Architectural Walkthroughs Using Portal Textures

Daniel G. Aliaga, Anselmo A. Lastra
Department of Computer Science
University of North Carolina at Chapel Hill
{ *aliaga* | *lastra* } *@cs.unc.edu*

ABSTRACT

This paper outlines a method to dynamically replace portals with textures in a cell-partitioned model. The rendering complexity is reduced to the geometry of the current cell thus increasing interactive performance. A portal is a generalization of windows and doors. It connects two adjacent cells (or rooms). Each portal of the current cell that is some distance away from the viewpoint is rendered as a texture. The portal texture (smoothly) returns to geometry when the viewpoint gets close to the portal. This way all portal sequences (not too close to the viewpoint) have a depth complexity of one. The size of each texture and distance at which the transition occurs is configurable for each portal.

Keywords: visibility culling, cells, portals, textures, sample points, morphing.

1. Introduction

The visualization of architectural spaces (buildings, ships or similar structures) requires large complex models with many geometric primitives. This type of model, however, has a property that can be exploited in order to reduce the rendering complexity: the space is typically divided by walls that occlude everything on the other side. Adjacent areas are only visible through certain openings (doors, windows, etc.). Past research has focused on dividing a model into cells (rooms or predetermined subsections of the model) and portals (doors, windows and other openings). Visibility culling algorithms are used to determine which other cells are visible from a particular viewpoint and view direction. Rendering is thus reduced to the geometry of the visible cells. Exact pre-processing algorithms [Airey90, Teller91] as well as conservative run-time algorithms [Luebke95] have been developed.

In this paper, we further simplify the rendering by conditionally replacing the cells visible through a portal with a texture. Consequently, the system only needs to render the geometry of the cell containing the viewpoint and a few textured mapped polygons. Furthermore, this approach alleviates the sudden decreases in performance when a complex cluster of cells becomes visible. If the viewpoint approaches a portal, the *portal texture* will (smoothly) return to geometry, allowing the viewpoint to move into the adjacent cell.

This technique is a specialization of the general use of *impostors* introduced by [Maciel95]. Portions of a static 3D model can be automatically or manually replaced with representations that are faster to render, namely 2D textures. The textures display imagery that is an approximate representation of the underlying geometry but the costs of rendering are independent of the model complexity. We can control the accuracy of the representation by regulating the number of textures. This provides us with a mechanism to control the quality of the images we are seeing, at the expense of texture memory and perhaps of swapping to the texture store of the graphics accelerator. Solving the general problem of deciding where to place textures in order to improve rendering performance is difficult [Shade96, Schaufler96, Aliaga96]. The cells and portals framework allows us to formulate a set of concrete and efficient algorithms for replacing geometry with textures.

In the following section (Section 2), we present the overall problem of dynamically replacing portals with textures and discuss various strategies. In Section 3, we describe our algorithm for (smoothly) replacing the cells visible through a portal with textures. Section 4 briefly describes our implementation, while Section 5 presents the results we have obtained. Finally, we end with future work (Section 6) and some conclusions (Sections 7).

2. Replacing Portals with Textures

In this section, we review the technique of culling to a portal and describe the notion of portal textures. We examine the possible ways of selecting the viewpoints for the portal textures, and the possible strategies for rendering the textures. Finally, we describe a method for smoothing the transition from portal texture to geometry (and vice versa).

2.1 Portal Culling and Portal Texture Culling

Based on the location of walls and other opaque surfaces, a model can be partitioned into cells [Airey90, Teller91]. Each cell contains a list of portals, each of which defines an opening through which an adjacent cell may be seen. Figure 1 shows the top view of a cell-partitioned model. The viewpoint is inside the *view* cell. Since the view frustum only intersects a subset of the portals of the view cell, the cells attached to each visible portal are recursively traversed to compute all of the visible cells.

Since the model contains the location of all portals, we can compute textures to be placed in the location of the

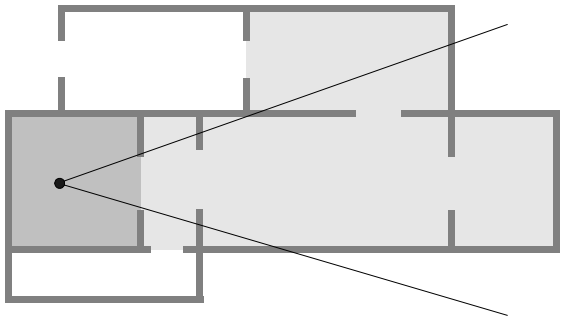


Figure 1: Portal Culling - The cell shown in dark gray is the view cell. Those shown in lighter gray are potentially visible and must be rendered.

otherwise transparent portal openings. At run-time, we render the view cell normally. All visible portals of the view cell are rendered as textures and no adjacent cells are actually rendered, despite being visible. Figure 2 illustrates the reduced set of cells that need to be rendered. As the viewpoint approaches a portal, we switch to rendering the geometry of the cell behind the portal. Once the viewpoint enters the adjacent cell, it becomes the view cell and the previous cell will now be rendered as a portal texture.

