58

POINT SAMPLE RENDERING

by

J.P. Grossman

Hon. B.Sc., Mathematics (1996)

University of Toronto

Submitted to the

DEPARTMENT OF ELECTICAL ENGINEERING AND COMPUTER SCIENCE

In Partial Fulfillment of the Requirements for the Degree of

Master of Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

August, 1998

[Sep[illegible]

Signature of Author [illegible]
Department of Electrical Engineering and Computer Science, August, 1998

Certified by [illegible]
William J. Dally

Accepted by [illegible]
Arthur C. Smith, Chair, Department Committee on Graduate Students

POINT SAMPLE RENDERING

by

J.P. Grossman

Submitted to the Department of Electrical Engineering and Computer Science
on August 7, 1998 in Partial Fulfillment of the
Requirements for the Degree of Master of Science in
Electrical Engineering and Computer Science

ABSTRACT

We present an algorithm suitable for real-time, high quality rendering of complex objects. Objects are represented as a dense set of surface point samples which contain colour, depth and normal information. These point samples are obtained by sampling orthographic views on an equilateral triangle lattice. They are rendered directly and independently without any knowledge of surface topology. We introduce a novel solution to the problem of surface reconstruction using a hierarchy of Z-buffers to detect tears. The algorithm is fast, easily vectorizable, and requires only modest resources.

Thesis Supervisor: William J. Dally
Title: Professor of Electrical Engineering and Computer Science

Acknowledgements

Thanks to Bill Dally, Seth Teller, Pat Hanrahan, John Owens, Scott Rixner and Shana Nichols for their valuable comments on the structure and content of portions of this thesis. Thanks especially to Shana Nichols for her careful proofreading and unwavering support.

Thanks also to the following people for granting us permission to use their Inventor models: Barbara Cutler ('sailboat'), Brendon Glazer ('toybird'), Michael Golding ('gavel'), Stephen Ho ('tree' and 'pine tree'), and Tara Schenkel ('chair' and 'wheel'). These models were created as part of a computer graphics course taught by Seth Teller at M.I.T.

This research was carried out as part of a project, led by Bill Dally, to design an inexpensive media processor capable of real time 3D graphics. A big thanks to Bill for his vision, his drive, and his encouragement; we miss you here at MIT.

This work was sponsored by the Defense Advanced Research Projects Agency (DARPA) under DARPA Order E254 and monitored by the Army under Contract DABT63-96-C-0037.

Contents

1 Introduction

The research presented in this thesis was motivated by a desire to render complex objects in real time without the use of expensive hardware. This ubiquitous goal has recently led to the use of images, rather than polygons, as primitives. Unlike traditional polygon systems in which rendering time is proportional to scene complexity, in image based systems rendering time is proportional to the number of pixels in the source images or the output image. The techniques of view interpolation [Chen93, Chen95], epipolar geometry [Laveau94], plenoptic modeling [McMillan95], light fields [Levoy96], lumigraphs [Gortler96], view based rendering [Pulli97], nailboards [Schaufler97] and post rendering 3D warping [Mark97] have all demonstrated the speed and viability of image based graphics.

Despite the success of these approaches, they suffer in varying degrees from a large appetite for memory, noticeable artifacts from many viewing directions, and an inability to handle dynamic lighting. It is therefore desirable to develop an algorithm which features the speed of image based graphics without these associated problems. In this thesis we will show that **Point Sample Rendering**, which uses points rather than images as primitives, is able to achieve this ideal.

In Point Sample Rendering, objects are modeled as a dense set of surface point samples. These samples are obtained from orthographic views and are stored with colour, depth and normal information, enabling Z-buffer composition, Phong shading, and other effects such as shadows. Motivated by image based rendering, point sample rendering is similar in that it also takes as input existing views of an object and then uses these to synthesize novel views. It is different in that (1) the point samples contain more geometric information than image pixels, and (2) these samples are *view independent* - that is, the colour of a sample does not depend on the direction from which it was obtained.

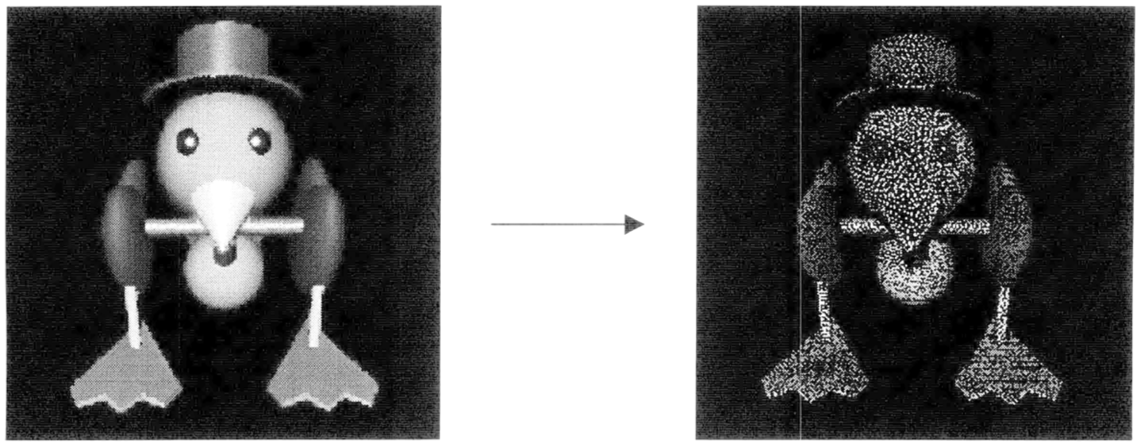


Figure 1.1: Point sample representation of an object.

The use of points as a rendering primitive is appealing due to the speed and simplicity of such an approach. Points can be rendered extremely quickly; there is no need for polygon clipping, scan conversion, texture mapping or bump mapping. Like image based rendering, point sample rendering makes use of today's large memories to represent complex objects in a manner that avoids the large render-time computational costs of polygons. Unlike image based representations, which are view dependent and will therefore sample the same surface element multiple times, point sample representations contain very little redundancy, allowing memory to be used efficiently.

The primary goal of this thesis is to establish Point Sample Rendering as a viable algorithm for rendering complex objects in real time. We will outline the challenges of Point Sample Rendering and discuss solutions to the problems that arise. Chapter 2 reviews a number of different image-based rendering systems in order to better understand their advantages and shortcomings. Chapter 3 introduces point sample rendering in greater depth, and chapter 4 discusses our solution to the fundamental challenge of point sample rendering – the reconstruction of continuous surfaces. Chapters 5-7 give details for a complete point sample rendering system in the context of our solution to the surface reconstruction problem. We present some results in chapter 8; finally in chapters 9 and 10 we discuss the strengths and weaknesses of point sample rendering, suggest some areas for future research, and conclude.

2 Image Based Rendering

The ideas of using points and images as rendering primitives have evolved separately over time. However, the particular Point Sample Rendering algorithm which will be discussed in this thesis was primarily motivated by, and is similar to, image based rendering. It is therefore instructive to review a variety of image based rendering techniques in order to understand our decision to use points for a real time rendering system despite the recent success of images in this domain.

2.1 A Brief History of Image Based Rendering

The first paper on Image Based Rendering was arguably a 1976 paper entitled "Texture and Reflection in computer Generated Images" [Blinn76]. In this paper Blinn and Newell formalized the idea, which had been previously suggested in [Catmull75], of applying a texture to the surface of an object. They went on to introduce the concept of environment maps which can be used to simulate reflection on curved, polished surfaces. Both of these ideas make use of pre-computed images to enhance the complexity of a rendered scene. Textures are typically small repeating patterns, whereas environment maps attempt to capture the reflection of the entire surroundings using images.

A natural extension of environment maps is to represent entire scenes as collections of images. In [Lipman80] part of a small town was modeled by recording panoramic images at 10 foot intervals along each of several streets and recording these images to videodisc. A user could then virtually drive through these streets and look in any direction while doing so. Camera rotation was continuous, but camera translation was discrete – there was no interpolation between neighbouring viewpoints. In [Chen93] this idea was extended to provide continuous motion between viewpoints by morphing from one to the next. However, the virtual camera was constrained to lie on the lines joining adjacent viewpoints, and the linear interpolation of the optical flow vectors which was used for morphing provided approximate view reconstruction only. In [Laveau94] and [McMillan95] the optical flow information was combined with knowledge of the relative positions of the cameras used to acquire the images. This allowed the researchers to provide *exact* view reconstruction from arbitrary viewpoints.

The idea of using images to represent single objects (as opposed to entire scenes) was mentioned in [Chen93] and realized by the 'object movies' described in [Chen95]. This system again used morphing to approximately reconstruct novel views of an object. Another approach to generating approximate novel views is to simulate 3D movement by applying affine transformations to existing views. In effect, this treats the object as a flat rectangle textured with its image. To construct a nearby view, the textured rectangle is simply rendered from the new viewpoint. This method has the advantage of being able to make use of standard texture mapping hardware. In [Maciel95] these 'imposters' were precomputed and selected for use at render time. In [Schaufler95] they were dynamically generated using polygons and then re-used for as many frames as possible. This latter approach was also used in the Talisman rendering system [Toborg96] to provide higher refresh rates than would be possible using polygons alone.

One of the problems encountered with imposters is that because they are flat, object intersections are not handled correctly. To address this problem, Schaufler added depth information to the imposters [Schaufler97]. The resulting 'nailboards' (imposters with depth) are still rendered using the same 2D affine transformation as before. However, the depth information is used to compute approximate depths for pixels in the destination image; object intersections can then be dealt with using a Z-buffer. Alternately, the depth information could be used to compute an exact 3D warp as in [Mark97].

It was argued in [McMillan95] that all image based graphics algorithms can be cast as attempts to reconstruct the plenoptic function (light flow in all directions). This approach is used explicitly by [Levoy96] (Light Field Rendering) and [Gortler96] (Lumigraphs). Rather than forward mapping images to construct views, they attempt to directly represent the 4D function which gives the colour and intensity of the light travelling along any ray in free space. Images can then be constructed by ray tracing this data structure directly. In [Levoy96] a light field rendering system is presented which is capable of computing images in real time. Since then a number of researchers have investigated extensions and applications of the light field concept [Wong97, Camahort98, Dischler98, Miller98].

The past two years have seen an abundance of research in image based graphics. If the proceedings of the annual SIGGRAPH conference provide any sort of measure of a topic's importance, then the SIGGRAPH '98 proceedings, which contain two full paper sessions devoted to image based rendering, indicate the prominence of this area of research in the computer graphics community. A complete summary of all ongoing work in this field is beyond the scope of this thesis; in this section we have simply provided an overview of the foundations upon which current research is based.

2.2 Problems with Image Based Graphics

Several of the techniques mentioned in section 2.1 are capable of rendering complex objects in real time. However, there are three basic problems that one encounters using an image based rendering paradigm: geometric artifacts, static lighting, and large memory requirements. The following sections briefly outline these problems; in Chapter 3 we will see how they can be avoided by using Point Sample Rendering.

2.2.1 Geometric Artifacts

It is impossible to accurately reconstruct a novel view of an object from existing views without knowing anything about the object's geometry. For example, consider the two views of a teapot shown in Figure 2.1 (a) and (b). Suppose we wish to construct an in-between view by applying a transformation to the two existing views and then combining the resulting images. Without any geometric information, there is no way to ensure that the spouts from the two images will be correctly aligned, and we will invariably end up with the geometric artifact shown in Figure 2.1c. In [Levoy96] this problem is addressed by artificially blurring the image as in Figure 2.1d.

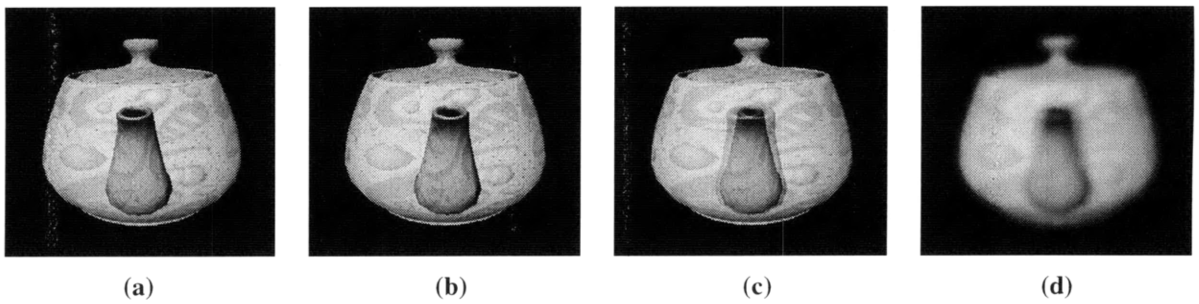


Figure 2.1: **(a, b)** Two views of a teapot. **(c)** Without any geometric information, any attempt to combine the two views to construct an in-between view will invariably result in a double image of the spout. **(d)** Artificially blurring the image to hide geometric artifacts.

In practice, with the notable exception of imposters which treat objects as textured rectangles, nearly all image based rendering algorithms make use of some sort of geometric information. This information is either in the form of correspondences between images [Chen93, Chen95, Laveau94, McMillan95], explicit per-pixel depth information with known transformations between views [Dally96, Pulli97], or an approximation of the object's shape which is used to alleviate, but not eliminate, the geometric artifacts [Gortler96].

The extent to which an image based rendering algorithm suffers from geometric artifacts depends on the manner in which it makes use of geometry. This is not to say that one should always maintain accurate depth information and precisely calibrated transformations between images in order to perform exact 3D warps; there are two reasons to seek algorithms with less of a reliance on geometry. First, this information may simply not be available when real objects are being modeled. The modeler may not have access to an accurate range scanner, or may be generating the model from pre-existing photographs. Second, it may not be possible to represent the object using images with depth. Two objects which defy such a representation are presented in [Gortler96]. One is a hairy stuffed lion, the other is a glass fruit bowl containing fruit. The former contains detail which is too fine to represent using one sample per pixel, the later contains pixels with visible depth complexity greater than 1 (fruit seen through glass).

2.2.2 Static Lighting

Another side effect of rendering without the complete geometric information available in polygon systems is the inability to use dynamic lighting. This is usually seen as an advantage rather than a drawback of image based rendering: the lighting, although static, is free. It requires no computation at render time since it is captured in the images of the object. Furthermore, it will contain all of the complex and usually computationally intensive lighting effects that are present in the original images such as shadows, specular highlights and anisotropic effects.

Notwithstanding these advantages of static lighting, there are certain applications which demand dynamic lighting. In [Wong97] the light field system was extended to support dynamic lighting by storing a Bidirectional Reflectance Distribution Function at each pixel, essentially turning a 4 dimensional data structure into a 6 dimensional data structure to take into account the lighting directions. However, this change increases model size significantly and slows down rendering; an SGI Indigo 2 was required to achieve interactive frame rates.

2.2.3 Large Memory Requirements

Since a large number of images are required to properly sample an object, image based techniques are fairly memory intensive. This makes it difficult to render scenes containing multiple objects in an image based fashion. In most cases it also necessitates the use of compression which slows down the rendering process. Table 2.1 gives typical uncompressed model sizes for various rendering techniques which appear in the literature; clearly the models are not getting any smaller over time.

Rendering Technique	Typical Uncompressed Model Size
QuickTime VR [Chen95]	1MB*
Light Fields [Levoy96]	400MB
Lumigraph [Gortler96]	1.125GB
Light Field + BRDF [Wong97]	4GB*
Uniformly Sampled Light Field [Camahort98]	11.3GB

Table 2.1: Typical Image Based Graphics model sizes. * estimate based on content of reference

3 Point Sample Rendering

In image based graphics, views of an object are sampled as a collection of view dependent image pixels. In point sample rendering, the views are sampled as a collection of infinitesimal view-independent surface points. This subtle shift in paradigm allows us to address the problems presented in section 2.2 without sacrificing rendering speed. By storing an exact depth for each point we are able to avoid geometric artifacts. By storing the normal, shininess and specular colour we are able to implement dynamic lighting. Finally, the use of a view independent primitive allows us to dramatically reduce memory requirements. Rather than sampling a given surface element from all directions as in image based graphics, we sample it once and then *compute* its appearance from an arbitrary direction using the current lighting model (Figure 3.1).

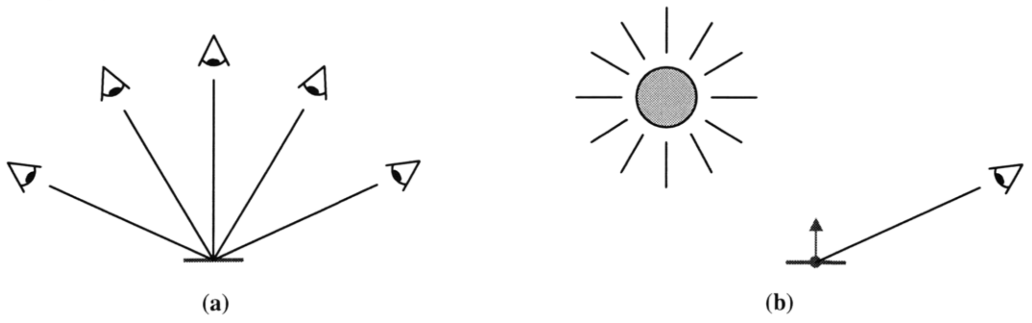


Figure 3.1: **(a)** Image Based Rendering stores views of a surface element from every direction. **(b)** Point Sample Rendering stores a single point with colour and normal and computes views from arbitrary directions using the current lighting model

Points have often been used to model 'soft' objects such as smoke, clouds, dust, fire, water and trees [Reeves83, Smith84, Reeves85]. The idea of using points to model solid objects was mentioned nearly two decades ago by Csuri et. al. [Csuri79] and was briefly investigated by Levoy and Whitted some years later for the special case of continuous, differentiable surfaces [Levoy85]. In 1988 Cline et. al. generated surface points from computed tomography data in order to quickly render parts of the human anatomy [Cline88]. More recently Max and Ohsaki used point samples, obtained from orthographic views and stored with colour, depth and normal information, to model and render trees [Max95]. However, almost all work to date dealing with a point sample representation has focused on using it to generate a more conventional representation such as a polygon mesh or a volumetric model [Hoppe92, Turk94, Curless96].

