# A New Technique for Rendering Complex Portals

Nick Lowe and Amitava Datta, *Member*, *IEEE Computer Society*

**Abstract**—In this paper, we identify a general paradigm for portal-based rendering and present an image-space algorithm for rendering complex portals. Our general paradigm is an abstraction of portal-based rendering that is independent of scene geometry. It provides a framework for flexible and dynamic scene composition by connecting cells with transformative portals. Our rendering algorithm maintains a visible volume in image-space and uses fragment culling to discard fragments outside of this volume. We discuss our implementation in OpenGL and present results that show it provides correct rendering of complex portals at interactive rates on current hardware. We believe that our work will be useful in many applications that require a means of creating dynamic and meaningful visual connections between different sets of data.

**Index Terms**—Portal-based rendering, fragment-culling, dual-depth-buffer, scene composition, complex portals.

## 1 Introduction

Portal-based rendering has been used extensively for accelerating visibility determination in complex architectural scenes. Portals are used to partition a scene into cells such that another cell is only visible if a portal that bounds that cell is visible. Due to the nature of indoor scenes, a number of constraints are usually placed on portal characteristics; portals are usually simple convex and planar polygons that lie on cell boundaries. Some previous work has indicated that portals can be used for purposes other than fast visibility determination. For example, transformative portals are very useful for representing mirrors [11] and in composing scenes by linking cells [19]. However, a framework for general transformative portals has not been explored in the past due to the limitations placed on portals.

In this paper, our aim is to present a higher-level abstraction of the portal-based rendering paradigm that allows for nonconvex and nonplanar transformative portals. We refer to this abstraction as our *general portal paradigm* and to portals with such characteristics as *complex portals*. We suggest that our general portal paradigm may provide a flexible framework for creating dynamic scenes. Since current portal-based rendering techniques that are based on geometric clipping would be impractical for complex portals, we also provide a rendering technique based on image-space fragment culling.

### 1.1 Review of Portal-Based Rendering

We provide a brief background on portal-based rendering. The purpose of most previous work is to accelerate visibility determination by using basic portals. Our aim in this paper is quite different. We concentrate on the correct visibility determination required for rendering complex portals.

The idea behind conventional portal-based rendering is that observers inside a building can only see a room if they are either inside it or they can see into it through a doorway or window (or a series of doorways and windows). This simple idea provides the foundation for a very efficient means of generating a *potentially visible set (PVS)* of rooms for a given viewpoint. Rendering only the PVS minimizes the amount of data sent through the rendering pipeline and can greatly improve overall rendering speed. The portal technique was originally proposed by Jones [8] as a method for hidden-line removal. It saw renewed popularity in the early 1990s with Airey et al. [3], Teller [17], and Luebke and Georges [11] primarily offering application, reinterpretation, and optimization. In the late 1990s, Aliaga and Lastra [4] used image-based rendering techniques with portal-based rendering in order to significantly reduce the rendering complexity of massive models. The technique has also become quite popular in commercial products and real time 3D engines (notably the Unreal Engine [1] and Tyberghein's Crystal Space engine [19]).

Most previous work dictates that, in order to render a scene using a portal-based technique, it must first be partitioned into convex polyhedral regions, called *cells*, that are separated by convex planar polygons, called *portals*. At runtime, the rendering algorithm uses the partition information to determine and render only visible regions. This two-stage approach is very similar to other visibility determination techniques such as the octree and BSP-tree (binary space partition tree). A good overview of these visibility determination techniques is given by Möller and Haines [13]. In all of these techniques, region adjacency is stored in a graph structure that is traversed at runtime to determine visible regions. In portal-based rendering, the graph structure generated by partitioning the scene is called the *cell and portal graph (CPG)*. This graph represents the adjacency and visibility relationships between cells. The nodes of the graph represent cells and the edges between

• *The authors are with the School of Computer Science & Software Engineering, University of Western Australia, 35 Stirling Highway, Crawley, Perth, WA 6009, Australia.*
*E-mail: {nickl, datta}@csse.uwa.edu.au.*

*Manuscript received 11 Dec. 03; revised 4 May 2004; accepted 10 May 2004.*
*For information on obtaining reprints of this article, please send e-mail to: tvcg@computer.org and reference IEEECS Log Number TVCG-0122-1203.*

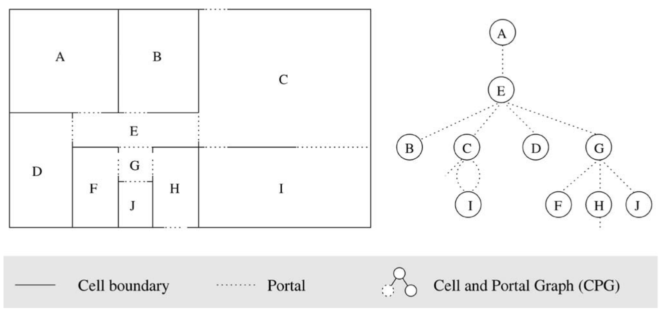


Fig. 1. The image to the left illustrates how a basic set of rooms would be decomposed into *cells* and *portals*. The cells are labeled A through I and the portals are represented by dashed lines on cell boundaries. The image to the right is the corresponding *cell and portal graph (CPG)*. Each cell is a node in the graph and each portal is a graph edge.

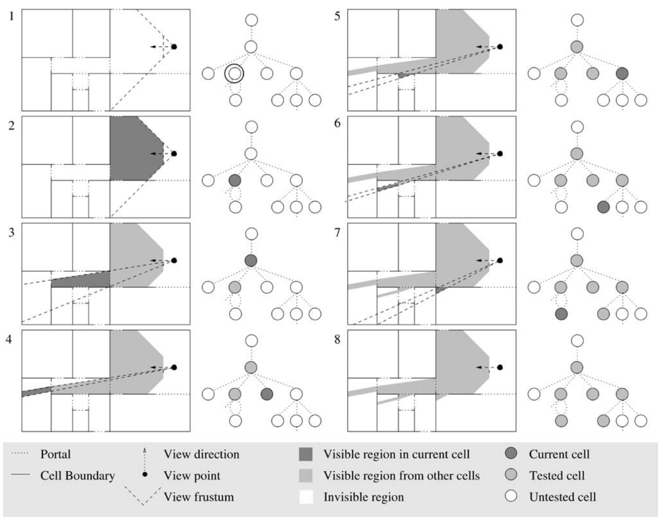


Fig. 2. This figure illustrates the portal-based rendering algorithm applied to the scene from Fig. 1. Rendering starts in the cell containing the viewpoint. Image 1 shows the camera placed in the scene. The initial visible volume is the space within the camera view frustum. The visible portion of the current cell is shown as dark gray in Image 2. Within this area, a portal is visible. Therefore, for each visible portal, a new visible volume is calculated representing the space visible through that portal.

cells represent the shared portals between cells. Fig. 1 illustrates a scene partitioned into cells and portals. It also shows the corresponding CPG graph. The *portal-based rendering algorithm* determines the PVS of a scene by depth-first traversal of its CPG. Traversal begins in the node representing the cell containing the viewpoint and an edge is only followed if the portal it represents is visible from the viewpoint. Since cell visibility is determined by the visibility of a portal into it, this ensures that only visible cells are visited. Fig. 2 demonstrates the portal-based rendering algorithm using the CPG of the basic scene from Fig. 1.

