A Fragment Culling Technique for Rendering Arbitrary Portals

Nick Lowe and Amitava Datta

School of Computer Science & Software Engineering
University of Western Australia
Perth, WA 6009
Australia
`{nickl,datta}@csse.uwa.edu.au`

Abstract. Portal-based rendering traditionally describes techniques that involve rendering scenes that have been partitioned into cells connected by portals. The partition information is exploited to determine visibility information. Recently, portal-based rendering has also been used to describe scenes composed from cells and transformative portals. Interesting scenes can be composed by using cell topologies that would never emerge from scene partitioning. Although some constraints have been removed to allow for scene composition, many still exist. The surfaces of portals are necessarily planar and convex polygons, usually with a low maximum number of vertices. These constraints are imposed to simplify clipping that would otherwise become geometrically complex. In this paper, we analyze a technique to simulate complex geometric clipping using fragment culling and integrate this into an algorithm to render arbitrary portals. Finally we provide some examples of interesting portal-based environments that are enabled by our algorithm.

1 Introduction

Portal rendering is a widely used method for rendering indoor scenes in computer graphics. One of the critical issues in rendering indoor architectural scenes is the determination of visible regions. In portal rendering, an indoor scene is usually decomposed into cells and each cell can have two different kinds of surfaces, opaque walls or transparent portals. If the current viewpoint is within a particular cell, an opaque wall obstructs visibility. On the other hand, a portal is a window to the other cells and objects in other cells are visible through a portal.

More recently, portal rendering has been used to construct scenes from independent cells by using transformative portals. Although portal rendering has evolved into a more general paradigm for describing connected spaces, the portals themselves are still kept rather simplistic. In all prior systems, portals are necessarily planar and convex polygons with a small maximum number of vertices.

Jones [7] pioneered portal-based rendering when he utilized scene partitioning to facilitate visibility determination in architectural scenes. In a pre-processing stage, Jones manually decomposed indoor scenes into convex polyhedral cells separated by convex polygonal portals. He then used cell adjacency information and portal visibilities to

Fig. 1. This figure illustrates a real-time digital art application that utilizes our arbitrary portal rendering algorithm. In the top right corner of the image there is a floating portal that leads to another world. Its surface is non-planar and geometrically complex, and it does not exist on a cell boundary. These characteristics make it very difficult to represent in a conventional portal system.

quickly determine visible polygons at runtime. Since Jones' technique involves complete decomposition it can become unwieldy for scenes with a large number of polygons.

Subsequent work by Teller [14] automated the process of complete visibility determination with an intensive pre-processing step that enabled conservative PVS (potentially visible set) calculation for static scenes. More recent work by Luebke and Georges [10] focused on runtime PVS determination without preprocessing. This allowed them to apply portal-rendering techniques to dynamic scenes. Whereas previous work had focused purely on efficient visibility determination, Luebke and Georges' work demonstrated some more interesting uses for portal-rendering. Teller had suggested that portals may be used to model mirrors by allowing an adjacent cell to be the source cell reflected about the portal plane. Luebke and Georges implemented some basic mirror portals, but stopped short of the "dynamic, fully recursive case". See also the paper by Aliaga and Lastra [2] for using textures on portal surfaces instead of rendering the parts visible through the portal.

Tyberghein's open source game engine, Crystal Space [5], utilizes a portal system that is perhaps closer to the dynamic, fully recursive case. Tyberghein models mirror portals as a subset of general transformative portals. Each portal contains a transformation matrix that is applied to its destination cell contents before they are rendered. This allows for mirror portals as well as other interesting effects. He also enables portals that exist within cells (rather than on cell boundaries) by clipping destination cell contents to the portal plane. Application of a dynamic transformative portal system to the digital arts was

investigated in recent work by Grant and Lowe [13]. Figure 1 is a screenshot of their *Shaman* demo.

