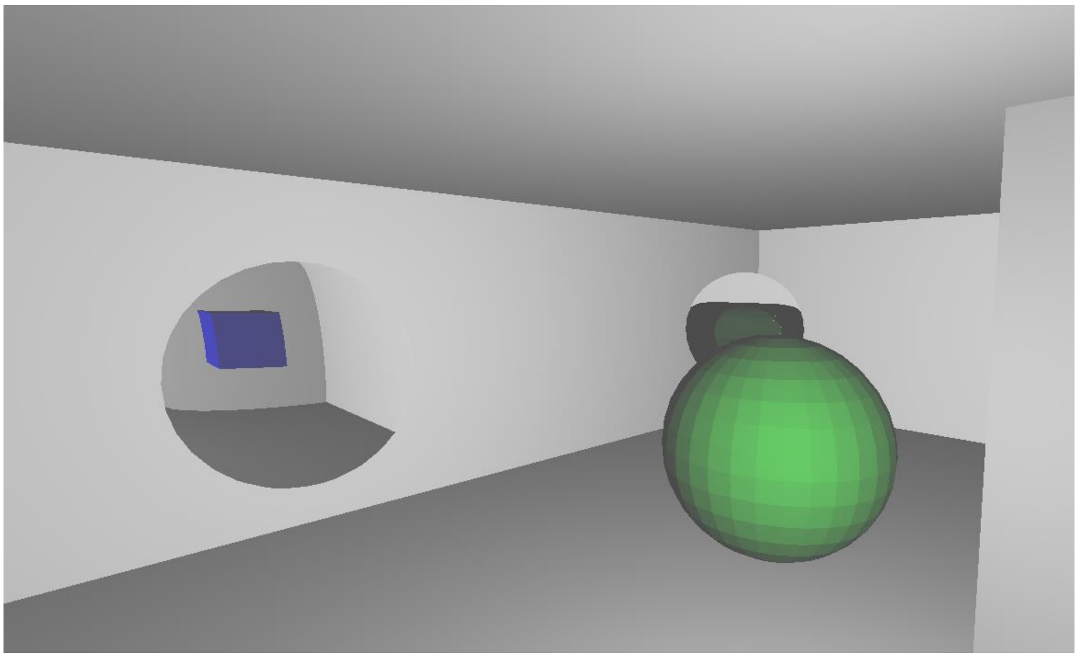
Can real-time raytracing contribute to or create gameplay features?



September 2020

Supervisor: [redacted]

Author: [redacted]

Student ID: [redacted]

Abstract

With the advent of NVIDIAs RTX line of graphics cards, raytracing features have become widely accessible to mainstream audiences and developers. While modern day video games make use of raytracing features, these are limited to enhancing graphical fidelity and used in conjunction with traditional rasterization rendering. This project aims to utilise GPU-based raytracing to create gameplay features and compare and evaluate them against existing implementations in traditional game engines that use raster rendering. One of the main features presented is a fully ray-traced “Portal” and how it is implementation in raytracing differs from a rasterization-based alternative.

Acknowledgements

[redacted]

Contents

Figures

Introduction

Since the inclusion of NVIDIAs RTX line of graphics cards, especially with the recent announcement of their latest series, raytracing features are becoming more prevalent in modern video games than ever before, however due to the substantial impact caused by real-time raytracing these features are used in conjunction with traditional raster rendering to improve the graphical quality of existing environments or create more realistic effects rather than as an outright replacement. Although the term raytracing is usually used in the context of computer graphics, the theory of it was first applied in the early 16th century to paint environments using perspective projection(Albrecht Durer, 1538). This practice was further expanded on in 1968 using an algorithm which utilised perspective projection and introduced rays and object intersections to view an environment(Arthur Appel, 1968). The algorithm was later given the term "ray casting"(Scott Roth, 1982) and would find use in early video games such as WolfenStein 3D(ID Software, 1982). A similar algorithm named "Recursive ray tracing" expanded on the original, adding Refraction, reflection and shadow rays which could produce realistic renders such as the one shown in figure 1.

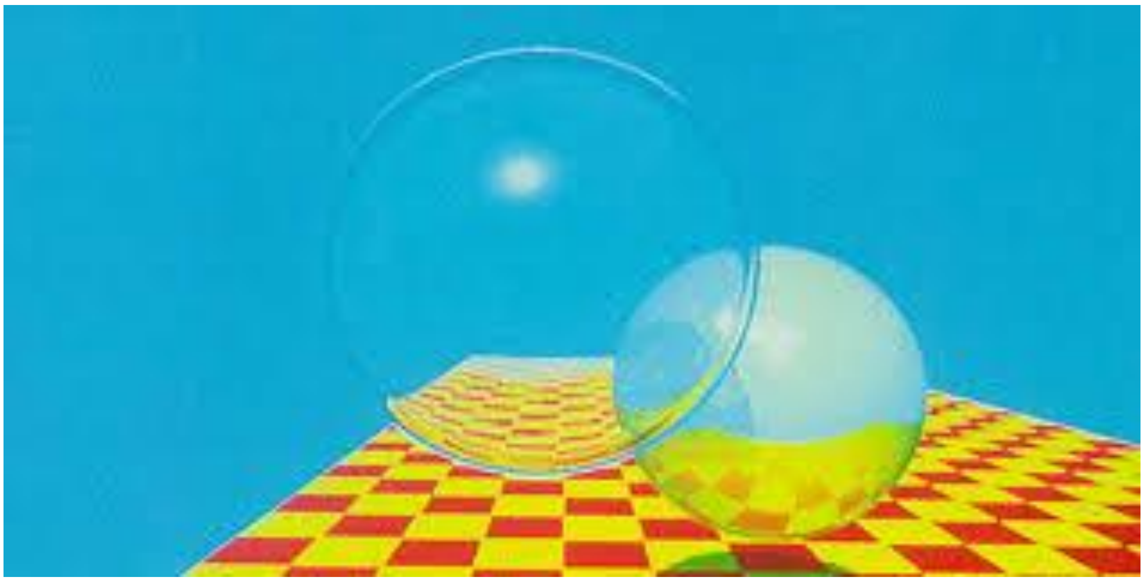


Figure 1 Render of a scene using raytracing Techniques, Turner Whitted, 1979

These ray traced effects such as reflections and soft shadows are what modern graphics cards are now able to produce in real-time, although they still require using a both a hybrid of raytracing and raster rendering in order for the applications to run at acceptable frame-rates. Before being used in real-time applications, ray tracing was a favoured method for offline rendering, specifically in animated movies as it could sequentially generate each frame with realistic lighting and visuals that in some cases could take over 15 hours to generate(Per Christensen et al, 2006). Another benefit of raytracing is that it can simulate camera based visual effects such as depth of field or specific aperture shapes(Kolb et al, 1995) which are commonly used in animated films. The visual effects created using raytracing are commonly used to create realistic imagery or to improve upon the visual created by rasterization in a hybrid format, however raytracing can also be used to create interesting visual effects that can be used in a gameplay environment. This paper aims to present two implementations of portals using both rasterization and raytraced rendering techniques and compare the technical aspects of both. Another aim of the project is to identify any new features that are presented when implementing portals using raytracing and if they can be used in the context of gameplay.

Literature review

Raytracing can be simplified into two aspects. Ray intersections which determines if and how a ray has intersected an object, and a shading algorithm which determines the pixel colour to return from the object based on the object's material and the lights in a scene. The shading algorithm an become increasingly complex as a more realistic result is required, however the ray intersection algorithms have stayed relatively the same over the years with minor changes to reduce the computational requirements.

Ray Intersections

A ray ***R*** can be defined with an origin ***O*** and a direction ***(t)D*** as

$$R = O + (t)D$$

where the direction is normalized, and ***t*** is a position along the ray. A ray intersection is where the point along a ray intersects with the geometry in its path. One common example is a ray-triangle intersection which is relatively quick to performance and allows for the use of triangle meshes(3D models that are made up of multiple triangles). A triangle is made up of three points *v0*, *v1*, and *v2* each representing the corner of the triangle. Testing if a ray intersects with a triangle can be done by testing if a point exists within it. A point **P** in a triangle can be defined as

$$P(u, v) = (1 - u - v) * v0 + u * p1 + v * p2$$

where both u and v are both equal to or bigger than zero, but added together are less than one(Moller, Trumbore, 1997). A point from a ray that intersects with a point from a triangle can be defined as

$$R(t) = (1 - u - v) * v0 + u + p1 + v * p2$$

While the Moller-Trumbore algorithm offers a fast intersection computation, there are algorithms that claim to run even faster in almost all cases(Weber, Baldwin, 2016). Ray intersections can be done with a variety of different primitives including boxes, which can be used to contain other primitives in a bounding volume used in acceleration structures.

GPU Raytracing

Most ray tracing calculations were traditionally done on the CPU as they could take advantage of ray-traced parallelism(Turner Whitted,1979) where each individual ray could be traced independently of others. Many developments to the raytracing algorithm were made over the years to reduce the time to render for each frame, such as using multiprocessor solutions(Stuart A Green, Derek J Paddon, 1990) and parallel processing(Kobayashi et al, 1987). In terms of real-time raytracing, the BRL-CAD modelling system utilized REMRT/RT tool which could render at several frames per second(Mike Yuus, 1987). 2010 saw a change in the hardware used in raytracing, as NVIDIA launched their Optix ray tracing engine that could utilise their GPUs(Graphical Processing Unit) and other “highly parallel architectures”(Steven G Parker et al, 2010) to run instances of fully ray-traced environments in real-time. The change to GPUs was also accelerated due to the use of GPU-based AI de-noising present in both the Nvidia Optix engine and their latest RTX line of graphics cards to clear up noise produced by lighting algorithms such as Global illumination. Monte-Carlo based global illumination or path-tracing is an algorithm for producing realistic lighting of an environment, however it is limited by the impact on performance it brings to the application as the number rays required by the algorithm increases. In order to maintain an acceptable framerate, the number of rays is decreased, however this leaves behind “noise” in an image where a ray has not intersected(Chakravarty R. Alla Chaitanya et al, 2017). The answer to this is to use an AI trained to remove the noise from the final image, which produces similar results but with much higher performance, allowing this algorithm to be used in real-time and modern video games. An example of the before and after of AI-based denoising can be seen in figure 2.

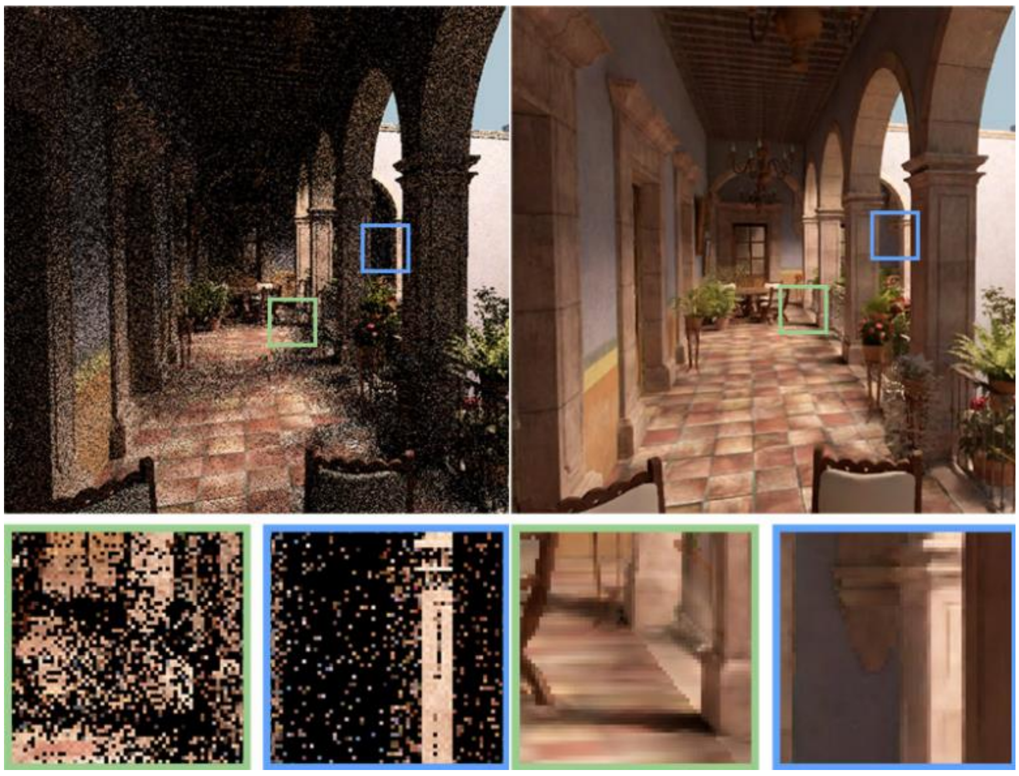


Figure 2, Before(left) and after(right) of ai-denoising in a scene, Chakravarty R. Alla Chaitanya et al, 2017

Voxel engines have also shown to be a good base for implementing raytraced rendering(Alexander Majercik et al, 2018) as the methods for traversing voxel environments mirror how a ray traverses an acceleration structure. Voxel environments that are represented as cubes can utilise AABBs(Axis-Aligned Bounding Box) which provide a fast ray-intersection test while also being stored in data structures such as Octrees or KD-trees that can be used as the acceleration structure for the raytracing algorithm making raytracing a viable alternative to raster rendering when used in voxel engines. Though real-time raytracing has been used in many modern-day video games and applications, most of its uses are to expand or add to existing graphical effects.

