IMPOSTORS FOR PARALLEL INTERACTIVE COMPUTER GRAPHICS

BY

ORION SKY LAWLOR

B.S., University of Alaska at Fairbanks, 1999
B.S., University of Alaska at Fairbanks, 1999
M.S., University of Illinois at Urbana-Champaign, 2001

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2005

Urbana, Illinois

Abstract

We demonstrate an interactive parallel rendering system based on the impostors technique. Impostors increase the latency tolerance of an interactive rendering system, which allows us to use the power of a parallel machine even at high resolutions and framerates. Impostors also decrease the required rendering bandwidth, which makes possible the interactive use of a variety of advanced rendering techniques. These techniques are demonstrated by the interactive high-quality rendering of very large detailed models on large distributed-memory parallel machines.

To TRUTH, without which everybody would be lying.

Acknowledgments

This work was made possible by the efforts of hundreds of teachers and friends over the span of nearly three decades. I can only mention a few here.

Thanks to my adviser, Dr. Kale, who provided me continual support and a steady stream of good ideas. May you always have enough good students to implement your grand designs. Thanks to my committee members, for forbearance, constructive suggestions, and excellent classes. Thanks to the members, past and present, of the parallel programming lab—first for camaraderie, but also for long-suffering patience and help with my bugs. May your pizza never run out.

Thanks to my wife, who for me left her beloved homeland and toiled under a harsh alien sky for four years. Thanks to dad, who taught me the amazing results of endless patience, craftsmanship, and hard work. Thanks to my mother, who instilled in me a fascination with and respect for the dialectic; and who demanded I receive a world-class education while growing up 120 miles from the nearest stoplight.

Table of Contents

Chapter

Chapter 1

Introduction

A central goal of computer graphics research is to **accurately** render **large** environments full of **detailed** geometry very **quickly**. For example, global illumination methods can accurately render the details of surface and subsurface light transport, but have well-known limitations on model size and speed. Highly detailed models and scene databases are available, but they render slowly. Modern graphics hardware rendering can render fairly large models quickly, but sacrifices rendering quality and light transport detail. Because none of today's systems provide enough accuracy, size, detail, and speed, the present goal of computer graphics is to simply increase these quantities.

The approach we pursue is motivated by a simple observation: in interactive rendering, as the camera moves through the scene, the appearance of most objects does not dramatically change from frame to frame. Typical rendering methods re-render all the objects for each frame; but the existing *impostors* [MS95] or *image caching* [SS96] approach renders each object once, then caches and reuses the rendering over several frames. The 2D rendering of each object is stored as an alpha-blended texture image, and can be drawn as a texture-mapped polygon—an "impostor." Because drawing a texture is faster than rendering the object, the impostors approach can be used to amortize the cost of complicated objects and high-quality rendering over several frames. We describe the features, advantages, and disadvantage of the impostors approach to image assembly in Chapter 3.

The enabling technology we explore is to render the impostors using a parallel server. The challenge with this approach is that the scene partitioning into impostors, and the amount of rendering effort required for each part of the scene, are both highly viewpoint dependent. The adaptive and dynamic nature of the problem makes impostor rendering difficult to implement in parallel. To support this and other dynamic parallel

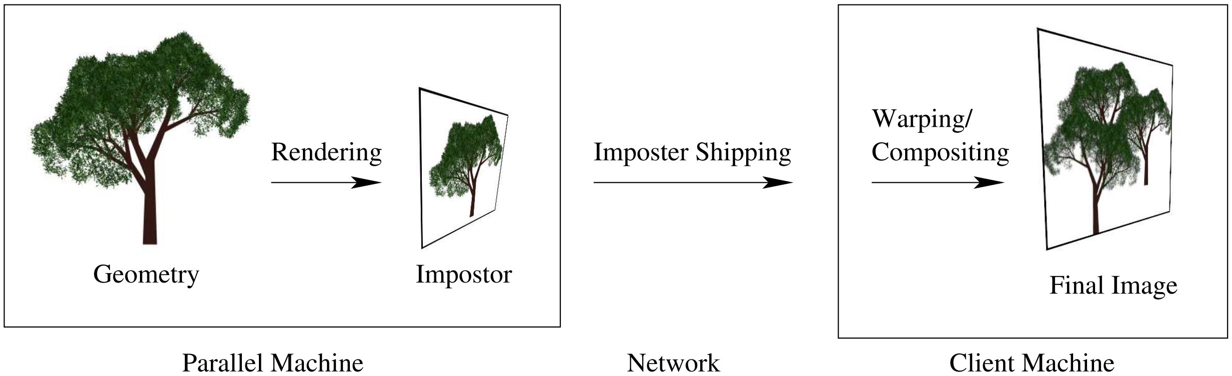


Figure 1.1. Our client/server graphics architecture. The parallel server keeps the model geometry, and renders pieces of it to impostor textures. These textures are shipped across the network to the client machine, which assembles and displays them.

problems, we have developed a production-quality parallel work migration and load balancing infrastructure [LK03], as described in Chapter 4 and Appendix A. We then used this parallel infrastructure to develop a high-performance parallel impostor library and set of applications.

The parallel machine that renders impostors is potentially miles away from the user. Impostors thus must be transmitted to a serial client machine sitting on the user's desk. The client machine then assembles the 3D scene by warping and compositing together individual 2D impostor images, using ordinary graphics hardware in the usual way. The overall architecture and data flow is as shown in Figure 1.1. Our client/server network protocol, PCS, and object communication language, the PUP framework, are described in Appendix B.

The underlying performance equations governing the parallel impostors method are derived and analyzed in Chapter 2. The main result of this analysis is that the performance of a parallel impostor system is generally not limited by the rendering rate. Instead, the performance-limiting design numbers are normally the network bandwidth to the client, and the impostor reuse rate. The impostor reuse rate, which measures how often impostors can be reused before being re-rendered, is a function of the impostor depth range, which is in turn determined by how large the world is, and how well we decompose the world geometry into impostors.

A realtime rendering system must produce on the order of a hundred million pixels per second, which is only 10ns per pixel—not enough time to use particularly sophisticated rendering techniques. But we show that because of the speedup of parallelism and the reuse of impostors, the parallel impostors technique allows each processor to spend up to a microsecond on each pixel, and yet still maintain high-quality realtime performance. This additional rendering time allows us to substantially improve the **accuracy** and **detail** of the resulting images. For example, we describe affordable techniques to improve the rendering accuracy by antialiasing the impostor geometry, computing good approximations to both direct as well as indirect lighting, and procedurally generating needed detail, as described in Chapter 7.

The leisurely rendering rate made possible by parallel impostors also allows us to render very **large** geometries at real-time frame-rates. We first demonstrate this by examining the performance of the overall parallel system for a simple but large-scale particle rendering system in Chapter 6. We then demonstrate a more complicated example by generating and exploring an extremely large, detailed model of the University of Illinois at Urbana-Champaign campus in Chapter 9.

Chapter 2

Fundamentals

This section analyzes the fundamental relationships that motivate and limit the parallel impostors technique. The overall conclusion of this section is that the speedup provided by parallel computation, and the rendering reuse provided by impostors, can both amplify a surprisingly slow rendering rate to produce a high resolution high quality realtime system.

2.1 Graphics Hardware Performance

We have performed a detailed cost analysis for the rendering performance of modern graphics hardware, as shown for a variety of cards in Appendix E.2. The analysis shows that the achieved *fill rate*, or overall system throughput as measured in pixels per second, drops dramatically for small triangles; but as we show, impostors can restore the fill rate without affecting image quality.

The time $t_\triangle$ to draw a triangle on modern graphics hardware is well modeled by

$$t_\triangle = \max(\alpha, \beta(s + \gamma r)) \tag{2.1}$$

Here, α is the triangle setup time, typically around 100ns/triangle. β is the pixel time, typically around 2ns/pixel. α is also the inverse of the maximum triangle rate (triangles per second), and β the inverse of the pixel fill rate (pixels per second). s is the total area in pixels in the triangle, and r is the number of rows of pixels in the triangle. γ is the per-row pixel pipeline startup time, measured in pixels per row. We find $\gamma = 3$ pixels/row fits most modern cards well; Appendix E.2 gives measured α and β for a variety of graphics hardware.

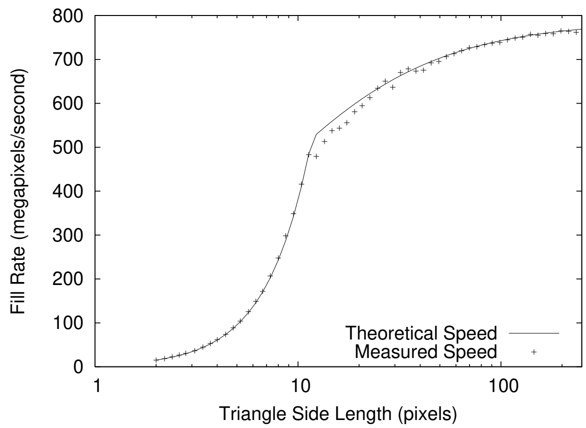


Figure 2.1. Achieved fill rate for nVidia GeForce3/Pentium 4.

The max() in the performance model is a natural result of the on-chip parallelism of modern graphics hardware. In a pipeline, throughput is limited by the slowest component, and this model contains two pipeline components: α, to represent vertex and triangle setup; and the β term, which represents row setup and pixel rendering.

The first thing to notice about this model is that in order to fully utilize the card's fill rate, the fill rate β term must dominate. If the triangle rate α term dominates, we could increase the area s of each triangle without increasing the per-triangle time. To see this another way, examine the achieved pixel fill rate B_C (pixels per second):

$$B_C = \frac{s}{t_\triangle} = \frac{s}{\max(\alpha, \beta(s+\gamma r))}$$

$$B_C = \min(\frac{s}{\alpha}, \frac{1}{\beta(1+\gamma r/s)}) \tag{2.2}$$

For small triangles, the triangle setup α term limits the overall performance. Thus to achieve close to peak fill rate, the triangles must be large. For current hardware, "large" means triangles on the order of 10 pixels across—Figure 2.1 shows a typical plot of this; Appendix E.2 shows the effect in detail.

But to accurately represent curved geometry, we would prefer to render using small triangles, often just a few pixels across. As a piecewise linear approximation to curved geometry, triangles of physical size h have a geometric discretization error in $O(h^2)$, so directly using large triangles instead of small triangles may create unacceptable geometric distortion.

Impostors provide a solution to this problem. By first rendering a set of accurate, small triangles into a single large impostor, we can then perform more efficient rendering with the larger impostor. If the source triangles are very small, on the order of a single screen pixel, rendering the larger impostor could be 50 times faster, because rendering the large impostor is fillrate-dominated, while rendering the small triangles is triangle setup-dominated. Of course, we still have to prepare the impostor image somehow, but if impostors are reused across many frames, this startup overhead can be amortized away.

This situation is remarkably similar to that encountered in parallel computing when sending many small messages—the per-message costs overwhelms the per-byte cost, and link utilization is low. The similar solution in parallel computing is to use message combining [KKV03], where small messages are assembled into larger messages.

2.2 Impostor Parallax Update Rate

As the camera moves, our 2D impostor images must be updated to follow the true 3D geometry of the object they represent. But impostors only provide some benefit if they can be reused for several frames; otherwise we might as well just draw the geometry directly every frame instead of drawing the geometry to impostors, then impostors to the screen. We define a key system parameter R as the pixel-area-averaged impostor reuse rate: that is, the number of frames a typical impostor pixel is displayed before being updated. If $R = 100$, the average impostor pixel will be displayed a hundred times before it is updated, and impostors can provide a significant performance improvement. If $R = 1$, the average impostor pixel is displayed exactly once, and hence impostors provide little benefit. This section attempts to calculate the impostor reuse rate R. To do this, we analyze how often the impostors must be updated, as this geometric update rate limits the impostor reuse rate, and hence the benefit provided by impostors.

Consider the situation shown in Figure 2.2. A camera, with view frustum shown at the left, images a plane of color samples, as shown at left. Because the camera only measures color, and cannot directly measure depth, we can replace the true scene geometry, such as the tree at right, with any transformed version that has the same colors when viewed from the camera. That is, for any camera view ray we can replace all the geometry along that ray with anything that produces the same overall color. We could choose to replace the world with one point sample along each ray, thus replacing the world with a cloud of colored points similar to the output of a laser range scanner device. But in practice it is much easier to project local

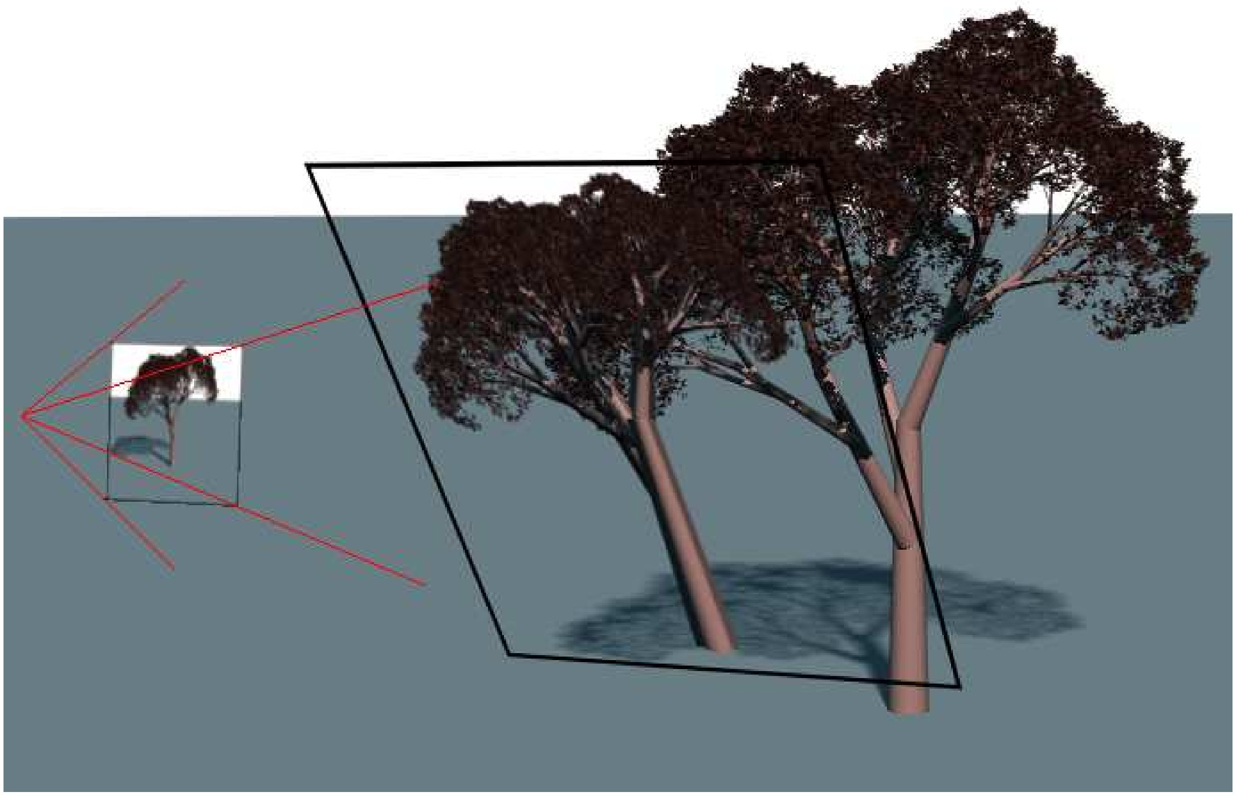


Figure 2.2. From the camera (left), an impostor (middle) projects to exactly the same image as the true geometry (right).

pieces of geometry onto a single surface, the impostor surface, whose shape and orientation we are free to choose. In this work, we choose to project the geometry for each object onto a plane; this allows us to use standard camera transforms to project the world geometry onto our impostors, as well as using standard texture-mapping to draw the impostors onscreen.

Consider first what happens if the camera rotates, without changing the center of projection. Because each view ray still starts at the eye and heads into the scene, our impostors still match the geometry exactly, for any rotation of the camera. That is, camera rotation never causes impostors to update; so if the camera only rotated, the impostor reuse rate R would be infinite.

We spend the remainder of the section analyzing parallax—simple translation of the camera. We analyze this simple case in detail to bound the geometric reprojection limits for our method. In our system, geometric change is the only reason impostors are ever updated. In general, impostors might need to be updated for other reasons such as deforming model geometry, changing specular reflections, or other view-dependent lighting effects; but parallax changes the appearance of almost any object. Also, we analyze (and use) simple planar impostors, though a similar analysis holds for non-planar "meshed impostors" [SDB97].

Consider a thin bar aligned with the z axis and centered at world coordinates $(0,z)$, as shown in Figure 2.3. Our perspective model is simply $s = kx/z$, which converts world coordinates (x,z) to screen pixels s. The eye and the center of projection are hence both at the origin, and k is the distance to the projection plane, or equivalently the number of pixels seen at a 45 degree field of view.

We begin by choosing the impostor plane z, shown by the dashed line, and project our object onto a texture lying in this plane. Initially, the projection is perfect—we can't tell the object has been replaced by the impostor, because the impostor is identical along each view ray, and can even be made pixel-for-pixel identical. We label the distance from the impostor plane to the most distant object point d, the maximum depth of the object beyond the impostor plane.

If the camera moves down the x axis, to $(-f,0)$, this is equivalent to moving the object to the new position $(f,0)$. If after moving the camera we re-use the old impostor, there will be a quantity Δx of projection error between the impostor and the true projected geometry. This error will normally be worst at the extrema of the object, so we examine the projected shift of the closest corner point.

Because $\triangle ABE$ and $\triangle CDE$ are similar triangles, the projected geometric error Δx of the new corner is:

$$\frac{\Delta x}{d} = \frac{f + \Delta x}{z}$$

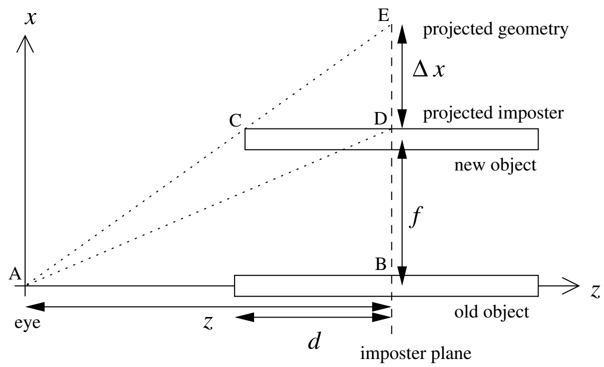


Figure 2.3. Parallax analysis for impostors.

$$\Delta x(\frac{1}{d} - \frac{1}{z}) = \frac{f}{z}$$
$$\Delta x(\frac{z-d}{zd}) = \frac{f}{z}$$
$$\Delta x = \frac{fd}{z-d}$$

Projecting into screen space, we find

$$\Delta s = k\frac{\Delta x}{z} = k\frac{fd}{z(z-d)}$$

If we fix a maximum screen-space error Δs, we can solve for the maximum allowable camera motion f:

$$f = \frac{z(z-d)\Delta s}{kd} \tag{2.3}$$

Taking the camera velocity as V meters per second and the framerate as H frames per second, the impostor is guaranteed to be reused for at least R frames:

$$R = \frac{fH}{V} = \frac{z(z-d)\Delta sH}{kdV} \tag{2.4}$$

That is, the minimum reuse rate is proportional to the screen error tolerance and framerate; and inversely proportional to the screen resolution, impostor depth, and camera velocity. For distant impostors with $z >> d$, the reuse rate is proportional to the square of the distance from the camera z. This squared proportionality means distant impostors can be reused an immense number of times.

Figure 2.4 shows isocontours of the maximum allowed camera motion f, for various impostor depths and distances. Figure 2.5 gives the numbers of frames R that an impostor can be reused before the screen

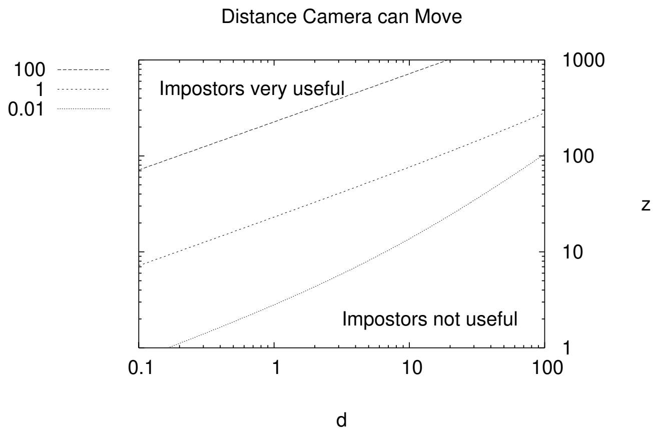


Figure 2.4. Isocontours of the bound, in meters, on the distance that the camera can safely move for an impostor to stay below one pixel of parallax error, for various impostor depths d (horizontal axis) and distances z (vertical axis), both in meters. Resolution is 1024x768 with a 90 degree horizontal field of view.

error exceeds one pixel. Distant or very flat impostors, plotted in the top and left of these plots, can tolerate enormous amounts of camera motion, and hence can be reused for a large number of frames. This means impostors work very well for distant or flat geometry. Very deep or very close impostors, on the bottom and right, can almost never be reused, as even a small camera motion causes a visible amount of parallax change. This means impostors provide little benefit when the geometry's depth range is on the order of the distance to the camera.

The impostors technique is thus particularly suited to navigating large virtual environments. Not only do most environments consist of many nearly flat objects, such as building walls and the ground, but the vast majority of the geometry in a large environment is far away.

2.3 Rendering and Network Bandwidth

The final fundamental limiting factor we consider is the bandwidth between each component of our system.

Render. In the proposed architecture, impostors begin by being rendered on the parallel machine. The rendering bandwidth B_R, in pixels per second per processor, depends heavily on the CPU speed and ren-

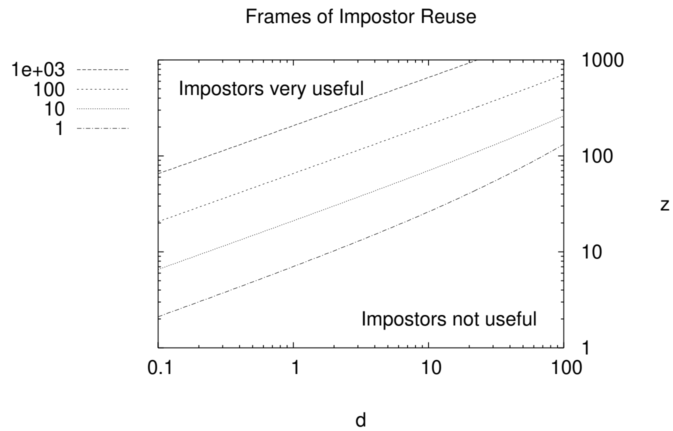


Figure 2.5. As above, but shows the number of frames each impostor can be re-used, in frames. Assumes the camera moves perpendicular to the view direction at V =20kmph and the framerate is H =60hz.

dering quality. Clearly, a high-quality antialiased radiosity-illuminated image will take much longer than a simple flat-shaded polygon rendering. A typical aggregate rendering rate for high-quality splats drawn in software might be one million finished pixels per second per processor, which is 4MB/s per processor. When multiple processors are used, the rendering rate is increased by the parallel speedup P.

Network. Rendered impostors are shipped to the client over a TCP-based connection called CCS, as described in Section B.1. CCS can saturate fast ethernet (100baseT), providing a network bandwidth B_N of 10MB/s. Gigabit ethernet should increase the data rate to 50-100MB/s. Shipping 32-bit pixels means the network compression rate C_N is 1/4 pixels/byte. Using run-length encoding should approximately double the compression rate, while a lossy compression such as JPEG or S3TC could increase it fourfold or more. Either encoding would increase the compression and decompression CPU cost, and have some impact on image quality.

Upload. Once in the client's main memory, the impostors must be uploaded onto the graphics card. Even a 4x AGP graphics card interface has a theoretical bandwidth of 1GB/s; but typical measured upload bandwidths B_U, including mipmapping time, range from 100-300MB/s (see Appendix E.2 for measurements). The upload compression rate C_U is 1/4 pixels/byte for 32-bit pixels.

Compositing. The final step is to composite the impostors on the graphics card into the framebuffer for display. The achievable fill rate B_C for alpha blended, textured and projected polygons is 80M-1200Mpix/s (250MB/s-5GB/s pixel bandwidth). The actual achieved fill rate depends on the triangle area, as shown in Equation 2.1.

The impostors method allows rendered, shipped, and uploaded frames to be reused on the graphics card several times. We define the area-averaged impostor reuse rate, bounded by Equation 2.4, as R reuses per pixel. This increases the effective bandwidth of everything up to the final compositing step by a factor of R.

Because some pixels are drawn several times, especially with overlapping impostors, overdraw restricts the overall system performance. We define the area-averaged rendered depth complexity as D pixels drawn per screen pixel.

Finally, because this is a pipeline, throughput is limited by the slowest component. The overall delivered screen bandwidth B in screen pixels per second is simply the minimum rate of rendering, network transmission, uploading, and compositing:

$$B = \frac{\min(B_R PR, B_N C_N R, B_U C_U R, B_C)}{D}$$

The lowest numerical bandwidth values are for rendering bandwidth B_R. However, the advantage of our architecture is that rendering bandwidth can be scaled up by adding processors until rendering is no longer the bottleneck. In practice, load imbalance and other parallel efficiency losses mean the parallel speedup P may be substantially lower than the number of processors; Section 4 describes methods for improving the parallel efficiency.

Because the graphics card bandwidth B_C is so much larger than the other bandwidths, in order to take advantage of the graphics card's fill rate we clearly must reuse impostors a significant amount. To prevent an uncompressed ethernet network with $B_N C_N$ of 2.5Mpix/s from limiting the overall performance, we would have to reuse each shipped impostor an average of $R = 25$-150 times, which according to Figure 2.5 is only acceptable for very flat or very distant impostors. Clearly, texture compression (high C_N), gigabit ethernet (high B_N), or some degree of error tolerance are necessary for most scenes.

As an example illustrating how the proposed system components can be balanced, consider the following scenario. First, choose rendering algorithms so that geometry is rendered with a bandwidth of $B_R = 1$ Mpix/s/cpu, with a parallel speedup of $P = 32$, for an aggregate rendering bandwidth of 32 Mpix/s. Ship the resulting impostors over the network using run-length encoding with $C_N = 0.5$ pixels/byte and gigabit

Method	d	a	Limiting factor
Screen Shipping	Large	Large	Network (low R: no reuse)
Point-Based	Small	Small	Triangle Rate (low a: tiny triangles)
Parallel-Plane	Small	Large	Fill Rate (high D: overdraw)

Table 2.1. Various extreme cases for the impostors method, for various impostor depth ranges d and areas a, and their limitations.

ethernet with $B_N = 60$MB/s, to provide a network data rate of 30Mpix/s. The impostors would then be reused on the graphics card approximately $R = 10$ times, which exactly matches a fill rate bandwidth of $B_C = 300$ Mpix/s. Assuming a depth complexity of $D = 2$, the overall delivered pixel rate would then be $B = 150$Mpix/s, enough for a 75Hz framerate at 1600x1200 resolution. Such a system would fully utilize nearly all of its components, for excellent performance.

2.4 System Design Extrema

It is useful to examine the limiting cases of this parallel rendering system, as summarized in Table 2.1.

Screen Shipping. One could agglomerate all the geometry of the scene into a single large impostor, then ship this one screen-filling impostor to the client. This is the screen shipping idea used by many parallel rendering systems, including nearly all parallel raytracers [Sto98] and other systems such as Chromium [HHN+02]. However, because the depth range for the scene is huge, Equation 2.3 shows the allowable amount of camera motion is tiny—that is, the client can never reuse the screen image unless the camera is absolutely stationary. With a reuse rate of $R = 1$, the rendering and network bandwidth thus required for even a moderate resolution and framerate using this technique is enormous—just shipping a 1024x768 screen at 30HZ in 32-bit color would require 95MB/s of data, which would saturate even gigabit ethernet. That is, screen shipping is limited by the network data shipping rate.

Point-based Rendering. To avoid the frequent updates of large impostors, we could instead decompose all the geometry of the scene into very tiny impostors. As the impostor depth range d drops to zero, Equation 2.4 shows that the number of frames of reuse R goes to infinity. This means our very tiny impostors could be reused indefinitely; after receiving the initial set of impostors, the client would never need anything more from the server. This is essentially point-based rendering, which has the well known problem that the per-triangle cost of modern graphics cards causes low performance when drawing very small polygons, as shown in Section 2.1. That is, point-based rendering is limited by the graphics card's triangle rate.

Parallel-plane Rendering. To avoid the display overhead of very small polygons, we could decompose the geometry into a series of impostors representing very thin slices along the Z axis, such as the Layered Impostors technique [Sch98]. Because the depth range of the impostors is small, the impostors would rarely need to be updated. Because the screen area of the impostors is large, the impostors make good use of the graphics card's fill rate. However, because all the impostors overlap, there is an incredible amount of overdraw D, so the delivered screen performance is low.

Efficient impostor-based system designs are not found at these extrema, which stress only one component of the system. Instead, an efficient system will use a balanced approach, intended to utilize all components of the system equally. This means using impostors to represent small, relatively flat portions of the scene geometry, and changing the impostor decomposition based on the viewpoint to keep the screen-space size of each impostor reasonable.

Chapter 3

Impostors

An image *impostor* is a 2D stand-in for real 3D geometry. The technique itself predates even computer graphics.

The painting style *trompe l'oeil* (to fool the eye) is the technique of using 2D shading to create the appearance of a 3D object, as shown in Figure 3.1. Examples of this technique date back to Greek and Roman times.

In theater, *backdrops* are huge painted pieces of fabric hung behind the stage. Backdrops are used to simulate large spaces (for example, outdoor scenes) or complicated sets (for example, an ornate building interior) while staying within space, cost, and construction time constraints. Matte paintings have served the same purpose in films for over a hundred years.

Trompe l'oeil, backdrops, and matte paintings all share the disadvantage that the depicted scene does not change when the viewpoint changes—that is, these paintings display no parallax. This makes them most effective from far away, where parallax is less noticeable. In addition, in the theater, viewers do not move; while in films, the viewpoint motion is carefully controlled.

Parallax is still an important consideration for impostors in computer graphics. The main techniques we use for achieving parallax include using multiple overlapping impostors at different depths, and adaptively re-rendering the impostors as the viewpoint changes.

3.0.1 Impostors in Computer Graphics

Environment maps [BN76] or skyboxes are a backdrop-style technique used in computer graphics. These precomputed background images normally map a view direction to a color. Like physical backdrops, they

Figure 3.1. A painting in the *trompe l'oeil* style by William Michael Harnett, 1848-1892.

display no parallax; the image does not depend on the viewer location. Many virtual environments use some variation of this technique to display "far away" geometry like the sky and distant mountains.

Billboards are precomputed, alpha-blended 2D textures that always drawn facing the viewer. For example, a classic computer graphics method for rendering trees is to precompute one large billboard with a tree image, then face the tree billboard towards the viewer.

Sprites, like billboards, are precomputed and always drawn facing the viewer, but there can be a separate image for a small set of different viewpoints. For example, the monsters in Id Software's 1993 "Doom" are sprites with up to eight viewpoints distributed around a horizontal circle, along with several animation frames. The distinction between a sprite and billboard is fuzzy; sometimes "sprite" merely means a billboard rendered with less sophisticated antialiasing or alpha blending.

The term *impostor* originates in a paper by Maciel and Shirley [MS95], which defines an impostor as anything that replaces actual geometry. They statically precompute texture-mapped polygon impostors for fixed pieces of geometry from a fixed set of viewpoints aligned with the coordinate axes.

Shade et al. [SLS$^+$96] build impostor images at runtime from a hierarchical scene graph. Shade allows large subtrees of the scene graph to be replaced by impostors, and shows a system that scales to very large databases. Like Shade, we also build impostors on the fly; the main difference is we render the impostors on a parallel machine, and with high quality.

Schaufler et al. [SS96] also describe a dynamically generated impostors system, and has detailed performance analysis regarding the impostor update rate and the use of out-of-date impostors. Schaufler's work also does not consider parallel or higher-quality rendering.

Aliaga [Ali98] describes a fully automated model decomposition and rendering system based on impostors, and presents a technique for warping the surrounding geometry to match an out-of-date impostor texture. His fundamental model representation is the "cells and portals" scheme, and he always places impostor planes along portals.

The impostors described above do not include per-pixel depth information—these impostors are 2D planar quadrilaterals located in 3D space. Chen and Williams [CW93] gave a good early account of the possibilities for and problems with fully 3D image-based rendering, including the possibilities of warping and resampling images based on per-pixel depth. A single impostor texture can survive dramatic viewpoint change if the texture is warped according to depth, but this requires an expensive per-pixel texture resampling and must deal with the holes caused by occluded regions in the original texture becoming visible. In our implementation, other than regular 3D planar projection we perform no impostor warping—instead, as the viewpoint shifts, the impostor textures are regenerated. Nothing prevents the parallel impostors technique from taking advantage of these more sophisticated impostors, however.

Sillion et al. [SDB97] describe an impostor-based rendering system that overlays the impostor image on a coarse mesh representing the geometry. This approach has a higher impostor reuse rate than simple flat impostors.

Schaufler [Sch98] describe Layered Impostors, a multi-pass rendering system which stores a per-pixel depth along with the impostor texture. By storing the depth in the alpha buffer, the geometry at a single layer of the impostor can be extracted using the alpha test. Schaufler then renders the all the impostor's layers from back to front. This approach can survive dramatic camera motion, but is limited by the graphics card's fill rate, because there is a huge amount of overdraw caused by the many layers.

Shade et al. [SGHS98] describe Layered Depth Images, an impostor-like image-based technique that includes several depths at each pixel. Shade gives a careful analysis of methods for resampling images with depth, including hole filling.

Decoret et al. [DSSD99] describe multi-meshed impostors, a system for constructing a set of meshed impostors that accurately capture parallax for complex urban geometry. They also include a discussion of impostor update prioritization, but in the end prioritize simply based on screen-space error.

Torborg et al. describe the Talisman [TK96] system, a tile-based hardware 2D image compositing system that approximates 3D warps with local 2D affine tile transforms. Torborg explicitly makes the argument that conventional Z-buffer rendering can only provide transparency, antialiasing, and anisotropic depth filtering at enormous cost; but image-based systems can provide these features cheaply. We use alpha-blended impostors to achieve many of the benefits of the Talisman system on conventional graphics hardware.

Mark Harris' cloud rendering work [Har02] caches the expensive rendering of true 3D clouds with simpler 2D impostors to achieve excellent rendering performance. Like our work, he amortizes out the cost of an expensive, accurate rendering by reusing impostor images.

3.1 Impostors and the Z Buffer

The biggest difference between the simple impostor technique and the usual graphics pipeline is the lack of per-pixel impostor depth information. There are a number of compelling reasons to omit depth information.

First, Z-buffer rendering is incompatible with transparency, and to allow antialiasing, the impostors we use are partially transparent at the object boundaries. Second, sending 24 bits of depth with each impostor pixel would double the uncompressed size of the impostor, which could halve the rate at which impostors can be sent across the network. Third, when performing splat-based particle rendering, a unique depth value per pixel may be difficult to compute or even define. Finally, graphics card programmable shaders only recently gained the ability to adjust a pixel fragment's depth value, and many cards still lack this ability, so an impostor depth value may not even be usable on the client.

All these disadvantages of the Z buffer lead us to instead primarily rely on the well-known per-object painter's algorithm. We traverse the scene's impostors in back-to-front order, alpha-compositing new impostors in depth order. Because nearer impostors are drawn later, they will properly occlude more distant impostors. Relying on the painter's algorithm then allows us to avoid the expense of computing, sending, and compositing per-pixel depths; and allows us to use transparency and antialiasing to improve the appearance of our impostors.

The method we use to traverse our scene database in approximate Z order is described in Section 9.2. Because an exact painter's algorithm is difficult to implement efficiently for arbitrary geometry, we do use the Z buffer on the client as a final defense against occlusion errors.

3.2 Impostor Plane Choice

As noted in Section 2.2, we are free to choose the plane that we project geometry onto when forming our impostors. For a fixed camera position, there is no reason to prefer one plane over another. But when the camera moves, our choice of impostor plane can significantly affect the resulting impostor update rate. For the smallest possible update rate, we should choose the plane that minimizes the impostor depth; the maximum distance of any part of the the object to the impostor plane. But we can also attempt to match parallax, so that if impostors are not updated, the resulting scene, though technically incorrect, is still plausible.

For example, Figure 3.2 shows our impostor-based rendering system drawing four trees. There are actually five impostors here: one per tree, and one large impostor covering the ground. We choose the impostor planes for each tree to pass through the base of the tree trunk. This ensures the tree's base always meets its shadow exactly. The impostor plane for the ground is always aligned with the major directions of the ground.

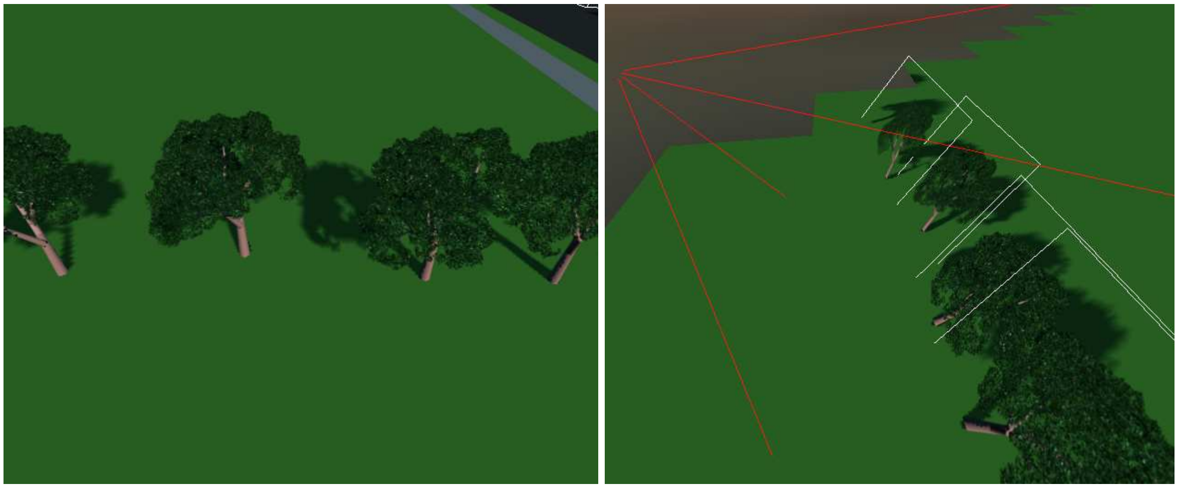


Figure 3.2. On the left, the view through a camera of a small set of geometry. On the right, the same scene viewed from a new viewpoint, showing that the "geometry" actually consists of a set of planar impostors, one for each tree. From the camera's point of view, the impostors are indistinguishable from the geometry.

3.3 Impostors Pitfalls and Solutions

There are situations where the impostors technique provides little benefit.

Low reuse. If the cached images have a low probability of reuse (low view coherence), it may be faster to skip the cache and simply draw the object onscreen directly. We mitigate this disadvantage by using a hybrid rendering algorithm that normally draws impostors, but switches to the usual polygon-based rendering for objects that show low view coherence, such as very nearby objects.

Overhead for simple objects. For simple models, the cost of drawing the object and the cost of drawing an impostor of the object can be nearly equal, so the image cache provides little benefit. However, larger models are always desired, so in practice an image cache should often help.

Changing objects. If the model changes dramatically, the cache must be invalidated. For largely static scenes, like natural environments or cities, this is rare; but for time-varying scientific datasets it is the norm. Frequently-changing geometry may have to bypass the impostor rendering system and be drawn directly on the client.

3.4 Proposed Architecture

Though impostors form the theoretical and practical basis of our technique, the basic technology is well understood, so only a small amount of work is needed to improve the performance and appearance of our impostors.

3.4.1 Impostor Update

The simple parallax-based impostor update equations of Section 2.2 cannot be used for arbitrary combinations of camera and object motion. Instead, we determine if an update is required by examining a set of "key points", for which we explicitly compute and compare the screen projection of the actual 3D location with the screen projection of the 2D impostor data. By default, the eight corners of the bounding box are used as key points. Impostors may also need to be updated because of intrinsic change, such as object deformation.

3.4.2 Spatial Discontinuity Hiding

The boundaries between impostors can cause visual artifacts like seams, holes, and other artificial discontinuities. These boundaries can be handled by blending, which replaces discontinuities with blurring; by keeping objects up-to-date to sub-pixel accuracy so the discontinuities are never visible; or by carefully shifting adjacent objects to parallax-match along each seam, like Aliaga [Ali98] . Because the eye is insensitive to large, low-frequency geometric errors, a carefully chosen geometry decomposition may be able to tolerate a great deal of error.

3.4.3 Temporal Discontinuity Hiding

When a new impostor image arrives, using it onscreen immediately can lead to a jarring visual "pop" as the new image replaces the old, because human vision is quite sensitive to sudden temporal change. We implement the classic solution to this problem of softening the transition using a time-dependent crossfade. A more advanced technique would be to delay the update until some occluder crosses in front of the object, as the visual system is quite insensitive to this kind of obscured change.

3.4.4 Impostor Geometry Decomposition

The hardware performance model Equation 2.1 shows that the graphics card's delivered fill rate drops substantially for small triangles. But the impostor parallax update rate Equation 2.4 shows that the reuse rate is low for nearby large, thick impostors. For high performance, we must stay between these two extremes—distant geometry should be aggregated into large impostors to improve the fill rate, while nearby geometry should be decomposed into small impostors to improve the impostor reuse rate.

In our system, the geometry for each impostor is represented by exactly one parallel array element. Thus as the camera moves and we change the set of impostors, we must change the set of parallel objects that represent those impostors. While a conventional parallel system such as MPI provides little support for this, the Charm++ Array Manager [LK03] directly supports the creation and deletion of parallel objects. We will hence create and destroy array elements as we coalesce and split the geometry they represent. Our method for actually splitting and merging the geometry is described in Section 9.5.

Chapter 4

Parallel Rendering

The method we propose performs impostor rendering on a parallel machine. This section summarizes the extensive prior work on parallel rendering, and describes the parallel rendering architecture we use.

The central problem not solved by most existing parallel rendering infrastructures regards the relatively low bandwidth between the client and server. Although network links capable of gigabytes per second exist, these high-performance networks tend to only work over short distances, be very expensive, or both. The vast majority of scientific users at major research centers have connections that deliver only 10-100MB/s.

