Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Control Engineering and Information Technology

Balázs Kovács

APPLICATION OF RENDER GRAPHS FOR OPTIMIZING RENDERING PIPELINES

Scientific Students' Association Report

Advisor

András Máté Fridvalszky

BUDAPEST, 2023

Table of Contents

Abstract **4**

Összefoglaló **5**

1 Introduction **6**

1.1 Render Graphs 7

2 Existing Render Graph Implementations **8**

2.1 Unreal Engine 8

2.2 Frostbite: FrameGraph 10

2.3 Other Applications of Graphs in Rendering 12

2.3.1 Unity Engine: Shader Graph 12

2.3.2 DirectX 12: Work Graphs 12

3 Related Graph Theory **14**

3.1 Directed Acyclic Graphs 14

3.1.1 Topological Ordering 14

3.1.2 Topological Sort 15

3.1.3 Application of DAGs in Render Graphs 16

3.2 Chordal Graphs 16

3.2.1 Interval Graphs 17

3.3 Breadth-First Search 17

3.3.1 Algorithm pseudocode 17

4 Designing a Render Graph based Renderer **18**

4.1 Resources 18

4.2 Nodes 19

4.3 Setup Phase 19

4.3.1 Builder Interface 20

4.3.2 Editor Interface 20

4.4 Compilation Phase 22

5 Implemented Techniques **23**

5.1 Deferred Shading 23

5.2 Ambient Occlusion 24

5.3 Anti-Aliasing 25

5.3.1 Spatial Anti-Aliasing Techniques 26

5.3.2 Post-Processing Anti-Aliasing Techniques 26
5.4 Gaussian Blur.. 27
5.5 Ray Tracing... 28
6 Optimizing the Rendering Pipeline 30
6.1 Task Ordering 30
6.1.1 Sequential Execution of Tasks... 30
6.1.2 Parallelizing Execution Using Multiple Queues............................... 30
6.2 Resource aliasing 31
6.2.1 Algorithm for Resource Usage Range-based Memory Usage Optimization. 31
6.2.2 Optimality 32
6.2.3 Time complexity analysis 34
6.2.4 Results.. 34
6.2.5 Memory aliasing 36
6.3 Existing solutions... 37
7 Conclusion and Future Work 38
8 Bibliography 39

Abstract

With the rise of next generation low-level graphics and compute APIs such as DirectX 12 by Microsoft and Vulkan by the Khronos Group, developers became responsible for initializing the rendering context, memory allocations, resource lifetime management and much more. Subsequently, developing an easily extensible, highly efficient renderer that allows for creation of complex pipelines with ease has grown increasingly difficult.

In my work, I will demonstrate the use of render graphs, a high-level representation of render passes and resources to vastly simplify the creation of complex rendering pipelines. The render graph system provides developers with a visual way of orchestrating complex rendering pipelines while hiding away much of the low-level operations, such as explicit resource barriers, lifetime management and memory allocations. This also allows for various background optimizations, such as resource repurposing which can end up saving significant amounts of memory. The use of render graphs has become the standard in modern game engine development: the most common example being Unreal Engine's Render Dependency Graph, or Frostbite's FrameGraph developed by DICE.

I will also present a working render graph implementation that I developed using the Vulkan API.

Összefoglaló

A modern új generációs alacsony szintű grafikus és számítási API-ok megjelenésével (például a Microsoft által fejlesztett DirectX 12, vagy a Khronos Group által specifikált Vulkan) a fejlesztők felelőssé váltak a renderelési környezet inicializálásáért, memória-allokációkért, az erőforrások élettartamának kezeléséért és még sok másért. Ennek következtében jelentősen nehezebbé vált egy könnyen bővíthető, hatékony grafikus motor fejlesztése, amely lehetővé teszi a bonyolult renderelési csővezetékek egyszerű létrehozását.

A dolgozatomban bemutatom a Render Graph-ok használatát, ami egy magas szintű reprezentációja a render pass-oknak és erőforrásoknak. Segítségével jelentősen megkönnyíthető a bonyolult renderelési csővezetékek létrehozása. A Render Graph rendszer egy vizuális eszközt kínál a fejlesztők számára, mellyel könnyedén le lehet írni a renderelési folyamatot. A rendszer elrejti a fejlesztők elől az alacsony szintű műveletek nagy részét, mint az erőforrások, és azok élettartamának kezelését, valamint a memória foglalásokat. Ez továbbá lehetőséget nyújt különböző optimalizációkra, például az erőforrások újrahasznosítására, amivel akár jelentős mennyiségű memóriát takaríthatunk meg. A Render Graph-ok használata elterjedt a modern játékmotor fejlesztésben, erre a leggyakrabban emlegetett példa az Epic Games által fejlesztett Unreal Engine, és annak a Render Dependency Graph modulja, vagy a DICE Frostbite motorjának FrameGraph-ja.

