Impostors for Parallel Interactive Computer Graphics

Orion Lawlor
Department of Computer Science
University of Illinois at Urbana-Champaign
olawlor@acm.org

April 1, 2004

Abstract

We introduce 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 the parallel machine even at high resolutions and framerates. Impostors also decrease the required rendering bandwidth, which makes possible the interactive use of advanced rendering techniques like antialiased raytracing. These techniques are demonstrated by the interactive high-quality rendering of very large higly detailed models.

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. Modern graphics hardware rendering can render large models quickly, but sacrifices accurate light transport. 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 the object once, then caches and reuses the rendering over several frames. The rendering is stored as an alpha-blended texture, and can be rendered as a texture-mapped polygon—an impostor. Because rendering an impostor is faster than rendering the object, the impostor approach amortizes the rendering cost of complicated objects over several frames.

The enabling technology we present is to **quickly** 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. This means that for good parallel load balance, the parallel backend must shift responsibility for rendering different parts of the scene between processors. To perform this load balancing, we use our parallel work migration and load balancing system [LK03], as described in Section 4.

Using a parallel machine provides computational power that can be used to improve the rendering **accuracy** and **detail**. We describe techniques to antialias the impostors, compute both direct as well as partial indirect lighting, and procedurally generate needed detail, as described in Section 6.

Using impostors and parallel machines allows us to render very **large** geometries, such as large virtual environments, at real-time frame-rates. We will demonstrate this by exploring an extremely large, detailed model of the University of Illinois at Urbana-Champaign campus, as described in detail in Section 5.

The architecture we propose renders impostors on a parallel machine potentially miles away, then transmits them to the serial client machine sitting on the user's desk. The client machine then assembles the 3D scene by drawing individual 2D impostor images as textured polygons using conventional graphics hardware. The overall data flow is as shown in Figure 1, the underlying equations governing this method are derived and analyzed in Section 2, and the features and advantages of the impostors approach to image assembly are described in Section 3.

2 Fundamentals

This section analyzes the fundamental relationships that motivate and limit the usability of impostors. The overall conclusion of this section is that impostors provide a degree of flexibility that an intelligent rendering system can exploit to solve existing problems with rendering efficiency.

2.1 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 A.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 to draw a triangle on modern graphics hardware is well modeled by

$$t = \max(\alpha, \beta(s + \gamma r)) \tag{1}$$

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

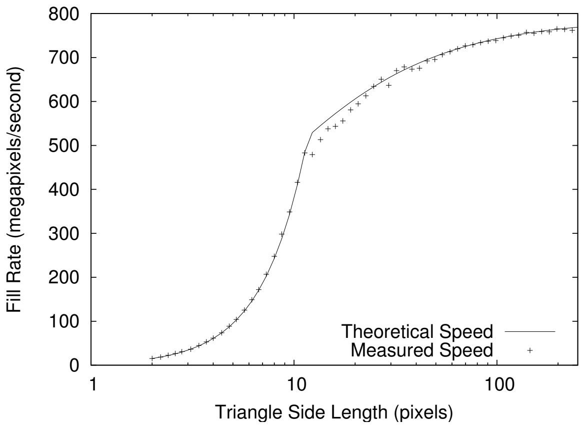

Figure 2. Achieved fill rate for nVidia GeForce3/Pentium 4.

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 A.2 gives measured α and β for a variety of graphics hardware.

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):

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

$$B_C = \min(\frac{s}{\alpha}, \frac{1}{\beta(1 + \gamma r/s)}) \tag{2}$$

For small triangles, the triangle setup α term limits the overall performance. Thus to achieve

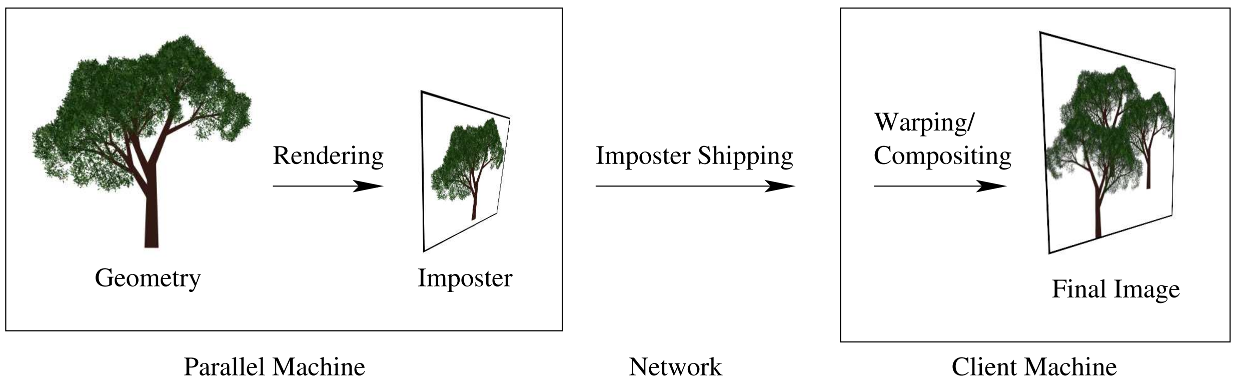

Figure 1. Our client/server graphics architecture.

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 shows a typical plot of this; Appendix A.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.

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 Update Rate

As the camera moves, our 2D impostor images must be updated to follow the true 3D geometry of the object they represent. This section analyzes how often the impostors must be updated, as this geometric update rate places fundamental limitations on the benefit provided by impostors. The simplest effect to analyze, and the worst case of camera motion, is parallax. We analyze this simple case in detail to bound the geometric reprojection limits for our method. Note that 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 is something all objects will have to handle. Also, we analyze (and use) simple planar impostors; a similar analysis holds for non-planar "meshed impostors".

Consider a thin bar aligned with the z axis and centered at world coordinates $(0, z)$, as shown in Figure 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 pixel-for-pixel identical. We la-

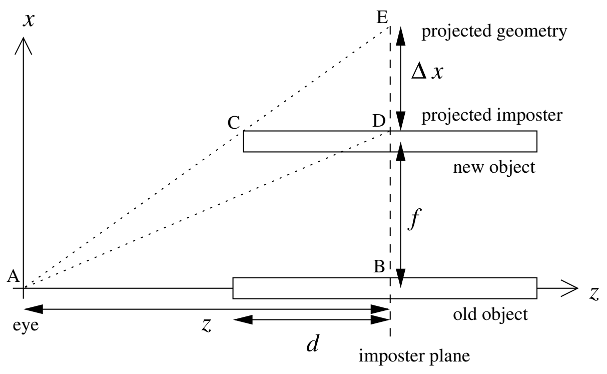

Figure 3. Parallax analysis for impostors.

bel the distance from the impostor plane to the most distant object point d, the 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:

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

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

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

$$\Delta x = \frac{fd}{z-d}$$

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 Δx, we can solve for the maximum allowable camera motion f:

$$f = \frac{z(z-d)\Delta s}{kd} \qquad (3)$$

	$d = 0.05$	$d = 0.25$	$d = 1$	$d = 5$
$z = 1$	0.04	0.01	-	-
$z = 5$	0.97	0.19	0.04	-
$z = 25$	24.37	4.83	1.17	0.2
$z = 100$	390.4	77.93	19.34	3.71

Table 1. Bound, in meters, on the distance the camera can safely move for an impostor to stay below one pixel of parallax error. Resolution is 1024x768 with a 90 degree horizontal field of view. Impostor distance z and depth d are in meters.

	$d = 0.05$	$d = 0.25$	$d = 1$	$d = 5$
$z = 1$	1	1	1	1
$z = 5$	10	2	1	1
$z = 25$	263	52	12	2
$z = 100$	4216	841	208	40

Table 2. Number of frames each impostor can be re-used, if the camera moves perpendicularly at V =20kmph and the framerate is H =60hz.

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:

$$R = \frac{fH}{V} = \frac{z(z-d)\Delta sH}{kdV} \qquad (4)$$

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.

Table 1 gives examples of the allowed camera motion f, and Table 2 gives the numbers of frames R the impostor can be reused before the screen error exceeds one pixel. Distant or very flat impostors can tolerate enormous amounts of camera motion, and hence can be reused for a large number of frames. Very deep or very close impostors can almost never be reused, as even a small camera motion causes a visible amount of parallax change.

2.3 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 rendering 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 100,000 to one million finished pixels per second, which is 400KB/s-4MB/s per processor. 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 4.3.2. 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 for tree-like impostors; while a lossy compression such as S3TC could increase it fourfold, with increased CPU cost and 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 A.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 80-350Mpix/s (250MB/s-1.5GB/s pixel bandwidth). The actual achieved fill rate depends on the triangle area, as shown in Equation 1.

The impostors method allows rendered, shipped, and uploaded frames to be reused on the graphics card several times. We take the area-averaged impostor reuse rate, bounded by Equation 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 take 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:

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

The lowest numerical bandwidth values are for rendering bandwidth B_R; but rendering bandwidth can theoretically 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 $R = 25$-150 times, which according to Table 2 is only acceptable for very flat or very distant impostors. Clearly, texture compression (high C_N) or gigabit ethernet (high B_N) will be necessary for most scenes.

A balanced high-performance system might render with a bandwidth of $B_R = 1$ Mpix/s/cpu with a parallel speedup $P = 32$, for a rendering bandwidth of 32 Mpix/s. The impostors would be shipped over the network using run-length encoding with $C_N = 0.5$ pixels/byte and gigabit 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 the fill rate bandwidth of $B_C = 300$ Mpix/s. Assuming a depth complexity $D = 2$, 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 Extrema

It is useful to examine the limiting cases of this parallel rendering system, as summarized in Table 3.

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 parallel raytracers [Sto98] and other systems such as Chromium [HHN$^+$02]. However, because the depth range for the scene is huge, Equation 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 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 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 decompositions 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.