This means we cannot simply render everything on the server, and ship the resulting image to the client, because the network bandwidth to the client is not sufficient for this. We alleviate the network bandwidth problem by reusing impostor images on the client, but impostors introduce dynamic and adaptive behavior that is difficult to handle with existing parallel programming interfaces, such as MPI [MPI94].

Existing programming interfaces such as MPI are well-suited to the screen-shipping parallel rendering method. With screen shipping, the decomposition of geometry is normally simple and static, and the start and end of each frame provides a clear, well-defined place to synchronize work between processors.

Impostors do not fit existing interfaces so well. For example, we often want to divide the model into thousands of impostors—far more than the number of processors. We want to assign the geometry for these impostors to processors based on load balance, but unlike the standard "tile pool" solution to screen-shipping load imbalance problems, we want impostors to move between processors infrequently. Our migratable object interface, described in Section 5.1, fits this need exactly.

When the client viewpoint changes, we want to be able to broadcast the new viewpoint to all the processors. But new viewpoints are not a synchronizing call, because rendering continues before and after the new viewpoint. Our efficient support for collective operations, even in the presence of migrations, fits this need well.

4.1 Prior Work: Parallel Rendering

There is an immense amount of prior work in essentially offline parallel rendering, including parallel raytracers [Sto98]; shear-warp and raycasting volume rendering; and diffuse, specular, and large-model radiosity. Our work is different in that we attempt to provide full-framerate interactive speed even over slow connections, not just a high-quality rendering.

Molnar et al. [MCEF94] provide a good taxonomy for possible methods to parallelize the usual feed-forward graphics pipeline exemplified by OpenGL. The main differentiating factor is the stage at which graphics data is sent across processors. *Sort-first*, or screen-space subdivision, means the geometric primitives are sent across processors before rasterization, and a rasterizer is only responsible for a small piece of the final output image. *Sort-last*, or object-space subdivision means primitives are rendered locally, and the rasterized products are then assembled across the network and composited.

UNC's 1990's PixelFlow machine [MEP92] is a characteristic sort-last architecture. PixelFlow consisted of an array of custom SIMD chips, each of which renders a set of primitives into a local framebuffer. The framebuffer outputs are then composited together via a high-performance hardware compositing network.

Recent work on the Chromium [HHN$^+$02] architecture provides a flexible, high-performance parallel rendering system for clusters. Chromium provides a call-compatible OpenGL replacement library and an assortment of backend processing implementations. In the typical sort-first usage, each processor of a parallel application uses the library to describe geometry, which is hashed into screen-space buckets. The geometry for each region of the screen is sent off and rendered on a separate rendering pipeline, which typically uses a real OpenGL implementation on a set of rendering nodes. The open-source implementation supports a number of other possible configurations, including sort-last. Our parallel impostors technique is clearly a sort-last technique, because objects are rendered into impostors independently before being sent across the network.

We could in theory similarly implement our parallel impostors technique using ordinary OpenGL calls to describe the geometry. However, our library would then have to reconstruct object boundaries from the stream of geometry data in order to construct impostors. Producing a good quality partitioning of an arbitrary set of geometry into impostors would be quite challenging, and nearly impossible in real time. In addition, the application would still have to traverse all the visible geometry of a scene, including geometry for impostors which do not need to be re-rendered.

Ward et al. describe the Holodeck Ray Cache [WS99], a lightfield filled at runtime with rays rendered by a parallel machine. Ward uses a master-slave ray distribution approach, and assembles the rays at the client. It is difficult to reconstruct an artifact-free image from an arbitrary set of rays, so achieving good image quality with this approach is challenging.

Mark [Mar99] showed a parallel rendering system where the parallel server provided a low-framerate stream of images plus depth, and the client used an image-based warping technique to interpolate a higher-framerate display. Because the client does not have true geometry information, this approach must somehow deal with holes in the generated images.

Wald et al. describe a parallel raytracer [WSB01] capable of displaying 50 million polygon models at interactive rates. This is a classic sort-last architecture based on screen shipping. They get good results using a screen-tile-based central work queue for parallel load balancing. Like any screen-shipping approach, their performance is limited by the network bandwidth to the display client. We compare the performance of impostors with screen shipping in detail in Section 6.2.

LaMar and Pascucci [LP03] present a parallel impostor rendering system for large transparent volume datasets. They based the scene decomposition on a k-D tree, and presented reasonable performance results. Unlike our work, their implementation was limited to shared-memory machines, largely because of their single global work queue. Global work queues do not scale well on the more common distributed-memory machines, while our prioritized, explicit load balancing system runs on distributed memory machines and does scale well [KRSS93].

4.2 Proposed Client/Server Architecture

The client, a regular OpenGL program, connects to the parallel server using our TCP/IP-based protocol CCS, as described in Section B.1. The server is a parallel machine. As the client's viewpoint changes, normally every frame, the client sends the latest viewpoint to the parallel server.

The parallel server receives viewpoint updates, renders new impostors if necessary, and sends impostor images back to the client. On receiving a new viewpoint, the server broadcasts the viewpoint to the parallel

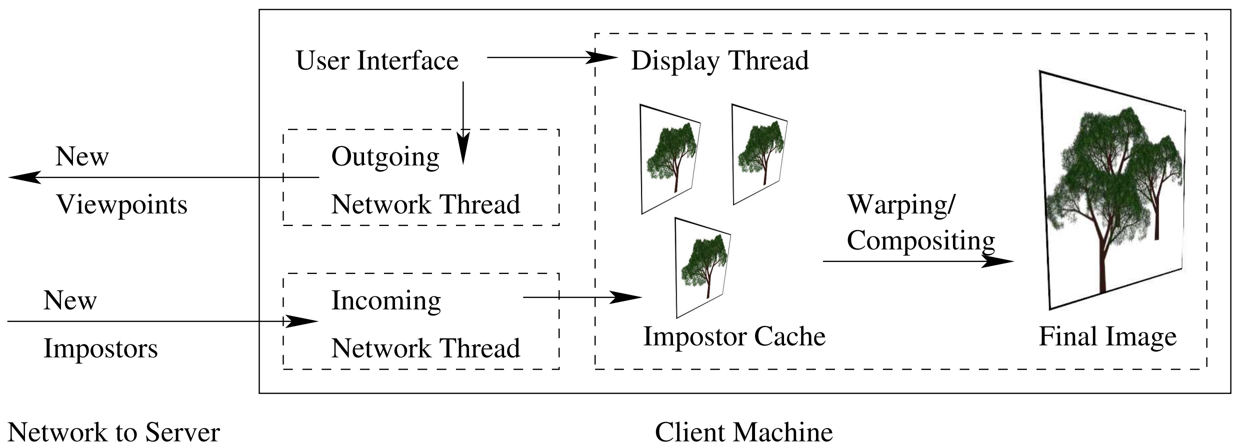


Figure 4.1. Architecture of client, showing threads that handle various tasks.

objects that represent each piece of geometry. If the geometry is offscreen, or if the previously rendered impostor for that geometry is still within the screen error tolerance, the object does nothing. If the old impostor is no longer adequate, the geometry object puts itself in the render queue. Because the actual rendering is separate from the rendering decision, we can prioritize the rendering process. As rendering generates new impostor texture images, the textures are batched up and sent off to the client.

The client receives updated impostor images from the network, unpacks them, and inserts them into the impostor cache. For display, the client traverses the scene, making OpenGL calls to render the cached impostors in back-to-front order. Of course, some impostors will not be visible in some frames, and will be culled during the traversal.

When the viewpoint changes, an impostor texture will only be updated once the request has been sent to the parallel machine, the parallel machine has rendered a new impostor, and the impostor has been shipped back to the client. Delaying the actual client display for this entire cycle would cause dropped frames whenever any one of these steps is delayed (due to, for example, network interference). Thus in our system the display is handled by a separate client thread, which is decoupled from the network impostor updates. The display thread thus always renders the currently available impostors, regardless of whether a better impostor is on the way. Network input and output are both handled by separate threads on the client, as shown in Figure 4.1. This loosely synchronized architecture performs well even in the presence of significant network performance variation.

Chapter 5

Parallel Infrastructure

We build our parallel rendering system on top of our existing robust, high-performance general purpose parallel infrastructure Charm++ [KK96]. In particular, the key pieces our impostor rendering system requires are our existing parallel work migration layer, the Charm++ Array Manager; our client/server interface CCS; and our C++ object introspection system, the PUP framework.

5.1 Charm++ Array Manager

The Charm++ Array Manager allows parallel objects called *array elements* to be created, destroyed, moved between processors of the parallel machine, send and receive messages, and participate in broadcasts and reductions. Array elements and the array manager are described in more detail in Appendix A and our published work [LK03].

Each piece of geometry to be rendered as an impostor is represented by one array element, a parallel object. Many of the operations we require are naturally supported by the array manager, and are listed below.

- All geometry, of various types, is kept in a single array whose type is an abstract geometry base class. An actual piece of geometry, such as a tree, will be of some concrete subclass, but in a single application there are often several different types of geometry, so the single array will actually contain several different concrete subclasses for different elements. Deriving all impostors from a single base class allows generic operations, such as viewpoint distribution, load balancing, and startup and shutdown, to use C++ virtual methods to uniformly access all geometry. The array manager directly supports this C++ idiom of an array of pointers to subclasses.

- Each time the viewpoint changes, the client informs the server, which then decides if any of the existing impostors need to be rendered. The server implements this decision by broadcasting the new viewpoint to all existing impostors, which make their decision based on their geometric reprojection error and their new projected screen resolution. The Charm++ array manager performs this broadcast efficiently, by broadcasting to all processors, and then delivering to the local array elements. Our "broadcast count" technique [LK03] allows this efficient broadcast to work properly even if elements migrate during the broadcast.

- As the viewpoint changes, new array elements are inserted into the array to represent newly-visible geometry, or to better represent nearby geometry. The array manager directly and efficiently supports these element insertions, and begins delivering messages and broadcasts to the new elements as soon as they are created.

- The automatic Charm++ load balancer monitors the load on each processor [Bru00], and uses the Charm++ array manager to migrate work between processors to improve load balance. Performance results from this migration-based load balancing are shown in Section 6.2.1. The Charm++ array manager allows these migrations to occur even when communication and collective operations are ongoing.

Overall, the Charm++ array manager provides a variety of features that make the implementation of an impostor-based parallel rendering system simpler and more efficient. But the Charm++ array manager is by no means impostor-specific, and forms a key component of dozens of other internal and external projects.

5.2 Converse Client/Server

All communication between client and server is performed using our CCS protocol, a TCP/IP-based request/response protocol similar to HTTP. The CCS protocol is described in more detail in Appendix B.1 and our published work [JLK04].

CCS is able to saturate a fast ethernet connection, works behind firewalls, and includes support for clients written in a variety of languages, including C and Java. CCS has proven to be a useful and flexible tool for solving a variety of problems.

5.3 PUP Framework

We use our PUP framework, a general-purpose method for manipulating the contents of C++ objects, to perform a variety of necessary housekeeping tasks on our impostor objects. The design and implementation of the PUP framework is described in detail in Appendix B.2, our published work [JLK04], and the Charm++ programming manual [Par04a].

In this work, the PUP framework is used to perform all of the tasks listed below for all possible impostor objects. Normally, each of these tasks would require a special piece of code for each kind of impostor or geometry. But the PUP framework allows us to completely decouple the task, which specifies what to do with the data, from the particular C++ object, which defines where the data is stored and what the data represents.

- In any client/server application, we must properly format requests and responses sent down the wire. We define all client/server communication in terms of the exchange of objects, which are serialized and deserialized to a standard network byte stream using the PUP framework's builtin "PUP_toNetwork4" and "PUP_fromNetwork4" tools.

- At the start of a run, the geometric representation of each impostor must normally be read from a disk file. During model construction, the geometry must be created and then written to those files. The campus dataset, described in Section 9, uses the PUP framework to perform disk I/O using "PUP_toDisk" and "PUP_fromDisk". This use of the PUP framework is a lightweight version of the object persistence schemes found in object-oriented databases [Dit91].

- For load balance, we migrate array elements from one processor to another. On a distributed-memory machine, this requires us to collect all the geometry represented by the array element into a network message, send the geometry across the parallel machine's network, and unpack the geometry on the other side. In this work we use the PUP framework, via the array manager, to perform the actual data movement for each object migration.

The PUP framework is also used in Charm++ for parameter marshaling, the initial distribution of read-only variables, datastructure inspection and modification for debugging, checkpoint and restart of parallel jobs, and several other tasks.

Chapter 6

Parallel Impostor Performance

In this chapter, we measure the performance of parallel impostors. For test problems, we use both a simple, straightforward interactive particle rendering system as well as the campus model described in later sections.

The simple problem we examine in this section is to render a large set of particles, as shown in Figure 6.1. For up to several million particles, this problem can be solved at interactive rates with simple, straightforward serial rendering. However, for tens or hundreds of millions of particles, the amount of memory required to represent the particles approaches multiple gigabytes, and a serial machine is no longer capable of even efficiently representing the problem data.

This is a model of problems actually encountered in computational cosmology [MQWS02], where the particles represent portions of matter in the early solar system or universe. A typical large-scale cosmology simulation dump might consist of 100 million particles for each of 100 different time snapshots. Scientists would like to explore this data volume interactively, to better understand the phenomena the simulation represents. The aggregate memory and I/O rate requirements for performing this sort of visualization in real time can only be satisfied via a fairly high degree of parallelism (e.g., using 32 disks of a cluster simultaneously); so making the rendering parallel as well is a sensible choice.

This section analyzes the parallel performance of the impostors scheme. We must then begin by deciding how to even evaluate the performance of such a parallel system.

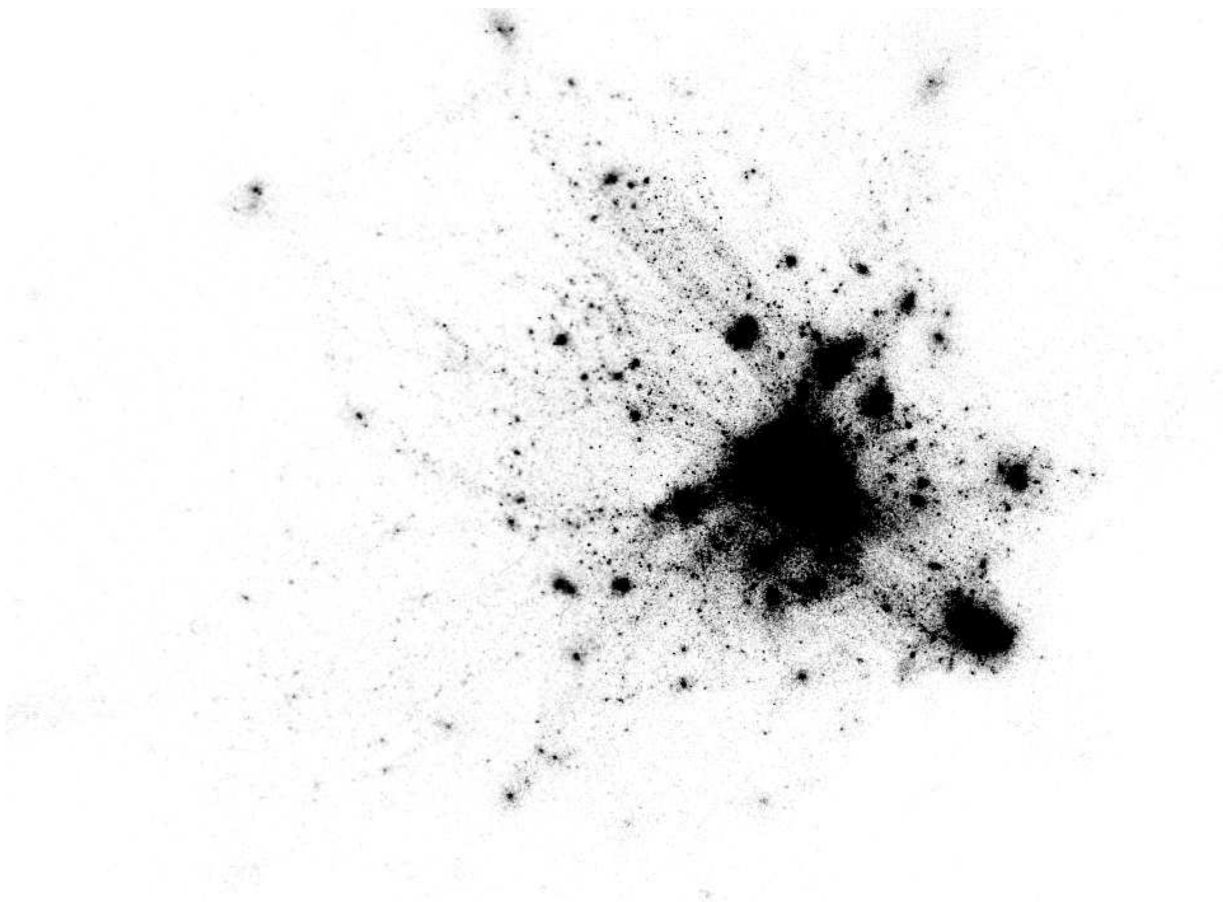


Figure 6.1. Large particle dataset encountered in cosmology.

6.1 Performance Evaluation

We analyze several aspects of the performance of our impostor-based rendering system. To perform a quantitative comparison, we must first define some quality metric with which to measure and compare delivered system performance; then we must build a system to measure this metric for various systems.

6.1.1 Quality Metric

The stereotypical graphics performance metric is simply delivered frames per second at a specified image resolution. But this is not a useful metric for an impostor-based system, because if we ignore the quality of the resulting images, an impostor-based rendering system can be tuned to deliver any desired framerate (with potentially large amounts of parallax error or distortion). Hence, for example, a low framerate yet low-distortion impostor system may be preferable to a very high framerate but high-distortion system.

Hence we must somehow consider both the temporal and spatial quality of the image stream we generate. There exist almost arbitrarily complicated perceptually based spatio-temporal moving image quality metrics, such as the "Color Moving Pictures Quality Metric" [vdBLV96], which compares image streams based on a perceptual color space passed through a biologically-motivated filter bank of oriented spatial filters.

But because the error induced by impostors is typically simple translation error, we instead use the simpler quality metric of time-integrated per-pixel squared differences. Formally, to compare a generated stream of images $g(x,y,t)$ and a reference image stream $r(x,y,t)$, we define the time-dependent image error $e(t)$ and total error e as the integrals:

$$e(t) = \sqrt{\frac{\int_{x,y} \|g(x,y,t) - r(x,y,t)\|^2 \, dxdy}{\int_{x,y} 0.1 \, dxdy}} \tag{6.1}$$

$$e = \frac{\int_t e(t)\, dt}{\int_t 1 \, dt}$$

The scaling factor of 0.1 is of course arbitrary. Note that $e = 0$ if and only if $g == r$.

6.1.2 Quality Evaluation Implementation

Interactive graphics programs produce their output on the hardware framebuffer of the graphics card. Hence to evaluate the performance of our program, we must simply compare the framebuffer with known-good values. Unfortunately, as measured in detail in Appendix E.2, modern graphics cards have relatively slow framebuffer readback performance—on the order of a hundred million pixels per second for fast cards. Hence the obvious technique of simply reading back the framebuffer to the host machine at every frame would consume the vast majority of both CPU and graphics card time. Further, read-back frames are generated at a significant fraction of a gigabyte per second, so even if the generated frames could be read back, they cannot be stored in memory except for very short periods.

Unfortunately, dramatically filling up and slowing down the client machine with framebuffer readback would severely perturb the running program, because the network and parallel machine would continue to operate at full speed. Without performing the analysis on a completely different machine from the client (e.g., by capturing and comparing the video output from the DVI port), we cannot entirely eliminate the Heisenberg-like impact of our measurement on the system.

However, several techniques can be used to minimize the measurement overhead:

- Perform readback to a texture, shrink the texture, and readback the smaller version. This improves performance because readback to texture is typically around tenfold faster, per pixel, than readback to host; but this does trade spatial resolution for speed.

- Perform readback on less than the full color resolution, such as using only 16 bits per pixel. This reduces the color precision and does not normally speed up readback, but reduces the data rate in subsequent processing.

- Perform readback only on a subset of frames. This is essentially a Monte Carlo evaluation of the integral 6.1, so frames should be randomly selected to avoid "sampling resonance" and hence systematic measurement error. This technique trades temporal sampling accuracy for speed.

- Immediately stream the read-back frames to disk for later analysis. Disk speeds of 20-50MB/s are typical; and disk writes involve very little overhead. This technique allows analysis of thousands of frames without significant memory overhead.

Consider the use of these techniques for a full-size 1600x1200 full-framerate 60Hz image generation system. Because 1600x1200 is 1.92MPix/frame, 60 frames/second is a pixel rate of about 110 MPix/second, or a data rate of nearly half a gigabyte per second. This is greater than the framebuffer readback rate of most graphics cards, and certainly exceeds achievable disk bandwidth, so we readback to a texture, and downsample to 800x600, and readback at 16-bit per pixel color precision. At 60Hz, the data rate is still over

57 MB/s, which exceeds most disks' performance, so we stochastically drop every other frame (on average) for a data rate of 28 MB/s, which most workstation disks can handle.

6.1.3 Quality Evaluation Infrastructure

We have implemented the quality metric and image-stream recording techniques described above as a library. To perform an evaluation:

1. During an interactive run, we record a 100Hz series of real viewpoints. These are stored to a viewpoint stream file.

2. Offline, we generate a series of perfect "reference" frames by rendering from each stored viewpoint. These frames are stored in a 100Hz reference image stream file.

3. In realtime, we feed the new viewpoints to the parallel machine and collect and display impostors. The resulting image frames are written to a generated image stream file.

4. Offline, we read and compare the reference and generated image streams using the quality metric described in Section 6.1.1.

All four of these phases, and ordinary interactive use, are supported inside the same client program. The evaluation infrastructure is handed control only at the start and end of each frame; nothing else is changed for each run.

6.2 Simple Parallel Performance Results

We present several parallel performance results for the simple particle rendering system.

These results are for the test problem of a 50 million particle astrophysics point dataset. Each particle occupies 20 bytes for a mass, position, and index, so the entire dataset is about a gigabyte in size. Particles are distributed into an space-filling octree, with lists of particles stored at the leaves of the octree. We render the particles using our high-quality software-based splatting system, as described in detail in Section 7.2.1.1. In these examples, the dataset is rendered on a large cluster of dual-processor 550MHz Pentium III Xeon nodes connected with 100mbit Ethernet.

Table 6.1 shows the most glaring benefit of parallelism—although these slow processors can only render about 2 million particles per second, 48 processors can render over 80 million particles per second. For this experiment, we simulated an infinitely fast network by disabling impostor shipping—impostors are rendered and then immediately discarded.

Processors	4	8	16	24	32	48
MParticles/second	7.14	15.71	32.71	49.18	65.49	81.68

Table 6.1. Real parallel rendering performance for test problem; assuming an infinitely fast impostor shipping network.

Processors	4	8	16	24	32	48
Framerate	61.864 fps	68.742 fps	65.628 fps	62.731 fps	63.993 fps	64.828 fps
Error	0.182655	0.139420	0.127121	0.127309	0.125537	0.135257

Table 6.2. Impostor-based system parallel rendering performance for model problem and real image shipping network.

Processors	4	8	16	24	32	48
Framerate	0.170 fps	0.285 fps	0.458 fps	0.543 fps	0.589 fps	0.681 fps
Error	0.568708	0.482714	0.420433	0.400707	0.388897	0.371073

Table 6.3. Screen shipping parallel performance for the same problem and same machine.

Table 6.2 shows the overall performance of the rendering system using impostors. Our impostors system indeed delivers a steady 60Hz display, and adding processors improves image quality. Table 6.3 shows the same system using screen shipping. For this system, screen shipping results in an unusably low framerate. Despite the fact that it assembles perfect, error-free individual images, because of the low framerate screen shipping also has very high time-averaged error. Screen shipping also fails to scale, because no matter how quickly it can assemble images, it cannot ship screenfuls to the client quickly enough. The performance of our parallel impostors system is also limited by the network bandwidth to the client, but impostor reuse decreases the impact of the client network on overall performance.

Processors	4	8	16	24	32	48
No-reuse Impostors framerate	0.196 fps	0.373 fps	0.372 fps	0.377 fps	0.391 fps	0.399 fps
Screen-Ship framerate	0.170 fps	0.285 fps	0.458 fps	0.543 fps	0.589 fps	0.681 fps
No-reuse Impostors Mbytes/frame	4.90	4.90	4.90	4.90	4.90	4.90
Screen-Ship Mbytes/frame	1.92	1.92	1.92	1.92	1.92	1.92

Table 6.4. Impostors without impostor reuse compared with screen shipping.

Size Limit (particles)	30K	100K	300K	1M
Number of Impostors	6644	2059	743	302
Framerate	18.397 fps	63.993 fps	98.032 fps	130.847 fps
Error	0.140231	0.125537	0.188875	0.221558
Overdraw	12.1	10.2	9.5	8.9
Client Drawing Time	18.76 ms	7.20 ms	3.20 ms	1.70 ms
Pixels Uploaded per Second	4401440.31	4274055.35	3619257.76	3213813.34
Impostors Uploaded per Second	443.59	222.49	112.67	60.98

Table 6.5. Performance as a function of the maximum amount of geometry per impostor. Octree nodes with more particles than the limit are subdivided.

Table 6.4 compares screen shipping with impostors where we have completely disabled impostor reuse—every frame, we render all the impostors from scratch. As can be seen, reuse provides the majority of the benefit from impostors. In fact, because different impostors overlap onscreen, without reuse impostors can actually cause more data to be shipped to the client than screen shipping, and the data rate the the client limits performance on large numbers of processors. For 4 or 8 processors, where rendering is the bottleneck, impostors actually still provide a small performance improvement over screen shipping. The reason for this is even without reuse, each impostor is shipped to the client as soon as it is rendered, which overlaps client communication with rendering work. This is beneficial compared to screen shipping, where the entire image must be assembled before any data is sent to the client.

Table 6.5 shows the performance impact of changing the size of the impostors. Generally, having too many impostors to draw makes the client display slow, but having too few impostors requires more frequent updates and hence (for a limited update rate) increases the overall error. We find having approximately 100,000 particles per impostor (approximately 40ms rendering grainsize) provides the best overall performance for this problem.

6.2.1 Automatic Load Balancing

The CHARM++ array manager [LK03] interfaces with the CHARM++ automatic load monitoring and load balancing system [BK00]. This existing Charm++ automatic load balancing system [Bru00] monitors the computational load on each processor and the load generated by each array element. If significant load imbalance is detected, a load balancer can migrate array elements between processors to improve the load balance.

An example of this migration improving the performance of a screen-shipping parallel rendering system is shown in Figure 6.2. This figure shows a timeline view from the CHARM++ performance visualization tool Projections [KKZL03]. In this figure, dark grey represents rendering work, light grey image assembly, and checkered load balancing. On the left, before balancing, because of the octree decomposition used, processors 4-6 do most of the work. Load balancing migrates some work off these overloaded processors onto less heavily loaded processors, improving processor utilization and overall performance. With load imbalance, the screen shipping approach takes 1.5 seconds per frame; after balancing, the frame time is 1.1 seconds.

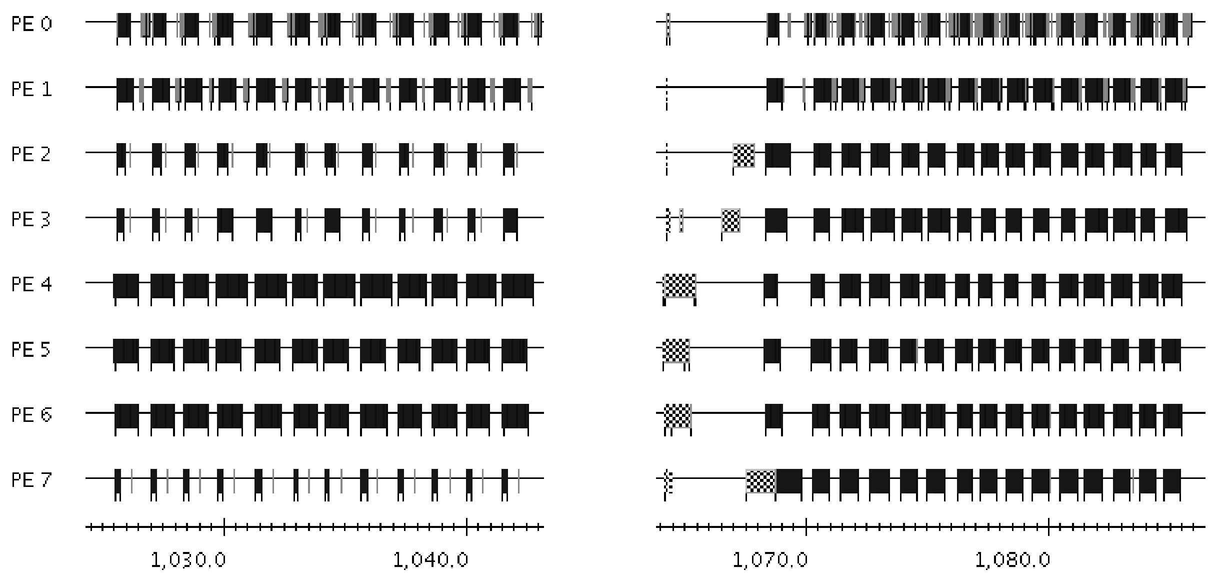


Figure 6.2. Timeline showing image assembly performance before and after several object migrations for load balance.

Table 6.6 shows the performance improvement of load balancing for several different numbers of processors. The benefit is relatively small, on the order of 10% even for a large number of processors. The reason for this is the octree decomposition we use is already relatively well balanced, as it is derived from the tree decomposition used in the astrophysics simulation.

6.3 Campus Parallel Performance Results

We present several parallel performance results for rendering the campus model. The rendering was performed at 1000 by 700 display resolution, rendered on a large cluster of dual-processor AMD Athlon MP 2000+ nodes connected with 100mbit Ethernet. Whenever possible, both CPUs of a node were used simultaneously.

These results are for a simple 10-second long path through the campus model, moving north along the west side of the Digital Computer Lab, and then west facing Beckman Institute, as shown in Figure 6.3. The camera path is a pure translation, and includes no camera rotation to avoid biasing the results in favor of impostors, for which rotation is free. However, to avoid startup transients the parallel impostor system is given the opportunity to prepare impostors for all visible objects (which normally takes less than two seconds) before the actual measurement begins.

Processors	4	8	16	24	32	48
Framerate Before	0.171 fps	0.287 fps	0.432 fps	0.529 fps	0.513 fps	0.652 fps
Framerate After	0.170 fps	0.284 fps	0.457 fps	0.542 fps	0.598 fps	0.679 fps

Table 6.6. Framerate before and after load balancing for screen shipping. Load balancing provides a small but increasing amount of improvement on large numbers of processors.

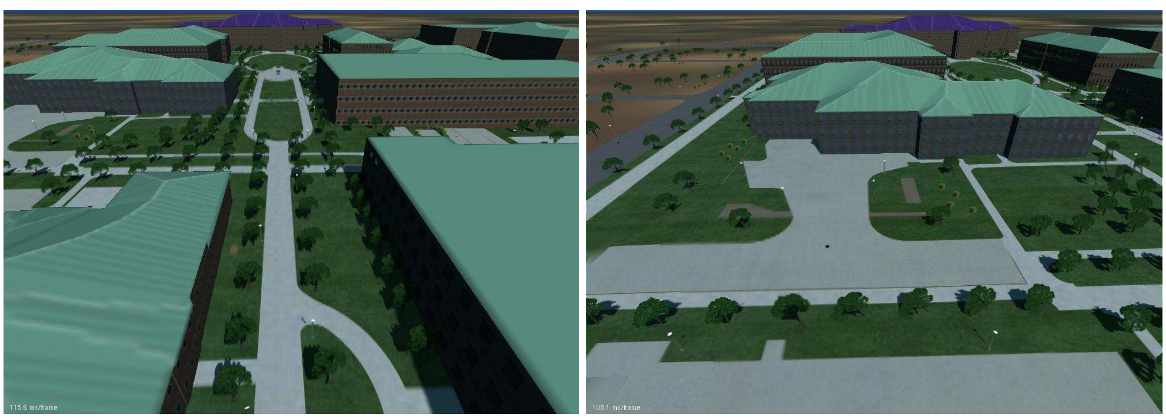


Figure 6.3. Path through the campus model used for performance evaluation. The left image shows the view near the beginning in the path, the right image shows the view from near the end.

Table 6.7 summarizes our results. In this table, “serial” means the campus rendered on one machine using only conventional OpenGL rendering on graphics hardware,[1] without impostors. Conventional serial rendering includes none of the advanced features described in later chapters, such as antialiasing, soft shadows, and adaptively expanded ground. Because of the complexity of the model—for example, each leaf is treated as as separate geometric object, and there are 2.7 million leaves rendered in the average frame—conventional serial rendering has a poor framerate. Because of the poor framerate, despite the fact that each image it prepares is perfect (with no geometric error, as would be found in an impostor system), conventional rendering has significant time-averaged pixel error.

The other columns of the table show our parallel impostors rendering system using the same client machine, but using the specified number of processors to render impostors using antialiasing, soft shadows, and adaptively expanded geometry as described in later chapters. As we add processors, the framerate slowly improves as newly-uncovered geometry is more quickly replaced by impostors instead of being drawn slowly on the client. Again because updated impostors arrive more quickly, the time-averaged error drops as we add processors, with diminishing returns setting in at about sixteen processors. The “No-Reuse Framerate” row shows the benefit derived from impostor reuse—if all impostors must be redrawn and shipped across the network every frame, impostors just slow the overall system down.

We examined one component of the geometry rendering rate, the total number of tree leaves rendered per second. This rendering rate increases substantially as we add processors. Although the serial machine renders leaves using the graphics hardware, and the parallel machine renders in software, because leaves are entirely geometry-rate limited and because of the efficient software implementation described in Section 7.2.1.1, the graphics hardware cannot render leaves substantially faster than software rendering. Note also that unlike in the previous section, time is also spent rendering buildings, the ground, and other objects—the campus is not composed entirely of leaves.

The network data rate required by the impostor rendering system is only a few megabytes per second, which is easily satisfied using CCS on fast ethernet. Particularly interesting is the system’s overall delivered fill rate, in finished screen pixels per second. Because of all the tiny overdrawn geometry in this model, conventional rendering makes very poor use of the graphics hardware’s fill rate potential. As the fill rate row shows, impostors allow the graphics hardware to deliver far more finished pixels per second.

[1] The client graphics card is an nVidia Quadro FX 500 (NV34GL).

	Serial	2	8	16	32	64
Framerate	1.47 fps	26.68 fps	29.95 fps	31.78 fps	32.12 fps	31.88 fps
Error	0.557	0.222	0.188	0.169	0.165	0.163
No-Reuse Framerate	1.47 fps	0.17 fps	0.54 fps	0.71 fps	0.73 fps	0.69 fps
Render Rate (Mleaf/second)	3.81	7.81	49.24	82.36	109.95	132.26
Network Rate (Mpixels/second)	-	0.108	0.514	0.682	0.757	0.811
Fill Rate (Mpixels/second)	0.98	18.68	20.96	22.24	22.48	22.32

Table 6.7. Performance numbers for the campus rendering problem rendered using various amounts of hardware.

	Serial	2	8	16	32	64
Prioritized Error	-	0.222	0.188	0.169	0.165	0.163
FIFO Error	-	0.219	0.214	0.213	0.198	0.185

Table 6.8. Benefit from prioritized rendering.

Table 6.8 shows the benefit of using prioritized rendering. In an impostor system, we can and must choose the order in which to re-render impostors. Our system prioritizes the impostor rendering and shipping order based on the very simple metric of screen diameter, estimated from the object's bounding sphere and the distance to the camera. For small numbers of processors, replacing this prioritization system with rendering in simple request-time order (first-in, first-out or FIFO) results in much higher overall error. The reason for this is that for the constants of this problem, almost all geometry quickly becomes out of date and could be re-rendered. But this means treating all rendering requests equally leads to the neglect of the few large and crucially important rendering requests, and hence wastes most rendering effort on the many distant unimportant impostors. With enough processors, there is time to satisfy almost all rendering requests, so the rendering quality delivered by FIFO finally begins to approach that of a prioritized scheme.

Overall, these results demonstrate that even on a reasonable number of processors, parallel impostors can indeed render very large models with high quality.

6.4 Network Compression

The primary performance bottleneck in many parallel rendering situations is the network link between the server and the client. Hence one way to improve overall performance is to compress the images being shipped across the network.

We analyzed four methods for image compression.

- Portable Network Graphics (PNG)[2] is a entropy-encoded lossless compressed image format. PNG fully supports alpha-channel transparency.
- Joint Photographic Experts Group (JPEG)[3] is a quantized DCT spatial frequency-based lossy compressed image format. We used the common version of JPEG, which does not support alpha channels directly. Because alpha channels are required by impostor-based systems, the alpha channel would have to be extracted and compressed separately, though for our performance analysis we ignore this. We did not use the latest wavelet-based JPEG2000 standard; though it does support a full alpha channel, the only noncommercial library for JPEG2000[4] is still approximately 20x slower than plain JPEG at compression and decompression.

[2] We used libpng 1.0.12/zlib 1.1.3 for our performance tests.

[3] We used The Independent JPEG Group's JPEG-6b library for our performance tests.

[4] JasPer 1.701.0

Figure 6.4. Impostor image used to test compression methods.

- Uncompressed images, shipped as plain 32-bit per pixel values. Most older graphics image formats are essentially uncompressed, including TGA, PPM, and most flavors of TIFF and BMP.

- Run-length encoding, as specialized for impostor images. This version of run-length encoding eliminates completely transparent regions at the start and end of each row, and ships the remainder of each row as ordinary 32-bit pixels. Run-length encoding is often used in TIFF files.

Our implementation of each of these compression methods runs completely in-memory and in-process,[5] and never touches any disk files. We analyzed each compression method for a range of square image sizes, from 4 pixels square to 1024 pixels square. The image under test is a fairly typical impostor image of a tree, shown in Figure 6.4. This image consists mainly of empty space, which seems fairly typical of impostors of objects with arbitrary shapes.

Figure 6.5 and Table 6.9 plot, for each compression method, the network time to ship the compressed images across a network at 10 megabytes per second, as well as the CPU time to compress and decompress the images. Compression time is spent in parallel on the server, and is hence not as important as decom-

[5]To do this, we use the libpng png_set_read/write_fn and jpeglib fill_input_buffer interfaces, and perform our own image buffer management.

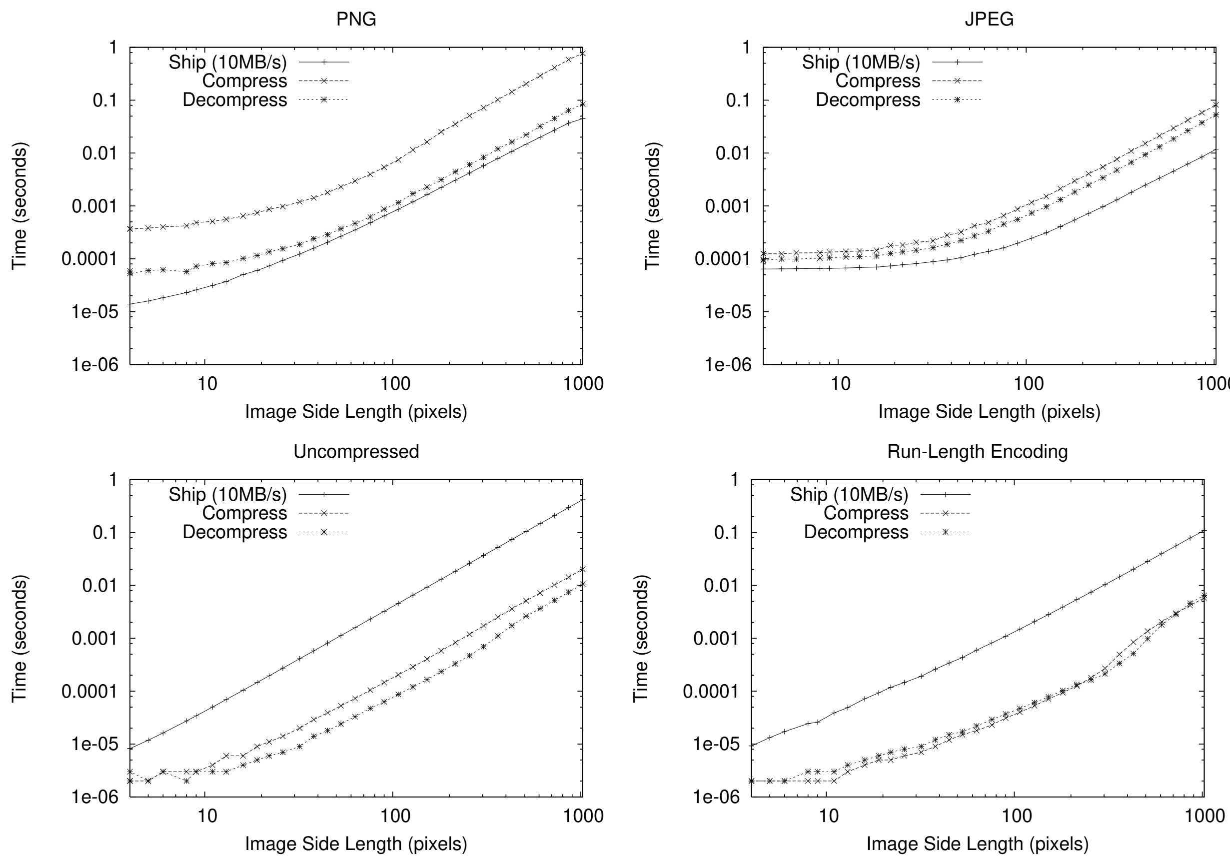


Figure 6.5. Breakdown of the time spent to compress, send over the network, and decompress images of various sizes for various image compression methods. Top left is PNG, top right is JPEG, bottom left is uncompressed, and bottom right is run-length encoding.

pression time spent on the client. Further, because shipping, compression, and decompression are each only stages of the pipelined display process, none may be the bottleneck.

But if we limit our focus to simply shipping images, our overall performance will be limited by the slowest component. Hence Figure 6.6 plots the time per image, as limited by the slowest component. Overall, PNG is limited by the expense of its compression stage, which by design performs an exhaustive search to determine the best representation for each image tile. Uncompressed and run-length encoded images are both always limited by the network data rate, as compression and decompression are very simple but the resulting images are large. Run-length encoding was always both smaller and faster than uncompressed images in this test, with a more significant performance advantage for larger images.

