Minimizing Draw Calls, A GPU Approach

by

Juul Joosten

Abstract

Rendering high fidelity scenes comes at the cost of performance and with the emerging games in *virtual reality*(VR) these scenes need to additionally be rendered multiple times. By the use of *graphics processing unit*(GPU) *occlusion culling* that drives *indirect rendering*, we can remove expensive draw calls and save on performance, without losing visual fidelity or introduce latency to the rendering pipeline.

Contents

Chapter 1

Introduction

Over the years, computer graphics(CG) in games have made vast leaps towards engaging players in highly immersive worlds. Due to the detail that is added by artists and level designers in terms of objects, texturing, lighting and modeling, these scenes require vast amounts of draw calls and shading operations. With the introduction of the next gen rendering APIs in mid 2015[5]; new methods have become accessible for a wider range of devices to lower this burden. Since the visual richness of games is constantly growing and virtual reality(VR) has effectively required games to additionally render their scenes multiple times, the need to minimize high fidelity draw calls has increased. *Compute shaders* in combination with *rasterization* and *indirect rendering* could increase the performance for current games and games to come.

This dissertation focuses on bringing an accessible and informational introduction to *indirect rendering* by the use of *GPU occlusion culling* described in the NVidia presentation called *Advanced Scene Graph Rendering Pipeline*[2]. By leveraging the *rasterizer*, *compute shaders* and *indirect rendering* we can efficiently remove occluded draw calls from the command list and thus increase performance or save power.

1 Overview

Vast leaps towards engaging and emerging visual worlds have created a need for high performance rendering pipelines. GPU occlusion culling facilitates a zero latency method that filters occluded draw calls from the command list. To be able to create, maintain and improve this pipelines, knowledge about the techniques involved is required.

2 Problem Description

With the progression towards high quality stylized and realistic rendering, we draw in more creative ways and thus put more stress on the GPU. In the current generation of games, a draw call could use multiple textures, buffers, up to hundreds of lights and complex shading algorithms to achieve high quality visuals. Since VR is on the rise, all these high quality visuals additionally need to be rendered twice to account for the stereoscopic view of the hardware. To achieve real-time rendering results for current games and sustain growth in visual fidelity for future games; there are two paths towards better performance.

- *Improve the hardware* where due to more power or better efficiency in instruction handling, better performance is achieved.
- *Optimize the software* where due to better use of the available hardware and knowledge about the application; the best path for the rendering instructions is enforced.

This dissertation will focus on the latter and improve performance by making optimized use of the hardware available.

Chapter 2

Implementation

1 Overview

1.1 Indirect Rendering

Indirect rendering is a technique in which the draw commands are created in a GPU buffer. This GPU buffer can be modified and submitted to the command list by the GPU without CPU interference. Prior to this technique, results from the GPU had to be send back to the CPU for processing the results; introducing latency or a CPU and GPU stall.

1.2 Occlusion Culling

Occlusion culling is a technique which determines the invisible objects from a predetermined viewpoint.

1.3 Introduction to GPU Occlusion Culling

By rasterizing occludee[1] geometry against occluder geometry filled depth buffer, indirect occlusion culling determines visibility and then alters the indirect rendering commands before they are submitted to the command queue on the GPU. No frame latency is introduced into the rendering pipeline.

[1] object that uses occlusion to determine visibility

Indirect Occlusion Culling consists of several passes, all of which are crucial in determining which draw calls contribute visually to the screen. The individual passes will be explained per chapter with source code snippets and images. A simplified overview of the passes is given below.

- Clear occluder geometry filled depth unordered access view(UAV)
- Downsample previous frames **static** depth buffer (Optional)
- Reproject (downsampled) **static**[2] depth buffer into cleared UAV
- Downsample reprojected depth buffer
- Copy downsampled reprojected depth buffer as shader resource view(SRV) to bound depth buffer
- Rasterize dynamic occluders or frustum edge static occluders into the depth buffer for better occluding results (Optional)
- Fill visibility buffer by rasterizing occluder geometry against the depth buffer
- Gather indirect draw calls based on visibility buffer results
- Render non occluded indirect draw calls

2 Frustum Culling

Generally the first step in visibility determination is frustum culling. Each objects *occluder geometry* is tested against the 6 frustum planes to determine if the object is within or on the edge of the cameras view.

The object is added to the render list when it passes the frustum culling check, this means that it will be considered as contributing to the view in this first pass.

[2]The depth buffer is required to only contain static geometry, since reprojecting dynamic geometry will not correspond in correct depths for the actual geometry in this frame.

Figure 2.1: In this image we can see the scene with some of the stages described in the overview. From left to right; previous frame depth buffer, reprojected depth buffer, downsampled reprojected depth buffer. Note that we move in a downwards and backwards motion, so the previous frames depth samples are projected away from us. This is visible at the side and bottom edges of the reprojection and downsampled reprojection buffer.

2.1 Implementation

GPU occlusion culling is a screen space technique, which means that the cameras frustum is already taken into account when occluded draw calls are rejected from the command list. Therefore frustum culling has no direct effect on the visually contributing draw calls. However, the rasterization of the occluder geometry could potentially benefit from frustum culling. Frustum culling can be done on the GPU using a compute shader in combination with an execute indirect call; which would make the whole pipeline GPU driven. Alternatively frustum culling could be done on the CPU, this would decrease the workload on the GPU and could potentially be used for other culling needs within the application. Note that for this technique, frustum culling only accelerates the visibility determination step.

3 Depth Buffer Reprojection

The next step is depth buffer reprojection. Since we rasterize the occlusion geometry against the previous frames static geometry depth buffer, it yields better results when we reproject the depth fragments into their current frames positions. The most significant benefit of this approach is that the visibility pass wont occlude against faulty depth positions. This could otherwise produce missing geometry in the scene. There are also some general rules and tips that should be obliged, listed below.

- *Only reproject a depth buffer with static geometry.* Reprojecting dynamic depth samples can yield false occluder information; resulting in the possible loss of visually contributing draw calls.

- *Depth reprojection is only required when the camera moves*, as long as the reprojected depth buffer or static geometry depth buffer of the previous frame is saved, we can reuse this for every frame in which the camera does not move. *Note that dynamic occluders would need to be rasterized into this depth buffer each frame, and thus it is important that the reprojected depth buffer is stored without the dynamic occluder information.*

