode != 2)
            {
              if(colinterp)
                {
                  utf.rgba = (((unsigned)(tr*tempw+0.5f)) & 0x0FF) << 24 |
```

```
            (((unsigned)(tg*tempw+0.5f)) & 0x0FF) << 16 |                 1710
            (((unsigned)(tb*tempw+0.5f)) & 0x0FF) << 8;
          }
        else
          {
            // color is average of vertices
            utf.rgba = (((unsigned)(r+0.5f)) & 0x0FF) << 24 |
              (((unsigned)(g+0.5f)) & 0x0FF) << 16 |
              (((unsigned)(b+0.5f)) & 0x0FF) << 8;
          }
        if (colalpha != 0)                                                1720
          {
            if(colinterp)
              utf.rgba |= ((unsigned)(ta*tempw+0.5f)) & 0x0FF;
            else
              utf.rgba |= ((unsigned)(a+0.5f)) & 0x0FF;
          }
      }
    if (texmode > 1)
      {
        utf.u = tu * tempw;                                               1730
        utf.v = tv * tempw;
      }
    if (lit && usedir)
      {
        if(litinterp)
          utf.intensity = ti * tempw;
        else
          utf.intensity = i;
      }
                                                                          1740
    magic_perf_fragment();

    Stage3(&tp->pInfo, &utf);

  }
else
  {
    if(gotrow == 1) // we were in the prim, and then left
      {
        break;                                                            1750
      }
  }

tL0+=adx0; tL1+=adx1; tL2+=adx2;

// increment tInterp in dx for r,g,b,a,u,v,i, and w1
tr+=rdx; tg+=gdx; tb+=bdx;
```

```
        ta+=adx;
        tu+=udx; tv+=vdx;
        ti+=idx;                                                    1760
        tw1+=w1dx;
      }
    }

}
static inline void texwrap(float *coord, unsigned mode)
{
  signed intpart;
  // *coord: map down to 0->1 range
  // mode: (0=none,1=repeat, 2=mirror,3=clamp)                      1770

  intpart = (signed)(*coord);

  if(mode == 1)
    {
      *coord = (*coord) - (float)intpart; // fractional part
      if(*coord < 0)
        *coord = 1 + *coord;
    }
  else if(mode == 2)                                                1780
    {
      if(*coord < 0)
        *coord = -*coord;

      if( intpart % 2 ) // if it's odd, do a reverse mapping
        *coord = 1 - (*coord - (float)intpart);
      else
        *coord = *coord - (float)intpart;
    }
                                                                    1790
  // for none and clamp, leave as is.
}

static inline void doFragment(Fragment *f, ZBufData *z,
                              unsigned nousez, unsigned nowritez)
{
  unsigned alpha;
  unsigned addr;
  unsigned temp1, temp;
  unsigned *zloc;                                                   1800

  addr = f->x+f->y*VWIDTH;

  if(!nousez)
    {
```

```
    zloc = &z->buf[addr];
    if( *zloc < f->z )
      {
        return;
      }                                                                  1810
  }

alpha = f->rgba & 0xFF;

if(alpha != 0x0FF)
  { // there's some alpha, read from framebuf

    if( z->fbmode & FBMODE_BACK )
      {
        temp1 = fb_read_pixel_rawaddr(addr, 0);                          1820
        temp = ( ((((f->rgba>>24) * alpha) + 255 )>>8) +
                 ((((temp1>>24) * (255-alpha)) + 255)>>8)) << 24;
        temp |= ( (((((f->rgba<<8)>>24) * alpha) + 255 )>>8) +
                  (((((temp1<<8)>>24) * (255-alpha)) + 255)>>8)) << 16;
        temp |= ( (((((f->rgba<<16)>>24) * alpha) + 255 )>>8) +
                  (((((temp1<<16)>>24) * (255-alpha)) + 255)>>8)) << 8;

        fb_set_pixel_rawaddr(addr, temp, 1, 0);
      }
                                                                         1830
    if( z->fbmode & FBMODE_FRONT )
      {
        temp1 = fb_read_pixel_rawaddr(addr, 1);
        temp = ( ((((f->rgba>>24) * alpha) + 255 )>>8) +
                 ((((temp1>>24) * (255-alpha)) + 255)>>8)) << 24;
        temp |= ( (((((f->rgba<<8)>>24) * alpha) + 255 )>>8) +
                  (((((temp1<<8)>>24) * (255-alpha)) + 255)>>8)) << 16;
        temp |= ( (((((f->rgba<<16)>>24) * alpha) + 255 )>>8) +
                  (((((temp1<<16)>>24) * (255-alpha)) + 255)>>8)) << 8;
                                                                         1840
        fb_set_pixel_rawaddr(addr, temp, 0, 1);
      }


  }
else
  {
    // note, optimizing with knowlege of exactly what FBMODE_BACK and _FRONT
    // are... be careful!
    fb_set_pixel_rawaddr(addr, (f->rgba ^ alpha),                        1850
                         z->fbmode & FBMODE_BACK,
                         (z->fbmode & FBMODE_FRONT)>>1);
  }
```

```
  if(nowritez == 0)
    {
      if(nousez)
        zloc = &z->buf[addr];

      *zloc = f->z;                                                          1860
    }
}


void Stage3(PrimInfo * pi, UntexFragment * utf)
{
  Fragment fm;
  unsigned texrgba;
  unsigned dotex; // do texture mapping
  unsigned rgbaxlyl, rgbaxlyh, rgbaxhyl, rgbaxhyh; // 4 texture samples for bilinear    1870
  signed ut,vt, ut1,vt1;
  TexEntry *pTEntry;
  int i;

  unsigned texinterp, textile, texalpha, colalpha;
  unsigned texmode, lit, useamb, usedir;
  unsigned nousez, nowritez;

  // unpack bitfields for better performance
  texinterp = pi->p.Mode.texinterp;                                          1880
  textile = pi->p.Mode.textile;
  texalpha = pi->p.Mode.texalpha;
  colalpha = pi->p.Mode.colalpha;
  texmode = pi->p.Mode.texmode;
  lit = pi->p.Mode.lit;
  useamb = pi->p.Mode.useamb;
  usedir = pi->p.Mode.usedir;
  nousez = pi->p.Mode.nousez;
  nowritez = pi->p.Mode.nowritez;
                                                                             1890

  pTEntry = &ptm->pTexEntryTable[pi->TextureID];

  // see if we're actually doing texture mode
  dotex = 0;
  if (texmode >= 2 && pi->TextureID < ptm->MaxTextures)
    {
      if(pTEntry->valid == 1)
        dotex = 1;
    }                                                                        1900
```

```
  texrgba = 0;

#ifdef NOTEXCACHE

  invalidate_variable(pTEntry->pBegin,
                      pTEntry->Width * pTEntry->Height
                      * sizeof(unsigned));
#endif //NOTEXCACHE
                                                                        1910
  fm.x = utf->x;
  fm.y = utf->y;
  fm.z = utf->z;

  if(dotex)
    {
      magic_perf_texel();

      if(texinterp == 0)
        { // nearest neighbor                                           1920
          signed tempu, tempv;
          //map u,v into 0-1 range, based on pi->p.Mode.textile
          //(0=none,1=repeat, 2=mirror,3=clamp)

          texwrap(&utf->u, textile);
          texwrap(&utf->v, textile);

          //scale up to texel index
          utf->u *= (float)pTEntry->Width;
          utf->v *= (float)pTEntry->Height;                             1930

          // truncate u,v down to int.
          tempu = (signed)utf->u;
          tempv = (signed)utf->v;

          // do clamping
          if( textile == 3)
            {
              if(tempu < 0) tempu = 0;
              if(tempu >= pTEntry->Width) tempu = pTEntry->Width - 1;   1940
              if(tempv < 0) tempv = 0;
              if(tempv >= pTEntry->Width) tempv = pTEntry->Width - 1;
            }


          if(tempu >= 0 && tempv >= 0 &&
             tempu < pTEntry->Width && tempv < pTEntry->Height)
            texrgba = pTEntry->pBegin[tempu+pTEntry->Width*tempv];
          else
```

```
      texrgba = 0;                                                    1950


  }
else
  { // bilinear filtering

    //shift u,v by 0.5 texel, so "0" is centered at a texel
    utf->u -= 0.5/(float)pTEntry->Width;
    utf->v -= 0.5/(float)pTEntry->Height;
                                                                      1960
    //map u,v into 0-1 range, based on pi->p.Mode.textile
    //(0=none,1=repeat, 2=mirror,3=clamp)
    texwrap(&utf->u, textile);
    texwrap(&utf->v, textile);

    //scale up to texel index
    utf->u *= (float)pTEntry->Width;
    utf->v *= (float)pTEntry->Height;

    // do clamping                                                    1970
    if( textile == 3)
      {
        if(utf->u < 0.0f) utf->u = 0.0f;
        if(utf->u > (float)pTEntry->Width - 1.0f)
          utf->u = pTEntry->Width - 1;
        if(utf->v < 0.0f) utf->v = 0.0f;
        if(utf->v > (float)pTEntry->Height -  1.0f)
          utf->v = pTEntry->Height - 1;
      }
                                                                      1980

    //truncate u,v down to ut,vt to get lower,
    //and add one to get upper (for rgbaxl/hyl/h)
    ut = (signed) utf->u;
    vt = (signed) utf->v;
    ut1 = ut+1;
    vt1 = vt+1;

    //in case ut1 or vt1 wraps around
    if(ut1 >= pTEntry->Width)                                         1990
      {
        if(textile == 3) // clamp
          ut1 = ut;
        if(textile == 1) // repeat
          ut1 = 0;
        if(textile == 2) // mirror
          ut1 = (ut == 0) ? 0 : ut - 1;
```

```
  }
if(vt1 >= pTEntry->Height)
  {                                                                2000
    if(textile == 3) // clamp
      vt1 = vt;
    if(textile == 1) // repeat
      vt1 = 0;
    if(textile == 2) // mirror
      vt1 = (vt == 0) ? 0 : vt - 1;
  }


if(ut >= 0 && vt >= 0 &&                                           2010
   ut < pTEntry->Width && vt < pTEntry->Height)
  rgbaxlyl = pTEntry->pBegin[ut+pTEntry->Width*vt];
else
  rgbaxlyl = 0;

if(ut >= 0 && vt1 >= 0 &&
   ut < pTEntry->Width && vt1 < pTEntry->Height)
  rgbaxlyh = pTEntry->pBegin[ut+pTEntry->Width*vt1];
else
  rgbaxlyh = 0;                                                    2020

if(ut1 >= 0 && vt >= 0 &&
   ut1 < pTEntry->Width && vt < pTEntry->Height)
  rgbaxhyl = pTEntry->pBegin[ut1+pTEntry->Width*vt];
else
  rgbaxhyl = 0;

if(ut1 >= 0 && vt1 >= 0 &&
   ut1 < pTEntry->Width && vt1 < pTEntry->Height)
  rgbaxhyh = pTEntry->pBegin[ut1+pTEntry->Width*vt1];              2030
else
  rgbaxhyh = 0;

// blend between four corners
texrgba = (((unsigned)((rgbaxlyl&0xFF)*(1.0f-(utf->u-(float)ut)) +
                       (rgbaxhyl&0xFF)*(utf->u-(float)ut)))&0xFF)
  |((((unsigned)(((rgbaxlyl>>8)&0xFF)*(1.0f-(utf->u-(float)ut)) +
                 ((rgbaxhyl>>8)&0xFF)*(utf->u-(float)ut)))&0xFF)<<8)
  |((((unsigned)(((rgbaxlyl>>16)&0xFF)*(1.0f-(utf->u-(float)ut)) +
                 ((rgbaxhyl>>16)&0xFF)*(utf->u-(float)ut)))&0xFF)<<16)    2040
  |(((unsigned)((rgbaxlyl>>24)*(1.0f-(utf->u-(float)ut)) +
                (rgbaxhyl>>24)*(utf->u-(float)ut)))<<24);

texrgba = (((unsigned)((texrgba&0xFF) * (1.0f-(utf->v-(float)vt)) +
                       ((rgbaxlyh&0xFF)*(1.0f-(utf->u-(float)ut)) +
```

```
                        (rgbaxhyh&0xFF)*(utf->u-(float)ut)) *
                       (utf->v-(float)vt)))&0xFF)
    | ((((unsigned)(((texrgba>>8)&0xFF) * (1.0f-(utf->v-(float)vt)) +
                    (((rgbaxlyh>>8)&0xFF)*(1.0f-(utf->u-(float)ut)) +
                     ((rgbaxhyh>>8)&0xFF)*(utf->u-(float)ut)) *                    2050
                    (utf->v-(float)vt)))&0xFF)<<8)
    | ((((unsigned)(((texrgba>>16)&0xFF) * (1.0f-(utf->v-(float)vt)) +
                    (((rgbaxlyh>>16)&0xFF)*(1.0f-(utf->u-(float)ut)) +
                     ((rgbaxhyh>>16)&0xFF)*(utf->u-(float)ut)) *
                    (utf->v-(float)vt)))&0xFF)<<16)
    | (((unsigned)((texrgba>>24) * (1.0f-(utf->v-(float)vt)) +
                   ((rgbaxlyh>>24)*(1.0f-(utf->u-(float)ut)) +
                    (rgbaxhyh>>24)*(utf->u-(float)ut)) *
                   (utf->v-(float)vt)))<<24);
                                                                                   2060
    }
  }


// truncate alphas in texrgba and utf->rgba, according to pi->p.Mode.texalpha and colalpha
//   and pi->alphaThresh

if(texalpha == 0)
  texrgba |= 0x0FF;
else if(texalpha == 2)                                                             2070
  { // hard alpha
    if( (texrgba & 0x0FF) >= pi->alphaThresh )
      texrgba |= 0x0FF;
    else
      texrgba &= 0xFFFFFF00;
  }

if(colalpha == 0)
  utf->rgba |= 0x0FF;
else if(colalpha == 2)                                                             2080
  { // hard alpha
    if( (utf->rgba & 0x0FF) >= pi->alphaThresh )
      utf->rgba |= 0x0FF;
    else
      utf->rgba  &= 0xFFFFFF00;
  }


// sort of a hack - if we're in blend mode, and
// either col or tex is hard alpha, and is 0x00 (under                            2090
// the threshold), then make the final alpha 0x00
// this is so we can have a hard-alpha texture be
// blended with a color map without creating a soft
```

```
// alpha result. Note that such a prim will be out-of-order
if(texmode == 3)
  {
    if(texalpha == 2 && (texrgba&0xFF) == 0)
      utf->rgba &= 0xFFFFFF00;
    if(colalpha == 2 && (utf->rgba&0xFF) == 0)
      texrgba &= 0xFFFFFF00;                                          2100
  }

// now blend texrgba and utf->rgba, according to pi->p.Mode.texmode
//  (0=none, 1=color, 2=tex, 3=blend, 4=texdecal, 5=coldecal, 6=modulate)
//  and pi->ColTexBalance (0 = all tex, 1 = all color) -> utf->rgba

switch(texmode)
  {
    unsigned alpha;
  case 1: // color only                                               2110
    fm.rgba = utf->rgba;
    break;
  case 2: // texture only
    fm.rgba = texrgba;
    break;
  case 3: // col/tex blend
  case 4: // tex on top of color decal
  case 5: // color on top of tex decal
    switch(texmode)
      {                                                               2120
      case 3: // col/tex blend
        alpha = pi->ColTexBalance * 256;
        if(alpha >= 256) alpha = 255;
        break;
      case 4: // tex on top of color decal
        alpha = 255 - (texrgba & 0xFF);
        break;
      case 5: // color on top of tex decal
        alpha = (utf->rgba & 0xFF);
        break;                                                        2130
      default: // this shouldn't happen
      }


    // fixed point modulation
    fm.rgba = ( ((((utf->rgba>>24) * alpha) + 255 )>>8) +
              ((((texrgba>>24) * (255-alpha)) + 255)>>8)) << 24;
    fm.rgba |= ( (((((utf->rgba<<8)>>24) * alpha) + 255 )>>8) +
              (((((texrgba<<8)>>24) * (255-alpha)) + 255)>>8)) << 16;
    fm.rgba |= ( (((((utf->rgba<<16)>>24) * alpha) + 255 )>>8) +      2140
              (((((texrgba<<16)>>24) * (255-alpha)) + 255)>>8)) << 8;
```

```
    fm.rgba |= ( (((((utf->rgba<<24)>>24) * alpha) + 255 )>>8) +
                 (((((texrgba<<24)>>24) * (255-alpha)) + 255)>>8));
    break;
  case 6: // color/tex modulated
    // max of each is 255 - treated as 1.0 (fixed point modulation)
    fm.rgba = ((((utf->rgba>>24) * (texrgba>>24)) + 255 )>>8) << 24;
    fm.rgba |= (((((utf->rgba<<8)>>24)
                 * ((texrgba<<8)>>24)) + 255 )>>8) << 16;
    fm.rgba |= (((((utf->rgba<<16)>>24)                                    2150
                 * ((texrgba<<16)>>24)) + 255 )>>8) << 8;
    fm.rgba |= ((((utf->rgba<<24)>>24) * ((texrgba<<24)>>24)) + 255 )>>8;
    break;
  case 0:
  default:
    fm.rgba = 0x000000FF;
  }

// finally, modulate output with light values, based on
//   pi->p.Mode.lit, useamb, usedir, pi->ambColor,                         2160
//   pi->dirColor, utf->intensity into fm.rgba.
if(lit)
  {
    unsigned temp, ambrgb=0, dirrgb=0, intens, tempr, tempg, tempb;


    // calculate ambient component - light values times intensity (i field)
    //                                  modulated with surface color
    if(useamb)
      { // more fixed point modulation. fun!                               2170
        intens = (pi->ambColor & 0xFF);
        temp = ((((pi->ambColor>>24) * intens) + 255 )>>8) << 24;
        temp |= (((((pi->ambColor<<8)>>24) * intens) + 255 )>>8) << 16;
        temp |= (((((pi->ambColor<<16)>>24) * intens) + 255 )>>8) << 8;
        ambrgb |= ((((temp>>24) * (fm.rgba>>24)) + 255 )>>8) << 24;
        ambrgb |= (((((temp<<8)>>24)
                   * ((fm.rgba<<8)>>24)) + 255 )>>8) << 16;
        ambrgb |= (((((temp<<16)>>24)
                   * ((fm.rgba<<16)>>24)) + 255 )>>8) << 8;
      }                                                                    2180
    // calculate directional component - light values times intensity
    //                                  modulated with surface color

    if(usedir)
      {
        intens = utf->intensity*256;
        if(intens > 255) intens = 255;

        temp = ((((pi->dirColor>>24) * intens) + 255 )>>8) << 24;
```

```
        temp |= (((((pi->dirColor<<8)>>24) * intens) + 255 )>>8) << 16;         2190
        temp |= (((((pi->dirColor<<16)>>24) * intens) + 255 )>>8) << 8;
        dirrgb |= ((((temp>>24) * (fm.rgba>>24)) + 255 )>>8) << 24;
        dirrgb |= (((((temp<<8)>>24)
                  * ((fm.rgba<<8)>>24)) + 255 )>>8) << 16;
        dirrgb |= (((((temp<<16)>>24)
                  * ((fm.rgba<<16)>>24)) + 255 )>>8) << 8;
      }

    // saturate-add the two together.
    // only affects color values, alpha stays the same!                     2200

    tempr = (ambrgb>>24) + (dirrgb>>24);
    if(tempr > 255) tempr = 255;
    tempg = ((ambrgb<<8)>>24) + ((dirrgb<<8)>>24);
    if(tempg > 255) tempg = 255;
    tempb = ((ambrgb<<16)>>24) + ((dirrgb<<16)>>24);
    if(tempb > 255) tempb = 255;

    fm.rgba &= 0x0FF;
    fm.rgba |= tempr << 24 | tempg << 16 | tempb << 8;                      2210
  }

  magic_perf_texfragment();

  if(fm.rgba & 0x0FF)
    {
    doFragment(&fm, pzbd, nousez, nowritez);
    }
}
                                                                            2220
```

Appendix C

Full Implementation Code Listing

C.1 Common-sw.h

```
// include file for Common-sw.S
// Ken Taylor 2004 Master's Thesis

#ifndef COMMON_SW_H
#define COMMON_SW_H

#define invalidate_word(val) __rgcc_one_input("ainv %0, 0", val)
#define flush_word(val) __rgcc_one_input("afl %0, 0", val)

                                                                                10
// magic instructions for performance measurements.
#define magic_perf_startbusywait()    ASM_VOLATILE_3(magc $0,$0, 0xfed0)
#define magic_perf_endbusywait()      ASM_VOLATILE_3(magc $0,$0, 0xfed1)
#define magic_perf_fragment()         ASM_VOLATILE_3(magc $0,$0, 0xfed2)
#define magic_perf_texfragment()      ASM_VOLATILE_3(magc $0,$0, 0xfed3)
#define magic_perf_texel()            ASM_VOLATILE_3(magc $0,$0, 0xfed4)
#define magic_perf_drawnprim()        ASM_VOLATILE_3(magc $0,$0, 0xfed5)
#define magic_perf_startintbusywait() ASM_VOLATILE_3(magc $0,$0, 0xfed6)
#define magic_perf_endintbusywait()   ASM_VOLATILE_3(magc $0,$0, 0xfed7)
                                                                                20

void flush_variable(void *, unsigned);
void invalidate_variable(void *, unsigned);


#endif //COMMON_SW_H
```

C.2 Common-sw.S

```
// flush_variable and invalidate_variable defined here

        .text
        .align 2

        .global flush_variable
        .ent    flush_variable
        # $4 = address to startflush, $5 = bytes to flush (4 bytes per word)
flush_variable:
        addu    $8, $0, $4      # copy start to temp reg                        10
        addu    $9, $8, $5      # upper limit = start + # bytes
fv_loop:
        beq-    $8, $9, fv_done
        afl     $8, 0           # flush
        addiu   $8, $8, 4
        j       fv_loop
fv_done:
        ainv    $8, 0
        lw      $0, 0($8)       #initiate a load of memory to make sure
                                #flush is 100% complete before continuing        20
        jr $31
        .end flush_variable


        .global invalidate_variable
        .ent    invalidate_variable
        # $4 = address to startflush, $5 = bytes to flush (4 bytes per word)
invalidate_variable:
        addu    $8, $0, $4      # copy start to temp reg
        addu    $9, $8, $5      # upper limit = start + # bytes                  30
iv_loop:
        beq-    $8, $9, iv_done
        ainv    $8, 0           # invalidate
        addiu   $8, $8, 4
        j       iv_loop
iv_done:
        jr $31
        .end invalidate_variable
```

C.3 render_datatypes.h

```
// render_datatypes.h
//
// Ken Taylor, MIT Master's Thesis 2004
//
// Last Updated: 5/18/2004
//
// This file defines data types that are shared between different
// stages of the pipeline, such as texture memory maps, primitive blocks,
// untextured fragment blocks, and textured fragment blocks.
//                                                                              10
// Data types that are only used for one stage are defined in the
// respecitve stage headers.

#ifndef RENDER_DATATYPES_H
#define RENDER_DATATYPES_H


// unit used for linked list of texture memory allocation
// space. See TexManager description for detail.
typedef struct _TexAllocation {                                                 20

  unsigned ID; // Texture ID, index into texEntry table.

  unsigned *pBegin; // pointer to beginning in texture memory
  unsigned *pEnd;   // pointer to one word beyond end in texture memory
  // pEnd should equal pBegin + (Width*Height*4) in bytes (not in pointer arith! in
  // pointer arith there'd be no 4!) - so is it necessary? - maybe not, but it's useful.

  struct _TexAllocation *pNext; // next item in list
  struct _TexAllocation *pPrev; // previous item in list                        30

} TexAllocation;


// unit used for quick-lookup array of textures. Adding a texture
// is slower though, as it requires a linear search through this array
// for an empty spot. See TexManager for more.
typedef struct _TexEntry {
  unsigned valid:1; // is there an entry here?
  unsigned updated; // has it been changed since the last time this proc        40
  // accessed it? bit field... one per column
  unsigned Width; // Texture width
  unsigned Height; // Texture height
  // texture size in memory is implicitly Width*Height*4 bytes
```

```
  unsigned *pBegin; // pointer to beginning in texture memory
  TexAllocation *pAlloc; // pointer to tex allocation data.
} TexEntry;

// tex entries are in stage3's memory, while tex allocation data is          50
// in stage0's memory... so i put width, height, and an extra pbegin in
// tex entry for quicker lookup.


// texManager is a texture memory manager structure shared by
// Stage1 and Stage3. It consists of a pointer to texture memory,
// and some basic info on the total room and room left in texture memory,
// a doubly linked list of allocated textures, in order they appear in memory,
// for linear search of space for a new texture, and a quick index
// that maps texture ids to allocation blocks, for quick texture lookup.     60
typedef struct _TexManager {

  unsigned *pTexMemory;   // a pointer to texture memory.
  // textures are word-aligned and stored as rgba.

  unsigned TexMemorySize; // total size of texture memory
  unsigned TexMemoryFree; // total free space in texture memory
  unsigned MaxTextures;   // maximum number of textures that can be listed
  unsigned NumTextures;   // current number of textures being listed
                                                                             70
  TexAllocation * pAllocHead; // head of texture allocation list (NULL if none)
  TexAllocation * pAllocTail; // tail of texture allocation list (NULL if none)

  TexEntry * pTexEntryTable; // table of texture entries for quick lookup
  // (texture ID is an index into the tex entry array.)

} TexManager;


// vertex used in transformed primitives                                     80
typedef struct _Vertex {

  // x and y are normalized between -1 and 1.
  float x;
  float y;
  float z;

  // w1 is one over the normalization factor w, and is used for
  //
  float w1;                                                                  90

  // texture coordinates (divided by old w after perspective divide)
  float u;
```

```
  float v;

  // rgba are floats internally, for correct color interpolation
  // (they're all divided by old w), although on input they're
  // packed into one 32-bit color (rgba).
  float r;
  float g;                                                              100
  float b;
  float a;

  // directional light intensity should also be interpolated correctly
  // for gourad shading, and this will be intensity/(old w). But for
  // flat shading, the average is used, and this is just intensity with
  // no extra division. Intensity is scaled from 0 to 1.

  float intensity;
                                                                        110
} Vertex;


// ModeBits are the primitive-level rendermode descriptors.
// they're stored in a separate structure as a bitmask to make
// things a bit more convenient.

typedef struct _ModeBits {

  unsigned draw:1; // whether to actually draw it                      120

  unsigned lit:1; // 0 means fullbright. 1 means final color depends on lighting.
  unsigned useamb:1; // whether or not to use ambient light
  unsigned usedir:1; // whether or not to use directional light

  unsigned texmode:3;
  // 0 = neither texture nor color (renders black)
  // 1 = no texture (color only)
  // 2 = no color (texture only)
  // 3 = tex/color blend (see ColTexBalance)                            130
  // 4 = decal, texture on top
  // 5 = decal, color on top  (in decal mode, the "bottom" can be seen through the
  //                           "top"'s alpha)
  // 6 = tex/color modulated (useful to make textures translucent)

  unsigned texalpha:2;
  // 0 = no alpha
  // 1 = soft alpha
  // 2 = hard alpha (cut between 0 and 100% at alphaThresh point)
                                                                        140
  unsigned colalpha:2;
```

```
    // 0 = none
    // 1 = soft
    // 2 = hard

    unsigned colinterp:1;
    // 0 = average vertices
    // 1 = smooth interpolation (perspective correct)

    unsigned litinterp:1;
    // 0 = average vertices (flat shading)
    // 1 = smooth interpolation (perspective correct gourad shading)

    unsigned texinterp:1;
    // 0 = nearest-neighbor
    // 1 = bilinear filtering.
    // note the implementation of texinterp isn't quite parallel to col and lit interp.
    // col and lit interp interpret the vertex parameters. texinterp interpolates between
    // pixels in a texture map - texture mapping *always* does perspective correct
    // interpolation of parameters.

    unsigned outoforder:1;
    // for the RenderState, they hold the user's input, as such:
    // 0 = default (in order for any soft alpha/translucency, or if
    //              either nousez or nowritez is on. out of order otherwise)
    // 1 = always out of order (optimization, basically)
    // but when being passed down the pipeline, they hold the heuristic result
    // of whether to treat it as in-order or out-of-order, as such:
    // 0 = always in-order
    // 1 = always out-of-order
    // see Stage1_datatypes.h

    unsigned textile:2;
    // 0 = no tiling (rest will be treated as black, with zero alpha)
    // 1 = repeat
    // 2 = repeat mirrored
    // 3 = clamp (outer pixel values repeated forever)

    unsigned nousez:1;
    // 0 = default (checks z buffer before updating)
    // 1 = don't check the z buffer, always update

    unsigned nowritez:1;
    // 0 = default (writes to z buffer when updating, unless alpha = 0)
    // 1 = don't write to the z buffer when updating.

} ModeBits;
```

```
// PrimaryPrimInfo is data that is needed by stage 4, which is less        190
// than that needed by earlier stages.
typedef struct _PrimaryPrimInfo {

  ModeBits Mode;

  // in RenderState, this is the next sequence number to use
  //   (see Stage1_datatypes.h)
  // when being passed down the pipe, this is the sequence number
  //   of the current primitive
  unsigned SeqNum;                                                         200

} PrimaryPrimInfo;


// PrimInfo holds data that is global to a particular
// primitive. This data should be sent once per primitive
// in the otherwise fragment-based streams between
// Stage2 and 3, and Stage3 and 4. Number of fragments
// in the prim will be sent separately, as that's more of
// a flow control value than an inherent part of the primitive.            210
typedef struct _PrimInfo {

  // data needed by last stage stored in PrimaryPrimInfo. other data
  // needed by previous stages stored in this struct (PrimInfo)
  PrimaryPrimInfo p;

  unsigned TextureID;

  // balance between color and texture for color-texture blend mode
  float ColTexBalance; //(0 = all tex, 1 = all color)                      220

  // alpha threshold for "hard alpha" modes
  unsigned alphaThresh;

  // the ambient light intensity here is the modulation of the
  // actual light intensity, and the primitive's light reflection factor.
  // to modulate two numbers on a zero-to-255 scale, a and b:
  // ((a * b) + 255 ) >> 8
  // - intermediates need 16 bits!
                                                                           230
  // ambient light color and intensity rgbi
  unsigned ambColor;

  // directional light color rgb (no intensity sent between stages at the
  // prim level, this is calculated and sent per vertex in first stage
  // from light intensity, primitive light reflection, and the dot product
  // of the normal and the light direction. After stage 2, directional light
```

```
  // intensities are interpolated and sent in the fragments.)
  unsigned dirColor;
                                                                        240
  // lighting modulation is finally applied to fragment color data either after
  // texture lookup in stage 3 or during recombining in stage 4 (before background
  // blending. Probably stage 3. (though putting off all combining to stage 4
  // could take advantage of visibility culling...) decided: put it in stage 3,
  // since stage 4 needs to get as much pixel throughput as possible

  // NOTE: RenderState in Stage1-datatypes.h uses this data structure, but
  //        the light values mean different things. *both* amb and dir store
  //        the intensity field, but it's the basic light intensity, and not the
  //        modulated-with-prim-reflection intensity.                  250

} PrimInfo;


// TransPrim is a transformed primitive (gone through projective
// transform and perspective division) sent from Stage1 to Stage2.
// Vertex coords are in normalized screen coordinates (-1 to 1).
typedef struct _TransPrim {

  PrimInfo pInfo;                                                       260

  Vertex v[3];

} TransPrim;


// UntexFragment is a fragment before texture mapping/blending
// has been done. As such, it still stores interpolated texture
// coordinates and light intensity. It's passed from Stage2 to Stage3.
typedef struct _UntexFragment {                                         270

  // x and y are clipped and in screen coordinates now.
  // (so Stage2 needs to know about the screen dimension)
  // top left is 0,0, increasing positively down and right
  unsigned x;
  unsigned y;

  // z is interpolated - although nonlinear, it's monotonic
  // (ie, this isn't the *real* z for the point, but it works for
  //  a z-buffer)                                                       280
  // fixed point representation still.
  signed z;

  // correctly interpolated texture coords, still normalized
  // to texture size (0 to 1. Greater or less implies
```

```
  // wrapping)
  float u;
  float v;

  // correctly interpolated color and alpha, packed together,                290
  // on a zero-to-255 scale for each field (0% to 100%). This
  // can result in gradient artifacts, but I'm not planning on
  // having too many passes over the data, so storing each
  // field as 8 bit now should suffice.
  unsigned rgba;

  // correctly interpolated directional light intensity, still a float.
  float intensity;

} UntexFragment;                                                             300


// Fragment is a fragment ready for blending with the
// framebuffer, passed from Stage3 to Stage4.
typedef struct _Fragment {

  // screen coordinates still
  unsigned x;
  unsigned y;
                                                                             310
  // fixed point still
  signed z;

  // final color and alpha from combining texture and color
  // based on mode bits. 0-to-255 scale still. lighting
  // information is blended by stage3 for now (todo: this might change)
  unsigned rgba;

} Fragment;
                                                                             320
// tag words sent down pipe
#define RENDER_P_PRIM     0
#define RENDER_P_FLUSH    1
#define RENDER_P_FRAG     0
#define RENDER_P_ENDPRIM 1

#endif //RENDER_DATATYPES_H
```

C.4 Stage1-datatypes.h

```
// Stage1_datatypes.h
//
//  Ken Taylor, MIT Master's Thesis 2004
//
// Last Updated: 5/18/2004
//
// This file defines data types that are used by the first
// stage of the pipeline (transform & lighting, aka geometry).
//
// For datatypes that are shared between stages, see render_datatypes.h                10

#ifndef STAGE1_DATATYPES_H
#define STAGE1_DATATYPES_H

#include "render_datatypes.h"

// InputVertex holds vertex information as
// it's being inputted from the user
// these are all in model coordinates. The ModelToView matrix
// projects them to the current view (this is the combination                          20
// of the ModelToWorld transform and the WorldToView matrix).
// Normals can be turned into world coordinates (for lighting
// calculation) by the NormalToWorld transform (which is the
// ModelToWorld transform inverted and transposed).
// See "RenderState" for all these matrices.
typedef struct _InputVertex {

  // coordinates
  float x;
  float y;                                                                              30
  float z;

  // normal
  float nx;
  float ny;
  float nz;

  // tex coords
  float u;
  float v;                                                                              40

  // color+alpha
  unsigned rgba;

} InputVertex;
```

```
// InputPrim stores vertices of a primitive
// as it's being inputted from the user
//  things like ModeBits, TextureID, and lighting info,                50
//  which are carried over between prims, are stored
//  in RenderState
typedef struct _InputPrim {

  InputVertex v[3];
  // vertices after the 3rd are ignored.

} InputPrim;

                                                                       60
// TransformMatrix holds 4x4 matrices used for
// transformations. elements are m[row][column]
typedef float TransformMatrix[4][4];


// RenderState stores information on the current
// rendering state, from transform matrices to current render and blending
// modes to current texture, color, and normal. Basically, any information
// that can be common between primitives.
typedef struct _RenderState {                                          70

  // just one bit of renderstate_updated information isn't enough,
  // because all m parallel procs need to see it before it can be
  // cleared (m = 4 in my case). So a bit vector of m bits is used.
  // when a proc updates the renderstate, it sets them all to 1. Then
  // when each proc sees a 1 in its spot, it loads the new state and
  // clears just that spot.

  unsigned Updated;
                                                                       80
  // the user specifies these two matrices.

  TransformMatrix ModelToWorld;
  TransformMatrix WorldToView;

  // these matrices are pre-calculated whenever the user
  // changes one of the two above.

  TransformMatrix ModelToView; // WorldToView*ModelToWorld
  TransformMatrix NormalToWorld; // (ModelToWorld^-1)^T                 90

  // per-vertex data that can be carried between vertices
  // normal
```

```
float nx;
float ny;
float nz;

// color+alpha
unsigned rgba;
                                                                                100
// PrimInfo is from render_datatypes.h
// PrimInfo contains per-primitive data that can be carried
//  between primitives: p.Mode, TextureID, ColTexBalance,
//  p.SeqNum, and alphaThresh
// And also Global lighting params: ambColor, and dirColor
//  these are used differently here than when PrimInfo is passed
//  between stage 1 and stage 2 - ambColor and dirColor
//  both use the intensity field, and it's the base light intensity
//  and not the modulated light+reflectivity intensity.
// p.Mode.outoforder and p.SeqNum have different meanings when            110
// stored here in RenderInfo versus being passed down the pipe.
// see comments in render_datatypes.h and below for info.