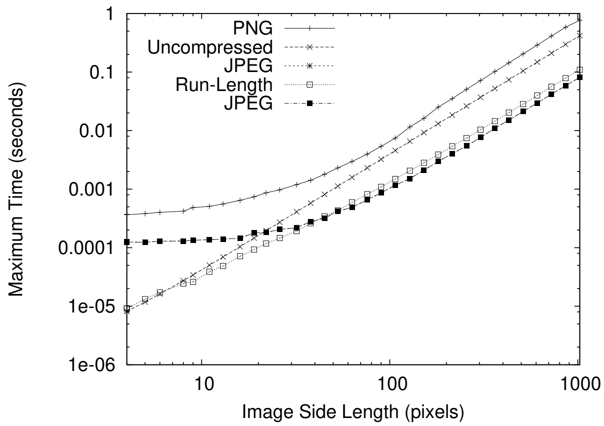


Figure 6.6. For various image sizes and compression types, the maximum of the compression, shipping, and decompression times. This is the throughput-limited time to ship an image of this type. Assumes a 10MB/s network.

JPEG produces very small compressed images, and is hence always limited by its compression and decompression speed. JPEG is relatively slow for small images, less than 40 pixels on a side, which is likely due to the complexity of the JPEG library and processing. However, for larger images, for a 10MB/s network, JPEG is actually slightly faster overall than run-length encoding. There are three reasons not to use JPEG: first, JPEG is a lossy compression scheme; second, the limiting factor for JPEG is the CPU speed of the client, so a faster network will not help it; and finally, JPEG cannot directly represent the impostor alpha channel.

In conclusion, for fast networks a simple run-length encoding technique seems to be the best choice for an impostor-based system, as it is smaller than uncompressed images, but still requires a very small amount of CPU time. For slow networks and fast processors, JPEG would also be a reasonable choice.

	Pixels	Compressed Size	Compress Time	Decompress Time
PNG	32 x 32	1229 bytes	1.195 ms	0.187 ms
JPEG	32 x 32	881 bytes	0.219 ms	0.162 ms
Uncompressed	32 x 32	4096 bytes	0.020 ms	0.009 ms
Run-Length	32 x 32	1916 bytes	0.007 ms	0.009 ms
PNG	128 x 128	11947 bytes	11.535 ms	1.708 ms
JPEG	128 x 128	3105 bytes	1.512 ms	0.965 ms
Uncompressed	128 x 128	65536 bytes	0.288 ms	0.121 ms
Run-Length	128 x 128	20636 bytes	0.053 ms	0.061 ms

Table 6.9. Compression times and sizes for various compression methods and image sizes.

Chapter 7

Quality Rendering Enabled by Parallel Impostors

In this chapter, we describe a set of techniques to render high-quality parallel impostors. This extends a different axis of the parallel impostors benefit space: improved rendering quality.

Rendering geometry for an impostor is quite similar to rendering for ordinary display. Because an impostor is simply a raster image of the object, virtually any technique can be used to render impostors. For example, impostors could be rendered using the classic feed-forward z-buffer algorithm on graphics hardware, a scanline algorithm in software, splat-based rendering, image-based techniques, distribution raytracing, or even physically-based radiation transport.

In fact, the main difference between ordinary realtime rendering and rendering for a parallel impostor is that we can spend much more time rendering when using parallel impostors, because of impostor reuse and the additional compute power provided by the parallel machine. In this chapter, we explore several different methods to use that time to significantly improve image quality. In Section 7.1, we explore how to improve the appearance of impostors by antialiasing the geometry during rendering. In Section 8, we explore several novel methods for improving the accuracy of the impostor lighting. In Section 7.2, we describe how we render recursively generated foliage with high quality.

7.1 Antialiased Impostor Rendering

Antialiasing is the problem of filtering the arbitrarily high resolution of the world onto the fixed and low resolution of the graphics device. The aliasing problem arises if we consider pixels as point samples of the world. Because the sampling locations of two adjacent pixels may land arbitrarily far apart, the pixels may receive completely different colors even in very similar regions, or may receive identical colors even in very different regions.

The aliasing problem is illustrated in the left portion of Figure 7.1. Note how in the distance the tiny regular checkerboard pattern has been aliased into an apparently random pattern, with artificial large-scale variations.

The aliasing problem can be theoretically analyzed using sampling theory. The true distribution of light across the camera view plane can be treated as an unknown signal, and pixels as samples of this signal. In the spatial frequency domain, regular point sampling corresponds to a comb filter of infinite extent, which means tiny, arbitrarily high spatial frequency features still pass through the filter unattenuated, and are erroneously reconstructed as large, low frequency features.

For textures, various well-known methods exist to filter out these aliases. By far the most popular is mipmapping, which simply selects the appropriate image size from a stacked pyramid of successively lower resolution images. An "anisotropic" technique, that allows the texture to be compressed different amounts in different directions, is the ellipsoidal weighted average, as described by Heckbert [Hec89]. As illustrated in Figure 7.1, these methods work quite well for texture filtering; and these methods are today implemented reasonably well on graphics hardware.

But aliasing still appears in modern images. The reason is that texture filtering only eliminates aliasing caused by features in textures, but geometry can alias as well.

Conventional rendering, including Z-buffer systems as well as simple raytracing, performs simple point sampling of the geometry in the scene. This results in objectionable aliasing artifacts along object boundaries, and can cause small or thin geometric features to break up or vanish entirely. Many methods exist to avoid these rendering artifacts; and most of these are useful with impostor-based rendering.

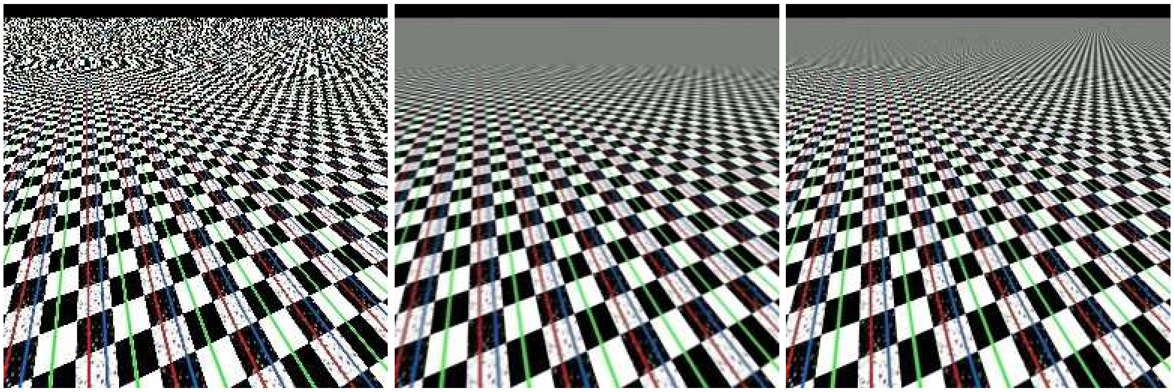


Figure 7.1. Simple checkerboard pattern extending to infinity, used to illustrate various texture antialiasing methods. On the left, aliased point sampling. In center, conventional mipmapping. On the right, anisotropic ellipsoidal weighted averaging.

7.1.1 Antialiasing via Supersampling

The most obvious and widely used technique to reduce geometric aliasing artifacts is supersampling—simply taking more than one sample per pixel. As noted by Cook, Porter, and Carpenter [CPC84], taking extra samples suffices to reduce aliasing along object boundaries (spatial antialiasing), as well as in texture contents, and even across time within the frame (temporal antialiasing). Modern graphics hardware supports a variety of sampling patterns with a very modest degree of supersampling, typically from 2 to 16 samples per pixel.

The worst-case expected variance of a point-sampled image can be determined by the central limit theorem. If the true light distribution within a pixel has standard deviation σ, a pixel's estimate based on n random point samples will have standard deviation approaching:

$$\sigma_p = \frac{\sigma}{\sqrt{n}}$$

For the common case of simple sub-pixel geometry, the well-known technique of stratified sampling can improve the quality of the estimate somewhat. The actual improvement is to approximately $\sigma_p = \frac{\sigma}{n^{3/4}}$ when there are few edges, or even $\sigma_p = \frac{\sigma}{n}$ when there are no edges [Mit96].

However, both theory and practice show that to achieve smooth images, many samples are required. For a problem pixel halfway along a pure black/white edge, $\sigma = 0.5$. To even approach the limits of human vision, we might take $\sigma_p = 0.01$. Then for stratified sampling, where $\sigma_p = \frac{\sigma}{n^{3/4}}$, $n = 184$ samples are required, which is normally not affordable for realtime rendering. For a more complicated pixel, where only the central limit theorem holds, $n = 2500$.

However, in practice, users are willing to tolerate surprising amounts of variation σ_p along edges, so supersampling is useful even for online rendering.

7.1.2 Analytic Antialiasing

Hart [HCK+99] has described several techniques for analytically filtering image functions with a known analytical form, including modulated quadrics and noise functions. McCool [McC95] describes box spline filtering, a fully analytic technique that supports arbitrarily complicated pixel basis functions.

We use a simple trapezoid-based analytically antialiased rendering system to render impostor geometry. Like Duff [Duf89], we compute the per-pixel area of a polygon by computing a contour integral. Conceptually, we decompose the polygon into half-infinite trapezoids, extending each edge of the polygon infinitely far to the right. These trapezoids can then be rasterized by walking down the scanlines they touch, and adding the signed, fractional area of their leading edge to their scanlines. Then to extract the polygon coverage for each pixel of a scanline, we simply sum up these signed fractional areas. This technique, and our implementation, both work for arbitrarily complicated polygon shapes, including shapes with self-intersections and holes. The antialiased rasterization process produces a set of (partially transparent) pixel spans, so arbitrary code can then use these spans.

Because rasterization is decoupled from pixel movement, we can then use a raytracer to provide color samples that fill an antialiased polygon outline. This results in analytic antialiasing for a raytracer. Or we can use the spans to fill in a soft shadow map, as described in Section 8.1.

Figure 7.2 demonstrates both sampling-based and analytic approaches to geometric antialiasing. Moving left to right, note first the problems with the pure point-sampled version on the left—the complicated tree geometry is replaced with random noise, and the polygon edges fall in sharp stair-steps. Stratified samples improve the image significantly, but random sampling noise is still present even at high sampling rates, and especially visible along smooth polygon boundaries. Analytic antialiasing results in a completely smooth image, similar to the high-sampling rate point sampled image, but without sampling noise.

However, note that with analytic filtering, the boundary between the two polygons in the bottom row of Figure 7.2 is visible as a light line, where the background partially shows through the adjacent edges of the polygons. The problem here is that the first polygon is drawn, its boundary is composited against the background before the second polygon is drawn. But compositing the boundary pixels twice with 50%

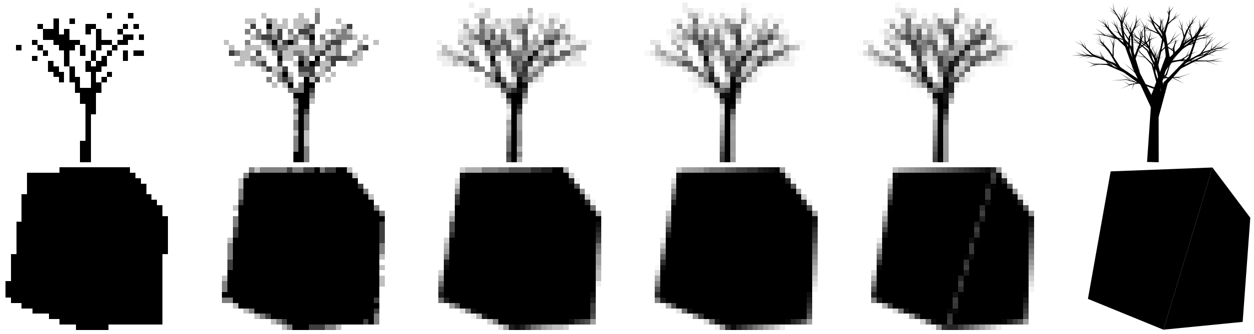


Figure 7.2. Comparison of several approaches to antialiasing. Top row is a tree outline, bottom row is two separate but adjacent polygons. From left to right, 30x30 pixel raster images prepared with: on extreme left, aliased single-sample pixels; in center 4, 16, and 64 stratified random samples per pixel; trapezoid-based analytic filtering; and on the extreme right, the true vector outline.

foreground color is not equivalent to compositing them once with 100% foreground color, so some portion of the background color gets mixed into the foreground and still remains after compositing.

A metaphor may be useful here. Compositing operations treat a pixel as a well-mixed bucket of paint. Compositing simply pours off some of the existing mix, and pours in new color. Initially, pixels are clear, so the bucket is full of clear water. If we pour off half the water, fill the remainder with pure black, pour off half that mix, and refill with black (i.e., perform two compositing steps), we end up with 75

There are two well-known solutions to this "double compositing" problem. If the fraction of remaining background color can be identified, then after all polygons are drawn any remaining background color can be removed. For example, we could initialize the background alpha channel to 0, then divide out any remaining background alpha after compositing. Alternately, each polygon can be slightly expanded, so it entirely covers its border pixels. This guarantees each border pixel will be composited at 100% coverage at least once, but also causes some geometric error. For impostor rendering, we use both these double compositing solutions in different contexts—for ground images, where we can identify the background, we use the alpha division technique; while for freestanding impostors, where the background cannot be clearly identified, we use polygon expansion.

7.1.3 Antialiasing Impostors

Any of the above techniques could be used to antialias impostor images.

Texture maps take advantage of the client graphics hardware's builtin texture antialiasing. By preparing impostors with antialiased edges, even object boundaries can be antialiased at very low cost. We prepare impostors with antialiased edges using the analytic trapezoid-based rasterization system described above.

Because an impostor is rendered for a fixed display orientation, a useful technique is to use anisotropically filtered geometry and texture while rendering the impostor. Impostor display can then use the hardware's simple and fast interpolation strategy, but because the impostor is already anisotropically filtered, the display quality can approach that of true anisotropic texturing.

This technique is used for ground rendering, where we choose the ground resolution to approximately match the screen resolution. The various ground regions (such as roads, sidewalk, and grass) are then rendered using trapezoidal filtering.

7.2 Trees and other Foliage

Both trees and other foliage are procedurally generated using an *Iterated Function System* (IFS). That is, instead of storing a list of branches and leaves, we store a small list of matrices that generate sub-branches from a main branch. The first branch is the trunk, and smaller branches recursively expand from it. At some fixed cutoff, we draw leaves at the end of the current set of branches. To generate bounding volumes for this recursively generated procedural geometry, we use our IFS bounding method, described in Appendix C. As described in the subsequent sections, trees are rendered using a simple splat-based renderer for the leaves, and antialiased raytraced cone segments for the trunk and branches.

For very nearby trees, we switch from using impostors to procedurally generating and rendering the tree geometry on the client directly. This avoids the low reuse rate for impostors of extremely close objects.

7.2.1 Leaf Rendering

Each leaf is, when far away, rendered as a single splat, with alpha scaled to the leaf's pixel-area coverage. When a leaf projects to more than a pixel, the leaf is drawn as an antialiased polygon. When closer yet, a leaf is drawn as a textured polygon. Leaf colors are pseudorandomly selected from a distribution of leaf colors.

For very distant trees, we artificially decrease the level to which we expand the leaves, and scale up the effective leaf size to keep the total area covered constant. This level-of-detail shift makes rendering very distant trees substantially faster, as shown by the jumps in the rendering time shown in Figure 7.7.

7.2.1.1 Splat Rendering

Because a typical outdoor scene contains more leaves than pixels, many of them quite distant, we must be able to efficiently render very small leaves. We choose to render distant leaves as antialiased splats—sub-pixel colored dots.

Our initial software implementation assumed some true geometric shape for the leaf, and calculated the per-pixel partial coverage by intersecting the leaf shape with each pixel. Unfortunately, this flexible, principled implementation took 4us per splat, which limits us to just 250K points per second.

Our current software implementation uses the following optimizations:

- Splat shapes are restricted to one-square-pixel boxes. This allows us to use a small lookup table that stores the coverage information for several adjacent pixels. The table is indexed by the sub-pixel splat offset, and contains a coverage value for the 4 adjacent pixels. This pushes the slow geometric intersection computations away from the critical path.
- Splat alpha blending is done using Intel MMX instructions. This allows all 4 channels (red, green, blue, and alpha) to be performed simultaneously.
- Splat colors, coverages, and locations are passed in as fixed-point numbers. This gives up some convenience, but avoids the surprisingly slow Intel floating-point to integer conversion.

Our optimized software implementation takes approximately 80ns per splat, or over 12M points per second, which is competitive with quality antialiased splats on graphics hardware. Perspective projection, clipping, and other processing normally take another approximately 50ns per splat.

A 100x100 array of antialiased splats drawn using our implementation in 1.6ms is shown in Figure 7.3. The same array drawn using conventional one-pixel aliased splats is shown in Figure 7.4. Note how the regular lines of splats break up and wobble in the aliased version. The effect is even worse in animation.

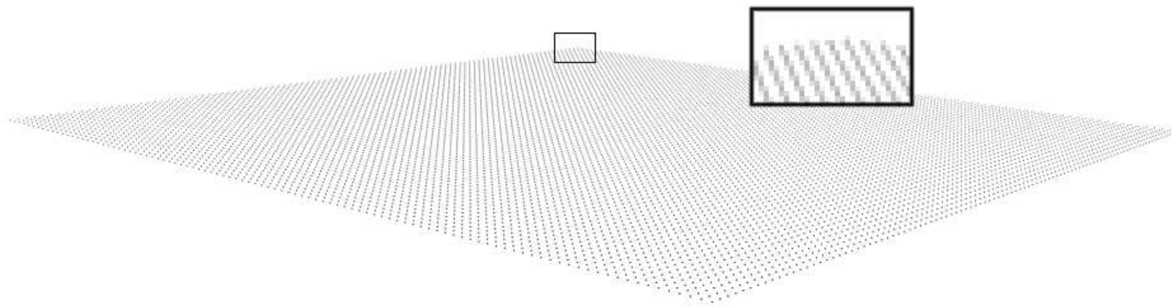


Figure 7.3. 100x100 array of antialiased splats. Note the smooth, regular lines.

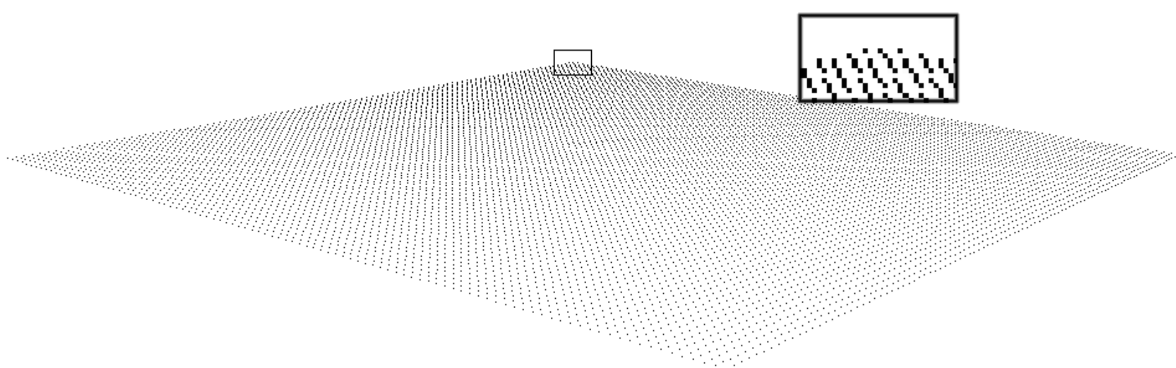


Figure 7.4. 100x100 array of conventional aliased splats. Note the broken, jumpy lines.

7.2.1.2 Leaf Clusters

To draw a tree, our initial implementation traversed the branches of the tree IFS in a purely recursive fashion. Starting at the root, we expanded the various subtrees until we reached the leaf level, then drew leaves. Because expanding a subtree means performing a matrix multiply to find the new subtree's coordinate system, this meant we performed one matrix multiply per leaf. But after optimizing our leaf rendering, this per-leaf matrix multiply was actually the bottleneck in overall tree rendering performance.

The obvious workaround for this problem is to expand and store all the leaves of the tree. This avoids doing any matrix multiplies at runtime, but has a high memory overhead because every possible leaf must be stored. A typical tree might have on the order of 100,000 leaves, which works out to several megabytes of leaves per tree.

We can avoid both the per-leaf matrix multiply and the high memory cost of pre-expansion using a hybrid method. We begin each rendering by recursively expanding subtrees at the high levels of the tree. Once we are within k levels of the individual leaves, we switch to a precomputed *leaf cluster*, a simple list of 3D leaf locations expressed in the coordinate system of the branch k levels above. Because all the subtrees of an IFS are identical up to an affine transformation, we only need to store one leaf cluster for the entire tree, and render it in its parent's coordinate system. We choose to cache $k = 4$ levels of the tree, which cuts the number of matrix multiplies for a $b = 3$ branch tree by a factor of $b^k = 81$. Using leaf clusters thus does not affect the image, but dramatically reduces the number of matrix multiplies needed to traverse the IFS, which accelerates rendering substantially.

7.2.2 Z Sorting

Trees are composed of many individual leaves and branches. To draw these in the correct back-to-front order for the painter's algorithm, we use a sorting algorithm to order the rendering process.

Our final bottleneck in our initial implementation was sorting, as the standard C library qsort routine took longer, per element, than our splat-based leaf renderer. The time spent sorting can be reduced in two ways. First, we can sort fewer elements, by only sorting leaf clusters instead of individual leaves. Second, we can sort faster by using a faster sort implementation. For example, we found the C++ standard library std::sort was several times faster than qsort, largely due to the inlining of the element comparison routine. By using a few bits of the floating-point exponent and mantissa as a sort key, the fastest sort for large trees is a two-pass radix sort.

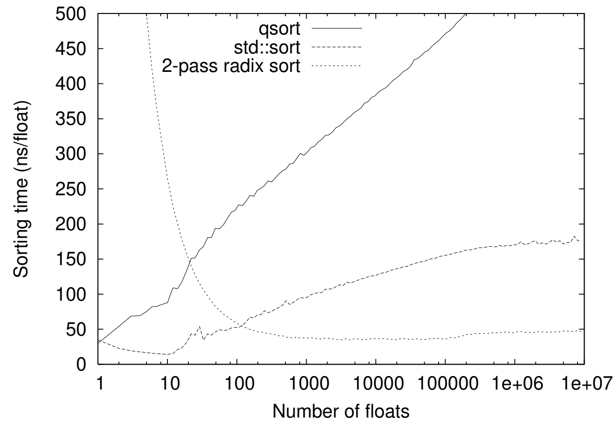


Figure 7.5. Time per element for various sorting methods.

The sort time for various sorting implementations and problem sizes is shown in Figure 7.5. qsort and std::sort are both randomized $O(n \lg n)$ expected time comparison-based methods, and the time per element indeed increases with n. Radix sort of n elements is $O(n)$, but there is a large fixed overhead in building the histogram tables, so comparison-based sorting is faster for small input sizes.

Leaf clusters and individual branches are both sorted by their centroid, and splatted into the impostor in back-to-front order. An example of a tree rendered using these techniques is shown in Figure 8.7.

7.3 Buildings

The walls of buildings on campus can be extruded directly from their outline. Because walls are quite flat, to achieve a high impostor reuse rate we simply render each wall as a separate impostor aligned with the plane of the wall.

Very distant windows are drawn in as simple recessed dark rectangles. Closer windows use multitexture on the client to alpha-composite the wall image atop a static neighborhood environment map. In each case, we can control the reflection intensity and add lit dust by adjusting the transparency and opacity of the window section.

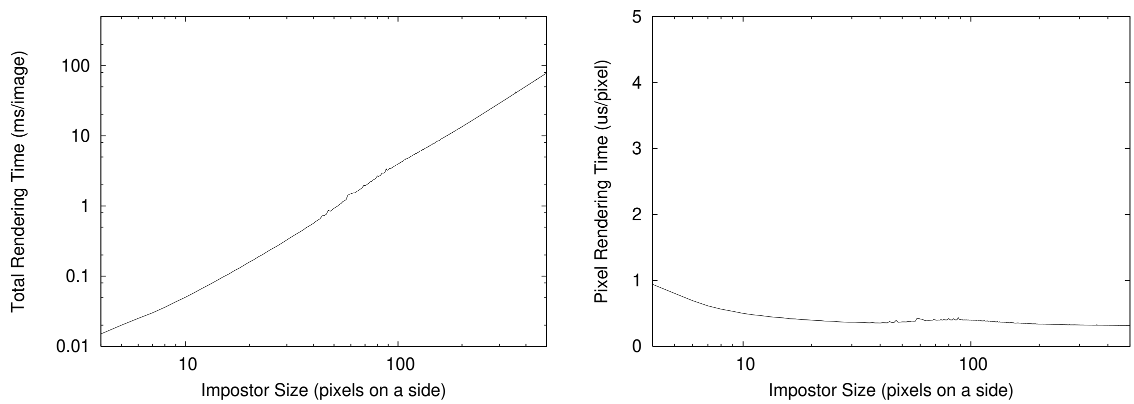


Figure 7.6. Overall time to render a finished polygon impostor, in milliseconds per impostor (left) and microseconds per pixel (right), for various output image sizes.

7.4 Other Geometry

The geometry we have described above such as trees and buildings is *procedurally generated*, where nothing is stored but the basic object parameters, and the geometry of the object is freshly synthesized at rendering time. Other geometry that is difficult to represent this way will be rendered using simple textured polygons read from a file using the usual feed-forward Z-buffer rendering system. We use this pregenerated geometry to represent lightpoles and traffic lights.

7.5 Overall Rendering Performance

Figure 7.6 shows the performance of our antialiased polygon renderer, discussed in Section 7.1, on a 1.6GHz Pentium M processor. The drop in per-pixel time as image sizes increase is caused by the amortization of our antialiased row setup as scanlines get longer and longer.

The times in this section also includes full lighting, using our soft shadow map technique. The time to prepare a soft shadow map is approximately one microsecond per shadowmap pixel.

Figure 7.7 shows the performance of our tree renderer, as discussed in Section 7.2. The jumps in rendering time happen when we render an additional level of geometry for the level-of-detail filtering.

Note that even with the optimizations presented in this chapter, all these rendering methods are still too slow to simply run directly on the client. But the reuse of impostors allows us to amortize away the rendering cost for these high-quality images, and using the parallel machine allows us to perform the rendering on

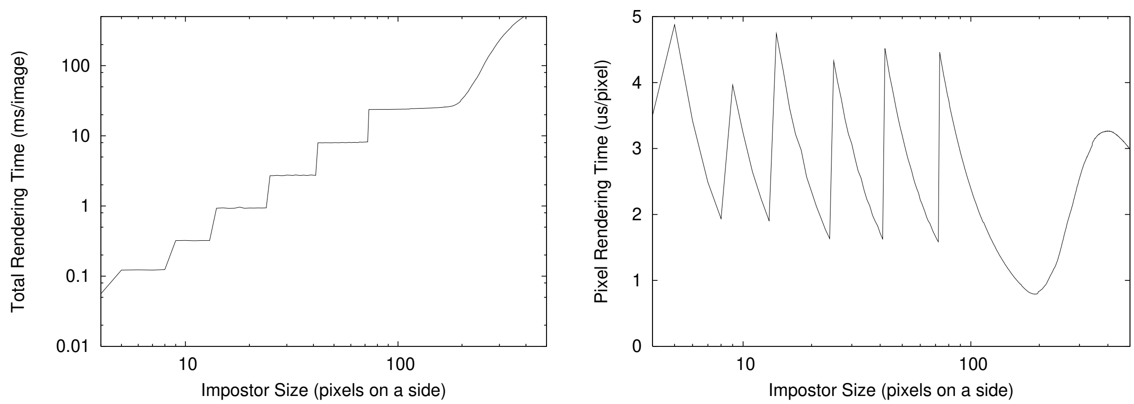


Figure 7.7. Overall time to render a finished tree, in milliseconds per tree (left) and microseconds per pixel (right), for various output image sizes.

many processors in parallel. Overall, these high-quality techniques fit nicely within our target rendering rate of approximately 1M rendered pixels per second per processor.

Chapter 8

Lighting Complex Scenes

This chapter describes techniques for computing both direct and indirect illumination. Direct illumination is light that travels directly from a light source to the illuminated object. For example, sunlight is considered direct light. Indirect illumination, by contrast, arrives at an object after being reflected one or more times. For example, sky light is considered indirect illumination.

In our system, lighting is computed on the parallel server, and is included in the pixels of the impostors when shipped to the client—that is, impostor images contain radiance, not albedo. This means the color of an impostor pixel does not change with viewpoint, which may seem to imply we are restricted to completely diffuse (Lambertian) surfaces, but in fact even pre-lit impostors actually support arbitrary surface BRDFs. For a non-diffuse surface, as the view changes, a surface's impostors could be re-rendered to display new colors driven by the surface reflectivity, in addition to the usual geometric parallax change. Because outdoor scenes consist largely of complicated, nearly diffuse objects, geometric change dominates the re-rendering decision, and is hence the only re-rendering criterion we consider.

8.1 Direct Lighting

Direct light, such as sunlight, is often approximated by a point light source, which casts a hard-edged shadow. But in reality an area light source casts a soft shadow, with a fully black umbra surrounded by the partially bright penumbra.

Figure 8.1 shows the geometry of an occluder, light source, and shadow. We define the "inner penumbra" as the portion of shadow where part, but less than half of the light source is visible. We define the "outer

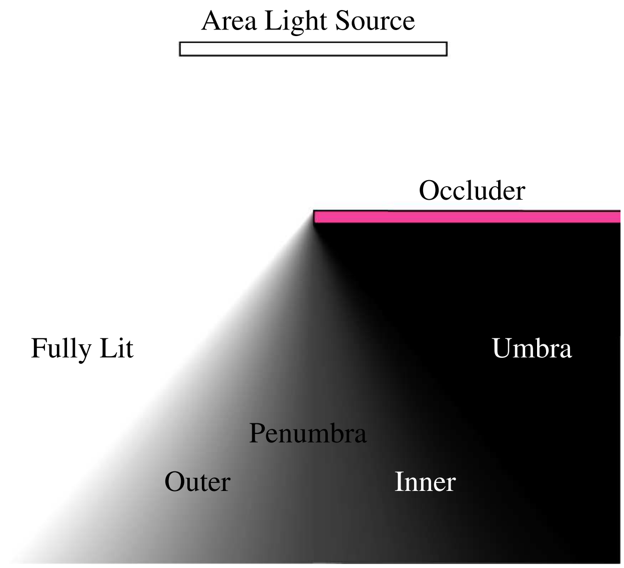


Figure 8.1. Cross-section of a simple soft shadow scene.

penumbra" as the portion of shadow where more than half of the light source is visible, but not all. The dividing line between the inner and outer penumbra is just the shadow cast by a point light source in the center of the area light source.

8.1.1 Prior Shadow Work

There are three classes of techniques to compute shadows in computer graphics. Raytracing can trivially determine if a light source is visible from a point and can hence determine extremely accurate shadow boundaries. *Shadow volumes* are a screen-space technique which extrudes, rasterizes, and counts signed object silhouette crossings for each pixel—visible surfaces that lie beneath a nonzero number of silhouettes are in shadow. Finally, *shadow maps* rasterize the scene from the light source's point of view, and store the depth to the first occluder—objects behind the first occluder are in shadow. Broadly, raytracing is generally quite slow, shadow volumes are slow for objects with complicated silhouettes (such as trees), and shadow maps display sampling, aliasing, and depth precision problems.

There exist a variety of methods, such as the "penumbra wedge" soft shadow volume technique of Assarsson [Ass03], that can accurately compute physical penumbrae. However, few of these techniques are affordable in real time, especially for objects with very high silhouette complexity such as foliage.

A good survey of existing soft shadow modifications to shadow map techniques is presented by Hasenfratz et al. [HLHS03]. Many older algorithms exist which compute soft shadows as a combination of several hard shadows, which is equivalent to approximating the extended light source as a collection of point light sources. The usual limitations of point sampling mean these approaches, like distribution raytracing, require a very large number of samples to average away aliasing problems, and are hence normally only usable offline.

A tech report by Parker et al [PSS98] introduced a widespread technique for creating soft shadows, by pasting on ficticious geometry to make the boundary of all objects fuzzy, allowing some light to leak through along the boundaries. This is of course physically inaccurate, and in particular generates soft shadows which do not properly expand as they propagate.

Haines' Shadow Plateaus [Hai01] use a silhouette meshing technique, which generated physically correct shadows for a fixed planar receiver. Non-planar shadow receivers are not computed correctly by this method, however, and the method is slow when the silhouette complexity is high.

Brabec and Siedel [BS02] show a slow "nearest occluder" search process based on a standard depth map for computing soft shadows. This search process is difficult to implement in graphics hardware, and quickly becomes the bottleneck for reasonably blurry shadows.

Our work is similar to the realtime soft shadow-map technique of Kirsch and Döllner [KD03]. In this method, the geometry is first rendered to the shadow depth map normally. The depth map is then used to compute a second "shadow-width map" which stores the width of the soft shadow—the distance to the soft shadow's edge. To evaluate the shadow intensity at a point, the distance from the point to the shadow depth map is combined (using a lookup table) with the shadow-width map value to produce a soft shadow. Because the shadow-width map is only computed for points inside the shadow depth map, this technique can only capture the inner penumbra—the computed soft shadows will never extend beyond the object's hard shadow, which is physically incorrect. Though they show an interesting method to compute the shadow-width map from the shadow depth map on the graphics hardware, their method requires many passes (8 passes with 7-way multitexturing in their example) and is hence relatively slow. Finally, because of the way their formulation interpolates the shadow-width map, thin dark ridges can appear near the soft shadow's outer boundary.

Chan and Durand's "smoothies" technique [CD03] also begins with the usual shadow depth map, but like our technique adds two additional buffers: the smoothie depth and smoothie alpha. The smoothie buffer is computed by traversing the object silhouettes, and extruding a "smoothie" to provide a soft shadow along each silhouette. At rendering time, the depth map and smoothie maps are combined to produce soft shadows. Because the smoothies are pasted to the outsides of the objects, this method can only produce outer penumbrae, which is again physically incorrect. And because the smoothies are computed based on silhouettes, the method will be inefficient for objects with high silhouette complexity, such as foliage.

Wyman and Hansen's penumbra maps [WH03] are similar to smoothies, and have a similar set of drawbacks. Both techniques also require the shadow receiver to be known while preparing the soft shadow map.

Arvo, Hirvikorpi, and Tyystjärvi [AHT04] presented an interesting soft-shadow implementation where the penumbra properties are computed in screen space. As with other work, these shadows are not physically accurate, and performing the computation in screen space results in various artifacts, especially for complex occluders. A screen space implementation does not allow for antialiased geometry to be rendered. The performance of this technique is also limited by the many passes required for their soft shadow propagation phase.

8.1.2 Penumbra Limit Shadow Formulation

In this section, we present a formulation, which we call "penumbra limit shadows," which has the following attributes.

- Geometrically and radiometrically physically exact, at least for the common case of an infinitely distant radially symmetric light source occluded by a straight edge. The method includes both inner and outer penumbrae.
- Visually plausible everywhere.
- The shadow receiver geometry need not be known at shadow generation time. That is, a single shadow map can cast physically correct shadows on surfaces at any depth.
- Trivial to compute, and involves zero arbitrary parameters.
- Can be derived from antialiased geometry, not aliased pixel-sampled geometry.
- Highly interpolation friendly, with very mild sampling and interpolation artifacts.

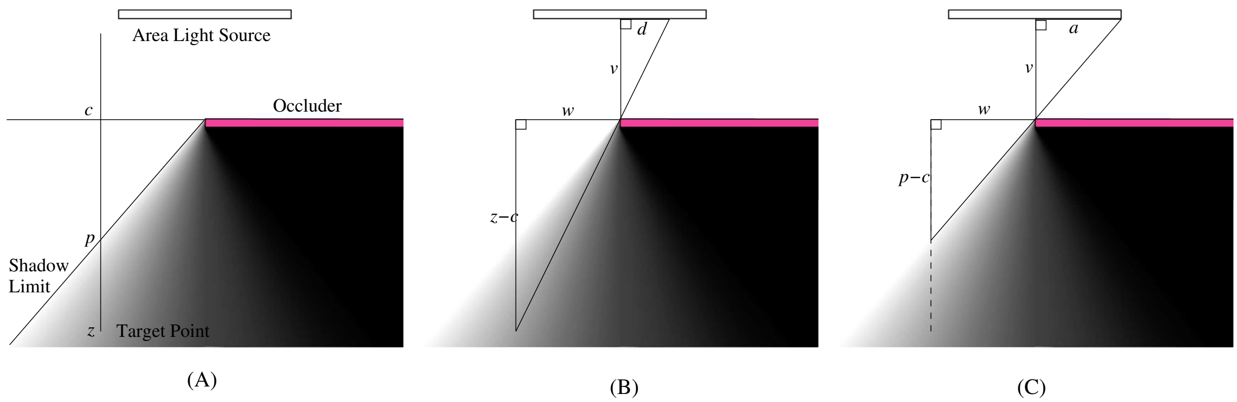


Figure 8.2. Derivation of penumbra limit equations.

Consider first the simplest 2D case: an extended but infinitely distant light source directly overhead, and a half-infinite occluder perpendicular to the light source. Our shadowmap parameterization runs along parallel vertical lines, with increasing depth in the downward direction.

As shown in Figure 8.2 (A), we compute and store two depths for each light-source ray. The first depth, c, is the depth of the occluder corner that casts the shadow. This value only changes when switching between soft shadows cast by different objects or different parts of the same object. The second depth we store, p, is the penumbra limit depth. In the outer penumbra, which we consider first, p is the actual depth to the start of the penumbra, and is hence always less than c and decreases away from the object.

We want to find the fraction of the light source visible from some target point. Call the depth of that point z. As shown in Figure 8.2 (B), two similar right triangles are formed by the target point and occluder corner, and the occluder corner and light source hit point. For the moment consider only the right half of the light source, and take the horizontal target/occluder distance as w, the light source height v, and the visible length of the light source d. Then by similar triangles

$$\frac{w}{z-c} = \frac{d}{v}$$

As shown in Figure 8.2 (C), another two similar right triangles are formed by the penumbra limit and occluder corner, and the occluder corner and light source corner. Taking the right-half light source length as a, again by similar triangles

$$\frac{p-c}{w} = \frac{v}{a}$$

Now multiplying the two above equations,

$$\frac{p-c}{w}\frac{w}{z-c} = \frac{d}{v}\frac{v}{a}$$

$$\frac{p-c}{z-c}=\frac{d}{a}$$

Note that w and v cancel, leaving $\frac{d}{a}$ on the right hand side. This is visible length of the right half of the light source divided by the the length of the light source, or the fraction of the right half of the light source that is visible. Then for the outer penumbra, the total fraction l of the entire light source that is visible from the target point is

$$l=\frac{1}{2}+\frac{1}{2}\frac{p-c}{z-c} \tag{8.1}$$

Now consider the inner penumbra. By mirror symmetry the same analysis above still applies, except now $\frac{d}{a}$ gives the fraction of the light source that is not visible. This means in the inner penumbra,

$$l=\frac{1}{2}-\frac{1}{2}\frac{p-c}{z-c}$$

Kirsch and Döllner approximate c using the usual shadow map depth. This is approximately equal to c for flat surfaces. They store the shadow width w (in Figure 8.2). Because they only consider positive shadow widths, their method only supports inner penumbrae.

Chan and Durand store c as the smoothie depth, and store a positive value proportional to $-(p-c)$ as the "smoothie alpha". This method thus only supports outer penumbrae. Wyman and Hansen's penumbra maps use Z_{v_i} for c and Z_F for p.

To uniformly support inner and outer penumbrae, we can instead simply change our definition of p in the inner penumbra. We choose to define the signed distance $p_i=(p-c)$ in the inner penumbra, and $p_i=-(p-c)$ in the outer penumbra. Then both inner and outer penumbrae are exactly represented by

$$l=\frac{1}{2}+\frac{1}{2}\frac{p_i}{z-c} \tag{8.2}$$

Here we store and interpolate p_i and c in the shadow map, and compute z for the target pixel during shadow lookup. We also store the starting shadow depth d, where any points with $z>=d$ are treated as fully lit.

Our rather strange definition for the penumbra limit surface p_i has a number of benefits. First, it means we need not store or test whether each pixel is is in the inner or outer penumbra. Second, the surface p around one occluder line forms an inverted "V" centered on c. The sharp edge of the surface p is difficult to sample and interpolate properly. However, the surface p_i around an occluder is simply a plane—this makes interpolation in p_i exact under bilinear interpolation, and works well even at very low shadowmap resolutions. We use the usual technique of storing the signed value p_i as an unsigned value biased up by half its range.

The largest benefit of this definition is that the midpoint of the shadow, $l = \frac{1}{2}$, lies along the zero-crossing line of the plane p_i. This means unlike with other shadow representations, with penumbra limit shadows, the shadow midpoint can be located more precisely than a shadowmap or even screen pixel. This makes penumbra limit shadows very tolerant of low shadow map resolutions, and provides us the opportunity to perform high-quality shadow antialiasing.

8.1.3 Lighting Computation

Equation 8.2 above gives us the fraction of the light source that is visible. For a uniform Lambertian 1D line source, this is equal to the fraction of the light that is visible. But for a 2D Lambertian disk light source, this linear visibility measure isn't quite the same as the fraction of the source's light we can see. That is, covering up the first 10% of the diameter of the light source decreases the amount of incoming light by less than 6%, because the portion of the disk we begin covering is fairly narrow.

The fact that for a disk, visibility and illumination have a nonlinear relationship is well-known, but a large number of authors [CD03] [WH03] [BS02] [Hai01] have erroneously described the visibility-illumination relationship as "sinusoidal". Parker even gives an analytic expression for this proported sinusoid [PSS98], then approximates it with a cubic polynomial interpolant.

However, the actual visibility-illumination relationship is not a sinusoid. Figure 8.3 plots Parker's sinusoid, Parker's cubic interpolant, the infinitesimal approximation described below, the exact coverage, and a linear interpolation for a light source measuring 10 degrees across. The difference between the sinusoid and reality peaks at about 25% coverage, where the difference is just over 4.9% of absolute illumination.

For a large enough disk light source, the cosine factor for incoming illumination begins to play a role in the area calculation; or equivalently we should calculate the light areas on the hemisphere. By the time this effect is significant, however, the light source is normally so large that most soft-shadow schemes are already highly inaccurate.