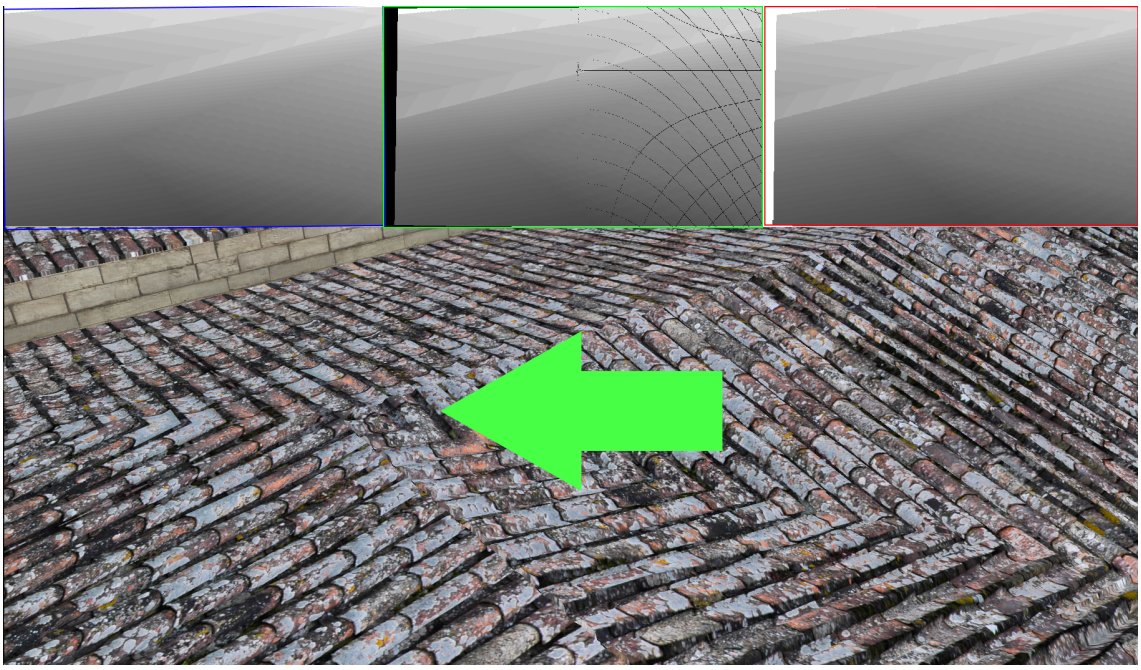

Figure 2.2: A rotation in the left direction shows how new information for the edge of the screen, is not available yet. This can be observed by the black left edge on the reprojected view(green) and the white left edge on the downsampled reprojected view(red). The blue view holds the previous frames unaltered depth buffer.

3.1 Issues With Reprojection

When moving or rotating quickly, reprojection will cause the samples to move over a large area on the screen. This results in holes where there is no screen space information available for correct reprojection. This can be seen as the red area in Figure **??**. In section 3.2 we will explain some of the improvements that can be made to overcome the data loss of reprojection.

3.2 Implementation

The reprojection of static geometry is done by transforming the depth sample from *normalized device coordinates*(NDC) to the previous frames view space and then project it into the current frames NDC and store it at its reprojected screen space position. The reprojection of dynamic geometry requires an

extra step, since reprojection only accommodates for the camera movement. In section 3.2 we will explain a way of adding dynamic occluder data into the visibility determination step. The following steps will be focussed on reprojecting the static depth data.

Clear Reprojection UAV

The first step in depth buffer reprojection is clearing the UAV that is used to store the reprojected depth values. The UAV is cleared to 0.0 so an atomic max operation will yield a value that is closest to the far plane[3].

Reproject

In our implementation of depth reprojection, we do a cheap downsample pass of the static geometry depth buffer to increase the performance of the reprojection and visibility tests, both of which performance is directly linked to work area resolution. The algorithm uses atomics and thus the floating point depth value needs to be interpreted as an unsigned integer value for the InterlockedMax to work. According to the floating point specification[4], **positive** floating point values are encoded with no restrictions to interpret them as unsigned integers and do valid max[4] operations on them.

```
#define DEPTH_BIAS 0.001

cbuffer r_cbvViewInfo: register(b0)
{
        struct viewStateStruct
        {
                float4x4 ProjectionMat;
                float4x4 ViewMat;
        } viewInfo;
};

cbuffer r_cbvPrevViewInfo: register(b1)
{
        struct prevViewStateStruct
        {
                float4x4 InverseViewProj;
        } prevViewInfo;
};

SamplerState r_point_clamp : register(s3);
```

[3]The algorithm and clear value might need to be altered when another depth buffer range is used. Our implementation has a near plane of 0.0 and a far plane of 1.0.

[4]We take the maximum value of the down sample pass for reprojection since it makes our algorithm more conservative and limits the amount of wrongly occluded geometry.

