The Curl Graphics2d Immediate Mode Rendering API

By

Morgan McGuire

Submitted to the Department of Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements for the Degrees of
Bachelor of Science in Computer Science and Engineering
and Master of Engineering in Electrical Engineering and Computer Science
at the Massachusetts Institute of Technology

May 22, 2000 [June 2000]

Author [illegible]
Department of Electrical Engineering and Computer Science
May 22, 2000

Certified by [illegible]
Stephen Ward
Thesis Supervisor

Accepted by [illegible]
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

Title: The Curl Graphics2d Immediate Mode Rendering API
Author: Morgan McGuire
Advisor: Stephen Ward
Date: May 22, 2000
Degree: Bachelor of Science in Computer Science and Engineering and Master of Engineering in Electrical Engineering and Computer Science

The Curl Graphics2d API (Graphics2d) is a 2d immediate mode rendering API for the Curl internet content language. The unique capabilities of the Curl language enable the API to take the form of a graphics language rather than a simple library of routines. This graphics language offers a basis set of primitives and an extension mechanism that allows 2d and 3d graphics without sacrificing performance.

Graphics2d attempts to achieve minimalist simplicity by identifying fundamental concepts in 2d rendering and representing each of those concepts with a single interface class or method. These interfaces may be implemented by user (parties outside of Curl, possibly executing untrusted code) code to interact with other modules built to the API. However, user code will typically use the implementation provided with Curl to build graphics intense applications rather than extending the API directly.

As with any large API, the philosophy and crucial design points comprise the essence of Graphics2d, not the particulars of the order of arguments or name of a specific method. Therefore, rather than present an exhaustive list of entry points with some commentary on each one, in this document I describe the design goals and constraints of the Graphics2d API, key or demonstrative API design decisions and implementation architectures for it. The full documentation of the commercial implementation of the API is available from Curl Corporation.

The scope of the project is indicated by the size of the commercial implementation. Before macro expansion (much of the code base is machine generated by macros, an extremely dense coding practice), the commercial implementation has over 3281 entry points, including 295 classes with 2852 methods and 34 macros. It totals over 170 thousand lines of code in the Curl language.

Morgan McGuire has worked on computer graphics and imaging at Morgan Systems, the NEC Research Institute and the IBM T. J. Watson Research Center. He is currently employed by Curl Corporation as senior architect of the computer graphics group.

Contents

I. Introduction

Design Goals

The Curl[1] Graphics2d API (Graphics2d) is a 2d immediate mode rendering API for the Curl internet content language. Curl is different from other languages in that it is suitable for describing a broad range of content. Particular kinds of content that Curl is intended to support include (with currently popular languages for expressing that kind of content in parenthesis):

- marked up text (HTML)
- marked up data (XML)
- raw computation (C, Java[2])
- rapidly developed component based software applications (Visual Basic[3], Java)
- system scripts (Perl, SH, TCL)
- interactive, media-rich documents (JavaScript[4], DHTML, Shockwave[5])
- high performance media-rich games (C, C++)

By supporting all of these kinds of content within a single language syntax and single underlying API, Curl also enables new kinds of content to emerge. Because its task is so varied, Curl must be more sophisticated than other point solution languages than C or HTML. The compiler must be able to achieve the performance of a compiler-friendly language like C. The syntax must be simple, approachable, and easy to learn like HTML and VB. The support libraries must be high

---

[1] Curl is produced by and is a trademark of Curl Corporation, 400 Technology Square, 8th floor, Cambridge, MA 02139

[2] Java is a platform independent language that relies on byte code interpreters in web browsers for execution. Java is produced by and is a trademark of Sun Microsystems, 901 San Antonio Road, Palo Alto, CA 94303

[3] Visual Basic is the visual development version of the Basic programming language. It is used for rapid development of GUI applications for Microsoft Windows and is the most popular programming language today. Visual Basic and Windows are produced by and are trademarks of Microsoft Corporation, One Microsoft Way, Redmond, Washington 98052-6399

[4] JavaScript is a simple scripting language for web browsers that uses Java-like syntax. JavaScript is produced by and is a trademark of Sun Microsystems and is implemented by Netscape Communications Corporation.

[5] Shockwave is a Windows-only browser plug-in that displays animated content created in Macromedia Director using the programming language Lingo. Shockwave, Director, Macromedia, and Lingo are produced by and are trademarks of Macromedia, Inc., 600 Townsend St., San Francisco, CA 94103

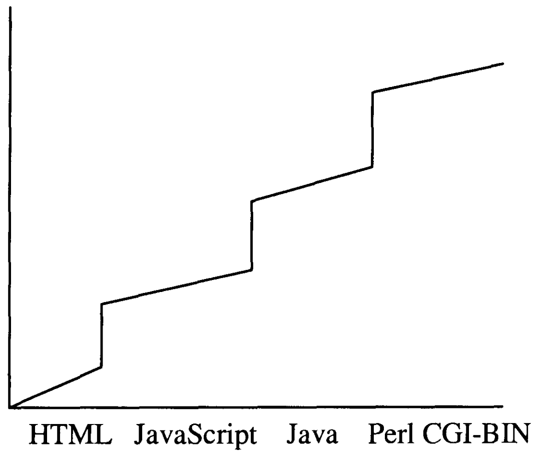
level and application oriented like Java, Shockwave, and Visual Basic. Syntax must be concise and fluid, allowing code to reflect the problem, not the language syntax.

A variety of technologies are available in Curl to meet these goals. Among them are a JIT compiler with a full "evaluate" function, parameterized types, an extensible syntax and compiler, garbage collection and explicit memory management, OOP support with full multiple inheritance, abstract classes, overriding, field accessor methods, closures, first class functions, and redirected fields (cascading property sheets). As a result, a library/API for the Curl language has a large number of requirements because of all of the target content, but may also make use of many tools. The Graphics2d API is a collection of compiler extensions, functions, methods, classes, and interface classes for enabling 2d retained mode graphics. A Curl compiler extension is a source to source transformation procedure that has access to compile time data (including compile-time data types and parameters to parameterized classes) and the power to create new compile time structures like data types. A compiler extension is also one of the ways to allow untrusted code to safely execute instructions at a more privileged level like unsafe memory accesses. A compiler extension may output code that has a more privileged access level if it determines that the conditions are safe (such as a raw memory access within a well bounded loop). Compiler extensions are also used to introduce new syntax into the language, perform targeted optimizations, and invoke code written in other languages.

The Graphics2d API must also exhibit the core design philosophy of the Curl content language. The following design goals exist from the union of good design practices, the content requirements, and the philosophic basis of Curl:

- Gentle slope
- Experimentalist
- User extensible
- High performance
- Platform independent
- Easy to understand
- Safe and secure
- Internet oriented
- Strong abstractions

The first three of these concepts, gentle slope, experimentalist, and user extensible, benefit from detailed explanation. The *gentle slope* philosophy of language design[6] involves smoothing the rough learning curve associated with mixed language programming. To develop content for the internet today, developers may begin in HTML. Learning new tags in HTML is simple, so the learning curve is steadily linear once the basic syntax is understood and developers begin to accumulate a base of HTML knowledge. When they have reached the limitations of HTML, JavaScript is required to add more functionality. But JavaScript programming has no similarity to HTML programming. Almost all of the syntax, conventions, and techniques learned during HTML programming are useless to a JavaScript programmer. So the learning curve has a sudden discontinuity, where a developer must stop and retrain for a new technology just to add a small amount of functionality. The first line of JavaScript can be more costly than any program the developer then writes in it because if may take a long time to learn the new language. This continues as more powerful but inconsistent languages like Java, C, and Perl are added into the mix. If a unified syntax, set of conventions and design paradigms were present through the continuum of content development languages, a developer would rarely face a discontinuity in the learning curve; it would form a gentle slope. A developer seeking to augment his or her skill set in the language will spend only incrementally more training time to gain incrementally more ability, rarely hitting a point where previous knowledge is inapplicable.



The Un-gentle Slope of the current Web developer learning curve

An *experimentalist* attitude is complements a gentle slope language. Many web developers developed their initial skill set by examining the source code of web pages and mutating it to produce their own. I attribute a large part of the success of the web, in terms of the huge amount of content produced in the recent past, to the conditions that promoted experimentalism. These include low cost of error (a broken HTML tag will not display, while broken C code may crash the operating system), wide availability of real example code (the entire web!), fast debugging cycle (reloading a page in a browser is almost instantaneous compared to compilation time for

[6] cite Ward et. al

most programs), and low cost of entry (editors and browsers are freely available). Curl should embrace all of these concepts to foster experimentalism, and extend them with more behaviors that are appropriate for a language: helpful error messages (error messages should be constructed to anticipate the problem and send the developer in the right direction), simple APIs (there should be a single interface to each major concept), small null programs. Small null programs means that the boilerplate code required to set up a program should be kept to a minimum (i.e. the "null program" should be tiny). This is a serious concern in modern programming. Microsoft Visual C++, the preferred tool for sophisticated development for the Windows platform, requires over a hundred lines of complicated code to open an empty window and receive events. Initializing the companion Direct X high performance graphics library may take another five hundred lines of detailed code. In contrast, HTML and JavaScript require almost no boilerplate code whatsoever before content and have few "magic incantations" that programmers must memorize. Finally, experimentalist APIs should act like good teachers or homework assignments: by their very nature, they should inspire developers to ask "what if I do this?" and subtly guide developers in the right direction. Functionality is useless if users can't get access to it. One hindrance to gaining access is often documentation. Documentation is essential to an API, but it is unrealistic to expect a developer to read and comprehend all of the documentation before using the API. Instead, a small piece of sample code should encourage programmers to try and mutate it to find new functionality.

A *user extensible* API provides a set of interfaces for communication and allows user code, as well as system code, to implement those APIs. Sufficient tools should be provided so that user implementations can approach the performance of internally developed implementations. User extensibility is important because it allows developers to harness the full power of the system and use it in ways the designers did not originally envision. For example, the Curl Graphics2d API has an interface called `Renderer` that is used as a drawing target. Typically, user widget code is given a renderer in its paint method so that it can render a visual interface onto the screen. Because the API is user extensible, a non-Curl Corporation developer can create a class that implements the `Renderer` interface and supply it to a 3rd party widget to discover how it draws itself. This functionality has been demonstrated to enable the rendering of widgets in contexts not originally envisioned, such as to the texture map of a three dimensional surface inside of a virtual reality world. Also, users can create their own alpha blending operations and image filters and use them to extend existing `Renderer` implementations.

To summarize, the goal of the Graphics2d API is to create a 2d rendering language and engine suitable for diverse content that is user extensible, high performance and exhibits other properties of good design. The structure of the API is a series of interfaces for fundamental graphics concepts. These interfaces allow user-defined classes to communicate with each other for purposes of exchanging graphic data. Specific implementations of those concepts can respond to queries (via method invocations) to describe themselves to unknown objects, thus enabling user extensibility and platform independence through strong abstractions. The primary goal is not to provide rendering functionality, but interfaces. That the API is provided with implementations to allow rendering to the screen, printer, and off-screen images is a secondary concern.

Graphics hardware trends

The theoretical approach to graphics ensures that the Graphics2d API will not quickly become obsolete. In order to provide compatibility with accelerator hardware both today and in the future, it was essential to keep the properties of current hardware as well as general trends in computer graphics in mind during the design and implementation of the API.

Before addressing some of those trends and presenting some criticism of the current direction of graphics hardware, it is appropriate to note the incredible level of realism current graphics hardware paired with current software techniques make possible as compared to the systems of a decade ago. New algorithms and hardware advances have made it possible to support in real time a level of graphics support that was previously restricted to the domain of offline photorealistic rendering systems. These include reflections, transparency, perspective correct texturing, parametric curves, perfect hidden surface culling/clipping, and rendering of very large worlds.

A number of short term trends have dominated the realtime PC computer graphics space and the process of optimizing code for that target. For the most part, these trends are now a part of history and not issues that need to be considered in a modern API. They have, of course been replaced with new trends and issues. It is important to begin by debunking some misplaced conventional wisdom and explaining the reality of the current situation.

Floating point data representations have been traditionally avoided in favor of integer-operation based fixed point representations because integer operations were typically faster. This introduced complexity into code, difficult to handle overflow cases, and additional instructions to handle fixed point multiples and float to integer conversion, but it resulted in a net performance increase. This is no longer the case. With multiple pipelines in today's super scalar processors, not only are floating point operations near the cost of integer operations, a net speed increase can be obtained by using floating point over integer operations. This is because in a system where multiple FPUs and ALUs are available, using a FPU for math operations frees the ALU for other operations like loop indexing and bounds checking. Using floating point all the way down into the rasterizer makes code simpler, reduces bugs, and does not affect performance negatively[7].

Early graphics hardware accelerators provided a small amount of on-card memory for storing sources (textures). With the AGP bus that allows graphics hardware to communicate directly with memory, texture memory is now effectively limitless, because main memory may be used.

The expensive operation on today's hardware is a graphics accelerator pipeline stall. Accelerator cards are typically clocked at a lower rate than the main processor and have pipelines that may be hundreds of stages. Any operation that causes a pipeline flush (like reading from the destination buffer) can take up to milliseconds.

The amount of data moving around on a graphics card is staggering. Current data rates are measured in megapixels. Running an application at 1600x1200 resolution at 80 frames per second is considered a realistic, if aggressive, performance target. This is a data rate of 196,608 megapixels per second! At this rate it is simply impossible to communicate per-pixel data to a graphics card with modern bus speeds. Instead, hardware is taking over more and more functionality from the main processor. Perspective correct single-pass multitexturing, geometry transformations, and illumination are all being computed on graphics cards. Unfortunately, in the quest for the ultimate pixel rate, graphics cards are taking over this functionality but limiting the developer's ability to control more than a few parameters of the rendering algorithms. This leads to a largely homogeneous set of applications as well as a severe limitation on graphics innovation in software. Some of this process is necessary; certain algorithms will be standardized and form the basis set of operations. However, limiting the geometry description and transformation

[7] Fixed point representations are still essential for controlling precision, however. 64-bit floating point resolves most precision problems (52 mantissa bits is a lot) but the storage demands can be significant, impacting cache behavior, and thereby hurt performance.

capabilities, shading options, PVS determination and texturing methods is not a healthy trend, and will eventually backfire because it is too close to the current breed of applications and prevents new application models from evolving. For example, current virtual reality games rely on the main processor for their PVS computations and some rendering because their algorithms are substantially different from what cards can provide. A limited set of accelerated functionality will hinder the handling of new video compression forms and increasingly larger and more complicated 3d scenes.

General purpose graphics languages are an increasingly important area of research. Areas that are already receiving commercial attention are shading languages and texturing languages (procedural textures). These will replace our current concept of multi-pass rendering or single-pass multi-texturing. As this becomes a reality, graphics hardware data formats will become even more highly differentiated (i.e. incompatible). Parameterized shape (curve) rendering will become common.

In order to provide high performance for general programming of graphics units, including texturing and shading, it is necessary that "Turing complete" graphics processors will be accessible to programmers with DSP instructions and branching logic. Right now, programmers only have access to the API layer for high-level procedures like "draw polygon" and setting certain complex registers/stacks like the "perspective matrix register." As more processing moves onto the graphics processor and reprogramable gate arrays become available (FPGAs, RAM processors, etc.), and immediate mode graphics languages are necessary, it makes sense that graphics hardware will make the full instruction set available and it will be possible to download code as well as data onto a graphics card. I will return to this topic in the conclusion and reflect on how the Curl API fits into such a world as well as current technology trends that I believe will foster the graphics hardware revolution.

Despite the high powered graphics machines I have defined as today's PC target, underpowered devices exist and will be with us for some time. Although the graphics power of the PC and set-top consoles has grown incredibly, more and more underpowered devices are adding graphics capabilities. Many PDAs have the graphics limitations that PCs had a decade ago, but developers would like write-once run-everywhere code to work on both PCs and PDAs. Complicated graphics displays are appearing everywhere — even on gas pumps and cash registers. The combination of the growing popularity of portable and wearable devices and the need for clear, information dense, and engaging display interfaces creates an increased the need for sophisticated graphics APIs for underpowered devices, as well as on the incredibly powerful PCs and game consoles.

Conventions

The following conventions are used in this API and document.

COMMENTS

Comments in Curl are denoted by a double bar "||". Comments and inline documentation (contained in a {doc-next ...} block) always precede the code they refer to.