In this paper, we investigate a technique to enable arbitrary portal surfaces, in order to develop an elegant system that is useful for real-time applications. Our main motivation is to allow portals to be placed anywhere in a cell, rather than only on the cell boundary. Moreover, we allow the portal surface to be general and not necessarily of convex polygonal shape. The main problem in allowing arbitrary surfaces as portals is the clipping of destination cell objects on the portal surface. We propose a fragment culling technique to enable this. Our approach is based on the current trend in computer graphics research to use powerful graphics cards for doing complex operation in hardware, rather than in software. We use the NVIDIA GeForce 4 series cards for our implementation.

The rest of the paper is organized as follows. We discuss the relative advantages and disadvantages of restricted and arbitrary portals in Section 2. Our method for rendering arbitrary portals is discussed in Section 3. We discuss our implementation in Section 4. Finally, we conclude with some discussions in Section 5.

2 Background

We discuss conventional and arbitrary portals and their relative advantages and disadvantages in this section.

2.1 Conventional Portals

Portals are commonly designed with many characteristics that can be exploited for fast clipping and subsequent efficient rendering. These constraints reduce the complexity of clipping operations by simplifying the input domain. The most common constraints are the following:

1. Portal surfaces are planar and convex polygons with a low maximum number of vertices.
2. Portals exist on cell boundaries.

The contents of an adjacent cell need to be clipped to the view frustum generated by the viewpoint and the portal surface. Convex planar polygons generate convex hull frusta that are simple to clip to. The complexity of clipping scales linearly with the number of vertices. Hence, the number of vertices is kept low. Also, when portals exist on cell boundaries, all adjacent cell data is known to exist on the opposite side of the portal surface. Therefore, adjacent cell data does not need to be clipped to the portal surface.

2.2 Arbitrary Portals

Although portals satisfying the above constraints are very useful for efficiency, more general portals could offer more flexibility in designing dynamic virtual environments. We define arbitrary portals as portals that are not held to the conventional constraints. That is, portals with the following characteristics:

1. The portal surface may be any arbitrary surface.
2. The portal may exist anywhere within a cell and connect to any space within an adjacent cell.

Whereas a conventional portal will produce a view frustum with the viewpoint, an arbitrary portal may not. Non-planar portals would create complex view volumes that the adjacent cell contents should be clipped to. The view volume surface may not be a convex hull and it may contain a large number of vertices. As a result, geometric clipping could be very difficult and computationally expensive.

3 Approach

We describe the development of our ideas in this section. First we clarify our terminology:

Frame buffer : The frame buffer refers to the colour buffer and all ancillary buffers. (This is the same frame buffer definition as used by OpenGL.)

Fragment : A fragment is generated during rasterisation of a primitive. It corresponds to a pixel in the frame buffer, but it is not part of the frame buffer.

As mentioned in the previous section, arbitrary portal rendering would require clipping to a potentially complex view volume. Since a geometric solution would be very expensive, we investigated a view-dependent fragment level approximation using fragment culling. We needed to define a mechanism to test whether or not a fragment lies within a portal view volume, and a technique to discard or accept the fragment accordingly.

The most common tools for fragment culling are the depth buffer, stencil buffer, and alpha test. Both the depth buffer and stencil buffer algorithms use per-pixel buffers (the same size and shape as the color buffer) and simple fragment attribute testing to determine whether a fragment should be accepted or discarded. The alpha test simply accepts or discards a fragment based on its alpha value. Dominé [4] observed that discarding fragments using the alpha test could provide reasonable performance improvements. See the book by Möller and Haines [12] for further details.

While the depth and stencil buffers are very useful tools, they do not support enough flexibility for more complex fragment processing. Notably, culling that requires testing of fragment depth values to a depth range. In an algorithm to correctly render transparent surfaces, Mammen [11] used dual *virtual pixel maps* as depth buffers in a multi-pass algorithm that culled fragments to a depth range. Later, Diefenbach [3] observed that Mammen's approach could be optimized by simple buffer swapping and be applied to arbitrary surface clipping. He describes a *dual depth buffer* which consists of two functionally equivalent depth buffers, one of which is available for writing and each with its own compare function.

