Master of Science in Engineering: Game and Software Engineering
January 29, 2023

Technical Report

A Render Graph Implementation for Direct3D 12

PA2526: Stort spelprojekt

Nadhif Ginola

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

Summary

This report is intended to cover the theory, detailed implementation, and profiling results of a render graph implemented in Direct3D 12 for the game Rogue Robots.

A render graph's primary purpose is to alleviate developer burden when implementing rendering effects by automating tedious processes. Moreover, it can also reduce GPU resource memory usage by utilizing memory aliasing.

The proposed render graph is conceptually built on a task graph which is modified to support cases unique to rendering, supports naive memory aliasing, and omits issues such as load-balancing and GPU multi-queue support.

Results show that the render graph scales poorly in performance for both execution and building, where a graph with 64 passes or below is preferred, but the solution performs sufficiently well given the context of the game project.

In conclusion, implementing a render graph was a great learning experience and it has made the implementation and modification of rendering effects easier for all contributors to the project, along with VRAM savings.

Contents

Chapter 1

Introduction

Writing code for Direct3D 12 can be tedious, where low-level memory, and resources and their states need to be manually managed along with resource access synchronization. As a result, implementing rendering effects may take a lot of time. A render graph is a solution to many such problems.

This report will present the theory, implementation, and profiling results of the render graph used for Rogue Robots.

Chapter 2

Theory

2.1 Direct3D 12

Direct3D 12 is an application programming interface (API) that provides low-level access to the graphics processing unit (GPU) and its resources. The application is responsible for managing the GPU memory and subsequent resource allocations [1]. Using Direct3D 12, the GPU can be leveraged in order to rasterize images for real-time rendering using polygon meshes and material properties.

2.2 Resource State Management

In Direct3D 12, the application must ensure that resources are in the correct state before issuing commands which access them. In addition, the application must also ensure that GPU resources are not modified or removed while still in use. This can be challenging, as the responsibility for managing the hardware state was largely on graphics drivers prior to Direct3D 12. To assist with this process, the API introduces several new features such as command lists for recording commands, and resource barriers for synchronization and state management.

2.2.1 Resource Barrier

A resource is a set of subresources and barriers manage state on a per-subresource level. There exist three types of resource barriers: Transition, Unordered Access View (UAV), and Aliasing. Transition barriers are used to transition subresources from one state to another, UAV barriers are used to make UAV (read-write) results visible to subsequent UAV accesses of the resource, and aliasing barriers are used to "activate" a resource that has an overlapped memory mapping with another resource.

2.3 Memory Aliasing

In order to reduce memory usage in Direct3D 12, memory is re-used. This is done by identifying mutually exclusive use of GPU resources in time, therefore allowing safe memory re-use [2]. In turn, a degree of foresight on the applications resource usage is required to appropriately allocate memory for optimal memory re-use.

2.4 Render Pass

Conceptually, a render pass can refer to any GPU workload, which may consume input data and always generates a set of output data [3]. Outputs are typically rendered images, but can also be the results of compute shader work stored in buffers. Such outputs can then be consumed as inputs in subsequent render passes.

2.5 Task Graph

A task graph is commonly a Directed Acyclic Graph (DAG) where a vertex in the graph represents an atomic unit of work (task) and where a directed edge "A $\rightarrow$ B" represents a work order dependency where A must be done before B. Once the DAG has been in topological order, the graph execution is complete. Its acyclic nature prevents deadlocks and ill-defined traversal paths [4].

Chapter 3

Methodology

This chapter explains, in detail, the problem domain, what the render graph is in the context of this project, and how the render graph was designed and implemented.

3.1 Problem Domain

The primary aspects that this render graph deals with are usability and GPU memory usage.

Given that the game can only consume 512 megabytes (MB) of GPU video memory at most, it was critical to implement measures that allow memory usage reductions. The purpose of the graph is to ease the developer's burden when implementing rendering code. Additionally, load balancing and multi-queue GPU operation and synchronization issues are omitted due to their complexity and limited implementation time.

