Thesis no: MSCS-2015-03

Complex Transformative Portal Interaction

Markus Tillman

Faculty of Computing
Blekinge Institute of Technology
SE-371 79 Karlskrona, Sweden

This thesis is submitted to the Faculty of Computing at Blekinge Institute of Technology in partial fulfillment of the requirements for the degree of Master of Science in Computer Science. The thesis is equivalent to 20 weeks of full time studies.

Contact Information:
Author:
Markus Tillman
E-mail: matg10@student.bth.se

University advisor:
Prashant Goswami
Department of Creative Technologies

Faculty of Computing
Blekinge Institute of Technology
SE-371 79 Karlskrona, Sweden

Internet : www.bth.se
Phone : +46 455 38 50 00
Fax : +46 455 38 50 57

Abstract

Context. A portal in computer graphics is an opening which connects two spaces together. Portals can be used for occlusion culling for indoor environments or wormhole-like effects. This thesis address the latter and focus on how objects interact with such portals.
Objectives. The objectives are to provide a solution to how objects can interact with complex portals in real-time with focus on visual (and physical) correctness and also present a background to how simple and complex portals work.
Methods. A hybrid approach of a geometry and image technique is used to render portals. Intersection techniques and a technique related to constructive solid geometry is used to solve object-portal interactions. The research methodology used is implementation and simple analysis of the results is performed.
Results. The results show that the implementation of the object-portal interaction scales exponentially. In the worst case it has a complexity of $O(n^2 * m^2)$ where n and m are the number of triangles in the object and portal respectively. Increasing the number of triangles in the object shape is more costly than increasing the number of triangles in the portal shape by the same amount. The results were not compared to previous knowledge as no results have been published of other object-portal interaction methods. The rendering of portals scales linearly with the number of triangles used to represent it.
Conclusions. This thesis extends the state-of-the-art portal rendering system and adds a solution to object-portal interaction of complex shapes. It also provides a detailed background into the fundamentals of portals and their nature. The thesis is of interest to those who want object-portal interaction of both simple and complex portals used in gameplay and special effects without restriction on portal placement and shape, with the exception that portals may not have holes in their shape in the direction an intersecting object is moving.

Keywords: Portal, Interaction, Intersection, Rendering.

Acknowledgements

I thank my supervisor Prashant Goswami for providing advice and feedback on portal interactivity and academic writing which no doubt helped improve the thesis. I thank my home review panel of esteemed judges; my family, who helped improve the quality of the thesis. I also thank Anton Petersson for help with the camera frustum shrinking.

List of Figures

List of Tables

List of Algorithms

Contents

Chapter 1

Introduction

A portal in the context of computer graphics is an opening which connects two spaces together. These openings are traditionally windows, doors and other logical openings between two rooms (cells). This kind of portal is called a *non-transformative portal* as it simply connects two adjacent cells without the need of a transformation matrix [1]. The portal concept was first introduced by Jones in 1971 and later became an area of interest with the work by Airey et al. and Teller & Séquin [2–5]. Non-transformative portals have been used in games such as *Unreal Tournament* (1999, Epic Games), *Doom* (1993, id Software) as well its successors *Quake* (1996, id Software) and *Quake 2* (1997, id Software). They have seen much attention as they speed up the rendering of indoor environments as only cells and their contents visible through the portals need to be rendered [2–12]. Figure 1.1 illustrates the use of non-transformative portals in an indoor environment of cells.

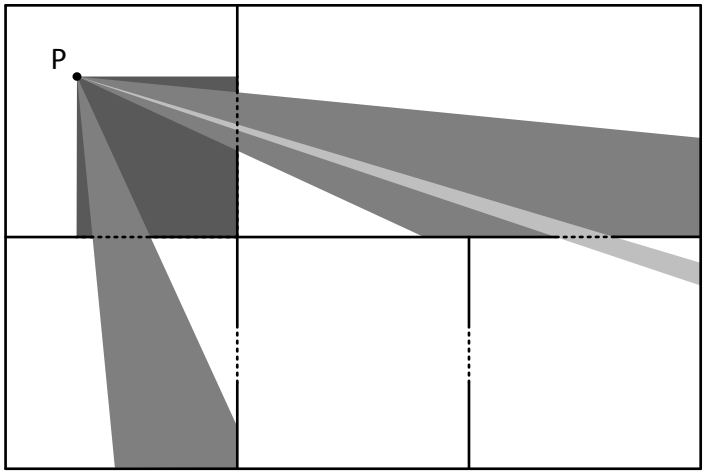


Figure 1.1: Indoor environment of cells connected by non-transformative portals marked in dotted lines. P is the viewing position with visibility from P through the portals to other cells shown in gray.

A *transformative* portal is a portal coupled with a transformation matrix [1]. This matrix allows the portal to connect *any* two spaces of different scale and also provides rotation between the two spaces. Unlike non-transformative portals, transformative portals may exist anywhere and not only on the border between two adjacent cells. It can therefore be used as a mirror for instance [1,5,8], [13, p. 670]. Physically impossible worlds created by the Dutch artist M.C. Escher (1898-1972) can be created virtually using these transformative portals [14]. A game that uses the scaling provided by the transformation matrix is in development by Pillow Castle[1]. Other games that use transformative portals are *Prey* (2006, Take-Two Interactive Software), *Portal 1* (2007, Valve Corporation) and its sequel, *Portal 2* (2011, Valve Corporation). Little academic research has focused on transformative portals [1,14,15]. Figure 1.2 shows an example of a transformative portal which connects a location in a cell with another location in another cell.

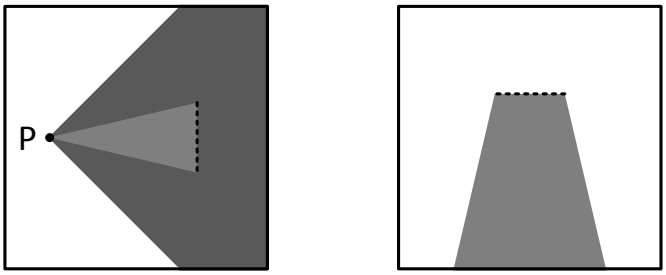


Figure 1.2: Example of a transformative portal connecting two cells. The portal and its boundary at the connected space is marked in dotted lines. P is the position of the viewer. Visibility from P is shown in gray. The visibility from P to the portal in the left cell continues through to its destination in the right cell using translation and rotation.

A portal is considered *simple* if its shape is convex[2] and planar [1]. Opposed to a simple portal, a *complex* portal's shape is non-convex and/or non-planar, which means it may even have a volume [1]. This volume can be considered as a third, intermediary space. Portal-object interaction involves objects and portals intersecting each other and cutting (slicing) the objects against the surfaces and borders of the portals as well as moving the objects through the portals to the space the portals are connected to. Portal interaction has never been done with complex transformative portals except for an unpublished paper by Vedel-Larsen [16]. Portal interaction of simple transformative portals are only present in games and are limited to specific locations like walls, or that connected portals are

[1] `http://www.pillowcastlegames.com/`

[2] A polygon is convex if all its interior angles are 180°or less, i.e., a line between any two points on the surface of the polygon is fully inside the polygon.

limited to exist in different rooms where they are not visible from the same point of view.

The aim of the thesis is to extend the state-of-the-art portal rendering system with transformative, portal-object interaction without restriction on portal shape and location which can run in real-time. The goal is to provide an object-portal interaction method that is visually (and physically) correct. Secondary goals include providing a detailed background to portals and how they work as well as the difference between simple & complex portals. As such, the following research questions are defined:

1. Can complex portal-object interaction be achieved in *real-time*?
2. Is the implemented object-portal interaction method *visually correct*?

In this thesis *real-time* is defined as the ability to execute under $16\frac{2}{3}$ ms, or above 60 frames per second. For object-portal interaction to be considered *visually correct*, an object shall be able to travel through a portal seamlessly without clipping issues and always be visible when looking into the portal it travels through.

The portal rendering system is a hybrid of a geometry and an image technique which conservatively culls objects invisible through portals and renders the portals using a stencil and a dual-depth buffer. The developed solution uses a variety of techniques to solve object-portal interaction. Triangle and oriented bounding box intersection tests are used to determine if an object intersects a portal. A technique related to constructive solid geometry is used to cut an object against a portal's surface and border, despite the object and/or portal having no volumes. The research methodology used in this thesis is implementation, and simple quantitative and qualitative analyses of the results are performed.

Results show that the implementation of the object-portal interaction scales exponentially. In the worst case it has a complexity of $O(n^2 * m^2)$ where n and m are the number of triangles in the object and portal respectively. Increasing the number of triangles in the object shape is more costly than increasing the number of triangles in the portal shape by the same amount. The results were not compared to previous knowledge as no results have been published of other object-portal interaction methods. The rendering of portals is inexpensive in comparison and scales linearly with the number of triangles used to represent it.

A method for objects interacting with complex portals is presented that allows objects to move through complex portals and be cut (sliced) against their surfaces and borders. The state-of-the-art portal rendering system is extended to discard portals which lie between the camera's near plane and the surfaces of other portals and also discard portals which lie behind already rendered portals. A background to how portals work and how simple and complex portals differ is presented. The thesis is of interest to those who want object-portal interaction of both simple and complex portals used in gameplay and special effects without restriction on portal placement and shape with the exception that portals may not have holes in their shape in the direction an intersecting object is moving.

The rest of the thesis is structured as follows: The next chapter discusses related work. Chapter 3 presents the details of the method to portal-object interaction. The performance of the implementation is presented in chapter 4 and an analysis and discussion of the results are presented in chapter 5 and chapter 6. Finally, the thesis ends with conclusions and future work in chapter 7.

Chapter 2

Related Work

Jones and many others use a geometry-based technique to render portals [2–8, 14]. Clipping issues occur when a portal intersects itself or other portals using geometry-based techniques. This is even more common for transformative portals due to the increased geometric clipping they involve [1, 8]. These kinds of techniques therefore usually limit the complexity of the portals to simple convex, planar shapes to achieve fast, correct clipping and efficient rendering [1, 17].

Aliaga & Lastra used a texturing technique to render portals [6]. Their technique is a special case of impostors[1] [18]. They first pre-render images where the viewer looks into the portals from different viewpoints and angles. During rendering they use the images that were generated closest to the current view point and use image warping to interpolate between them to provide a smooth transition to texture the portals with. This interpolated image is smoothly transitioned into 3D geometry seen through the portal as a viewer gets closer and walks through it. Texturing techniques have been used in render engines such as the Unreal and Crystal Space engines [19, 20].

In 2005, Lowe and Datta presented an image-based rendering technique that made the rendering of complex portals considerably easier [1]. It addresses the issues of correct clipping that geometry-based techniques suffer from by performing the portal visibility test in screen space. They perform no culling which means that all scene data is sent through the graphics pipeline for rendering. They use a stencil buffer to distinguish pixels that belong to a portal as well as the level of portal recursion. After they move the camera to the space the portal connects to, they use the first (near-)depth buffer to exclude any pixels that are between the camera's near plane (0 depth) and any portal pixels. The second (far-)depth buffer works like the normal depth buffer and works opposite to the near-depth buffer. As the visibility test is performed in screen space, the technique does not have the constraint that any geometry for the portals is needed; the portals can be anything that can be rendered into screen space. The technique also allows cells to be concave unlike previous work [1, 8].

Petersson has presented an image-space and geometry hybrid technique with

[1] An impostor is a billboard (a 2D image that is always facing the viewer) that subtly replaces a 3D object.

transformative portals [15]. He focused on speeding up the technique developed by Lowe and Datta while maintaining visual quality. His technique adds the constraint that every object, including portals, has a bounding volume. He used view frustum culling with these bounding volumes to exclude objects and portals that are guaranteed to not contribute to the final image. The culling was recursively repeated for each visible portal after the view frustum had been shrunk to fit their bounding volumes. The objects that were unculled were sent to the graphics card for the final visibility test performed by a dual depth buffer.

In previous work, a *cell-and-portal graph (CPG)* is created by partitioning the scene into cells and portals [1, 2, 5–9, 15]. The graph represents the adjacency and visibility between cells through connecting portals. Exploiting these relationships speeds up the rendering of indoor scenes. A node in the graph is a cell which contains an arbitrary number of portals that each connect to a cell. Automatic partitioning and CPG generation has been the focus of previous papers [9–11, 21]. Automatically generating the CPG can however result in undesirable and missing portals which requires a designer's attention [21]. Traversing the CPG creates a *potentially visible set (PVS)* of the scene. The PVS contains all the potentially visible cells and the objects they contain. It can be pruned by culling the cells and portals against the view frustum, shrinking the frustum to fit each visible portal, or find sightlines through portal sequences from the root node (cell) [2, 5, 6, 8, 15]. Due to the PVS's relative simplicity over exact visibility calculations, it has become popular when rendering portals [4, 5, 7, 8, 15].

Chapter 3

Method

3.1 Portals, how do they work?

Non-transformative portals are openings between two cells and are much simpler than transformative portals [1]. They only hold a (non-visible) shape, usually convex like a simple rectangle. A visibility frustum can be created from the current point of view by using the boundaries of their shapes [5, 15]. This means that anything that is outside this frustum, i.e., occluded by the surrounding walls of the cell the portal is in, can be safely be omitted from rendering as they are guaranteed to be invisible.

Transformative portals are more complex than non-transformative portals and will be discussed in the following section.

3.1.1 Transformative portals

To connect a portal to a location a transformation matrix is used, hence the name *transformative* portal [1]. This location can be anywhere, even the same location as the portal itself. If there is no scaling nor rotation in the transformation matrix and the location is the same as the portal position, it is essentially a *non-transformative* portal. Transformative portals are similar to the concept of wormholes; light and objects can travel through them from one point in space to another using a shortcut in space. However, they are simplified to be made up of triangles to be useful in computer graphics. A transformative portal in this thesis is defined as geometry coupled with a world matrix for placing the geometry in world space and a transformation matrix to move objects to space the portal connects to. The position and space where the portal exists will henceforth be referred to as the source position and source environment. The position and space that the portal connects to will be called the destination position and the destination environment.

A transformative portal can enlarge or shrink objects that move through it by applying the scale in its transformation matrix to the object. An illustration of an objects travels through a transformative portal is shown in fig. 3.2.

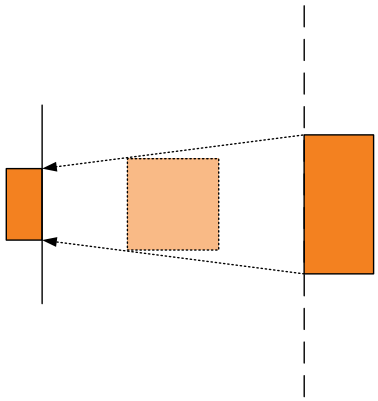


Figure 3.1: An illustration of how transformative portals work with objects. Portal on the left is marked with vertical solid black lines and connects to the space to its right using a scale of two. Its imaginary border at the destination environment is marked in vertical dotted black lines. The orange square is moving from right to left and traveling through the portal, existing in both spaces at the same time. The part that has traveled through the portal is shrunk down by half of the square's original size.