1 Introduction

Ever since the introduction of low-level graphics application programming interfaces (APIs) like DirectX 12 (2015), Vulkan (2016) and Metal, they have only become an increasingly popular choice for rendering engines. These APIs target high-performance real-time 3D rendering applications (interactive media) and highly parallelized computing. For example, today we see that a lot of videogames receive DirectX 12 (and less commonly Vulkan) support through engine upgrades. Compared to older APIs such as DirectX 11 and OpenGL, they more accurately resemble how current generation GPUs work and offer higher performance and more efficient CPU and GPU usage.

Traditionally GPUs executed a single workload in parallel on multiple GPU cores, where multiple workloads were run after each other. That was until we saw the most recent improvements in GPU architecture, allowing for hardware parallelization of multiple workloads. Making use of hardware parallelization using the new, low-level API, developers can achieve the benefits mentioned earlier by submitting workloads to different “queue families”.

However, these benefits come at the cost of developers now being responsible for setting up the hardware interface and rendering context themselves in an explicit and verbose manner. It takes a lot of code to get a simple example running while using these APIs, and it only gets more difficult if one would like to do something even remotely complicated. For this reason, if one would like to use such an API reasonably for implementing a rendering engine, it would be a great idea to first build an abstraction layer over the API or develop a comprehensive rendering hardware interface (RHI). This is easier said than done, as creating one that can utilize these APIs to their full extent requires careful planning and designing, especially if our requirements include supporting multiple APIs.

1.1 Render Graphs

There are numerous ways of creating abstraction layers for these low-level APIs, one of them is the idea of Render Graphs. Render Graphs are a high-level description of graphics operations and resources used to render a scene. The system also has full control over the lifetime and usage of most render resources. This brings a well-defined structure for the rendering code, but also takes care of error-prone, low-level operations in the background such as automatic resource transitions and memory allocations.

As of 2023 the use of Render Graphs has become a standard in modern game engine development: The most common example is the render dependency graph (RDG) from Unreal Engine. From various Game Developers Conference (GDC) presentations we know that there are other existing implementations used in engines like Frostbite by DICE [1] and Anvil by Ubisoft [2].

2 Existing Render Graph Implementations

Using a Render Graph system is a popular choice among modern renderers. In the following section I will present various popular engines that implement their own version of Render Graphs and other similar graph-based techniques.

2.1 Unreal Engine

Unreal Engine (UE) is a state-of-the-art game engine developed by Epic Games, created by Tim Sweeney, and first released in 1998. Despite its origins as a game engine, Unreal Engine has become a powerful real-time 3D creation tool used in a variety of other fields such as movie production. The latest generation, Unreal Engine 5 is widely considered to be one of the most technologically advanced game engines today. Its major features include Nanite, Lumen and Virtual Shadow Maps. The engine's source code is publicly available and can be viewed on GitHub under the condition that one joins the Epic Games developers organization.

Nanite is a virtualized geometry system that allows for an increase in geometry complexity by multiple orders of magnitude while handling the level of detail (LOD) automatically. Lumen is a fully dynamic global illumination solution that eliminates the need for lightmaps while allowing for real-time behavior of light sources.