## 1.2 Review of Image-Space Methods

In this paper, we devise a scheme for rendering portals that cannot be rendered efficiently using geometric techniques, that is, portals that are transformative, nonconvex, and nonplanar. A common approach to solve similar problems is the use of *image-space methods*. Here, we provide a brief background on image-space methods.

An image of a scene is a discretization of that scene from a given view. The idea behind image-space methods is that operations on this scene discretization will produce appropriate results for that view. For example, masking pixels in image-space can produce the effect that parts of the scene are not visible. However, the scene geometry has not changed. To produce the same effect with geometric operations would require calculating the projection of the masked pixels onto the scene and actually removing the corresponding geometry. Clearly, image-space methods are view-dependent, and geometric operations are not. However, complex geometric operations may be simulated by very basic image-space operations such that the required image-space processing for each frame will actually be less for a dynamic scene. Also, as scene complexity increases, geometric operations tend to scale exponentially. In contrast, image-space operations tend to scale linearly with the size of the output image.

In most real time systems, rendering is accomplished by rasterization. Rasterization generates *fragments* that are subject to image-space *fragment operations* before they may contribute to the output image. *Fragment culling* techniques are fragment operations that accept or reject fragments based on their properties and the contents of *ancillary buffers* that are the same width and height as the output image. Perhaps the most common fragment culling technique is the *depth-buffer* technique. The depth-buffer, also called the z-buffer, is an ancillary buffer that stores distances from the view-point to fragments that have contributed to the output image. By using a *depth test*, fragments can be accepted or rejected based on whether or not they are closer to or further from the viewpoint. For opaque scenes, robust view-dependent visibility determination is easily achieved by only accepting the fragments nearest to the viewpoint. A good review of depth-buffer applications is provided by Theoharis et al. [18].

While the depth-buffer is a very useful tool, it only stores a single depth value. Thus, it cannot natively support fragment operations that require culling to a depth range. This has led to a number of similar techniques in areas that require a depth-range test (such as translucent surfaces rendering). In an algorithm, to correctly render translucent surfaces, Mammen [12] used dual *virtual pixel maps* as depth buffers in a multi-pass algorithm that culled fragments to a depth range. Rossignac and Wu [14] used a *depth-interval buffer* for shading from CSG (constructive solid geometry). Later, Diefenbach [6] observed that Mammen's approach could be optimized by simple buffer swapping and used for view-dependent 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.

The rest of this paper is organized as follows: We outline our *General Portal Paradigm* in Section 2. The details of our general portal rendering algorithm are given in Section 3. A version of the algorithm that uses common features of graphics hardware is presented in Section 4. Section 5 provides details of our implementation using OpenGL. We discuss the results in Section 6. Finally, we provide concluding remarks in Section 7.

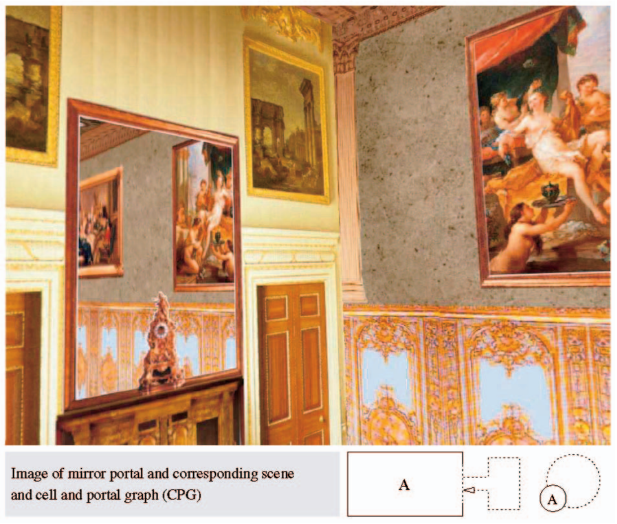


Image of mirror portal and corresponding scene
and cell and portal graph (CPG)
A
A

Fig. 3. This figure illustrates a cell containing a *mirror portal*. Here, a mirror effect is produced by creating a portal that connects a cell to itself with a reflective transformation about the plane of the portal. The top-down view of the scene and the cell and portal graph are shown to the bottom right.

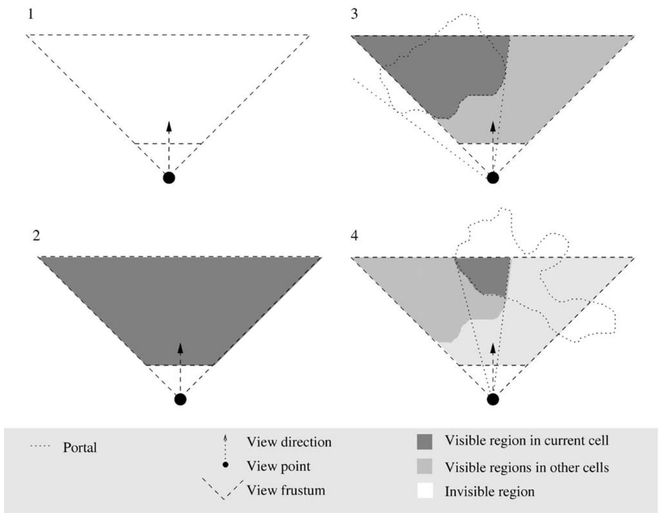


Fig. 4. This figure illustrates the visible volume of space for a camera and portals. The first image is a camera defined by a viewpoint, view direction, and view frustum. Image 2 shows the visible volume for the cell containing the viewpoint. Everything within the camera frustum is potentially visible. In Image 3, a portal is added to the cell. The visible volume for the adjacent cell is the volume of space within the frustum, but behind the portal surface. In the final image, a portal is added to the adjacent cell. The visible volume through that portal is the volume behind the portal surface and within the visible volume of the previous cell.