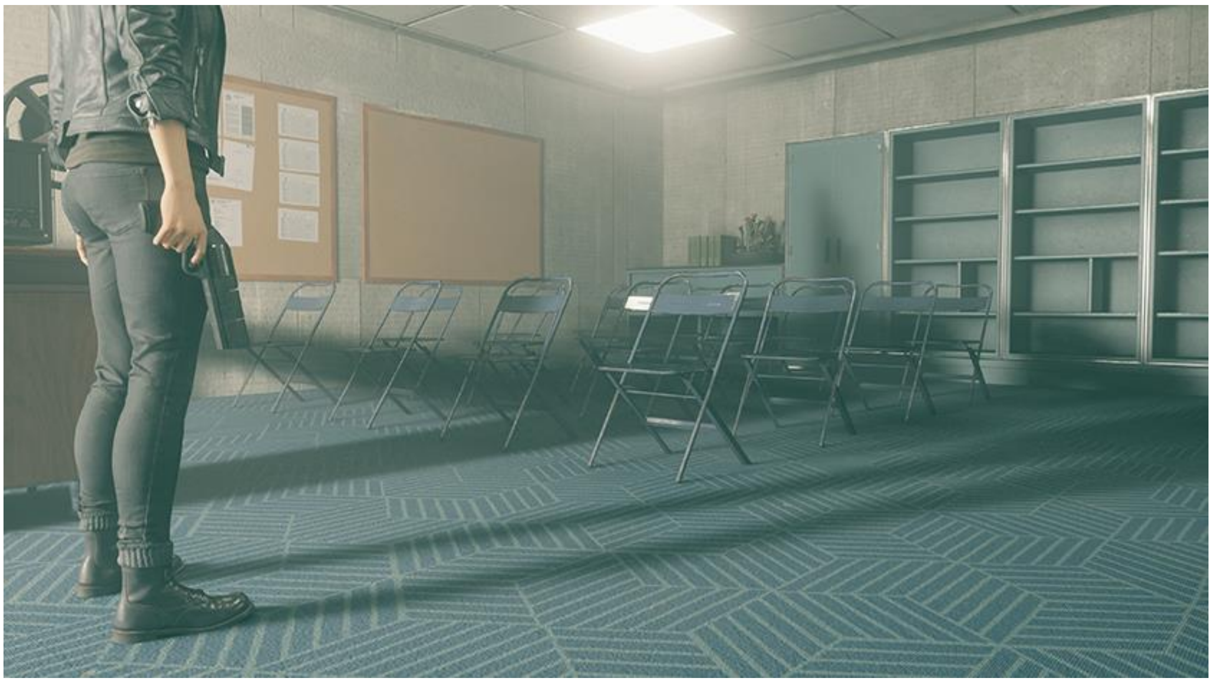


Figure 3, Real-time raytraced shadows in Control, Nvidia, 2019

One of the effects of raytracing is real-time shadows on objects that would otherwise not support or use them which allows players to view the shadows on objects that are not in their view. Shadow maps are used to create shadows in traditional raster rendering which work well for scenes with a small number of large objects, however when there are numerous small objects in a scene these shadows are usually omitted due to the heavy performance cost. These shadows can be displayed or have their detail increased by using real-time raytraced shadowing as shown in Figure 3 with less of an impact on performance(Nvidia, 2019). As well as Realtime shadows, the windows 10 edition of Minecraft(Mojang, 2020) launched a beta version that includes global illumination and real-time reflections. Both graphical features can alter how a game can be played, for instance providing light to an area indirectly, or reflecting a view to the player they would not otherwise be able to see. Stay in the Light(Sunside Games,

2019) is a video game that expands on real-time reflections by using a mirror to view the scene behind a player, looking out for monsters or points of interest that the player otherwise cannot view such as in Figure 4. While reflections can be produced using raster-rendering, these are usually reflecting static environments and cannot reflect the dynamic objects that real-time raytracing can. The importance of viewing the scene using the visual effect means that the effect itself has become a gameplay feature that the player must use or interact with in order to progress.

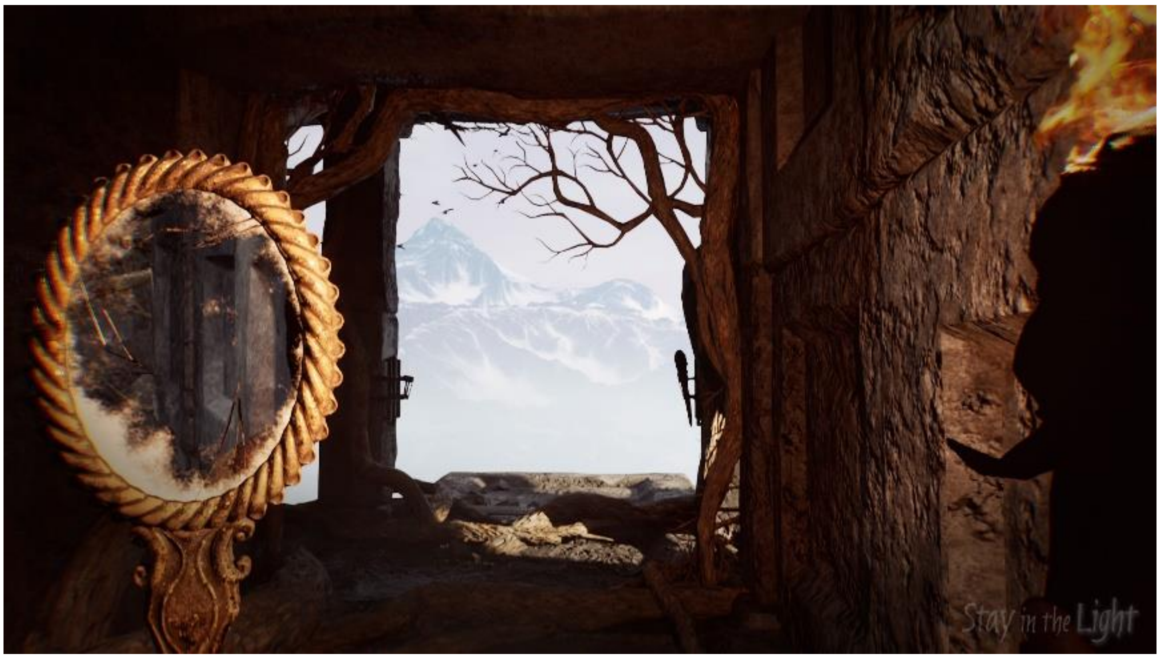


Figure 4, Realtime raytraced reflections in Stay in The Light, Sunside Games, 2019

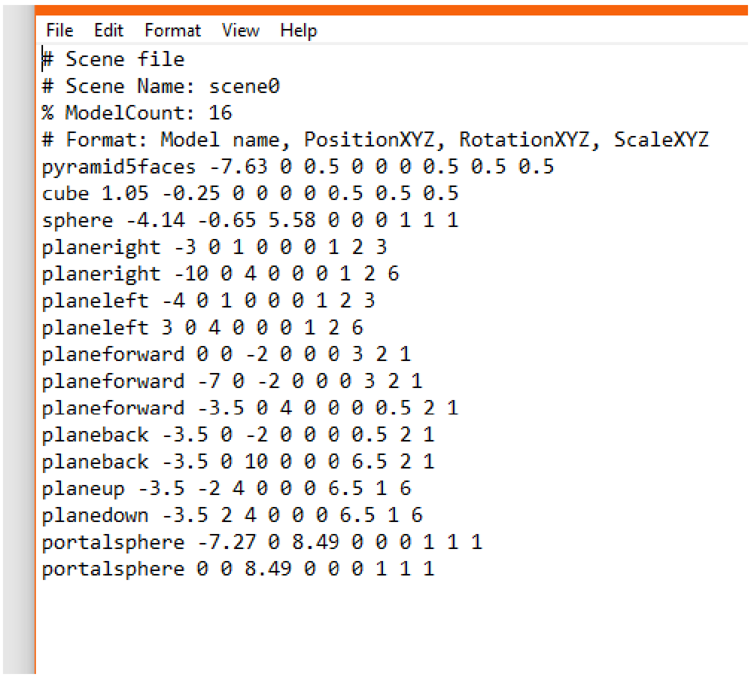
Visual effects like these can be used to create gameplay features and even be the centre-point of a game. Narbacular Drop(Nuclear Monkey Software, 2005) is a game which allowed a player to place two linking portal objects onto the geometry of a scene. The view from one side of a portal is visible from the other, which the player uses in order to complete puzzles. The team behind Narbacular Drop eventually went on to make the spiritual successor Portal(Valve, 2007) which built upon the existing techniques to create a puzzle game using the portals mechanics. While both games contained portals, neither of them uses a form of raytracing to visualise them and are both done using raster rendering. Narbacular drop uses render textures to render its portal views, while Portal uses stencil buffers(David Kircher, 2018). The render texture method uses multiple cameras to render the scene, then applying the output to the portal objects texture before finally rendering the entire scene. If the view of a portal contains itself, this becomes a recursive implementation where multiple renders at different offsets are required and applied to a view until this recursion value hits zero and the final output can be rendered from the main camera. Although there are not many situations where a player is required to look through multiple recursions, there are instances where the player must use visual information from one portal view in order to progress throughout the level which they would not otherwise be able to achieve.

Though render textures are relatively simple to implement, they suffer from low performance at high recursion values(David Kircher, 2018) as a single frame may require multiple renders. Another issue of render textures is that visual effects such as anti-aliasing are not available on the portal views due to them being rendered to a flat texture instead of buffers. On the other hand, stencil rendered portals output the cameras view to the stencil buffer instead of a texture, and the only viewable section of the output is from the portals model. The scenes first render pass is of opaque objects, which allows the cut out of the portal model to be used in the stencil buffer. This method also means that visual effects such as anti-aliasing will work as instead of rendering a texture, which allows the view to be rendered in full by the main camera. Although the visual effect of the portals is convincing, there are issues created due to the implementation being created using raster rendering. One example is that after the second recursion, the view of the next recursive portal is copied from a previous one and stretched to fit the view due to the heavy impact on performance required to render multiple recursions. Another consequence of the portals using raster rendering is that light from one side of a portal cannot affect the environment of its counterpart, for instance lighting up a dark room that contains one portal, and placing another in a lighter room to allow the light to pass through.