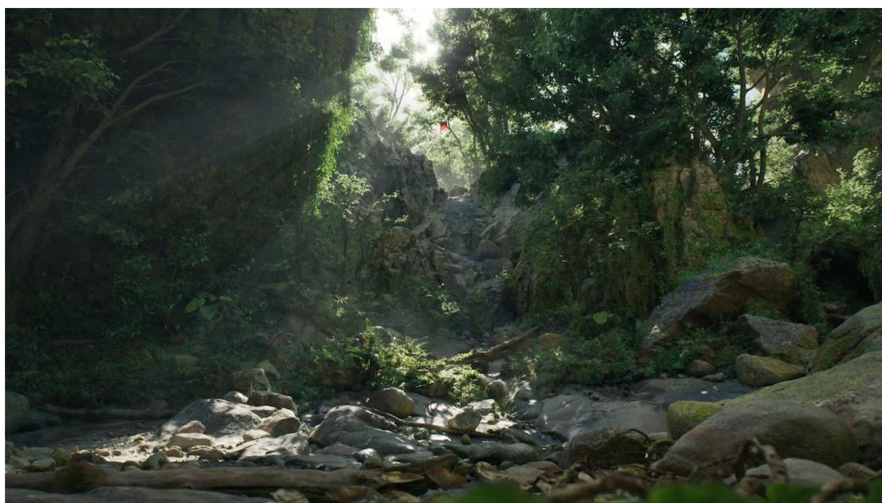

Figure 2.1: Screenshot taken from the Unreal Engine 5.2 real-time showcase.

All these new rendering techniques are enabled by the underlying Render Dependency Graph (RDG) API [3], which they also refer to as “Render Graph”. The

RDG includes all the expected features of a render graph system: synchronizing asynchronous commands, allocation of transient resources with optimal lifetimes and memory aliasing, automatic resource transitioning, culling unused nodes and resources, API usage and resource dependency validation and visualization of the graph structure.

The high-level rendering code of Unreal Engine should be written using RDG. Developers can build their own, custom rendering pipelines in code using the Render Graph Builder API. While it is recommended that for common passes developers use the collection of utility functions provided by UE, they also have the ability to define custom passes.

The RDG splits the rendering pipeline into two phases: the setup and execute timelines. We as developers can set up the graph during the setup timeline by creating resources and adding passes using a Builder instance. Then, after calling execute, the graph is compiled and executed. All render hardware interface (RHI) commands are deferred into pass lambdas, which are called on the execution timeline.

The allocation of underlying RHI resources is delayed until execution. Resources, buffers, and textures specifically can be either transient or external whether its lifetime is constrained to the graph. Transient resources can potentially alias with other resources with disjointed lifetimes. External resources lifetimes extend outside of the graph. These can be existing RHI resources, or resources extracted from the graph after execution.

The RDG insights plugin can be used as a debugging and diagnostic tool and for visualization of the RDG structure. This visualization can be used to inspect various properties of the graph such as: resource lifetimes, asynchronous compute fences and overlaps.

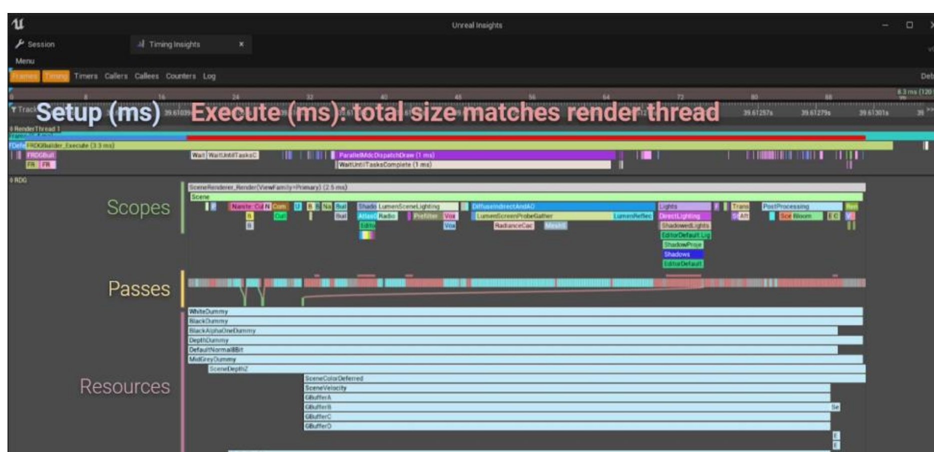

Figure 2.2: RDG Insights plugin interface. (Source: [3])

2.2 Frostbite: FrameGraph

Frostbite is the in-house game engine of DICE, a subsidiary of Electronic Arts (EA). First developed for first-person shooter games, its use since then has been expanded to a variety of genres. The renderer of Frostbite produces photorealistic images comparable to that of Unreal Engine. Due to Frostbite being a proprietary engine there is not much publicly available information about its design and architecture, except from a few presentations from the Game Developers Conference.

