Abstract

In this thesis an image-space and geometry hybrid algorithm for portal rendering is constructed and evaluated in terms of correctness and performance. A number of tests are defined and out of these three are selected and executed. Based on the results of these tests, the conclusion is that a hybrid approach to portal rendering is a valid option for increasing the speed without sacrificing the correctness of an image-space portal rendering algorithm.
Contents

1 Introduction 3
  1.1 Background ................................................. 3
  1.2 Goal and purpose ........................................... 7
  1.3 Research questions ......................................... 7
  1.4 Method ..................................................... 7
  1.5 Report layout ............................................... 8

2 Construction 8
  2.1 Cell and portal graph ....................................... 9
  2.2 Image space algorithm ...................................... 9
  2.3 Image space and geometry hybrid algorithm .................. 12

3 Experiment design 12
  3.1 Test specifications .......................................... 13
    3.1.1 Test 1: Number of visible objects ..................... 13
    3.1.2 Test 2: Number of out-of-view or occluded objects ..... 13
    3.1.3 Test 3: Clipping behavior ............................. 13
  3.2 Selected tests .............................................. 13

4 Results 13
  4.1 Test 1: Number of visible objects ......................... 14
  4.2 Test 2: Number of out-of-view or occluded objects .......... 14
  4.3 Test 3: Clipping behavior .................................. 14

5 Discussion 14

6 Conclusions 15

7 Future work 16
  7.1 Tests not included in this thesis .......................... 16
    7.1.1 Test 4: Number of portal recursions .................. 16
    7.1.2 Test 5: Number of discarded portals ................ 16
    7.1.3 Test 6: Screen resolution ............................. 16

8 Acknowledgments 17
1 Introduction

In the game development field, one often wants to create the effect of wormholes. Richard F. Holman [5] explains that “wormholes are solutions to the Einstein field equations for gravity that act as ‘tunnels’, connecting points in space-time in such a way that the trip between the points through the wormhole could take much less time than the trip through normal space”. Throughout this thesis, instead of wormhole, the term portal will be used and defined as a visual link between two locations. These two locations can be either adjacent to each other, or not. Typically, players (and potentially other objects in the game as well) would be able to travel through this portal as if it was a magical doorway into the remote location. However, this thesis will only focus on the visual aspect of portals; how to achieve the effect of seeing a remote or adjacent location through a portal, not on objects traveling through portals. In the case or portals linking remote locations together, we will use the term transformative portals.

Doing this in a 3d world is not trivial. Especially not with adequate performance for interactive games. The game Natural Selection 2 (NS2) from 2012 is a very recent example where portals play a huge part in the game mechanics. It’s a multiplayer game with two teams trying to capture and hold certain areas of a mostly indoor level. One of the teams has the ability to construct “phase gates”, which are basically transformative portals by the definition used in this thesis. This then allows that team’s players to travel instantly across the map between the locations of the portals. However, the portals in NS2 only allows players to travel through them, but not see through them. Instead they appear as a surface with a texture. It would, in this author’s view, enhance the game experience and create a more fun game mechanic if players were allowed to see through the portals. As it is, players have to blindly jump through the portals without knowing what’s going on at the remote location.

As to why it is this way in NS2, Charlie Cleveland, Game Director at Unknown Worlds Entertainment, explains: “It’s just extra work to add portals to Spark[1]. We always wanted to add this but ran out of time - and now it’s not a priority”. This shows that portals aren’t trivial and that the field is worth exploring. Following is an introduction to portal culling and a walk-through of previous work in this area.

1.1 Background

There are a number of acceleration algorithms under the family name Portal Culling, sometimes Portal Rendering. The goal of portal culling algorithms is to speed up rendering of indoor environments, by taking advantage of the fact that walls often act as large occluders. Typically the world is divided into cells and portals. Basically, cells are rooms and portals are doorways and windows that link adjacent rooms together. In order to achieve faster rendering of such a scene, the algorithms try to exclude cells that aren’t visible from rendering. To determine which cells are visible and which aren’t, the algorithms track the camera’s field of view (adjustable angle, typically around 110-120 degrees). If a portal is in the field of view and not occluded by a wall, then the associated cell on the other side needs to be rendered, otherwise it can be discarded. The algorithm then recursively examines the next cell for visible portals [4].