Hence a useful tool is a 2D approximation, exact only for an infinitesimal light source, but in practice useful for real light sources. Consider a 2D disk of unit radius, clipped off by a straight vertical line as in Figure 8.4. We want to compute the area of this clipped disk, as it gives the visible area of the light source. To do this, we decompose the clipped disk into a wedge and a triangle. If a fraction t of the disk's diameter is

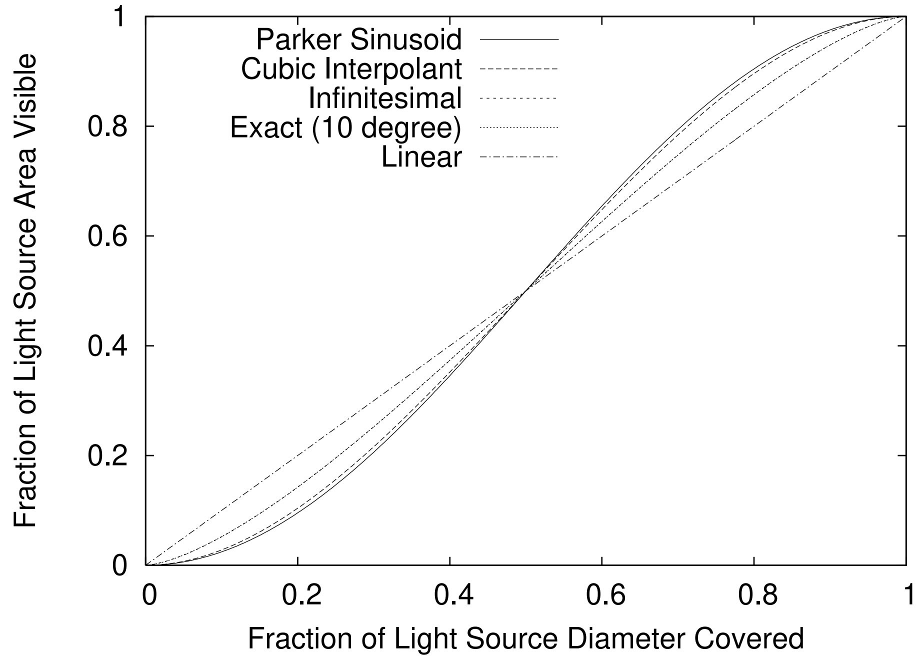


Figure 8.3. The amount of illumination coming from an infinitesimal disk light source occluded by a straight edge, using various published approximations, an infinitesimal approximation, and reality.

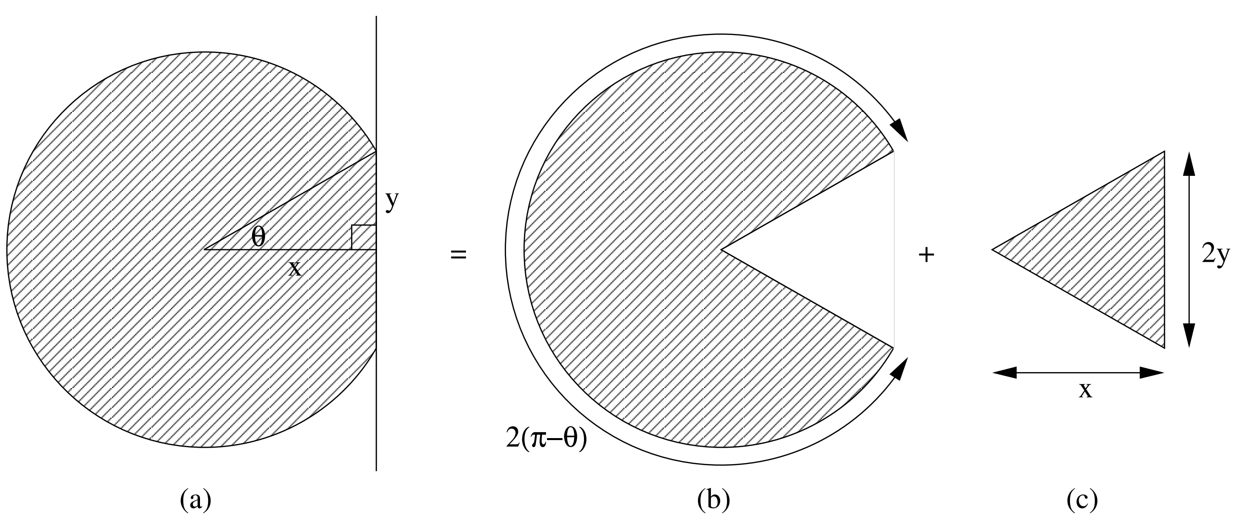


Figure 8.4. Computing the area of a partially obscured disk (a) as the sum of the area of a wedge (b) and an isosceles triangle (c).

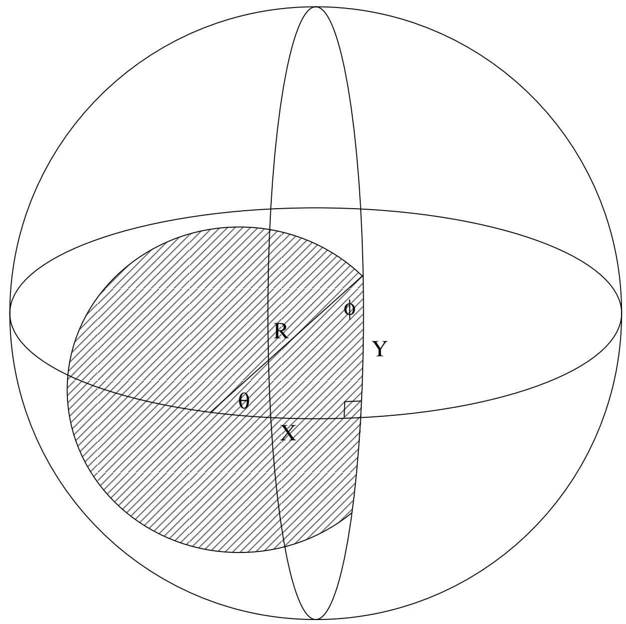


Figure 8.5. Computing the area of a partially obscured disk on the unit sphere.

still visible, then the signed distance from the disk's center to the nearest point on the occluder is $x = 2(t - \frac{1}{2})$. We can then construct a right triangle between this point, the disk center, and the last occluded point. One side of this triangle has length x, and the hypotenuse is the disk's unit radius, so by the Pythagorean Theorem the remaining side has length $y = \sqrt{1-x^2}$. The arc included by this triangle measures $\theta = \cos^{-1} x$, so the arc length of our wedge is $2(\pi - \theta)$ and hence the wedge's area is just $\pi - \theta$. The remainder of the clipped disk is an isosceles triangle that is simply our right triangle plus a mirror image, so the isosceles triangle has area xy.

Hence the area of the clipped disk in Figure 8.4 is just the area of the wedge plus the area of the isosceles triangle

$$\pi - \cos^{-1} x + x\sqrt{1-x^2}$$

And the fraction of the entire disk's area, which for a uniform light source is the fraction of non-occluded illumination, is then equal to

$$\frac{\pi - \cos^{-1} x + x\sqrt{1-x^2}}{\pi}$$

The irradiance arriving from a clipped disk source of uniform brightness is exactly equal to the radiance of the source times the visible area of the clipped disk in steradians. We can calculate this area on the hemisphere in exactly the same way we calculated the area in the plane: we again decompose the clipped disk into a wedge and triangle, and compute the areas in the same way, but now using the area and length relations of spherical trigonometry[SG77] instead of plane geometry. Begin with a disk of radius R, measured in radians. The distance from the center of the disk to the occluding plane if a fraction t of the diameter is visible is $X = 2R(1-t)$. The Spherical Pythagorean Theorem then gives us $\cos Y = \frac{\cos R}{\cos X}$. Because R is across from a right angle, the interior angles of our right triangle can be derived from the Spherical Sine Theorem as $\sin\theta = \frac{\sin Y}{\sin R}$ and $\sin\phi = \frac{\sin X}{\sin R}$. The angular extent of our wedge is again $2(\pi-\theta)$, so the solid angle area of our wedge is then $2(\pi-\theta)(1-\cos R)$ steradians. Via Girard's Theorem the area of the remaining isosceles triangle is $2(\theta+\phi+\frac{\pi}{2}-\pi)$ steradians. The total area of the clipped disk is thus $2((\pi-\theta)(1-\cos R)+(\theta+\phi-\frac{\pi}{2}))$ steradians. In computing these relationships, one can avoid 2π periodicity problems in computing the inverse sine for negative X by always calculating using a positive X, and for negative X inverting the resulting area by subtracting it from the area of the whole disk, $2\pi(1-\cos R)$ steradians.

For a light source that measures 10 degrees across or smaller, the infinitesimal approximation presented above differs from reality by less than 0.02% absolute illumination. However, as Haines points out [Hai01], the visibility-illumination relationship can trivially be precalculated and stored in a lookup table, so the efficiency of this computation is not particularly important. The visibility-illumination lookup table could even store the light from an arbitrarily complicated radially symmetric light source, which would allow us to include the sun's corona and some of the sky's light, or the fuzzy shape of the sun shining through a thin cloud layer.

In the images presented here, we use the infinitesimal light source approximation to a uniform disk, stored into a 256-entry lookup table.

8.1.4 Supporting Arbitrary Surfaces

The preceding sections describe how to apply penumbra limit shadows along a single edge. But in reality, we must compute shadows for more complicated geometry. Because complicated geometry can create extremely complicated shadows, we cannot hope to efficiently compute physically exact shadows, except perhaps along straight edges. As with any fixed-complexity system, overlapping soft shadows are only approximated; and depending on how the shadows overlap the system may create or destroy radiant energy as the light propagates.

Kirsch and Döllner support arbitrary geometry by first rendering a depth image, then extruding the penumbra interior from the discontinuities in the depth image. This has the disadvantage that extruding the depth image is difficult to perform efficiently on graphics hardware, although Kirsch and Döllner present a clever algorithm based on repeated readback to compute this reasonably quickly. Chan and Durand instead extract silhouettes from the geometry, then render these silhouettes directly to the smoothie buffer. This has the disadvantage that the silhouette complexity can be quite high, but is easy to perform (for Smoothies or penumbra limit shadows) on the GPU.

In our implementation, we chose to compute penumbra limit shadows from a single depth buffer, because the silhouette complexity of foliage can be extremely high. In principle, propagating soft shadow boundaries across the shadow map is relatively easy. We treat each depth discontinuity in the shadow map as a shadow-casting edge, and propagate that edge's cone of shadow outwards across the shadow map, decreasing in height as it propagates.

The first difficulty we encounter is that when the soft shadows cast by two different edges meet, we cannot in general exactly represent the combined shadow, because combined shadows can have arbitrarily high spatial complexity but our shadow map has fixed complexity. Hence we must throw away some of the shadow detail in order to project the true shadow shape onto the space of functions we can represent. One principled approach to this projection is to calculate the true shadow surface (e.g., via sampling) and then project to the representable space; a good example of this sort of approach are Deep Shadow Maps, by Lokovic and Veach [LV00]. However, computing the true shadow surface for our distributed light sources is too slow to perform online.

Instead, we use the simple technique of combining separate shadows according to the heuristic "highest limit surface wins." That is, given the large class of soft shadows covering a shadow map pixel, we take the uppermost shadow and ignore all deeper shadows. This is not physically correct in general, but does give the physically correct answer for simple 2D halfplane occluders, which is the most we can hope for, given our simple shadow representation.

Hence, shadowmap propagation amounts to starting a shadow cone from each discretized depth discontinuity edge, and keeping, at each pixel, the highest shadow cone. A trivial implementation of this approach would simply check all possible interactions between e depth discontinuities and a pxp grid of pixels, which would run in time $O(ep^2)$. But there can be many depth discontinuities in the worst case, so this running time is too slow: $O(p^4)$ if there are as many depth discontinuities as pixels, which can be the case for foliage.

To speed up shadow propagation, we transform it into an incremental per-pixel operation. Note first that a cone, including the cone of shadow extending from a depth discontinuity, can be computed incrementally around a single point by iterating the operation “take max of my neighbors, minus the cone slope.” On a discretized grid, each iteration requires one pass through all the pixels, and hence takes $O(p^2)$ work. But in the worst case, we would need up to p iterations for a cone starting at one edge of the grid to propagate to the other side. We could thus propagate shadows by initializing the shadow heights at each depth discontinuity and iterating p times, and this indeed drops our run time to $O(p^3)$. But it is possible to do still better. Kirsch and Döllner present a GPU algorithm that iterates while comparing with successively larger “neighborhoods;” this algorithm computes the shadowmap in $O(p^2 \lg p)$.

In fact, we do even better yet using a sequential “sweep” algorithm to propagate shadows. Figure 8.6 shows how the sweep algorithm functions. In the first pass, shown on the left, we scan through the shadow map in raster order starting at the top left, calculating the effect on each pixel from all its “incoming neighbors”: its higher neighbors, and all its neighbors to the left in its row. We define a pixel as “complete” if all its incoming neighbors are complete, and it includes the effect of those neighbors. The figure shows the complete pixels a certain point in the computation using shading.

By induction, we start with a complete pixel, and calculate the effect of that pixel on all its outgoing neighbors. This makes the next pixel complete, and we then repeat. We start the induction at the top left pixel, which has no incoming neighbors and is hence complete, and proceed in raster order through the grid. After one top-to-bottom pass, every pixel includes the soft shadows cast by edges above and to the left of it. We then flip the definition of incoming and outgoing, and make one bottom-to-top pass, as shown on the right in Figure 8.6. This “sweep” algorithm thus propagates soft shadows across the shadow map via two iterations over the pixels, using $O(p^2)$ time, which is optimal.

8.1.5 Examples

A simple scene containing a tree and a single polygon is shown in Figure 8.7. The corresponding penumbra limit maps are shown in Figure 8.8.

A more complicated scene is shown in Figure 8.9. Note that this scene includes soft shadows cast by tree leaves (rendered using particles), tree trunks (rendered as raytraced cones), billboard streetlights, and building polygons. Penumbra limit shadows can properly expand soft shadows from any geometry that can be rendered into a depth image.

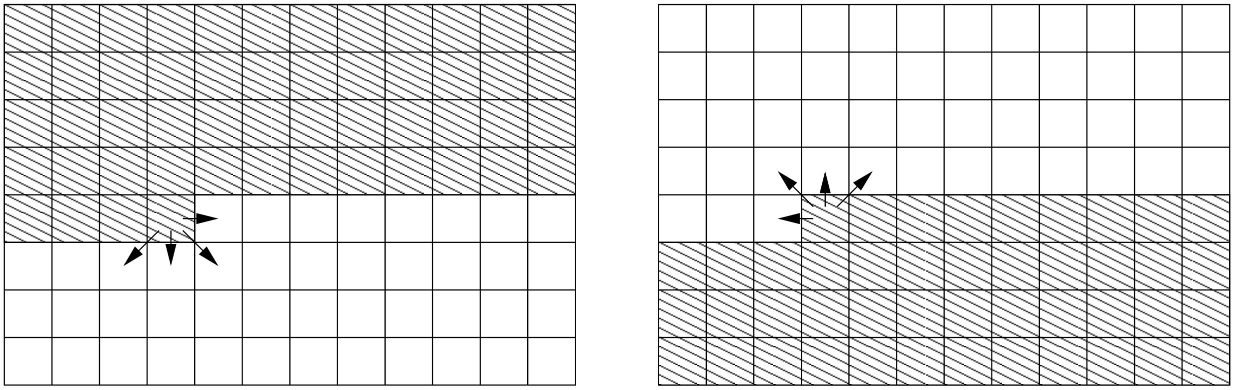


Figure 8.6. Illustrating the sweep algorithm to propagate soft shadows across the shadow map. Shaded region illustrates completed pixels at one step; arrows indicate direction of information flow, from incoming to outgoing, during the sweep.

8.1.6 Soft Shadow Comparison

Figure 8.10 compares shadow images computed using a variety of techniques. In the top row, we examine the shadow cast on the ground plane; the bottom row shows the difference between this image and the exact soft shadow. The model is a single triangle skewed up from the ground, illuminated by an infinitely distance circular light source. The top right triangle corner touches the ground plane, the top left corner is raised slightly, and the bottom corner is raised far off the surface. The shadow map resolution is just 32 x 32 pixels; and each output image is 512 x 512 pixels.

The exact, physical soft shadow is shown at the extreme left. This image was computed via Monte Carlo integration, also known as distribution raytracing, taking 4,000 sample rays per pixel to cover the area light source. With this huge number of samples, the image took approximately 10 minutes to produce, but all the other shadow methods can be calculated at framerate even in software. Hence exact shadows, at least calculated using random sampling, are far too slow to actually use.

Penumbra limit shadows are shown next, and display a few minor problems. At the large blurry model corner, penumbra limit shadows are a bit darker than the exact image. Because of the low shadowmap resolution, penumbra limit shadows cannot capture the sharp corner at the top left. But as claimed, up to depth quantization error, penumbra limit shadows are exact along straight occluder edges. Because penumbra limit shadows are so interpolation friendly, as shown on the left half of Figure 8.11, the impact of the low shadowmap resolution is difficult to even discern.

Compare this situation with that of Smoothies [CD03], which are compared to only the outer penumbra of the perfect image (computed by scaling and shifting the output light). The smoothie depth and smoothie

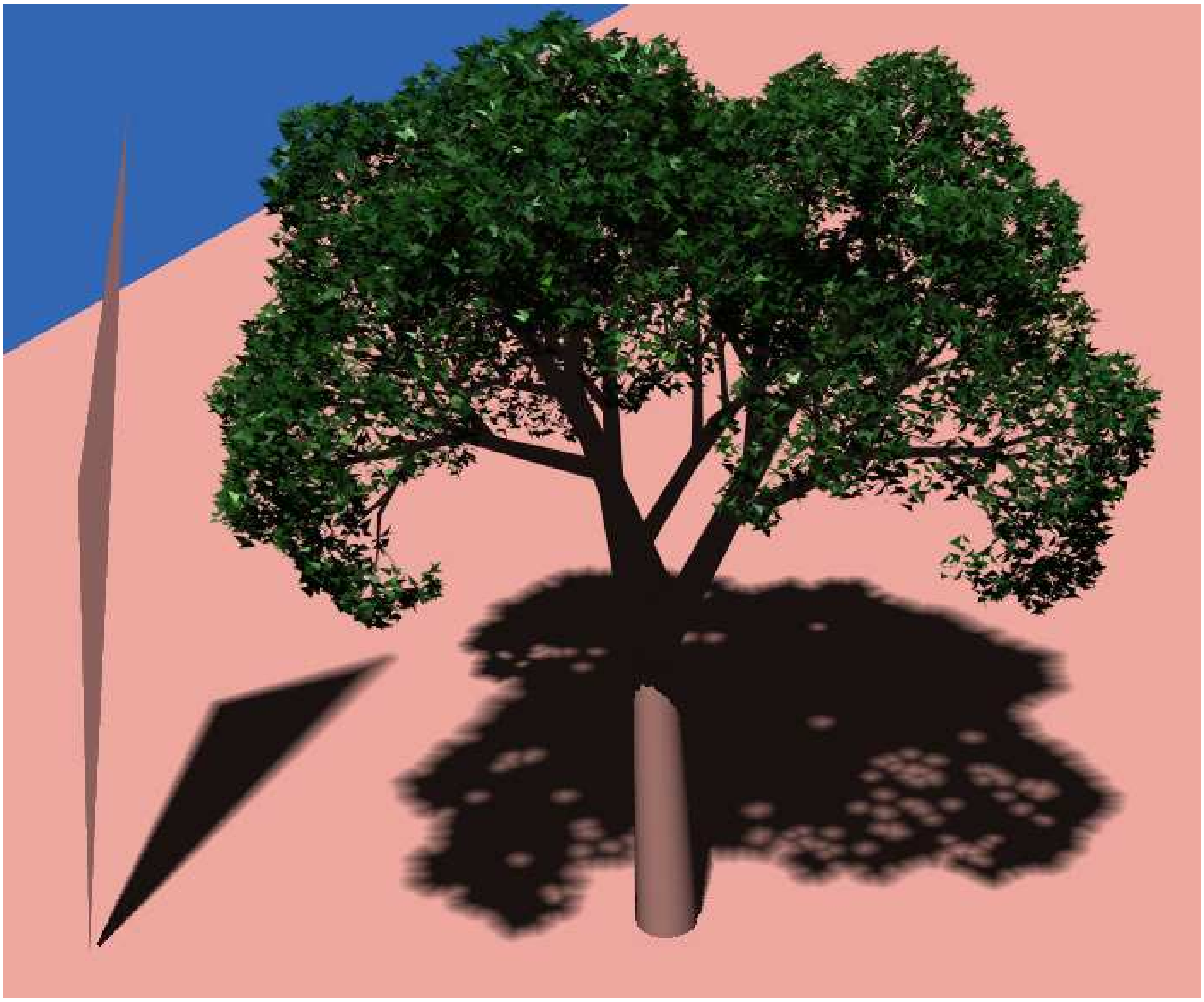


Figure 8.7. A scene with soft shadows rendered via the penumbra limit map technique. Light is cast by an extended light source offscreen to the left, which has an angular size of three degrees. The light is occluded by a small polygon standing on its tip (left) and a tree (right). Note how the polygon's shadow gets softer as it travels. The shadow map resolution is 1024x1024.

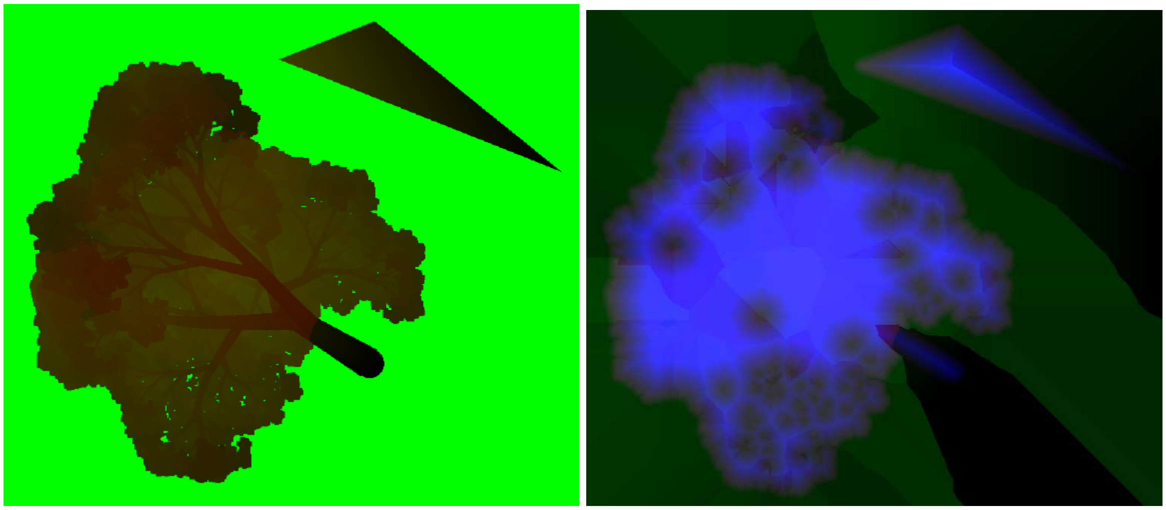


Figure 8.8. Portions of the actual penumbra map image. Left is the original depth map, with the red channel showing the closest geometry and the green channel showing the farthest, where darker values indicate farther depths from the light. Right is the actual penumbra limit map—red is the depth to the first portion of shadow, green is the depth to the active occluder c, and blue is the signed penumbra limit depth p.

alpha from which Smoothies computes soft shadows always contain discontinuities along the model and shadow edges, as shown in the center section of Figure 8.11. These discontinuities would lead to interpolation artifacts if the depths themselves were interpolated. Because of this, the soft shadow computation for Smoothies is performed only at shadowmap pixel centers, and the resulting lighting is interpolated across to the actual sample point. This is entirely analogous to Reeves' filtering, described in detail below. The primary disadvantage of this approach is that because the lighting is only computed at shadowmap pixel centers, these pixels show up prominently in the resulting shadows, as can be seen in the figure as stair stepping along the top edge of the polygon.

The single-sample technique of Kirsch and Döllner [KD03] interpolates shadow depths rather than shadowed intensities, and are hence not directly limited by the resolution of the shadow map. In addition to only covering the inner penumbra, this technique suffers from interpolation artifacts along the shadow boundary. The situation is shown on the right half of Figure 8.11. Because the shadow depth (used as the denominator of their intensity equation) drops to zero along a shadow boundary, any nonzero shadow width will be amplified into a thin dark ridge. Because the shadow width is discretely sampled, it will not always reach zero before the shadow depth, resulting in annoying dark ridges around the shadow, which are clearly visible in Figure 8.10.

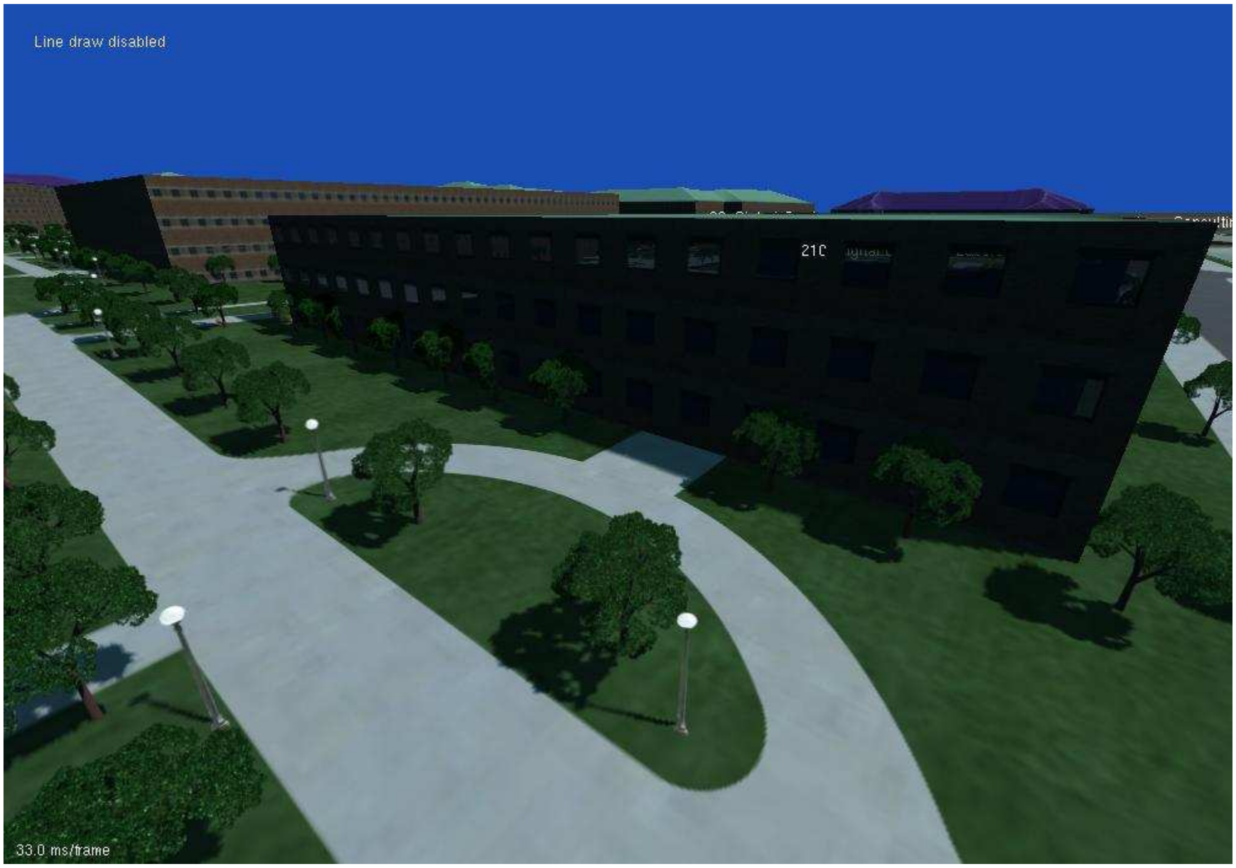
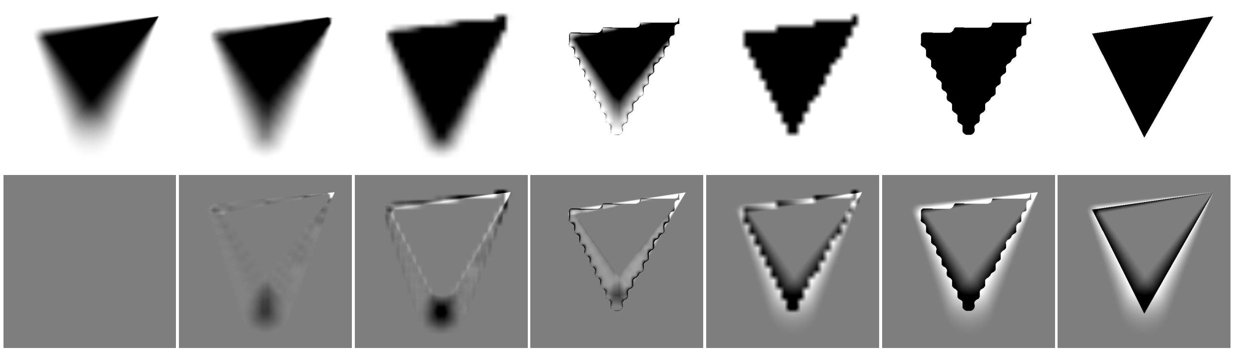


Figure 8.9. Campus scene rendered with soft shadows via the penumbra limit shadow map.



Exact Penumbra Limit Smoothies Kirsch & Doellner Reeves Filtering Hard Shadow Map Shadow Volumes

Figure 8.10. Soft shadows computed using a variety of techniques. The shadow map resolution is just 32 x 32.

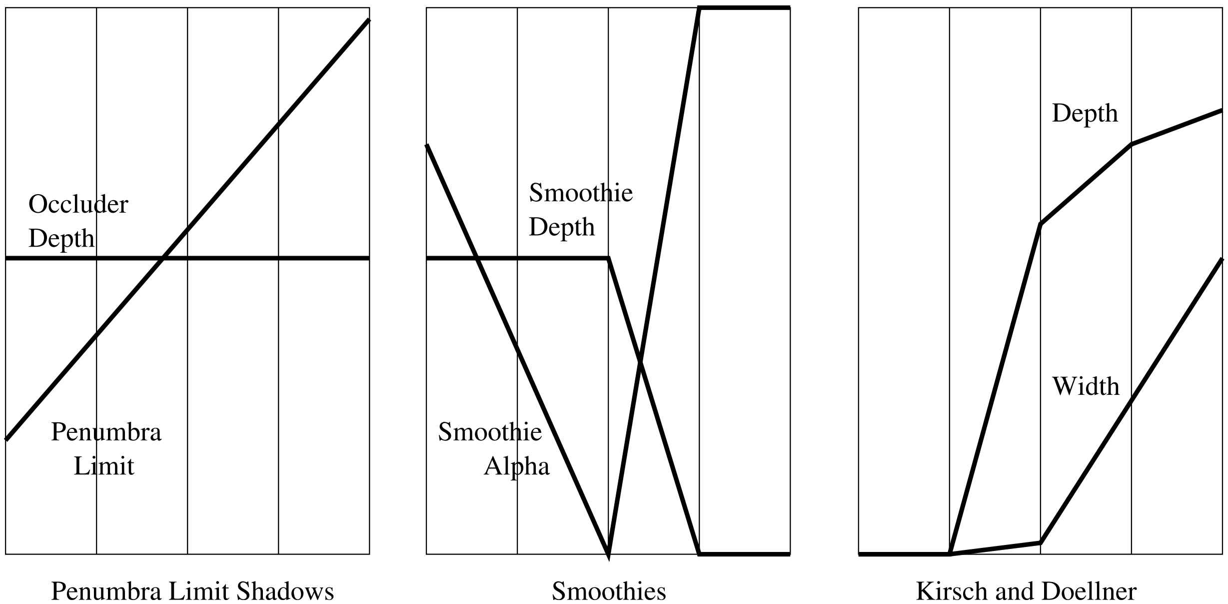


Figure 8.11. Different values (vertical axis) for five shadow map pixels (horizontal axis) centered on an object edge, for three different soft shadow map techniques. An occluder ends in the center pixel, with the outer penumbra to the left and the inner penumbra to the right.

Reeves' "percentage closer" shadow filtering [RSC87] simply performs shadow lighting at the exact shadow map pixels, then interpolates the resulting shadows. Compared to simple hard shadow maps, where the depth is interpolated before being shadowed, Reeves' technique results in a smoother image. However, the shadow sampling grid is still clearly visible, and softness of the shadow is entirely a result of sampling and has no relationship with the actual physical penumbra.

Finally, on the extreme right we show ordinary shadow volumes. Shadow volumes are not subject to the sampling problems normally associated with shadow maps. However, it is difficult to efficiently make shadow volumes work for soft shadows, and the performance of shadow volumes is poor for scenes with high silhouette complexity.

Overall, for a given shadow map resolution, penumbra limit maps provide substantially more accurate shadows than previously presented methods, and provide this accuracy at affordable cost.

8.1.7 Parallel Direct Lighting

Because the scene is largely static, we could precompute coarse versions of the lighting, including a coarse penumbra limit map. But because the scene is large, we cannot possibly precompute everything, and must dynamically generate appropriately filtered higher-resolution lighting at runtime. Hence before rendering an object, we first compute and propagate a shadow map of appropriate coverage and resolution for that object.

It is not quite as easy to compute a small piece of a penumbra limit map as a conventional shadow map, because soft shadows propagate across the map. However, because the scene has a known depth range, it is easy to bound the distance a soft shadow can expand. If we then enlarge our depth image by this distance, we can use the enlarged depths to compute an exact penumbra limit map over the region of interest.

For a relatively small light source like the sun, a soft shadow expands by less than a half meter over 50m of height, so this shadow expansion normally amounts to only a few shadowmap pixels. This means shadow maps can be computed in parallel with little wasted work computing overlapping depth images.

8.2 Impostor Global Illumination

Indirect or "global" illumination is all light that does not arrive directly from a pointlike light source. For example, sunlight is direct illumination; while sky light, and light reflected off buildings, trees, and the ground are all indirect illumination. Because of the all-pairs nature of global illumination (any two objects in the scene can exchange light), the effects of indirect illumination are difficult to compute efficiently, and are hence often ignored.

We will compute sky light and other indirect illumination using a coarse uniform *voxel/cubemap lighting grid* spread throughout the scene. A voxel/cubemap lighting grid captures the incoming illumination from a discrete set of directions at a discrete set of locations. This representation can be visualized as a regular 3d (voxel) grid of space locations, with cubemaps capturing the incoming illumination in all directions at each location, as illustrated in Figure 8.12.

For analysis, we take the voxel grid as a cube discretized into v voxels along each side, or $V = v^3$ voxels total; while each cubemap face is discretized into d directions, for $D = 6d^2$ direction samples per cubemap. Hence this representation uses three spatial and two angular directions, for a total of $VD = 6v^3d^2$ illumination samples, which is a full five-dimensional radiance field. We take as E the total number of distinct light reflecting elements, which for accuracy are often small patches of individual objects.

Clearly, many scenes will require thousands of voxels to adequately represent the subtle variation of light across space—that is, many scenes need high spatial resolution. However, we may need only a few dozen direction samples to adequately capture the important variations of light with direction at a particular point.

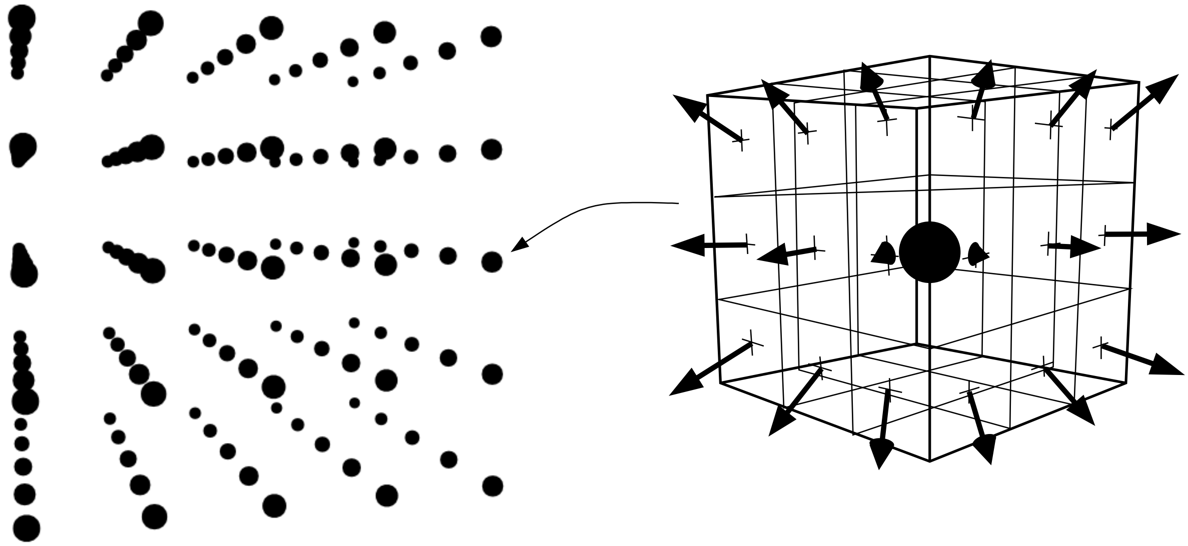


Figure 8.12. Voxel/cubemap lighting grid. We keep a regular array of locations, shown here on the left as a 5 x 5 x 5 voxel grid. At each voxel, we capture the incoming illumination using a cubemap, shown on the right at 3 x 3 (pixels per face) resolution.

This required angular resolution is especially low for mostly diffuse objects, which blur away any higher-frequency angular changes. Low directional complexity is one reason why low-order spherical harmonics are often used to represent incoming and outgoing light.

8.2.1 Prior Work

The classic implementation of global illumination is the transport matrix, as formulated by Cohen and Greenberg [CG85]. For each pair of elements, fill one entry of the "transport matrix" with the element-element "form factor", the fraction of radiant energy leaving the first element that arrives on the second element, which is a function of the element geometry and intervening occluders. Solving for a global illumination solution is thus simply a matter of solving a matrix fixed-point equation involving the transport matrix, element reflectance, and incoming illumination.

The main problem with this approach, like many global illumination methods, is speed. The transport matrix has E^2 elements, and would take an absurd $O(E^3)$ time to solve outright. Cohen and Greenberg use an iterative method to find light from the first k bounces by iterating k times, for a cost of $O(kE^2)$. Using sparse matrix techniques or multigrid methods can cut the time slightly. Other authors have cast the usual transport matrix formulation of radiosity onto the GPU, such as Carr et al. [CHH03], although still with poor overall performance.

The main advance in recent years in global illumination has been the development of discrete sampling-based methods. These include Jensen's Photon Maps [Jen96] and Keller's Instant Radiosity [Kel97], which both trace out discrete light rays starting from the light sources and accumulating on surfaces. This technique was recently implemented on the GPU by Larsen et al [LC04]. These techniques are quite useful, but achieving a good sample distribution is difficult, especially for large, open environments.

In physical radiation transport, a well-known technique is to discretize space, S, and directions, N, in a fashion similar to the voxel/cubemap method often called the discrete-ordinates method or simply SN transport. This technique is widely used in radiation transport, though often on unstructured grids; for a typical parallel implementation see the Sweep3d program of Plimpton et al. [PHBI00]. The method we present here can be seen as an efficient implementation of this sweep-style radiosity computation.

The voxel/cubemap representation of light can be seen as a generalization of Levoy and Hanrahan's Lightfield [LH96]. A lightfield is a 4D radiance function, parameterized on two location axes (typically a point in an incoming grid) and two direction axes (typically a point on an outgoing grid). Lightfields do not capture occlusion, and are hence not useful for representing or calculating the light near occluding objects.

The voxel/cubemap representation of light we use is identical to Greger's irradiance volume [GSHG98]. Greger filled his irradiance volume by taking independent raytraced illumination samples, so his technique was quite slow. Nijasure [NPG] showed this same voxel/cubemap approach can also be implemented on the GPU.

Unfortunately, rendering low-resolution cubemaps at many different voxels is inefficient, because all the scene geometry must be reprojected for all V voxels' viewpoints, so the per-vertex geometry cost dominates rendering. Hence for example Nijasure [NPG] used only a $V = 64$ voxel grid, which is only enough to capture the spatial variation of the most trivial scenes. However, the cubemaps in this case were $d = 16$ pixels across ($D = 1536$ directions), but they were then projected to 9 spherical harmonic coefficients, so the effective angular resolution is still quite low.

8.2.2 Impostor Global Illumination Technique

Though our representation is the same, our method for calculating a voxel/cubemap lighting grid is different from previously described work. Instead of calculating a cubemap for the light from all directions for one voxel, then moving to next voxel, we calculate the light for one group of directions for all voxels, then move to the next group of directions.

Consider the light arriving at one plane of voxels from a particular direction. This plane of light can be captured by imaging with an orthogonal camera, with the camera focal plane oriented along the plane of voxels. Because we can project the geometry for an entire plane of voxels at once, the number of times the geometry needs to be projected with this method is just vD. For a typical useful $v = 100$ and $D = 25$, the geometry needs to be projected only $vD = 2,500$ times, versus at least $V = v^3 = 1,000,000$ projections for previous voxel/cubemap methods.

In fact, we can render the voxels even more efficiently. Rather than starting from scratch to render each plane of voxels, we can reuse the samples computed on the preceding plane. For a given direction, the illumination samples for the voxels in one plane differ from the voxels in the previous plane for only two reasons. First, there is a bulk 3D offset as light propagates between the planes; for a single light direction, this can be computed as a simple 2D image shift. Second, geometry between the planes has affected the incoming light; this can be computed by rendering the small slice of geometry that lies between the voxel planes. Because rendering small slices means we can avoid repeating much of the geometric projection work for each plane, we thus only require D separate geometric projections. Because D is normally thousands of times smaller than V, this approach saves substantial amounts of work.

Because we replace all the geometry from previous planes with an image, we call this method impostor global illumination. The biggest advantage of this technique is that it replaces the large amounts of per-vertex work, in projecting all the geometry to all the cubemaps, with large amounts of per-pixel work, most of which involves resampling relatively large per-plane texture images. Because the geometry is only used to render slices, this technique is easy to implement, and trivially supports textured surfaces and arbitrary BRDF's. Because the technique traverses all of space, it is easy to include subsurface scattering effects or volume BRDF's, such as participating media.