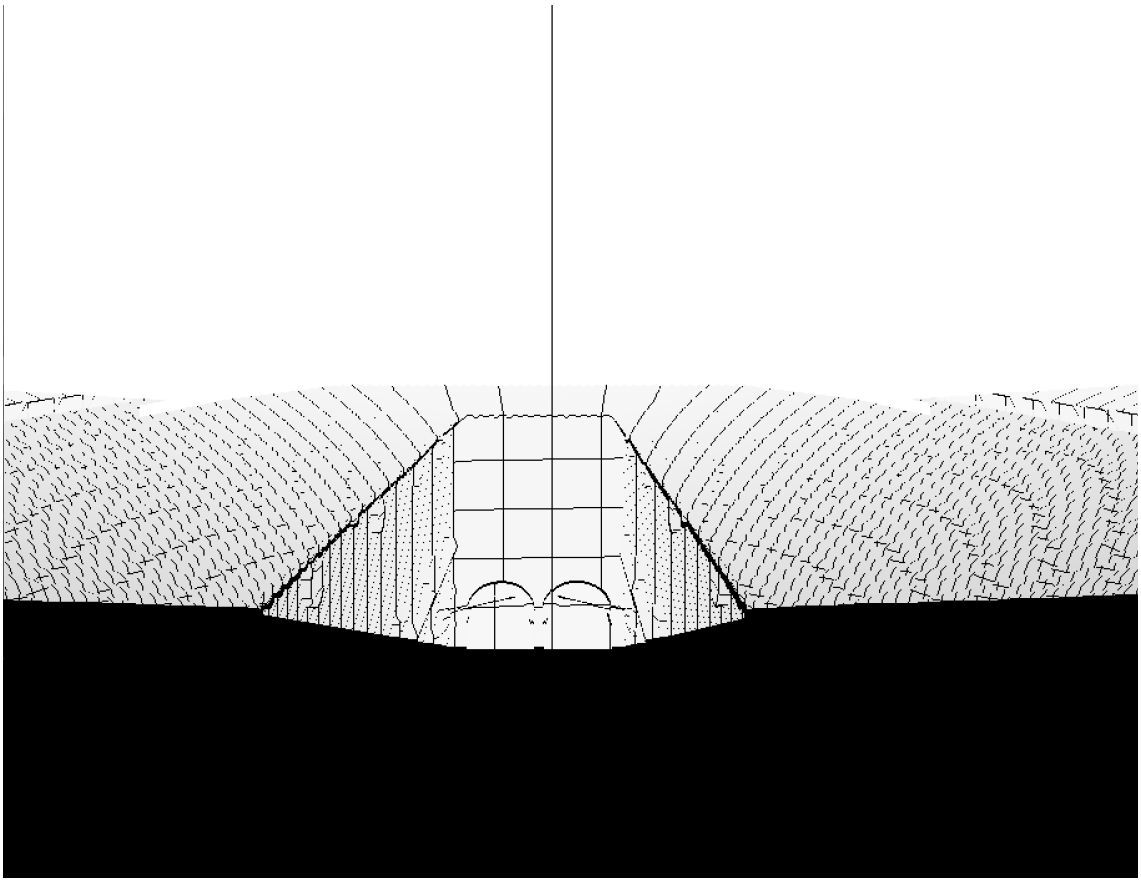

Figure 2.3: A reprojected depth buffer might look like this.

```
Texture2D<float>         prevDepthBuffer         : register(t2); // SRV
RWTexture2D<uint>        reprojDepthBuffer       : register(u3); // UAV

[numthreads(NUM_THREADS_X, NUM_THREADS_Y, NUM_THREADS_Z)]
void CSMain(uint3 pixelID : SV_DispatchThreadID)
{
        if (pixelID.x < HALF_SCREEN_WIDTH &&
                pixelID.y < HALF_SCREEN_HEIGHT)
        {
                float2 screenSpaceUV = float2(pixelID.xy + 1) / float2(
                    HALF_SCREEN_WIDTH, HALF_SCREEN_HEIGHT);

                float4 depthGather = prevDepthBuffer.GatherRed(r_point_clamp
                    , screenSpaceUV, uint2(0, 0));

                // when we only have samples of the near plane distance, we
                    can ignore this patch
                float depthGatherSum = depthGather.x + depthGather.y +
                    depthGather.z + depthGather.w;
                if (depthGatherSum == 0.0)
                        return;

                // we always take the maximum value of the previous depth
                    buffer to make sure that we always have the worst case
                    depth and don't occlude visible geometry
                float prevFrameDepth = max(depthGather.x, max(depthGather.y,
                     max(depthGather.z, depthGather.w)));

                // unproject
                float4 screenSpaceToNDC = float4((screenSpaceUV * 2.0) -
                    1.0, prevFrameDepth, 1.0);
                screenSpaceToNDC.y = 1.0 - screenSpaceToNDC.y;
                float4 prevFrameViewSpacePos = mul(prevViewInfo.
                    InverseViewProj, screenSpaceToNDC);
                prevFrameViewSpacePos /= prevFrameViewSpacePos.w;

                // reproject
                float4 currentFrameClipSpace = mul(viewInfo.ProjectionMat,
                    mul(viewInfo.ViewMat, prevFrameViewSpacePos));
                float3 currentFrameNDC = currentFrameClipSpace.xyz /
                    currentFrameClipSpace.w;
                currentFrameNDC.y = 1.0 - currentFrameNDC.y;
                float2 currentFrameScreenPos = saturate((currentFrameNDC.xy
                    + 1.0) * 0.5) * float2(HALF_SCREEN_WIDTH,
                    HALF_SCREEN_HEIGHT);

                int2 currentFrameReprojectScreenPos = floor(
                    currentFrameScreenPos);
                InterlockedMax(reprojDepthBuffer[
                    currentFrameReprojectScreenPos], asuint(currentFrameNDC.
                    z + DEPTH_BIAS));
        }
}
```

Figure 2.4: The downsampled reprojected depth buffer might look like this.

Downsample

As can be seen in 2.3, floating point inaccuracy and missing information can cause artifacts in the final reprojected depth buffer. Since these lines are reprojected to the far plane; all occludee geometry rasterized against these lines will pass the visibility test, which results in visually occluded draw calls being submitted and processed without any visual contribution to the scene. An extra downsample pass is performed upon the reprojected depth buffer to minimize the artifacts and acquire a solid depth buffer reprojection.

```
SamplerState r_point_clamp : register(s2);

Texture2D<float>        reprojDepthBuffer               : register(t0); //
    SRV
RWTexture2D<float>      downSampleDepthBuffer   : register(u1); // UAV

#define allZero(x) (!any(x))

[numthreads(NUM_THREADS_X, NUM_THREADS_Y, NUM_THREADS_Z)]
void CSMain(uint3 pixelID : SV_DispatchThreadID)
{
        if (pixelID.x < QUART_SCREEN_WIDTH &&
                pixelID.y < QUART_SCREEN_HEIGHT)
```

```
        {
                float2 uv = float2(pixelID.xy + 0.5) / float2(
                    QUART_SCREEN_WIDTH, QUART_SCREEN_HEIGHT);

                float4 gatheredSamples = reprojDepthBuffer.GatherRed(
                    r_point_clamp, uv, uint2(0, 0));

                if (allZero(gatheredSamples))
                {
                        downSampleDepthBuffer[pixelID.xy] = 1.0;
                }
                else
                {
                        // we always take the maximum value of the previous
                            depth buffer, to make sure that we always have
                            the worst case depth and dont occlude visible
                            geometry
                        float downSampledDepth = max(gatheredSamples.x, max(
                            gatheredSamples.y, max(gatheredSamples.z,
                            gatheredSamples.w)));

                        downSampleDepthBuffer[pixelID.xy] = downSampledDepth
                            ;
                }
        }
}
```

Adding Occluder Geometry By Rasterization

Because the reprojected depth buffer is limited to static screenspace information and dynamic objects can not be reprojected; a game will often require additional occluder geometry in the reprojected depth buffer. By rasterizing the *minimum bounding volume* as occlusion geometry into the reprojected depth buffer this is easily achieved. Similarly missing occluder information on the edges of the screen can be added by rasterizing occluder geometry from the objects that collide with the cameras frustum. This would result in better overall culling efficiency.

4 Determine Visibility

To determine visibility, we rasterize the occludee bounding volumes against the reprojected depth buffer. After the visibility buffer generation, a gather pass will create a new indirect command list with the non occluded draw calls.

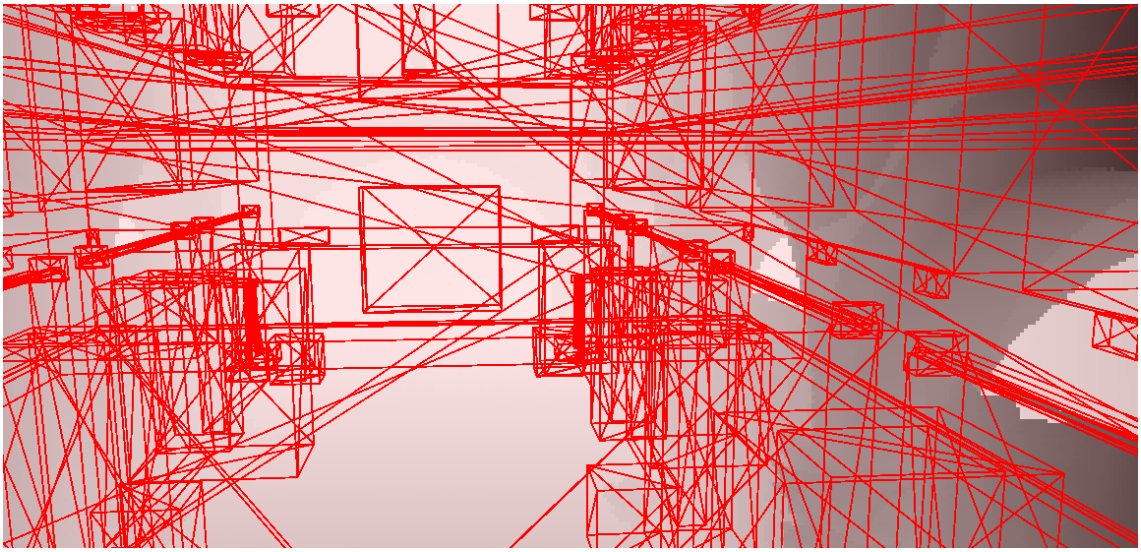

Figure 2.5: Rasterization of occludee volumes against the reprojected depth buffer.

4.1 Implementation

Fill Visibility Buffer

The first step of visibility determination is rasterizing the occludee bounding volumes against the downsampled reprojected depth buffer as can be seen in Figure 2.5. For this pass we make sure that no render target is bound since we are not interested in visual results. The fragments will be depth tested less then or equal to the reprojected depth buffer. If the fragments pass this test, we write a 1 in the visibility buffer[2] at the location of the mesh index. Note that the PSMain shader has a race condition when writing to the visibility buffer. This does not give any issues, since the output value is static per occludee.