USERS AND SECURITY

A *user* or *developer* is a programmer writing code in Curl who does not work on the developing Curl language itself. An *end-user* runs applications written in the Curl language. *Untrusted code* is Curl code that is of either of unknown origin (and therefore, potentially malicious) or does not have any requirements to invoke potentially dangerous methods, and therefore can (or must) be executed in a mode where it is unable to do serious harm to the end-user's computer. This includes *safety* and *security* restrictions. A safety restriction prevents untrusted code from violating the language semantics or crashing/corrupting the computer. Raw memory (i.e. not bounds checked) accesses and illegal casts are examples of potential safety violations. A security restriction prevents access to routines that may violate the end-user's privacy or be used maliciously. Unrestricted file and network access are examples of potential security violations. Often, a safety violation can be exploited to violate security. *Privilege* is the level of access code is given to routines that have the potential to violate safety or security. A *privileged* routine is one that may only be invoked under certain circumstances and is not generally callable. This mechanism is often used in Graphics2d to allow a compiler extension to execute code that the developer who invoked that extension does not have privilege to access.

It is considered good practice to write code so that it may be executed at the lowest (most restricted) privilege level possible for the functionality that code contains. A lot of effort was put into enabling untrusted code to perform fast graphics using Graphics2d. This area of research has not been widely represented in the literature of computer graphics. However, the rapid growth of high level languages in commercial practice, widespread access to the internet and high performance graphics hardware, and general interest in (potentially untrusted) component based software design will lead to more effort in this area in the future[8].

RENDER/DRAW/PAINT

The following verb terminology is used in the API names for the act of "drawing":

Render A onto self

Draw A onto B

Paint self onto B (as in VB, Windows, and Java's PAINT methods)

This terminology is used more loosely (and somewhat interchangeably) in the text.

OUT ARGUMENT

To avoid excessive memory allocation, many routines take an argument that may be used as storage space for the result. This argument is called out. Mutation of the out argument should not be depended on; always check the return value.

TYPESETTING

Language names are formatted in `large, fixed type`. Code examples are formatted in `small fixed type`. Key words and new terms immediately preceding their definitions are *italicized.*

Outline of this Document

In this introduction I presented the design goals and constraints of the Curl Graphics2d API, general industry trends that influenced the API design, and conventions used in the naming and presentation of the API. The following section, *Structure of Curl Graphics Packages*, describes how the Curl Graphics2d API fits into the Curl language API as a whole. The *API Overview* presents the API from a high-level, discussing how it represents fundamental concepts in computer graphics and how the parts interact. The *Interfaces* section describes each programmatic interface in detail, giving design justification for specific points in the architecture

[8] Curl is a very unique web content solution in this regard. The current graphics solutions for the web all fail to achieve the combination of safe and fast with any kind of generality. These include Java (safe but slow), Shockwave (moderately safe and moderately fast, but extremely limited), and DirectX based solutions (completely unsafe but fast).

and describing the major data structures employed in an efficient implementation of the API. The *Conclusion* section reflects on the API as a whole as well as the commercial implementation of the API produced at Curl Corporation, and discusses new directions for computer graphics. The appendices include real world examples of the API in use as well as the documentation from a prerelease version of Curl. All code examples are written in the Curl language unless otherwise noted.

Acknowledgements

My advisor and friend, and CTO of the Curl Corporation, Steve Ward, contributed greatly to the ideas contained in this thesis and made it possible to implement it in a real world setting.

The members of the Curl Graphics Team:

Aaron Orenstein
Ben Landon
Damian Frank
Paul Metzger
Jonathan Zarge
Leonard MacMillan (consultant)

were instrumental in providing guidance and feedback as well as working with me to reduce theory to practice. Ben Landon is responsible for the implementation of the `Region` hierarchy, particularly the difficult intersection and union code. The `ControlPointBasis` concept and implementation are due to Aaron Orenstein. David Kranz, Mat Hostetter, Bert Halstead and the rest of the Curl team also participated in the implementation of the API and exerted influence over sections of the design. In particular, David Hollingsworth and I collaborated on the first draft of the Renderer API and many of the key ideas there are a result of his input. David Kranz, Mat Hostetter, Dan Nussbaum, and Aaron Orenstein collaborated with me to implement the unit system described in the Renderer section.

I first worked with an extensible graphics language for the internet on the Laser Graphics Language project with Laura Wollstadt at Morgan Systems. Many aspects of Laser inspired the Curl 3d and 2d APIs. I am grateful for her input.

Jim Miller, Richard Kelsey, Steve Adams, and Olin Shivers gave advice and guidance with Laser 1.0. Seth Teller taught Course 6.837: Computer Graphics at MIT and encouraged my early ray tracer work, thereby inspiring Laser 2.0.

II. Structure of Curl Graphics Packages

Graphics2d is only one of a layered structure of Curl graphics APIs. It is the general intent of the Curl graphics architecture that multiple self-complete yet compatible APIs will exist for graphics, and that in general a user will write to the level of the highest (most abstract) API that meets his or her needs, dropping down to lower levels only when it is necessary to break the abstractions present at that level. At the highest level, Modeling APIs are extremely application oriented and allow users to instantiate and combine complex predefined objects to rapidly develop application functionality. It is also possible for users to create their own object classes that conform to the modeling interfaces. In order to provide a lot of out of the box functionality, these Modeling APIs must make assumptions about the kinds of graphics tasks that will be undertaken by applications. When these assumptions are not valid for a certain application, it is necessary for a developer to augment the Modeling API with a lower level API. This process also defines a gentle slope for user education. Developers may begin by building 3d worlds and 2d user interfaces from provided modeling classes, then tackle raw retained mode and immediate mode only when that level of sophistication is needed. Their knowledge of the high level Modeling APIs is still useful because all of the APIs are compatible, so Modeling level objects may be combined with lower level objects. The Modeling API is **not** a "learner API" or "unsophisticated API" that must be abandoned to get serious work done. A sophisticated developer will still choose do as much work as possible at the Modeling level and will only dip down to lower levels when required. This is somewhat analogous to assembly language programming versus C programming versus C++ programming. C++ has much more functionality than C, so sophisticated developers work at that level. It is sometimes necessary to augment a C++ program with C techniques because C++ is more application targeted and discourages certain C paradigms. When even C will not suffice, advanced programmers will write assembly code by hand. Dropping down a level means forgoing certain kinds of language support and writing much more code (thereby opening oneself to bugs and maintenance problems), however it provides access to a more general interface that has the capability to be more expressive if used correctly.

The Modeling APIs are retained mode and assume a retained mode context that provides for object interactions. For example, the GUI Toolkit, the 2d Modeling API, allows users to create widgets and connect them to each other with very few lines of code. These widgets have built in assumptions about a containing dialog or document context, windowing system, and UI gestures.

The Retained Mode APIs provide basic scene graph capability and retained mode functionality. Users can construct objects that will automatically refresh their on-screen image, and may relate them to one another in traditional scene graph hierarchies containing transformation nodes and other properties. Interfaces for event handling, but not event delivery (which is left to the user) appear at this level.

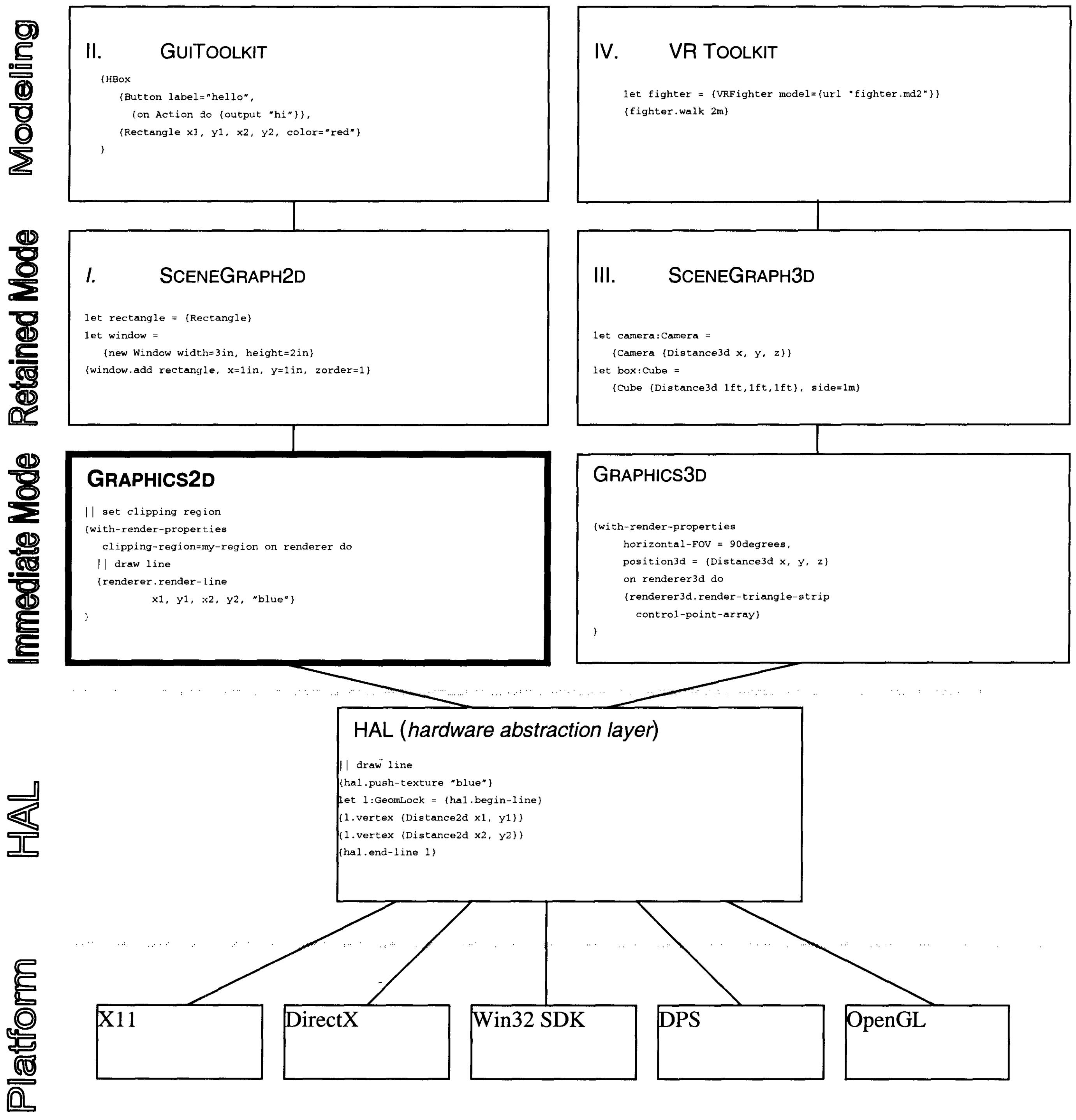
Immediate Mode APIs allow for direct, non-persistent rendering calls. They provide imperative interfaces to clipping regions, transformation matrices and other rendering state and establish interfaces and languages for 2d and 3d graphics. The Graphics2d API described in this document is at this layer. Although Graphics2d can be used for 3d rendering, Graphics3d is better suited for that task because it moves higher up the 3d graphics pipeline to perform projection and 3d clipping itself, simplifying 3d immediate mode rendering at the expense of some generality.

The Hardware Abstraction Layer API provides a virtual hardware interface with matrix stacks, texture registers, and the like. It uses no high level abstractions for geometry and provides only minimal (rectangle) clipping support. The Graphics2d guarantees of renderer state integrity are not preserved at this level. The HAL is a Curl substitute for interfaces like OpenGL[9] and DirectX, intended to provide an alternative and improved interface to those APIs. Untrusted users may be denied access to the HAL because security and safety can be violated at this level (consuming excessive system resources, reading information that other components rendered to the screen, rendering outside of restricted bounds). The HAL must be reimplemented as part of the process of porting Curl to new platforms.

The Platform APIs are an implementation detail and are not available to users outside of the Curl system at all. These APIs trampoline various external graphics APIs into Curl so that the HAL may be implemented using all Curl code without linking to external API back ends.

[9] OpenGL is SGI's industry standard low level graphics API. OpenGL is a trademark of 1600 Amphitheatre Parkway, Mountain View, CA 94043. See http://www.opengl.org for more information.

Conceptual Structure of Curl Graphics Packages



```
{HBox
   {Button label="hello",
     {on Action do {output "hi"}},
   {Rectangle x1, y1, x2, y2, color="red"}
}
```

```
let fighter = {VRFighter model={url "fighter.md2"}}
{fighter.walk 2m}
```

```
let rectangle = {Rectangle}
let window =
   {new Window width=3in, height=2in}
{window.add rectangle, x=1in, y=1in, zorder=1}
```

```
let camera:Camera =
   {Camera {Distance3d x, y, z}}
let box:Cube =
   {Cube {Distance3d 1ft,1ft,1ft}, side=1m}
```

```
|| set clipping region
{with-render-properties
   clipping-region=my-region on renderer do
  || draw line
  {renderer.render-line
          x1, y1, x2, y2, "blue"}
}
```

```
{with-render-properties
     horizontal-FOV = 90degrees,
     position3d = {Distance3d x, y, z}
     on renderer3d do
     {renderer3d.render-triangle-strip
       control-point-array}
}
```

```
|| draw line
{hal.push-texture "blue"}
let l:GeomLock = {hal.begin-line}
{l.vertex {Distance2d x1, y1}}
{l.vertex {Distance2d x2, y2}}
{hal.end-line l}
```

III. API Overview

The previous section described the relationships between all of the graphics APIs in Curl. In this section, the Graphics2d API is introduced and the major interfaces are described.

Graphics Concepts

Designing a graphics language appropriate for all kinds of 2d rendering requires establishing the basis set of 2d graphics concepts. The Curl 2d graphics system recognizes six fundamental concepts that must be represented to present a basis set on which all higher level graphics ideas can be represented. These are fundamental because all 2d rendering can be expressed using only these concepts but none of the concepts can be expressed in terms of each other. Each concept is represented by one or more interface classes in the API. These concepts are (with interface classes denoted in parentheses):

• Source	(`Texture`)
• Destination	(`Renderer`)
• Geometry	(`Region`, also `Distance2d`, `Path`, `Font`)
• Drawing Operation	(`DrawOperation`)
• Interpolation	(`ControlPoint`, `ControlPointBasis`)
• Communication medium	(`Pixmap`)

One major contribution of the Curl graphics system is the presentation of a single interface for each concept. This makes the API easy to learn. Each of the fundamental concepts of rendering is represented by an interface. Once a user is familiar with the idea and role of a destination, he or she will find Renderer to be a straightforward interface to a destination. A simple explanation of the role of each concept is:

Geometry describes what to draw. A destination is where you draw it. A source provides the texture to fill the area. A drawing operation describes how to combine the new image with the existing background. Various properties, like color and 3d depth, are interpolated across the surface when rendered. Pixmap allows interfaces to have a single format to use when communicate between themselves and with user code about large numbers of pixels.

Using a single API for each rendering concept, as opposed to alternative APIs depending on the nature of the source, destination, etc., also allows polymorphic rendering functionality. For example, in Curl a widget has a single method to render itself, regardless of the target. When rendering to the screen, an off screen target, a printer, or an atypical output device like a plotter, 35mm slide processor, or T-shirt printer, exactly the same rendering calls are made. In fact, the widget is often unaware of the format of the actual rendering target. It is therefore a necessity that all interfaces are independent of device particulars including resolution, color depth, and color space. To allow hinting and tweaking, however, a renderer can be queried for these properties.

One weakness of many prior graphics APIs is that they offer complete device independence or querying, but not both. APIs that are completely device independent, including Java Graphics2d, OpenGL, and X11, prevent developers from determining when they are invoking a routine that will be emulated in software rather than hardware, and can have performance problems as a result. APIs that offer only querying and do not mandate support of all functionality by implementations, such as DirectX, can guarantee performance but require developers to write complicated querying and initialization code for simple situations. Curl requires that all implementations implement the full API, however it also provides device capability querying so that performance conscious developers can avoid routines that are implemented via software emulation of hardware functionality.

The presentation of single interfaces for sources, destinations, geometry and other graphics concepts allows developers to create libraries such as the image filter library which efficiently implement graphics algorithms but present a plug-and-play component interface. This is a well understood system programming technique, but has been slow to be adopted in the real time graphics community due to the extreme performance demands that group faces[10]. The Curl graphics API is able to present high level abstractions without sacrificing performance due to a mixture of newly available technologies: the extensible Curl compiler, new graphics algorithms, and the fact that inexpensive high performance graphics hardware is now widely distributed.