3 Impostors

An image *impostor* is a 2D standin 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 4. 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.

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 3. Various extreme cases for the impostors method, for various impostor depths d and areas a, and their limitations.

Figure 4. A painting in the *trompe l'oeil* style by William Michael Harnett, 1848-1892.

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.

3.1 Prior Work: Impostors

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 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 method for rendering trees was 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 Doom were 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 precomputed texture-mapped polygon impostors for fixed pieces of geometry from fixed viewpoints.

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 also did not consider parallel or antialiased rendering.

Aliaga [Ali98] describes a fully automated portal-based rendering system based on impostors, and presents a technique for warping the surrounding geometry to match an impostor texture.

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 being used with 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] described 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 overdraw caused by the many texture passes each layer.

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.

3.2 Objections

Impostors, like any technique, has its drawbacks.

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 can 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.

3.2.1 Missing Z

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 cards 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.

There are also several inherent limitations to the usual hardware Z-buffer depth algorithm. The Z-buffer consumes memory bandwidth during rasterization approximately equal to that consumed by framebuffer writes. The Z-buffer is not useful when combining partially transparent objects, which must instead be drawn in strict depth order. The pass-fail nature of the Z-buffer depth test prevents us from antialiasing the edges of polygons. Finally, the finite precision of the depth buffer leads to quantization errors (z-buffer fighting) for scenes with a large range of depths, such as scenes consisting of both nearby foliage and far away mountains.

All these disadvantages of the Z buffer lead us to instead use the well known per-object painter's algorithm. We traverse the scene's impostors in back-to-front order, alpha-compositing impostors as we go. This allows us to avoid the expense of computing, sending, and compositing per-pixel depths; allows us to use transparency and antialiasing to improve the appearance of our impostors; and avoids flashing Z-buffer roundoff errors. The method we use to traverse our scene database in Z order is described in Section 5.2.

3.3 Planned Work

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.

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 will determine if an update is required by examining a set of "key points", for which we will explicitly compute and compare the screen projection of the actual 3D location with the screen projection of the 2D impostor data. Initially, we will use the eight corners of the bounding box as key points. Impostors may also need to be updated because of intrinsic change, such as object deformation.

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 subpixel 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, judiciously chosen image planes may be able to conceal a great deal of error.

Temporal discontinuity hiding. When we get a new impostor image, 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. Classic solutions to this problem soften the transition using time or position dependent crossfades. An 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.

4 Parallel Rendering

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

4.1 Prior Work: Parallel

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; 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, which would be difficult and unlikely to produce a good quality partitioning. In addition, the application would have to describe all the geometry of a scene, including that 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.

4.2 Proposed Architecture

The client, a regular OpenGL program, will connect to the parallel server using our TCP/IP-based protocol CCS, as described in Section 4.3.2. 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 will receive viewpoint updates, render new impostors if necessary, and send impostor images back to the client. On receiving a new viewpoint, the server will broadcast the viewpoint to the parallel objects that represent each piece of geometry. If the previously rendered impostor for that geometry is no longer adequate, the geometry object enqueues itself to be redrawn. Because rendering is separate from the redraw 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 database, 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 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 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 5. This loosely synchronized architecture should be able to perform well even in the presence of significant network performance variation.

4.3 Parallel Infrastructure

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

4.3.1 Charm++ Array Manager

The Charm++ Array Manager[LK03] allows parallel objects called *array elements* to be cre-

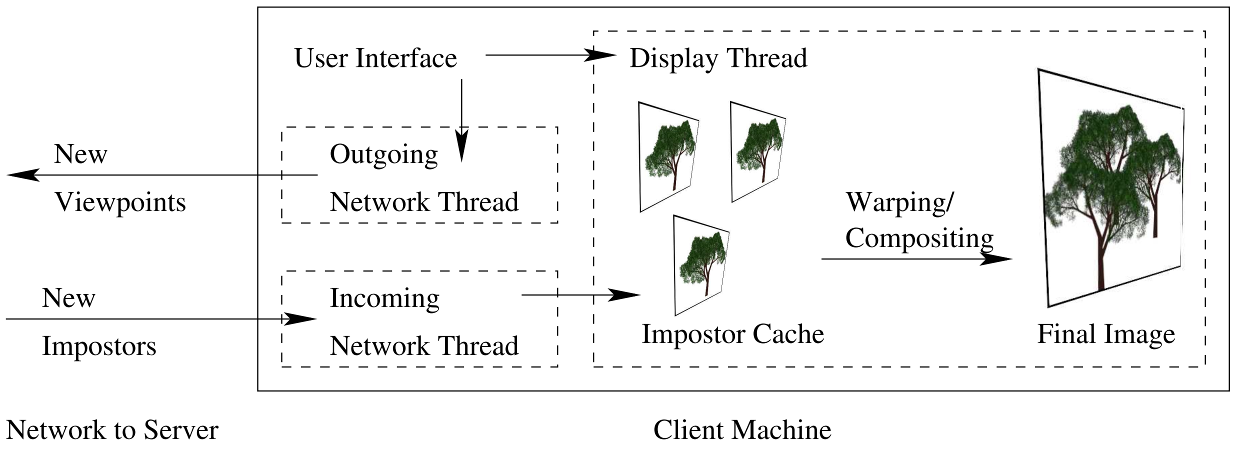

Figure 5. Architecture of client, showing threads that handle various tasks.

ated, destroyed, moved between processors of the parallel machine, send and receive messages, and participate in broadcasts and reductions. Each piece of geometry to be rendered as an impostor will be represented as an array element parallel object.

Array element broadcasts will be used to distribute the latest user viewpoint. Array elements will send each other messages to coordinate lighting, rendering, impostor merging, and splitting tasks.

4.3.2 Converse Client/Server

The *Converse Client/Server* (CCS) protocol [JLK04] is a simple protocol based on TCP/IP. CCS runs portably on all UNIX machines with sockets as well as Windows machines, and includes authentication. CCS allows 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 visualizer 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. After the server has processed the request, it responds with a block of binary response data. This simple request/response protocol allows information to be injected into and extracted from a running parallel program.

Because the client opens the TCP connection for a CCS request, CCS can be used by clients behind firewalls or NAT routers. When CCS is running over the unsecured internet, it can be run in a secure authentication mode[Par04], which uses a SHA-1 hash of the request, a nonce, and a shared secret key for authentication. Authentication prevents arbitrary users from injecting messages, but because of export regulations we do not provide network encryption.

4.3.3 PUP Framework

CCS provides a byte stream between client and server; but for object-level communication we use the existing *PUP framework* [JLK04], a general-purpose method for manipulating the contents of C++ objects. The PUP framework is also used to migrate objects between processors on the parallel machine, and perform I/O.

CHARM++ originally required users to write explicit data copying pack and unpack routines for each object, as well as a size routine to determine the outgoing message size before packing. The motivation for PUP is that the routines used to compute the object size, pack an object into a message, and unpack an object from a message all must match up exactly. Everything that 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 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. The PUP framework uses C++ operator overloading to elegantly express this object-field recursion.

The PUP interface used by CCS translates messages to network byte order during the packing process, and translates back to native byte order during unpacking. This allows clients and servers with different machine architectures to interoperate, without any additional effort. A different implementation of the same interface reads and writes objects to binary or ASCII disk files, which is used for object I/O.

Finally, the PUP framework provides a class registration mechanism to remotely instantiate subclasses when the caller only knows about the superclass. This is used to instantiate the display objects that hold impostors generated by remote parallel objects.

4.4 Planned Work

We have already created most of the parallel infrastructure that will be needed, but achieving good parallel utilization and load balance will require additional research.

4.4.1 Geometry Decomposition

The hardware performance model Equation 1 shows that the graphics card's delivered fill rate drops substantially for small triangles. But the impostor parallax update rate Equation 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 will be represented by exactly one parallel array element. Thus as the camera moves, as 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 5.4.

4.4.2 Network Compression

As the primary bottleneck for performance may be the link between client and server, some amount of data compression may be beneficial. We will analyze the bottlenecks and compare the performance of uncompressed, run-length encoded, entropy encoded (gzip), and lossy frequency-domain encoded (JPEG) images.

4.4.3 Lighting

Rendering impostors is completely parallel, but certain operations such as lighting, described in Section 6.4, will require different pieces of geometry to communicate. For example, all the geometry should share a single lighting map. To implement this in parallel, we will distribute the lightmap into tiles stored by a dedicated set of array elements. During lighting, we will merge the lightmap contributions from different pieces of geometry. During rendering, each piece of geometry will access its own portion of the lightmap. Because lightmap access has excellent spatial and temporal locality, it should be possible to implement this scheme efficiently.

4.4.4 Parallel Prioritization

A critical task for the parallel renderer will be allocating rendering cycles—deciding which pieces of geometry to draw, from which viewpoints, and at what resolution. Consider that most impostors in the world are either behind the viewer, obscured by other geometry, or too far away to be seen; so with an imperfect prioritization scheme we could waste all our rendering cycles drawing irrelevant geometry.

We will design and quantitatively evaluate the performance of quality rendering prioritization systems. The evaluation will be performed by sending the parallel machine each camera position of a stored camera path, and computing the frame-by-frame RMS error of the resulting renderings, compared with a perfect set of renderings computed offline.

We will examine prioritization systems involving combinations of these factors:

- Geometry's screen area in pixels, not counting clipped or occluded areas.
- Current impostor's projection error, in screen pixels.
- Perceptual importance, including contrast and visible detail.
- Estimated rendering time required, to encourage short jobs to go first.
- Request age in frames, to prevent starvation.

For example, one promising approach is to set the priority as the product of the screen area, error, and importance divided by the estimated rendering time. This attempts to maximize the per-pixel perceived image quality per second.

