Image-Based Exploration of Massive Online Environments

Siddhartha Chaudhuri Daniel Horn Pat Hanrahan Vladlen Koltun

Stanford University *

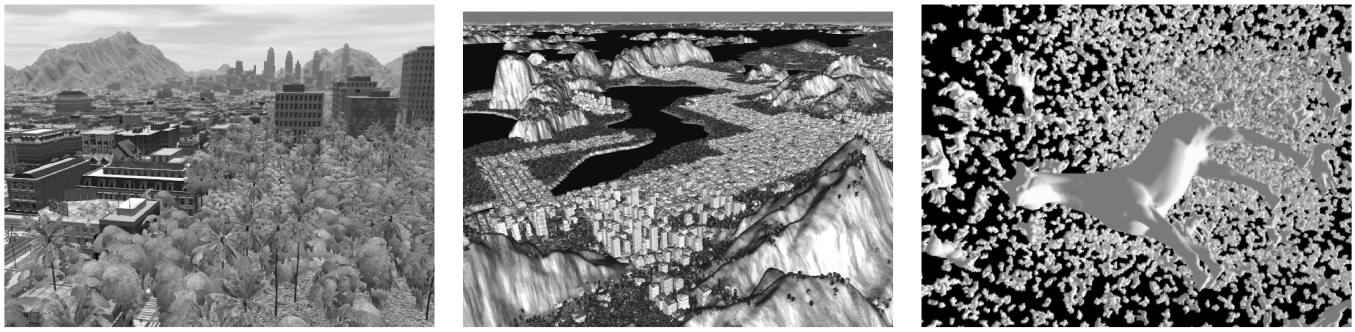


Figure 1: *Two scenes rendered interactively over a broadband network on commodity hardware. (Left and center) A 25-billion polygon, 10,000 km^2 Earth-like scene. (Right) A fully three-dimensional scene with 3 billion polygons.*

Abstract

This paper presents a system for interactive exploration of massive, detailed virtual environments over a broadband network. We build upon the hierarchical image-based framework pioneered by Shade et al. [1996] and Schaufler and Stürzlinger [1996], introducing key adaptations for scalability. A cluster of servers maintain a hierarchy of depth images of bounded regions of the scene. A client displays the scene using a logarithmic set of depth images that can be maintained under constant bandwidth independent of scene size. We report on techniques used to overcome the daunting visual quality issues encountered with image-based rendering of general unstructured scenes with billions of polygons on commodity hardware over a wide-area network. Experimental results are reported on scenes that exemplify the extreme demands of large-scale online worlds.

CR Categories: I.3.2 [Computer Graphics]: Graphics Systems—Distributed/network graphics;

Keywords: network graphics, image-based rendering, depth images, orthoviews, splatting, massive virtual environments

1 Introduction

Detailed three-dimensional models of vast geographic environments are becoming common. Services such as Google Earth and Microsoft Virtual Earth provide large geo-referenced databases of imagery and terrain data, and three-dimensional models of buildings and vegetation are rapidly being integrated [Google Inc. 2008]. Semi-autonomous modeling of extended urban areas is now within reach [Bosse et al. 2004; Thrun and Montemerlo 2005]. Online worlds that do not correlate with real spaces already host large numbers of participants [Miller 2007]. Such worlds now contain hundreds of thousands of continuous acres covered with detailed user-generated architecture and flora.

A critical bottleneck in engineering such systems is the display of complex 3D environments to many simultaneous users in real time over the Internet. Popular virtual worlds contain terabytes of data and are far too large to fit on any single machine. Further, they change over time and contain many different types of objects. A common solution in practice is to transmit and display only nearby objects. The typical cutoff radius is a few hundred meters or less, and the abrupt transition to empty space is frequently obscured by fog. This hinders participants' experience by precluding long or panoramic views and limits the visual impact of the worlds.

*e-mail:{sidch,danielrh,hanrahan,vladlen}@cs.stanford.edu

The computer graphics research community has advanced a variety of promising approaches to the display of large environments. Our application scenarios feature a combination of extreme scale and complexity, large numbers of distinct heterogeneous objects, and high visibility (see Figure 1). These factors conspire against exclusive reliance on visibility preprocessing, geometric simplification, and impostors. They also limit the applicability of approaches specialized for urban models and terrains [Cignoni et al. 2007; Losasso and Hoppe 2004].

This paper evaluates the suitability of hierarchical image-based rendering [Shade et al. 1996; Schaufler and Stürzlinger 1996] for handling massive online worlds. The first contribution of our work is adapting the framework to guarantee performance under limited bandwidth. To this end, each cell in the hierarchy contains image-based representations of only the corresponding bounded region of the scene, and the representations are collectively valid for all possible viewpoints. Our second major focus is high visual quality. We thus undertake a systematic exploration of the visual artifacts stemming from the large-scale use of image-based rendering. We report practical approaches to handling these issues, including a method for splatting depth images representing large collections of heterogenous objects.

The main contribution of our work is demonstrating the viability of image-based rendering at previously unseen scales. We evaluate the system on a 10,000km^2 model with 25 billion polygons. The model depicts a realistic Earth-like environment with sea and landmasses covered by forests and cities. To the best of our knowledge, this is the largest pre-modeled general environment reported in the literature of online exploration to date[1]. We also report results on a fully three-dimensional test scene, consisting of about 30,000 objects, each with roughly 100,000 polygons.

2 Related Work

Image-based systems. Shade et al. [1996] and Schaufler and Stürzlinger [1996] represent a scene with a hierarchy of image-based impostors. Impostors are periodically recomputed to remain valid for the current viewpoint. More distant impostors can be recomputed less frequently. Chang et al. [1999] use Layered Depth Images (LDIs) [Shade et al. 1998] in each cell of the hierarchy. By eliminating parallax error and reducing disocclusion artifacts, LDIs can be valid for a greater range of possible viewpoints. Max

[1]Note the contrast to instance-based worlds of Wand et al. [2001] and systems such as Google Earth that focus on terrain and nearby objects.

[1996], in the context of tree rendering, suggests using multiple images to cover the full range of possible viewing directions – an insight adopted by our system.

Aliaga et al. [1999] describe the landmark Massive Model Rendering system (MMR), which divides the scene into a grid of cells and represents the distant parts of the scene, as seen from the center of a cell, with a cubemap of six textured depth meshes [Darsa et al. 1998; Sillion et al. 1997]. A bottleneck of the system for our application domain is the preprocessing load: Each per-cell set of textured depth meshes represents the entire scene and generating these sets for all cells takes quadratic time.