Similarly, a transformative portal can also make the connected space look larger or smaller by conceptually spreading out or focusing incoming light from the destination environment, shown in fig. 3.1.

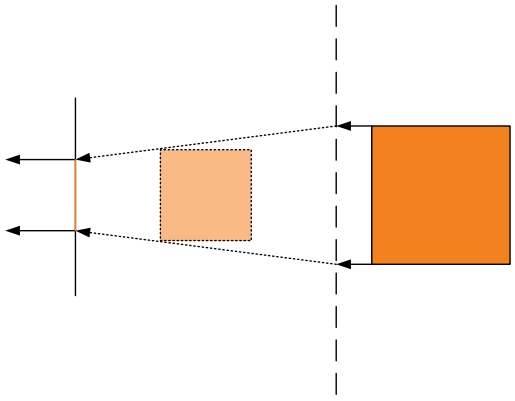


Figure 3.2: An illustration of how transformative portals work with light. Portal on the left is marked with vertical solid black lines and connects to the space to its right using a scale of two. Its imaginary border at the destination environment is marked in vertical dotted black lines. Light bounces of the orange square at the destination and hits the imaginary border of the portal which is then moved to its source environment after it focuses the light.

An observer standing in front of the portal in fig. 3.2 and looking to the right will see the square through the portal's surface shrunk down to half its *actual* size. If an observer stands between the portal and its destination facing toward the square, the observer receives no light that bounce of the cube as it is intercepted by the portal.

All light that hits the shape of the portal at the *destination* environment is transported through the portal to the source environment, but not the other way. This means it should be black in the shape of the portal at the destination environment. This is defined as a *one-way* portal as it only allows light to travel *one way*. An example of an implemented simple, one-way transformative portal is shown in fig. 3.3.

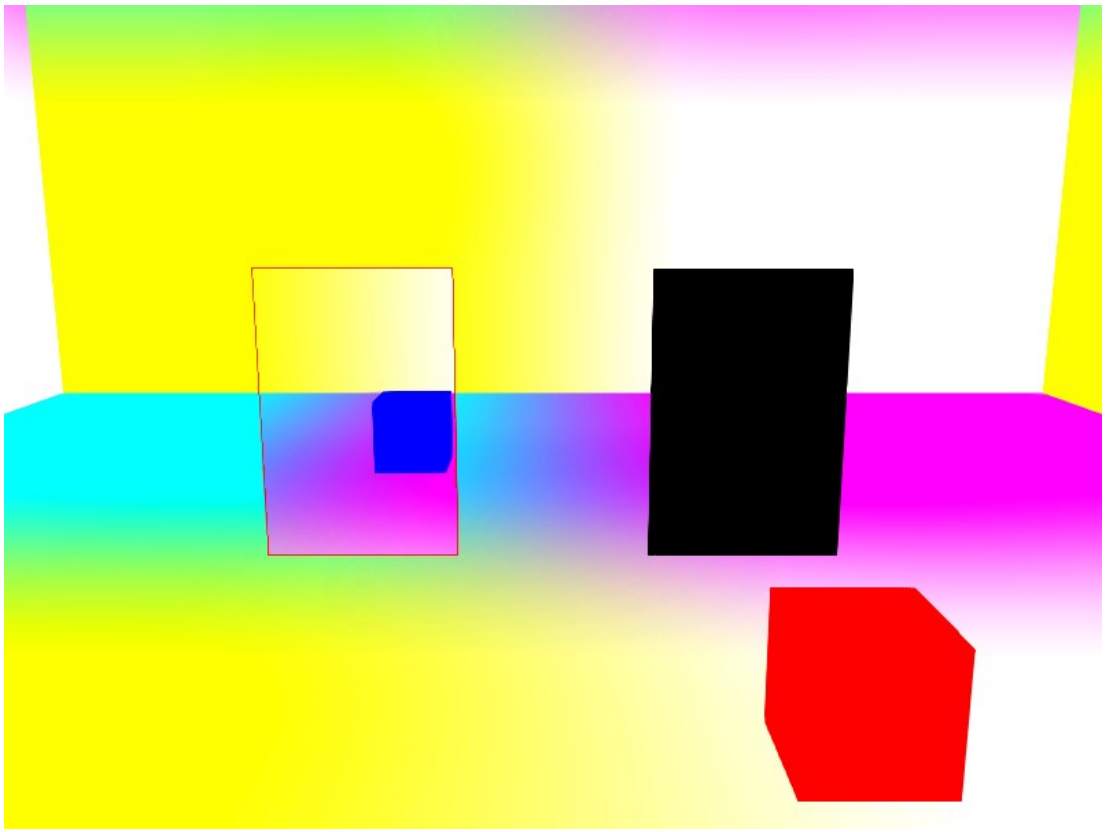


Figure 3.3: An example of a simple one-way transformative portal. The portal is located at the left in the image. The portal shape's boundary is marked in red lines. The portal connects to the space between the two cubes to its right. The shape of the portal at the destination is marked in black. The blue (top) cube positioned at the right behind the black portal shape can be seen through the portal to the left.

Now, imagine walking around the portal, the view would then become like in fig. 3.4.

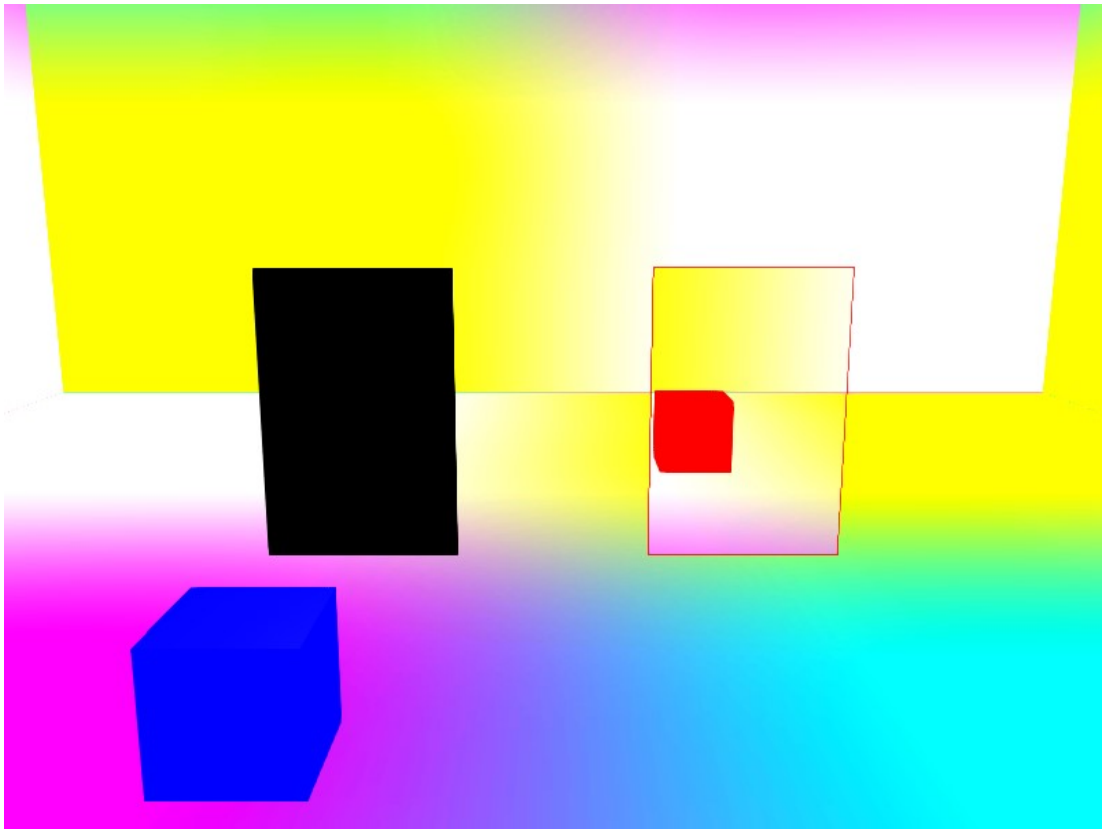


Figure 3.4: An example of a simple one-way transformative portal. The portal is located at the right in the image. The portal shape's boundary is marked in red lines. The portal connects to the space between the two cubes to its left. The shape of the portal at the destination is marked in black. The red (top) cube positioned at the left behind the black portal shape can be seen through the portal to the right.

What will happen to the *source* light that hits the source portal in the source environment? Since a one-way portal is defined to transport only destination light to the source environment, what should happen to the light? One of the following three scenarios may be an acceptable solution:

1. The light is absorbed and the portal looks as in fig. 3.3 and fig. 3.4.
2. The light bounces off the portal and is effectively a two-way mirror where the source and destination light is mixed.
3. The light hits the portal but stays in the source environment without any changes. Note that the source and destination light will mix as in scenario 2.

Figure 3.5 shows how each scenario would look like.

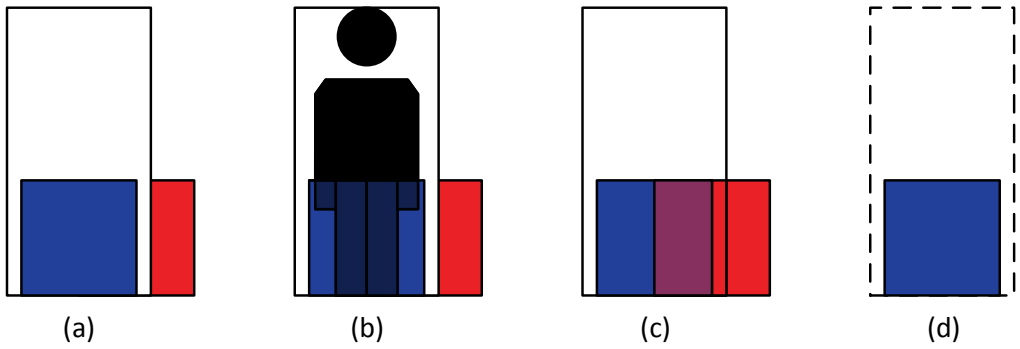


Figure 3.5: Different scenarios of how source light interacts with one-way portals. Three portals connect to a space where there is a blue square (d). The imaginary boundary of the portals at the destination is marked in black dotted lines. Behind each portal is a red square. The viewer is modeled as a black stick figure standing in front of the second portal. (a) Source light is absorbed by the portal. (b) Source light bounces off the portal. (c) Source light hits the portal but stays in the source environment.

The portal in fig. 3.3 and fig. 3.4 is *one-way*. This means that there is no second portal at the destination environment that connects back to the first portal. Using two portals connecting back to each other creates a loop and behaves more like a real connection between two arbitrary points in space. Actually, any number of portals can be connected to each other. For instance, portal one at point A connects to point B, portal two at point B connects to point C and portal three at point C connects back to point A. A portal system consisting of more than two portals can be viewed as a circular singly linked list. A *two-way* portal system of the example in fig. 3.3 is shown in fig. 3.6.

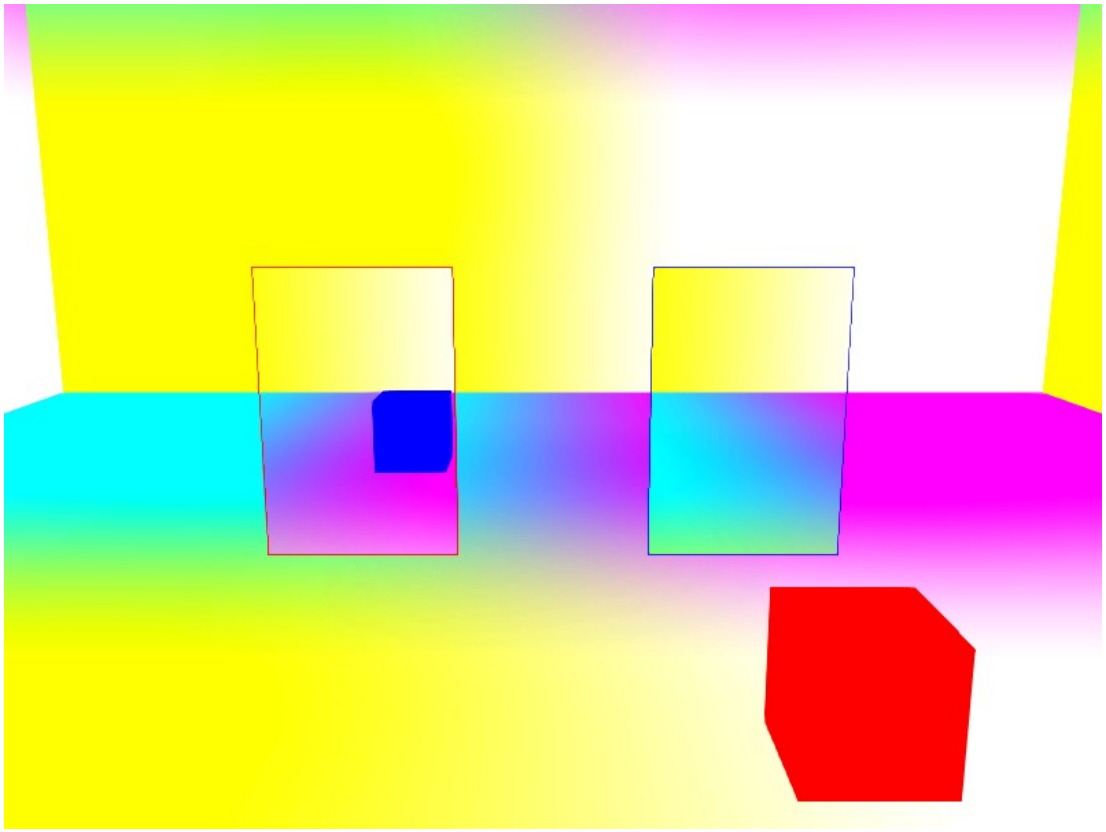


Figure 3.6: An example of a simple two-way transformative portal. The first portal is located at the left in the image with its boundary marked in red lines and the second portal is located at the right in the image with its boundary marked in blue lines. The blue cube is located behind the portal to the right and can be seen through the portal to the left.

The transformation matrix is used to translate, rotate and scale the destination environment, or more accurately, the camera's view matrix. No limitation is put on the translation and rotation of the transformation matrix but one should be put on scaling if a two-way portal system is used. For instance, if the first portal is twice as large as the second portal in all dimensions, the destination environment, as viewed from the first portal, should be scaled up by a factor of two. Similarly, the size of any object traveling through the portal system should be shrunk by half or enlarged by a factor of two, depending on which portal it entered.