We will also investigate the use of estimated future values for these prioritization factors. This could allow speculative "prefetching" work, at least when the camera path is reasonably predictable. In particular, we will evaluate the use of impostors projected, not from the current camera position, but from the estimated camera position halfway though the imposter's lifetime. With perfect camera prediction, this pre-warping technique could double the impostor reuse rate.

One complicating factor in prioritization is the fact that most of the world's geometry is offscreen or occluded—for example, all the geometry behind the viewer or hidden by nearby buildings. This means we must be prepared to quickly stop rendering geometry that becomes hidden. More difficult is the fact that in an interactive setting, geometry can become visible quite quickly, for example when panning the camera or passing beyond the corner of a building. For these situations, predicting and rendering currently invisible geometry may be very important.

We anticipate accurate prioritization will be difficult to obtain, but very important for overall system efficiency.

4.4.5 Parallel Load Balancing

The existing Charm++ automatic load balancing system [Bru00] will monitor the computational load on each processor and the load generated by each array element. If significant load imbalance is detected, the system will migrate array elements between processors to improve the load balance.

We expect this history-based load balancing method will appear to provide good load balance, in the sense that all processors will remain busy most of the time. However, for good delivered performance, all processors must not only be busy, but must be busy doing important, high-priority work. We will use our extensive infrastructure for parallel prioritization to perform this prioritized load balancing [KRSS93].

In particular, we will use the impostor update rate Equation 4 to estimate the time until the next required update based on the current amount of error, and the priorities described above to evaluate the priority of that update. If the expected future prioritized load distribution is uneven, we will migrate array elements to resolve the imbalance.

Note that because much of our geometry is procedurally generated, the individual array elements representing each piece of geometry should require only a small amount of memory. Thus migrating these array elements should be very efficient.

5 Large Model

The motivating example and performance benchmark for our method 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. Ground truth data is readily available. Finally, a large virtual environment which includes substantial quantities of distant geometry fits well with the parallel impostors technique, due to the high reuse of distant impostors shown in Table 2.

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.

5.1 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 6. 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.

In this sense, the mesh representation is a perfectly ordinary expand-on-demand quadtree. The well-known problem with quadtrees, however, is that when a quad is refined but the adjacent quad is not refined, a crack appears between the two quads. Lindstrom and Pascucci's contribution is a preprocessing bottom-up error propagation step that allows the runtime top-down traversal to respect refinement boundaries, so cracks never appear.

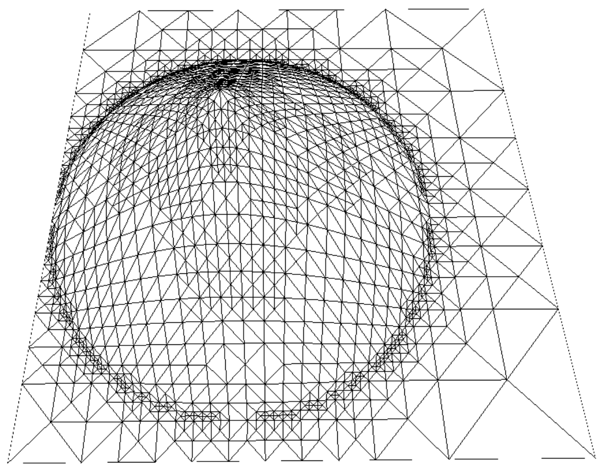
Figure 7. Mesh for terrain rendering of a cone sticking up out of a plane. Note how the quads are split around the axis of the cone, conforming to the surface curvature.

The method we use differs slightly from Lindstrom and Pascucci's in that we explicitly treat the mesh as a set of quadrilaterals (quads), which allows us to 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, as shown in Figure 7.

Rather than representing the entire mesh as a strict hierarchy, we divide the preprocessed mesh up into independent rectangular tiles. This eliminates the top levels of the hierarchy, and allows the tiles to be paged in across the network.

5.2 Depth Traversal

As described in Section 3.2.1, we will not 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.

As we recursively traverse the terrain quadtree, we can choose 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 the geometry in back-to-front order. Where this becomes difficult is when the camera is within the scene,

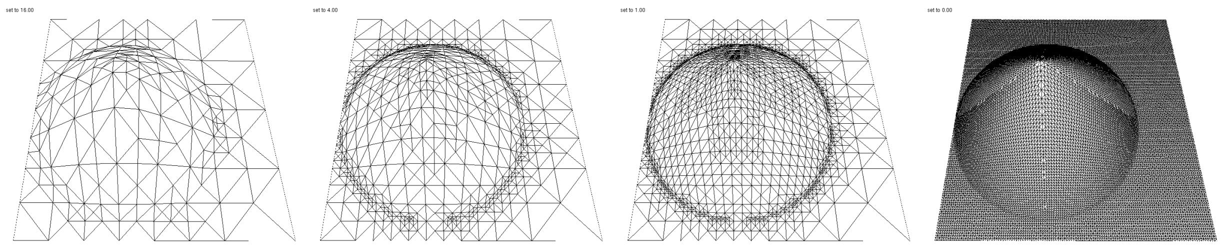

Figure 6. Mesh for terrain renderings with different screen-space error criteria: 16 pixels, 4 pixels, 1 pixel, 0 pixels.

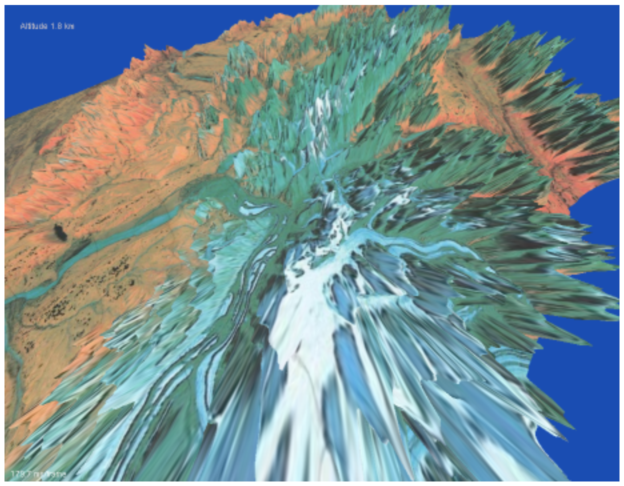

Figure 8. Extreme wideangle view looking down from 1.8km above Denali National Park in Alaska. This image is correctly rendered in back-to-front (outside to inside) order, which demonstrates the correctness of our painter's algorithm depth sort.

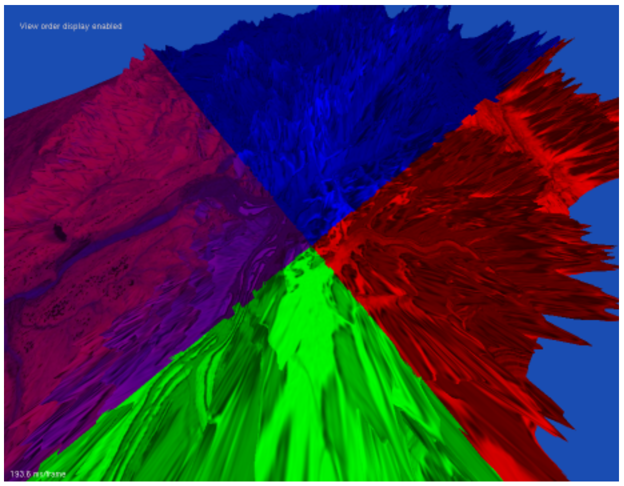

Figure 9. The same view as Figure 8, showing the order in which the terrain is traversed. The order starts from the left in the purple region, from the top in the blue region, from the right in the red region, and from below in the green region.

as in Figure 8. In this case, the child processing order must be determined separately at each node. Figure 9 is colored by the number of the first processed child; 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.

5.3 Data Sources

An immense amount of data is available for the campus.

AutoCAD Maps. Like many campus operations and maintenance departments, UIUC maintains a number of AutoCAD files describing 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.

We use the OpenDWG [All04] library to read the AutoCAD, and custom tools to bring the drawings into the UTM map projection from the Illinois State Coordinate System (ILCS). An example of a section of the utilities map is shown in Figure 10.

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

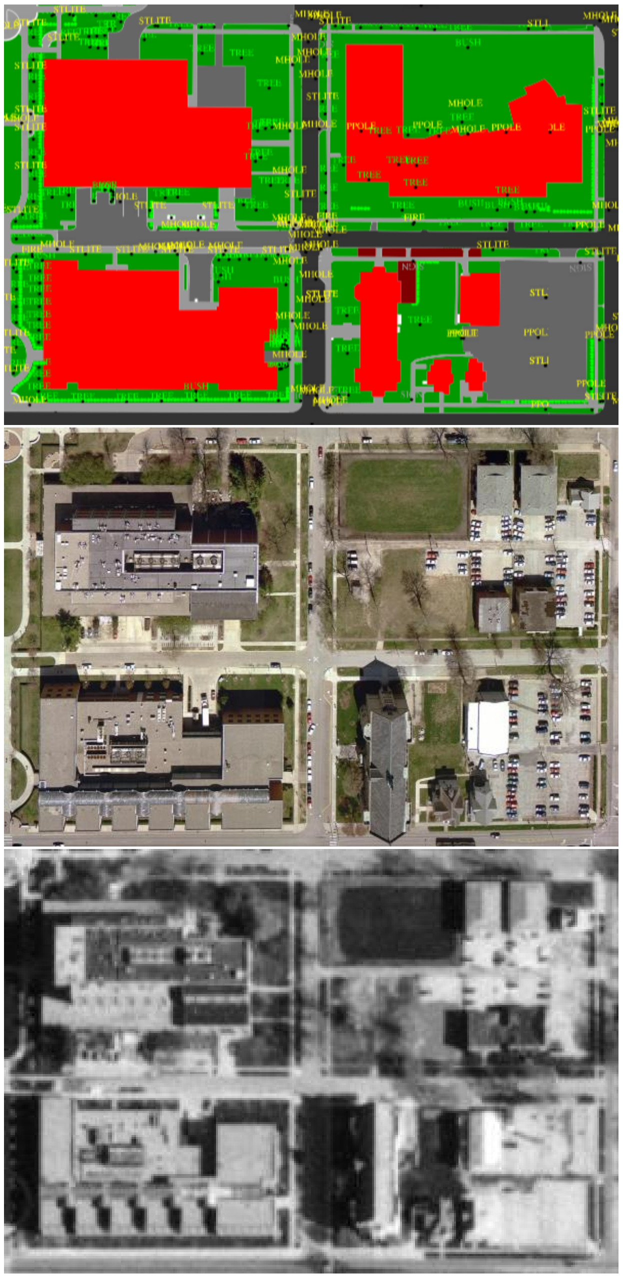

Figure 10. Four blocks from the 2004 campus utilities map (top) and 1999 aerial photo (middle), and 1998 USGS orthophoto (bottom) near Mattews and Springfield in Urbana, Illinois. Note how the Siebel Center in the top right, constructed in 2003, does not appear in the lower two photos.

pixel and a 250 scale with a ground resolution of approximately 12cm per pixel, as shown in Figure 10.