2.2 Texture Selection Strategies

A portal can be viewed from multiple view directions as well as multiple viewpoints. Since a single texture only

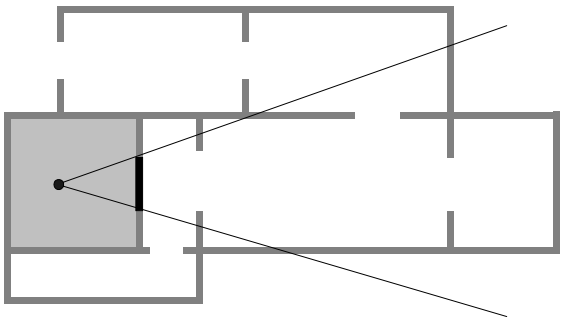


Figure 2: Portal Texture - only the view cell needs to be rendered. The portal (shown in black) is rendered as a single textured polygon.

produces a perspectively correct image from one viewpoint, we need to do some additional work to improve image quality when using portal textures. There are two main approaches to this problem: using image warping, or using multiple textures. The former corresponds closely to image-based rendering [Chen93] and plenoptic modeling [McMillan95, Mark97]. This approach uses depth information at each pixel to warp an image to a new viewpoint. Algorithms to re-project the images and resolve changes in the visibility of the pixels are the subject of current research. This method also does not take advantage of standard rendering hardware, thus we chose to sample the geometry behind a portal from multiple viewpoints. In the models that we have encountered, we have found the use of multiple portal textures to produce decent overall image quality. We sacrifice a controlled amount of image quality for performance.

There are various possible methods for selecting the portal texture viewpoints. We have classified these methods into three categories:

- Model independent viewpoints: define a regularly spaced set of viewpoints spanning the space on the front side of a portal (Figure 3a) without regard to particular model characteristics.
- Model dependent viewpoints: define a subset of viewpoints that do not necessarily span the entire front side of a portal. This approach requires some knowledge about model characteristics, such as the typical portal viewing directions. For example, consider a hallway with a portal to a connecting room. The portal will typically only be viewed from acute angles. By the time the viewpoint is in front of the portal, the portal will need to be rendered using geometry (Figure 3b).

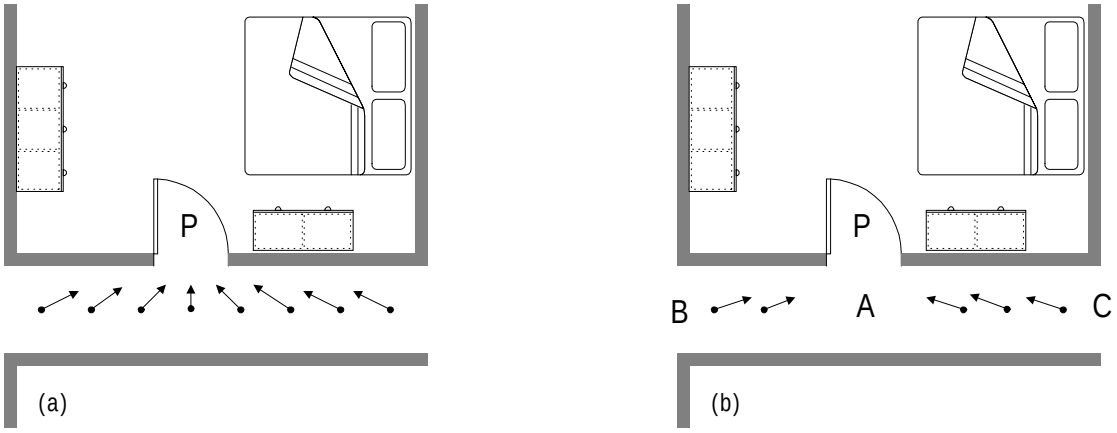


Figure 3: (a) Model-independent and (b) model-dependent viewpoints for portal **P**. In this case, since we expect to represent the portal as geometry by the time we arrive at location **A**, it is advantageous to only render portal textures from the most likely view locations in area **B** and **C**.