3.2 User-facing API

The user-facing API is based on the API exposed by Frostbite's Frame Graph by DICE and relies on supplying C++ lambda expressions to declare resources, their usage intent, and the runtime rendering code [5]. For simplicity, a lambda expression can be defined as a function that can be called at a later point in time. An example of a task submission to the graph can be seen in Figure 3.1, or Figure 3.2.

```
rg.AddPass<ShadowPassData>("Shadow Pass",
    [&](ShadowPassData&, RenderGraph::PassBuilder& builder)
    {
        builder.DeclareTexture(RG_RESOURCE(ShadowDepth), RGTextureDesc::DepthWrite2D(DepthFormat::D32, 1024, 1024, m_shadowMapCapacity));
        builder.WriteDepthStencil(RG_RESOURCE(ShadowDepth), RenderPassAccessType::ClearPreserve,
            TextureViewDesc(ViewType::DepthStencil, TextureViewDimension::Texture2D_Array, DXGI_FORMAT_D32_FLOAT)
            .SetArrayRange(0, m_shadowMapCapacity));

    },
    [&, shadowDrawFunc = shadowDrawSubmissions](const ShadowPassData&, RenderDevice* rd, CommandList cmdl, RenderGraph::PassResources&) mutable
    {
        rd->Cmd_SetViewports(cmdl, Viewports().Append(0.f, 0.f, 1024.f, 1024.f));
        rd->Cmd_SetScissorRects(cmdl, ScissorRects().Append(0, 0, 1024, 1024));

        rd->Cmd_SetIndexBuffer(cmdl, m_globalEffectData.meshTable->GetIndexBuffer());

        // Fills shadowmaps chronologically
        rd->Cmd_SetPipeline(cmdl, m_shadowPipe);
        u32 nextMap = 0;
        for (u32 i = 0; i < m_activeSpotlights.size(); ++i)
        {
            if (m_activeSpotlights[i].shadow)
                shadowDrawFunc(rd, cmdl, m_singleSidedShadowDraws[m_activeShadowCasters[i].singleSidedBucket], nextMap++, m_activeSpotlights[i].shadow.value());
        }
    });
```

Figure 3.1: Pass to render scene depth into a texture

```
struct PassData
{
    RGResourceView input;
    RGResourceView output;
};

rg.AddPass<PassData>("SSAO Blur Horizontal",
    [&](PassData& passData, RenderGraph::PassBuilder& builder)
    {
        builder.DeclareTexture(RG_RESOURCE(AOBlurred), RGTextureDesc::ReadWrite2D(DXGI_FORMAT_R16G16B16A16_FLOAT, m_renderWidth / 2, m_renderHeight / 2));

        passData.input = builder.ReadResource(RG_RESOURCE(AOBlurredUnfinished), D3D12_RESOURCE_STATE_NON_PIXEL_SHADER_RESOURCE,
            TextureViewDesc(ViewType::ShaderResource, TextureViewDimension::Texture2D, DXGI_FORMAT_R16G16B16A16_FLOAT));
        passData.output = builder.ReadWriteTarget(RG_RESOURCE(AOBlurred),
            TextureViewDesc(ViewType::UnorderedAccess, TextureViewDimension::Texture2D, DXGI_FORMAT_R16G16B16A16_FLOAT));

    },
    [&](const PassData& passData, RenderDevice* rd, CommandList cmdl, RenderGraph::PassResources& resources)
    {
        // clear
        rd->Cmd_ClearUnorderedAccessFLOAT(cmdl,
            resources.GetTextureView(passData.output), { 1.f, 1.f, 1.f, 1.f }, ScissorRects().Append(0, 0, m_renderWidth / 2, m_renderHeight / 2));

        if (m_graphicsSettings.ssao)
        {
            rd->Cmd_SetPipeline(cmdl, m_boxBlurPipe);
            {
                auto args = ShaderArgs()
                    .AppendConstant(m_globalEffectData.globalDataDescriptor)
                    .AppendConstant(m_currPfDescriptor)
                    .AppendConstant((u32)(m_renderWidth / 2.f))
                    .AppendConstant((u32)(m_renderHeight / 2.f))
                    .AppendConstant(resources.GetView(passData.input))
                    .AppendConstant(resources.GetView(passData.output))
                    .AppendConstant(0)
                    .AppendConstant(1);
                rd->Cmd_UpdateShaderArgs(cmdl, QueueType::Compute, args);

                // ...
                auto xGroup = (u32)std::ceilf(m_renderWidth / 2.f / 8.f);
                auto yGroup = (u32)std::ceilf(m_renderHeight / 2.f / 8.f);
                rd->Cmd_Dispatch(cmdl, xGroup, yGroup, 1);
            }

        }
    });
```