## 2 A General Portal Paradigm

The general idea behind portal-based rendering is that *cells are only visible through portals to them*. Jones [8] found this idea very useful in formulating an efficient means to determine visibility for indoor scenes. Later, Teller [17] suggested that mirrors could be modeled by portals that connect a cell to itself but apply a reflective transformation before rendering the cell a second time. Luebke and Georges [11] implemented limited *mirror portals* and found that general mirror portals introduced significant clipping problems. They stopped short of the dynamic, fully recursive case. Fig. 3 illustrates a mirror portal. Tyberghein's Crystal Space engine [19] uses a more general portal scheme. Each portal has an associated coordinate transformation that is applied before the adjacent cell is rendered. Also, portals are unidirectional and any cell may be adjacent to any portal. This means that mirror portals are only a subset of a larger set of *transformative portals* that can be used for novel scene construction.

Previous work on transformative portals demonstrates that portals can be used for novel scene construction rather than for just visibility determination. Novel scenes can be created by modifying portal transformations and links. However, transformative portals come at the cost of increased geometric clipping. This has been the primary problem with previous transformative portal systems. Often, constraints are introduced to reduce clipping issues. In all previous systems, portals are necessarily convex and planar polygons. Also, portals are not permitted to intersect. In the earliest systems, this was guaranteed because portals had to exist at cell boundaries. However, even in the more recent Crystal Space engine [19], which allows portals to exist in (and link to) any space within a cell, portals are still not allowed to intersect.

We propose that the restrictions on portal characteristics limit the usefulness of portals for dynamic scene composition. For example, an application may wish to animate a portal shape to convey some information on the characteristics of the adjacent cell. The portal shape must remain planar, so the animation is somewhat limited. If the portal surface becomes concave, it must be decomposed into a set of convex portals. If the portal is moving, the system must ensure that it does not intersect any other portals. Clearly, these restrictions are due to implementation details (geometric clipping), rather than on a conceptual basis. We propose a more abstract portal paradigm in which portals are not similarly restricted. Previous work is based on the concept that portals are necessarily convex and planar polygons that connect cells and cannot intersect other portals. We believe that a more useful paradigm is simply that cells contain data, portals connect them, and both can be rasterized. We hope that, in many cases, this approach can provide an elegant solution to the problem of creating dynamic links between related data in a virtual or visualization environment.

## 3 The General Algorithm

Our general portal paradigm describes a system in which portals can be used to create meaningful visual links between cells. Scenes created by linking cells with complex portals will require a different approach to visibility determination than normal scenes. However, the basic requirements for visibility determination remain the same. Visibility determination amounts to determining the *visible volume* of space and rejecting anything outside of this volume. For normal scenes, the visible volume of space is the space that lies within the camera *view frustum* and that is not occluded by other objects. For a scene with portals, each cell will have a different visible volumes that depends on the visibility of the portal that links to it. The visible volume for a normal scene and for a scene with a portal is illustrated in Fig. 4. An abstract portal rendering algorithm for our

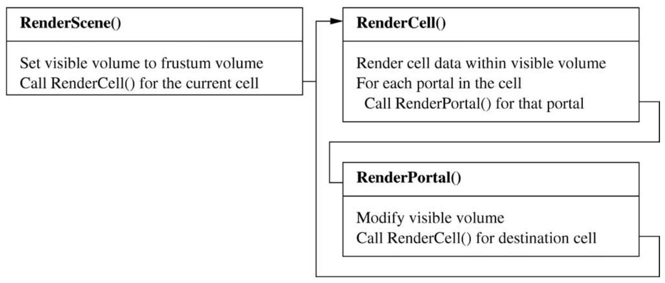

RenderScene()
Set visible volume to frustum volume
Call RenderCell() for the current cell
RenderCell()
Render cell data within visible volume
For each portal in the cell
Call RenderPortal() for that portal
RenderPortal()
Modify visible volume
Call RenderCell() for destination cell

Fig. 5. This figure illustrates the most abstract form of a portal-based rendering algorithm. It defines the key operations that are required for portal-based rendering without specifying a method for visibility determination. Previous approaches have specified geometric operations within the basic algorithm.

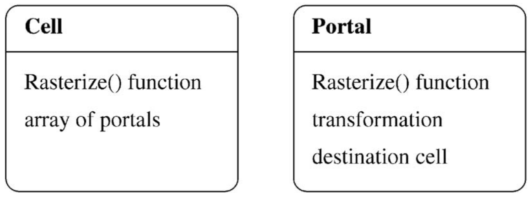


Fig. 6. This figure illustrates our cell and portal data structures. The cell data structure contains a rasterization function and a set of portals. The portal data structure also contains a rasterization function. It also contains a destination cell pointer (representing the cell that it connects to) and a transformation (representing the coordinate frame transformation from the cell containing the portal to the destination cell). Note that neither data structure contains any geometric data. Since our algorithm operates purely at the fragment level, it does not require geometric information.

general portal paradigm is illustrated in Fig. 5. The abstract algorithm dictates that only cell data within the visible volume is rendered. How this is achieved is independent of the abstract algorithm and the best approach will vary depending on the characteristics of the cell data and the underlying graphics hardware.

For polygon rendering systems, determining what lies within the visible volume can be performed as a geometric operation or a fragment operation. As a geometric operation, polygons are clipped to the visible volume before they are rasterized. As a fragment operation, polygons are rasterized and then each fragment generated is tested and either accepted or rejected. Gross geometric operations (such as frustum culling) are often used to discard a large amount of scene data that is not visible, but basic fragment operations (such as the depth-buffer technique) are used to resolve visibility for detailed geometry.

The algorithm we have developed is a fragment culling technique designed for complex portal rendering. Since a scene containing complex portals is characteristically different from a normal scene, the way in which fragments are tested for visibility is quite different. The idea behind our algorithm is that we define the visible volume in screen-space with two depth values per screen pixel. The first one is the near-depth value. It represents the portal surface through which the cell is visible. Any fragment nearer than this value is not visible through the portal. The second one is the far-depth value and is used to store the nearest valid fragment rendered thus far. Any fragment further than this value is occluded by the nearer fragment and should be discarded.

Since this is a fragment approach, actual geometry is not required. The only requirement is that cells and portals produce fragments. The data structures we use to describe complex cells and portals are illustrated in Fig. 6. The `Rasterize()` functions in both structures generate fragments and return the number of fragments rendered (that is, fragments that pass all fragment tests). This information is used to determine when to stop the recursive algorithm. When rasterizing a portal produces no visible fragments, the adjacent cell will not be visible. Therefore, traversal of the CPG for that cell can cease.

## 4 An Algorithm with Hardware Support

Our general algorithm illustrated in Fig. 5 communicates the flow of our algorithm, but does not provide details on how we update or clip to the visible volume. These details are specific to graphics hardware. In this section, we provide a more specific algorithm for hardware with a frame-buffer that contains a stencil-buffer and dual-depth-buffer. Most consumer-level graphics hardware should fall into this category. The stencil-buffer is widely supported and Everitt [7] has shown that a dual-depth-buffer can be emulated using two other widely supported features: the depth-buffer and shadow-maps. Therefore, we have chosen to use these buffers for fragment culling.