[10] Real time graphics programmers inhabit a world where linear time algorithms are too slow because data sets contain millions of objects, hardware pipelines have hundreds of stages that can take milliseconds to stall, megabytes of data must be moved at rates and profile times are discussed in cycle counts.

To allow untrusted code to perform pixel level operations safely but efficiently, a series of building blocks are provided for image/pixmap manipulation:

• 2d arrays suitable for vectorization	`(Map-of, NumMap-of, FixMap-of)`
• Image transformations	`(ImageFilter)`
• Iterators	`(for)`
• Conversion methods	`(to-Pixmap, to-Texture)`

API Analogs

A *source* is an immutable producer of texture data. Typical sources are gradients, texture maps and solid colors. "Texture" and "source" are common terms for sources in other graphics systems. The `Texture` class defines an interface through which consumers of the texture can express queries as method invocations. Textures model a mathematical two-dimensional color function that has infinite resolution. Sources are immutable, specifically disallowing the possibility of having a source that is also a destination.

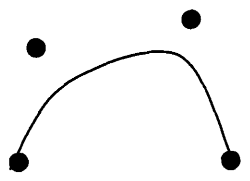
A *destination* is a target that can be rendered onto, the counterpart to a source. Destinations present a write-only interface to the objects that draw on them. "Target," "context," and "on/off-screen bitmap" are some terms that other graphics systems use for destinations. The `Renderer` interface is used for all destinations in Curl. It provides two sets of methods, *fundamental* and *derived.* The fundamental methods take a minimum number of arguments and render the four kinds of immediate mode primitives: paths, regions, pixmaps, and text. These are the methods that must be implemented by a subclass of `Renderer`. The derived methods are by default implemented in terms of the fundamental ones. A subclass implemented using a back end that is capable of accelerating the derived methods may override these methods for added performance. The `with` compiler extension is used to set properties on the renderer such as the current drawing texture.

Geometry is a mathematical description of a shape. A geometry interface is necessary to allow different parts of the graphics system to communicate about the shapes to be rendered. The `Region` interface describes finite 2d areas (which may be discontinuous) to be filled or used as clipping regions. Multiple implementations of Region are provided, including `PolygonRegion`, `ConvexPolygonRegion`, `RectangleRegion`, and

EllipseRegion. Path describes an infinitely thin (possible discontinuous) directed 2d mathematical curve. Font describes the geometry of a set of text characters. Points, vectors, and distances are described using the built-in unit (symbolic analysis) system for handling vectors and scalars with real world units like distance. The Distance data type uses floating point encoding to store a distance, and the compiler performs unit checking to verify that instances of it are only added to, subtracted from, and assigned to other distance variables. When two variables are multiplied, divided or otherwise mathematically operated on, a new result type is derived with the appropriate units. Distance2d is the 2d vector version of Distance. Fraction and Fraction2d indicate a unitless value normalized to the range [0, 1] (these are useful for the parameter in linear interpolation as well as texture coordinates). Float and Float2d are general unitless numbers. Area is the type of the product of two Distance2ds, Resolution is the type of the reciprocal of a Distance, and so on. Special compiler support allows the benefits of unit checking (described in the *Region* subsection of the *Interface* section) without additional runtime cost.

A *drawing operation* is an equation that determines how to combine the existing destination pixels of an image with source data to produce a new image on the destination. Paletted (indexed color) graphics systems historically used bitwise operations (AND, OR, XOR, etc.) to loosely approximate effects like translucency, transparency, and inversion. Modern graphics systems provide "alpha-blending" modes that combine source and destination pixels in more complicated ways based on true-color operations rather than bitwise manipulation. The DrawOperation interface provides a user-extensible framework for specifying arbitrary operations that combine source and destination pixels while rendering.

Control points are locations in space with associated properties that are used to describe geometry. As shown in the diagram on the right, the control points may not actually be contained within the geometry itself. To describe properties that may vary across geometry, properties such as color, depth (z), and texture coordinates can be specified at each control point. The ControlPoint interface provides a user extensible framework for specifying such properties and allowing those properties to be interpolated across a region during clipping and rendering. Note that four or more control points with independent values of any property lead to an


A Bezier Curve and its Control Points

ambiguous interpolation space. It is necessary to establish an unambiguous value for any property at all points in space. If this is not done, subsequent renderings will exhibit different properties after clipping or rotation, a highly undesirable result for animations. Users are therefore required to provide control points that exhibit a consistent property field over all 2d space. The `ControlPointBasis` interface assures this by examining an array of `ControlPoints` and producing a consistent function over x and y for each interpolated property. Renderers take these bases as arguments rather than raw control points and can therefore be assured of having consistent properties.

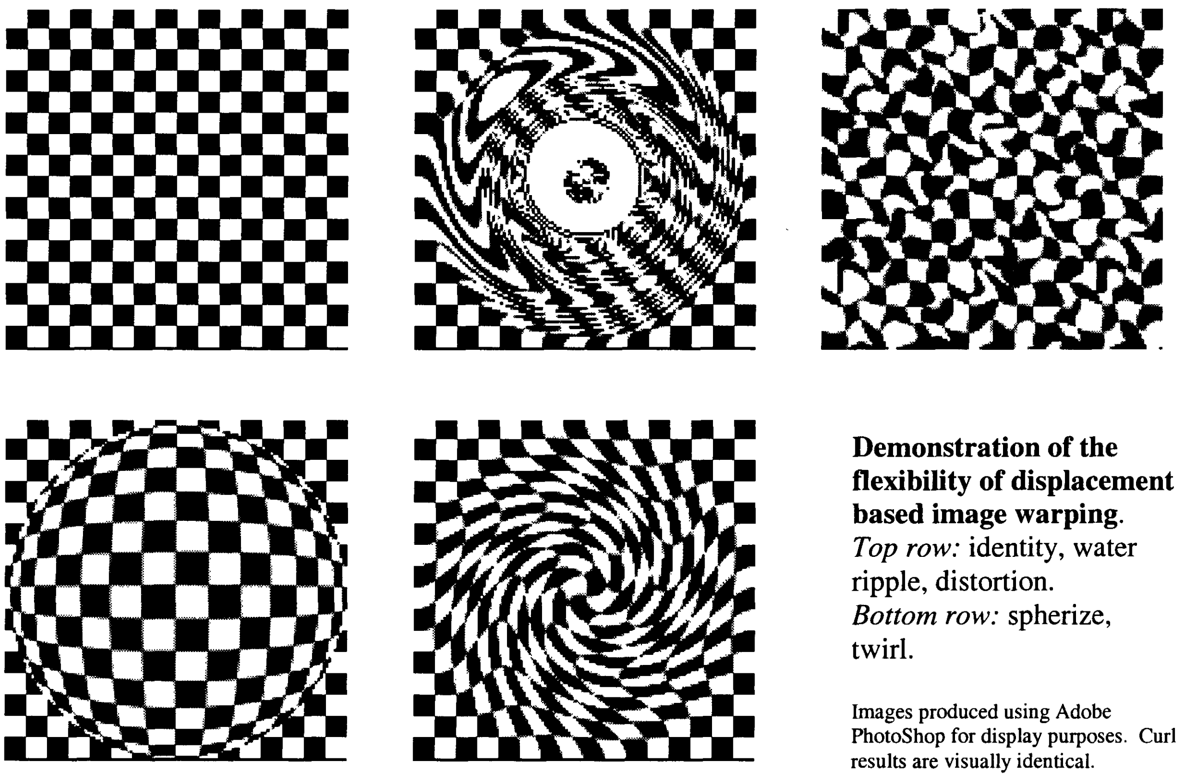
The `Color` subclass of the `Texture` interface provides a device independent medium for expressing color data. However, because it is immutable and somewhat heavyweight (`Color` stores color space information and encodes all color data in floating point) it is too inconvenient and inefficient to serve as a *communication medium* between interfaces. The `Pixel` class serves that purpose. It contains four channels storing fractions with a resolution of 1/256. These are named alpha, red, green, and blue and are typically used to represent a device dependent color with those channels in the W^{3}C *sRGB* color space. `Pixmap` is a two dimensional array of `Pixels`. It is a specific parameterization of the more general `Map-of` construct, which is a parameterized 2d graphics array. Other subclasses of Map-of like `NumMap-of` and `FixMap-of` work with `Pixmap` to provide a vectorized image processing language subset similar to the scientific programming language Matlab[11].

The `for-map-element` compiler syntax (sometimes referred to as "for-pixel") allows untrusted code to efficiently iterate over a subregion of a pixmap or other `Map-of`. It is specially designed for graphics, using the same rasterization rules as the renderer interface, allowing iteration over parallel maps, nonrectangular iteration bounds, and supporting parallelization by not guaranteeing the iteration order. `for-map-element` also safely lifts bounds checks, allowing untrusted code to safely execute raw memory accesses.

In addition to this extension, other building blocks are available to users to allow fast, safe and simple pixel manipulation code. These include `Spectrum` and `DisplacementMesh`.

[11] Matlab is produced by and is a trademark of The MathWorks, Inc., 3 Apple Hill Drive, Natick, MA 01760-2098

`Spectrum` supports paletted (indexed color) graphics by providing a lookup table that is capable of mapping a `FixMap` (fixed point numeric 2d array) to a pixmap. Paletted graphics are less popular in modern systems, but are still essential for certain graphic effects. `DisplacementMesh` provides morphing capabilities for all `Map-of` parameterizations. It represents a two dimensional grid of connected control points. Through a programmatic interface, the control points can be displaced from the regular grid. This distorted grid can then be used to spatially map the elements of `Map-of` to another. Many image warping effects can be implemented using this functionality. Some examples are shown below:



Demonstration of the flexibility of displacement based image warping.
Top row: identity, water ripple, distortion.
Bottom row: spherize, twirl.

Images produced using Adobe PhotoShop for display purposes. Curl results are visually identical.

Because Pixmap is the primary image data exchange format of the graphics system, most classes support a `to-Pixmap` method (when applicable). The Graphics2d API provides an image filter interface. Although this does not operate on pixmaps directly, by turning pixmaps into textures and vice versa is possible to apply these image filters is to pixmap data.

The `Spectrum`, `for-pixel` extension, `DisplacementMesh` and `ImageFilter` building blocks are not intended for use as standalone objects. Many interesting pixel-level animations can be performed safely and efficiently by untrusted code using combinations of these building blocks, however. As such, they represent a simple pixel-level graphics sublanguage within Graphics2d. No high-level pixel manipulation language with this level of power, safety, and simplicity is currently available to developers. Many graphics APIs, like OpenGL and DirectX, do not provide pixel manipulation facilities at all. As an integral part of the Curl platform independent web based content language, the pixel processing routines are a major feature and new level of web functionality.

IV. Interfaces

In this section, I describe each of the major interfaces of the API. Specific design decisions are called out and implementation details are given to demonstrate the techniques that led to an efficient implementation of the API.

Texture

`Texture` is an interface for sources. It is required that all classes that implement this interface are immutable so that sharing of texture data may safely be undertaken. It initially appeared that this restriction might not be necessary given careful dependency tracking and very clear indications of when copying would and world not occur. However, without the guarantee of immutability, it proved prohibitively difficult to keep the API simple and achieve efficient implementation using graphics hardware. Effective implementation of copy-on-mutate semantics within a language may eliminate the need for this restriction.

Texture supports sampling methods with three levels of (subjective) quality, accessors that indicate the resolution of the texture, accessors indicating the wrapping modes, and a method that converts the texture to a pixmap. For most real time rendering, renderers convert textures to native data formats via the `to-Pixmap` method, then perform standard texture mapping using that native texture. Most of the implementation of textures is involved in making that process efficient. As hardware becomes more capable of handling large textures and algorithmic textures, the texture interface will scale well. Multi-texturing, the process of combining multiple basic textures to provide a richer set, is likely to be the first case accelerated, but full procedural textures may be possible in the future.

```
{define-class public abstract Texture
  {getter abstract {friend-of Renderer} {device-table}:DeviceTable}
  {getter public {uniform?}:bool}

   || infinite resolution?
  {getter public {continuous?}:bool}

   || usually nearest
  {getter abstract public {get-fast u:Fraction, v:Fraction}:Pixel}

   || usually bilinear
  {getter abstract public {get-good u1:Fraction, v1:Fraction}:Pixel}

   || usually bilinear
  {getter abstract public {get-best u1:Fraction, v1:Fraction,
                                    u2:Fraction, v2:Fraction}:Pixel}
  {getter abstract public {horizontal-texture-mode}:WrapMode}
  {getter abstract public {vertical-texture-mode}:WrapMode}
```

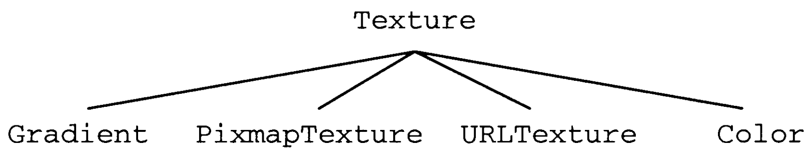
```
    {getter abstract public {max-horizontal-samples}:float}
    {getter abstract public {recommended-horizontal-samples}:float}
    {getter abstract public {min-horizontal-samples}:float}
    {getter abstract public {max-vertical-samples}:float}
    {getter abstract public {min-vertical-samples}:float}
    {getter abstract public {recommended-vertical-samples}:float}
    {getter abstract public {quality}:Quality
    {method abstract public {to-Pixmap out:Pixmap=null,
                        width:int=self.recommended-horizontal-samples,
                        height:int=self.recommended-vertical-samples}:Pixmap}
}

{define-enum public WrapMode clamp, tile}

|| Textures can't defer their quality setting
{define-enum public Quality fast, good, best, defer}
```

Texture Interface

Several implementations are provided in addition to the interface. `Gradient` allows creation of simple 2d linear color gradients as are found in popular business presentation graphics packages. Because gradients are stored as color points in space, not images, they have infinitely fine resolution and take little space. `PixmapTexture` and `URLTexture` are image textures,



Inheritance diagram for textures

initialized respectively from a Pixmap and URL. The primary implementation difference between them is when the data is available. A `URLTexture` may not download the image data from the URL it is initialized with until rendering occurs and the texture is actually needed.

QUALITY

Because most consumers of textures will render to finite resolution displays, the query methods allow sampling. To meet the needs of real-time rendering systems as well as those that produce high-end photorealistic or scientific output, three sampling quality levels are supported: fast, good, and best. Three levels are specifically supported rather than a continuum. Most graphics algorithms have a very fast approximation intended for real-time use as well as a "best" approximation using a complex mathematical model. For example, nearest-neighbor sub sampling of discrete pixmap textures versus integration over continuous representations of color data. Introducing a "good" level allows some flexibility on both the producer and consumer's part, as the "best" case may be particularly slow and the "fast" case a particularly poor approximation. Although it is subjective, allowing a "good" quality level allows the interface implementer to provide a reasonable implementation for general purpose use in addition to the extremes. For texturing, this may be bilinear or trilinear interpolation across a MIP map.

The sampling methods provide three quality levels for the caller to choose from. Most renderers will ignore these methods however and use the `to-Pixmap` method to generate texture data, then use their internal texturing mechanism at the desired quality setting (usually nearest-neighbor, bilinear, or trilinear).

The actual quality setting used when rendering is a combination of a property of the renderer and a property of the texture. If the renderer's texture quality setting is `defer` (the default), then the quality setting of the texture is used. This design allows quality levels to be attached to textures, but gives the renderer the ultimate power to override those settings. Allowing textures to set the level is desirable in many instances, for example, when foreground and background elements are to be rendered at different quality levels. It is one of the goals of the API that components will be reused in ways their designers never intended. This implies that a renderer may be attempting to render in real time objects that were not designed for high performance. By allowing the renderer to override quality settings, the application can set a low quality setting for speed and override the potentially slower quality settings individual textures request. Likewise, high quality texturing may be enforced on objects that normally are rendered at low quality.

FORMAT NEGOTIATION

To create high performance implementations of the Graphics API, it is desirable to use the native formats of underlying hardware and graphics API's for data storage. Using the native format allows texturing to occur in hardware, which frequently reduces the main processor cost of rendering from tens of cycles per pixel to effectively zero. Even if native formats are used during rendering but not for storage, a (often substantial) cost must be incurred when data is converted between the device independent and hardware-specific formats. For texture data, this cost can be on the order of hundreds of thousands of main processor cycles per texture used and it is therefore extremely undesirable to pay that cost per-frame.