3.1.2 Object-Portal Interaction

An object that intersects the surface of a portal without a volume exists in two spaces; the source environment from which the object came from and the desti-

nation environment to which a part of the object has been teleported to. The piece of the object that is on the other side of the source portal in the direction the object is traveling in the source environment is the part that is teleported to the destination environment. The rest of the object stays in the source environment. An object that intersects the *border* of a portal is sliced by it in the same direction, leaving one piece in the source environment and another piece in the destination environment after the object is done interacting with the portal. Figure 3.7 illustrates what should happen when objects intersect a simple portal.

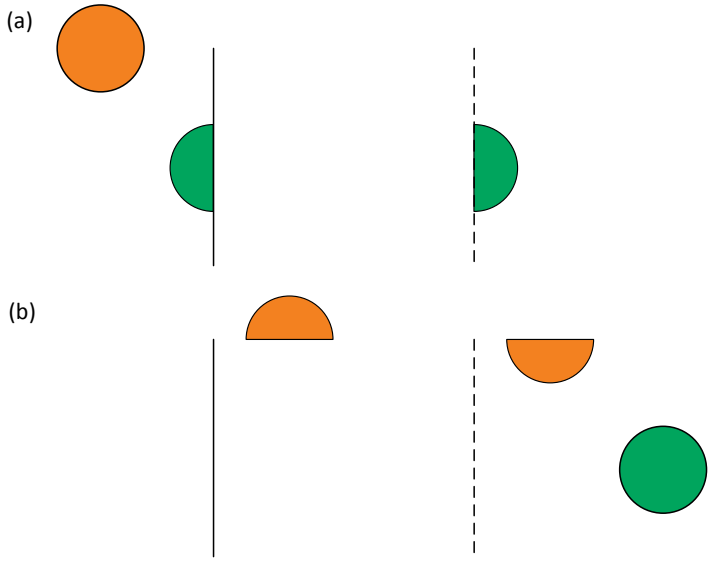


Figure 3.7: Two spheres interacting with a simple one-way portal. The portal is marked as a solid black line and its imaginary boundary at the destination environment is marked in black dotted lines. The two spheres are moving from left to right. (a) The bottom sphere (green) is cut against the portals surface and exists in the source and destination environments. (b) The bottom sphere (green) has traveled through the portal to the destination environment without being sliced. The top sphere (orange) has been sliced at the portal border in the direction it is moving.

A way to avoid slicing the objects against the portal *borders* is to put a frame around the portal that they can collide with. This makes sure that all parts of an object is within the borders of a portal when it travels through it. No slicing is necessary if the portal is limited to exist on a surface, like a wall for instance. Another method can be used for solid portals which have a volume; an object can be considered tethered to the source environment as long as *any* part of it is outside the volume of the portal, eliminating the need to slice the object. Figure 3.8 illustrates how this would work.

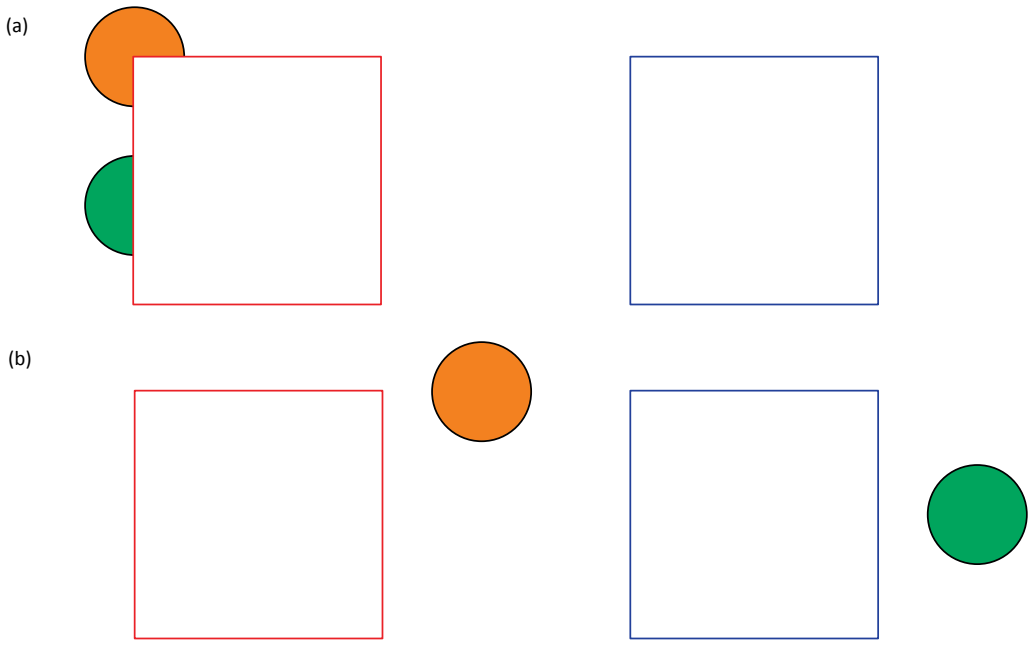


Figure 3.8: Two spheres interacting with a volumetric portal. The (red) square at the left is the source portal and the (blue) square at the right is the destination. The spheres are tethered to the source environment as long as any part of them remains outside the volume. The two spheres are moving from left to right. (a) The spheres are interacting with the source portal. (b) The spheres are done interacting with the source portal. The top sphere (orange) remains in the source environment as a part of it was always outside the portal's volume whereas the bottom sphere (green) went inside the portal and was moved to the destination environment.

This system works fine as long as the portal is *one-way*. If an object is inside a *two-way* portal it means it is inside both the source and destination portals at the same time. If the object were to be teleported to the destination after entering the source portal, it would be teleported back and forth between the two until the object has traveled past the volumes of the portals, leaving the object in either the source or destination environment without knowing which. A way to solve this problem is to teleport the object as soon it intersects the boundary of the source portal *after* it has been determined it was inside its volume. Note that for all complex portals the methods of using a frame or slicing the objects can be used but the special case of a two-way portal with a volume still needs to be handled to avoid constant teleporting.

3.2 Rendering system

The following computer graphic elements will be used; color, depth, stencil buffer, view frustum. The color buffer is where image data of the scene is written to after it has passed depth and stencil tests. The depth of an object is written to the depth buffer if it passes the depth test. Likewise, the stencil value is subjected to an operation if the stencil test is accepted. The stencil buffer is used for special effects like compositing (portals, mirrors), decaling, silhouettes and shadows [1]. For more information about buffers, see appendix A. The view frustum is the space in which objects may be visible. This region of space looks like that of a pyramid with a truncated top aimed toward the camera position [13, p. 19].

The rendering system is based on the system developed by Lowe and Datta [1]. Their system is extended to handle outdoor environments as well as a near-depth value check to discard portals that are closer than the portal they are visible through. The culling system is self-implemented with some hints from Petersson's solution [15]. Figure 3.9 illustrates the flow of the portal rendering system.

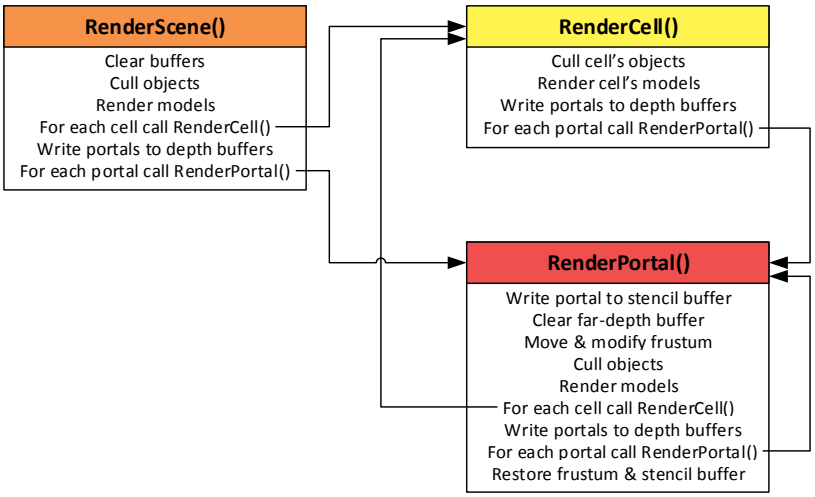


Figure 3.9: Flow of the portal rendering system. First all buffers are cleared. Objects are then culled against the camera frustum. All visible "normal" geometry is then rendered before each visible cell, and lastly any visible portal. The cell's contents are culled and rendered, including the cell itself. Any portals in the cell are then rendered. First, a portal is written to the stencil buffer. The far-depth buffer is then cleared at the corresponding locations in the stencil buffer. The camera frustum is shrunk to fit the portal boundary and then moved to the portal's destination where culling is performed. Objects at the destination visible through the current portal are rendered. The camera frustum and stencil buffer are then restored to their previous states.

[1] `https://msdn.microsoft.com/en-us/library/windows/desktop/bb205074(v=vs.85).aspx`

Two depth buffers are used to correctly clip geometry at the destination environment against the portal's surface. Portal depth values are written to the near-depth buffer and is necessary to clip geometry at the destination environment between the camera's near plane and portal surface. Fig. 3.10 illustrates the need for a near-depth buffer.

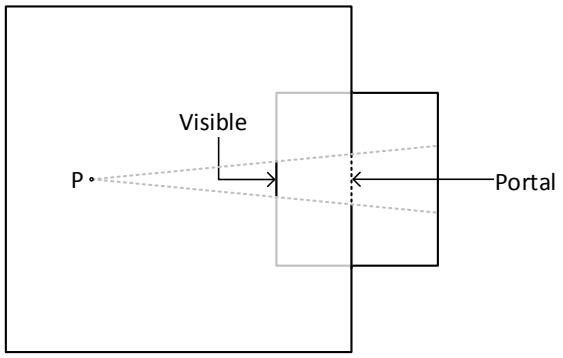


Figure 3.10: A portal connecting two cells. The portal connects the border of the cell to the left with the middle of the cell to the right. P is the camera position and its visibility through the portal is showed in gray dotted lines. Geometry from the cell to the right is incorrectly rendered in front of the portal.

The buffers are cleared at the beginning of each frame. The color buffer is cleared to a user-specified color and the far-depth buffer values are set to 1. The near-depth buffer is cleared to 0 as it works opposite to the far-depth buffer: Any depth values less than the ones in the near-depth buffer are discarded when rendering objects, including portals, but only when they are rendered through another portal. The *recursionDepth* variable is set to 0. Objects of the scene are then culled against the camera's frustum. Models are rendered first to ensure correct results. Cells are rendered after models. If the camera is inside a cell, it is rendered first. If inside a cell, the cell's objects are culled against the camera frustum. See algorithm 3.2.1 for how the scene is rendered.

Before rendering a cell and its models, the stencil reference value is set to *recursionDepth*, stencil test to "equal", stencil pass operation to "keep" and depth test to "less" with writes to the far-depth buffer enabled. If the depth of a rendered pixel of a cell or model is less than the corresponding value in the near-depth buffer, the pixel is discarded and therefore not written to the color or far-depth buffers. After all non-portal models have been rendered, all portals in the cell are rendered to the far and near-depth buffers, unlike Lowe & Datta who first write to the far-depth buffer and then copy it to the near-depth buffer [1]. While writing to the near- and far-depth buffers, the near-depth values from the previous set of rendered portals is used to discard pixels depending on the value of

Algorithm 3.2.1 Render scene

```
function RenderScene
    for pixel in buffers do
        colorBuffer[pixel] ← color
        farDepthBuffer[pixel] ← 1
        stencilBuffer[pixel] ← 0
        nearDepthBuffer[pixel] ← 0
    unculledObjects ← CullObjects(allObjects, camera.cullFrustum))
    for model in unculledObjects do
        RenderModel(model)
    // Start with the cell the camera is in, if any.
    for cell in unculledObjects do
        RenderCell(cell, 0)
    RenderPortalsToFarAndNearDepthBuffers(unculledObjects, recursionDepth)
    for portal in unculledObjects do
        RenderPortal(portal, 0)
```

recursionDepth. Portal pixels which are closer than the portal that they are visible through (when *renderDepth* is greater than zero) are discarded. Pixels which are farther away than already rendered pixels for portals that are *not* rendered through other portals (when *renderDepth* is zero) are also discarded. The first discard is necessary to avoid rendering portals which lie between the camera's near plane and the portal surface they are rendered through. The second discard is to prevent portals from being rendered on top of other portals when they are occluded by them. See algorithm 3.2.2 for pseudocode on how to render cells.

Algorithm 3.2.2 Render cell

```
1: function RenderCell(cell, recursionDepth)
2:     unculledObjects ← CullObjects(cell.objects, camera.cullFrustum)
3:     RenderModel(cell)
4:     for model in unculledObjects do
5:         RenderModel(model)
6:     RenderPortalsToFarAndNearDepthBuffers(unculledObjects, recursionDepth)
7:     for portal in unculledObjects do
8:         RenderPortal(portal, recursionDepth)
```

After writing all portals in the current space to the near and far-depth buffers, a portal is rendered to the stencil buffer by setting the stencil reference value to *recursionDepth*, stencil test to "equal", stencil pass operation to "increment" and depth test to "equal" with depth writes disabled. If zero pixels were written to the stencil buffer then return from the function and render the next portal. This can

happen if the portal is really far away, an object is occluding it, or it is invisible. Before the destination environment can be rendered, the far-depth buffer must be cleared at the portal pixels (where stencil value now is *recursionDepth* + 1). This is done by first incrementing the *recursionDepth* value by one and then setting the stencil reference value to the new *recursionDepth* value, stencil test to "equal", stencil pass operation to "keep" and depth test to "always" with writes to the far-depth buffer enabled and then render a full-screen triangle with a depth of 1 in normalized device coordinates (NDC). Figure 3.11 illustrates why this is necessary.

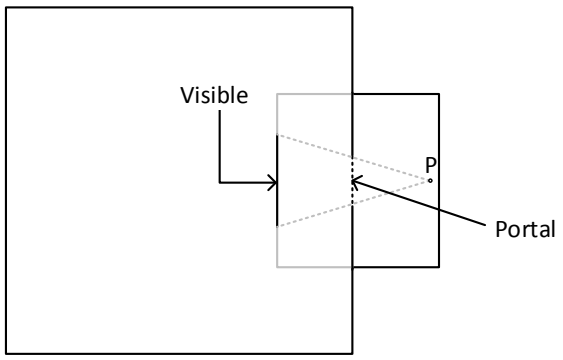


Figure 3.11: A portal connecting two cells. The portal connects the border of the cell to the left with the middle of the cell to the right. P is the camera position and its visibility through the portal is showed in gray dotted lines. Geometry from the cell to the right is incorrectly rendered through the portal.