Figure 3.2: Pass to render a half-resolution horizontal blur compute workload

3.3 Importing External Resources

For resources that have lifetimes longer than a single frame, they can be imported with a name and can be used just as any other resource. An entry state and desired exit state for the resource can also be declared on graph execution entry and exit respectively (see Figure 3.3). This allows for any external rendering code to integrate trivially with the graph. Additionally, the underlying resource of imported resources can be changed during runtime (see Figure 3.4).

```
m_rgResMan->ImportTexture(RG_RESOURCE(NoiseSSAO), m_ssaoNoise, D3D12_RESOURCE_STATE_COPY_DEST, D3D12_RESOURCE_STATE_COPY_DEST);
m_rgResMan->ImportBuffer(RG_RESOURCE(SamplesSSAO), m_ssaoSamples, D3D12_RESOURCE_STATE_COPY_DEST, D3D12_RESOURCE_STATE_COPY_DEST);
```

Figure 3.3: Importing texture

```
// Change backbuffer resource for this frame
m_rgResMan->ChangeImportedTexture(RG_RESOURCE(Backbuffer), m_sc->GetNextDrawSurface());
```

Figure 3.4: Changing resource of imported texture

3.4 Designing the Graph

3.4.1 Writing To A Single Resource

Assume two render passes A and B which are post-processing effects done on a single resource X with the intent of A happening before B. This would entail that both A

and B have to read from and write to X and generates a bidirected cycylic graph (see Figure 3.5), which does not satisfy the requirements for a task graph.

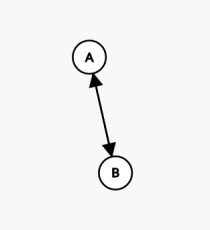

Figure 3.5: Bidirected cyclic graph

In order to solve this, the pass submission order and graph-level resource aliasing are used to make this unambiguous. The submission of pass B can use the information of currently submitted passes. Given that A has written to X (referred to as X0), the submission of B can alias the underlying graphics resource represented by X0, create a new graph resource X1, and write to X1, thereby resolving the graph. From the graph authors' perspective, the resource is always referred to as X, while it is internally being versioned.

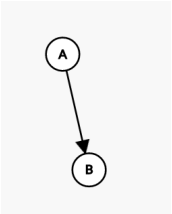

Figure 3.6: Directed acyclic graph

3.4.2 Simultaneous Read-Writes Due To Aliasing

Assume a case where A writes to X0, then B reads from X0, then C writes to X1 (aliased X0). A graph generated from such requirements is ill-defined on a topological graph traversal (see Figure 3.7) as traversing (A, B, C) and (A, C, B) would generate different results. The solution is two-fold. Firstly, such a case must be considered illegal (see Figure 3.7), which is trivial by evaluating all resources within the same depth. Secondly, the graph must be extended such that the intent is clearly translated into the graph traversal.

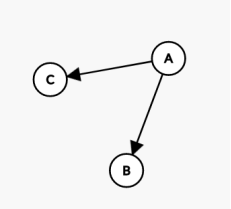

Figure 3.7: Possible ill-defined case

Graph Disambiguation

Assuming that the graph author submits the passes in the order of intent, a proxy dependency B $\rightarrow$ C can be inserted upon submission of pass C which disambiguates the graph and details the author's intent clearly (see Figure 3.9). X is consumed once aliased and can no longer be used.

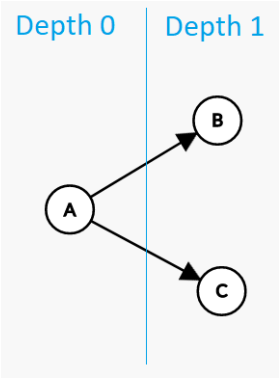

Figure 3.8: Figure 3.7 with depth

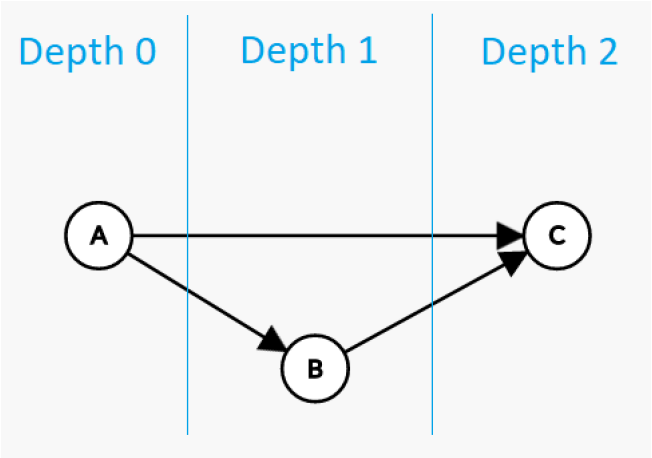

Figure 3.9: Disambiguated graph

3.5 Graph Resource Types

Internally, graph resources are explicitly identified as either Declared, Aliased, or Imported. Declared refers to resources that are graph-local and were declared as a part of a pass. Aliased refers to graph-level aliased resources. Imported refers to externally imported resources.