The downside of this approach is that it only directly supports a regular grid of location and direction samples. Most scenes have high variation in the sampling detail required, and in particular many scenes only require illumination samples at surfaces, not everywhere in space. However, it would be possible to implement this scheme at multiple resolutions, to better represent light near surfaces. The fixed and generally low direction resolution of our approach makes it difficult to represent highly coherent light, such

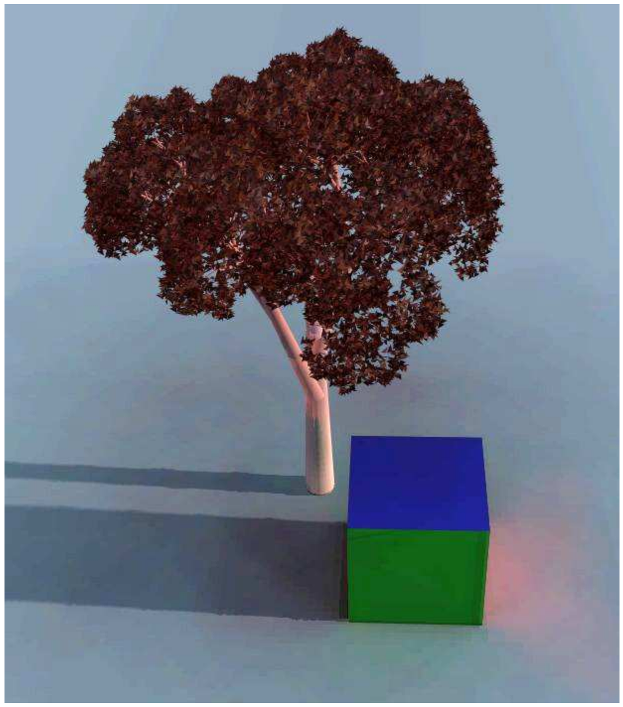


Figure 8.13. Tree scene with global illumination computed via Impostor Global Illumination.

as sunlight (which is better handled using the shadow map above) or caustics (which are better handled using photon maps).

An example of this technique used to compute global illumination is shown in Figure 8.13. Sunlight streams in from the left; sky light from above. Note the red light reflecting off the front of the cube, and the subtle darkening around the bases of the tree and cube. The voxel spatial resolution for this image is $v = 128$ voxels per side, or $V = 2097152$ voxels total; direction resolution is $d = 3$ directions per cubemap side, or $D = 26$ directions total. An unoptimized software implementation propagated the light for this image in approximately six seconds.

An example of a set of per-direction images for several planes of this scene's voxels is shown in Figure 8.14. Each individual image shows the appearance of the world from a particular direction at a particular plane of voxels. Eight directions are shown, arranged vertically; for eleven different planes of voxels, arranged horizontally.

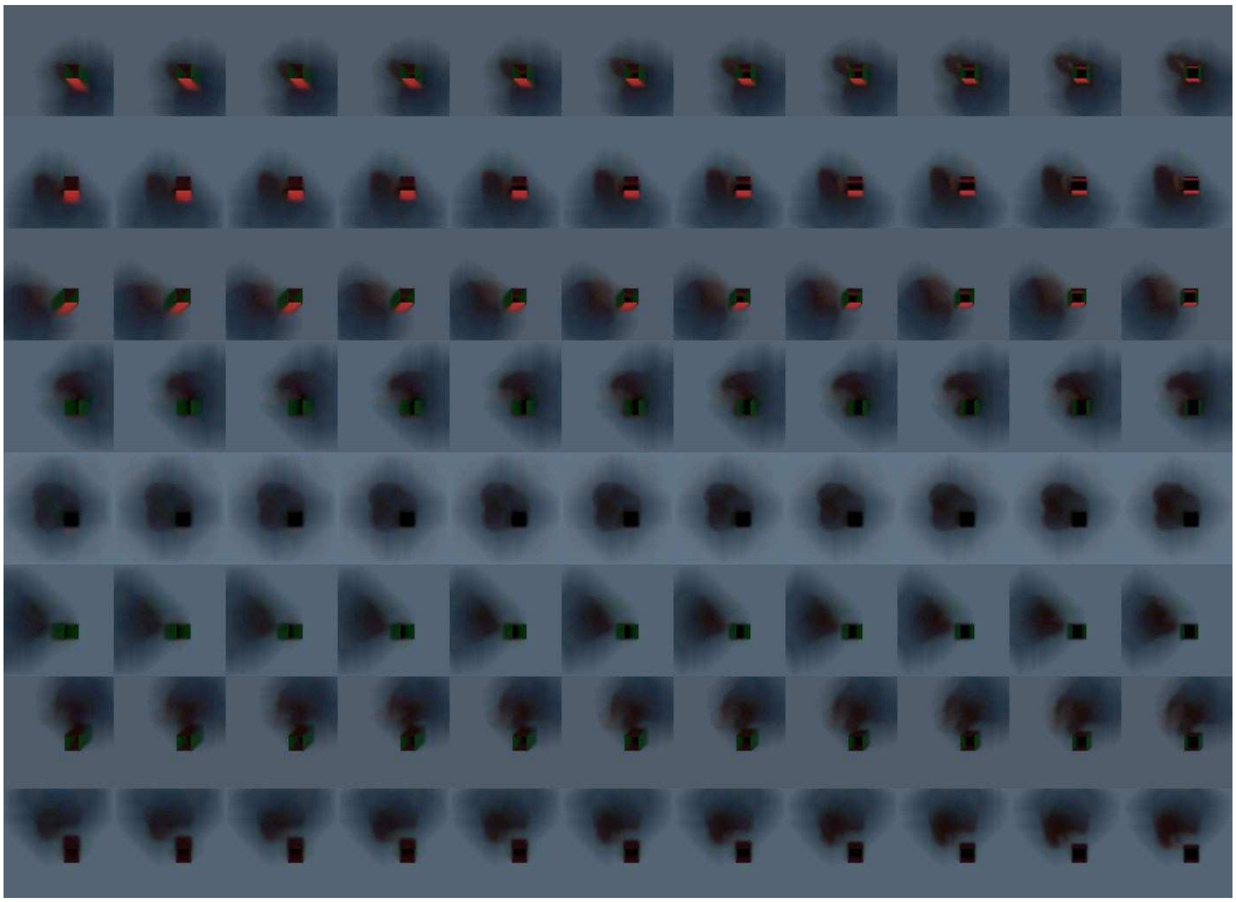


Figure 8.14. Subset of the radiance images for various directions (rows) along various voxel planes (columns) computed during one Impostor Global Illumination pass. The sweep plane for this example is horizontal; the first, highest-Z planes are shown on the right; the last, lowest-Z planes are shown on the left. This slice passes through the cube above; the sky-shadow of the tree is visible as a dark background blur. Note the bright red showing up as we pass through the lit face of the box, visible from the top three rows.

8.2.3 Impostor Global Illumination Filtering

The technique described so far is simply a more efficient method to compute the usual voxel/cubemap lighting grid. But we would like to do better than the usual cubemap approach. Particularly for low cubemap resolutions, the projection to cubemap pixels can alias the incoming geometry and miss bright, important features. The problem is that an ordinary camera treats each direction as entirely discrete, a directional impulse; instead of as a smooth angular function, a directional basis function with some angular and hence location spread. Using a smooth directional basis corresponds to depth-of-field filtering on the voxel camera, or to performing a distance-dependent geometry blur.

An inexpensive approximation to this directional filtering is to slightly blur the previous voxel plane's image. Then light which has propagated in from far away will have been blurred many times and hence be smeared out in space; or, equivalently, geometry's outgoing light spreads out more and more as it propagates.

Because the only operations in this technique are rendering slices of geometry to large textures, and shifting and blurring textures, this technique can be implemented very efficiently in the GPU.

8.2.4 Impostor Global Illumination Plausibility

Impostor global illumination matches the true behavior of light in a number of ways.

- Radiant energy is neither created nor destroyed, except by emission and absorption by geometry.
- The response of the system to an input of radiant energy is linear and translation invariant (up to voxel resolution).

However, there are a number of ways in which impostor global illumination does not match the true behavior of light.

- In a vacuum, light traveling in given direction continues in that direction forever. In our system, eventually light in a given direction will blur out with neighboring directions.
- Light's impulse response is perfectly rotationally invariant. But because we discretize angles, the response of our system is somewhat lumpy in both space and directions. Luckily, this is normally difficult to detect.

Figure 8.15. Campus illuminated by sky light (top), sun light only (middle), and sun and sky light (bottom). The top and bottom images are calibrated photographs acquired with a digital camera; the sun-only image is computed from these images by subtracting the sky light image from the sun and sky light image.

8.3 Lighting Summary

Outdoor scenes are lit primarily by the sun, but sky light and indirect illumination also play an important role. As shown in the middle photo in Figure 8.15, without sky light, areas in shadow appear unnaturally black. We have presented two techniques, penumbra limit shadows, and impostor global illumination, that are capable of reproducing both the direct and indirect lighting in this scene.

Chapter 9

Large Model Support

This chapter describes a very large model to be rendered using parallel impostors. This extends the final axis of the parallel impostors benefit space: large environments. Normally large, detailed models are difficult to render, because the large quantities of highly detailed geometry result in a high per-triangle cost. Impostors are very useful for rendering large environments because the impostor reuse rate for distant geometry is high (as shown in Figure 2.5). This means the cost of rendering distant geometric detail can be amortized over many frames, thus accelerating overall system throughput. Parallelism can provide further acceleration, by having multiple processors expand geometric detail simultaneously.

Our motivating example is an extremely large, detailed model of the campus of the University of Illinois at Urbana-Champaign. This is a useful environment to analyze, because it includes a variety of natural and artificial objects, so our impostor technique can be examined in a real, unbiased environment. An immense quantity of machine-readable data is already available for the campus, and can be readily assembled into a large model.

We will focus on building and viewing the campus model from outside the buildings and at ground level. The basic representation for the model is an elevation image, or *height map* decorated with trees, buildings, and other geometry. Because the height map, trees, and buildings are all represented recursively, the overall scene graph we describe is a recursive tree.

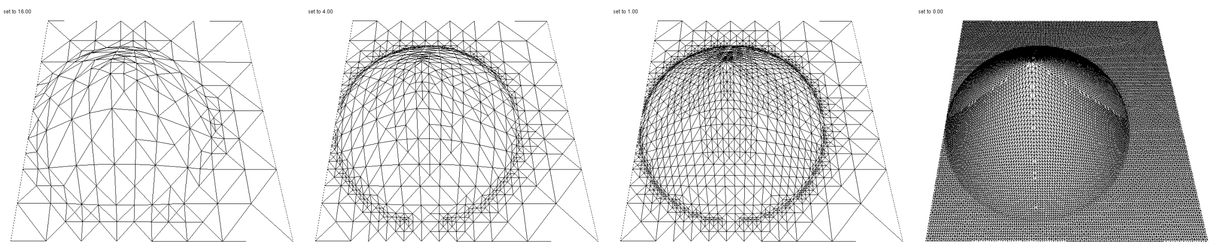


Figure 9.1. Mesh for terrain renderings with different screen-space error criteria: 16 pixels, 4 pixels, 1 pixel, 0 pixels.

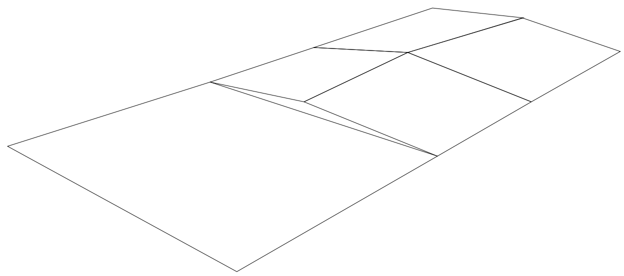


Figure 9.2. When a quad that is not expanded (left), is adjacent to a quad that is expanded (right), the non-conforming nodes from the expanded quad may cause a visible crack or gap.

9.1 Height Maps

The height map representation we use is derived from Lindstrom and Pascucci's terrain simplification method [LP02]. This method displays the terrain by recursively expanding a spatial tree, terminating the expansion when the screen space error drops below a threshold, as shown in Figure 9.1. One advantage of this recursive expansion is that we can perform very efficient view culling by simply not expanding those subtrees that lie completely offscreen.

The mesh representation is thus a perfectly ordinary expand-on-demand quadtree. The well-known problem with quadtrees, however, is that when a quad is expanded but the adjacent quad is not expanded, a crack can appear between the two quads, as shown in Figure 9.2. Lindstrom and Pascucci's contribution is a simple and efficient method to avoid cracks. To avoid cracks, this method makes the expansion decision at each vertex, not each quad. Because vertices are shared between adjacent quads, this ensures all adjacent quads have identical refinements along their common edge.

Figure 9.3 shows how we choose to expand the geometry of a single quad, based on the expansion decisions at vertices—black for expansion and white for no expansion. By induction, the corner vertices are always black. If the center vertex is black, we must connect it to all the corners and all black side vertices. If all four corner vertices of a sub-quad need to be expanded, then we recursively expand that subquad.

We choose our expansion criterion to guarantee that if the center vertex is white, and does not need to be expanded, then we do not need to expand any vertex of the quad. We guarantee this by propagating error bounds from edge vertices to center vertices in exactly the same way as Lindstrom and Pascucci. This allows the expansion search as soon as we hit a white center vertex; without this, we would have to check every possible vertex in the graph.

One advantage our explicit quad-based approach is that we can decompose the quads into triangles along either diagonal, rather than decomposing them in a fixed way as in Lindstrom and Pascucci. We always split the quads along the axis that better represents the geometry. The benefit of this approach is shown on the right half of Figure 9.4, where our quad-based mesh conforms to the curvature of the terrain. The lower error of our approach satisfies the same screen error bound while creating about 10

9.2 Depth Traversal

As described in Section 3.1, to support antialiasing we do not primarily use the Z-buffer to resolve depth. Instead, we use the simple "render from back to front" painter's algorithm. We have developed a novel technique for traversing a height field in strict depth order from an arbitrary viewpoint.

To work with the painter's algorithm, we must draw the terrain geometry in "back-to-front" order. The simplest way to do this would be to first generate all the geometry, then perform a giant depth sort, then drawn all the geometry in depth order. This global sort would be inefficient, because the generated geometry is large and sorting is slow.

A global sort is also unnecessary, because the painter's algorithm only requires geometry to be drawn in locally back-to-front order—that is, each primitive should only be drawn after any deeper *overlapping* primitive. This means we can afford to expand the geometry in some arbitrary order, such as an order that respects the model's hierarchical structure, and still guarantee the painter's algorithm works.

We perform our depth sort as we recursively traverse the terrain quadtree, by carefully choosing the order in which to traverse the child nodes and hence expand their geometry. By processing child nodes in back-to-front order, we extract their geometry in a local depth order. Because depth is not a global property, but depends on the camera location, we must expand the terrain in a camera-location dependent order. In

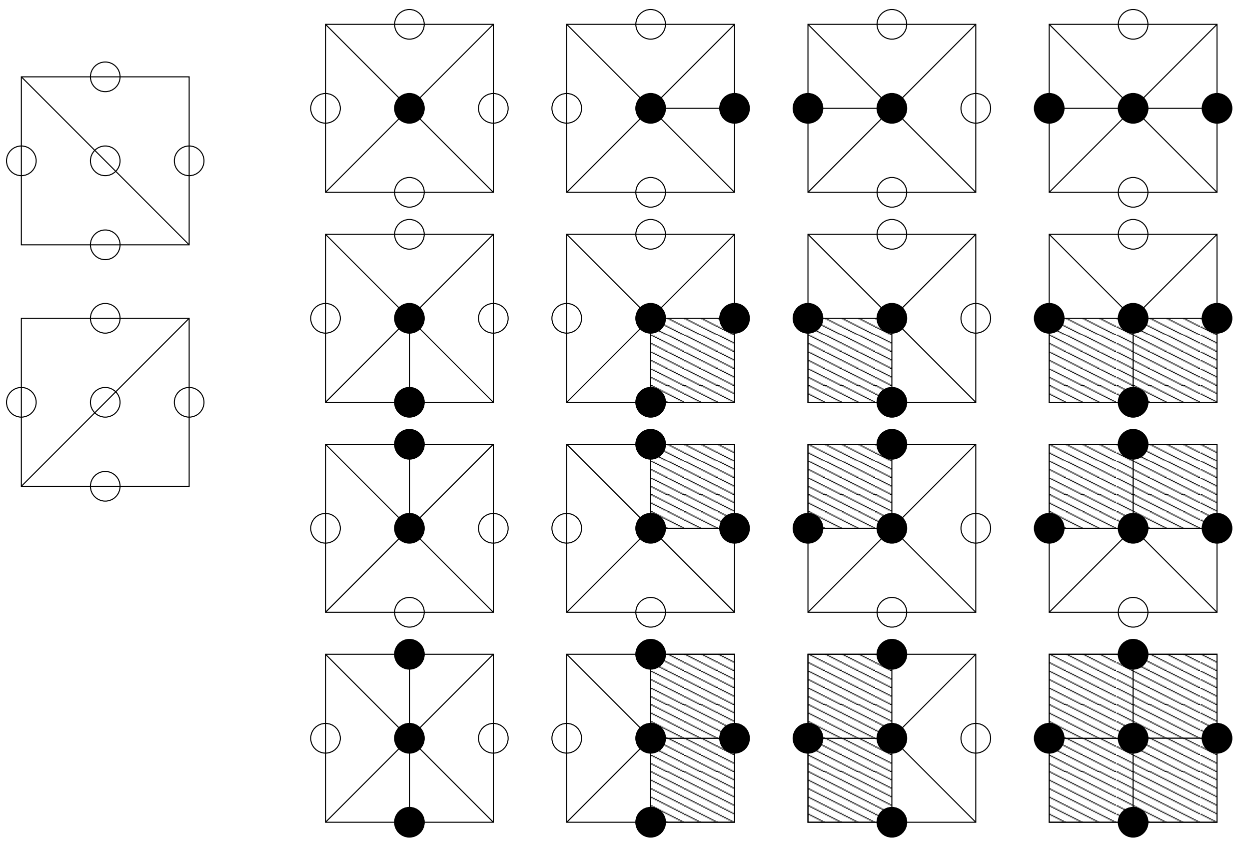


Figure 9.3. All possible decompositions of a quad used while expanding a terrain. White points are not expanded; black points require expansion. The decomposition produces either triangles, shown in outline here, or recurses to finer quads, shown shaded here.

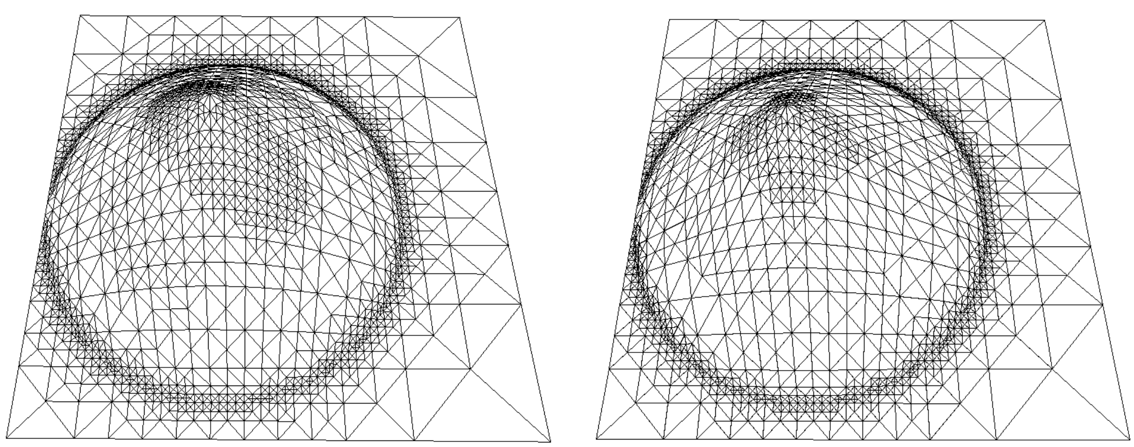


Figure 9.4. Mesh for terrain rendering of a cone sticking up out of a plane. On the left, Lindstrom's triangle-strip based method. On the right, our quad-based method. Note how on the right the quads are split around the axis of the cone, conforming to the surface curvature using fewer primitives than on the left.

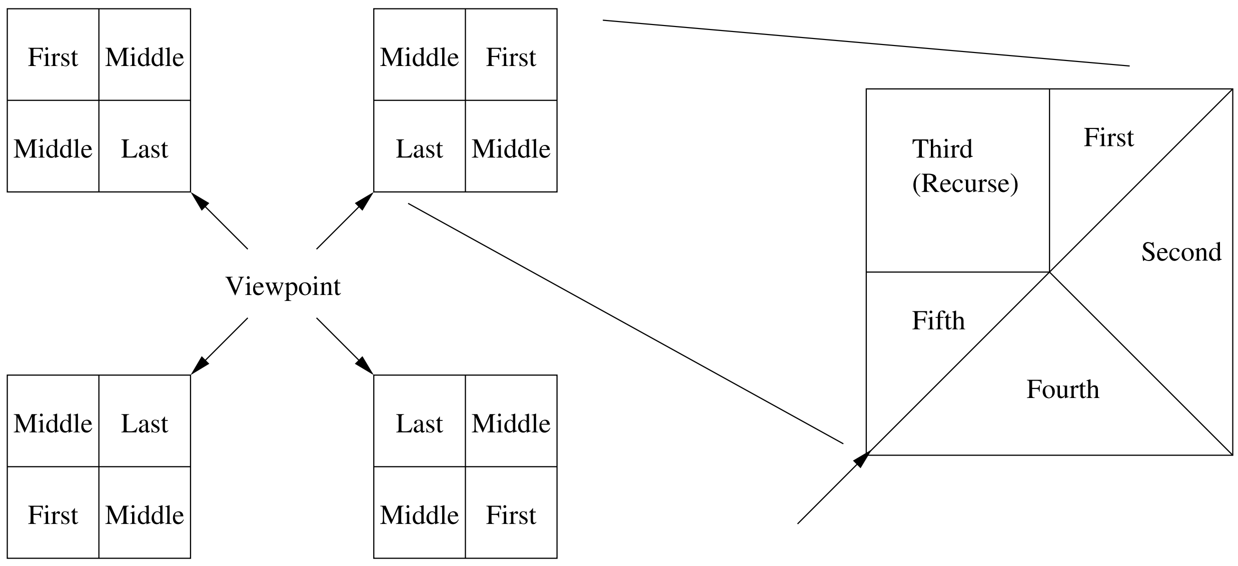


Figure 9.5. Viewpoint-dependent terrain rendering order. Left portion shows the four basic orderings for each diagonal direction. Right portion shows various complications encountered when decomposing into a mix of recursively drawn quads and triangles.

broad strokes, we simply draw each quad's children in depth order, as shown in the left portion of Figure 9.5. Because the edges of quads, where overlap could occur, run only vertically and horizontally, it is sufficient to consider just four camera view directions, in each of the diagonal directions or quadrants.

The process isn't quite as simple as ordering the recursive traversal, however, because a quad may be decomposed into triangles as well as simple recursive children. Also, each portion of a quad may also include objects scattered over the terrain, which we store associated with each quad they touch. In this case, we must ensure that each piece of a quad is drawn before any overlapping closer pieces. An example of a more complicated rendering order is shown in Figure 9.6.

To perform this view-dependent decomposition efficiently at runtime, we precalculate a table that lists both the decomposition of a quad into pieces, as shown in Figure 9.3, as well as the order the pieces must be drawn in, as shown in Figure 9.5. We keep a separate table for each of the four diagonal camera orientations. This results in an efficient but correct view order traversal.

This method properly renders the scene from back-to-front, as shown in Figure 9.7. The elevation data for this image comes from the USGS DEM, and the texture from a Landsat image of Denali National Park. DEM and texture were assembled by the author [Law01]. In this wide-angle case, the child processing order must be determined separately at each node. The right portion of this figure is colored by the processing

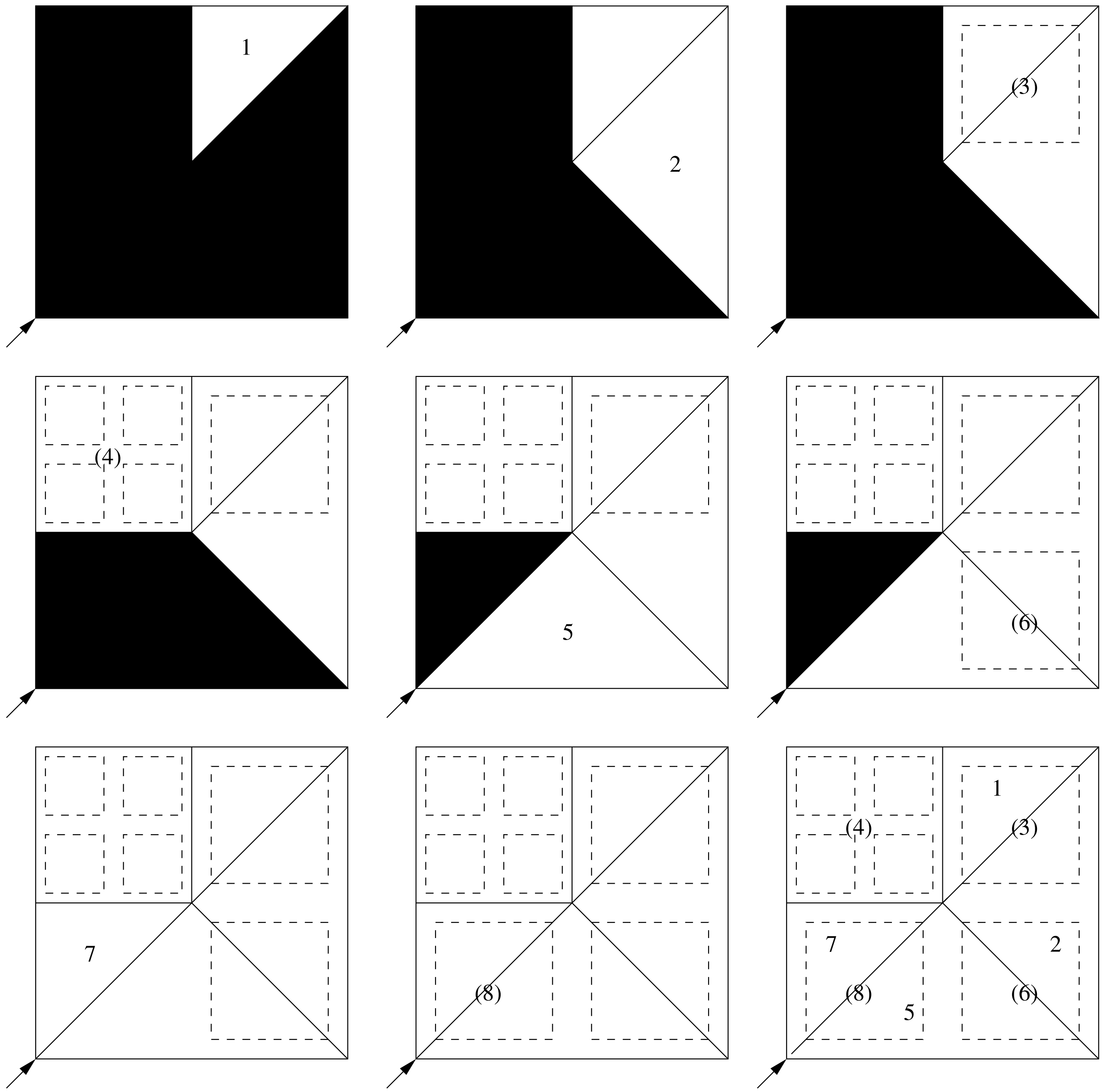


Figure 9.6. Steps in rendering one complicated terrain quad, including objects. Black represents areas that have not been drawn yet. Step 1 draws a small triangle in the back portion of the quad; 2 a large triangle; after the ground beneath them has been drawn, step 3 draws the objects on the back quad; 4 recursively expands a side quad and its objects; 5 draws another large triangle; this lets us drawn the objects in another side quad 6; 7 draws the final terrain triangle; and 8 draws the objects on the final quad.

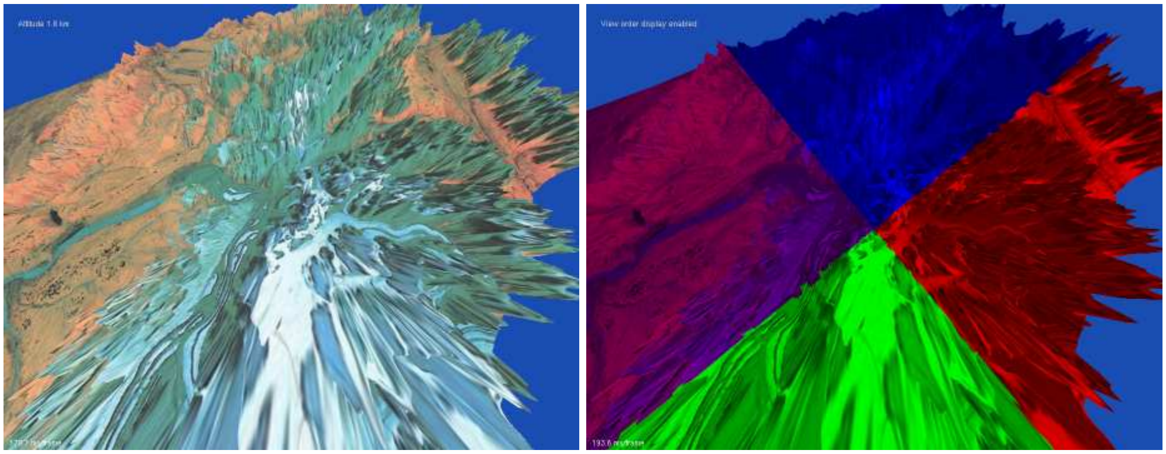


Figure 9.7. Extreme wideangle view looking down from 1.8km above Denali National Park in Alaska. This image is correctly rendered in back-to-front (outside to inside) order using our painter's algorithm depth sort for terrain. The right image shows the different orders the parts of the terrain are rendered in.

order; note how the order changes as the camera view vector crosses the X and Y axes of the terrain, which run along the diagonals of the image.

9.3 Data Sources and Integration

An immense and diverse body of digital data is available for the UIUC campus. There are, however, a variety of practical difficulties encountered when attempting to integrate this data into a single coherent model.

- Disparate data formats. Data is available in vector image files such as AutoCAD, PostScript, or PDF; as well as raster image files including JPEG and TIFF. Tools must thus be written to accept this data and convert it into some directly usable format.

- Disparate measurement units and coordinate systems. Data comes in meters, feet, and arbitrary units such as pixels; and is projected to some map projection. Sadly, data is very rarely tagged with the correct measurement units and coordinate systems, and many file formats do not even allow these tags. Hence at least some human labor is often required to recognize the coordinate system and units for each data source, and then convert each one to some common system.

- Data ambiguity, often because of limitations in the file format. For example, virtually all file formats only store 2D data, but actually represent some 3D reality. In some cases, such as wall or building outlines, it is relatively straightforward to extrude 2D features into 3D. For other cases, such as reconstructing closed regions given a set of (not quite closed) region outlines, this can be quite difficult. Many of these files are intended to convey certain aspects of the system to a human observer; and hence it is not always obvious how a machine can interpret them at all.

This section describes our approaches for mitigating these difficulties, and the data sources we used to build our campus model.

9.3.1 Coordinate System

To build a single model, we require a single common coordinate system, which for geographic data is typically a map projection, as well as a single set of measurement units. The factors involved in this choice include the accuracy required by the system, the coordinate systems of the input data sources, the ease of conversion between systems, and, rather surprisingly, machine precision.

Machine precision affects many commonly used map projections because the projection coordinate origin is typically quite far from the point of interest. For example, the origin for the Universal Transverse Mercator (UTM) map projection lies at the equator, which is approximately 4,400 kilometers from Illinois. A 32-bit floating-point Y coordinate of 4,400,000 meters has a mantissa quantization error of 0.5 meters, which is far too coarse to position objects and the camera accurately.[1] Because 32-bit floating-point coordinates are used throughout the graphics pipeline, this quantization error means we cannot use most map projections directly.

To avoid this quantization, we store model coordinates with 64-bit floating-point values. Before rendering, we shift the model origin to lie at the camera location, which avoids the quantization problems associated with the direct use of map coordinates.

9.3.1.1 Map Projection

A huge variety of map projections exist, all of which attempt to project inherently 3D reality onto a 2D coordinate plane while preserving distances and angles. Because the Earth is curved, any approximation becomes fairly inaccurate—failing to preserve angles, distances, or both—for areas more than several hundred kilometers across.

[1] As an aside, note that 32-bit fixed point, because of the missing exponent field, can maintain significantly higher precision with global range.

Geographic projection is the simplest projection—the map projection coordinates of a point are simply the latitude and longitude of the point. Though simple, geographic projection becomes highly distance and angle-distorting toward the poles, as one degree of longitude near the pole is much smaller than one degree of latitude. The geographic projection is the common interchange standard, and was the gridding scheme used by older USGS elevation data.

The Universal Transverse Mercator (UTM) map projection[Sny87] splits the Earth at the poles and unwraps a set of 60 tall thin "zones" distributed every 6 degrees of longitude around the equator. Within a zone, distances and angles can be measured with reasonable accuracy. Illinois is in UTM zone 16.

The Illinois Coordinate System (ILCS) map projection[Sny87] is defined by Illinois state law (765 ILCS 225, the Illinois Coordinate System Act) as an orthogonal system with a specified origin. This is the projection used by the campus basemaps.

Converting between map projections involves a few rather subtle aspects. The angle used for latitude is "geodetic"—that is, it measures the angle of a surface normal on the Earth. Measuring the angle of the position vector from the center of the Earth gives "geocentric" latitude, which isn't quite the same thing because the Earth isn't quite a sphere.[2] The standard approximation treats the Earth as an ellipsoid using a known semimajor axis and eccentricity. A variety of ellipsoids exist; but the WGS84 semimajor axis and eccentricity are often used.

Converting between different map projections typically involves a conversion to and from these (geodetic) geographic coordinates, and is hence complicated and slow. But locally, if two different map projections are both useful (i.e., neither is highly distorting) then they are very nearly equivalent up to some 2D affine transformation. Thus for computational purposes over a small region, we can simply precompute a fixed affine transformation relating the two map projections, which is simpler and much faster than using the actual projections. In practice, if two map projections are brought into affine correspondence at a point, the error 5km away from using the affine correspondence instead of the true projections is normally less than 1mm.

In this work, we convert all maps to UTM map coordinates, in meters, and use UTM easting and northing as our x and y axes. The UIUC campus, in UTM coordinates, lies at coordinate range $x = (393000 - 397100)$ meters and $y = (4437500 - 4441600)$ meters.

[2] The rotation of the Earth flattens the poles and expands the equator slightly.

To avoid the precision problems described in the previous section, we place our model origin at UTM map coordinates (395000,4440000) meters, and elevation 210 meters. This keeps the ranges of x, y, and z coordinates reasonable.

9.3.2 USGS Elevation and Topographic Maps

The United States Geological Survey publishes a variety of relatively coarse elevation and land use data[Sur04] for the entire United States.

We use the USGS 1:24K scale Digital Elevation Model (DEM) data as our source for elevation data. This dataset is provided as a raster image of elevations, at 10m pixel spacing, projected to the UTM map projection. Figure 9.8 shows the elevation map used for the campus model.

Although DEMs are simply a rectangular array of heights, they are normally stored in their own special file formats, of which the USGS uses several. The Spatial Data Transfer Standard (SDTS) is the modern standard, which includes extensive metadata, including scale and projection information, along with the elevation records. SDTS data files can be read with the FIPS 173 decoder library, available from the USGS. Older USGS DEMs are in a terse, fixed-record-length ASCII format, and typically use the geographic (latitude/longitude) projection.

The zero elevation for a DEM is defined as mean sea level, which is difficult to define far from the coast. DEMs are thus referenced from an extrapolated whole-Earth mean sea level “datum”, such as the WGS84 datum. This can differ from an ellipsoid representation of the Earth by dozens of meters of elevation, but this variation varies quite smoothly, and so is only important for absolute positioning.

Because a DEM stores elevations referenced from sea level, the curvature of the Earth means DEM elevations are not directly usable as planar z coordinates. If we treat the Earth locally as a sphere of radius r, moving a horizontal distance x tangent to the surface of the Earth puts us a vertical distance h off the Earth’s surface, where $r^2 = x^2 + (r-h)^2$ as shown in Figure 9.9. Thus $h = r - \sqrt{r^2 - x^2}$. For $r = 6378500$ meters and $x = 2000$ meters, half the width of the campus, the curvature drop $h = 0.31$ meters, which is a non-negligible vertical change.

We compensate for the Earth’s curvature subtracting the vertical drop h to every sample of the DEM elevation, using the center of the image as the reference point. The resulting height is then used as the distance up along the z axis. This approximation ignores the true variation of the “vertical” direction across

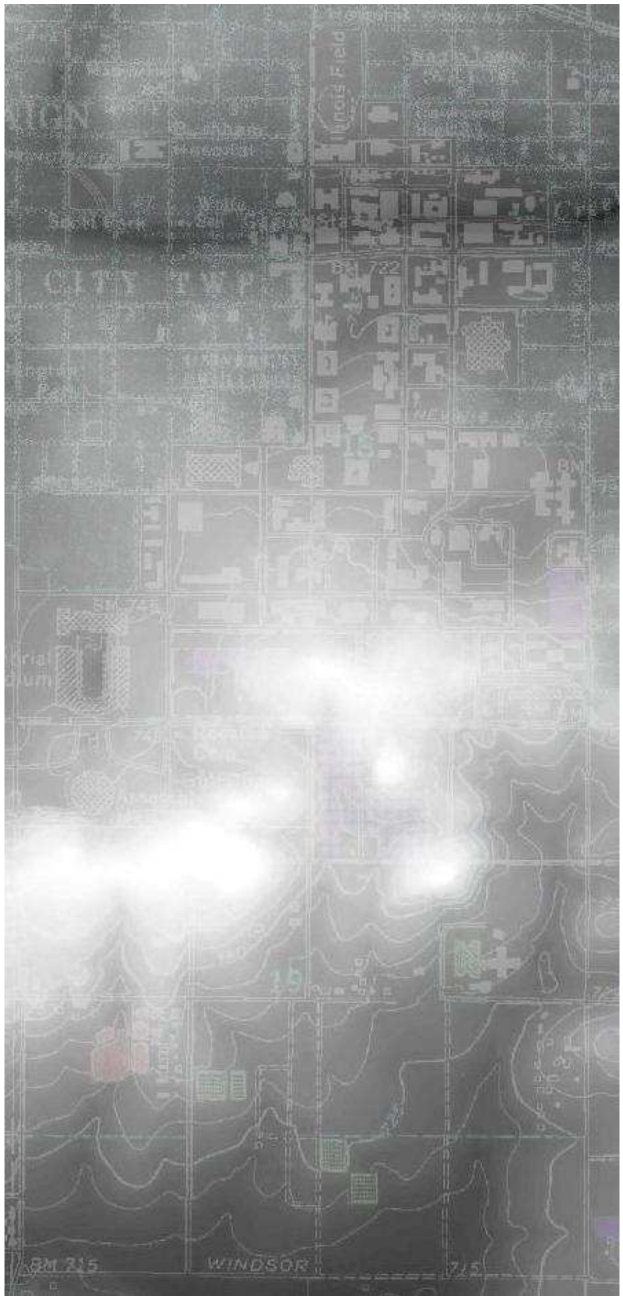


Figure 9.8. A 4.7 kilometer high section of the USGS 10m elevation map, with the USGS topographic map overlaid. The region displayed stretches north-south from University Avenue to Windsor Road; and from 1st Street in Champaign to Lincoln Avenue in Urbana.

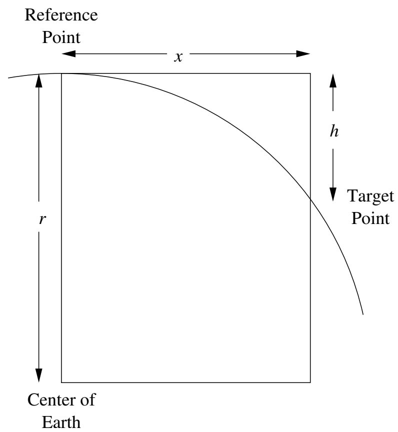


Figure 9.9. Calculating the height difference caused by the curvature of the Earth.

the surface of the Earth, but over 5km this variation only amounts to about 1/20 of a degree, and for our purposes can be ignored.

9.3.3 AutoCAD Maintenance Maps

The UIUC Operations and Maintenance division maintains maps describing several different aspects of the campus. The utilities map, for example, describes vector contours for every building, street, sidewalk, body of water, and grassy area on campus, as well as the location of every tree, bush, streetlight, power pole, manhole and sewer drain on campus. Building maps describe, for each floor, the locations of all walls, doors, stairs, and windows.

Many of these maps are stored as highly layered 2D vector AutoCAD files. We use the excellent, freely available OpenDWG [All04] library to read the AutoCAD files. We then process the 2D maps into a usable state using several small custom tools.

AutoCAD is designed to output drawings to 2D pen plotters, and is hence line-centric. But to build a convincing virtual world, we need 2D areas that represent portions of 3D space, not just outlines. Luckily, many features in modern AutoCAD files are enclosed by closed polylines which can easily be rasterized into areas and then projected into space. Older AutoCAD files may bound areas by disjoint sets of line segments, from which actual areas may be difficult to identify or even impossible to reconstruct. A good reference on AutoCAD topological cleanup is the Master's thesis of Lewis [Lew96].

The campus maps are measured in either feet or inches, and most are registered to the campus basemaps, which are in the ILCS map projection. Processing thus involves first bringing the drawings into the UTM map projection from the Illinois State Coordinate System (ILCS). Near UIUC, ILCS meters can be converted to UTM zone 16 meters via the 2D affine matrix (note that this is only a slight rotation and a shift):

$$\begin{bmatrix} 0.999649 & 0.014986 & 80745.669 \\ -0.014986 & 0.999649 & 4063234.149 \end{bmatrix}$$

9.3.3.1 Utilities Map

The map data are then separated out by layer. An example of a section of the main Operations and Maintenance utilities map, UTIL2002r2004, is shown in Figure 9.10. We extract the following ground coverage information from this utilities map. Luckily, these layers all contain closed polylines, and the layers' areas are all disjoint.

- Building outlines are extracted from the BUILDING layer.
- All roads are outlined in the STREET layer. This gives the regions covered by roads all the way to the curbline. Unfortunately, by default the roads wrap around each block, but do not properly terminate at the map boundaries. Hence we must add an external wrapper to end the region bounded by roads.
- Pedestrian sidewalks are outlined in the SIDEWALK layer.
- Parking lots and related access roads are outlined in the PARKING_SERVICE_DRIVES layer.
- Grass and other landscaped regions are listed in the GROUNDS layer.
- Rivers (such as the Boneyard Creek flowing through campus) and lakes (such as the settling ponds on the south campus farms) are outlined on the WATER layer.
- Soccer and tennis fields are listed on the RECREATION_AREA layer.