From a presentation given by Yuriy O'Donnell at GDC, a rendering Engineer on the frostbite team, we know that the engine uses a similar system to render graphs called FrameGraph [1]. Overtime, the functionality and complexity of the rendering system has scaled up massively. Rendering systems became tightly coupled and as a result suffered from limited extensibility. With the renderer code growing from 4k to 16k source lines of code (SLOC), requiring explicit resource management with implementations differing across teams, the code became expensive to maintain and extend.

The answer to these problems was designing a new, modular and extensible renderer architecture. This led to the creation of FrameGraph and Transient Resource System. Like render graphs, the FrameGraph is a high-level description of render passes and resources with full knowledge of the frame. The FrameGraph is supported by the Transient Resource System, which is responsible for resource allocation and memory aliasing.

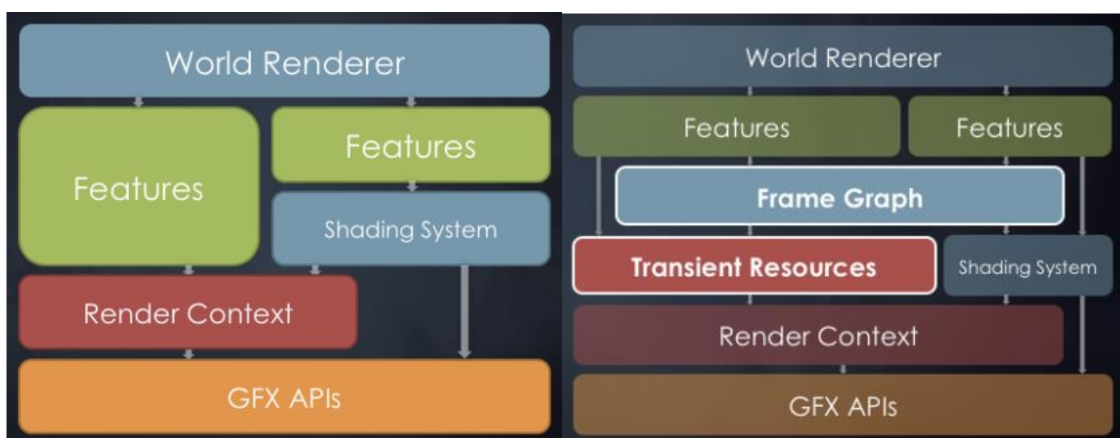

Figure 2.3: Comparing the old and new renderer architecture of Frostbite. (Source: [1])

The FrameGraph's ambitions align with what we would expect from a render graph system. Moving away from immediate mode rendering, the rendering code is now

organized as efficient self-contained modules called passes. Among many things error-prone operations such as resource management, async compute and memory barriers were simplified. Previously these had to be explicitly managed by the developers. FrameGraph implements a multi-phase retained mode rendering API (setup, compile, execute) with the graph being rebuilt from scratch every frame to support dynamically changing rendering configurations.

Just like other render graph systems, during the setup phase developers can define passes with input and output resources. These are all virtual at this stage. Like in Unreal Engine's RDG, resources can be either transient or external depending on whether its lifetime is constrained to the graph or not. External resources include the backbuffer, history buffer for temporal anti-aliasing and more. Resource parameters are derived based on its uses and creation is delayed until first use. Passes are declared using C++ lambda functions, because creating per pass C++ classes is inconvenient, as they require a lot of boilerplate code and break code flow.

The compilation phase is as expected: unused resources and passes are culled, then resource lifetimes are calculated, and finally concrete GPU resources are allocated based on usage. Finally, during the execution phase the passes are iterated over and their callback functions are called. The system also provides tools for visualization and debugging.

We can conclude that by using a render graph-like architecture, the Frostbite team managed to turn the old and hard to maintain renderer into one that is easily extensible and is more fit for complex rendering pipelines used today.

2.3 Other Applications of Graphs in Rendering

The use of graphs in computer graphics is not limited to the rendering pipeline. The new Work Graphs specification [16] provides a way of organizing GPU work at a low-level using graphs and various tools exist that use a node-based editor to allow users to for example create shaders.

2.3.1 Unity Engine: Shader Graph

Unity is a widely popular cross-platform game engine. In Unity we see another application of graphs in rendering, shader graphs. The Shader Graph is a node-based system that lets developers visually author shaders and see results real-time.