PrimInfo pInfo;

// other global lighting params:

// light direction, in world coordinates
float ldx;
float ldy;                                                                      120
float ldz;

// ambient light reflectivity, from 0 to 255
unsigned ambreflect;

// directional light reflectivity, from 0 to 255
unsigned dirreflect;

// amb light is defaulted to 100% fullbright if not changed
                                                                                130
// is there even a directional light?
unsigned dirdefined:1;

// there's an optimization where instead of transforming every normal
// for light calculations, the light is transformed into model space.
// however, this doesn't work for Nonorthogonal transforms, such as
// shearing and nonisotropic scaling.

// sequence numbers for in-order rendering (soft alpha prims)
//  notice the seq # algorithm requires that whether a prim is              140
//  in-order or out-of-order to be known apriori. For color alpha,
```

```
// we can do a rough heuristic on the vertex values to determine
// whether a "soft" rendering poly will have any actual soft values.
// but if it's possible for a texture to create "soft" alpha holes,
// we have to assume that the poly will be soft, and treat it as in-order.

// so, whether a poly should be treated as in-order or not is a complex
//  function of its vertex colors, texmode, texalpha, colalpha, outoforder,
//  nousez and nowritez. But this is worthwile to calculate, as
//  putting things out of order helps gain a lot more parallelism.          150

// anyway, there's the current sequence number given to in-order prims.
// SeqNum now defined in PrimInfo.p. See render_datatypes.h

// and the "lagged" sequence number given to out-of-order prims. This
// number is n+1, where "n" is the last SeqNum given
// to an out-of-order prim. Rollover is dealt with by flushing the
// pipeline before sending the prim with SeqNum 0 down. With a large
// enough number space for sequence numbers, this shouldn't be
// too much of a performance hit.                                          160

unsigned LaggedSeqNum;

// In the final stage, an in-order prim will only go if the rendering
// sequence number down there (different than our SeqNum) is equal to
// the prim's SeqNum, and will increment the rendering sequence
// number when done. (in-order prims also go a whole prim at a time).
// However, an out-of-order prim will go whenever its SeqNum
// is less than or equal to the rendering sequence number. Out-of-
// order prims can be split up into any size groups of fragments, as        170
// long as they only update the sequence number when they've rendered
// the last of their fragments.

} RenderState;

#endif //STAGE1_DATATYPES_H
```

C.5 ZBuf_datatypes.h

```
// ZBuf_datatypes.h
//
//  Ken Taylor, MIT Master's Thesis 2004
//
// Last Updated: 5/18/2004
//
// this file defines inter-stage shared variables between stage1
// and stage4, mostly for z-buffering.

#ifndef ZBUF_DATATYPES_H                                              10
#define ZBUF_DATATYPES_H

#define FBMODE_NONE 0
#define FBMODE_BACK 0x01
#define FBMODE_FRONT 0x02
#define FBMODE_BOTH 0x03

typedef struct _ZBufData {

  unsigned fbmode;                                                    20
  // fbmode: 00 - don't render anywhere
  //         01 - render to back buffer
  //         10 - render to front buffer
  //         11 - render to both

  signed buf[VWIDTH*VHEIGHT];

} ZBufData;

                                                                      30
#endif //ZBUF_DATATYPES_H
```

C.6 render_cmds.h

```
// render_cmds.h
//
//  Ken Taylor, MIT Master's Thesis 2004
//
// Last Updated: 5/18/2004
//
// This file holds defines for all the render opcodes.

#ifndef RENDER_CMDS_H
#define RENDER_CMDS_H                                         10

// command mode only
// no params
#define RENDER_BEGINSCENE        1

// scenestream only
// no params. . wait for reply from renderer before continuing
#define RENDER_ENDSCENE          2

// scenestream only                                           20
// params are float  x,y ,z,u,v
#define RENDER_VERTEX            3

// param is unsigned rgba
#define RENDER_COLOR             4

// param is 16 floats, one row at a time
#define RENDER_MODELMATRIX       5

// param is 16 floats, one row at a time                      30
#define RENDER_VIEWMATRIX        6

// param is 3 floats: x,y,z
#define RENDER_NORMAL            7

// param for these is unsigned:
#define RENDER_SET_LIT           8
#define RENDER_SET_USEAMB        9
#define RENDER_SET_USEDIR        10
#define RENDER_SET_TEXMODE       11                           40
#define RENDER_SET_TEXALPHA      12
#define RENDER_SET_COLALPHA      13
#define RENDER_SET_COLINTERP     14
#define RENDER_SET_LITINTERP     15
#define RENDER_SET_TEXINTERP     16
```

```
#define RENDER_SET_OUTOFORDER    17
#define RENDER_SET_TEXTILE       18
#define RENDER_SET_NOUSEZ        19
#define RENDER_SET_NOWRITEZ      20
#define RENDER_SET_TEXTUREID     21                                          50

// param is a float
#define RENDER_COLTEXBALANCE     22

// param is unsigned
#define RENDER_ALPHATHRESH       23

// param is unsigned rgbi
#define RENDER_AMBCOLOR          24
                                                                             60
// param is unsigned rgbi
#define RENDER_DIRCOLOR          25

// param is 3 floats, x,y,z
#define RENDER_DIRLIGHT          26

// param is unsigned
#define RENDER_AMBREFLECT        27

// param is unsigned                                                         70
#define RENDER_DIRREFLECT        28

// command mode only
// takes unsigned param - which page (or both)
//        and rgbx param for color
#define RENDER_CLEARFB           29

// command mode only
// no params
#define RENDER_CLEARZ            30                                          80

// command mode only
// param is unsigned
#define RENDER_SETPAGE           31

// command mode only
// param is unsigned (wait for vsync)
#define RENDER_FLIPPAGE          32

// command mode only                                                         90
// params are sizex, sizey
// returns either a token or -1 (0xFFFFFFFF) if no space
#define RENDER_ALLOCATE_TEXTURE  33
```

```
// command mode only
// param is unsigned token
#define RENDER_DEALLOC_TEXTURE   34

// command mode only
// param is token, then length of data in words, then data for          100
//  texture
#define RENDER_UPLOAD_TEXTURE    35

// command mode only
// no params, returns unsigned total size available
#define RENDER_TEXMEM_AVAIL      36

// command mode only
// no params
#define RENDER_COMPACT_TEXMEM    37                                     110

// command mode only
// params are x,y,page(s),rgbx
#define RENDER_WRITE_FB          38

// command mode only
// params are starting x,y, page(s), length, then rgbx's
#define RENDER_WRITE_FB_BLOCK    39

// command mode only                                                    120
// params are x,y,page
// returns unsigned rgbx
#define RENDER_READ_FB           40

// command mode only
// params are starting x,y, page, length
// returns many unsigned rgbx according to length
#define RENDER_READ_FB_BLOCK     41

// command mode only                                                    130
// params are x,y,signed value
#define RENDER_WRITE_Z           42

// command mode only
// params are starting x,y, length, then signed z's
#define RENDER_WRITE_Z_BLOCK     43

// command mode only
// params are x,y
// returns signed z                                                     140
#define RENDER_READ_Z            44
```

```
// command mode only
// params are starting x,y, length
// returns many signed z's according to length
#define RENDER_READ_Z_BLOCK      45

// command mode only
// no params, returns nothing
#define RENDER_RESET             46                                150

// command mode only
// no params, returns nothing
#define RENDER_HALT              47


#endif RENDER_CMDS_H
                                                                   160
```

C.7 Stage1-Main.c

```
// Stage1-Main.c
//
//  Ken Taylor, MIT Master's Thesis 2004
//
// Last Updated: 5/18/2004
//
// This file implements the startup and control code for tile 0.
// When tile 0 goes into scenestream mode, it branches into code
// which runs under "Stage1-Common.c". Assembly helper functions
// for this code are in Stage1-Main-sw.S, while assembly functions
// for scenestream and that are otherwise common among the tiles
// are in Stage1-sw.S.


#include "module_test.h" // includes raw.h
#include "raw_compiler_defs.h" // for PASS (testing)
#include "Stage1-datatypes.h" // shared datatypes, includes render_datatypes.h
#include "ZBuf_datatypes.h" // for z-buffer/stage-4 interaction
#include "render_framebuffer.h"
#include "render_cmds.h" // command defines
#include "Common-sw.h" // flush/invalidate

// Start up the switch (code in assembly)
void setup_switch_main(void);
void setup_interrupts(void);
void setup_switch_scenestream(void);

#define gdn_send_hdr(F, l, u, oY, oX, dY, dX) \
  gdn_send(F<<29|l<<24|u<<20|oY<<15|oX<<10|dY<<5|dX)

// functions from Stage1-Common.c that are used
void MatrixMatrixMult(TransformMatrix X, TransformMatrix M,
                      TransformMatrix Y);
void MatrixInvTrans(TransformMatrix X, TransformMatrix Y);


RenderState *prs;
TexManager *ptm;
ZBufData *pzbd;

// clears the renderstate (prs) to its initial values
void ClearRenderState()
{
  int i,j;
```

```
  prs->Updated = 0xE;
  for(i = 0; i < 4; i++)  // init all matrices to the identity matrix
    for(j = 0; j < 4; j++)
      {
        prs->ModelToWorld[i][j] = prs->WorldToView[i][j] =              50
          prs->ModelToView[i][j] = prs->NormalToWorld[i][j] =
          (i == j) ? 1.0f : 0.0f;
      }

  prs->nx = 1.0f;
  prs->ny = 0.0f;
  prs->nz = 0.0f;
  prs->rgba = 0x000000FF;
  prs->pInfo.p.Mode.lit = 0;
  prs->pInfo.p.Mode.useamb = 0;                                         60
  prs->pInfo.p.Mode.usedir = 0;
  prs->pInfo.p.Mode.texmode = 1;
  prs->pInfo.p.Mode.texalpha = 1;
  prs->pInfo.p.Mode.colalpha = 1;
  prs->pInfo.p.Mode.colinterp = 1;
  prs->pInfo.p.Mode.litinterp = 0;
  prs->pInfo.p.Mode.texinterp = 0;
  prs->pInfo.p.Mode.outoforder = 0;
  prs->pInfo.p.Mode.textile = 1;
  prs->pInfo.p.Mode.nousez = 0;                                         70
  prs->pInfo.p.Mode.nowritez = 0;
  prs->pInfo.TextureID = 0;
  prs->pInfo.ColTexBalance = 0.5f;
  prs->pInfo.alphaThresh = 128;
  prs->pInfo.ambColor = 0;
  prs->pInfo.dirColor = 0;
  prs->ldx = 1.0f;
  prs->ldy = 0.0f;
  prs->ldz = 0.0f;
  prs->ambreflect = 255;                                                80
  prs->dirreflect = 255;
  prs->dirdefined = 0;
}



// takes cmd, and executes it.
// returns true if going into scenestream, false if not.
// potentially uses prs, ptm, pzbd.
unsigned doCommand(unsigned cmd)                                        90
{
  int i,j;
```

```
switch(cmd)
  {
  case RENDER_BEGINSCENE:
    static_send(0);
    static_send(0);
    return 1;
    break;                                                              100

  case RENDER_COLOR:
    static_send(1);
    prs->rgba = static_receive();
    static_send(0);
    prs->Updated = 0x0E;
    break;

  case RENDER_MODELMATRIX:
    static_send(16);                                                    110
    for(i = 0; i < 4; i++)
      for(j = 0; j < 4; j++)
        {
          prs->ModelToWorld[i][j] = static_receive_f();
        }

    static_send(0);

    MatrixMatrixMult(prs->ModelToView, prs->WorldToView,
                     prs->ModelToWorld);                                120

    MatrixInvTrans(prs->NormalToWorld, prs->ModelToWorld);

    prs->Updated = 0x0E;

    break;

  case RENDER_VIEWMATRIX:
    static_send(16);
    for(i = 0; i < 4; i++)                                              130
      for(j = 0; j < 4; j++)
        {
          prs->WorldToView[i][j] = static_receive_f();
        }

    static_send(0);

    MatrixMatrixMult(prs->ModelToView, prs->WorldToView,
                     prs->ModelToWorld);
                                                                        140
    prs->Updated = 0x0E;
```

```
  break;

case RENDER_NORMAL:
  static_send(3);
  prs->nx = static_receive_f();
  prs->ny = static_receive_f();
  prs->nz = static_receive_f();
                                                            150
  static_send(0);

  prs->Updated = 0x0E;
  break;

case RENDER_SET_LIT:
  static_send(1);
  prs->pInfo.p.Mode.lit = static_receive();
  static_send(0);
  prs->Updated = 0x0E;                                      160
  break;

case RENDER_SET_USEAMB:
  static_send(1);
  prs->pInfo.p.Mode.useamb = static_receive();
  static_send(0);
  prs->Updated = 0x0E;
  break;

case RENDER_SET_USEDIR:                                     170
  static_send(1);
  prs->pInfo.p.Mode.usedir = static_receive();
  static_send(0);
  prs->Updated = 0x0E;
  break;

case RENDER_SET_TEXMODE:
  static_send(1);
  prs->pInfo.p.Mode.texmode = static_receive();
  static_send(0);                                           180
  prs->Updated = 0x0E;
  break;

case RENDER_SET_TEXALPHA:
  static_send(1);
  prs->pInfo.p.Mode.texalpha = static_receive();
  static_send(0);
  prs->Updated = 0x0E;
  break;
```

```
case RENDER_SET_COLALPHA:                                          190
  static_send(1);
  prs->pInfo.p.Mode.colalpha = static_receive();
  static_send(0);
  prs->Updated = 0x0E;
  break;

case RENDER_SET_COLINTERP:
  static_send(1);
  prs->pInfo.p.Mode.colinterp = static_receive();                  200
  static_send(0);
  prs->Updated = 0x0E;
  break;

case RENDER_SET_LITINTERP:
  static_send(1);
  prs->pInfo.p.Mode.litinterp = static_receive();
  static_send(0);
  prs->Updated = 0x0E;
  break;                                                           210

case RENDER_SET_TEXINTERP:
  static_send(1);
  prs->pInfo.p.Mode.texinterp = static_receive();
  static_send(0);
  prs->Updated = 0x0E;
  break;

case RENDER_SET_OUTOFORDER:
  static_send(1);                                                  220
  prs->pInfo.p.Mode.outoforder = static_receive();
  static_send(0);
  prs->Updated = 0x0E;
  break;

case RENDER_SET_TEXTILE:
  static_send(1);
  prs->pInfo.p.Mode.textile = static_receive();
  static_send(0);
  prs->Updated = 0x0E;                                             230
  break;

case RENDER_SET_NOUSEZ:
  static_send(1);
  prs->pInfo.p.Mode.nousez = static_receive();
  static_send(0);
  prs->Updated = 0x0E;
```

```
  break;

case RENDER_SET_NOWRITEZ:                                            240
  static_send(1);
  prs->pInfo.p.Mode.nowritez = static_receive();
  static_send(0);
  prs->Updated = 0x0E;
  break;

case RENDER_SET_TEXTUREID:
  static_send(1);
  prs->pInfo.TextureID = static_receive();
  static_send(0);                                                    250
  prs->Updated = 0x0E;
  break;

case RENDER_COLTEXBALANCE:
  static_send(1);
  prs->pInfo.ColTexBalance = static_receive_f();
  static_send(0);
  prs->Updated = 0x0E;
  break;
                                                                     260
case RENDER_ALPHATHRESH:
  static_send(1);
  prs->pInfo.alphaThresh = static_receive();
  static_send(0);
  prs->Updated = 0x0E;
  break;

case RENDER_AMBCOLOR:
  static_send(1);
  prs->pInfo.ambColor = static_receive();                            270
  static_send(0);
  prs->Updated = 0x0E;
  break;

case RENDER_DIRCOLOR:
  static_send(1);
  prs->pInfo.dirColor = static_receive();
  static_send(0);
  prs->Updated = 0x0E;
  break;                                                             280

case RENDER_DIRLIGHT:
  static_send(3);
  prs->ldx = static_receive_f();
  prs->ldy = static_receive_f();
```

```
    prs->ldz = static_receive_f();
    static_send(0);
    prs->dirdefined = 1;
    prs->Updated = 0x0E;
    break;                                                          290

  case RENDER_AMBREFLECT:
    static_send(1);
    prs->ambreflect = static_receive();
    static_send(0);
    prs->Updated = 0x0E;
    break;

  case RENDER_DIRREFLECT:
    static_send(1);                                                 300
    prs->dirreflect = static_receive();
    static_send(0);
    prs->Updated = 0x0E;
    break;

  case RENDER_CLEARFB:
    {
      int i;
      unsigned page, rgbx;
                                                                    310
      static_send(2);
      page = static_receive(); // page (like z->fbmode)
      rgbx = static_receive(); // rgbx
      static_send(0);

      for(i = 0; i < VWIDTH * VHEIGHT; i++)
        {
          fb_set_pixel_rawaddr(i, rgbx, page & FBMODE_BACK,
                               (page & FBMODE_FRONT)>>1);
        }                                                           320
    }
    break;

  case RENDER_CLEARZ:
    {
      int i;
      static_send(0);
      static_send(0);

      for(i = 0; i < VWIDTH*VHEIGHT; i++)                           330
        {
          pzbd->buf[i] = 0x7FFFFFFF;
        }
```

```
    // flush zbd
    flush_variable(pzbd, sizeof(ZBufData));
  }

  break;
                                                                    340
case RENDER_SETPAGE:
  static_send(1);
  // page .. FBMODE_NONE, BACK, FRONT, BOTH
  pzbd->fbmode = static_receive();
  flush_word(&pzbd->fbmode);
  static_send(0);
  break;

case RENDER_FLIPPAGE:
  static_send(1);                                                   350
  if(static_receive())// wait for vsync?
    fb_flip_page_vsync();
  else
    fb_flip_page();
  static_send(0);
  break;

case RENDER_ALLOCATE_TEXTURE:
  {
    unsigned sizex, sizey;                                          360
    unsigned totalsize;
    signed nexttexID, texID = -1;
    TexAllocation *pAlloc = 0;

    // get next texture id to use
    for(nexttexID = 0; ptm->pTexEntryTable[nexttexID].valid && nexttexID
          < ptm->MaxTextures; nexttexID++);

    static_send(2);
    sizex = static_receive();                                       370
    sizey = static_receive();

    if(nexttexID < ptm->MaxTextures)
      {
        totalsize = sizex*sizey;

        if(ptm->pAllocHead == 0)
          {
            // base case, all memory is clear
            if(totalsize*sizeof(unsigned) <= ptm->TexMemorySize)   380
              {
```

```
          pAlloc = (TexAllocation*)malloc(sizeof(TexAllocation));
          if(pAlloc)
            {
              ptm->pAllocHead = ptm->pAllocTail = pAlloc;
              pAlloc->ID = texID = nexttexID;
              pAlloc->pBegin = ptm->pTexMemory;
              pAlloc->pEnd = ptm->pTexMemory + totalsize;
              pAlloc->pNext = 0;
              pAlloc->pPrev = 0;                                          390
              ptm->TexMemoryFree -= totalsize*sizeof(unsigned);
            }
        }
    }
  else
    {
      // search for a block large enough to hold the texture
      if(totalsize*4 <= ptm->TexMemoryFree)
        {
          TexAllocation * pta;                                            400
          unsigned * lastbegin = ptm->pTexMemory +
            ptm->TexMemorySize/sizeof(unsigned);
          for(pta = ptm->pAllocTail; pta != 0;
              lastbegin = pta->pBegin, pta = pta->pPrev)
            {
              if(lastbegin - pta->pEnd >= totalsize)
                {
                  // here's a spot that will work, allocate it right after
                  // pta->pEnd, and allocate new texallocation at pAlloc.
                  pAlloc = (TexAllocation *) malloc(sizeof(TexAllocation));   410
                  if(pAlloc)
                    {
                      if(pta->pNext)
                        pta->pNext->pPrev = pAlloc;
                      else
                        ptm->pAllocTail = pAlloc;

                      pAlloc->pNext = pta->pNext;
                      pAlloc->pPrev = pta;
                      pta->pNext = pAlloc;                                420
                      pAlloc->ID = texID = nexttexID;
                      pAlloc->pBegin = pta->pEnd;
                      pAlloc->pEnd = pAlloc->pBegin + totalsize;
                      ptm->TexMemoryFree -= totalsize*sizeof(unsigned);
                    }
                }
            }

          if(pta == 0)
```

```
                { // went all the way to the beginning, do final check                     430
                  // if there's room at beginning, allocate new texalloc at pAlloc.

                  if(lastbegin - ptm->pTexMemory >= totalsize)
                    {
                      pAlloc = (TexAllocation*)malloc(sizeof(TexAllocation));
                      if(pAlloc)
                        {
                          ptm->pAllocHead->pPrev = pAlloc;
                          pAlloc->pNext = ptm->pAllocHead;
                          pAlloc->pPrev = 0;                                               440
                          ptm->pAllocHead = pAlloc;
                          pAlloc->ID = texID = nexttexID;
                          pAlloc->pBegin = ptm->pTexMemory;
                          pAlloc->pEnd = pAlloc->pBegin + totalsize;
                          ptm->TexMemoryFree -= totalsize*sizeof(unsigned);
                        }
                    }
                } // if(pta == 0)
            } // if(totalsize*4 <= ptm->TexMemoryFree)
        } // else ( if(ptm->pAllocHead == 0))                                              450
    } // if(nexttexID < ptm->MaxTextures)

  if(pAlloc && texID != -1)
    {
      // allocated the texture correctly, now add the texture entry
      ptm->pTexEntryTable[texID].valid = 1;
      ptm->pTexEntryTable[texID].updated = 0x0F;
      ptm->pTexEntryTable[texID].Width = sizex;
      ptm->pTexEntryTable[texID].Height = sizey;
      ptm->pTexEntryTable[texID].pAlloc = pAlloc;                                          460
      ptm->pTexEntryTable[texID].pBegin = pAlloc->pBegin;

      ptm->NumTextures++;

      // and flush texentry out
      flush_variable(&ptm->pTexEntryTable[texID], sizeof(TexEntry));

      // don't need to flush TexAllocation, as only the main
      // processor (this one) ever uses that data.
      // likewise, don't need to flush TexManager out.                                    470

    }

  static_send(1);
  static_send(texID);

}
```

```
  break;

case RENDER_DEALLOC_TEXTURE:                                              480
  {
    signed id;
    TexAllocation *pAlloc;
    static_send(1);
    id = static_receive();

    if(id >= 0 && id < ptm->MaxTextures)
      {
        pAlloc = ptm->pTexEntryTable[id].pAlloc;
        if(ptm->pTexEntryTable[id].valid && pAlloc)                         490
          {
            if(pAlloc->pNext)
              pAlloc->pNext->pPrev = pAlloc->pPrev;
            else
              ptm->pAllocTail = pAlloc->pPrev;

            if(pAlloc->pPrev)
              pAlloc->pPrev->pNext = pAlloc->pNext;
            else
              ptm->pAllocHead = pAlloc->pNext;                              500

            ptm->TexMemoryFree += (pAlloc->pEnd - pAlloc->pBegin)
              *sizeof(unsigned);
            ptm->NumTextures--;
            free(pAlloc);
            ptm->pTexEntryTable[id].pAlloc = 0;
            ptm->pTexEntryTable[id].valid = 0;
            ptm->pTexEntryTable[id].updated = 0xF;

            flush_variable(&ptm->pTexEntryTable[id], sizeof(TexEntry));     510

            // don't need to flush TexAllocation, as only the main
            // processor (this one) ever uses that data.
            // likewise, don't need to flush TexManager out.
          }
      }

    static_send(0);
  }
  break;                                                                    520

case RENDER_UPLOAD_TEXTURE:
  {
    unsigned size, temp;
    signed id;
```

```
      unsigned *texbegin = 0, *texend = 0, *texiter;

      static_send(2);
      id = static_receive();
      size = static_receive();                                                    530
      static_send(0); // reading back 0

      // hack: at this point, static network sends us
      //         next word unconditionally

      if(size > 0)
        {
          if(id >= 0 && id < ptm->MaxTextures)
            if(ptm->pTexEntryTable[id].valid && ptm->pTexEntryTable[id].pAlloc)
              {                                                                   540
                texiter = texbegin = ptm->pTexEntryTable[id].pAlloc->pBegin;
                texend = ptm->pTexEntryTable[id].pAlloc->pEnd;
              }

          // get first word
          temp = static_receive();
          size--;

          if(texiter != texend)
            *texiter = temp;                                                      550

          texiter++;

          static_send(size); // get rest of words
          for( ;texiter < texend && size > 0; size--, texiter++)
            {
              *texiter = static_receive();
            }

          // program screwed up, but try not to lock up...                        560
          for( ; size > 0; size-- )
            static_receive();

          // flush out what we uploaded
          if(texbegin != texend)
            flush_variable(texbegin, (texend - texbegin)*sizeof(unsigned));

          static_send(0); // reading back 0

        }                                                                         570

    }
    break;
```

```
case RENDER_TEXMEM_AVAIL:
  static_send(0);
  static_send(1);
  static_send(ptm->TexMemoryFree); // texmem avail
  break;
                                                                580
case RENDER_COMPACT_TEXMEM:
  // TODO
  static_send(0);
  static_send(0);
  break;

case RENDER_WRITE_FB:
  {
    int x,y;
    unsigned page,rgbx;                                         590
    static_send(4);
    x = static_receive(); // x
    y = static_receive(); // y
    page = static_receive(); // page
    rgbx = static_receive(); // rgbx
    fb_set_pixel_rgbx(x, y, rgbx,
                      (page & FBMODE_BACK) != 0,
                      (page & FBMODE_FRONT) != 0);

    static_send(0);                                             600
  }
  break;

case RENDER_WRITE_FB_BLOCK:
  {
    unsigned length;
    // TODO
    static_send(4);
    static_receive(); // x
    static_receive(); // y                                      610
    static_receive(); // page
    length = static_receive(); // length
    static_send(0);

    if(length > 0)
      {
        // first rgbx in block
        static_receive();
        length--;
        static_send(length);                                    620
        for( ; length > 0 ; length--)
```

```
          static_receive(); // rest of rgbx's
        static_send(0);
      }

  }
  break;

case RENDER_READ_FB:
  {                                                                 630
    int x,y;
    unsigned page;
    static_send(3);
    x = static_receive(); // x
    y = static_receive(); // y
    page = static_receive(); // page - 0 = back, 1 = front
    static_send(1);
    static_send(fb_read_pixel(x, y, page));
  }
  break;                                                            640

case RENDER_READ_FB_BLOCK:
  {
    unsigned length;
    // TODO
    static_send(4);
    static_receive(); // x
    static_receive(); // y
    static_receive(); // page
    length = static_receive(); // length                            650
    static_send(length);
    for( ; length > 0; length--)
      static_send(0); // read rgbx
  }

  break;

case RENDER_WRITE_Z:
  {
    unsigned x,y;                                                   660
    signed val;
    static_send(3);
    x = static_receive(); // x
    y = static_receive(); // y
    val = static_receive(); // val
    pzbd->buf[x+VWIDTH*y] = val;
    flush_word(&pzbd->buf[x+VWIDTH*y]);
    static_send(0);
  }
```

```
  break;                                                        670

case RENDER_WRITE_Z_BLOCK:
  {
    unsigned length;
    // TODO
    static_send(3);
    static_receive(); // x
    static_receive(); // y
    length = static_receive(); // length
    static_send(0);                                             680

    if(length > 0)
      {
        // first val in block
        static_receive();
        length--;
        static_send(length);
        for( ; length > 0 ; length--)
          static_receive(); // rest of val's
        static_send(0);                                         690
      }

  }
  break;

case RENDER_READ_Z:
  {
    unsigned x,y;
    static_send(2);
    x = static_receive(); //x                                   700
    y = static_receive(); //y
    static_send(1);
    invalidate_word(&pzbd->buf[x+VWIDTH*y]);
    static_send(pzbd->buf[x+VWIDTH*y]); // val
  }
  break;

case RENDER_READ_Z_BLOCK:
  {
    unsigned length;                                            710
    // TODO
    static_send(3);
    static_receive(); // x
    static_receive(); // y
    length = static_receive(); // length
    static_send(length);
    for( ; length > 0; length--)
```

```
        static_send(0); // read val
      }
      break;                                                        720

    case RENDER_RESET:
      {
        int i;
        static_send(0);
        static_send(0);

        // clear out render state
        ClearRenderState();
                                                                    730
        // clear out texture memory

        // (assuming that everything is set up correctly!)
        for(i = 0; i < ptm->MaxTextures; i++)
          {
            if(ptm->pTexEntryTable[i].valid)
              {
                ptm->pTexEntryTable[i].valid = 0;
                free(ptm->pTexEntryTable[i].pAlloc);
              }                                                     740
          }

        ptm->TexMemoryFree = ptm->TexMemorySize;
        ptm->NumTextures = 0;
        ptm->pAllocHead = 0;
        ptm->pAllocTail = 0;


        // clear out z buffer
        pzbd->fbmode = FBMODE_BACK;                                 750

#ifdef INIT_ZBUF
        for(i = 0; i < VWIDTH*VHEIGHT; i++)
          {
            pzbd->buf[i] = 0x7FFFFFFF;
          }

        // flush zbd
        flush_variable(pzbd, sizeof(ZBufData));
#else                                                               760
        flush_variable(pzbd->fbmode, sizeof(unsigned));
#endif

        // TODO: init framebuffer if compiler define says so
```

```
        // tell fb to reset
        fb_reset();
      }
      break;
                                                                770
    case RENDER_HALT:
      static_send(0);
      static_send(0);
      // halting!
      while(1);
      break;

    default:
      static_send(0);
      static_send(0);                                           780
    }

  return 0;
}

void begin(void)
{
  // things we need to do on initial bootup:
  //// set up the interrupt handlers
  setup_interrupts();                                           790
  raw_set_status_EX_MASK(0x00000000); // all off
  raw_user_interrupts_on();
  raw_interrupts_on();

  // for framebuffer code
  //       funny desty (sender x,y = 0)
  fb_init_fbhdr(0,0);

  //// allocate shared memory for stage1.
  prs = (RenderState*)malloc(sizeof(RenderState));              800

  //// set renderstate to startup defaults, and invalidate
  ClearRenderState();
  // only initialize seqnum at startup, not on reset since stage4
  // never gets wind of a reset happening
  prs->pInfo.p.SeqNum = 1;
  prs->LaggedSeqNum = 1;
  flush_variable(prs, sizeof(RenderState));

  //// send shared memory pointers to other tiles in stage1.       810

  gdn_send_hdr(0, 1, 0, 0, 0, 0, 1);
  gdn_send(prs);
```

```
gdn_send_hdr(0, 1, 0, 0, 0, 0, 2);
gdn_send(prs);
gdn_send_hdr(0, 1, 0, 0, 0, 0, 3);
gdn_send(prs);

//// wait for acknowledgements back
gdn_receive();                                                          820
gdn_receive();
gdn_receive();

//// (we don't really need to ack on stage 2- they don't do
////  anything except in scenestream, and the network can buffer
////  for them until they're ready)

//// ask stage3 to allocate tex mem, and tex control structures.
gdn_send_hdr(0, 1, 0, 0, 0, 2, 0);
gdn_send(0);                                                            830

//// wait for response from stage 3, and store pointers.
ptm = (TexManager *)gdn_receive();

//// tell stage4 to allocate z-buffer and other shared mem it needs
gdn_send_hdr(0, 1, 0, 0, 0, 3, 0);
gdn_send(0);

//// wait for response from stage4, and store pointer
pzbd = (ZBufData*)gdn_receive();                                        840

while(1)
  {
    // going into command mode:
    //// set up the static network
    setup_switch_main();

    //// send acknowledgement out to renderhost, telling it we're booted
    //// and ready for commands.
    static_send(1);                                                     850

    // command mode:
    //// loop reading command, performing action, sending back
    ////  responses if necessary. Everything can be done with memory
    ////  accesses (make sure to flush changed memory) and gdn messages
    ////  (for writing to framebuffer) except scenestream mode.
    while(!doCommand(static_receive()));

    // going into scenestream:
    //// make sure memory changes are all flushed. if any pixels were   860
    ////  written, do one more read and wait for response to make sure
```

```
//// it's serialized.
prs->Updated = 0x0; // tiles always invalidate renderstate on start
flush_variable(prs, sizeof(RenderState));
flush_variable(ptm, sizeof(TexManager));
flush_variable(pzbd->fbmode, sizeof(unsigned));
fb_read_pixel(0,0,0);

//// enable gdn_avail interrupt (gdn messages sent when it's time
//// to endscene and flush pipe)                                          870
raw_set_status_EX_MASK(0x00000020);

//// set up the static network (and send 1, then 0)
setup_switch_scenestream();
static_send(1);
static_send(0);

//// send one word out of static network to tell it we're ready to
//// accept messages/
static_send(1);                                                           880

// scenestream mode:
DoSceneStream(prs, 0);

// old wordy description of scenestream follows:
//// loop with algorithm defined in Stage1-Common.c. Commands are run
//// with no acknowledgements. When endscene command occurs, host will
//// wait for an acknowledgement before sending any more. Tile that gets
//// endscene will then perform a flush, which consists of sending a gdn
//// message to all the other Stage1 tiles, and turning the gdn_avail       890
//// interrupt off for itself. All the tiles then send a specially encoded
//// primitive to the next stage and down the pipe. They then go back to
//// their normal scenestream mode. When the primitive gets to the
//// bottom stage, the bottom stage cleans up, and sends a
//// gdn message back to the tile that started the flush (this tile
//// number has to be both encoded in its original gdn message to the
//// other tiles, and in the special flush primitive).

//// When the flushing tile gets all 4 messages, then it knows that the
//// flush is complete, and sends another gdn message to all 4 tiles with a    900
//// special code that means the scene is done (this is needed because
//// the same flushing mechanism is used to clear the pipeline when the
//// sequence number rolls over). All tiles then reset their static networks
//// to the starting position. Tile 0 then leaves scenestream (gdn_avail
//// interrupt can be same for everyone)

// from scenestream back to command mode:
//// turn off gdn_avail interrupt
raw_set_status_EX_MASK(0x00000000); // all off
```

```
                                                                                910
      // invalidate render state, if needed
      invalidate_variable(&prs->Updated, sizeof(unsigned));
      if( prs->Updated & 0x01)
        invalidate_variable(prs, sizeof(RenderState));

      prs->Updated &= 0xE;

      //// (note, all this will happen at beginning of while loop ->)
      ////// set the static network back up
      ////// send acknowledgement of endscene back out to renderhost        920
      ////// go to "command mode" above
    }

  // async resets?
  //// too complex to drain out dynamic network state. won't implement these

  // sync resets? halts?
  //// these can be implemented as standard commands. all other tiles will
  //// be blocked waiting for something.
                                                                                930

}
```

C.8 Stage1-Main-sw.S

```
// setup_switch from starsearch/examples/multi_tile/static_net/mixed/compute_sw.S

        .text
        .align 2
.global setup_switch_main

.ent setup_switch_main
setup_switch_main:
        mtsri   SW_FREEZE, 1        // Freeze the switch.
        la      $8, sw_start        // Get switch starting address.          10
        mtsr    SW_PC, $8           // Set the switch PC.
        mtsri   SW_FREEZE, 0        // Get with switch running.
        jr      $31                 // Return.
.end setup_switch_main

        .swtext
        .align 3
// Start of switch code.
sw_start:
        nop             route $csto->$cWo   // First word from proc to west      20
                                            // to start render_host

read:   nop             route $cWi->$csti   // send first word to proc
        move    $1, $csto                   // count of words to read
        BEQZD $1, $1, reply
readL:  BNEZD $1, $1, readL route $cWi->$csti

reply:  move    $1, $csto                   // count of words in reply
        BEQZD $1, $1, read
replyL: BNEZD $1, $1, replyL route $csto->$cWo                                   30

        j       read
```

C.9 Stage1-Aux.c

```
// Stage1-Aux.c
//
//  Ken Taylor, MIT Master's Thesis 2004
//
// Last Updated: 5/18/2004
//
// This file implements the startup and control code for tiles > 0
// in stage 1. Does some basic setup, then stays in a scenestream
// loop.
                                                                    10
#include "module_test.h" // includes raw.h
#include "raw_compiler_defs.h" // for PASS (testing)
#include "Stage1-datatypes.h" // shared datatypes, includes render_datatypes.h
#include "Common-sw.h" // flush/invalidate

#define gdn_send_hdr(F, l, u, oY, oX, dY, dX) \
  gdn_send(F<<29|l<<24|u<<20|oY<<15|oX<<10|dY<<5|dX)

void setup_switch_scenestream(void);
                                                                    20
void begin(void)
{
  RenderState *prs;
  int tileNum;

  // set up interrupts
  setup_interrupts();
  raw_set_status_EX_MASK(0x00000000); // all off
  raw_user_interrupts_on();
  raw_interrupts_on();                                              30

  tileNum = raw_get_abs_pos_x();

  // receive global pointer to render state from tile 0
  prs = gdn_receive();

  // invalidate cache
  invalidate_variable(prs, sizeof(RenderState));

  // send back ack                                                  40
  gdn_send_hdr(0,1,0,0,tileNum,0,0);
  gdn_send(0);

  // turn gdn_avail interrupt on for scenestream
  raw_set_status_EX_MASK(0x00000020);
```

```
  // we just stay in scenestream for the rest of the time
  while(1)
    {
      // reset switch to start of cycle                                    50
      setup_switch_scenestream();
      static_send(1);
      static_send(0);

      // do scene stream until an endprim occurs.
      DoSceneStream(prs, tileNum);
    }
}
                                                                          60
```

C.10 Stage1-Common.c

```
// Stage1-Common.c
//
// Ken Taylor, MIT Master's Thesis 2004
//
// Last Updated: 5/18/2004
//
// This file implements the code used in common across stage1 to
// run scenestreaming mode.