Some methods take the particular scene geometry into account in positioning images. Architectural walkthroughs can be accelerated by placing impostors for regions visible through *portals* [Rafferty et al. 1998]. More generally, Aliaga and Lastra [1999] selectively replace octree cells with images to guarantee a minimum framerate. Jeschke et al. [2005] reduce redundancy by noting that impostors for more distant cells are valid over a greater range of viewpoints. Cignoni et al. [2007] introduced BlockMaps for urban models.

Wilson and Manocha [2003] choose viewpoints to maximally capture portions of the scene invisible from previously selected ones. The depth buffers of images rendered from previous viewpoints are used to approximate the "known" portions of the scene.

The priority rendering approach of Regan and Pose [1994] observed that distant environment maps can be rendered less frequently than those up close, implying a constant number of updates at constant velocity. Similar arguments apply to mipmaps [Tanner et al. 1998].

Other rendering acceleration techniques. Geometric level-of-detail techniques [Luebke et al. 2002] produce coarse representations for distant objects. The method has been successfully applied to terrains: mipmapped textures [Williams 1983] coupled with clipmaps for prefetching and culling [Tanner et al. 1998; Losasso and Hoppe 2004] make it possible to explore the Earth's terrain in real-time.

Airey et al. [1990] and Teller and Séquin [1991] pioneered visibility culling algorithms for architectural models. A long line of research pursues visibility techniques for urban environments, culminating with the approach of Leyvand et al. [2003]. Cohen-Or et al. [2003] provide an excellent survey.

Point-based systems [Rusinkiewicz and Levoy 2000] handle huge models in real time by sampling points from the surfaces of the models. Rusinkiewicz and Levoy [2001] also develop view-dependant progressive transmission that adapts point-based methods for remote rendering. Wand et al. [2001] randomly sample and render triangles from the scene.

Remote rendering. As model size increases, the server may take on some rendering tasks to ease the load on the client. Schmalstieg [1997] performs culling and occlusion detection on the server. Mann and Cohen-Or [1997] and Yoon and Neumann [2000] present image-based approaches to remote rendering. Schmalstieg and Gervautz [1996] and Teler and Lischinski [2001] consider bandwidth-limited remote walkthroughs and develop heuristics to guide the transmission of geometry.

3 Assumptions and Properties

The following assumptions underlie some of our design choices.

Steady world. We assume that most spatially significant objects (mountains, buildings, large vegetation, etc.) are not in continuous motion. This is a well-established assumption in online exploration of massive environments and reflects the evident realities of the physical world. Our system does support a bounded quantity of moving objects, which are simply rendered as geometry.

Bounded velocity. To maintain a seamless view over constant network bandwidth, the participant's movement speed must be bounded. In our system the velocity limit grows exponentially as the participant moves away from geometric detail, allowing rapid zooming towards and away from populated regions. Speeds beyond the limit are also supported, but visual quality degrades correspondingly.

Bounded density. The world can grow to arbitrary size, but the local complexity of geometry must be bounded by a constant. This means that the density of geometry in any particular area (say, number of polygons within a cubic meter) cannot exceed a certain threshold. This is again an accepted assumption in the online exploration literature [Polis et al. 1995]. It guarantees a bound on the amount of local geometry (within a fixed radius around the viewpoint) that can be reached in a unit of time within the velocity bound.

Our system has the following properties, which are not together present in any previous work.

- It is appropriate for worlds with fine detail at large scales.
- It can accommodate any environment, not just special classes such as urban models and terrains. It can handle an unstructured and unpredictable variety of geometric content. Virtually no human intervention is required to adapt the algorithm to the specific structure of the scene.
- The server-side storage overhead is comparable to or smaller than the original world data.
- Server-side preprocessing takes linear time. It is easily parallelizable on commodity clusters.
- Changes to the environment are quickly and efficiently processed and the update can be transmitted to clients within a short (though not real-time) period of time.
- The required bandwidth is bounded as a function of maximal user velocity and is independent of the size of the world. In practice, the consumed bandwidth is only fractionally larger than the bandwidth required to maintain nearby geometry.
- The lighting and shading of the environment can be customized in real time on any individual client (Figure 3).

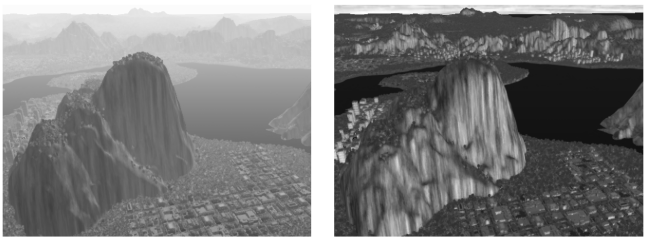


Figure 3: *Different client-side lighting for the same environment.*

4 System Architecture

Our system has a distributed client-server architecture, in which each client renders its view of the world using geometry for nearby objects and a set of depth images for more distant ones.

4.1 Server

The server-side preprocessing partitions the world geometry into a regular grid of cells. These cells form the leaves of an octree. The subdivision can have different levels along different axes if the environment has different extent along them.

The model is distributed among available server nodes for preprocessing. Each of n servers is responsible for a collection of subtrees that represent $1/n$-th of the world.

For each cell of the octree, the preprocessing creates orthographic depth images, or *orthoviews*, of the cell's content from a number of *canonical directions*. Computation and storage costs increase linearly with the number of directions, hence we strive to keep this number small. We use a subset of the principal directions of a cube, namely $\{-1, 0, +1\}^3 \setminus \{0, 0, 0\}$. For our test scenes this provides good visual quality (Figure 5). This choice would not be suitable for some environments, like a regular grid of objects where the axes of maximum occlusion are precisely these principal directions: for such scenes a different set of canonical directions would

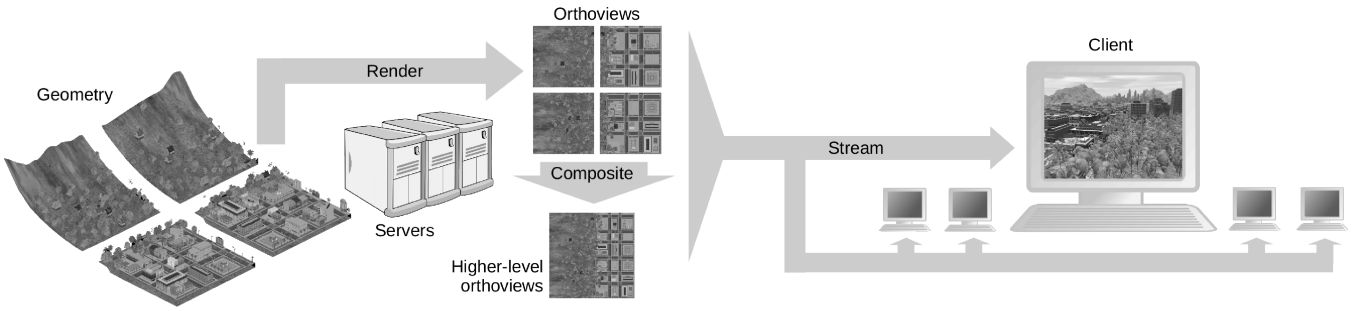


Figure 2: *An overview of the system.*

be chosen. Verifying that the utilized set of canonical directions is appropriate for a given environment is one of the few responsibilities of the system operator.