The first portal culling algorithm was introduced by Airey [3] and later improved by among others Teller and Hanrahan [13, 4]. Luebke [11] presents a simplified and more effective version

\[\text{Spark is the name of the game engine developed by Unknown Worlds Entertainment and used in Natural Selection 2.}\]
Figure 1: Cells are enumerated from A to E. Openings joining cells together are portals. Only geometry visible through portals is rendered. In this case cell B is completely discarded from rendering because the field of view doesn’t intersect any of it’s portals.
Illustration by Anton Petersson, 2013.

of portal culling. Under “Future Work”, they bring up that the algorithm can be extended to encompass rendering of mirrors and some problems related to that. This is done by associating a transform or warp matrix, which can be further utilized for composing scenes. Extending the algorithm further to also include effective rendering of transformative portals is also possible [2][10]. When a transformative portal is outside the field of view, the remote cells can be excluded from rendering, thereby achieving a rendering speedup similar to that of discarded adjacent cells.
Both kinds of portals (transformative and non transformative) have typically been limited to planar convex polygons in previous work on portals. Lowe and Datta show an algorithm for rendering complex portals without such restrictions. In their algorithm, a portal can be any mesh. Actually, it doesn’t even need to be a mesh, the only criteria is that a portal can be rasterized. Their algorithm focuses on correct visibility determination and is done in image-space. This means that no view frustum culling is performed. Instead, all the geometry of a partially visible cell is sent down the rendering pipeline. In order to determine which fragments contribute to the final image, each fragment is tested against a depth range and a stencil buffer.

View frustum culling is a technique that tries to exclude objects that are out of view. The view frustum is typically defined by the projection matrix used and consists of six planes: left, top, right, bottom, near and far. The near plane forms a limit on how close an object can be to the camera and still be visible. The far plane puts a similar limit on how far the camera can see. When using perspective projection, the view frustum can be described as a pyramid, truncated at the top and bottom by the near and far planes. Objects (fully) outside this volume do not need to be rendered as they won’t affect any pixels on screen. Objects partially inside the volume (intersecting one or more planes) and objects fully inside must be rendered. So, using view frustum culling, the system can test every object in a scene against the view frustum to determine if it needs to be rendered or not.

In this thesis, a hybrid approach will be evaluated in terms of correctness and rendering speed. This hybrid algorithm will combine Lowe and Datta’s image-based algorithm with previous geometry-based algorithms in order to render complex transformative portals with view frustum culling. In geometry-based algorithms, the view frustum is shrunk to fit closely around a portal and then used for frustum culling when rendering the cell that is visible through the portal. That way, depending on the geometry layout of a cell, a varying amount of geometry can be discarded before

---

2The most common form of projection used in 3d games. The main characteristic of perspective projection is that parallel lines appear to join together at a distant point on the horizon and doesn’t appear parallel.
being sent through the rendering pipeline. For transformative portals, the view frustum needs to be transformed as well, before using it to cull geometry in any target cell.

Figure 3: Portal culling with view frustum culling. The view frustum is shrunk every time it passes through a portal, and more objects can be discarded. Discarded objects are marked with a red diagonal line. Note that the portals themselves can be discarded by the view frustum culling test. The portals connected to cell B are discarded this way. Illustration by Anton Petersson, 2013.

Other related work was done by Soumyajit Deb and Ankit Gupta in 2008. Previous occlusion culling algorithms (portal culling is a form of occlusion culling) have focused on static scenes. Deb and Gupta show an algorithm that focuses on static and dynamic environments. They use an object graph (typically called a scene graph) which they evaluate every frame, given a camera location. When evaluating this graph, they don’t start from the root of the tree every frame, instead they use spatial and temporal coherency [7].