Methods and Materials

As real-time raytracing is relatively recent advancement there are few game engine or APIs capable of rendering a fully ray-traced or even hybrid rendered environment, however since late 2018 DirectX12 has included an extension called DirectX Raytracing into its development build which allows for developers to create and view an environment using either fully ray-traced or hybrid rendering. Though raytracing and rasterization are separate techniques for rendering, their results are produced using similar graphics pipelines. DirectX12 stores and uses the data of a model using buffers, descriptors, descriptor tables and a descriptor heap. The buffers store both the index, vertex and normal data of a model as well as the uv data if the application supports textures. Each buffer is bound to a descriptor which describes the what the buffer is and its contents to the GPU. Multiple descriptors of the same types, such as constant buffer views(CBVs), Unordered access views(UAVs) and shader resource views(SRVs) can be grouped together to form a descriptor table. The descriptor heap is a collection of descriptor tables grouped together, usually in a single frame for fast access on the GPU. Buffers are not limited to storing data for a single object and in this instance the index or vertex data of multiple objects could be stored in a single buffer, and using individual data using this method requires storing and accessing the start index for each separate model. While storing the data into a single buffer is more performant, it also means that accessing the data becomes more difficult. This situation is where a developer needs to balance the usability versus performance while also making sure the code remains readable, where the correct answer is highly dependent on the situation. In this instance, the application splits up each model’s index and vertex data into their own buffers, as there are areas in the project where the data on individual models are required such as identifying which object should be shaded as a portal and which object should be shaded using default diffuse lighting.

The Models used in the implementations were created in blender 2.9(Blender Foundation, 2020), as it can easily export models to the different typed required by both DirectX12 and Unity3D. Blender also contains the functionality to automatically triangulate a model, which creates two triangles from a ‘face’. Although Unity can automatically triangulate faces of imported models, this functionality is not present in DirectX12 and the imported models are read in triangle-by-triangle. Model data such as the index and vertex information are read in from the models file using a custom file reader for specific file types such as .PLY or .OBJ file formats. The .PLY format was chosen because it contains the vertex normal for each individual vertex, whereas other formats such as .OBJ reuse the vertex normal for a single triangle which made it simple to implement. The downside to using this format is that Unity3D cannot read in .PLY files without an extension, which means that the same model needs to be created twitch into two different formats used by each application. Creating and reading in multiple objects separately is not efficient therefore the project required a method of combining these files into a single level file to read in instead. While a level editor can be created in directX12, there are already existing engines that contain level editors to design and edit an environment in. These engines can also export the required file format needed, saving development time in on an optional component for this project. The unity3D engine provides all the tool required to edit a level without any additional changes required, and its scripting component is used to create and export a .TXT file that contains the information for each models filename, position, rotation and scale as shown in Figure 5. To import the level into the directX12 application, a scene loader reads in the .TXT file and uses the filenames included to load the corresponding model data into buffers. The position, rotation and scale data for each model is applied for each model instance during the building of the acceleration structure, which is a component of the main raytracing pipeline.



```
# Scene file
# Scene Name: scene0
% ModelCount: 16
# Format: Model name, PositionXYZ, RotationXYZ, ScaleXYZ
pyramid5faces -7.63 0 0.5 0 0 0 0.5 0.5 0.5
cube 1.05 -0.25 0 0 0 0 0.5 0.5 0.5
sphere -4.14 -0.65 5.58 0 0 0 1 1 1
planeright -3 0 1 0 0 0 1 2 3
planeright -10 0 4 0 0 0 1 2 6
planeleft -4 0 1 0 0 0 1 2 3
planeleft 3 0 4 0 0 0 1 2 6
planeforward 0 0 -2 0 0 0 3 2 1
planeforward -7 0 -2 0 0 0 3 2 1
planeforward -3.5 0 4 0 0 0 0.5 2 1
planeback -3.5 0 -2 0 0 0 0.5 2 1
planeback -3.5 0 10 0 0 0 6.5 2 1
planeup -3.5 -2 4 0 0 0 6.5 1 6
planedown -3.5 2 4 0 0 0 6.5 1 6
portalsphere -7.27 0 8.49 0 0 0 1 1 1
portalsphere 0 0 8.49 0 0 0 1 1 1
```

Figure 5 File Format for reading in the objects in a scene, Authors Image, 2020

Using Unity3D as the level editor for the DirectX12 application does mean that making changes to the level requires using two different programs and exporting and important a new scene each time, however it also means that level designs can be made early on and focus can be put into the portal implementations instead. Unity3D also contains scenes to switch between which for this project allows one scene to be used as a level editor, and another scene to be used as an implementation of raster portals. Using Unity3D for the raster portals allows the directX12 application to focus on making a fully raytraced environment without worrying about implementing fallbacks to using raster rendering instead.

Acceleration structures can be described as a way to reduce the computational complexity of a scene by lowering the amount of necessary object intersections for each ray which is especially important for real-time raytracing as in some cases, up to 95% of the time spent rendering a scene is spent on the computation of object intersections(Akira Fujimoto, 1983). The structures job is to group up objects together so that a ray can easily tell what objects it is likely to intersect. There are various types of acceleration structures such as kd-trees or uniform grids, however directX12’s raytracing pipeline uses bounding volume hierarchies(BVH). The idea behind a BVH is that the objects in a scene are divided up and grouped into a bounding volumes, such as an axis-aligned bounding box, or a bounding sphere. Multiple bounding volumes can again be grouped into their own containing bounding volumes creating a tree structure and ends when the process reaches the root of the tree containing all objects in the scene. When a ray intersects with a bounding volume, it will then check to see which of the two child volumes or objects it should intersect with, therefore any of the objects or volumes contained in the un-intersected volume need no further consideration(Timothy L. Kay, 1986).

The acceleration structure in this implementation is split up into both bottom level acceleration structures(BLAS) and top-level acceleration structures(TLAS), with the BLAS containing the mesh information of an object including its vertices, indices and their data formats. This is also where the object is given a geometry type, for instance for triangle geometry or procedural geometry and if the object should render as opaque or not. After this data is read in, the bottom level structure also creates instance descriptors for each object which are used to set how it should be positioned, rotated and scaled in the scene. The top level acceleration structure is made up of multiple bottom level structures that make up the geometry of an entire level, it also contains the ability to create and set its own instances which can be used to duplicate groups of models using existing vertex and index buffers which can save on memory usage. This multi-layered acceleration structure is efficient as the TLAS can search for an individual object, while the BLAS can search for an individual triangle in that object for a ray to hit.

The pipeline contains root signatures to bind the raytracing resources to the graphics pipeline, there are two primary root signatures each containing their own individual resources that are separated into global and local parameters. While both root signature types are used to bind resources to the GPU, local root signatures contain data for the application in shaders. Global parameters include, the output view for the final raytracing output, the acceleration structure and the constant scene buffers, while the local parameters have two constant parameters for the models and portals, as well as a parameter for the vertex and index buffers. Part of the pipeline is to construct the shader tables and to apply their constant buffers at the start of the application. As models and objects can be shaded in different ways, the application must identify and construct shader tables for each unique shader including their ray generation, hit and miss shaders. Both models and portals contain constant buffers that determine how the model should appear, but portals also contain a constant buffer that links to another portal which is unique so must be separated into its own hit group. The raytracing shader determines how the scene and its objects are depicted onto the screen. The ray generation function is used to create primary rays from the camera and apply them to the final render texture output. If a ray intersects and object it will call the corresponding hit shader for that object which determines how the object should be coloured, then return this colour to the ray gen function. If no object is hit there is usually a miss shader which is used to return a default colour used at the background of a scene. Using a constant buffer to store the portal locations was a simple solution which allows for quick access to this data while in the raytracing shader, however it also means that updating this information at runtime requires rebuilding the shader tables which is where the constant buffers are applied.

Implementations

The current implementations include the visuals for portals in both a raster and fully raytraced engine. A portal in these implementations is a fictional doorway between two linked point, where the views of one portal can be seen through another. The raster implementation of portals is faithful to the implementation in Narbacular Drop which uses render textures, however the raytraced version introduces the use of portals on three-dimensional objects that are not possible or very difficult to produce using rasterization. The render texture method of portals is relatively simple and quick to implement which makes it a suitable choice for prototyping the effect. Both the Unity3D and DirectX12 engine uses the same level in order to better compare the similarities and differences between each implementations. The level is an enclosed room made up of planes, three objects including a cube, five-sided pyramid and a uv-sphere(sphere with visible faces) as well as two portals applied to various meshes as shown in figure 6.

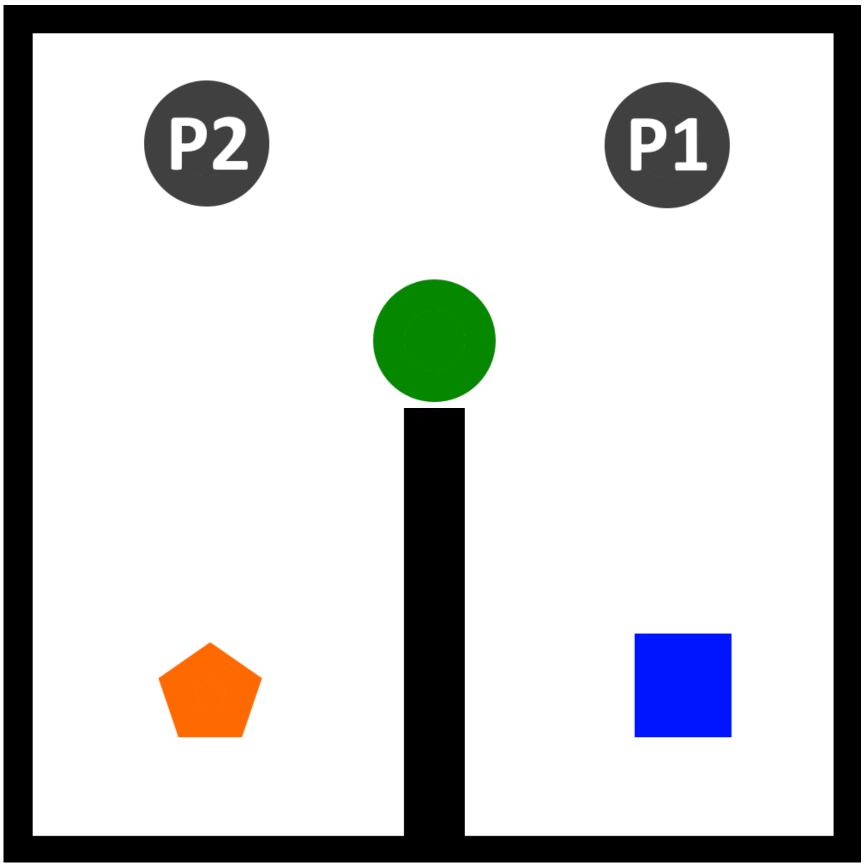


Figure 6, Top-Down view design of the level used in both portal implementations, Authors Image, 2020

The design of the level allows the user to understand the views returned from a portal due to it being clear which side of the room contains which shape. For example, if the user's camera is next to the blue sphere, then the only way to view the orange pyramid is to look through the portal, and if the portal was a mirrored reflection, then they would see the blue cube instead. While more interesting levels could be created, the current level provides enough information to the user for the implementations to be developed. Another benefit to creating small and relatively simple scenes is that it becomes easier to stop irregularities when developing the implementations which may go unnoticed in larger scale levels.

Raster Portal Implementation

A portal in the Unity3D engine is contains a polygon mesh, a unique camera, render texture and a fragment shader which is applied to the model. The only mesh supported in this implementation is a quad as one instance of a portal effect can only be applied to a single flat surface which contains the uv data from zero to one for each axis. The fragment shader uses the uv coordinates to only display the view from the portal mesh instead of a stretched view of the entirety of the cameras render. Each portal camera is a child of its portal object and its position and rotation are determined by the corresponding position and rotation of the primary camera used to view the scene creating a matrix transform that positions the cameras relative to the players view as seen in figure 7. Each camera is assigned to a portal and holds a reference to both the portal it is linked to as well as the main camera. While this method would benefit from being automated, it works in the current level which only contains a fixed number portals. In the unity editor, the cameras forward directions are shown using a green ray for the main camera and a red ray for the portal's camera.