Orthoviews have color, depth and normal components. Their resolution is determined as follows. Assume that ignoring occlusion, each visible surface is seen along some canonical direction at an angle of at most θ from the perpendicular (e.g. the face views of the cube give $\theta \approx 55$ degrees). Let a leaf cell have linear extent s, the nearest objects displayed via depth images be at distance r, and the output resolution be $p \times p$ pixels for a field-of-view of (horizontal) angle α. An orthoview resolution of $q \times q$ pixels, where

$$q = \frac{p}{\cos\theta} \times \frac{s}{2\,r\tan\frac{\alpha}{2}},$$

implies that adjacent orthoview pixels for a surface sampled at angle θ from the perpendicular reproject to adjacent pixels on the output display.

Orthoviews for leaf cells are created on the respective servers by rendering the contents of each cell from geometry. The rendering is done with no shading, to allow customized shading on the client side. The server node reads in the model for a cell, renders the orthoviews, and then compresses and writes them to permanent storage.

Orthoviews for non-leaf cells are created from their children using 2D compositing (Figure 4). Consider a cell that is part of a subtree handled by a single server. To create an orthoview from a particular projection direction, the corresponding orthoview images from the eight children cells are composited. The resulting image has the same resolution as the original eight but covers twice as much of the world along each axis. The depth of an output pixel is the closest of the depths of all candidate pixels intersecting its projected area in the source images; the color and normal are interpolated from unoccluded candidates within a threshold depth of the closest (see Figure 4).

This compositing can be performed efficiently in graphics hardware. We set up a double-size output buffer, representing the parent's image space. For each child orthoview, we render an axis-aligned quad parallel to the image plane. The quad covers half of the output buffer along each axis. When rasterized, its fragments correspond bijectively to pixels in the child orthoview. We adjust the depth of each fragment to locate it correctly in the parent's projection volume, and assign it the corresponding color and normal. This procedure automatically removes hidden pixels. The result is a double-size depth image that we downsample. Each 2x2 block of pixels maps to a single output sample that is assigned depth, color, and normal as described above.

Orthoviews for the highest-level cells that transcend server boundaries are assembled analogously, the only distinction being that images are exchanged using a network linking the servers. Each high-level cell is assigned in a bottom-up process to a server that handles one of the children cells. Since the highest levels of a hierarchy contain exponentially fewer nodes than the lower ones, this final stage is orders of magnitude faster than the intra-server assembly of lower levels.

Scalability. For a leaf node, a block of geometric data is loaded and its orthoviews are rendered exactly once. For higher-level nodes, orthoviews are composited rapidly using image-based operations. The entire preprocessing step takes linear time and the generated images consume linear space.

By traversing the tree depth-first and caching data on the current path to the root in main memory, each orthoview is written to permanent storage exactly once, never reloaded from it, and transmitted across a network connection at most once (during the top-level inter-server assembly). At any moment, the number of orthoviews in a server's main memory is proportional to the height of the subtree assigned to it; hence the maximum RAM consumed by a server process is logarithmic in the extent of the scene. The time and permanent storage requirements per server are inversely proportional to the number of servers.

Updates. When the model changes locally, the orthoview hierarchy can be quickly updated, since only images on paths from affected cells to the root need to be recomputed. If the octree has n leaf cells, of which k are affected by the change, the time taken to update the orthoviews is $\mathrm{O}(k \log n)$. If the leaf cells are contiguous, this update time is $\mathrm{O}(k + \log n)$.

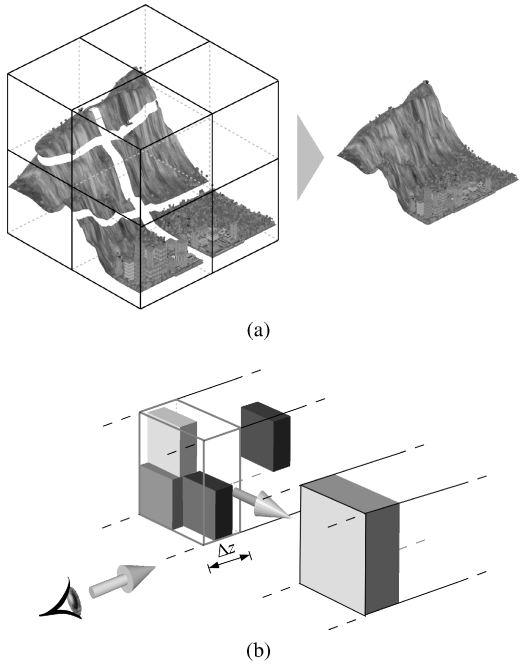


Figure 4: *(a) Child orthoviews are composited to form the parent orthoview. (b) A parent pixel (right) is assigned the minimum depth of child pixels (left) projecting onto it, and the average color and normal of the children in its four quadrants that are within a threshold depth Δz of the minimum.*

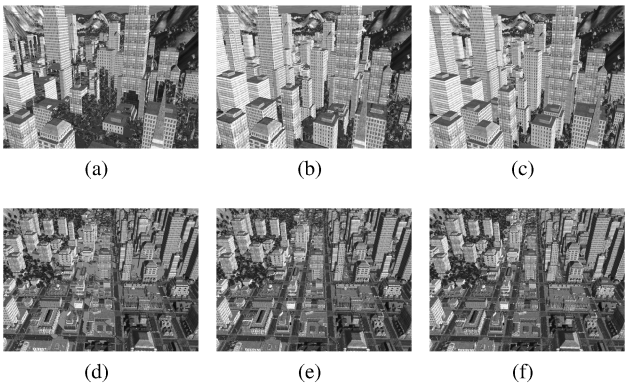


Figure 5: *Varying the number of orthoviews. (a) A view rendered with 5 available canonical projections per cell, showing severe disocclusion artifacts. (b) 9 canonical projections perform considerably better. (c) A reference image rendered from geometry. Note that this is a particularly challenging scene with many long, narrow spaces between buildings. (d) 2, (e) 3, and (f) 5 orthoviews selected per cell by the client, from a maximum of 9 canonical projections. Disocclusion errors from the use of just 2 orthoviews (highlighted in pink) are largely solved by adding a third view. Additional views yield small improvements.*

4.2 Client

A client renders all objects within a fixed radius r around the viewpoint as geometry. r is the minimum distance at which an orthoview pixel (ignoring surface orientation) projects to a single pixel in the final image. Thus all objects within a certain radius have guaranteed image quality. The rest of the world is displayed by rendering a small number of orthoviews (3 by default, see Figure 5) for each cell in a set of octree cells. The orthoviews are chosen to make small angles with the viewing direction. For constant angular resolution, the number of relevant cells from each level of the hierarchy is constant. Thus the entire environment outside radius r can be represented using a logarithmic number of cells. Both geometry and orthoviews are streamed from the servers.