- The *Vertex Shader* transforms the occludee bounding volume to clip space so that the fragments can be depth tested against the reprojected depth buffer. It also clears the visibility buffer.

```
PSInput VSMain(VSInput vsInput, uint objectIndex : SV_InstanceID)
{
        PSInput result;

        float3 modelPos = mul(r_modelMatrices[objectIndex], float4(
            vsInput.position.xyz,1.0));
```

```
        float3 worldPos = mul(r_worldMatrices[objectIndex], float4(
            modelPos, 1.0));

        result.position = mul(viewInfo.ProjectionMat, mul(viewInfo.
            ViewMat, float4(worldPos, 1.0)));

        result.objectIndex = objectIndex;

        // reset visibility
        r_visibility[objectIndex] = 0;

        return result;
}
```

- The *Pixel Shader* populates the visibility buffer. *Note the [earlydepthstencil] compiler setting, this tells the HLSL compiler to force early depth-stencil testing prior to fragment shader execution*[3].

```
[earlydepthstencil]
void PSMain(PSInput input)
{
        // if the pixel shader is invoked, the fragment is visible and
             we thus need to render the object
        r_visibility[input.objectIndex] = 1;
}
```

Populate Indirect Draw Argument Buffer

To populate the indirect draw argument buffer, we gather the visible draw calls from the predefined GPU indirect draw argument buffer, this buffer holds all the draw call arguments for the scene and its draw arguments are mapped to the occludee indices; therefore the visibility buffer maps to the corresponding draw arguments.

- The *Compute Shader* walks over each visibility buffer entry and appends its corresponding draw arguments to the execute indirect draw arguments buffer when the visibility buffer entry is set to 1.

```
StructuredBuffer<uint>
            r_visibility            : register(t0); // SRV
StructuredBuffer<IndirectCommandArgs>
    r_inputCommands         : register(t1); // SRV
AppendStructuredBuffer<IndirectCommandArgs>             outputCommands
              : register(u2); // UAV

void VSMain(uint vertexID : SV_VertexID)
{
        if (r_visibility[vertexID] == 1)
        {
```

```
                outputCommands.Append(r_inputCommands[vertexID]);
        }
}
```

- On some GPUs there is a slight performance improvement when a *vertex shader* is used instead of a *compute shader* when the work group count is of a relatively small amount[1]. Note that this did not work on the AMD that we have tested.

```
// EXECUTE INDIRECT HELPER STRUCTS
struct VertexBufferView
{
        uint2 Address;
        uint Size;
        uint Stride;
};

struct IndexBufferView
{
        uint2 Address;
        uint Size;
        uint Format;
};

struct DrawIndexedInstancedArgs
{
        uint IndexCountPerInstance;
        uint InstanceCount;
        uint StartIndexLocation;
        int  BaseVertexLocation;
        uint StartInstanceLocation;
};

struct IndirectCommandArgs
{
        IndexBufferView IndexBuffer;
        VertexBufferView VertexBuffer;
        uint modelMatrixIndex;
        uint albedoTextureIndex;
        DrawIndexedInstancedArgs DrawIndexedInstanced;
        uint _padding;
};


cbuffer rc_constants : register(b3)
{
        uint maxNumIndirectCommands;
};

StructuredBuffer<uint>
            r_visibility            : register(t0); // SRV
StructuredBuffer<IndirectCommandArgs>
    r_inputCommands         : register(t1); // SRV
AppendStructuredBuffer<IndirectCommandArgs>             outputCommands
              : register(u2); // UAV
```

```
[numthreads(NUM_THREADS_X, NUM_THREADS_Y, NUM_THREADS_Z)]
void CSMain(uint3 groupId : SV_GroupID, uint groupIndex :
    SV_GroupIndex)
{
        uint meshIndex = (groupId.x * NUM_THREADS_X) + groupIndex;

        if (meshIndex < maxNumIndirectCommands)
        {
                if (r_visibility[meshIndex] == 1)
                {
                        outputCommands.Append(r_inputCommands[
                            meshIndex]);
                }
        }
}

// vertex shader variant of compute shader written above
void VSMain(uint meshIndex : SV_VertexID)
{
        if (r_visibility[meshIndex] == 1)
        {
                outputCommands.Append(r_inputCommands[meshIndex]);
        }
}
```

5 Render Visible Geometry

The pipeline described so far has worked towards supplying the execute indirect command pipeline with the required data. Now that the data is gathered, execute indirect can render the visibly contributing geometry based on the occlusion results.

5.1 Implementation

When using execute indirect, there are certain limitations that need to be taken into account when designing the rendering pipeline for the game. An example would be that all the potentially requested textures need to be bound to the execute indirect program, since the GPU has no prior knowledge on which objects will be occluded.

Execute Indirect Limitations

By using execute indirect, there are certain limitations to the ability of the indirect buffer. The limitations are listed below and should be taken into

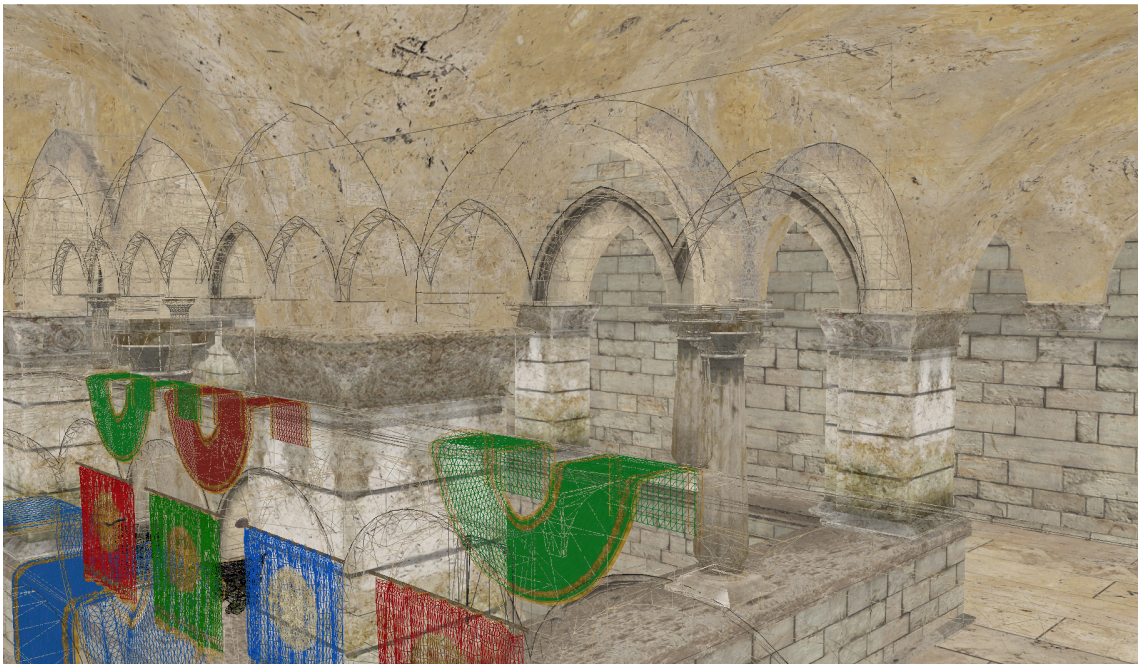

Figure 2.6: An interior view of sponza that shows the non occluded geometry. The wireframe visualizes the non-occluded geometry on top, which makes it easy to see potential errors. Note that the occlusion culling does its job and only renders visually contributing geometry in this view.