We recognize that fragment culling within a portal view volume can be achieved by using two existing tools: the stencil buffer and the dual depth buffer. The stencil test can be used to cull fragments in u and v screen coordinates (on the viewplane), and the dual depth buffer can be used to cull fragments in the w screen coordinate (in the view direction).

Fig. 2. This figure illustrates our algorithm for rendering arbitrary portals. When rendering each frame, the current cell is the cell in which the camera viewpoint resides. When objects are rendered they are subject to the stencil test, near depth test, and far depth test. These tests ensure that only meshes and portals that are visible are rendered.

4 Algorithm

First we define our data structures:

- Cell :** A cell contains a set of meshes and portals.
- Portal :** A portal contains a surface and a pointer to an adjacent cell.

Our algorithm is illustrated in Figure 2. It uses the stencil buffer and dual depth buffer. We always use the dual depth buffer for near and far depth testing. Hence, we refer to the two buffers as the *near depth buffer* and *far depth buffer*. The far depth buffer is the only depth buffer available for writing. The main methods in our algorithm are *Render Frame*, *Render Cell*, and *Render Portal*. Here, we discuss each method in detail.

Render Frame: The frame buffer is first cleared. This sets all frame buffer elements to defaults. These relevant defaults are:

- The stencil buffer elements are set to marked.
- The near depth buffer elements are set to represent the camera near plane
- The far depth buffer elements are set to represent the camera far plane.

Once the frame buffer is cleared, the cell in which the camera viewpoint resides is rendered.

Render Cell: This method renders a cell's meshes and portals. When it is invoked, the frame buffer is conditioned to determine the visibility of fragments generated by rendering the cell's contents. If the camera resides in this cell, the frame buffer has been conditioned by the *Render Frame* method to accept all fragments as possibly visible. Otherwise, the frame buffer has been conditioned by calls to the *Render Portal* method. In this case, the stencil test culls fragments that are outside the projection of the portal surface in screen space. The near depth test culls fragments that would appear on the wrong side of the portal surface (the side nearer the camera), and the far depth test resolves visibility within the cell. The cell's portals are also rendered subject to these tests.

Render Portal: This method has three main roles:

1. Condition the stencil, near depth, and far depth buffers such that the adjacent cell is rendered correctly.
2. Invoke adjacent cell rendering.
3. On exit, ensure that the stencil and far depth buffers are conditioned such that the source cell is rendered correctly.

The portal surface needs to be placed in the near depth buffer. This is achieved by rendering it to the far depth buffer and then swapping the depth buffers. The near depth test now ensures that adjacent cell fragments are only accepted if they are behind the portal. The far depth buffer is then cleared so that it can be used for visibility determination within the adjacent cell.

Since the near depth buffer may contain depth information other than portal surface, we need to ensure that adjacent cell fragments are only accepted if they are visible through the portal. This is achieved by marking visible portal surface pixels in the stencil buffer. However, the current stencil buffer will be needed when rendering of the source cell resumes. Therefore, a copy of the stencil buffer is stored and then the portal surface is marked in the stencil buffer. The frame buffer is now correctly conditioned and the adjacent cell is rendered.

After rendering the adjacent cell, we need to restore the depth buffer and stencil buffer so that fragments generated when rendering other portals within the source cell will be correctly culled. This involves restoring the far depth buffer and the stencil buffer to the state they were before the adjacent cell was rendered. Recall that the near depth buffer is never available for writing, and that depth buffers were swapped earlier. So a depth buffer swap now restores the far depth buffer to the correct state. The stencil buffer is restored from the stored copy.