#include "raw_compiler_defs.h"                                          10
#include "raw.h"
#include "Stage1-datatypes.h"
#include "render_cmds.h"
#include "Common-sw.h" // flush/invalidate

void signal_proc_ready(void);

#define gdn_send_hdr(F, l, u, oY, oX, dY, dX) \
  gdn_send(F<<29|l<<24|u<<20|oY<<15|oX<<10|dY<<5|dX)
                                                                        20

// matrix multiply! r = M*e
void MatrixMult(float *r0, float *r1, float *r2, float *r3,
                TransformMatrix M,
                float e0, float e1, float e2, float e3)
{
  *r0 = M[0][0]*e0 + M[0][1]*e1 + M[0][2]*e2 + M[0][3]*e3;
  *r1 = M[1][0]*e0 + M[1][1]*e1 + M[1][2]*e2 + M[1][3]*e3;
  *r2 = M[2][0]*e0 + M[2][1]*e1 + M[2][2]*e2 + M[2][3]*e3;
  *r3 = M[3][0]*e0 + M[3][1]*e1 + M[3][2]*e2 + M[3][3]*e3;              30
}

// matrix to matrix multiply! X = M*Y
void MatrixMatrixMult(TransformMatrix X, TransformMatrix M,
                      TransformMatrix Y)
{
  int i,j,k;
  for(i = 0; i<4; i++)
    for(j = 0; j<4; j++)
      {                                                                 40
        X[i][j] = 0;
        for(k = 0; k < 4; k++)
          {
            X[i][j] += M[i][k] * Y[k][j];
          }
```

```
    }
}

// invert and transpose a matrix! X = (Y^-1)^T
// note: used only for normals, so:                                      50
// - only computing the adjoint, not the inverse, and,
// - only computing it for the upper left 3x3 (no translation on normals)
// code from http://www.gignews.com/realtime020100.htm, accessed 4/20/04
void MatrixInvTrans(TransformMatrix X, TransformMatrix Y)
{
  X[0][0] = Y[1][1] * Y[2][2] - Y[1][2] * Y[2][1];
  X[0][1] = Y[1][2] * Y[2][0] - Y[1][0] * Y[2][2];
  X[0][2] = Y[1][0] * Y[2][1] - Y[1][1] * Y[2][0];
  X[1][0] = Y[2][1] * Y[0][2] - Y[2][2] * Y[0][1];
  X[1][1] = Y[2][2] * Y[0][0] - Y[2][0] * Y[0][2];                       60
  X[1][2] = Y[2][0] * Y[0][1] - Y[2][1] * Y[0][0];
  X[2][0] = Y[0][1] * Y[1][2] - Y[0][2] * Y[1][1];
  X[2][1] = Y[0][2] * Y[1][0] - Y[0][0] * Y[1][2];
  X[2][2] = Y[0][0] * Y[1][1] - Y[0][1] * Y[1][0];
  X[0][3] = X[1][3] = X[2][3] = 0;
  X[3][3] = 1;
}


unsigned pipeHDR;                                                        70

void SendFlushPrim(unsigned tileNum)
{
  gdn_send(pipeHDR | 2<<24);
  gdn_send(RENDER_P_FLUSH);
  gdn_send(tileNum);
}

void FlushPipeline(unsigned tileNum)
{                                                                        80
  // turn gdn_avail interrupt off
  raw_set_status_EX_MASK(0x00000000); // all off

  // send gdn messages with our tile number to all other stage 1 tiles.
  if(tileNum != 0)
    {
      gdn_send_hdr(0,1,0,0,tileNum,0,0);
      gdn_send(tileNum);
    }
  if(tileNum != 1)                                                       90
    {
      gdn_send_hdr(0,1,0,0,tileNum,0,1);
      gdn_send(tileNum);
```

```
    }
  if(tileNum != 2)
    {
      gdn_send_hdr(0,1,0,0,tileNum,0,2);
      gdn_send(tileNum);
    }
  if(tileNum != 3)                                          100
    {
      gdn_send_hdr(0,1,0,0,tileNum,0,3);
      gdn_send(tileNum);
    }

  // send flush command down pipe

  SendFlushPrim(tileNum);

  // wait for all gdn responses from last stage.              110
  gdn_receive();
  gdn_receive();
  gdn_receive();
  gdn_receive();

  // turn gdn_avail back on.
  raw_set_status_EX_MASK(0x00000020);

}
                                                            120
// a helper function - reads from the sn,
// and  decrements the block variable. if it's
// zero, reads a new one from the static network.
// assumes that *b is currently >0!
static inline unsigned block_receive(unsigned *b)
{
  unsigned temp = static_receive();
  if(--(*b) <= 0)
    *b = static_receive();
  return temp;                                              130
}
static inline float block_receive_f(unsigned *b)
{
  float temp;
  temp = static_receive_f();
  if(--(*b) <= 0)
    *b = static_receive();
  return temp;
}
                                                            140
// rs = pointer to the global RenderState structure
```

```
    //if(blockLength > 0)
    //{
    //   rs->rgba = block_receive(&blockLength);
    //   rs->Updated = 0x0F;
    //}
// tileNum = our tile number, used for Updated checking and
//            flushing
void ExeSceneStream(RenderState * rs, unsigned tileNum)
{                                                                     150
  int i,j;
  unsigned blockLength;
  unsigned numVerts = 0; // number of vertices we've got (don't render
  //                          without 3.
  InputPrim ip; // data stored on input
  TransPrim tp; // object to output

  float tnx,tny,tnz; // temp normal in world coords, for lighting
  float tempz; // temp z before converting to fixed point.
                                                                      160
  unsigned unordered = 0; // is it an unordered prim, or ordered?
  unsigned isendscene = 0;

  unsigned visible = 0; // is visible or was clipped?

  unsigned flushmode = 0;

  // first, invalidate rs->Updated.
  invalidate_word(&rs->Updated);
                                                                      170
  // check to see if our bit index is 1
  if( rs->Updated & (1 << tileNum))
    //if it is, invalidate all of rs in the cache
    invalidate_variable(rs, sizeof(RenderState));
  else
    {
      // else, just invalidate rs->pInfo.p.SeqNum and rs->LaggedSeqNum
      invalidate_word(&rs->pInfo.p.SeqNum);
      invalidate_word(&rs->LaggedSeqNum);
    }                                                                 180

  // set our bit in rs->Updated to be 0
  rs->Updated &= ~(1 << tileNum);

  // clear input prim
  for(i = 0; i<3; i++)
    {
      //ip.v[i].x = 0.0f;
      //ip.v[i].y = 0.0f;
```

```
    //ip.v[i].z = 0.0f;                                                    190
    ip.v[i].nx = 1.0f;
    ip.v[i].ny = 0.0f;
    ip.v[i].nz = 0.0f;
    //ip.v[i].u = 0.0f;
    //ip.v[i].v = 0.0f;
    ip.v[i].rgba = 0;

  }
                                                                           200
// read blockLength from static network
blockLength = static_receive();


// now, loop while blockLength!=0:
while(blockLength > 0)
  {
    unsigned cmd;
    //// read next command. decrement blockLength. if blockLength = 0 now,
    ////  read in blockLength.                                             210
    cmd = block_receive(&blockLength);

    //// switch on command:
    ////  read in command's data. update:
    ////   ip, *rs, numVerts.
    ////    when making a new vertex, get data from *rs.
    ////    when changing normals or whatnot, update *rs.
    ////   keep track of blockLength, if it reaches 0,
    ////    read in blockLength again. if read blockLength = 0, break.
    //// if command is endscene.... set isendscene = 1                     220

    switch(cmd)
      {
      case RENDER_ENDSCENE:
        isendscene = 1;
        break;

      case RENDER_VERTEX:
        if(blockLength > 0)
          {                                                                230
            ip.v[numVerts].x = block_receive_f(&blockLength);
            ip.v[numVerts].y = block_receive_f(&blockLength);
            ip.v[numVerts].z = block_receive_f(&blockLength);
            ip.v[numVerts].u = block_receive_f(&blockLength);
            ip.v[numVerts].v = block_receive_f(&blockLength);
            ip.v[numVerts].nx = rs->nx;
            ip.v[numVerts].ny = rs->ny;
```

```
        ip.v[numVerts].nz = rs->nz;

        ip.v[numVerts].rgba = rs->rgba;                              240

        numVerts++;
      }

    break;

  case RENDER_COLOR:
    if(blockLength > 0)
      {
        rs->rgba = block_receive(&blockLength);                      250
        rs->Updated = 0x0F;
        flush_word(&rs->rgba);
      }
    break;

  case RENDER_MODELMATRIX:
    if(blockLength > 0)
      {
        for(i = 0; i < 4; i++)
          for(j = 0; j < 4; j++)                                     260
            {
              rs->ModelToWorld[i][j] = block_receive_f(&blockLength);
            }

        MatrixMatrixMult(rs->ModelToView, rs->WorldToView,
                         rs->ModelToWorld);

        MatrixInvTrans(rs->NormalToWorld, rs->ModelToWorld);

        rs->Updated = 0x0F;                                          270
        flush_variable(rs->ModelToWorld, sizeof(TransformMatrix));
        flush_variable(rs->ModelToView, sizeof(TransformMatrix));
        flush_variable(rs->NormalToWorld, sizeof(TransformMatrix));
      }
    break;

  case RENDER_VIEWMATRIX:
    if(blockLength > 0)
      {
        for(i = 0; i < 4; i++)                                       280
          for(j = 0; j < 4; j++)
            {
              rs->WorldToView[i][j] = block_receive_f(&blockLength);
            }
```

```
        MatrixMatrixMult(rs->ModelToView, rs->WorldToView,
                         rs->ModelToWorld);

        rs->Updated = 0x0F;
        flush_variable(rs->WorldToView, sizeof(TransformMatrix));           290
        flush_variable(rs->ModelToView, sizeof(TransformMatrix));
      }
    break;

  case RENDER_NORMAL:
    if(blockLength > 0)
      {
        rs->nx = block_receive_f(&blockLength);
        rs->ny = block_receive_f(&blockLength);
        rs->nz = block_receive_f(&blockLength);                              300

        rs->Updated = 0x0F;
        flush_word(&rs->nx);
        flush_word(&rs->ny);
        flush_word(&rs->nz);
      }
    break;

  case RENDER_SET_LIT:
    if(blockLength > 0)                                                      310
      {
        rs->pInfo.p.Mode.lit = block_receive(&blockLength);

        rs->Updated = 0x0F;
        flushmode = 1;
      }
    break;

  case RENDER_SET_USEAMB:
    if(blockLength > 0)                                                      320
      {
        rs->pInfo.p.Mode.useamb = block_receive(&blockLength);

        rs->Updated = 0x0F;
        flushmode = 1;

      }
    break;

  case RENDER_SET_USEDIR:                                                    330
    if(blockLength > 0)
      {
        rs->pInfo.p.Mode.usedir = block_receive(&blockLength);
```

```
        rs->Updated = 0x0F;
        flushmode = 1;

      }
    break;
                                                                        340
  case RENDER_SET_TEXMODE:
    if(blockLength > 0)
      {
        rs->pInfo.p.Mode.texmode = block_receive(&blockLength);

        rs->Updated = 0x0F;
        flushmode = 1;
      }
    break;
                                                                        350
  case RENDER_SET_TEXALPHA:
    if(blockLength > 0)
      {
        rs->pInfo.p.Mode.texalpha = block_receive(&blockLength);

        rs->Updated = 0x0F;
        flushmode = 1;

      }
    break;                                                              360

  case RENDER_SET_COLALPHA:
    if(blockLength > 0)
      {
        rs->pInfo.p.Mode.colalpha = block_receive(&blockLength);

        rs->Updated = 0x0F;
        flushmode = 1;

      }                                                                 370
    break;

  case RENDER_SET_COLINTERP:
    if(blockLength > 0)
      {
        rs->pInfo.p.Mode.colinterp = block_receive(&blockLength);

        rs->Updated = 0x0F;
        flushmode = 1;
                                                                        380
      }
```

```
  break;

case RENDER_SET_LITINTERP:
  if(blockLength > 0)
    {
      rs->pInfo.p.Mode.litinterp = block_receive(&blockLength);

      rs->Updated = 0x0F;
      flushmode = 1;                                                    390

    }
  break;

case RENDER_SET_TEXINTERP:
  if(blockLength > 0)
    {
      rs->pInfo.p.Mode.texinterp= block_receive(&blockLength);

      rs->Updated = 0x0F;                                               400
      flushmode = 1;

    }
  break;

case RENDER_SET_OUTOFORDER:
  if(blockLength > 0)
    {
      rs->pInfo.p.Mode.outoforder = block_receive(&blockLength);
                                                                        410
      rs->Updated = 0x0F;
      flushmode = 1;

    }
  break;

case RENDER_SET_TEXTILE:
  if(blockLength > 0)
    {
      rs->pInfo.p.Mode.textile = block_receive(&blockLength);           420

      rs->Updated = 0x0F;
      flushmode = 1;

    }
  break;

case RENDER_SET_NOUSEZ:
  if(blockLength > 0)
```

```
    {                                                              430
      rs->pInfo.p.Mode.nousez = block_receive(&blockLength);

      rs->Updated = 0x0F;
      flushmode = 1;

    }
  break;

case RENDER_SET_NOWRITEZ:
  if(blockLength > 0)                                              440
    {
      rs->pInfo.p.Mode.nowritez = block_receive(&blockLength);

      rs->Updated = 0x0F;
      flushmode = 1;

    }
  break;

case RENDER_SET_TEXTUREID:                                         450
  if(blockLength > 0)
    {
      rs->pInfo.TextureID = block_receive(&blockLength);

      rs->Updated = 0x0F;
      flush_word(&rs->pInfo.TextureID);
    }
  break;

case RENDER_COLTEXBALANCE:                                         460
  if(blockLength > 0)
    {
      rs->pInfo.ColTexBalance = block_receive_f(&blockLength);

      rs->Updated = 0x0F;
      flush_word(&rs->pInfo.ColTexBalance);
    }
  break;

case RENDER_ALPHATHRESH:                                           470
  if(blockLength > 0)
    {
      rs->pInfo.alphaThresh = block_receive(&blockLength);

      rs->Updated = 0x0F;
      flush_word(&rs->pInfo.alphaThresh);
    }
```

```
    break;

  case RENDER_AMBCOLOR:
    if(blockLength > 0)
      {
        rs->pInfo.ambColor = block_receive(&blockLength);

        rs->Updated = 0x0F;
        flush_word(&rs->pInfo.ambColor);

      }
    break;

  case RENDER_DIRCOLOR:
    if(blockLength > 0)
      {
        rs->pInfo.dirColor = block_receive(&blockLength);

        rs->Updated = 0x0F;
        flush_word(&rs->pInfo.dirColor);

      }
    break;

  case RENDER_DIRLIGHT:
    if(blockLength > 0)
      {
        rs->ldx = block_receive_f(&blockLength);
        rs->ldy = block_receive_f(&blockLength);
        rs->ldz = block_receive_f(&blockLength);

        rs->dirdefined = 1;

        rs->Updated = 0x0F;
        flush_word(&rs->ldx);
        flush_word(&rs->ldy);
        flush_word(&rs->ldz);

      }
    break;

  case RENDER_AMBREFLECT:
    if(blockLength > 0)
      {
        rs->ambreflect = block_receive(&blockLength);

        rs->Updated = 0x0F;
        flush_word(&rs->ambreflect);
```

```
      }
    break;

  case RENDER_DIRREFLECT:                                             530
    if(blockLength > 0)
      {
        rs->dirreflect = block_receive(&blockLength);

        rs->Updated = 0x0F;
        flush_word(&rs->dirreflect);

      }
    break;
                                                                      540
  default:

  }
}

if(flushmode)
  {
    flush_variable(&rs->pInfo.p.Mode, sizeof(ModeBits));
  }
                                                                      550

if(isendscene)
  {
    //// flush the pipeline
    FlushPipeline(tileNum);

    //// send endscene gdn message to all stage1 tiles, including ourself.
    ////    cleanup will occur in DoSceneStream after this proc leaves.
    gdn_send_hdr(0,1,0,0,tileNum,0,0);
    gdn_send(0xFFFFFFFF);                                             560
    gdn_send_hdr(0,1,0,0,tileNum,0,1);
    gdn_send(0xFFFFFFFF);
    gdn_send_hdr(0,1,0,0,tileNum,0,2);
    gdn_send(0xFFFFFFFF);
    gdn_send_hdr(0,1,0,0,tileNum,0,3);
    gdn_send(0xFFFFFFFF);

  }
else
  {                                                                   570
    unsigned wordsleft, gdnleft;
    unsigned *ptr;
```

```
if( numVerts >= 3 )
  {

    //// set unordered according to heuristic
    ////  from rs->pInfo.p.Mode data, and alpha values in ip.v[0,1,2].

    if(rs->pInfo.p.Mode.outoforder == 1)                                    580
      unordered = 1;
    else
      {
        unsigned vertextrans =
          (((ip.v[0].rgba & 0x0FF) != 0 || (ip.v[1].rgba & 0x0FF) != 0 ||
            (ip.v[2].rgba & 0x0FF) != 0) &&
           ((ip.v[0].rgba & 0x0FF) != 255 || (ip.v[1].rgba & 0x0FF) != 255 ||
            (ip.v[2].rgba & 0x0FF) != 255));

        unordered = !                                                       590
          (rs->pInfo.p.Mode.nousez || rs->pInfo.p.Mode.nowritez ||
           (rs->pInfo.p.Mode.texalpha == 1 &&
            ((rs->pInfo.p.Mode.texmode & 0x2) || // 2,3,6...
             ((rs->pInfo.p.Mode.texmode & 0x4) && // 4, 5
              vertextrans))) ||
           (rs->pInfo.p.Mode.colalpha == 1 && vertextrans &&
            (rs->pInfo.p.Mode.texmode == 1 || rs->pInfo.p.Mode.texmode == 3
             || rs->pInfo.p.Mode.texmode == 6)));
      }
                                                                            600
    //// if ordered:
    ////  set rs->LaggedSeqNum to rs->pInfo.p.SeqNum + 1
    if (!unordered)
      {
        rs->LaggedSeqNum = rs->pInfo.p.SeqNum + 1;
      }

    //// increment rs->pInfo.p.SeqNum
    rs->pInfo.p.SeqNum++;
                                                                            610
    //// just flush Updated, SeqNum, LaggedSeqNum
    flush_word(&rs->LaggedSeqNum);
    flush_word(&rs->pInfo.p.SeqNum);

  }

flush_word(&rs->Updated);

////
////  check to see if rs->pInfo.p.SeqNum has wrapped around                620
//// (if we start it at 1 on startup, 0 will mean wraparound!)
```

```
////  note: doesn't matter if laggedseqnum is wrapped around before
////        seqnum, since first prim to use laggedseqnum will have
////        a seqnum = laggedseqnum anyway
//// if it has, flush the pipeline before continuing....
if(rs->pInfo.p.SeqNum == 0)
  FlushPipeline(tileNum);

////
//// send token to switch, telling it that rs has been flushed,             630
////  (and pipe flushed if wraparound) and that next tile can start.
static_send(1);

// clear transprim
for(i = 0; i < 3; i++)
  {
    tp.v[i].x = 0.0f;
    tp.v[i].y = 0.0f;
    tp.v[i].z = 0;
    tp.v[i].w1 = 1.0f;                                                      640
    tp.v[i].u = 0.0f;
    tp.v[i].v = 0.0f;
    tp.v[i].r = 0.0f;
    tp.v[i].g = 0.0f;
    tp.v[i].b = 0.0f;
    tp.v[i].a = 0.0f;
  }


// copy info over to transprim                                              650
tp.pInfo = rs->pInfo;
tp.pInfo.p.Mode.outoforder = unordered;
tp.pInfo.p.Mode.draw = 0;
if(unordered)
  {
    tp.pInfo.p.SeqNum = rs->LaggedSeqNum;
  }
else
  {
    tp.pInfo.p.SeqNum--;                                                    660
  }

if( numVerts >= 3 )
  {

    // do screenspace transform
    for(i = 0; i < 3; i++)
      {
        MatrixMult(&tp.v[i].x, &tp.v[i].y, &tp.v[i].z, &tp.v[i].w1,
```

```
                    rs->ModelToView,                                                670
                    ip.v[i].x, ip.v[i].y, ip.v[i].z, 1.0f);

          // todo: optimize for when we don't use z coordinate?
        }

      // clipping. necessary:
      //   - backface culling
      //   - at least dropping polys behind near plane
      // would be nice TODO:
      //   - full frustum culling                                                       680
      //   - clipping to near plane, including splitting triangles, regen vertices+values
      //       (this can create more than one prim - complexifying this code!)

      visible = 1;

      // near plane and singularity dropping
      if( tp.v[0].w1>0 && (tp.v[0].z <= -tp.v[0].w1) ||
          tp.v[0].w1<0 && (tp.v[0].z >= -tp.v[0].w1) ||
          tp.v[1].w1>0 && (tp.v[1].z <= -tp.v[1].w1) ||
          tp.v[1].w1<0 && (tp.v[1].z >= -tp.v[1].w1) ||                                690
          tp.v[2].w1>0 && (tp.v[2].z <= -tp.v[2].w1) ||
          tp.v[2].w1<0 && (tp.v[2].z >= -tp.v[2].w1)
          || abs(tp.v[0].w1) <= 1e-100
          || abs(tp.v[1].w1) <= 1e-100
          || abs(tp.v[2].w1) <= 1e-100)
        visible = 0;

      // backface culling - vertices are defined clockwise when prim facing the screen
      if( visible )
        {                                                                               700
          //if point 2 is on rhs of point 0->1 vector, keep...
          // ax + by + c > 0
          // a = y1 - y0, b = x0 - x1, c = y0x1-y1x0

          if((tp.v[1].y - tp.v[0].y)*tp.v[2].x + (tp.v[0].x - tp.v[1].x)*tp.v[2].y
              + tp.v[0].y * tp.v[1].x - tp.v[1].y*tp.v[0].x >= 0 )
            visible = 0;
        }

    }                                                                                   710


  if(visible)
    {
      tp.pInfo.p.Mode.draw = 1;
```

```
// ambient light modulation
if(tp.pInfo.p.Mode.lit && tp.pInfo.p.Mode.useamb)
  { // modulate amb intensity with reflectivity                              720
    unsigned intens = rs->pInfo.ambColor & 0x0FF;
    rs->pInfo.ambColor &= 0xFFFFFF00;
    // max of each is 255 - treated as 1.0 (fixed point modulation)
    rs->pInfo.ambColor |= ((rs->ambreflect * intens) + 255 ) >> 8;
  }

if(!rs->dirdefined)
  tp.pInfo.p.Mode.usedir = 0;

// directed light modulation                                                 730
if (tp.pInfo.p.Mode.lit && tp.pInfo.p.Mode.usedir)
  for(i = 0; i < 3; i++)
    {
      float w;
      float prod;
      //// transform normal into world coordinates tnx tny tnz
      MatrixMult(&tnx, &tny, &tnz, &w,
                 rs->NormalToWorld,
                 ip.v[i].nx, ip.v[i].ny, ip.v[i].nz, 1.0f);
                                                                             740
      // normalize normal
      w = 1.0f/sqrtf(tnx*tnx+tny*tny+tnz*tnz);
      tnx *= w;
      tny *= w;
      tnz *= w;

      //// dot product tn<xyz> with rs->ld<xyz> (be careful of sign!)
      prod = - tnx*rs->ldx - tny*rs->ldy - tnz*rs->ldz;

      tp.v[i].intensity = (prod <= 0) ? 0 : ( prod * ((float)rs->dirreflect)/255.0f  750
        * ((float)(rs->pInfo.dirColor & 0x0FF))/255.0f );
    }


// perspective division!
for(i = 0; i < 3; i++)
  {
    tp.v[i].w1 = 1/tp.v[i].w1;
    tp.v[i].x = tp.v[i].w1 * tp.v[i].x;
    tp.v[i].y = tp.v[i].w1 * tp.v[i].y;                                      760
    if (!(tp.pInfo.p.Mode.nowritez && tp.pInfo.p.Mode.nousez))
      tp.v[i].z = tp.v[i].w1 * tp.v[i].z;
    if (tp.pInfo.p.Mode.texmode != 0 && tp.pInfo.p.Mode.texmode != 2)
      {
        tp.v[i].r = tp.v[i].w1 * (ip.v[i].rgba >> 24);
```

```
          tp.v[i].g = tp.v[i].w1 * (( ip.v[i].rgba << 8 ) >> 24);
          tp.v[i].b = tp.v[i].w1 * (( ip.v[i].rgba << 16 ) >> 24);
        }
      if (tp.pInfo.p.Mode.colalpha != 0)
        tp.v[i].a = tp.v[i].w1 * (ip.v[i].rgba & 0x0FF);                    770
      if (tp.pInfo.p.Mode.texmode > 1)
        {
          tp.v[i].u = tp.v[i].w1 * ip.v[i].u;
          tp.v[i].v = tp.v[i].w1 * ip.v[i].v;
        }
      if(tp.pInfo.p.Mode.lit && tp.pInfo.p.Mode.usedir)
        tp.v[i].intensity = tp.v[i].w1 * tp.v[i].intensity;
    }
  }
                                                                             780
  // stream all of tp south with however many gdn messages it takes.

  //  (max 31 byte payload...) - first byte tells it whether it's
  //  a prim or a flush command. Rest is tp structure

  if(visible)
    magic_perf_drawnprim();

  wordsleft = sizeof(TransPrim)/sizeof(unsigned);
  ptr = (unsigned*) &tp;                                                     790

  gdnleft = wordsleft >= 30 ? 30 : wordsleft;

  gdn_send(pipeHDR | (gdnleft+ 1)<<24);
  gdn_send(RENDER_P_PRIM);

  for(; wordsleft > 0; wordsleft--, gdnleft--, ptr++)
    {
      if(gdnleft <= 0)
        {                                                                    800
          gdnleft = wordsleft >= 31 ? 31 : wordsleft;
          gdn_send(pipeHDR | gdnleft<<24);
        }

      gdn_send(*ptr);
    }

}

////                                                                         810
//// signal switch that we're ready for more input
signal_proc_ready();
```

```
  ////
  //// and that's it! quit and main program will call us for next
  ////  prim, unless there was an endscene command
}

extern volatile unsigned doFlush;                                   820
extern volatile unsigned FlushProcNum;
extern volatile unsigned endPrim;

// loops executing scene stream commands
// sends pipeline flush south when necessary
// exits on endprim.
void DoSceneStream(RenderState * rs, unsigned tileNum)
{
  unsigned gotInput = 0;
  doFlush = endPrim = 0;                                            830
  pipeHDR = tileNum<<10|1<<5|tileNum;

  while(!endPrim)
    {
      magic_perf_startbusywait();
      while(!(gotInput = raw_get_status_SW_BUF1() & 0x000000E0)
            && !doFlush && !endPrim);
      magic_perf_endbusywait();

      if(doFlush)                                                   840
        {
          doFlush = 0;
          SendFlushPrim(FlushProcNum);
        }
      if(gotInput)
        {
          gotInput = 0;
          ExeSceneStream(rs, tileNum);
        }
    }                                                               850
}
```

C.11 Stage1-sw.S

```
// interrupt code inspired by starsearch/module_tests/interrupts/external/tests.S
        .text
        .align 2

# interrupt vector
ivec:   j       HNDL_GDN_AVAIL

# Copy ivec down to 0x50
        .global setup_interrupts
        .ent    setup_interrupts                                          10
setup_interrupts:
        addiu   $9, $0, %lo(ivec)
        aui     $9, $9, %hi(ivec)
        ilw     $12, 0($9)
        isw     $12, 0x50($0)
        jr      $31
        .end setup_interrupts

// cache-free saving point for interrupt
.swtext                                                                   20
gdn_avail_save1:        .word 0
gdn_avail_save2:        .word 0

.text

HNDL_GDN_AVAIL:
        swsw    $2, %lo(gdn_avail_save1)($0)
        swsw    $3, %lo(gdn_avail_save2)($0)

        addu    $2, $0, $cgni                                             30

        // if gdn message is a tile num
        sltiu   $3, $2, 4
        BEQ     $3, $0, hga_endprim

        // set doFlush and FlushProcNum
        la      $3, FlushProcNum
        sw      $2, 0($3)
        la      $3, doFlush
        addiu   $2, $0, 1                                                 40
        sw      $2, 0($3)
        j       hga_done

hga_endprim:
        // else if gdn message is all 1s
```

```
        // set endPrim
        la      $3, endPrim
        addiu   $2, $0, 1
        sw      $2, 0($3)
                                                                    50
hga_done:

        swlw    $2, %lo(gdn_avail_save1)($0)
        swlw    $3, %lo(gdn_avail_save2)($0)

        dret

// a pointer to the shared memory, stored here so the assembly
// can access it.
.data                                                               60
                .global doFlush
                .global FlushProcNum
                .global endPrim
doFlush:        .word 0
FlushProcNum: .word 0
endPrim:        .word 0
```

C.12 Stage1-sw-0.S

```
// scene streaming static code, tile 0

        .text
        .align 2
.global setup_switch_scenestream

.ent setup_switch_scenestream
setup_switch_scenestream:
        mtsri   SW_FREEZE, 1        // Freeze the switch.
        la      $8, SceneStream_SWBegin // Get switch starting address.          10
        mtsr    SW_PC, $8           // Set the switch PC.
        mtsri   SW_FREEZE, 0        // Get with switch running.
        jr      $31                 // Return.
.end setup_switch_scenestream

// "signals" the switch state machine that the proc is
// ready by forcing its pc into a different state.
// Don't call this if the switch is already in a "ready"
//  state or it'll get messed up!
.global signal_proc_ready                                                       20
.ent signal_proc_ready
signal_proc_ready:
        mtsri   SW_FREEZE, 1
        nop
        nop
        nop
        nop
        mfsr    $8, SW_PC
        // note: the logic here is assuming a certain ordering
        // of states in the memory for <= >= comparison                         30
        la      $9, ADone
        beq     $8, $9, spr_Done_Active
        la      $9, Done_Active
        beq     $8, $9, spr_Done_Active
        la      $9, Counting_1_Ready
        sltu    $9, $8, $9
        bne     $9, $0, spr_Passing
        la      $9, Take_Turn
        sltu    $9, $8, $9
        bne     $9, $0, spr_Counting                                            40
        la      $9, Pass_Turn
        sltu    $9, $8, $9
        beq     $9, $0, spr_Pass_Turn
        j       spr_done
```

```
spr_Done_Active:
        // it can only be at ADone if it *just* sent the last
        // part of the prim to the tile.
        la      $8, Done_Active_Ready
        j       spr_done                                          50

spr_Passing:
        la      $9, Passing_1_Busy
        la      $10, Passing_1_Ready
        subu    $9, $8, $9
        addu    $8, $9, $10
        j       spr_done

spr_Counting:
        la      $9, Counting_1_Busy                               60
        la      $10, Counting_1_Ready
        subu    $9, $8, $9
        addu    $8, $9, $10
        j       spr_done

spr_Pass_Turn:
        la      $9, Pass_Turn
        la      $10, Take_Turn
        subu    $9, $8, $9
        addu    $8, $9, $10                                       70

spr_done:
        mtsr    SW_PC, $8           // Set the switch PC.
        nop
        nop
        nop
        nop
        mtsri   SW_FREEZE, 0        // Get with switch running.
        jr      $31
                                                                  80
.end signal_proc_ready

        .swtext
        .align 3

SceneStream_SWBegin:
        // get a 1 and a zero from processor
        MOVE $1, $csto
        MOVE $0, $csto
                                                                  90
        // send processor token west to start scenestream
        // (only processor 1 does this)
        NOP     route $csto->$cWo
```

```
// start in Active state
//----------------------------------------//
Active:
        // get size of next data chunk (route sizes to proc)
        MOVE $2, $cWi              route $cWi->$csti
        // if chunk is zero, end prim                                        100
ADone: BEQZD $2,$2, Done_Active

ACount: // route chunk to processor
        BNEZD $2,$2, ACount        route $cWi->$csti
        j       Active

//----------------------------------------//
// (On ProcReady -> Done_Active_Ready+offset, pc set by proc)
Done_Active:
        // send a token east from proc                                       110
        J       Counting_1_Busy     route $csto->$cEo

//----------------------------------------//
Done_Active_Ready:
        // send a token east from proc
        J       Counting_1_Ready    route $csto->$cEo

//----------------------------------------//
Passing_1_Ready:
        // get the size of next data chunk, pass it east                     120
        MOVE $2, $cWi       route $cWi->$cEo

        // if chunk is zero, end prim!
        BEQZD $2,$2, Counting_2_Ready

P1RCount:
        BNEZD $2,$2, P1RCount route $cWi->$cEo
        j       Passing_1_Ready

Passing_2_Ready:                                                             130
        // get the size of next data chunk, pass it east
        MOVE $2, $cWi       route $cWi->$cEo

        // if chunk is zero, end prim!
        BEQZD $2,$2, Counting_3_Ready

P2RCount:
        BNEZD $2,$2, P2RCount route $cWi->$cEo
        j       Passing_2_Ready
                                                                             140
Passing_3_Ready:
```

```
        // get the size of next data chunk, pass it east
        MOVE $2, $cWi         route $cWi->$cEo

        // if chunk is zero, end prim!
        BEQZD $2,$2, Take_Turn

P3RCount:
        BNEZD $2,$2, P3RCount route $cWi->$cEo
        j         Passing_3_Ready                                    150

//------------------------------------------//
// (On ProcReady -> Passing_N_Ready+offset, pc set by proc)
Passing_1_Busy:
        // get the size of next data chunk, pass it east
        MOVE $2, $cWi         route $cWi->$cEo

        // if chunk is zero, end prim!
        BEQZD $2,$2, Counting_2_Busy
                                                                     160
P1BCount:
        BNEZD $2,$2, P1BCount route $cWi->$cEo
        j         Passing_1_Busy

Passing_2_Busy:
        // get the size of next data chunk, pass it east
        MOVE $2, $cWi         route $cWi->$cEo

        // if chunk is zero, end prim!
        BEQZD $2,$2, Counting_3_Busy                                 170

P2BCount:
        BNEZD $2,$2, P2BCount route $cWi->$cEo
        j         Passing_2_Busy

Passing_3_Busy:
        // get the size of next data chunk, pass it east
        MOVE $2, $cWi         route $cWi->$cEo

        // if chunk is zero, end prim!                               180
        BEQZD $2,$2, Pass_Turn

P3BCount:
        BNEZD $2,$2, P3BCount route $cWi->$cEo
        j         Passing_3_Busy

//------------------------------------------//
Counting_1_Ready:
        BNEZ  $cEi, Passing_1_Ready
```

```
                                                                          190
Counting_2_Ready:
        BNEZ  $cEi, Passing_2_Ready

Counting_3_Ready:
        BNEZ  $cEi, Passing_3_Ready
        J       TTP1

//------------------------------------------------//
// (On ProcReady -> Counting_1_Ready+offset, pc set by proc)
Counting_1_Busy:                                                          200
        BNEZ  $cEi, Passing_1_Busy

Counting_2_Busy:
        BNEZ  $cEi, Passing_2_Busy

Counting_3_Busy:
        BNEZ  $cEi, Passing_3_Busy
        J       PTP1

//------------------------------------------------//                      210
Take_Turn:
        // wait for token from east (from last proc)
        MOVE $3, $cEi
TTP1: j        Active

//------------------------------------------------//
// (On ProcReady -> Take_Turn+offset, pc set by proc)
Pass_Turn:
        // wait for token from east (from last proc)
        MOVE $3, $cEi                                                     220
        // send a token east
PTP1: j        Counting_1_Busy route $0->$cEo

//------------------------------------------------//
```

C.13 Stage1-sw-1.S

```
// scene streaming static code, tile 1

        .text
        .align 2
.global setup_switch_scenestream

.ent setup_switch_scenestream
setup_switch_scenestream:
        mtsri   SW_FREEZE, 1        // Freeze the switch.
        la      $8, SceneStream_SWBegin // Get switch starting address.        10
        mtsr    SW_PC, $8           // Set the switch PC.
        mtsri   SW_FREEZE, 0        // Get with switch running.
        jr      $31                 // Return.
.end setup_switch_scenestream