USGS Maps. The United States Geological Survey publishes a variety of data for the entire United States. We will use the USGS 1:24K scale digital elevation model (DEM) data as our source for overall terrain features. Figure 11 shows the elevation map.

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 12.

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

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 we have developed a geometric and radiometric calibration profile. 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.

5.4 Planned Work

A substantial amount of work remains on the campus model.

Integration. The data sources in the previous section must be registered into a single coordinate system. We we will use the UTM map projection, because the USGS data is already in this coordinate system. UTM provides a regular 2D coordinate plane measured in meters.

Culling. Our height map traversal algorithm does not perform occlusion culling, which could avoid drawing scenery that is obscured by buildings. We will extend our height map to perform occlusion culling in substantially the same manner as view culling: during the preprocess we will compute and propagate 2D angular view infor-

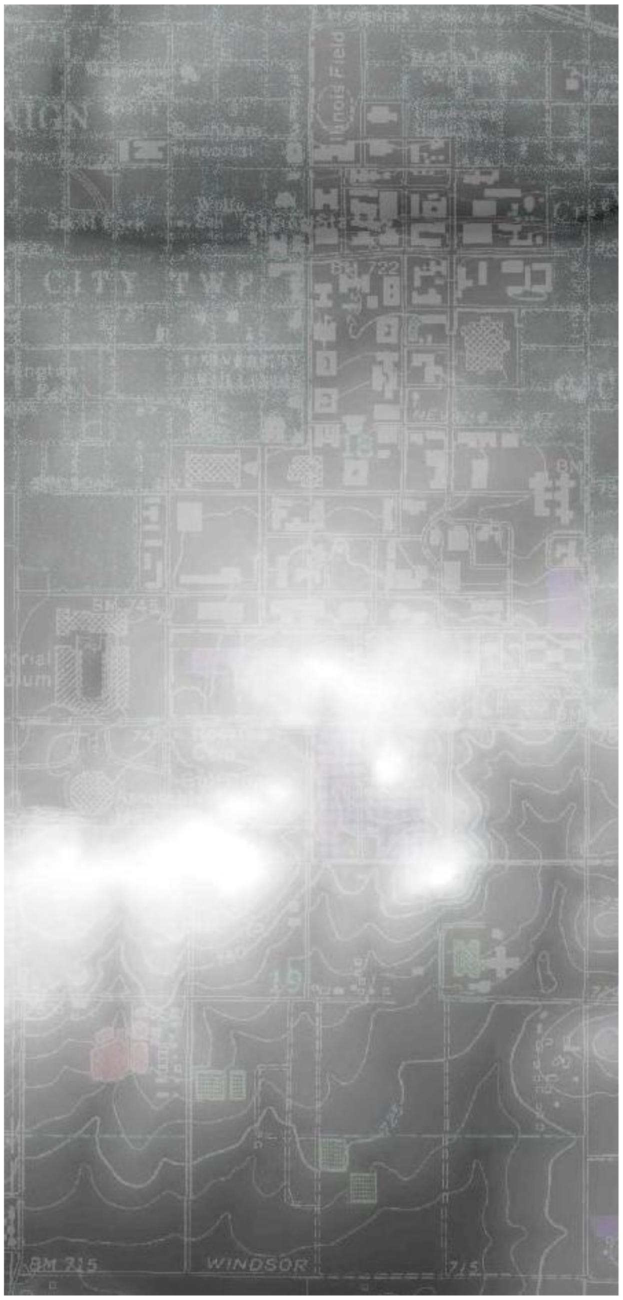

Figure 11. 4.7 square kilometers from the USGS 10m elevation map, with the USGS topo map overlaid. The region displayed stretches north-south from University to Windsor; and from 1st Street in Champaign to Lincoln Avenue in Urbana.

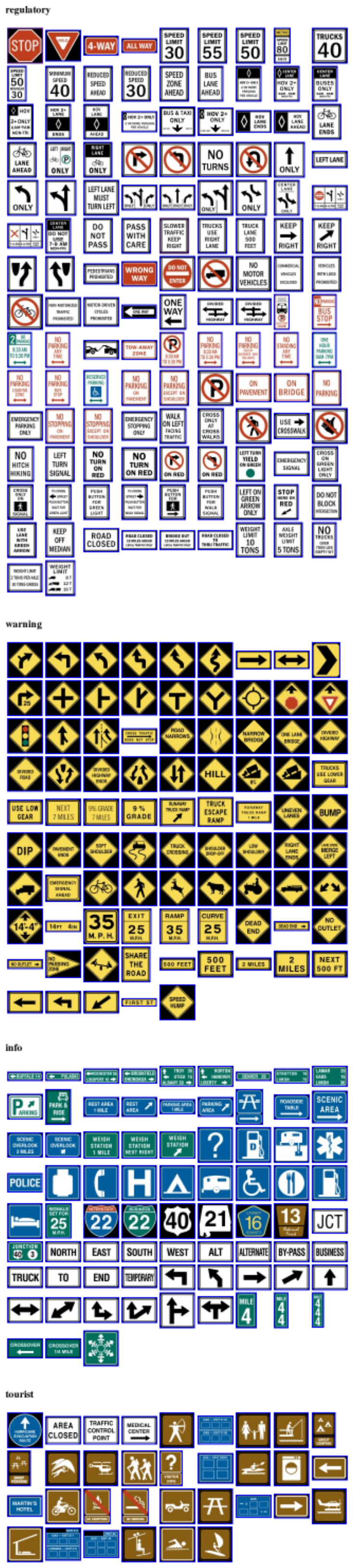

Figure 12. A partial index of public domain vector-graphics road sign images.

mation up the bounding hierarchy, then during traversal we will perform early exit testing to cull out obscured geometry.

Splitting and Merging. As the camera moves, we will 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 will 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 will only use this data to track the locations of exterior floors, doors, and windows, and hence decorate 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.

Photo Analysis. The campus maps provide an excellent classification of the university grounds into grass, sidewalk, roads, parking lots, and buildings. However, consider the roads—the maps describe the boundaries of the roadway, but give no details on the composition or color of the road. We plan to use the campus aerial photos to extract this additional appearance information for all parts of the campus.

It is easy to overlay a photograph as a texture on existing geometry, but this technique is limited by the photograph's resolution and also mixes together the desired and the unwanted geometry (such as cars on the roadway) and lighting (such as shadows and sky light). For example, the photographs in Figure 10 are incorrect where new buildings have been constructed since the photos were taken.

Instead, we plan to use the existing geometry to estimate and divide out the sky and direct sun lighting, then use the photographs to initialize a procedurally generated albedo texture model. Projecting the image onto a statistical texture model will essentially filter out any errors in albedo estimation or unwanted geometry. We expect a very simple stationary texture model will suffice to represent the large variations in the colors of grass and pavement, with a multiplicative detail texture to provide the structured small-scale detail.

6 Rendering

Rendering geometry for an impostor is quite similar to rendering for 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 scan-line algorithm in software, splat-based rendering, image-based techniques, distribution raytracing, or even physically-based radiation transport.

In this section we will describe our approach for rendering the campus model. The aim is to minimize modeling effort and maximize image quality, given the target rendering rate of around a million pixels per second. As such, there may be sufficient compute time available to pursue high-quality rendering techniques such as distribution ray tracing, quality splatting, or cone tracing.

6.1 Antialiasing

Conventional interactive rendering performs simple point sampling of object boundaries, which results in objectionable stair-step aliasing along object boundaries. Texture maps take advantage of the client graphics hardware's builtin texture antialiasing. By preparing our impostors with antialiased edges, even object boundaries are antialiased at very low cost. We will prepare impostors with antialiased edges using our analytic trapezoid-based rasterization system.

Because an impostor is rendered for a fixed display orientation, another promising technique is to include anisotropic texturing while preparing the impostor. Impostor display could then use the hardware's simple and fast interpolation strategy, but because the impostor is already anisotropically filtered, the display quality should approach that of true anisotropic texturing.

6.2 Reflections

Reflection, including blurry reflection, can be integrated into the impostor images just like all other appearance quantities. However, because unlike diffuse emission, reflected images can change drastically depending on the viewpoint. Thus, for nearby geometry including reflections in the impostor would result in a very short impostor lifetime.

Instead, we will normally compute reflections on the client machine, using the usual environment map techniques commonly found on graphics hardware. The environment map could be: a fixed precomputed image; a new image rendered by the client based on the current impostors; or could even be rendered on and sent from the parallel server.