- Single viewpoint: a single viewpoint, usually facing the portal. This method is economical and works well when the possible or likely view directions to the portal are restricted.

In general, model-dependent viewpoints produce better results than model-independent viewpoints because they take advantage of the user's domain knowledge of the model to reduce the number of textures.

At run-time, we choose the portal texture that most closely represents the view of the geometry behind the portal from the current viewpoint. As the viewpoint moves, we continuously switch to the best texture. This generates a visual effect commonly known as *popping*. By increasing the number of textures, we can control the extent of the popping.

We have experimented with blending portal textures but have not found the results to be visually pleasing.

2.3 Texture Creation Strategies

We've discussed strategies for selecting portal texture viewpoints. Another question is *when* to render the textures. If we decide to use a small number of portal textures, it's probably best to render all of them at start up. However, if we prefer to use many textures in order to improve image quality, we may not be able to render them all at start up. The next simplest strategy is to render portal textures as they are needed, and cache them for reuse.

Rendering portal textures on demand works quite well for the common application of architectural walkthroughs. In practice, users don't fly quickly through the model. They usually go to a room and examine an area in detail before proceeding to another portion of the model. Demand rendering of textures will result in slower performance when the user first enters a cell. However, as the user works in an area, that area will "sweeten" and performance will increase. This is analogous to the use of a cache to take advantage of locality.

2.4 Smooth Transition Strategies

The single portal-texture case, when only one texture is used to represent the portal from all directions, is very interesting because of its low cost. Unfortunately, this case may result in a very noticeable transition from texture to geometry (or vice versa). This is one of the worst examples of popping. We can eliminate this abrupt transition by smoothly warping the geometry represented by the portal texture from its current incorrectly projected position to its correctly projected position (or vice versa). This *morphing* strategy [Aliaga96] is very effective and may be accomplished at very low cost (the mathematics of the warp will be explained in Section 3.3). This warp can be efficiently implemented using the graphics hardware matrix stack.

We can also use the morphing strategy to ease the transition to or from geometry even when using multiple portal textures. Although it is less important in this case, since the computational cost is minimal, it is worthwhile.

3. Portal Textures

In this section, we describe the algorithms we chose for the portal textures system. First, we detail our texture viewpoint selection strategy. Afterwards, we explain our morphing algorithm more precisely.

3.1 Overall Algorithm

Our system replaces geometry behind portals with portal textures sampled from a constrained set of model

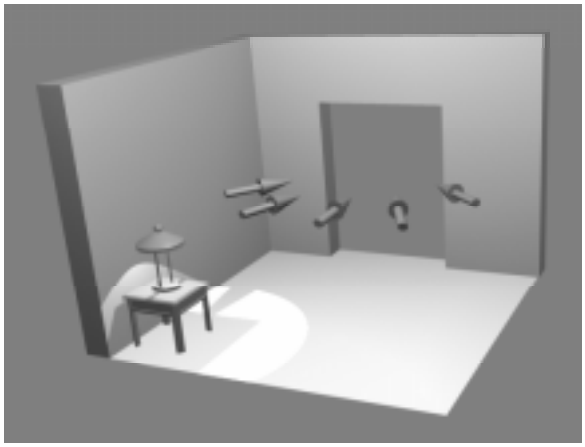


Figure 4: Constrained portal texture viewpoints lying on a semicircle in front of the portal at eye height.

dependent viewpoints. We smoothly change from rendering the portal texture to geometry (and vice versa) by warping the geometry from its projected position (on the portal texture) to the correct projection for the current viewpoint. The geometric warping is particularly useful if we wish to reduce overall texture memory use by using only a single texture per portal.

3.2 Constrained Model Dependent Sampling

When visualizing architectural models, we typically walk at about the same height (although we perhaps change "floors"). Without loss of generality, we assume that our head movement is typically left/right and forward/backward. We may also gaze up or down at any time. This reduces the number of necessary textures. For each portal, we allow the modeler to define a set of viewpoints constrained to lie on a semicircle of some radius on the front-side of the portal. We assume that portals are typically perpendicular to the "floor" of the model and thus fix the semicircle to lie at some typical viewing height for each portal (Figure 4).

For each portal of the model, we need to define (or assume reasonable default) values for the following parameters:

(a) Viewing Height: the typical viewing height of the portal.

(b) Sampling Distance: the radius of the constraining semicircle from the portal.

(c) Transition Distance: the distance from the portal at which to perform a portal texture to geometry transition (or vice versa).