5 Implementation

Now we discuss the implementation of our algorithm. We used C with OpenGL for rendering. We also used the SGIX_shadow [1] and NV_register_combiners extensions [9]. Our development system is a Pentium 4 system with a NVIDIA GeForce4 graphics card.

Fig. 3. This figure illustrates how the fragments tests defined in our algorithm map to our implementation. The stencil and far depth tests map directly to the standard OpenGL tests, but OpenGL does not natively provide a near depth test. Therefore, we emulate a near depth test by using standard texel generation (in conjunction with a shadow map), register combiners, and alpha testing.

The stencil test and far depth tests used by our algorithm map directly to OpenGL's standard stencil and depth tests. However, no near depth test is provided by OpenGL or by vendor extensions. Therefore, we were forced to emulate a near depth buffer.

In his paper on order-independent transparency, Everitt [6] describes a technique to emulate any number of virtual depth buffers on hardware that supports shadow maps (introduced by Weigand [15]), texture shaders [8], and register combiners [9]. Everitt uses the shadow map as an auxiliary per-pixel depth test by matching its resolution to the frame buffer and projecting it from the camera viewpoint. Everitt's technique is rather intolerant to variance in depth values. So he uses texture shaders to interpolate shadow map depth values in the same way the depth buffer interpolates them. We use a similar technique to emulate a near depth buffer. Unlike Everitt's technique, our application is tolerant to variance. Therefore, we do not need to use texture shaders.

We implemented the near depth buffer in the following way. A shadow map texture is generated at the same resolution as the screen by copying the depth buffer. This is projected from the viewpoint by projective texture mapping using OpenGL texture coordinate generation. The shadow map test is configured to set texels to zero for fragments nearer than the stored depth value (and one otherwise), and the register combiners are configured to multiply the fragment alpha by the texel value. Thus, fragments that are nearer than the stored depth have alpha values of one, and those farther have their original alpha values. Finally, the alpha test is set to discard fragments with alpha of zero.

Figure 3 illustrates how the stages in our ideal pipeline translate to the stages in our implementation. The stencil test and far depth test map directly to standard OpenGL tests,

Fig. 4. This figure illustrates our implementation. The objects in the source and destination cells and the portal are in completely general positions. All images are from the same camera. Images *A* and *B* illustrate input. Image *A* shows the cell containing the viewpoint. The green sphere is a mesh, and the grey teapot is a portal (rendered as a mesh). Image *B* shows the portal's destination cell. It contains two meshes: a distant wall and a blue sphere. Images *C* and *D* illustrate processing. Image *C* shows the destination cell subject to the dual depth test. The blue sphere intersects the portal surface and is partially clipped. Therefore, the inside of the blue sphere is visible after clipping. The inside surface is rendered in pink for clarity. Image *D* shows the destination cell subject to the stencil test. The final image *E* is the result combining all fragment tests.

but the near depth test is implemented combining custom shadow mapping (indicated by texture generation), register combiners, and the alpha test. Some results from our implementation are shown in Figure 4.

5.1 Discussion

Features : The benefits of our algorithm are those commonly attributed to fragment level solutions. It provides a discrete runtime solution for visibility determination in all scenes in which cell contents and portal surfaces can be rasterised. The algorithm has linear cost growth with an increase in the number of fragments generated by rasterisation. It is invariant to the geometric complexity of surfaces and does not require geometry sorting or clipping. It is indifferent to geometry changes such as cell transformations or dynamic deformable portal surfaces. All these features are very difficult to support in a conventional portal system.

Limitations : There are a number of limitations to our implementation. However, these are not necessarily intrinsic to our algorithm, as they can be resolved by better hardware support.

When rendering a portal, the visible portal surface is marked by incrementing the stencil value of pixels that pass the stencil test. Thus, the depth of recursion possibly is limited by the depth of the stencil buffer. This is not necessarily a problem. An 8-bit stencil buffers allows for 255 levels, which is adequate for our intended applications. However, if required, the stencil buffer can be copied into a texture when the maximum depth is reached and then cleared and reused. When required it can be restored from the texture.