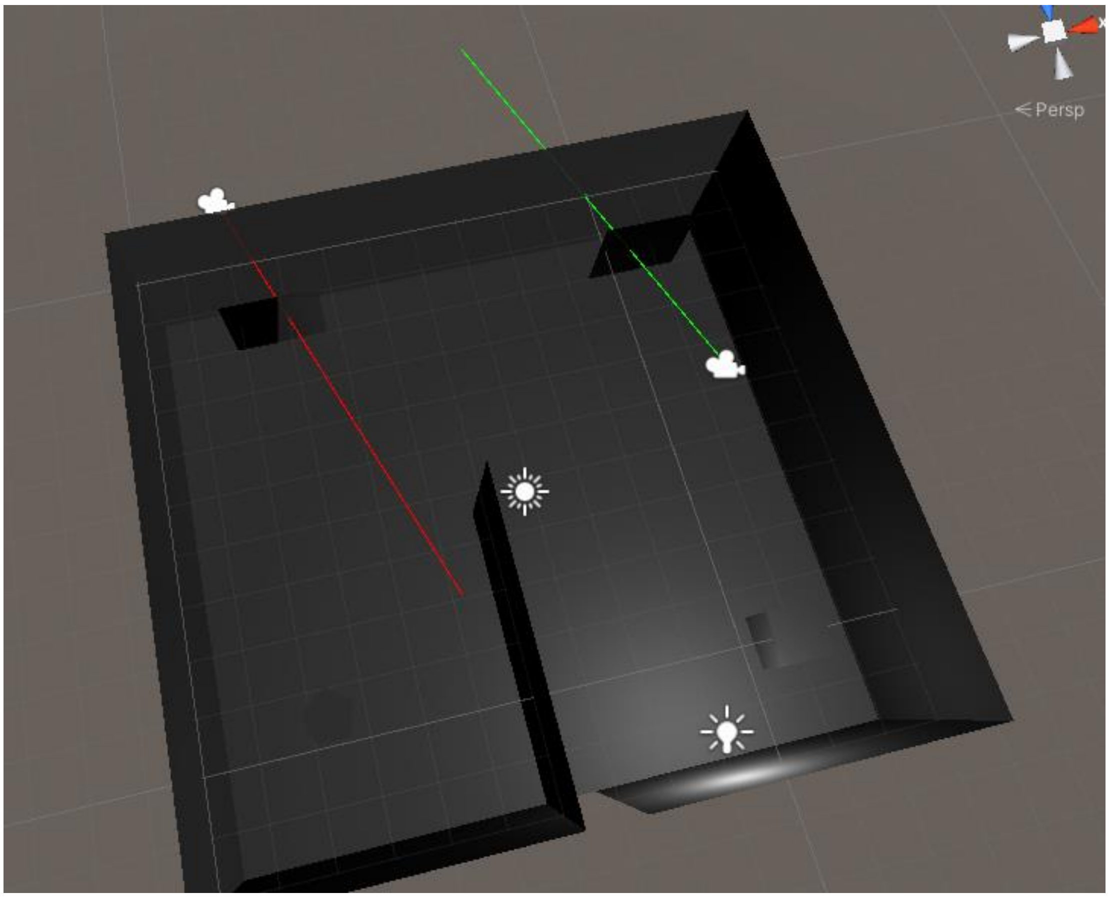


Figure 7, Player cameras view direction(green) and linked portals rotated view direction to match(red) in Unity3D, Authors Image, 2020

As the player camera moves and rotates around a scene, the portal cameras will match the view and render their individual views to a render texture before the main camera renders a frame. The views from each portal camera must be rendered first in order to view them in the final camera output. An example of the visuals produced with the effect can be seen in figure 8, where the view from one portal can be seen on the texture of another. Each portal contains a render texture component which stores the output of each camera, and this texture is then applied to the surface of a linked portal. If a portal can be seen from the view of another portal, for instance if two portals are facing each other than a new method is required in order to render multiple iterations depending on the angle and view of the main camera. For each unique view from a portal, an additional camera is required to make another render before the previous camera, which stops when either the final camera view no longer contains a portal, if a user defined constraint is finally reached. The constraint is usually in the form of a number which stops the number of renders going above a threshold where performance of the renders negatively impacts the performance of the application. Due to the constraint stopping additional renders, the view from the portal can contain blank spots where either a fallback shader or undefined behaviour is shown instead, however methods exist to hide or alleviate this problem such as applying fog or lowering the resolution of the view for each iteration of the portal views until they become indiscernible. In order to view these recursions, the portals must be setup to render the recursions back-to-front until the final view from the main camera is rendered. This can become difficult to setup, as each camera needs to know if it should render before another and can have a large impact on the performance of the application as more cameras and needed to render the scene. A solution to this problem is presented by David Kircher who ran into this issue during the development of Portal, where after the second camera recursion, any future recursions would reuse the same render from the previous camera and stretch the view to appear like a new recursion. Although this produces a “fake” view to the player, Kircher decided that the performance gained using the method was worthwhile as there were no situations in the game where the player was required to look through more than one or two recursions of a portal, therefore it would not impact gameplay while boosting the performance of the program in almost all instances.

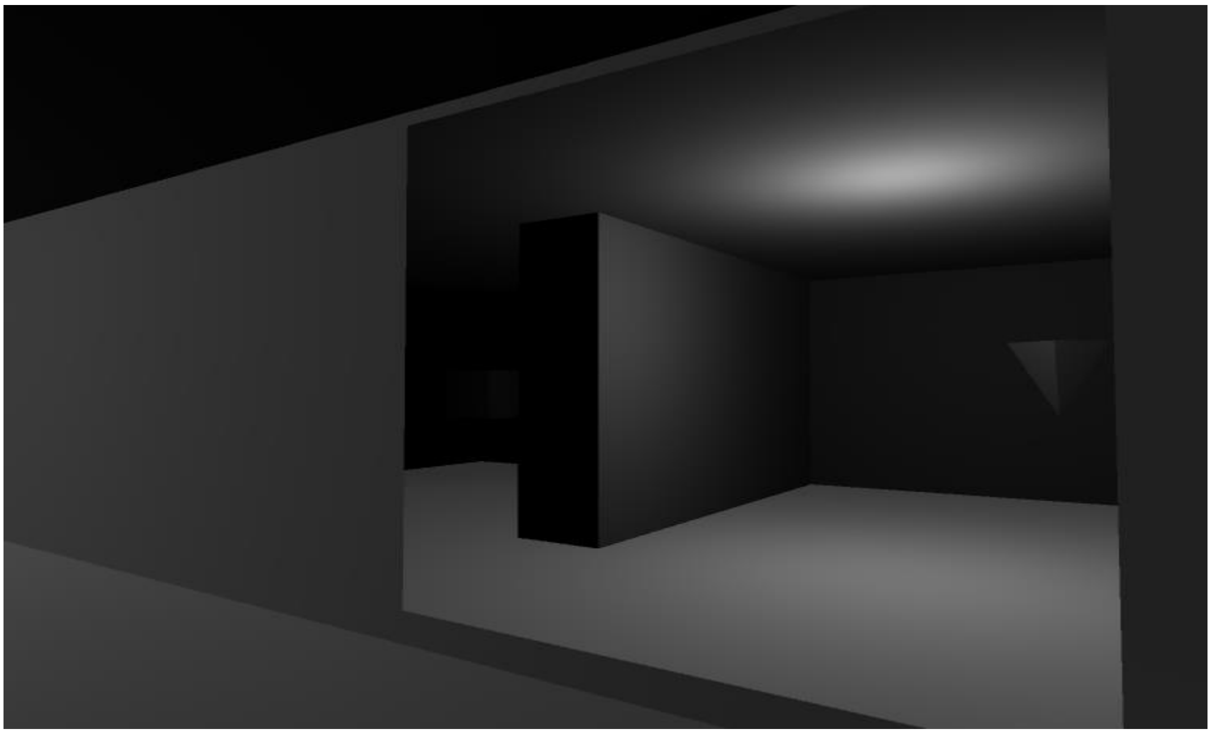


Figure 8 Showcasing portal effect using render texture method in the Unity3D engine, Authors Image, 2020

The implementation of portals in the unity3D engine presented serves as a prototype for testing the visual effects of a portal and the expected visual results that a raytraced alternative should produce. The application also serves as a basis to later make comparisons as the setup between both implementations differs greatly and each implementation contains its own unique issues. Due to the level design being done in a separate scene, it is easy to make fast changes to a scene and view them in this implementation, which takes longer for the raytraced version as it the scene needs to be exported and imported into the correct folders.

Raytraced Portal Implementation

A portal in the raytracing engine is created in three individual stages, starting with the creation of a model that is assigned as a portal object. For a portal to function, it must be linked to another unique instance of a portal and for this reason the assignment process is done during the level creation. During the level loading process, a portal is identified using the model's filename, which uses a separate function to create and load the model when compared to the rest of the objects in the level. The second stage of the portal creation involves assigning a unique hit group for the portals model which informs the program which hit shader to use when shading an object. The last stage of the implementation is a unique closest hit shader which is used by the portal models. The portal hit shader shares similarities with the default shader as they both obtain the normal direction of the triangle for further computation, however the difference lies in how the object is shaded. The default shader computes the diffuse lighting and returns a shaded colour, however the portal hit shader constructs the parameters for a new ray to be traced in the scene. Using a constant buffer slot, the position of a linked portal can be shared and accessed from inside the hit shader which is used to obtain a local hit position on an object. When the object is hit, its position is stored and by subtracting the objects position provides an offset that can be used on to obtain an origin point for a ray as shown in figure 9. This method works for any mesh as the linked portals geometry does not affect the final position of the offset, which also means that two linked portals can have unique models.

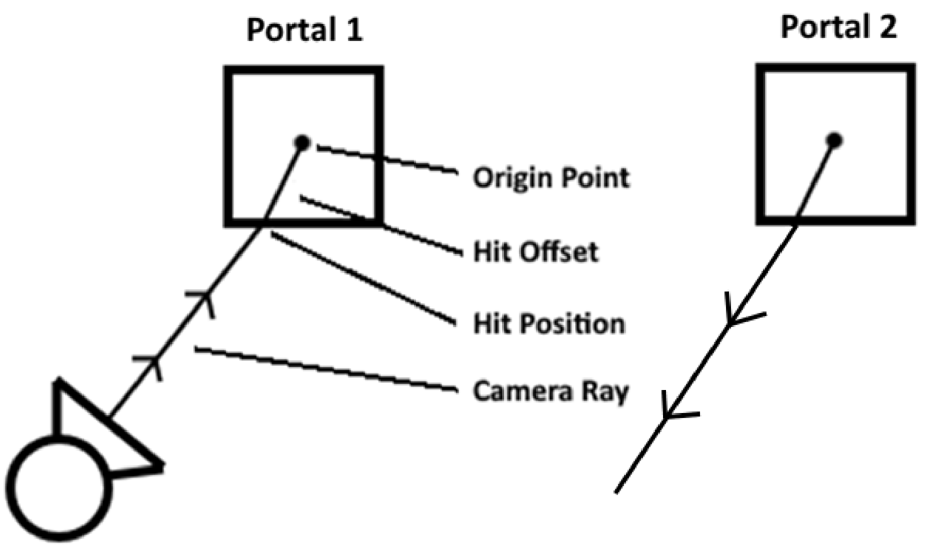


Figure 9 Using an offset to position a new ray from a portal, Authors Image, 2020

The direction of a ray can change the visual effect of a portal, for instance if the direction is the same as the original camera direction, then the result is a window that can view the scene in the same direction in the world as the camera. A reversed forward direction would produce results closer to the raster implementations visual where a ray going into one portal come out of another at the same angle as shown in figure 10. In DirectX12, the z component of a direction vector with XYZ coordinates is used as the forward direction, therefore by multiplying the z component by minus one will produce a revered direction from the camera.