(d) Viewing Angles: the set of points on the semicircle to use as texture viewpoints. We have found the use of multiple disjoint angular spans of equally spaced viewpoints to yield good results.

3.3 Morphing

In order to perform a smooth portal texture to geometry transition (or vice versa), we need to smoothly re-project the geometry behind the portal from its projected position on the texture to its correct position [Aliaga96]. This warp corresponds to an inferred perspective warp [Wolberg90]. In addition, the transformation must be set up so that z-buffering works properly.

Setup

A portal texture corresponds to an image of the model seen through a portal, defined by four vertices $\mathbf{v}_0$-$\mathbf{v}_3$ and a sample point $\mathbf{p}_a$. We denote this viewing frustum projection by [$\mathbf{v}_0$-$\mathbf{v}_3$, $\mathbf{p}_a$]. In general, when we wish to return the cells behind the portal to geometry, our current viewpoint will be at some point $\mathbf{p}_b$. We need to smoothly re-project the geometry over the next several (e.g., five) frames, from the projection [$\mathbf{v}_0$-$\mathbf{v}_3$, $\mathbf{p}_a$] to the projection [$\mathbf{v}_0$-$\mathbf{v}_3$, $\mathbf{p}_b$]. This implies that, despite having our eye at point $\mathbf{p}_b$, we need to project the geometry onto the portal plane as if we were at some position between $\mathbf{p}_a$ and $\mathbf{p}_b$. Thus, we re-project the geometry using the interpolated view frustum [$\mathbf{v}_0$-$\mathbf{v}_3$, $\mathbf{p}_i$] where $\mathbf{p}_i$ is a point along the line segment $\mathbf{p}_a$-$\mathbf{p}_b$. Then, we use an inferred perspective transformation to warp the projection *plane*, defined by frustum [$\mathbf{v}_0$-$\mathbf{v}_3$, $\mathbf{p}_i$], to appear as if it were seen from the current viewpoint $\mathbf{p}_b$.

The current frustum, [$\mathbf{v}_0$-$\mathbf{v}_3$, $\mathbf{p}_b$], can be expressed using a model-space transformation $\mathbf{M}_b$ and a projection $\mathbf{P}_b$. Similarly, the interpolated frustum can be defined by a model-space transformation $\mathbf{M}_i$. and a projection $\mathbf{P}_i$. The final (warped) frustum is defined by $\mathbf{M}_w = \mathbf{P}_i\mathbf{M}_i$. and $\mathbf{P}_w = \mathbf{W}_{ib}$, where $\mathbf{W}_{ib}$ is the perspective warp from $\mathbf{p}_i$ to $\mathbf{p}_b$. This sequence of transformations is illustrated in Figure 5.

To construct the warp matrix $\mathbf{W}_{ib}$, we employ a four-corner mapping (assuming planar quadrilaterals). We project the vertices $\mathbf{v}_0$-$\mathbf{v}_3$ using $\mathbf{P}_i\mathbf{M}_i$ and $\mathbf{P}_b\mathbf{M}_b$ and use their projected positions to construct the four corner mapping.

Proper Occlusion

In order to resolve occlusion properly, we must set up the matrix $\mathbf{W}_{ib}$ so that the final transformation matrix will produce z-values that correspond to the projection onto [$\mathbf{v}_0$-$\mathbf{v}_3$, $\mathbf{p}_i$]. In essence, we wish to transform the x and y coordinates and pass the original projected z-value through the warp unaffected (at least until the homogeneous divide). We can accomplish this by placing the nine coefficients of the warp matrix as follows:

$$\begin{bmatrix} a & b & 0 & c \\ d & e & 0 & f \\ 0 & 0 & 1 & 0 \\ g & h & 0 & i \end{bmatrix}$$

4. Implementation

We implemented the portal-textures system on a Silicon Graphics Onyx (250 MHz R4400, 2GB main memory) with Infinite Reality graphics (containing 64MB of texture memory) and on an Indigo2 (250 MHz R4400, 128MB memory) with Max Impact graphics (and 4MB of texture memory). The system is coded in C++, uses the OpenGL graphics library, and employs a user-configurable amount of host memory and texture memory.

At run-time, the system renders the portal textures to host memory, and loads the textures into the texture memory of the graphics accelerator as needed (using the texture binding and copy commands of OpenGL). For simplicity, all textures are 256x256 pixels in size and 8 bits per color component. If there is no free space in the texture memory, we replace older portal textures that are no longer in view. To decide which textures in accelerator memory to replace, we use a simple working-set algorithm. Portal textures are usually computed on demand, although they could instead be computed at start up.