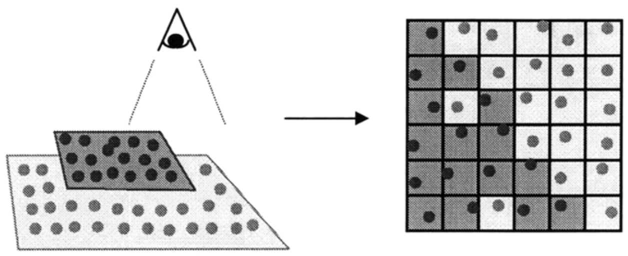


Figure 3.2: A dark surface is closer to the viewer than a light surface. When the surfaces are rendered using point samples, the points on the dark surface are more spread out. Some screen pixels are 'missed', and the light surface peeps through.

The fundamental challenge of point sample rendering is the reconstruction of continuous surfaces. In polygon rendering, continuous surfaces are represented exactly with polygons and displayed accurately using scan conversion. However, point sample rendering represents surfaces only as a collection of points which are forward-mapped into the destination image. It is entirely possible for some pixels to be 'missed', causing tears to appear in surfaces (Figure 3.2). It is therefore necessary to somehow fix these surface holes. Furthermore, this issue should be addressed in a manner that does not seriously impact rendering speed. The algorithm presented in [Levoy85] makes use of surface derivatives and is therefore applicable only to differentiable surfaces. In [Max95] no attempt is made to address this problem at render time, but it is alleviated by sampling the trees at a higher resolution than that at which they are displayed. Although this does not completely eliminate gaps in the output image, such gaps are not problematic as they appear to the viewer as spaces between leaves rather than errors in the final image. In chapter 3 we will examine this problem more closely and present a solution which produces good results with little overhead.

A secondary challenge of point sample rendering is the automatic generation of efficient point sample representations for objects. In this thesis we restrict our attention to synthetic objects, which allows us to sample the objects from arbitrary directions without worrying about the inaccuracies associated with real cameras and range scanners. As in image based rendering, we will take the approach of viewing an object from a number of different directions and storing samples from each of these views. The problem is to (i) select an appropriate set of directions from which to sample the object, and (ii) select a suitable set of point samples from each direction such that the entire set of samples forms a complete representation of the object with minimal redundancy.

The remainder of this thesis details our approach to object modeling and rendering using point samples. In order to test and evaluate our ideas, we have implemented a software system which takes as input object descriptions in Open Inventor format and produces point sample models as output. These models are then displayed in a window where they may be manipulated (rotated, translated and scaled) in real time by the user. The software system was used to generate all of the statistics and renderings which appear in this thesis.

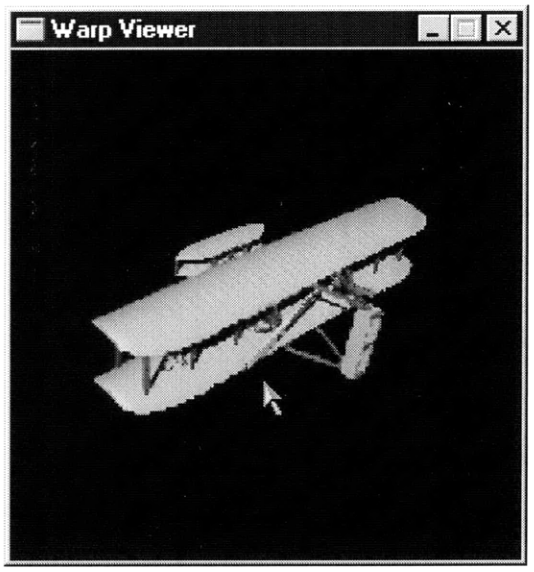


Figure 3.3: Interactive viewer

4 Surface Reconstruction

Given a set of surface point samples, it is easy enough to transform them to screen space and map them to pixels. When a point sample is mapped to a pixel, we say that the pixel is **hit** by the point sample. When a pixel is hit by points from multiple surfaces a Z-buffer can be used to resolve visibility. The difficulty is that, in general, not all pixels belonging to a surface will be hit by some point from that surface. For example, Figure 3.2 shows a dark surface that has holes in it because some of the pixels were 'missed'. A light surface which is further away is visible through these holes.

It is necessary to somehow reconstruct these surfaces so that they appear continuous when rendered. Note that it is *not* necessary to compute an internal representation for the surfaces as in [Hoppe92]; we simply need to ensure that they are properly reconstructed in the output image.

4.1 Previous Approaches

One approach to the problem of surface reconstruction is to treat the point samples as vertices of a triangular mesh which can be scan-converted as in [Mark97]. We have rejected this approach for a number of reasons. First and foremost, it is slow, requiring a large number of operations per point sample. Second, it is difficult to correctly generate the mesh without some a priori knowledge of the object's surface topology; there is no exact way to determine which points should be connected to form triangles and which points lie on different surfaces and should remain unconnected. Invariably one must rely on some heuristic which compares depths and normals of adjacent points. Third, it is extremely difficult to ascertain whether or not the entire surface of the object is covered by the union of all triangles from all views, especially if we are retaining only a subset of the point samples from each view. Fourth, when a concave surface is sampled by multiple views, the triangles formed by points in one view can obscure point samples from other views (Figure 4.1a). This degrades the quality of rendered images by causing pixels to be filled using the less accurate colour obtained from triangle interpolation rather than the more accurate point sample which lies behind the triangle. Fifth, noticeable artifacts result when we attempt to combine triangles from multiple views in a single image. Far from merging seamlessly to form a single continuous surface, triangles from different views appear incoherent, as shown in Figure 4.1b.

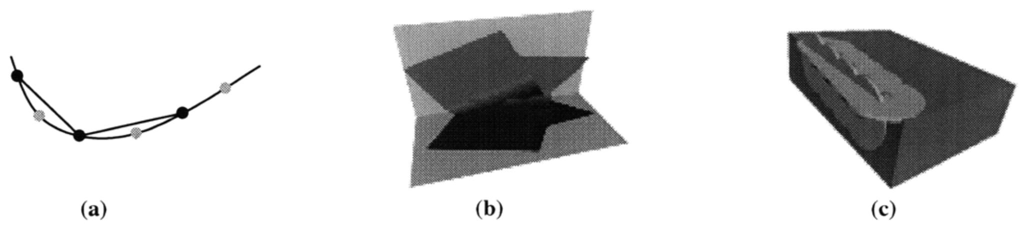


Figure 4.1: **(a)** A curved surface is sampled by two views (shown in one dimension). When the views are combined, triangles from one view (connected dark points) obscure point samples from the other (light points). **(b)** Part of an object consists of a light surface intersecting a dark surface at 90° (shown in lighter shades of gray). This corner is sampled by two different views. When triangles from these two views are combined by scan-converting them into a single Z-buffered image (shown in darker shades of gray), the triangles appear incoherent due to the different rates at which they interpolate from light to dark. **(c)** 'Overshooting' in splatting.

Another approach is the use of 'splatting', whereby a single point is mapped to multiple pixels on the screen, and the colour of a pixel is the weighted average of the colours of the contributing points. This method was used by Levoy and Whitted [Levoy85]; it has also been used in the context of volume rendering [Westover90] and image based graphics [Mark97]. However, this is again slow, requiring a large number of operations per point. Furthermore, choosing the size and shape of the splat to ensure that there are no tears in any surface is an extremely difficult problem. One interesting idea is to use point sample normals to draw small oriented circles or quadrilaterals, but this results in 'overshoot' near corners as shown in Figure 4.1c.

4.2 Adequate Sampling

In order to maximize speed, our solution essentially ignores the problem altogether at render time, as in [Max95]. To see how this can be achieved without introducing holes into the image, we start with the simplifying assumptions that the object will be viewed orthographically, and that the target resolution and magnification at which it will be viewed are known in advance. We can then, in principle, choose a set of surface point samples which are dense enough so that when the object is viewed there will be no holes, independent of the viewing angle. We say that an object or surface is **adequately sampled** (at the given resolution and magnification) if this condition is met. To fashion this idea into a working point sample rendering algorithm, we need to answer the following three questions:

- How can we generate an adequate sampling using as few point samples as possible?
- How do we display the object when the magnification or resolution is increased?
- What modifications must be made to the algorithm in order to display perspective views of the object?

These questions will be addressed in the following sections.

4.3 Geometric Preliminaries

We begin by establishing a geometric condition which guarantees that a set of point samples forms an adequate sampling of a surface. Suppose we overlay a finite triangular mesh on top of a regular array of square pixels. We say that a pixel is **contained** in the triangular mesh if its center lies within one of the triangles (Figure 4.2a). As before, we say that a pixel is **hit** by a mesh point if the mesh point lies inside the pixel.

Theorem If the side length of each triangle in the mesh is less than the pixel side length, then every pixel which is contained in the mesh is hit by some point in the mesh.

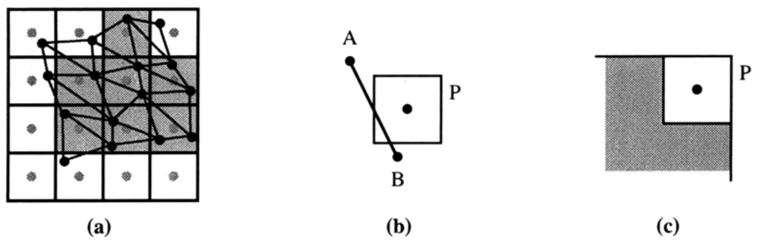


Figure 4.2: **(a)** Pixels contained in a mesh. **(b)** If one of A, B, C lies above P and another lies below, then the distance between them is greater than the side length of P. **(c)** A, B, C lie in the shaded region.

Proof Our proof is by contradiction - suppose instead that some pixel P is contained in the mesh but is not hit by any mesh point. Then there is some mesh triangle ABC such that ABC contains the center of P, but A, B, C lie outside of P. Now we cannot have one vertex of ABC above (and possibly to the left/right of) P and another below P, as then the distance between these vertices would be greater than the side length of P (Figure 4.2). Similarly, we cannot have one vertex to the left of P and another vertex to the right of P. Without loss of generality, then, assume that A, B, C all lie to the left of and/or below P (Figure 4.2c). The only way for a triangle with vertices in this region to contain the center of P is if one vertex is at the upper left corner of P and the other is at the lower right, but then the distance between these vertices is greater than the side length of P, contradiction.

Corollary Suppose a surface is sampled at points which form a continuous triangular mesh on the surface. If the side length of every triangle in the mesh is less than the side length of a pixel at the target resolution (assuming unit magnification), then the surface is adequately sampled.

Proof This follows directly from the theorem and the fact that when the mesh is projected orthographically onto the screen, the projected side lengths of the triangles are less than or equal to their unprojected lengths.

4.4 Sampling

The previous section provides some direction as to how we can adequately sample an object. In particular, it suggests that to minimize the number of samples, the distance between adjacent samples on the surface of the object should be as large as possible but less than *ds*, the pixel side length at the target resolution (again assuming unit magnification). In order to obtain such a uniform sampling, we sample orthographic views of the object on an equilateral triangle lattice. Now it is possible for the distance between adjacent samples on the object to be arbitrarily large on surfaces which are nearly parallel to the viewing direction (Figure 4.3a). However, suppose we restrict our attention to a surface (or partial surface) S whose normal differs by no more than θ from the projection direction at any point, where θ is a 'tolerance angle'. If we sample the orthographic view on an equilateral triangle lattice with side length *ds*·cosθ, we obtain a continuous triangular mesh of samples on S in which each side length is at most (*ds*·cosθ) / cosθ = *ds* (Figure 4.3b). Hence, S is adequately sampled.

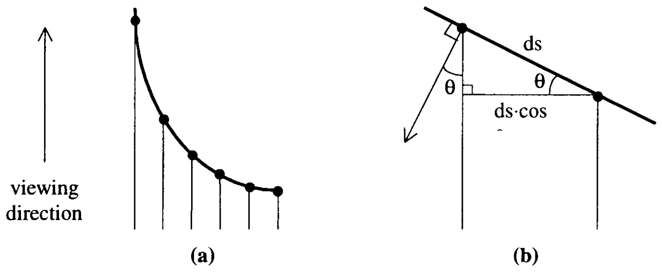


Figure 4.3: **(a)** When an orthographic view is sampled on a regular lattice, the samples on the surface of the object can be arbitrarily far apart. **(b)** If the normal of a surface is everywhere within θ of the viewing direction and the spacing between lattice points is *ds*·cosθ, then the distance between adjacent samples on the surface of the object will be less than or equal to *ds*.

Suppose the object being modeled is smooth and convex, where by "smooth" we mean that the surface normal exists and doesn't change too quickly (we will neglect to derive a more precise definition for smooth since this would not add anything to the discussion). Given θ, we can choose a set of projections such that any direction in space will be within θ of some projection direction. In particular, the normal at each point on the surface of the object will differ by no more than θ from some projection direction. It follows that if we use this set of projections to sample the object then the entire object will be adequately sampled.

Unfortunately, most interesting objects exhibit a tendency to be neither smooth nor convex. We will not, therefore, use the corollary of section 4.3 to 'prove' that an object is being adequately sampled. We simply use it to provide theoretical motivation for an algorithm which, ultimately, must be verified experimentally.

We conclude with three comments on sampling. First, although we have referred multiple times to 'surface meshes' we do *not* store any surface mesh information with the point samples, nor is it necessary to do so. It suffices for there to exist *some* continuous triangulation of the points on a surface which satisfies the conditions given in section 4.2. Second, as we collect point samples from multiple orthographic projections of an object, a given surface patch will in general appear in, and be sampled by, several projections. Hence, this patch need not be adequately sampled by some single projection so long as it is adequately sampled by the union of samples from all projections. Finally, the preceding discussion brings to light the advantage of using an equilateral triangle mesh. If we were to use a square lattice instead, we would need to decrease the spacing between lattice points to *ds*·cosθ/√2 in order to compensate for the longer diagonals within the lattice squares. This would increase the total number of samples in a projection by 73%.

4.5 Magnification and Perspective

The concept of an 'adequately sampled' surface does not solve the surface reconstruction problem; if we wish to magnify the object we are once again confronted by the original dilemma. However, suppose that the object is adequately sampled at the original resolution/magnification, and suppose we decrease the image resolution by the same amount that the magnification is increased. The object will again be adequately sampled at this new resolution/magnification. Thus, we can render the object at the lower resolution and resample the resulting image to the desired resolution.

This works well for reconstructing orthographic views of the object, but when we try to adapt the solution to perspective views we encounter a problem: because of the perspective transform, different parts of the object will be magnified by different amounts. A straightforward solution would be to choose a single lower resolution taking into account the *worst case* magnification of the object, i.e. the magnification of the part closest to the viewer. However, this unfairly penalizes those parts of the object which would be adequately sampled at a higher resolution, and it unnecessarily degrades the quality of the final image.

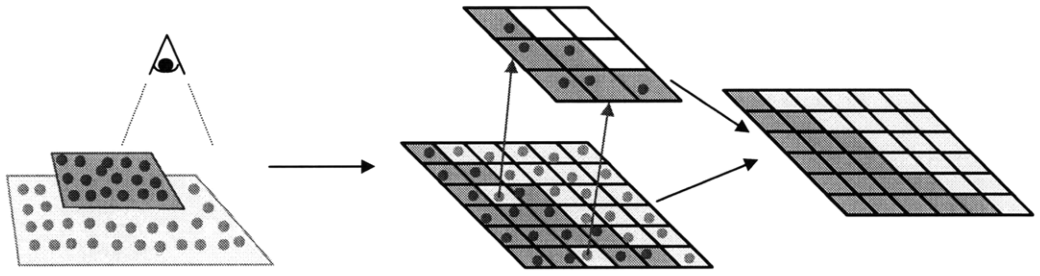


Figure 4.4: The dark surface is rendered using a higher level depth buffer in which there are no gaps. This depth buffer is used to detect and eliminate holes in the image.

To address this problem, we introduce a *hierarchy* of lower resolution depth buffers as in [Green93]. The lowest buffer in the hierarchy has the same resolution as the target image, and each buffer in the hierarchy has one half the resolution of the buffer below. Then, in addition to mapping point samples into the destination image as usual, each part of the object is rendered in a depth buffer at a low enough resolution such that the points cannot spread out and leave holes. When all parts of the object have been rendered, we will have an image which, in general, will contain many gaps. However, it is now possible to detect and eliminate these holes by comparing the depths in the image to those in the hierarchy of depth buffers (Figure 4.4). The next two sections provide details of this process.

4.6 Finding gaps

The easiest way to use the hierarchical depth buffer to detect holes is to treat a pixel in the k^{th} depth buffer as an opaque square which covers exactly 4^k image pixels. We can then make a binary decision for each pixel by determining whether or not it lies behind some depth buffer pixel. If a pixel is covered by a depth buffer pixel, then we assume that it does not lie on the foreground surface, i.e. it is a hole. In the final image this pixel must be recoloured by interpolating its surroundings. If a pixel is *not* covered by any depth buffer pixels, then we assume that it *does* lie on the foreground surface, and its colour is unchanged in the final image. This method is fast and easy to implement. However, it produces pronounced blocky artifacts, particularly at edges, as can be seen in Figure 4.5.