These fields comprise a total of 12,033 separate closed outlines, with 610,636 vertices listed. Much information remains in the utilities map to be extracted, including many varieties of manholes, storm sewers, and utility lines.

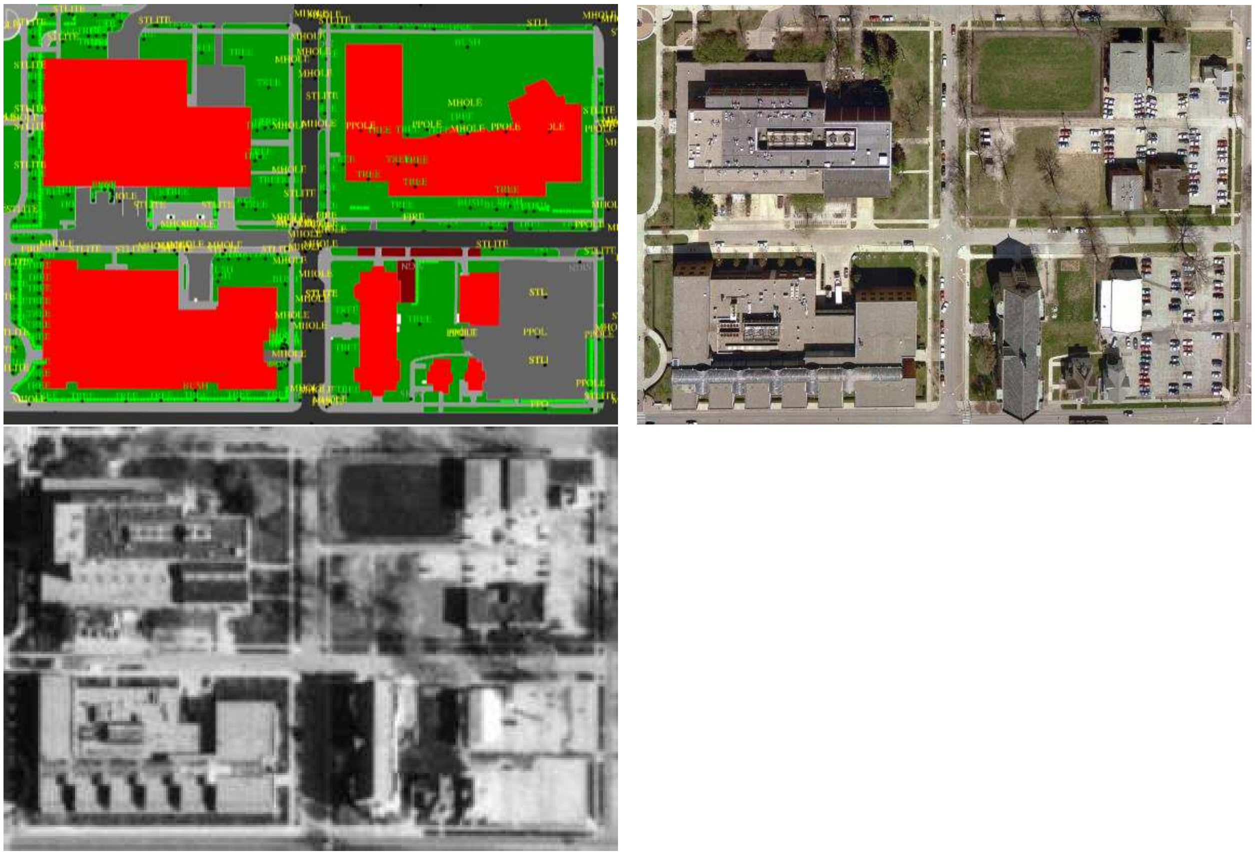


Figure 9.10. Four blocks from the 2004 campus utilities map (top) and 1999 aerial photo (middle), and 1998 USGS orthophoto (bottom) near Goodwin and Springfield Avenue in Urbana, Illinois. Note how the Siebel Center in the top right, constructed in 2003, does not appear in the lower two photos.

9.3.3.2 Basemaps

We extract the following information from the UIUC basemaps. The basemaps were prepared from a photogrammetric survey in 1993.

- Trees (TREE) and bushes (BUSH) are listed as inserts in the LANDSCAPE layer.
- Electrical power poles (PPOLE), street lights (STLITE), fire hydrants (FIRE), and electrical boxes (EBOX) are listed as inserts in the UTILITIES layer.

There are 13,334 trees, 3,454 bushes, 2,560 street lights, 1,334 power poles, and 282 fire hydrants listed on these maps.

9.3.4 University Map Website

The above maps only give building outlines, not building names or operations and maintenance building numbers. This makes the maps difficult to relate to other data source, such as building outlines, which are indexed by building number.

Hence to associate each building outline with a building number, a useful data source is the university map website [oIaUC04], which provides a clickable image map of the campus. This website uses the building number to identify buildings internally, so building outlines and numbers can be easily extracted from the web page text.

These maps are in their own coordinate system, likely related to the PDF conversion of the original AutoCAD source data. The coordinate system (in pixels) appears to be related to UTM meters by the 2D affine matrix:

$$\begin{bmatrix} 1.374704 & -0.008082 & 393608.718 \\ -0.012479 & -1.373144 & 4441509.0 \end{bmatrix}$$

9.3.5 Road Signs

The United States Federal Highway Administration publishes a specification for road signs, the Manual for Uniform Traffic Control Devices [oTFHA04], which includes vector images of every official road sign. We have extracted these roadsigns using our custom Postscript interpreter, and can easily render them as antialiased images of any size. A partial index is shown in Figure 9.11.

The road signs in the model are dynamically generated from these vector versions.

9.4 Roof Extrusion

For conventional architecture, walls can trivially be extruded from a building outline. However, pitched roofs are more difficult to reconstruct, especially for complicated building outlines.

The remote sensing literature contains several computer-vision based methods that can assist a photogrammetry operator with the semi-automatic extraction of roof shapes [CW99]. However, none of these photogrammetric methods are fully automated.

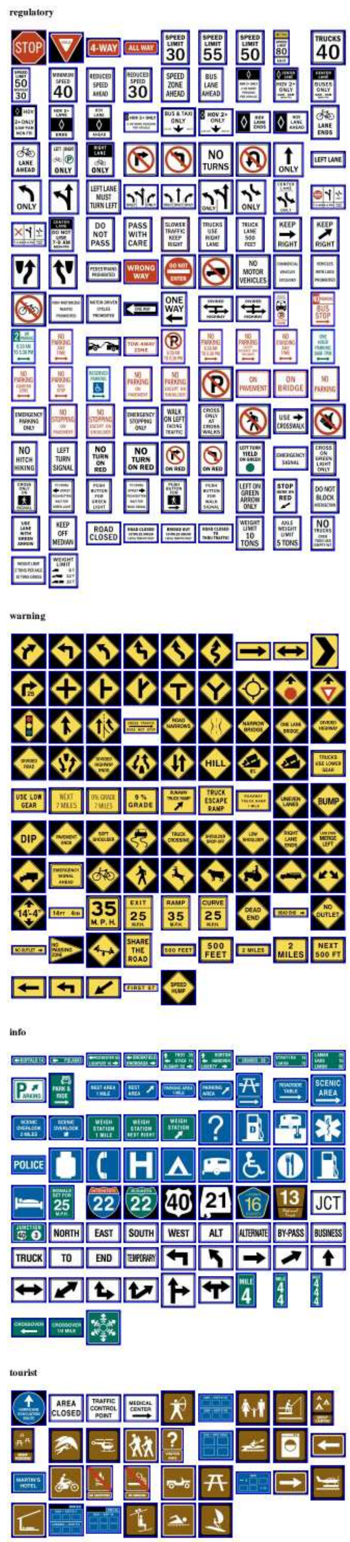


Figure 9.11. A partial index of public domain vector-graphics road sign images.

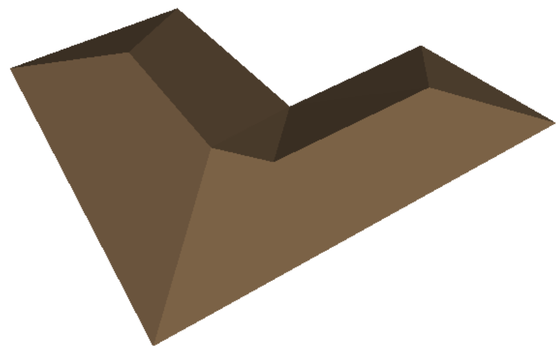


Figure 9.12. A roof for a simple building generated via the straight skeleton.

9.4.1 Straight Skeleton Roof Extrusion

A more interesting and fully automated method is presented by Brenner [Bre00]. This method is based on the "straight skeleton" of the building outline polygon. The straight skeleton [AAA$^+$95] of a polygon is the shape swept out by shrinking the polygon by sweeping each edge of the polygon inward at a fixed rate, and merging vertices as they intersect. Equivalently, we can imagine constructing a roof by intersecting tilted planes extruding from each wall. This is also equivalent to a spacetime plot of a 2D polygon burning away under a non-entropy-conserving (translational) burning function.

The straight skeleton is quite similar to the medial axis (the set of points equidistant to at least two input points), and is in fact identical to the medial axis for a convex shape. An example of a 3D roof generated from the straight skeleton is shown in Figure 9.12, and the same roof generated using a medial axis technique (described below) is shown in Figure 9.12.

Eppstein and Erickson [EE99] presented a variety of interpretations of the straight skeleton, and presented a relatively efficient algorithm for computing the straight skeleton from a set of input vertices.

9.4.2 Medial Axis Roof Extrusion

Real roof outlines contain symmetries, colinearities, and other degeneracies that make computing the exact straight skeleton difficult. As an engineering solution we currently use a robust Voronoi-based method to approximate the medial axis during roof construction. A Voronoi diagram consists of a set of Voronoi cells, one for each input point, containing the region of space closer to that input point than any other. Voronoi

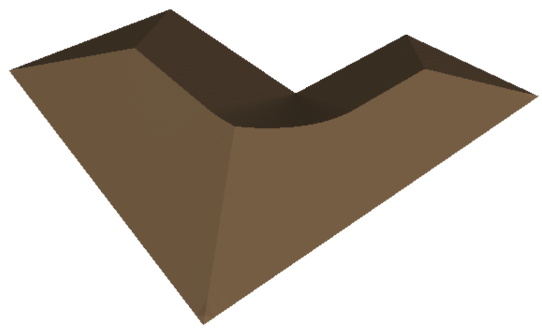


Figure 9.13. The same roof generated via the medial axis.

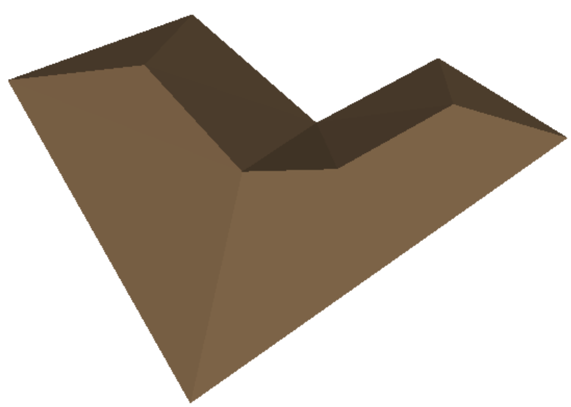


Figure 9.14. The same roof generated via the medial axis, then simplified.

cells are bounded by Voronoi edges (consisting of points equidistant from two input points) and Voronoi vertices (points equidistant from at least three input points). For a densely-sampled 2D closed outline, the Voronoi diagram is a good approximation of the medial axis.

We form a triangulation of the shape interior by modifying the Voronoi diagram of the densely-sampled outline of the building shape. For each Voronoi edge, we connect both Voronoi vertex endpoints to the equidistant input set points that define the edge. These new "cell" edges always connect a Voronoi vertex to one of its defining set points, and because Voronoi cells are convex and set points lie in the interior of their cells, these new edges are guaranteed never to cross. For Voronoi vertices that lie outside the original shape, we throw away the outside vertex and all edges connected to it. Finally, we add in the edges of the original shape outline, removing any Voronoi edges they cross.

The steps involved are shown in Figure 9.15. The original building outline points are shown as small red triangles; and the densely-sampled outline vertices, original outline edges, and Voronoi vertices in black. Figure 9.15(a) shows the original Voronoi diagram, with Voronoi edges shown in green, and the Voronoi edges to infinity omitted. Figure 9.15(b) shows the same shape with our cell edges inserted and shown in blue. Figure 9.15(c) shows the same diagram with outside Voronoi vertices removed.

For an input shape without acute angles, this process results in a set of edges that form a new triangulation of the points along the shape outline as well as points along the medial axis (the interior Voronoi vertices). However, where the input shape has highly acute angles, the Voronoi vertices for set points may lie outside the shape, and hence those set points may not have edges to Voronoi vertices. Figure 9.15(d) shows a roof shape with these very small angles. As is well known in Voronoi triangulation, for small angles, the relevant Voronoi vertex may lie outside the shape; so we will filter out these vertices. To handle these small angles, we detect set points without any cell edges to Voronoi vertices, and propagate new edges to these points from topologically adjacent set points that have cell edges to Voronoi vertices. Figure 9.15(e) shows these new cell edges in red.

The triangulation scheme we have described relies on the well-known Voronoi diagram, which we compute using Fortune's sweep-line algorithm [For87]. Because none of the edges cross—original set edges, surviving Voronoi edges, or new cell edges—the scheme results in a valid triangulation for the set points and Voronoi vertices for the entire building shape. Despite the many degeneracies present in building outlines, this technique appears to be quite robust; a real building outline is shown in Figure 9.16.

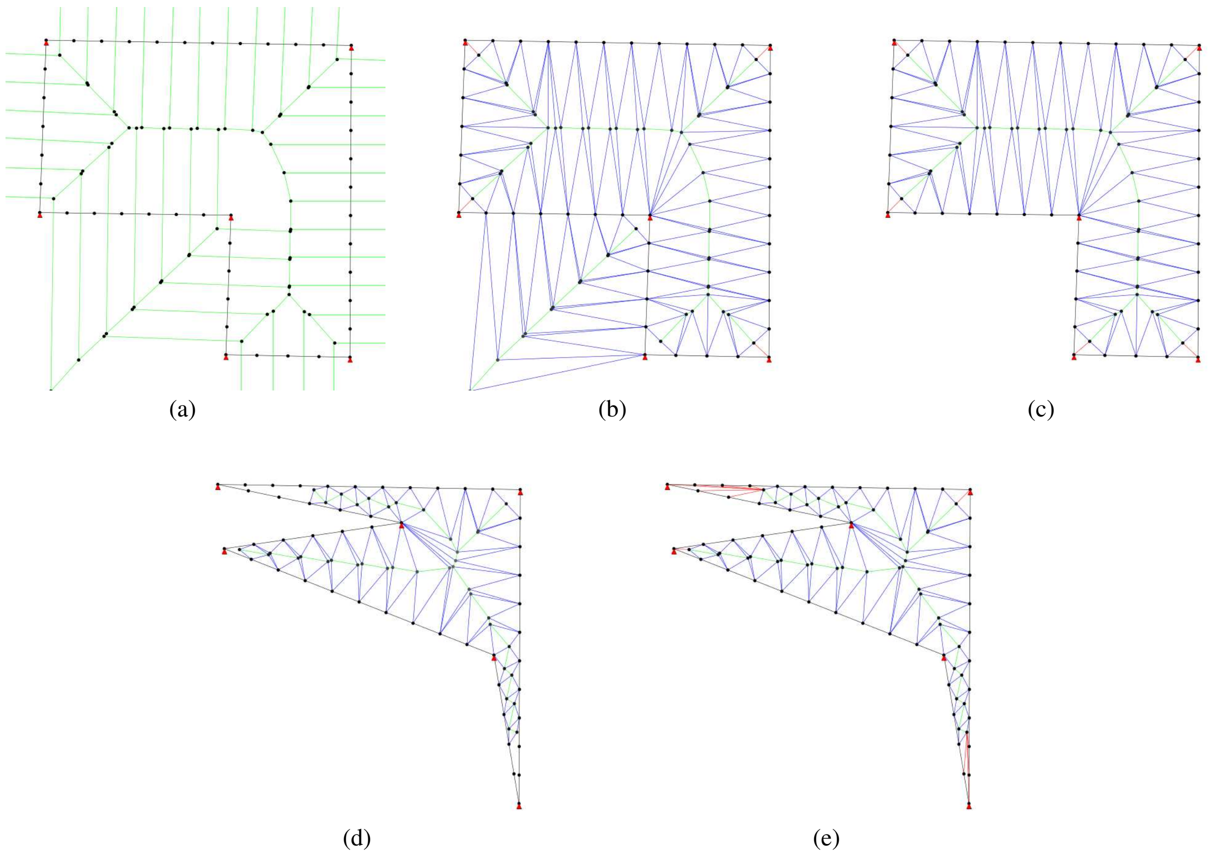


Figure 9.15. Steps involved in constructing a roof triangulation. (a) Voronoi triangulation. (b) Cell edges inserted. (c) Exterior Voronoi vertices removed. (d) Missing cell edges for small angles. (e) Cell edges added along small angles.

To inflate this triangulation to 3D, we set the elevation of all set points to 0, and set the elevation of each Voronoi vertex to the roof slope (rise over run) times the distance from the Voronoi vertex to its defining set points. This results in a good approximation of the roof shape, although along convex corners the smooth curves of the resulting profile appear a bit strange, as seen in Figure 9.13.

To remove these curves, and to speed up roof rendering, as a final step we simplify the roof triangulation. We use Garland's quadric-based simplification scheme [GH97], which works quite well when instructed to produce an output mesh with twice as many triangles as the original building outline had edges. A simplified roof is shown in Figure 9.14.

By outlining the Voronoi edges along the medial axis, we can highlight the roof ridgelines. By drawing in (a subset of) the new cell edges added, we can depict the water channels in a corrugated metal roof. This allows us to generate roof texture while we generate roof geometry, as shown in Figure 9.17.

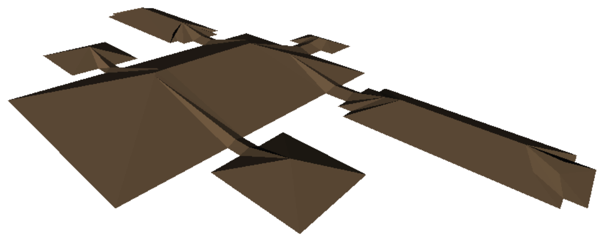


Figure 9.16. A complicated building roof generated via the medial axis, then simplified.

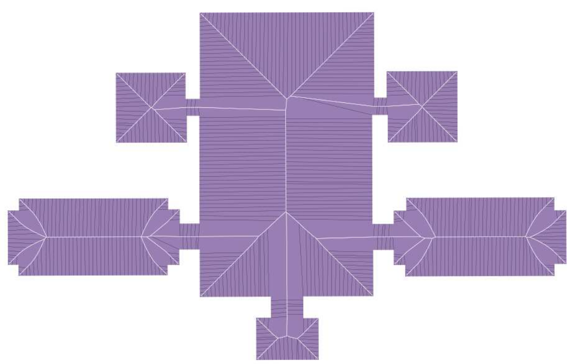


Figure 9.17. A complicated building roof texture generated by outlining the medial axis and cell edges.

9.5 Future Work

The accuracy and efficiency of the model could be improved in a number of ways.

Culling. Our height map traversal algorithm does not perform occlusion culling, which could avoid drawing scenery that is obscured by buildings. The height map could be modified to perform occlusion culling in substantially the same manner as view culling. To do this, during the preprocess one could compute and propagate 2D angular view information up the bounding hierarchy, then during traversal perform early exit testing to cull out obscured geometry.

Splitting and Merging. As the camera moves, we split and merge the geometry represented by each impostor. Splitting a set of geometry means replacing the geometry by its children in the scene graph. For example, splitting might replace a single tree by separate impostors for its trunk and branches; replace small tree branches by separate impostors for its twigs and leaves; replace a building by separate impostors for each wall; or replace a small wall section by individual bricks. Merging is simply the inverse process—a set of children are replaced by their common parent. While splitting and merging could in theory occur between arbitrary geometry, we only use decompositions that respect the scene graph.

Interiors. Extensive data is also available for building interiors; but because our focus is on exteriors, we only use this data to track the locations of exterior walls. Building interiors would be most naturally supported by a separate portal-based system, which would complement the terrain-based system we use for exteriors.

Aerial Photos. On April 7, 1999, the university commissioned an aerial survey of the campus, producing a series of 9 inch prints which were scanned at 600dpi into 5000x5000 pixel full color digital images. There are two passes, a 500 (feet of ground per inch of film) scale with a ground resolution of approximately 24cm per pixel and a 250 scale with a ground resolution of approximately 12cm per pixel, as shown in Figure 9.10.

These photos have thus far only been used to manually verify the contents of the maps. Automated analysis of these photos remains future work.

Ground Photos. Finally, we have a series of panoramic photos taken from ground level using a handheld digital camera. The digital camera photos are in EXIF format, for which geometric and radiometric calibration profiles are available. The images can thus be processed into distortion-free, linear-light images.

We expect to use the ground photos to calibrate, verify, and fill in holes in the aerial photos.

Chapter 10

Conclusion

We have presented a wide variety of work related to parallel impostor-based rendering. The unifying theme to this work is that the additional performance delivered by both impostors and parallelism allows new, higher-quality rendering schemes to be used to render large worlds. We now summarize our contributions.

- We have designed and implemented a high-performance, production-quality parallel migratable work system, the CHARM++ array elements described in Appendix A. In particular, this system efficiently supports broadcasts, used to distribute the impostor viewpoint; messaging, used in the rendering of impostors; and migration, used for load balancing. The system includes our novel method to support collective operations when migrations are ongoing.

- We have presented two useful tools for constructing high-performance client/server applications, the CCS protocol in Appendix B.1 and the PUP object serialization framework in Appendix B.2. CCS is used to connect the visualization client to the parallel server, and PUP is used to easily format all I/O and network messages.

- We have presented a high-performance rendering architecture, parallel impostors, as described in Chapter 4. We presented a theoretical analysis of the benefits of impostors in Chapter 2.2 and measured the actual benefits in Chapter 6, both of which show parallelism and impostors together can allow dramatically increased performance and improved image quality compared to existing methods.

- We have described several rendering techniques that are made affordable in real time by the extra rendering performance of both parallelism and impostors. These include an integrated approach to antialiasing explained in Section 7.1; a method to approximate blurry shadows, the penumbra limit map technique described in Section 8.1; and a method to compute global illumination, the impostor global illumination technique described in Section 8.2.

- We have described a series of techniques for synthesizing and rendering large-scale scenes in Chapter 9. These include techniques for handling and integrating various data sources, including maps and DEMs; as well as the Voronoi roof extrusion method in Section 9.4.

We have presented parallel impostors, a general technique that exploits view coherence to decrease the latency and bandwidth required for interactive 3D rendering. We have shown how this technique enables us to utilize the rendering power of large parallel machines to extend the state of the art in interactive rendering quality, by making smooth, fully antialiased rendering with high-quality lighting affordable. Finally, we have demonstrated the technique is applicable to very large outdoor environments.

Bibliography

[AAA$^+$95] Oswin Aichholzer, Franz Aurenhammer, David Alberts, , and Bernd Gartner. A novel type of skeleton for polygons. *Journal of Universal Computer Science*, 1(12):752–761, 1995.

[AHT04] Jukka Arvo, Mika Hirvikorpi, and Joonas Tyystjärvi. Approximate soft shadows with an image-space flood-fill algorithm. 2004.

[Ali98] Daniel G. Aliaga. *Automatically Reducting and Bounding Geometric Complexity by Using Images*. PhD thesis, University of North Carolina at Chapel Hill, 1998.

[All04] Open Design Alliance, 2004. http://opendwg.org.

[Ass03] Ulf Assarson. *A Real-Time Soft Shadow Volume Algorithm*. PhD thesis, Chalmers University of Technology, 2003.

[BK99] Robert K. Brunner and Laxmikant V. Kalé. Adapting to load on workstation clusters. In *The Seventh Symposium on the Frontiers of Massively Parallel Computation*, pages 106–112. IEEE Computer Society Press, February 1999.

[BK00] Robert K. Brunner and Laxmikant V. Kalé. Handling application-induced load imbalance using parallel objects. In *Parallel and Distributed Computing for Symbolic and Irregular Applications*, pages 167–181. World Scientific Publishing, 2000.

[BKdSH01] Milind Bhandarkar, L. V. Kale, Eric de Sturler, and Jay Hoeflinger. Object-Based Adaptive Load Balancing for MPI Programs. In *Proceedings of the International Conference on Computational Science, San Francisco, CA, LNCS 2074*, pages 108–117, May 2001.

[BN76] James Blinn and Martin Newell. Texture and reflection in computer generated images. *Communications of the ACM*, 19(10):542–546, October 1976.

[Bre00] Claus Brenner. In *Proc. XIX ISPRS Cong., Int. Archives of Photogrammetry*, pages 85 – 92, 2000.

[Bru00] Robert K. Brunner. Versatile automatic load balancing with migratable objects. TR 00-01, January 2000.

[BS02] Stefan Brabec and Hans-Peter Seidel. Single sample soft shadows using depth maps. 2002.

[CC03] Hseuh-Ting Chu and Chaur-Chin Chen. On bounding boxes of iterated function system attractors. *Computers & Graphics*, 27:407–414, 2003.

[CD03] Eric Chan and Frédo Durand. Rendering fake soft shadows with smoothies. In *Eurographics Symposium on Rendering*, 2003.

[CG85] Michael F. Cohen and Donald P. Greenberg. The hemi-cube: a radiosity solution for complex environments. In *SIGGRAPH*, pages 31–40. ACM Press, 1985.

[CHH03] Nathan A. Carr, Jesse D. Hall, and John C. Hart. Gpu algorithms for radiosity and subsurface scattering. In *Proc. Graphics Hardware*, July 2003.

[CPC84] R.L. Cook, T. Porter, and L. Carpenter. Distributed ray tracing. In *SIGGRAPH Proceedings*, pages 137–145, 1984.

[CW93] Shenchang Eric Chen and Lance Williams. View interpolation for image synthesis. In *SIGGRAPH Proceedings*, pages 279–288. ACM Press, 1993.

[CW99] Shih-Hong Chio and Shue-Chia Wang. Semi-automatic system for roof reconstruction based on 3d linear segments. In *Proceedings of 20th Asian Conference on Remote Sensing (ACRS)*, 1999.

[Dit91] Klaus R. Dittrich. Object-oriented database systems: The notion and the issues. In *On Object-Oriented Database Systems*, pages 3–12. 1991.

[DSSD99] Xavier Decoret, François Sillion, Gernot Schaufler, and Julie Dorsey. Multi-layered impostors for accelerated rendering. *Computer Graphics Forum (Eurographics '99)*, 18(3), 1999.

[Duf89] Tom Duff. Polygon scan conversion by exact convolution. pages 154–168, 1989.

[EE99] David Eppstein and Jeff Erickson. Raising roofs, crashing cycles, and playing pool: Applications of a data structure for finding pairwise interactions. *Discrete and Computational Geometry*, 22(4):569–592, 1999.

[For87] S. Fortune. Sweepline algorithms for voronoi diagrams. *Algorithmica*, 2:153–174, 1987.

[GH97] Michael Garland and Paul Heckbert. Surface simplication using quadric error metrics. In *Proc. SIGGRAPH'97*, pages 209–216, 1997.

[GSHG98] Gene Greger, Peter Shirley, Philip M. Hubbard, and Donald P. Greenberg. The irradiance volume. *IEEE Computer Graphics and Applications*, 18(2):32–43, March 1998.

[Hai01] Eric Haines. Soft planar shadows using plateaus. *J. Graph. Tools*, 6(1):19–27, 2001.

[Har02] Mark J. Harris. Real-time cloud rendering for games. In *Proceedings of Game Developers Conference*, March 2002.

[HCK+99] John C. Hart, Nate Carr, Masaki Kameya, Stephen A. Tibbitts, and Terrance J. Coleman. Antialiased parameterized solid texturing simplified for consumer-level hardware implementation. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, pages 45–53. ACM Press, 1999.

[HD91] John C. Hart and Thomas A. DeFanti. Efficient antialiased rendering of 3-D linear fractals. *Computer Graphics*, 25(3), 1991.

[Hec89] Paul S. Heckbert. Master's thesis, University of California, Berkeley, June 1989.

[HHN+02] Greg Humphreys, Mike Houston, Ren Ng, Randall Frank, Sean Ahern, Peter D. Kirchner, and James T. Klosowski. Chromium: a stream-processing framework for interactive rendering on clusters. In *SIGGRAPH Proceedings*, pages 693–702. ACM Press, 2002.

[HLHS03] Jean-Marc Hasenfratz, Marc Lapierre, Nicolas Holzschuch, and François Sillion. A survey of real-time soft shadows algorithms. In *Eurographics*. Eurographics, Eurographics, 2003. State-of-the-Art Report.

[HLK03] Chao Huang, Orion Lawlor, and L. V. Kalé. Adaptive MPI. In *Proceedings of the 16th International Workshop on Languages and Compilers for Parallel Computing (LCPC 03)*, College Station, Texas, October 2003.

[Hut81] J. Hutchinson. Fractals and self-similarity. *Indiana University Mathematics Journal*, 30(5):713–747, 1981.

[Jen96] Henrik Wann Jensen. Global illumination using photon maps. In *Rendering Techniques '96*, pages 21–30. Springer-Verlag, 1996.

[Jia01] Xiangmin Jiao. *Data Transfer and Interface Propagation in Multicomponent Simulations*. PhD thesis, University of Illinois at Urbana-Champaign, 2001.

[JLK04] Rashmi Jyothi, Orion Sky Lawlor, and L. V. Kale. Debugging support for charm++. In *PADTAD Workshop for IPDPS 2004*. IEEE Press, 2004.

[KBJ+96] L. V. Kale, Milind Bhandarkar, Narain Jagathesan, Sanjeev Krishnan, and Joshua Yelon. Converse: An Interoperable Framework for Parallel Programming. In *Proceedings of the 10th International Parallel Processing Symposium*, pages 212–217, April 1996.

[KD03] Florian Kirsch and Juergen Döllner. Real-time soft shadows using a single light sample. 11(1), 2003.

[Kel97] Alexander Keller. Instant radiosity. In *SIGGRAPH*, pages 49–56, 1997.

[KK93] L.V. Kalé and S. Krishnan. CHARM++: A Portable Concurrent Object Oriented System Based on C++. In A. Paepcke, editor, *Proceedings of OOPSLA'93*, pages 91–108. ACM Press, September 1993.

[KK96] L. V. Kale and Sanjeev Krishnan. Charm++: Parallel Programming with Message-Driven Objects. In Gregory V. Wilson and Paul Lu, editors, *Parallel Programming using C++*, pages 175–213. MIT Press, 1996.

[KKD00] Laxmikant V. Kalé, Sameer Kumar, and Jayant DeSouza. An adaptive job scheduler for timeshared parallel machines. Technical Report 00-02, Parallel Programming Laboratory, Department of Computer Science, University of Illinois at Urbana-Champaign, Sep 2000.

[KKV03] L. V. Kale, Sameer Kumar, and Krishnan Vardarajan. A Framework for Collective Personalized Communication. In *Proceedings of IPDPS'03*, Nice, France, April 2003.

[KKZL03] Laxmikant V. Kalé, Sameer Kumar, Gengbin Zheng, and Chee Wai Lee. Scaling molecular dynamics to 3000 processors with projections: A performance analysis case study. In *Terascale Performance Analysis Workshop, International Conference on Computational Science(ICCS)*, Melbourne, Australia, June 2003.

[KRSS93] L.V. Kale, B. Ramkumar, V. Saletore, and A. B. Sinha. Prioritization in parallel symbolic computing. In T. Ito and R. Halstead, editors, *Lecture Notes in Computer Science*, volume 748, pages 12–41. Springer-Verlag, 1993.

[Law01] Orion Lawlor. Denali landsat image and 3d dem, 2001. http://charm.cs.uiuc.edu/users/olawlor/projects/2001/denali/.

[LC04] B. D. Larsen and N. Christensen. Simulating photon mapping for real-time applications. In Alexander Keller Henrik Wann Jensen, editor, *Eurographics Symposium on Rendering*, June 2004.

[Lew96] Rick Lewis. Generating three-dimensional building models from two-dimensional architectural plans. Master's thesis, UC Berkeley EECS Department, 1996.

[LH96] Marc Levoy and Pat Hanrahan. Light field rendering. pages 31–42, 1996.

[LH03] Orion Sky Lawlor and John Hart. Bounding iterated function systems using convex optimization. In *Proceedings of Pacific Graphics 2003*, pages 283–292, October 2003.

[LK03] Orion Sky Lawlor and L. V. Kalé. Supporting dynamic parallel object arrays. *Concurrency and Computation: Practice and Experience*, 15:371–393, 2003.

[LP02] Peter Lindstrom and Valerio Pascucci. Terrain simplification simplified: A general framework for view-dependent out-of-core visualization. *IEEE Transactions on Visualization and Computer Graphics*, 8(3):239–254, 2002.

[LP03] Eric LaMar and Valerio Pascucci. A multi-layered image cache for scientific visualization. In *IEEE Symposium on Parallel and Large-Data Visualization and Graphics*, pages 61–68, 2003.

[LV00] Tom Lokovic and Eric Veach. Deep shadow maps. In *SIGGRAPH 2000 Proceedings*, August 2000.

[Mar99] William R. Mark. *Post-Rendering 3D Image Warping: Visibility, Reconstruction, and Performance for Depth-Image Warping*. PhD thesis, University of North Carolina at Chapel Hill, April 1999.

[McC95] Michael D. McCool. Analytic antialiasing with prism splines. In *SIGGRAPH Proceedings*. ACM Press, 1995.

[MCEF94] Steve Molnar, Michael Cox, David Ellsworth, and Henry Fuchs. A sorting classification of parallel rendering. In *IEEE Computer Graphics and Applications*, volume 14-4, pages 23–32, July 1994.

[MEP92] Steven Molnar, John Eyles, and John Poulton. Pixelflow: high-speed rendering using image composition. In *SIGGRAPH Proceedings*, pages 231–240. ACM Press, 1992.

[Mit96] Don P. Mitchell. Consequences of stratified sampling in graphics. In *SIGGRAPH Proceedings*, pages 277–280. ACM Press, 1996.

[MPI94] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*, May 1994.

[MQWS02] Lucio Mayer, Thomas Quinn, James Wadsley, and Joachim Stadel. Formation of giant planets by fragmentation of protoplanetary disks. *Science*, 298:1756–59, 2002.

[MS95] Paulo W. C. Maciel and Peter Shirley. Visual navigation of large environments using textured clusters. In *Proceedings of the 1995 symposium on Interactive 3D graphics*, pages 95–ff. ACM Press, 1995.

[NPG] Mangesh Nijasure, Sumantra Pattanaik, and Vineet Goel. Interactive global illumination in dynamic environments using commodity graphics hardware. In *Proceedings of Pacific Graphics 2003*, pages 450–454. IEEE Computer Society.

[oIaUC04] University of Illinois at Urbana-Champaign. Campus map website, 2004. http://www.uiuc.edu/ricker/CampusMap.

[oTFHA04] United States Department of Transportation Federal Highway Administration. Manual on uniform traffic control devices, 2004. http://mutcd.fhwa.dot.gov.

[Par04a] Parallel Programming Laboratory, University of Illinois, Urbana-Champaign. *The Charm++ Programming Language Manual, Version 6.0*, Jan 2004.

[Par04b] Parallel Programming Laboratory, University of Illinois, Urbana-Champaign. *Converse Programming Manual*, Jan 2004.

[PHBI00] Steve Plimpton, Bruce Hendrickson, Shawn Burns, and Will McLendon III. Parallel algorithms for radiation transport on unstructured grids. In *Proceedings of Super Computing 2000*. http://sc2000.org, 2000.

[PSS98] Steven Parker, Peter Shirley, and Brian Smits. Single sample soft shadows. Technical Report UUCS-98-019, Computer Science Department, University of Utah, October 1998. http://www.cs.utah.edu/ bes/papers/coneShadow.

[PZKK02] James C. Phillips, Gengbin Zheng, Sameer Kumar, and Laxmikant V. Kalé. NAMD: Biomolecular simulation on thousands of processors. In *Proceedings of SC 2002*, Baltimore, MD, September 2002.

[Ric96] Jonathan Rice. Spatial bounding of self-affine iterated function system attractor sets. In *Graphics Interface*, pages 107–115, May 1996.

[RK99] Parthasarathy Ramachandran and L. V. Kalé. Mulitlingual debugging support for data-driven and thread-based parallel languages. In *Lecture Notes in Computer Science: Proc. of 12th International Workshop on Languages and Compilers for Parallel Computing (LCPC '99)*, pages 236–250. Springer-Verlag, August 1999.

[RSC87] William T. Reeves, David H. Salesin, and Robert L. Cook. Rendering antialiased shadows with depth maps. In *SIGGRAPH Proceedings*, pages 283–291. ACM Press, 1987.

[San96] Sanjeev Krishnan and L. V. Kale. A parallel array abstraction for data-driven objects. In *Proceedings of Parallel Object-Oriented Methods and Applications Conference*, February 1996.

[Sch98] Gernot Schaufler. Per-object image warping with layered impostors. In *Proceedings of the 9th Eurographics Workshop on Rendering '98*, pages 145–156, June 1998.

[SDB97] François Sillion, George Drettakis, and Benoit Bodelet. Efficient impostor manipulation for real-time visualization of urban scenery. *Computer Graphics Forum (Eurographics '97)*, 16(3):207–218, 1997.

[SG77] William Marshall Smart and Robin Michael Green. *Text-Book on Spherical Astronomy*. Cambridge University Press, 1977.

[SGHS98] Jonathan W. Shade, Steven J. Gortler, Li-Wei He, and Richard Szeliski. Layered depth images. *Computer Graphics*, 32(Annual Conference Series):231–242, 1998.

[SLS+96] Jonathan Shade, Dani Lischinski, David H. Salesin, Tony DeRose, and John Snyder. Hierarchical image caching for accelerated walkthroughs of complex environments. *Computer Graphics*, 30(Annual Conference Series):75–82, 1996.

[Sny87] John P. Snyder. Map projections–a working manual. *U.S. Geological Survey Professional Paper 1395*, 1987.

[SS96] Gernot Schaufler and Wolfgang Stürzlinger. A three dimensional image cache for virtual reality. *Computer Graphics Forum*, 15(3):227–236, 1996.

[Sto98] John Stone. An efficient library for parallel ray tracing and animation. Master's thesis, Dept. of Computer Science, University of Missouri Rolla, 1998. http://jedi.ks.uiuc.edu/j̃ohns/.

[Sur04] United States Geological Survey. The national map, 2004. http://seamless.usgs.gov/.

[TK96] Jay Torborg and James T. Kajiya. Talisman: commodity realtime 3d graphics for the pc. In *SIGGRAPH Proceedings*, pages 353–363. ACM Press, 1996.

[vdBLV96] Christian J. van den Branden Lambrecht and Olivier Verscheure. Perceptual quality measure using a spatio-temporal model of the human visual system. *Proceedings of the SPIE*, 2668:450–461, 1996.

[WH03] Chris Wyman and Charles Hansen. Penumbra maps: Approximate soft shadows in real-time. pages 202–207, 2003.

[WS99] Gregory Ward and Maryann Simmons. The holodeck ray cache: an interactive rendering system for global illumination in nondiffuse environments. *ACM Transactions on Graphics*, 18(4):361–368, 1999.

[WSB01] Ingo Wald, Philipp Slusallek, and Carsten Benthin. Interactive distributed ray-tracing of highly complex models. In *Proceedings of the EUROGRAPHICS Workshop on Rendering*, pages 274–285, June 2001.

Appendix A

Parallel Array Support

All computation in our parallel rendering backend is built using Charm++ Arrays, which this chapter describes in detail.

The contents of this chapter expand on work also presented in [LK03].

A.1 Introduction

A perennial problem in computer programming is naming—if we want something, how and who do we ask for it? This problem is especially acute on parallel machines, because communicating with remote processors is slow.

The situation is akin to the (physical) postal system. We wish to deliver a message to a person—how can we get the message to them? The imperfect solution adopted by the postal system is to use physical addressing—we somehow obtain, and then specify, the exact physical location where the person can be found. The problem, of course, is that when a person moves, their mail will either be forwarded (which is slow and duplicates delivery effort), misdelivered (delivered to the current occupant of that address), or lost (returned to the sender or dropped completely).

The well-known parallel messaging standard MPI[MPI94] also uses physical addressing—messages in MPI are sent to a particular process number and tag. Like the postal service, if the computational entity the message is destined for moves, the message will either have to be manually forwarded or (more likely) will be misdelivered or lost. This means computational entities in MPI programs either never move, or can only be moved with a great deal of intricate, error-prone programming effort.

While it is easy to deliver messages to a physical address, it is evident that moving objects are difficult to support. We present a simple, well-known solution to this problem—a layer of indirection between objects and physical addresses.

Our framework allows an object to be referenced by a globally unique, problem-domain, user-assigned logical address called an "array index". All communication is via the array index, which allows the object to be migrated in a completely general and user-transparent way.

In this appendix, we show how to support message delivery, creation, deletion, and migration scalably and efficiently using this logical addressing scheme. We present a solution to the problem of broadcasts and reductions in the presence of ongoing migrations. Finally, we present performance data from several actual applications built using this system.

A.1.1 Partition Decomposition

Many of today's emerging high-end parallel applications are characterized by irregular and dynamic computational structure. These problems demand techniques such as latency hiding and dynamic load balancing to achieve good parallel performance. However, it requires significant programming effort to incorporate these techniques into a parallel program written using the prevalent processor-centric programming model exemplified by MPI.

We abandon this processor-centric model for an object-centric model. We divide the computational work into a large number of parallel objects. Parallel objects resemble processors in that they are self-contained, and can send and receive messages. Unlike processors, however, they can logically addressed, created or deleted, scheduled dynamically, and migrated at run-time to improve load balance. Thus the programmer specifies which actions are to be computed in parallel, and the system decides when and where these actions execute.

Our parallel construct to support this approach is called a dynamic parallel object array, and is built on top of Charm++ [KK96]. Individual objects are called array elements, and can send and receive messages, participate in broadcasts and reductions, and migrate as needed. Each element of the array is identified by a unique array index, which may be variable-length. Because elements can be individually scheduled and migrated, an "object array" is very different from the "array objects" found in HPF, POOMA, P++, Global Arrays and elsewhere. In our construct, each element of the array is a relatively coarse-grained C++ object, with full support for remote method invocation. Unlike Concurrent Aggregates, Linda, or Orca, there is no duplication or replication—each message send addresses exactly one array element across the entire machine.