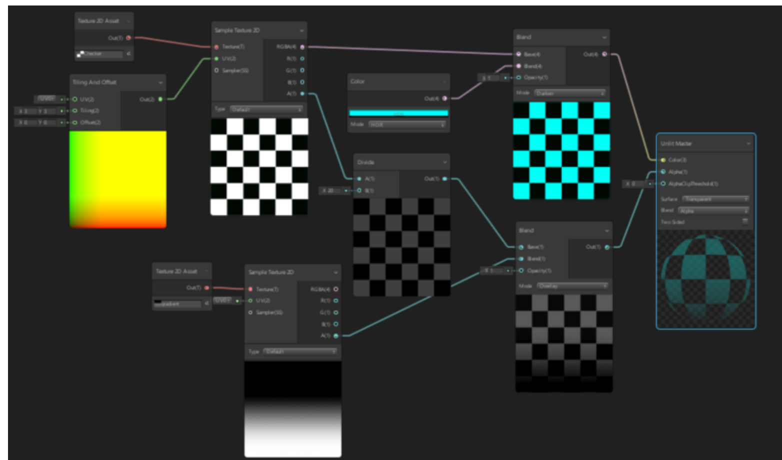

Figure 2.4: Interface of the Unity Shader Graph

Shader Graph provides a large variety of predefined nodes for developers to work with. It is also possible to create “Custom Function” nodes with custom inputs and outputs and functionality defined by HLSL (High-Level Shader Language) code written by the developer.

Compared to render graphs, this system works at a higher level, by using it we cannot modify the rendering pipeline itself but create materials.

2.3.2 DirectX 12: Work Graphs

Work graphs are a new system for GPU based work creation in DirectX 12. The preview for this feature was released recently in June of 2023 with the specification available for viewing on GitHub. The motivation for work graphs was to address some limitations in the programming model of GPUs and by doing so enabling the development

of future technologies. For example, the new rendering features of Unreal Engine 5 such as Lumen and Nanite are hitting the limits of the current compute shader paradigm.

Traditionally compute workloads are dispatched to the GPU, then based on its result with a round trip back to the CPU we determine what subsequent work the GPU needs to do. Work graphs eliminate the need for this round trip by allowing running shader threads to dispatch new workloads on-demand. The system then can schedule the requested work as soon as the GPU has the capacity to execute it. Therefore, the system is asynchronous in nature. Work graphs are a graph of nodes where running shader threads can request invocations of other nodes, without waiting for them to launch. The graph is acyclic, with the exception that a node can output to itself.

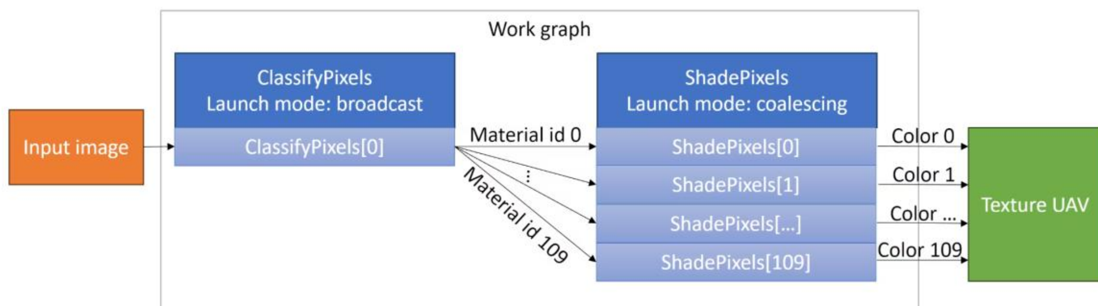

Figure 2.5: Example Work graph. (Source: [4])

This model is comparable to that of hardware-accelerated ray tracing. Ray tracing shaders already have the ability to call other shaders. Starting from a ray generation shader intersection a variety of shader types; any hit, closest hit, miss, and callable shaders are called. In concept, callable shaders are the most similar to work graphs. Callable shaders can be explicitly invoked from another shader in code via `CallShader()` in HLSL and `executeCallableEXT()` in GLSL.

Developers can experiment with work graphs using preview drivers. As of now only AMD provides publicly available drivers with support for work graphs. In contrast to Render Graphs, work graphs aim to solve a problem of GPU workload scheduling at a vastly lower level.