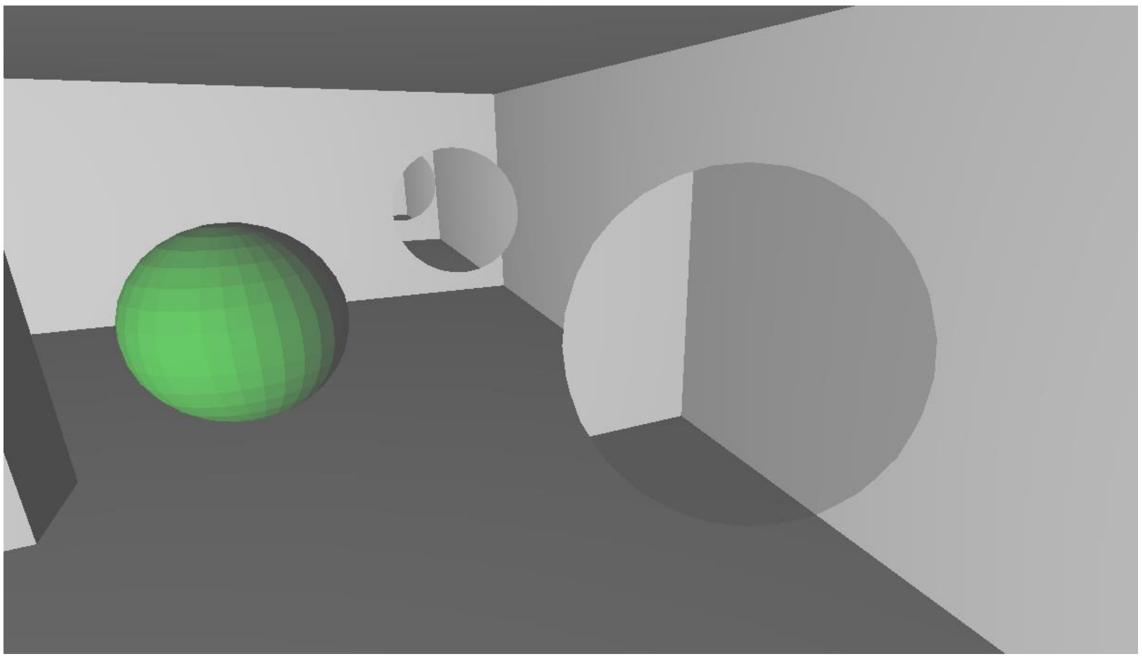


Figure 10, Using an equal direction to the main camera, each portal displays the view from the same direction, Authors Image, 2020

Rays in the raytracing shader are cast using the TraceRay function. This contains parameters such as a ray which contains an origin and direction, the top level acceleration structure of a scene, a Ray Flag which specifies ray operations such as only rendering opaque or skipping procedural primitives, and an hit group offset contribution value for use in shader table calculations. The function also contains a geometry stride used to skip certain types of geometry in a bottom-level acceleration structure, as well as a miss shader index used to obtain the miss shader when a ray does not hit any object in the scene. Finally, the function contains a colour payload which is used to return the colour of the object hit by the ray(this is after the pixel is shaded by a hit shader). The TraceRay function is called within the hit shader which contains the same parameters as the original camera ray but with a different origin and direction which means it can return the same visual payload colour that can potentially hit another instance of a portal, creating a recursion that does not end until the final ray hits a non-portal object. In specific instances this can produce an unusually high number of recursions such as if the user is looking directly into a portal which faces another portal such as in figure 11. The small dark circle produced by the recursions is the result of those pixels not updating once the lighting of that area changes. A preferable method would be to apply fog to each iteration to make the final recursion a less abrupt change, or to have a clause in the function to return a separate colour that can be controlled by the developer.

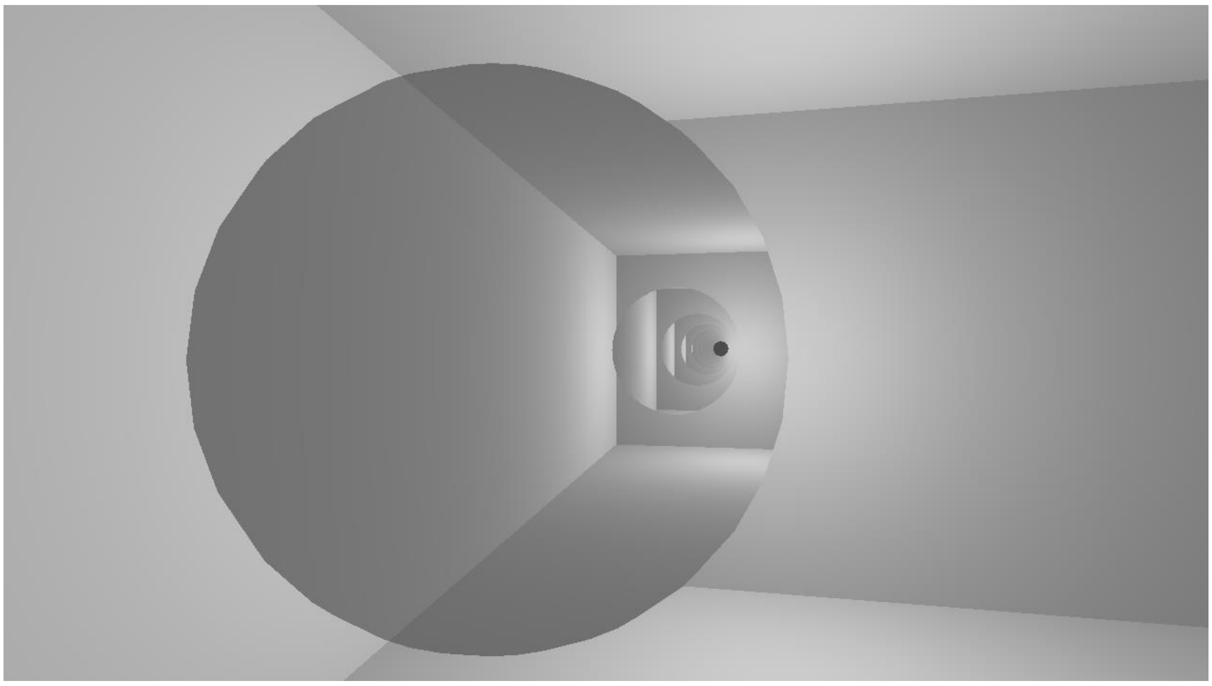


Figure 11 Portal recursions caused by two aligned portals. Small dark circle caused by max recursion limit reached, Authors Image, 2020

This would negatively impact the performance of the application as the number of rays required to produce the image would be substantial, however this is circumvented by applying a maximum number of recursions available to non-primary TraceRay calls, this value for the number of recursions allowed is set during the setup of the raytracing component.

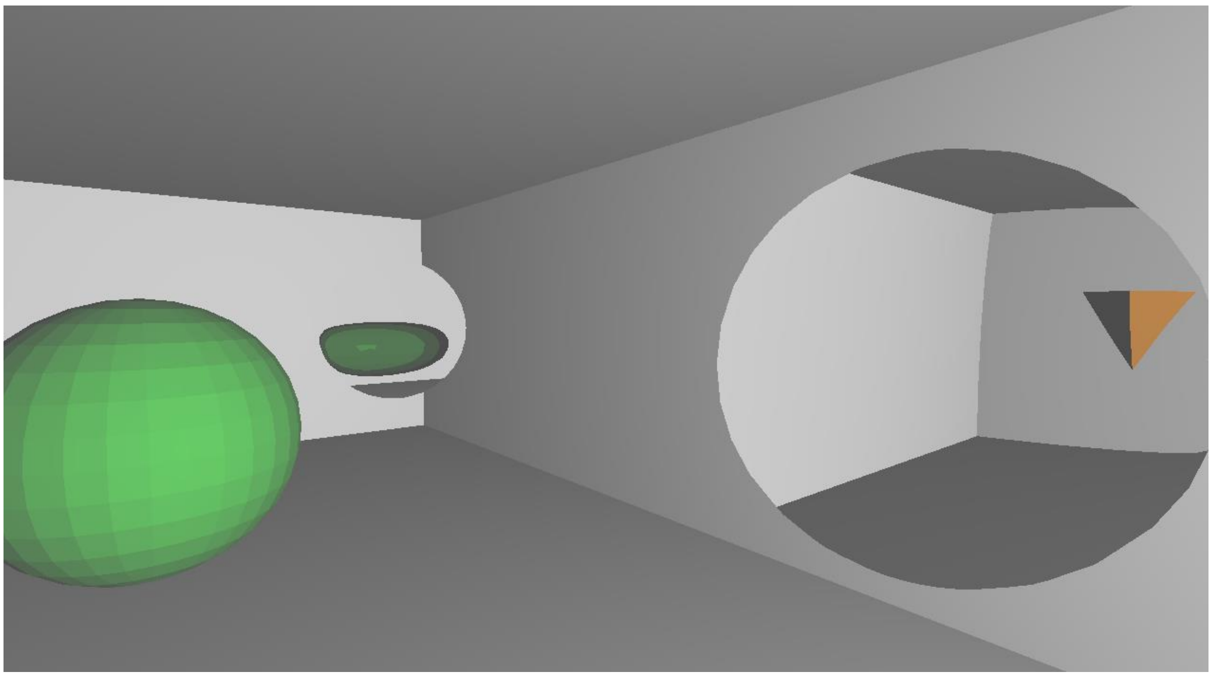


Figure 12 Image displaying the use of sphere meshes for a three-dimensional portal view, authors image, 2020

While the raster implementation is limited to using flat quads for the portal views, the raytraced portals can be applied to a triangle-based three-dimensional mesh that is loaded into the level such as the spheres shown in figure 12. While up close the appearance of the portal looks two-dimensional, however as the camera moves further away the view begins to curve around the sphere. This effect is more difficult to implement in a raster engine, especially with the texture rendering approach as the texture will end up stretched to fit a curved object and there would be difficulty in culling parts of the object that should not be seen from the cameras perspective. One way to view the shape of the portal is to apply diffuse lighting present in the default hit shader and multiply it by the final colour which produces a shaded portal view as seen in figure 13. Using this method allows for the faces of the spherical portal to be visible while it still produces the visuals using the new rays, which makes it easier to understand how the view becomes warped as it curves around the sphere.

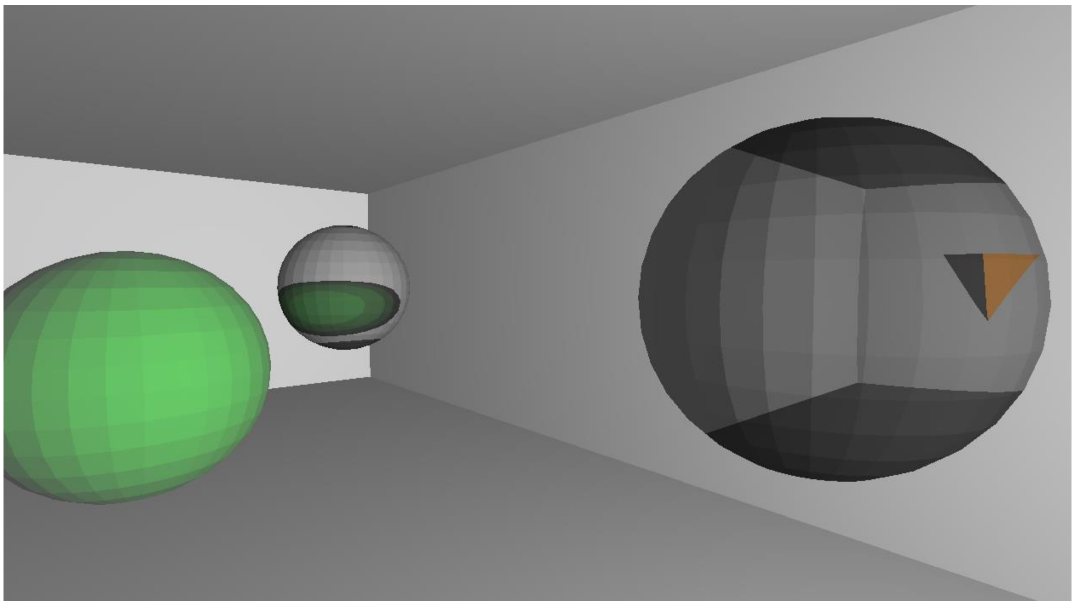


Figure 13, Image showcasing diffuse lighting on the portals model to see the shape of a portal, Authors Image, 2020

Diffuse Lighting

In order to shade the environment present in the raytraced portal implementation, the raytracing shader calculates diffuse lighting for each ray-triangle intersection. The function for calculating the diffuse lighting requires both the ray hit position defined as

$$O + (t)D$$

and the surface normal of the hit. The surface normal **N** of a triangle can be calculated as

$$N = vn0 + bx * (vn1 - vn0) + by * (vn2 - vn0)$$

where vn are the vertex normals of the triangle and bx and by are the barycentric coordinates.