// "signals" the switch state machine that the proc is
// ready by forcing its pc into a different state.
// Don't call this if the switch is already in a "ready"
//  state or it'll get messed up!
.global signal_proc_ready                                                      20
.ent signal_proc_ready
signal_proc_ready:
        mtsri   SW_FREEZE, 1
        nop
        nop
        nop
        nop
        mfsr    $8, SW_PC
        // note: the logic here is assuming a certain ordering
        // of states in the memory for <= >= comparison                        30
        la      $9, Active
        sltu    $9, $8, $9
        bne     $9, $0, spr_Idle
        la      $9, ADone
        beq     $8, $9, spr_Done_Active
        la      $9, Done_Active
        beq     $8, $9, spr_Done_Active
        la      $9, Counting_1_Ready
        sltu    $9, $8, $9
        bne     $9, $0, spr_Passing                                            40
        la      $9, Take_Turn
        sltu    $9, $8, $9
        bne     $9, $0, spr_Counting
        la      $9, Pass_Turn
        beq     $8, $9, spr_Pass_Turn
```

```
        j       spr_done

spr_Idle:
        la      $9, Idle_Busy
        la      $10, Idle_Ready                                        50
        subu    $9, $8, $9
        addu    $8, $9, $10
        j       spr_done

spr_Done_Active:
        // it can only be at ADone if it *just* sent the last
        // part of the prim to the tile.
        la      $8, Done_Active_Ready
        j       spr_done
                                                                       60
spr_Passing:
        la      $9, Passing_1_Busy
        la      $10, Passing_1_Ready
        subu    $9, $8, $9
        addu    $8, $9, $10
        j       spr_done

spr_Counting:
        la      $9, Counting_1_Busy
        la      $10, Counting_1_Ready                                  70
        subu    $9, $8, $9
        addu    $8, $9, $10
        j       spr_done

spr_Pass_Turn:
        la      $8, Take_Turn

spr_done:
        mtsr    SW_PC, $8           // Set the switch PC.
        nop                                                            80
        nop
        nop
        nop
        mtsri   SW_FREEZE, 0        // Get with switch running.
        jr      $31

.end signal_proc_ready

        .swtext
        .align 3                                                       90

SceneStream_SWBegin:
        // get a 1 and a zero from processor
```

```
        MOVE $1, $csto
        MOVE $0, $csto
        j IRP1

// start in IRP1 state
//---------------------------------------//
Idle_Ready:                                                          100
        // send token west
        NOP         route $cEi->$cWo
        // wait for token
IRP1:   MOVE $3, $cWi
        J       Take_Turn

//---------------------------------------//
// (On ProcReady -> Idle_Ready+offset, pc set by proc)
Idle_Busy:
        // send token west                                           110
        NOP         route $cEi->$cWo
        // wait for token
IBP1:   MOVE $3, $cWi
        J       Pass_Turn

//---------------------------------------//
Active:
        // get size of next data chunk (route sizes to proc)
        MOVE $2, $cWi               route $cWi->$csti
                                                                     120
        // if chunk is zero, end prim
ADone: BEQZD $2,$2, Done_Active

ACount: // route chunk to processor
        BNEZD $2,$2, ACount         route $cWi->$csti
        j       Active

//---------------------------------------//
// (On ProcReady -> Done_Active_Ready+offset, pc set by proc)
Done_Active:                                                         130
        // send a token east from proc
        J       Counting_1_Busy     route $csto->$cEo

//---------------------------------------//
Done_Active_Ready:
        // send a token east from proc
        J       Counting_1_Ready    route $csto->$cEo

//---------------------------------------//
Passing_1_Ready:                                                     140
        // get the size of next data chunk, pass it east
```

```
        MOVE $2, $cWi        route $cWi->$cEo

        // if chunk is zero, end prim!
        BEQZD $2,$2, Counting_2_Ready

P1RCount:
        BNEZD $2,$2, P1RCount route $cWi->$cEo
        j         Passing_1_Ready
                                                                              150
Passing_2_Ready:
        // get the size of next data chunk, pass it east
        MOVE $2, $cWi        route $cWi->$cEo

        // if chunk is zero, end prim!
        BEQZD $2,$2, Idle_Ready

P2RCount:
        BNEZD $2,$2, P2RCount route $cWi->$cEo
        j         Passing_2_Ready                                             160

//-----------------------------------------//
// (On ProcReady -> Passing_N_Ready+offset, pc set by proc)
Passing_1_Busy:
        // get the size of next data chunk, pass it east
        MOVE $2, $cWi        route $cWi->$cEo

        // if chunk is zero, end prim!
        BEQZD $2,$2, Counting_2_Busy
                                                                              170
P1BCount:
        BNEZD $2,$2, P1BCount route $cWi->$cEo
        j         Passing_1_Busy

Passing_2_Busy:
        // get the size of next data chunk, pass it east
        MOVE $2, $cWi        route $cWi->$cEo

        // if chunk is zero, end prim!
        BEQZD $2,$2, Idle_Busy                                                180

P2BCount:
        BNEZD $2,$2, P2BCount route $cWi->$cEo
        j         Passing_2_Busy

//-----------------------------------------//
Counting_1_Ready:
        BNEZ  $cEi, Passing_1_Ready route $cEi->$cWo
```

```
Counting_2_Ready:                                                        190
        BNEZ  $cEi, Passing_2_Ready route $cEi->$cWo
        J       IRP1

//--------------------------------------------------------//
// (On ProcReady -> Counting_1_Ready+offset, pc set by proc)
Counting_1_Busy:
        BNEZ  $cEi, Passing_1_Busy  route $cEi->$cWo

Counting_2_Busy:
        BNEZ  $cEi, Passing_2_Busy  route $cEi->$cWo                     200
        J       IBP1

//--------------------------------------------------------//
Take_Turn:
        j       Active          route $1->$cWo

//--------------------------------------------------------//
// (On ProcReady -> Take_Turn+offset, pc set by proc)
Pass_Turn:
        // send a token east                                             210
        j       Counting_1_Busy route $0->$cEo, $0->$cWo

//--------------------------------------------------------//
```

C.14 Stage1-sw-2.S

```
// scene streaming static code, tile 2

        .text
        .align 2
.global setup_switch_scenestream

.ent setup_switch_scenestream
setup_switch_scenestream:
        mtsri   SW_FREEZE, 1        // Freeze the switch.
        la      $8, SceneStream_SWBegin // Get switch starting address.         10
        mtsr    SW_PC, $8           // Set the switch PC.
        mtsri   SW_FREEZE, 0        // Get with switch running.
        jr      $31                 // Return.
.end setup_switch_scenestream

// "signals" the switch state machine that the proc is
// ready by forcing its pc into a different state.
// Don't call this if the switch is already in a "ready"
//  state or it'll get messed up!
.global signal_proc_ready                                                   20
.ent signal_proc_ready
signal_proc_ready:
        mtsri   SW_FREEZE, 1
        nop
        nop
        nop
        nop
        mfsr    $8, SW_PC
        // note: the logic here is assuming a certain ordering
        // of states in the memory for <= >= comparison                     30
        la      $9, Active
        sltu    $9, $8, $9
        bne     $9, $0, spr_Idle
        la      $9, ADone
        beq     $8, $9, spr_Done_Active
        la      $9, Done_Active
        beq     $8, $9, spr_Done_Active
        la      $9, Counting_1_Ready
        sltu    $9, $8, $9
        bne     $9, $0, spr_Passing                                         40
        la      $9, Take_Turn
        sltu    $9, $8, $9
        bne     $9, $0, spr_Counting
        la      $9, Pass_Turn
        beq     $8, $9, spr_Pass_Turn
```

```
        j       spr_done

spr_Idle:
        la      $9, Idle_Busy
        la      $10, Idle_Ready                                         50
        subu    $9, $8, $9
        addu    $8, $9, $10
        j       spr_done

spr_Done_Active:
        // it can only be at ADone if it *just* sent the last
        // part of the prim to the tile.
        la      $8, Done_Active_Ready
        j       spr_done
                                                                        60
spr_Passing:
        la      $9, Passing_1_Busy
        la      $10, Passing_1_Ready
        subu    $9, $8, $9
        addu    $8, $9, $10
        j       spr_done

spr_Counting:
        la      $9, Counting_1_Busy
        la      $10, Counting_1_Ready                                   70
        subu    $9, $8, $9
        addu    $8, $9, $10
        j       spr_done

spr_Pass_Turn:
        la      $8, Take_Turn

spr_done:
        mtsr    SW_PC, $8               // Set the switch PC.
        nop                                                             80
        nop
        nop
        nop
        mtsri   SW_FREEZE, 0            // Get with switch running.
        jr      $31

.end signal_proc_ready

        .swtext
        .align 3                                                        90

SceneStream_SWBegin:
        // get a 1 and a zero from processor
```

```
        MOVE $1, $csto
        MOVE $0, $csto

        j IRP1

// start in IRP1 state
//---------------------------------------//                    100
Idle_Ready:
        // send token east to west
        NOP         route $cEi->$cWo
        // wait for token
IRP1:   MOVE $3, $cWi
        J       Take_Turn

//---------------------------------------//
// (On ProcReady -> Idle_Ready+offset, pc set by proc)
Idle_Busy:                                                      110
        // send token east to west
        NOP         route $cEi->$cWo
        // wait for token
IBP1:   MOVE $3, $cWi
        J       Pass_Turn

//---------------------------------------//
Active:
        // get size of next data chunk (route sizes to proc)
        MOVE $2, $cWi               route $cWi->$csti           120

        // if chunk is zero, end prim
ADone: BEQZD $2,$2, Done_Active

ACount: // route chunk to processor
        BNEZD $2,$2, ACount         route $cWi->$csti
        j       Active

//---------------------------------------//
// (On ProcReady -> Done_Active_Ready+offset, pc set by proc)   130
Done_Active:
        // send a token east from proc
        J       Counting_1_Busy     route $csto->$cEo

//---------------------------------------//
Done_Active_Ready:
        // send a token east from proc
        J       Counting_1_Ready    route $csto->$cEo

//---------------------------------------//                    140
Passing_1_Ready:
```

```
        // get the size of next data chunk, pass it east
        MOVE $2, $cWi        route $cWi->$cEo

        // if chunk is zero, end prim!
        BEQZD $2,$2, Idle_Ready

P1RCount:
        BNEZD $2,$2, P1RCount route $cWi->$cEo
        j       Passing_1_Ready                                         150

//--------------------------------------//
// (On ProcReady -> Passing_N_Ready+offset, pc set by proc)
Passing_1_Busy:
        // get the size of next data chunk, pass it east
        MOVE $2, $cWi        route $cWi->$cEo

        // if chunk is zero, end prim!
        BEQZD $2,$2, Idle_Busy
                                                                        160
P1BCount:
        BNEZD $2,$2, P1BCount route $cWi->$cEo
        j       Passing_1_Busy

//--------------------------------------//
Counting_1_Ready:
        BNEZ $cEi, Passing_1_Ready route $cEi->$cWo
        J       IRP1

//--------------------------------------//                              170
// (On ProcReady -> Counting_1_Ready+offset, pc set by proc)
Counting_1_Busy:
        BNEZ $cEi, Passing_1_Busy route $cEi->$cWo
        J       IBP1

//--------------------------------------//
Take_Turn:
        j       Active          route $1->$cWo

//--------------------------------------//                              180
// (On ProcReady -> Take_Turn+offset, pc set by proc)
Pass_Turn:
        // send a token east
        j       Counting_1_Busy route $0->$cEo, $0->$cWo

//--------------------------------------//
```

C.15 Stage1-sw-3.S

```
// scene streaming static code, tile 3

        .text
        .align 2
.global setup_switch_scenestream

.ent setup_switch_scenestream
setup_switch_scenestream:
        mtsri   SW_FREEZE, 1        // Freeze the switch.
        la      $8, SceneStream_SWBegin // Get switch starting address.
        mtsr    SW_PC, $8           // Set the switch PC.
        mtsri   SW_FREEZE, 0        // Get with switch running.
        jr      $31                 // Return.
.end setup_switch_scenestream


// "signals" the switch state machine that the proc is
// ready by forcing its pc into a different state.
// Don't call this if the switch is already in a "ready"
//  state or it'll get messed up!
.global signal_proc_ready
.ent signal_proc_ready
signal_proc_ready:
        mtsri   SW_FREEZE, 1
        nop
        nop
        nop
        nop
        mfsr    $8, SW_PC
        // note: the logic here is assuming a certain ordering
        // of states in the memory for <= >= comparison
        la      $9, Active
        sltu    $9, $8, $9
        bne     $9, $0, spr_Idle
        la      $9, ADone
        beq     $8, $9, spr_ADone
        la      $9, Pass_Turn
        beq     $8, $9, spr_Pass_Turn
        j       spr_done

spr_Idle:
        la      $9, Idle_Busy
        la      $10, Idle_Ready
        subu    $9, $8, $9
        addu    $8, $9, $10
```

```
        j       spr_done

spr_ADone:
        // it can only be at ADone if it *just* sent the last
        // part of the prim to the tile.                                    50
        la      $8, Idle_Ready
        j       spr_done

spr_Pass_Turn:
        la      $8, Take_Turn

spr_done:
        mtsr    SW_PC, $8           // Set the switch PC.
        nop
        nop                                                                 60
        nop
        nop
        mtsri   SW_FREEZE, 0        // Get with switch running.
        jr      $31

.end signal_proc_ready


        .swtext
        .align 3                                                            70

SceneStream_SWBegin:
        // get a 1 and a zero from processor
        MOVE $1, $csto
        MOVE $0, $csto

        j       IRP1

// start in IRP1 state
//--------------------------------------//                                  80
Idle_Ready:
        // send token from proc to west
        NOP             route $csto->$cWo
        // wait for token
IRP1:   MOVE $3, $cWi
        J       Take_Turn

//--------------------------------------//
// (On ProcReady -> Idle_Ready+offset, pc set by proc)
Idle_Busy:                                                                  90
        // send token from proc to west
        NOP             route $csto->$cWo
        // wait for token
```

```
IBP1:  MOVE $3, $cWi
       J        Pass_Turn

//-----------------------------------------//
Active:
       // get size of next data chunk (route sizes to proc)
       MOVE $2, $cWi              route $cWi->$csti                 100

       // if chunk is zero, end prim
ADone: BEQZD $2,$2, Idle_Busy

ACount: // route chunk to processor
       BNEZD $2,$2, ACount        route $cWi->$csti
       j        Active

//-----------------------------------------//
Take_Turn:                                                          110
       j        Active       route $1->$cWo

//-----------------------------------------//
// (On ProcReady -> Take_Turn+offset, pc set by proc)
Pass_Turn:
       // send a token west
       j        IBP1         route  $0->$cWo

//-----------------------------------------//
                                                                    120
```

C.16 Stage2-Common.c

```
// Stage2-Common.c
//
//  Ken Taylor, MIT Master's Thesis 2004
//
// Last Updated: 5/18/2004
//
// This file implements the code used in common across stage2,
//  stage2 does rasterization and perspective correct interpolation
//  of parameters across the triangle sent from stage1. It sends
//  untextured/unblended fragments to stage3.                          10
#include "raw_compiler_defs.h"

#include "raw.h"
#include "render_datatypes.h"
#include "Common-sw.h"

void setup_switch(void);

void begin(void)
{                                                                      20
  int tileNum;
  int ii,j;
  TransPrim tp; // the prim we input
  UntexFragment utf; // fragmetns we output
  float ulx;
  float uly;
  float lrx;
  float lry;

  unsigned correctinterp; // are we doing any perspective-correct interpolation?   30

  float w10,w11,w12;

  // line equations: L - line value for top left corner axul+byul+c
  // (incremental    tL - temp value for each line
  //  model for      ttL - temp value for each pixel
  //  better         adx - x increment for L
  //  performance)   bdy - y increment for L
  float L0, tL0, adx0, bdy0; // v0->v1
  float L1, tL1, adx1, bdy1; // v1->v2                                 40
  float L2, tL2, adx2, bdy2; // v2->v0
  // clockwise, so rhs is < 0:
  //  a = y2-y1, b=x1-x2, c=y1x2-y2x1

  // plane equations, x - top left corner value
```

```
//                    tx - temp for each line
//                    ttx - temp for each pixel
//                    xdx - x increment for x
//                    xdy - y increment for x
// don't do incremental model for z, do calc for                     50
// each point (better accuracy?)
float za, zb, zc;
float r, tr, rdx, rdy;
float g, tg, gdx, gdy;
float b, tb, bdx, bdy;
float a, ta, adx, ady;
float u, tu, udx, udy;
float v, tv, vdx, vdy;
float i, ti, idx, idy;
float w1, tw1, w1dx, w1dy;                                           60
float me, b1e, b2e, detM1;
// precalc me = x2y3-x3y2
//         b1e = z2y3-z3y2
//         b2e = x2z3-x3z2
// det M  = x1(y2-y3)-y1(x2-x3)+me
// det B1 = z1(y2-y3)-y1(z2-z3)+b1e
// det B2 = x1(z2-z3)-z1(x2-x3)+b2e
// det B3 = -x1(b1e)-y1(b2e)+z1(me)
// A = detB1/detM, B=detB2/detM, C=detB3/detM
// Ax+By+C = z                                                       70

// when w1 is interped, invert and multiply each
// other interpreted value by it, for perspect correct.

//---this is where code actually begins---

tileNum = raw_get_abs_pos_x();


// start static network                                              80

setup_switch();

// loop forever:
while(1)
  {
    unsigned word;

    //// read next word from gdn, also send byte south.
    static_send(word = gdn_receive());                               90

    //// if it's a flush byte
    if(word == RENDER_P_FLUSH)
```

```
  {
    ////// send next byte (proc #) south
    static_send(gdn_receive());
  }
else
  {
    unsigned nousez, nowritez, texmode, colinterp, colalpha;                    100
    unsigned lit, usedir, litinterp;
    unsigned wordsleft, *ptr;

    ////// read in tp from gdn
    wordsleft = sizeof(TransPrim)/sizeof(unsigned);
    ptr = (unsigned *)&tp;

    for( ; wordsleft > 0; wordsleft--, ptr++)
      {
        (*ptr) = gdn_receive();                                                 110
      }

    ////// stream tp.pInfo south

    wordsleft = sizeof(PrimInfo)/sizeof(unsigned);
    ptr = (unsigned*)&tp.pInfo;

    for( ; wordsleft > 0; wordsleft--, ptr++)
      {
        static_send(*ptr);                                                      120
      }

    if(!tp.pInfo.p.Mode.draw)
      {
        // don't draw it, just pass the priminfo south (for sequencing)
        static_send(RENDER_P_ENDPRIM);
        continue;
      }

    // unpack bitfields for better performance                                  130
    nousez = tp.pInfo.p.Mode.nousez;
    nowritez = tp.pInfo.p.Mode.nowritez;
    texmode = tp.pInfo.p.Mode.texmode;
    colinterp = tp.pInfo.p.Mode.colinterp;
    colalpha = tp.pInfo.p.Mode.colalpha;
    lit = tp.pInfo.p.Mode.lit;
    usedir = tp.pInfo.p.Mode.usedir;
    litinterp = tp.pInfo.p.Mode.litinterp;

    ////// get inverses of (w^-1) values, to multiply with                     140
    //////  parameters to get their real values.
```

```
w10 = 1/tp.v[0].w1;
w11 = 1/tp.v[1].w1;
w12 = 1/tp.v[2].w1;

// scale prim's x and y values to screen space, using compiler defs.
// VWIDTH, VHEIGHT. scale so -1 -> 0, and 1 -> VWIDTH/HEIGHT
// (pixels are centered on .5 steps - pixel 0 is at 0.5,
// pixel 1 is at 1.5, etc.
tp.v[0].x = tp.v[0].x*(VWIDTH/2.0f) + VWIDTH/2.0f;                    150
tp.v[0].y = tp.v[0].y*(VHEIGHT/2.0f) + VHEIGHT/2.0f;
tp.v[1].x = tp.v[1].x*(VWIDTH/2.0f) + VWIDTH/2.0f;
tp.v[1].y = tp.v[1].y*(VHEIGHT/2.0f) + VHEIGHT/2.0f;
tp.v[2].x = tp.v[2].x*(VWIDTH/2.0f) + VWIDTH/2.0f;
tp.v[2].y = tp.v[2].y*(VHEIGHT/2.0f) + VHEIGHT/2.0f;

// for bounding box
//  minimum is lowest n such that n+0.5 is
//   greater than or equal to lowest pixel coord.
//  maximum is highest n such that n+0.5 is                               160
//   less than or equal to highest pixel coord.
//  we want the bounding box to be the lowest and highest n+0.5 that's
//   within the prim.
//  3.1 ... min at 3, max at 2 ... 3.6 min at 4, max at 3

ulx = (signed)(tp.v[0].x+0.5f);
lrx = (signed)(tp.v[0].x-0.5f);
uly = (signed)(tp.v[0].y+0.5f);
lry = (signed)(tp.v[0].y-0.5f);
                                                                      170
for(ii = 1; ii < 3; ii++)
  {
    ulx = (ulx <= (signed)(tp.v[ii].x + 0.5f))
      ? ulx : (signed)(tp.v[ii].x + 0.5f);
    uly = (uly <= (signed)(tp.v[ii].y + 0.5f))
      ? uly : (signed)(tp.v[ii].y + 0.5f);
    lrx = (lrx >= (signed)(tp.v[ii].x - 0.5f))
      ? lrx : (signed)(tp.v[ii].x - 0.5f);
    lry = (lry >= (signed)(tp.v[ii].y - 0.5f))
      ? lry : (signed)(tp.v[ii].y - 0.5f);                            180
  }

if(ulx < 0) ulx = 0;
if(ulx >= VWIDTH) ulx = VWIDTH - 1;
if(lrx < 0) ulx = 0;
if(lrx >= VWIDTH) lrx = VWIDTH - 1;
if(uly < 0) uly = 0;
if(uly >= VHEIGHT) uly = VHEIGHT - 1;
if(lry < 0) uly = 0;
```

```
if(lry >= VHEIGHT) lry = VHEIGHT - 1;                                        190

// move to center of pixels
ulx+=0.5;
uly+=0.5;
lrx+=0.5;
lry+=0.5;

////// set up plane equations for each line, z, rgba, uv, intensity:
// a = y2-y1, b=x1-x2, c=y1x2-y2x1 // rhs is inside // clockwise faces front
adx0 = tp.v[1].y - tp.v[0].y;                                                200
bdy0 = tp.v[0].x - tp.v[1].x;
L0 = adx0*ulx + bdy0*uly + tp.v[0].y*tp.v[1].x - tp.v[1].y*tp.v[0].x;
adx1 = tp.v[2].y - tp.v[1].y;
bdy1 = tp.v[1].x - tp.v[2].x;
L1 = adx1*ulx + bdy1*uly + tp.v[1].y*tp.v[2].x - tp.v[2].y*tp.v[1].x;
adx2 = tp.v[0].y - tp.v[2].y;
bdy2 = tp.v[2].x - tp.v[0].x;
L2 = adx2*ulx + bdy2*uly + tp.v[2].y*tp.v[0].x - tp.v[0].y*tp.v[2].x;

correctinterp = 0;                                                           210

// these values are the same for all parameters for x,y
me  = tp.v[1].x*tp.v[2].y - tp.v[2].x*tp.v[1].y;
detM1 = 1/(tp.v[0].x*(tp.v[1].y-tp.v[2].y)
          -tp.v[0].y*(tp.v[1].x-tp.v[2].x)+me);


if(!nousez || !nowritez)
  {
    // set up z interp:                                                      220
    b1e = tp.v[1].z*tp.v[2].y - tp.v[2].z*tp.v[1].y;
    b2e = tp.v[1].x*tp.v[2].z - tp.v[2].x*tp.v[1].z;
    za = detM1*(tp.v[0].z*(tp.v[1].y-tp.v[2].y)
              -tp.v[0].y*(tp.v[1].z-tp.v[2].z)+b1e);
    zb = detM1*(tp.v[0].x*(tp.v[1].z-tp.v[2].z)
              -tp.v[0].z*(tp.v[1].x-tp.v[2].x)+b2e);
    zc = detM1*(tp.v[0].z*me - tp.v[0].x*b1e - tp.v[0].y*b2e);
  }

if (texmode != 0 && texmode != 2)                                            230
{
  if(colinterp)
    {
      //set up r,g,b interp
      correctinterp = 1;
      b1e = tp.v[1].r*tp.v[2].y - tp.v[2].r*tp.v[1].y;
      b2e = tp.v[1].x*tp.v[2].r - tp.v[2].x*tp.v[1].r;
```

```
    rdx = detM1*(tp.v[0].r*(tp.v[1].y-tp.v[2].y)
              -tp.v[0].y*(tp.v[1].r-tp.v[2].r)+b1e);
    rdy = detM1*(tp.v[0].x*(tp.v[1].r-tp.v[2].r)                    240
              -tp.v[0].r*(tp.v[1].x-tp.v[2].x)+b2e);
    r = rdx*ulx + rdy*uly +
      detM1*(tp.v[0].r*me - tp.v[0].x*b1e - tp.v[0].y*b2e);

    b1e = tp.v[1].g*tp.v[2].y - tp.v[2].g*tp.v[1].y;
    b2e = tp.v[1].x*tp.v[2].g - tp.v[2].x*tp.v[1].g;
    gdx = detM1*(tp.v[0].g*(tp.v[1].y-tp.v[2].y)
              -tp.v[0].y*(tp.v[1].g-tp.v[2].g)+b1e);
    gdy = detM1*(tp.v[0].x*(tp.v[1].g-tp.v[2].g)
              -tp.v[0].g*(tp.v[1].x-tp.v[2].x)+b2e);                250
    g = gdx*ulx + gdy*uly +
      detM1*(tp.v[0].g*me - tp.v[0].x*b1e - tp.v[0].y*b2e);

    b1e = tp.v[1].b*tp.v[2].y - tp.v[2].b*tp.v[1].y;
    b2e = tp.v[1].x*tp.v[2].b - tp.v[2].x*tp.v[1].b;
    bdx = detM1*(tp.v[0].b*(tp.v[1].y-tp.v[2].y)
              -tp.v[0].y*(tp.v[1].b-tp.v[2].b)+b1e);
    bdy = detM1*(tp.v[0].x*(tp.v[1].b-tp.v[2].b)
              -tp.v[0].b*(tp.v[1].x-tp.v[2].x)+b2e);
    b = bdx*ulx + bdy*uly +                                          260
      detM1*(tp.v[0].b*me - tp.v[0].x*b1e - tp.v[0].y*b2e);
  }
else
  {
    // color is average of vertices
    r = (tp.v[0].r*w10 + tp.v[1].r*w11 + tp.v[2].r*w12)/3;
    g = (tp.v[0].g*w10 + tp.v[1].g*w11 + tp.v[2].g*w12)/3;
    b = (tp.v[0].b*w10 + tp.v[1].b*w11 + tp.v[2].b*w12)/3;
  }
if (colalpha != 0)                                                   270
  {
    if(colinterp)
      {
        // set up a interp
        correctinterp = 1;

        b1e = tp.v[1].a*tp.v[2].y - tp.v[2].a*tp.v[1].y;
        b2e = tp.v[1].x*tp.v[2].a - tp.v[2].x*tp.v[1].a;
        adx = detM1*(tp.v[0].a*(tp.v[1].y-tp.v[2].y)
                  -tp.v[0].y*(tp.v[1].a-tp.v[2].a)+b1e);            280
        ady = detM1*(tp.v[0].x*(tp.v[1].a-tp.v[2].a)
                  -tp.v[0].a*(tp.v[1].x-tp.v[2].x)+b2e);
        a = adx*ulx + ady*uly +
          detM1*(tp.v[0].a*me - tp.v[0].x*b1e - tp.v[0].y*b2e);
      }
```

```
      else
        {
          // alpha is average of vertices
          a = (tp.v[0].a*w10 + tp.v[1].a*w11 + tp.v[2].a*w12)/3;
        }                                                                    290
    }
}

if (texmode > 1)
  {
    //set up u,v interp
    correctinterp = 1;

    b1e = tp.v[1].u*tp.v[2].y - tp.v[2].u*tp.v[1].y;
    b2e = tp.v[1].x*tp.v[2].u - tp.v[2].x*tp.v[1].u;                         300
    udx = detM1*(tp.v[0].u*(tp.v[1].y-tp.v[2].y)
                -tp.v[0].y*(tp.v[1].u-tp.v[2].u)+b1e);
    udy = detM1*(tp.v[0].x*(tp.v[1].u-tp.v[2].u)
                -tp.v[0].u*(tp.v[1].x-tp.v[2].x)+b2e);
    u = udx*ulx + udy*uly +
      detM1*(tp.v[0].u*me - tp.v[0].x*b1e - tp.v[0].y*b2e);

    b1e = tp.v[1].v*tp.v[2].y - tp.v[2].v*tp.v[1].y;
    b2e = tp.v[1].x*tp.v[2].v - tp.v[2].x*tp.v[1].v;
    vdx = detM1*(tp.v[0].v*(tp.v[1].y-tp.v[2].y)                             310
                -tp.v[0].y*(tp.v[1].v-tp.v[2].v)+b1e);
    vdy = detM1*(tp.v[0].x*(tp.v[1].v-tp.v[2].v)
                -tp.v[0].v*(tp.v[1].x-tp.v[2].x)+b2e);
    v = vdx*ulx + vdy*uly +
      detM1*(tp.v[0].v*me - tp.v[0].x*b1e - tp.v[0].y*b2e);
  }

if (lit && usedir)
  {
      if(litinterp)                                                          320
        {
          // set up intensity interp
          correctinterp = 1;

          b1e = tp.v[1].intensity*tp.v[2].y - tp.v[2].intensity*tp.v[1].y;
          b2e = tp.v[1].x*tp.v[2].intensity - tp.v[2].x*tp.v[1].intensity;
          idx = detM1*(tp.v[0].intensity*(tp.v[1].y-tp.v[2].y)
                      -tp.v[0].y*(tp.v[1].intensity-tp.v[2].intensity)+b1e);
          idy = detM1*(tp.v[0].x*(tp.v[1].intensity-tp.v[2].intensity)
                      -tp.v[0].intensity*(tp.v[1].x-tp.v[2].x)+b2e);         330
          i = idx*ulx + idy*uly +
            detM1*(tp.v[0].intensity*me - tp.v[0].x*b1e - tp.v[0].y*b2e);
        }
```

```
        else
          {
            // intens is average of vertices
            i = (tp.v[0].intensity*w10 + tp.v[1].intensity*w11
                 + tp.v[2].intensity*w12)/3;
          }
      }                                                          340

  if(correctinterp)
    {
      // set up w1 interp

      b1e = tp.v[1].w1*tp.v[2].y − tp.v[2].w1*tp.v[1].y;
      b2e = tp.v[1].x*tp.v[2].w1 − tp.v[2].x*tp.v[1].w1;
      w1dx = detM1*(tp.v[0].w1*(tp.v[1].y−tp.v[2].y)
                 −tp.v[0].y*(tp.v[1].w1−tp.v[2].w1)+b1e);
      w1dy = detM1*(tp.v[0].x*(tp.v[1].w1−tp.v[2].w1)          350
                 −tp.v[0].w1*(tp.v[1].x−tp.v[2].x)+b2e);
      w1 = w1dx*ulx + w1dy*uly +
        detM1*(tp.v[0].w1*me − tp.v[0].x*b1e − tp.v[0].y*b2e);


    }

  ////// from uly to lry
  for(ii = (int)uly; ii <= (int)lry; ii++)
    {                                                            360
      unsigned gotrow = 0;
      tL0 = L0; tL1 = L1; tL2 = L2;
      L0+=bdy0; L1+= bdy1; L2+=bdy2;

      // also initialize incremental interp for r, g, b, a, u, v, i, and w1 for row
      // (surrounding everything by if clause adds too much overhead
      if(colinterp)
        {
          tr = r; tg = g; tb = b;
          r+=rdy; g+=gdy; b+=bdy;                                370
          ta = a;
          a+=ady;
        }
      tu = u; tv = v;
      u+=udy; v+=vdy;
      if(litinterp)
        {
          ti = i;
          i+=idy;
        }                                                        380
      tw1 = w1;
```

```
w1+=w1dy;

// from ulx to lrx
for(j = (int)ulx; j <= (int)lrx; j++)
  {
    if( tL0 < 0 && tL1 < 0 && tL2 < 0)
      {
        float tempw;
                                                                    390
        gotrow = 1;

        // fill in utf with x,y.
        utf.x = j;
        utf.y = ii;

        // fill in as necessary: z (scaled), u/w1,
        // v/w1, rgba (packed+scaled+/w1), intensity/w1
        if(!nousez || !nowritez)
          {                                                         400
            // note: this loses precision. TODO: take full
            // advantage of signed fixed point precision
            // somehow? (use software double-sized ints?)
            float tempz;

            tempz = za * ((float)j + 0.5f) + zb * ((float)ii + 0.5f) + zc;
            utf.z = tempz*(signed)(0x7FFFFFFF);
          }

        if(correctinterp)                                           410
          {
            tempw = 1/tw1;
          }

        if (texmode != 0 && texmode != 2)
          {
            if(colinterp)
              {
                utf.rgba = (((unsigned)(tr*tempw+0.5f)) & 0x0FF) << 24 |
                  (((unsigned)(tg*tempw+0.5f)) & 0x0FF) << 16 |     420
                  (((unsigned)(tb*tempw+0.5f)) & 0x0FF) << 8;
              }
            else
              {
                  // color is average of vertices
                utf.rgba = (((unsigned)(r+0.5f)) & 0x0FF) << 24 |
                  (((unsigned)(g+0.5f)) & 0x0FF) << 16 |
                  (((unsigned)(b+0.5f)) & 0x0FF) << 8;
              }
```

```
        if (colalpha != 0)                                          430
          {
            if(colinterp)
              utf.rgba |= ((unsigned)(ta*tempw+0.5f)) & 0x0FF;
            else
              utf.rgba |= ((unsigned)(a+0.5f)) & 0x0FF;
          }
      }
    if (texmode > 1)
      {
        utf.u = tu * tempw;                                         440
        utf.v = tv * tempw;
      }
    if (lit && usedir)
      {
        if(litinterp)
          utf.intensity = ti * tempw;
        else
          utf.intensity = i;
      }
                                                                    450
    magic_perf_fragment();
    // stream utf out on static network - first word indicates a utf
    // (vs end-of-prim)
    static_send(RENDER_P_FRAG);

    wordsleft = sizeof(UntexFragment)/sizeof(unsigned);
    ptr = (unsigned *)&utf;

    for( ; wordsleft > 0; wordsleft--, ptr++)
      {                                                             460
        static_send(*ptr);
      }
  }
else
  {
    if(gotrow == 1) // we were in the prim, and then left
      {
        break;
      }
  }                                                                 470

tL0+=adx0; tL1+=adx1; tL2+=adx2;

// increment tInterp in dx for r,g,b,a,u,v,i, and w1
tr+=rdx; tg+=gdx; tb+=bdx;
ta+=adx;
tu+=udx; tv+=vdx;
```

```
                ti+=idx;
                tw1+=w1dx;
            }
        }

        ////// send end-of-prim word out on sn
        static_send(RENDER_P_ENDPRIM);

        ////// i think that's it... end loop!
    }
  }
}
```

480

490

C.17 Stage2-sw.S

```
// setup_switch from starsearch/examples/multi_tile/static_net/mixed/compute_sw.S

        .text
        .align 2
.global setup_switch

.ent setup_switch
setup_switch:
        mtsri   SW_FREEZE, 1        // Freeze the switch.
        la      $8, sw_start        // Get switch starting address.              10
        mtsr    SW_PC, $8           // Set the switch PC.
        mtsri   SW_FREEZE, 0        // Get with switch running.
        jr      $31                 // Return.
.end setup_switch

        .swtext
        .align 3
// Start of switch code.
sw_start:
        j sw_start      route $csto->$cSo     // Everything goes south!          20
```

C.18 Stage3-Common.c

```
// Stage3-Common.c
//
//  Ken Taylor, MIT Master's Thesis 2004
//
// Last Updated: 5/18/2004
//
// This file implements the code used in common across stage3, which
//  does texture lookup and blending in the pipeline, and sends
//  textures and (mostly blended) fragments south to stage 4.
                                                                        10
#include "raw.h"
#include "raw_compiler_defs.h"
#include "render_datatypes.h"
#include "Common-sw.h" // flush/invalidate

#define gdn_send_hdr(F, l, u, oY, oX, dY, dX) \
  gdn_send(F<<29|l<<24|u<<20|oY<<15|oX<<10|dY<<5|dX)

void setup_switch(void);
                                                                        20



static inline void texwrap(float *coord, unsigned mode)
{
  signed intpart;
  // *coord: map down to 0->1 range
  // mode: (0=none,1=repeat, 2=mirror,3=clamp)

  intpart = (signed)(*coord);
                                                                        30
  if(mode == 1)
    {
      *coord = (*coord) - (float)intpart; // fractional part
      if(*coord < 0)
        *coord = 1 + *coord;
    }
  else if(mode == 2)
    {
      if(*coord < 0)
        *coord = -*coord;                                               40

      if( intpart % 2 ) // if it's odd, do a reverse mapping
        *coord = 1 - (*coord - (float)intpart);
      else
        *coord = *coord - (float)intpart;
```

```
    }

  // for none and clamp, leave as is.
}

void begin(void) {
  unsigned tileNum;
  TexManager *tm;
  PrimInfo pi;
  UntexFragment utf;
  Fragment fm;
  unsigned texrgba;
  unsigned dotex; // do texture mapping
  unsigned rgbaxlyl, rgbaxlyh, rgbaxhyl, rgbaxhyh; // 4 texture samples for bilinear
  signed ut,vt, ut1,vt1;
  TexEntry *pTEntry;
  unsigned invalidateTex; // do we need to invalidate tex mem (first prim after a flush)
  int i;


  tileNum = raw_get_abs_pos_x();
  // if we're tile 0 in the row
  if(tileNum == 0)
    {
      //// allocate texture structures, and flush them.
      tm = (TexManager*)malloc(sizeof(TexManager));
      tm->pTexMemory = (unsigned*)malloc(TEXMEMSIZE*sizeof(unsigned));
      tm->TexMemorySize = tm->TexMemoryFree= TEXMEMSIZE;
      tm->pTexEntryTable = (TexEntry*)malloc(TEXENTRIES*sizeof(TexEntry));
      tm->MaxTextures = TEXENTRIES;
      for(i = 0; i < TEXENTRIES; i++)
        {
          tm->pTexEntryTable[i].valid = 0;
          tm->pTexEntryTable[i].updated = 0;
          flush_variable(&tm->pTexEntryTable[i], sizeof(TexEntry));
        }
      tm->NumTextures = 0;
      tm->pAllocHead = 0; // the funny thing, is that the allocation list will be in
      tm->pAllocTail = 0; // stage 1's memory, though don't really need it.

      flush_variable(tm, sizeof(TexManager));

      //// wait for gdn message (from control tile)
      gdn_receive();

      //// send pointers to tex struct to other 3 tiles in this row
      gdn_send_hdr(0, 1, 0, 2, 0, 2, 1);
      gdn_send(tm);
```

```
    gdn_send_hdr(0, 1, 0, 2, 0, 2, 2);
    gdn_send(tm);
    gdn_send_hdr(0, 1, 0, 2, 0, 2, 3);
    gdn_send(tm);

    //// wait for 3responses back
    gdn_receive();                                                    100
    gdn_receive();
    gdn_receive();

    //// send pointers to texture structures back to control tile
    gdn_send_hdr(0, 1, 0, 2, 0, 0, 0);
    gdn_send(tm);
  }
else
  {
    //// tm = pointer read from gdn                                    110
    tm = (TexManager*)gdn_receive();

    //// send gdn message back to tile 0 in row
    gdn_send_hdr(0, 1, 0, 2, tileNum, 2, 0);
    gdn_send(0);
  }

// set up static network
setup_switch();
                                                                      120
invalidateTex = 1;

// loop forever!
while(1)
  {
    unsigned word;
    // note: static network needs flow control now, since it's both sending data
    //        from north to us and from us to south.
    // here's how it works: first word from north goes to us.
    //     (marker of prim vs flush or fragment vs endprim)               130
    // then it waits for a count of number of subsequent words
    //     to sent to us, and does so.
    //     flush: one, for proc #
    //     prim: size of prim info
    //     endprim: zero
    //     fragment: size of untextured fragment info
    // then it waits for a count of number of words to send south
    //     from us, and does so.
    //     flush: two: flush marker, and proc #
    //     prim: size of primary prim info + 1 for marker                 140
    //     endprim: one, the endprim marker
```

```
//     fragment: size of fragment info + 1 for marker
// and it repeats.