Multi-texturing is currently disabled in our system. This was omitted to simplify the register combiners and could be added by modifying the register combiners. However, the shadow map will always require a texture. Thus the number of textures that can be applied while using our algorithm will always be one less than the maximum number provided by hardware. That is, except for the cell that the camera resides in since near depth testing would not be required.

6 Conclusion

Although many advances have been made in portal-based techniques, all existing techniques still have similar limitations. Portal rendering involves a lot of geometric operations to determine cell visibility. Accordingly, portals are kept geometrically simplistic. Portal surfaces are always convex and planar polygons and usually restricted to a low number of vertices. This is to simplify and reduce clipping operations that scale linearly in cost with the geometric complexity.

We have presented an algorithm that enables arbitrary portal rendering by using fragment culling. Using our technique, portals with any surface that connect any spaces within cells can be rendered correctly. Our algorithm scales linearly in cost with the number of fragments generated.

We will later re-implement our algorithm in a higher level shading language. More immediately, we plan to utilise the ARB_fragment_program recently ratified by the OpenGL ARB and now supported by hardware. Our technique will be integrated into an arbitrary portal system once we have developed a complementary geometry culling technique that ensures that only necessary fragment tests are performed.

Acknowledgments. The authors would like to thank three anonymous referees for many suggestions that improved the presentation of the paper. This research is partially supported by an Australian Postgraduate Award, Western Australian Interactive Virtual Environments Center (IVEC) and Australian Partnership for Advanced Computing (APAC).

References

1. SGIX_shadow: <http://oss.sgi.com/projects/ogl-sample/registry/SGIX/shadow.txt>, 2001.
2. Daniel G. Aliaga and Anselmo A. Lastra. Architectural walkthroughs using portal textures. *IEEE Visualization '97*, pages 355–362, November 1997. ISBN 0-58113-011-2.
3. P. Diefenbach. Pipeline rendering: Interaction and realism through hardware-based multi-pass rendering. *PhD Dissertation, Department of Computer Science, University of Pennsylvania*, 1996.
4. S. Dominé. Alpha test tricks. Whitepaper, NVIDIA Corporation, 2001.
5. J. Tybergheir et al. Crystal space 3d engine. <http://crystal.sourceforge.net>, 2002.
6. C. Everitt. Interactive order-independent transparency. Whitepaper, NVIDIA OpenGL Applications Engineering, 2001.
7. C. B. Jones. A new approach to the 'hidden line' problem. *Computer Journal*, 14(3):232–237, August 1971.
8. M. J. Kilgard. NV_texture_shader. http://oss.sgi.com/projects/ogl-sample/registry/NV/texture_shader.txt, November 2001.
9. M. J. Kilgard. NV_register_combiners. http://oss.sgi.com/projects/ogl-sample/registry/NV/register_combiners.txt, February 2002.
10. D. Luebke and C. Georges. Portals and mirrors: Simple, fast evaluation of potentially visible sets. *1995 Symposium on Interactive 3D Graphics*, pages 105–106, April 1995. ISBN 0-89791-736-7.
11. A. Mammen. Transparency and antialiasing algorithms implemented with the virtual pixel maps technique. *IEEE Computer Graphics & Applications*, 9(4):43–55, 1989.
12. T. Möller and E. Haines. *Real-Time Rendering*. A K Peters, 1999.
13. G. Taylor and N. Lowe. Phantoms in the corridor: Portal systems and the digital mind. In *Proceedings of the Biennale of Electronic Arts Perth*, 2002.
14. S. Teller. Visibility computations in densely occluded polyhedral environments. Technical Report CSD-92-708, Berkeley, California, U.S.A., 1992.
15. T. F. Wiegand. Interactive rendering of csg models. *Computer Graphics Forum*, 15(4):249–261, 1996. ISSN 0167-7055.