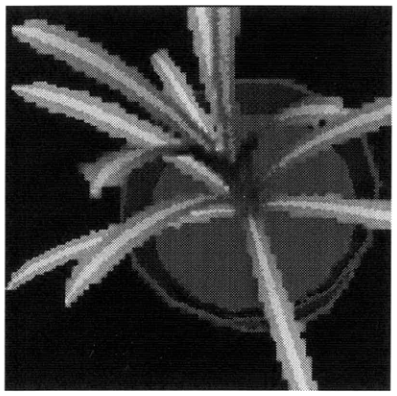


Figure 4.5: Treating depth buffer pixels as opaque squares produces blocky artifacts

In Figure 4.6 we see an example of how this problem is caused. The edge of a light foreground surface is rendered over a dark background in the $k = 2$ depth buffer (Figure 4.6a). Each depth buffer pixel which is hit by some point sample becomes a 4x4 opaque square, and *all* background pixels which lie behind these squares are treated as holes and discarded (Figure 4.6b). This creates large gaps in the image; many pixels are assumed to be holes even though in reality they are far from the edge of the surface. When these gaps are filled using interpolated colours, the surface colour bleeds into the background, creating noticeable artifacts.

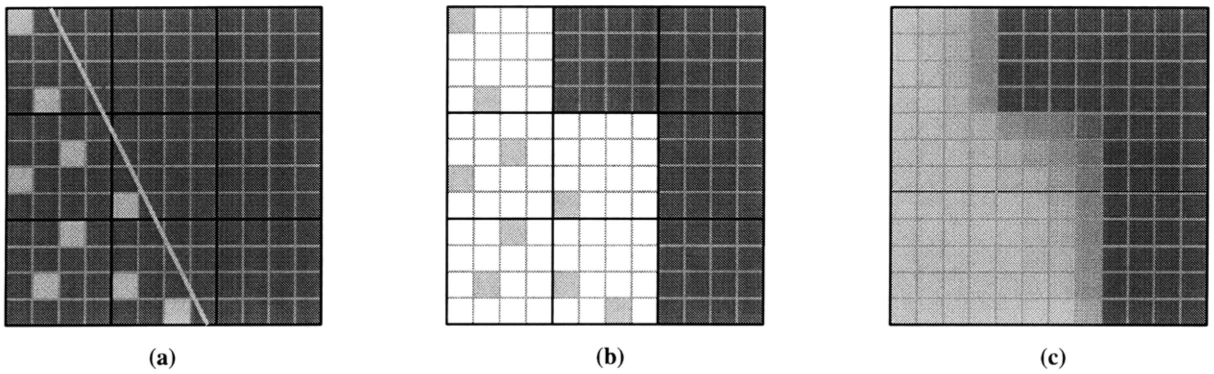


Figure 4.6: **(a)** A light foreground surface is rendered in the $k = 2$ depth buffer. The image pixels and depth buffer pixels are indicated using light and dark lines respectively. The light diagonal line indicates the actual edge of the foreground surface, and the light squares indicate pixels which are hit by point samples. **(b)** Background pixels which lie behind depth buffer pixels are discarded. **(c)** Final image: when the holes are filled the surface colour spreads out into the background.

Our strategy for dealing with this problem is to replace the binary decision with a continuous one. Each pixel will be assigned a weight in [0, 1] indicating a 'confidence' in the pixel, where a 1 indicates that the pixel is definitely in the foreground and a 0 indicates that the pixel definitely represents a hole and its colour should be ignored. The weights will then be used to blend between a pixel's original and interpolated colours. To see how these weights can be obtained, we observe that: (i) if a pixel is surrounded by depth buffer pixels which are closer to the viewer, then it is certainly a hole and should have weight 0; (ii) if a pixel is surrounded by depth buffer pixels which are further away from the viewer, then it is certainly in the foreground and should have weight 1; (iii) if a pixel is surrounded by depth buffer pixels both closer to and further away from the viewer, then it lies near the edge of the surface and should have a weight between 0 and 1. It therefore makes sense to consider *multiple* depth buffer pixels when assigning a weight to a single image pixel.

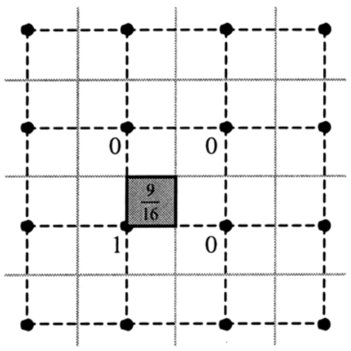


Figure 4.7: Computing the coverage of an image pixel. Depth buffer pixels are shown with solid gray outlines; the square mesh with vertices at their centers is shown using dashed lines. One vertex lies in front of the image pixel (coverage 1); the other three lie behind it (coverage 0). Using bilinear interpolation, the coverage of the pixel is 9/16.

We accomplish this by treating the k^{th} depth buffer as a square mesh with vertices at the depth buffer pixel centers. We then compute the amount by which the k^{th} depth buffer 'covers' an image pixel as follows: the center of the image pixel lies inside one of the squares of the mesh. The depth at each corner of the square is compared to the depth of the image pixel; a corner is assigned a *coverage* of 1 if it lies in front of the pixel and 0 otherwise. Finally, we take the coverage of the pixel to be the bilinear interpolation of these four corner coverages (this is reminiscent of the "Percentage Closer Filtering" used in [Reeves87] to generate antialiased shadows). The total coverage of a pixel is then the sum of the coverages from each depth buffer (capped at 1), and *weight* = 1 - *coverage.* This is illustrated in Figure 4.7.

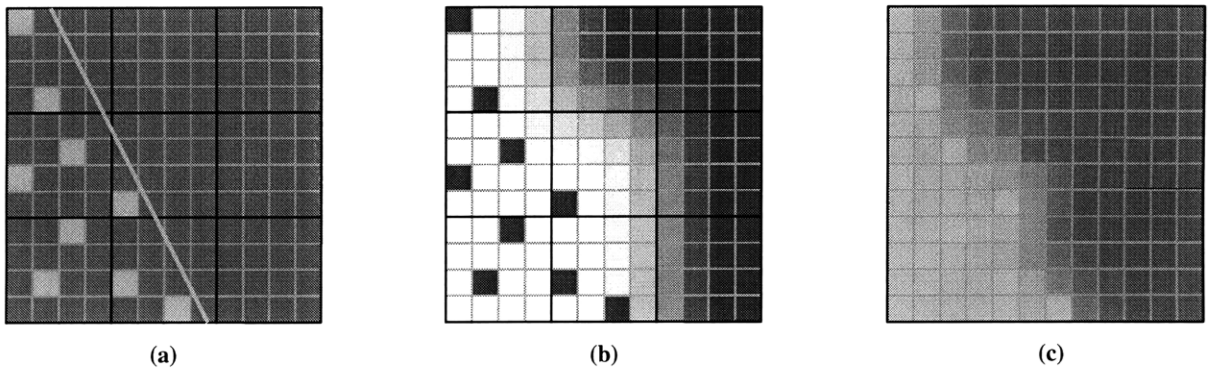


Figure 4.8: **(a)** Rendering the same light surface as in Figure 4.6a. **(b)** Weights assigned to image pixels (larger weights are darker). **(c)** Final image: the artifacts are less noticeable.

In Figure 4.8 we see how the example of Figure 4.6 is handled by this new approach. Figure 4.8b shows the weights that are assigned to image pixels using bilinear interpolation. These weights are used to blend between each pixel's original and interpolated colours; in the final image the artifacts are much less prominent (Figure 4.8c). Finally, Figure 4.9 provides a direct comparison of the two approaches; the latter method clearly produces better results.

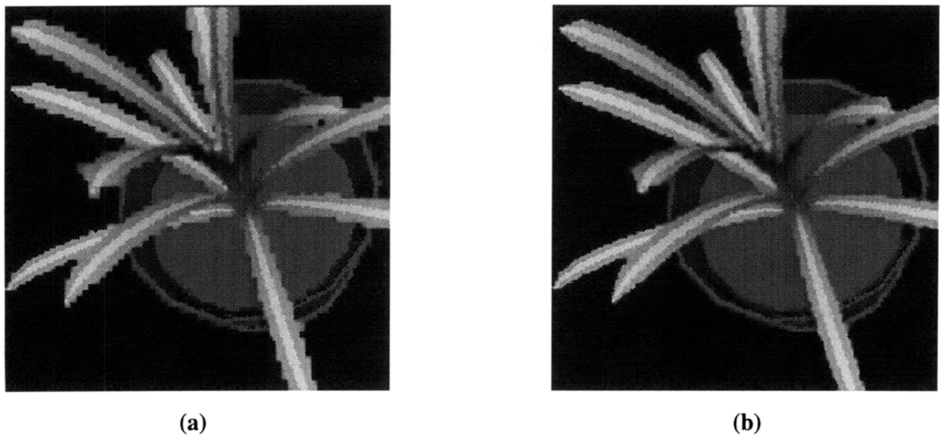


Figure 4.9: Comparison of methods for finding gaps. **(a)** Opaque squares, binary decisions. **(b)** Square mesh, bilinear coverages.

4.7 Filling gaps

To fill the gaps in the final image we must compute a colour for pixels with weight less than 1. This problem is not specific to point sample rendering; it is the general problem of image reconstruction given incomplete information. Our solution is a variation of the two phase "pull-push" algorithm described in [Gortler96] (which, in turn, was based on the work presented in [Mitchel87]). In the 'pull' phase we compute a succession of lower resolution approximations of the image, each one having half the resolution of the previous one. In the 'push' phase, we use these lower resolution images to fill gaps in the higher resolution images. This is illustrated in Figure 4.10 with a picture of a mandrill.

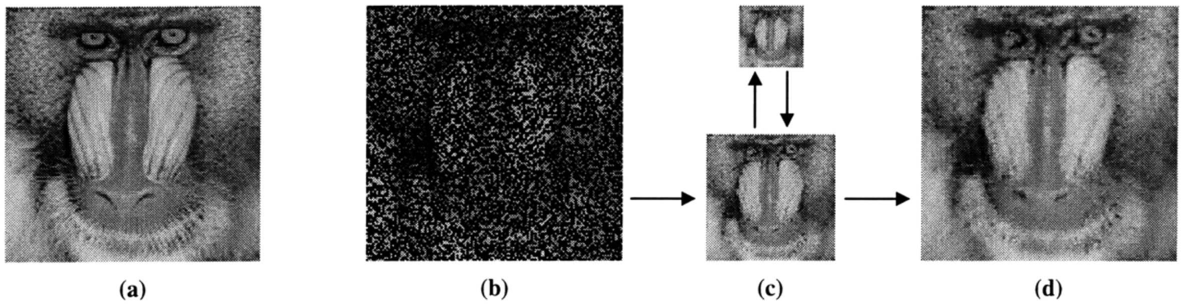


Figure 4.10: The pull-push algorithm. **(a)** Original image. **(b)** Incomplete image; 50% of the pixels have been discarded. **(c)** Lower resolution approximations. **(d)** Reconstruction.

The lower resolution approximations of the image are computed by repeatedly averaging 2x2 blocks of pixels, taking the weight of the new pixel to be the sum of the weights of the four pixels, capped at 1. Specifically, if $c^k_{x,y}$ and $w^k_{x,y}$ are the colour and weight of the (x, y) pixel in the kth low resolution approximation, then:

$$c^{k+1}_{x,y} = \frac{w^k_{2x,2y}c^k_{2x,2y} + w^k_{2x+1,2y}c^k_{2x+1,2y} + w^k_{2x,2y+1}c^k_{2x,2y+1} + w^k_{2x+1,2y+1}c^k_{2x+1,2y+1}}{w^k_{2x,2y} + w^k_{2x+1,2y} + w^k_{2x,2y+1} + w^k_{2x+1,2y+1}}$$

$$w^{k+1}_{x,y} = MIN(1, \quad w^k_{2x,2y} + w^k_{2x+1,2y} + w^k_{2x,2y+1} + w^k_{2x+1,2y+1})$$

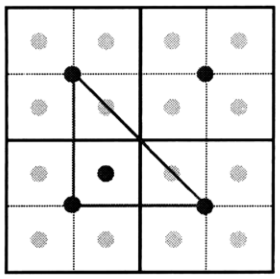
In the 'push' phase, to compute the final colour $c'^{k}_{x,y}$ of the (*x*, *y*) pixel in the *k*th low resolution image, we inspect its weight $w^{k}_{x,y}$. If $w^{k}_{x,y} = 1$ then we simply set $c'^{k}_{x,y} = c^{k}_{x,y}$. Otherwise, we compute an interpolated colour $\tilde{c}^{k}_{x,y}$ using as interpolants the pixels in the (*k*+1)st image. We then set

$$c'^{k}_{x,y} = w^{k}_{x,y} c^{k}_{x,y} + (1 - w^{k}_{x,y}) \tilde{c}^{k}_{x,y}$$

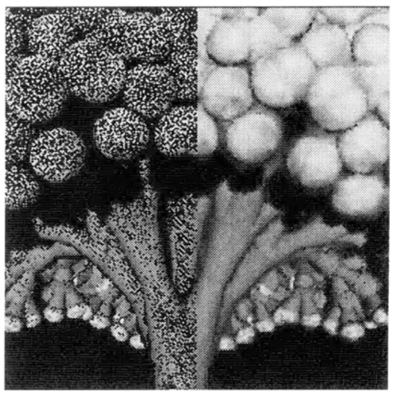
If we were to use bilinear interpolation to compute $\tilde{c}^{k}_{x,y}$, then we would interpolate the four closest pixels in the (*k*+1)st image with weights 9/16, 3/16, 3/16 and 1/16 (since the interpolation is always from one level up in the hierarchy, the weights never change). However, if we approximate these weights as ½, ¼, ¼ and 0, then the algorithm runs significantly faster (over 6 times faster using packed RGB arithmetic; see Appendix) without any noticeable degradation in image quality. Hence, we interpolate the three closest pixels in the (*k*+1)st image with weights ½, ¼ and ¼ (Figure 4.11a). For example,

$$\tilde{c}^{k}_{2x+1,2y+1} = \frac{1}{2} c'^{k+1}_{x,y} + \frac{1}{4} c'^{k+1}_{x+1,y} + \frac{1}{4} c'^{k+1}_{x,y+1}$$

with similar expressions for $\tilde{c}^{k}_{2x,2y}$, $\tilde{c}^{k}_{2x+1,2y}$, and $\tilde{c}^{k}_{2x,2y+1}$.



(a)



(b)

Figure 4.11: **(a)** An interpolated colour is computed using the three closest lower resolution pixels as shown. **(b)** Before gaps are filled (left) and after (right).

5 Object Modeling

The goal of the modeling process is to generate an adequate sampling of the object for a given target resolution/magnification using as few samples as possible. In this chapter we will describe the approach taken by our software system. The system is capable of automatically generating point sample representations for arbitrary synthetic objects; there is no user intervention.

In what follows we will refer to two different coordinate systems: object space and projection space. Object space is the coordinate system for the entire object; in these coordinates the object lies inside a unit sphere centered at the origin. Projection space is a per-projection coordinate system whose origin coincides with the object space origin and whose Z-axis points towards the viewer.

5.1 Obtaining Point Samples

The target resolution is specified by the user; the target magnification is assumed to be 1. The first step is to choose a set of orthographic projection directions from which to sample the object. We chose to do this in a data-independent manner, using the same set of projections for each object. While this does not guarantee that the object will be adequately sampled by the union of samples from all projections, it greatly simplifies the sampling process, and a judicious choice of projection directions will adequately sample most objects with high probability. In particular, a large number of projection directions distributed evenly about the sphere of directions will adequately sample most objects. We used the 32 directions obtained by subdividing the sphere of directions into octants, subdividing each of the eight resulting spherical triangles into 4 smaller triangles by joining the midpoints of the sides, then taking the centers of these 32 spherical triangles.

As explained in section 4.4, rather than sampling the orthographic projections on a square lattice at the target resolution, we sample on a denser equilateral triangle lattice. The density of the lattice is determined by the tolerance angle; if the tolerance angle is θ then the spacing between samples is *ds*·cosθ (where *ds* is the pixel side length), which gives us $2/\sqrt{3}\cos^2\theta$ samples per pixel. We typically use θ = 25° which translates to 1.4 samples per pixel.

Each point sample contains a depth, an object space surface normal, diffuse colour, specular colour and shininess. Table 5.1 gives the amount of storage associated with each of these; an entire point will fit into three 32 bit words. Surface normals are quantized to one of 32768 unit vectors. These vectors are generated using the same procedure described previously to generate projection directions, except that there are 6 levels of subdivision rather than 1, producing 32768 spherical triangles. The vectors are stored in a large lookup table which is accessed when the points are shaded at render time.

Field	# bits	Description
Depth	32	Single precision floating point
Object space normal	15	quantized to one of 32768 vectors
Diffuse colour	24	8 bits of red, green, blue
Specular colour	16	6 bits of green, 5 bits of red, blue
Shininess	8	Exponent for Phong shading
Total	**95**	

Table 5.1: Data associated with each point sample

The samples in a projection are divided into 8x8 **blocks**. This allows us to compress the database by retaining only those blocks which are needed while maintaining the rendering speed and storage efficiency afforded by a regular lattice (a regular lattice can be rendered quickly by using incremental calculations to transform the points, and it can be stored more compactly than a set of unrelated points as we only need to store one spatial coordinate explicitly). In addition, the use of blocks makes it possible to amortize visibility tests over groups of 64 points.