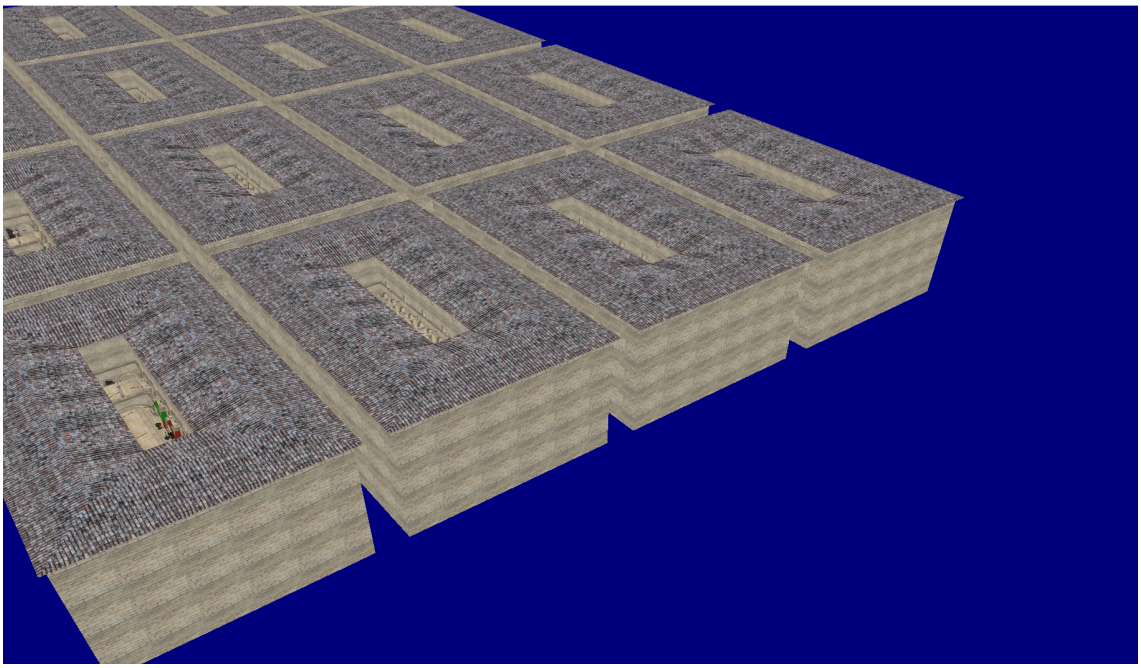

Figure 2.7: Sponza in a 4x4 grid.

Figure 2.8: The difference between GPU occlusion culled and regularly rendered scene shown in 2.7. Chapter 3 section 1.2 shows the performance gains for this particular view.

account when deciding to move towards indirect rendering. *Note that the limitations listed below are for DX12 and might differ from limitations or possibilities with other APIs*

- *Only ability to set Root Constants as value, Root SRV (buffer only), Root UAV (buffer only), Root CBV, Vertex Buffers and Index Buffer.*
- *The execute indirect call must end with either a draw call or a dispatch call*; so adjusting root views or constants is only allowed before the draw or dispatch call.

For this example, a simple vertex and pixel shader are used to demonstrate the rendering ability. We use an SRV texture array to render the multiple meshes with individual textures.

- *Vertex Shader*

```
cbuffer r_cbvViewInfo : register(b0)
{
        struct viewStateStruct
        {
                float4x4 ProjectionMat;
                float4x4 ViewMat;
        } viewInfo;
};

cbuffer rc_constants : register(b1)
{
        uint modelMatrixIndex;
        uint albedoTextureIndex;
};

StructuredBuffer<float4x4> r_modelMatrices : register(t28);

Texture2D<float4> albedoTextures[26] : register(t2);

// NOTE: r_linear_wrap is a static sampler declared in GFX Root
    Signature extraction
// NOTE: r_linear_wrap samples MinMagMip Linear, UV Wrap
SamplerState r_linear_wrap : register(s3);

struct VSInput
{
        float3 position : POSITION;
        float2 uvCoords0 : TEXCOORD;
};

struct PSInput
{
        float4 position : SV_POSITION;
        float2 uvCoords0 : TEXCOORD0;
```

```
};

PSInput VSMain( VSInput vsInput, uint instanceIndex : SV_InstanceID)
{
        PSInput vsOUT;

        const float3 worldPos = mul(r_modelMatrices[modelMatrixIndex],
             float4(vsInput.position, 1.0));
        vsOUT.position = mul(viewInfo.ProjectionMat, mul(viewInfo.
            ViewMat, float4(worldPos, 1.0)));
        vsOUT.uvCoords0 = vsInput.uvCoords0.xy;

        return vsOUT;
}
```

- *Pixel Shader*

```
float4 PSMain(PSInput psIN) : SV_TARGET
{
        float4 outColor = albedoTextures[albedoTextureIndex].Sample(
            r_linear_wrap, psIN.uvCoords0);

        return outColor;
}
```

Chapter 3

Results

1 Results

To determine the performance of GPU occlusion culling in a game like environment, several cameras have been added to the Sponza scene. At these locations benchmarks are performed which will showcase the culling performance. A moving camera was added to give the users an idea on the culling performance in a dynamic case. Note that the static cameras could improve their runtime performance by disabling the reprojection and downsample pass, since a regular static geometry depth buffer would suffice for the culling operations. This was explicitly not done in this benchmark to make sure that the cost of reprojection and downsampling is visible.

1.1 Hardware

The benchmark locations have been measured on various hardware configurations to see the differences between hardware vendors.

Specs	Nvidia 950 GTX	Intel 5100 Iris
OS	Windows 10 x64	Windows 10 x64
GPU	Nvidia 950 GTX	Intel 5100 Iris
GPU VRam	2048MB GDDR5	1.5GB of System Ram
GPU Driver	368.22	20.19.15.4424
DX12 Feature Level	12_1	12_0
CPU	AMD 8350FX Eight-Core at 4GHz	Intel i5-4278U at 2.6GHz
RAM	16GB	8GB

Figure 3.1: View from camera one.

1.2 Timings

For the benchmarks a windowed resolution of 1280x720 was used.

Camera 1

The view from camera one is shown in Figure 3.1. Note that this view is located facing away from the Sponza grid, therefore frustum culling for the CPU rendering path is most effective.

Camera 1	GPU Occlusion Culled			CPU Frustum Culled		
Nvidia 950GTX	min	max	average	min	max	average
Total Draw Time	0.327ms	0.419ms	0.400ms	0.198ms	0.259ms	0.220ms
Clear UAV	0.010ms	0.011ms	0.011ms	0.000ms	0.000ms	0.000ms
Reproject	0.052ms	0.055ms	0.053ms	0.000ms	0.000ms	0.000ms
Downsample Reproject	0.016ms	0.019ms	0.017ms	0.000ms	0.000ms	0.000ms
Copy Reproject to Depth	0.010ms	0.011ms	0.011ms	0.000ms	0.000ms	0.000ms
Rasterize Occludees	0.048ms	0.066ms	0.053ms	0.000ms	0.000ms	0.000ms
Clear Append Buffer	0.002ms	0.003ms	0.003ms	0.000ms	0.000ms	0.000ms
Gather Visible Objects	0.006ms	0.008ms	0.007ms	0.000ms	0.000ms	0.000ms
Draw Visible Objects	0.154ms	0.246ms	0.221ms	0.000ms	0.000ms	0.000ms