//// read byte from static network
word = static_receive();
if(word == RENDER_P_FLUSH)
  {
    //// if it's a flush

    // read proc #
    static_send(1);
    word = static_receive();

    //// send flush and next by(proc #) south
    static_send(2);
    static_send(RENDER_P_FLUSH);
    static_send(word);

    invalidateTex = 1;
  }
else
  {
    unsigned texinterp, textile, texalpha, colalpha;
    unsigned texmode, lit, useamb, usedir;
    unsigned wordsleft, *ptr;

    ////// read in pi
    wordsleft = sizeof(PrimInfo)/sizeof(unsigned);
    static_send(wordsleft);
    ptr = (unsigned*)&pi;

    for( ; wordsleft > 0; wordsleft--, ptr++)
      {
        (*ptr) = static_receive();
      }

    ////// send pi.p south
    wordsleft = sizeof(PrimaryPrimInfo)/sizeof(unsigned);
    static_send(wordsleft+1);
    ptr = (unsigned*)&pi.p;

    static_send(RENDER_P_PRIM);

    for( ; wordsleft > 0; wordsleft--, ptr++)
      {
        static_send(*ptr);
      }
```

```
      if(pi.p.Mode.draw)                                                          190
        {

          // unpack bitfields for better performance
          texinterp = pi.p.Mode.texinterp;
          textile = pi.p.Mode.textile;
          texalpha = pi.p.Mode.texalpha;
          colalpha = pi.p.Mode.colalpha;
          texmode = pi.p.Mode.texmode;
          lit = pi.p.Mode.lit;
          useamb = pi.p.Mode.useamb;                                              200
          usedir = pi.p.Mode.usedir;

          pTEntry = &tm->pTexEntryTable[pi.TextureID];
          invalidate_variable(pTEntry, sizeof(TexEntry));

          // see if we're actually doing texture mode
          dotex = 0;
          if (texmode >= 2 && pi.TextureID < tm->MaxTextures)
            {
              if(pTEntry->valid == 1)                                             210
                dotex = 1;
            }

          texrgba = 0;

#ifndef NOTEXCACHE
          if(dotex && pTEntry->updated & (1 << tileNum))
            {
#endif //NOTEXCACHE
              // invalidate the texture                                           220
              invalidate_variable(pTEntry->pBegin,
                                  pTEntry->Width * pTEntry->Height
                                  * sizeof(unsigned));

              pTEntry->updated &= ~(1 << tileNum);
              flush_variable(&pTEntry->updated, sizeof(unsigned));
#ifndef NOTEXCACHE
            }
#endif // NOTEXCACHE
                                                                                  230
        }

      while(1)
        {
          ////// read next byte. if it's a utf byte:
          if(static_receive() == RENDER_P_FRAG)
            {
```

```
//read in utf
wordsleft = sizeof(UntexFragment)/sizeof(unsigned);
static_send(wordsleft);                                          240
ptr = (unsigned*)&utf;

for( ; wordsleft > 0; wordsleft--, ptr++)
  {
    (*ptr) = static_receive();
  }

//copy x,y,z from utf to fm.

fm.x = utf.x;                                                    250
fm.y = utf.y;
fm.z = utf.z;

if(dotex)
  {
    magic_perf_texel();

    if(texinterp == 0)
      { // nearest neighbor
        signed tempu, tempv;                                     260
        //map u,v into 0-1 range, based on pi.p.Mode.textile
        // (0=none,1=repeat, 2=mirror,3=clamp)

        texwrap(&utf.u, textile);
        texwrap(&utf.v, textile);

        //scale up to texel index
        utf.u *= (float)pTEntry->Width;
        utf.v *= (float)pTEntry->Height;
                                                                 270
        // truncate u,v down to int.
        tempu = (signed)utf.u;
        tempv = (signed)utf.v;

        // do clamping
        if( textile == 3)
          {
            if(tempu < 0) tempu = 0;
            if(tempu >= pTEntry->Width)
              tempu = pTEntry->Width - 1;                        280
            if(tempv < 0) tempv = 0;
            if(tempv >= pTEntry->Width)
              tempv = pTEntry->Width - 1;
          }
```

```
    if(tempu >= 0 && tempv >= 0
       && tempu < pTEntry->Width
       && tempv < pTEntry->Height)
      texrgba = pTEntry->pBegin[tempu+
                          pTEntry->Width*tempv];                    290
    else
      texrgba = 0;

  }
else
  { // bilinear filtering

    //shift u,v by 0.5 texel, so "0" is centered at a texel
    utf.u -= 0.5/(float)pTEntry->Width;
    utf.v -= 0.5/(float)pTEntry->Height;                            300

    //map u,v into 0-1 range, based on pi.p.Mode.textile
    // (0=none,1=repeat, 2=mirror,3=clamp)
    texwrap(&utf.u, textile);
    texwrap(&utf.v, textile);

    //scale up to texel index
    utf.u *= (float)pTEntry->Width;
    utf.v *= (float)pTEntry->Height;
                                                                    310
    // do clamping
    if( textile == 3)
      {
        if(utf.u < 0.0f) utf.u = 0.0f;
        if(utf.u > (float)pTEntry->Width - 1.0f)
          utf.u = pTEntry->Width - 1;
        if(utf.v < 0.0f) utf.v = 0.0f;
        if(utf.v > (float)pTEntry->Height -  1.0f)
          utf.v = pTEntry->Height - 1;
      }                                                             320

    //truncate u,v down to ut,vt to get lower,
    // and add one to get upper (for rgbaxl/hyl/h)
    ut = (signed) utf.u;
    vt = (signed) utf.v;
    ut1 = ut+1;
    vt1 = vt+1;

    //in case ut1 or vt1 wraps around
    if(ut1 >= pTEntry->Width)                                       330
      {
        if(textile == 3) // clamp
          ut1 = ut;
```

```
      if(textile == 1) // repeat
        ut1 = 0;
      if(textile == 2) // mirror
        ut1 = (ut == 0) ? 0 : ut - 1;
    }
  if(vt1 >= pTEntry->Height)
    {                                                          340
      if(textile == 3) // clamp
        vt1 = vt;
      if(textile == 1) // repeat
        vt1 = 0;
      if(textile == 2) // mirror
        vt1 = (vt == 0) ? 0 : vt - 1;
    }

  if(ut >= 0 && vt >= 0 && ut < pTEntry->Width
     && vt < pTEntry->Height)                                  350
    rgbaxlyl = pTEntry->pBegin[ut+pTEntry->Width*vt];
  else
    rgbaxlyl = 0;

  if(ut >= 0 && vt1 >= 0 && ut < pTEntry->Width
     && vt1 < pTEntry->Height)
    rgbaxlyh = pTEntry->pBegin[ut+pTEntry->Width*vt1];
  else
    rgbaxlyh = 0;
                                                               360
  if(ut1 >= 0 && vt >= 0 && ut1 < pTEntry->Width
     && vt < pTEntry->Height)
    rgbaxhyl = pTEntry->pBegin[ut1+pTEntry->Width*vt];
  else
    rgbaxhyl = 0;

  if(ut1 >= 0 && vt1 >= 0 && ut1 < pTEntry->Width
     && vt1 < pTEntry->Height)
    rgbaxhyh = pTEntry->pBegin[ut1+pTEntry->Width*vt1];
  else                                                         370
    rgbaxhyh = 0;

  // blend between four corners
  texrgba = (((unsigned)((rgbaxlyl&0xFF)*
                         (1.0f-(utf.u-(float)ut)) +
                         (rgbaxhyl&0xFF)*
                         (utf.u-(float)ut)))&0xFF)
    |((((unsigned)(((rgbaxlyl>>8)&0xFF)*
                   (1.0f-(utf.u-(float)ut)) +
                   ((rgbaxhyl>>8)&0xFF)*                       380
                   (utf.u-(float)ut)))&0xFF)<<8)
```

```
            |((((unsigned)(((rgbaxlyl>>16)&0xFF)*
                              (1.0f-(utf.u-(float)ut)) +
                              ((rgbaxhyl>>16)&0xFF)*
                              (utf.u-(float)ut)))&0xFF)<<16)
            |(((unsigned)((rgbaxlyl>>24)*(1.0f-(utf.u-(float)ut)) +
                              (rgbaxhyl>>24)*(utf.u-(float)ut)))<<24);

          texrgba = (((unsigned)((texrgba&0xFF) *
                                  (1.0f-(utf.v-(float)vt)) +                          390
                                  ((rgbaxlyh&0xFF)*
                                   (1.0f-(utf.u-(float)ut)) +
                                   (rgbaxhyh&0xFF)*
                                   (utf.u-(float)ut)) * (utf.v-(float)vt)))&0xFF)
            | ((((unsigned)(((texrgba>>8)&0xFF) *
                              (1.0f-(utf.v-(float)vt)) +
                              (((rgbaxlyh>>8)&0xFF)*
                               (1.0f-(utf.u-(float)ut)) +
                               ((rgbaxhyh>>8)&0xFF)*
                               (utf.u-(float)ut)) * (utf.v-(float)vt)))&0xFF)<<8)   400
            | ((((unsigned)(((texrgba>>16)&0xFF) *
                              (1.0f-(utf.v-(float)vt)) +
                              (((rgbaxlyh>>16)&0xFF)*
                               (1.0f-(utf.u-(float)ut)) +
                               ((rgbaxhyh>>16)&0xFF)*
                               (utf.u-(float)ut)) * (utf.v-(float)vt)))&0xFF)<<16)
            | (((unsigned)((texrgba>>24) * (1.0f-(utf.v-(float)vt)) +
                              ((rgbaxlyh>>24)*(1.0f-(utf.u-(float)ut)) +
                               (rgbaxhyh>>24)*(utf.u-(float)ut)) *
                              (utf.v-(float)vt)))<<24);                              410
        }
    }


    // truncate alphas in texrgba and utf.rgba,
    // according to pi.p.Mode.texalpha and colalpha
    // and pi.alphaThresh

    if(texalpha == 0)
      texrgba |= 0x0FF;                                                              420
    else if(texalpha == 2)
      { // hard alpha
        if( (texrgba & 0x0FF) >= pi.alphaThresh )
          texrgba |= 0x0FF;
        else
          texrgba &= 0xFFFFFF00;
      }

    if(colalpha == 0)
```

```
    utf.rgba |= 0x0FF;                                                      430
  else if(colalpha == 2)
    { // hard alpha
      if( (utf.rgba & 0x0FF) >= pi.alphaThresh )
        utf.rgba |= 0x0FF;
      else
        utf.rgba  &= 0xFFFFFF00;
    }

  // sort of a hack - if we're in blend mode, and
  // either col or tex is hard alpha, and is 0x00 (under                    440
  // the threshold), then make the final alpha 0x00
  // this is so we can have a hard-alpha texture be
  // blended with a color map without creating a soft
  // alpha result. Note that such a prim will be out-of-order
  if(texmode == 3)
    {
      if(texalpha == 2 && (texrgba&0xFF) == 0)
        utf.rgba &= 0xFFFFFF00;
      if(colalpha == 2 && (utf.rgba&0xFF) == 0)
        texrgba &= 0xFFFFFF00;                                              450
    }


  // now blend texrgba and utf.rgba, according to pi.p.Mode.texmode
  //  (0=none, 1=color, 2=tex, 3=blend,
  //   4=texdecal, 5=coldecal, 6=modulate)
  //  and pi.ColTexBalance (0 = all tex, 1 = all color) -> utf.rgba

  switch(texmode)
    {                                                                       460
      unsigned alpha;
    case 1: // color only
      fm.rgba = utf.rgba;
      break;
    case 2: // texture only
      fm.rgba = texrgba;
      break;
    case 3: // col/tex blend
    case 4: // tex on top of color decal
    case 5: // color on top of tex decal                                    470
      switch(texmode)
        {
        case 3: // col/tex blend
          alpha = pi.ColTexBalance * 256;
          if(alpha >= 256) alpha = 255;
          break;
        case 4: // tex on top of color decal
```

```
      alpha = 255 - (texrgba & 0xFF);
      break;
    case 5: // color on top of tex decal                                480
      alpha = (utf.rgba & 0xFF);
      break;
    default: // this shouldn't happen
    }


  // fixed point modulation
  fm.rgba = ( ((((utf.rgba>>24) * alpha) + 255 )>>8) +
              ((((texrgba>>24) *
                 (255-alpha)) + 255)>>8)) << 24;                        490
  fm.rgba |= ( (((((utf.rgba<<8)>>24) * alpha) + 255 )>>8) +
              (((((texrgba<<8)>>24) *
                 (255-alpha)) + 255)>>8)) << 16;
  fm.rgba |= ( (((((utf.rgba<<16)>>24) * alpha) + 255 )>>8) +
              (((((texrgba<<16)>>24) *
                 (255-alpha)) + 255)>>8)) << 8;
  fm.rgba |= ( (((((utf.rgba<<24)>>24) * alpha) + 255 )>>8) +
              (((((texrgba<<24)>>24) *
                 (255-alpha)) + 255)>>8));
  break;                                                                500
case 6: // color/tex modulated
  // max of each is 255 - treated as 1.0 (fixed point modulation)
  fm.rgba = ((((utf.rgba>>24) *
              (texrgba>>24)) + 255 )>>8) << 24;
  fm.rgba |= (((((utf.rgba<<8)>>24) *
               ((texrgba<<8)>>24)) + 255 )>>8) << 16;
  fm.rgba |= (((((utf.rgba<<16)>>24) *
               ((texrgba<<16)>>24)) + 255 )>>8) << 8;
  fm.rgba |= ((((utf.rgba<<24)>>24) *
              ((texrgba<<24)>>24)) + 255 )>>8;                          510
  break;
case 0:
default:
  fm.rgba = 0x000000FF;
}

// finally, modulate output with light values, based on
//   pi.p.Mode.lit, useamb, usedir, pi.ambColor,
//   pi.dirColor, utf.intensity into fm.rgba.
if(lit)                                                                 520
  {
  unsigned temp, ambrgb=0, dirrgb=0, intens, tempr, tempg, tempb;


  //  calculate ambient component - light values times
```

```
//    intensity (i field) modulated with surface color
if(useamb)
  { // more fixed point modulation. fun!
    intens = (pi.ambColor & 0xFF);
    temp = ((((pi.ambColor>>24) *                                530
              intens) + 255 )>>8) << 24;
    temp |= (((((pi.ambColor<<8)>>24) *
               intens) + 255 )>>8) << 16;
    temp |= (((((pi.ambColor<<16)>>24) *
               intens) + 255 )>>8) << 8;
    ambrgb |= ((((temp>>24) *
                 (fm.rgba>>24)) + 255 )>>8) << 24;
    ambrgb |= (((((temp<<8)>>24) *
                 ((fm.rgba<<8)>>24)) + 255 )>>8) << 16;
    ambrgb |= (((((temp<<16)>>24) *                              540
                 ((fm.rgba<<16)>>24)) + 255 )>>8) << 8;
  }
//  calculate directional component - light values times
//    intensity  modulated with surface color

if(usedir)
  {
    intens = utf.intensity*256;
    if(intens > 255) intens = 255;
                                                                 550
    temp = ((((pi.dirColor>>24) *
              intens) + 255 )>>8) << 24;
    temp |= (((((pi.dirColor<<8)>>24) *
               intens) + 255 )>>8) << 16;
    temp |= (((((pi.dirColor<<16)>>24) *
               intens) + 255 )>>8) << 8;
    dirrgb |= ((((temp>>24) *
                 (fm.rgba>>24)) + 255 )>>8) << 24;
    dirrgb |= (((((temp<<8)>>24) *
                 ((fm.rgba<<8)>>24)) + 255 )>>8) << 16;          560
    dirrgb |= (((((temp<<16)>>24) *
                 ((fm.rgba<<16)>>24)) + 255 )>>8) << 8;
  }

//  saturate-add the two together.
//  only affects color values, alpha stays the same!

tempr = (ambrgb>>24) + (dirrgb>>24);
if(tempr > 255) tempr = 255;
tempg = ((ambrgb<<8)>>24) + ((dirrgb<<8)>>24);                   570
if(tempg > 255) tempg = 255;
tempb = ((ambrgb<<16)>>24) + ((dirrgb<<16)>>24);
if(tempb > 255) tempb = 255;
```

```
            fm.rgba &= 0x0FF;
            fm.rgba |= tempr << 24 | tempg << 16 | tempb << 8;
          }

        magic_perf_texfragment();
                                                                      580
        // stream fm south
        wordsleft = sizeof(Fragment)/sizeof(unsigned);
        static_send(wordsleft+1);
        ptr = (unsigned*)&fm;

        static_send(RENDER_P_FRAG);

        for( ; wordsleft > 0; wordsleft--, ptr++)
          {
            static_send(*ptr);                                        590
          }
      }
    else
      {
        //it's an end prim byte

        // we have nothing to read
        static_send(0);

        // send end prim byte south                                   600
        static_send(1);
        static_send(RENDER_P_ENDPRIM);
        break;
      }
    }
  }
  // end loop
}

}                                                                     610
```

C.19 Stage3-sw.S

```
// setup_switch from starsearch/examples/multi_tile/static_net/mixed/compute_sw.S

        .text
        .align 2
.global setup_switch

.ent setup_switch
setup_switch:
        mtsri   SW_FREEZE, 1        // Freeze the switch.
        la      $8, sw_start        // Get switch starting address.          10
        mtsr    SW_PC, $8           // Set the switch PC.
        mtsri   SW_FREEZE, 0        // Get with switch running.
        jr      $31                 // Return.
.end setup_switch

        .swtext
        .align 3
// Start of switch code.
sw_start:       nop         route $cNi->$csti      // send first word to proc
        move    $1, $csto                   // count of words to read         20
        BEQZD $1, $1, reply
readL: BNEZD $1, $1, readL route $cNi->$csti

reply:  move    $1, $csto                   // count of words in reply
        BEQZD $1, $1, sw_start
replyL: BNEZD $1, $1, replyL route $csto->$cSo

        j       sw_start
```

C.20 Stage4-Common.c

```
// Stage4-Common.c
//
//  Ken Taylor, MIT Master's Thesis 2004
//
// Last Updated: 5/18/2004
//
// This file implements the code used in common across stage4, which
//  take textured fragments, does z-buffer check if necessary,
//  and updates the framebuffer.
                                                                     10
#include "raw_compiler_defs.h"
#include "render_datatypes.h"
#include "ZBuf_datatypes.h"
#include "render_framebuffer.h"
#include "Common-sw.h" // flush/invalidate

// defined in .S file
extern volatile unsigned taketurn;
extern volatile unsigned seqnum;
extern volatile unsigned inthdr;                                     20

void setup_switch(void);
void setup_interrupts(void);

// TODO: this is common
#define gdn_send_hdr(F, l, u, oY, oX, dY, dX) \
  gdn_send(F<<29|l<<24|u<<20|oY<<15|oX<<10|dY<<5|dX)

// process a fragment, checking+ updating z buffer and updating fb as necessary
static inline void doFragment(Fragment *f, ZBufData *z,              30
                              unsigned nousez, unsigned nowritez)
{
  unsigned alpha;
  unsigned addr;
  unsigned temp1, temp;
  unsigned *zloc;

  addr = f->x+f->y*VWIDTH;

  if(!nousez)                                                        40
    {
      zloc = &z->buf[addr];
      invalidate_word(zloc);
      if( *zloc < f->z )
        {
```

```
      return;
    }
  }

alpha = f->rgba & 0xFF;                                                       50

if(alpha != 0x0FF)
  { // there's some alpha, read from framebuf

    // turn interrupt off
    raw_user_interrupts_off();

    if( z->fbmode & FBMODE_BACK )
      {
        temp1 = fb_read_pixel_rawaddr(addr, 0);                               60
        temp = ( ((((f->rgba>>24) * alpha) + 255 )>>8) +
               ((((temp1>>24) * (255-alpha)) + 255)>>8)) << 24;
        temp |= ( (((((f->rgba<<8)>>24) * alpha) + 255 )>>8) +
                 (((((temp1<<8)>>24) * (255-alpha)) + 255)>>8)) << 16;
        temp |= ( (((((f->rgba<<16)>>24) * alpha) + 255 )>>8) +
                 (((((temp1<<16)>>24) * (255-alpha)) + 255)>>8)) << 8;

        fb_set_pixel_rawaddr(addr, temp, 1, 0);
      }
                                                                              70
    if( z->fbmode & FBMODE_FRONT )
      {
        temp1 = fb_read_pixel_rawaddr(addr, 1);
        temp = ( ((((f->rgba>>24) * alpha) + 255 )>>8) +
               ((((temp1>>24) * (255-alpha)) + 255)>>8)) << 24;
        temp |= ( (((((f->rgba<<8)>>24) * alpha) + 255 )>>8) +
                 (((((temp1<<8)>>24) * (255-alpha)) + 255)>>8)) << 16;
        temp |= ( (((((f->rgba<<16)>>24) * alpha) + 255 )>>8) +
                 (((((temp1<<16)>>24) * (255-alpha)) + 255)>>8)) << 8;
                                                                              80
        fb_set_pixel_rawaddr(addr, temp, 0, 1);
      }


    // turn interrupt back on
    raw_user_interrupts_on();
  }
else
  {
    // note, optimizing with knowlege of exactly what                         90
    // FBMODE_BACK and _FRONT are... be careful!
    fb_set_pixel_rawaddr(addr, (f->rgba ^ alpha),
                         z->fbmode & FBMODE_BACK,
```

```
                    (z->fbmode & FBMODE_FRONT)>>1);
    }

  if(nowritez == 0)
    {
      if(nousez)
        zloc = &z->buf[addr];                                         100

      *zloc = f->z;
      flush_word(zloc);
    }
}

void begin(void) {

  ZBufData * zbd;
  PrimaryPrimInfo ppi;                                                110
  Fragment fm[FRAGBLOCKS];
  int tileNum;
  int i, j;
  unsigned invz = 1;
  unsigned myinthdr, myseqnum,mytaketurn; // used for non-volatile access
  unsigned mynousez, mynowritez; // isolate at prim level (bitfield values)

  taketurn = 0;
  myseqnum = seqnum = 0;
                                                                      120
  // set up static network
  setup_switch();

  tileNum = raw_get_abs_pos_x();
  // inthdr for sending token to next tile
  myinthdr = inthdr = 1<<24| 3<<15| tileNum<<10| 3<<5| (tileNum+1)%4;

  // set up interrupts
  setup_interrupts();
  raw_set_status_EX_MASK(0x00000000); // all off                     130
  raw_user_interrupts_on();
  raw_interrupts_on();

  // for framebuffer code
  fb_init_fbhdr(3, tileNum);

  // if we're tile 0 in the row
  if(tileNum == 0)
    {
      zbd = (ZBufData*)malloc(sizeof(ZBufData));                      140
```

```
//initialize zbd to zero
zbd->fbmode = FBMODE_BACK;

#ifdef INIT_ZBUF
 for(i = 0; i < VWIDTH*VHEIGHT; i++)
   {
     zbd->buf[i] = 0x7FFFFFFF;
   }
                                                                    150
 // flush zbd
 flush_variable(zbd, sizeof(ZBufData));

#else

 flush_variable(zbd->fbmode, sizeof(unsigned));
 flush_variable(zbd->buf, sizeof(unsigned));

#endif
                                                                    160
// wait for gdn message from main tile
gdn_receive();

//// initialize framebuffers to all black (optional step with compiler def?)
#ifdef INIT_FB_BLACK
// TODO
#endif

//// send pointers to zbd to other 3 tiles in row
gdn_send_hdr(0, 1, 0, 3, 0, 3, 1);                                  170
gdn_send(zbd);
gdn_send_hdr(0, 1, 0, 3, 0, 3, 2);
gdn_send(zbd);
gdn_send_hdr(0, 1, 0, 3, 0, 3, 3);
gdn_send(zbd);

//// wait for 3 responses back
gdn_receive();
gdn_receive();
gdn_receive();                                                      180

//// send pointer to zbd back to main tile.
gdn_send_hdr(0, 1, 0, 3, 0, 0, 0);
gdn_send(zbd);

//// turn on gdn_avail.
raw_set_status_EX_MASK(0x00000020);

//// send first round-robin token, seq # 1, to next tile.
```

```
    gdn_send_hdr(0, 1, 0, 3, 0, 3, 1);                                   190
    gdn_send(1);
  }
else
  {
    // read gdn from tile 0
    zbd = (ZBufData*)gdn_receive();

    //// turn on gdn_avail
    raw_set_status_EX_MASK(0x00000020);
                                                                         200
    //// send gdn message back to tile 0
    gdn_send_hdr(0, 1, 0, 3, tileNum, 3, 0);
    gdn_send(1);
  }

// (with gdn_avail on, can be interrupted. Interrupt should:
//// check extern shared variable to see if we want to take turn. if so,
////// place seq num in another extern shared variable and change
//////    reset variable to signal to main loop that you got the turn
//// if not, send token to next tile in loop.                           210

while(1)
  {
    unsigned word;

    // read word from static network
    word = static_receive();

    // if it's a flush byte
    if(word == RENDER_P_FLUSH)                                           220
      {
        // next word is tile num
        word = static_receive();

        //// send gdn message to tile that started flush
        raw_user_interrupts_off();
        gdn_send_hdr(0, 1, 0, 3, tileNum, 0, word);
        gdn_send(0);
        raw_user_interrupts_on();
                                                                         230
        // flush may mean z->fbmode was changed between frames. invalidate!
        invz = 1;
      }
    else
      {
        unsigned wordsleft, *ptr;
```

```
// read in ppi from sn
wordsleft = sizeof(PrimaryPrimInfo)/sizeof(unsigned);
ptr = (unsigned*)&ppi;                                              240

for( ; wordsleft > 0; wordsleft--, ptr++)
  {
    (*ptr) = static_receive();
  }

mynousez = ppi.Mode.nousez;
mynowritez = ppi.Mode.nowritez;

if(invz)                                                            250
  {
    invalidate_word(&zbd->fbmode);
    invz = 0;
  }

// read in next word from sn
word = static_receive();

// if not an end-of-prim marker
if( word == RENDER_P_FRAG )                                         260
  {
    if(!ppi.Mode.outoforder)
      {
        //// wait for token to be = seqnum
        magic_perf_startbusywait();
        taketurn = 1;
        do
          {
            while(taketurn);
            myseqnum = seqnum;                                      270
            mytaketurn = taketurn = (ppi.SeqNum != myseqnum);
            // if not our turn in the sequence
            if(mytaketurn)
              {
                gdn_send(myinthdr);
                gdn_send(myseqnum);
              }
          } while(mytaketurn);
        magic_perf_endbusywait();
                                                                    280
        //// do first fragment
        //// go through all fragments until endprim
        do
          {
            fm->x = static_receive();
```

```
          fm->y = static_receive();
          fm->z = static_receive();
          fm->rgba = static_receive();

          if(fm->rgba & 0x0FF)                                              290
            doFragment(fm, zbd, mynousez, mynowritez);

        } while( static_receive() == RENDER_P_FRAG );

      //// give up token to next stage (seqnum+1)
      gdn_send(myinthdr);
      gdn_send(myseqnum+1);

    }
  else //(unordered)                                                        300
    {
      unsigned endprim = 0; // did we get an endprim header word
      unsigned newfrag; // have we started to get the newfrag's
      //////////////////////// header word yet
      unsigned numfrags, i;
      while(!endprim)
        {
          // read next fm
          fm->x = static_receive();
          fm->y = static_receive();                                         310
          fm->z = static_receive();
          fm->rgba = static_receive();

          if(fm->rgba & 0x0FF)
            numfrags = 1;
          else
            numfrags = 0;

          newfrag = 0;
                                                                            320
          taketurn = 1;
          mytaketurn = 1;

          while(numfrags < FRAGBLOCKS)
            {
              // wait for something to appear on the SN
              magic_perf_startbusywait();
              while(!(raw_get_status_SW_BUF1() & 0x000000E0))
                {
                  if(!taketurn)                                             330
                    {
                      myseqnum = seqnum;
                      if(ppi.SeqNum > myseqnum || numfrags == 0)
```

```
          {
            gdn_send(myinthdr);
            gdn_send(myseqnum);

            taketurn = 1;
          }
        else                                                              340
          {
            mytaketurn = 0;
            break;
          }
      }
  }
magic_perf_endbusywait();

if(!mytaketurn)
  break;                                                                  350

newfrag = 1;

if(endprim = (static_receive() == RENDER_P_ENDPRIM))
  break;

// wait for prim data now
magic_perf_startbusywait();
while(!(raw_get_status_SW_BUF1() & 0x000000E0))
  {                                                                       360
    if(!taketurn)
      {
        myseqnum = seqnum;
        if(ppi.SeqNum > myseqnum || numfrags == 0)
          {
            gdn_send(myinthdr);
            gdn_send(myseqnum);

            taketurn = 1;
          }                                                               370
        else
          {
            mytaketurn = 0;
            break;
          }
      }
  }
magic_perf_endbusywait();

if(!mytaketurn)                                                           380
    break;
```

```
    fm[numfrags].x = static_receive();
    fm[numfrags].y = static_receive();
    fm[numfrags].z = static_receive();
    fm[numfrags].rgba = static_receive();

    newfrag = 0;

    if(fm[numfrags].rgba & 0x0FF)                                        390
      numfrags++;
  }

// in case we haven't got taketurn yet
magic_perf_startbusywait();
if(mytaketurn)
  {
    do
      {
        while(taketurn);                                                 400
        myseqnum = seqnum;
        mytaketurn = taketurn = (ppi.SeqNum > myseqnum);
        // if not our turn in the sequence
        if(mytaketurn)
          {
            gdn_send(myinthdr);
            gdn_send(myseqnum);
          }
      } while(mytaketurn);
  }                                                                      410
magic_perf_endbusywait();

// do it
for(i = 0 ; i < numfrags; i++)
  doFragment(&fm[i], zbd, mynousez, mynowritez);

// give up token
gdn_send(myinthdr);
if(endprim)
  {                                                                      420
    gdn_send(myseqnum+1);
  }
else
  gdn_send(myseqnum);

// need to read in next header if we didn't above
if(!newfrag)
  {
    endprim = (static_receive() == RENDER_P_ENDPRIM);
```

```
                    if(endprim)                                          430
                      {
                        // oops, prim ends! better catch the sequence # and
                        //  increment it
                        taketurn = 1;
                        while(taketurn);
                        gdn_send(myinthdr);
                        gdn_send(seqnum+1);
                      }
                  }
                                                                         440
              } // while(!endprim)
          } // else (ppi.Mode.outoforder)
        } // if (word == RENDER_P_FRAG)
      else
        {
          // empty prim, treat as unordered, increment seqnum
          // when it gets to our value or greater.
          magic_perf_startbusywait();
          taketurn = 1;
          do                                                             450
            {
              while(taketurn);
              myseqnum = seqnum;
              mytaketurn = taketurn = (ppi.SeqNum > myseqnum);
              // if not our turn in the sequence
              if(mytaketurn)
                {
                  gdn_send(myinthdr);
                  gdn_send(myseqnum);
                }                                                        460
            } while(mytaketurn);
          magic_perf_endbusywait();

          gdn_send(myinthdr);
          gdn_send(myseqnum+1);
        }
    } // else (word != RENDER_P_FLUSH)
  } // while(1)

//        remember to turn off gdnavail if expecting response from        470
//        fb, and to turn it on before sending token away.
//  - wait for token means set taketurn = 1, and wait for it to be 0.
//     then you can chek seqnum (token seqnum) against ppi.SeqNum
//  - give up token by sending gdn message to next tile modulo
//     total tiles in row (m), with seqnum as body of the token.
// NOTE: no method here to restart tokens at zero. not really needed.
//        when they wraparound, pipeline will be flushed, and things
```

```
//        will continue as normal. so don't reset
//        seqnum to 0 at top, either!
                                                              480
}
```

C.21 Stage4-sw.S

```
// setup_switch from starsearch/examples/multi_tile/static_net/mixed/compute_sw.S
// interrupt code inspired by starsearch/module_tests/interrupts/external/tests.S

        .text
        .align 2
.global setup_switch

.ent setup_switch
setup_switch:
        mtsri   SW_FREEZE, 1        // Freeze the switch.                     10
        la      $8, sw_start        // Get switch starting address.
        mtsr    SW_PC, $8           // Set the switch PC.
        mtsri   SW_FREEZE, 0        // Get with switch running.
        jr      $31                 // Return.
.end setup_switch

        .swtext
        .align 3
// Start of switch code.
sw_start:                                                                      20
        j sw_start      route $cNi->$csti       // Everything comes from north!