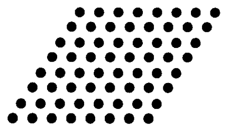


Figure 5.1: 8x8 block of point samples

There is a tradeoff involved in the choice of the block size. With smaller blocks it is easier to retain exactly those parts of a projection which are needed, resulting in smaller databases and faster rendering times. On the other hand, there is a fixed amount of overhead associated with each block (visibility calculations, initialization of incremental calculations, etc). With smaller blocks, this overhead is amortized over fewer points. We have found an 8x8 block to be optimal in terms of rendering speed (see section 5.3).

The texture of the object is sampled using an Elliptical Weighted Average filter [Heckbert89]. To use this filter, one would normally project a small circle onto the tangent plane of the sample point from the direction of view, producing an ellipse in the tangent plane (Figure 5.2a). This ellipse is then mapped into texture space and taken to be the support of an elliptical gaussian filter. However, this is a view-dependent filter whereas we desire our samples to be view-*independent.* We therefore use a *circle* in the tangent plane of the sample point with diameter equal to the length of a pixel diagonal (Figure 5.2b); this circle is then mapped to an ellipse in texture space as usual.

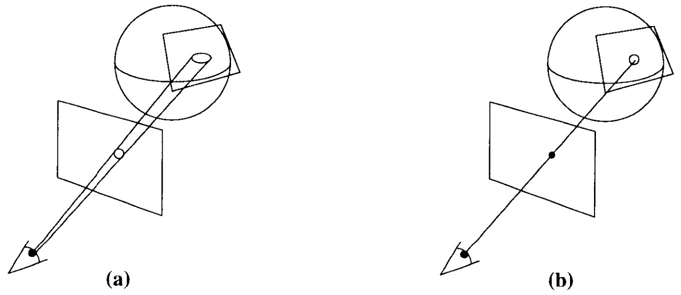


Figure 5.2: View dependent texture filtering (a) and view independent texture filtering (b).

5.2 Selection of Blocks

The union of point samples from all projections is assumed to form an adequate sampling of the object. The next task in the modeling process is to find a subset S of blocks which is still an adequate sampling but contains as little redundancy as possible. For this we use a greedy algorithm.

We start with no blocks ($S = \phi$). We then step through the list of orthographic projections; from each projection we select blocks to add to S. A block is selected if it corresponds to a part of the object which is not yet adequately sampled by S. Rather than attempt to exactly determine which surfaces are not adequately sampled, which is extremely difficult, we employ the heuristic of both ray tracing the orthographic projection and reconstructing it from S, then comparing the depths of the two images thus obtained (Figure 5.3). This tests for undersampled surfaces by searching for pixels which are 'missed' by the reconstruction. If a pixel is missing from a surface then, by definition, that surface is not adequately sampled. However, if no pixels are missing from a surface it does *not* follow that the surface must be adequately sampled, as there may be some other rotated/translated view in which the surface will contain a hole. Thus, to enhance the quality of the test it is repeated several times using various translations and rotations for the view. Typically, we use every combination of four rotations (0°, 22.5°, 45°, 67.5°) and four translations (either no offset or a one half pixel offset in both x and y), a total of sixteen tests.

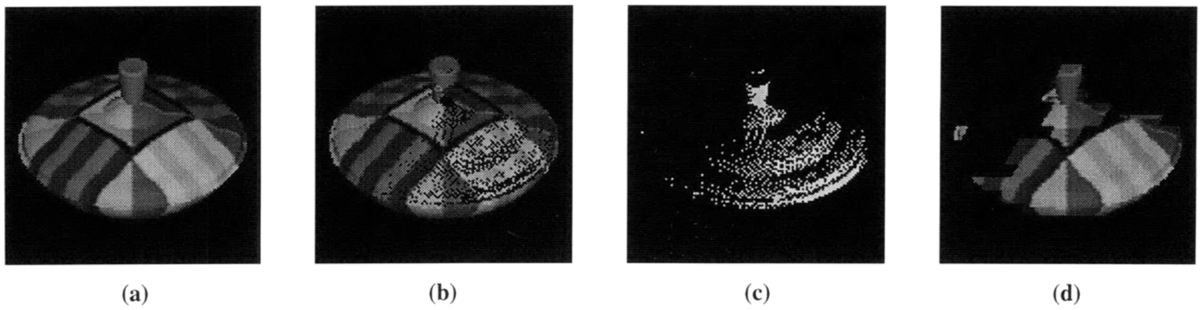


Figure 5.3: Block Selection. **(a)** Ray traced projection. **(b)** Projection reconstructed from current set of blocks. **(c)** Pixels where (a) and (b) have different depths. **(d)** Blocks added to set.

Since we start with no blocks, there will be a bias towards including blocks from earlier projections. Indeed, for the first two projections, which are taken from opposite directions, all the blocks which contain samples of the object will necessarily be included. To eliminate this bias we make a second pass through the projections in the same order; for each projection we remove from S any of its blocks that are no longer needed. This second pass has been observed to decrease the total number of blocks in the set by an average of 22%.

5.3 Notes on Parameters

There are a number of different parameters involved in object modeling, and for the most part it is difficult to determine *a priori* what the optimal settings for these parameters will be. For this we must resort to experimentation.

We varied the tolerance angle and block size for each of the five models shown in Figure 5.4. These models were selected to represent a wide range of object characteristics, from the large smooth surfaces of 'top' to the intricate features of 'kittyhawk'.

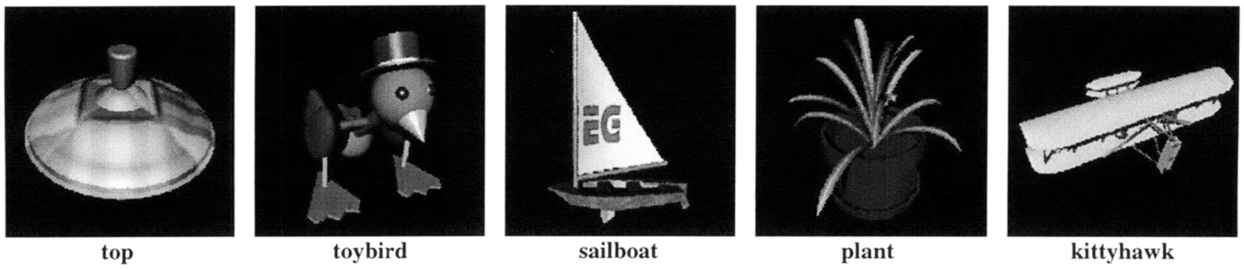
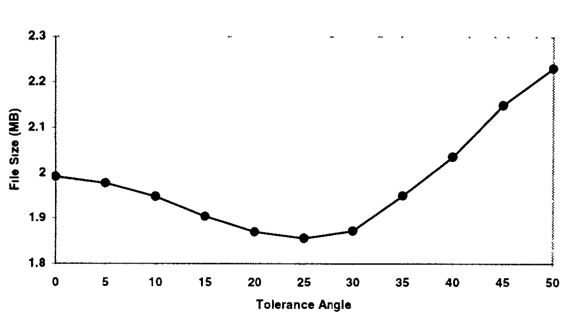


Figure 5.4: Models used to find optimal threshold angle, number of projections and block size

Figure 5.5 shows the average file size and rendering time as a function of tolerance angle; both are minimized for θ = 25° (1.4 samples per pixel). Figure 5.6 shows the average file size and rendering time as a function of block size. As expected, file size increases monotonically with block size due to the fact that larger blocks make it more difficult to retain only those portions of projections which are needed. The optimal block size in terms of rendering speed is 8x8.

One choice that *was* theoretically motivated was the decision to sample on an equilateral triangle lattice as opposed to a more traditional square lattice (section 4.4). Of course, there is nothing to prevent the use of a square lattice in the modeling algorithm; we simply do not expect it to perform as well. Figure 5.7 shows the average file size and rendering time as a function of tolerance angle for square lattice sampling; both are minimized for θ = 45° (2 samples per pixel). As predicted, the square lattice is inferior to the equilateral triangle lattice, producing models which are on average 14% larger and 5% slower.



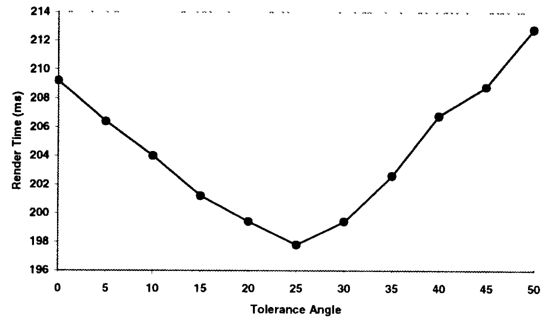
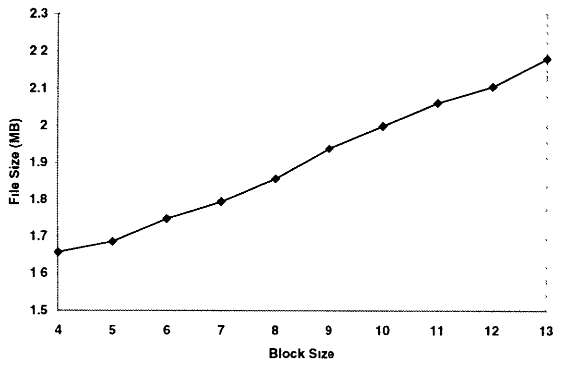


Figure 5.5: Average file size and render time as a function of tolerance angle.



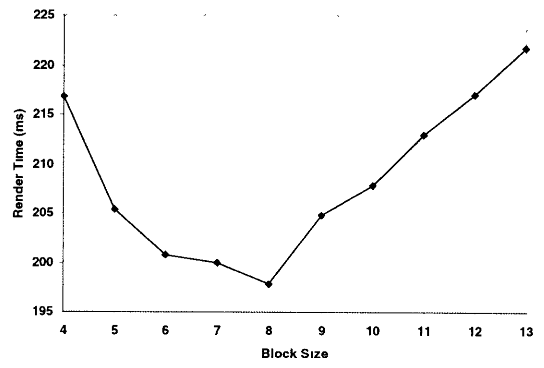
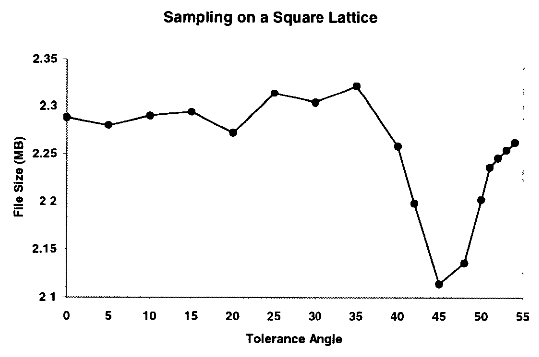


Figure 5.6: Average file size and render time as a function of block size



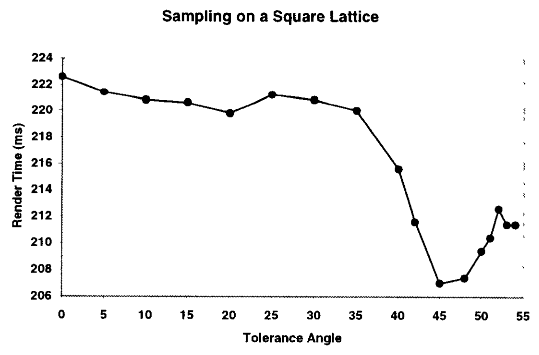


Figure 5.7: Average file size and render time as a function of tolerance angle for square lattice samplıng.

6 Visibility

The simplest way to construct an image from a point sample representation is to render *all* of the blocks. However, from a given viewpoint only a subset of the blocks are visible. If we can somehow identify this subset then we can save time by rendering only those blocks which contribute to the final image. Since it is generally impractical to compute this set exactly, we are interested in ways of quickly and conservatively estimating a set of visible blocks.

6.1 View Frustum Clipping

The first method which is used to test a block's visibility is to see if it lies within the **view frustum** – the region of space that is in the camera's field of view. As usual, the view frustum is defined by a 'near' plane, a 'far' plane, and the four planes which pass through the camera and the sides of the screen (Figure 6.1). We use a block's bounding sphere to determine whether or not the block lies entirely outside one of these planes. If it does, then we don't need to render it.

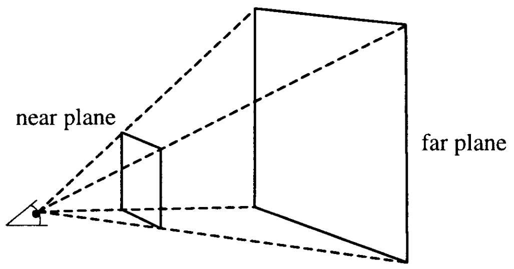


Figure 6.1: View Frustum

6.2 Spatial Subdivision

If a block lies inside the view frustum, then whether or not it is visible depends on whether or not it is obscured by some other part of the object, which in turn depends on the position of the camera. A brute force approach to determining visibility, then, is to subdivide space into a finite number of regions and to associate with each region a list of potentially visible blocks. However, we chose not to implement a visibility test based on spatial subdivision due to the large amount of storage required. Since this is a three dimensional problem, the number of bits of storage which are needed is proportional to the number of blocks times the cube of the resolution of the spatial subdivision.

6.3 Visibility Masks

One way to lessen the amount of storage required is to reduce the dimensionality of the problem. If we restrict our attention to viewpoints which lie outside the convex hull of the object, then whether or not a block is visible depends only on the *direction* from which it is viewed. Visibility as a function of direction is a two dimensional problem which we tackle using **visibility masks.**

The visibility mask is based on Normal Masks as introduced by Zhang and Hoff [Zhang97]. The sphere of directions is subdivided into 128 spherical triangles using the procedure described in section 5.1 with two levels of subdivision. Each triangle represents a set of directions. To each block, we assign a bitmask of length 128 - one bit per triangle. The k^{th} bit in a block's bitmask is 1 if and only if the block is visible (from a viewpoint outside the convex hull of the object) from some direction which lies inside the k^{th} triangle. We call this bitmask the **visibility mask** for that block.

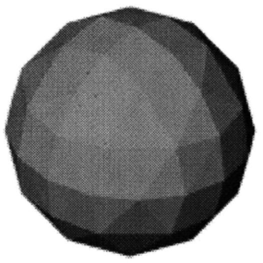


Figure 6.2: The sphere of directions is subdivided into 128 triangles. Each triangle contains a set of directions which is represented by a single bit in the visibility mask.

To compute the visibility masks, we render the entire set of blocks orthographically from a number of directions in each triangle (typically ten directions per triangle). Each point is tagged with a pointer to its block; this tag is used to determine the subset of blocks which contribute to the orthographic images. The visibility masks in this subset of blocks are updated by setting the appropriate triangle bit.

The simplest way to make use of these masks at render time is to construct a visibility mask for the screen. In this mask, the kth bit is set if and only if some direction from a screen pixel to the eye lies inside the kth triangle on the triangulated sphere of directions. We then AND this mask with each block's mask; a block is definitely not visible if the result of this test is zero (illustrated in two dimensions in Figure 6.3).

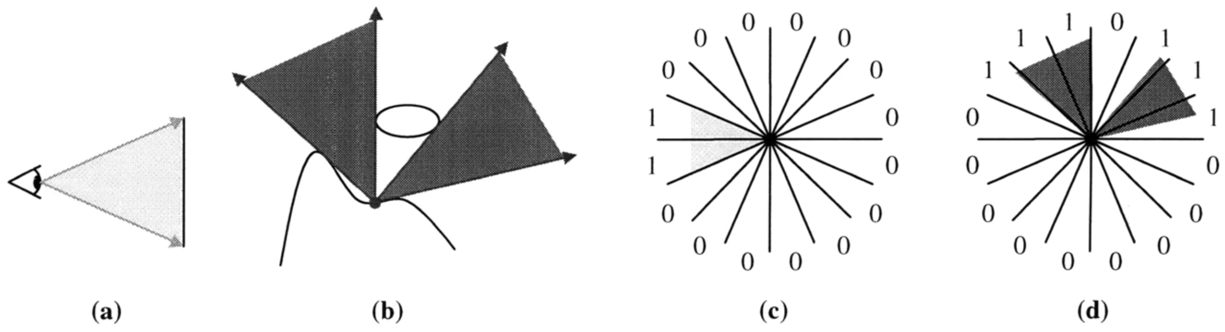


Figure 6.3: Using visibility masks to test visibility. **(a)** Eye-screen directions. **(b)** Set of directions from which a point is visible. **(c)** Screen mask generated from set of screen-eye directions. **(d)** Visibility mask of point. We can deduce that the point is not visible from the fact that the AND of the two masks is zero.

As explained in [Zhang97], however, using a single mask for the whole screen is overly conservative. We are, in effect, using the entire set of directions from the eye to the screen to approximate the set of directions subtended by a single block. In reality the latter set is almost certainly much smaller, so if we can improve the approximation then we will improve the rate of culling.

To this end, we subdivide the screen into 2^n x 2^n square regions. In order to be able to quickly determine which region contains a given block, we organize the regions into a binary space partitioning (BSP) tree of depth $2n$. This also enables us to gracefully handle the case in which a block straddles two or more regions by searching for the smallest node of the BSP tree which contains the block. A mask is generated for each node of the BSP tree, and each block is tested using the mask of the smallest node that contains the block. We note that the mask for each square region specifies which spherical triangles it intersects when these triangles are projected onto the screen. These masks can therefore be computed quickly and exactly by scan converting the projected triangles on the 2^n x 2^n screen. The mask of each internal BSP tree node is then the OR of the masks of its two children.