If reflections are rare, such as in outdoor scenes, they could be rendered entirely on the client using the full geometry of the reflective object. If reflections are more common, or more detail is required, the server could compute a normal map image which the client would use to compute per-pixel reflection.

6.3 Motion blur

Motion blur is an important effect for quickly-moving objects. Motion blur is normally computed by averaging together a set of samples distributed in time along the interval between frames. Rendering motion blur via this set of samples is simple, but because the cost is thus linear in the blur distance (e.g., a 16-pixel blur requires 16 separate renderings), motion blur is rarely performed in interactive rendering systems.

However, impostors provide two important improvements for motion blur. First, texture maps can be redrawn again and again quite cheaply, which decreases the cost of motion blur computed using the usual algorithm. Second, unlike polygons, impostors have the property that a rendering of two impostors can be itself treated as an impostor. This property is used by our novel algorithm for efficiently computing motion blur: fast exponentiation blur.

We define an operator T^i that represents a shift by i/n of the frame time. Conventional motion blur calculates the output blur image B as the average of the input geometry G at each of the shifts T^i:

$$B = \frac{\sum_{i=0}^{n-1} T^i G}{n}$$

But if the operator T has the "polynomial" property, that $T^i = T^{i-j}T^j$, then we can calculate B more quickly by factorizing the product as (for even n):

$$B = \frac{(1 + T^{n/2}) \sum_{i=0}^{n/2-1} T^i G}{n}$$

For example, if M is a rotation/translation/scale 2D image matrix that takes one frame to the next, and we define $T^i = M^{i/n}$ as simple matrix exponentiation, then T has the polynomial property, since $T^{i-j}T^j = M^{(i-j)/n}M^{j/n} = M^{i/n}M^{-j/n}M^{j/n} = M^{i/n}$. Note that matrix exponentials, and hence fast exponentiation blurring, can handle simple linear translation, but also rotation, scaling, skew, and any combination thereof.

The factorization process corresponds to computing a large blur (e.g., with $n = 16$) by first computing a small blur image (e.g., with $n/2 = 8$), call it S, then computing $S + T^{n/2}S$—that is, adding a shifted version of the small blur image back into itself. Note that this is exactly the same repeated self-multiplication trick used in the numerical fast exponentiation algorithm.

We can repeat this factorization process to calculate the blur in just $\lg n$ steps (for n a power of two):

$$B = \frac{\prod_{j=0}^{j=\lg n-1} (1 + T^{2^j}) G}{n}$$

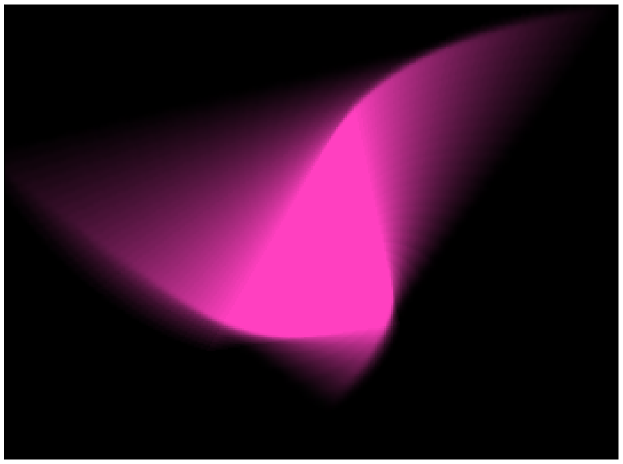

Figure 13. A small central purple polygon, motion blurred for a rotating zoom using fast exponentiation blurring.

This blur is initialized by first drawing the input geometry once at the identity shift T^0. The image is then repeatedly read back, shifted twice as far as before, and added into itself. This can all be implemented quite efficiently on the graphics card using either render-to-texture or render and copy-to-texture. Figure 13 shows a simple polygon blurred using fast exponentiation blurring on the graphics card.

In our parallel rendering system, we will perform motion blurring on the client machine. The server could in theory also perform blurring, but the blurring would then be delayed by the time taken to ship the blurred images across the network, which may be too limiting. Only a limited degree of blurring may be affordable for slower graphics cards, such as blurring only the nearby impostors, or only computing an average blur for the entire screen.

6.4 Lighting

Outdoor scenes are lit by the sun, but sky light and indirect illumination play a very important role in outdoor scenes. As shown in the middle photo in Figure 14, without sky light, areas in shadow appear unnaturally black. This section describes techniques for representing both sun and sky light.

In our prototype, all impostor lighting will be computed on the parallel server, and hence in-

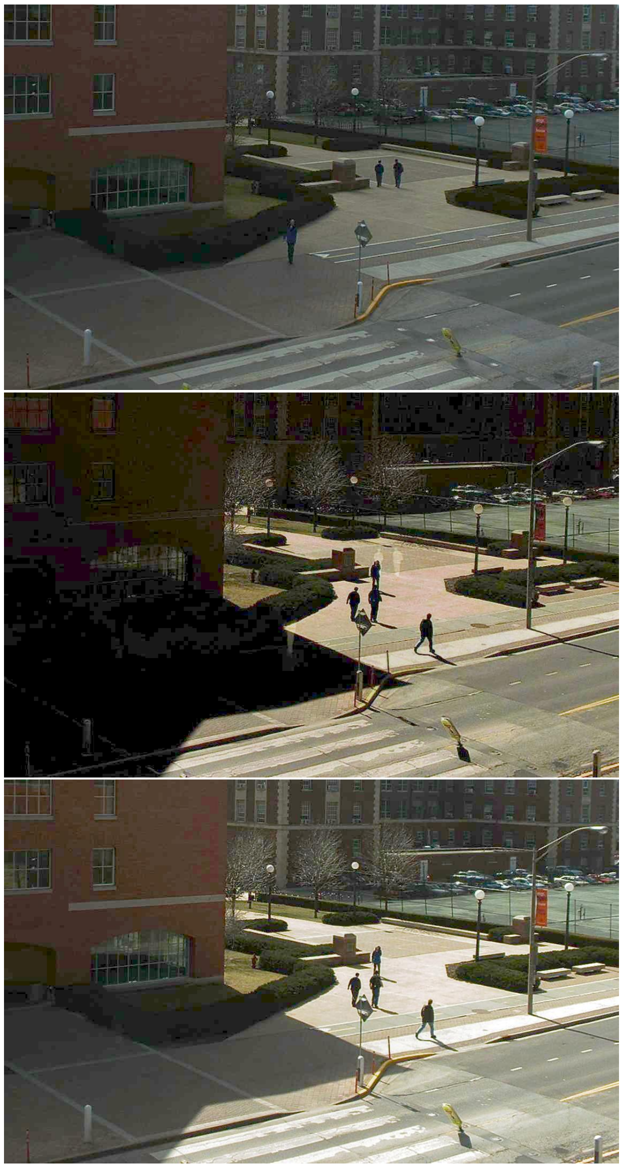

Figure 14. Campus illuminated by sky light (top), sun light only (middle), and sun and sky light (bottom). The top and bottom images are calibrated photographs acquired with a digital camera; the sun-only image is computed from these images by subtracting the sky light image from the sun and sky light image.

cluded in the impostors when shipped to the client. Because the scene is largely static, we can precompute the expensive portions of the lighting, such as the generation of the shadow and sky-light maps.

6.4.1 Direct Lighting

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

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 is extremely accurate. Shadow volumes are a screen-space technique which extrudes, rasterizes, and counts object silhouette edges to determine which points 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. Broadly, raytracing is generally quite slow, shadow volumes are slow for objects with complicated silhouettes (such as trees), and shadow maps display sampling and resolution problems.

A good survey of existing soft shadow modifications to these techniques is presented by Hasenfratz et al. [HLHS03]. We begin with the realtime soft shadow-map technique of Kirsch and Döllner [KD03], which can be computed fully on the graphics hardware, but only represents the inner penumbra. We have extended this technique to capture the outer penumbra, which allows it to be physically correct for at least some cases. We have also found a formulation of this technique which is substantially free from interpolation artifacts.

Our novel formulation stores two depths for each light-source ray: c, the depth of the first occluder (this can be approximated by the usual shadow map depth); and p, the signed depth of the beginning ("limit") of the penumbra. The sign of the penumbra limit p is chosen so it decreases away from the object. From light source to the first occluder, the light source is fully visible; from first occluder to the beginning of the penumbra, the light source is fully obscured; beyond the end of the penumbra, more and more of the light source is visible.

Inside the penumbra, the fraction L of the light source visible at a depth z from the light source is simply:

$$L(z) = 0.5 + 0.5 * \frac{c - p}{c - z}$$

It can be shown that this *penumbra limit map* can exactly represent the shadow of a simple half-infinite plane occluder illuminated by an infinitely distant light source. For arbitrary occluders or light sources, the shadow complexity of a light-source ray can be arbitrarily large, so the penumbra limit map is no longer exact, but it provides a very fast yet reasonable approximation, as shown in Figure 15. The corresponding penumbra limit maps are shown in Figure 16.

We currently generate the penumbra limit map in software, which our unoptimized implementation takes under one second to compute a 1024x1024 penumbra limit map. It should be trivial to traverse the penumbra limit map in one pass of the current generation of programmable graphics hardware, though for computing impostors on the parallel machine we may render them in software.

The disadvantages of this technique are that, like any shadow map, we must choose an appropriate resolution for the penumbra limit map. 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.

6.4.2 Indirect Lighting

We will approximate sky light and other indirect illumination using a coarse uniform *voxel lighting grid* throughout the scene. This approach calculates only the vertical direction of Greger's irradiance volume [GSHG98]. While Greger and later work calculates the lighting based on a cubemap or other irradiance samples, we propose a faster approximation.