Calculating the lighting is done using a dot product of the normalized direction of a light minus the ray hit position and the surface normal defined as

$$A = N \cdot L$$

where **N** is the surface normal and L is the normalized light direction. The final colour of the object is the albedo colour multiplied by both the lights colour and the value of the dot product, an example of this lighting is presented in figure 14. In a scene with only a single light source there are surfaces on an object that will have no shading apply to them, therefore a minimum ambient lighting level is applied to these objects. While raytracing can produce environments with realistic lighting, only diffuse lighting is required in order to view the portal implementations visuals.

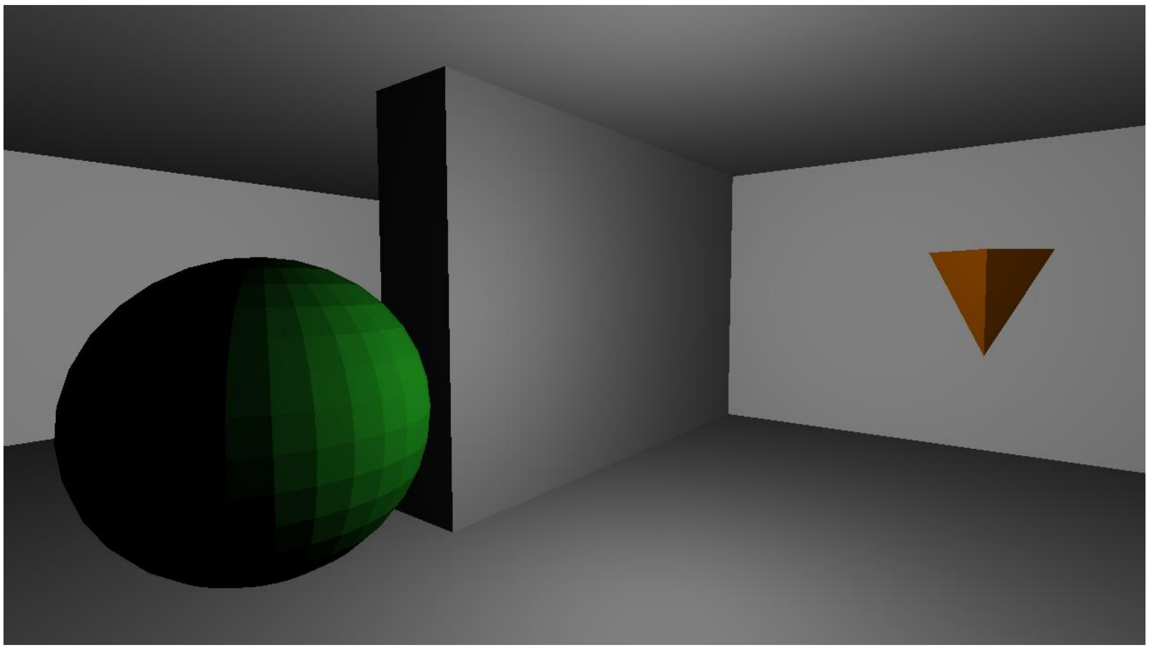


Figure 14 Diffuse lighting applied to multiple objects by a single light source, Authors Image, 2020

Evaluation

Hardware

Although real-time raytracing is a flagship feature of the RTX line of graphics cards by Nvidia, the raytracing implementation has been developed on a GTX 1070 card, which while it can support raytracing does not have some of the features present in the RTX cards such as AI-denoising or DLSS(Deep Learning Super Sampling). While this does not provide an optimal environment for evaluating the performance of the raytraced implementation, the frame-rate of the application does not drop below 140 frames-per-second and has not made a noticeable effect on the visuals even in strenuous circumstances such as when calculating multiple recursion rays per frame. An ideal testing environment would use a later line of RTX cards such as the 2000 series or preferably the recent 3000 series.

Implementation Evaluations

The difficulty of implementing the raytraced portals is that scene scales with its complexity, as each portals model is required to be identified before the application runs to give it the portals hit shader instead of the default model shader. Currently this is managed inside the shader table creation, however in a larger setting this could be more efficient by using a dedicated shader table creator class to bind models to the correct shaders. This method would also scale with future shaders as currently applying any shader other than the default one must be done manually on a per-object basis, whereas in an optimal setting each object that gets imported by the level should have the name of its shader read in from a file. One of the benefits of this implementation is that when a ray intersects with a portal, it produces another ray that for is unique to the first but can the same results. Any visual effects or shaders that appear from the primary rays that appear in the final image will automatically appear in the portals image without further effort. This is an intrinsic part of raytracing that should not be overlooked as to produce this effect using the raster rendering method, each camera would need to be setup render with these visual effects added as well. Figure 10, which showcased how different directions could alter the view of the portal could be expanded upon into its own gameplay mechanic, as a user could alter the direction of the portal rays which would give them a different view of the environment without moving the camera. This effect can also be done in the raster implementation by allowing the user control of the portal cameras, which would also change its final view.

One problem presented with the raster implementation, as well as a problem that was identified during the development of Portal(D. Kircher, 2018) is that objects that are between the view of a portal and its corresponding camera appear in the render texture. An example of this problem is presented in figure 15, where a green sphere is placed behind one of the portals, and when looking through the linked portal at certain angles will display the green sphere as it occludes the view of the portal from the camera. In his talk, David Kircher describes this problem with the term “Banana Juice” as it is a problem that is not easy to describe. The solution provided by Kircher is to create a clip plane behind a portal so that level geometry or objects behind the portal are not visible. This solution will work for two-dimensional quads as there is an obvious forward and backward direction of a portal to create a clip plane, however for three-dimensional portal objects this method will not work as there are no obvious direction on objects such as a sphere. Another solution would be to calculate if any objects near the portal are between the portal and the camera every frame, however this would cause an unwanted drop in performance, especially with a large number of objects surrounding a portal. While creating a workaround for this issue can be complex, the raytraced portals do not encounter this issue as portal rays are produced on the surface of a portal instead of behind it which means objects cannot occlude the view.

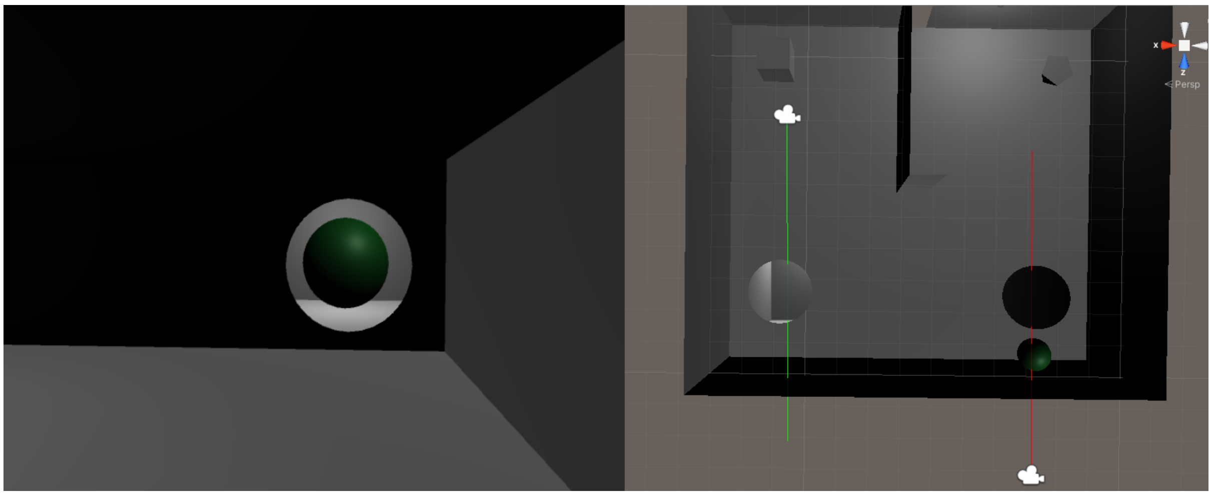


Figure 15, Showcasing objects occluding the view of a portal camera, Authors Image, 2020

With changes to the raster implementations fragment shader, the portal effect can be applied to curved surfaces such as spheres just like in the raytraced version, however the effect of the portal is still two-dimensional. This is due to how the raster portals use a camera that is behind the linked portal rendering the geometry, and so the projected view is only a two-dimensional image that appears as a circle that is always looking at the camera. The raytraced version however returns the view from rays that can hit a sphere at different positions in space, which can produce a true three-dimensional image on the spheres surface. A comparison of these two effects can be seen in Figure 16.

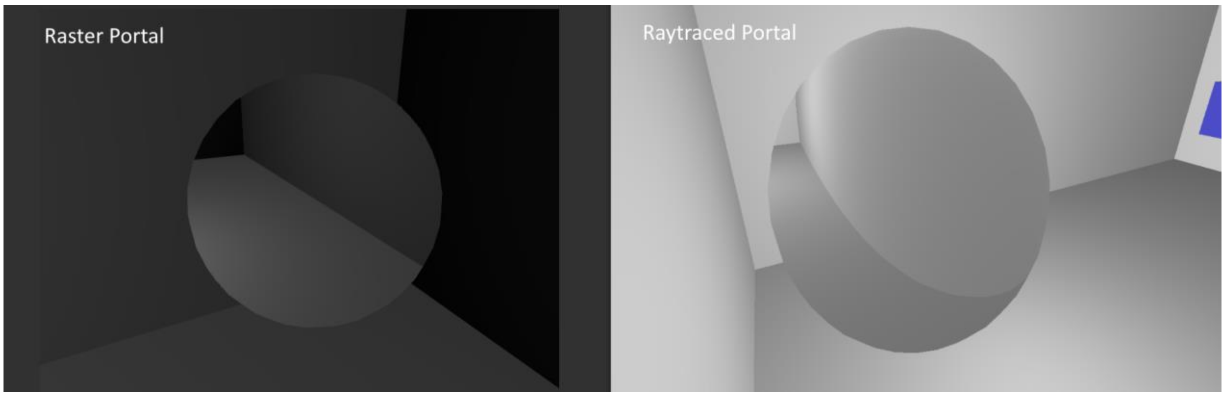


Figure 16, Raster portal(left) on curved surface showing no changes and Raytraced portal(right) showing warped results of 3D portal, Authors Image, 2020

Another indicator that the raytraced portal is three-dimensional is that when an object becomes warped enough, it wraps around on a curved surface creating a warped representation of the object on the surface of the portal as shown in figure 17. The green left side of the green sphere ends up wrapping around curved surface of the portal creating an interesting effect not possible on a two-dimensional mesh.