The contents of each cell are maintained as a collection of geometric primitives organized in a spatial partitioning tree (an octree). When a cell is flagged as visible, its contents are culled to the current frustum and rendered. Portals are culled to the current frustum using their screen-space bounding rectangle.

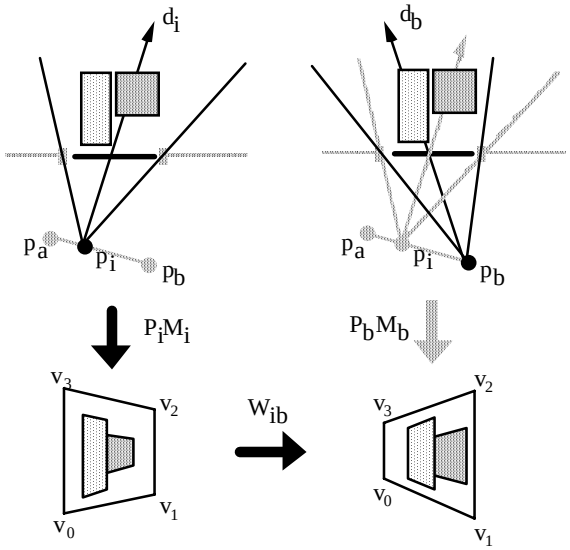


Figure 5: Sequence of transformations for morphing geometry. We first project along $\mathbf{d}_i$ onto the plane of the portal ($\mathbf{P}_i\mathbf{M}_i$). We then re-project from $\mathbf{p}_i$ to $\mathbf{p}_b$ ($\mathbf{W}_{ib}$).

The overall visibility-determination algorithm is summarized by the following pseudo-code (the top-level function is assumed to be initially called with the view cell and the view frustum):

```
Visibility(cell, frustum) {
  Mark cell visible
  Cull cell to frustum
  Foreach portal {
     Cull portal to frustum
     if (portal is visible) {
        if (portal in transition)
           Initialize transition
        else if (portal is texture)
           Choose best texture sample
        else if (portal is geometry)
           Visibility(portal's adjacent cell,
                      culled frustum)
     }

     if (portal in transition) {
        Next transition step
        if (portal->texture finished)
           Choose best texture sample
        if (texture->portal finished)
           Visibility(portal's adjacent cell,
                      culled frustum)
     }
  }
}
```

5. Results

5.1 Performance

We tested our system using two architectural models. The first model, named *Brooks House* (Color plates 1-5), is that of a large one-story house modeled using 528,000 polygons. The second model, the *Haunted House* (Color plates 6-7), is of a two-story house and consists of 214,000 polygons. Both of these models have been divided into cells and portals. The more complex Brooks House has 19 cells and 52 portals, while the Haunted House has 7 cells and 12 portals.

We traversed a path through each model and recorded the number of primitives rendered per frame as well as the overall frame time (Figures 6 through 9) using (a) view-frustum culling [Clark76], (b) portal culling, and (c) portal-texture culling. For these experiments, we created portal textures for every degree, over viewing directions ranging from 30 to 120 degrees in front of the portals, and pre-computed the textures (the next section has some observations about the cost of portal texture rendering).

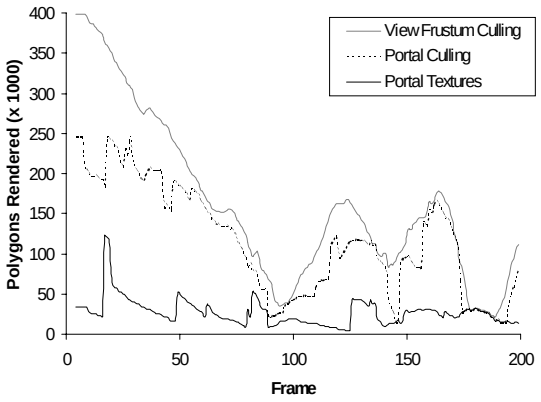


Figure 6: Number of polygons rendered for the Brooks House model using traditional view-frustum culling, portal culling, and the portal textures system.

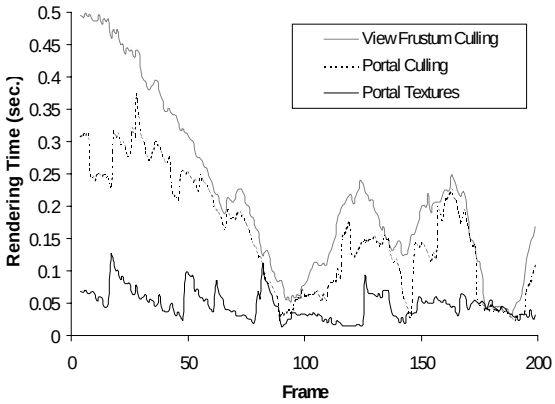


Figure 7: Frame times of the same path on the Brooks House model using view-frustum culling, portal culling, and the portal textures system.