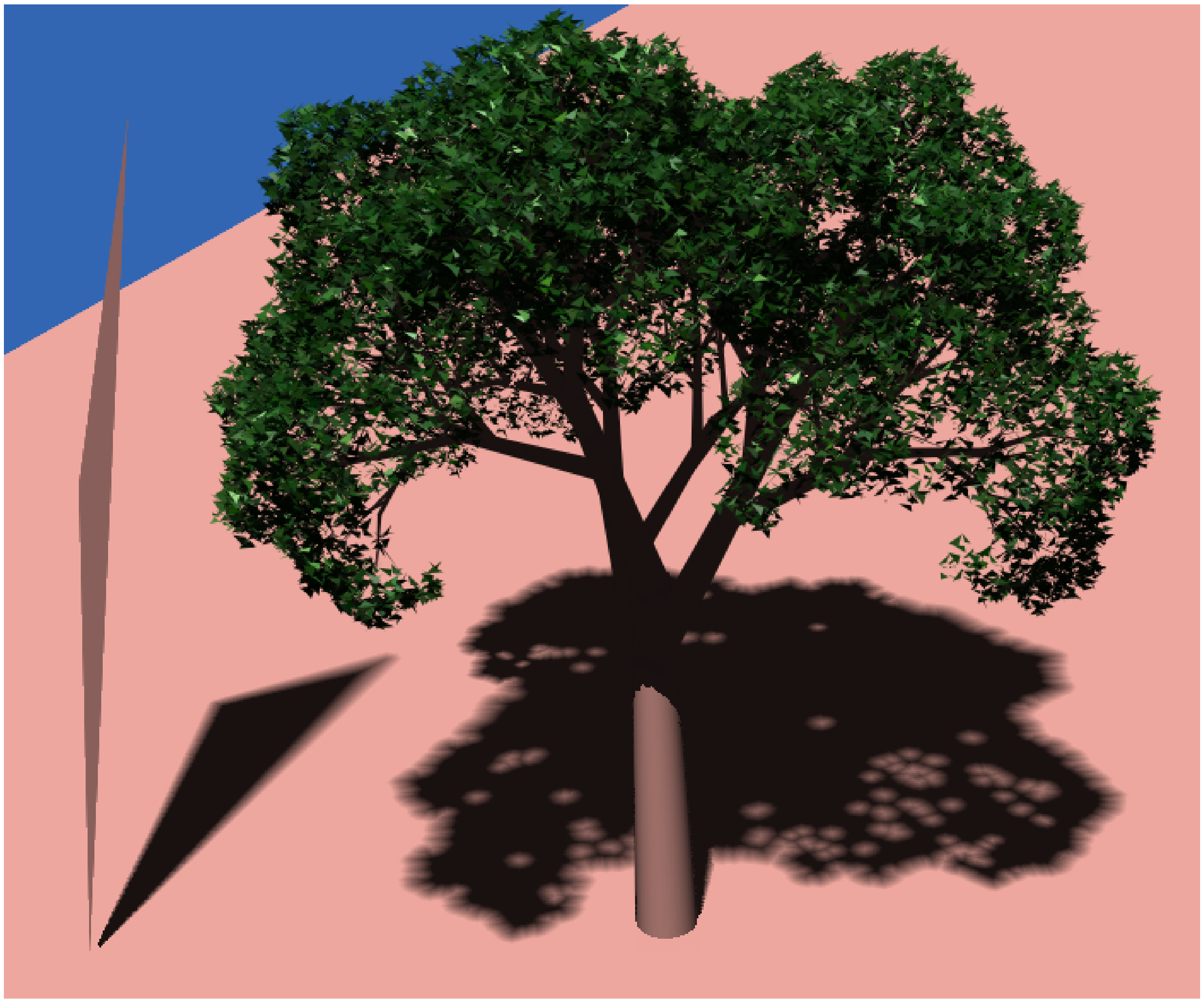

Figure 15. A scene with soft shadows rendered via the penumbra limit map technique. Light is cast by an extended light source offscreen to the left, which has an angular size of three degrees. The light is occluded by a small polygon standing on its tip (left) and a tree (right). Note how the polygon's shadow gets softer as it travels. The shadow map resolution is 1024x1024.

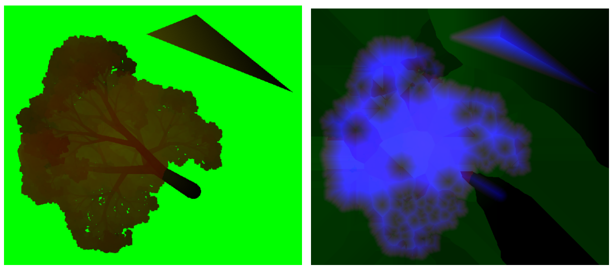

Figure 16. Portions of the actual penumbra map image. Left is the original depth map, with the red channel showing the closest geometry and the green channel showing the farthest, where darker values indicate farther depths from the light. Right is the actual penumbra limit map—red is the depth to the first portion of shadow, green is the depth to the active occluder c, and blue is the signed penumbra limit depth p.

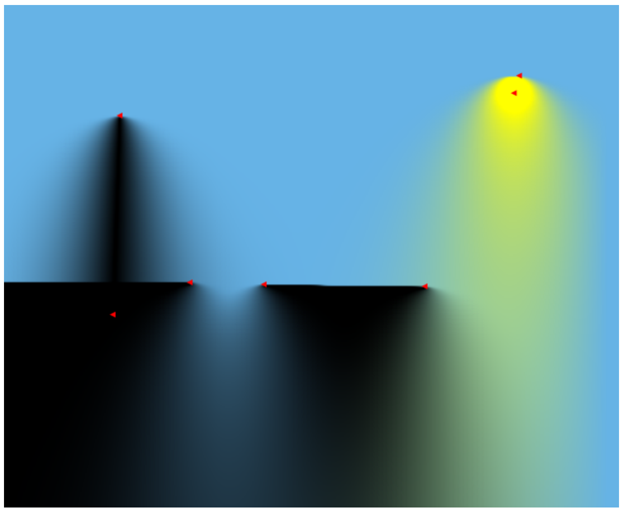

Figure 17. A cross section of the sky lighting computed by our one-sweep approximation to global illumination. Blue sky light diffuses down from the top and is blocked by several black shapes, and a yellow sphere reflects light downwards.

To calculate the lighting grid, we first rasterize the geometry into a voxel grid. Each voxel contains an isotropic approximation to the contained geometry's occlusion and emission, where emission includes diffuse reflection as well as source brightness terms. Our representation of this isotropic occlusion and emission is simply an RGBA alpha-weighted color: low alpha corresponds to small amounts of occlusion, high alpha to large amounts of occlusion, and RGB is the emissive color.

This voxel grid could actually be rendered by the usual volumetric rendering techniques; but we will use a simple directional approach to compute sky lighting. We will sweep through the occlusion and emission voxel grid from top down: higher voxels occlude sky light and cast emitted light toward nearby lower voxels, as illustrated in Figure 17. This diffusion-based approach has the disadvantage that light spreads in all directions, rather than traveling in straight lines like actual light in a vacuum. However, radiant energy is neither created nor destroyed during propagation; and the system is actually physically correct for a situation with an appropriate participating media.

The result is a voxel lighting grid giving a local approximation of the illumination arriving from above, including sky and indirect lighting. For higher quality, one could add sweeps from other directions to capture illumination from below and from the sides. The natural generalization of this technique to many directions is the discrete-angles sweep method for radiation transport described by Plimpton et al. [PHBI00].

6.5 Trees/Foliage

Trees and other foliage will be procedurally generated using an *Iterated Function System* (IFS). This will allow us to use our IFS bounding method [LH03] to construct a tight bounding volume for the procedurally generated geometry. Trees will be rendered using a simple splat-based renderer for the leaves, and antialiased line segments for the trunk and branches.

Each leaf is, when far away, rendered as a single splat, with alpha scaled to the leaf's area coverage. When close up, the leaf is drawn as an antialiased leaf texture map. Leaf colors are pseudorandomly selected from a distribution of leaf colors; with the pseudorandom seed value set consistently for each leaf cluster.

To speed up rendering for the leaves, we only actually expand the IFS maps for the high levels of the tree; once we are close to the individual leaves, we switch to a precomputed *cluster* of 3D leaf locations. Using leaf clusters does not affect the image, but reduces the number of matrix multiplies needed during display dramatically.

We perform the Z-sort for trees using a two-pass radix sort. 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 15.

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

6.6 Buildings

The buildings on campus are predominantly rectangular, so we will render each wall as a separate impostor. The buildings are also almost all brick, so we will work to procedurally generate the appearance of brick. The wall impostors can always be drawn aligned with the walls, so even up close, individual bricks can be rendered as simple aliased rectangles. Mortar lines can be omitted for distant walls, and drawn in as antialiased lines for nearby walls.

Very distant windows will be drawn in as simple recessed dark rectangles. Closer windows will use texture combiners to alpha-composite the wall image atop a static neighborhood environment map. For very nearby windows, we will dynamically generate the reflected image on the client by drawing in a stenciled, reflected version of the nearby geometry. In each case, we can control the reflection intensity and add lit dust by adjusting the transparency and opacity of the wall impostor.

6.7 Sky

Our sky will be procedurally rendered, based on a single-bounce analytical approximation to Rayleigh and Mie scattering similar to Musgrave's sky model [Mus93]. This should allow us to compute the color of the sky for any time of day, location, and atmospheric condition.

6.8 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 will use this pregenerated geometry to represent cars, lightpoles, and traffic lights.

7 Conclusion

This paper contains a variety of work related to impostor-based rendering. We now summarize our contributions.