The camera's state is saved as it will need to be restored after a portal has been rendered. A new frustum is calculated after the far-depth buffer has been cleared at the portal pixels. This new frustum is *only* used for culling, and not for rendering. The camera frustum is shrunk to fit a portal's bounding volume by modifying the projection matrix's field of view (top-down angle in radian), aspect ratio and near & far clip distances. The camera's forward, and up & right vectors consequently, are also modified. Note that if a portal is planar and aligned to the camera's near plane, the camera's near plane can be moved up to the portal's surface. If it is unaligned but planar, oblique view frustum could be used to move the near plane to the portal [22]. The way of calculating a new frustum presented in this thesis assumes that portal surfaces are irregular. Algorithm 3.2.3 shows pseudocode for how a portal is rendered.

To calculate a new forward vector and near clip distance, the points of the bounding volume in world space is transformed to NDC using the camera's view-projection matrix. 2D minimum (min) and maximum (max) points are generated

Algorithm 3.2.3 Render portal

```
1:  function RenderPortal(portal, recursionDepth)
2:      if recursionDepth > 255 then return recursionDepth
3:      RenderPortalToStencilBuffer(recursionDepth)
4:      nrOfPixelsDrawn ← query.data
5:      if nrOfPixelsDrawn == 0 then return recursionDepth
6:      recursionDepth ← recursionDepth + 1
7:      for pixel in stencilBuffer do
8:          if stencilBuffer[pixel] == recursionDepth then
9:              farDepthBuffer[pixel] ← 1
10:     saveState ← camera
11:     camera.cullFrustum ← CalculateNewCullFrustum(portal.boundingVolume, camera)
12:     camera ← Transform(portal.position, portal.transformationMatrix)
13:     unculledObjects ← CullObjects(allObjects, camera.cullFrustum)
14:     for model in unculledObjects do
15:         RenderModel(model)
16:     // Start with the cell the camera is in, if any.
17:     for cell in unculledObjects do
18:         RenderCell(cell, recursionDepthl)
19:     RenderPortalsToFarAndNearDepthBuffers(unculledObjects, recursionDepth)
20:     for portal in unculledObjects do
21:         RenderPortal(portal, recursionDepth)
22:     camera ← saveState
23:     for pixel in stencilBuffer do
24:         if stencilBuffer[pixel] == recursionDepth then
25:             stencilBuffer[pixel] ← stencilBuffer[pixel] − 1
```

from the transformed points. The z value of *min* & *max* are set to the smallest z value of any of the transformed points. The x & y values of *min* & *max* are clamped to between -1 and 1 to keep the new frustum size equal or smaller than the original. The z value is clamped to 0[2] if it is negative. Top-left and bottom-right corners are then derived from *min* (bottom-left) & *max* (top-right) corners. These two points will be used to calculate aspect ratio, field of view and far clip distance later. See fig. 3.12 for an illustration how the points are calculated.

[2]Direct3D use a range between 0 and 1 for depth values.

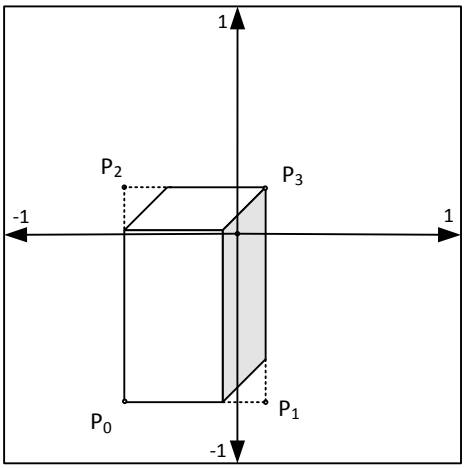


Figure 3.12: Calculating corner points of a bounding volume in normalized device coordinates. P_0 and P_3 are the minimum and maximum corners of the bounding volume and P_1 and P_2 are the bottom-right and top-left corners derived from P_0 and P_3.

The points are then transformed back to world space with the inverse of the matrix used earlier. The new forward vector is then calculated as follows:

$$F_{new} = P\hat{\vec{M}} * |F| \tag{3.1}$$

Where F_{new} is the new forward vector, P is the camera position, M is the middle point between min & max in world space and F is the camera's current forward vector. An illustration of these are shown in fig. 3.13.

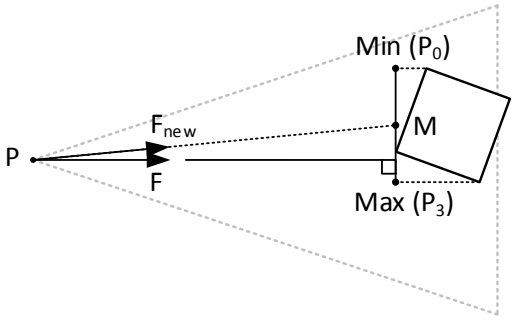


Figure 3.13: Calculating a new forward vector. Top-down view of a camera frustum in world space. P is the position of the camera and F is its forward vector. The camera frustum is showed in dotted gray lines. Min and Max are the minimum and maximum corner points of the square in relation to F. M is the middle point between Min and Max. The new forward vector F_{new} is between P and M.

The new near clip distance is calculated as shown in eq. (3.2).

$$Near_{new} = \vec{PC} \cdot \hat{\vec{PM}} - e - \epsilon \tag{3.2}$$

Where $Near_{new}$ is the new near clip distance, P is the camera position, C is the bounding volume's center position, e is the "extent" of the bounding box to the normalized $\vec{PC}$ vector. ϵ is a small value to prevent floating point rounding errors when rendering the destination environment on the portal surface. The "extent" of an oriented bounding box (OBB) to a normalized vector is calculated as such:

$$e = \sum_{i \in u,v,w} h_i |\hat{u}_i * \hat{v}| \tag{3.3}$$

Where i is one of the three dimensions of the OBB, $\hat{u_i}$ is one of the unit vectors of the OBB and h_i is its half length in dimension i. $\hat{v}$ is the normalized direction in which the extent apply to. Figure 3.14 illustrates the components used to calculate the new near clip distance.

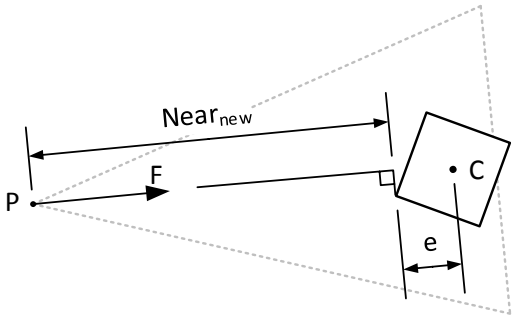


Figure 3.14: Calculating a new near clip distance. Top-down view of a camera frustum in world space with its frustum shown in dotted gray lines. P is the position of the camera and F is its forward vector. e is the "extent" of the square from C to $\hat{F}$. $Near_{new}$ is the new near clip distance and the shortest parallel distance from P to the square.

To calculate a new field of view, aspect ratio and far clip distance, the corner points calculated earlier (see fig. 3.12) are transformed back to NDC using the *new* forward vector and near clip distance. New *min* and *max* points are calculated from the four transformed points. Due to non-orthogonal projection and/or due to the shape of the bounding volume, the distance between the origin and *min* and *max* might differ. This will lead to a frustum that is not covering the whole bounding volume. This is fixed by setting *min* and *max* to each others negative maximum value so that they mirror each other. In fig. 3.15, the distance from *min* to origin is greater than the distance from *max* to origin in both X and Y axes. *max* is set to *min*'s negative values.

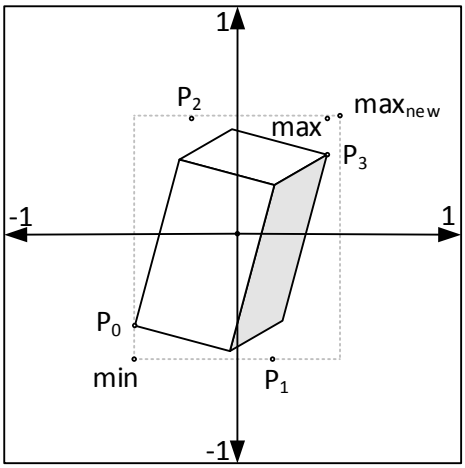


Figure 3.15: Re-calculating corner points of a bounding volume in normalized device coordinates using a new forward vector. *min* is the minimum values and *max* is the maximum x & y values of the points P_0, P_1, P_2 and P_3. The new frustum border is based on *min* & max_{new} and is shown in gray dotted lines.

The *min* & *max* distances in the Z axis are set to 0 which projects them onto the new near plane. New top-left and bottom-right corners are derived from *min* & *max*. These four corner points are then transformed to world space. The new aspect ratio can then be calculated according to eq. (3.4)

$$AR_{new} = |tr_{near_{new}} - tl_{near_{new}}| \,/\, |tl_{near_{new}} - bl_{near_{new}}| \tag{3.4}$$

Where AR_{new} is the new aspect ratio, $tr_{near_{new}}$, $tl_{near_{new}}$, and $bl_{near_{new}}$ are the top right, top left, and bottom left corners of the new near plane in world space. The new field of view is calculated as follows:

$$FOVY_{new} = 2tan^{-1}(|tl_{near_{new}} - bl_{near_{new}}| \,/\, 2Near_{new}) \tag{3.5}$$

Where $FOVY_{new}$ is the new top-down field of view of the camera frustum and $Near_{new}$ is the new near clip distance from eq. (3.2).

A new far clip distance is necessary if the original viewing distance is to be preserved as using the original far clip distance would exclude objects when culling them, see marked (red) area in fig. 3.16. The new far clip distance is the distance between the camera position and the center point on the *new* far plane. To calculate this point, first the corner points of the *old* far plane are calculated by finding the points of intersection on it from the camera position to the four corner points on the new near plane. The maximum distance between the camera position and any of these intersection points is used to calculate the bottom-left

and top-right corners on the *new* far plane. The center point is calculated by adding these two points together and dividing the result by 2. Equation (3.6) shows how the new far clip is calculated and fig. 3.16 illustrates the parts used to calculate the new far clip.

$$\begin{aligned} Far_{new} = |(p + bl_{near_{new}} - p * d + \\ p + tr_{near_{new}} - p * d) \,/\, 2 - p)); \end{aligned} \tag{3.6}$$

Where Far_{new} is the new far clip distance, $p+bl_{near_{new}}-p*d$ and $p+tr_{near_{new}}-p*d$ are the top-left and top-right corners of the new far plane. d is the maximum distance between the camera and the four corners on the *old* far plane.

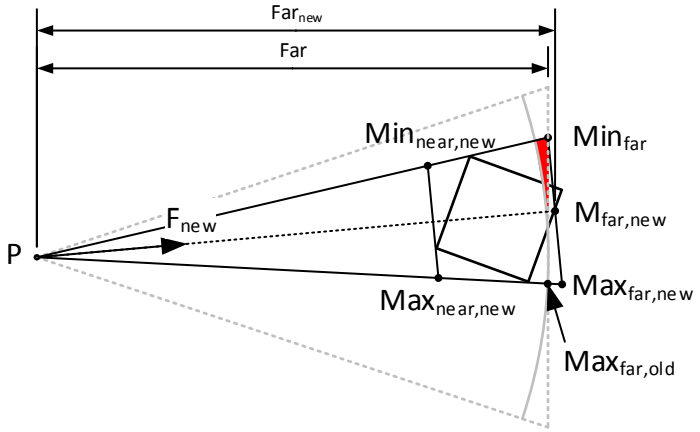


Figure 3.16: Calculating a new far clip distance. Top-down view of a camera frustum. P is the position of the camera with its *old* frustum borders showed in gray dotted lines. Far is the current far clip distance, shown as an arc in gray around P. Far_{new} is the new far clip distance, which is the length of the vector from P to $M_{far,new}$. $M_{far,new}$ is the center point on the new far plane. $Min_{near,new}$ and $Max_{near,new}$ are the bottom-left and top-right corners on the new near plane. Min_{far} is (in this example) the bottom-left corner of both the old and new far planes. $Max_{far,old}$ is the top-right corner of the old far plane and $Max_{far,new}$ is the top-right corner of the new far plane. Objects in the marked area (red) are excluded from rendering if Far is used instead of Far_{new}.

After the new frustum has been calculated, the camera is moved to the portal's destination environment and transformed by its transformation matrix. The camera position is first translated by the portal position and then multiplied by the portal's transformation matrix according to eq. (3.7).

$$P_{new} = P * T(-S) * Tr \tag{3.7}$$

Where P_{new} is the new position and P the old one. T is the translation matrix made up of negative S which is the position of the portal in the source environment. Tr is the portal's transformation matrix. The cameras right, up and forward vectors of its view matrix are then multiplied by the portal's transformation matrix as in eq. (3.8).

$$V_{new} = V(R, U, F) * Tr \tag{3.8}$$

Where V_{new} is the new view matrix and V the old one. R, U and F are the right, up, and forward vectors of the camera's view matrix. Objects at the destination environment are then culled against the new frustum and rendered using the new view matrix. Non-portal objects are rendered first for correct rendering.

The camera is then restored to its previous state and the portal is "un"-rendered from the stencil buffer after all unculled objects have been rendered. This is necessary to render other portal destinations to the correct places on the buffers. To "un"-render a portal, the stencil reference value is set to *recursionDepth*, stencil test to "equal", stencil pass operation to "decrement" and depth test to "always" with writes to the far-depth buffer disabled. A full-screen triangle with a depth of 1 in NDC is then rendered.

3.3 Implementation

The system was implemented using C++11 (Visual studio 2013 professional) with the Direct3D 11.0 API for rendering. It is used to query the number of pixels written to the stencil buffer using its query class. Since Direct3D 11.0 doesn't support a second depth buffer, the near-depth buffer was implemented as a render target view for writing and as a shader resource view for reading. CGAL's Delaunay triangulation is used to triangulate vertex lists when slicing an object against a portal [23]. Non-minimal oriented bounding boxes are used as the bounding volume of each object, including portals. Scenario 1 where source light is absorbed by the portal is used in this implementation.

3.3.1 Portals

In this implementation a transformative portal is defined as a 3D mesh coupled with a world matrix for placing it in world space, a transformation matrix to teleport objects and the camera to its destination when rendering the destination environment. Unlike normal models it has no textures nor color and its vertices consist only of position data. It also has an optional pointer to another portal to prevent infinite recursion (this is related to the similar problem of constant teleporting mentioned in section 3.1.2). To avoid checking if the camera ended up in a cell after applying the portal's transformation matrix, an optional pointer

to a cell may also be used. This pointer is set in a pre-processing step and ignored during rendering if null.