        .text
        .align 2

# interrupt vector
ivec:   j       HNDL_GDN_AVAIL
                                                                               30
# Copy ivec down to 0x50
        .global setup_interrupts
        .ent    setup_interrupts
setup_interrupts:
        addiu   $9, $0, %lo(ivec)
        aui     $9, $9, %hi(ivec)
        ilw     $12, 0($9)
        isw     $12, 0x50($0)
        jr      $31
        .end setup_interrupts                                                  40

// cache-free saving point for interrupt
.swtext
gdn_avail_save1:        .word 0
gdn_avail_save2:        .word 0
```

```
.text

HNDL_GDN_AVAIL:
        magc   $0, $0, 0xfed6  // start busywait                                  50
                               // (don't want spinning token to count as active)
        swsw   $2, %lo(gdn_avail_save1)($0)
        swsw   $3, %lo(gdn_avail_save2)($0)

        la     $2, taketurn
        lw     $3, 0($2)
        BEQ    $3, $0, hga_notaketurn
// they are taking the turn, place seq num in shared variable
// and reset taketurn
        sw     $0, 0($2)                                                          60
        addiu  $2, $2, 4 // taketurn+4 = seqnum
        sw     $cgni, 0($2)

        j      hga_done

hga_notaketurn:
// not taking the turn, send seqnum to next tile
        addiu  $2, $2, 8 // taketurn+8 = inthdr
        lw     $2, 0($2)
        addu   $cgno, $2, $0                                                      70
        addu   $cgno, $cgni, $0

hga_done:
        swlw   $2, %lo(gdn_avail_save1)($0)
        swlw   $3, %lo(gdn_avail_save2)($0)

        magc   $0, $0, 0xfed7 // end busywait
        dret

// a pointer to the shared memory, stored here so the assembly                   80
// can access it. make sure the C program or someone touches
// this point before turning on interrupts so that it doesn't
// cache miss... (though this doesn't guarantee no cache miss...
// maybe it won't be a problem)
.data
               .global taketurn
               .global seqnum
               .global inthdr
taketurn:      .word 0
seqnum:        .word 0                                                            90
inthdr:        .word 0
```

Appendix D

Verification Framework Code Listing

TODO: directory structure

D.1 RenderInterface.bc

```
include("<dev/basic.bc>");

if (LookupSymbolHash(gSymbolTable, "gMagicInstrHMS") == NULL)
  include("<dev/magic_instruction.bc>");

include("render_host.bc");
include("render_framebuffer.bc");

{
  local result;                                                          10

    result = dev_render_host_init(15);

    if (result == 0)
      exit(-1);

    result = dev_render_framebuffer_init(11);

    if (result == 0)
     exit(-1);                                                           20

}
```

D.2 render_framebuffer.bc

```
// render_framebuffer
//   device that interfaces from the RAW processor to a framebuffer/video DAC.
//   Can optionally record all RAW and framebuffer interactions for use in
//  testing a verilog drop-in for this module in the future. (can we use
//  PLI to put it in here directly? look into this...)
//   Also, can optionally display a constantly-updated image of what's
//  in the framebuffer at any point for debugging/verification/cool
//  purposes. (What good is a graphics card if you don't know what it's
//  rendering?
                                                                    10
// if i include this, it gives me "redefined" warnings
// if i don't, DrawLine is for some reason not linked when the reset
// routines are called. arrrgh!
include("<bug/graphics.bc>");

//
// dev_render_framebuffer_init
//

fn dev_render_framebuffer_init(ioPort)                              20
{
  local rfbStruct = hms_new();
  local result;

  rfbStruct.ioPort = ioPort;
  rfbStruct.fb_interface = hms_new();

  result = SimAddDevice("RenderFrameBuffer",
                        "dev_render_framebuffer_reset",
                        "dev_render_framebuffer_calc",              30
                        rfbStruct);

  if (result == 0)
  {
    printf("// **** render_framebuffer: failed to add device to port %d\n",
        ioPort);
    return 0;
  }

  result = dev_render_framebuffer_hw_init(rfbStruct.fb_interface);   40

  if(result == 0)
  {
    return 0;
  }
```

```
  // handle to the device
  return rfbStruct;
}                                                                    50

fn reset_fb_controller(rfbStruct)
{
  rfbStruct.fb_interface.PAGE = 1; // active page is visible page
  rfbStruct.fb_interface.nWE = 1;
  rfbStruct.fb_interface.nOE = 1;
  rfbStruct.fb_interface.nCS0 = 1;
  rfbStruct.fb_interface.nCS1 = 1;
}
                                                                     60
global gLastFBHW = 0;

NativeFunctionLink("atoi",1);

//
// dev_render_framebuffer_hw_init
//

fn dev_render_framebuffer_hw_init(fb_interface)
{                                                                    70
  local rfbhwStruct = hms_new();
  local result;

  gLastFBHW = rfbhwStruct;

  rfbhwStruct.fb_interface = fb_interface;

  rfbhwStruct.vheight = 480;
  rfbhwStruct.vwidth = 640;
                                                                     80


  arg_process(
    & fn(argv, foundArg)
    {
      rfbhwStruct.vheight = atoi(gArgv[foundArg+1]);
      printf("vheight set to %d\n", rfbhwStruct.vheight);
    },
    "-render_vheight");
                                                                     90
  arg_process(
    & fn(argv, foundArg)
    {
```

```
    rfbhwStruct.vwidth = atoi(gArgv[foundArg+1]);
    printf("vwidth set to %d\n", rfbhwStruct.vwidth);
  },
  "-render_vwidth");

rfbhwStruct.showdisplay = 0;
// 32 bit memory (24 used for r,g,b)                                    100
rfbhwStruct.fb0 = malloc(4 * rfbhwStruct.vwidth * rfbhwStruct.vheight);
rfbhwStruct.fb1 = malloc(4 * rfbhwStruct.vwidth * rfbhwStruct.vheight);

if(arg_scan("-render_showdisplay") != -1)
{
  rfbhwStruct.height = rfbhwStruct.vheight*2 + 1;
  rfbhwStruct.width = rfbhwStruct.vwidth;
  rfbhwStruct.display = wgxCreateWindow(rfbhwStruct.height,rfbhwStruct.width);
  if(rfbhwStruct.display != 0)
  {                                                                     110
    rfbhwStruct.displaydata = wgxReturnDrawingArea(rfbhwStruct.display);
    rfbhwStruct.showdisplay = 1;
    rfbhwStruct.realtimeupdate = 1;
  }
}

result = SimAddDevice("RenderFrameBufferHW",
                      "dev_render_framebuffer_hw_reset",
                      "dev_render_framebuffer_hw_calc",
                      rfbhwStruct);                                     120

if (result == 0)
{
  printf("// **** render_framebuffer_hw: failed to add device");
  return 0;
}

// handle to the device
return rfbhwStruct;
}                                                                       130

//
// dev_render_framebuffer_hw_reset
//

fn dev_render_framebuffer_hw_reset(rfbhwStruct)
{
  local i;
  local c;
                                                                        140
  // todo: set VSYNC with a timer
```

```
  rfbhwStruct.fb_interface.VSYNC = 1;

  // clear framebuffers
  for (i = 0; i < rfbhwStruct.vwidth * rfbhwStruct.vheight; i++)
  {
    rfbhwStruct.fb0[i] = 0x00000000;
    rfbhwStruct.fb1[i] = 0x00000000;
  }
                                                                150
  // set up the background
  if(rfbhwStruct.showdisplay == 1)
  {
    wgxSetForegroundColor(rfbhwStruct.display,
        wgxCreateColor(rfbhwStruct.display, 0x0000, 0x0000, 0x0000));
    wgxFillRectangle(rfbhwStruct.display, 0,0, rfbhwStruct.width, rfbhwStruct.height);
    c = wgxCreateColor(rfbhwStruct.display, 0xFFFF, 0x3333, 0xFFFF);
    wgxSetForegroundColor(rfbhwStruct.display, c);
    wgxDrawLine(rfbhwStruct.display, 0, rfbhwStruct.vheight,
                rfbhwStruct.width-1, rfbhwStruct.vheight);        160
    DrawLine(rfbhwStruct.displaydata, rfbhwStruct.height, rfbhwStruct.width,
            0, rfbhwStruct.width, rfbhwStruct.vheight, rfbhwStruct.vheight,
            c);
    wgxFlush(rfbhwStruct.display);
  }
}

//
// dev_render_framebuffer_hw_calc
//                                                              170

fn dev_render_framebuffer_hw_calc(rfbhwStruct)
{
  local i;
  local j;

  if(isatty(0)) // hacky way of not switching if there's no shunt. see default.bug
    switch_to_text();
  while(1)
  {                                                             180
      if(rfbhwStruct.fb_interface.nWE == 0)
      {
        //printf("hw write!\n");
        dev_render_framebuffer_hw_write(rfbhwStruct, rfbhwStruct.fb_interface.A,
                                        rfbhwStruct.fb_interface.D,
                                        !rfbhwStruct.fb_interface.nCS0,
                                        !rfbhwStruct.fb_interface.nCS1);
      }
      else if(rfbhwStruct.fb_interface.nOE == 0)
```

```
        {                                                                    190
          local page = -1;

          //printf("hw read!\n");
          if (rfbhwStruct.fb_interface.nCS0 == 0
              && rfbhwStruct.fb_interface.nCS1 == 1)
            page = 0;
          else if (rfbhwStruct.fb_interface.nCS1 == 0
                   && rfbhwStruct.fb_interface.nCS0 == 1)
            page = 1;
                                                                             200
          rfbhwStruct.fb_interface.D = dev_render_framebuffer_hw_read(rfbhwStruct,
            rfbhwStruct.fb_interface.A, page);
        }

        yield;
    }
}

//
// dev_render_framebuffer_hw_write                                           210
//

fn dev_render_framebuffer_hw_write(rfbhwStruct, address, data, p0, p1)
{
  local i;
  local j;
  local c;

  //printf("hw write: address: %05X data: %08X p0: %d p1: %d\n",
        address, data, p0, p1);                                              220

  if(rfbhwStruct.showdisplay == 1)
  {
    c = wgxCreateColor(rfbhwStruct.display, (data >> 24)<<8,
        ((data <<8) >> 24)<<8, ((data << 16) >> 24)<<8);
    j = address / rfbhwStruct.vwidth;
    i = address % rfbhwStruct.vwidth;
    //printf("hw write display: color: %08X i: %d j: %d\n", c, i, j);
  }
                                                                             230
  if(p0)
  {
    rfbhwStruct.fb0[address] = data;
    render_drawpoint(rfbhwStruct,i,j,c);
  }
  if(p1)
  {
```

```
    rfbhwStruct.fb1[address] = data;
    render_drawpoint(rfbhwStruct,i,j+rfbhwStruct.vheight+1,c);
  }                                                                   240

  if(rfbhwStruct.showdisplay == 1)
    wgxFlush(rfbhwStruct.display); // flush slows it down, but
                                   // lets it update properly on step.
}

//
// dev_render_framebuffer_hw_read
//
                                                                      250
fn dev_render_framebuffer_hw_read(rfbhwStruct, address, page)
{
  if(page == 0)
    return rfbhwStruct.fb0[address];
  else if(page == 1)
    return rfbhwStruct.fb1[address];
  else
    return 0xBAADBEEF;
}
                                                                      260
//
// render_drawpoint
//

// plots the point x,y in the realtime display (if it exists),
// both directly and to the back buffer.
fn render_drawpoint(rfbhwStruct,x,y,color)
{
  if(rfbhwStruct.showdisplay == 1)
  {                                                                   270
    if(rfbhwStruct.realtimeupdate == 1)
    {
      //printf("drawing point at x=%d,y=%d color %08X\n", x, y, color);
      wgxSetForegroundColor(rfbhwStruct.display, color);
      wgxDrawLine(rfbhwStruct.display, x,y,x,y);
    }
    *(rfbhwStruct.displaydata+((x+rfbhwStruct.width*y)<<2)) = color;
  }
}
                                                                      280
//
// render_realtimeupdate_on
//

fn render_realtimeupdate_on()
```

```
{
  gLastFBHW.realtimeupdate = 1;
}

//                                                                  290
// render_realtimeupdate_off
//

fn render_realtimeupdate_off()
{
  gLastFBHW.realtimeupdate = 0;
}

//
//  render_refresh_display                                          300
//

// to be called by user, to refresh the fb image from back buffer
// (slow!)
fn render_refresh_display()
{
  if(gLastFBHW != 0)
  {
    if(gLastFBHW.showdisplay == 1)
    {                                                               310
      wgxCommitImage(gLastFBHW.display);
    }
  }
}

//
// dev_render_framebuffer_reset
//

fn dev_render_framebuffer_reset(rfbStruct)                          320
{
  reset_fb_controller(rfbStruct);
}

//
// dev_render_framebuffer_calc
//

fn dev_render_framebuffer_calc(rfbStruct)
{                                                                   330
  local ioPort = rfbStruct.ioPort;
  local hdr;
  local dc; // don't care
```

```
local length;
local sY;
local sX;

local word;
local cmd;
local page;                                                                    340
local address;

local blength;

while(1)
{
      if(isatty(0)) // hacky way of not switching if there's no shunt. see default.bug

        switch_to_text();
      //printf("hello! and welcome to the framebuffer calc loop!\n");              350
      yield;

      // pull a command off the dynamic network
      hdr = threaded_general_io_receive(machine, ioPort);
      yield;
      DecodeDynHdr(hdr, &dc, &length, &dc, &sY, &sX, &dc, &dc);
      //printf("framebuffer: recieve header: %08X\n", hdr);
      //printf("framebuffer: recieve length: %d, sY: %d, sX: %d\n", length, sY, sX);

      if(length < 1)                                                           360
        continue; // ERROR!

      word = threaded_general_io_receive(machine, ioPort);
      yield;
      length--;

      cmd = word >> 29;
      page = (word << 11) >> 30;
      address = (word << 13) >> 13;
                                                                               370
      //printf("framebuffer: recieve word: %08X\n", word);
      //printf("framebuffer: recieve cmd: %01X page: %01X addr: %05X\n",
      //         cmd, page, address);

      if( cmd == 0b111 )
      {
        //printf("reset!\n");
        // reset
        reset_fb_controller(rfbStruct);
      }                                                                        380
      else if( cmd == 0b110 )
```

```
{
  //printf("reserved!\n");
  //reserved
}
else if( cmd & 0b100 )
{
  //printf("page flip!\n");
  // pageflip
  if( cmd & 0b001 )                                                    390
  {
    //printf("waiting for vsync\n");
    // wait for vsync
    while(!rfbStruct.fb_interface.VSYNC)
      yield;
  }

  rfbStruct.fb_interface.PAGE = !rfbStruct.fb_interface.PAGE;

  if( cmd & 0b001 )                                                    400
  {
    // send reply
    // send "word" back to tile
    hdr = ConstructDynHdr(0,1,0,0,0,sY,sX);
    threaded_general_io_send(machine,ioPort,hdr);
    yield;
    threaded_general_io_send(machine,ioPort,word);
    yield;
  }
}                                                                      410
else
{
  // read or write

  if( cmd & 0b010 )
  {
    //printf("block!\n");
    // block
    if( length < 1 )
      continue; // ERROR!                                              420

    blength = threaded_general_io_receive(machine, ioPort);
    yield;
    length--;
  }
  else
    blength = 1;

  if ( cmd & 0b001 )
```

```
{                                                                     430
  //printf("read!\n");
  // read
  hdr = ConstructDynHdr(0, blength, 0, 0, 0, sY, sX);
  threaded_general_io_send(machine, ioPort, hdr);

  rfbStruct.fb_interface.nWE = 1;
  rfbStruct.fb_interface.nOE = 0;

  // active page if specified, else assume other page.
  // can't read from both pages and can't read from none!            440

  if(    ((page & 0b01) && rfbStruct.fb_interface.PAGE == 0)
      || (!(page & 0b01) && rfbStruct.fb_interface.PAGE == 1))
  {
    rfbStruct.fb_interface.nCS1 = 1;
    rfbStruct.fb_interface.nCS0 = 0;
  }
  else
  {
    rfbStruct.fb_interface.nCS1 = 0;                                  450
    rfbStruct.fb_interface.nCS0 = 1;
  }

  for( ; blength > 0 ; blength-- )
  {
    rfbStruct.fb_interface.A = address;
    yield;
    threaded_general_io_send(machine, ioPort, rfbStruct.fb_interface.D);
    address++;
  }                                                                   460

  rfbStruct.fb_interface.nOE = 1;
  rfbStruct.fb_interface.nCS1 = 1;
  rfbStruct.fb_interface.nCS0 = 1;

  // note: deadlock possibility if tile accidentally sent
  // a too-long message, as we started sending a reply before
  // the whole message was drained. Well, this is the tile's fault.

}                                                                     470
else
{
  //printf("write!\n");
  // write

  rfbStruct.fb_interface.nCS1 =
    !(((page & 0b01) && rfbStruct.fb_interface.PAGE == 1) ||
```

```
          ((page & 0b10) && rfbStruct.fb_interface.PAGE == 0));

        rfbStruct.fb_interface.nCS0 =                                          480
          !(((page & 0b01) && rfbStruct.fb_interface.PAGE == 0) ||
            ((page & 0b10) && rfbStruct.fb_interface.PAGE == 1));

        // note: this timing may be naive. A more complete
        // implementation would settle address and data, then
        // strobe WE. Keeping it simple for now, but might
        // make better in the future. TODO.

        rfbStruct.fb_interface.nOE = 1;
                                                                               490
        for( ; blength > 0 && length > 0; blength--)
        {
          rfbStruct.fb_interface.A = address;
          rfbStruct.fb_interface.nWE = 1; // in case io_receive yields!

          rfbStruct.fb_interface.D = threaded_general_io_receive(machine, ioPort);
          length--;

          rfbStruct.fb_interface.nWE = 0;
                                                                               500
          if(grhost_perfDoPerf)
            grhost_perfNumPixels++;

          yield;
          address++;
        }

        rfbStruct.fb_interface.nWE = 1;
        rfbStruct.fb_interface.nCS1 = 1;
        rfbStruct.fb_interface.nCS0 = 1;                                       510
      }
    }


    // drain the rest of the message
    // (SHOULDN'T HAVE TO, BUT JUST IN CASE)
    for( ; length > 0 ; length-- )
    {
      //printf("WARNING extra word!\n");
      word = threaded_general_io_receive(machine, ioPort);                     520
      yield;
    }
  }
}
```

D.3 render_host.bc

```
// render_host
//   device that sends rendering commands to graphics processor
//  generally streams them in on static network
//   front-side interface left undefined, depends on host interface
//  commands piped in from a controlling process for debugging / simulation

NativeFunctionLink("pipe",1);

  // performance measurement info
global grhost_perfSceneStreamCycles = 0;                                          10
global grhost_perfStage1CyclesActive = 0; // only counted in scenestream these 4
global grhost_perfStage2CyclesActive = 0;
global grhost_perfStage3CyclesActive = 0;
global grhost_perfStage4CyclesActive = 0;
global grhost_perfRenderCycles = 0;
global grhost_perfNumFrames = 0;
global grhost_perfNumPrims = 0;
global grhost_perfNumPixels = 0; // get this data from the framebuffer
global grhost_perfNumFrags = 0; // get this through magic instructions
global grhost_perfNumTexFrags = 0; // get this through magic instructions         20
global grhost_perfNumTexels = 0; // also through magic instructions
global grhost_perfNumDrawnPrims = 0; // magic instr.

global grhost_perfDoPerf = 0;  // count rendercycles
global grhost_perfSceneStream = 0; // count scenestream + active %'s
global grhost_perfBusyWait;
global grhost_perfIntBusyWait; // busywait to use in interrupts.


                                                                                  30
//
// dev_render_host_init
//

fn dev_render_host_init(ioPort)
{
  local rhostStruct = hms_new();
  local result;
  local i;
                                                                                  40
  rhostStruct.ioPort = ioPort;

  result = SimAddDevice("RenderHost",
                        "dev_render_host_reset",
                        "dev_render_host_calc",
```

```
                    rhostStruct);

if (result == 0)
{
  printf("// **** render_host: failed to add device to port %d\n", ioPort);       50
  return 0;
}

grhost_perfBusyWait = malloc(4*16);
grhost_perfIntBusyWait = malloc(4*16);

dev_render_perf_reset(0);

// these two state variables shouldn't be reset on the general
// perf reset command                                                                60
grhost_perfSceneStream = 0;
for(i = 0; i < 16; i++)
{
  grhost_perfBusyWait[i] = 0;
  grhost_perfIntBusyWait[i] = 0;
}


result = SimAddDevice("RenderProfiler",
                    "dev_render_perf_reset",                                        70
                    "dev_render_perf_calc",
                    0);

if (result == 0)
{
  printf("// **** render_host: failed to add device to port %d\n", ioPort);
  return 0;
}
                                                                                    80
rhostStruct.usingpipe = 0;
rhostStruct.outofdata = 0;

arg_process(
  & fn(argv, foundArg)
  {
    rhostStruct.usingpipe = 1;
    rhostStruct.piper = malloc(2*4);
    rhostStruct.pipew = malloc(2*4);
                                                                                    90
    // pipe[0] is reading, pipe[1] is writing
    pipe(rhostStruct.piper);
    pipe(rhostStruct.pipew);
```

```
    if ((rhostStruct.pid = fork()) == 0)
    {
      close(rhostStruct.pipew[1]);
      dup2(rhostStruct.pipew[0], 0); // pipew is new stdin
      close(rhostStruct.piper[0]);                                        100
      dup2(rhostStruct.piper[1], 1); // piper is new stdout

      // so ctrl-C doesn't kill child. see btl/system/util.bc:shell
      setpgrp();

      execlp("/bin/sh","/bin/sh", "-c",gArgv[foundArg+1],0);
    }
    else
    {
      close(rhostStruct.piper[1]);                                        110
      close(rhostStruct.pipew[0]);
      rhostStruct.pipewf = fdopen(rhostStruct.pipew[1],"a");
    }

    // F_SETFL = 4
    // O_NONBLOCK = 04000
    fcntl(rhostStruct.piper[0], 4, 04000);

    //printf("// started render_host client: %s\n", gArgv[foundArg+1]);
  },                                                                      120
  "-render_hostcmd");

// add magc instruction handler
// rs = 0,
// imm = fed0 - start busy waiting (nothing getting done)
//       fed1 - end busy waiting (doing work again)
//       fed2 - count a fragment
//       fed3 - count a textured fragment
//       fed4 - count a texel
//       fed5 - count a drawn primitive (nonclipped)                      130
//       fed6 - start busy waiting in an interrupt
//       fed7 - end busy waiting in an interrupt
listi_add(gMagicInstrHMS.theList,
        & fn(procNum, rs, imm, result_ptr)
          {
            if (rs == 0)
            {
              switch (imm)
              {
              case 0xfed0:                                                140
                grhost_perfBusyWait[procNum] = 1;
```

```
                return 1;
              case 0xfed1:
                grhost_perfBusyWait[procNum] = 0;
                return 1;
              case 0xfed2:
                if(grhost_perfDoPerf)
                  grhost_perfNumFrags++;
                return 1;
              case 0xfed3:                                              150
                if(grhost_perfDoPerf)
                  grhost_perfNumTexFrags++;
                return 1;
              case 0xfed4:
                if(grhost_perfDoPerf)
                  grhost_perfNumTexels++;
                return 1;
              case 0xfed5:
                if(grhost_perfDoPerf)
                  grhost_perfNumDrawnPrims++;                           160
                return 1;
              case 0xfed6:
                grhost_perfIntBusyWait[procNum] = 1;
                return 1;
              case 0xfed7:
                grhost_perfIntBusyWait[procNum] = 0;
                return 1;

              default:
                return 0;                                               170

              }
            }

            return 0;
          }
    );

  // handle to the device
  return rhostStruct;                                                   180
}

fn dev_render_perf_reset(dummy)
{
  local i;

  // performance measurement info
  grhost_perfSceneStreamCycles = 0;
  grhost_perfStage1CyclesActive = 0; // only counted in scenestream these 4
```

```
  grhost_perfStage2CyclesActive = 0;                                                190
  grhost_perfStage3CyclesActive = 0;
  grhost_perfStage4CyclesActive = 0;
  grhost_perfRenderCycles = 0;
  grhost_perfNumFrames = 0;
  grhost_perfNumPrims = 0;
  grhost_perfNumPixels = 0; // get this data from the framebuffer
  grhost_perfNumFrags = 0; // how to get this? magic instructions?
  grhost_perfNumTexFrags = 0; // get this through magic instructions
  grhost_perfNumTexels = 0;
  grhost_perfNumDrawnPrims = 0; // magic instr.                                     200
  grhost_perfDoPerf = 0;  // count rendercycles
}


// cycles counting up performance data in global variables
fn dev_render_perf_calc(dummy)
{
        while(1)
        {
                if(grhost_perfDoPerf)                                               210
                {
                  grhost_perfRenderCycles++;
                  if(grhost_perfSceneStream)
                  {
                    grhost_perfSceneStreamCycles++;
                    grhost_perfStage1CyclesActive +=
                        ((Proc_GetStallReason(Machine_GetProc(machine,0)) == 0)
                         &&(!(grhost_perfBusyWait[0]||grhost_perfIntBusyWait[0]))) +
                        ((Proc_GetStallReason(Machine_GetProc(machine,1)) == 0)
                         &&(!(grhost_perfBusyWait[1]||grhost_perfIntBusyWait[1]))) +   220
                        ((Proc_GetStallReason(Machine_GetProc(machine,2)) == 0)
                         &&(!(grhost_perfBusyWait[2]||grhost_perfIntBusyWait[2]))) +
                        ((Proc_GetStallReason(Machine_GetProc(machine,3)) == 0)
                         &&(!(grhost_perfBusyWait[3]||grhost_perfIntBusyWait[3])));
                    grhost_perfStage2CyclesActive +=
                        ((Proc_GetStallReason(Machine_GetProc(machine,4)) == 0)
                         &&(!(grhost_perfBusyWait[4]||grhost_perfIntBusyWait[4]))) +
                        ((Proc_GetStallReason(Machine_GetProc(machine,5)) == 0)
                         &&(!(grhost_perfBusyWait[5]||grhost_perfIntBusyWait[5]))) +
                        ((Proc_GetStallReason(Machine_GetProc(machine,6)) == 0)      230
                         &&(!(grhost_perfBusyWait[6]||grhost_perfIntBusyWait[6]))) +
                        ((Proc_GetStallReason(Machine_GetProc(machine,7)) == 0)
                         &&(!(grhost_perfBusyWait[7]||grhost_perfIntBusyWait[7])));
                    grhost_perfStage3CyclesActive +=
                        ((Proc_GetStallReason(Machine_GetProc(machine,8)) == 0)
                         &&(!(grhost_perfBusyWait[8]||grhost_perfIntBusyWait[8]))) +
                        ((Proc_GetStallReason(Machine_GetProc(machine,9)) == 0)
```

```
                                &&(!(grhost_perfBusyWait[9]||grhost_perfIntBusyWait[9]))) +
                              ((Proc_GetStallReason(Machine_GetProc(machine,10)) == 0)
                                &&(!(grhost_perfBusyWait[10]||grhost_perfIntBusyWait[10]))) +     240
                              ((Proc_GetStallReason(Machine_GetProc(machine,11)) == 0)
                                &&(!(grhost_perfBusyWait[11]||grhost_perfIntBusyWait[11])));
                          grhost_perfStage4CyclesActive +=
                              ((Proc_GetStallReason(Machine_GetProc(machine,12)) == 0)
                                &&(!(grhost_perfBusyWait[12]||grhost_perfIntBusyWait[12]))) +
                              ((Proc_GetStallReason(Machine_GetProc(machine,13)) == 0)
                                &&(!(grhost_perfBusyWait[13]||grhost_perfIntBusyWait[13]))) +
                              ((Proc_GetStallReason(Machine_GetProc(machine,14)) == 0)
                                &&(!(grhost_perfBusyWait[14]||grhost_perfIntBusyWait[14]))) +
                              ((Proc_GetStallReason(Machine_GetProc(machine,15)) == 0)            250
                                &&(!(grhost_perfBusyWait[15]||grhost_perfIntBusyWait[15])));
                      }
                  }
                  yield;
          }
}


//
// dev_render_host_reset                                                                        260
//
//

fn dev_render_host_reset(rhostStruct)
{
        // we really should close the process here and restart it
}

NativeFunctionLink("strtoul", 3);
NativeFunctionLink("fdopen", 2);                                                                270




//
// dev_render_host_readfromhost(rhostStruct, buf)
//
// reads 11 bytes (next command) in from host and stores it in buf

fn dev_render_host_readfromhost(rhostStruct, buf)
{                                                                                               280
  local numread = 0;
  local nr = 0;

  while(numread < 11)
  {
```

```
    nr = read(rhostStruct.piper[0], buf+numread, 11−numread);
    if(nr == −1)
      {
        yield;
        continue; // assuming EAGAIN is only error. Watch out for lockups!      290
      }
    else
      numread += nr;

    if(nr == 0)
    {
      printf("//**** render_host: out of data\n");
      rhostStruct.outofdata = 1;
      break;
    }                                                                          300
}


}

//
// render_report_profiling()
//

fn render_report_profiling()                                                   310
{
  printf("RenderCycles: %d StreamCycles: %d\n", grhost_perfRenderCycles,
        grhost_perfSceneStreamCycles);
  printf("ActiveCycles*4 Stage 1: %d, 2: %d, 3: %d, 4: %d\n",
         grhost_perfStage1CyclesActive,
         grhost_perfStage2CyclesActive,  grhost_perfStage3CyclesActive,
         grhost_perfStage4CyclesActive);

  printf("Frames: %d, Prims: %d, DrawnPrims: %d,\nFrags: %d, \
TexFrags: %d, Texels: %d, Pixels: %d\n",                                       320
        grhost_perfNumFrames, grhost_perfNumPrims, grhost_perfNumDrawnPrims,
        grhost_perfNumFrags, grhost_perfNumTexFrags, grhost_perfNumTexels,
        grhost_perfNumPixels);

}

//
// dev_render_host_calc
//
                                                                               330
fn dev_render_host_calc(rhostStruct)
{
  local ioPort = rhostStruct.ioPort;
```

```
local buf = malloc(11);
local status;


// wait for a sn message to come in, to know the
// processor is ready.
threaded_static_io_receive(machine, ioPort);
if(isatty(0)) // hacky way of not switching if there's no shunt. see default.bug
  switch_to_text();

while(1)
{
      if(rhostStruct.usingpipe)
      {
        local data;
        dev_render_host_readfromhost(rhostStruct, buf);
        if( !rhostStruct.outofdata )
        {
          //printf("// render_host: read \"%.10s\" \n", buf);

          if(strncmp(buf, "wait", 4) == 0)
          { // wait for a word response
            threaded_static_io_receive(machine, ioPort);
          }
          else if(strncmp(buf, "halt", 4) == 0)
          { // pause the simulation
            gInterrupted = 1;
          }
          else if(strncmp(buf, "read", 4) == 0)
          { // send data to program from static network
            local bytestosend = 0;
            local j;

            //printf("// render_host requests read\n");

            dev_render_host_readfromhost(rhostStruct, buf);

            if(!rhostStruct.outofdata)
            {
              bytestosend = strtoul(buf, 0, 16);
              //printf("// bytes to send: %d\n", bytestosend);

              for(j = 0; j < bytestosend; j++)
              {
                data = threaded_static_io_receive(machine, ioPort);
                //printf("// data: 0x%08x\n", data);
                fprintf(rhostStruct.pipewf, "0x%08x\n", data);
                fflush(rhostStruct.pipewf);
```

```
      }

    }
  }
  else if(strncmp(buf, "debug", 5) == 0)
  { // print out a debug string
    dev_render_host_readfromhost(rhostStruct, buf);
    if(!rhostStruct.outofdata)
    {                                                                    390
      // (don't comment out this printf!)
      printf("// render_host: DEBUG: %.11s", buf);
    }

  }
  else if(strncmp(buf, "pstart", 6) == 0)
  { // start profiling
      grhost_perfDoPerf = 1;
  }
  else if(strncmp(buf, "pstop", 5) == 0)                                 400
  { // stop profiling
      grhost_perfDoPerf = 0;
  }
  else if(strncmp(buf, "penterss", 8) == 0)
  { // enter scene stream
      grhost_perfSceneStream = 1;
  }
  else if(strncmp(buf, "pexitss", 7) == 0)
  { // exit scene stream
      grhost_perfSceneStream = 0;                                        410
  }
  else if(strncmp(buf, "pframe", 6) == 0)
  { // mark the beginning of a frame
      if(grhost_perfDoPerf)
        grhost_perfNumFrames++;
  }
  else if(strncmp(buf, "pprim", 5) == 0)
  { // mark the beginning of a prim
      if(grhost_perfDoPerf)
        grhost_perfNumPrims++;                                           420
  }
  else if(strncmp(buf, "preport", 7) == 0)
  { // report profiling info so far
      render_report_profiling();
  }
  else if(strncmp(buf, "preset", 6) == 0)
  { // reset profiling info
      dev_render_perf_reset(0);
  }
```

```
        else                                                                    430
        { // send data to proc
          // data comes in as 32-bit numbers in hex format
          data = strtoul(buf, 0, 16);
          //printf("// render_host: read \"%.10s\" sending %08X\n", buf, data);

          threaded_static_io_send(machine, ioPort, data);
        }
      }
    }
                                                                                440
    if(rhostStruct.outofdata)
      break;

    yield;
  }

  // shouldn't get down here unless the feeder process died
  NativeFunctionLink("waitpid", 3);

  waitpid(rhostStruct.pid, &status, 0);                                         450
  printf("// render_host: exit status of child: %d (%08X)\n",
      (status & 0xFF00) >> 8, status);
}
```

D.4 render_framebuffer.h

```
// render_framebuffer.h
//
// Ken Taylor, MIT Master's Thesis 2004
//
// Last Updated: 5/28/2004
//
// contains functions to interact with framebuffer. used by
// stage1 in command mode and stage4 in scenestream

#ifndef RENDER_FRAMEBUFFER_H                                         10
#define RENDER_FRAMEBUFFER_H

#include "raw_compiler_defs.h"

unsigned fbhdr;
unsigned fbhdr_1;
unsigned fbhdr_2;

#define gdn_send_or(var1, var2)     __rgcc_two_input("or $cgno, %0, %1", var1, var2)
                                                                     20
// init fbhdr values for fast gdn sending
void fb_init_fbhdr(unsigned sendery, unsigned senderx)
{
  //      funny sendery senderx   desty
  fbhdr = 3<<29|(sendery&0x1F)<<15|(senderx&0x1F)<<10|3<<5;
  fbhdr_1 = fbhdr | 1<<24;
  fbhdr_2 = fbhdr | 2<<24;
}

static inline void fb_set_pixel_rawaddr(unsigned addr, unsigned rgbx, int bp, int ap)    30
{
  unsigned cmda, cmdb;

  // 0b000 command for write single
  cmda = (bp << 20) | (ap << 19);
  cmdb = (addr&0x07FFFF);

  // note: these functions take 2 cycles to send. For
  // higher performance, put in assembly later?
    // (or maybe release build with optimization would be faster?       40

  /*print_string("SETPIXELRGBX***");
  print_hex(fbhdr_2);
  print_hex(cmd);
  print_hex(rgbx);*/
```

```
  gdn_send(fbhdr_2);
  gdn_send_or(cmda, cmdb);
  gdn_send(rgbx);

}

// set a pixel in the framebuffer
static inline void fb_set_pixel_rgbx(int x, int y, unsigned rgbx, int bp, int ap)
{
  fb_set_pixel_rawaddr(y*VWIDTH+x, rgbx, bp, ap);
}

static inline void fb_set_pixel(int x, int y, unsigned r, unsigned g,
                                unsigned b, int bp, int ap)
{
  unsigned data;

  // r, g, b, unused
  data = (r << 24) | ((g << 24) >> 8) | ((b << 24) >> 16);

  fb_set_pixel_rgbx(x, y, data, bp, ap);
}


// send a flip page command to framebuffer (no wait for vsync)
static inline void fb_flip_page()
{
  gdn_send(fbhdr_1);
  gdn_send(0x80000000);
}

// wait for vsync
static inline void fb_flip_page_vsync()
{
  unsigned temp;

  gdn_send(fbhdr_1);
  gdn_send(0xA0000000);

  temp = 0;

  // uhm... hope you're not expecting any other GDN messages!
  while( temp != 0xA0000000 )
    temp = gdn_receive();
}

// send a reset command to framebuffer
```

```
static inline void fb_reset()
{
  gdn_send(fbhdr_1);
  gdn_send(0xE0000000);
}

static inline void fb_set_pixel_block_rawaddr(unsigned addr, unsigned *bytes,            100
                                              unsigned length, int bp, int ap)
{
  unsigned cmda, cmdb;
  int i;

  cmda = (0x2 << 29) | (bp << 20) | (ap << 19);
  cmdb =  (addr&0x07FFFF);

  gdn_send(fbhdr|((length+2)<<24));
  gdn_send_or(cmda,cmdb);                                                                  110
  gdn_send(length);
  for(i=0; i<length; i++)
    gdn_send(bytes[i]);
}

// set a block of pixels
static inline void fb_set_pixel_block(int x, int y,  unsigned *bytes,
                                      unsigned length, int bp, int ap)
{
  fb_set_pixel_block_rawaddr(y*VWIDTH+x, bytes, length, bp, ap);                           120
}

// read a pixel
// page = 0 for back, =1 for active
static inline unsigned fb_read_pixel_rawaddr(unsigned addr, int page)
{
  unsigned cmda, cmdb;
  cmda = (0x1 << 29) | ((!page) << 20) | (page << 19);
  cmdb =(addr&0x07FFFF);
                                                                                           130
  gdn_send(fbhdr_1);
  gdn_send_or(cmda, cmdb);
  return gdn_receive(); // hope you're not expecting any other gdn messages!
}

static inline unsigned fb_read_pixel(int x, int y, int page)
{
  return fb_read_pixel_rawaddr(y*VWIDTH+x, page);
}
                                                                                           140
static inline void fb_read_pixel_block_rawaddr(unsigned addr, unsigned *bytes,
```

```
                                        unsigned length, int page)
{
  unsigned cmda, cmdb;
  int i;

  cmda = (0x3 << 29) | ((!page) << 20) | (page << 19);
  cmdb = (addr&0x07FFFF);

  gdn_send(fbhdr_2);                                                    150
  gdn_send_or(cmda, cmdb);
  gdn_send(length);

  for(i = 0; i < length; i++)
    bytes[i] = gdn_receive();
}

static inline void fb_read_pixel_block(int x, int y, unsigned *bytes,
                                   unsigned length, int page)
{                                                                       160
  fb_read_pixel_block_rawaddr(y*VWIDTH+x, bytes, length, page);
}

#endif // RENDER_FRAMEBUFFER_H
```

D.5 render_client.h

```
// render_client.h
// Ken Taylor Master's Thesis 2004
// Last Updated: 5/18/04
//
// This file provides user-friendly functions to interact with the
// RAW rendering processor. Right now, they just output hex values
// to stdout, which are to be piped into render_host.bc by setting the
// -render_hostcmd btl argument to the name of the executable. But
// they could possibly be made into some sort of library interface in the
// future if the render slave is ever implemented in real hardware.
// Another possibility is wrapping OpenGL to call these functions, so
// OpenGL apps could be run.