- We have designed a novel high-performance rendering architecture, parallel impostors. We have shown parallelism and impostors can allow dramatically increased performance and improved image quality.
- We have formulated a new performance model for modern graphics hardware, described in Section 2.1 and tested in detail in Appendix A.2.
- We have described a novel impostor-based method to compute motion blur, the fast exponentiation blur of Section 6.3. This method of blurring is asymptotically faster than known approaches.
- We have described a novel method to approximate blurry shadows, the penumbra limit map technique of Section 6.4.1.

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. We have shown how this technique can be used to extend the state of the art in interactive rendering quality, by making fully antialiased rendering affordable. Finally, we have demonstrated the technique is applicable to very large outdoor environments.

References

[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.

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

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

[CW93] Shenchang Eric Chen and Lance Williams. View interpolation for image synthesis. In *Proceedings of the 20th annual conference on Computer graphics and interactive techniques*, pages 279–288. ACM Press, 1993.

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

[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.

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

[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.

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

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

[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.

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

[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.

[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.

[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.

[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.

[Mus93] F. Kenton Musgrave. *Methods for Realistic Landscape Imaging*. PhD thesis, Yale University, 1993.

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

[Par04] 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.

[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.

[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.

[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/~johns/.

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

[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.

A Graphics Operations

This section describes the basic graphics hardware features used in this work.

A.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.

A.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 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.

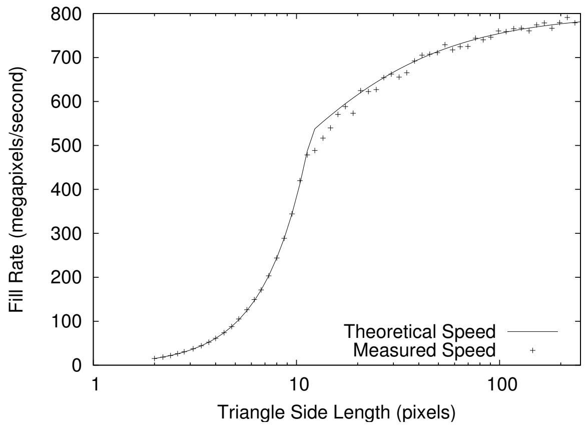

Figure 18. Fill rate for nVidia GeForce3/Athlon.

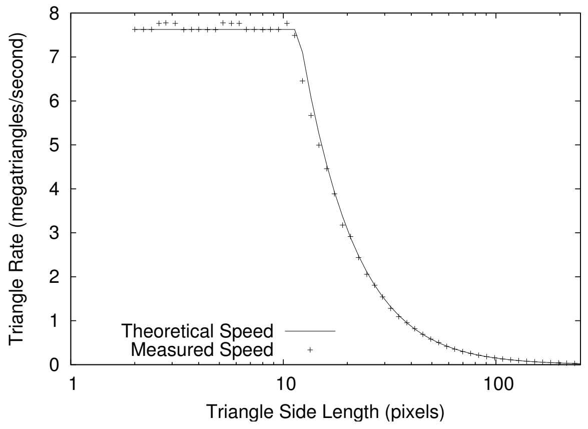

Figure 19. Triangle rate for nVidia GeForce3/Athlon.

A.2.1 nVidia GeForce3/Athlon

Figure 18 and Figure 19 show 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 a plain color pixel takes 1.250 ns. Adding texturing increases the cost to 2.091 ns. Alpha blending and texturing costs 2.616 ns. Depth and texturing costs 2.497 ns. Alpha, depth buffer, and texturing costs 3.022 ns. Mipmapped texture upload (with MIPMAP_SGIS) costs 1.282 us + 43.749 ns/pixel.

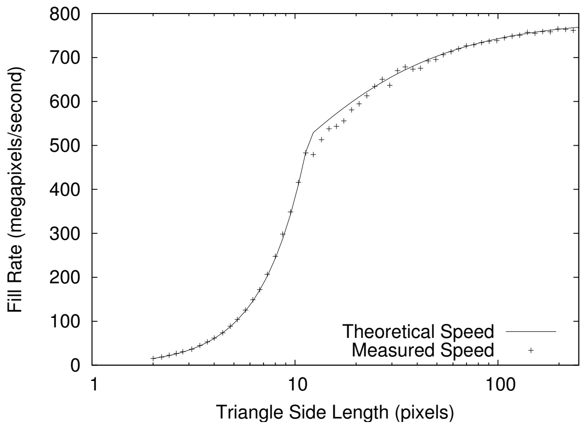

Figure 20. Fill rate for nVidia GeForce3/Pentium 4.

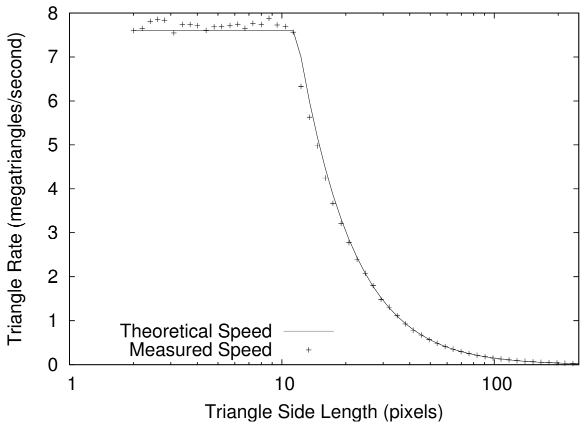

Figure 21. Triangle rate for nVidia GeForce3/Pentium 4.

A.2.2 nVidia GeForce3/Pentium 4

Figure 20 and Figure 21 show 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 a plain color pixel takes 1.427 ns. Adding texturing increases the cost to 2.250 ns. Alpha blending and texturing costs 2.812 ns. Depth and texturing costs 2.636 ns. Alpha, depth buffer, and texturing costs 3.144 ns. Mipmapped texture upload (with MIPMAP_SGIS) costs 1.244 us + 15.588 ns/pixel.

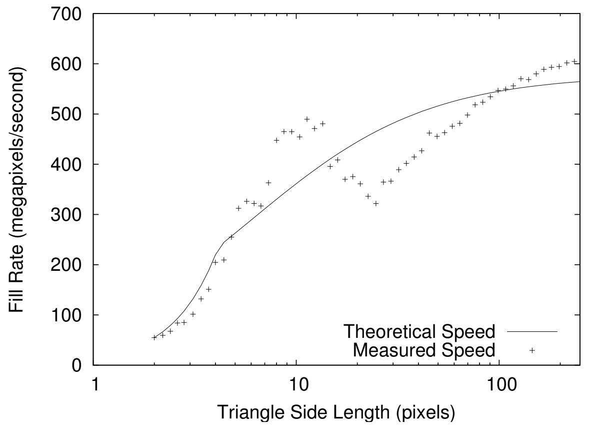

Figure 22. Fill rate for ATI Mobility Radeon 9000.

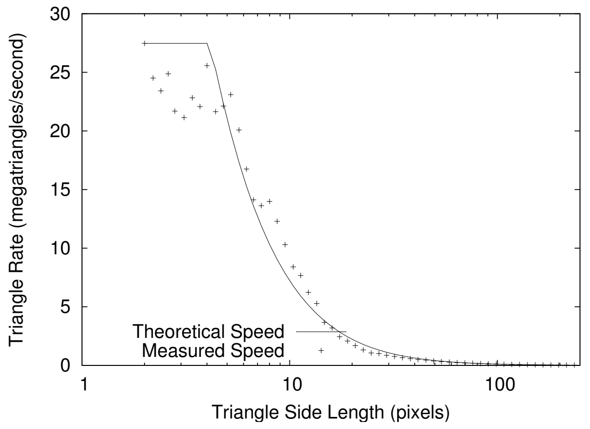

Figure 23. Triangle rate for ATI Mobility Radeon 9000.

A.2.3 ATI Mobility Radeon 9000

Figure 22 and Figure 23 show 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 a plain color pixel takes 1.586 ns. Adding texturing increases the cost to 3.863 ns. Alpha blending and texturing costs 6.108 ns. Depth and texturing costs 5.974 ns. Alpha, depth buffer, and texturing costs 8.205 ns. Mipmapped texture upload (with MIPMAP_SGIS) costs 0.923 us + 12.838 ns/pixel.

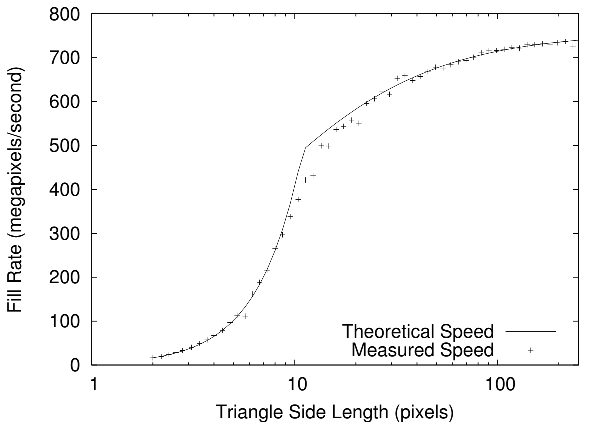

Figure 24. Fill rate for nVidia GeForce2 Ti.

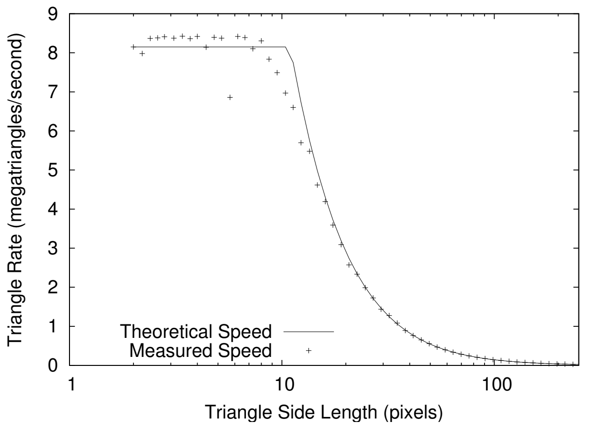

Figure 25. Triangle rate for nVidia GeForce2 Ti.