Unfortunately, hardware graphics API implementations tend to store textures in immutable, highly optimized (unpublished) forms to optimize rendering for cache performance and graphics processor-level code vectorization. This leads to the drawback of using native formats for storage: they restrict the feature set of the high level API by making it impossible to move data fluidly between different devices. For example, a font object may be created for displaying text on the screen. Under most low-level graphics API's (such as the Microsoft Windows GDI or Bitstream FontFusion), font data is device-specific because it contains resolution specific hinting, is encoded for the rendering capabilities of the target, and is loaded from the target's set of installed fonts. It is not possible to take the font object created under one low-level API and hand it to another, or even a different rendering context created under the same low-level API. In the example, transferring the font to a printer target will fail.

The Curl Graphics2d API is bound by the realities of low-level APIs. However, it uses its privileged position of being more abstract, and therefore more expressive of high level concepts, to transcend the limitations of lower APIs with abstraction techniques whenever possible. By allowing the developer to work at a higher level, more information is available than in traditional languages and more relevant optimizations can be performed. Consider the case of optimizing machine code versus high level language code as an analogy. Optimizations attempted at the machine code level are inherently limited, because the information about what high-level operation is being performed is obfuscated during compilation. A bounds checker working at the machine code level must be very complicated to perform compile time determination of whether an array access is out of bounds, even for a simple loop, because the loop structure and memory accesses have been decomposed into processor instructions. At that level, an array access may appear very similar to a simple local variable reference, and a computation on an array index will look like generic arithmetic. In the high level code, however, a simple examination of all array reference operations is possible, and loop structure is clearly marked. See the `for` compiler syntax for more examples of this philosophy in action.

In the implementation of the Curl Graphics2d system on top of low-level APIs, the most common optimization in the renderer is reducing the cost of conversion between native formats, making low level incompatibilities into a high level capabilities. The general mechanism used for this is a high level abstraction that acts like a memoizer or cache. `PixmapTexture` and `UrlTexture` are two classes that implement the `Texture` interface and use this mechanism to perform format negotiation between low-level native texture formats. Note that Curl has no native texture format of its own; anything that implements the `Texture` interface can be used as a texture, so the implementer chooses the data format.

`DeviceTable` is the class that implements the device cache as a table mapping `Renderer` instances to device-specific texture handles. The interface is similar to hash table, with get and set methods. Given a `DeviceTable`, a renderer can request the native texture handle it has previously stored in that table, and may insert new values. When a renderer prepares to render a surface, it requests the texture handle previously stored. This is accomplished by extracting the device table from the texture via the `Texture.device-table` accessor, then invoking `DeviceTable.get` with the argument of the renderer itself. If the resulting value is null, the renderer generates a new device-specific native texture and stores a handle to it back into the texture so that subsequent rendering calls using this texture may proceed without the format negotiation step. A device specific texture is generated by requesting a pixmap representing the texture at the resolution required for rendering via `Texture.to-Pixmap`, then invoking the appropriate low-level API and storing the resulting handle back into the device cache. Note that the Curl data-exchange pixmap format is designed to be binary compatible with the input to most native texture creation methods (32-bit ARGB stored in row-major, top-down/left-right order).

In order to allow the device specific native formats to be deallocated when the corresponding renderers are no longer in use (and is garbage collected), the device table only maintains a *weak pointer* to the renderers used as keys. A weak pointer is a pointer that does not prevent garbage collection of an object. When an object is garbage collected, all weak references to it are set to null. A special weak hash table is used that drops pointers to corresponding values when the weakly referenced keys are garbage collected. Under this architecture, when a renderer is garbage collected, all device specific texture data it has cached in individual textures will also be garbage collected. Because most low-level APIs use explicit memory management, it is usually necessary to wrap the native data handle in a Curl class with a finalizer method so that the low-level resources can be explicitly deallocated on garbage collection of the associated Curl object.

```
{define-class public DeviceTable
  private table:{WeakKeyHashTable-of Renderer, any}

  {method public {init}
    set self.table = {new {WeakKeyHashTable-of Renderer, any}}
  }
  {method public {set renderer:Renderer, value:any}:void
    set self.table[renderer] = value
  }
  {method public {get renderer:Renderer}:any
    {if {self.table.key-exists? renderer} then
      {return self.table[renderer]}
    else
      {return null}
    }
  }
}
```

DeviceTable helper class

A texture must always maintain its own internal data so that it can implement the `to-Pixmap` and `get-` methods. Even after a renderer has cached its own proprietary texture format inside of a texture, it must be possible to generate a pixmap for that texture. The cache can be flushed to recover memory, the low-level API may drop textures during context switches, and the texture may be used with a different renderer.

BUILT-IN TEXTURES

One of the major shortcomings of the VRML web-graphics interface was the need to download huge images for use as textures. Even with relatively high bandwidth, large images can take significant time to download. The increasing volume of image, video, and audio content on the web suggests that designers and developers will continue grow their demands for rich content to saturate any amount of bandwidth available in the future. To allow lightweight applications to still contain rich content, part of the Curl Graphics2d API is a content library. This includes a large assortment of high quality textures that are guaranteed to be installed on every client machine.

One important innovation of the Curl language is the use of URLs to replace all filenames. This simple convention allows all interfaces to be internet enabled, a key factor missing from many web technologies. Curl also provides its own URL scheme, "curl://". Part of the Curl scheme name space is devoted to client side rich content. The directory (in the URL sense, it is not a file system directory) "curl://textures" contains the textures that are guaranteed to be available. Many of these are tiling 256 x 256 textures with a wide range of applicability, like images of grass, bricks, plants, and fences. Some would make poor textures by themselves, like flares, checkerboards, and white noise, but are intended to be combined to customize the other textures.

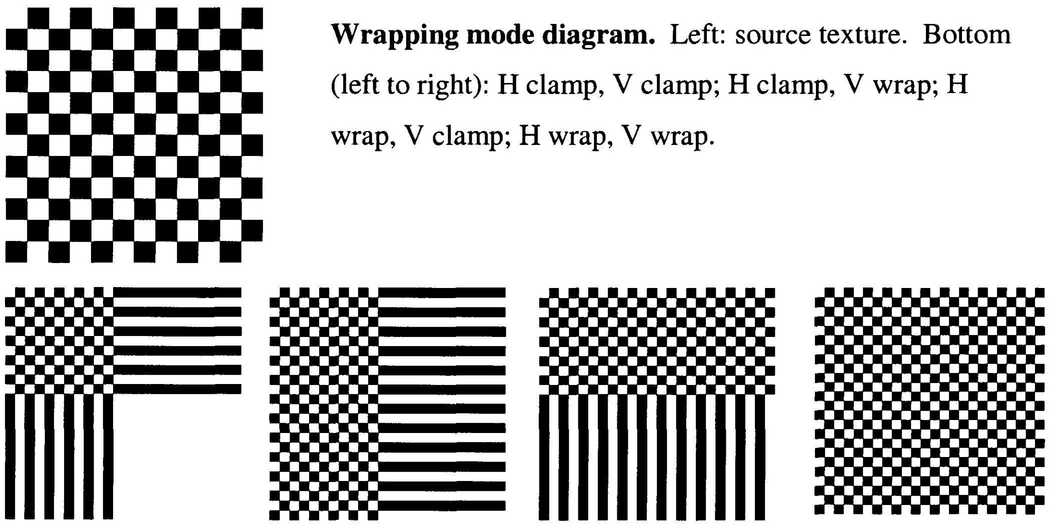
Because Curl provides an image filter library, it is possible to achieve thousands of visually unique textures from a small set by changing the properties like the hue, focus, orientation, and contrast of standard textures or by combining multiple textures using blending, modulation, accumulation, etc. The W^{3}C's HTML16 standard color set is also available by name from this directory.

With the richness of the built-in texture library and the image filter library, it is possible to create applications with richly textured surfaces (including high quality 3d applications) without requiring end-users to download large texture files.

Wrap Modes

Texture coordinates are normalized to the interval [0, 1], regardless of the resolution or aspect ratio of the texture. This is a standard practice that allows textures to be replaced with other textures or the same one at a different resolution without requiring texture coordinates to be recomputed. Coordinates outside of the [0, 1] interval are allowed. They are interpreted with respect to the texture wrapping mode. The horizontal and vertical wrapping modes are controlled independently.

When a wrapping mode is set to *clamp*, the border texels (Texture analog of pixels) are extended infinitely parallel to the axis. When a wrapping mode is set to *wrap*, the entire texture tiles along that axis with a periodicity of 1.0. The diagram below demonstrates the four possible combinations:



Wrapping mode diagram. Left: source texture. Bottom (left to right): H clamp, V clamp; H clamp, V wrap; H wrap, V clamp; H wrap, V wrap.

The checkerboard texture is chosen to specifically illustrate the wrapping modes. Ordinarily, the clamp mode is used with textures that contain a central detail surrounded by a uniform color.

When textures are used with coordinates outside the [0, 1] interval and the wrap mode, it is best to choose textures designed to tile innocuously. These textures may be inherently periodic, like the checkerboard above, or may be complicated patters than only reveal themselves to be periodic on close examination. Many of the built-in textures were carefully designed to tile well without their periodicity being obvious to the casual observer.

It is interesting to note that a small fraction of all possible periodic patterns are provably impossible to encode using rectangular tiling. These include the Penrose tiles, which are the only example known to mathematics of simple tiling patterns that cannot be represented using rectangles. This class of textures (in addition to non-periodic textures, of course) are the only ones that cannot be represented using the `Texture` interface. This is presented as an interesting curiosity, not a serious limitation. Most texture interfaces only allow pixmap based textures, so the Curl inclusion of procedural textures, gradients, lazy loading URL-textures and colors creates a substantially more flexible than the APIs graphics developers are currently using.

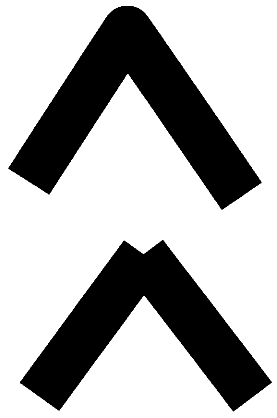
Renderer

All objects that act as rendering targets in Curl support the `Renderer` interface. This enables code that draws to be written to a single interface, whether the actual target is on screen, off screen, a printer, or an atypical output device. Adherence to this rule gives the API both flexibility and simplicity. Users need only learn a single rendering API, and may then use that API for all drawing purposes. This interface is augmented by the higher level, application oriented interfaces found in the Graphics3d API and Modeling APIs, however, users can always gain access to the underlying renderer if desired under those APIs.

FUNDAMENTAL VS. DERIVED METHODS

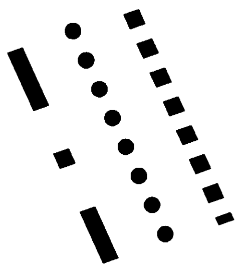
The four fundamental rendering methods, render-path, render-region, render-pixmap, and render-text, are chosen because those rendering primitives are distinct from one another both theoretically and in practice. Each is rendered with a different set of hints and algorithms and thus they are not decomposable into one another. In addition, low-level APIs and hardware tend to have different implementations for each of these primitives.

A *path* is an infinitely thin mathematical curve. Path rendering, also called *stroking*, involves rendering a contour that follows the curve of the path, given parameters like color and thickness. Important features in a path stroking algorithm include: mitering (joining), continuity, stippling, and treatment of thin lines. Mitering involves special handling of the joints where a line bends at a sharp corner. A naïve path rendering algorithm might stroke each line segment of a path separately. This can result in the broken joint shown in the bottom half of the mitering diagram on the right. A correct path stroking algorithm will fix joints by capping them with a circular, square, or triangular end. The image on the top of the mitering diagram demonstrates a round cap. The mitering method is one of the options that controls path rendering. The external (obtuse angle) and internal (acute angle) may be independently set to one of `Cap.circle`, `Cap.rectangle`, `Cap.triangle`.

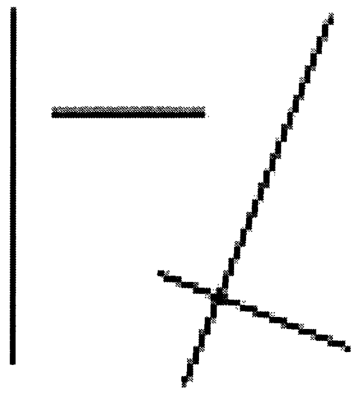

Mitering. *Top:* mitered joint. *Bottom:* non-mitered joint.

The enumeration class is named `Cap` because it is also used for the `end-cap` property. This property indicates how the ends of a thick line are rendered. I chose "rectangle," "circle," and "triangle" rather than the traditional printer's names "butt cap," "round," "sharp" because they are more expressive and easy to remember. Other path-specific options include the stroke thickness and stipple method.

Stippling is the process of applying a stencil like pattern over a line while rendering. This is the effect that generates dotted and dashed lines, as demonstrated on the right. The `stipple` renderer property allows setting the pattern to an arbitrary bit vector. The `end-cap` property is used to determine shape of the edges between transparent and solid portions of the line.


Stippled lines

In the terminology of modern raster graphics, stippling is equivalent to modulating the alpha channel of the texture with a binary mask. From a design point of view, it is appealing to eliminate the separate concept of stippling and require that users create textures that use the alpha channel for stippling. As dotted and dashed lines are commonly requested features, I feel it does not justify the amount of extra work and confusion on the user's part to create separate textures when the renderer can handle the special case in such a direct manner. Also, many devices provide direct support for stippling that would be unable to reconstruct the stipple request from a texture with alpha channel. As an end to end argument, it is better to keep the API at the high level where users can specify the effects they desire and the renderer implementations can choose to use native stippling, texture modulation, or other methods as appropriate to implement those requests.


Antialiased lines

Rendering of thin diagonal lines must also be undertaken carefully. The `Renderer` interface guarantees that any path stroked with a thickness greater than zero meters must be both continuous and visible. This allows user code to render thin lines in a device independent manner without concern for various artifacts because renderers are required to use high quality path rendering algorithms. Unfortunately, for raster devices this can cause all lines with a thickness of less than two pixels on the target device to be indistinguishably rendered as single pixel thick lines. Antialiasing techniques can improve this. Path antialiasing under the renderer can be performed during the scan conversion process by weighting the color contribution to each pixel by a function of either the distance between the path and the pixel or the area that the mathematical region formed by stroking the path would occupy in pixels[12]. In addition to smoothing the inherently jagged edges of diagonal lines by effecting low-pass filtering of the ideal image rather than subsampling, antialiasing algorithms can simulate subpixel positioning by blending.

The vertical line in the above figure has been rendered with a stroke thickness of exactly one pixel using an antialiasing algorithm. The line is centered slightly to the left of the center of the column of dark shaded pixels, so that antialiasing algorithm shades those pixels at approximately 90% intensity and the adjacent column to the left at approximately 10% intensity. This visually positions the line at a higher resolution than the display is actually capable of. An exactly centered line with a thickness equal to one half the pixel size of the display can be rendered with an intensity of 50%, presenting the appearance of a half pixel thick line and resolving the

[12] The *Sampling Theory* section (14.10.3) of the *Quest for Visual Realism* in Computer Graphics: Principles and Practice 2nd edition by Foley, van Dam, Feiner and Hughes gives many standard antialiasing algorithms as well as an excellent explanation of the sampling issues involved in antialiasing.

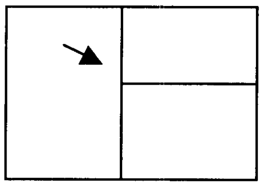
ambiguity present with thin lines in a non-antialiasing system. Also, antialiasing for subpixel position allows paths rendered with a stroke thickness equal to an even number of pixels to be properly centered.

To help users achieve a high level of control over the pixel level effects of rendering, renderers provide query methods in addition to strong pixel level guarantees. These include `Renderer.antialias-paths?`, `Renderer.antialias-text?`, `Renderer.antialias-regions?`, `Renderer.pixel-size`, `Renderer.resolution`, and `Renderer.unit-size`. The `antialias-` accessors indicate whether this renderer will attempt to use antialiasing algorithms on the various kinds of primitives. The pixel size and resolution are inverses of each other. The *unit size* of a device is the size that that device recommends using for "thin lines." Frequently, graphic designers want a "1 pixel border" or space as part of an image. Such a constraint is both difficult to accommodate and inherently a bad idea for a resolution independent graphics system (the following subsection on regions describes Curl's resolution independent graphics interfaces). The output device may have pixels that are on the order of a tenths of a centimeter (cell phone display) or are thousandths of a centimeter (printer). The pixel size is available because it is sometimes needed, although it is recommended that developers write special case code that is enabled when the resolution exits the intended range. As an alternative, developers may use multiples of unit size in their code. The unit size is approximately one one-hundredth of an inch, but it may grow or shrink slightly to be an even multiple of the actual pixel size. A one-pixel line may display well on screen but is likely to be almost invisibly thin on a printer and possibly huge on a low resolution display. In contrast, a one-unit line will appear to be approximately the same weight on all devices. The implementer of an extremely low resolution device (like the cell phone) may choose to set unit size to zero, indicating that thin lines are not a good idea on that device.