There is naturally a tradeoff involved in choosing n - larger n will result in improved culling, but more overhead. We have found that $n = 2$ gives the best performance on average (i.e. a 4 x 4 subdivision of the screen).

6.4 Visibility Cones

One of the basic visibility tests used in polygon rendering is "backface culling"; there is no need to render polygons whose normals are pointing away from the viewer. We can adapt this test to point sample rendering. We begin with the observation that, similar to polygons, the tangent plane at each point sample defines a half space from which the point is never visible. Taking the intersection of these half spaces over all the points in a block, we obtain a region in space, bounded by up to 64 planes, from which no part of the block is ever visible. It would be quite expensive to attempt to exactly determine whether or not the camera lies inside this region. Instead, we can approximate the region by placing a cone inside it. This **visibility cone** gives us a fast, conservative visibility test; no part of the block is visible from any viewpoint within the cone.

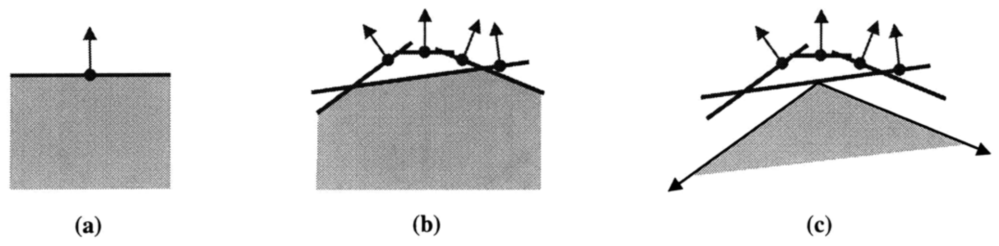


Figure 6.4: Visibility cone in two dimensions. **(a)** Half space from which point is never visible. **(b)** Region from which no part of the block is visible. **(c)** Cone inside region.

In what follows we describe one possible way to choose this cone. Let p_i ($1 \le i \le 64$) denote the points in a block, and let n_i denote their normals. The half space from which the i^{th} point is never visible is then defined by

$$(x - p_i)\cdot n_i \le 0 \qquad \textbf{(6.1)}$$

We wish to define the axis of the cone parametrically by

$$p + tn \quad t \in \mathbf{R} \qquad \textbf{(6.2)}$$

Take

$$p = \frac{\sum p_i}{64} \qquad \textbf{(6.3)}$$

And choose n such that $\min\limits_i (n \cdot n_i)$ is maximized. The rationale for this is that we want the cone to be as large as possible, so the apex angle should be as large as possible, so the worst discrepancy between n and some n_i should be minimized. We can solve for n exactly using the observation that it must be the spherical circumcircle of three of the n_i, so there are only a finite number of candidates to test. Note that all points in a block are obtained from a single orthographic projection taken from some direction d. It follows that $(-d)\cdot n_i > 0$ for each i, so by definition of n

$$\min_i (n \cdot n_i) \ge \min_i (-d \cdot n_i) > 0 \qquad \textbf{(6.4)}$$

Next, we find an apex q for the cone. Since q must on the line defined by equation **(6.2)**, it is of the form

$$q = p + t_0 n \qquad \textbf{(6.5)}$$

Since q must lie in each of the half spaces defined in **(6.1)**, for each i we have

$$(p + t_o n - p_i)\cdot n_i \le 0 \quad \Rightarrow \quad t_0 \le \frac{(p_i - p)\cdot n_i}{n \cdot n_i} \qquad \textbf{(6.6)}$$

Note that, from equation **(6.4)**, we do not have to worry about this denominator being zero or negative. We therefore set

$$t_0 = \min_i \left(\frac{(p_i - p) \cdot n_i}{n \cdot n_i} \right) \qquad \textbf{(6.7)}$$

Next we must choose the angle α between the axis of the cone and the sides of the cone. If β_i is the angle between n and n_i, then in order to ensure that the cone lies inside each of the half spaces in **(6.1)** we must have $\alpha \leq 90 - \beta_i$ for each i (Figure 6.5).

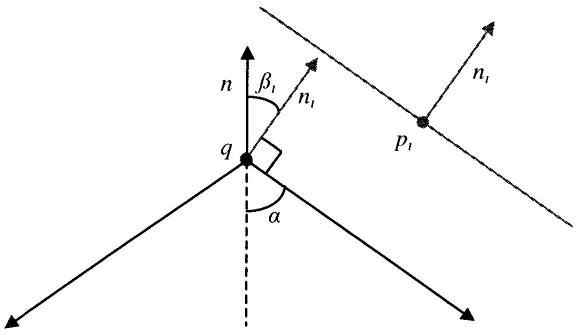


Figure 6.5: Constraining the apex angle of the visibility cone. The cone has apex q and axis n. The i^{th} point is at p_i with normal n_i. The angle between n and n_i is β_i. To ensure that the cone lies in the half space defined by $(x - p_i) \cdot n_i \leq 0$, we must have $\alpha \leq 90 - \beta_i$.

Since we want the apex to be as large as possible given this constraint, we take

$$\alpha = \min_i (90 - \beta_i) \qquad \textbf{(6.8)}$$

The cone is then defined by

$$\frac{(x - q) \cdot n}{|x - q|} \leq -\cos\alpha \qquad \textbf{(6.9)}$$

The following definition is equivalent and requires less computation to determine whether or not a point x lies inside the cone:

$$(x - q) \cdot \frac{n}{\cos\alpha} < 0 \qquad \textit{and} \qquad \left((x - q) \cdot \frac{n}{\cos\alpha} \right)^2 \geq (x - q)^2 \quad \textbf{(6.10)}$$

The cone is thus defined by the apex q and the vector $v = \dfrac{n}{\cos\alpha}$. Note that

$$\begin{aligned} \cos\alpha &= \cos\left(\min_i (90 - \beta_i) \right) = \max_i \left(\cos(90 - \beta_i) \right) = \max_i \left(\sin\beta_i \right) \\ &= \max_i \left(\sqrt{1 - \cos^2\beta_i} \right) = \sqrt{1 - \min_i \cos^2\beta_i} = \sqrt{1 - \min_i (n \cdot n_i)^2} \end{aligned} \qquad \textbf{(6.11)}$$

6.5 Experimental Results

In order to verify that the savings afforded by the visibility mask and the visibility cone outweigh their overhead, we rendered the five objects shown in Figure 5.4 and the eight shown in Figure 6.6 (thirteen in total) both with and without visibility cones and using a varying number of bits for the visibility masks. In Figure 6.7 we see that the best speedup - slightly over 18% - results from a combination of visibility cones and 128 bit visibility masks. Using only 32 bits for the visibility mask yields nearly the same speedup, so this smaller mask size may in some cases be preferable for reasons of simplicity and storage efficiency.

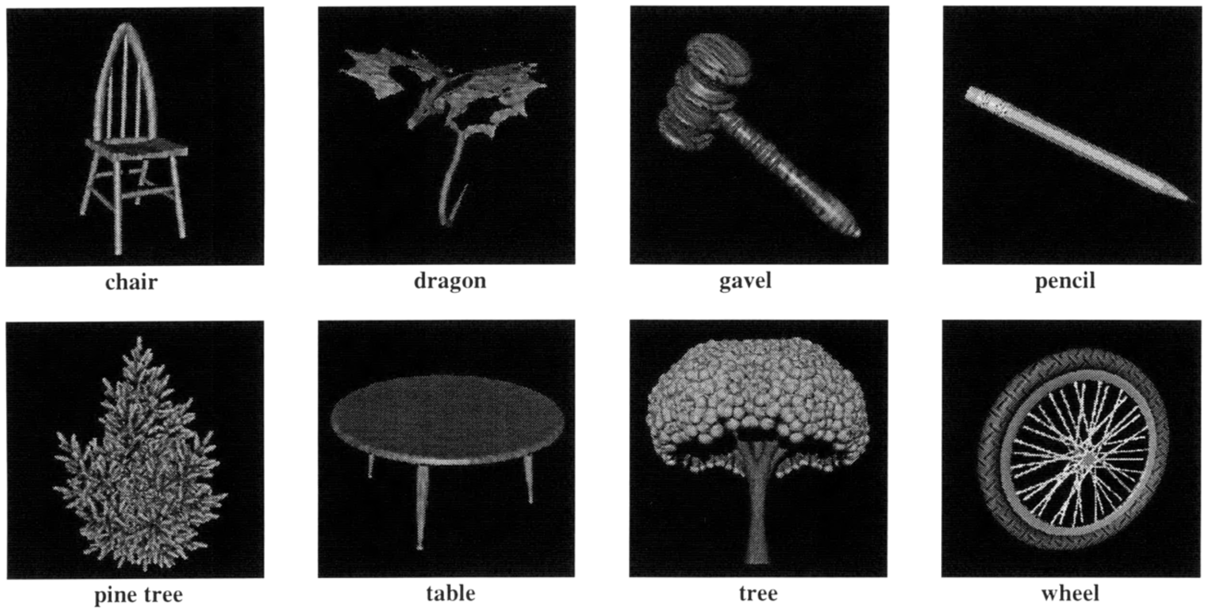


Figure 6.6: Models used for visibility experiments.

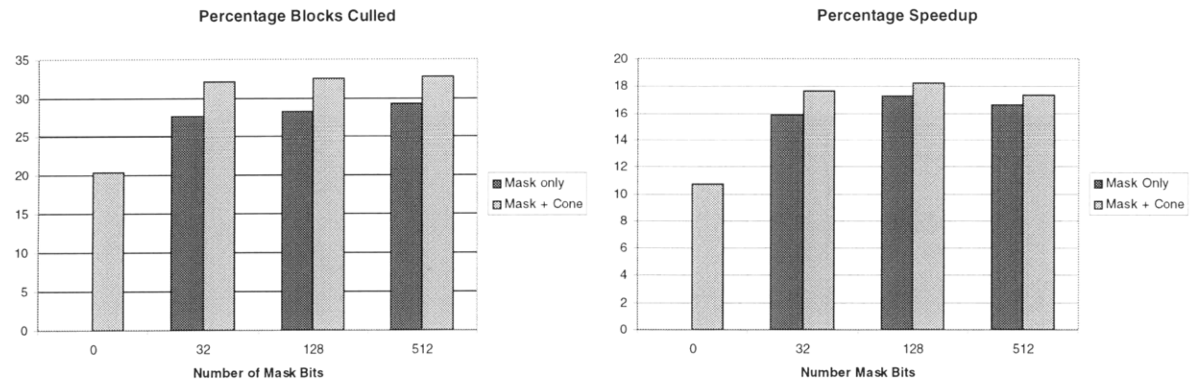


Figure 6.7: Average percentage blocks culled and average percentage speedup as a function of visibility tests used.

To find the best value for the parameter n mentioned in Section 6.3 (used to subdivide the screen into 2^n x 2^n square regions), we again rendered each of the thirteen models varying n from 0 (no subdivision) to 6 (64x64 subdivision). Figure 6.8 show the average percentage improvement in culling and percentage speedup over the base case $n = 0$ (no subdivision) as a function n. As expected, the culling improves as n is increased, but there is a tradeoff between the savings in rendering time and the overhead of the visibility test. The optimal speedup occurs at $n = 2$ (4x4 subdivision).

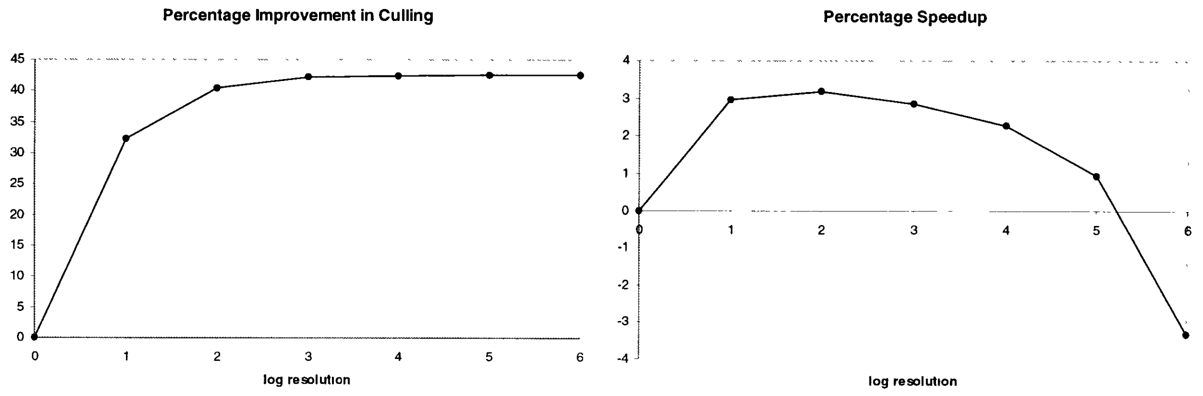


Figure 6.8: Average percentage improvement in culling and percentage speedup as a function of screen subdivision factor.

7 Rendering

At this point we are ready to put together the pieces of the previous chapters and discuss the point sample rendering algorithm in its entirety. There are five steps in the point sample rendering pipeline:

1. **Visibility Testing.** We conservatively estimate a set of visible blocks by clipping against the view frustum and testing the visibility cone and the visibility mask of each block.

2. **Block Warping.** Each potentially visible block is **warped** [Wolberg 90] into the destination image and the hierarchical Z-buffer. Points are not shaded at this stage; they are simply copied into the image buffer.

3. **Finding Gaps.** After all blocks have been warped we use the depths in the hierarchical Z-buffer to detect surface holes as described in section 4.6.

4. **Shading.** Shading is performed in screen space. Each non-background pixel with non-zero weight is shaded according to the current lighting model.

5. **Filling Gaps.** Finally, the surface holes which were detected in the third step must be coloured appropriately as described in section 4.7.

Three of these steps have been dealt with previously. In this chapter we will discuss the procedure used to warp blocks (sections 7.1 and 7.2) as well as the shading algorithm (section 7.3).

7.1 Basic Warping

In this section we will describe the basic Point Sample Rendering algorithm which maps each point sample to the nearest pixel in the destination image. We will show how the computations can be optimized using incremental calculations. We ignore for the time being the hierarchical Z-buffer which will be discussed in section 7.2.

Recall that the points in a block are stored in the coordinates of the orthographic projection from which the block was obtained. We begin, then, by computing, for each projection, the transformation from projection space to *camera space*. By camera space we mean the coordinate system in which the camera is at the origin and the screen is a square of side 2 centered and axis-aligned in the plane $z = 1$. We can write this transformation as

$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \mathbf{A}\begin{bmatrix} x \\ y \\ z \end{bmatrix} + \mathbf{v} = \begin{bmatrix} A_{xx} & A_{xy} & A_{xz} \\ A_{yx} & A_{yy} & A_{yz} \\ A_{zx} & A_{zy} & A_{zz} \end{bmatrix}\begin{bmatrix} x \\ y \\ z \end{bmatrix} + \begin{bmatrix} v_x \\ v_y \\ v_z \end{bmatrix} \qquad \textbf{(7.1)}$$

where (x, y, z) are the projection space coordinates of a point, (x', y', z') are the camera space coordinates, **A** is an orthogonal matrix giving the rotation and scale components of the transformation, and **v** is a translation vector. We can then compute the integer screen coordinates (u, v) from the camera space coordinates with a perspective divide and a scale/offset which maps [-1, 1] to [0, N] as follows:

$$u = \left\lfloor \frac{N}{2}\left(\frac{x'}{z'}+1\right)\right\rfloor \qquad v = \left\lfloor \frac{N}{2}\left(\frac{y'}{z'}+1\right)\right\rfloor \qquad \textbf{(7.2)}$$

where $\lfloor\cdot\rfloor$ denotes the greatest integer function and N is the resolution of the image. The complete transformation from (x, y, z) to (u, v) is thus

$$u = \left\lfloor \frac{N}{2}\left(\frac{A_{xx}x + A_{xy}y + A_{xz}z + v_x}{A_{zx}x + A_{zy}y + A_{zz}z + v_z} + 1\right)\right\rfloor$$

$$v = \left\lfloor \frac{N}{2}\left(\frac{A_{yx}x + A_{yy}y + A_{yz}z + v_y}{A_{zx}x + A_{zy}y + A_{zz}z + v_z} + 1\right)\right\rfloor \quad \textbf{(7.3)}$$

If we use the straightforward approach of computing the full transformation for each point, then we need 9 additions and 9 multiplications to compute *x'*, *y'*, *z'*, 1 reciprocal to compute 1/*z'*, and 2 more additions and 4 more multiplications to compute *u, v* for a total of 11 additions, 13 multiplications and 1 reciprocal for each point. We can reduce this to 11, 11 and 1 by absorbing the factor of N/2 in the precomputed matrix coefficients.

The destination image is Z-buffered. Since shading is performed in screen space after all blocks have been warped, points must be copied to the image 'as is'; at each image pixel we must be able to store a point's diffuse colour, specular colour, normal and shininess. The following pseudocode outlines the warping procedure:

```
for each point
      compute u, v, z'
      if (z' < depth(u, v))
            depth(u, v) = z'
            copy point to pixel(u, v)
      end if
end for
```

where `depth(u, v)` denotes the Z-buffer depth at (*u*, *v*) and smaller depths are closer to the viewer.