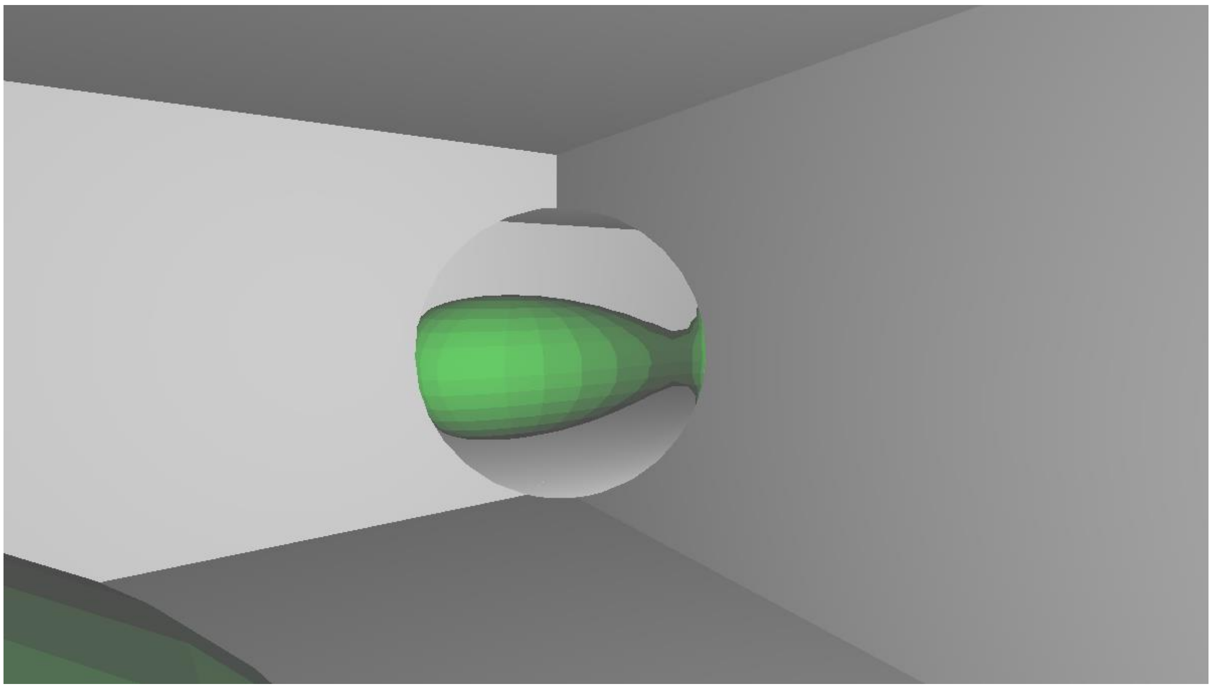


Figure 17, Raytraced portal showing object wrapping on a curved surface, Authors Image, 2020

The portal hit shader can be applied to any polygon mesh unlike the raster equivalent, and the implementation allows for each individual portal to have its own unique mesh, however this can create instances where one portals mesh can be seen from the view if it obscures another. This also means that the final visuals from the first portal are not affected by the geometry of the second portal. Although this is the intended functionality, there could be interesting visuals produced if the second portals geometry effected the new ray's origin and direction as it comes through the portal. There is also a visual bug which occurs when reaching the max number of recursions from a portal hit. If a portal is hit while the recursion limit is reached, the pixel colours are not updated and use the previous frames colour. This effect produces a 'smear' such as the one in figure 18 is visible when the camera views the portal in motion. With regards to performance, the portals function well when they don't take up a large amount of the cameras view, however the frames-per-second drop dramatically as the camera moves toward a portal. This is due to the number of rays required per frame increasing as more of the original primary rays from a camera hit the portals surface as seen in figure x. While these rays hit the surface of the portal, they are potentially running intersection calculations on the geometry behind the portal as well. In this implementation the scenery is made up of simple planes with only two triangle intersections require per plane, however in a more complex scene the performance impact of the portals may become an issue as the number of rays increases exponentially.

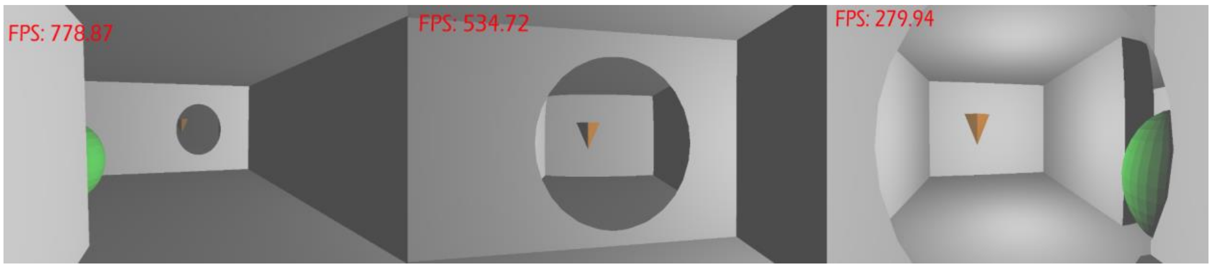


Figure 18, Images showcasing decreasing fps as camera gets closer to a portal, Authors image, 2020

Another issue is that creating a portal is done at the start of the application, and a portal is identified by its filename such as “portalsphere” instead of “sphere”. This means that portals in the current implementation cannot be adding during runtime of the application as the scene would need to be rebuilt, and each type of portal would need its own unique filename. Solutions to this problem such as swapping which hit shader is used on a model also would not work due to a portal requiring a linked portal in its constant buffers in order to function, and without this the program enters undefined behaviour or crashes. Portal links would need to be changed at runtime which would require rebuilding the shader tables as this is where the constant buffers are applied. There are also visual errors that appear on meshes in the raytraced application in the form of incorrectly coloured pixels such as the ones shown in figure 19. Although the cause for these incorrect pixels is largely unknown, one potential reason is due to floating point errors which cause rays to slip between the edges of the triangles in a mesh, as the colours of the pixels does match the scenery behind the object. While this issue is reproducible, the effect on the visuals is minimal as only a few pixels are incorrect, and the issue Is less noticeable during camera motion.

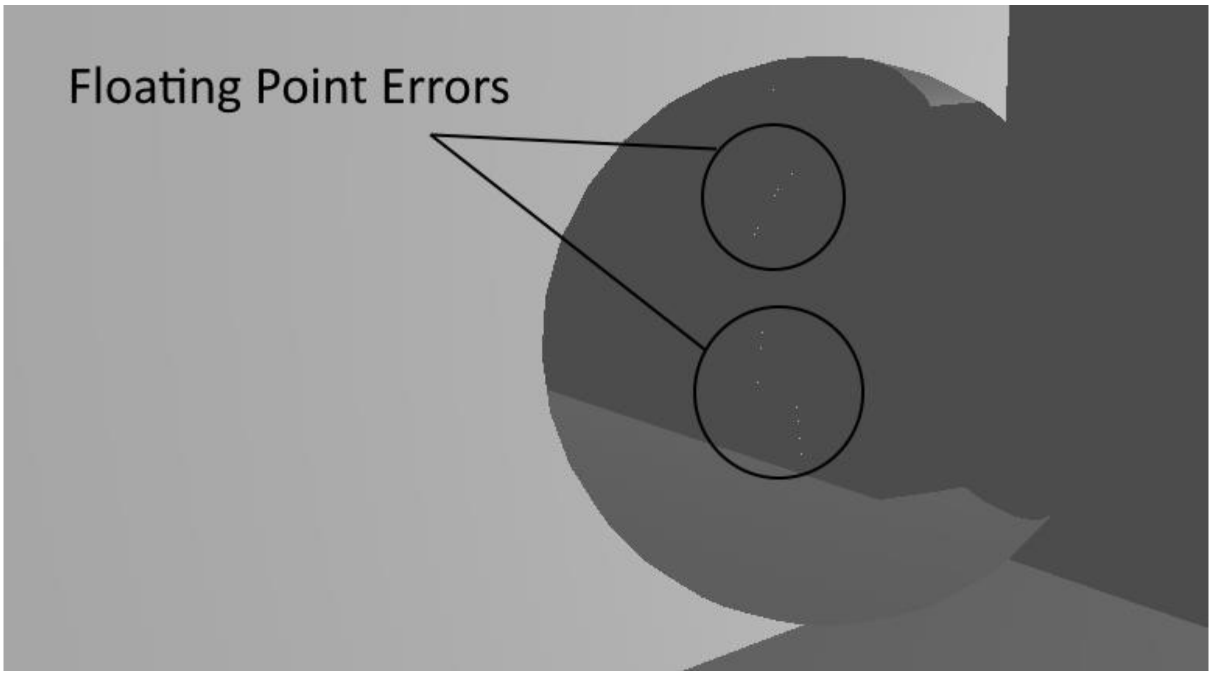


Figure 19, Potential Floating-point errors causing incorrect pixel colours, Authors image, 2020

One of the downsides to the portals using recursive rays to produce their visuals is that with a maximum recursion value set, the view from a portal will display a lower number of recursions than the main camera. This becomes a larger issue as the number of max recursions decreases, for example if there are is only two max recursions set in the program, then reflective or portal surfaces wont be able to be seen from the view of a portal as both portals and reflections share the same amount of recursive rays. This issue will also be present when implementing global illumination or “path tracing” which also uses recursive rays to produce realistic lighting, as the lower number of recursive rays would reduce the quality of the image produced. Although an obvious solution is to use a unique value for portal recursions directX12 does not support adding any new values for this, consequently the only solution is to increase the number of maximum recursions allowed which will impact the performance. Comparing the performance of each implementation is difficult as there are a few factors that affect the outcomes, first instance although the Unity3D engine has a profiling option, it is difficult to distinguish the impact that one camera render has over another, as all renders are grouped into a single block. It is also difficult the judge the impact on performance between each implementation as they are both created in two different engines, for instance the directX12 application is built from the ground up specifically for the portal implementation in mind while the Unity3D implementation was added into a setup designed for a game.

The scalability of the raster implementation may potentially have problems with screen space visual effects, for example in order to use SSAO(screen space ambient occlusion) each camera is required to render the scene with this effect applied which can become increasingly expensive in performance. Screen space image effects can be applied to a final render, however in order to view these effects in the portals view texture, it also needs to be applied to each portal's camera view before the main camera renders the scene. Due to how the raytracing implementation is setup, no matter how far into the recursions a ray is, it will always return the same hit shaders used throughout the scene. This means that if the hit shader has certain features such as ray traced ambient occlusion or global illumination, then all portal views will also contain these features and no additional effects need to be added to the scene in order to view them. While the chosen raytracing workflow uses triangle intersections, raytracing can also render procedural meshes which use their own intersection code. This would allow for the portals to appear on the surfaces of procedural surfaces and dynamic surfaces not available to the triangle rendering.

Finally, both implementations could benefit from having more interaction between the user and the applications. In both implementations, the user is able to move the camera with the mouse and the WASD keys on a keyboard, however they are currently a passive viewer rather than an active participant. Allowing the user to create their own portals or objects and move them in real-time would not only make the user more of an active participant but could also allow for them to view or create new and interesting visuals not seen with the current builds. More interaction could also find potential problems or features that one implementation has that is not present in the other which is useful for both users and developers.

Further Development

To expand the portal implementation further, the portal hit shader should be edited to allow portals to work when on separate alignments. Currently the hit shader assumes that the portals share the same rotation, however any objects that have unique rotations will have an unexpected outcome as it only reverses the ray direction. In order use this shader on rotated objects, the ray direction coming out of the portal must be modified by the linked portals rotation. This method would also enable using other models for the portals instead of flat planes, however the linked portal would also need the same model to have the expected portal view. The current raytracing shader used in the application only implements basic lighting of objects which is enough to view the portal implementation, however it does not display the complex lighting calculations are that only possible using raytracing when compared to the raster implementation. Lighting effects such as raytraced shadows and reflections would be visible from portal but not provide any new interactions, however global illuminated lighting would allow for a portal to interact with the environment by allowing light to pass through and light up darker areas. While these lighting features are not necessary to view the portal implementation, they do would expand on what raytraced portals are capable of when compared to the raster rendered alternative.

Another aspect of the implementation that could improve is the acceleration structure and object management. The current version of these features allows for both the level and objects to be created at the beginning of the application, however it does not allow for these to be updated during runtime. In order to support these features, key parts of the implementation would require structural changes. One of the major changes would be to allow for reconstruction of the descriptor heap for adding or removing descriptors at runtime which would store the buffers of new or removed objects. Along with the descriptor heap, the shader tables would also require either reconstructing or editing to assign the correct shader to new objects. The acceleration structure would also need to be rebuilt when an object is either added or removed from the scene as a single object can change the structure of the tree. Although rebuilding the acceleration structure is a costly procedure it is not done every frame update, instead only rebuilding when changes to the scene take place in the same frame. An implementation of portals in the unity engine using stencil buffer method would allow the comparison of each method as well as a better comparison to the ray traced version, however a better solution to this would to implement raster rendering in the DirectX12 application. While the current application uses the DirectX Raytracing workflow, raster rendering is what DirectX12 uses by default and implementing the ability to swap between both raster and raytrace rendering in real-time would improve the ability to compare both features. Currently, testing performance of the implementations is flawed as there is difficulty discerning the impact on performance that the rendering takes up compared to other functions of the application. For a fair test to be made, the implementations would need to be done in the same engine, which would allow for both a visual and technical evaluation of each rendering technique. Besides the large changes to the project, there are small improvements that can be made to increase the functionality of the application. While objects can have separate colours, these are applied manually during the shader creations on a per-object basis. Each object has a new constant buffer applied to it which contains a colour, however more fluid approach to this would be to have the objects albedo imported along with the model data during the level creation. Along with object colour, the portal data is also applied during the shader creation step however this should also be done during level loading as it would allow for the application to become more data driven.