Occlusion culling algorithms such as the one presented by Deb and Gupta are called online occlusion culling since scene visibility is fully evaluated at runtime. Other schemes, such as portal rendering, requires the scene to be divided into cells and portals as an automatic or manual preprocessing stage. Such algorithms are categorized as offline. For the game Killzone 3, good quality occluder objects were generated at level export, with the option for artists to manually tag
occluders as well [1]. Remedy Entertainment also used an online algorithm for Alan Wake [12].

1.2 Goal and purpose

The purpose of this work is to create a hybrid portal rendering algorithm capable of rendering complex transformative portals without clipping issues. The goal is to verify that Datta’s image-space algorithm can be combined with elements from previous geometry-based algorithms in order to speed up rendering, while still avoiding clipping issues.

1.3 Research questions

1. Can rendering of complex transformative portals be sped up through the use of view frustum culling?

2. Can clipping issues still be avoided when using view frustum culling?

1.4 Method

In order to answer the research questions Lowe’s and Datta’s algorithm was implemented using C++ and OpenGL. The hybrid algorithm was also implemented. Then, it was a trivial task to verify whether or not the hybrid algorithm could avoid the clipping issues that Teller and Hanrahan mentions [13].

The implementations also allowed for performance measurements. Frames were counted during an interval of 1 second and from that the average time to render a frame (frametime) was calculated. This measuring was repeated every second while the program was running. Everytime a new value was calculated, it was sent to standard output. When running the tests, these outputs were piped to text files on disk. This was the method used for all performance benchmarking in this thesis. This is explained further in section 2.

Performance was measured and compared between the two algorithms for a number of scenarios. The point with the view frustum culling of the hybrid algorithm is to achieve better performance by discarding objects not visible through portals. Therefore, two relevant test parameters are number of visible objects and number of discarded objects. Note that the pure image space algorithm also properly discards objects that are occluded or out of view, but it does so at a later stage in the rendering pipeline, at the pixel level. Another very interesting parameter is the number of portal recursions, i.e. how many nested portals the field of view intersects, as they require special attention by the algorithms. The resolution of the rendering window is also an important parameter, since it should affect the cost of image-space operations.

These are the independent variables that were examined:

1. Number of visible objects
2. Number of discarded objects
3. Number of portal recursions
4. Window resolution
Objects, as mentioned above, could be any 3d object as long as they are equally demanding to render. In these experiments the objects were simple textured cubes. The dependent variables are frame-time and correctness. Frame-rate is not a variable of it’s own, but rather an alternative representation of frame-time.

This is a quantitative approach.

1.5 Report layout

Thus far, the area of research has been introduced and motivated. Goals and research questions are explained, followed by a brief description of the method used for answering these questions.

Following is a thorough description of the algorithms and related areas in section 2. Then, in section 3 a number of specific tests are briefly defined and motivated. Out of these tests, three were selected as a way of limiting the thesis because of time constraints. The selected tests are explained in greater detail. In section 4 the results of the experiments are presented, followed by a discussion in section 5 and conclusions in section 6. Finally, some ideas for future work in this area have been compiled in section 7.

2 Construction

In this section the construction of the two algorithms will be explained in detail. First off is a brief introduction to the concept of frame buffers. The buffers described here are all very common. They are: the color-, depth- and the stencil buffer and the algorithms described in this section relies heavily on these.

The color buffer is just a memory area, typically on the graphics accelerator, containing the color values for every pixel of the screen. They can be stored in a number of ways including 16 bits per pixel and 24 bits per pixel (typically 8 bits for each of the color components red, green and blue). In real-time graphics applications these color values are written as the current frame is rendered [4]. However, writing to the color buffer is usually controlled by a number of test functions. The most commonly used is the depth test.