The Brooks House model was rendered on an Onyx with Infinite Reality graphics hardware. Figures 6 and 7 show the number of polygons rendered and frame time, respectively. We achieved an overall speedup of 3.3 with portal textures vs. portal culling and of 4.6 vs. view-frustum culling. The Haunted House model was rendered on an Indigo2 with Max Impact graphics hardware. The results are plotted in Figures 8 and 9. Speedups were 2.6 for portal textures over portal culling and 4.3 for portal textures over view-frustum culling. Typically, one to three portal textures are rendered in each frame.

Notice how the large variations in rendering performance have been significantly reduced. This is due primarily to the fact that a texture can be rendered in time independent of the complexity of the geometry it represents. However, computing textures on demand may also introduce some longer frame times.

5.2 Portal Texture Creation

What if we decide to compute portal textures on demand? Figure 10 shows frame rate with textures always computed on demand (a "cold cache") and sampling ranges of one and ten degrees per portal texture. We used the Brooks House model and the same path used for the results described in the previous section. Here the overall speedups are 1.8 for the finely sampled case and 2.8 for the coarsely sampled rendering.

Although the performance was still quite good, we have lost the very steady frame rate that we achieved with the pre-computed portal textures. In fact, we sometimes peaked above the portal-culling case because some scenes in the model forced us to render textures for several rooms in one frame (a room visible in a doorway that was visible in another doorway, etc). However, in general we expect that our performance will be bounded by the

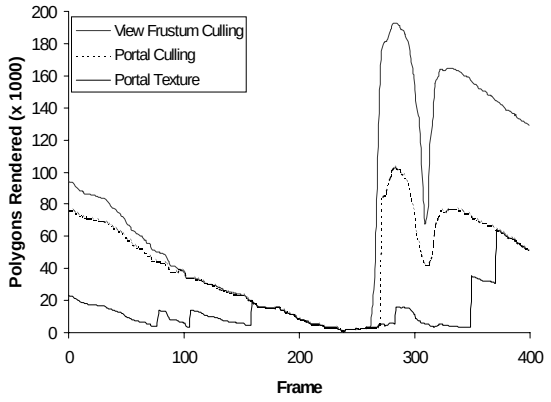


Figure 8: Number of polygons rendered for the Haunted House model using view-frustum culling, portal culling, and the portal textures system.

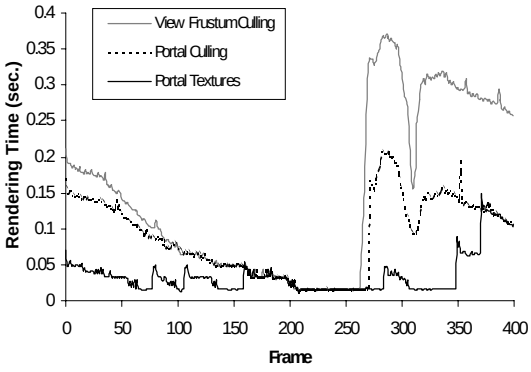


Figure 9: Frame times of the Haunted House model on an Indigo2 with Max Impact graphics accelerator.

geometry rendered using only portal culling, plus some extra overhead.

Computing portal textures on demand is a good strategy if the typical use of the walkthrough program is to examine areas of the model in detail. Although the frame rate may be low when we first enter a room, it will get better as we stay in that room.

5.3 The Single Portal-Texture Case

The use of a single portal texture per portal gives us the best and steadiest performance, albeit with lowest image fidelity. However, we have observed that with morphing at transitions, the user feels very comfortable interacting with the model. Since it is trivial to pre-compute all of the portal textures when loading a model, variations in frame rate are small. This is the best choice for a tight rendering budget.

5.4 Image Quality

Increased performance is achieved at the expense of some image quality. In general, a greater number of textures, together with properly configured portal parameters and the use of morphing, will improve the image quality. In our video, we show the image quality produced by varying the number of textures. One texture per degree gives excellent quality but may be too expensive for some applications.