A *region* is a finite 2d area bounded by a path. Region rendering has significantly more restrictive constraints under Graphics2d than rendering of other primitives. This is because unlike pixmaps, which have few possible variations on the rendering algorithm, and fonts and paths, which are rendered in separate pieces, regions frequently abut and the edge cases must be handled correctly and consistently to avoid a large number of potential artifacts. The primary region rendering requirement is:

Mathematically adjacent, non-intersecting regions must be drawn without overdraw or underdraw (intervening space).

This means that on a raster display completely filled by non-overlapping regions, every pixel is colored by exactly one region. This requirement is essential to prevent a large number of edge artifacts that are observed in graphics systems. The Sony Playstation and Windows GDI are infamous for their overdraw and underdraw problems. When regions are rendered with blending effects, overdraw (setting the same pixel twice) blends the same pixel twice, generating visible edges in otherwise translucent surfaces. Underdraw leaves holes that can be seen through, possibly to a black background. One exception to the no underdraw/overdraw rule is allowed. This is the case of T-junctions. A T-junction, as shown in the diagram on the left, is a place where three or more regions are adjacent to each other and not all of the polygons have a vertex at that point. Because of the inherent precision limitations of digital computers for storing real number values, it is unrealistic to require that the edge without a vertex at that intersection be rendered so as to not underdraw or overdraw the edges that have a vertex at the intersection. This is a well understood problem and is simply remedied by inserting a vertex into the previously unbroken edge.


T-junction

The polygon rendering implementations in modern[13] graphics hardware all exhibit the property of no underdraw/no overdraw and it is possible to efficiently achieve in software only implementations (provided some care is taken), so these are reasonable restrictions. Many of the problems observed in current software applications are due to poorly specified high level APIs, not the underlying hardware.

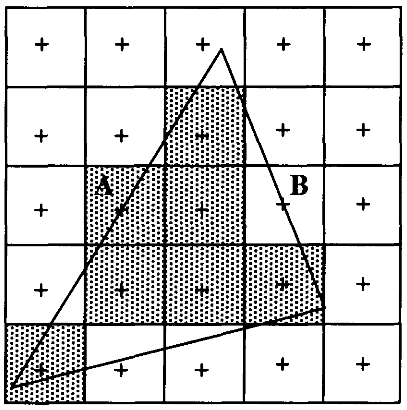
[13] circa 1996 and later

In order to ensure that the pixels along the edge of the display are rendered consistently between different raster devices, it is necessary to specify the pixel rounding rules in addition to the adjacency properties. These rounding rules are not intended to address text/font/region alignment. Most good path and font rendering algorithms perform subpixel positioning with antialiasing or round the rendering coordinates in order to center the path or glyph on a pixel center. This makes it very difficult to align regions, paths, and text to within more than a half pixel. Also, the rounding rules are not rules for rounding vertices, but for determining whether a pixel that is fractionally overlapped by a polygon is rendered or not. All vertices are (semantically) kept as real numbers in distance units throughout the entire rendering process[14].

The rounding rules are expressed through the following pixel fill rule (sometimes called the *top-left* rule because it favors the top and left edges):

1. Fill a pixel if its center is completely within the region (not on the border) *or*
2. It is on the border of the region *and*
    i. The normal to the edge of the region has direction 45 degrees $\leq \theta < 225$ degrees and no vertex is on the pixel center *or*
    ii. A vertex lies on the pixel center and a ray cast from the pixel center at 315 degrees enters the body of the region.

The diagram on the right demonstrates these rules. In the diagram, the grid marks pixel boundaries, the diagonal lines form a triangular region to be filled, the crosses mark pixel centers, and the shaded pixels are those that will be filled when the region is rendered.

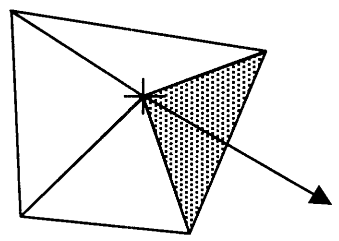


Region rounding rules. Pixel A is rounded in and pixel B is rounded out by the top-left rule.

[14] In addition to being "mathematically correct" and easier to implement, it is actually faster to rasterize in floating point than fixed point/integer on a modern processor like the Intel PIII chip. This is because the FPU is on a separate pipeline from the ALU, so integer blending operations can proceed in parallel to rasterization of the subsequent scan line.

Most of the shaded pixels are filled because of rule 1. Pixel A is filled because the edge exactly crosses the pixel center and the normal is between 45 and 225 degrees. The *normal* is the vector perpendicular to the edge, pointing out of the region. If the counter-clockwise oriented tangent to the region boundary is in the direction (*dx*, *dy*), the normal at that point is (*-dy*, *dx*). For the purpose of these rules, 45 degrees points diagonally upwards and to the right ad 225 degrees points diagonally downwards and to the left. This describes a 180 degree arc such that exactly fifty percent of the edge cases will fill the pixel. If an edge pixel is not filled, it will be filled by an adjacent region. Pixel B is such a case. The pixel center is on the boundary, however the normal is less than 45 degrees, so that pixel will be filled by a region rendered to the right of the triangle and not by the triangle itself.

Vertex cases cannot be handled using just the normal because there are two edge normals at a vertex, one for each edge. Also, at a vertex, filling that pixel 50% of the time is not sufficient; while only two regions can meet at an edge, any number may meet at a vertex. To handle this case, a ray is cast diagonally downwards from the pixel center. If that ray immediately enters the region, that region fills the pixel, otherwise that region does not fill the pixel. The diagram on the left shows four regions meeting at a pixel center. A ray is cast downwards to the left, and the region that is immediately entered fills the pixel. If the ray travels along an edge, the region owning the edge with the 45 degree normal fills the pixel.


Four regions sharing a vertex at a pixel center

Renderers that do not produce discrete output (e.g. printers) may omit the scan conversion process for combinations of Region, DrawOperation, and ControlPoints where the result will not differ. For example, solid colored regions may be rendered without explicit scan conversion.

The boundaries of the Region should be rendered with as little error as possible on high resolution devices, even when scan converting. One way of accomplishing this is applying a clipping region of the intersection of the clipping region and the region to be rendered, then performing scan conversion at lower resolution and transferring the scan converted, Pixmap output to the drawable using the new clipping region.

Fonts contain glyph descriptions. These descriptions may be device specific. Like path rendering, font rendering algorithms are designed to produce visually pleasing, high contrast results containing thin lines, rather than provide strong guarantees about the relationship of one rendered glyph to another. The Graphics2D API imposes few constraints on the font rendering of implementations. Most font renderers will provide subpixel positioning of text characters, as well as antialiasing. The provided Renderer implementation uses a commercial font rendering API to produce 256 levels of translucency for antialiasing in the output and subpixel positions to a 4x4 subpixel grid. The same draw-operation and texture that are used for region rendering are used for text characters. The font baseline, the line on which the characters sit, is transformed according to the transformation property of the renderer. Text characters may be additionally transformed by the text-transformation property (which is taken relative to the transformation). This allows rendering of text at an angle to the baseline. The diagram on the right demonstrates the use of text-transformation and transformation together.

Text-transformation vs. transformation. *Left:* 90° transformation, identity text-transformation. *Right*: 90° transformation, negative 90° text-transformation

A *pixmap* is a rectangular grid of pixel values (colors). Pixmap rendering is extremely straightforward given region rendering capabilities. Rendering of a pixmap can be implemented by creating a `PixmapTexture` then rendering a rectangle onto the target using that texture. It is treated as a primitive and given its own fundamental method because hardware often provides special support for rendering of pixmaps and because creating pixmap textures then performing rendering with them requires making at least three copies of the pixmap data, a major efficiency loss. Those three copies are the initial clone operation when the `PixmapTexture` is created, the pixmap to native texture format conversion, and the final copying of the native texture format to the destination.

Low level API's and hardware often have special support for blitting pixmaps that is not available for general texturing. By using this to avoid the copying and possibly provide improved performance (or avoid consuming the texturing module), significant performance gains can be made. These are significant enough to outweigh the possible increase in simplicity, and thereby elegance by not considering pixmaps to be a primitive.

THE WITH COMPILER EXTENSION

The `with` compiler extension allows user code to modify and restore the state of the renderer. A compiler extension is used primarily to prevent potentially untrusted user code from corrupting the state of the renderer, or reading its state (consumers of a renderer are prohibited from reading, and thereby potentially mutating, existing state). Use of an extension also allows a custom syntax. The `with` extension allows user code to effectively "push" a new value for a state variable, execute rendering calls, then "pop" that value, restoring the old state. An example of the with macro is:

```
{with texture = "red" on renderer do
   {renderer.render-line x1, y1, x2, y2}
   {renderer.render-line x3, y3, x4, y4}
}
```

With example1

This code is semantically equivalent to:

```
{renderer.render-line x1, y1, x2, y2, texture = "red"}
{renderer.render-line x3, y3, x4, y4, texture = "red"}
```

With example 2

If the same property will be in effect for a number of rendering calls (in these examples, the two line rendering methods), it is more convenient to set the property once using a `with` block, rather than repeating it for each rendering call. Because only properties that are very likely to change on each rendering call are supported as keyword arguments by the render- methods, many properties can only be set using a `with` block. The real utility of the `with` extension is not either of these features, however. Consider the alternative to using `with`, where code that uses a renderer is allowed to access renderer state through field access syntax, in the style of traditional graphics APIs like the Windows GDI or Java Graphics2D:

```
let old-texture:Texture = renderer.texture
set renderer.texture = new-texture
...  || code block A
{my-child.paint renderer}
...  || code block B
set renderer.texture = old-texture
```

With example 3 (hypothetical alternative to the `with` extension)

This alternative has three particularly bad shortcomings. First, `new-texture` may be the same as `old-texture`. When this occurs, extraneous setting and resetting of that property will occur. For the texture property, the performance impact is not significant. For a relative property like the clipping region or transformation, the overhead will be much higher as setting those properties is not a simple pointer assignment but may require more complex computation and memory allocation. Second, if an exception is thrown between assignment of the new texture and restoration of the old texture, the restoration code will be skipped and the renderer's state will not be properly reset. These first two issues can be overcome by careful discipline on the part of the programmer to provide hazard-free logic that detects assignment of a property to its current value and diligent use of exception handling code. In common practice, programmers often fail to exercise such discipline in other languages, so requiring them to use a compiler extension that will insert that code for them will lead to fewer bugs.

The third problem is one that cannot be overcome by programmer disciple when untrusted components are available or strong abstractions are required. The code makes a call to `my-child`'s paint method. If the system were designed such that renderer properties could be set without a guarantee of the initial state being restored, there is no way for the calling code to verify that `my-child` has restored the texture appropriately. For texture this is less of a concern, but often a parent object desires to set the clipping region and transformation for its child, invoke a method on the child, and be able to safely assume that the properties were restored at the end of the child's method.

The clipping region and transformation cases are necessary for providing strong graphic abstractions. Rendering calls are made by an object within a reference frame called *object space*. A parent object should be able to prevent a child object from determining its position in absolute coordinates (*world space* coordinates). This can be necessary to satisfy security concerns (e.g. not allowing an untrusted object to extract any information about the device/platform it is running on that could somehow be used to violate security). It is also desirable for providing an abstraction where an object is completely isolated from its environment and thereby prevented from violating the abstraction layer.

In With example 3, the caller knows that `my-child` can directly mutate renderer properties, and therefore has no strong guarantee that the renderer state will be correct when code block B executes. In the actual API, as shown in with example 4,

```
{with texture = new-texture on renderer do
  ...  || code block A
  {my-child.paint renderer}
  ...  || code block B
}
```

With example 4

the calling code is guaranteed that there is no way for `my-child` to corrupt the rendering state, even if an exception is thrown from the paint method.

With relative properties like the clipping region and transformation a child object needs to be able to set the property relative to the current value. The `with` compiler extension also prevents buggy code and untrusted, malicious code from enlarging the clipping region (and thereby gaining access to draw outside of its own boundaries). When `with` is used to set the clipping region, the actual operation performance is an intersection of the current clipping region with the new clipping region. At the end of the block `with` restores the old clipping region. If the clipping region property was directly accessible, as in the style of With example 3, then there would be no way to restrict code from growing the clipping region, or even worse, maintaining a pointer to the clipping region and later mutating it.

```
{with clipping-region = new-region on renderer do …}
```

Clipping region example

Absolute Properties

The following absolute properties are available to be set via the `with` extension:

- `texture`
- `font`
- `color-draw-operation`
- `alpha-draw-operation`
- `light-map`
- `stroke-thickness`
- `stipple-pattern`
- `mitering-method`
- `quality`
- `depth-buffer-mode`

The with extension maps code setting absolute properties of the form:

```
{with <property> = <value-exp> on <renderer-exp> do
       <body-exp>
}
```

to code like:

```
let <renderer>:Renderer = <renderer-exp>
let <render-data>:RenderData = <renderer>.render-data
let <old-property>:<property-type> = <render-data>.<property>
{try

    <body>

  finally do
    set <render-data>.<property> = <old-property>
}
```

where *renderer*, *render-data*, and *old-property* are generated names, and *property-type* is determined by compile time lookup. Note that the `render-data` accessor of `Renderer` is privileged access so that the extension and the class itself can read that value but user code is prohibited from seeing the accessor. This preserves the invariant that users cannot corrupt the state.

The rendering state is present on the `RenderData` class rather than the renderer itself to simplify the process of implementing the Renderer interface as well as abstract all of the state code into a separate class for readability. If the Renderer interface contained state accessors, implementers would have to trampoline each method to an underlying field rather than just the single accessor.

Relative Properties

The following relative properties are available to be set via the `with` extension:

- `transformation`
- `texture-transformation`
- `text-transformation`
- `clipping region`

These are called *relative properties* because the new value is combined with the old value (i.e. is set relative to the old value). For the three transformations, this is done by saving the old transformation matrix to the program stack and producing a new transformation that is the product of the user supplied value and the old transformation. For the clipping region, the user supplied clipping region is intersected by the old clipping region after storing the old clipping region on the program stack.