3.3.2 Rendering system

The color buffer used in this implementation supports 8 bits of unsigned normalized integer per channel, including alpha. The far-depth and stencil buffers are implemented as a normal depth-stencil buffer that uses 32 bits, 24 bits unsigned normalized float for depth and 8 bits unsigned integer for stencil values. Stencil tests and operations apply to both front and back-facing triangles. The resource of the near-depth buffer uses a 32-bit floating-point format, is written to using a render target view and read from the graphical processing unit (GPU) using a shader resource view of matching formats. After the near-depth buffer has been used as a shader resource view for reading, it is unbound from the shader to use it as a render target view for writing.

Mentioned in section 3.2, when writing to the near and far-depth buffers, the near-depth values from the previous set of rendered portals is used to discard pixels depending on the value of *recursionDepth*. The resource used for the near-depth buffer cannot be read and written to at the same time. A copy of the near-depth buffer is used to solve this. The near-depth buffer is copied after writing a set of portals to the near and far-depth buffers.

Culling of objects is done using OBB-plane intersection on each of the six planes of the camera frustum [13, pp. 755-757]. After applying a portal's transformation matrix on the camera and there's no cell destination stored in the portal, the camera's position is tested against the planes of the bounding volume of all cells to see if the camera is inside one of them, in which case it is rendered first.

3.3.3 Object-portal interaction

The object-portal interaction is divided into five steps:

1. Intersection determination between object and portal.
2. Find "container" of the portal.
3. Find intersections between object and container.
4. Separate vertices inside and outside container, add intersections, triangulate them and remove extra triangles.
5. Create objects from remaining triangles & apply portal's transformation matrix to object inside container.

Before an object can interact with a portal it must first be determined that the two intersect each other. Since triangles are the lowest primitives used, triangle

intersection is used. Bounding volume intersection is first performed for each object and portal. Triangle intersection between the triangles of the object and portal is performed if the object's bounding volume intersects or is inside the bounding volume of the portal. If the two intersect each other, the object shall be sliced against the portal in the direction the object is traveling (travel-vector).

To find the needed intersections to slice the object against the portal, the "container" of the portal must first be found. This is done by first projecting the portal's vertices to the plane with the opposite travel-vector as its normal and then rotate them to the XY-plane. The 2D outer border of the vertices is then found where there is a triangle edge between them. This is done by first finding the top-most vertex and then successively go clock-wise to the next vertex of the triangles. If the current and next vertices are both part of the same triangle, and make up an outer edge of the portal, the latter point is saved. To test if the edge between the vertices make up the outer edge, the angle between it and the previous edge is tested. The edge between any two vertices of the same triangle that has the lowest angle to the previous edge make up the outer border of the projected triangles.

The original vertices of the portal that make up the 2D border are then copied and moved far enough in the travel vector's direction to make sure that the object can fit inside the container. The distance moved is the maximum length of the portal's and object's bounding boxes added together. Triangles are then created between the original vertices and the moved ones. These triangles make up the sides of the container where the portal itself is its bottom (see fig. 3.18).

The object is then tested against the container using ray-triangle intersection to find where they intersect each other [13, p. 747]. For each triangle in the object and container, three rays are shot from one vertex to the next in each triangle and tested against the other object's triangles (a portal is also an object). The resulting barycentric coordinates or distance value from the intersection is used to interpolate the vertices of the object to slice. More than one identical intersection may be found since triangle edges may be shared among triangles. These are removed after finding all intersections. Figure 3.17 illustrates the intersection test between the object and container.

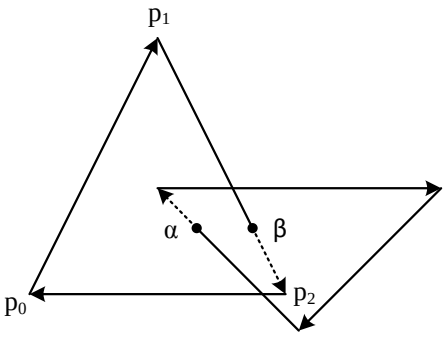


Figure 3.17: Triangle intersection and vertex interpolation between two triangles. A ray is shot from each vertex to the next in each triangle and tested against the other for intersection. p_0, p_2 and p_2 are the vertices of the first triangle. α and β are the vertices where the two triangles intersect each other.

The vertex where the *container* triangle intersects the *object* triangle is calculated as follows:

$$\alpha = (1 - u - v)\boldsymbol{p_0} + u\boldsymbol{p1} + v\boldsymbol{p2} \tag{3.9}$$

The vertex where the *object* triangle intersects the *container* triangle is calculated as follows:

$$\beta = (1 - t)\boldsymbol{p_1} + \boldsymbol{p_2} * t \tag{3.10}$$

Where t is the distance from the origin of the ray to the point of intersection.

Ray-triangle intersection is also used to separate vertices of the object inside the container from vertices outside it. A ray is shot from each vertex in the object in the opposite direction of the object's travel-vector. A vertex is inside the container if the associated ray hits a portal triangle, and outside it if no triangles are hit. Since the rest of the triangles of the container (the sides) are parallel to the direction of the rays, they are excluded from the intersection test.

The intersections calculated earlier are then added to these two lists of vertices. The first list contains the slice of the object to teleport to the destination environment and the second list contains the slice that shall stay in the source environment. Each list of vertices are then triangulated using CGAL's Delaunay triangulation [23]. This creates new triangles which are not part of the original mesh. These are removed if their center-point does not exist on any triangle of the original object. Triangles may be generated on the wrong side of the container if the portal is non-planar. Incorrect triangles may be created from the first list of vertices where the portal bends in the direction of the object's travel-vector. Incorrect triangles may be created from the second list where the portal bends in

the opposite direction. These are removed from the first list by shrinking the container slightly around its center to prevent floating-point errors when performing line-triangle intersection between the second list of triangles and the container. Triangles with a line that intersects the container are discarded. Figure 3.18 illustrates how these triangles are removed.

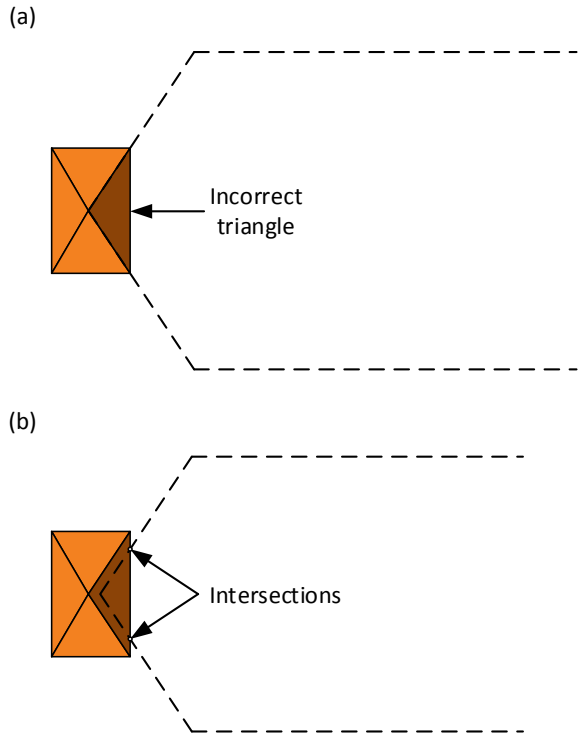


Figure 3.18: Removing triangles inside the portal container. The container is marked with dotted black lines. The triangles of the object to stay in the source environment are marked in orange. The triangle that is inside the container and shall be removed is marked in brown. (a) Triangles have been created from the intersections and the vertices outside the container. (b) The container has been shrunk around its center to prevent floating-point errors. Triangles which intersect the container are removed.

Incorrect triangles are removed from the second list of triangles in the same way except the container is enlarged instead of shrunk before performing the intersection testing. Finally, two new objects are created from the remaining triangles in the lists. The new objects inherit scale, position, rotation and textures from the original object and the portal's transformation matrix is applied to the object inside the container.

The way of slicing an object against a portal presented in this section is performed once more when the two no longer intersect each other to create the final,

correct slice(s). Otherwise, the slices from the previous frame where the object was intersecting both the *portal* and its container is used instead of the intersection against the other part of the portal *container*. This problem is illustrated in fig. 3.19.

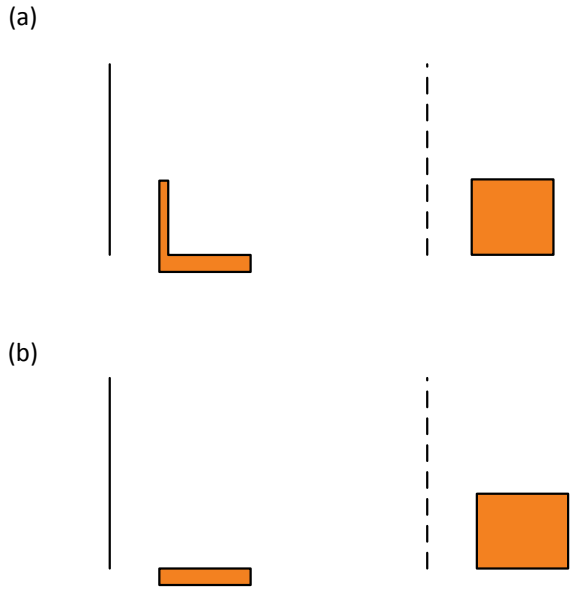


Figure 3.19: Extra slicing. A square moving from left to right is sliced at the portal's bottom border. The portal is marked with black solid lines and its destination with dotted lines. (a) The object interacts the portal without slicing it once more after the two no longer intersect, resulting in incorrect slicing. (b) The object is sliced against the portal *container* once more after they no longer intersect, resulting in correct slices.

Chapter 4

Results

The executable program was compiled using Visual Studio 2013 Professional 32-bit release mode with all default compiler optimizations enabled and executed using one thread. As few processes with as little activity as possible were kept running during the data gathering. Power management mode for the graphics card was set to prefer maximum performance to limit fluctuations. Likewise, the processor was set to run at maximum capacity. The program was run using the following specifications:

Graphics card	NVIDIA GeForce GTX 670 [1]
Processor	Intel(R) Core(TM) i5-3570K
Motherboard	Asus P8Z77-V LX
Main memory	2x Corsair CMZ8GX3M2A1886C9 @ 1333mhz
Operating system	Windows 8.1 Pro 64-bit

Table 4.1: Environment used to generate the results.

1000 samples were taken for each test in this chapter and each one was rendered at a resolution of 800x600. No portal recursions were performed for any of the tests.

The results shown in table 4.3 were gathered using combinations of the shapes presented in table 4.2. Only one object and one portal were used for each test. The camera was positioned parallel to the portal directed at it for the tests. Both object and portal were static and not moving. The results were gathered using Windows QueryPerformance() functions.

For the portal rendering tests in 4.4, circle shapes of significantly more triangles were used. The camera was positioned in front of the portal and the same scene of a room (cube) was used for each test. The portal connected to the space shortly behind it, ensuring no portal recursion occurred. As the shape differ slightly between each test, so does the number of portal pixels written to the buffers. The portal rendering results were gathered using Direct3D's ID3D11Query interface.

[1] Drivers updated on 09-June-2015.

#	**Shape name**	# **of triangles**
1	Triangle	1
2	Square	2
3	Circle4	4
4	Circle8	8
5	Circle16	16
6	Circle32	32
7	Circle64	64
8	Circle128	128
9	Circle256	256
10	Cone	8
11	Cube	12
12	Sphere	48

Table 4.2: Object and portal shapes used in different tests and the number of triangles each shape has.

Test#	Object shape	Portal shape	Slicing (ms) Mean	Max
1	Triangle	Triangle	0.19	0.27
2	Triangle	Square	0.24	0.60
3	Triangle	Circle4	0.34	0.59
4	Triangle	Circle8	0.50	1.21
5	Triangle	Circle16	0.81	1.09
6	Triangle	Circle32	1.39	2.16
7	Triangle	Circle64	2.86	4.17
8	Triangle	Circle128	6.38	9.47
9	Triangle	Circle256	15.93	20.81
10	Square	Triangle	0.23	0.58
11	Circle4	Triangle	0.40	0.57
12	Circle8	Triangle	0.64	1.00
13	Circle16	Triangle	1.14	1.48
14	Circle32	Triangle	2.26	3.73
15	Circle64	Triangle	4.00	6.54
16	Circle128	Triangle	9.98	13.16
17	Circle256	Triangle	24.63	29.94
18	Cone	Cone	6.82	8.23
19	Cube	Cone	13.84	20.28
20	Sphere	Cone	34.29	41.93

Table 4.3: The object-portal interaction execution (slicing) time of different combinations of object and portals shapes.

Test#	# of pixels	# of triangles	Render (ms) Mean	Max
A	65926	65538	0.67	1.06
B	65922	131076	1.01	1.40
C	65920	262152	1.54	3.23
D	65920	524304	1.79	2.13
E	65920	1048608	2.26	2.60
F	65920	2097216	3.28	4.53
G	65928	4194432	5.50	5.94
H	65968	8388864	10.72	11.7
I	66028	16777728	20.90	22.25

Table 4.4: The portal rendering time using different number of triangles to represent the portal shape. Around 66000 portal pixels were written to the stencil buffer for each test.

Tests 18-20 are shown in figs. 4.1 to 4.3. Figures 4.4 to 4.7 show how a cube is sliced against portal surfaces and borders as it travels through portals.

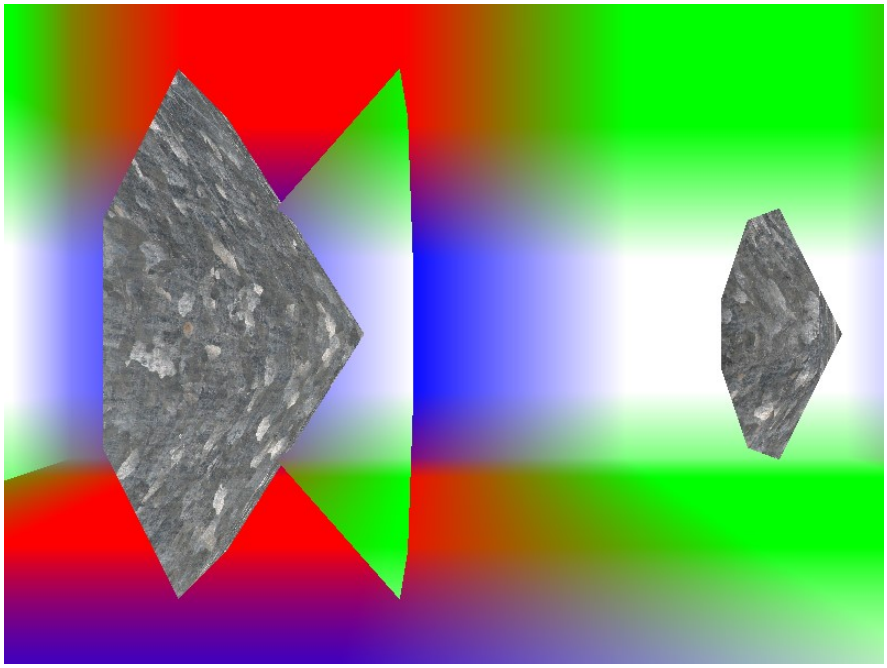


Figure 4.1: Slice object against portal surface. A cone is intersecting a complex portal (cone-shaped). The cone is sliced against the portal's surface.