The amount of texture memory on the graphics accelerator has not proven to be a problem since we only needed one to three textures per frame, and the cost of texture replacement is quite low. We have timed the texture-paging rate of our SGIs. On our Onyx/IR, we can page a 256x256 texture from host to texture memory in ~1.8 milliseconds. Our Indigo2/Max Impact pages the same size texture in ~2.2 milliseconds. Thus our texture memory and texture paging requirements are well within reason.

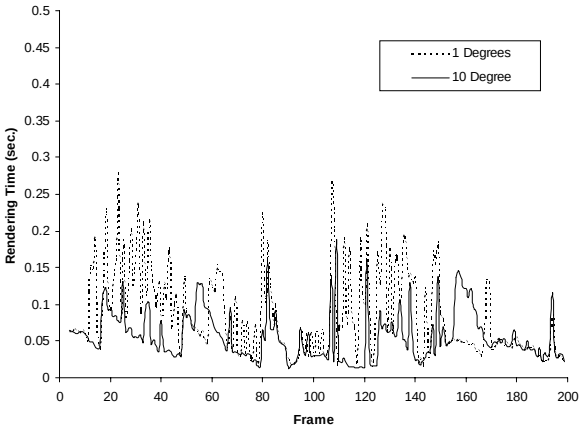


Figure 10: Cold-cache frame times for the Brooks House model using one portal texture sample for every 10 degrees vs. one portal texture per degree.

6. Future Work

The extensions of most immediate benefit to this system are those that might automatically compute the best portal viewpoints and view directions to use for creating the textures. Perhaps through the use of exact cell and portal visibility calculations [Airey90, Teller91], we could locate areas of the model from which each portal is visible and sample only from those areas.

Once we gather more experience with portal textures, we may decide to reduce the constraint that texture viewpoints must lie on a single semicircle. Perhaps modelers will wish to have more freedom to place texture viewpoint locations. An interactive portal-texture placement program would be useful.

At the moment, we are not taking advantage of idle time to render portal textures that may be needed in the future. We could enhance the system to perform incremental rendering in one of several ways. The easiest would be to pre-render the portal textures nearest to our current viewpoint whenever the graphics system is idle (e.g., when the user is not moving). An approach that would give better performance even when moving, is to assign a portion of each frame time to the rendering of future portal textures. We could perhaps use a simple prediction of the next several viewpoints in order to determine the best set of textures to pre-render.

The portal-textures approach is best suited to diffuse environments. This has not been a problem in practice because models for architectural walkthroughs are typically pre-illuminated using radiosity methods. However, if we wish to add specular effects, we may be able to store additional parameters, such as surface normals, at each texel and defer shading [Lastra95].

Finally, we could look to plenoptic warping [McMillan95] in an effort to eliminate popping while also reducing texture memory usage. To make this practical would require specialized hardware.

7. Conclusions

We have demonstrated the benefits of using portal textures. This approach allows us to reduce the rendering complexity principally to that of the cell containing the viewpoint. The adjacent cells, visible through the portals, are represented by textures. For models where the use of portals is advantageous, portal textures increase performance by a constant factor, based on the number of frames a texture can be reused. Improvement is especially good when long sequences of portals are present.

We have shown two cases, one where a number of portal textures are stored (in host or texture memory) for each portal and another where only a single portal texture is stored. Morphing can be used to eliminate the visual popping effect at portal texture to geometry transitions. Using multiple portal textures can trade off quality for speed. The single portal-texture case is especially interesting because (using trivial pre-computation) it dampens the fluctuations in frame time that occur when several rooms become suddenly visible through doorways.

8. Acknowledgments

We are greatly indebted to Dave Luebke for many fruitful discussions about the portals system. We would also like to thank the Ultra64/Nintendo64 SGI Team for the opportunity to try out portal textures on their hardware. Furthermore, we would like to thank Leonard McMillan for his insights regarding re-projections. In addition, we are also grateful to the many late night inhabitants of the graphics lab.

The Brooks House model is courtesy of many generations of students and members of the UNC Walkthrough team. The Haunted House model was created by Dave Luebke and Mike Goslin.

This research was supported in part by the Defense Advanced Research Projects Agency, ISTO Order No. A410 and DABT63-93-C-0048 ("Enabling Technologies and Application Demonstrations for Synthetic Environments"), the National Science Foundation, Grant No. MIP-9306208, and a University of North Carolina Dissertation Fellowship.

References

[Airey90] John Airey, John Rohlf and Frederick Brooks. “Towards Image Realism with Interactive Update Rates in Complex Virtual Building Environments”, *Symp. on Interactive 3D Graphics*, 41-50, 1990.

[Aliaga96] Daniel G. Aliaga. "Visualization of Complex Models Using Dynamic Texture-Based Simplification", *IEEE Visualization ‘96*, 101-106, 1996.