Renderer Interface

The renderer interface described in this section is:

```
{define-class public abstract Renderer
   field privileged render-data:RenderData = {RenderData}
   {method public {init}}

   ||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
   || Hardcore (fundamental) methods
   ||
   || These are the primitive operations. The arguments may be big,
   || bad, and scary. Casual users are encouraged to use the fluffy methods.
   {method public {render-path path:Path}:void}

   {method abstract public {render-region-fundamental
              region:Region,
              basis:ControlPointBasis,
              error-if-unsupported?:bool=false}:void}


   {method abstract public {render-path-fundamental
              path:Path,
              basis:ControlPointBasis,
              error-if-unsupported?:bool=false}:void}

   {method public {render-line x1:Distance, y1:Distance, x2:Distance, y2:Distance,
                          error-if-unsupported?:bool=false}:void

   {method public {render-region region:Region, error-if-unsupported?:bool=false}:void}
```

```
{method public {get-Glyph code:char,
                          font:GfxFont=self.render-data.font,
                          error-if-unsupported?:bool=false,
                          code-is-glyph-index?:bool=false}:Glyph}

{method public {render-Glyphs glyphs:{FastArray-of Glyph},
                              locations:{FastArray-of Distance2d},
                              color:Texture=self.render-data.texture,
                              error-if-unsupported?:bool=false}:void}

{method public {render-String location:Distance2d,
                              string:StringInterface,
                              font:GfxFont=self.render-data.font,
                              color:Texture=self.render-data.texture,
                              error-if-unsupported?:bool=false}:void}
{getter abstract public {unit-size}:Distance}
{getter abstract public {pixel-size}:Distance}
{getter abstract public {resolution}:Resolution}
{method public {apply-filter dest-xy:Distance2d, width:Distance,
                             height:Distance, filter:ImageFilter,
                             use-destination?:bool=false,
                             error-if-unsupported?:bool = false,
                             ...}:void}
{getter public {drawable}:Drawable}

{method public {get-clipping-region into:Region=null}:void}
{getter public {clipping-region-empty?}:bool}

{getter public {antialias-text?}:bool}
{getter public {antialias-regions?}:bool}
{getter public {antialias-paths?}:bool}

||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
||
|| All of the methods in this section can be implemented using the
|| fundamental methods. Subclass implementers only need to override these
|| to provide efficiency, not capability.


{method public {render-Pixmap dst-xy:Distance2d,
                              dst-width:Distance,
                              dst-height:Distance,
                              src:Pixmap,
                              src-x:int,
                              src-y:int,
                              src-width:int,
                              src-height:int,
                              error-if-unsupported?:bool = false}:void}

{method public {render-rectangle x1:Distance,
                                 y1:Distance,
                                 x2:Distance,
                                 y2:Distance,
                                 uv1:Fraction2d={Fraction2d 0, 0},
                                 uv2:Fraction2d={Fraction2d 1, 1},
                                 color:Texture=self.render-data.texture,
                                 error-if-unsupported?:bool=false}:void}

{method public {render-rectangular-path
                  x1:Distance,
                  y1:Distance,
                  x2:Distance,
                  y2:Distance,
                  color:Texture=self.render-data.texture,
                  stroke-thickness:Distance=self.render-data.stroke-thickness,
                  error-if-unsupported?:bool=false}:void}
```

```
    || Draws a polygon using the NZ winding rule
    {method public {render-polygon
                      color:Texture=null,
                      coords:{FastArray-of Distance}=null,
                      error-if-unsupported?:bool=false,
                      ...}:void}

    {method public {render-ellipse
                      x1:Distance,
                      y1:Distance,
                      x2:Distance,
                      y2:Distance,
                      start:Angle=0deg,
                      stop:Angle=320deg,
                      color:Texture=null,
                      error-if-unsupported?:bool=false}:void}

    {method public {render-elliptic-path
                      x1:Distance,
                      y1:Distance,
                      x2:Distance,
                      y2:Distance,
                      stroke-thickness:Distance=0m,
                      color:Texture=null,
                      error-if-unsupported?:bool=false
                  }:void}


    {method public {render-char x:Distance,
                                y:Distance,
                                c:char,
                                error-if-unsupported?:bool=false}:void}

    || If your renderer/device performs its own clipping region maintenance
    || rather than using the clipping region provided by RenderData,
    || override Renderer.push-clipping-region, Renderer.pop-clipping-region
    || and Renderer.object-to-device-space.
    {method privileged {push-clipping-region r:Region}:void}
    {method privileged {pop-clipping-region}:void}
    {method privileged {object-to-device-space r:Region}:Region}
}
```

Renderer interface

Region

The `Region` interface describes mathematical 2d areas suitable for use as clipping regions or rendering. A general purpose implementation, `GenericRegion`, describes the area enclosed by a closed `Path`. Although this class can be used to represent any region, performing intersection and union operations on `GenericRegions` may be less efficient than using one of the more limited implementations like `PolygonRegion` or `RectangleRegion`. These subclasses are designed to provide compact representations of certain kinds of regions and allow fast intersection and union operations.

All of the region methods are functional; they do not mutate the object they are applied to. This allows code to be written in a straightforward manner. In order to avoid memory allocation, the methods take an optional argument called `out`. When this argument is specified, the methods attempt to mutate `out` to hold the result of the computation. If this is possible, `out` will be mutated and returned. The behavior of mutating `out` should never be depended on, so it is essential that users always use the return value. The cases where it is not possible to use a provided out are ones where the object provided is of a subclass that cannot represent the result. For example, the union of two circular regions cannot be stored in a `RectangleRegion`. The common case of polygon-polygon clipping or polygon-rectangle clipping can be easily supported with efficiency using this method since the results are predictable, but the methods do not prohibit more general clipping. The clone method is deep so as to guarantee that the cloned region may be safely mutated without affecting the original one. All regions can be converted into `GenericRegions`.

The geometry of a `GenericRegion` is the area enclosed by a closed `Path`. A `Path` is composed of an array of zero (the empty path) or more `Distance2ds` and an array of `PathOperations`. The Distance2ds are 2d vectors with distance units (e.g. meters, inches) that form the control points of the path[15]. The operations describe how the curve is affected by each control point. The four operations available are MOVE, LINE, CURVE, and ARC. MOVE indicates that the path is disjoint and resumes at this control point. LINE connects two control points with a straight line. CURVE indicates that this segment is a cubic bezier curve (spline) which curves between the next three control points. ARC is a section of an ellipse, which is encoded with an oriented bounding box for the ellipse and start and stop points. Arc segments could be encoded using fewer points, however this would prevent a powerful feature of the path representation. The segment encoding is designed such that the control points form the convex hull of the path. Among other nice geometric properties, this means that the bounding box of the control points is always a valid bounding box for the path (or a generic region formed from it). Because all control points are absolute (i.e. not relative to other control points), it is possible to efficiently transform a path or generic region by transforming all of the points in the vertex array, and the new bounding box can be found by computing the bounding box of those points.

[15] Originally, the control points were of type ControlPoint so that arbitrary data could be attached to each vertex. To reduce the storage demands of regions, the auxiliary data (texture coordinates, colors) was removed and required to be separately supplied by the user as a parallel set of ControlPoints.

The path and generic region implementation is very similar to the Postscript path and region representation. This was done intentionally to facilitate interaction with Postscript printers as well as Display Postscript rendering engines like the Macintosh OSX rendering API.

REGION INTERFACE

The region interface is:

```
{define-class public abstract Region
    {method public abstract {clone}:Region}
    {method public abstract {to-GenericRegion}:GenericRegion}
    {getter public abstract {convex?}:bool}
    {getter public abstract {rectangular?}:bool}
    {getter public abstract {empty?}:bool}
    {method public abstract {Region.union other:Region, out:Region=null}:Region}
    {method public abstract {Region.intersect other:Region,
                                out:Region=null}:Region}
    {method public abstract {Region.transform
                              spatial-transformation:Transformation2d,
                              out:Region=null}:Region}
    {method public abstract {get-bounds
                              error-if-empty?:bool=true
                          }:{return
                                x-min:Distance,
                                y-min:Distance,
                                x-max:Distance,
                                y-max:Distance}}
     || unspecified for border cases
    {method public abstract {contains-point? point:Distance2d}:bool}

    {method public abstract {get-horizontal-spans
                                start-points:{Array-of Distance},
                                lengths:{Array-of Distance},
                                y:Distance}:void}
    {method public abstract {get-spans
                                start-points:{Array-of Distance2d},
                                end-points:{Array-of Distance2d},
                                start:Distance2d,
                                end:Distance2d}:void}
}
```

Region interface

DrawOperation

Drawing operations combine source and destination pixels to produce a new destination value. Historically, these were bitwise operations like XOR, COPY, AND, etc. In a modern graphics system they are often referred to as *alpha-blending* modes because they often use the source pixel's alpha channel as a parameter of the combination algorithm. In Graphics2d, all drawing operations implement the `DrawOperation` interface.

Two drawing operations may be specified on a renderer. The color draw operation is the algorithm used to combine the R, G, and B channels of the source and destination, possibly as a function of the source and destination A channel. The alpha draw operation describes how to combine the source and destination A channels.

The `DrawOperation` interface provides a method, `DrawOperation.combine` that implements an algorithm for combining two channels as a function of two alpha values. All of the other methods are for optimization purposes, allowing consumers of the interface to detect special cases (such as when the destination channels are not used) and optimize for them, or operate on multiple channels simultaneously (often, all channels can be operated on together packed into a single word).

The interface class also provides constants for some commonly used drawing operations. The interface is shown below, with the class hierarchy and an explanation of some of the operations following.

DrawOperation Interface

Some code and documentation is given inline to better explain the default behavior and constants.

```
{define-class abstract public DrawOperation
   || result-Channel = (src-Channel * src-A) + dst-Channel
   let public constant Accumulate:DrawOperation = {new _DrawOperationAccumulate}
   || result-Channel = src-Channel + dst-Channel
   let public constant Add:DrawOperation = {new _DrawOperationAdd}
   || result-Channel = (src-Channel * src-A) + (dst-Channel * (1 - src-A))
   let public constant Blend:DrawOperation = {new _DrawOperationBlend}
   || result-Channel = src-Channel + dst-Channel * (1 - src-A)
   let public constant BlendPremul:DrawOperation = {new _DrawOperationBlendPremul}
   || result-Channel = dst-Channel
   let public constant Destination:DrawOperation = {new _DrawOperationDestination}
   let public constant DontCare:DrawOperation = {new _DrawOperationDontCare}
   || result-Channel = (src-Channel * src-A) + (src-Channel * dst-Channel * (1 - src-A))
   let public constant Glass:DrawOperation = {new _DrawOperationGlass}
   || result-Channel = 1 - dst-Channel
   let public constant InvertDestination:DrawOperation =
     {new _DrawOperationInvertDestination}
   || result-Channel = 1 - src-Channel
   let public constant InvertSource:DrawOperation = {new _DrawOperationInvertSource}
   || result-Channel = if src.alpha > 0.5 then src-Channel else dst-Channel
   let public constant Mask:DrawOperation = {new _DrawOperationMask}
   || result-Channel = src-Channel * dst-Channel
   let public constant Modulate:DrawOperation = {new _DrawOperationModulate}
   || result-Channel = 1
   let public constant One:DrawOperation = {new _DrawOperationOne}
   || result-Channel = src-Channel
   let public constant Source:DrawOperation = {new _DrawOperationSource}
   || result-Channel = 0
   let public constant Zero:DrawOperation = {new _DrawOperationZero}
```

```
    || result-Channel = src-Channel * value + dst-Channel * (1 - value)
    {define-proc public {BlendConstantAlpha value:Fraction}:ParameterizedDrawOperation
        {return {_DrawOperationBlendConstantAlpha value}}}
    || result-Channel = (color-channel * src-A) + (dst-Channel * (1 - src-A))
    {define-proc public {BlendConstantColor color:Color}:DrawOperation {return
        {_DrawOperationBlendConstantColor color}}}
    || result-Channel = value
    {define-proc public {Constant value:Fraction}:ParameterizedDrawOperation {return
        {_DrawOperationConstant value}}}

    {method public abstract {combine dst:Fraction,
                                     dst-alpha:Fraction,
                                     src:Fraction,
                                     src-alpha:Fraction
                            }:Fraction}

    || variations on combine:
    {method public {combine-Pixels dst:Pixel,
                                   src:Pixel
                   }:Pixel
     || The default implementation of this function combines all four
     || channels using {combine} on each channel.
     {return {Pixel.create {self.combine dst.a, dst.a, src.a, src.a},
                           {self.combine dst.r, dst.a, src.r, src.a},
                           {self.combine dst.g, dst.a, src.g, src.a},
                           {self.combine dst.b, dst.a, src.b, src.a}}}}

     {method public {combine-Pixels-alpha dst:Pixel,
                                          src:Pixel
                   }:Pixel
       || The default implementation of this function just calls combine-Pixels
       {return {self.combine-Pixels dst, src}}}


     {doc-next
      {purpose
          This function combines two pixel color channels and returns the
          combined color channels in pixel form.
      }
      {notes The result alpha channels are undefined and should be ignored.}
      {notes This is an optimized version of combine-Pixels}}
     {method public {combine-Pixels-color dst:Pixel,
                                          src:Pixel
                   }:Pixel
      || The default implementation of this function just calls combine-Pixels
      {return {self.combine-Pixels dst, src}}
     }

     {getter public {uses-destination?}:bool
      || The default implementation returns true.  This is generally
      || sub-optimal (since it may not use the destination), but is
      || guaranteed to work if they forgot to implement it.
      {return true}}

     {getter public {uses-source?}:bool
      || The default implementation returns true.  This is generally
      || sub-optimal (since it may not use the source), but is
      || guaranteed to work if they forgot to implement it.
      {return true}}

     {define-DrawOperation-draw-strip-methods non-final}
}
```

The DrawOperation interface

COMMON DRAWOPERATIONS

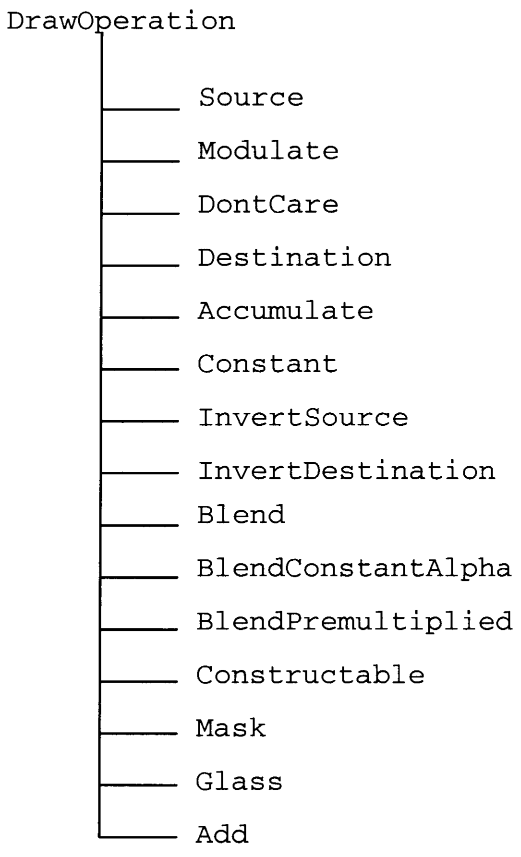
Some common drawing operations are provided as class constants. These enabled code like the following:

```
{with-render-properties color-draw-operation=DrawOperation.Source on renderer do
    ...
}
```

Example of setting a drawing operation on a renderer

without requiring allocation of a new instance of the `SourceDrawOperation` class. This practice also conveniently groups the common drawing operations into a single namespace.

Most importantly, however, it allows consumers of the interface to recognize some of the common operations and optimize for them. For example, the OpenGL based Renderer implementation provided with the API recognizes all of the provided drawing operations and uses the corresponding OpenGL drawing operation in graphics hardware instead of actually invoking the methods of the class.



DrawOperation class hierarchy

The class hierarchy is shown in the diagram on the right. Source is the operation that overwrites the destination pixels with source pixels. Destination is the NOP drawing operation; it semantically overwrites the destination pixels with themselves. InvertSource and InvertDestination write the inverted source and destination values to destination, respectively. Because the channels of pixels are semantically defined on a 0 – 1 scale, these perform 1 – *c* on each channel. Constant writes a specified constant value to each channel.

The Add operation writes the vector sum of source and destination channels to the destination.

Accumulate performs an add, but it modulates the source channels by the source alpha value. This simulates the application of paint to the destination, where the source alpha values indicate the opacity level of the paint (thickness of each coat). Many graphics programs refer to this as a *spray paint* or *airbrush effect.*

Blend is the operation that is sometimes misleadingly referred to as *alpha blending* (Graphics2d uses the term alpha blending to refer to all draw operations). The result is a linear interpolation between the source and destination channel where the interpolation parameter is the source alpha channel (0 = all destination, 1 = all source). This operation is used for fading effects and anti-aliasing. The Mask operation is an optimization of this operation for binary source alpha channels. When the source alpha channel is greater than .5, the result is the source, otherwise the result is equal to the destination. This is used for transparent blitting operations and is suitable for rendering transparent GIF images, un-antialiased sprites, and other images with transparent pixels. Note that 0.5 is not perfectly representable in the pixel representation where one byte is used per channel, so Mask happens to equally divide the alpha spectrum into transparent and non-transparent values. The operation is named Mask because the process of transparent blitting is referred to as *masking* because it can be performed using AND and OR logicial operations using a transparency bit mask.

BlendConstantAlpha is the Blend operation, but it uses a constant interpolation parameter rather than the source alpha channel. This provides a simpler interface for uniform fade effects because it is not necessary to set the source alpha values. BlendPremultipled is another optimization of the Blend operation. It is the same as that operation, however the source channels are assumed to have already been modulated by the source alpha value. For software based rendering, this can provide a performance increase if the same source values are repeatedly blended.

The Glass operation filters the destination by the source. This produces the effect of colored glass, and is useful for rendering glass surfaces, particularly in 3d. Modulate modulates the destination by the source. This produces the effect of source colored light illuminating the destination. Light maps are rendered using this operation.

ConstructableDrawOperation is a simple wrapper class that produces a draw operation given a procedure. It is not as efficient as implementing a subclass of DrawOperation but may be easier for naïve users to use.