#ifndef RENDER_CLIENT_H
#define RENDER_CLIENT_H

#include "../render_cmds.h"
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <stdlib.h>
#include <errno.h>
#include <math.h>

// helper functions for sending raw values
void RSendPVal(void *val)
{
  //printf("RSendPVal\n");
  printf("0x%08x\n", *((unsigned *)val));
}

void RSendUnsigned(unsigned val)
{
  //printf("RSendUnsigned\n");
  RSendPVal(&val);
}

unsigned RReadUnsigned()
{
  int numread = 0, nr = 0;
  char buf[11];

  while(numread < 11)
    {
      fflush(stdout);
```

```
      nr = read(STDIN_FILENO, &buf[numread], 11-numread);
      if(nr == -1)
        {
          fprintf(stderr, "ERRNO = %d\n", errno);
          break;                                                   50
        }
      else
        numread +=nr;

      if(nr == 0)
        return 0;
    }

  return strtoul(buf, 0, 16);
}                                                                  60

void RSendSigned(signed val)
{
  //printf("RSendSigned\n");
  RSendPVal(&val);
}

signed RReadSigned()
{
  unsigned temp;                                                   70
  temp = RReadUnsigned();
  return *((signed *)&temp);
}

void RSendFloat(float val)
{
  //printf("RSendFloat\n");
  RSendPVal(&val);
}
                                                                   80
float RReadFloat()
{
  unsigned temp;
  temp = RReadUnsigned();
  return *((float *)&temp);
}

void RSendMeta(const char *val)
{
  char send[12];                                                   90
  int i;

  for( i = 0; i < 10; i++)
```

```
    send[i] = '*';

  send[10] = '\n';
  send[11] = '\0';

  strncpy( send, val, ((strlen(val)>11) ? 11 : strlen(val)) );
                                                                 100
  printf(send);
}


// RMETA commands are for the bc code and not sent to the
// Raw processor
void RMETAHalt()
{
  RSendMeta("halt");
}                                                                110

void RMETAWait()
{
  RSendMeta("wait");
}

void RMETARead(unsigned num)
{
  RSendMeta("read");
  RSendUnsigned(num);                                            120
}

void RMETADebug()
{
  RSendMeta("debug");
}


// token to know if we're in scenestream of not, to know how
// to do flow control                                            130

unsigned g_RInSceneStream;

// camera and perspective matrices, stored as state so user only
// has to change one at a time

float g_RPerspective[4][4]; // eyespace -> canonical coords
float g_RCamera[4][4]; // worldspace -> eyespace

// standard client commands                                      140
void RInit()
```

```
{
  int i,j;
  g_RInSceneStream = 0;
  for(i = 0; i < 4; i++)
    for(j = 0; j < 4; j++)
      {
        if(i == j)
          g_RPerspective[i][j] = g_RCamera[i][j] = 1;
        else                                                        150
          g_RPerspective[i][j] = g_RCamera[i][j] = 0;
      }

}

void RBeginScene()
{
  if(!g_RInSceneStream)
    {
      RSendUnsigned(RENDER_BEGINSCENE);                             160
      RMETAWait();
      RSendMeta("penterss");
      g_RInSceneStream = 1;
    }
}

void RBeginPrim()
{
  if(g_RInSceneStream)
    {                                                               170
      RSendMeta("pprim");
    }
}

void REndPrim()
{
  if(g_RInSceneStream)
    {
      RSendUnsigned(0);
    }                                                               180
}

void REndScene()
{
  if(g_RInSceneStream)
    {
      RSendUnsigned(1);
      RSendUnsigned(RENDER_ENDSCENE);
      REndPrim();
```

```
      RMETAWait();                                                   190
      RSendMeta("pexitss");
      g_RInSceneStream = 0;
    }
}

void RVertex(float x, float y, float z, float u, float v)
{
  if(g_RInSceneStream)
    {
      RSendUnsigned(6);                                              200
      RSendUnsigned(RENDER_VERTEX);
      RSendFloat(x);
      RSendFloat(y);
      RSendFloat(z);
      RSendFloat(u);
      RSendFloat(v);
    }
}

void RColorRGBA(unsigned rgba)                                       210
{
  if(g_RInSceneStream)
    RSendUnsigned(2);
  RSendUnsigned(RENDER_COLOR);
  RSendUnsigned(rgba);
}

void RColor(unsigned R, unsigned G, unsigned B, unsigned A)
{
  RColorRGBA( (R&0x0FF)<<24|(G&0x0FF)<<16|(B&0x0FF)<<8|(A&0x0FF) );  220
}

// TODO: software-based pushmatrix/popmatrix functionality?

void RModelMatrix(float M[4][4])
{
  int i,j;

  if(g_RInSceneStream)
    RSendUnsigned(17);                                               230

  RSendUnsigned(RENDER_MODELMATRIX);

  for(i = 0; i<4 ; i++)
    for(j = 0; j<4 ; j++)
      RSendFloat(M[i][j]);
}
```

```
void RViewMatrix(float M[4][4])
{                                                                    240
  int i,j;

  if(g_RInSceneStream)
    RSendUnsigned(17);

  RSendUnsigned(RENDER_VIEWMATRIX);

  for(i = 0; i<4 ; i++)
    for(j = 0; j<4 ; j++)
      RSendFloat(M[i][j]);                                           250
}

// matrix to matrix multiply! X = M*Y
void MatrixMult(float X[4][4], float M[4][4], float Y[4][4])
{
  int i,j,k;

  for(i = 0; i<4; i++)
    for(j = 0; j<4; j++)
      {                                                              260
        X[i][j] = 0;
        for(k = 0; k < 4; k++)
          {
              X[i][j] += M[i][k] * Y[k][j];
          }
      }
}

void RUpdateMatrices()
{                                                                    270
  float M[4][4];
  MatrixMult(M, g_RPerspective, g_RCamera);
  RViewMatrix(M);
}

void RSetProjMatrix(float M[4][4])
{
  int i,j;
  for(i = 0; i < 4; i++)
    for(j = 0; j < 4; j++)                                           280
      g_RPerspective[i][j] = M[i][j];

  RUpdateMatrices();
}
```

```
void RProjMatrix(float l, float r, float t, float b, float n, float f)
{
  g_RPerspective[0][0] = 2.0f*n/(r-l);
  g_RPerspective[0][1] = 0;
  g_RPerspective[0][2] = -(r+l)/(r-l);
  g_RPerspective[0][3] = 0;

  g_RPerspective[1][0] = 0;
  g_RPerspective[1][1] = 2.0f*n/(b-t);
  g_RPerspective[1][2] = -(b+t)/(b-t);
  g_RPerspective[1][3] = 0;

  g_RPerspective[2][0] = 0;
  g_RPerspective[2][1] = 0;
  g_RPerspective[2][2] = (f+n)/(f-n);
  g_RPerspective[2][3] = -2.0f*n*f/(f-n);

  g_RPerspective[3][0] = 0;
  g_RPerspective[3][1] = 0;
  g_RPerspective[3][2] = 1;
  g_RPerspective[3][3] = 0;

  RUpdateMatrices();
}

// u = angle between center and right edge
// v = angle between center and top edge
// from http://en.wikipedia.org/w/wiki.phtml?title=3D_projection&oldid=2227008
// updated Jan 2004, accessed 4/20/04
void RProjMatrixUV(float u, float v, float n, float f)
{
  g_RPerspective[0][0] = 1.0f/tan(u);
  g_RPerspective[0][1] = 0;
  g_RPerspective[0][2] = 0;
  g_RPerspective[0][3] = 0;

  g_RPerspective[1][0] = 0;
  g_RPerspective[1][1] = -1.0f/tan(v);
  g_RPerspective[1][2] = 0;
  g_RPerspective[1][3] = 0;

  g_RPerspective[2][0] = 0;
  g_RPerspective[2][1] = 0;
  g_RPerspective[2][2] = (f+n)/(f-n);
  g_RPerspective[2][3] = -2.0f*n*f/(f-n);

  g_RPerspective[3][0] = 0;
  g_RPerspective[3][1] = 0;
```

```
  g_RPerspective[3][2] = 1;
  g_RPerspective[3][3] = 0;

  RUpdateMatrices();

}
                                                                        340
void ROrthMatrix(float l, float r, float t, float b, float n, float f)
{
  g_RPerspective[0][0] = 2.0f/(r−l);
  g_RPerspective[0][1] = 0;
  g_RPerspective[0][2] = 0;
  g_RPerspective[0][3] = −(r+l)/(r−l);

  g_RPerspective[1][0] = 0;
  g_RPerspective[1][1] = 2.0f/(b−t);
  g_RPerspective[1][2] = 0;                                             350
  g_RPerspective[1][3] = −(b+t)/(b−t);

  g_RPerspective[2][0] = 0;
  g_RPerspective[2][1] = 0;
  g_RPerspective[2][2] = 2.0f/(f−n);
  g_RPerspective[2][3] = −(f+n)/(f−n);

  g_RPerspective[3][0] = 0;
  g_RPerspective[3][1] = 0;
  g_RPerspective[3][2] = 0;                                             360
  g_RPerspective[3][3] = 1;

  RUpdateMatrices();
}

// VRP = viewer point
// FP = look-at point
// UP = up-direction point
//  0 = x, 1 = y, 2 = z.
// method from http://www.siggraph.org/education/materials/           370
//              HyperGraph/viewing/view3d/3dview1.htm
// accessed 4/20/04
// also used 04_transformations.pdf
void RCameraMatrix(float VRP[3], float FP[3], float UP[3])
{
  float N[3], // normalized eye vector
    UV[3], // up vector
    V[3], // upwards normal to N
    U[3]; // right hand normal to N
                                                                        380
  int i;
```

```
  // compute eye vector
  for(i = 0; i < 3; i++)
    N[i] = FP[i] - VRP[i];

  // normalize
  for(i = 0; i < 3; i++)
    N[i] = N[i]/sqrt(N[0]*N[0] + N[1]*N[1] + N[2]*N[2]);
                                                                              390
  // compute up vector
  for(i = 0; i < 3; i++)
    UV[i] = UP[i] - VRP[i];

  // compute upwards normal
  for(i = 0; i < 3; i++)
    V[i] = UV[i] - (N[0]*UV[0]+N[1]*UV[1]+N[2]*UV[2])*N[i];

  // normalize
  for(i = 0; i < 3; i++)                                                      400
    V[i] = V[i]/sqrt(V[0]*V[0] + V[1]*V[1] + V[2]*V[2]);

  // U = N x V
  U[0] = -N[1]*V[2] + N[2]*V[1];
  U[1] = -N[2]*V[0] + N[0]*V[2];
  U[2] = -N[0]*V[1] + N[1]*V[0];

  g_RCamera[0][0] = U[0];
  g_RCamera[0][1] = U[1];
  g_RCamera[0][2] = U[2];                                                     410
  g_RCamera[0][3] = -VRP[0]*U[0] - VRP[1]*U[1] - VRP[2]*U[2];

  g_RCamera[1][0] = V[0];
  g_RCamera[1][1] = V[1];
  g_RCamera[1][2] = V[2];
  g_RCamera[1][3] = -VRP[0]*V[0] - VRP[1]*V[1] - VRP[2]*V[2];

  g_RCamera[2][0] = N[0];
  g_RCamera[2][1] = N[1];
  g_RCamera[2][2] = N[2];                                                     420
  g_RCamera[2][3] = -VRP[0]*N[0] - VRP[1]*N[1] - VRP[2]*N[2];

  g_RCamera[3][0] = 0;
  g_RCamera[3][1] = 0;
  g_RCamera[3][2] = 0;
  g_RCamera[3][3] = 1;

  RUpdateMatrices();
}
```

```
                                                        430
void RNormal(float x, float y, float z)
{
  if(g_RInSceneStream)
    RSendUnsigned(4);

  RSendUnsigned(RENDER_NORMAL);

  RSendFloat(x);
  RSendFloat(y);
  RSendFloat(z);                                        440
}

// 1 or 0
void RSetLit(unsigned s)
{
  if(g_RInSceneStream)
    RSendUnsigned(2);

  RSendUnsigned(RENDER_SET_LIT);
                                                        450
  RSendUnsigned(s);
}

// 1 or 0
void RSetUseAmb(unsigned s)
{
  if(g_RInSceneStream)
    RSendUnsigned(2);

  RSendUnsigned(RENDER_SET_USEAMB);                     460

  RSendUnsigned(s);
}

// 1 or 0
void RSetUseDir(unsigned s)
{
  if(g_RInSceneStream)
    RSendUnsigned(2);
                                                        470
  RSendUnsigned(RENDER_SET_USEDIR);

  RSendUnsigned(s);
}

#define R_TEXMODE_NONE 0
#define R_TEXMODE_COLOR 1
```

```
#define R_TEXMODE_TEXTURE 2
#define R_TEXMODE_BLEND 3
#define R_TEXMODE_TEXDECAL 4
#define R_TEXMODE_COLDECAL 5
#define R_TEXMODE_MODULATE 6

void RSetTexMode(unsigned s)
{
  if(g_RInSceneStream)
    RSendUnsigned(2);

  RSendUnsigned(RENDER_SET_TEXMODE);

  RSendUnsigned(s);
}

#define R_ALPHA_NONE 0
#define R_ALPHA_SOFT 1
#define R_ALPHA_HARD 2

void RSetTexAlpha(unsigned s)
{
  if(g_RInSceneStream)
    RSendUnsigned(2);

  RSendUnsigned(RENDER_SET_TEXALPHA);

  RSendUnsigned(s);
}

void RSetColAlpha(unsigned s)
{
  if(g_RInSceneStream)
    RSendUnsigned(2);

  RSendUnsigned(RENDER_SET_COLALPHA);

  RSendUnsigned(s);
}

void RSetColInterp(unsigned s)
{
  if(g_RInSceneStream)
    RSendUnsigned(2);

  RSendUnsigned(RENDER_SET_COLINTERP);

  RSendUnsigned(s);
```

```
}

void RSetLitInterp(unsigned s)
{
  if(g_RInSceneStream)                                   530
    RSendUnsigned(2);

  RSendUnsigned(RENDER_SET_LITINTERP);

  RSendUnsigned(s);
}

void RSetTexInterp(unsigned s)
{
  if(g_RInSceneStream)                                   540
    RSendUnsigned(2);

  RSendUnsigned(RENDER_SET_TEXINTERP);

  RSendUnsigned(s);
}

void RSetOutOfOrder(unsigned s)
{
  if(g_RInSceneStream)                                   550
    RSendUnsigned(2);

  RSendUnsigned(RENDER_SET_OUTOFORDER);

  RSendUnsigned(s);
}

#define R_TEXTILE_NONE 0
#define R_TEXTILE_REPEAT 1
#define R_TEXTILE_MIRROR 2                               560
#define R_TEXTILE_CLAMP 3

void RSetTexTile(unsigned s)
{
  if(g_RInSceneStream)
    RSendUnsigned(2);

  RSendUnsigned(RENDER_SET_TEXTILE);

  RSendUnsigned(s);                                      570
}

void RSetNoUseZ(unsigned s)
```

```
{
  if(g_RInSceneStream)
    RSendUnsigned(2);

  RSendUnsigned(RENDER_SET_NOUSEZ);

  RSendUnsigned(s);                                                580
}

void RSetNoWriteZ(unsigned s)
{
  if(g_RInSceneStream)
    RSendUnsigned(2);

  RSendUnsigned(RENDER_SET_NOWRITEZ);

  RSendUnsigned(s);                                                590
}

void RSetTextureID(unsigned s)
{
  if(g_RInSceneStream)
    RSendUnsigned(2);

  RSendUnsigned(RENDER_SET_TEXTUREID);

  RSendUnsigned(s);                                                600
}

void RColTexBalance(float s)
{
  if(g_RInSceneStream)
    RSendUnsigned(2);

  RSendUnsigned(RENDER_COLTEXBALANCE);

  RSendFloat(s);                                                   610
}

void RAlphaThresh(unsigned s)
{
  if(g_RInSceneStream)
    RSendUnsigned(2);

  RSendUnsigned(RENDER_ALPHATHRESH);

  RSendUnsigned(s);                                                620
}
```

```
void RAmbColorRGBI(unsigned rgbi)
{
  if(g_RInSceneStream)
    RSendUnsigned(2);
  RSendUnsigned(RENDER_AMBCOLOR);
  RSendUnsigned(rgbi);
}

void RAmbColor(unsigned R, unsigned G, unsigned B, unsigned I)
{
  RAmbColorRGBI( (R&0x0F)<<24|(G&0x0F)<<16|(B&0x0F)<<8|(I&0x0F) );
}

void RDirColorRGBI(unsigned rgbi)
{
  if(g_RInSceneStream)
    RSendUnsigned(2);
  RSendUnsigned(RENDER_DIRCOLOR);
  RSendUnsigned(rgbi);
}

void RDirColor(unsigned R, unsigned G, unsigned B, unsigned I)
{
  RDirColorRGBI( (R&0x0F)<<24|(G&0x0F)<<16|(B&0x0F)<<8|(I&0x0F) );
}

void RDirLight(float x, float y, float z)
{
  if(g_RInSceneStream)
    RSendUnsigned(4);

  RSendUnsigned(RENDER_DIRLIGHT);
  RSendFloat(x);
  RSendFloat(y);
  RSendFloat(z);
}

void RAmbReflect(unsigned s)
{
  if(g_RInSceneStream)
    RSendUnsigned(2);

  RSendUnsigned(RENDER_AMBREFLECT);

  RSendUnsigned(s);
}
```

```
void RDirReflect(unsigned s)                                          670
{
  if(g_RInSceneStream)
    RSendUnsigned(2);

  RSendUnsigned(RENDER_DIRREFLECT);

  RSendUnsigned(s);
}

#define R_PAGE_NONE 0                                                 680
#define R_PAGE_BACK 0x01
#define R_PAGE_FRONT 0x02
#define R_PAGE_BOTH 0x03

void RClearFBRGBX(unsigned page, unsigned rgbx)
{
  if(!g_RInSceneStream)
    {
      RSendUnsigned(RENDER_CLEARFB);
      RSendUnsigned(page);                                            690
      RSendUnsigned(rgbx);
    }
}

void RClearFB(unsigned page, unsigned R, unsigned G, unsigned B)
{
  RClearFBRGBX(page, (R&0x0FF)<<24 | (G&0x0FF)<<16 | (B&0x0FF)<<8);
}

void RClearZ()                                                        700
{
  if(!g_RInSceneStream)
    {
      RSendUnsigned(RENDER_CLEARZ);
    }
}

void RSetPage(unsigned page)
{
  if(!g_RInSceneStream)                                               710
    {
      RSendUnsigned(RENDER_SETPAGE);
      RSendUnsigned(page);
    }
}

void RFlipPage(unsigned waitforvsync)
```

```
{
  if(!g_RInSceneStream)
    {                                                        720
      RSendUnsigned(RENDER_FLIPPAGE);
      RSendUnsigned(waitforvsync);
    }
}

unsigned RAllocateTexture(unsigned sizex, unsigned sizey)
{
  if(!g_RInSceneStream)
    {
      RSendUnsigned(RENDER_ALLOCATE_TEXTURE);                730
      RSendUnsigned(sizex);
      RSendUnsigned(sizey);
      RMETARead(1);
      return RReadUnsigned();
    }
  return -1;
}

void RDeallocTexture(unsigned t)
{                                                            740
  if(!g_RInSceneStream)
    {
      RSendUnsigned(RENDER_DEALLOC_TEXTURE);
      RSendUnsigned(t);
    }
}

void RUploadTexture(unsigned t, unsigned length, unsigned *data)
{
  if(!g_RInSceneStream)                                      750
    {
      unsigned *ptr;
      RSendUnsigned(RENDER_UPLOAD_TEXTURE);
      RSendUnsigned(t);
      RSendUnsigned(length);
      for(ptr = data; ptr <= data + length; ptr++)
        {
          RSendUnsigned(*ptr);
        }
    }                                                        760
}

unsigned RTexMemAvail()
{
  if(!g_RInSceneStream)
```

```
    {
      RSendUnsigned(RENDER_TEXMEM_AVAIL);
      RMETARead(1);
      return RReadUnsigned();
    }
  return 0;
}

void RCompactTexMem()
{
  if(!g_RInSceneStream)
    {
      RSendUnsigned(RENDER_COMPACT_TEXMEM);
    }

}

void RWriteFBRGBX( unsigned x, unsigned y, unsigned page, unsigned rgbx)
{
  if(!g_RInSceneStream)
    {
      RSendUnsigned(RENDER_WRITE_FB);
      RSendUnsigned(x);
      RSendUnsigned(y);
      RSendUnsigned(page);
      RSendUnsigned(rgbx);
    }
}

void RWriteFB( unsigned x, unsigned y, unsigned page,
               unsigned R, unsigned G, unsigned B)
{
  RWriteFBRGBX(x,y, page, (R&0x0FF)<<24 | (G&0x0FF)<<16 | (B&0x0FF) << 8);
}

void RWriteFBBlock( unsigned x, unsigned y, unsigned page,
                    unsigned length, unsigned *data)
{
  if(!g_RInSceneStream)
    {
      unsigned * ptr;
      RSendUnsigned(RENDER_WRITE_FB_BLOCK);
      RSendUnsigned(x);
      RSendUnsigned(y);
      RSendUnsigned(page);
      RSendUnsigned(length);
      for(ptr = data; ptr < data + length; ptr++)
        RSendUnsigned(*ptr);
```

```
    }
}

unsigned RReadFB( unsigned x, unsigned y, unsigned page)
{
  if(!g_RInSceneStream)
    {                                                              820
      RSendUnsigned(RENDER_READ_FB);
      RSendUnsigned(x);
      RSendUnsigned(y);
      RSendUnsigned(page);
      RMETARead(1);
      return RReadUnsigned();
    }
  return 0;
}
                                                                   830
void RReadFBBlock( unsigned x, unsigned y, unsigned page,
                   unsigned length, unsigned *data)
{
  if(!g_RInSceneStream)
    {
      unsigned * ptr;
      RSendUnsigned(RENDER_READ_FB_BLOCK);
      RSendUnsigned(x);
      RSendUnsigned(y);
      RSendUnsigned(page);                                         840
      RSendUnsigned(length);
      RMETARead(length);
      for(ptr = data; ptr < data + length; ptr++)
        {
          *ptr = RReadUnsigned();
        }
    }
}

void RWriteZ( unsigned x, unsigned y, signed val)                  850
{
  if(!g_RInSceneStream)
    {
      RSendUnsigned(RENDER_WRITE_Z);
      RSendUnsigned(x);
      RSendUnsigned(y);
      RSendSigned(val);
    }
}
                                                                   860
```

```
void RWriteZBlock( unsigned x, unsigned y, unsigned length, signed *data)
{
  if(!g_RInSceneStream)
    {
      signed * ptr;
      RSendUnsigned(RENDER_WRITE_Z_BLOCK);
      RSendUnsigned(x);
      RSendUnsigned(y);
      RSendUnsigned(length);                                            870
      for(ptr = data; ptr < data + length; ptr++)
        RSendSigned(*ptr);
    }
}

signed RReadZ( unsigned x, unsigned y)
{
  if(!g_RInSceneStream)
    {
      RSendUnsigned(RENDER_READ_Z);                                     880
      RSendUnsigned(x);
      RSendUnsigned(y);
      RMETARead(1);
      return RReadUnsigned();
    }
  return 0;
}

void RReadZBlock( unsigned x, unsigned y, unsigned page,
                  unsigned length, signed *data)                        890
{
  if(!g_RInSceneStream)
    {
      signed * ptr;
      RSendUnsigned(RENDER_READ_Z_BLOCK);
      RSendUnsigned(x);
      RSendUnsigned(y);
      RSendUnsigned(length);
      RMETARead(length);
      for(ptr = data; ptr < data + length; ptr++)                       900
        {
          *ptr = RReadUnsigned();
        }
    }
}

void RReset()
{
  if(!g_RInSceneStream)
```

```
    {                                                       910
      RSendUnsigned(RENDER_RESET);
      RInit();
    }
}

// note: doesn't do a RMETAHalt by default
void RHalt()
{
  if(!g_RInSceneStream)
    {                                                       920
      RSendUnsigned(RENDER_HALT);
    }
}
#endif // RENDER_CLIENT_H
```

D.6 triangletest.c

```c
// triangletest.c
// started as simply a recreation of hostcmd-triangletest.txt
// using the render_client library, for testing.
//
// now is a full testsuite for the performance of the arch.
//
// Ken Taylor 5/18/04

#include "render_client.h"
#include <math.h>

void drawtriangles(unsigned alpha)
{
    RBeginPrim();
    RColor(0xFF, 0, 0, alpha);
    RVertex(0,0,0,0,0);
    RColor(0, 0xFF, 0, 0xFF);
    RVertex(0.5, 0, 0, 1, 0);
    RColor(0, 0, 0xFF, 0xFF);
    RVertex(0, 0.5, 0, 0, 1);
    REndPrim();

    RBeginPrim();
    RColor(0xFF, 0xFF, 0, alpha);
    RVertex(0.9, 0.5, 0, 1, 0);
    RColor(0xFF, 0, 0xFF, alpha);
    RVertex(0.9, 0.75, 0, 1, 1);
    RColor(0, 0xFF, 0xFF, alpha);
    RVertex(0.75, 0.75, 0, 0, 1);
    REndPrim();

    RColor(0xFF, 0xFF, 0xFF, alpha);

    RBeginPrim();
    RVertex(-0.5, -0.5, 0, 0, 1);
    RVertex(-0.4, -0.7, 0, 0.5, 0);
    RVertex(-0.3, -0.5, 0, 1, 1);
    REndPrim();

    RBeginPrim();
    RVertex(-0.5, 0, 0, 0, 0);
    RVertex(-0.4, 0, 0, 1, 0);
    RColor(0xFF, 0, 0, alpha);
    RVertex(-0.4, 0.1, 0, 1, 1);
    REndPrim();
```

```
    RBeginPrim();
    RColor(0x50, 0, 0xc0, alpha);
    RVertex(-0.7, 0.9, 0, 0, 1);
    RVertex(-0.7, 0.01, 0, 0, 0);                                          50
    RColor(0, 0xFF, 0xCC, 0xFF);
    RVertex(-0.4, 0.45, 0, 1, 0.5);
    REndPrim();

    RBeginPrim();
    RColor(0xFF, 0xC0, 0x40, 0xFF);
    RVertex(0, 0, 0, 0, 1);
    RColor(0, 0xFF, 0x88, alpha);
    RVertex(0, -1, 0, 0, 0);
    RColor(0xFF, 0xFF, 0xFF, alpha);                                       60
    RVertex(1, 0, 0, 1, 1);
    REndPrim();

}

// sets a matrix with a certain angle and (2D) scale
//  not a general transform! only works for this test!
//  assumes the rest of the matrix looks like an identity!
void SetMatrix(float Rot[4][4], double angle, double scale)
{                                                                          70
  Rot[1][1] = Rot[0][0] = scale * cos(angle);
  Rot[1][0] = scale * sin(angle);
  Rot[0][1] = -Rot[1][0];
}


  // need to render several different scales. possible variables:
  // RSetTexMode( R_TEXMODE_NONE, COLOR, TEXTURE, BLEND,
  //              TEXDECAL, COLDECAL, MODULATE)
  // RSetTexAlpha(R_ALPHA_NONE,SOFT,HARD                                   80
  // translucent or not.
  // RSetColAlpha(R_ALPHA_NONE,SOFT,HARD
  // RSetColInterp(1,0)
  // RSetTexTile(R_TEXTILE_NONE, REPEAT, MIRROR, CLAMP)
  // RSetTexInterp(1,0)
  //
  // RSetLit(1,0)
  // RSetUseAmb(1,0)
  // RSetUseDir(1,0)
  // RSetLitInterp(1,0)                                                    90
  //
  // RSetNoUseZ(1,0);
  // RSetNoWriteZ(1,0);
```

```
// RSetOutOfOrder(1,0);
// for full feature testing
// for each size:
// (for simplicity:
//   - don't care about all modes. care mostly about:
//     - uncolored/textured, unlit, noz, outoforder (baseline)
//     - colored/flatcolored/interpcolored                                   100
//     - untex/nearesttex/bilintex
//       - if both textured and colored, use blend
//       - texture should have no transparent spots
//       - color should be soft alpha
//     - unlit/flatdiramblit/interpdiramblit
//     - fullz,zreadonly,zwriteonly,noz
//     - transparent, opaque
//       - for zreadonly,zwriteonly,noz,transparent:
//         - in order, out of order.
                                                                             110

// so table looks like:
// texalpha = x
// colalpha = SOFT
// textile = REPEAT
// useamb = 1
// usedir = 1
//
// texmode colinterp texinterp lit, litinterp
//  NONE       0         0     0    0    // baseline                         120
//  COLOR      0         x     0    x
//  COLOR      1         x     0    x
//  COLOR      0         x     1    0
//  COLOR      1         x     1    0
//  COLOR      0         x     1    1
//  COLOR      1         x     1    1

//  TEXTURE    x         0     0    x
//  TEXTURE    x         1     0    x
//  TEXTURE    x         0     1    0                                        130
//  TEXTURE    x         1     1    0
//  TEXTURE    x         0     1    1
//  TEXTURE    x         1     1    1

//  BLEND      0         0     0    x
//  BLEND      1         0     0    x
//  BLEND      0         0     1    0
//  BLEND      1         0     1    0
//  BLEND      0         0     1    1
//  BLEND      1         0     1    1                                        140
//  BLEND      0         1     0    x
```

```
// BLEND    1       1    0    x
// BLEND    0       1    1    0
// BLEND    1       1    1    0
// BLEND    0       1    1    1
// BLEND    1       1    1    1

#define OTVSIZE 24

struct _OuterTestVec {                                                    150
  unsigned texmode;
  unsigned colinterp;
  unsigned texinterp;
  unsigned lit;
  unsigned litinterp;
  const char *name;
} OTV[OTVSIZE] = { {R_TEXMODE_COLOR  , 0, 0, 0, 0 , "col0x0x"},
                  {R_TEXMODE_COLOR  , 1, 0, 0, 0 , "col1x0x"},
                  {R_TEXMODE_COLOR  , 0, 0, 1, 0 , "col0x10"},
                  {R_TEXMODE_COLOR  , 1, 0, 1, 0 , "col1x10"},             160
                  {R_TEXMODE_COLOR  , 0, 0, 1, 1 , "col0x11"},
                  {R_TEXMODE_COLOR  , 1, 0, 1, 1 , "col1x11"},
                  {R_TEXMODE_TEXTURE, 0, 0, 0, 0 , "texx00x"},
                  {R_TEXMODE_TEXTURE, 0, 1, 0, 0 , "texx10x"},
                  {R_TEXMODE_TEXTURE, 0, 0, 1, 0 , "texx010"},
                  {R_TEXMODE_TEXTURE, 0, 1, 1, 0 , "texx110"},
                  {R_TEXMODE_TEXTURE, 0, 0, 1, 1 , "texx011"},
                  {R_TEXMODE_TEXTURE, 0, 1, 1, 1 , "texx111"},
                  {R_TEXMODE_BLEND  , 0, 0, 0, 0 , "bln000x"},
                  {R_TEXMODE_BLEND  , 1, 0, 0, 0 , "bln100x"},             170
                  {R_TEXMODE_BLEND  , 0, 0, 1, 0 , "bln0010"},
                  {R_TEXMODE_BLEND  , 1, 0, 1, 0 , "bln1010"},
                  {R_TEXMODE_BLEND  , 0, 0, 1, 1 , "bln0011"},
                  {R_TEXMODE_BLEND  , 1, 0, 1, 1 , "bln1011"},
                  {R_TEXMODE_BLEND  , 0, 1, 0, 0 , "bln0100"},
                  {R_TEXMODE_BLEND  , 1, 1, 0, 0 , "bln1100"},
                  {R_TEXMODE_BLEND  , 0, 1, 1, 0 , "bln0110"},
                  {R_TEXMODE_BLEND  , 1, 1, 1, 0 , "bln1110"},
                  {R_TEXMODE_BLEND  , 0, 1, 1, 1 , "bln0111"},
                  {R_TEXMODE_BLEND  , 1, 1, 1, 1 , "bln1111"} };           180

  // for each mode:
  // (note, trans only matters in color and blend modes)
  // trans, nousez, nowritez, ooo
  //   0      0       0       x
  //   0      0       1       0
  //   0      1       0       0
  //   0      1       1       0
  //   1      0       0       0
```

```
//  1   0   1   0
//  1   1   0   0
//  1   1   1   0
//  0   0   1   1
//  0   1   0   1
//  0   1   1   1 // use this one for the NONE above (baseline)
//  1   0   0   1
//  1   0   1   1
//  1   1   0   1
//  1   1   1   1

#define ITVSIZE 15

struct _InnerTestVec {
  unsigned alpha;
  unsigned nousez;
  unsigned nowritez;
  unsigned unordered;
  const char *name;
} ITV[ITVSIZE] = { {0xFF, 0, 0, 1, "itv000x"},
                   {0xFF, 0, 1, 0, "itv0010"},
                   {0xFF, 1, 0, 0, "itv0100"},
                   {0xFF, 1, 1, 0, "itv0110"},
                   {0x88, 0, 0, 0, "itv1000"},
                   {0x88, 0, 1, 0, "itv1010"},
                   {0x88, 1, 0, 0, "itv1100"},
                   {0x88, 1, 1, 0, "itv1110"},
                   {0xFF, 0, 1, 1, "itv0011"},
                   {0xFF, 1, 0, 1, "itv0101"},
                   {0xFF, 1, 1, 1, "itv0111"},
                   {0x88, 0, 0, 1, "itv1001"},
                   {0x88, 0, 1, 1, "itv1011"},
                   {0x88, 1, 0, 1, "itv1101"},
                   {0x88, 1, 1, 1, "itv1111"} };

unsigned TexID = -1;
unsigned Texture[8*8] = { 0xFFFF00FF, 0xFFFF00FF, 0xFFFF00FF, 0xFFFF00FF,
                          0xFFFF00FF, 0xFFFF00FF, 0xFFFF00FF, 0xFFFF00FF,
                          0xFFFF00FF, 0xFFFF00FF, 0x00FFFFFF, 0xFFFF00FF,
                          0xFFFF00FF, 0x00FFFFFF, 0xFFFF00FF, 0xFFFF00FF,
                          0xFFFF00FF, 0xFFFF00FF, 0xFFFF00FF, 0xFFFF00FF,
                          0xFFFF00FF, 0xFFFF00FF, 0xFFFF00FF, 0xFFFF00FF,
                          0xFFFF00FF, 0x000000FF, 0xFFFF00FF, 0xFFFF00FF,
                          0xFFFF00FF, 0xFFFF00FF, 0x000000FF, 0xFFFF00FF,
                          0xFFFF00FF, 0x000000FF, 0xFFFF00FF, 0xFFFF00FF,
                          0xFFFF00FF, 0xFFFF00FF, 0x000000FF, 0xFFFF00FF,
                          0xFFFF00FF, 0xFFFF00FF, 0x000000FF, 0xFFFF00FF,
                          0xFFFF00FF, 0x000000FF, 0xFFFF00FF, 0xFFFF00FF,
```