3.6 Render Graph Building

Unlike Frostbite's Frame Graph, which rebuilds the Frame Graph every frame, this render graph builds only on demand. The construction of the graph consists of a number of ordered steps, which will be covered in chronological order.

3.6.1 Building Adjacency Maps

In this phase, adjacency maps for each pass are created. For each recorded pass (A), all other passes (B) are compared by performing an intersection on the set of output resources of A with the set of input resources of B. If the intersection results in a non-empty set, a directed edge A $\rightarrow$ B can be formed.

3.6.2 Sorting Passes Topologically

Given the passes and their adjacency list, the directed acyclic graph can be topologically sorted using the Depth-First-Search (DFS) based sorting algorithm [6].

3.6.3 Building Dependency Levels

The graph is traversed in topological order to assign depth levels to each vertex. Using the depth, vertices are sorted into a per-depth bucket (dependency level). All vertices within a dependency level are independent of each other and can, in theory, be recorded in parallel.

3.6.4 Tracking Resource Lifetimes

Resource lifetimes are separated into a graph resource lifetime and a render resource lifetime represented by a pair of numbers [x, y], where they represent the depth at which the resource was first and last accessed respectively. This phase resolves lifetimes by going through all passes and all their input and output resources.

Graph Resource Lifetime (GRL) refers to the lifetime of a named graph resource, such as X0: [0, 4] which was created by a pass with depth 0 and was last accessed by a pass with depth 4.

Render Resource Lifetime (RRL) refers to the lifetime of the underlying resource that a graph resource represents. Its lifetime is the [min, max] of all graph resources which refer to it.

3.6.5 Realizing Resources

Declared graph resources are iterated over and their resource descriptions are used to create the underlying resource using D3D12 Memory Allocator [7].

Memory aliasing has been implemented for textures only, as they commonly have a larger memory footprint than buffers.

All declared textures are iterated over in memory size order, from largest to smallest, which can be referred to as A. For each other texture B, if the lifetimes of A and B do not overlap, add B to a list of aliasable resources for A, and mark B as handled. This results in a number of sets.

For each resulting set, enough memory is allocated to store the largest texture and creates placed resources within that set. Aliasing barriers are inserted into the correct dependency levels by iterating over the set in GRL order.

3.6.6 Resolving Barriers

For each dependency level, all passes are iterated over where the desired states of inputs and outputs are collected. Using the state information, both transition and UAV barriers can be added, the state tracked over the dependency level iteration, and redundant state transitions ignored.

Given that all resources within a dependency level are considered, the graph can enforce exclusive resource access for a pass when a write on the resource is detected or apply a combined read state.

UAV barriers are always inserted if two consecutive read-write access states were requested. The graph must ensure that any potential changes are visible for subsequent access.

3.7 Render Graph Execution

The graph is executed in every frame and consists of the following steps:

- Recreation of resource access views

- Graph Execution
- State Management

3.7.1 Recreation of Resource Access Views

Given the fact that the rendering resource of an imported resource can be changed at run-time, the graph recreates all views for all resources used. Recreation is only required for imported resources, but a catch-all solution was implemented given other priorities in the project and never revisited.

3.7.2 Graph Execution

The graph is traversed in dependency-level steps. For simplicity, only a single command list is used for the whole graph, to avoid complications with implicit state transitions [8] and because load balancing is omitted from the problem domain.

Before a dependency level is executed, all the barriers submitted for that dependency level are submitted (if any) at once. Afterward, the rendering code for each pass is executed with an automatic scoped render pass.

3.7.3 State Management

At the end of the graph execution, the state of imported resources is transitioned into their desired exit state, and declared resources have their state transitioned into their initial state prior to graph execution. During the graph execution, the state of resources is dynamically tracked.

3.8 Measuring Graph Execution Time

The measurement of the graph execution time is divided into three categories based on the graph dimensions (depth, breadth):

- Increasing depth (see Figure 3.10)
- Increasing breadth (see Figure 3.11)
- Fixed depth and breadth with varying graph forms (see Figure 3.12)

For each category, 4 times were taken, where the last 3 are the most expensive sub-calls which make up the total execution time:

- Total execution time
- Recreation of Resource Views
- Pass Executions
- State Management

For brevity, the final pass is omitted from the dimensions. Each pass generates and reads a single resource except the passes which have no incoming and outgoing edge respectively. The graph-authored rendering code is empty, therefore, the costs recorded are implementation-defined.