Conclusions

While the visual effects of a portal can be produced using both raytracing and rasterization, the raytracing implementation offers more depth in the effects that can be produced as it can be applied to three-dimensional objects instead of just flat surfaces. Handling the portals also requires less setup in raytracing than the raster alternative which requires maintaining individual cameras and render textures per portal. The current implementation showcases a demo of the portal effect using raytracing, however in order to perform a complete evaluation of the performance impact that the raytraced portal have, a larger and more complex scene would be required. Raytraced portals also offer more customizable features that would affect the visuals produced from portal such as more complex lighting calculations being made in the portals hit shader or changing the visuals produced from recursive rays to have their own interactions. There are also different directions that the current project can be taken in. Improving the raytracing shader to support additional graphical effects such as shadows, reflections and more complex lighting could help showcase how the portals can be used from a gameplay perspective, however another direction would be to support one portals geometry affecting the visuals producing by the first portal. Both directions would help showcase how the overall portal effect can be used in a gameplay environment. Raytraced portals can offer visuals equal to their raster alternative required in games such as Portal or Narbacular Drop and can even alleviate some of the issues present in those implementations, however this does come at a moderate cost of performance, that increases as the use gets closer to a portal. Further testing should be done on the performance impact of the portals, especially as the current implementation is done on a previous line of graphics cards not designed with raytracing in mind. Although raytraced portals can be done on three-dimensional meshes that create unique visuals, it is unclear if these visuals can provide an impact to gameplay outside of being interesting to look at.

References

Dürer, A. (1538) Underweysung der Messung.

IBM Research Centre (2020) *Some techniques for shading machine renderings of solids*. [online]. Available from: http://people.reed.edu/~jimfix/math385/lec01.1/Light/Appel.pdf [Accessed 6 February 2020].

Whitted, T. (1979) *An improved illumination model for shaded display*. [online]. Available from: https://dl.acm.org/doi/10.1145/358876.358882 [Accessed 5 February 2020].

ID Software (1992) *Gameplay screenshot from WolfenStein 3D*. [online]. Available from: https://drh1.img.digitalriver.com/DRHM/Storefront/Company/zenimax/images/screenshots/Wolf_older/Wolf3D/0000002417.1920x1080.jpg [Accessed 06/02/20].

Pixar Animation Studios (2006) *Ray Tracing for the Movie 'Cars'*. [online]. Available from: https://graphics.pixar.com/library/RayTracingCars/paper.pdf [Accessed 6 February 2020].

T. Moller, B. Trumbore, (1997), Fast Minimum Storage Ray/Triangle Intersection, AB Chalmers University of Technology, available from: https://cadxfem.org/inf/Fast%20MinimumStorage%20RayTriangle%20Intersection.pdf [accessed July 2020].

M. Weber, Baldwin, (2016), Fast Ray-Triangle Intersections by Coordinate Transformation, State University of New York at Geneseo, available from: http://jcgt.org/published/0005/03/03/ [accessed July 2020].

S. Green & Paddon, (2020) A highly flexible multiprocessor solution for ray tracing. International Journal of Computer Graphics ,The Visual Computer Volume 6 pp. pp 62–73. [Accessed July February 2020].

Parker, S. et al (2010) OptiX: A General-Purpose Ray Tracing Engine. NVIDIA, [online]. Available from: https://research.nvidia.com/sites/default/files/pubs/2010-08_OptiX-A-General/Parker10Optix.pdf [Accessed July 2020].

Chaitanya, C. (2017) Interactive Reconstruction of Monte Carlo Image Sequences using a Recurrent Denoising Autoencoder. [online]. Available from:

https://research.nvidia.com/sites/default/files/publications/dnn_denoise_author.pdf [Accessed July 2020].

A.Majercik et al, NVIDIA, (2018), A Ray-Box Intersection Algorithm and Efficient Dynamic Voxel Rendering, available from: http://jcgt.org/published/0007/03/04/, [accessed July 2020].

Nvidia, 2019, Control: Multiple Stunning Ray-Traced Effects Raise The Bar For Game Graphics, available from: https://www.nvidia.com/en-gb/geforce/news/control-rtx-ray-tracing-dlss-out-now/ [accessed July 2020]

Nvidia, 2019, EXPERIENCE MINECRAFT WITH RTX BETA, available from: https://www.nvidia.com/en-gb/geforce/campaigns/minecraft-with-rtx/ [accessed July 2020]

Sunside Games (2019) *Stay in The Light [Computer game]*

Nuclear Monkey Software, (2005), Narbacular Drop [Computer Game]

Valve Software, (2007), Portal [Computer Game]

D. Kircher, (2018), Portal Problems - Lecture 11 - CS50's Introduction to Game Development 2018

A. Fujimoto et al, (1983), ARTS: Accelerated Ray-Tracing System, Graphica Computer Corporation, available at: http://www.dca.fee.unicamp.br/~leopini/DISCIPLINAS/IA725/ia725-12010/Fujimoto1986-4056861.pdf [accessed July 2020]

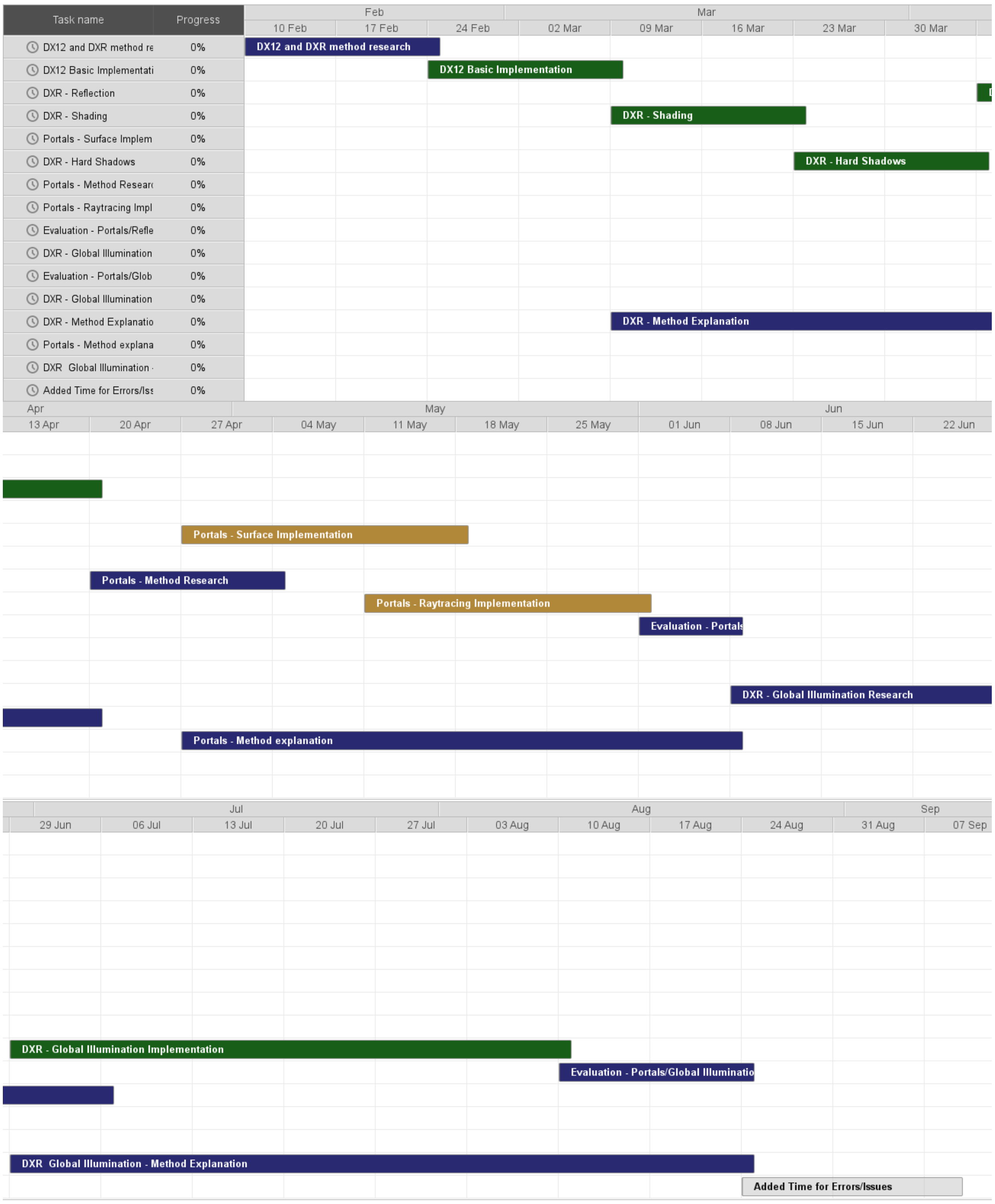
T. Kay et al(1986), Ray tracing complex scenes, SIGGRAPH '86: Proceedings of the 13th annual conference on Computer graphics and interactive techniques, available at: https://dl.acm.org/doi/10.1145/15922.15916 [accessed July 2020]

C. Kolb, D. Mitchell, P. Hanrahan, (1995), A realistic camera model for computer graphics, In Proceedings of the 22nd annual conference on Computer graphics and interactive techniques (SIGGRAPH '95), available at: https://dl.acm.org/doi/10.1145/218380.218463 [accessed June 2020]

M. J. Muuss, (1987), RT & REMRT: Shared Memory Parallel and Network Distributed Ray-tracing Programs, Proceedings of 4th Computer Graphics Workshop, Cambridge, MA, USA, pp 86-98.

Blender foundation, 2020, Blender 2.9 [computer program], available at: https://www.blender.org/

Appendices



Task name	Progress
DX12 and DXR method re	0%
DX12 Basic Implementati	0%
DXR - Reflection	0%
DXR - Shading	0%
Portals - Surface Implem	0%
DXR - Hard Shadows	0%
Portals - Method Researc	0%
Portals - Raytracing Impl	0%
Evaluation - Portals/Refle	0%
DXR - Global Illumination	0%
Evaluation - Portals/Glob	0%
DXR - Global Illumination	0%
DXR - Method Explanatio	0%
Portals - Method explana	0%
DXR Global Illumination ·	0%
Added Time for Errors/Iss	0%

Feb
Mar
10 Feb
17 Feb
24 Feb
02 Mar
09 Mar
16 Mar
23 Mar
30 Mar
DX12 and DXR method research
DX12 Basic Implementation
DXR - Shading
DXR - Hard Shadows
DXR - Method Explanation
Apr
May
Jun
13 Apr
20 Apr
27 Apr
04 May
11 May
18 May
25 May
01 Jun
08 Jun
15 Jun
22 Jun
Portals - Surface Implementation
Portals - Method Research
Portals - Raytracing Implementation
Evaluation - Portals
DXR - Global Illumination Research
Portals - Method explanation
Jul
Aug
Sep
29 Jun
06 Jul
13 Jul
20 Jul
27 Jul
03 Aug
10 Aug
17 Aug
24 Aug
31 Aug
07 Sep
DXR - Global Illumination Implementation
Evaluation - Portals/Global Illuminatio
DXR Global Illumination - Method Explanation
Added Time for Errors/Issues

Project Proposed Timeline – The beginning of the project followed closely with the timeline, aside from the original DirectX12 project setup taking longer than anticipated however the inclusion of global illumination was not included in the final build due to the difficulty of its inclusion as well as the project putting a larger priority on the portal implementation. Another reason this feature was not included is that development time was spent making a raster-based alternative of the portals which was used to both compare and prototype the feature. Other visual effects such as reflections and shadows were also skipped as completing the portals was deemed more important to the project, and these features would serve mainly to complement the overall visuals of the final visuals.