In order to clearly communicate our detailed algorithm, we first clarify our definitions for the frame buffer and important fragment tests. We then present the functions that comprise our algorithm and discuss them in detail.

### 4.1 The Frame-Buffer

We define the frame-buffer as a color-buffer (used for storing the output image) and two ancillary buffers: the stencil-buffer and the dual-depth-buffer. The stencil-buffer is a general purpose buffer that is most commonly used to mark pixels that meet an application-specific criterion, such as back-facing pixels. We use the stencil-test to mark visible portal fragments. The dual-depth-buffer contains two buffers: the near-depth-buffer and the far-depth-buffer. Each contains depth values normalized between the near and far planes of the frustum. When the frame-buffer is cleared, the stencil values are set to zero, the near-depth values are set to 0.0 (representing the camera near plane) and the far-depth values are set to 1.0 (representing the camera far plane).

### 4.2 Fragment Tests

Rasterization is the process of producing fragments. Fragments are generated for screen coordinates (u, v) and contain a color value and a normalized eye-space depth value. Each fragment generated by rasterization is subject to a number of fragment tests. These tests operate on the incoming fragment and the corresponding frame-buffer elements (those at the same screen-coordinates). Some tests also depend on other variables. The `stencil_reference_value` is used for the stencil test and it can be set to any positive integer. The `far_depth_function` is used for the far-depth test and it can be set to one of `less_than` or `equal_to`. The fragment tests are as follows:

- The `stencil_test` returns true if the stencil reference value is the same as the corresponding stencil-buffer value.
- The `near_depth_test` returns true if the fragment depth value is greater than the corresponding near-depth-buffer value.
- The `far_depth_test` depends on the `far_depth_function`. If the far depth function is `equal_to`, the far-depth test returns true if the fragment depth value is equal to the corresponding far-depth-buffer value. Similarly for `less_than`, the far-depth test returns true if the fragment depth value is less than the corresponding far-depth-buffer value.

## 4.3 Updating the Frame-Buffer

If a fragment passes all fragment tests, the corresponding frame-buffer elements are updated. There is one variable that helps define how the frame-buffer is updated. The *stencil operation* is used to determine how the stencil-buffer element is updated. It can be set to one of: `increment`, `do_nothing`, `decrement`. The frame-buffer is then updated as follows (note that the near-depth-buffer is never updated):

- The *color-buffer element* is set to the fragment color.
- The *stencil-buffer element* is set according to the stencil operation. For stencil operations `increment` and `decrement`, the value is incremented and decremented, respectively. If the stencil operation is `do_nothing`, the element is not modified.
- The *far-depth-buffer element* is set to the fragment depth.

## 4.4 Variables

We do not require any frustum or camera structure, but we do require a cell variable, `current_cell`, and a recursion depth counter, `recursion_depth`, to identify the cell containing the viewpoint and the recursion depth of the algorithm.

## 4.5 Functions

The complex portal rendering algorithm is comprised of three functions: `RenderScene()`, `RenderCell()`, and `RenderPortal()`,

```
Function1 RenderScene()
Require: current_cell is defined

   Condition the frame-buffer
   Clear the frame-buffer
   Disable writing to the color-buffer

   Condition the fragment tests
   Disable the near-depth test
   $stencil\_reference\_value \Leftarrow recursion\_depth$
   $stencil\_operation \Leftarrow do\_nothing$
   $far\_depth\_function \Leftarrow less\_than$

   Render the current cell
   Call RenderCell() for current_cell
```

The `RenderScene()` function (Function 1) conditions the frame-buffer, fragment tests, and frame-buffer update variables to their correct initial states. That is, the states required to correctly render the cell containing the viewpoint. The color-buffer is cleared to the background color. The stencil-buffer is set to all 0s. All elements in the near-depth-buffer are set to 0.0 (representing the camera near plane). Finally, all elements in the far-depth-buffer are set to 1.0 (representing the camera far plane). Here, the near-depth test could be disabled as an optimization. Since all values are 0.0, the test will always pass. The stencil reference value is set to 0. Since all elements in the stencil-buffer are initialized to 0, all fragments should pass the stencil test. This is the desired action because all fragments generated in the current cell are potentially visible from the viewpoint. The stencil operation is set to `do_nothing` so that the fragments generated by cell rasterization will not affect the stencil buffer. Only portals should affect the stencil buffer because only they can reduce the visible volume. The far-depth test is set to `less_than` so that only the nearest fragment generated by rasterizing the current cell is drawn. Now that the frame-buffer and fragment tests are conditioned correctly, the recursive algorithm is started by rendering the cell containing the viewpoint.

```
Function 2 RenderCell()
   Rasterize portals to far-depth-buffer
   for all portals in this cell do
      Call the portal's Rasterize() function
   end for

   Rasterize the cell to the color-buffer and far-depth-buffer
   Enable writing to the color-buffer
   Call this cell's Rasterize() function
   Disable writing to the color-buffer

   Condition the near-depth-buffer
   Copy the far-depth-buffer to the near-depth-buffer

   Render all portals in this cell
   for all portals in this cell do
      Call RenderPortal() for the portal
   end for
```

The `RenderCell()` function (Function 2) renders a cell. When entering this function, the frame-buffer and fragment tests are correctly conditioned such that only fragments within the visible volume can pass all fragment tests. The near-depth-buffer and stencil-buffer are conditioned to identify visible pixels and each pixel's far-depth has been cleared so that the far-depth test can be used for visibility determination between fragments within this cell. Portals are rasterized to the far-depth-buffer and the cell is rasterized to the far-depth-buffer and the color-buffer. At this point, the far-depth-buffer contains all data for the nearest portal fragments. Therefore, it can be used as the near-depth-buffer for all portals in this cell. Thus, it is copied to the near-depth-buffer and the portals within the current cell are rendered.