7.1.1 Incremental Calculations

We can reduce the number of operations required by using incremental calculations to take advantage of the fact that the points have been sampled on a regular lattice. Let (x_i, y_i, z_i) be the projection space coordinates of the i^{th} point in a block; let (x_i', y_i', z_i') be its camera space coordinates, and (u_i, v_i) its integer screen coordinates. If $(x_{i-1}, y_{i-1}, z_{i-1})$ and (x_i, y_i, z_i) are adjacent points on the same row of a block, then

$$x_i = x_{i-1} + dx \qquad \textit{and} \qquad y_i = y_{i-1} \quad \textbf{(7.4)}$$

where *dx* is the fixed spacing between point samples. If we define

$$\begin{aligned} p_i &= A_{xx}x_i + A_{xy}y_i + v_x \\ q_i &= A_{yx}x_i + A_{yy}y_i + v_y \\ r_i &= A_{zx}x_i + A_{zy}y_i + v_z \end{aligned} \quad \textbf{(7.5)}$$

then we can compute (u_i, v_i) much more efficiently as follows:

$$\begin{array}{lll} p_i = p_{i-1} + A_{xx}dx & q_i = q_{i-1} + A_{yx}dx & r_i = r_{i-1} + A_{zx}dx \\ x_i' = p_i + A_{xz}z_i & y_i' = q_i + A_{yz}z_i & z_i' = r_i + A_{zz}z_i \end{array}$$

$$u_i = \left\lfloor \frac{N}{2}\left(\frac{x_i'}{z_i'} + 1\right)\right\rfloor \qquad v_i = \left\lfloor \frac{N}{2}\left(\frac{y_i'}{z_i'} + 1\right)\right\rfloor \quad \textbf{(7.6)}$$

This now requires 8 additions, 5 multiplications and 1 reciprocal per point, a significant improvement. Here we are again assuming that N/2 is absorbed into other constants, and we note that $A_{xx}dx$, $A_{yx}dx$, and $A_{zx}dx$ are not counted as multiplications as these values can be pre-computed.

To move from one row points in the block to the next we use a slightly different set of constants, but the derivation is exactly the same. Since we sample on an equilateral triangle lattice, equation **(7.4)** becomes:

$$x_t = x_{i-8} + \frac{dx}{2} \qquad and \qquad y_i = y_{i-8} + \frac{\sqrt{3}}{2}dx \tag{7.4'}$$

We can then adjust the update equations for p_t, q_t and r_t accordingly.

7.1.2 Optimized Incremental Calculations

We can, in fact, compute (u_t, v_t) using even fewer operations. To see how this is possible, we rewrite **(7.3)** using the definitions in **(7.5)**, then add and subtract a constant from each fraction:

$$\begin{aligned} u_t &= \left\lfloor \frac{N}{2}\left(\frac{p_i + A_{xz}z_t}{r_i + A_{zz}z_t} + 1\right)\right\rfloor = \left\lfloor \frac{N}{2}\left(\frac{p_t - \frac{A_{xz}}{A_{zz}}r_t}{r_i + A_{zz}z_i} + \frac{A_{xz}}{A_{zz}} + 1\right)\right\rfloor \\ v_t &= \left\lfloor \frac{N}{2}\left(\frac{q_t + A_{yz}z_t}{r_t + A_{zz}z_t} + 1\right)\right\rfloor = \left\lfloor \frac{N}{2}\left(\frac{q_t - \frac{A_{yz}}{A_{zz}}r_t}{r_t + A_{zz}z_t} + \frac{A_{yz}}{A_{zz}} + 1\right)\right\rfloor \end{aligned} \tag{7.7}$$

Thus, if we define

$$\begin{aligned} p_t' &= p_i - \frac{A_{xz}}{A_{zz}}r_t \\ q_t' &= q_t - \frac{A_{yz}}{A_{zz}}r \\ r_t' &= r_t \end{aligned} \tag{7.8}$$

then we can rewrite **(7.6)** as follows:

$$\begin{aligned} &p_t' = p_{i-1}' + \left(A_{xx} - \frac{A_{xz}A_{zx}}{A_{zz}}\right)dx \qquad q_t' = q_{i-1}' + \left(A_{yx} - \frac{A_{yz}A_{zx}}{A_{zz}}\right)dx \qquad r_t' = r_{t-1}' + A_{zx}dx \\ &\qquad\qquad z_t' = r_t + A_{zz}z_t \\ &u_i = \left\lfloor \frac{N}{2}\left(\frac{p_t'}{z_t'} + \frac{A_{xz}}{A_{zz}} + 1\right)\right\rfloor \qquad v_t = \left\lfloor \frac{N}{2}\left(\frac{q_i'}{z_i'} + \frac{A_{yz}}{A_{zz}} + 1\right)\right\rfloor \end{aligned} \tag{7.6'}$$

This now requires 6 additions, 3 multiplications and 1 reciprocal per point. Again we assume that N/2 is absorbed into other constants, and we do not count additions and multiplications of constants that can be precomputed. The equations for moving from one row to the next are again derived analogously.

7.2 Hierarchical Z-buffer

In order to implement the surface reconstruction algorithm presented in Chapter 4, each block must be warped into the hierarchy of Z-buffers at a low enough resolution such that its points can't spread out and leave holes. If the image resolution is *NxN*, then the k^{th} depth buffer has resolution $\left\lceil \frac{N}{2^k} \right\rceil \times \left\lceil \frac{N}{2^k} \right\rceil$ where $\lceil \cdot \rceil$ denotes the least integer function. To select an appropriate depth buffer for the block, we start with the assumption that the block is an adequate sampling for orthographic views with unit magnification at *MxM* pixels. Suppose that the object is being rendered with magnification γ, and let $d = \min_i z_i'$ where z_i' is the camera space depth of the i^{th} point in the block as before. Then the worst case effective magnification of the block, taking into account the change in resolution, the object magnification, and the perspective projection, is

$$\frac{\gamma N}{Md} \tag{7.9}$$

To compensate for this, we need to choose the depth buffer such that

$$\frac{\gamma N}{Md2^k} \le 1 \tag{7.10}$$

Thus, we take

$$k = \max\left(0, \left\lceil \log\left(\frac{\gamma N}{Md}\right)\right\rceil\right) \tag{7.11}$$

It is fairly expensive to compute $d = \min_i z_i'$ exactly. As an approximation, we can instead use the distance *d'* from the camera to the bounding sphere of the block. This provides a lower bound for *d* and is much easier to calculate.

If $k = 0$, then the block is being warped into the usual depth buffer only and we do not need to make any modifications to the warping procedure of section 7.1. If $k > 0$, then we need to augment the warping procedure as follows:

```
for each point
        compute u, v, z'
        if (z' < depth(0, u, v))
                depth(0, u, v) = z'
                copy point to pixel(u, v)
                u' = u/2^k
                v' = v/2^k
                if (z' + threshold < depth(k, u', v'))
                        depth(k, u', v') = z' + threshold
        end if
end for
```

where `depth(k, u, v)` denotes the k^{th} Z-buffer depth at (u, v).

Note that instead of simply testing the point's Z value against the value in the depth buffer, we first add a small threshold. The reason for this is that the depth buffer will be used to filter out points which lie behind the foreground surface; the use of a threshold prevents points which lie *on* the foreground surface from being accidentally discarded. For example, suppose 3 points from a foreground surface and 1 point from a background surface are mapped to four different image pixels contained in a single depth map pixel (shown in one dimension in Figure 7.1). Without the threshold, all but the nearest of these points will be discarded (Figure 7.1a). The use of a threshold prevents this from happening (Figure 7.1b), allowing us to use all or most of the surface points when computing the final image. We found a good choice for the threshold to be three times the image pixel width; smaller thresholds produce images of lower quality, while larger thresholds begin to have difficulty discriminating between foreground and background surfaces. Figure 7.2 shows the striking difference in image quality which results from the use of a threshold; the left image was rendered using a threshold of three times the image pixel width.

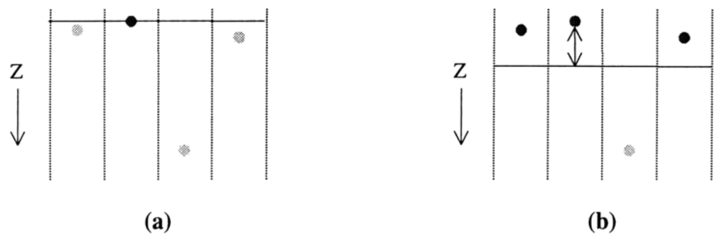


Figure 7.1: Four points are mapped to different image pixels contained in a single depth map pixel. **(a)** No depth threshold - only the nearest point is retained. **(b)** The use of a small threshold prevents points on the same surface as the nearest point from being discarded

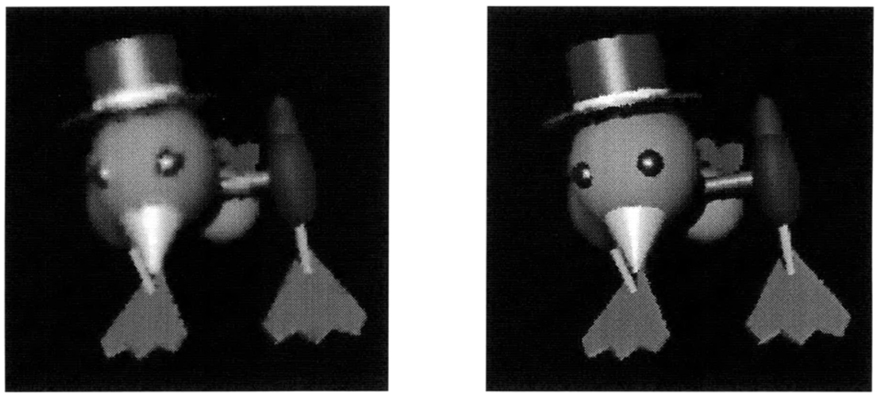


Figure 7.2: Depth threshold (left) vs. no threshold (right)

7.3 Shading

After all the blocks have been warped and weights have been assigned to image pixels as described in section 4.6, the non-background pixels with positive weight must be shaded according to the current lighting model. Performing shading in screen space after pixels have been weighted saves time by shading only those points which contribute to the final image. For our purposes the lighting model is Phong shading with shadows, but one could certainly implement a model which is more or less sophisticated as needed.

Since point sample normals are stored in object space, we perform shading calculations in object space to avoid transforming the normals to another coordinate system. We therefore need to recover object space coordinates for the points contained in the image pixels. Since we do not store the exact camera space x' and y' coordinates for points when they are copied to pixels in the image buffer, we assume that they are located at the pixel centers. Using the z' coordinates which are stored in the image Z-buffer, we can then transform the points from camera space to object space. The pixel centers form a regular square lattice, so this transformation can be done quickly using incremental calculations.

It is worth noting that because of the dense reflectance information contained in point sample models, the resulting specular highlights are of ray-trace quality, as can be seen in Figure 7.3.

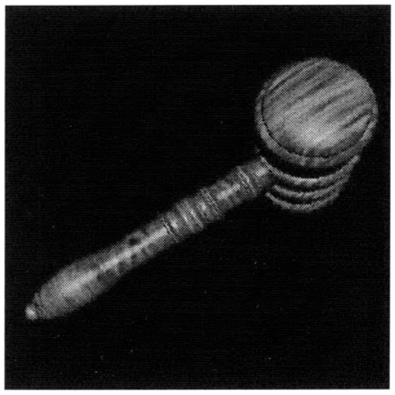


Figure 7.3: High quality specular highlights.

7.3.1 Shadows

Shadows are computed using shadow maps [Williams78]. For each light, we render the object from the light's viewpoint and use a hierarchical depth map to store Z values. Since we are *only* interested in creating a depth map, we don't have to worry about copying points' reflectance information into a destination image. The warping procedure is thus greatly simplified. For each block we again choose an appropriate level *k* in the depth buffer hierarchy and then warp the points as follows:

```
for each point
        compute u', v', z'
        if (z' < depth(k, u', v'))
                depth(k, u', v') = z'
        end if
end for
```

To calculate how much of a given point is in shadow, we first transform the point to *light space*, the coordinate system of the light's hierarchical depth map. We then compute a coverage for the point using almost exactly the same procedure as that used in section 4.6 for detecting gaps. The only difference is that we subtract a small bias from the point's depth before comparing it to the depths in the shadow map, as in [Williams78] and [Reeves87], to prevent surfaces from casting shadows on themselves due to small inaccuracies in the computation of the points' light space coordinates. In our case, these inaccuracies arise primarily from the fact that the points' *x'* and *y'* camera space coordinates are rounded to pixel centers.

Since the warping procedure used to create shadow maps is so simple, it is extremely fast. As a result, shadows can be computed with much less overhead than would normally be expected from an algorithm which re-renders the object for each light source. For example, Figure 7.4 was rendered at 256x256 pixels with three light sources - two directional lights and a spotlight. It took 204ms to render without shadows and 356ms with shadows - an overhead of less than 25% per light source.

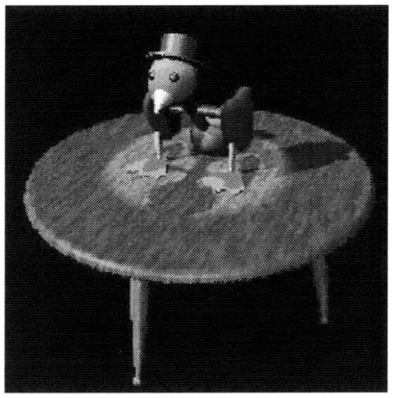


Figure 7.4: Fast shadows.

7.3.2 Backfacing Points

Since the normals of the points are not inspected when they are copied to the image buffer, it is entirely possible for some of them to be pointing away from the viewer. For solid objects, these points necessarily lie behind a foreground surface and will thus usually be filtered out (Figure 7.5a). However, points that lie near a corner may be missed by the filtering process because of the small threshold that is used to avoid discarding points on the foreground surface (Figure 7.5b). This can cause incorrectly coloured pixels to appear in the final image (Figure 7.6a). To fix this problem, we must eliminate backfacing points by assigning them a weight of zero (Figure 7.6b). This is performed during shading, which is when the points' normals are extracted.

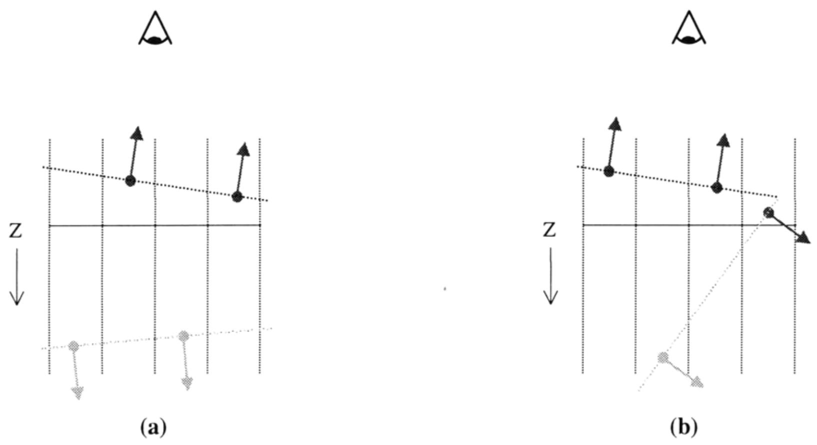


Figure 7.5: **(a)** Most backfacing points are filtered out because they lie behind a foreground surface. **(b)** Backfacing points near a corner can sneak in.

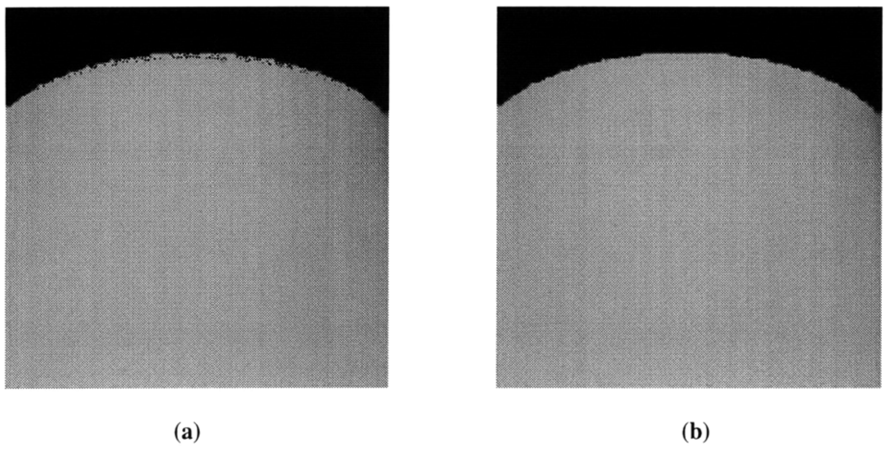


Figure 7.6: Close up of the edge of a cylinder. **(a)** Backfacing points are not eliminated; some pixels are coloured incorrectly. **(b)** Backfacing points are eliminated.