A deep vs wide graph comparison can give insight into the usage limitations of the graph. Extreme measurement by 4x increases in depth/breadth measurements was used to better highlight configurations in which the graph remains, and stops being usable.

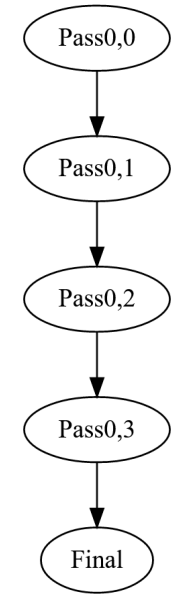

Figure 3.10: Graph with dimensions (4, 1)

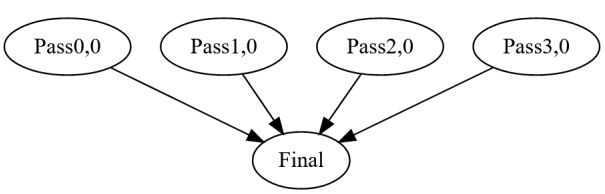

Figure 3.11: Graph with dimensions (1, 4)

3.9 Measuring Graph Memory Usage

Given that the gains of aliasing are heavily dependent on the context, only the results of the gains within the context of this project will be shown. As the render graph uses pools of memory, the memory which is actually used by the graph can be queried.

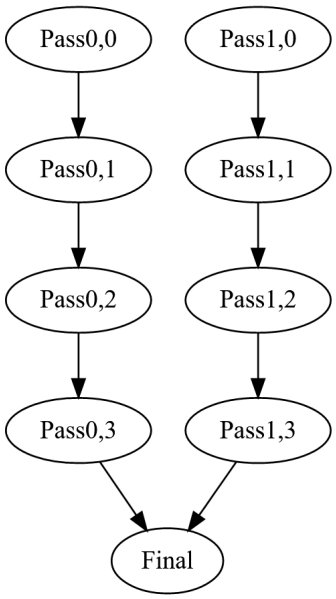

Figure 3.12: Graph with dimensions (4, 2)

Chapter 4

Results

Measurements for the results are taken on a Laptop with:

- AMD Ryzen 5900HX
- RTX 3070
- 16 GB RAM

Unnecessary processes (such as background processes, Discord, Steam, etc.) were disabled over the duration of the profiling.

The game is running with 1920x1080 in-game resolution using Release build, with a runtime budget of 16.66 ms assumed.

Measurements are taken using Mean-Time-Per-Call (MTPC) using TracyProfiler [9] with scoped profiling around the relevant code.

4.1 Graph Execution Overhead

Figures 4.2 - 4.4 show that an increase in depth or breadth makes little difference and that the graph is perfectly fit for usage with 64 passes. Using 256 passes is cutting it very close given the run-time budget, especially when the timing is overhead only.

As for how the timing increases with the number of passes, a linear increase would suggest a 4x increase in time for each data point. Roughly a 4-5x increase in time can be observed given higher depth/width, which indicates slightly worse than a linear relationship.

Additionally, the form of the graph (width-breadth variety) seems to have a slight, but minimal effect on the total execution time of the graph.

4.2 Graph Build Time

Figures 4.5 - 4.8 show that 256 passes in not feasible given a run-time budget of 16.66 ms. Using 64 passes is only viable if it does not occur every frame, where a rebuild can cause a hitch depending on available frame time. Moreover, the costs of adjacency map creation can be observed to overtake the costs of realizing resources from 256 passes and onward.

4.3 Graph Memory Usage

Given the render graph used for the game:

- Memory usage with memory aliasing disabled: **142.875 MB**
- Memory usage with memory aliasing enabled: **116.8125 MB**
- Total memory reduction with aliasing: **26.0625 MB** (18% reduction)