The depth test is used to make sure that pixels from objects closer to the viewer don’t get overwritten by pixels from objects further away. This is where the depth buffer comes into play. It contains the z depth information about every pixel in the color buffer. The depth value \((-1 \leq z \leq 1\) for a pixel is then compared to the z-value of any incoming pixel. If the z-value of the incoming pixel is less than that already in the buffer at that location, the new z-value is written to the depth buffer. At least in the normal mode of operation, but the actual depth test function can be set to something else. For instance, it can be set to require that incoming z-values are equal to the ones already in the buffer, which is useful when implementing the image space complex portal rendering algorithm. The z-buffer algorithm allows for primitives to be rendered in any order, which is very useful [4].

The stencil buffer can be used to create masks and limit rendering to certain areas of the screen (color buffer). It is usually of the same resolution as the color buffer but with fewer bits per value. Typical bit counts are 1 to 8 bits [4]. In OpenGL, it is possible to set a clear value, a test function and an operation to perform when the test succeeds. The buffer can then be cleared using the set clear value. Associated with the test function is a reference value. Together they allow the application to, for instance, only render to the screen pixels where the stencil values are set to 1.
The operation can be set for three possible outcomes of the stencil and depth test, and only affects the stencil buffer itself:

1. When the stencil test fails.
2. When the stencil test passes but the depth test fails.
3. When the stencil test and the depth test passes.

For any of the above outcomes any of the operations can be set. Some of the operations are keep, replace, increase and decrease. Keep leaves the stencil buffer unchanged, replace replaces the stencil value with the reference value and increase and decrease increases or decreases the current stencil value, respectively [9].

2.1 Cell and portal graph

The cell and portal graph is the data structure that enables portal rendering. It has two main node types: Portal and Cell. Every cell can have an arbitrary number of portals. In the implementation used for this thesis, number of portals per cell was restricted to max 100 out of programming convenience. Every portal has a target cell and a transform. For non-transformative portals, this might also be called an adjacency graph [4]. In the implementation used here, each cell also had a separate scenegraph. So when rasterizing a cell, the scene graph is rendered. The scene graph contains all the objects that should be rendered when the cell is rasterized.

2.2 Image space algorithm

This algorithm makes complex transformative portals possible. It performs visibility determination at the fragment stage, which is rather late in the rendering pipeline. It is recursive and can handle scenes where portals are seen through portals. However, if we allow ourselves to ignore the recursive case for now, here is a high level description of the simple (and typical) case where the viewer is located inside a cell, A, and sees a remote cell B through the portal P. See figure 4.

![Figure 4: Example scene. Viewer is in cell A and sees cell B through portal P. Illustration by Anton Petersson, 2013.](image)

In this example, cell A rasterizes to all green pixels while cell B rasterizes to all blue pixels. Also, cell A has a circular object in front of the portal. The portal itself is triangular. Here are the steps that the algorithm performs when rendering the above example.
Figure 5: From left to right: Depth buffer after step 2, depth buffer after step 3, color buffer after step 3, stencil buffer after step 4, color buffer after step 7. These images are not actually rendered buffers, but simplified illustrations. Illustrations by Anton Petersson 2013.

1. Determine the current cell, i.e. the one that the viewer is in. The current cell is A.

2. Write the portal P to the depth buffer.

3. Write the cell A to the color buffer as well as depth buffer. Notice that this might obscure some parts of the portal, generating lesser depth values for those pixels in the depth buffer. Also, since the depth buffer already contains the rasterized image of the portal, some pixels from the cell rasterization might get discarded. That means that the pixels in the color buffer corresponding to the portal will have the clear value, i.e. even though the portal itself isn’t rasterized to the color buffer, it’s still visible at this point. See the third image in figure 5. Notice the black pixels.

4. Rasterize the portal to the stencil buffer, but discard all pixels that don’t have the same depth values as the corresponding fragments in the depth buffer. This step creates a mask in the stencil buffer, which will be used in the next next step to discard all fragments that shouldn’t be visible through the portal.

5. Clear the depth buffer at the portal pixels.

6. Transform the camera by the matrix associated with the portal. This is what creates the illusion that cell B is just on the other side of the portal.