**Funtion 3** RenderPortal()

```
**Mark visible portal pixels in stencil-buffer**
disable *near-depth-test*
$stencil\_reference\_value \Leftarrow recursion\_depth$
$stencil\_operation \Leftarrow increment$
$far\_depth\_function \Leftarrow equal\_to$
call Rasterize() for the portal

**Stop if portal is not visible**
**if** no marked pixels **then**
    enable *near-depth-test*
    $stencil\_operation \Leftarrow do\_nothing$
    $far\_depth\_function \Leftarrow less\_than$
    return
**end if**

**Clear depth values of portal pixels in far-depth-buffer**
$stencil\_reference\_value \Leftarrow (recursion\_depth + 1)$
$stencil\_operation \Leftarrow do\_nothing$
disable *far-depth-test*
render full-screen quad with depth of 1.0

**Render portal destination cell**
enable *near-depth-test*
$far\_depth\_function \Leftarrow less\_than$
enable *far-depth-test*
transform coordinate system
call RenderCell() for destination
return to previous coordinate system

**Unmark pixels in stencil-buffer**
$stencil\_reference\_value \Leftarrow (recursion\_depth + 1)$
$stencil\_operation \Leftarrow decrement$
disable *near_depth_test*
disable *far_depth_test*
render full-screen quad
enable *near_depth_test*
enable *far_depth_test*
```

The `RenderPortal()` function (Function 3) conditions the frame-buffer, fragment tests, and frame-buffer update variables for cell rendering. It then renders the adjacent cell. After the adjacent cell has been rendered, it returns the frame-buffer and all variables to their previous state. The first step in conditioning the frame-buffer is marking the visible portal pixels in the stencil-buffer. Upon entering this function, the far-depth-buffer contains the nearest fragments, including those generated by rasterization of the source cell's portals. Therefore, any fragment generated by portal rasterization that has depth equal to that stored in the far-depth-buffer indicates a visible portal pixel. To mark these pixels, we increment the corresponding stencil-buffer value. If no pixels are marked, the portal is not visible and the function can exit. However, it should return to the correct state before doing so.

Now that the visible pixels are marked in the stencil-buffer, the far-depth values of these pixels currently represent the portal surface. These values need to be cleared so that the far-depth test can be used for visibility determination for the adjacent cell. In order to clear these values, we use the stencil test so that only marked pixels are affected and set the near-depth values to 1.0 (representing the far plane). We then apply the portal transformation and render the adjacent cell. Once the adjacent cell is rendered, we return to the state before the portal was rendered. That is, we return to the source cell's coordinate system and unmark pixels in the stencil-buffer.

## 5 Implementation

We implemented our algorithm in C using OpenGL 1.4 [16] and the GL_NV_occlusion_query [5] extensions to OpenGL. 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 [7] describes a technique to emulate any number of virtual depth buffers on hardware that supports shadow maps (introduced by Williams [21]), texture shaders [9], and register combiners [10]. 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 application 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 just use a depth component texture. We also do not need to use register combiners because all the functionality we require is provided by standard texture application modes and the alpha test.

We implemented the near-depth-buffer slightly differently. 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 depth component texture modifies the alpha value of the fragment such that fragments outside of the visible volume are assigned an alpha value of zero. Finally, the alpha test is set to discard fragments with alpha of zero. Fig. 7 illustrates how the stages in our ideal pipeline map to the stages in our implementation. The stencil test and far-depth test map directly to standard OpenGL tests, but the near-depth test is implemented combining custom shadow mapping (indicated by texel generation) and the alpha test. We use the occlusion query extension to count the number of pixels a portal covers. If the portal covers no pixels, the destination cell does not need to be rendered.

Clearly, any other fragment test that can provide the same functionality as our current implementation would be applicable to our complex portal rendering algorithm. Many new developments in programmable graphics hardware look promising. For our next implementation, we plan to develop a simple fragment shader that uses the OpenGL

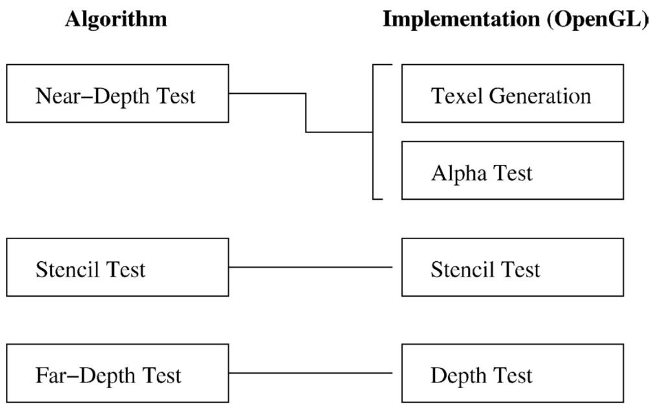


Fig. 7. This figure illustrates how the fragment tests defined in our algorithm map to our implementation. The stencil and far-depth tests map directly to the standard OpenGL tests, but OpenGL does not natively provide a near-depth test. Therefore, we emulate a near-depth test by using standard texel generators (in conjunction with a shadow map) and the alpha test.

Shading Language [15] and an offscreen buffer or texture to represent the near-depth-buffer.

## 6 Results and Discussion

We devised our tests to verify the correctness of our algorithm. However, we also provide some general performance numbers for our implementation. These are detailed in Table 1. Our test machine was a Pentium-4 2.4GHz system with 1GB of RAM and an Nvidia Geforce4 Ti4600 graphics card. For all tests, the image resolution was 512 by 512 pixels.

The first test for our algorithm was very basic. We created two similar cells containing spheres and connected them with a portal. We used a function that draws a well-known teapot for the portal rasterization function. We then inspected the results using the stencil-buffer and dual-depth-buffer, both together and independently. The results of this basic test are illustrated in Fig. 8. These results are consistent with our expectations. The dual-depth test alone produces incorrect fragment-culling because the near-depth-buffer contains all source cell surfaces (rather than only the portal surface). The stencil test alone correctly masks nonportal pixels, but does not clip the destination cell data to the portal surface. Used together, they produce

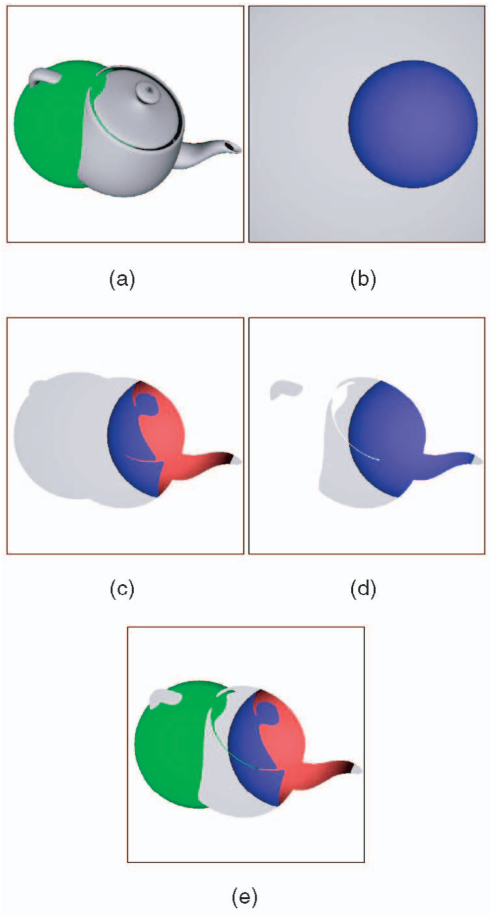


Fig. 8. This figure illustrates the input, processing, and results of our complex portal rendering implementation. All images are taken from the same camera. (a) and (b) illustrate input. (a) shows the cell containing the viewpoint. The green sphere is a mesh and the gray teapot is a portal (rendered as a mesh). (b) shows the portal's destination cell. It contains two meshes: a distant wall and a blue sphere. (c) and (d) illustrate processing. (c) shows the destination cell subject to the dual-depth test (the inside surface of the sphere is rendered in pink). (d) shows the destination cell subject to the stencil test. The final image, (e), is the result combining all fragment tests.

the correct results. We did not do any performance testing for this example.