The DontCare operation is intentionally undefined. It is provided to allow one part of the system to indicate to another that it does not care what happens to a channel. The consumer of the interface, often a renderer, may then use that channel as a scratch space, leave it alone, or write any value of its choosing into that channel. This is useful because often the code invoking a routine does not care what will happen to one or more channels as a result and would like to allow the renderer to use the fastest algorithm available for some other operation, possibly corrupting the channel in the process. The Renderer's alpha-draw-operation property is set to DontCare by default. It is very rare to read the alpha channel from the destination, so the renderer is allowed to corrupt that value unless the user specifically requests a different alpha drawing operation.

ControlPoint

A control point describes a point in a multi-valued 2d field function. Given three non-degenerate (that is, not colinear) control points, that function is described over all 2d space. A control point basis[16] describes the entire field function, and is created from three such control points.

The field function values are values that are useful to interpolate over a rendered region, such as color, texture coordinates, light map coordinates, 3d depth, and specular highlight color. Those specific values are supported by provided implementations. Users can extend the system to interpolate other values, for example 3d surface normal vectors for Phong shading.

Three or more control points are created by user code to describe the shading of a region. A control point basis is then created from those control points and supplied to a renderer render call along with the geometry. The renderer then transforms the basis and uses it to perform per-pixel shading. For hardware acceleration, a renderer may recognize certain control point bases and perform the actual interpolation and shading in hardware. The following is a tour of the methods, followed by the programmatic interface definition.

`ControlPoint.clone` returns a clone of self such that `ControlPoint.interpolate-from` on that clone will not mutate the original instance. `Set-from` clones in the other

[16] The idea of control point bases is entirely due to Aaron Orenstein, as is his clever implementation of it.

direction, copying its argument's rep over its own state. The interpolate methods linearly interpolate between control points. A number of variations are provided to meet differing constraints like simplicity or not allocating memory.

The `draw-hline` method draws a horizontal line of texture onto the destination pixmap. It is not responsible for the application of the renderer's draw operation. It is called by a renderer to draw a horizontal raster line of a `Region` (i.e. read the source texture after the renderer has transformed, clipped and rasterized the region).

The location methods return the location portion of the control point as a 2d or 3d vector. Separate x, y, and z accessors provide access to individual components. The rhw accessor is the *reciprocal to the homogeneous variable w*, and is a function of 3d depth. It is used for perspective rendering and depth buffering. The `transform` method transforms a control point by a 2d transformation.

The factory method produces a factory for control points of the same most derived type as the one it is invoked on. This may be overriden to reduce memory allocation of procedures.

The `new-ControlPointBasis` method returns a ControlPointBasis appropriate for this type of control point. The line version of this method produces a 1d basis for line rendering.

The `project-rhw` method performs a 3d -> 2d perspective projection of this control point subject to the provided rhw value. That rhw value is used to set the rhw accessor as well. Although the Graphics2d API does not explicitly perform 3d rendering or transformations, this method allows perspective projection and rendering using only 2d semantics.

The `find-plane` method finds three control points that represent the field function overspecified by an array of control points. This is used to create a basis. This process is necessary because an arbitrary subset of three points from the array, through guaranteed to be on the field function, may be degenerate and form a line or single point, or a low aspect triangle and thereby lead to a low precision `ControlPointBasis`.

The interface follows.

```
{define-class abstract public ControlPoint
  {method abstract public {set-from c:ControlPoint}:void}
  {method abstract public {interpolate-from
        a:ControlPoint, blend-factor:Fraction, b:ControlPoint}:void}
  {method final public {interpolate-self blend-factor:Fraction,
                              other:ControlPoint}:void}
  {method abstract public {draw-hline
        length:int, dst:Pixmap, x:int, y:int, dst-depth:DepthMap,
        dst-depth-offset:int, render-data:RenderData, basis:ControlPointBasis,
        pixel-size:Distance}:void}
  {getter public {location2d}:Distance2d}
  {setter public {location2d loc:Distance2d}:void}
  {getter public {location3d}:Distance3d}
  {setter public {location3d loc:Distance3d}:void}
  {getter abstract public {x}:Distance}
  {setter abstract public {x x:Distance}:void}
  {getter abstract public {y}:Distance}
  {setter abstract public {y y:Distance}:void}
  {getter public {z}:Distance}
  {setter public {z z:Distance}:void}
  {getter public {rhw}:float}
  {setter public {rhw rhw:float}:void}
  {method public abstract {transform
        spatial-transformation:Transformation2d,
        texture-transformation:TextureTransformation}:void}
  {getter abstract public {factory}:Factory}
  {method abstract public {new-ControlPointBasis
        cross-product:{Quantity-of "(distance^2)"},
        a:ControlPoint, b:ControlPoint, c:ControlPoint,
        work0:ControlPoint, work1:ControlPoint}:ControlPointBasis}
  {method abstract public {new-ControlPointBasis-line
        length:Distance, a:ControlPoint, b:ControlPoint,
        work0:ControlPoint, work1:ControlPoint}:ControlPointBasis}
  {method abstract public {project-rhw rhw:Fraction}:void}
  {method abstract public {print out:TextOutputStream}:void}
  {define-proc public {find-plane
             rep:{Array-of ControlPoint},
             start:int=0, length:int=rep.size, best-method?:bool=false}:{return
                  out0:ControlPoint, out1:ControlPoint,
                  out2:ControlPoint, non-colinear?:bool}}
}
```

ControlPoint Interface

Most user programs need control points that have arbitrary combinations of a few basic properities (texture coordinates, color, depth, light map coordinates). A helper macro is provided to create subclasses so that user code does not actually have to subclass and override methods to create new control points of these form.

```
{define-ControlPoint [{metavar Access}] {metavar Name}
   {inherits {metavar cp-property-list}}}
```

Syntax of the define-ControlPoint macro

The syntax is chosen to mimic define-class. Access specifies the protection attributes, which must be one of public, subclass, package, and private. The default is package. Name specifies the name of the new type. The cp-property-list specifies the control point properties of the newly defined `ControlPoint` subclass, separated by commas. The property list must include either xy (2d) or xyz (3d). The complete list of valid properties is:

• xy	2d location, expressed as a `Distance2d`.
• xyz	3d location, expressed as a `Distance3d`.
• color	Color information, expressed as a `Color`.
• specular	Specular highlight information, expressed as a `Color`.
• texture	Texture coordinate, expressed as a `Fraction2d`.
• lightmap	Light map coordinate, expressed as a `Fraction2d`.

To instantiate a {code ControlPoint} that was defined by calling `define-ControlPoint`, the user provides the initial values for the properties for which it is defined, separating them by commas. The following shows an example of defining a new control point type and creating an instance of that type:

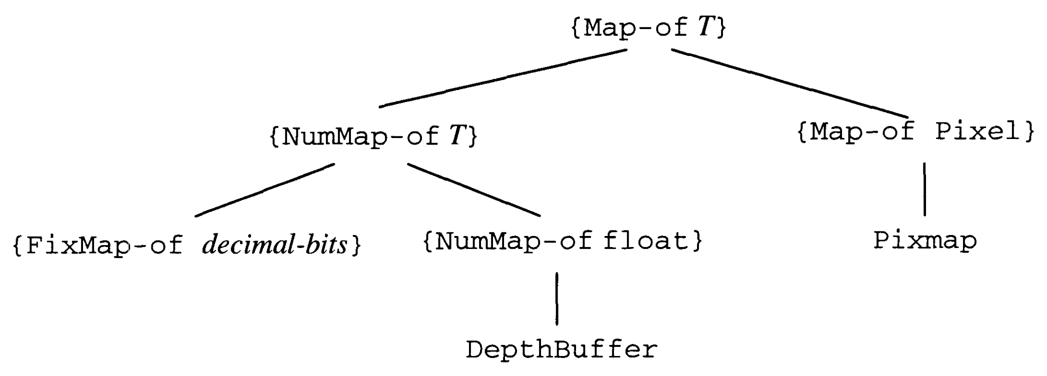
```
{define-ControlPoint public CpType {inherits xy, color}}
{CpType {Distance2d 0.25in, 0.25in}, {sRGB 1, 0, 0, 1}}
```

New ControlPoint type creation and instantiation

Pixmap

Pixmap is a two dimensional array of Pixels with semantics appropriate for graphics algorithms. A Pixel is an efficient 4 byte representation of a color in the sRGB color space. Pixmap is used for communication between user code and Graphics2d interfaces, between Graphics2d interfaces, and between Graphics2d and underlying APIs or hardware. The internal representation of Pixmap as little-endian packed ARGB bytes stored in left to right, top to bottom order is chosen because it is supported by most low level APIs in addition to being very efficient for the internal purposes of Graphics2d.

Pixmap is a specific parameterization of Map-of, the parameterized 2d array that provides `get-` and `set-` methods with semantics appropriate for graphics algorithms. `DepthBuffer`, `NumMap-of` and `FixMap-of` are other important classes in the same hierarchy. The Map-of inheritance hierarchy is shown below.



`Map-of` Inheritance Hierarchy

`Map-of` provides the basic 2d array interface suitable for graphics. The upper-left corner of the rectangular `Map-of` is its origin, from which the positive x-axis points to the right, and the positive y-axis points down. Internally the Map-of has a 1d array of elements. This array can be extracted using `Map-of.underlying-FastArray` for more efficient operation on the underlying structure. Element 0 of the underlying FastArray is the element in the upper-left corner of the `Map-of`. Element `Map-of.max-offset` is the element at the lower-right corner of the `Map-of`, and corresponds to element (`Map-of.max-x`, `Map-of.max-y`) using the 2d addressing scheme. There are five methods for indexing a Map-of, as shown below. The following code examples assume that `m` is a `{Map-of T}` and `f` is a `{FastArray-of T}` equal to `m.underlying-FastArray`. Each of the examples is given for the "set" method. A corresponding "get" method exists for each.

- `{m.set x,y}` or `m[x, y]`
  The basic set method has the same syntax and semantics of an array set. If either argument is out of bounds an exception is thrown. This method is safe, secure, and likely to trigger exceptions when there are bugs in the calling code. It is therefore the most commonly used method and it is named and given syntax to simplify and encourage its use over the other variants. This method, when inlined, costs one multiply, one addition, two comparisons, one logic operation, one branch, and two pointer dereferences per pixel, making it too inefficient

for some high performance code. The four bounds checks x >= 0, x < width, y >= 0, y <= width are reduced to two by using unsigned comparisons. Language safety can be maintained while reducing this to a single comparison of {unsafe-cast uint32, x + y * width} < {unsafe-cast uint32, width * height}, however this would violate the semantics of giving an exception in the case where x is out of bounds but x + y * width happens to still be in bounds of the underlying 1d array.

- `{m.unsafe-set x, y}`
  This set method lifts the bounds checks. It will violate safety and possibly corrupt memory if x or y is out of bounds. This method is callable only by privileged code. The cost of this method, when inlined, is one multiply, one addition, and two pointer dereferences per pixel. This is more efficient than the basic set method but may still be too slow for high performance applications.
- `{m.safe-set x,y}`
  This set method and its associated get method are appropriate for many graphics algorithms where boundary cases can be ignored and it is known that the straightforward implementation accesses out-of-bounds indices. For example, a simple blur filter may average a pixel against its eight nearest neighbors. This makes untrusted (unprivileged) code both more convenient to write as well as more efficient. Explicit code for the boundary cases of an algorithm can often occupy more space than the code for the common case. By eliminating boundary case code and allowing the safe-get/safe-set methods to handle boundaries implicitly, bugs can be avoided (you can't have bugs in code that doesn't exist) and the user code will be much simpler. With the safe methods, accesses out of bounds simply fall through to default values. The efficiency boost comes from combining the user code bounds check for correctness and the system bounds check for safety into a single operation. Of course, one must be careful when using the safe methods because exceptions will not be thrown when an algorithm goes out of bounds where it is not supposed to. It is a good practice to put assertions before loops and non-speed critical accesses to make sure that algorithms implemented using safe methods are functioning as intended. The specific behavior of the methods for the out of bounds cases is that safe-get returns the empty value (the empty value for a Map-of may be set by the user), and safe-set does nothing. These methods have the same cost as the basic set/get methods.
- `{f.set offset}` or `f[offset]`
  This is the 1d indexing method. The user may obtain a pointer to the underlying {FastArray-of T} from the Map-of and perform 1d array accesses on it. This eliminates the cost of the multiplication, addition, and extra pointer dereference required for the 2d methods at the

expense of the convenience of using 2d indexing. This method is useful when user code is accessing elements in some manner where the 1d indices are computable using some simple method like addition or a lookup table, as compared to computing x + y * width at each pixel. The cost of this method is one pointer dereference, one branch, and one comparison.

- `{f.unsafe-set offset}`
  The 1d unsafe set method strips every non-essential part of the element access. The cost is a single pointer dereference (assuming a load relative instruction on the underlying machine). This method is inconvenient, unsafe, and can violate language semantics. It is used only by privileged code in cases where absolute performance is required.

Direct pixel indexing is too inefficient as well as too complicated for many applications. In addition to direct pixel indexing, a number of operations that affect many pixels are provided. These are highly optimized parts of the system that may exceed the performance that a user implementation could provide. They also provide highly optimized, reliable implementations out of the box, allowing users to focus on higher level code.

`Map-of.render-copy` is a powerful tool for performing horizontal and vertical line drawing, rectangle filling, clearing a rectangular region, and performing blits (block image transfers). This is a traditional blitter. `NumMap-of` and `Pixmap` subclasses allow use of `DrawOperations` with render-copy.

`Map-of.render-line` is provided for debugging purposes. Unlike a `Renderer`, it does not provide any guarantees about which elements are written to. This is a very convenient method for getting a general idea of what an algorithm under development is doing. For example, when debugging a polygon rasterizer, it is convenient to use render-line to see the polygon bounds to within a few elements.

The `-clone` methods perform common whole-image transformations, such as rotation and reflection.

The `Map-of.distort` method uses a `DisplacementMesh` to implement image warps as described in the beginning of this section. Any image transform that can be described by mapping the elements of one distorted grid to another can be described and implemented efficiently and simply using the distort method, including rotations, twirl, “sphereize,” image morphing, perspective, scale, and translation. Of course, some error is introduced in this process and it is often desirable to perform simple operations like translation and scale using the render-copy method.

The most powerful multi-element operation method is the for-map-element iterator. This is described below, following the Map-of and Pixmap interface code.

```
{define-class public {Map-of T:Type} {inherits Serializable}
   public field empty-value:T
   {method public {init width:int, height:int, empty-value:T=self.default-empty-value,
                     initial-value:T=empty-value}:void}
   {method public {clone-into p:{Map-of T}}:void}
   {method public {new-like-self width:int=self.width, height:int=self.height,
                              empty-value:T=self.empty-value,
                              initial-value:T=empty-value}:{Map-of T}}
   {getter final inline public {width}:int {return self._width}}
   {getter final inline public {height}:int {return self._height}}
   {getter final inline public {underlying-FastArray}:{FastArray-of T}}
   {method final inline privileged {unsafe-set x:int, y:int, new-value:T}:void}
   {method final inline privileged {unsafe-get x:int, y:int}:T}
   {method final inline public {set x:int, y:int, new-value:T}:void}
   {method final inline public {get x:int, y:int}:T}
   {method final inline public {safe-get x:int, y:int}:T}
   {method final inline public {safe-set x:int, y:int, new-value:T}:void}
   {method public {clear new-value:T=self.empty-value}:void}
   {method public {shift dx:int, dy:int, wrap?:bool=false,
                      pad-value:T=self.empty-value}:void}
   {method public {flip-vertical}:void}
   {method public {flip-horizontal}:void}
   {method final public {clone}:{Map-of T}}
   {method public {shift-clone dx:int, dy:int, wrap?:bool=false,
                             pad-value:T=self.empty-value}:{Map-of T}}
   {method public {pad-crop-clone left-pad:int=0, bottom-pad:int=0,
                              right-pad:int=0, top-pad:int=0,
                              pad-value:T=self.empty-value}:{Map-of T}}
   {method public {embed-clone width:int, height:int,
                             halign:String="center", valign:String="center",
                             pad-value:T=self.empty-value}:{Map-of T}}
   {method public {rotate-90-clone}:{Map-of T}}
   {method public {rotate-180-clone}:{Map-of T}}
   {method public {flip-horizontal-and-vertical-clone}:{Map-of T}}
   {method public {rotate-270-clone}:{Map-of T}}
   {method public {flip-vertical-clone}:{Map-of T}}
   {method public {flip-horizontal-clone}:{Map-of T}}
   {method public {render-line x1:int, y1:int, x2:int, y2:int, value:T}:void}
   {method public {render-copy dst-x:int, dst-y:int, src:{Map-of T}, src-x:int,
                            src-y:int, width:int, height:int}:void}
   {method public {render-hline x:int, y:int, length:int, value:T}:void}
   {method public {render-vline x:int, y1:int, y2:int, value:T}:void}

   {method public {distort displacement-mesh:DisplacementMesh,
                        out:{Map-of T}={self.new-like-self},
                        smooth?:bool=false}:{Map-of T}}
   {method public inline final {in-bounds? x:int, y:int}:bool}
   {getter final inline public {max-x}:int}
   {getter final inline public {max-y}:int}
   {getter final inline public {max-offset}:int}
   {getter final inline public {size}:int}
   {method privileged {unsafe-free}:void}
}
```

Map-of interface

```
{define-class public Pixmap {inherits {Map-of Pixel}, Texture}
   field public ignore-alpha?:bool
   {method public {init width:int,
                        height:int,
                        ignore-alpha?:bool=false,
                        empty-value:Pixel={Pixel.create-from-uint8 0, 0, 0, 0},
                        initial-value:Pixel={Pixel.create-from-uint8 0, 0, 0, 0}
                  }:void}

}
```

Pixmap interface

The for-map-element Iterator

Although the normal language `for` loop iteration syntax can be used with maps, `for-map-element` (also known as "`for-pixel`") is a more powerful and flexible construct that is designed specifically for graphics developers. In order to allow compilers to better parallelize, unroll, and otherwise optimize iteration by reordering, the iteration order is unspecified. The iterator is guaranteed to execute the loop body exactly once for each element in the bounds specified, but no guarantee is made as to the order in which those elements will be processed.