7. Rasterize the target cell, i.e. the cell that the portal is connected to, B. Fragments will be written to the color buffer using the mask in the stencil buffer. Fragments will also be depth tested both to the near and far depth buffers.

As mentioned above, the algorithm makes use of dual depth buffers. The reason for this is to avoid clipping errors. Since complex portals can have arbitrary shapes, and also since the algorithm isn’t making use of view frustum culling, there is a real possibility that geometry from the target cell that shouldn’t contribute to the final image gets rendered anyway. See figure 6. This is why a near depth buffer is needed, in addition to the far depth buffer. Standard hardware today has built in support for only one depth buffer as well as depth testing. This means that one of the buffers will need to be implemented some other way. In the implementation used here, the near


\[3\] Normally all objects share the same view transform. Transformative portals allow for scenes to be composed using multiple view transforms.
depth buffer was implemented using a depth texture and a specialized fragment shader. The depth buffer contents were copied to the depth texture whenever it was needed (typically once per cell with portals, per frame). The fragment shader would then lookup depth values from the depth texture while rendering objects in a cell, and discard those fragments that were closer than the portal surface they were seen through. The near depth test protects against the same clipping issues that Luebke mentions [11].

Another thing worth noting about this implementation is that in the recursive case, where portals are seen through other portals, it keeps track of the recursion level and keeps the stencil buffer updated with the current portal pixels set to the current recursion level. The stencil test is then set to only pass if the stored stencil value for a given pixel equals the stencil reference value. The stencil reference value is set to the current recursion level when rendering the objects of a cell. The result of this is that only the pixels of the current portal can be overwritten when rendering the target cell.

Also, in order to determine when to stop the recursion, the ARB_occlusion_query extension [6] is used. Before rendering the portals of a cell the second time, the extension is used to query the pipeline in order to determine how many pixels in the color buffer that would be affected by
rendering the portal. If the answer is zero for all portals in a cell, the recursion can stop.

### 2.3 Image space and geometry hybrid algorithm

This algorithm expands upon the image-space algorithm by adding view frustum culling. It works the same way, but imposes assumptions that the cells only contain geometry. The algorithm proposed here requires all objects in a cell including the portals to have bounding volumes. In this implementation, axis aligned bounding boxes (AABBs) were used. AABBs will often over estimate the size of the objects they encapsulate, but aren’t very complex to use in intersection tests. The over estimation characteristic allows for conservative culling. Conservative culling techniques never discard any objects that are part of the visible set [4], i.e. objects that should contribute to the final image. The downside of this is that they might include more objects than are part of the visible set, i.e. objects that shouldn’t contribute to the final image.

View frustum culling can be implemented in world space, testing objects against the planes of the frustum. If an object is outside the frustum, it can be discarded. In the context of portal rendering, view frustum culling can be used first to test all the objects in the active cell for visibility. Then it can be used to determine which of the cell’s portals that are visible. If a portal can be determined to be outside of the view frustum, the entire target cell of that portal can be discarded from rendering. When rendering the target cell of a portal, the view frustum can be shrunk to fit closely around the portal. A smaller frustum is beneficial in this context, as more objects will be classified as outside of the frustum by the intersection tests [4].

In the implementation used in this thesis, the frustum culling is performed by projecting all bounding volumes to the unit cube and generate two-dimensional AABBs. These AABBs are then intersection tested against the current “cull box”, which is initially the whole screen, and then shrinks around portals as their target cells are rendered. This algorithm uses the same cell and portal graph as the purely image-space based algorithm.