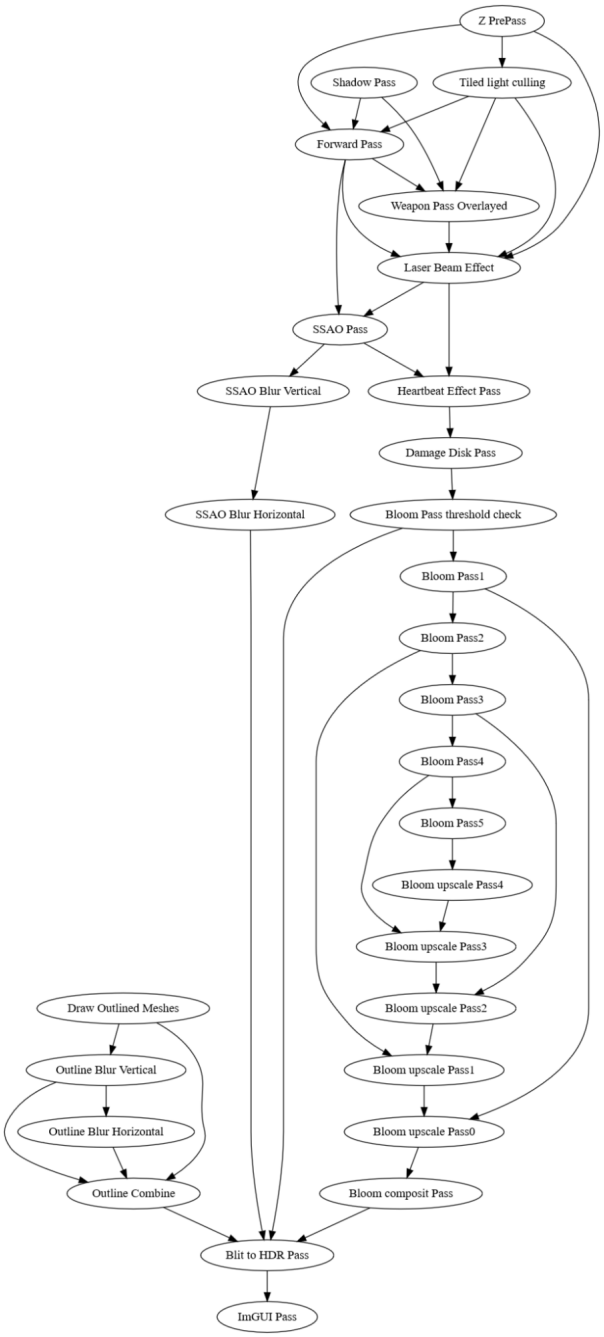

Figure 4.1: Render Graph used in-game (omitting particle work)