Camera 1	GPU Occlusion Culled			CPU Frustum Culled		
Intel 5100 Iris	min	max	average	min	max	average
Total Draw Time	4.958ms	6.271ms	5.477ms	1.887ms	4.884ms	3.115ms
Clear UAV	0.067ms	0.103ms	0.072ms	0.000ms	0.000ms	0.000ms
Reproject	1.988ms	2.414ms	2.133ms	0.000ms	0.000ms	0.000ms
Downsample Reproject	0.038ms	0.092ms	0.049ms	0.000ms	0.000ms	0.000ms
Copy Reproject to Depth	0.016ms	0.037ms	0.020ms	0.000ms	0.000ms	0.000ms
Rasterize Occludees	0.782ms	1.185ms	0.926ms	0.000ms	0.000ms	0.000ms
Clear Append Buffer	0.006ms	0.016ms	0.007ms	0.000ms	0.000ms	0.000ms
Gather Visible Objects	0.008ms	0.018ms	0.009ms	0.000ms	0.000ms	0.000ms
Draw Visible Objects	1.322ms	2.020ms	1.694ms	0.000ms	0.000ms	0.000ms

Camera 2

The view from camera two is shown in Figure 3.2. Note that this view is located facing towards the Sponza grid, therefore frustum culling for the CPU rendering path is less effective.

Figure 3.2: View from camera two.

Camera 2	GPU Occlusion Culled			CPU Frustum Culled		
Nvidia 950GTX	min	max	average	min	max	average
Total Draw Time	0.660ms	0.841ms	0.777ms	2.587ms	2.907ms	2.625ms
Clear UAV	0.010ms	0.012ms	0.011ms	0.000ms	0.000ms	0.000ms
Reproject	0.059ms	0.063ms	0.060ms	0.000ms	0.000ms	0.000ms
Downsample Reproject	0.016ms	0.019ms	0.018ms	0.000ms	0.000ms	0.000ms
Copy Reproject to Depth	0.010ms	0.011ms	0.011ms	0.000ms	0.000ms	0.000ms
Rasterize Occludees	0.062ms	0.074ms	0.066ms	0.000ms	0.000ms	0.000ms
Clear Append Buffer	0.003ms	0.004ms	0.003ms	0.000ms	0.000ms	0.000ms
Gather Visible Objects	0.010ms	0.013ms	0.011ms	0.000ms	0.000ms	0.000ms
Draw Visible Objects	0.445ms	0.643ms	0.573ms	0.000ms	0.000ms	0.000ms

Camera 2	GPU Occlusion Culled			CPU Frustum Culled		
Intel 5100 Iris	min	max	average	min	max	average
Total Draw Time	6.737ms	8.401ms	7.401ms	18.712ms	23.479ms	20.795ms
Clear UAV	0.067ms	0.102ms	0.072ms	0.000ms	0.000ms	0.000ms
Reproject	1.926ms	2.350ms	2.082ms	0.000ms	0.000ms	0.000ms
Downsample Reproject	0.039ms	0.085ms	0.049ms	0.000ms	0.000ms	0.000ms
Copy Reproject to Depth	0.016ms	0.036ms	0.020ms	0.000ms	0.000ms	0.000ms
Rasterize Occludees	1.075ms	1.490ms	1.233ms	0.000ms	0.000ms	0.000ms
Clear Append Buffer	0.006ms	0.015ms	0.007ms	0.000ms	0.000ms	0.000ms
Gather Visible Objects	0.011ms	0.029ms	0.014ms	0.000ms	0.000ms	0.000ms
Draw Visible Objects	2.897ms	4.140ms	3.443ms	0.000ms	0.000ms	0.000ms

Camera 3

The view from camera three is shown in Figure 3.3. Note that this view is located as an overview on the Sponza grid, therefore frustum culling for the CPU rendering path is less effective.

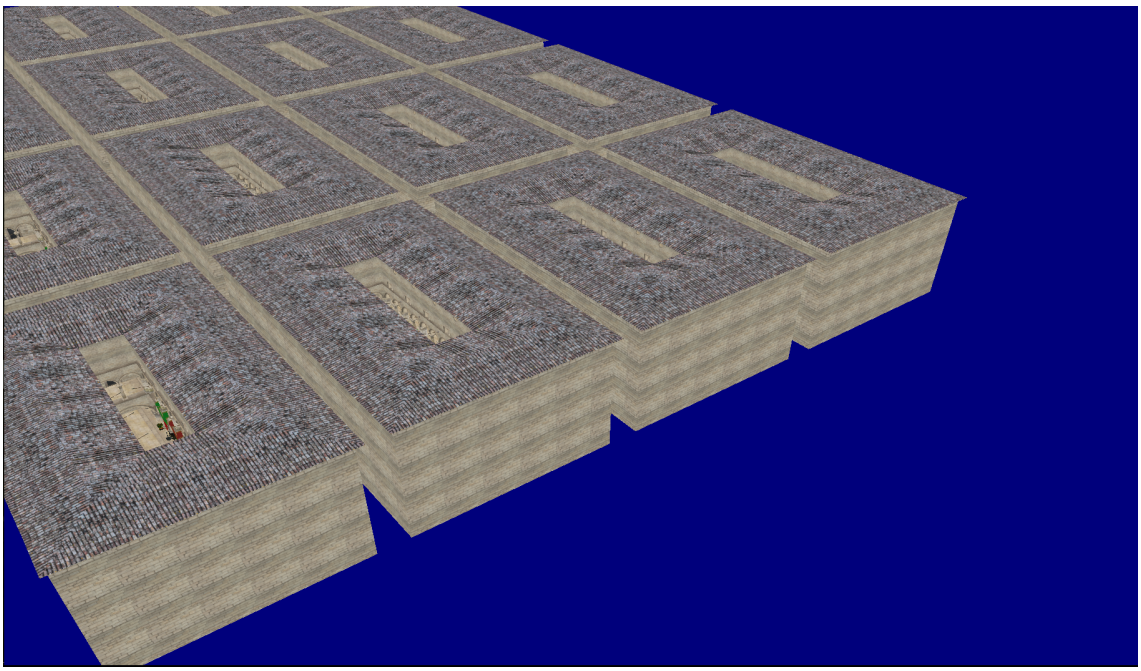

Figure 3.3: View from camera three.

Camera 3	GPU Occlusion Culled			CPU Frustum Culled		
Nvidia 950GTX	min	max	average	min	max	average
Total Draw Time	0.680ms	0.745ms	0.718ms	3.412ms	3.566ms	3.425ms
Clear UAV	0.011ms	0.012ms	0.011ms	0.000ms	0.000ms	0.000ms
Reproject	0.053ms	0.056ms	0.054ms	0.000ms	0.000ms	0.000ms
Downsample Reproject	0.016ms	0.018ms	0.017ms	0.000ms	0.000ms	0.000ms
Copy Reproject to Depth	0.010ms	0.011ms	0.011ms	0.000ms	0.000ms	0.000ms
Rasterize Occludees	0.087ms	0.094ms	0.089ms	0.000ms	0.000ms	0.000ms
Clear Append Buffer	0.003ms	0.003ms	0.003ms	0.000ms	0.000ms	0.000ms
Gather Visible Objects	0.011ms	0.014ms	0.012ms	0.000ms	0.000ms	0.000ms
Draw Visible Objects	0.461ms	0.520ms	0.496ms	0.000ms	0.000ms	0.000ms