Regan and Pose [1994] show that a cell in a hierarchy need only be updated when the user crosses the boundary of a cell of the same size. Thus, nearby octree cells change at twice the frequency of the next level of the hierarchy, which changes at twice the frequency of the next, etc. The required bandwidth is therefore a constant factor of the viewer's movement speed and is independent of the scale and complexity of the world. Streaming local geometry adds a constant overhead.

We stress that this bandwidth cap is only an amortized estimate. Certain planes bound cells from all levels of the hierarchy, so crossing these implies that all such levels must be updated. A good prefetching scheme (Section 6) spreads out the download over time, allowing us to closely approach the amortized estimate.

If the viewer is not near any geometric objects, orthoviews for nearby cells are null and need not be rendered. This allows the maximal movement speed achievable within a certain bandwidth cap to increase exponentially as the viewpoint moves away from detailed information. Also, by modifying a *resolution bias* parameter, users may force undersampling of regions, enabling thinner clients to use fewer orthoviews at the expense of visual quality.

5 Visual Quality

5.1 Server-side antialiasing

We antialias data at every level of the tree. At the base level, geometry is rendered with 2x antialiasing. When compositing (and simultaneously downsampling) a set of child orthoviews to create the parent orthoview, the depth of an output pixel is the closest of the depths of all candidate pixels intersecting its projected area in the child orthoviews; the color and normal are interpolated from all unoccluded candidates. This allows aggregated detail to be filtered up the tree, so that a high-level pixel accurately represents the large scene volume it covers with a proportional number of subsamples. This yields superior visual quality over regular screen-space antialiasing, where a constant number of subsamples are used regardless of how much of the scene the pixel covers (Figure 6).

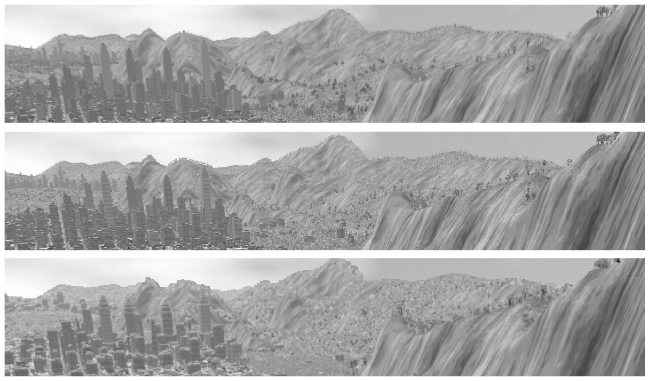


Figure 6: *The benefits of server-side antialiasing. (Top) Geometry rendered with 4x4 antialiasing at 640x480. (Center) Geometry rendered at 2072x1554, with the same antialiasing, and downsampled to 640x480, showing vegetation on distant hills. (Bottom) Our system rendering antialiased orthoviews at 640x480, clearly showing foliage even in the far distance.*

5.2 Client-side splatting

Many authors have addressed the issue of reconstructing and rendering object representations from a point set. One approach is to recover a collection of surface patches [Darsa et al. 1998; Sillion et al. 1997], which are then simplified and rendered as triangle meshes. Another is to splat points individually, using grid spacing and normal information to adjust splat kernels for gap-free coverage of a surface [Botsch et al. 2005]. In our setting, the first approach produces a prohibitively large number of polygons with a consequent performance hit, and the second doesn't deal well with disconnected splats that represent distinct objects or object parts. We use a hybrid approach to point splatting that combines the advantages of both techniques. Our method has the following steps:

1. In a preprocessing pass, compute the connectivity of each orthoview pixel with its eight neighbors.

2a. If a pixel is highly connected, assume it is part of a larger surface, model it as an elliptical disk with the corresponding normal (an approximation to a more refined Gaussian kernel), and project this disk onto the output image plane.

2b. If a pixel is not highly connected, render it as a sphere (an approximation to a symmetric Gaussian).

A comparison of this approach with alternatives is shown in Figure 8. Below we present each of the stages in detail.

Connectivity computation. Two neighboring pixels with image-space depths d_1, d_2, world-space positions $\mathbf{p_1}, \mathbf{p_2}$ and unit normals $\hat{\mathbf{n}}_\mathbf{1}, \hat{\mathbf{n}}_\mathbf{2}$ are considered disconnected from each other if either of the following two conditions fail to hold, for pre-specified $\tau_1, \tau_2 \in \mathbb{R}$ and $\hat{\mathbf{p}} = \frac{\mathbf{p_1}-\mathbf{p_2}}{\|\mathbf{p_1}-\mathbf{p_2}\|}$:

$$\text{Depth: } \|d_1 - d_2\| < \tau_1.$$
$$\text{Normal: } |\hat{\mathbf{n}}_\mathbf{1} \cdot \hat{\mathbf{p}}| < \tau_2 \text{ and } |\hat{\mathbf{n}}_\mathbf{2} \cdot \hat{\mathbf{p}}| < \tau_2.$$

Experiments revealed that these two conditions were not sufficient (Figure 8). We therefore added a third condition, for $\tau_3 \in \mathbb{R}$:

$$\text{Sphericality: } \hat{\mathbf{n}}'_\mathbf{1} \cdot \hat{\mathbf{n}}_\mathbf{2} < \tau_3, \text{ where } \hat{\mathbf{n}}'_\mathbf{1} = \hat{\mathbf{n}}_\mathbf{1} - 2(\hat{\mathbf{n}}_\mathbf{1} \cdot \hat{\mathbf{p}})\hat{\mathbf{p}}.$$

Intuitively, the normals at two points anywhere on a sphere are reflections of each other in the plane orthogonally bisecting the line segment that connects the points. We found this property to be correlated with connectivity in our experiments.

In a second pass, we transitively close the connectivity relation within every 2x2 block of pixels. Thus, within such blocks, any two pixels that were connected by a path are made connected by an edge. In a third pass, if a pixel is connected to more than four neighbors it is labeled *connected*, otherwise it is labeled *disconnected*. This flag is stored in the lowest bits of the depth buffer. Figure 7 illustrates this procedure.