For further tests, we created two visually distinct cells: The *cube room* and the *landscape* illustrated in Fig. 9. To add a portal to a cell, we require a rasterization function, a coordinate transformation, and a destination cell. Fig. 10 shows how we add a portal to the cube room. First, we use the rasterization function for a surface (in this case, a simple cube). Then, we connect it to the landscape with a slight scaling transformation. We also add a portal from the landscape back to the cube room, resulting in a recursive portal loop. Videos 1 and 2 show basic cube portals connecting the two cells. Video 3 shows a recursive portal loop animated by modifying the two portals' orientations and transformations.

The final test for our algorithm was to check for pixel-level correctness for animated geometrically complex portals. Fig. 11 shows two complex surfaces with overlapping sections. We created portals using the rasterization functions for these surfaces and connected them to the cube

TABLE 1
Performance Results of Our Implemented Algorithm

| *Test Description* | *Frames Per Second* | | | *Depth of Recursion* | | |
|---|---|---|---|---|---|---|
| | *Min* | *Max* | *Avg* | *Min* | *Max* | *Avg* |
| Cube Room to Landscape (Video 1: non-recursive) | 102 | 112 | 102 | - | - | - |
| Landscape to Cube Room (Video 2: non-recursive) | 108 | 123 | 116 | - | - | - |
| Cube Room to Landscape (no video: recursive) | 25 | 100 | 46 | 0 | 12 | 6 |
| Landscape to Cube Room (Video 3 : recursive) | 22 | 53 | 27 | 3 | 8 | 7 |
| Long Demonstration Video (Video 6 : recursive) | 13 | 166 | 45 | 0 | 12 | 4 |

*The minimum frame rate corresponds to the maximum depth of recursion and vice versa.*

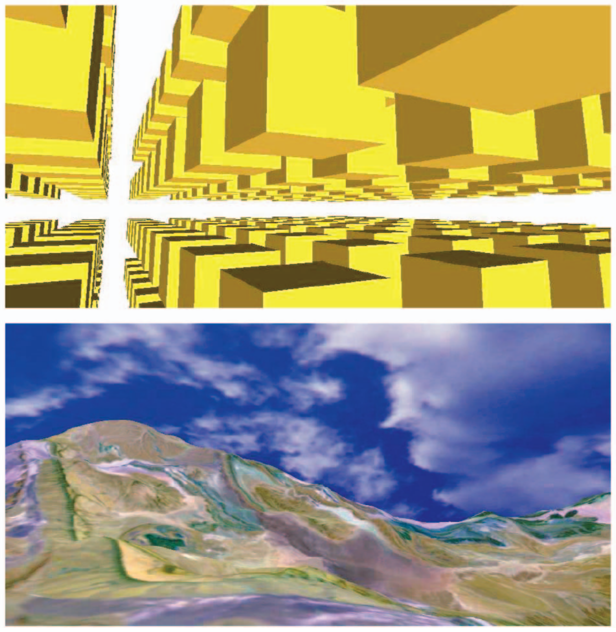


Fig. 9. This figure shows our two test cells. The first is the *cube room* and the second is the *landscape*. We chose these two environments because they are easily distinguishable.

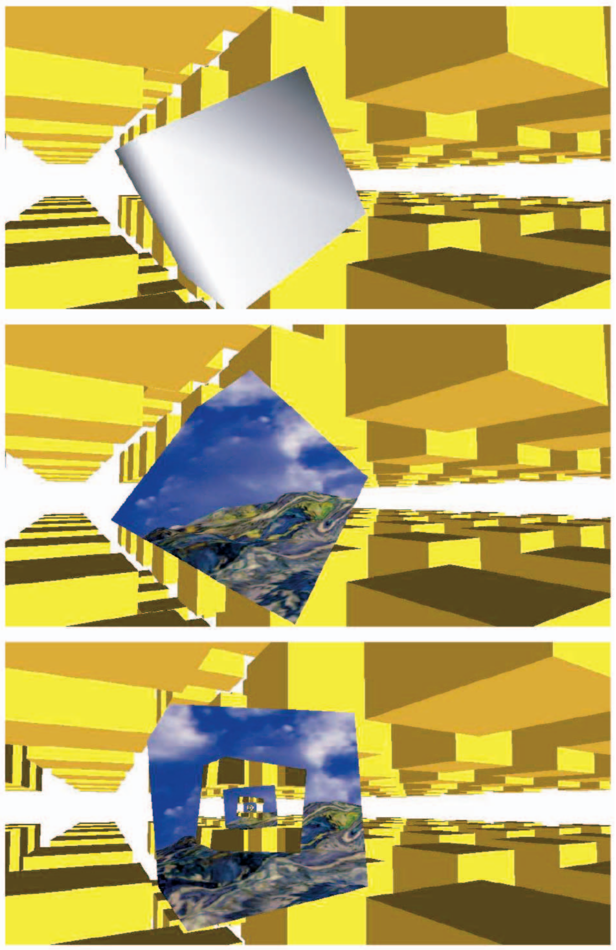


Fig. 10. This figure illustrates adding a portal to the cube room cell. The first image is the surface of the portal. In this example, we have chosen a cube for visual simplicity. In the second image, a portal is created by using the surface rasterization function and linking to the landscape cell (with a slight scale transformation). In the final image, we have also added a portal to the landscape cell back to the cube room cell.

room and landscape cells. The visibility between the two portals, as well as the visibility of the destination cells through them, is correct. This is illustrated well by Videos 4 and 5, in which we animate the two portal surfaces passing through each other. Fig. 12 is a series of images from the video. We did no performance testing for this test as we were checking for correctness. We have also produced another high-quality video showing recursive complex portals. The performance of this long demonstration video is described in Table 1. More details and videos of most tests are available from our website: http://www.csse.uwa.edu.au/~nickl/TVCG.