A.2.4 nVidia GeForce2 Ti

Figure 24 and Figure 25 show 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 a plain color pixel takes 1.542 ns. Adding texturing increases the cost to 2.668 ns. Alpha blending and texturing costs 4.480 ns. Depth and texturing costs 3.708 ns. Alpha, depth buffer, and texturing costs 5.499 ns. Mipmapped texture upload (with MIPMAP_SGIS) costs 1.528 us + 33.313 ns/pixel.

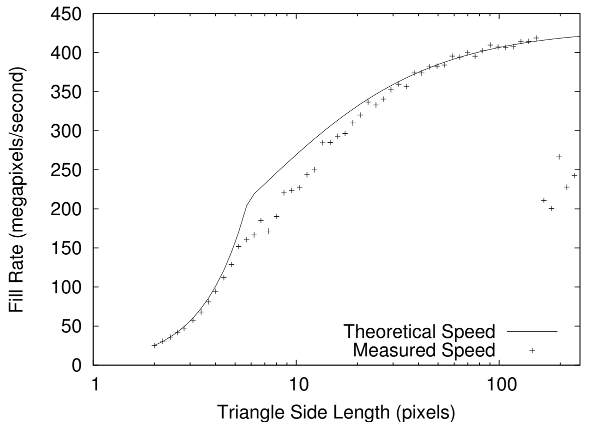

Figure 26. Fill rate for nVidia GeForce4 MX 440.

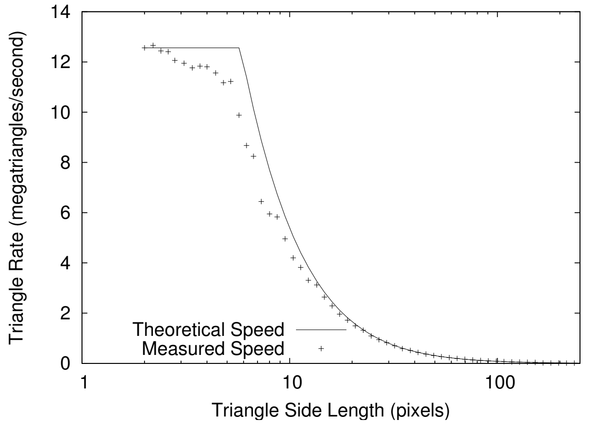

Figure 27. Triangle rate for nVidia GeForce4 MX 440.

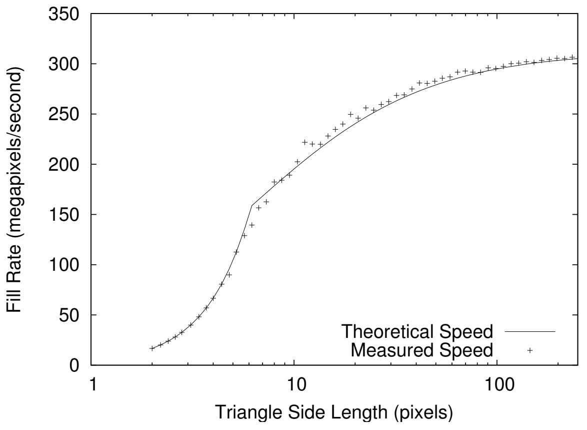

Figure 28. Fill rate for nVidia Quadro2 MXR.

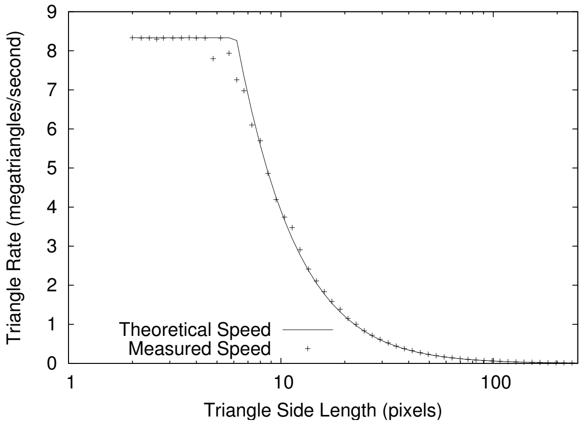

Figure 29. Triangle rate for nVidia Quadro2 MXR.

A.2.5 nVidia GeForce4 MX 440

Figure 26 and Figure 27 show 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 a plain color pixel takes 2.572 ns. Adding texturing increases the cost to 4.951 ns. Alpha blending and texturing costs 6.888 ns. Depth and texturing costs 7.143 ns. Alpha, depth buffer, and texturing costs 9.212 ns. Mipmapped texture upload (with MIPMAP_SGIS) costs 0.690 us + 10.165 ns/pixel.

A.2.6 nVidia Quadro2 MXR

Figure 28 and Figure 29 show 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 a plain color pixel takes 3.240 ns. Adding texturing increases the cost to 5.303 ns. Alpha blending and texturing costs 9.344 ns. Depth and texturing costs 7.644 ns. Alpha, depth buffer, and texturing costs 12.106 ns. Mipmapped texture upload (with MIPMAP_SGIS) costs 1.206 us + 14.884 ns/pixel.

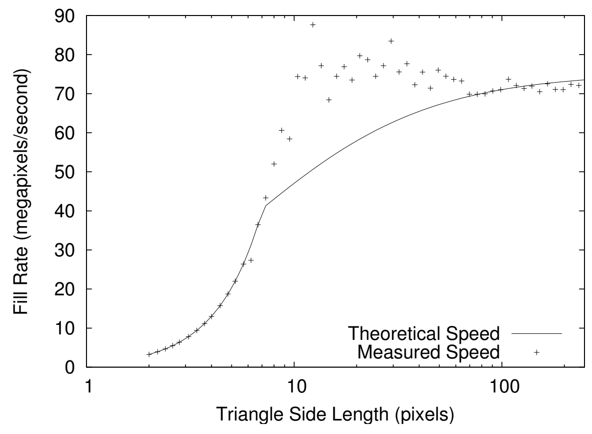

Figure 30. Fill rate for Sun Creator-3D.

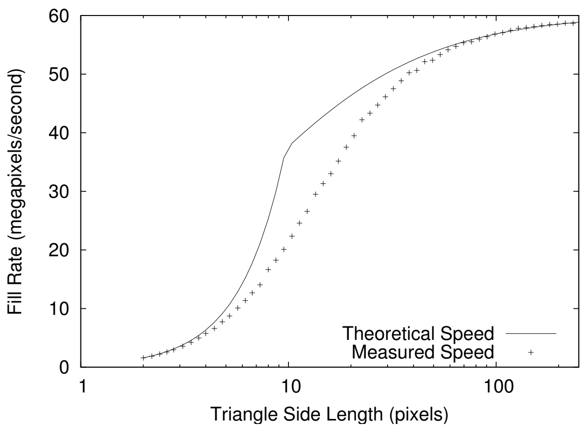

Figure 32. Fill rate for Mesa Software Rendering.

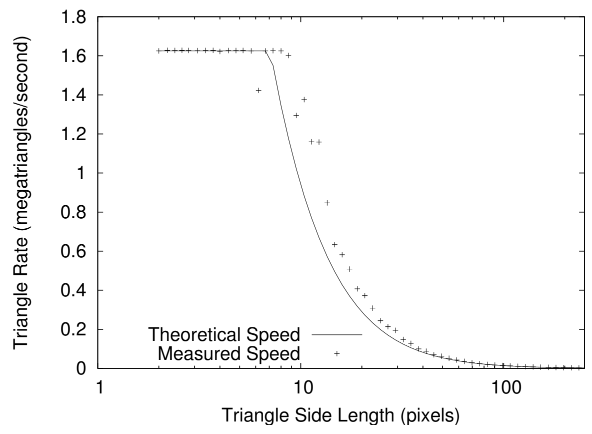

Figure 31. Triangle rate for Sun Creator-3D.

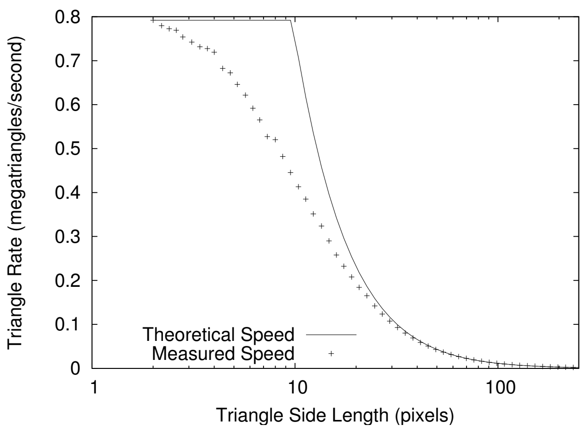

Figure 33. Triangle rate for Mesa Software Rendering.

A.2.7 Sun Creator-3D

Figure 30 and Figure 31 show 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 a plain color pixel takes 14.142 ns. Adding texturing increases the cost to 151.531 ns. Alpha blending and texturing costs 151.500 ns. Depth and texturing costs 151.536 ns. Alpha, depth buffer, and texturing costs 151.592 ns. Mipmapped texture upload (with MIPMAP_SGIS) costs not available.

A.2.8 Mesa Software Rendering

Figure 32 and Figure 33 show 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 a plain color pixel takes 16.745 ns. Adding texturing increases the cost to 180.310 ns. Alpha blending and texturing costs 222.258 ns. Depth and texturing costs 188.139 ns. Alpha, depth buffer, and texturing costs 230.198 ns. Mipmapped texture upload (with MIPMAP_SGIS) costs 1.210 us + 40.008 ns/pixel.