8 Results

Our test system was run as a single process on a 333MHz dual Pentium II with 512MB RAM. Table 8.1 gives file size, number of blocks and construction time for the fourteen models shown in Figure 8.1. The models were rendered with no shadows and a single directional light source at 256x256 pixels; Table 8.2 gives timing statistics for these settings. 'init' refers to the initialization stage during which all buffers are cleared and the screen visibility masks are constructed. The remaining times are for the five rendering steps described in Chapter 7. The largest of the models, 'Tree', was obtained from an Inventor model which contains 26,438 cones and ellipsoids and takes over 10 seconds to render using the Inventor Scene Viewer running on the same machine

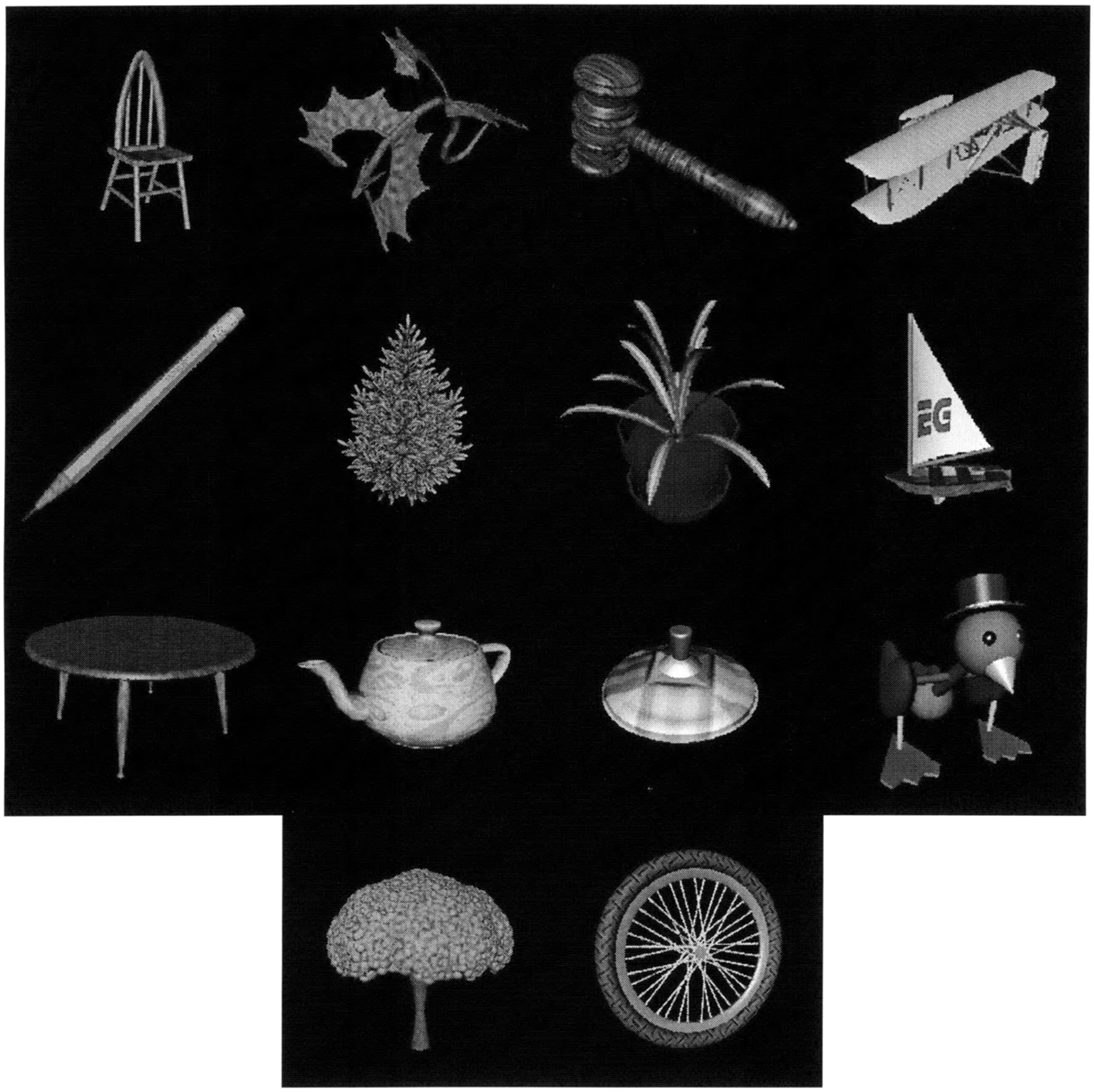


Figure 8.1: Models for which statistics are presented: Chair, Dragon, Gavel, Kittyhawk, Pencil, Pine Tree, Plant, Sailboat, Table, Teapot, Top, Toybird, Tree and Wheel.

	Chair	Dragon	Gavel	Kittyhawk	Pencil	Pine Tree	Plant
File size (MB)	1.32	0.96	1.20	1.73	0.38	7.12	2.33
Number of blocks	2666	2100	1779	3390	682	11051	3973
Construction time (hours : minutes)	0:15	0:18	0:08	1:02	0:09	1:35	0:34

	Sailboat	Table	Teapot	Top	Toybird	Tree	Wheel
File size (MB)	1.00	1.86	1.94	2.96	1.46	7.41	2.19
Number of blocks	1786	3128	2871	4473	2411	10694	4742
Construction time (hours : minutes)	0:13	1:18	0:34	0:43	0:11	4:08	1:04

Table 8.1: Model statistics

time (ms)	Chair	Dragon	Gavel	Kittyhawk	Pencil	Pine Tree	Plant
init	9	10	9	10	10	9	10
visibility	3	3	3	5	1	9	7
warp	44	36	38	62	15	226	74
find gaps	3	7	6	8	5	4	9
shade	14	18	20	23	12	25	24
pull	7	7	7	7	7	7	7
push	2	3	2	3	2	2	3
total	**82**	**84**	**85**	**118**	**52**	**282**	**134**

time (ms)	Sailboat	Table	Teapot	Top	Toybird	Tree	Wheel
init	9	10	10	9	10	9	10
visibility	2	6	5	6	4	11	8
warp	33	52	55	61	50	171	73
find gaps	6	7	7	3	8	6	4
shade	18	21	25	21	23	27	29
pull	7	7	7	6	7	7	6
push	2	3	2	2	3	3	2
total	**77**	**106**	**111**	**108**	**105**	**234**	**132**

Table 8.2: Timing statistics. All times are given in milliseconds for a software only implementation running on a 333MHz Pentium II.

Table 8.3 shows the extent to which the two visibility tests (visibility cone and visibility mask) are able to eliminate blocks in these images. Note that the visibility cone works quite well for objects with low curvature (e.g. Teapot, Top) but poorly for objects with high curvature (e.g. Chair, Pine Tree). This is due to the fact that surfaces with high curvature are visible from a larger set of directions. In most cases the visibility cone is outperformed by the visibility mask, but it is still able to provide some additional culling.

% culled	Chair	Dragon	Gavel	Kittyhawk	Pencil	Pine Tree	Plant
Neither	0	0	0	0	0	0	0
Cone only	13.5	16.1	19.3	14.3	23.8	1.2	29.3
Mask only	15.6	19.1	29.1	25.0	26.8	11.1	27
Both	19.0	24.1	31.9	26.6	30.5	11.6	34

% culled	Sailboat	Table	Teapot	Top	Toybird	Tree	Wheel
Neither	0	0	0	0	0	0	0
Cone only	27.2	26.0	32.1	39.6	23.0	4.9	20.3
Mask only	26.9	30.7	25.5	37.5	20.5	36.5	22.2
Both	31.0	36.0	35.3	46.4	27.5	36.8	25.3

Table 8.3: Culling statistics. Percentage of blocks eliminated as a function of which visibility tests are used.

We did not make any attempt to compress the models. The purpose of the test system was to investigate point sample rendering; compression is an orthogonal issue. Furthermore, the models are not so large as to make compression a necessity.

9 Discussion and Future Work

Point sample rendering allows complex objects to be displayed at interactive rates on today's inexpensive personal computers. It requires no hardware acceleration, it has only modest memory requirements, and it fully supports dynamic lighting.

The key to the speed of our algorithm is the simplicity of the procedure used to warp points into the destination image. All of the relatively expensive computations - shading, finding gaps, and filling gaps - are performed in screen space after all points have been warped. This places an absolute upper bound on the overhead of these operations, independent of the number and/or complexity of models being rendered. Ideally, one would also like the overall rendering time to be independent of model complexity. Although this is not the case, it is very nearly so; for example, the 'pine tree' inventor model is two orders of magnitude more complex than 'sailboat', yet the point sample rendering time is less than four times greater.

In the following sections we discuss some issues which were not fully addressed by this thesis and are therefore suitable subjects for further investigation.

9.1 Artifacts

For the most part, the images generated using point samples are of high quality, comparable or superior to images generated using polygons. However, they are not without artifacts. In particular, they suffer from rough edges and occasional surface holes.

9.1.1 Rough Edges

Without the precise geometric information available in polygon rendering, it is difficult to perform any sort of subpixel anti-aliasing, particularly at silhouette edges. There are clearly visible artifacts in Figure 8.1 due to this problem. A brute force solution is to increase the sampling density and render objects at double the image resolution. This works well (Figure 9.1b), but has the obvious disadvantage of being significantly slower. A more sophisticated variant of this solution would be to store both low and high resolution copies of each block, using the slower high resolution copy only for those blocks that lie on a silhouette edge. A less expensive alternative is to take the approach of [Levoy96] and artificially blur the image to hide the problem (Figure 9.1c).

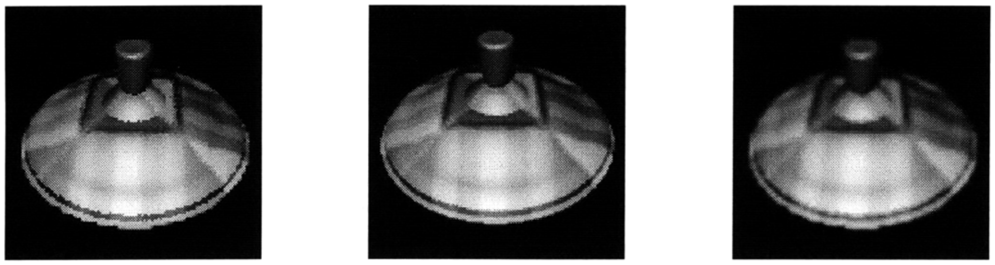

(a) (b) (c)

Figure 9.1: **(a)** Rendered at 128x128, no anti-aliasing. **(b)** Rendered at 256x256 and displayed at 128x128 (2x2 subpixel anti-aliasing). **(c)** Artificially blurred.

The noisy edges in Figure 9.1a are quite different from the 'jaggies' that one observes in polygon rendering. They seem, at first, to indicate a software or arithmetic error in the rendering process. However, they are simply an artifact of the manner in which surfaces are sampled. Figure 9.2 illustrates how these noisy edges can arise when rendering using point samples.

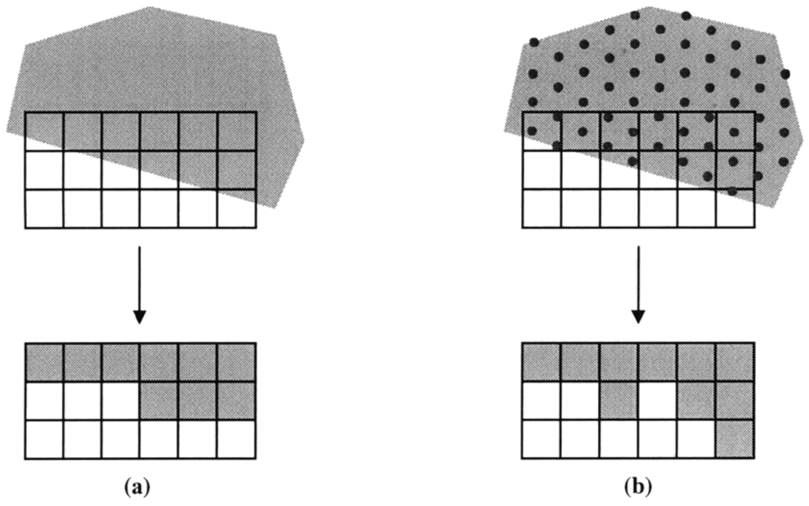


Figure 9.2: **(a)** A surface is rendered by scan converting polygons. Pixels are coloured if their centers lie inside a polygon, producing jagged edges. **(b)** A surface is rendered using point samples. Pixels are coloured if they are hit by a point sample, which can produce noisy edges.

9.1.2 Surface Holes

The algorithm described in section 4.6 to detect tears works well in general. In particular, it is guaranteed to eliminate surface holes in surfaces that have been adequately sampled and are rendered to a single level of the depth buffer hierarchy. However, it is possible for the blocks of a surface to be warped into two (or more) different levels of the hierarchy; in this case the algorithm provides no guarantee that pixels which lie at the interface between levels and represent holes will be given zero weight. In fact, we do occasionally see holes in such regions, as shown in Figure 9.3.

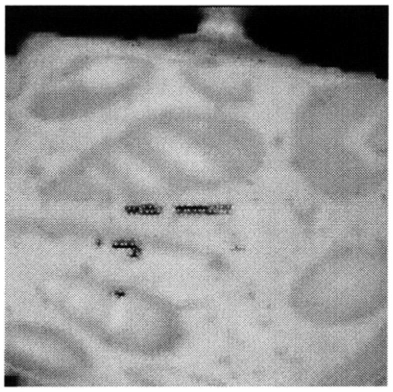


Figure 9.3: Surface holes occasionally appear at the interface between different levels of the depth buffer hierarchy.

It is not immediately clear how to deal with this problem. One possible approach is to detect when a pixel is partially covered by multiple levels of the depth buffer hierarchy. In this case the coverages can be weighted more heavily, for example they can be multiplied by a factor greater than one. This will make it more likely that background pixels in these in-between regions are given a coverage ≥ 1 and hence zero weight.

9.2 Object Modeling

The goal of the modeling process is to generate an adequate sampling of an object using as few point samples as possible. In chapter 5 we presented one possible approach; it is by no means the only one, and it can almost certainly be improved upon. This is definitely an area that merits further investigation.

9.2.1 Generating Point Samples

The algorithm in Chapter 5 made use of a single set of orthographic projections which did not depend on the object being modeled. The obvious alternative is to use a data *dependent* set of projections. The only constraint on the projections is that their union should provide an adequate sampling; one could therefore use geometric knowledge of the object to try to find a minimal set having this property. For simple objects such as 'Top' this set would contain relatively few projections, whereas for more complicated objects such as 'Kittyhawk' and 'Pine Tree' it would contain more.

One limitation of any algorithm which samples orthographic projections on a regular lattice is that it is difficult to properly sample thin spoke-like structures which are on the order of one pixel wide. As a result, such structures often appear ragged in rendered images (Figure 9.4). One possible solution to this problem is to vary the spacing between lattice points as needed. Another possible solution, which is itself an interesting area of investigation, is to have objects generate their own point samples rather than relying on some external sampling process. Each geometric primitive would produce a set of points which form an adequate sampling of the primitive, much in the same way that primitives 'dice' themselves into micropolygons in the Reyes rendering architecture [Cook87]. The difficulty lies in organizing these generated points into blocks or some other structure so that they may be stored and rendered efficiently.

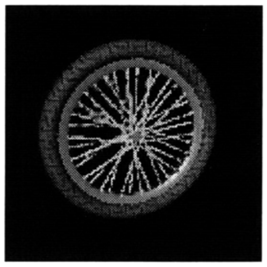


Figure 9.4: Ragged spokes.

9.2.2 Selection of Blocks

The greedy algorithm presented in Chapter 5 for selecting a set of blocks made no use of the fact that some blocks are 'better' than other blocks in the sense that they do a better job of sampling the surface of the object. In particular, a surface will be more densely sampled by projection directions close to the surface's normal. This observation could be used as the basis of an algorithm for selecting blocks. Each block could be scored based on the deviation of the point sample normals from the projection direction, or the spacing between the point samples on the surface of the object, or a combination of both. One could then walk through the entire set of blocks in the order of lowest score to highest score, discarding any blocks that are not needed to form an adequate sampling of the object. This algorithm would likely be much slower than the one presented in Chapter 5, but it could potentially produce a better selection of blocks.

9.2.3 Partial Blocks

An implicit assumption has been made that if a block is selected for inclusion in the final point sample model, then *all* the point samples in the block are retained. In fact there is no reason why this is necessary. If it can somehow be determined that a particular point sample is not needed to form an adequate sampling of the object, then that point sample can be discarded without harm. This decreases the disk storage space of the model, and reduces the amount of computation that must be performed at render time (using the notation of section 7.1.2, we would still need to compute p', q' and r' to preserve the integrity of the incremental calculations, but we would not need to compute z', u and v). It may not decrease the amount of memory required to store the model since, to optimize rendering time, the blocks may be stored as fixed-sized arrays of point samples.

Note that it is already necessary to handle partial blocks since not every ray which is traced in a given orthographic projection will intersect the object. In our software system, we use a bitmask to indicate which point samples are missing in the disk images, and we mark these point samples as invalid by giving them infinite depth when the blocks are loaded into memory. We can therefore discard unwanted point samples simply by marking them as invalid; we do not need to augment either the internal block representation or the rendering algorithm.

9.3 Hardware Acceleration

Although point sample rendering does not rely on hardware acceleration, it could certainly benefit from such support. Moreover, this hardware need not be special purpose; the block warping procedure is extremely simple and involves at most two conditionals, both of them Z-buffer tests. Furthermore, the algorithm exhibits a high degree of data locality as each point is warped independently and there are no references to global data structures such as texture maps or bump maps. The algorithm is therefore naturally vectorizable and parallelizable, and could easily be implemented on any SIMD or VLIW architecture

9.4 Real Objects

A topic which invariably arises in any discussion of modeling paradigms is the acquisition of models from real objects. In order to construct a point sample model for an object, it is necessary to somehow obtain both shape and reflectance information for that object. This is a challenging problem, but it has been addressed quite successfully by Sato, Wheeler and Ikeuchi [Sato97a]. In fact, one of the problems they encountered was that the models they generated - which included dense grids containing colour, normal and specular information - could not be rendered using standard software or hardware. It was therefore necessary for them to write software to display these models, and image generation was slow [Sato97b]. This problem would be solved by generating point sample representations of these models which would make full use of the reflectance information and could be rendered in real time.