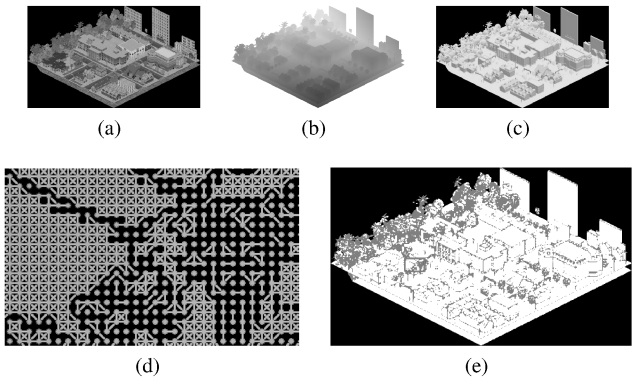


Figure 7: *(a, b, c) Color, depth and normal components of an orthoview. (d) The pairwise connectivity graph of a small section of the orthoview: dots mark original pixel centers and line segments indicate connectivity. (e) The resulting map of* connected *(white) and* disconnected *(grey) pixels, with the section of (d) marked in red. Note the difference between buildings and foliage.*

Connected splatting. To splat connected samples, we use a simplification of the method proposed by Botsch et al. [2005]. The splat is represented by center $\mathbf{c}$ and two tangent-space axes $\mathbf{u}$ and $\mathbf{v}$, given in homogeneous coordinates in the projection space of the orthoview by

$$\mathbf{u} = s_x \times \left(1, 0, -\tfrac{n_x}{n_z}, 0\right) \quad \text{and} \quad \mathbf{v} = s_y \times \left(0, 1, -\tfrac{n_y}{n_z}, 0\right),$$

where $\mathbf{n} = (n_x, n_y, n_z)$ is the sample normal and $s_x \times s_y$ is the size of an orthoview pixel. In the basis with origin $\mathbf{c}$ and axes $\mathbf{u}, \mathbf{v}$, the disk D of diameter $\sqrt{2}$ is chosen to represent the splat.

Let the matrix $\mathbf{M}$ transform points from the orthoview's projection space to screen space. Also, let the operator ρ convert the homogeneous vector (x, y, z, w) to the non-homogeneous equivalent $(x/w, y/w, z/w)$. The projected axes, in non-homogenous coordinates, are given by

$$\begin{aligned}\mathbf{u}' &\equiv (u'_x, u'_y, u'_z) = \rho(\mathbf{M}(\mathbf{c} + \mathbf{u})) - \rho(\mathbf{Mc}),\\ \mathbf{v}' &\equiv (v'_x, v'_y, v'_z) = \rho(\mathbf{M}(\mathbf{c} + \mathbf{v})) - \rho(\mathbf{Mc}).\end{aligned}$$

Assuming the splats are small (a reasonable assumption since we choose to display orthoviews with splats roughly the size of an on-screen pixel), and writing $\mathbf{c}' \equiv (c'_x, c'_y, c'_z, c'_w) = \mathbf{Mc}$, the axes can be approximated as

$$\mathbf{u}' \approx \frac{(\mathbf{Mu})_{xyz}}{c'_w} \quad \text{and} \quad \mathbf{v}' \approx \frac{(\mathbf{Mv})_{xyz}}{c'_w}.$$

The x and y components of $\rho(\mathbf{c}') \pm \frac{1}{2}(\mathbf{u}' \pm \mathbf{v}')$ are the corners of a screen-space parallelogram that approximately bounds the projected splat. We render the bounding square S of this parallelogram, parametrized as $[-0.5, 0.5]^2$. The matrix $\mathbf{T}$, given by

$$\begin{pmatrix} u'_x & v'_x \\ u'_y & v'_y \end{pmatrix},$$

transforms points from splat space to S. We discard all points $\mathbf{p} \in S$ such that $\|\mathbf{T}^{-1}\mathbf{p}\| > 1/\sqrt{2}$, thus approximating the projection of D. Note that the inverse mapping also gives texture coordinates across the splat, allowing per-fragment shading.

Disconnected splatting. For disconnected samples, we precompute the size of the world-space sphere bounding a single pixel rectangle of the orthoview. This size, scaled by the z coordinate of a sample's eye-space position to account for perspective, gives the projected size of the splat, and we simply render a circular splat of this size.

5.3 Popping

When the set of orthoviews rendered by a client changes, the change can be accompanied by a visual "popping" artifact if the new orthoviews appear different from the old ones and these differences are apparent. This can happen when an orthoview is replaced by its children, when a set of orthoviews are replaced by their parent, or when the set of canonical directions closest to the viewing angle changes.

Popping is most severe if the newly needed set of orthoviews is not yet present on the client, due to reduced network performance or excessively high user speeds. If nothing is done to ameliorate this, large parts of the environment will appear empty while new data is streamed in. To address this severe variety of popping we retain orthoviews used in previous frames until new data is received. The splatting algorithm automatically covers undersampled areas with correspondingly enlarged splats, providing a coarse but continuous appearance before more refined orthoviews arrive. However, the visual impact of not having the needed data is still apparent. To minimize the occurrence of such situations, a prefetching scheme is required that aims to anticipate the need for orthoviews ahead of time and stream them onto the client preemptively. This is reported in Section 6.2.

A milder form of popping appears when one set of orthoviews is swapped out for another, with both sets being present client-side. This visual artifact can be eliminated by ensuring that such swaps are only performed when the corresponding splats are sub-pixel. We were interested in pushing the limits of image-based rendering and found that trading off some mild (i.e., perceptible but not bothersome) popping for increased performance was sensible from our standpoint. We found that the visual impact of this artifact can be significantly reduced without compromising performance as follows. We treat the framebuffer from the previous frame, with colour and depth components, as a depth image (for a perspective projection). This depth image is reprojected by the current camera parameters and blended with the current frame. The blending weight is proportional to the time elapsed between the frames. Superimposed pixels are blended if they are within a depth threshold of each other.

This approach has two useful side-effects. First, the temporal smoothing completely eliminates flicker and z-fighting artifacts. Second, we can trivially simulate a variable amount of motion blur, by warping the previous frame to an intermediate projection rather than the current one.

We refrained from blending individual orthoviews, for performance reasons. We expect this compromise to become unnecessary in the face of increased graphics hardware performance.

6 Implementation

The system was implemented with OpenGL on Linux and tested on a variety of server and client machines. A basic HTTP server was set up on each server node. The clients request orthoviews using coordinate-based filenames that map to the appropriate server.