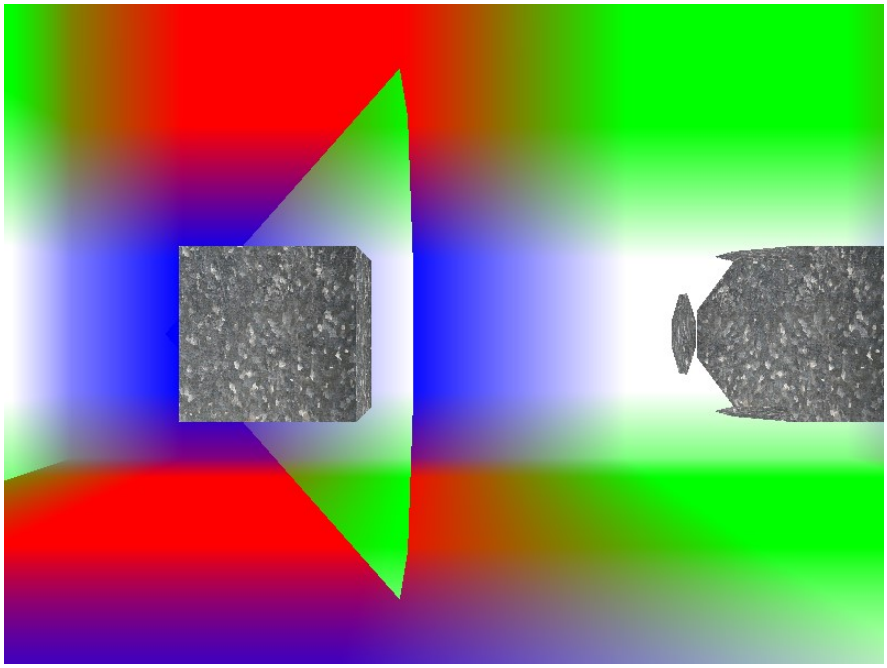


Figure 4.2: Slice object against portal surface. A cube is intersecting a complex portal (cone-shaped). The cube is sliced against the portal's surface.

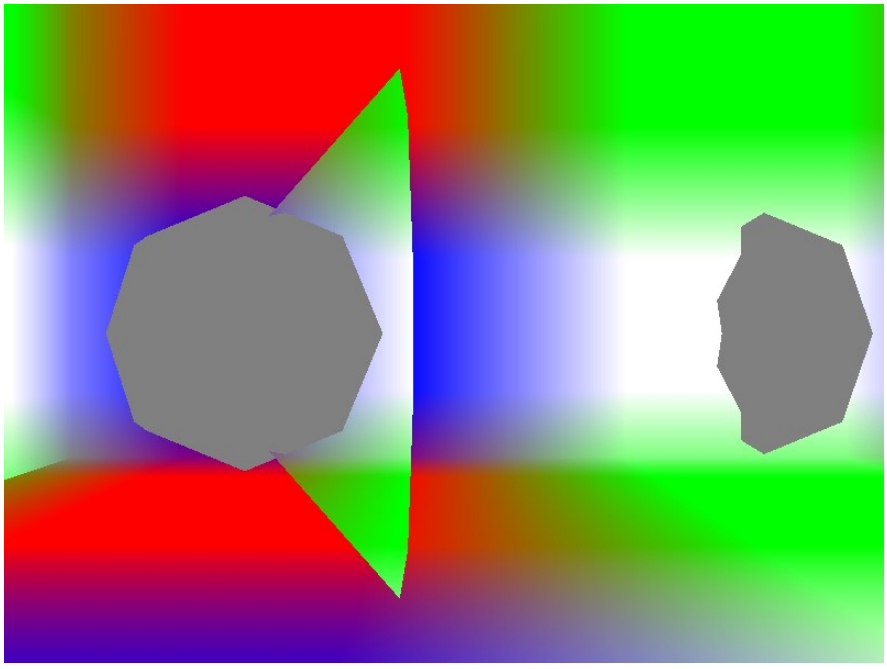


Figure 4.3: Slice object against portal surface. A sphere is intersecting a complex portal (cone-shaped). The sphere is sliced against the portal's surface.

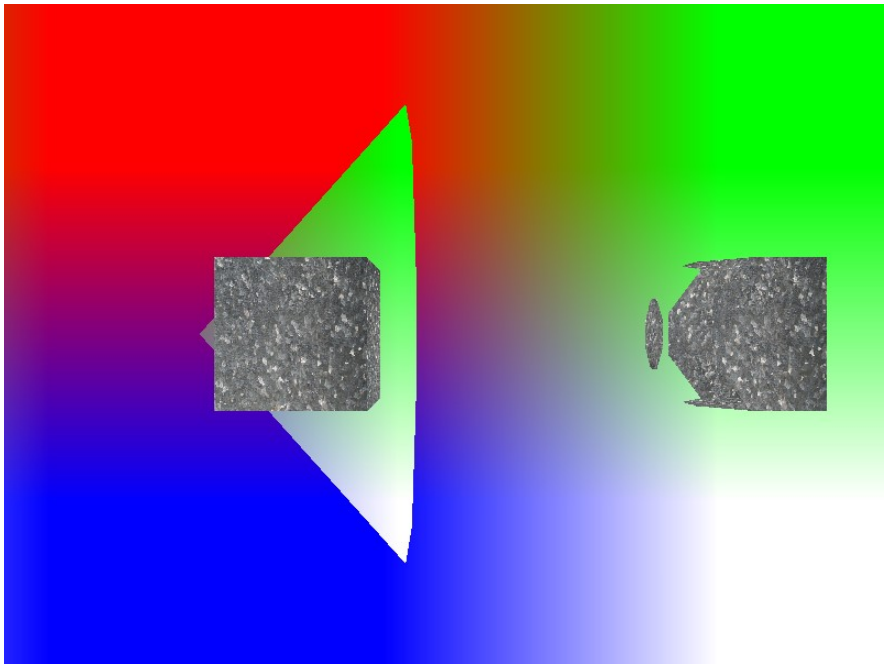


Figure 4.4: Slice object against portal surface. A cube is intersecting a complex portal (cone-shaped). The cube is sliced against the portal's surface.

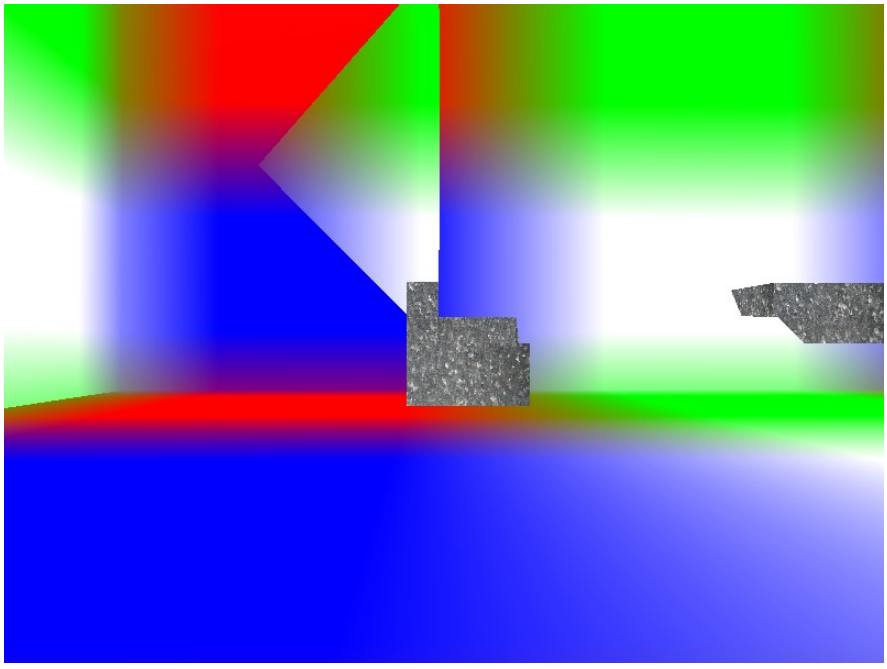


Figure 4.5: Slice object against portal border. A cube intersecting a complex portal (cone-shaped). The cube is sliced against the portal's border.

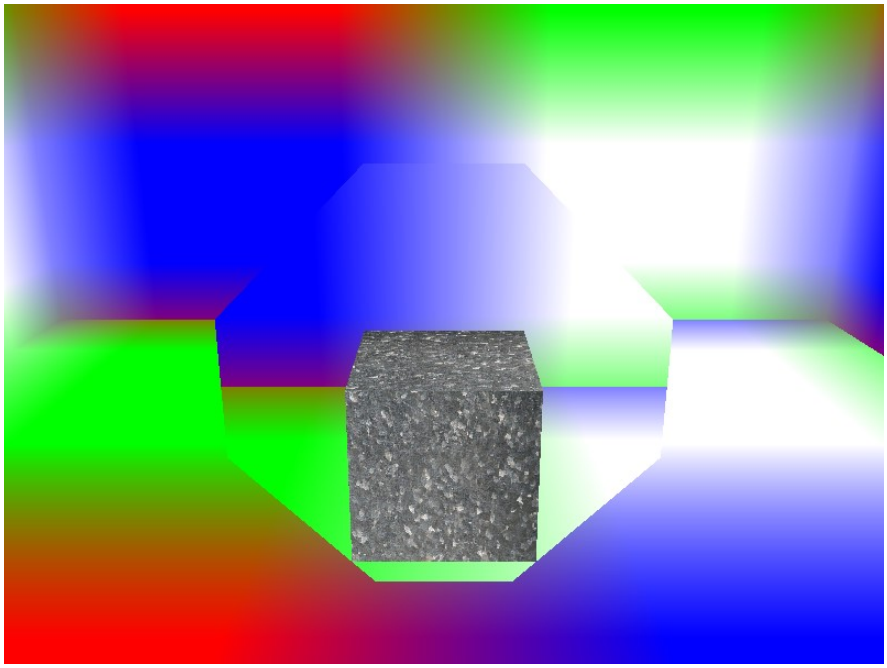


Figure 4.6: A cube moving through a portal (cone-shaped). The cube is (invisibly) sliced against the portal's surface and fully visible through the portal.

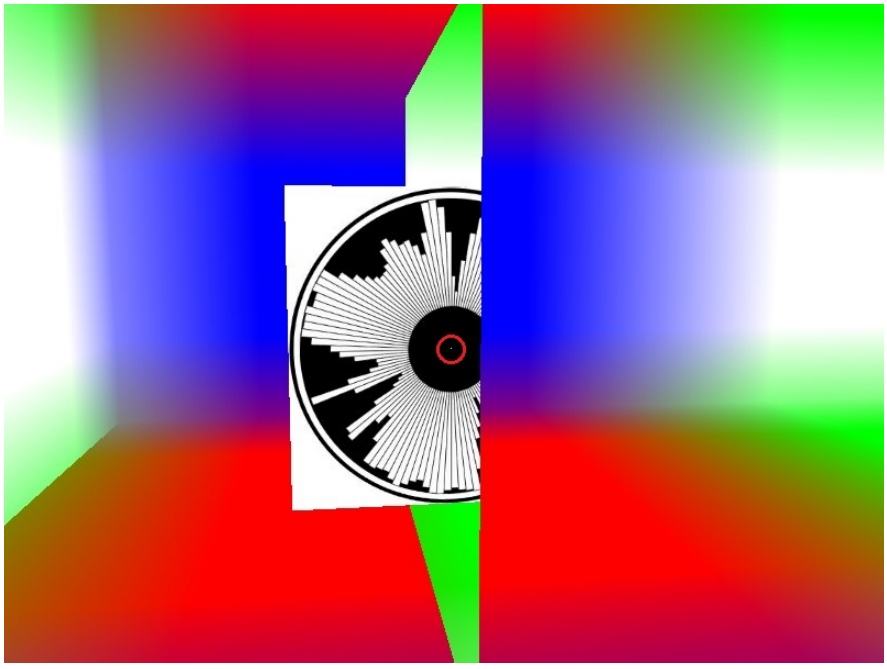


Figure 4.7: Robustness issue - render slices. A cube is intersecting a portal (square). A portal pixel is visible through the object (marked with red circle).

Chapter 5

Analysis

The results of test 1-17 from table 4.3 are summarized in fig. 5.1.

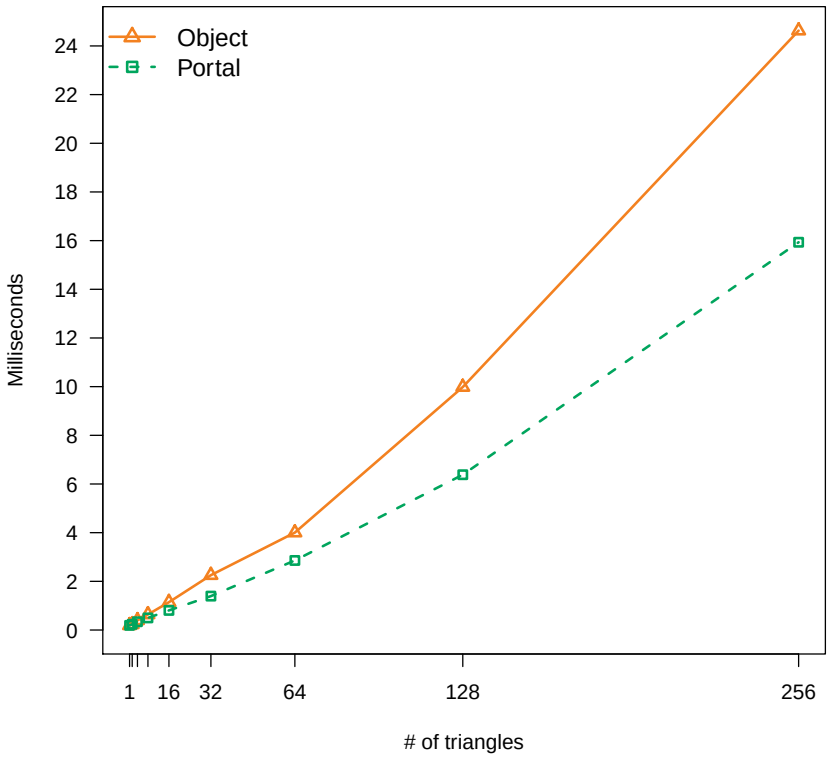


Figure 5.1: Plot of the average execution times of the object-portal interaction using different number of triangles for the object and portal shapes.