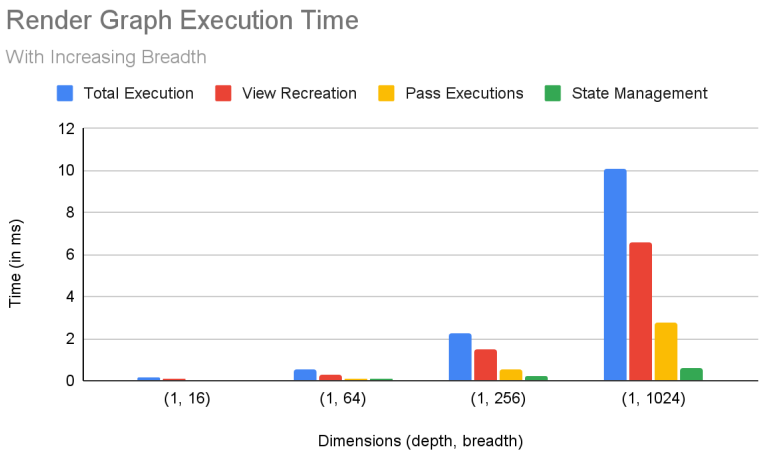

Figure 4.2: Execution timing 1

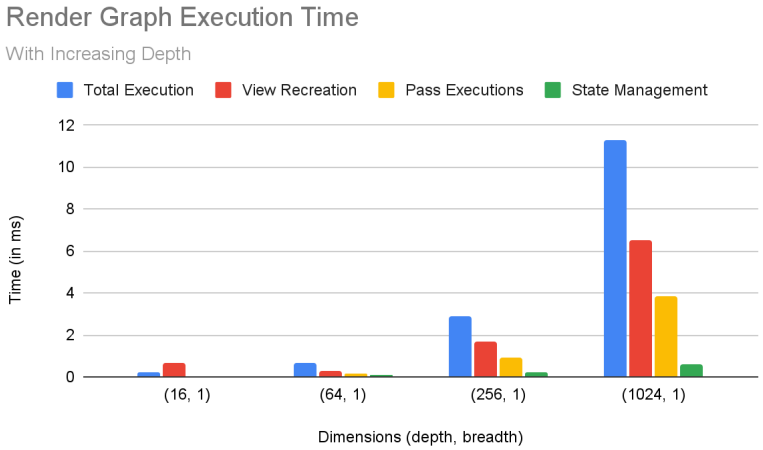

Figure 4.3: Execution timing 2

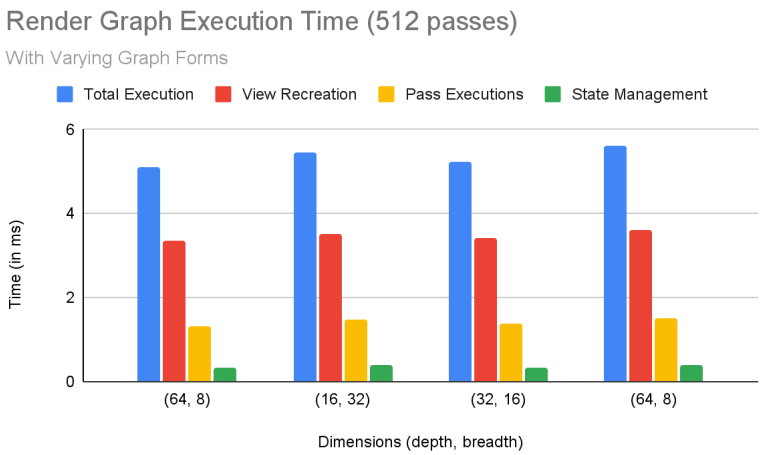

Figure 4.4: Execution timing 3

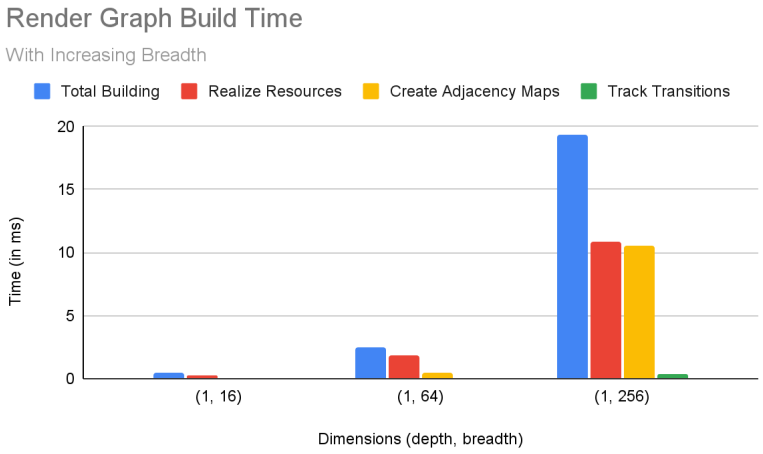

Figure 4.5: Build timing 1

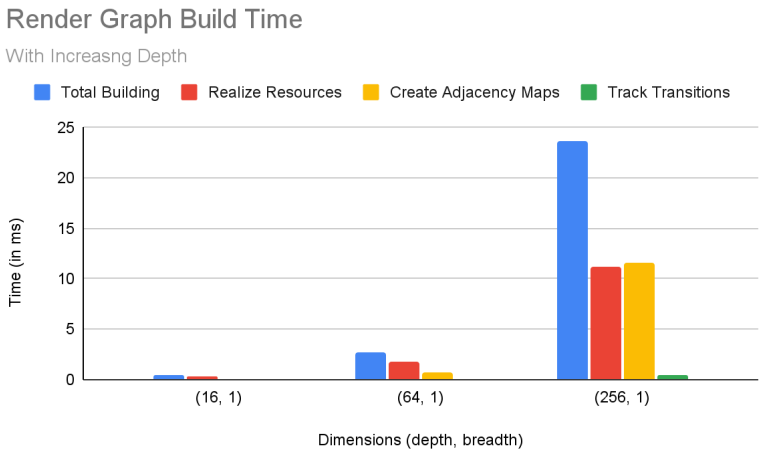

Figure 4.6: Build timing 2

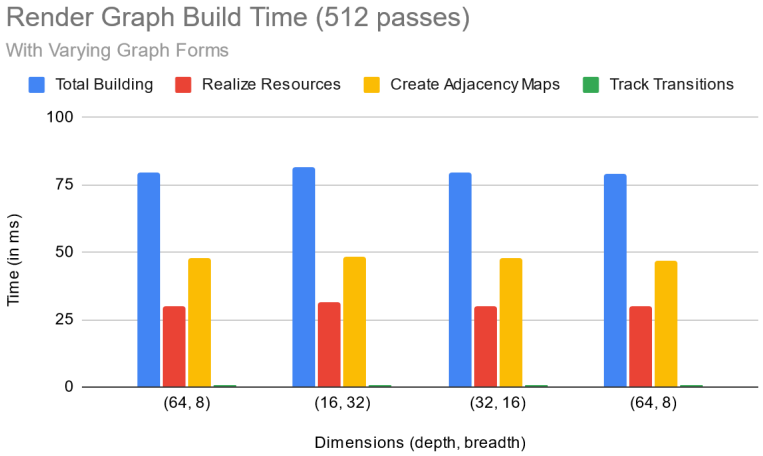

Figure 4.7: Build timing 3

Chapter 5

Other contributions

In addition to the Render Graph. I was responsible for implementing, maintaining, and assisting in the implementation of the majority of the core graphics functionality, abstractions, helpers, and data structures. My contribution includes, but is not limited to:

- Bindless rendering.
- Physically based shading.
- Normal mapping.
- Screen Space Ambient Occlusion.