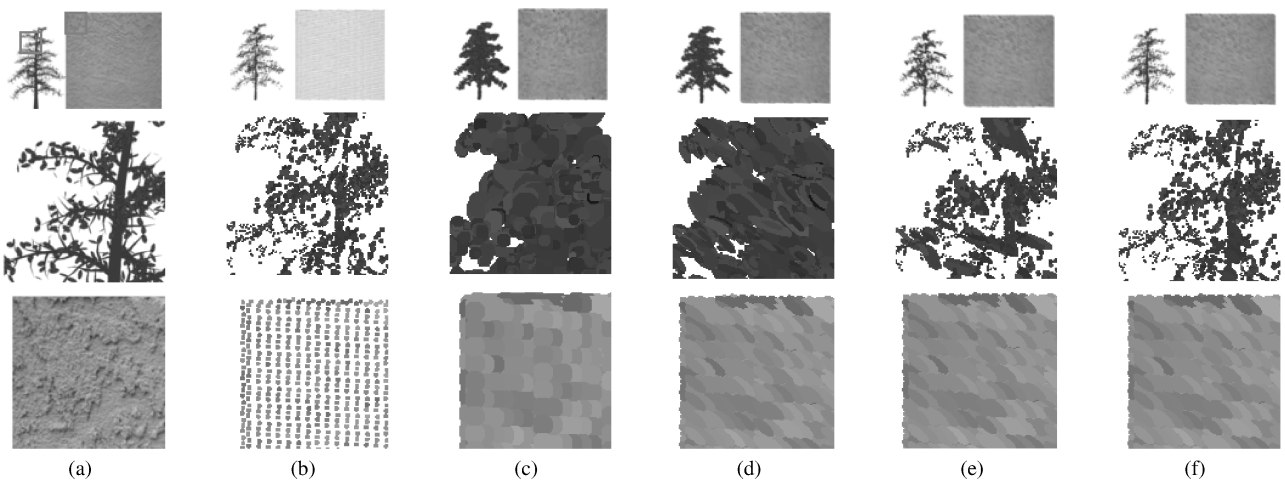


Figure 8: *Comparison of splat shapes. Each panel shows an image of a tree next to a wall, with two square sections from the image enlarged by a factor of 4 and placed below it. (a) Original geometry, rendered at high resolution, with the enlarged sections marked in red. (b–f) Comparison of splatting strategies for orthoviews going diagonally into the page at* $\approx 70^\circ$ *from the perpendicular: (b) a circular kernel bounded by the pixel's area, (c) a circular kernel bounding the pixel's area warped by the normal [Shade et al. 1998], (d) an elliptic kernel bounding the pixel's area warped by the normal [Botsch et al. 2005], (e) our algorithm using a circular or elliptic kernel according to the pixel's connectivity, computed* without *the sphericality criterion, and (f) our algorithm* with *the sphericality criterion. Our algorithm preserves both the appearance of distinct leaves on the tree and the connectivity of the wall.*

6.1 Representation and rendering

Orthoviews are represented as 512x512 24-bit color and normal maps, and 16-bit depth maps. While it is known that normals can be quantized to significantly smaller bit depths with little loss in shading quality, we retained the extra precision for our connectivity computations. For the depth data, we found in practice that 10 bits of precision were sufficient. The remaining 6 bits were used to store per-pixel connectivity data, which was pre-computed on the server.

Orthoviews are rendered by splatting every pixel. A pre-generated vertex buffer, with attached colors and normals, consumed too much GPU memory. Instead, we upload the orthoview's depth, color and normal channels as textures. (Colors and normals are S3-compressed and depth is shrunk to 12-bit.) A single generic vertex buffer is precomputed to index orthoview pixel locations.

Splat shape calculations are performed largely in the vertex shader. If the point has a valid depth value, we examine its connectivity bits. Based on this, we compute the splat square size and the transformation $\mathbf{T}^{-1}$ (see Section 5.2). The splat shading is computed from color, normal and lighting information.

Occlusion culling on orthoviews accelerates rendering, sometimes significantly (Figure 10). Each orthoview is divided into a uniform grid of tiles (4x4 by default). The vertex buffer is correspondingly ordered, so the first segment corresponds to the first tile. A bounding box is precomputed for depth samples in each tile. Before rendering a tile, a hardware occlusion query for its bounding box checks visibility.

We perform full-screen 4x4 antialiasing to reduce noise, improve the rendering of subpixel splats, and accurately warp the previous framebuffer for blending with the current frame (Section 5.3). We found that manually performing FSAA in an offscreen buffer was more effective than the system version.

6.2 Prefetching

For network prefetching, we maintain a number of "look-ahead" positions surrounding the viewpoint. These are placed at a distance equal to the size of the lowest-level octree cell currently in use. We compute the union of the sets of required orthoviews for these positions, as well as for the current viewpoint. This set is ordered first by increasing distance from the current position, and then by level in the octree. This constitutes the prefetching queue. The queue is updated when the viewpoint travels more than a certain distance from the last update location. In our implementation this distance is set to one tenth the size of the lowest-level octree cell currently in use. Background threads process the prefetching queue in order. A local cache holds prefetched orthoviews. An orthoview is downloaded if it is not already in the cache. When an orthoview is needed for the current viewpoint, it is located in the cache and added to the current rendered set. Nearby geometry is prefetched and handled similarly.

Due to transfer latency, we found that it is also crucial to prefetch orthoviews and geometry into GPU memory to avoid jerkiness. GPU prefetching is prioritized similarly to network prefetching, with the data being fetched from the local cache rather than the network. This level of prefetching is handled by the display thread.

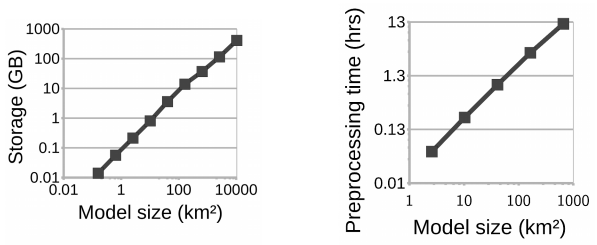
7 Experimental Results

We report experimental results on two scenes that were constructed to reflect the scale and generality of online worlds. The first is a 10,000 km^2, 25-billion polygon landscape with dense vegetation and architecture. The second is a fully three-dimensional scene with 29,264 complex models, each with 97,000 polygons, floating in space. The preprocessing of the first scene was handled in parallel by four workstations. A representative node has a quad core Intel Q6700 CPU and an NVIDIA 8800 GTX GPU. The preprocessing time was 10 hours. The second scene was preprocessed on a single server in 1 hour.