Camera 3	GPU Occlusion Culled			CPU Frustum Culled		
Intel 5100 Iris	min	max	average	min	max	average
Total Draw Time	6.703ms	8.485ms	7.208ms	23.944ms	31.509ms	26.701ms
Clear UAV	0.067ms	0.102ms	0.072ms	0.000ms	0.000ms	0.000ms
Reproject	0.753ms	2.529ms	2.276ms	0.000ms	0.000ms	0.000ms
Downsample Reproject	0.038ms	0.091ms	0.052ms	0.000ms	0.000ms	0.000ms
Copy Reproject to Depth	0.016ms	0.037ms	0.020ms	0.000ms	0.000ms	0.000ms
Rasterize Occludees	0.776ms	1.153ms	0.912ms	0.000ms	0.000ms	0.000ms
Clear Append Buffer	0.006ms	0.015ms	0.007ms	0.000ms	0.000ms	0.000ms
Gather Visible Objects	0.018ms	0.038ms	0.022ms	0.000ms	0.000ms	0.000ms
Draw Visible Objects	2.907ms	4.360ms	3.297ms	0.000ms	0.000ms	0.000ms

Camera 4

The view from camera four is shown in Figure 3.5. Note that this view is located as a top down view of the Sponza grid. Frustum culling for the CPU rendering path is less effective, there is also limited occlusion due to the top intake of the buildings.

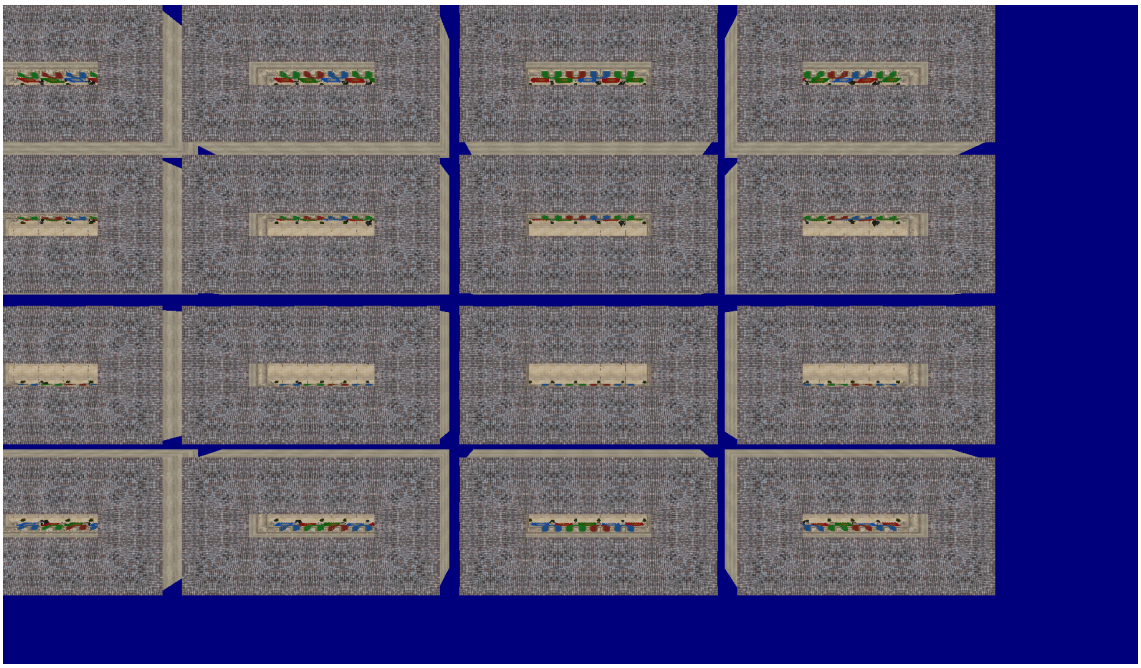

Figure 3.4: View from camera four.

Camera 4	GPU Occlusion Culled			CPU Frustum Culled		
Nvidia 950GTX	min	max	average	min	max	average
Total Draw Time	2.506ms	2.815ms	2.555ms	3.623ms	4.798ms	3.959ms
Clear UAV	0.011ms	0.011ms	0.011ms	0.000ms	0.000ms	0.000ms
Reproject	0.058ms	0.063ms	0.061ms	0.000ms	0.000ms	0.000ms
Downsample Reproject	0.016ms	0.019ms	0.017ms	0.000ms	0.000ms	0.000ms
Copy Reproject to Depth	0.010ms	0.011ms	0.011ms	0.000ms	0.000ms	0.000ms
Rasterize Occludees	0.122ms	0.126ms	0.124ms	0.000ms	0.000ms	0.000ms
Clear Append Buffer	0.003ms	0.003ms	0.003ms	0.000ms	0.000ms	0.000ms
Gather Visible Objects	0.023ms	0.027ms	0.025ms	0.000ms	0.000ms	0.000ms
Draw Visible Objects	2.228ms	2.533ms	2.276ms	0.000ms	0.000ms	0.000ms

Camera 4	GPU Occlusion Culled			CPU Frustum Culled		
Intel 5100 Iris	min	max	average	min	max	average
Total Draw Time	13.28ms	17.86ms	14.91ms	26.56ms	46.95ms	29.09ms
Clear UAV	0.06ms	2.21ms	0.25ms	0.00ms	0.00ms	0.00ms
Reproject	0.70ms	4.04ms	1.58ms	0.00ms	0.00ms	0.00ms
Downsample Reproject	0.04ms	0.10ms	0.05ms	0.00ms	0.00ms	0.00ms
Copy Reproject to Depth	0.02ms	0.04ms	0.02ms	0.00ms	0.00ms	0.00ms
Rasterize Occludees	0.88ms	1.27ms	0.98ms	0.00ms	0.00ms	0.00ms
Clear Append Buffer	0.01ms	0.014ms	0.01ms	0.00ms	0.00ms	0.00ms
Gather Visible Objects	0.05ms	0.09ms	0.06ms	0.00ms	0.00ms	0.00ms
Draw Visible Objects	10.92ms	13.26ms	11.47ms	0.00ms	0.00ms	0.00ms

Fly Through

This view starts within a Sponza building and flies over the building into another Sponza building. Note that this view flies towards the outer bounds of the Sponza grid and therefore frustum culling for the CPU rendering path is highly effective.