[Chen93] Shenchang Eric Chen and Lance Williams, "View Interpolation for Image Synthesis", *SIGGRAPH 93*, 279-288, 1993.

[Clark76] James Clark, "Hierarchical Geometric Models for Visible Surface Algorithms", *CACM*, **19** (10), 547 - 554, October 1976.

[Lastra95] Anselmo Lastra, Steven Molnar, Marc Olano and Yulan Wang, "Real-Time Programmable Shading", *Proc. Symp. on Interactive 3D Graphics*, 59-66, 1995.

[Luebke95] David Luebke and Chris Georges, "Portals and Mirrors: Simple, Fast Evaluation of Potentially Visible Sets", *Proc. Symp. on Interactive 3D Graphics*, 105-106, 1995.

[Maciel95] Paulo Maciel and Peter Shirley, “Visual Navigation of Large Environments Using Textured Clusters”, *Symp. on Interactive 3D Graphics*, pp. 95-102, 1995.

[McMillan95] Leonard McMillan and Gary Bishop, "Plenoptic Modeling: An Image-Based Rendering System", *SIGGRAPH 95*, 39-46, 1995.

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

[Schaufler96] Gernot Schaufler and Wolfgang Sturtzlinger, "A Three Dimensional Image Cache for Virtual Reality", *Eurographics '96*, 227-235, 1996.

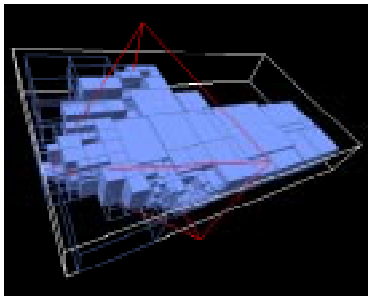
[Shade96] Jonathan Shade, Dani Lischinski, David H. Salesin, Tony DeRose and John Snyder, "Hierarchical Image Caching for Accelerated Walkthroughs of Complex Environments", *SIGGRAPH 96*, 75-82, 1996.

[Teller91] Seth Teller and Carlo H. Séquin, “Visibility Preprocessing For Interactive Walkthroughs”, *SIGGRAPH 91*, 61-69, 1991.

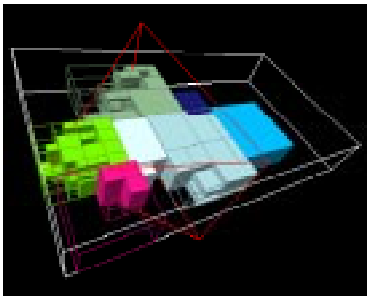
[Wolberg90] George Wolberg, **Digital Image Warping**, IEEE Computer Society Press, Los Alamitos, California, 1990.

Color Figure 1: View of Brooks House. The three portals have been replaced with textures.

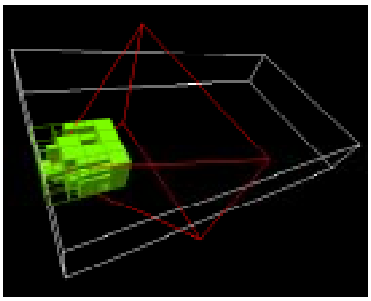
Color Figure 2: Wireframe of Brooks House. The three portal textures are outlined in white.



Color Figure 3: View-frustum culling. Blue boxes represent rendered octtree nodes.

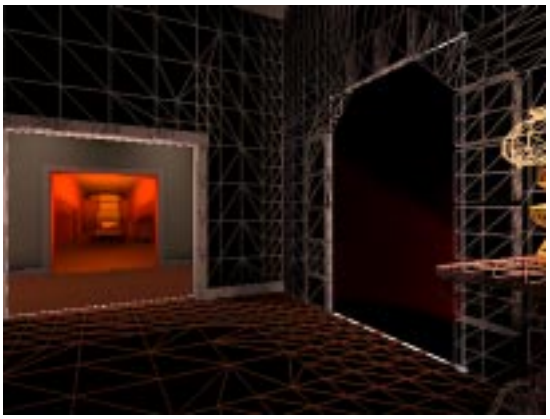


Color Figure 4: Portal culling. Each cell's rendered octtree nodes are shown in a different color.



Color Figure 5: Portal Textures. Only the cell containing the viewpoint needs to be rendered.

Color Figure 6: View of Haunted House. Two portals have been replaced with textures.



Color Figure 7: Wireframe view of Haunted House. Two portal textures are clearly visible (right portal is a dark stairway).