9.5 Decimation

The density at which an object is sampled depends on the approximate resolution and magnification at which the object is to be viewed. In this thesis we have discussed at length how to handle magnification, but we have not dealt with the issue of decimation. Of course the algorithm will work without modification if either the resolution or the magnification are decreased, but there are two reasons why this (null) solution is not satisfactory:

i) Many more points will be rendered than are necessary to display the object; this will slow down rendering considerably.

ii) This would lead to the types of aliasing artifacts which are seen in non mip-mapped textures.

A natural approach is to use a form of mip-mapping, storing multiple representations of the object at various magnifications. However, since it is not possible to interpolate between consecutive representations (as in mip-mapped textures), we would need to choose a single representation at render time based on the object's current magnification. This could lead to 'popping' artifacts as the object moves closer to or further away from the viewer. A better approach may be to mip-map at the block level; this would also take into account the fact that different parts of the object are magnified by different amounts.

10 Conclusion

Point Sample Rendering is an algorithm which features the speed of image based graphics with quality, flexibility and memory requirements approaching those of traditional polygon graphics. Objects are represented as a dense set of point samples which can be rendered extremely quickly. These point samples contain precise depth and reflectance information, allowing objects to be rendered in dynamically lit environments with no geometric artifacts. Because the point samples are view-independent, point sample representations contain very little redundancy, allowing memory to be used efficiently.

We have introduced a novel solution to the problem of surface reconstruction, using a combination of adequate sampling and hierarchical Z-buffers to detect and repair surface tears. This approach does not sacrifice the speed of the basic rendering algorithm, and in certain cases it guarantees that images will not contain holes. We have also shown how a variant of percentage closer filtering [Reeves87] can be used to eliminate artifacts which result from a straightforward utilization of the hierarchical Z-buffer.

Because of its flexibility, Point Sample Rendering is suitable for modeling and rendering complex objects in virtual environments with dynamic lighting such as flight simulators, virtual museums and video games. It is also appropriate for modeling real objects when dense reflectance information is available. Since Point Sample Rendering has low memory requirements, it can be implemented on inexpensive personal computers.

In this thesis, we have shown that it is possible to render directly from point samples. Our software system, implemented on a 333MHz Pentium II, has demonstrated the ability of Point Sample Rendering to render complex objects in real time. It has also shown that the resulting images are of high quality, with good shadows and ray-trace quality specular highlights. With further research, particularly in the areas discussed in chapter 9, Point Sample Rendering will become a mature, robust rendering algorithm.

Appendix – 32 Bit Packed RGB Arithmetic

Ordinarily, performing arithmetic on colour values requires repeating each operation three times – once for each colour channel. On a 32 bit machine we can sometimes avoid this repetition of operations by storing RGB values in the following packed form:

```
00RRRRRRRR00GGGGGGGG00BBBBBBBB00
```

This allows up to four such values to be added as regular integers without worrying about overflowing from one colour channel to the next; the result of such an addition will be of the form

```
RRRRRRRRRRGGGGGGGGGGBBBBBBBBBB00
```

We used packed RGB values to speed up the pull-push algorithm by a factor of nearly three.

Pull

Recall that the lower resolution image approximations are computed by repeatedly averaging 2x2 blocks of pixels, taking the weight of the new pixel to be the sum of the weights of the four pixels, capped at 1. If $c^k_{x,y}$ and $w^k_{x,y}$ are the colour and weight of the (x, y) pixel in the kth low resolution approximation, then

$$c^{k+1}_{x,y} = \frac{w^k_{2x,2y}c^k_{2x,2y} + w^k_{2x+1,2y}c^k_{2x+1,2y} + w^k_{2x,2y+1}c^k_{2x,2y+1} + w^k_{2x+1,2y+1}c^k_{2x+1,2y+1}}{w^k_{2x,2y} + w^k_{2x+1,2y} + w^k_{2x,2y+1} + w^k_{2x+1,2y+1}}$$

$$w^{k+1}_{x,y} = MIN(1,\ \ w^k_{2x,2y} + w^k_{2x+1,2y} + w^k_{2x,2y+1} + w^k_{2x+1,2y+1})$$

We cannot directly implement these equations using packed RGB arithmetic because of the wc product terms. However, suppose we define $d^k_{x,y} = w^k_{x,y}c^k_{x,y}$. We can then implement the equations using the following sequential steps:

$$w_{sum} = w^k_{2x,2y} + w^k_{2x+1,2y} + w^k_{2x,2y+1} + w^k_{2x+1,2y+1} \qquad \textbf{(1)}$$

$$w^{k+1}_{x,y} = MIN(1, w_{sum}) \qquad \textbf{(2)}$$

$$d^{k+1}_{x,y} = d^k_{2x,2y} + d^k_{2x+1,2y} + d^k_{2x,2y+1} + d^k_{2x+1,2y+1} \qquad \textbf{(3)}$$

$$if \quad w_{sum} > 1 \quad then \quad d^{k+1}_{x,y} = \frac{d^{k+1}_{x,y}}{w_{sum}} \qquad \textbf{(4)}$$

Step **(3)** can now be performed using packed RGB values without, as mentioned before, worrying about overflowing from one colour channel to the next. Now if $w_{sum} \le 1$ then we're done. We don't need to perform the division of step **(4)**, and the definition of $d^k_{x,y}$ ensures that if $w_{sum} \le 1$ then each of the three colour channel sums is ≤ 255, so $d^{k+1}_{x,y}$ is guaranteed to still have 0's in between the colour channels. If $w_{sum} > 1$, then we need to perform the division. In the general case this cannot be accomplished using packed arithmetic; we must unpack, perform three divisions, and repack. However, in the common case all the weights are zero or one, so we will be dividing by 2, 3 or 4. Division by 2 or 4 is trivial; to divide by 3 we multiply by 1/3. The binary expansion of 1/3 is .01010101..., so this can be accomplished as follows:

```
// create the bitmask 00111111110011111111001111111100
#define RGBMASK 0x3fcff3fc

// Fast divide by 3
// rgb is the result of the sum in step (3)
rgb = ((rgb >> 2) & RGBMASK) +
      (((rgb & 0xff3fcff0) >> 4) & RGBMASK) +
      (((rgb & 0xfc3f0fc0) >> 6) & RGBMASK) +
      (((rgb & 0xf03c0f00) >> 8) & RGBMASK);
```

Note that if $w_{sum} = 3$, then each colour channel is at most 3x255 before the divide, so after the divide we are once again guaranteed to have 0's in between the colour channels.

Push

Recall that an interpolated colour $\tilde{c}^k_{x,y}$ is computed using as interpolants the three closest pixels in the $(k+1)^{st}$ image with weights ½, ¼ and ¼, e.g.:

$$\tilde{c}^k_{2x+1,2y+1} = \frac{1}{2} c'^{k+1}_{x,y} + \frac{1}{4} c'^{k+1}_{x+1,y} + \frac{1}{4} c'^{k+1}_{x,y+1}$$

with similar expressions for $\tilde{c}^k_{2x,2y}$, $\tilde{c}^k_{2x+1,2y}$, and $\tilde{c}^k_{2x,2y+1}$. The interpolated colour is then blended with the pixel's original colour:

$$c'^k_{x,y} = w^k_{x,y} c^k_{x,y} + (1 - w^k_{x,y}) \tilde{c}^k_{x,y}$$

It is straightforward to compute the interpolated colour using packed arithmetic:

```
rgb_interp = ((rgb1 >> 1) + ((rgb2 + rgb3) >> 2)) & RGBMASK
```

where `RGBMASK` is as defined previously. To blend the two colours together we again, in the general case, need to unpack, perform three blends, and repack. However, in the common case (all weights are zero or one) we either don't need to blend ($w^k_{x,y} = 0$) or we don't need to compute an interpolated colour at all ($w^k_{x,y} = 1$).

References

[Blinn76] James F. Blinn, Martin E. Newell, "Texture and Reflection in Computer Generated Images", *Communications of the ACM* (SIGGRAPH '76 Proceedings), Vol 19, No. 10, 1976, pp. 542-547

[Camahort98] Emilio Camahort, Apolstolos Lerios, Donald Fussell, "Uniformly Sampled Light Fields", Proc. 9th Eurographics Workshop on Rendering, Vienna, Austria, June 1998, pp. 117-130

[Catmull75] Edwin A. Catmull, "Computer Display of Curved Surfaces", Proceedings of the Conference on Computer Graphics, Pattern Recognition and Data Structure, May 1975, pp. 11-17.

[Chen93] Shenchang Eric Chen, Lance Williams, "View Interpolation for Image Synthesis", Proc. SIGGRAPH '93. In *Computer Graphics* Proceedings, Annual Conference Series, 1993, ACM SIGGRAPH, pp. 279-285

[Chen95] Shenchang Eric Chen, "QuickTime® - An Image-Based Approach to Virtual Environment Navigation", Proc. SIGGRAPH '95. In *Computer Graphics* Proceedings, Annual Conference Series, 1995, ACM SIGGRAPH, pp. 29-37

[Cline88] H. E. Cline, W. E. Lorensen, S. Ludke, C. R. Crawford, B. C. Teeter, "Two Algorithms for the three-dimensional reconstruction of tomograms", *Medical Physics*, Vol. 15, No. 3, May-June 1988, pp. 320-327.

[Cook87] Robert L. Cook, Loren Carpenter, Edwin Catmull, "The Reyes Image Rendering Architecture", *Computer Graphics* (SIGGRAPH '87 Proceedings), Vol. 21, No. 4, July 1987, pp. 95-102

[Csuri79] C. Csuri, R. Hackathorn, R. Parent, W. Carlson, M. Howard, "Towards an Interactive High Visual Complexity Animation System", *Computer Graphics* (SIGGRAPH '79 Proceedings), Vol. 13, No. 2, August 1979, pp. 289-299

[Curless96] Brian Curless, Marc Levoy, "A Volumetric Method for Building Complex Models from Range Images", Proc. SIGGRAPH '96. In *Computer Graphics* Proceedings, Annual conference Series, 1996, ACM SIGGRAPH, pp. 303-312

[Dally96] William J. Dally, Leonard McMillan, Gary Bishop, Henry Fuchs, "The Delta Tree: An Object-Centered Approach to Image-Based Rendering", Artificial Intelligence memo 1604, Massachusetts Institute of Technology, May 1996.

[Dischler98] Jean-Michel Dischler, "Efficiently Rendering Macro Geometric Surface Structures with Bi-Directional Texture Functions", Proc. 9th Eurographics Workshop on Rendering, Vienna, Austria, June 1998, pp. 169 - 180

[Gortler96] Stephen J. Gortler, Radek Grzeszczuk, Richard Szeliski, Michael F. Cohen, "The Lumigraph", Proc. SIGGRAPH '96. In *Computer Graphics* Proceedings, Annual Conference Series, 1996, ACM SIGGRAPH, pp. 43-54

[Greene93] Ned Greene, Michael Kass, Gavin Miller, "Hierarchical Z-buffer Visibility", Proc. SIGGRAPH '93. In *Computer Graphics* Proceedings, Annual Conference Series, 1993, ACM SIGGRAPH, pp. 231-238

[Heckbert87] Paul S. Heckbert, "Ray Tracing Jell-O® Brand Gelatin", *Computer Graphics* (SIGGRAPH '87 Proceedings), Vol. 21, No. 4, July 1987, pp. 73-74

[Heckbert89] Paul S. Heckbert, "Fundamentals of Texture Mapping and Image Warping", Masters Thesis, Dept. of EECS, UCB, Technical Report No. UCB/CSD 89/516, June 1989

[Hoppe92] Hugues Hoppe, Tony DeRose, Tom Duchamp, John McDonald, Werner Stuetzle, "Surface Reconstruction from Unorganized Points", *Computer Graphics* (SIGGRAPH '92 Proceedings), Vol. 26, No. 2, July 1992, pp. 71-78

[Laveau94] S. Laveau, O.D. Faugeras, "3-D Scene Representation as a Collection of Images and Fundamental Matrices", INRIA Technical Report No. 2205, February 1994

[Levoy85] Mark Levoy, Turner Whitted, "The Use of Points as a Display Primitive", Technical Report TR 85-022, The University of North Carolina at Chapel Hill, Department of Computer Science, 1985

[Levoy96] Mark Levoy, Pat Hanrahan, "Light Field Rendering", Proc. SIGGRAPH '96. In *Computer Graphics* Proceedings, Annual Conference Series, 1996, ACM SIGGRAPH, pp. 31-42

[Lipman80] A. Lippman, "Movie-Maps: An Application of the Optical Video disk to Computer Graphics", Proc. SIGGRAPH '80, 1980

[Maciel95] Paulo W. Maciel, Peter Shirley, "Visual Navigation of Large Environments Using Textured Clusters", Proc. 1995 Symposium on Interactive 3D Graphics, April 1995, pp. 95-102

[Mark97] William R. Mark, Leonard McMillan, Gary Bishop, "Post-Rendering 3D Warping", Proc. 1997 Symposium on Interactive 3D Graphics, pp. 7-16

[Max95] Nelson Max, Keiichi Ohsaki, "Rendering Trees from Precomputed Z-Buffer Views", 6th Eurographics Workshop on Rendering, Dublin, Ireland, June 1995, pp. 45-54

[McMillan95] Leonard McMillan, Gary Bishop, "Plenoptic Modeling: an image-based rendering system", In *Computer Graphics* Proceedings, Annual Conference Series, 1995, ACM SIGGRAPH, pp. 39-46

[Miller98] Gavin Miller, Steven Rubin, Dulce Poncelen, "Lazy Decompression of Surface Light Fields for Precomputed Global Illumination", Proc. 9th Eurographics Workshop on Rendering, Vienna, Austria, June 1998, pp. 281-292

[Mitchell87] Don P. Mitchell, "Generating Antialiased Images at Low Sampling Densities", *Computer Graphics* (SIGGRAPH '87 Proceedings), Vol. 21, No. 4, July 1987, pp. 65-69

[Pulli97] Kari Pulli, Michael Cohen, Tom Duchamp, Hugues Hoppe, Linda Shapiro, Werner Stuetzle, "View-based Rendering: Visualizing Real Objects from Scanned Range and Color Data", *Rendering Techniques '97*, Proc. Eurographics Workshop, pp. 23-34

[Reeves83] William T. Reeves, "Particle Systems - A Technique for Modeling a Class of Fuzzy Objects", *Computer Graphics* (SIGGRAPH '83 Proceedings), Vol. 17, No. 3, July 1983, pp. 359-376

[Reeves85] William T. Reeves, "Approximate and Probabilistic Algorithms for Shading and Rendering Structured Particle Systems", *Computer Graphics* (SIGGRAPH '85 Proceedings), Vol 19, No. 3, pp. 313-322

[Reeves87] William T. Reeves David H. Salesin, Robert L. Cook, "Rendering Antialiased Shadows with Depth Maps", *Computer Graphics* (SIGGRAPH '87 Proceedings), Vol. 21, No. 4, July 1987, pp. 283-291

[Smith84] Alvy Ray Smith, "Plants, Fractals and Formal Languages", *Computer Graphics* (SIGGRAPH '84 Proceedings), Vol. 18, No. 3, July 1984, pp. 1-10

[Sato97a] Yoichi Sato, Mark D. Wheeler, Katsushi Ikeuchi, "Object Shape and Reflectance Modeling from Observation", Proc. SIGGRAPH '97. In *Computer Graphics* Proceedings, Annual Conference Series, 1997, ACM SIGGRAPH, pp. 379-387

[Sato97b] Yoichi Sato, Personal Communication (Question period following presentation of [Sato97a], SIGGRAPH 1997)

[Schaufler95] Gernot Schaufler, "Dynamically Generated Imposters", MVD '95 Workshop "Modeling Virtual Worlds – Distributed Graphics", Nov. 1995, pp. 129-136

[Schaufler97] Gernot Schaufler, "Nailboards: A Rendering Primitive for Image Caching in Dynamic Scenes", *Rendering Techniques '97*, Proc. Eurographics Workshop, pp. 151-162

[Toborg96] Jay Toborg, James T. Kajiya, "Talisman: Commodity Real-time 3D Graphics for the PC", *Computer Graphics* (SIGGRAPH '96 Proceedings), August 1996, pp. 353-363

[Turk94] Greg Turk and Marc Levoy, "Zippered Polygon Meshes from Range Images", Proc. SIGGRAPH '94. In *Computer Graphics* Proceedings, Annual Conference Series, 1994, ACM SIGGRAPH, pp. 311-318

[Westover90] Lee Westover, "Footprint Evaluation for Volume Rendering", *Computer Graphics* (SIGGRAPH '90 Proceedings), Vol. 24, No. 4, August 1990, pp. 367-376.

[Wolberg90] G. Wolberg, **Digital Image Warping,** IEEE Computer Society Press, Los Alamitos, California, 1990.

[Wong97] Tien-Tsin Wong, Pheng-Ann Heng, Siu-Hang Or, Wai-Yin Ng, "Image-based Rendering with Controllable Illumination", *Rendering Techniques '97*, Proc. Eurographics Workshop, pp. 13-22

[Zhang97] Hansong Zhang and Kenneth E. Hoff III, "Fast Backface Culling Using Normal Masks", Proc. 1997 Symposium on Interactive 3D Graphics, pp. 103-106