Increasing the number of triangles in both the portal and object shape seem to increase exponentially as an increasing amount of triangles are added to their shapes. Increasing the number of triangles in the object shape is more expensive than increasing the number of triangles in the portal shape by the same amount.

The results of test A-I from table 4.4 are summarized in fig. 5.2.

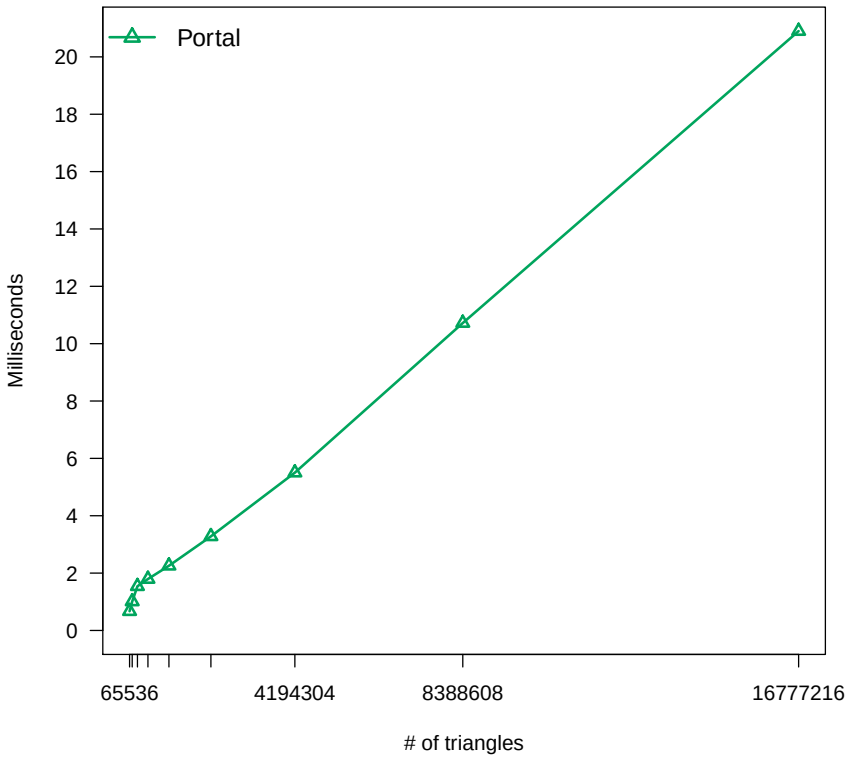


Figure 5.2: Plot of the average rendering times of portals using varying number of triangles to represent the portal shape.

The visual correctness of the object-portal interaction was qualitatively tested for each test in table 4.3 to see if the objects moved through the portals seamlessly without any clipping errors.

The research questions defined in the Introduction are answered as follows:

1. *Can complex portal-object interaction be achieved in real-time?*
   Yes, although the shapes of the object and portal must be simple for the implementation to execute in real-time on the hardware used. The object shape cannot contain many more triangles than 128 before the execution time is no longer considered real-time. Likewise, the portal shape must contain 256 or fewer triangles for the object-portal interaction to be in real-time. Cone and cube objects interacting with a cone portal are able to execute in real-time whereas a sphere object does not (see test 18-20 in table 4.3).

2. *Is the implemented object-portal interaction method visually correct?*
   Figures 4.4 and 4.5 show correct clipping (cutting) between the portal and object shapes. Figure 4.6 show the object seamlessly moving through the portal fully visible and visually unmodified. Figures 4.4 and 4.5 also show parts of the object through the portal seamlessly joining with the part still in the source environment, as does figs. 4.1 to 4.3. However, fig. 4.7 show a rendering artifact caused by slicing the object. This can happen along the surface of the portal where the object was sliced when a portal pixel overwrites an object pixel. A pixel-perfect rendering system is needed for the object-portal interaction to be entirely, visually, correct.

Chapter 6

Discussion

The cost of increasing the number of triangles in the object shape is non-linear as the triangulation does not scale linearly and in the worst case is $O(n^2)$ [23]. Likewise, increasing the number of triangles in the portal shape does not scale linearly as finding the border of its projected points in the worst case has a complexity of $O(n^2)$. As such, the worst case complexity of the implementation becomes $O(n^2 * m^2)$ where n and m are the number of triangles in the object and portal shapes respectively. The results show that the interaction between a portal made up of one triangle and an object of 128 triangles executed shortly under 10 ms. This limits the application of the implementation to simple object and portal shapes or to offline rendering. The results cannot be compared to previous knowledge as no results have been published of other object-portal interaction methods. Increasing the number of triangles in the object shape is more expensive than increasing the number of triangles in the portal shape by the same amount. This is because finding the container of a portal involves no intersecting testing whereas ray-triangle tests are performed to see which points of the object are inside the portal container. Another costly computation is that the separated points are triangulated and extra effort is required to remove unwanted triangles.

The rendering of portal scales linearly when increasing the number of triangles in the portal shape and the number of rendered portal pixels are similar. Note that the cost of portal rendering mostly depends on the number of objects rendered through it and how many times a portal is rendered through another. It is therefore important to shrink the culling frustum tightly around the portal shape.

A validity issue of the results is that the method was not extensively tested beyond what was reported in the results chapter. Floating-point errors have been considered for the object-portal interaction and are mitigated using epsilon values. The answer to research question 1 depends if visual correctness has been achieved which is biased by author subjectivity. This has been mitigated by comparing how objects move through portals to how they are normally rendered. Rendering artifacts likely caused by floating-point rounding errors seldom occur when portal pixels overwrite object pixels where the two intersect, see fig. 4.7. No epsilon value can be used in the portal-interaction method to mitigate the effect as it will increase the artifacts from other viewpoints.

Chapter 7
Conclusions and Future Work

An object-portal interaction system has been presented that allows objects to move through complex portals and be cut against their surfaces and borders. Unlike other work, the objects and portals can be of any shape and can exist anywhere. The state-of-the-art portal rendering system is extended to discard portals which lie between the camera's near plane and the surfaces of other portals and discard portals which lie behind already rendered portals. Its portal-culling method is extended to more tightly fit around the bounding volume of a portal. The thesis is of interest to those who want object-portal interaction of both simple and complex portals used in gameplay and special effects without restriction on portal placement and shape with the exception that portals may not have holes in their shape in the direction an intersecting object is moving.

The answers to the research questions are summarized as follows:

1. *Can complex portal-object interaction be achieved in real-time?*
   Yes, but the object and portal shapes must contain 256 triangles or fewer for the implementation to execute in what is defined as real-time on the hardware used.
2. *Is the implemented object-portal interaction method visually correct?*
   Yes, with the exception that portal pixels sometimes overwrite object pixels where the two intersect, causing the portal's connected space to be rendered on top of the object at these pixels.

An inherent limitation with non-volumetric portals is that an object which intersects a portal needs to be traveling in a direction or else it cannot be determined which part (slice) to move to the portal's destination. For volumetric portals, an object can be considered tethered to the source environment as long as *any* part of it is outside the volume of the portal. This eliminates the need to slice the object and the cost associated with it; the interaction is covered by the collision detection system.

The main limitations of the implementation are that the method of slicing objects does not consider portals with holes in them, that the slicing direction is static and that only the original object's shape is preserved. Portals may not have holes in them in the direction the object is moving because no internal borders

are considered. A change in the object's (parent) direction while it interacts with a portal is not considered and therefore the two derived slices cannot travel in different directions while their parent is interacting with the portal. If a solid object is sliced, no extra triangles are created along the cut to make the slices solid. Extra work is necessary to remove unwanted triangles from the triangulation method used and it does not differentiate between vertices with the exact same position which can result in wrong vertex data (beside position) for a triangle. This limits the triangulation to triangle strips or triangle lists where position data is unique. It also does not create front or back-facing triangles exclusively and requires that both objects and portals are made up of at least one triangle each.

As the far-depth (normal depth) buffer is cleared at the pixels where a portal is rendered, it is required that these pixels are overwritten when rendering the portal's destination or else old color data will remain in the color buffer at the corresponding far-depth pixels. In rare cases when a portal is rendered through another, it is possible that a few portal pixels are written to the stencil buffer when they shouldn't, causing infinite recursion. Since a normal stencil buffer of 8 bits is used, the maximum number of recursions is 256.

The portal interaction implementation is computationally expensive and optimizing it would make the method more appealing to real-time constricted applications. A good candidate to start with is to improve the triangulation to only create wanted triangles. It could also be extended to handle other vertex data beside position and to handle slices of solid objects. Another way to speed up the interaction would be to apply a bounding volume hierarchy which could greatly reduce the number of intersection tests and reduce the computation time. Furthermore, parallelizing the implementation using the GPU could substantially speedup the object-portal interaction. It could also be extended to handle portals with holes in them by finding the inner border(s) of a projected portal.

The intensity of the light coming through portals could be applied to their source environments to increase their realism. A light shaft coming out from the portal boundary could also be added to increase the effect. Since portals are distortions of space, it may be appealing to have two portals of different shapes connected to each other. A mapping system that maps points on the first shape to points on the second shape would then be needed. A collision system that handles slices transported to the portals connected space may also be worth exploring.

Appendix A

Buffers

The color buffer is where image data of the scene is written to. It is presented to the screen after rendering a frame. A common format for this buffer is 32 bits per pixel (8 bits per component; red, green, blue and alpha), normalized float values. This format allows $8^3 = 16777216$ different color combinations as well as 256 different alpha values which is sufficient in many cases. Writing to the color buffer is determined by the depth and stencil buffers if these are used.

The depth buffer is of the same size as the color buffer and contains the depth values ($0 \leq z \leq 1$ in Direct3D) of each pixel rendered and a depth test determines how these values are overwritten. This test is a simple Boolean comparison ($<$, $==$, $<=$, $>$, $!=$ $>=$), or "always" and "never" which simply means that the test will always or never pass, respectively. The two values compared is the one already in the depth buffer and the new rendered depth value. Normally, the depth test function is set to *less*, meaning that any newer pixels rendered with a depth values less than what is already in the depth buffer overwrites the current value and the pixel's color information is written to the color buffer. This makes it so that objects closer to the viewer are seen. One would normally use 32 bits (or 24 bits if an 8-bit stencil is used) for this value to get the highest amount of precision to avoid z-fighting [13, p. 833].

The stencil buffer is tied to the depth buffer and are of the same size. A stencil buffer is an integer mask used to render effects to certain areas on the color buffer. The stencil buffer has three tests, one for each condition below, to determine how values in the buffer are overwritten. The two values compared are the value already in the stencil buffer and a reference value set before rendering. An operation is performed on the corresponding value in the buffer if a stencil test is accepted. This operation can be one of the following[1]:

1. Keep existing stencil data.
2. Set the stencil data to 0.
3. Set the stencil data to the reference value.
4. Increment the stencil value by 1, and clamp the result.

[1]From the D3D11_STENCIL_OP enumeration: `https://msdn.microsoft.com/en-us/library/windows/desktop/ff476219(v=vs.85).aspx`

5. Decrement the stencil value by 1, and clamp the result.
6. Invert the stencil data.
7. Increment the stencil value by 1, and wrap the result if necessary.
8. Decrement the stencil value by 1, and wrap the result if necessary.

A stencil operation is invoked if one the following conditions are true[2]:

1. The stencil test failed.
2. The depth test failed.
3. The stencil and depth tests both passed.

Each of these conditions can have its own stencil operation, but only the one associated with the first condition that is true is executed.

[2]From the D3D11_DEPTH_STENCILOP_DESC structure: `https://msdn.microsoft.com/en-us/library/windows/desktop/ff476109(v=vs.85).aspx`

References

[1] N. Lowe and A. Datta, "A technique for rendering complex portals," *Visualization and Computer Graphics, IEEE Transactions on*, vol. 11, pp. 81–90, Jan 2005.

[2] C. B. Jones, "A new approach to the 'hidden line' problem," *The Computer Journal*, vol. 14, no. 3, pp. 232–237, 1971.

[3] J. M. Airey, "Increasing update rates in the building walkthrough system with automatic model-space subdivision and potentially visible set calculations," tech. rep., DTIC Document, 1990.

[4] J. M. Airey, J. H. Rohlf, and F. P. Brooks Jr, "Towards image realism with interactive update rates in complex virtual building environments," in *ACM SIGGRAPH Computer Graphics*, vol. 24, pp. 41–50, ACM, 1990.

[5] S. J. Teller and C. H. Séquin, "Visibility preprocessing for interactive walkthroughs," in *ACM SIGGRAPH Computer Graphics*, vol. 25, pp. 61–70, ACM, 1991.

[6] D. G. Aliaga and A. A. Lastra, "Visibility Culling using Portal Textures," *University of North Carolina at Chapel Hill, Chapel Hill, NC*, 1997.

[7] W. Jiménez, C. Esperança, and A. A. Oliveira, "Efficient algorithms for computing conservative portal visibility information," in *Computer Graphics Forum*, vol. 19, pp. 489–498, Wiley Online Library, 2000.

[8] D. Luebke and C. Georges, "Portals and mirrors: Simple, fast evaluation of potentially visible sets," in *Proceedings of the 1995 symposium on Interactive 3D graphics*, pp. 105–ff, ACM, 1995.

[9] D. Haumont, O. Debeir, and F. Sillion, "Volumetric cell-and-portal generation," in *Computer Graphics Forum*, vol. 22, pp. 303–312, Wiley Online Library, 2003.

[10] A. Lerner, Y. Chrysanthou, and D. Cohen-Or, "Breaking the walls: Scene partitioning and portal creation," in *Computer Graphics and Applications, 2003. Proceedings. 11th Pacific Conference on*, pp. 303–312, IEEE, 2003.

[11] A. Lerner, Y. Chrysanthou, and D. Cohen-Or, "Efficient cells-and-portals partitioning," *Computer Animation and Virtual Worlds*, vol. 17, no. 1, pp. 21–40, 2006.

[12] S. Teller and P. Hanrahan, "Global visibility algorithms for illumination computations," in *Proceedings of the 20th annual conference on Computer graphics and interactive techniques*, pp. 239–246, ACM, 1993.

[13] T. Akenine-Möller, E. Haines, and N. Hoffman, *Real-time rendering*. CRC Press, 2008.

[14] E. Orbons and Z. Ruttkay, "Interactive 3D Simulation of Escher-like Impossible Worlds," 2008.

[15] A. Petersson, "Bachelor Thesis Fast complex transformative portals," 2013.

[16] B. K. Vedel-Larsen, "Very Large Scale Environments," 2008.