The first scene has 10 octree levels and the aggregate size of all orthoviews is 412 GB . The second scene has 7 octree levels and the size of the preprocessed image-based hierarchy is 1.9 GB. This data compares favorably with the size of the original scenes, represented as geometry: 944 GB for the first scene and 79.5 GB for the second. Figure 9 shows the sizes and preprocessing times for increasingly large subtrees of the hierarchy for the first scene, demonstrating linear scaling.

We tested the handling of changes in the environment, triggering updates of regions of varying sizes in the first scene. These results are reported in Table 1. Changes of even large regions are processed quickly.

The primary test machine for the client was an eight-core workstation with 8GB RAM and an NVIDIA 285 GTX GPU. Tests were performed at a screen resolution of 640x480, with both geometry



(a) (b)

Figure 9: *(a) Storage vs model size. (b) Preprocessing time on a single machine vs model size. Both log-log graphs have slope* ≈ 1 *and show linear relationships.*

Size of modified region	*Update time (seconds)*
200m×200m	20.82
400m×400m	23.95
800m×800m	42.49

Table 1: *Time taken to update the hierarchy of orthoviews for the first scene after a region is modified.*

and orthoviews streamed over a broadband connection. Over the automated test path for the first scene shown in the accompanying video, the graphics card processed at most 25 million points per frame after frustum and occlusion culling (Figure 10).

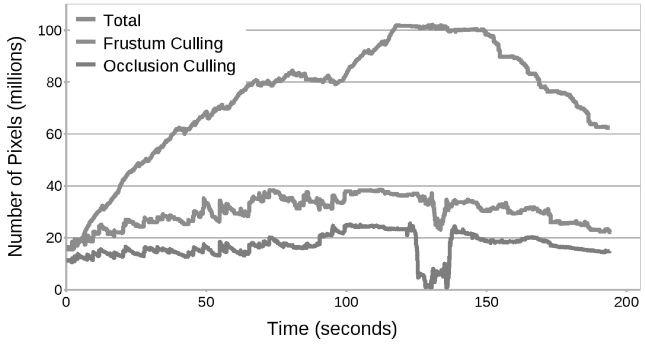


Figure 10: *The total number of orthoview pixels in memory (red), those in orthoviews intersecting the view frustum (green), and those actually processed by the graphics hardware after occlusion culling (blue), over the path shown in the accompanying video for the first scene. The sharp reduction in the number of processed pixels (and the resultant increase in framerate that can be observed in Figure 11(a)) around 120–140s corresponds to city navigation where building facades cause high occlusion, despite the presence of a large number of orthoviews in memory.*

Framerate and bandwidth utilization for paths from both scenes are shown in Figure 11. For comparison, rendering a representative view of the first scene from geometry took 14 minutes on the same machine.

8 Discussion

We investigated the application of hierarchical image-based rendering to massive online environments. Solutions to critical visual quality and performance issues were developed and evaluated. This work demonstrates that hierarchical image-based rendering is a viable approach to handling large-scale heterogenous online worlds.

The chief bottleneck of our approach is the point-rendering performance of graphics hardware. Our approach yields up to 25 million points per frame for our test scenes, stretching current consumer-grade hardware to the limit. For this reason we currently avoid blending between individual orthoviews (which would further eliminate popping artifacts) and variable shading and opacity within individual splats (which would further enhance visual quality). We chose not to use layered depth images for the same reason. Future growth in graphics hardware performance promises to make these improvements feasible within a small number of years.

Not all scenes are as large as ours, which were constructed as a long-term feasibility study. Smaller scenes can already be rendered at higher resolution, higher frame-rates, and with additional visual effects. Furthermore, a crucial advantage of our approach is that the client-side performance requirements grow only logarithmically with the size and complexity of the scene. Thus with every doubling of hardware point-rendering performance, the size and complexity of detailed online worlds that can be explored interactively on commodity machines is raised to the second power.

9 Acknowledgements

Chris Platz created the elevation map used in the primary test scene. The city block models were acquired from TurboSquid.com. This work was supported by NSF grants SES-0835601, CCF-0641402 and CNS-0831163, an Alfred P. Sloan Research Fellowship, and a PACCAR Inc. Stanford Graduate Fellowship.

References

AIREY, J. M., ROHLF, J. H., AND BROOKS, JR., F. P. 1990. Towards image realism with interactive update rates in complex virtual building environments. In *Proc. Symp. Interactive 3D Graphics*, ACM, 41–50.

ALIAGA, D. G., AND LASTRA, A. 1999. Automatic image placement to provide a guaranteed frame rate. In *Proc. SIGGRAPH*, ACM, 307–316.

ALIAGA, D. G., COHEN, J., WILSON, A., BAKER, E., ZHANG, H., ERIKSON, C., HOFF, K., HUDSON, T., STUERZLINGER, W., BASTOS, R., WHITTON, M., BROOKS, F., AND MANOCHA, D. 1999. MMR: An interactive massive model rendering system using geometric and image-based acceleration. In *Proc. Symp. Interactive 3D Graphics*, ACM, 199–206.

BOSSE, M., NEWMAN, P., LEONARD, J., AND TELLER, S. 2004. Simultaneous localization and map building in large-scale cyclic environments using the atlas framework. *Int'l J. Robotics Res. 23*, 12, 1113–1139.

BOTSCH, M., HORNUNG, A., ZWICKER, M., AND KOBBELT, L. 2005. High-quality surface splatting on today's GPUs. In *Proc. Eurographics Symp. on Point-Based Graphics*, 17–24.

CHANG, C.-F., BISHOP, G., AND LASTRA, A. 1999. LDI tree: a hierarchical representation for image-based rendering. In *Proc. SIGGRAPH*, ACM, 291–298.

CIGNONI, P., DI BENEDETTO, M., GANOVELLI, F., GOBBETTI, E., MARTON, F., AND SCOPIGNO, R. 2007. Ray-casted BlockMaps for large urban models streaming and visualization. *Comp. Graphics Forum 26*, 3.

COHEN-OR, D., CHRYSANTHOU, Y., SILVA, C., AND DURAND, F. 2003. A survey of visibility for walkthrough applications. *Trans. Vis. and Comp. Graphics 9*, 3, 412–431.

DARSA, L., SILVA, B. C., AND VARSHNEY, A. 1998. Walkthroughs of complex environments using image-based simplification. *Computers and Graphics 22*, 1, 55–69.

GOOGLE INC. 2008. *Google 3D Warehouse*. http://sketchup.google.com/3dwarehouse.

JESCHKE, S., WIMMER, M., SCHUMANN, H., AND PURGATHOFER, W. 2005. Automatic impostor placement for guaranteed frame rates and low memory requirements. In *Proc. Symp. Interactive 3D Graphics and Games*, ACM, 103–110.