For example, a large dynamic structural simulation modeled using the finite element method may include 10 million elements in an unstructured mesh. Using our method, the application programmer may decide to partition this mesh into 5,000 chunks using a mesh partitioner such as METIS or Chaco. Each chunk is then implemented as a parallel array element, making a 5,000 element object array. Elements can then communicate with one another without knowing (or worrying about) which processor they reside on.

This approach, which we have been exploring for the past several years, has several advantages:

- As the number of elements is typically much larger than the number of processors, each processor houses several objects. This leads to an adaptive overlap of computation and communication—while one object is waiting for its data, another object can complete its execution. Scheduling is done dynamically depending on which message arrives first, so this latency hiding requires no additional effort by the programmer.

- Each element's data is small, so our approach can even improve performance on a single processor because of better cache utilization.

- Most important, however, is the logical addressing scheme used to deliver messages. With this approach, the run-time system is free to migrate objects across the parallel machine as it pleases, without affecting the semantics of the user program.

The run-time system can use this freedom to effect measurement-based load balancing, for example. During the computation, it can measure the load presented by each element, along with the element communication patterns. It can then remap the objects so as to minimize load imbalance and communication overheads. Even for dynamic applications, such measurement-based load balancing works effectively when load patterns change either slowly or infrequently. We have demonstrated this dynamic load balancer in a production-quality molecular dynamics system, NAMD [PZKK02].

Brunner and Kale [BK99] describe the application-independent dynamic load balancing framework, along with several applications. For example, if the parallel program is using idle desktop workstations, the run-time system can vacate the processors when their owners start using them. Time-shared clusters can also be supported efficiently, by shrinking or expanding jobs [KKD00] to the available set of processors using object migration. In the current work, however, we confine our attention to the underlying array construct and its implementation.

Our approach is implemented in Charm++ [KK93], a parallel library for C++. However, to allow the many existing MPI codes to use the load balancing and other facilities of Charm++, we have implemented AMPI [HLK03], an adaptive implementation of MPI, atop Charm++. In AMPI, an MPI process is virtualized as a migratable thread running in an array element. The system thus simulates multiple MPI processors on each real processor, allowing latency hiding and migration-based load balancing. Thus this research is applicable to the wide of class of parallel applications written using MPI as well. For more details on this process, with results from several significant independently developed scientific simulation codes written in Fortran 90 and C++, see [BKdSH01].

Charm++ already included some support for object arrays [San96]. We added to that work dynamic insertion and deletion, arbitrary bitvector indices, scalable migration, and reductions and broadcasts that function with ongoing migrations.

A.2 Motivating Examples

We consider several examples to motivate the design and features of parallel arrays.

A.2.1 Quadtree

Consider a simple heat flow simulation application that discretizes its domain with an adaptive 2D mesh. The mesh is implemented as a quadtree, as shown in Figure A.1. Using a separate array element for each leaf of the quadtree would likely result in a tiny grain size and poor performance, so each array element is the root of a small subtree of the mesh.

A quadtree admits a natural indexing scheme—the directions taken at each level of the tree from the root. Concatenating a binary representation of these directions results in a variable-length bit string that uniquely identifies a leaf of the tree. For example, the element labeled 10:11 in Figure A.1 can be reached from the outer box by moving to the lower-left (10) subbox, then to that box's lower-right (11) subbox. For addressing messages to these array elements, our arrays support variable-length indices.

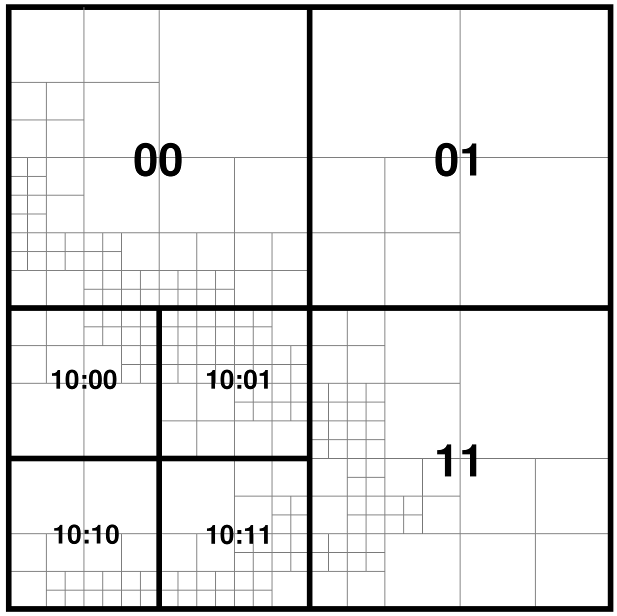


Figure A.1. Adaptive 2D quadtree mesh with seven elements, showing element array index and subtrees (in gray).

While running the program, the run-time load balancer collects[1] an object communication graph, as shown in Figure A.2, from instrumentation measurements built into the runtime system.

As the computation proceeds, elements will send each other messages to exchange temperatures with their neighbors. They will perform local calculations to propagate heat around their part of the mesh. In a steady-state problem, elements will occasionally contribute their local error values to a reduction to determine whether the convergence criteria have been satisfied. Once the convergence criteria have been met, the program will broadcast a "report results" message to all elements. The program may decide to create new array elements to refine an existing region. The program may delete array elements when coarsening a region. During the computation, the Charm++ run-time load balancer will migrate elements to improve the load balance. Clearly, the ongoing messaging, broadcasts, and reductions must continue to work even in the face of these migrations, creations, and deletions.

[1]The local portion of the object communication graph is collected independently on each processor. Load balancers then operate on the distributed graph.

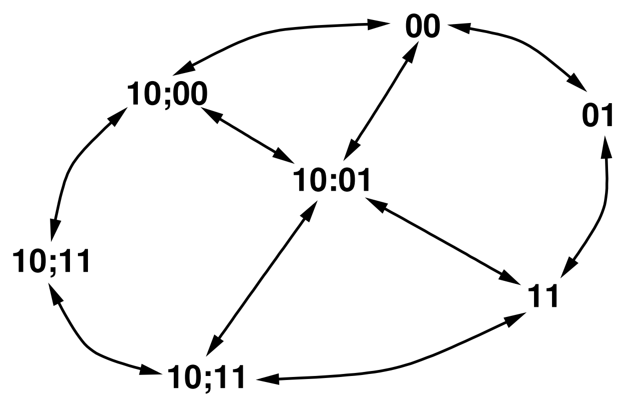


Figure A.2. Communication graph for example array.

A.2.2 Document Indexing System

Consider a parallel document indexing system. Each processor accesses a document and parses out a list of words found in the document. The document should then be linked into the document list for each word. In this case, the word itself (a character string) can be used as the array index, with the referenced parallel object storing the word's document list. The same word-indexed structure could be used to respond to queries.

New words (and hence array elements) will occasionally be encountered and created during the course of the computation. Over time, rarely-used words or misspellings could be deleted from the array. For load balance or better storage utilization, words could be dynamically migrated between processors. Occasionally, summary statistical information such as the average document list length or number of answered queries could be reduced over the entire array.

A.2.3 Collision Detection

Consider a "bucket-based" contact detection algorithm. Such a computation consists of a group of physical objects scattered through space, some of which may overlap. The problem is to find all overlapping pairs of objects. A slow algorithm is to simply consider all pairs of objects. A natural optimization is to first map objects into disjoint regions of space—buckets—and only consider pairs of objects that fall in the same bucket. Of course, a large object may cover several buckets.

If the buckets have a natural indexing (for example, if they form a quadtree or regular grid), then this can be used as the array index for the bucket. Then the collision detection algorithm is to send each object to the appropriate bucket, collide the objects in each bucket independently, and then collect the colliding objects across the array.

Once again, we find dynamic creation (when objects enter new regions of space), deletion (when no objects lie in a region), migration (for load balance), broadcasts (to begin the collision computation), and reductions (to collect the collisions) are all useful operations, and must all work together.

A.3 Message Delivery

These applications are difficult to support. In particular, the user may create an element at index 42 on some processor C, then send a message to it from processor A. A must be able to deliver the message despite the fact that A may never have communicated with C. Worse, 42 may migrate to some new processor D while the message is in transit.

Non-scalable methods for location determination are easy to imagine.[2] Processors could be required to broadcast the location of all new or migrated elements. This solution, however, would waste bandwidth and require every processor to keep track of every array element, which requires a non-scalable amount of storage. Alternatively, a central registry could maintain the locations of all array elements. This conserves bandwidth, but still may have enormous non-distributed storage requirements and also presents a serial bottleneck. Our solution conserves bandwidth, has modest storage requirements, and is well distributed.

A.3.1 Scalable Location Determination

To solve the location problem scalably, the system can map any array index to a home,[3] a processor that always knows where the corresponding element can be reached. The default index to home function simply returns the hashed array index modulo the number of processors; but we also support user-defined home functions. An element need not reside at its home processor, but must keep its home informed of its current location. In the example above, A will map the index 42 to its home processor B, which will know that 42 is currently living on processor C.

Thus, A sends its message to the home of 42, B, who then forwards the message to C. Since this forwarding is inefficient, C sends a (short circuit) routing update back to A, advising it to send future messages for 42 directly to C. This messaging and routing update is shown in Figure A.3.

[2] And frequently implemented in real code!

[3] This same concept is used in many Distributed Shared Memory (DSM) implementations.

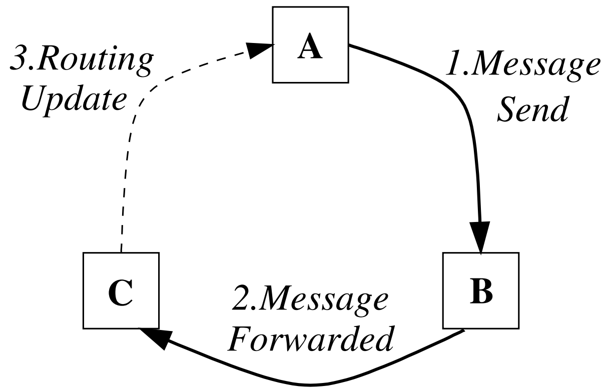


Figure A.3. Message forwarding among processors: A, the source; B, the home; and C, the destination.

Since elements and homes are scattered across the machine, most forwarded messages must cross the machine twice, wasting cross-section bandwidth. The forward-free alternative—A asks B where to send, B replies, A sends directly to C—may use less total bandwidth for large messages, but requires an additional hop in the critical path. Forwarding also generalizes more smoothly to the migration case. Finally, the forwarding approach works quite well when the element actually lives at its home. With either approach, the common case of repeated communication quickly settles to one hop—that is, with no additional communication overhead.

A simple generalization of this scheme is to use k separate mappings to assign k homes to each element.[4] Several homes allow messages to be forwarded via any[5] home, saving cross-machine bandwidth, but also requires elements to inform k processors when they are created or moved. With $k = p$, every processor knows the location of every element, eliminating forwarding; but creations, migrations, and deletions all require a broadcast. The best value of k depends on the relative frequency of message forwarding and creations, migrations, and deletions.

A.3.2 Creation

To create an element, the system need only inform the element's home and call the element constructor. If no processor is specified, the element is created by default at its home processor, which eliminates later message forwarding. It is an error[6] to attempt to create two elements at the same index.

[4] With multiple homes, one could imagine creating a system that could survive a node failure.

[5] Typically the home the fewest hops away, or the least loaded.

[6] We can easily detect this error when the element's home processor is notified of the second creation.

A message may arrive for an element before the element has been created—this could occur because the create message was delayed on the network. In this case, the system buffers these early messages at home until the element is finally created.

For other applications, a message that arrives for a nonexistent element should result in the creation of the element. For example, in a document indexing system, a document containing a new word that has no corresponding array element will send a "link to me" message using that word as an index. Since no processor has any record of an array element at that index, the message will be forwarded to the home processor for that index, which will recognize that the corresponding element does not exist.

Based on the application's desired semantics, a new array element could be created at that index to handle the message. The new element could be created on any processor. However, it is often most efficient to create the element either at the home processor ("createhome"), where future messages from other processors will be directed; or on the sending processor ("createhere"), which may soon send other messages for the element.

The application specifies the desired early-arrival semantics—buffering or create-on-demand—on a per-method basis in the interface file.

A.3.3 Deletion

To delete an element, the system invokes the object's destructor and informs the element's home processor. No other processors are informed. Any routing cache entries on other processors will remain unused until they eventually expire and are deleted.

More complex methods to reclaim deleted element routing cache entries could be used. When deleting an element, a processor could broadcast a deletion notice. Rather than broadcast, elements could keep track of, and only inform, processors that may have cached their location. A "deleted list" could propagate through the system at a low priority. These alternatives are all reasonable, but all use network bandwidth; simple expiration can be completely local and quite efficient. It is an error for a message to be sent to a deleted element. However, a new element is allowed to reuse an array index vacated by a deleted element.

A.3.4 Migration

Migration is always under user control—either explicitly, via a "migrate" call; or implicitly, by enabling run-time load balancing. To migrate an element, the system packs it into a message, sends it to its new location,

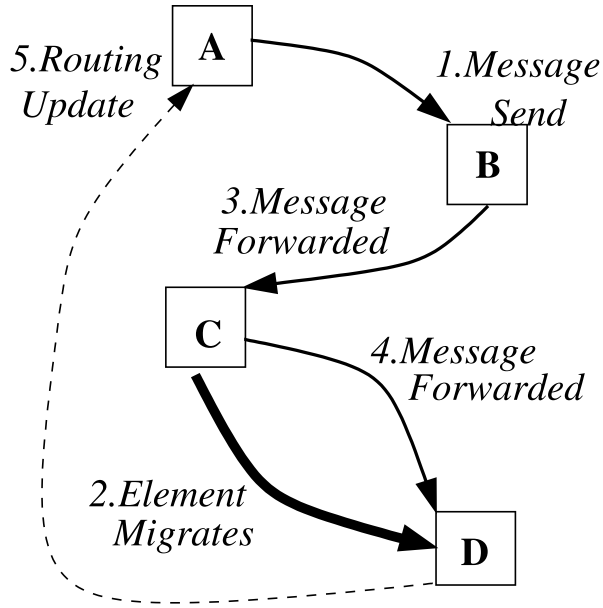


Figure A.4. Delivery may require several hops during an element migration.

and informs the element's home processor of its new location. A message that arrives for a departed element is forwarded to its last known location, with the usual short circuit routing update (just as in Section A.3.1) once it arrives. If an element migrates rapidly and repeatedly, messages may be forwarded an arbitrary number of times (see Figure A.4). Of course, migration is normally infrequent, so this pathological case is rare.

Processors that may have cached a migrator's old location are not informed of the migration. Any stale routing cache entries will be updated when the next message is sent. This lazy update prevents unnecessary traffic and keeps migrations fast. The alternative, to inform all others of your current location after each move, saves time on the first message at the cost of significantly more expensive migration.

Of course, for the common case of repeated communication with stationary elements, the system quickly settles to a single hop.

A.3.5 Protocol Diagram

Each processor must keep track of each array's local and home elements, as well as maintain a routing cache of "last-seen" locations. All this information can efficiently be kept in a per-processor, per-array hash table, keyed by the array index.

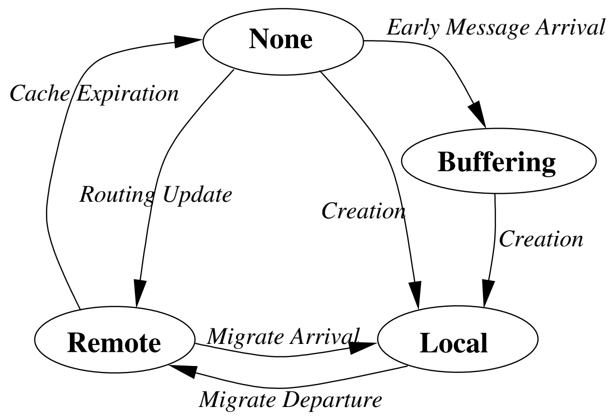


Figure A.5. Finite state machine that represents an array element on one processor.

To deliver a message addressed to an array index, the system looks the index up in its hash table. The represented element will be in one of these states:

- Local: the element is on this processor. Messages are delivered directly to the element.
- Remote: the element was last seen on another processor; i.e., we have a routing cache entry. Messages for the element are forwarded to that processor. Non-home remote pointers expire if they remain unused for too long.
- None: this processor has no idea where the element is located—the element is not listed in the hash table. Messages for such elements will be sent to their home; or if this is the home, buffered.
- Buffering: this processor has messages queued for the element, but the element has not yet been created, e.g., because the element creation message has not arrived yet. This state is only used on an element's home processor.

The element state can change according to the transitions of the finite state machine of Figure A.5.

A.4 Collective Operations

In addition to communicating point-to-point, array elements often need to participate in global operations such as broadcasts and reductions.

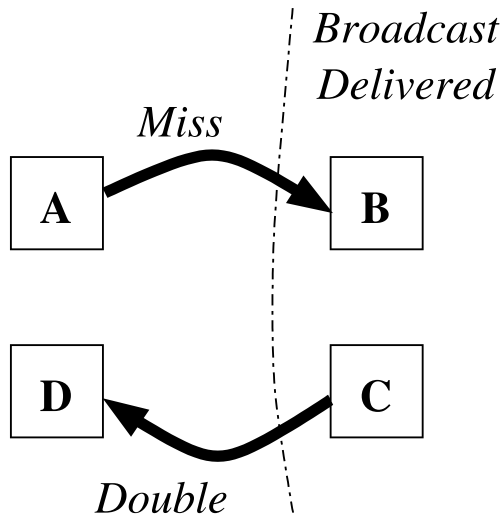


Figure A.6. Broadcast delivery errors. Processors A and D have not received the broadcast; processors B and C have.

A.4.1 Broadcasts

The semantics of a broadcast are that every existing array element will receive each broadcast message exactly once. Since processors have no shared clock, "existing" means created but not destroyed at the instant the broadcast is received on that processor. Array broadcasts are thus first sent to each processor, then delivered to each processor's current local elements. However, this is not enough if there are ongoing migrations.

For example, consider the case where a migrating element leaves processor A before the broadcast is delivered, and arrives on processor B where the broadcast has already been delivered. The migrator may miss the broadcast. Or, reversing the situation, an element may receive a broadcast on processor C, then migrate to D where the broadcast has not yet arrived. When the broadcast reaches D, the migrator may erroneously receive the broadcast again, as shown in Figure A.6.

To solve these problems, the broadcasts are serialized,[7] and processors and elements each maintain a broadcast count. When an element is created, it takes the creating processor's broadcast count.

[7] A broadcast, by definition, must reach every node. Serializing the broadcasts via a single node thus involves little additional cost.

To prevent duplicate delivery, when a broadcast arrives the system compares its count with each element's broadcast count. The system delivers the broadcast only if the count indicates the element has not yet received that broadcast.

To prevent missed broadcasts, the system maintains a buffer of past broadcasts. When an element arrives from migration, the system again compares its broadcast count with the element's. If the element missed any broadcasts while migrating, the element's count will be too low, and the element is brought up to date from the broadcast buffer. Old entries in the broadcast buffer will eventually become useless and must be reclaimed. [8]

Broadcast semantics are easy to enforce when an element is deleted—simply stop delivering broadcasts to the deleted element. When an element is created, it should receive all broadcasts that arrive at its birthplace after its creation; so a new element's broadcast count is initialized with the local processor's broadcast count.

A.4.2 Reductions

A reduction combines many values scattered across a parallel machine into a single value. A reduction function defines what "value" means and performs the combination. The semantics of a reduction are that each existing element will contribute exactly one value, and the reduction function will be applied to these values in an unspecified order.[9] As before, "existing" means created but not deleted at the time the local reduction completes. Of course, other work may proceed during the reduction.

Reductions can be implemented efficiently by first reducing the values within each processor (the local reduction), then reducing these values across processors. As with broadcasts, in the presence of migration this simple algorithm is not enough.

The problem is that during the time a migrating element is in transit, it belongs to no processor.[10] That is, the source processor cannot wait for the migrator's contribution because it already left; while the destination processor cannot know it is on the way, as shown in Figure A.7. Thus the source and destination processors might both complete their local reductions, missing the migrator. However, the reduction must wait until all elements, even migrators, have contributed.

One sensible solution is to count the number of contributed values as the reduction data is collected, and not allow the reduction to complete until the number of values matches the number of elements. Unfortu-

[8] This currently involves a timeout, but much better solutions are clearly possible.

[9] If the order matters, one can use the list-making reduction function to collect the data first.

[10] Note that sending another, separate message for synchronization only pushes the problem back one step.

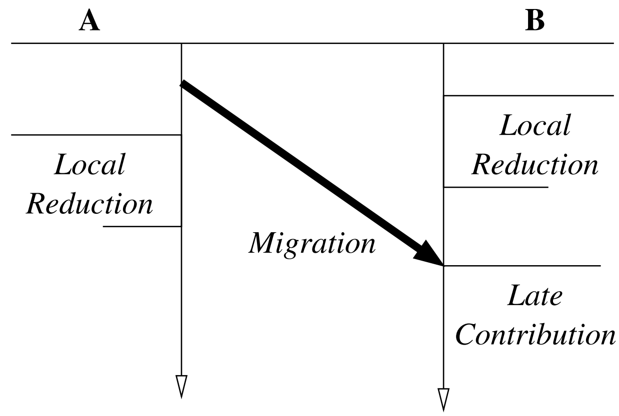


Figure A.7. Timeline: reduction skips a migrator. We must ensure the migrator's contribution is included.

nately, the total number of elements is not available on any processor; and a simple sum of the local element counts will still miss migrating elements.

The approach we use is to sum the net births—the total number of elements created on a processor minus the total destroyed on that processor.[11] Because of migration, this number may be negative if elements often migrate in and are destroyed (e.g., on "graveyard" processors).

Since for each processor p, the net births n_p is defined as:

$$n_p = c_p - d_p$$

Thus summed across all processors:

$$\sum_p n_p = \sum_p (c_p - d_p) = \sum_p c_p - \sum_p d_p$$

Summed across all processors, then, we have the total number of elements created but not yet deleted, which is the global element count.

Thus the reduction algorithm actually used is:

1. At each processor, collect contributed values from local elements until all current local elements have contributed. As with broadcasts, we use a per-element and per-processor reduction count to determine which elements still need to contribute. At that point, apply the reduction function to the collected values and add the result, contribution count, and the current net births to the across-processor reduction.

[11] The net births count is measured at the instant the local reduction completes.

2. Reduce the values and add the contribution count and net births across all processors.

3. As migrators make their late contributions, send their values directly to the root. Once the contribution count equals the total net births, return the reduced value to the user.

The reduction semantics are also slightly more difficult to enforce in the presence of creations and deletions. Element creations are relatively easy—the net births counter is incremented and the element receives the reduction count of the local processor.

If an element is deleted that has not yet contributed to the current reduction, we simply decrement the net births counter. If the element has already contributed, we must ensure it is included in the net births count for this reduction; but for future reductions it is not included. This is easy to implement with a simple delayed net births adjustment.

A.5 Performance

We have extensively analyzed the performance of the array support, as summarized below.

A.5.1 Theoretical

Notation

- p the number of processors on the parallel machine
- n the total number of array elements
- l_i the number of local array elements on processor i
- r_i the number of remote elements recently referenced by processor i
- h_i the number of elements with processor i as their home

Element creation and deletion, since they only involve the current processor and the element's home, require constant time and 1 message. Migration requires constant time and 2 messages[12]. Message delivery may require an unbounded number of messages, but only if the element migrates as fast as the message travels. Repeated messages to stationary elements take constant time and 1 message.

[12] One message transports the element, one updates the home processor's routing table.

The local, element-wise operations during reductions and broadcasts require time in $O(l_i)$ on processor i. Without migration, the cross-processor phase of a broadcast or reduction tree requires $p-1$ messages and completes in $\lceil \log_b p \rceil$ hops, where b is the tree branching factor (typically 2 to 16).

The storage consumed by the element hash table on processor i is in $O(l_i + r_i + h_i)$. If each element communicates with a bounded number of other elements, r_i is in $O(l_i)$. If elements and home processors are distributed relatively uniformly, l_i and h_i will both be near n/p. Subject to these assumptions, each processor's hash table requires storage in $O(n/p)$. In the worst case, l_i, r_i and h_i are all bounded by n, so the storage is still in $O(n)$.

A.6 Conclusions

We have presented efficient support for a general logical addressing scheme for parallel objects. The address, called an array index, is a user-defined structure, supporting multidimensional, sparse arrays as well as structures such as trees. Objects may be efficiently created, deleted, or migrated at any time; and even in the face of these operations, the system supports array-wide broadcasts and reductions efficiently. This system has proved to be a robust and useful foundation for several significant applications.

Appendix B

Network Communication Support

Our serial display client, written using OpenGL, connects to the parallel rendering program across the network using a protocol called Converse Client/Server (CCS). To extract program state, and represent impostors on the wire, both the client and server use the PUP framework to describe their C++ objects.

The contents of this chapter expand on work also presented in [JLK04].

B.1 CCS network interface

The Converse Client-Server (CCS) network interface [KBJ+96] enables Converse (and hence CHARM++) programs to act as parallel servers, responding to requests from the network. The server side of this interface is built into every CHARM++ program, and the client side is provided as a library for C and Java.

A CCS client, in this case the OpenGL client, connects to the server via a TCP connection and sends it a request, which consists of a string handler name and a block of binary request data. The CHARM++ runtime uses the handler name to look up and call the appropriate handler function from an extensible table. The handler function can then respond to the client with another block of binary data. This simple request/response protocol allows information to be injected into and extracted from a running parallel program.

For example, when the client sends the request name "lv3d_newViewpoint" with a new client view frustum, the parallel backend broadcasts this viewpoint to all affected geometry. When the client sends "lv3d_getViews", the server responds with any generated impostor images.

B.1.1 Network Protocol

CCS uses a simple request-response protocol identical to HTTP. The client opens a TCP connection to the server and sends a text request name and an arbitrary sequence of binary request data. The server processes the request, and sends back an arbitrary sequence of binary response data.

The previous version of CCS [RK99] used a slightly different protocol. Previously, to send response data back to the client, the server made a separate connection back to the client. This approach had the following disadvantages over the new version:

- Because two separate connections were made per request, for small requests the older protocol took approximately twice as much time.
- Because both client and server made connections to each other, neither one could be behind a firewall or NAT router. In the modern version, because the client opens the only TCP connection for a CCS request, CCS can be used by firewalled or NAT clients.
- Because the client had to create and listen on a server socket, it could not run in a sandbox. The modern CCS protocol allows clients to run from a sandboxed Java applet.
- Both client and server had significant security holes. In particular, it was possible for a malicious client to use the server to make connections to arbitrary third-party nodes, by simply listing the third party's IP and port number as the response address.

When CCS is running over the unsecured internet, it can be run in a secure authentication mode[Par04b], which attaches to each request and response an authentication ticket consisting of the SHA-1 hash of the request, a counter, a per-client nonce, and a shared secret key. The hash is computed both on the sender and receiver, and if they do not match, the message is considered a fabrication. Sending only the hash of the secret key keeps the key secret. The randomly generated nonce has both client and server portions to prevent man-in-the middle attacks on either client or server. The counter prevents message replay attacks. CCS authentication thus prevents arbitrary clients from injecting messages, but because of export regulations we do not also provide network encryption.

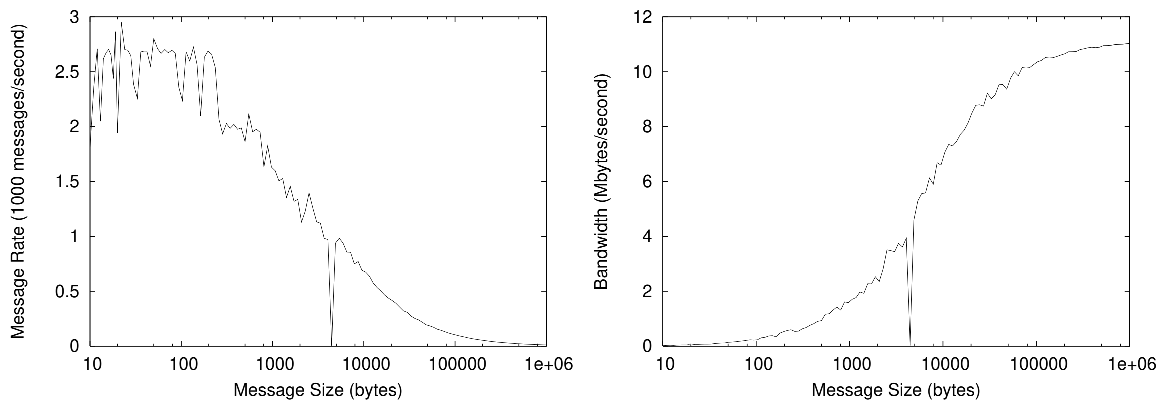


Figure B.1. Delivered CCS performance, in thousands of complete exchanges and millions of transmitted bytes per second, over fast ethernet.

B.1.2 CCS Performance

We have measured the actual performance of CCS in detail. Figure B.1 shows the delivered performance of CCS exchanges between two local Linux machines connected with fast ethernet, with a 174us round-trip IP ping time. For small messages, up to a kilobyte, the performance is connection-setup dominated, but still achieves over 2,000 messages per second. For large messages, over 100 kilobytes, throughput is limited by the network bandwidth. With long messages, CCS achieves better than 11 megabytes per second over the 100mbit link.

For our impostor based parallel rendering system, the parallel server sends back 100KB of impostors at a time. This is a small enough block of data that it adds only about 10ms of latency, but large enough that it has a chance to utilize the network bandwidth.

B.2 CHARM++ PUP framework

The PUP framework is a method to describe the in-memory layout of an object, and was originally designed to support object migration in CHARM++. To copy a complicated object from one processor to another, we must pack the object into a network message, ship the message to another processor, and finally unpack the message into an object on the other side. This is an extremely common operation, used in Java RMI serialization/deserialization, parameter marshalling and unmarshalling for CORBA communication, and even MPI derived datatypes. PUP stands for Pack/UnPack, and is a compact, efficient, and flexible method to perform this packing and unpacking of user objects for C++.

Because of the type safety and introspection capabilities of the Java language and virtual machine, Java can pack and unpack arbitrary objects automatically, without any further effort. CORBA requires the user to describe the format of each communicated object in a CORBA IDL file, which is preprocessed to generate pack and unpack code. MPI requires the user to build a “derived datatype” at runtime, using type construction library calls that list each field of each communicated object; because this is complicated, users often write explicit packing and unpacking code to ship complicated objects.

CHARM++ originally required users to write an explicit pack and unpack routine for each object, as well as a size routine to determine the outgoing message size before packing. The motivation for PUP is that the code used to size the message, pack an object into a message, and unpack an object from a message must match up exactly—everything that is packed must be unpacked, and vice versa. Writing three interrelated routines for every object is tedious, error-prone, and contributes to the burden of parallel programming.

In the PUP framework, sizing, packing, and unpacking are all is controlled by a single user-written subroutine called a *pup* routine. The *pup* routine simply calls a virtual method on each of the object’s fields, which are then sized, packed or unpacked as appropriate.

Consider a very simple C++ class with three fields:

```
class foo {
    int A;
    float B;
    long C;
public:
    ...
};
```

We define an abstract class named “PUP::er” with one virtual method named “bytes”, which takes the address of an object field and a description of the type of data in the field. A *pup* routine for foo would then just pass each of the foo object’s fields into the PUP::er.

```
void foo::pup(PUP::er &p) {
    p.bytes(&A,MPI_INT);
    p.bytes(&B,MPI_FLOAT);
    p.bytes(&C,MPI_LONG);
```

```
}
```

Because the PUP::er is given the address and data type of each of the objects' fields, it can perform arbitrary manipulations of those fields, including copying data into the fields, copying data out of the fields, or even building an MPI derived datatype using the field offsets.

```
// Compute total size of object fields
class SIZING_PUP_er : public PUP::er
{
public:
   int totalsize; // size of object
   SIZING_PUP_er() {totalsize=0;}

   virtual void
    bytes(void *field,int datatype) {
      totalsize+=size(datatype);
    }
};

// Copy data out of object fields
... define destbuf as PUP::er field ...
void PACKING_PUP_er::
bytes(void *field,int datatype) {
    memcpy(destbuf,field,size(datatype));
    destbuf+=size(datatype);
}

// Copy data into object fields
... define srcbuf as PUP::er field ...
void UNPACKING_PUP_er::
bytes(void *field,int datatype) {
```

```
    memcpy(field,srcbuf,size(datatype));
    srcbuf+=size(datatype);
}

// Build an MPI derived datatype
void MPI_DATATYPE::
bytes(void *field,int datatype) {
   displacements[n]=field-objbase;
   datatypes[n]=datatype
   n++;
}
```

It should be clear the very simple technique of calling a virtual method for each field of an object is quite powerful. CHARM++ actually uses the first three PUP::ers above to size network messages and copy data into and out of objects as they are sent across the network. The overhead for using the very general pup method to do the copy is exactly one virtual function call per field, which on many machines is faster than the memory copy itself. Other PUP::ers, not shown here, can read and write objects to and from disk, or even convert binary data formats between different machine architectures.

B.2.1 PUP operator

However, the application code must tediously pass to the "bytes" routine the address and datatype of each field of each object. Luckily, we can use C++ operator overloading to automatically extract the datatypes, and also to provide a simpler syntax:

```
void operator|(PUP::er &p,int &x)
{ p.bytes(&x,MPI_INT); }
void operator|(PUP::er &p,float &x)
{ p.bytes(&x,MPI_FLOAT); }
... and so on for other datatypes ...

void foo::pup(PUP::er &p) {
```

```
    p|A; // calls p.bytes
    p|B;
    p|C;
}
```

Users can treat operator| as a builtin operator, analogous to the << and >> C++ iostream operators. This operator overloading also provides a surprising benefit: we can now use the same syntax to pup user-defined classes that we use for builtin types like "int".

```
void operator|(PUP::er &p,foo &x)
    { x.pup(p); }

class bar {
    int I;
    foo F;
    ...
}
void bar::pup(PUP::er &p) {
    p|I;  // calls p.bytes
    p|F;  // calls foo::pup
}
```

Notice how C++'s operator overloading selects the appropriate way to pup the two fields I and F, even though the operator| call looks identical.

Operator overloading can also be applied to pup templated classes, with the template type determined by C++ type resolution. For example, we can easily define a pup operator for the standard C++ class std::vector. The elements of the vector are pup'd using their own pup operator, so they can be of any type.

```
template<class T>
void operator|(PUP::er &p,
              std::vector<T> &v)
{
```

```
    int length=v.size();
    p|length;
    p.resize(length);
    for (int i=0;i<length;i++)
        p|v[i];
}
```

This std::vector pup operator shows some of the strange beauty of using a single routine for both packing and unpacking. While packing, the length of x is known, the "p|length" call stores the length, and the "resize" call specifies the current size and hence does nothing. While unpacking, the length is initially zero, "p|length" extracts the true length, and the "resize" call actually allocates space in the vector for the new elements.

Because operator overloading follows the type system, we can now pup an array of ints, std::vector<int>; or a 2D array of foo objects, std::vector<std::vector<foo> >, using the same "p|x" syntax used to pup plain ints. CHARM++ includes builtin pup operators for std::vector, std::list, std::string, std::map, and std::multimap, templated over any object with a pup operator. PUP thus uses C++'s sophisticated type and template overloading system to achieve an effect similar to the type introspection ability built into Java.

B.2.2 PUP subclasses

The PUP framework as described above does a good job of traversing the fields of an object of a known type; but often we need to create C++ objects when the exact type is not yet known. For example, the client and server both describe an impostor using a concrete subclass of the abstract "interface" class CkView.

The server can easily pup its specific instance of the CkView to send to the client, by calling a virtual pup routine. But the client does not yet have an instance of the subclass of CkView, so the client does not yet have a pup routine to call. Worse, because different imposters are used in different applications and different places, the client does not know *a priori* which subclass of CkView it should instantiate.

Thus, the server needs to somehow communicate to the client which subclass to instantiate. The PUP framework provides a uniform, easy to use system to do this called PUP::able. A PUP::able superclass provides a virtual "ID" method that subclasses can use to expose their true subclass type. The server then stores the PUP::able ID, which identifies the subclass, before calling the subclass pup routine.

The client can then use the ID to instantiate the appropriate subclass before unpacking. The runtime support needed is simply a table mapping ID to a construction function, a function pointer that instantiates a new object of the appropriate subclass type. Client and server share a single registration system to assign class ID's and construction functions.

The PUP::able subclass identification system is used to communicate impostors, which have a single virtual method to perform drawing; as well as the overall drawable "universe", which has virtual methods for drawing, accepting new impostors from the network, and responding to user navigation input. This system allows client/server systems to enjoy the software engineering benefits of separating generic abstract interface classes from application-specific subclasses.

Appendix C

Bounding Iterated Function Systems

This section expands on our published work [LH03], which relates to bounding iterated function systems. We use iterated function systems to represent trees and other foliage in our campus model. We need good bounding volumes for this procedurally generated geometry in order to fit tight impostor textures, and we need to compute those bounds quickly because there are over 10,000 trees in the campus model.

C.1 Introduction to Iterated Function Systems

An Iterated Function System (IFS) consists of a finite set of functions w_m, which move points around in some space—typically, the functions w_m map $\mathbb{R}^n$ to $\mathbb{R}^n$. For example, the classic Sierpinski gasket's IFS, shown in Figure C.1, consists of three maps that contract 2D space by a factor of two towards the points $(0,0)$, $(1,0)$, and $(0.5,1)$.

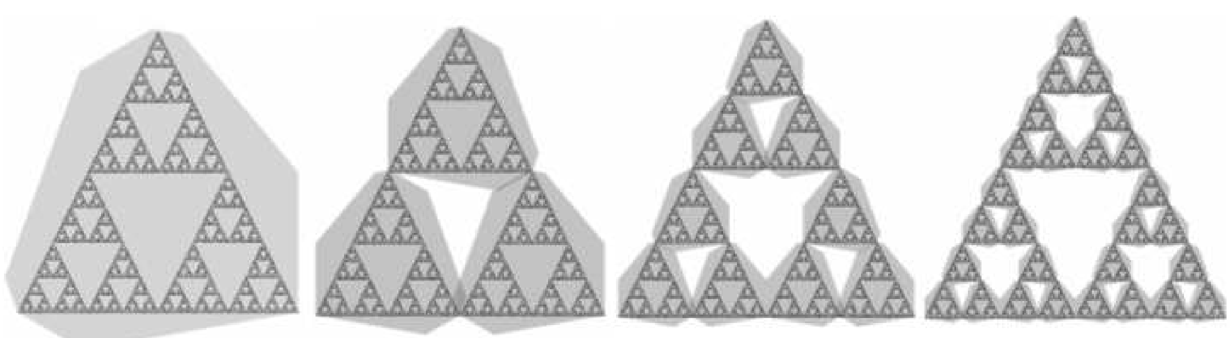


Figure C.1. Sierpinski's gasket, a classic IFS. If a bounding volume contains its images under the maps of an IFS, then it contains the attractor of the IFS.

The Hutchinson operator W maps a set of points to another set of points, and is defined as the union of each of the maps w_m. That is, given a subset of space B,

$$W(B) = \bigcup_{m \in M} w_m(B)$$

Under certain conditions on the w_m,[1] it can be shown that repeated applications of W always converge to a unique attractor A. That is, there exists a set A such that, starting with any bounded nonempty set B, $W(W(...(W(B))) = W^\infty(B) = A$. Furthermore, A is invariant under W—that is, $W(A) = A$. Convergent IFS, with a well-defined attractor, are the only IFS that will concern us.

We can now restate a recursive bounding theorem [Hut81, HD91], which states that if a shape bounds its image under the operator W, then the shape bounds the attractor, as demonstrated in Figure C.1.

Theorem 1. *Let B be a non-empty compact subset of $\mathbb{R}^n$ and let W be the Hutchinson operator of a convergent IFS on $\mathbb{R}^n$. If $W(B) \subset B$ then $W^\infty(B) \subset B$.*

Proof: A simple proof proceeds by induction on applications of W.

The base case is trivial since $W(B) \subset B$.

For the inductive step, assume for some k, $W^k(B) \subset B$ and let $C = W^k(B)$. The subset property $C \subset B$, is preserved in the image of each of the IFS maps $w_m(C) \subset w_m(B)$, and also in their unions

$$W(C) = \bigcup_{m \in M} w_m(C) \subset \bigcup_{m \in M} w_m(B) = W(B) \subset B$$

Thus $W(C) = W(W^k(B)) = W^{k+1}(B) \subset B$.

By induction, we have $W^\infty(B) \subset B$. □

C.2 Bounding Iterated Function Systems

Our basic approach will be to bound the IFS attractor using a bounding hull built from the intersection of a set of halfspaces. We will begin with a coarse bound constructed using existing techniques, then perform a refinement process.

We begin by computing a recursive sphere bound using the technique described by Hart [HD91], but with the matrix fixed-point acceleration of Rice [Ric96]. From this coarse recursive bounding sphere, we compute the positions of a set of even coarser (non-recursive) bounding halfspaces and then refine.

[1] A sufficient condition is that each w_m be Lipschitz contractive.

C.3 Bound Refinement for Iterated Function Systems

Bound refinement is the process of taking a loose bound and making it tighter. Our expression of the refinement process begins with a single (bounding) halfplane. The refinement process seeks to move that halfplane as far as possible without hitting the IFS attractor. To do this, we repeatedly replace a bound with its maps, resulting in a finer and finer approximation of the attractor. A complete bounding volume can then be extracted, one halfspace at a time, by separate searches for these outermost deeply-nested bounds.

We find this coarse-bound-and-refine strategy is both faster and more reliable than our previous published method based on convex optimization [LH03].

We describe and analyze three bound refinement schemes. For each scheme, we measured the volume of the resulting bound as well as the time taken to compute the bound. Free parameters for each type of refinement are the maximum depth to which we evaluate the IFS, and the number of times we repeat the bounding process.

- A heap-based search, ordered by the distance from the bounding halfplane. We use the heap to find the current outermost bound, and then recursively expand it. This method is the bounding equivalent of Hart's original IFS raytracing method [HD91], and is guaranteed to find exactly the same solution as an exhaustive search. The performance of this scheme depends on the IFS, but is generally nearly linear in depth. This scheme is also easy to implement—pseudocode for this search method is shown in Figure C.2.

- An exhaustive search over all possible maps: recursively expands all maps out to the search depth. Takes time which is exponential in the search depth, but is trivial to implement and guaranteed to find the smallest bound for a given depth.