### 3 Experiment design

The experiment is implemented as a native application written in C++11. In order to take advantage of hardware acceleration the application uses OpenGL 3.3 with the ARB_occlusion_query extension [6]. The binary program was built using Visual Studio 2012 Express Desktop Edition. It was built in 64bit release mode with full compiler optimizations enabled. The experiments were conducted on a laptop computer running Windows 7 Premium. A battery charger was connected the entire time and all battery saving features were disabled. Most other programs were shut down before running any experiment. All tests were run on the following computer hardware:

<table>
<thead>
<tr>
<th>Computer manufacturer</th>
<th>Samsung</th>
</tr>
</thead>
<tbody>
<tr>
<td>Computer model name</td>
<td>530U3C-A01SE</td>
</tr>
<tr>
<td>CPU</td>
<td>Intel Core i5-3317U</td>
</tr>
<tr>
<td>Graphics Accelerator</td>
<td>Intel HD4000</td>
</tr>
<tr>
<td>Max resolution</td>
<td>1366x768</td>
</tr>
<tr>
<td>System RAM</td>
<td>8GB 1600MHz</td>
</tr>
</tbody>
</table>

A simple scene was set up using simple algorithms for generating cell and object geometry together with hardcoded positions. Portal geometry and positioning was entirely hardcoded and so
the cells and portals graph was created at compile time. Since the scenes were hardcoded the code had to be modified and recompiled for every test run. One test parameter at a time was examined. All tests focuses on comparing the pure image-space algorithm with the image-space and geometry hybrid algorithm, while setting the test parameter to different values. During testing, the application was run with double buffering enabled as the glfw documentation doesn’t mention any way of disabling it. Akenine-Mölle suggests to avoid double buffering when measuring performance as the screen update rate in conjunction with synchronization might hide away performance differences. The independent and dependent variables of this study are outlined in section 1.4.

3.1 Test specifications

3.1.1 Test 1: Number of visible objects

The first test focuses on number of visible objects, to see how the two algorithms compares when no objects can be successfully discarded from rendering. This is a way of measuring the cost of failed culling, i.e. how much performance is lost just trying to determine which objects are visible? How much faster could the scene be rendered if no culling attempt was made? The test is run with a screen resolution of 1366x768. Number of visible objects is an independent variable. Frame-time and frames per second are dependent variables.

3.1.2 Test 2: Number of out-of-view or occluded objects

This test is setup using a scene with two cells, A and B. The viewer is located in cell A and looking through a portal into cell B. Cell B has a (varying) number of objects out of view. The test is performed in fullscreen mode with a resolution of 1366x768. The scene does not have any visible objects. Number of out-of-view objects is an independent variable. Frame-time and frames per second are dependent variables.

3.1.3 Test 3: Clipping behavior

This test was not a performance test but rather a quality test. A number of scenes were created with specialized portal and object placement, testing various aspects of the algorithm for correctness. The screen resolution for this test is constant at 1366x768. Correctness is a dependent variable that depends on the use of view frustum culling, as an independent variable.

3.2 Selected tests

In an effort to limit the scope of the study not all of the relevant tests have been experimented with. Originally, six tests were defined. Although all of them are likely to provide interesting insights into the characteristics of the algorithms, there was not enough time to pursue all of them. A number of tests were selected. Those are the ones listed above. The other three are listed at section 7.1 under Future work.

4 Results

In this section the results from running all of the tests are summarized in tables. The performance benchmarking tests (test one and two) generated hundreds of values. The last 100 values for
every tested case has been automatically selected, reformatted and compiled into tables. Then, a
program automatically calculated median, mean, min and max values. Those are the values that
are presented here. All time values presented in this section are in milliseconds. As for test three,
the test results were trivial to verify manually by looking at the images produced by the algorithms.