## 6.1 Limitations and Benefits

Our implementation provides exact fragment-level visibility determination when rendering complex portals. On our development platform, each test runs at interactive rates. However, there is considerable slow down when the depth of recursion of rendering is high (refer to Table 1). For example, this would occur when two portals reflect each other. This situation potentially renders a lot of geometry because two cells are visited recursively in a loop. Since each cell is rasterized each time it is visited, this can result in many rasterization operations. This was an expected outcome. Our primary concern for this work was to develop a paradigm and design a pixel-exact visibility technique. We anticipate using complementary techniques much like many real-time graphics systems today use some gross geometry culling with the depth-buffer rather than the depth-buffer alone. We also hope that new technologies developed for depth range operations, like the *delay stream* of Aila et al. [2], or other deferred rendering schemes, could be used for faster implementations.

Another characteristic of our implementation is the use of the stencil-buffer. The dual-depth-buffer alone can be used to cull fragments outside of a portal surface, provided it is the only surface in the near-depth-buffer. So, the stencil-buffer is not entirely necessary to produce an exact output image. However, it does provide a few advantages: First, since portal surfaces are logically opaque, the portions of the near-depth-buffer used by portals in the same source cell will be mutually exclusive. Therefore, we can use the stencil-buffer to mark these areas rather than regenerate the near-depth-buffer for each portal. This makes the number of complete near-depth-buffer writes the same as the traversal depth of the CPG, rather than the number of edges visited during traversal. Another benefit is the correctness of the resulting far-depth-buffer. Only stencil marked far-depth-buffer values are cleared for each portal and these values are rewritten with the correct values by fragments generated from rasterization of the adjacent cell. Therefore, the final depth values correspond to the actual rendered fragments. This would not occur if the entire far-depth-buffer was cleared for each portal. This is in contrast to many other depth range techniques, such as those derived from Wiegand's constructive solids geometry (CSG) meth-

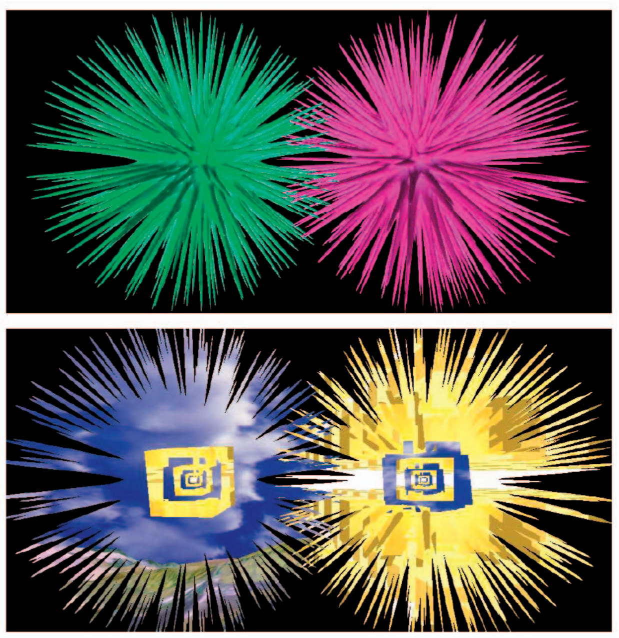


Fig. 11. This image shows two portals that use the complex object to represent their surfaces. They connect to the cube room cell and the landscape cell, respectively. The cube room and landscape cells also have cells connecting them. Notice that the visibility determination is handled perfectly at the pixel level.

od [20], that do not necessarily leave meaningful data in the depth-buffer.

In our implementation, the maximum depth of recursion is limited by the depth of the stencil-buffer. This is not necessarily a problem. An 8-bit stencil-buffer allows for 255 levels of recursion. This should be more than adequate for most applications. The maximum depth of recursion in our tests programs was only 12 (see Table 1). There are a number of limitations introduced because we store the near-depth-buffer as a depth texture: First, our implementation requires enough texture memory to store the depth texture. We do not see this as a significant problem, as current graphics cards tend to have quite a lot of texture memory. Second, since the near-depth-buffer is applied as a texture to all fragments, the number of textures that can be applied while using our algorithm will always be one less than the maximum number that can be applied by the graphics hardware.

## 7 Conclusion

We have presented a general paradigm for portal-based rendering that allows for the composition of scenes by connecting cells with complex portals. Although the foundation of portal-based rendering is in the use of portals for fast visibility determination, our inspiration was drawn from previous work that alluded to the applicability of portals in scene composition. This led to the development of our *general portal paradigm* as a potentially useful framework for creating visual connections between independent data sets.

Our general portal paradigm allows for *complex portals* that may not have the characteristics required for efficient

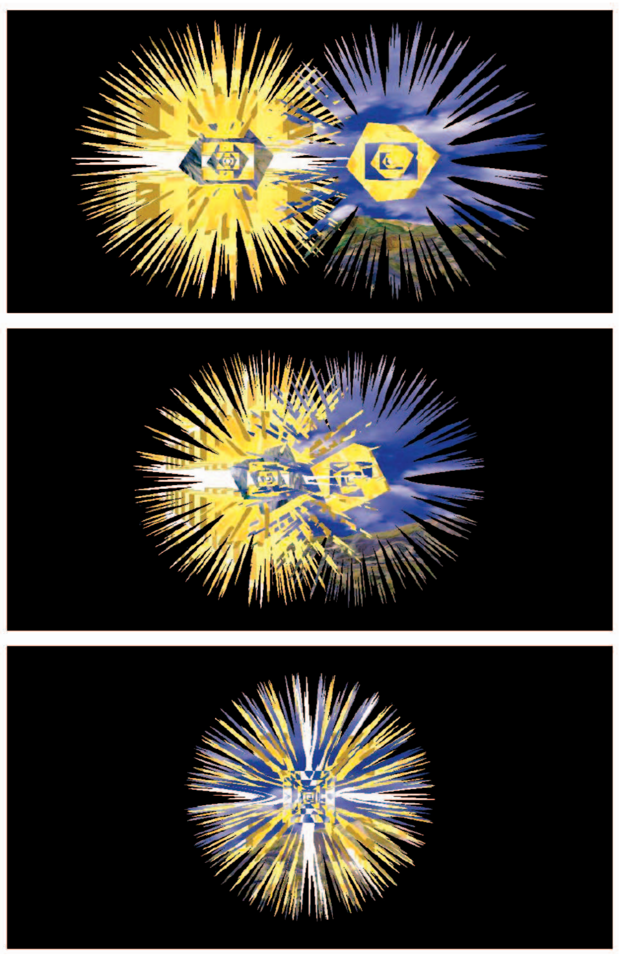


Fig. 12. This series of images shows that an object with very high geometric detail can easily be used as a portal in our complex portal rendering system. This is because the visibility determination using our algorithm works even when complex portals intersect. These images are taken from a demonstration program in which the portals are constantly animated. A geometric solution would be very computationally intensive.

rendering using conventional geometric techniques. Therefore, we developed an algorithm that uses fragment culling to correctly render complex portals. The performance of our algorithm is mostly invariant to the geometric complexity of the portals. Rather, its performance scales linearly with the number of fragments generated by portal and cell rasterisation. Our current implementation runs at interactive rates on consumer-level hardware using OpenGL.