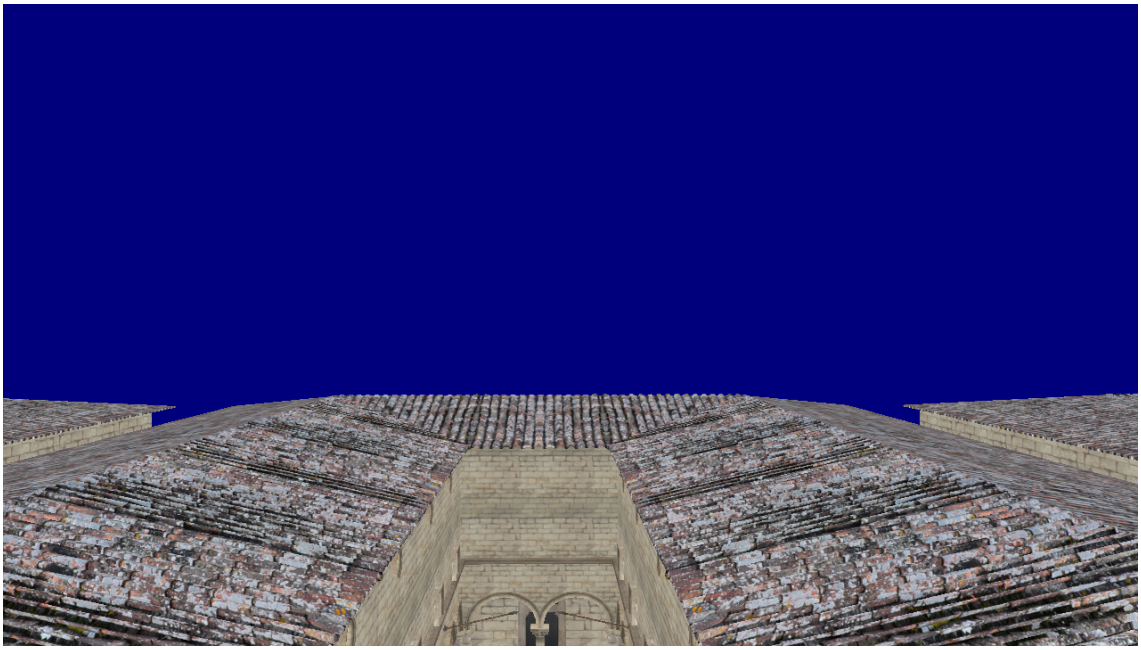

Figure 3.5: Moment capture from fly through camera.

Fly Through Camera	GPU Occlusion Culled			CPU Frustum Culled		
Nvidia 950GTX	min	max	average	min	max	average
Total Draw Time	0.324ms	1.928ms	0.567ms	0.082ms	2.624ms	0.990ms
Clear UAV	0.010ms	0.011ms	0.011ms	0.000ms	0.000ms	0.000ms
Reproject	0.052ms	0.062ms	0.056ms	0.000ms	0.000ms	0.000ms
Downsample Reproject	0.014ms	0.020ms	0.017ms	0.000ms	0.000ms	0.000ms
Copy Reproject to Depth	0.010ms	0.011ms	0.011ms	0.000ms	0.000ms	0.000ms
Rasterize Occludees	0.050ms	0.096ms	0.064ms	0.000ms	0.000ms	0.000ms
Clear Append Buffer	0.002ms	0.003ms	0.003ms	0.000ms	0.000ms	0.000ms
Gather Visible Objects	0.006ms	0.018ms	0.009ms	0.000ms	0.000ms	0.000ms
Draw Visible Objects	0.142ms	1.724ms	0.373ms	0.000ms	0.000ms	0.000ms

Fly Through Camera	GPU Occlusion Culled			CPU Frustum Culled		
Intel 5100 Iris	min	max	average	min	max	average
Total Draw Time	4.300ms	10.764ms	6.092ms	2.491ms	22.761ms	6.940ms
Clear UAV	0.067ms	0.099ms	0.072ms	0.000ms	0.000ms	0.000ms
Reproject	0.767ms	2.570ms	2.291ms	0.000ms	0.000ms	0.000ms
Downsample Reproject	0.039ms	0.110ms	0.060ms	0.000ms	0.000ms	0.000ms
Copy Reproject to Depth	0.016ms	0.048ms	0.023ms	0.000ms	0.000ms	0.000ms
Rasterize Occludees	0.726ms	1.537ms	1.020ms	0.000ms	0.000ms	0.000ms
Clear Append Buffer	0.005ms	0.015ms	0.007ms	0.000ms	0.000ms	0.000ms
Gather Visible Objects	0.008ms	0.036ms	0.013ms	0.000ms	0.000ms	0.000ms
Draw Visible Objects	0.436ms	6.701ms	2.023ms	0.000ms	0.000ms	0.000ms

Chapter 4

Conclusion

1 Conclusion

By studying the results we can see that GPU occlusion culling improves the rendering performance greatly without compromising on visual fidelity. In all but one of the tests cases on both Nvidia and Intel, GPU occlusion culling beats the CPU frustum culled technique. The only exception is the first test case, in which frustum culling plays a major role in culling the scene. We can see that the extra steps that are required for GPU occlusion culling with a moving camera, account for the overhead. Potentially these steps can be optimized to make the difference in cases such as these smaller. We can therefore conclude that GPU occlusion culling is a great improvement to a games rendering pipeline and can save performance up to 4.8 times[1]. GPU occlusion culling allowed all the test cases to perform in real-time 60fps on Intel and Nvidia; where the regular rendering path was regularly rendering around 30fps on the Intel test machine. The main challenge in using this technique is to setup a rendering pipeline that drives the indirect argument buffers. Reprojection and occludee rasterization are fairly straight forward to implement.

[1] Camera 3 for the Nvidia GTX 950

2 Future Work

2.1 Improvements Upon Depth Reprojection

Some of the proposed upgrades to this algorithm would cover improving the worst case scenarios that occur due to missing world space information in the reprojection step when moving or rotating quickly. Another improvement that I would have liked to have tried would be the addition of a dynamic occluder path as an example on how to rasterize a minimum bounding volume for dynamic objects.

2.2 Incorporating Level of Detail

An addition to the GPU based rendering pipeline could be to add an extra pass to the gather step. This pass would determine the meshes LOD level. Multiple append buffers could be implemented that then collect the visible draw arguments, sorted on LOD. By calling execute indirect for each of these append buffers, performance could be gained by rendering objects that are small or far away in a higher LOD level. Making the transformation and potentially the shading stage more efficient.

2.3 GPU driven Scene Rendering

Another addition towards a fully GPU driven scene renderer would be keep the objects in the scene in caches on the GPU. As soon as an object has not been referenced for X amount of frames and when this object is not in close proximity to the view frustum, this object could be removed from GPU memory and new objects could be uploaded into its space. This would enable full GPU driven scene management for open world games. To let the CPU know which objects can be overwritten, a readback buffer could be used that is periodically send to main system RAM to be read by the CPU. To make sure that popping does not happen, the highest LODs of all models could be stored in GPU memory at all times.

3 Acknowledgment

I would like to thank Leroy Sikkes for co-creation of the rendering framework that is extensively used in this project.

Special thanks go out to Daniel Wustenhoff, Jeremiah van Oosten, Jurre de Baare, Leroy Sikkes, Vladimir Bondarev and my collegues at Confetti Interactive for proof-reading, sparring ideas and supporting me in this project.

Bibliography

[1] Pierre Boudier Christoph Kubisch. Gpu-driven rendering. GPU Tech Conference - http://on-demand.gputechconf.com/gtc/2016/presentation/s6138-christoph-kubisch-pierre-boudier-gpu-driven-rendering.pdf, 2016.

[2] Evgeny Makarov. Advanced scene graph rendering pipeline. GPU Tech Conference - http://on-demand.gputechconf.com/gtc/2013/presentations/S3032-Advanced-Scenegraph-Rendering-Pipeline.pdf, 2013.

[3] Microsoft. *earlydepthstencil*, 2016. MSDN - https://msdn.microsoft.com/en-us/library/windows/desktop/ff471439(v=vs.85).aspx.

[4] Alex Fit-Florea Nathan Whitehead. *Precision and Performance: Floating Point and IEEE 754 Compliance for NVIDIA GPUs*. NVIDIA. http://developer.download.nvidia.com/assets/cuda/files/NVIDIA-CUDA-Floating-Point.pdf.

[5] Sean Pelletier. Why windows 10 and directx 12 represent a big step forward for gaming. Blog - https://blogs.nvidia.com/blog/2015/07/29/directx-12-windows-10/, 7 2015.