3 Related Graph Theory

Graphs are mathematical structures used to model relations between objects. In the context of graph theory graphs are an ordered pairing defined as $G = (V, E)$, where V is the set of vertices (also called nodes, points) and E is the set of edges that connect vertices together. A distinction is made between undirected and directed graphs. In an undirected graph, edges connect two vertices symmetrically, while in a directed graph edges connect two vertices asymmetrically.

In computer science it is not uncommon to reach for graph theory to help us solve various issues. Graphs have many applications including representing network topology, state machines and dependencies between several objects.

Dependency graphs are directed graphs that represent dependencies between tasks (a vertex of the graph) using directed edges. For some directed graphs it is possible to find a linear ordering of its vertices such that it respects the dependencies defined by the graph's edges. The graphs that fulfill this property are known as directed acyclic graphs.

The theories, definitions and algorithms mentioned in this section can be found in more detail in "*Introduction to Algorithms*". [6]

3.1 Directed Acyclic Graphs

A directed acyclic graph (DAG) is a directed graph with no directed cycles. Such graphs consist of vertices and edges, with each edge directed from one vertex to another, such that these directed edges never form a closed loop. We can verify if directed graphs are acyclic or not using the following theorem: "The directed graph $G = (V, E)$ is acyclic if and only if it can be topologically ordered." This property of directed acyclic graphs is particularly useful as it allows us to verify if a given directed graph is acyclic or not.

3.1.1 Topological Ordering

The topological ordering of a directed graph G can be defined as follows:

Let $(v_1, v_2, \dots, v_n)$ be an ordering of the vertices of a directed graph G. This list of vertices is a topological ordering of G if for every edge (x, y) the vertex x occurs earlier than y in the list.

In the context of dependency graphs directed acyclic graphs play a crucial role:

- The acyclic property of them ensures that there are no circular dependencies or infinite loops, making it possible to find a valid execution order of nodes.
- They allow for efficient algorithms to find topological ordering.
- Can be used to parallelize tasks, as nodes that do not depend on one another.
- Provides an intuitive way of representing complex dependencies.

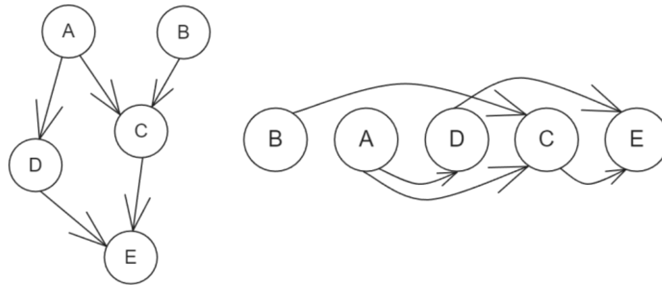

Figure 3.1: DAG and its Topological Ordering.

3.1.2 Topological Sort

The topological ordering of a directed graph can be derived by various algorithms in linear time $O(|V| + |E|)$ where V is the set of vertices and E is the set of edges. The two most common methods are Kahn's Algorithm [5] and depth-first Search (DFS).

Kahn's Algorithm (first described by Arthur B. Kahn in 1962) can also be used to find if it is acyclic or not. This algorithm works by strategically removing the edges of the graph.

3.1.2.1 Pseudocode for Kahn's Algorithm

```
L <- Empty list, this will contain the sorted nodes
S <- Set (or stack or queue) of nodes which have no incoming edges
while S is not empty do:
     remove a node v from S
     add v to L
     for each node w with an edge e from v to w do:
          remove edge e from the graph
          if w has no other incoming edges then:
               insert m into S
if graph has edges then:
     return error (the graph has at least one cycle)
else
     return L
```

3.1.3 Application of DAGs in Render Graphs

Render Graphs are essentially dependency graphs in which nodes that represent various functions are dependent on one another's resources. Therefore, we can use directed acyclic graphs to represent resource and data dependency through the rendering pipeline described by our Render Graph.

Using the properties of DAGs, we can perform various optimizations and find a linear execution order of the Render Graph by running topological sort on it.

One such optimization is the parallelization of various tasks that can be executed simultaneously. A commonly seen use case for this is asynchronous compute; the utilization of unused GPU resources (compute units, bandwidth, registers) by running compute dispatch calls asynchronously while rendering.