Our work has many potential application areas in which dynamic visual links between sets of data would be useful. Application domains include video games and interactive virtual environments, scientific visualization, and network distributed scenes. Our future directions involve investigating how to apply our work to such domains.

A video game could allow players to place portals to other areas that they would like to monitor. Using our system, the portal could be a dynamic, animated 3D model. The animation of the model could communicate information to the player, such as when an event occurs in the cell the portal links to. For example, the portal could deform erratically if a monster appears in its destination cell.

Visualization applications could benefit from complex portals. For example, a program could provide different portal rasterization functions for the same data set. Users could expose different characteristics of the data set by creating complex portals that use the corresponding rasterization functions. They could create a dynamic toolkit of transformed and shaped portals to provide meaningful views of the features they wish to interrogate. For example, a medical imaging program could provide portals that rasterize different aspects of scanned data. A doctor could then actively investigate many aspects simultaneously by using complex portals.

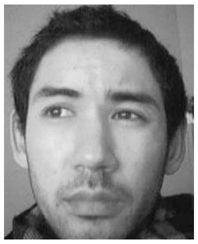
Another problem that could benefit from our work is that of connecting and visualizing distributed 3D data. By using portal rasterization functions that query servers for cell data, we could link cells that are distributed over a network. In this way, large virtual environments could be created by the efforts of many contributors who provide cell servers.

## Acknowledgments

The authors would like to thank Associate Editor Professor Anselmo Lastra and four anonymous reviewers for their extensive comments and suggestions that improved the presentation of the paper considerably. The first author's work is supported in part by an Australian Postgraduate Award and a Western Australian Interactive Virtual Environments Centre (IVEC) Scholarship. The second author's work is supported by the Western Australian Interactive Virtual Environments Centre (IVEC) and Australian Partnership in Advanced Computing (APAC).

## References

[1] Unreal engine, http://unreal.epicgames.com, 1998-2003.
[2] T. Aila, V. Miettinen, and P. Nordlund, "Delay Streams for Graphics Hardware," *ACM Trans. Graphics,* vol. 22, no. 3, pp. 792-800, 2003.
[3] J.M. Airey, J.H. Rohlf, and F.P. Brooks Jr., "Towards Image Realism with Interactive Update Rates in Complex Virtual Building Environments," *Proc. ACM Symp. Interactive 3D Graphics,* pp. 41-50, Mar. 1990.
[4] D.G. Aliaga and A.A. Lastra, "Architectural Walkthroughs Using Portal Textures," *Proc. IEEE Visualization '97,* pp. 355-362, Nov. 1997.
[5] M. Craighead, NV_occlusion_query,http://oss.sgi.com/projects/ogl-sample/registry/NV/occlusion_query.txt, Feb. 2002.
[6] P. Diefenbach, "Pipeline Rendering: Interaction and Realism through Hardware-Based Multi-Pass Rendering," PhD dissertation, Dept. of Computer Science, Univ. of Pennsylvania, 1996.
[7] C. Everitt, "Interactive Order-Independent Transparency," white paper, NVIDIA OpenGL Applications Eng., 2001.
[8] C.B. Jones, "A New Approach to the 'Hidden Line' Problem," *Computer J.,* vol. 14, no. 3, pp. 232-237, Aug. 1971.
[9] M.J. Kilgard, "NV_texture_shader," http://oss.sgi.com/projects/ogl-sample/registry/NV/texture_shader.txt, Nov. 2001.
[10] M.J. Kilgard, "NV_register_combiners," http://oss.sgi.com/projects/ogl-sample/registry/NV/register_combiners.txt, Feb. 2002.
[11] D. Luebke and C. Georges, "Portals and Mirrors: Simple, Fast Evaluation of Potentially Visible Sets," *Proc. 1995 Symp. Interactive 3D Graphics,* pp. 105-106, Apr. 1995.
[12] A. Mammen, "Transparency and Antialiasing Algorithms Implemented with the Virtual Pixel Maps Technique," *IEEE Computer Graphics and Applications,* vol. 9, no. 4, pp. 43-55, 1989.
[13] T. Möller and E. Haines, *Real-Time Rendering.* A.K. Peters, 1999.
[14] J. Rossignac and J. Wu, "Depth-Interval Buffer for Hardware-Assisted Shading from CSG: Accurate Treatment of Coincident Faces and Shadows," *Proc. Fifth Eurographics Graphics Hardware Workshop,* 1989.
[15] R.J. Rost, *OpenGL(R) Shading Language.* Boston: Addison Wesley Longman, 2004.
[16] M. Segal and K. Ashley, *The OpenGL Graphics System: A Specification (Version 1.4).* Silicon Graphics Inc., 2002.
[17] S. Teller, "Visibility Computations in Densely Occluded Polyhedral Environments," Technical Report CSD-92-708, Berkeley, Calif., 1992.
[18] T. Theoharis, G. Papaioannou, and E. Karabassi, "The Magic of the Z-Buffer: A Survey," *Proc. 2001 Int'l Conf. Central Europe Compter Graphics, Visualization, and Computer Vision (WSCG),* V. Skala, ed., pp. 379-386, 2001.
[19] J. Tyberghein, *Crystal Space 3D Engine,* http://crystal.sourceforge.net, 2002.
[20] T.F. Wiegand, "Interactive Rendering of CSG Models," *Computer Graphics Forum,* vol. 15, no. 4, pp. 249-261, 1996.
[21] L. Williams, "Casting Curved Shadows on Curved Surfaces," *Proc. Computer Graphics (SIGGRAPH '78),* vol. 12, pp. 270-274, Aug. 1978.



**Nick Lowe** received the Bachelor of Computer and Mathematical Sciences degree from the University of Western Australia. He is a PhD candidate in computer graphics and his thesis is focused on techniques and applications for arbitrary portal rendering. His interest is primarily in real-time rendering. He is a founding member of the 60Hz Real-time Rendering Group and a lead author of their Clean rendering libraries.

**Amitava Datta** received the MTech and PhD degrees in computer science from the Indian Institute of Technology, Madras. He is an associate professor in the Department of Computer Science and Software Engineering at the University of Western Australia. In 2001, he was a visiting professor at the Computer Science Institute, University of Freiburg, Germany. His research interests include parallel processing, optical computing, computer graphics, and mobile and wireless computing. He is a member of the editorial board for the *Journal of Universal Computer Science*. He is a member of the IEEE Computer Society.