The iterator syntax is:

```
{for-map-element [tag <tag>,] [<element>[:<T>] [at <x>[:int], <y>[:int]]
                              [in-rectangle <x1>, <y1>, <x2>, <y2>]
                              [in-region <region> with-resolution <resolution>]
                              in [no-read] [no-write] [<map>]]+ do
   <body>
}
|| only one of the in-region and in-rectangle clauses may be specified
```

For-map-element syntax

The *tag* is a loop name such that nested loops can be broken out of using the Curl language `break` command with *tag* as an argument. *Element* is the fresh variable name used to store the map element for which *body* is currently executing. Unlike the Curl `for` loop element, this value is assigned prior to execution of the body, then copied back into the map after the body has completed. This means that the loop can be used to safely set values via unchecked memory accesses without the body having a privilege level suitable to perform the assignments. At the beginning of each iteration the loop code generates the index of the next element. That element is fetched using the `Map-of.unsafe-get` (in practice, the offset version is used to avoid the multiply cost in computing the 1d array offset) operation and the result is stored in *element*. This is a safe operation because the loop code generates the index, and can therefore be guaranteed that that index is in bounds. After the body has executed, the current value of *element* is stored back into the map using `unsafe-set`. This is safe for the same reasons. If the user writing the body does not require the original values (i.e. *element* does not need to be read before each iteration), this can be indicated to the compiler via the hint *no-read*. Similarly, *no-write* indicates that there is no need to write the value back at the end of each iteration. If the code needs access to the two dimensional element indices, the *at* clause allows them to be specified (this is analogous to the *key* clause of `for`).

The *with-region* block allows specification of an arbitrary Region to iterate over instead of a rectangular area. Because `Map-of` coordinates are unitless and Regions specify geometry in device independent distance units like centimeters, the resolution of the `Map-of` must be specified as part of this block. This resolution is equivalent to the dot-pitch or pixel resolution graphics programmers are familiar with. Iterating over an arbitrary region is an excellent demonstration of harnessing the language's compiler extension power to abstract the operation to a level where more traditional compiler optimizations cannot compete. Many compilers can lift the bounds check operations from a simple loop. Sophisticated compilers can lift the bounds checks on a rectangular iteration such as the one specified by *in-rectangle*. We verified that both gcc, the GNU C compiler, and HotSpot, the Sun Java optimizing byte-code to native-binary JIT compiler, can lift bounds checks for simple rectangular iterations. When iterating over an arbitrary region, the Curl compiler knows that no bounds checks are necessary because it generates the code that performs the rasterization, so the indices can be trusted. By using a compiler extension, the high level concept of iterating over the elements inside of a shape is communicated directly from programmer to compiler. Without such an extension, the programmer's original intention is masked by complicated rasterization code. No traditional style compiler will be able to examine the code for a rasterizer and determine that the resulting indices are in bounds.

Because the iterator can loop over multiple Map-ofs at the same time, it provides a mechanism for allowing untrusted code to safely and efficiently perform a variety of per-pixel operations. Here is the user code for a depth-conditional bitblit. The pixels of a source pixmap are copied over the pixels of a depth pixmap if and only if an associated depth constraint is met. The iterator moves in parallel through three maps, performing a per-pixel depth comparison and copying source over destination when a depth threshold is satisfied.

```
{for-map-element source:Pixel in no-write source-pixmap,
                 dest:Pixel in no-read dest-pixmap,
                 depth:float in no-write depth-map do
   {if depth < clipping-plane then
      set dest = source
   }
}
```

Depth Conditional BitBlit

Operations like this one are very common in graphics programming. This exact process is used by many popular 3d games to render bill boarded (pixmap-based) 2d sprites into a 3d world. Allowing such an elegant coding solution as well as excellent runtime performance is a success of the API. Note that the above code could have been constrained to perform the operation in an arbitrary region, such as a circle or polygon, making it simple to write a user based shape renderer for maps.

Image Filters

Image filters describe an algorithm mapping one or more `Textures` to another. Typical image filters include GaussianBlur, Contrast, Sharpen, and Emboss. An image filter may have state that affects the result of applying it to one or more textures, for example, the blur radius of a GaussianBlur.

The interface provides two methods, one that outputs a new texture and one that is used by a renderer to apply an image filter to a destination "in place." The latter is very useful for paint programs and for rendering widgets with filtered effects.

```
{define-class abstract public ImageFilter
   || All ... args must be Textures
   {method abstract {create-texture ...}:Texture}
   || All ... args must be Textures
   {method abstract {friend-of Renderer} {apply
        renderer:Renderer, x1:Distance, y1:Distance,
        x2:Distance, y2:Distance, ...}:void}
}
```

ImageFilter interface

Both take an arbitrary number of textures as rest arguments (... in the Curl language). The create-texture method is straightforward; it chooses a resolution for the output texture from the inputs, applies the filter and returns the result. Sometimes this is accomplished by converting the input textures to Pixmaps, applying the filter algorithm and turning the resulting pixmap into a texture using the PixmapTexture class. To save memory and conserve resolution, it is possible to produce an image filter where the result of this method is a "procedural texture" that stores the image algorithm and inputs and reapplies them on the fly when the result texture's methods are invoked. For example, a filter that increases the lightness of a texture may output a result that points at the input texture. Whenever a method is called on the outputted texture, it will invoke the same method on the original input texture, modify the result by lightening it, and return that value. This is less time efficient than the pixmap method, however.

The `apply` method is used by a renderer to implement its own apply-filter method:

```
{renderer.apply-filter x1, y1, x2, y2, filter,
           use-destination?:bool, <textures>...}}
```

Applying an ImageFilter to a Renderer

This renderer method applies an image filter to the existing image on the destination if `use-destination?` is true by sampling a portion of the destination and converting it to a texture, then passing that to the image filter. The rectangular bounds supplied are passed through to the image filter. They do not restrict the area of effect of the filter (although the clipping region does). Instead, these identify the area that the identity filter would output its result to. Some filters place their output partially or entirely outside of these bounds. For example, the Rotate filter writes outside of these bounds for rotations that are not integer multiples of 90 degrees. It is possible to represent translations, scales, and other transformations using this mechanism, leading naturally to an image filter-bank mechanism.

Many image filters come with a simplified interface in the form of a procedure so that users do not have to construct an instance of the class for simple applications. For example, the blur and rotate procedures use stock instances of their associated image filter classes.

V. Conclusions

JIT Compilation

It is a serious challenge to produce a graphics API whose implementations can make full use of the processing resources available to it and give the developers using it high level flexibility and device independence. All current commercial APIs have failed to succeed at this challenge because they are too close to the hardware and do not provide sufficient room for abstraction. The presence of a true high-level source to native binary format just-in-time compiler allows such abstraction without sacrificing performance. Graphics languages are not a new concept; at the time of this writing, Pixar's RenderMan graphics language is a huge commercial success. However, the use of serious JIT compilation technology for graphics, not just interpreters or static compilers, and especially, for real time graphics, has been far underutilized. The only significant example available is the early blitters and clippers that relied on self-modifying code to produce "compiled" sprites. These blitters would iterate over the pixels in a pixmap and produce a series of machine instructions that rendered that pixmap without the usual branching logic. This is really a form of compression, and the same techniques can be applied to storing and intersecting clipping regions efficiently. Once CPUs became significantly complicated that branching logic was cheap and self-modifying code could stall the pipeline by invalidating the instruction cache, these techniques quickly waned.

Until around the year 2000, graphics performance advances depended on higher and higher level functionality on graphics cards and advances in potentially visible set determination algorithms (the algorithms that determine which surfaces need to be rendered because they are not occluded). The PVS determination work marked a real advance in computer graphics, but most of the graphics calls were misleading in terms of progress. The margins on graphics hardware grew narrow although consumer prices increased. Applications written to the graphics card APIs became homogeneous because they were all forced to use the same limited set of texturing, alpha blending and shading functionality. The cost of performing a custom alpha blend or image filter was prohibitive compared to using the standard ones, so no developer did anything new at the rendering level. The perceived performance gain was therefore misleading because it came at the expense of functionality and was part of what is now becoming an unsupportable trend; graphics hardware manufacturers can't afford to continue raising the price of graphics hardware, nor can they afford to be squeezed out by the cost of producing such dedicated mega-processors. Already architectures and APIs are moving away from this brute force, muscle-hardware approach towards using main memory and processor resources, and presenting more programmable graphics hardware. That is the sustainable trend, and that is where the JIT compiler and an API like Graphics2d are most essential. To fully make this point, it is necessary to describe the hardware architecture of the future in more detail.

I envision an architecture where a large number of processing units are available to the developer, each with highly targeted functionality. A programmer may see a handful of general purpose processors (replacing the CPU) running at different clock speeds and with different amounts of cache. These will be augmented by DSP processors suitable for generic audio, video, and encryption tasks. Matrix processors, similar to those available in high-end graphics workstations and supercomputers, will perform large matrix computations like the 4x4 multiplies that can easily dominate modern graphics processing. The fundamentals of a graphics pipeline will be available to accept high level 3d rendering calls, but points along that pipeline will be exposed and separately programmable processors for performing texturing, illumination, and shading computations. This is not to say that every piece of the rendering pipeline will be fully configurable. Some general primitives like depth-buffering, convolution-based filtering, perspective texturing of flat surfaces and matrix transformations are extremely unlikely to change significantly in the future and will make very good atomic operations. Others, like texturing and shading, offer too much richness to only allow developers to choose from a fixed set of operations. Making these customizable with general code will have a performance impact, but that is not to be feared. The RISC vs. CISC tradeoff comes from the observation that a microcoded implementation (i.e. built-in instruction) will always outperform an implementation built on top of the instruction set, however a reduced instruction set can allow more optimization potential and simplified architecture. Simplified architecture usually leads to higher clock speeds and lower latency due to short pipelines. The same is true of primitives in graphics. Offering a fixed set of eight or ten texturing modes is analogous to providing a special purpose polynomial evaluation routines. Those texture modes can be much faster than ones implemented using the general instruction set, however, adding more texturing modes increases the complexity of the hardware architecture so quickly that it can become unsustainable to add new modes or otherwise accelerate the hardware in the presence of such complexity. A computer architecture of many different processor elements with different instruction sets allows us to have our cake and eat it too. The texturing processor will have instructions suitable for performing texturing operations, however it will not have built in texturing modes. This incurs a CISC to RISC style performance hit for the common texturing modes but allows that to be recovered partly by the domain specific nature of the texturing instruction set. The advantage comes not just in the form of added functionality and generality but in a net performance gain, since the presence of a separate texturing processor frees other processors, and the simplified texturing processor can now be clocked at an extremely high rate. With reduced complexity also comes reduced cost. Graphics algorithms are highly parallelizable and often speed up linearly with the number of processors. Consider the case where a simple general texturing processor costs one-tenth the price of the special components that perform today's hard-coded texturing modes and runs at half the speed. For the same price point, it is possible to put ten general texturing processors into a machine instead of a single special purpose one and the resulting architecture will be five times faster.

Thrown into these ideas is the concept that the actual mixture of processors on a machine will change over time: advances in traditional reconfigurable processors as well as the further out potentials of bio-computing, quantum effect based computation and light based computation all indicate a trend towards reconfigurable processing capabilities. This mirrors the state that we observe today where end-user machines are highly heterogeneous, and therefore difficult to optimize applications for, but assumes that technological advances will reduce the cost (and waste) of discarding old hardware and thereby decrease the time constant over which some targeted user base can be assumed to be stable. With downloadable processor configurations, it may even be the case that a machine's architecture changes while it is running in order to better serve the currently executing applications.

I have described long range (perhaps science-fiction) changes to the hardware that performs computation. Some changes along these lines are very likely to occur in the near term, and some (heterogeneous configurations, multiple processors, programmable sub-processors) are already occurring in limited ways. These immediately raise the question for modern computer, API and language developers: *How will a programmer develop and optimize software for such a machine?* Our answer must be scalable to the time when the significant changes occur that reverse the current special-purpose/limited functionality trends.

This subsection was prefaced with an answer: JIT compilation and high level abstractions. The high-level abstractions encode the computational task at a level where it can be scheduled between multiple processors and the JIT enables that task to be performed efficiently on the processor it executes on.

The current trend of muscle-machine graphics acceleration technology will end. It will be replaced by architectures based on highly mobile code, both over the internet and inside of a computer[17]. In this environment the underlying hardware and software resources used may change during the execution of code. The practice of compiling relatively low level, platform specific code must give way to JIT compilation of high level, modeling code to arbitrary task specific hardware. Rather than simply serving as a hardware abstraction, a high level graphics API should focus on providing interfaces to fundamental graphics functionality. An API should be sufficiently abstract to facilitate the separation of low-level hardware implementation from high-level software expression, but be grounded in awareness of inherent constraints and the demands of real world applications.

Because of the performance demands, computer graphics practices are often extreme expressions of general computer science and software development practices. Many high performance graphics applications are written very close to the underlying hardware and completely rewritten for new generations of hardware. This is the mark of an immature engineering discipline. Mature disciplines define interfaces that will last through generations of technology and innovate behind solid abstractions. Performance is not a valid reason to deviate from this. The best optimization technique is always abstraction. The higher level a developer's communication with a compiler, the more opportunity the compiler has to take advantage of task specific resources. High level APIs and dynamic compilation techniques are two design components necessary to allow software to function in a less centralized environment where computing resources are ubiquitous but heterogeneous and escape versioning problems and technology cycles to achieve the robustness of appliances and tools, their mechanical counterparts.

Onward to 3d

2d immediate mode, which encompasses 3d immediate mode under the Graphics2d API, is relatively well understood. The API described in this document seeks to present a uniform API for 2d rendering that will scale to future architectures. The interesting problems in computer graphics for the next decade are in 3d retained mode, particularly the model creation aspect. A

[17] With distributed processing and relocatable processes, the boundary that effectively describes "a computer" is blurry.

stable immediate mode API is needed so that we can move on to focus on these new problems, not the details of displaying images on screen.

JIT compilation, graphics languages, and high level interfaces are technology that should be brought to bear on the 3d modeling and data manipulation problems. There is no indication that brute force approaches will be able handle the level of realism developers are beginning to demand from 3d applications. Realism in rendering is somewhere around 8000x8000 pixels at 100fps, which is within the grasp of modern technology. Realism in modeling demands trillions of objects (imagine modeling the actual earth or even a single city for a virtual reality application) and using contemporary storage techniques is beyond the storage capacity of modern technology, let alone the visible set determination and rendering capabilities. Modeling even a simple object to the standards of realism is extremely expensive and requires sophisticated technology. I eagerly look forward to the synthesis of the current batch of interesting 3d technologies: PVS algorithms, data exchange formats, image-based rendering, procedural/query based objects[18], data capture technologies, image recognition/scene reconstruction and wide scale networked VR environment management, as well as undiscovered approaches.

[18] These are advocated by the industry wide SEDRIS project as well as Curl Graphics3d, RenderMan, and Laser, and I have high hopes for this approach.