LEYVAND, T., SORKINE, O., AND COHEN-OR, D. 2003. Ray space factorization for from-region visibility. In *Proc. SIGGRAPH*, ACM, 595–604.

LOSASSO, F., AND HOPPE, H. 2004. Geometry clipmaps: terrain rendering using nested regular grids. *ACM Trans. Graphics 23*, 3, 769–776.

LUEBKE, D., WATSON, B., COHEN, J. D., REDDY, M., AND VARSHNEY, A. 2002. *Level of Detail for 3D Graphics*. Elsevier Science Inc.

MANN, Y., AND COHEN-OR, D. 1997. Selective pixel transmission for navigating in remote virtual environments. *Comp. Graphics Forum 16*, 3, C201–C206.

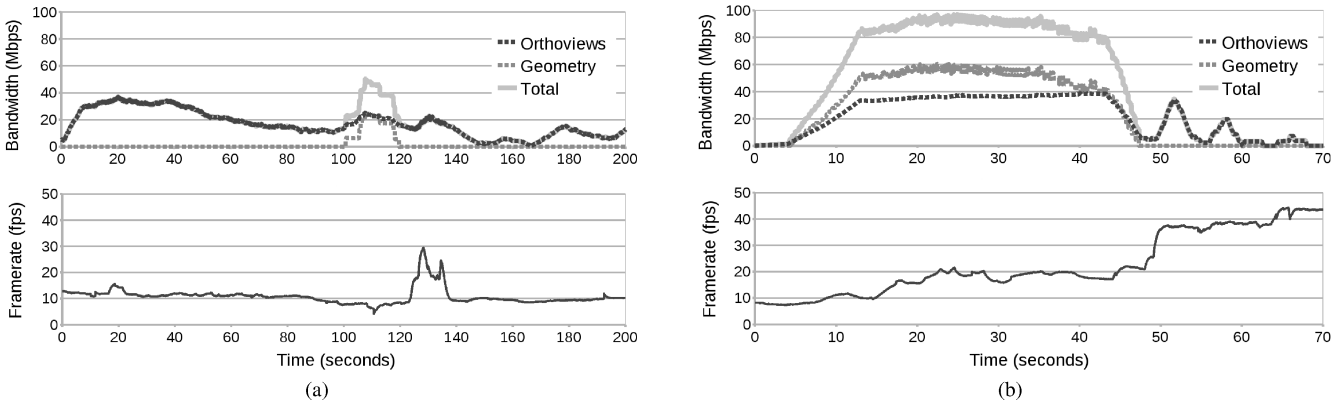


Figure 11: *Network bandwidth and framerate for the client over the automated test runs shown in the video for (a) the first and (b) the second scene.*

MAX, N. L. 1996. Hierarchical rendering of trees from precomputed multi-layer z-buffers. In *Rendering Techniques*, 165–174.

MILLER, G. 2007. The promise of parallel universes. *Science 317*, 1341–1343.

POLIS, M. F., GIFFORD, S. J., AND MCKEOWN JR., D. M. 1995. Automating the construction of large-scale virtual worlds. *Computer 28*, 7, 57–65.

RAFFERTY, M. M., ALIAGA, D. G., POPESCU, V., AND LASTRA, A. A. 1998. Images for accelerating architectural walkthroughs. *IEEE Comp. Graphics Appl. 18*, 6, 38–45.

REGAN, M., AND POSE, R. 1994. Priority rendering with a virtual reality address recalculation pipeline. In *Proc. SIGGRAPH*, ACM, 155–162.

RUSINKIEWICZ, S., AND LEVOY, M. 2000. QSplat: a multiresolution point rendering system for large meshes. In *Proc. SIGGRAPH*, ACM, 343–352.

RUSINKIEWICZ, S., AND LEVOY, M. 2001. Streaming QSplat: a viewer for networked visualization of large, dense models. In *Proc. Symp. Interactive 3D Graphics*, ACM, 63–68.

SCHAUFLER, G., AND STÜRZLINGER, W. 1996. A three dimensional image cache for virtual reality. *Comp. Graphics Forum 15*, 3, 227–236.

SCHMALSTIEG, D., AND GERVAUTZ, M. 1996. Demand-driven geometry transmission for distributed virtual environments. *Comp. Graphics Forum 15*, 3, 421–431.

SCHMALSTIEG, D. 1997. *The Remote Rendering Pipeline*. PhD thesis.

SHADE, J., LISCHINSKI, D., SALESIN, D. H., DEROSE, T., AND SNYDER, J. 1996. Hierarchical image caching for accelerated walkthroughs of complex environments. In *SIGGRAPH '96: Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, ACM, New York, NY, USA, 75–82.

SHADE, J., GORTLER, S., WEI HE, L., AND SZELISKI, R. 1998. Layered depth images. In *Proc. SIGGRAPH*, ACM, 231–242.

SILLION, F., DRETTAKIS, G., AND BODELET, B. 1997. Efficient impostor manipulation for real-time visualization of urban scenery. *Comp. Graphics Forum 16*, 3, C207–C218.

TANNER, C. C., MIGDAL, C. J., AND JONES, M. T. 1998. The clipmap: a virtual mipmap. In *Proc. SIGGRAPH*, ACM, 151–158.

TELER, E., AND LISCHINSKI, D. 2001. Streaming of complex 3D scenes for remote walkthroughs. In *Eurographics*, A. Chalmers and T.-M. Rhyne, Eds., vol. 20(3). Blackwell Publishing, 17–25.

TELLER, S. J., AND SÉQUIN, C. H. 1991. Visibility preprocessing for interactive walkthroughs. *SIGGRAPH Comp. Graphics 25*, 4, 61–70.

THRUN, S., AND MONTEMERLO, M. 2005. The GraphSLAM algorithm with applications to large-scale mapping of urban structures. *Int'l J. Robotics Res. 25*, 5-6, 403–430.

WAND, M., FISCHER, M., PETER, I., AUF DER HEIDE, F. M., AND STRASSER, W. 2001. The randomized z-buffer algorithm: interactive rendering of highly complex scenes. In *Proc. SIGGRAPH*, ACM, 361–370.

WILLIAMS, L. 1983. Pyramidal parametrics. *SIGGRAPH Comput. Graph. 17*, 3, 1–11.

WILSON, A., AND MANOCHA, D. 2003. Simplifying complex environments using incremental textured depth meshes. *ACM Trans. Graphics 22*, 3, 678–688.

YOON, I., AND NEUMANN, U. 2000. Web-based remote rendering with IBRAC. *Comp. Graphics Forum 19*, 3.