[17] Y. Yang and Y. Wang, "Walkthrough applications based on portal," in *Electronics, Computer and Applications, 2014 IEEE Workshop on*, pp. 277–283, IEEE, 2014.

[18] P. W. Maciel and P. Shirley, "Visual navigation of large environments using textured clusters," in *Proceedings of the 1995 symposium on Interactive 3D graphics*, pp. 95–ff, ACM, 1995.

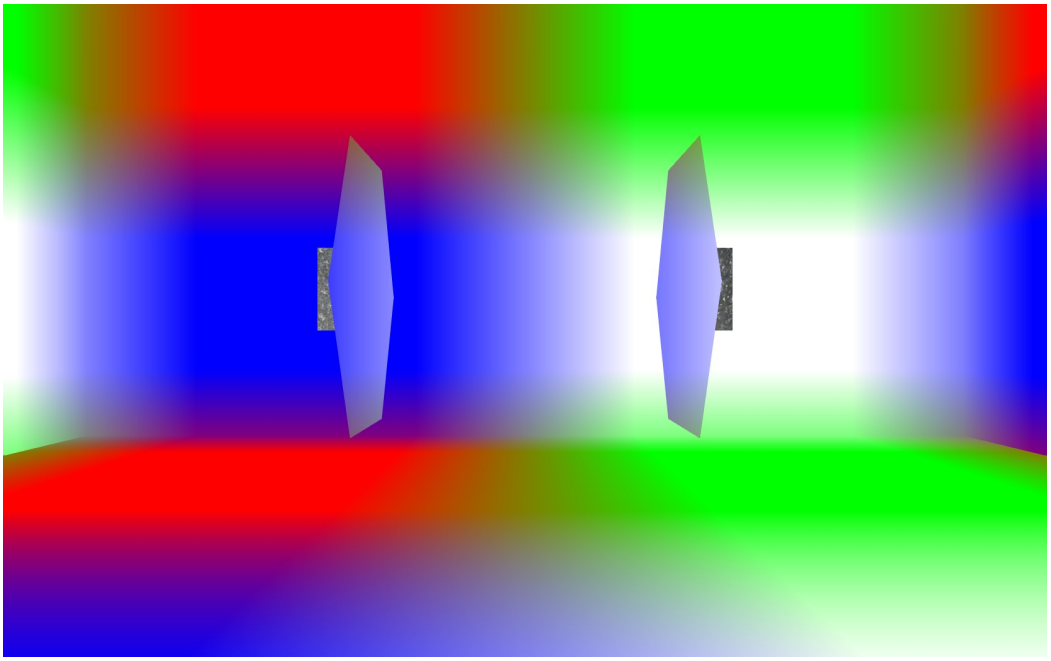
[19] "Unreal Engine." `https://www.unrealengine.com/`. [Online; accessed 2015-May-20].

[20] "Crystal Space Engine." `http://crystalspace3d.org/`. [Online; accessed 2015-May-20].

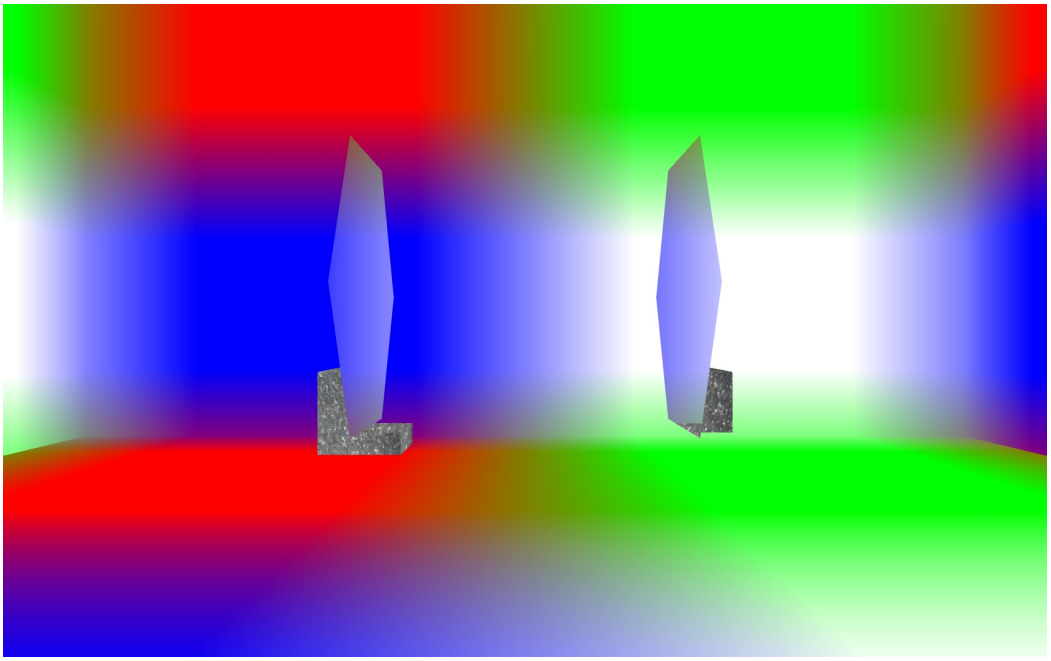
[21] S. Lefebvre, S. Hornus, *et al.*, "Automatic cell-and-portal decomposition," 2003.

[22] E. Lengyel, "Oblique View Frustum Depth Projection and Clipping," *Journal of Game Development*, vol. 1, no. 2, pp. 5–16, 2005.

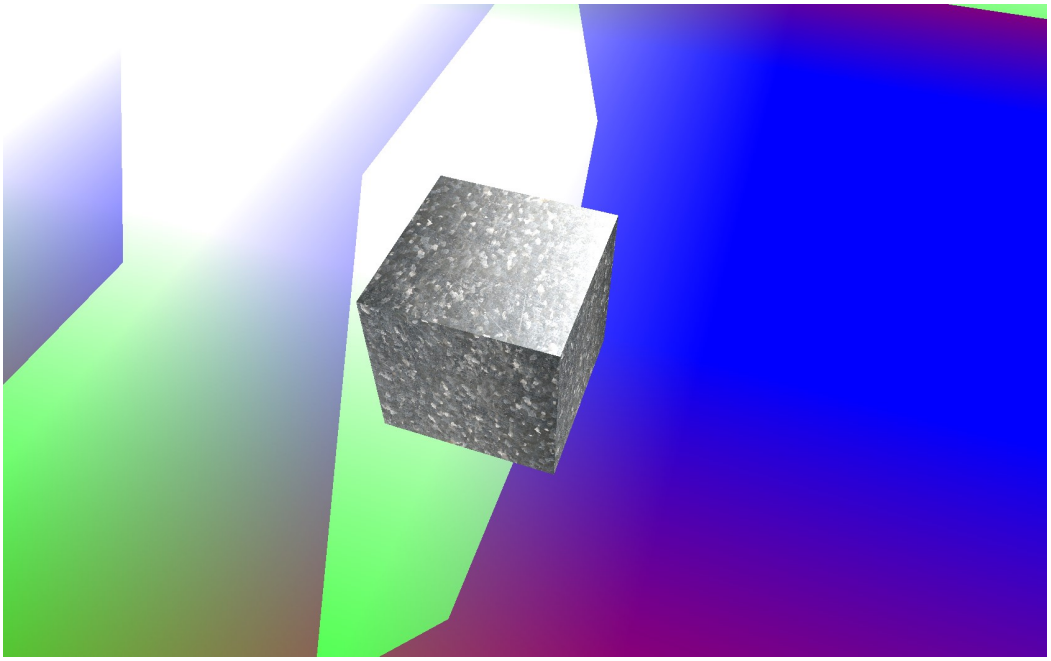
[23] The CGAL Project, *CGAL User and Reference Manual*. CGAL Editorial Board, 4.6 ed., 2015.



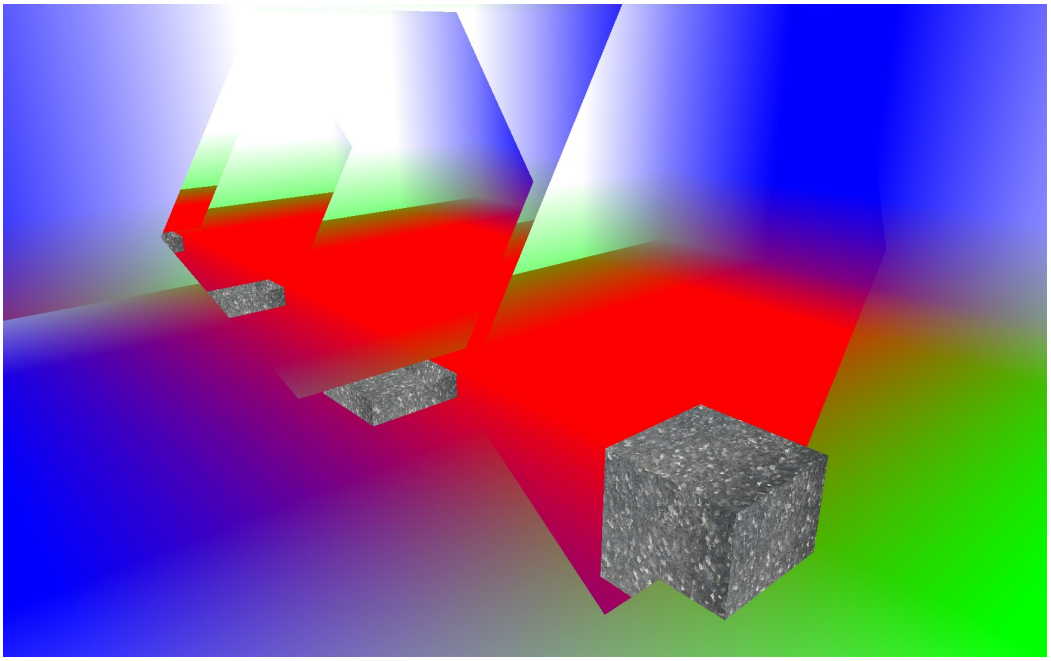
Cube sliced against surface of portal.



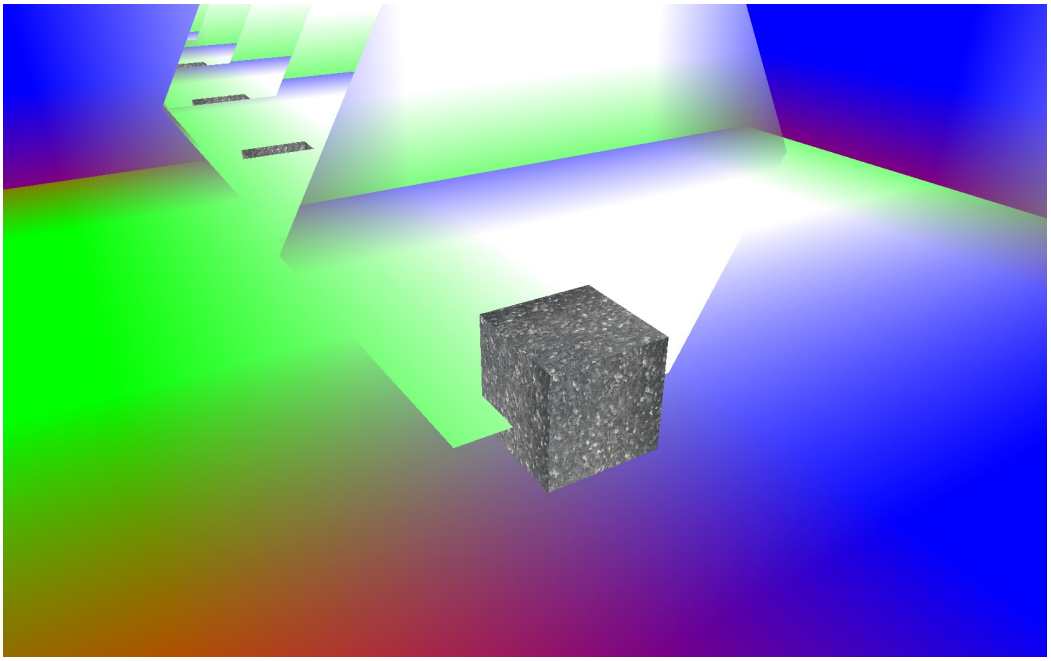
Cube sliced against surface and border of portal.



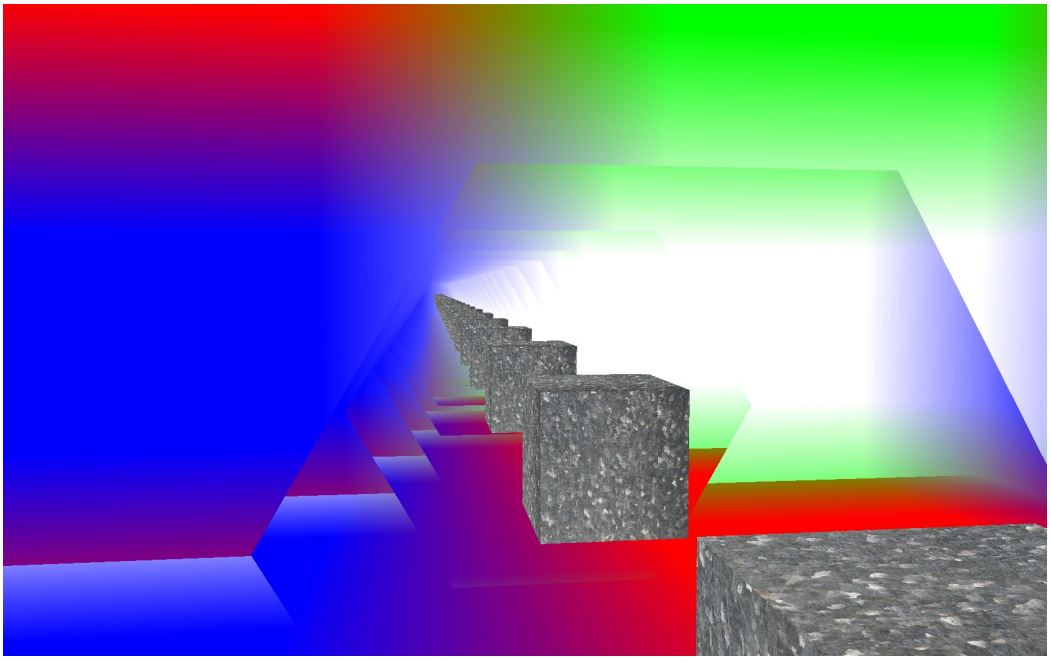
View of cube as it moves through a portal.



View of cube as a part of it moves through a portal and the other part stays in the source environment.



View of cube as a part of it moves through a portal and the other part stays in the source environment.



Portal recursion.