```
                     0xFFFF00FF, 0xFFFF00FF, 0xFFFF00FF, 0x000000FF,
                     0x000000FF, 0xFFFF00FF, 0xFFFF00FF, 0xFFFF00FF,
                     0xFFFF00FF, 0xFFFF00FF, 0xFFFF00FF, 0xFFFF00FF,         240
                     0xFFFF00FF, 0xFFFF00FF, 0xFFFF00FF, 0xFFFF00FF };



void DoOneTestSet(int o,int i, double scale)
{
  unsigned alpha;
  double angle = (0.0/360.0)*(2.0*M_PI);
  float Rot[4][4] = {1, 0, 0, 0,
                     0, 1, 0, 0,                                               250
                     0, 0, 1, 0,
                     0, 0, 0, 1};
  float Identity[4][4] = {1,0,0,0,
                          0,1,0,0,
                          0,0,1,0,
                          0,0,0,1};

  RReset();
  RSendMeta("preset");
                                                                               260
  RViewMatrix(Identity);
  SetMatrix(Rot, angle, scale);
  RModelMatrix(Rot);


  RSetTexMode(OTV[o].texmode);
  RSetColInterp(OTV[o].colinterp);
  RSetTexInterp(OTV[o].texinterp);
  RSetLit(OTV[o].lit);
  RSetLitInterp(OTV[o].litinterp);                                             270
  alpha = ITV[i].alpha;
  RSetNoUseZ(ITV[i].nousez);
  RSetNoWriteZ(ITV[i].nowritez);
  RSetOutOfOrder(ITV[i].unordered);

  RSetColAlpha(R_ALPHA_SOFT);
  RSetTexTile(R_TEXTILE_REPEAT);
  RSetUseAmb(1);
  RSetUseDir(1);
                                                                               280
  TexID = RAllocateTexture(8, 8);
  if(TexID == -1)
    {
      RMETADebug();
      RSendMeta("texerr!");
```

```
      RMETAHalt();
    }
  else
    RUploadTexture(TexID, 8*8, Texture);
                                                                        290

  RSetTextureID(TexID);

  RSendMeta("pstart");
  RBeginScene();

  RSendMeta("pframe");
  drawtriangles(alpha);

  RSendMeta("pframe");                                                  300
  drawtriangles(alpha);

  RSendMeta("pframe");
  drawtriangles(alpha);

  RSendMeta("pframe");
  drawtriangles(alpha);

  REndScene();
                                                                        310
}

void DoTests()
{
  int o,i;
  unsigned alpha;

  double scale;

  i = 12; // change for starting in the midst of a test                 320
  o = 4; // ditto

  for(; o < OTVSIZE; o++)
    {
      for(; i < ITVSIZE; i++)
        {
          RMETADebug();
          RSendMeta(OTV[o].name);
          RMETADebug();
          RSendMeta(ITV[i].name);                                       330
          RMETADebug();
          RSendMeta("s 0.01 ");
          DoOneTestSet(o,i,0.01);
```

```
RSendMeta("preport");

RMETADebug();
RSendMeta(OTV[o].name);
RMETADebug();
RSendMeta(ITV[i].name);
RMETADebug();                                    340
RSendMeta("s 0.05 ");
DoOneTestSet(o,i,0.05);
RSendMeta("preport");

/*
RMETADebug();
RSendMeta(OTV[o].name);
RMETADebug();
RSendMeta(ITV[i].name);
RMETADebug();                                    350
RSendMeta("s 0.1 ");
DoOneTestSet(o,i,0.1);
RSendMeta("preport");

RMETADebug();
RSendMeta(OTV[o].name);
RMETADebug();
RSendMeta(ITV[i].name);
RMETADebug();
RSendMeta("s 0.2 ");                             360
DoOneTestSet(o,i,0.2);
RSendMeta("preport");

RMETADebug();
RSendMeta(OTV[o].name);
RMETADebug();
RSendMeta(ITV[i].name);
RMETADebug();
RSendMeta("s 0.3 ");
DoOneTestSet(o,i,0.3);                           370
RSendMeta("preport");

RMETADebug();
RSendMeta(OTV[o].name);
RMETADebug();
RSendMeta(ITV[i].name);
RMETADebug();
RSendMeta("s 0.5 ");
DoOneTestSet(o,i,0.5);
RSendMeta("preport");                            380
```

```
            RMETADebug();
            RSendMeta(OTV[o].name);
            RMETADebug();
            RSendMeta(ITV[i].name);
            RMETADebug();
            RSendMeta("s 1.0 ");
            DoOneTestSet(o,i,1.0);
            RSendMeta("preport");
            */                                                          390
          }
        i = 0; // leave this line intact.
      }
  }


  int main(void)
  {
    RInit();
                                                                        400
    // get startup crap out of the way, for performance measuring
    RBeginScene();
    REndScene();

    DoTests();

    RMETAHalt();

    return 0;
  }                                                                     410
```

D.7 cubetest.c

```
// cubetest.c
//
// Ken Taylor 5/18/04

#include "render_client.h"
#include <math.h>

#define INVSQRT3 0.57735026918962576450915

void drawcube()                                              10
{
  // top
  //
  //  (-0.5,  0.5,  0.5) - ( 0.5,  0.5,  0.5)
  //          |         /          |
  //  (-0.5,  0.5, -0.5) - ( 0.5,  0.5, -0.5)
  //
  RColor(0xFF, 0, 0, 0xFF);
  RBeginPrim();
  RNormal(-INVSQRT3, INVSQRT3, -INVSQRT3);                    20
  RVertex(-0.5, 0.5, -0.5, 0,1);
  RNormal(-INVSQRT3, INVSQRT3, INVSQRT3);
  RVertex(-0.5, 0.5, 0.5, 0, 0);
  RNormal(INVSQRT3, INVSQRT3, INVSQRT3);
  RVertex(0.5, 0.5, 0.5, 1, 0);
  REndPrim();
  RBeginPrim();
  RNormal(-INVSQRT3, INVSQRT3, -INVSQRT3);
  RVertex(-0.5, 0.5, -0.5, 0,1);
  RNormal(INVSQRT3, INVSQRT3, INVSQRT3);                      30
  RVertex(0.5, 0.5, 0.5, 1, 0);
  RNormal(INVSQRT3, INVSQRT3, -INVSQRT3);
  RVertex(0.5, 0.5, -0.5, 1, 1);
  REndPrim();

  // front
  //
  //  (-0.5,  0.5, -0.5) - ( 0.5,  0.5, -0.5)
  //          |         /          |
  //  (-0.5, -0.5, -0.5) - ( 0.5, -0.5, -0.5)                 40
  //
  RColor(0, 0xFF, 0, 0xFF);
  RBeginPrim();
  RNormal(-INVSQRT3, -INVSQRT3, -INVSQRT3);
  RVertex(-0.5, -0.5, -0.5, 0,1);
```

```
RNormal(-INVSQRT3, INVSQRT3, -INVSQRT3);
RVertex(-0.5, 0.5, -0.5, 0, 0);
RNormal(INVSQRT3, INVSQRT3, -INVSQRT3);
RVertex(0.5, 0.5, -0.5, 1, 0);
REndPrim();                                                        50
RBeginPrim();
RNormal(-INVSQRT3, -INVSQRT3, -INVSQRT3);
RVertex(-0.5, -0.5, -0.5, 0,1);
RNormal(INVSQRT3, INVSQRT3, -INVSQRT3);
RVertex(0.5, 0.5, -0.5, 1, 0);
RNormal(INVSQRT3, -INVSQRT3, -INVSQRT3);
RVertex(0.5, -0.5, -0.5, 1, 1);
REndPrim();

// right                                                           60
//
//  ( 0.5,  0.5, -0.5) - ( 0.5,  0.5,  0.5)
//           |         /          |
//  ( 0.5, -0.5, -0.5) - ( 0.5, -0.5,  0.5)
//
RColor(0, 0, 0xFF, 0xFF);
RBeginPrim();
RNormal(INVSQRT3, -INVSQRT3, -INVSQRT3);
RVertex(0.5, -0.5, -0.5, 0,1);
RNormal(INVSQRT3, INVSQRT3, -INVSQRT3);                            70
RVertex(0.5, 0.5, -0.5, 0, 0);
RNormal(INVSQRT3, INVSQRT3, INVSQRT3);
RVertex(0.5, 0.5, 0.5, 1, 0);
REndPrim();
RBeginPrim();
RNormal(INVSQRT3, -INVSQRT3, -INVSQRT3);
RVertex(0.5, -0.5, -0.5, 0,1);
RNormal(INVSQRT3, INVSQRT3, INVSQRT3);
RVertex(0.5, 0.5, 0.5, 1, 0);
RNormal(INVSQRT3, -INVSQRT3, INVSQRT3);                            80
RVertex(0.5, -0.5, 0.5, 1, 1);
REndPrim();

// bottom
//
//  (-0.5, -0.5, -0.5) - ( 0.5, -0.5, -0.5)
//           |         /          |
//  (-0.5, -0.5,  0.5) - ( 0.5, -0.5,  0.5)
//
RColor(0xFF, 0, 0xFF, 0xFF);                                       90
RBeginPrim();
RNormal(-INVSQRT3, -INVSQRT3, INVSQRT3);
RVertex(-0.5, -0.5, 0.5, 0,1);
```

```
RNormal(−INVSQRT3, −INVSQRT3, −INVSQRT3);
RVertex(−0.5, −0.5, −0.5, 0, 0);
RNormal(INVSQRT3, −INVSQRT3, −INVSQRT3);
RVertex(0.5, −0.5, −0.5, 1, 0);
REndPrim();
RBeginPrim();
RNormal(−INVSQRT3, −INVSQRT3, INVSQRT3);                         100
RVertex(−0.5, −0.5, 0.5, 0,1);
RNormal(INVSQRT3, −INVSQRT3, −INVSQRT3);
RVertex(0.5, −0.5, −0.5, 1, 0);
RNormal(INVSQRT3, −INVSQRT3, INVSQRT3);
RVertex(0.5, −0.5, 0.5, 1, 1);
REndPrim();

// back
//
//  ( 0.5,  0.5,  0.5) - (-0.5,  0.5,  0.5)                      110
//            |          /          |
//  ( 0.5, -0.5,  0.5) - (-0.5, -0.5,  0.5)
//
RColor(0xFF, 0xFF, 0, 0xFF);
RBeginPrim();
RNormal(INVSQRT3, −INVSQRT3, INVSQRT3);
RVertex(0.5, −0.5, 0.5, 0,1);
RNormal(INVSQRT3, INVSQRT3, INVSQRT3);
RVertex(0.5, 0.5, 0.5, 0, 0);
RNormal(−INVSQRT3, INVSQRT3, INVSQRT3);                          120
RVertex(−0.5, 0.5, 0.5, 1, 0);
REndPrim();
RBeginPrim();
RNormal(INVSQRT3, −INVSQRT3, INVSQRT3);
RVertex(0.5, −0.5, 0.5, 0,1);
RNormal(−INVSQRT3, INVSQRT3, INVSQRT3);
RVertex(−0.5, 0.5, 0.5, 1, 0);
RNormal(−INVSQRT3, −INVSQRT3, INVSQRT3);
RVertex(−0.5, −0.5, 0.5, 1, 1);
REndPrim();                                                      130

// left
//
//  (-0.5,  0.5,  0.5) - (-0.5,  0.5, -0.5)
//          |          /          |
//  (-0.5, -0.5,  0.5) - (-0.5, -0.5, -0.5)
//
RColor(0, 0xFF, 0xFF, 0xFF);
RBeginPrim();
RNormal(−INVSQRT3, −INVSQRT3, INVSQRT3);                         140
RVertex(−0.5, −0.5, 0.5, 0,1);
```

```
  RNormal(-INVSQRT3, INVSQRT3, INVSQRT3);
  RVertex(-0.5, 0.5, 0.5, 0, 0);
  RNormal(-INVSQRT3, INVSQRT3, -INVSQRT3);
  RVertex(-0.5, 0.5, -0.5, 1, 0);
  REndPrim();
  RBeginPrim();
  RNormal(-INVSQRT3, -INVSQRT3, INVSQRT3);
  RVertex(-0.5, -0.5, 0.5, 0,1);
  RNormal(-INVSQRT3, INVSQRT3, -INVSQRT3);                                  150
  RVertex(-0.5, 0.5, -0.5, 1, 0);
  RNormal(-INVSQRT3, -INVSQRT3, -INVSQRT3);
  RVertex(-0.5, -0.5, -0.5, 1, 1);
  REndPrim();

}

int main(void)
{
  unsigned Texture[16*16] = {0xFF0000FF, 0xFF0000FF, 0xFF0000FF, 0xFF0000FF,   160
                             0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF,
                             0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF,
                             0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF,
                             0x00FF00FF, 0xFF0000FF, 0xFF0000FF, 0xFF0000FF,
                             0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF,
                             0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF,
                             0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF,
                             0x0000FFFF, 0xFF0000FF, 0xFF0000FF, 0xFF0000FF,
                             0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF,
                             0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF,   170
                             0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF,
                             0xFF0000FF, 0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF,
                             0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF,
                             0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF,
                             0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF,
                             0x00FF00FF, 0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF,
                             0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF,
                             0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF,
                             0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF,
                             0x0000FFFF, 0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF,   180
                             0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF,
                             0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF,
                             0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF,
                             0xFF0000FF, 0xFFFFFFFF, 0xFFFFFFcc, 0xFFFFFF88,
                             0xFFFFFF44, 0xFFFFFF00, 0xFFFFFFFF, 0xFFFFFFFF,
                             0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF,
                             0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF,
                             0x00FF00FF, 0xFFFFFFFF, 0xFFFFFFCC, 0xFFFFFF88,
                             0xFFFFFF44, 0xFFFFFF00, 0xFFFFFFFF, 0xFFFFFFFF,
```

```
                        0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF,     190
                        0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF,
                        0x0000FFFF, 0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF,
                        0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF,
                        0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF,
                        0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF,
                        0xFF0000FF, 0xFFFFFFFF, 0x00FF00FF, 0x0000FFFF,
                        0x0000FFFF, 0x0000FFFF, 0x00FF00FF, 0xFFFFFFFF,
                        0xFFFFFFFF, 0xFFFFFF00, 0xFFFFFF00, 0xFFFFFFFF,
                        0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF,
                        0x00FF00FF, 0xFFFFFFFF, 0x0000FFFF, 0xFFFF00FF,     200
                        0xFFFF00FF, 0xFFFF00FF, 0x0000FFFF, 0xFFFFFFFF,
                        0xFFFFFFFF, 0xFFFFFF00, 0xFFFFFF00, 0xFFFFFFFF,
                        0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF,
                        0x0000FFFF, 0xFFFFFFFF, 0x0000FFFF, 0xFFFF00FF,
                        0xFFFF00FF, 0xFFFF00FF, 0x0000FFFF, 0xFFFFFFFF,
                        0xFFFFFFFF, 0xFFFFFF00, 0xFFFFFF00, 0xFFFFFFFF,
                        0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF,
                        0xFF0000FF, 0xFFFFFFFF, 0x0000FFFF, 0xFFFF00FF,
                        0xFFFF00FF, 0xFFFF00FF, 0x0000FFFF, 0xFFFFFFFF,
                        0xFFFFFFFF, 0xFFFFFF00, 0xFFFFFF00, 0xFFFFFFFF,     210
                        0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF,
                        0x00FF00FF, 0xFFFFFFFF, 0x00FF00FF, 0x0000FFFF,
                        0x0000FFFF, 0x0000FFFF, 0x00FF00FF, 0xFFFFFFFF,
                        0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF,
                        0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF,
                        0x0000FFFF, 0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF,
                        0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF,
                        0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF,
                        0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF,
                        0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF,     220
                        0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF,
                        0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF,
                        0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF };

unsigned Texture2[8*8] = { 0xFFFF00FF, 0xFFFF00FF, 0xFFFF00FF, 0xFFFF00FF,
                        0xFFFF00FF, 0xFFFF00FF, 0xFFFF00FF, 0xFFFF00FF,
                        0xFFFF00FF, 0xFFFF00FF, 0x00FFFFFF, 0xFFFF00FF,
                        0xFFFF00FF, 0x00FFFFFF, 0xFFFF00FF, 0xFFFF00FF,
                        0xFFFF00FF, 0xFFFF00FF, 0xFFFF00FF, 0xFFFF00FF,
                        0xFFFF00FF, 0xFFFF00FF, 0xFFFF00FF, 0xFFFF00FF,     230
                        0xFFFF00FF, 0x000000FF, 0xFFFF00FF, 0xFFFF00FF,
                        0xFFFF00FF, 0xFFFF00FF, 0x000000FF, 0xFFFF00FF,
                        0xFFFF00FF, 0x000000FF, 0xFFFF00FF, 0xFFFF00FF,
                        0xFFFF00FF, 0xFFFF00FF, 0x000000FF, 0xFFFF00FF,
                        0xFFFF00FF, 0xFFFF00FF, 0x000000FF, 0xFFFF00FF,
                        0xFFFF00FF, 0x000000FF, 0xFFFF00FF, 0xFFFF00FF,
                        0xFFFF00FF, 0xFFFF00FF, 0xFFFF00FF, 0x000000FF,
```

```
                    0x000000FF, 0xFFFF00FF, 0xFFFF00FF, 0xFFFF00FF,
                    0xFFFF00FF, 0xFFFF00FF, 0xFFFF00FF, 0xFFFF00FF,
                    0xFFFF00FF, 0xFFFF00FF, 0xFFFF00FF, 0xFFFF00FF };      240


float M[4][4] = {1, 0, 0, 0,
                 0, 1, 0, 0,
                 0, 0, 1, 0,
                 0, 0, 0, 1};

float VRP[3] = {2.0,2.0,-3.0};
float FP[3]  = {0.0,0.0,0.0};
float UP[3]  = {0.0,10.0,0.0};                                                250
signed texID, texID2;

RInit();

// world coords oriented, x across, y up/down, z back/forth
//                        pos right  pos up   pos away

RCameraMatrix(VRP, FP, UP);
RProjMatrixUV(45.0*M_PI/180.0,45.0*M_PI/180.0,0.1,100);
                                                                              260
texID = RAllocateTexture(16, 16);

if(texID == -1)
  {
    RMETADebug();
    RSendMeta("texerr");
    //RMETAHalt();
  }
else
  {                                                                           270
    //RUploadTexture(texID, 16*16, Texture);
    //RSetTextureID(texID);
    //RSetTexMode(R_TEXMODE_BLEND);
    //RSetTexAlpha(R_ALPHA_HARD);
    //RColTexBalance(0.3);
  }
/*
texID2 = RAllocateTexture(8, 8);

if(texID2 == -1)                                                              280
  {
    RMETADebug();
    RSendMeta("texerr2");
    //RMETAHalt();
  }
```

```
else
  {
    RUploadTexture(texID2, 8*8, Texture2);
  }
*/

//void RAmbColor(unsigned R, unsigned G, unsigned B, unsigned I);
//void RDirColor(unsigned R, unsigned G, unsigned B, unsigned I);

RSetLitInterp(1);
RDirColorRGBI(0xFFFFFFFF);
//RAmbColorRGBI(0xFFFFFF88);
//RAmbReflect(0xFF);
RDirLight(-1/sqrt(1.5), -0.5/sqrt(1.5), -0.5/sqrt(1.5));
RDirReflect(0xFF);
RSetLit(1);
RSetUseAmb(0);
RSetUseDir(1);
RSetTexInterp(0);


//RSetNoUseZ(1);
//RSetNoWriteZ(1);
//RSetOutOfOrder(1);

RSendMeta("pstart");

RBeginScene();

RSendMeta("pframe");

M[0][3] = 0;
M[1][3] = 0;
M[2][3] = 0;
RModelMatrix(M);
drawcube();

M[2][3] = -1.5;
RModelMatrix(M);
RSetTextureID(texID2);
drawcube();

M[2][3] = 1.5;
RModelMatrix(M);
RSetTextureID(texID);
drawcube();

M[0][3] = -1.5;
```

```
  M[2][3] = -1.5;
  RModelMatrix(M);
  drawcube();

  M[2][3] = 0;
  RModelMatrix(M);
  drawcube();                                              340

  M[2][3] = 1.5;
  RModelMatrix(M);
  drawcube();

  M[0][3] = 1.5;
  M[2][3] = -1.5;
  RModelMatrix(M);
  drawcube();
                                                           350
  M[2][3] = 0;
  RModelMatrix(M);
  drawcube();

  M[2][3] = 1.5;
  RModelMatrix(M);
  drawcube();

  REndScene();
                                                           360
  RMETAHalt();

  return 0;
}
                                                           370
```

D.8 texturetest.c

```
// texturetest.c
//
// Ken Taylor 5/18/04

#include "render_client.h"
#include <math.h>

int main(void)
{
  float Rot[4][4] = {1, 0, 0, 0,
                     0, 1, 0, 0,
                     0, 0, 1, 0,
                     0, 0, 0, 1}; // 15 deg rotation around z axis
  float Identity[4][4] = {1,0,0,0,
                          0,1,0,0,
                          0,0,1,0,
                          0,0,0,1};

  unsigned Texture[16*16] = {0xFF0000FF, 0xFF0000FF, 0xFF0000FF, 0xFF0000FF,
                             0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF,
                             0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF,
                             0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF,
                             0x00FF00FF, 0xFF0000FF, 0xFF0000FF, 0xFF0000FF,
                             0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF,
                             0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF,
                             0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF,
                             0x0000FFFF, 0xFF0000FF, 0xFF0000FF, 0xFF0000FF,
                             0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF,
                             0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF,
                             0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF,
                             0xFF0000FF, 0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF,
                             0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF,
                             0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF,
                             0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF,
                             0x00FF00FF, 0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF,
                             0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF,
                             0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF,
                             0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF,
                             0x0000FFFF, 0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF,
                             0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF,
                             0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF,
                             0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF,
                             0xFF0000FF, 0xFFFFFFFF, 0xFFFFFFcc, 0xFFFFFF88,
                             0xFFFFFF44, 0xFFFFFF00, 0xFFFFFFFF, 0xFFFFFFFF,
                             0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF,
```

```
        0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF,
        0x00FF00FF, 0xFFFFFFFF, 0xFFFFFFCC, 0xFFFFFF88,
        0xFFFFFF44, 0xFFFFFF00, 0xFFFFFFFF, 0xFFFFFFFF,
        0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF,
        0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF,          50
        0x0000FFFF, 0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF,
        0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF,
        0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF,
        0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF,
        0xFF0000FF, 0xFFFFFFFF, 0x00FF00FF, 0x0000FFFF,
        0x0000FFFF, 0x0000FFFF, 0x00FF00FF, 0xFFFFFFFF,
        0xFFFFFFFF, 0xFFFFFF00, 0xFFFFFF00, 0xFFFFFFFF,
        0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF,
        0x00FF00FF, 0xFFFFFFFF, 0x0000FFFF, 0xFFFF00FF,
        0xFFFF00FF, 0xFFFF00FF, 0x0000FFFF, 0xFFFFFFFF,          60
        0xFFFFFFFF, 0xFFFFFF00, 0xFFFFFF00, 0xFFFFFFFF,
        0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF,
        0x0000FFFF, 0xFFFFFFFF, 0x0000FFFF, 0xFFFF00FF,
        0xFFFF00FF, 0xFFFF00FF, 0x0000FFFF, 0xFFFFFFFF,
        0xFFFFFFFF, 0xFFFFFF00, 0xFFFFFF00, 0xFFFFFFFF,
        0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF,
        0xFF0000FF, 0xFFFFFFFF, 0x0000FFFF, 0xFFFF00FF,
        0xFFFF00FF, 0xFFFF00FF, 0x0000FFFF, 0xFFFFFFFF,
        0xFFFFFFFF, 0xFFFFFF00, 0xFFFFFF00, 0xFFFFFFFF,
        0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF,          70
        0x00FF00FF, 0xFFFFFFFF, 0x00FF00FF, 0x0000FFFF,
        0x0000FFFF, 0x0000FFFF, 0x00FF00FF, 0xFFFFFFFF,
        0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF,
        0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF,
        0x0000FFFF, 0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF,
        0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF,
        0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF,
        0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF,
        0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF,
        0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF,          80
        0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF,
        0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF, 0xFFFFFFFF };

double angle = (0.0/360.0)*(2.0*M_PI);
signed texID;
unsigned alpha = 0xCC;


RInit();
                                                                 90
RViewMatrix(Identity);

RSetNoUseZ(1);
```

```
RSetNoWriteZ(1);
RSetOutOfOrder(1);

texID = RAllocateTexture(16, 16);

if(texID == -1)
  {                                                          100
    RMETADebug();
    RSendMeta("texerr");
    //RMETAHalt();
  }
else
  {
    RUploadTexture(texID, 16*16, Texture);
    RSetTextureID(texID);
    //#define R_TEXMODE_NONE 0
    //#define R_TEXMODE_COLOR 1                              110
    //#define R_TEXMODE_TEXTURE 2
    //#define R_TEXMODE_BLEND 3
    //#define R_TEXMODE_TEXDECAL 4
    //#define R_TEXMODE_COLDECAL 5
    //#define R_TEXMODE_MODULATE 6

    RSetTexMode(R_TEXMODE_COLDECAL);
    //RColTexBalance(0.2);

    //#define R_TEXTILE_NONE 0                               120
    //#define R_TEXTILE_REPEAT 1
    //#define R_TEXTILE_MIRROR 2
    //#define R_TEXTILE_CLAMP 3
    //RSetTexTile(R_TEXTILE_CLAMP);

    //#define R_ALPHA_NONE 0
    //#define R_ALPHA_SOFT 1
    //#define R_ALPHA_HARD 2

    RSetTexAlpha(R_ALPHA_NONE);                              130
    RSetColAlpha(R_ALPHA_HARD);
    RAlphaThresh(0xDD);

  }

RSendMeta("pstart");

RBeginScene();

//while(1)                                                   140
//  {
```

```
    Rot[1][1] = Rot[0][0] =(float) cos(angle);
    Rot[1][0] = (float)sin(angle);
    Rot[0][1] = -Rot[1][0];

    RSendMeta("pframe");

    RModelMatrix(Rot);

    RBeginPrim();                                              150
    RSetTexInterp(1);
    RColor(0xFF, 0x00, 0x00, 0xFF);
    RVertex(0, 0, 0, 0, 1);
    RColor(0, 0xFF, 0x00, alpha);
    RVertex(0, -0.5, 0, 0, 0);
    RColor(0x00, 0x00, 0xFF, alpha);
    RVertex(0.5, 0, 0, 1, 1);
    REndPrim();

    RBeginPrim();                                              160
    RSetTexInterp(0);
    RColor(0xFF, 0x00, 0x00, 0xFF);
    RVertex(-0.5, 0, 0, 0, 1);
    RColor(0, 0xFF, 0x00, alpha);
    RVertex(-0.5, -0.5, 0, 0, 0);
    RColor(0x00, 0x00, 0xFF, alpha);
    RVertex(0, 0, 0, 1, 1);
    REndPrim();

                                                               170
    angle += (15.0/360.0)*(2.0*M_PI);
    // }

  REndScene();

  RMETAHalt();

  return 0;
}
```

D.9 ordertest.c

```
// ordertest.c
// tests proper sequential ordering constraints
//
// Ken Taylor 5/18/04

#include "render_client.h"
#include <math.h>

int main(void)
{                                                                           10
  float Rot[4][4] = {1, 0, 0, 0,
                     0, 1, 0, 0,
                     0, 0, 1, 0,
                     0, 0, 0, 1}; // 15 deg rotation around z axis
  float Identity[4][4] = {1,0,0,0,
                          0,1,0,0,
                          0,0,1,0,
                          0,0,0,1};

  double angle = (0.0/360.0)*(2.0*M_PI);                                    20
  unsigned alpha = 0x88;

  RInit();

  RViewMatrix(Identity);

  //RSetNoUseZ(1);
  //RSetNoWriteZ(1);
  //RSetOutOfOrder(1);
                                                                            30
  RSendMeta("pstart");

  RBeginScene();

  //while(1)
  //{
      Rot[1][1] = Rot[0][0] =(float) cos(angle);
      Rot[1][0] = (float)sin(angle);
      Rot[0][1] = -Rot[1][0];
                                                                            40
      RSendMeta("pframe");

      RModelMatrix(Rot);

      RBeginPrim();
```

```
RColor(0xFF, 0, 0, 0xFF);
RVertex(-0.8,-0.2, 0.0, 0, 0);
RVertex(-0.6, 0.0, 0.5, 0, 0);
RVertex(-0.8, 0.2, 0.0, 0, 0);
REndPrim();                                                      50

RBeginPrim();
RColor(0, 0xFF, 0, 0xFF);
RVertex(-0.7,-0.2, 0.0, 0, 0);
RVertex(-0.5, 0.0, 0.5, 0, 0);
RVertex(-0.7, 0.2, 0.0, 0, 0);
REndPrim();

RBeginPrim();
RColor(0, 0, 0xFF, alpha);                                       60
RVertex(-0.6,-0.2, 0.0, 0, 0);
RVertex(-0.4, 0.0, 0.5, 0, 0);
RVertex(-0.6, 0.2, 0.0, 0, 0);
REndPrim();

RBeginPrim();
RColor(0xFF, 0, 0xFF, 0xFF);
RVertex(-0.5,-0.2, 0.0, 0, 0);
RVertex(-0.3, 0.0, 0.5, 0, 0);
RVertex(-0.5, 0.2, 0.0, 0, 0);                                   70
REndPrim();

RBeginPrim();
RColor(0xFF, 0xFF, 0, 0xFF);
RVertex(-0.4,-0.2, 0.0, 0, 0);
RVertex(-0.2, 0.0, 0.5, 0, 0);
RVertex(-0.4, 0.2, 0.0, 0, 0);
REndPrim();

RBeginPrim();                                                    80
RColor(0, 0xFF, 0xFF, alpha);
RVertex(-0.3,-0.2, 0.0, 0, 0);
RVertex(-0.1, 0.0, 0.5, 0, 0);
RVertex(-0.3, 0.2, 0.0, 0, 0);
REndPrim();

RBeginPrim();
RColor(0xFF, 0xFF, 0xFF, alpha);
RVertex(-0.2,-0.2, 0.0, 0, 0);
RVertex( 0.0, 0.0, 0.5, 0, 0);                                   90
RVertex(-0.2, 0.2, 0.0, 0, 0);
REndPrim();
```

```
    RBeginPrim();
    RColor(0xFF, 0, 0, 0xFF);
    RVertex(-0.1,-0.2, 0.0, 0, 0);
    RVertex( 0.1, 0.0, 0.5, 0, 0);
    RVertex(-0.1, 0.2, 0.0, 0, 0);
    REndPrim();

    RBeginPrim();
    RColor(0, 0xFF, 0, 0xFF);
    RVertex( 0.0,-0.2, 0.0, 0, 0);
    RVertex( 0.2, 0.0, 0.5, 0, 0);
    RVertex( 0.0, 0.2, 0.0, 0, 0);
    REndPrim();

    RBeginPrim();
    RColor(0, 0, 0xFF, 0xFF);
    RVertex( 0.1,-0.2, 0.0, 0, 0);
    RVertex( 0.3, 0.0, 0.5, 0, 0);
    RVertex( 0.1, 0.2, 0.0, 0, 0);
    REndPrim();

    RBeginPrim();
    RColor(0xFF, 0, 0xFF, alpha);
    RVertex( 0.2,-0.2, 0.0, 0, 0);
    RVertex( 0.4, 0.0, 0.5, 0, 0);
    RVertex( 0.2, 0.2, 0.0, 0, 0);
    REndPrim();

    RBeginPrim();
    RColor(0xFF, 0xFF, 0, alpha);
    RVertex( 0.3,-0.2, 0.0, 0, 0);
    RVertex( 0.5, 0.0, 0.5, 0, 0);
    RVertex( 0.3, 0.2, 0.0, 0, 0);
    REndPrim();

    RBeginPrim();
    RColor(0, 0xFF, 0xFF, alpha);
    RVertex( 0.4,-0.2, 0.0, 0, 0);
    RVertex( 0.6, 0.0, 0.5, 0, 0);
    RVertex( 0.4, 0.2, 0.0, 0, 0);
    REndPrim();


    //angle += (15.0/360.0)*(2.0*M_PI);
    //}

REndScene();
```

```
  RMETAHalt();

  return 0;
}
```

Note: ignore extraneous percent signs in URLs, they were caused by a bug in the bibliography layout code.

References

[1] K. Akeley and P. Hanrahan. CS448A: Real-time graphics architectures (lecture notes). [Online document, accessed 20 May 2004], Available HTTP: http://graphics.stanford.edu/courses/cs448a-01-fall/, October 2001.

[2] F. Durand and B. Cutler. The graphics pipeline: Projective transformations. [Online document, accessed 20 May 2004], Available HTTP: http://graphics.csail.mit.edu/classes/6.837/F03/lectures/13_transformat%ions.pdf, October 2003.

[3] F. Durand and B. Cutler. Lecture notes for 6.837 fall 2003. [Online document, accessed 20 May 2004], Available HTTP: http://graphics.csail.mit.edu/classes/6.837/F03/lectures.html, October 2003.

[4] F. Durand and B. Cutler. Texture mapping & other fun stuff. [Online document, accessed 20 May 2004], Available HTTP: http://graphics.csail.mit.edu/classes/6.837/F03/lectures/13_transformat%ions.pdf, October 2003.

[5] F. Durand and B. Cutler. Transformations. [Online document, accessed 20 May 2004], Available HTTP: http://graphics.csail.mit.edu/classes/6.837/F03/lectures/04_transformat%ions.pdf, October 2003.

[6] Michael Gordon et al. A stream compiler for communication-exposed architectures. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, CA USA, October 2002.

[7] Ziyad S. Hakura and Anoop Gupta. The design and analysis of a cache architecture for texture mapping. In *Proceedings of the 24th International Symposium on Computer Architecture*, pages 108–120, 1997.

[8] H. Hoffmann, V. Strumpen, and A. Agarwal. Stream algorithms and architecture. Technical Memo MIT-LCS-TM-636, Laboratory for Computer Science, MIT, March 2003.

[9] L. Lamport. A new solution of Dijkstra's concurrent programming problem. *Communications of the ACM*, 17(8):453–455, August 1974.

[10] L. Lamport. The mutual exclusion problem, parts I and II. *Journal of the ACM*, 33(2):313–384, April 1986.

[11] T. Möller and Eric Haines. *Real-Time Rendering*. Ak Peters Ltd, second edition, July 2002.

[12] S. Molnar, M. Cox, D. Ellsworth, and H. Fuchs. A sorting classification of parallel rendering. *IEEE CG&A*, pages 23–32, July 1994.

[13] R. K. Morley and P. Shirley. *Realistic Ray Tracing*. Ak Peters Ltd, second edition, July 2003.

[14] J. D. Owens et al. Polygon rendering on a stream architecture. *2000 SIGGRAPH / Eurographics Workshop on Graphics Hardware*, pages 23–32, August 2000.

[15] G. L. Peterson. Myths about the mutual exclusion problem. *Information Processing Letters*, 12(3), June 1981.

[16] M. Segal and K. Akeley. The design of the OpenGL graphics interface. Silicon Graphics Computer Systems (unpublished), 1994. Available from `http://graphics.stanford.edu/courses/cs448a-01-fall/design_opengl.pdf` [accessed 20 May 2004].

[17] C. L. Seitz et al. The hypercube communications chip. Display File 5182:DF:85, Department of Computer Science, California Institute of Technology, March 1985.

[18] M. Taylor. btl debugging — for idiot savants. [Online document, accessed 20 May 2004], Available HTTP: `http://cag.lcs.mit.edu/raw/memo/19/btl-debug.html`.

[19] M. Taylor. btl extension — for jedi masters. [Online document, accessed 20 May 2004], Available HTTP: `http://cag.lcs.mit.edu/raw/memo/19/btl-advanced.html`.

[20] M. Taylor. The raw prototype design document, v5.00. [Online document, accessed 20 May 2004], Available FTP: `ftp://ftp.cag.lcs.mit.edu/pub/raw/documents/RawSpec99.pdf`, October 2003.

[21] M. Taylor et al. The Raw microprocessor: A computational fabric for software circuits and general purpose programs. *IEEE Micro.*, March 2002.

[22] M. Taylor, W. Lee, S. Amarasinghe, and A. Agarwal. Scalar operand networks: On-chip interconnect for ILP in partitioned architectures. In *Proceedings of the International Symposium on High Performance Computer Architecture*, February 2003.

6318-47