4.1 Test 1: Number of visible objects

<table>
<thead>
<tr>
<th>Algorithm</th>
<th># Objects</th>
<th># Values</th>
<th>Median frame-time</th>
<th>Mean frame-time</th>
<th>min</th>
<th>max</th>
</tr>
</thead>
<tbody>
<tr>
<td>Image</td>
<td>100</td>
<td>100</td>
<td>11.818</td>
<td>11.819</td>
<td>11.733</td>
<td>11.917</td>
</tr>
<tr>
<td>Hybrid</td>
<td>100</td>
<td>100</td>
<td>11.791</td>
<td>11.729</td>
<td>11.655</td>
<td>11.929</td>
</tr>
<tr>
<td>Hybrid</td>
<td>300</td>
<td>100</td>
<td>14.282</td>
<td>13.936</td>
<td>13.000</td>
<td>14.912</td>
</tr>
</tbody>
</table>

4.2 Test 2: Number of out-of-view or occluded objects

<table>
<thead>
<tr>
<th>Algorithm</th>
<th># Objects</th>
<th># Values</th>
<th>Median frame-time</th>
<th>Mean frame-time</th>
<th>min</th>
<th>max</th>
</tr>
</thead>
<tbody>
<tr>
<td>Image</td>
<td>100</td>
<td>100</td>
<td>10.14</td>
<td>9.960</td>
<td>9.389</td>
<td>10.454</td>
</tr>
<tr>
<td>Hybrid</td>
<td>100</td>
<td>100</td>
<td>8.973</td>
<td>9.011</td>
<td>8.973</td>
<td>9.657</td>
</tr>
<tr>
<td>Image</td>
<td>500</td>
<td>100</td>
<td>10.903</td>
<td>10.864</td>
<td>10.787</td>
<td>11.022</td>
</tr>
<tr>
<td>Image</td>
<td>3000</td>
<td>100</td>
<td>27.405</td>
<td>31.834</td>
<td>26.684</td>
<td>39.000</td>
</tr>
</tbody>
</table>

4.3 Test 3: Clipping behavior

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Correct clipping</th>
</tr>
</thead>
<tbody>
<tr>
<td>Image</td>
<td>yes</td>
</tr>
<tr>
<td>Hybrid</td>
<td>yes</td>
</tr>
</tbody>
</table>

5 Discussion

So how does the collected test results help answer the research questions? The data collected
from test one shows that a geometry and image-space hybrid algorithm doesn’t necessarily have
to be slower than a pure image-space algorithm when rendering only visible objects through a
portal. Increasing the number of visible objects seems to slow down both algorithms by about the
same amount. An image-space portal rendering algorithm does not benefit from any gross culling
of objects in a scenario with only visible objects. This is very much expected. It is a waste of
computing power to try to find objects that can be discarded, when no object can be discarded.
Another thing to take away from the results of test one is by how little the frame rate decreases
as the number of objects are increased by a factor 2 or even 3. It does have a measurable impact
though.

The results from test 2 show very clearly the performance increase that geometry based opera-
tions can provide. The hybrid algorithm is measurably faster already at 100 discarded objects.
The performance gain achieved when using view frustum culling grows as the number of objects
increases. At 3000 objects, the hybrid algorithm is almost three times faster than the image-space algorithm. And only simple textured cubes were used in this test. The performance gain is likely to be even bigger when rendering more complex objects (for instance with advanced shading at the fragment level). It’s interesting that for discarded objects, the algorithms seem to scale similarly from 100 to 500 objects, but when testing with 3000 objects, the hybrid algorithm is much faster. This might be due to a performance bottleneck in the pipeline; at a certain number of processed objects, the actual object processing becomes the new (bigger) bottleneck. This could have been further analyzed using pipeline optimization techniques. Akenine-Möller describes some ways to locate the performance bottleneck in a rendering pipeline [4]. It would be interesting to test this further with other object quantities, in order to better identify how the algorithm scales.

Geometry based culling techniques are nothing new, especially not view frustum culling. Transformative portals aren’t new either, see section 1.1. Luebke’s and Teller’s work, among others, show that portal rendering can be done with a pure geometry approach. However, the clipping issues become difficult to handle and often leads to restrictions. For instance, a portal might need to be convex and lie in a plane. Transformative portals might need to be stuck to a wall of a cell. Lowe’s and Datta’s image-space algorithm allows for complex transformative portals without such restrictions. Their algorithm, however, focuses purely on correctness and also does not assume there is any geometry at all. In their algorithm, cells can be composed of images or anything that can be rasterized. What this hybrid algorithm does is impose such assumptions or limitations on an image-space algorithm. The hybrid algorithm does not work for non-geometrical scenes. The upside of imposing such restrictions is that it allows for further optimizations by quickly testing geometric objects for visibility. This is actually a typical case of trading flexibility for speed, while preserving quality.