- A greedy search, where at each level we only expand the outermost bound of the previous level. This is the algorithm described by Chu and Chen [CC03], and takes time linear in the search depth. This search is also fairly easy to implement, but care must be taken to include the non-outermost bounds, because during refinement a bound that was not outermost, and hence skipped over, may become outermost as we evaluate other bounds more finely. Because of these skipped bounds, the greedy algorithm's bounds are not as tight as those of exhaustive search.

Volumes for these three refinement schemes, for various search depths and number of refinement repetitions, are shown in Table C.1. As expected, heap and exhaustive search have identical volumes, and greedy bounds are never smaller, and often substantially larger. Times for each refinement scheme are shown in Table C.2. Surprisingly, for shallow depths, exhaustive testing is slightly faster than heap-based search even when it opens more volumes, because the overhead of maintaining the heap is relatively high. Overall, exhaustive search is good for small depths but becomes unusably slow even at moderate depths; greedy search is always fast but never finds tight bounding volumes; and heap-based search is fairly slow for small depths, but is the fastest way to obtain tight bounds at high depths.

The initial bounding volume is represented by its vertices, which makes searching for the outermost point on the bounding volume a simple "max" operation over the plane distance for the bounding volume vertices.

We find it is much faster if, during the search, instead of actually moving a parent bound's vertices under the maps of the IFS, we instead simply move the search plane by the inverse map and leave the vertices stationary. This is advantageous because the search plane (one vector and one scalar) is smaller and faster to map than the vertices (many vectors).

		Repeat 1	Repeat 2	Repeat 4	Repeat 8
Depth 1	Heap	24.55	24.55	24.55	24.55
	Exhaustive	24.55	24.55	24.55	24.55
	Greedy	24.55	24.55	24.55	24.55
Depth 2	Heap	19.03	17.12	16.22	16.10
	Exhaustive	19.03	17.12	16.22	16.10
	Greedy	21.27	19.32	17.46	16.56
Depth 4	Heap	9.24	7.92	7.57	7.55
	Exhaustive	9.24	7.92	7.57	7.55
	Greedy	17.22	14.38	12.00	11.03
Depth 8	Heap	5.88	5.67	5.66	5.66
	Exhaustive	5.88	5.67	5.66	5.66
	Greedy	17.22	14.30	11.92	10.96
Depth 16	Heap	5.48	5.48	5.48	5.48
	Greedy	17.22	14.30	11.92	10.96
Depth 32	Heap	5.47	5.47	5.47	5.47
	Greedy	17.22	14.30	11.92	10.96
Depth 48	Heap	5.47	5.47	5.47	5.47
	Greedy	17.22	14.30	11.92	10.96

Table C.1. Volume after refining an IFS.

		Repeat 1	Repeat 2	Repeat 4
Depth 1	Heap	12.8 us (144)	24.9 us (288)	49.0 us (576)
	Exhaustive	2.5 us (108)	4.3 us (216)	7.9 us (432)
	Greedy	2.9 us (108)	5.1 us (216)	9.5 us (432)
Depth 2	Heap	30.0 us (324)	58.8 us (648)	116.8 us (1296)
	Exhaustive	12.2 us (432)	23.7 us (864)	46.3 us (1728)
	Greedy	10.0 us (216)	19.2 us (432)	37.6 us (864)
Depth 4	Heap	75.1 us (1116)	129.5 us (1836)	234.4 us (3204)
	Exhaustive	63.5 us (4320)	126.8 us (8640)	253.6 us (17280)
	Greedy	13.2 us (432)	25.6 us (864)	50.3 us (1728)
Depth 8	Heap	307.4 us (4698)	458.3 us (6894)	751.3 us (11142)
	Exhaustive	4699.8 us (354240)	9360.6 us (708480)	18714.3 us (1416960)
	Greedy	19.7 us (864)	38.9 us (1728)	77.7 us (3456)
Depth 16	Heap	1245.5 us (17406)	1686.7 us (23724)	2566.5 us (36360)
	Greedy	32.2 us (1728)	64.9 us (3456)	130.4 us (6912)
Depth 32	Heap	3304.9 us (40824)	4294.9 us (54126)	6272.6 us (80730)
	Greedy	58.4 us (3456)	118.2 us (6912)	236.9 us (13824)
Depth 48	Heap	5317.9 us (63450)	6898.1 us (83646)	10043.6 us (124038)
	Greedy	83.1 us (5184)	169.2 us (10368)	342.3 us (20736)

Table C.2. Time taken in microseconds and (bounds opened) during refinement of an IFS.

```
double planeSearch(IFS, initBound, searchPlane, maxDepth)
{
  heap<BoundImage> H(searchPlane);
  H.put(initBound);
  while (H.not_empty())
  {
    BoundImage O=H.getOutermostBound();
    if (O.depth == maxDepth) break;
    for (each map m of the IFS)
      H.put(O under map m);
  }
  return O.extent(searchPlane);
}
```

Figure C.2. Pseudocode for the bound refinement process, using a heap-based lazy evaluation scheme. This search finds the outermost point (relative to an output bound search plane) of an initial bound under repeated maps from the IFS.

Appendix D

Spherical Harmonic Projection

We compute indirect lighting beforehand, and store, for each voxel of space, the lighting for a diffuse surface represented as low-order spherical harmonics. This section describes how we convert our per-direction lighting into a diffuse surface lighting.

D.1 Function Projection

Many functions are defined as a sum of products of scalar coefficients f_i and basis (or shape) functions F_i, or

$$f(x) = \sum_i f_i F_i(x)$$

Here, we seek the function f that best approximates an arbitrary function g—that is, we want to project g onto the space of functions f. For example, given the exact lighting g, we might want to project onto the space f of low-order spherical harmonics.

A least-squares approximation of g is equivalent, as also discussed by Jiao [Jia01], to imposing the Galerkin condition with test functions equal to f's basis functions. The Galerkin condition requires the approximation error to be orthogonal to each of the basis functions, or

$$\forall j \;\; 0 = \int_{x=-\infty}^{\infty} (f(x) - g(x))\, F_j(x)\, dx$$

$$\forall j \;\; \int_{x=-\infty}^{\infty} f(x) F_j(x)\, dx = \int_{x=-\infty}^{\infty} g(x) F_j(x)\, dx$$

Expanding the definition of f, we find

$$\forall j \ \sum_i f_i \int_{x=-\infty}^{\infty} F_i(x) F_j(x)\, dx = \int_{x=-\infty}^{\infty} g(x) F_j(x)\, dx$$

The equation for each j can be seen as a row in a matrix equation. On the right-hand side the function g is projected onto the basis functions F_j, while on the left-hand side we solve a matrix that compensates for nonunit and overlapping shape functions. This projection equation can be calculated as a series of integrations followed by a matrix-vector solve.

However, consider the case when the basis functions are othonormal—that is, when

$$\int_{x=-\infty}^{\infty} F_i(x)\ F_j(x)\ dx = \begin{cases} 1 & \text{if } i = j, \\ 0 & \text{else.} \end{cases}$$

In this case, the overlap compensation matrix (of $F_i F_j$ products) is the identity matrix; so no solve is actually needed. Luckily, many commonly used basis functions are orthonormal, including Fourier series, spherical harmonics, and inverse-area-scaled disjoint piecewise constant functions. Thus for these bases, Galerkin or least-squares function projection is simply a matrix-vector product, not a solve.

D.2 Cosine Lobe

We choose to define a modified cosine lobe function c centered on the unit vector $\hat{\mathbf{z}}$ for an input unit direction vector $\hat{\mathbf{d}}$ as

$$c_\alpha(\hat{\mathbf{d}}) = \max(0\,,\,(1-\alpha) + \alpha \hat{\mathbf{d}} \bullet \hat{\mathbf{z}})$$

We call this a cosine lobe function because $\hat{\mathbf{d}} \bullet \hat{\mathbf{z}} = \cos(\theta)$, where θ is the angle between $\hat{\mathbf{d}}$ and $\hat{\mathbf{z}}$.

This function is normalized such that $c_\alpha(\hat{\mathbf{z}}) = 1$. For $\alpha \leq 1$, this is the maximum value of c, so $c_\alpha(\hat{\mathbf{d}}) \in [0,1]$.

This function reduces to a number of commonly used functions in computer graphics.

- $c_{0.0}(\hat{\mathbf{d}}) = 1.0$, so with $\alpha = 0.0$, f represents uniform, unit illumination from all directions.
- $c_{1.0}(\hat{\mathbf{d}}) = \max(0\,,\,\hat{\mathbf{d}} \bullet \hat{\mathbf{z}})$, so with $\alpha = 0.0$, f represents the Lambertian illumination function if we take $\hat{\mathbf{z}}$ as the light source direction and $\hat{\mathbf{d}}$ as the surface normal.

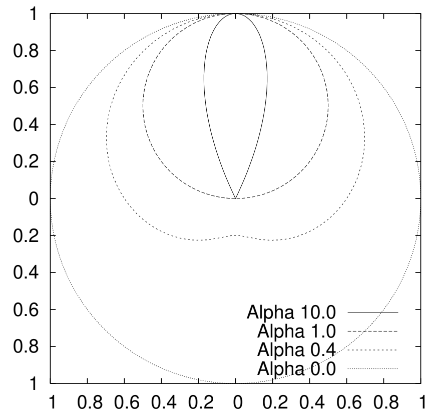


Figure D.1. The shifted cosine lobe function f.

- With $\alpha > 1.0$, c_α represents a small lobe centered around $\hat{\mathbf{z}}$, becoming increasingly narrow as α increases. Such a lobe would be useful to represent specular surfaces, by taking $\hat{\mathbf{z}}$ as the surface normal and $\hat{\mathbf{d}}$ as Blinn's halfway vector (average of view and light directions).
- c_∞ represents an infinitely narrow lobe, with width zero. This is not quite the Dirac delta function, because in the limit its integral is zero, not 1.

The function f is plotted in Figure D.1 for a series of α values.

D.2.1 Cosine Lobe Integrals

If the outgoing radiance leaving a small surface patch varies smoothly with direction, we can actually pre-calculate and project the outgoing radiance onto a small number of smoothly varying basis functions. To compute this outgoing radiance given the local incoming radiance, we must convolve the incoming light with the surface bidirectional reflectance distribution function (BRDF).

Hence if the incoming illumination and surface BRDF are such that we can approximate the outgoing radiance with a set of cosine lobes, projecting the outgoing radiance onto an orthonormal basis simply requires us to take the function product of a cosine lobe with the new basis.

We're interested in approximating the outgoing radiance in all directions $\hat{\mathbf{d}}$ over a sphere, so we need to calculate the definite integral

$$\int_{\bigcirc} cs = \int_{\theta=0}^{\pi} \int_{\phi=0}^{2\pi} c(\theta,\phi) s(\theta,\phi) \sin\theta \, d\phi \, d\theta$$

Here, θ is the angle down from the $\hat{\mathbf{z}}$ axis, and ϕ is the angle measured around the $\hat{\mathbf{z}}$ axis.

Because the spherical harmonic basis functions are orthonormal, we do not require a subsequent matrix solve step.

D.2.1.1 First Spherical Harmonic s_0

For the first spherical harmonic basis function, the constant function $s_0 = 1.0$,

$$\int_{\bigcirc} c_\alpha s_0 = \int_{\theta=0}^{\theta_0} \int_{\phi=0}^{2\pi} ((1-\alpha) + \alpha \cos\theta) \sin\theta d\phi d\theta$$

Here, if c_α reaches zero, θ_0 is the angle along which c_α first becomes zero. If c_α never becomes zero, which occurs for $\alpha < 0.5$, $\theta_0 = \pi$.

Taking the substitution $u = \cos\theta$, the integral above is equal to (interchanging the limits of integration to compensate for the negative in $du = -\sin\theta d\theta$)

$$2\pi \int_{u=u_0}^{u=1} (1-\alpha) + \alpha u du$$

$$2\pi \left[(1-\alpha)u + \alpha u^2/2\right]_{u=u_0}^{u=1}$$

If $\alpha < 0.5$, then $u_0 = -1$ and the integral is equal to $4\pi(1-\alpha)$ stradians. If $\alpha \geq 0.5$, $u_0 = \frac{\alpha-1}{\alpha}$ and the integral is equal to $\pi\frac{1}{\alpha}$ stradians.

D.2.1.2 Second Spherical Harmonic s_1

For the second spherical harmonic basis function, the linear function $s_1(\hat{\mathbf{d}}) = \hat{\mathbf{d}} \bullet \hat{\mathbf{v}}$,

$$\int_{\bigcirc} c_\alpha s_1 = \int_{\theta=0}^{\theta_0} \int_{\phi=0}^{2\pi} ((1-\alpha) + \alpha\cos\theta)\ (\cos\theta\ \hat{\mathbf{v}}.z + \sin\theta(\cos\phi\ \hat{\mathbf{v}}.x + \sin\phi\ \hat{\mathbf{v}}.y)) \sin\theta d\phi d\theta$$

θ_0 is defined as before; and we again take the substitution $u = \cos\theta$. The trigonometric ϕ terms integrate across 2π to 0; and we then have

$$2\pi\ \hat{\mathbf{v}}.z \int_{u=u_0}^{u=1} ((1-\alpha) + \alpha u) u du$$

$$2\pi\ \hat{\mathbf{v}}.z \left[(1-\alpha)u^2/2 + \alpha u^3/3\right]_{u=u_0}^{u=1}$$

If $\alpha < 0.5$, so $u_0 = -1$, the integral is equal to $\frac{4\pi}{3}\hat{\mathbf{v}}.z\alpha$ stradians. If $\alpha \geq 0.5$, the integral is equal to $\pi\hat{\mathbf{v}}.z\frac{\alpha-1/3}{\alpha^2}$ stradians.

Because our cosine lobe function is radially symmetric about the $\hat{\mathbf{z}}$ axis, only the component of $\hat{\mathbf{v}}$ along the z axis is relevant.

Appendix E

Graphics Operations

This section describes the basic graphics hardware features used in this work.

E.1 Hardware Operations

The parallel impostors technique requires only a few, widely available hardware operations.

One critical feature is the ability to upload textures onto the graphics card once, then repeatedly render using that texture. We currently use the OpenGL routine glBindTexture to create a texture, and glTexImage2D to copy in the image data. When replacing the impostor, we use glDeleteTextures to free the texture.

To draw an impostor, we currently use glBindTexture to activate the impostor's (previously uploaded) texture, then glBegin/glEnd to draw the impostor as a quadrilateral. We use four calls to glTexCoord2f/glVertex3f to describe the 2d texture coordinates and 3d vertex coordinates of the four corners of the impostor. The texture coordinates always cover the entire impostor texture.

E.2 Hardware Performance

We analyzed the triangle rate and fill rate for a variety of graphics hardware. In each of the following sections, we present the fill rate and triangle rate for drawing untextured single-color opaque right triangles with various short side lengths. The triangles are drawn using vertex arrays and glDrawElements, although numbers for glBegin/glVertex/glEnd are similar. Numbers are given for 66% vertex reuse (each triangle has only one new vertex), which should be typical of triangle strips; the performance appears similar for random-order triangles.

The detailed performance model is Equation 2.1, as described in Section 2.1. For triangles with sides less than about 10 pixels, all the graphics hardware we examined is triangle-rate limited; and the fill rate is quite poor. For triangles larger than about 100 pixels, all hardware is fill-rate limited. For triangles of intermediate size, triangle rate eventually trades off into fill rate.

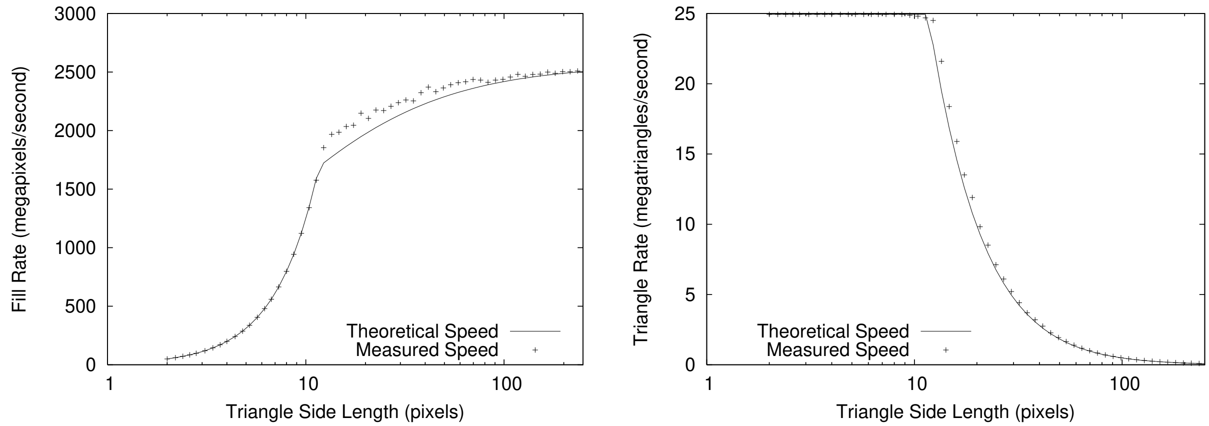


Figure E.1. Fill rate and triangle rate for nVidia GeForce6800/Athlon64.

E.2.1 nVidia GeForce6800/Athlon64

Figure E.1 shows the actual performance for a nVidia GeForce6800 with an AMD Athlon64 3200+ processor, running the 64-bit Linux 2.6.4 kernel with a 32-bit OpenGL program. We compare actual performance to the model $\alpha = 40.1$ ns, $\beta = 0.39$ ns/pixel.

Drawing each plain color pixel takes 0.396 ns/pixel. Adding texturing increases the cost to 0.814 ns/pixel. Alpha blending and texturing costs 0.838 ns/pixel. Depth and texturing costs 0.833 ns/pixel. Alpha, depth buffer, and texturing costs 0.870 ns/pixel. Mipmapped texture upload (with MIPMAP_SGIS) costs 1.253 us + 21.922 ns/pixel. Copying framebuffer pixels to a texture costs 0.285 ns/pixel. Copying framebuffer pixels to memory costs 5.759 ns/pixel.

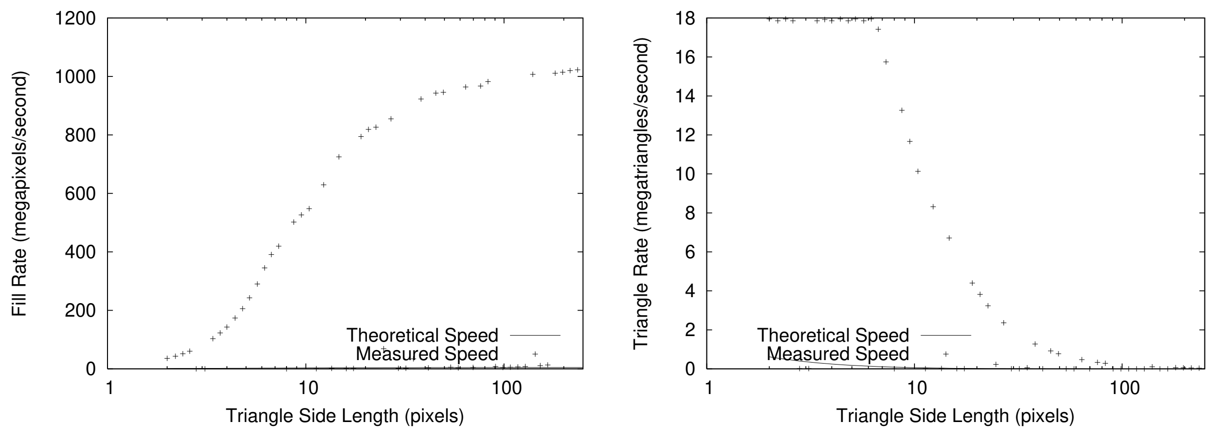


Figure E.2. Fill rate and triangle rate for nVidia QuadroFX 500.

E.2.2 nVidia QuadroFX 500

Figure E.2 shows the actual performance for a nVidia QuadroFX 500 (NV34GL rev a1) with a Pentium 4 3GHz under Linux 2.6.5, with nVidia driver version 61.11. We compare actual performance to the model $\alpha = 55.7$ ns, $\beta = 203.32$ ns/pixel.

Drawing each plain color pixel takes 0.970 ns/pixel. Adding texturing increases the cost to 1.518 ns/pixel. Alpha blending and texturing costs 2.159 ns/pixel. Depth and texturing costs 2.250 ns/pixel. Alpha, depth buffer, and texturing costs 2.902 ns/pixel. Mipmapped texture upload (with MIPMAP_SGIS) costs 1.281 us + 13.805 ns/pixel. Copying framebuffer pixels to a texture costs 1.737 ns/pixel. Copying framebuffer pixels to memory costs 9.660 ns/pixel.

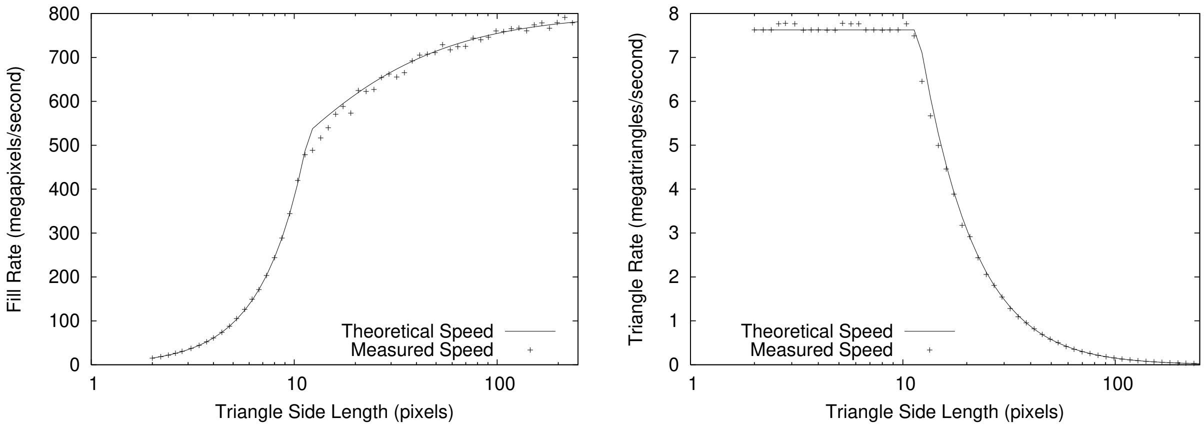


Figure E.3. Fill rate and triangle rate for nVidia GeForce3/Athlon.

E.2.3 nVidia GeForce3/Athlon

Figure E.3 shows the actual performance for a nVidia GeForce3 (ASUS 8200) with an AMD Athlon 1.3GHz processor, running Windows 2000 Professional. We compare actual performance to the model $\alpha = 131.1$ ns, $\beta = 1.25$ ns/pixel.

Drawing each plain color pixel takes 1.250 ns/pixel. Adding texturing increases the cost to 2.091 ns/pixel. Alpha blending and texturing costs 2.616 ns/pixel. Depth and texturing costs 2.497 ns/pixel. Alpha, depth buffer, and texturing costs 3.022 ns/pixel. Mipmapped texture upload (with MIPMAP_SGIS) costs 1.282 us + 43.749 ns/pixel. Copying framebuffer pixels to a texture costs 1.162 ns/pixel. Copying framebuffer pixels to memory costs 10.456 ns/pixel.

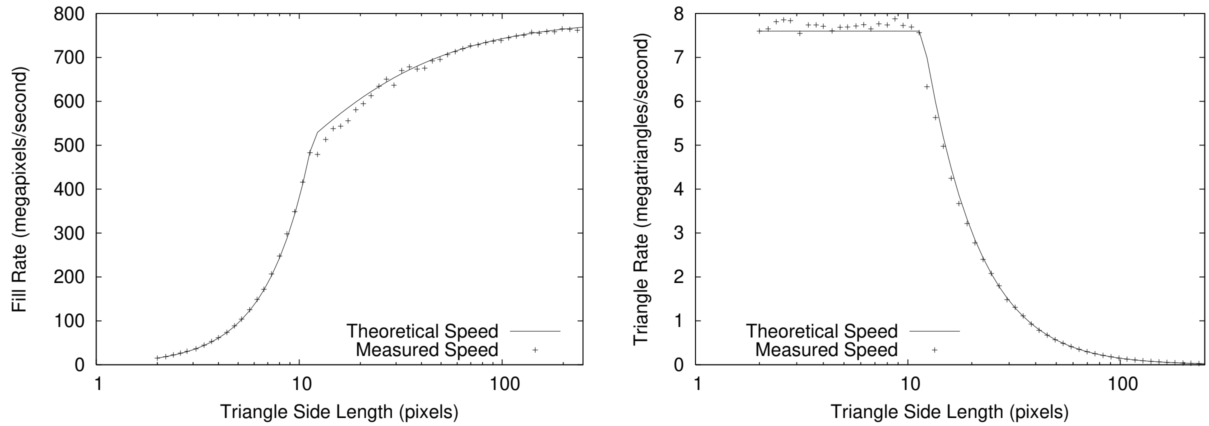


Figure E.4. Fill rate and triangle rate for nVidia GeForce3/Pentium 4.

E.2.4 nVidia GeForce3/Pentium 4

Figure E.4 shows the actual performance for a nVidia GeForce3 (ASUS 8200) with Intel Pentium 4 1.8Ghz/RDRAM under Linux 2.4.20, nVidia driver version 53.36. We compare actual performance to the model $\alpha = 131.6$ ns, $\beta = 1.27$ ns/pixel.

Drawing each plain color pixel takes 1.427 ns/pixel. Adding texturing increases the cost to 2.250 ns/pixel. Alpha blending and texturing costs 2.812 ns/pixel. Depth and texturing costs 2.636 ns/pixel. Alpha, depth buffer, and texturing costs 3.144 ns/pixel. Mipmapped texture upload (with MIPMAP_SGIS) costs 1.244 us + 15.588 ns/pixel. Copying framebuffer pixels to a texture costs 1.201 ns/pixel. Copying framebuffer pixels to memory costs 10.269 ns/pixel.

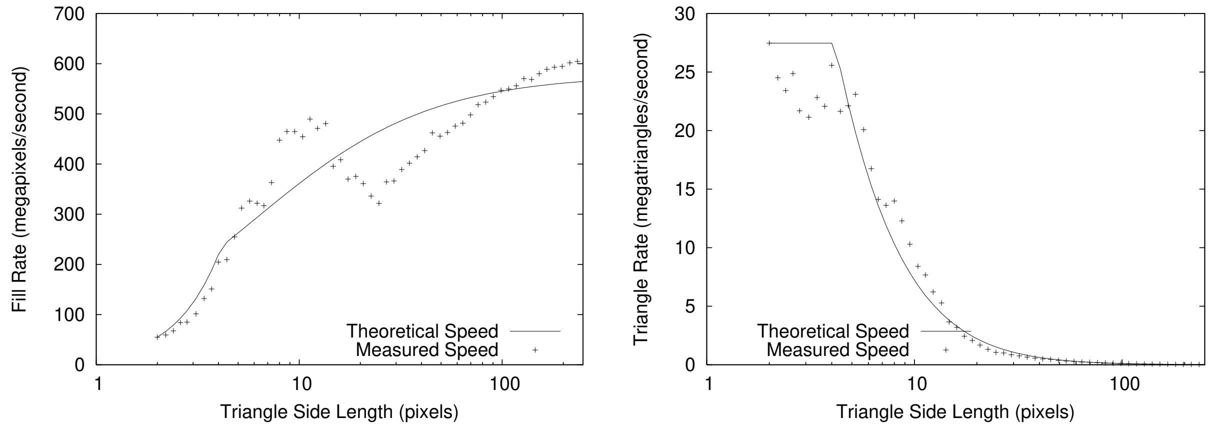


Figure E.5. Fill rate and triangle rate for ATI Mobility Radeon 9000.

E.2.5 ATI Mobility Radeon 9000

Figure E.5 shows the actual performance for an ATI Mobility Radeon 9000 (R250 Lf rev 1) with Intel Pentium M 1.6Ghz running Linux 2.4.25, ATI FireGL Linux kernel driver February 2004. We compare actual performance to the model $\alpha = 36.4$ ns, $\beta = 1.73$ ns/pixel.

Drawing each plain color pixel takes 1.586 ns/pixel. Adding texturing increases the cost to 3.863 ns/pixel. Alpha blending and texturing costs 6.108 ns/pixel. Depth and texturing costs 5.974 ns/pixel. Alpha, depth buffer, and texturing costs 8.205 ns/pixel. Mipmapped texture upload (with MIPMAP_SGIS) costs 0.923 us + 12.838 ns/pixel. Copying framebuffer pixels to a texture costs 6.486 ns/pixel. Copying framebuffer pixels to memory costs 27.652 ns/pixel.

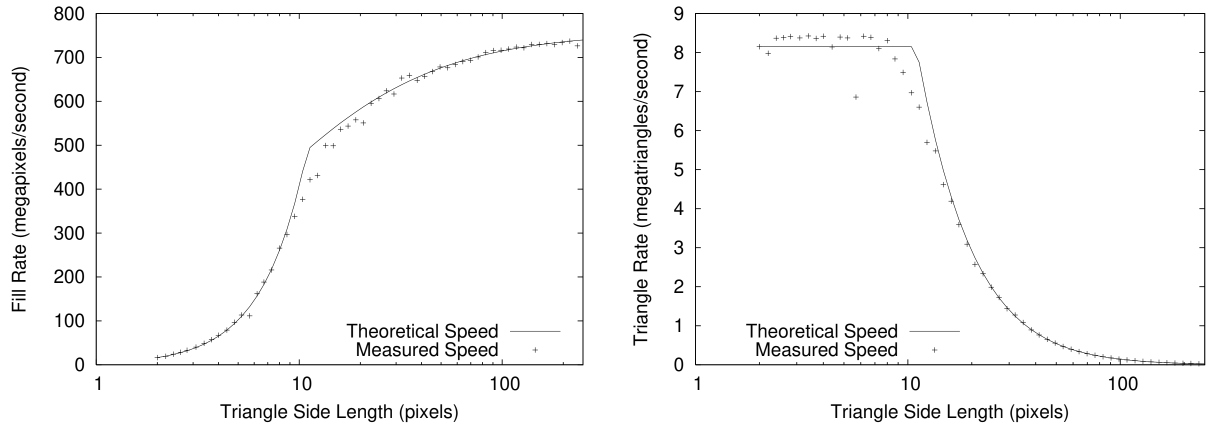


Figure E.6. Fill rate and triangle rate for nVidia GeForce2 Ti.

E.2.6 nVidia GeForce2 Ti

Figure E.6 shows the actual performance for a nVidia GeForce2 Ti (NV15 rev 164) with an AMD Athlon 1.25GHz running Linux 2.4.20, nVidia driver version 44.96. We compare actual performance to the model $\alpha = 122.7$ ns, $\beta = 1.32$ ns/pixel.

Drawing each plain color pixel takes 1.542 ns/pixel. Adding texturing increases the cost to 2.668 ns/pixel. Alpha blending and texturing costs 4.480 ns/pixel. Depth and texturing costs 3.708 ns/pixel. Alpha, depth buffer, and texturing costs 5.499 ns/pixel. Mipmapped texture upload (with MIPMAP_SGIS) costs 1.528 us + 33.313 ns/pixel. Copying framebuffer pixels to a texture costs 2.618 ns/pixel. Copying framebuffer pixels to memory costs 11.648 ns/pixel.

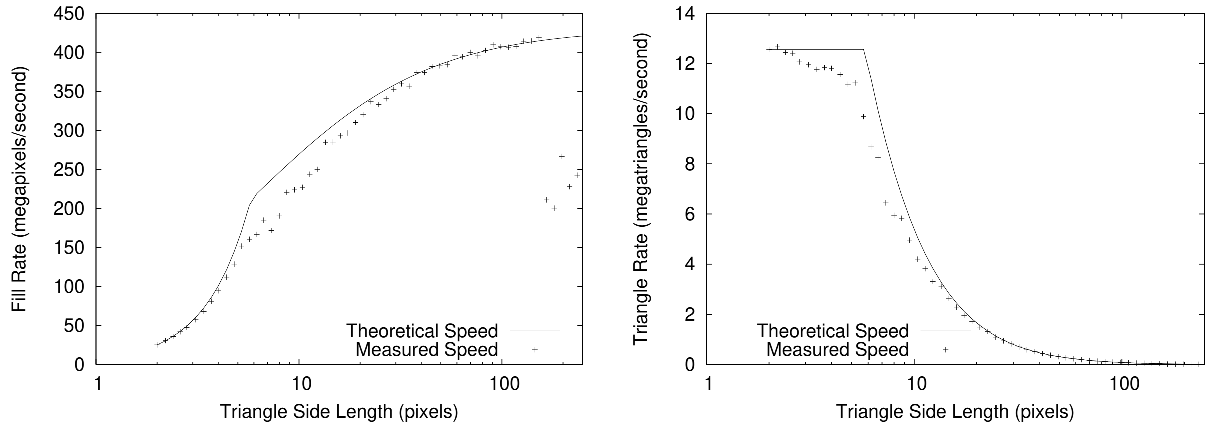


Figure E.7. Fill rate and triangle rate for nVidia GeForce4 MX 440.

E.2.7 nVidia GeForce4 MX 440

Figure E.7 shows the actual performance for a nVidia GeForce4 MX 440 (NV18 rev 162) with a Pentium 4 3GHz/Hyperthreading under Linux 2.4.22, with nVidia driver version 53.36. We compare actual performance to the model $\alpha = 79.6$ ns, $\beta = 2.32$ ns/pixel.

Drawing each plain color pixel takes 2.572 ns/pixel. Adding texturing increases the cost to 4.951 ns/pixel. Alpha blending and texturing costs 6.888 ns/pixel. Depth and texturing costs 7.143 ns/pixel. Alpha, depth buffer, and texturing costs 9.212 ns/pixel. Mipmapped texture upload (with MIPMAP_SGIS) costs 0.690 us + 10.165 ns/pixel. Copying framebuffer pixels to a texture costs 4.899 ns/pixel. Copying framebuffer pixels to memory costs 10.157 ns/pixel.

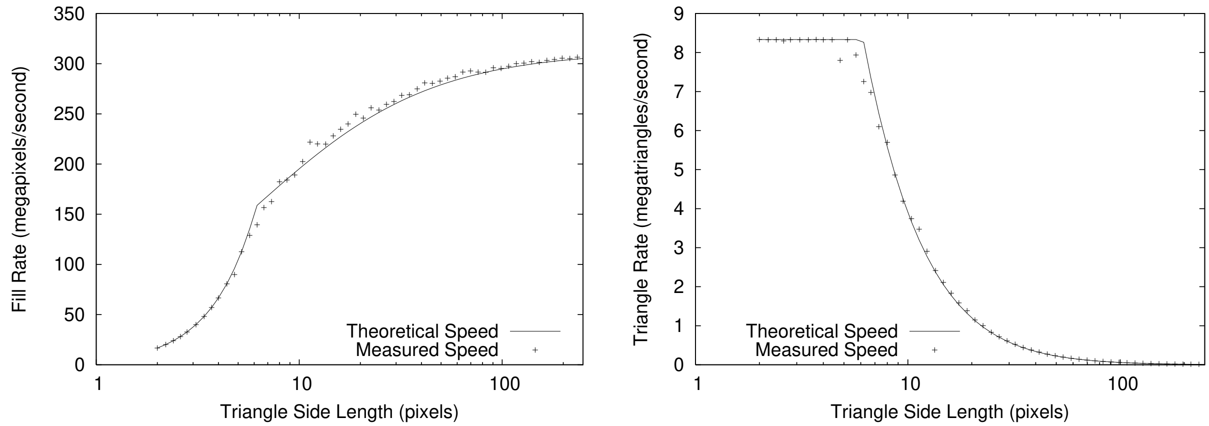


Figure E.8. Fill rate and triangle rate for nVidia Quadro2 MXR.

E.2.8 nVidia Quadro2 MXR

Figure E.8 shows the actual performance for a nVidia Quadro2 MXR (NV11 rev 178) with Intel Pentium 4 1.8Ghz/RDRAM under Linux 2.4.20, nVidia driver version 53.36. We compare actual performance to the model $\alpha = 120.0$ ns, $\beta = 3.20$ ns/pixel.

Drawing each plain color pixel takes 3.240 ns/pixel. Adding texturing increases the cost to 5.303 ns/pixel. Alpha blending and texturing costs 9.344 ns/pixel. Depth and texturing costs 7.644 ns/pixel. Alpha, depth buffer, and texturing costs 12.106 ns/pixel. Mipmapped texture upload (with MIPMAP_SGIS) costs 1.206 us + 14.884 ns/pixel. Copying framebuffer pixels to a texture costs 3.504 ns/pixel. Copying framebuffer pixels to memory costs 9.898 ns/pixel.

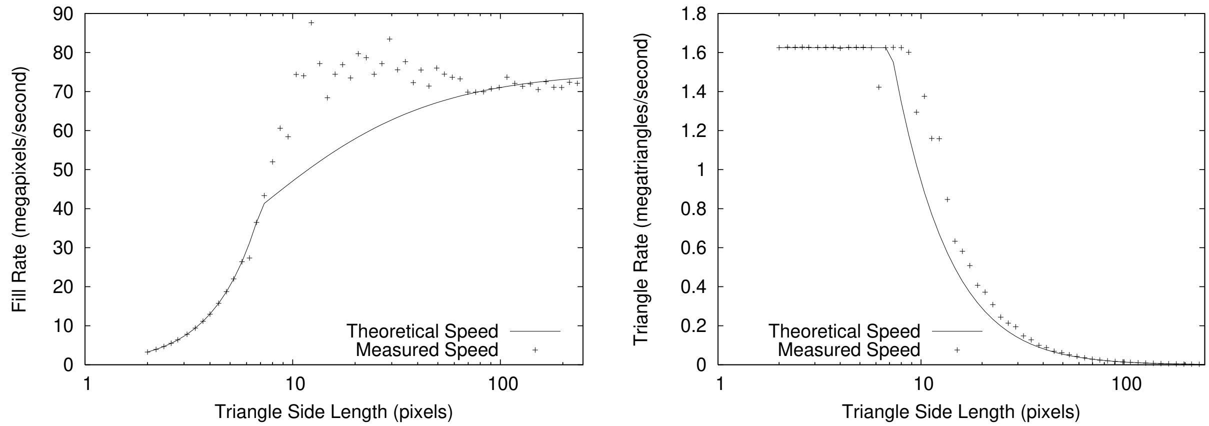


Figure E.9. Fill rate and triangle rate for Sun Creator-3D.

E.2.9 Sun Creator-3D

Figure E.9 shows the actual performance for a Sun Creator-3D graphics card in a SunBlade 1000 running Solaris 8, Sun OpenGL 1.2.2 for Solaris. We compare actual performance to the model $\alpha = 615.4$ ns, $\beta = 13.28$ ns/pixel.

Drawing each plain color pixel takes 14.142 ns/pixel. Adding texturing increases the cost to 151.531 ns/pixel. Alpha blending and texturing costs 151.500 ns/pixel. Depth and texturing costs 151.536 ns/pixel. Alpha, depth buffer, and texturing costs 151.592 ns/pixel. Mipmapped texture upload (with MIPMAP_SGIS) costs not available. Copying framebuffer pixels to a texture costs ns/pixel. Copying framebuffer pixels to memory costs ns/pixel.

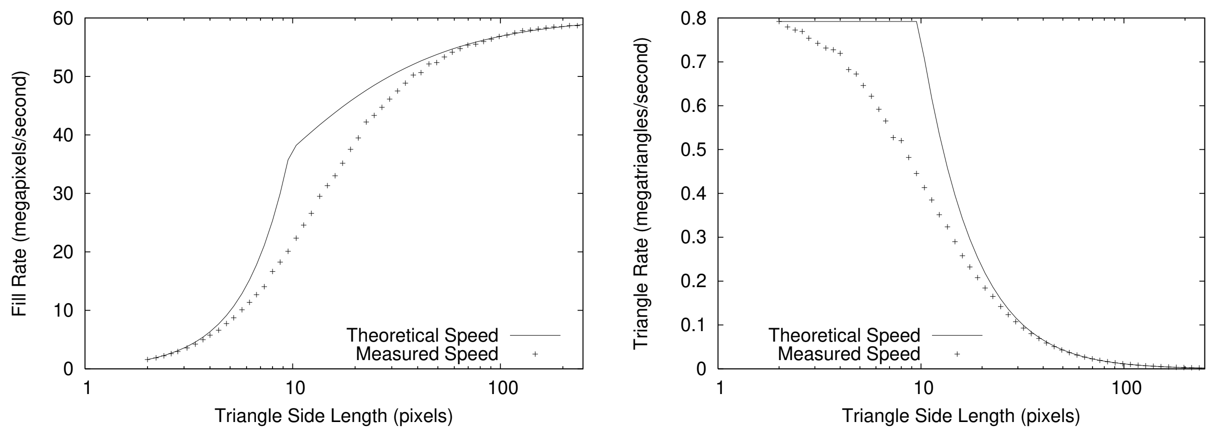


Figure E.10. Fill rate and triangle rate for Mesa Software Rendering.

E.2.10 Mesa Software Rendering

Figure E.10 shows the actual performance for software rendering using Mesa 4.0.4 and XFree86 4.3.0 on a Pentium M 1.6GHz running Linux. We compare actual performance to the model $\alpha = 1262.5$ ns, $\beta = 16.59$ ns/pixel.

Drawing each plain color pixel takes 16.745 ns/pixel. Adding texturing increases the cost to 180.310 ns/pixel. Alpha blending and texturing costs 222.258 ns/pixel. Depth and texturing costs 188.139 ns/pixel. Alpha, depth buffer, and texturing costs 230.198 ns/pixel. Mipmapped texture upload (with MIPMAP_SGIS) costs 1.210 us + 40.008 ns/pixel. Copying framebuffer pixels to a texture costs 23.867 ns/pixel. Copying framebuffer pixels to memory costs 124.728 ns/pixel.

Author's Biography

Orion Sky Lawlor was born on February 2, 1977 in California, but grew up in the tiny town of Glennallen, Alaska. He received two B.S. degrees, one in Computer Science and another in Mathematics, from the University of Alaska at Fairbanks in 1999. While at Fairbanks, he was affiliated with the Alaska Synthetic Aperture Radar Facility, where he developed algorithms for satellite radar processing and radar interferometry. For graduate work he transferred to the University of Illinois at Urbana-Champaign, where he worked with Dr. L. V. Kalé and the Parallel Programming Lab. After receiving his M.S. in Computer Science in 2001, while working on his PhD he also worked as staff with the Center for Simulation of Advanced Rockets, headed by Dr. Michael T. Heath, where his focus was on 3D mesh preparation and remeshing tools. After completing his PhD at Illinois in late 2004, Orion will return to Fairbanks, Alaska, where he will continue to pursue his research interests in parallel algorithms, interactive computer graphics, optical and radar remote sensing, and large models and geographic information systems.