Chapter 6

Discussion

Given that the test represents unrealistic graphs, the execution time results cannot be directly used for a realistic measure of usability. Realistic workloads have varying numbers of resources generated and consumed, with dependencies that are more complex than tested. As such, the results only give a glimpse of what could be expected given similar configurations.

Additionally, since the graph is not rebuilt every frame, less emphasis can be placed on the build times. In this implementation, rebuilds are intended for infrequent graphics setting changes. Should the graph be used for frequent in-the-moment changes, then the result weighs heavily on the viability of the graph.

The algorithm used for memory aliasing is naive given that only a single resource can be active given the memory allocated for a whole pack of resources. For example, if a 32 MB texture and a 2 MB texture are within the same aliasing group, 28 MB of that memory cannot be used. Memory is uniquely allocated for a pack and only used within that pack.

As a result, the implementation of the proposed render graph is only fit for use at 64 passes or below if a runtime budget of 16.66 ms is set, where even re-building should be kept to a minimum. Given the context of the project, this has worked without issues. Even for a hobby rendering project, this solution would be sufficient.

Chapter 7

Future Work

Additional work that can be done for this render graph can include the following, but are not limited to:

- Implementing a more sophisticated algorithm for memory aliasing for tighter memory reuse.
- Profiling a naive fork-join multi-threaded command list recording.
- Re-design barrier model with split barriers in mind.

Chapter 8

Conclusion and Reflections

In closing, implementing a render graph has been greatly beneficial for learning, practice, and for concrete results in the project. It has allowed better familiarity, more knowledge, and exploration with regard to a complex multifaceted problem in low-level rendering. Beginning from the first principle, a task graph, and modifying it to fit the rendering problem domain was challenging, but rewarding given that it allowed iteration, and finding dead-ends, which resulted in better understanding. Even if only a sliver of the whole problem was solved, it was rewarding and leaves a desire to learn more.

Most importantly, it has helped with the implementation and modification of various rendering techniques for all contributors by alleviating many tedious processes. Even if the resulting performance of the automatizing scales poorly, it has solved the issue well within the scale and context of the game project.

References

[1] stevewhims, *What is Direct3D 12 - Win32 apps*, en-us. [Online]. Available: https://learn.microsoft.com/en-us/windows/win32/direct3d12/what-is-directx-12- (visited on Jan. 8, 2023).

[2] stevewhims, *Memory Aliasing and Data Inheritance - Win32 apps*, en-us. [Online]. Available: https://learn.microsoft.com/en-us/windows/win32/direct3d12/memory-aliasing-and-data-inheritance (visited on Jan. 8, 2023).

[3] stevewhims, *Direct3D 12 render passes - Win32 apps*, en-us. [Online]. Available: https://learn.microsoft.com/en-us/windows/win32/direct3d12/direct3d-12-render-passes (visited on Jan. 8, 2023).

[4] *Task Graph | Our Pattern Language*. [Online]. Available: https://patterns.eecs.berkeley.edu/?page_id=609 (visited on Jan. 8, 2023).

[5] *GDC Vault - FrameGraph: Extensible Rendering Architecture in Frostbite*. [Online]. Available: https://www.gdcvault.com/play/1024612/FrameGraph-Extensible-Rendering-Architecture-in (visited on Jan. 8, 2023).

[6] *Topological sorting*, en, Page Version ID: 1123299686, Nov. 2022. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Topological_sorting&oldid=1123299686#Depth-first_search (visited on Jan. 8, 2023).

[7] *D3D12 Memory Allocator*, en-GB. [Online]. Available: https://gpuopen.com/d3d12-memory-allocator/ (visited on Jan. 8, 2023).

[8] stevewhims, *Using Resource Barriers to Synchronize Resource States in Direct3D 12 - Win32 apps*, en-us. [Online]. Available: https://learn.microsoft.com/en-us/windows/win32/direct3d12/using-resource-barriers-to-synchronize-resource-states-in-direct3d-12 (visited on Jan. 8, 2023).

[9] *Wolfpld/tracy: Frame profiler*. [Online]. Available: https://github.com/wolfpld/tracy (visited on Jan. 8, 2023).