It should be interesting to compare these results to a geometry based algorithm. That’s actually something that was never really considered during this thesis. A geometry-based solution should always be more efficient but perhaps a well optimized hybrid algorithm can come close. That’s assuming only convex, planar portals are used. A geometry-based solution capable of handling complex portals is probably too difficult or inefficient to implement correctly.

Other than that, the method used here should be an adequate choice for answering the research questions. In order for any worthwhile comparison between different algorithms of this complexity (especially utilizing graphics hardware) to be performed, both algorithms really need to be implemented and tried. It’s too difficult to theoretically come to valid conclusions, especially when it comes to performance.

6 Conclusions

The conclusions from this study are that an image-space portal rendering algorithm can benefit from lending some features from a geometry based algorithm. Greater performance can be achieved through view frustum culling in particular, without sacrificing image quality. The hybrid algorithm showed that it can be three times faster than the pure image-space algorithm for a given scene complexity. In no scenario tested in this thesis did the geometry-based additions cause any significant decrease in performance.
7 Future work

This section contains a collection of thoughts and ideas on how the work in this study might be expanded upon.

The first thing that comes to mind is to find out more about the actual performance characteristics of the hybrid portal rendering algorithm as outlined in section 2.3. This could be done by experimenting with the tests defined in section 7.1; this thesis doesn’t cover them. Also it would be very interesting to do some general pipeline optimization and evaluation, to find current bottlenecks. It could be interesting to look into whether or not frame buffer objects or ARB_occlusion_query2 can provide additional performance. Also, one can experiment with hierarchial bounding volumes to minimize the number of intersection tests.

Other things that might be interesting is to consider whether or not light sources on one side of a transformative portal should affect the shading of the target cell, and if there are any special things to consider if yes. This could be interesting both from a game design perspective and a computer graphics one.

Lastly, how to handle objects intersecting with a transformative portal, for instance when traveling through it seamlessly. This was supposed to be a part of this thesis from the start but was rejected early because of the time constraints. How should an object, a player character for instance, appear to a viewer when the object has partly travelled through the portal? How can the renderer make sure the object is rendered correctly for all viewers? Maybe the object needs to temporarily have multiple world positions. Also consider the fact that portals are one-way. Two-way portals are simulated using two or more portals. The issue of clipping an object correctly when seamlessly travelling through a portal is also interesting. If a viewer is located next to a transformative portal, which might also be complex, how can the object passing through the portal be correctly clipped or masked so that it doesn’t stick out on the other side of the portal geometry? There might also be some interesting game design issues related to this.

7.1 Tests not included in this thesis

7.1.1 Test 4: Number of portal recursions

In this test a series of scenes with increasing number of portal recursions are examined. Number of visible objects in every cell is constant. Number of objects out of view or occluded are also constant. Screen resolution is 1366x768.

7.1.2 Test 5: Number of discarded portals

In this test a series of scenes with increasing number of discarded/culled portals are evaluated. No portal recursion occurs. A discarded portal means an entire cell can be discarded from rendering. This test compares how quickly the two algorithms can determine that a portal can be discarded. The scenes does not include any objects in any of the cells.

7.1.3 Test 6: Screen resolution

In this test a single ”typical” scene with two cells and 3 portals is used. The portals are all located in the same cell as the viewer. One of them is visible, the other two aren’t. Number of objects in/out of view are kept constant. Various screen resolutions are tested.
8 Acknowledgments

I would like to thank my supervisor Charlotte Svennersten for her guidance and feedback on things such as input, variables and analysis which undoubtedly led to a better thesis. I would also like to thank her for all the encouragement, which I think really made a difference.

References