3.2 Chordal Graphs

Chordal graphs are a subset of perfect graphs in which all cycles of four or more vertices have a chord. A chord is defined as an edge that is not part of a cycle but connects two vertices of said cycle, meaning that every induced cycle should have exactly three vertices (triangulated graphs). A graph is chordal if and only if it has a perfect elimination ordering. A perfect elimination ordering of a chordal graph can be found in linear time using the lexicographic breadth-first search algorithm [7], the verification of the found ordering can also be done in linear time. Therefore, it is possible to recognize a chordal graph in linear time. As a result, several problems such as graph coloring can be solved in polynomial time for chordal graphs.

Perfectly orderable graphs are graphs whose vertices can be ordered in such a way that a greedy coloring algorithm with that ordering optimally colors every induced subgraph of the given graph.

An application of perfect elimination orderings is finding a maximum clique of a chordal graph in polynomial-time. As chordal graphs are perfect graphs, the size of the maximum clique equals the chromatic number of the chordal graph and since they are also perfectly orderable an optimal coloring can be obtained by applying a greedy algorithm to its vertices in the reverse of its perfect elimination ordering.

3.2.1 Interval Graphs

Interval graphs are a special subset of chordal graphs. An interval graph is an undirected graph formed from a set of intervals, with a vertex for each interval and an edge between vertices whose interval intersect. Therefore, the edge set of I interval graph is defined as $E(I) = \{(v_i, v_j) \mid S_i \cap S_j \neq 0\}$

As interval graphs are chordal graphs, they are also perfect graphs. They can be recognized in linear time, and an optimal graph coloring can be found in linear time.

3.3 Breadth-First Search

One of the most known and used algorithms that operates on graphs is the breadth-first search (BFS). The algorithm traverses the given graph such that it first explores all nodes at the current depth level before moving on to the nodes of the next depth level and stops when a node that satisfies a given property is found. Breadth-first search can also be used to determine which nodes can be reached from the root node by running the algorithm so that it does not search for a node but returns the set of visited nodes.

3.3.1 Algorithm pseudocode

```
G <- Graph
R <- Root node, a vertex of graph G
Q <- Let Q be a queue
add R to Q
mark R as visited
while Q is not empty:
     v <- dequeue vertex from Q
     for all neighbors w of v in graph G
          if w is not visited
               add w to Q
               mark w as visited
```

Later, this algorithm will be useful during the Render Graph compilation phase to cull nodes that are not connected to the input graph.

4 Designing a Render Graph based Renderer

It is for a reason that the rendering process is commonly referred to as a pipeline. The rendering process usually is organized as a sequence of multiple stages of rendering work where each stage consumes and produces GPU resources. Passes depend on one another based on the resources they use. Graphs are natural candidates to model these relations between stages. Using the relations, we can build a directed acyclic graph that describes the rendering pipeline. The Render Graph is exactly this, a dependency graph where edges represent resource usage and are drawn between two vertices: one that produces the resource and one that consumes the resource.

This gives us the opportunity to create a system that has high-level knowledge of the rendering pipeline that renders a frame and determines an optimal way to execute it ahead of time. A render graph system is expected to automatically take care of low-level operations (memory barriers, state transitions) and optimize the rendering pipeline (memory usage, execution order). This system can also be used to cull redundant stages and validate the correctness of the described pipeline.

4.1 Resources

Resources primarily represent GPU resources such as images, buffers and acceleration structures. However, it is also possible to extend these with “special” types of resources including but not limited to camera data and scene objects.

The Render Graph system is the owner of GPU resources and is responsible for managing their lifetime. Resources store the underlying GPU resource and are produced, consumed, and stored in heterogeneous containers by nodes.

Taking these requirements into consideration, we can define a bare minimum Resource base class. The specific types of resources then can be implemented by inheriting from this class, for example: ImageResource that stores an Image as its underlying resource.

In my implementation I included image, top-level acceleration structure, camera data and scene object data resources. The last two are external resources to the graph, which are owned by a scene, and exist for the sole purpose of making scene data available for nodes to consume.

4.2 Nodes

In the context of Render Graphs, a node represents a self-contained unit of work, a render pass. Nodes consume and produce resources which are represented by input and output GPU resources and stored in a heterogeneous container. Nodes also need to be able to represent various GPU workloads: graphics, compute and if applicable hardware accelerated ray tracing and expose various configurable parameters of these workloads. These are the properties all nodes will share, therefore when working with inheritance we can define the bare minimum Node base class as following:

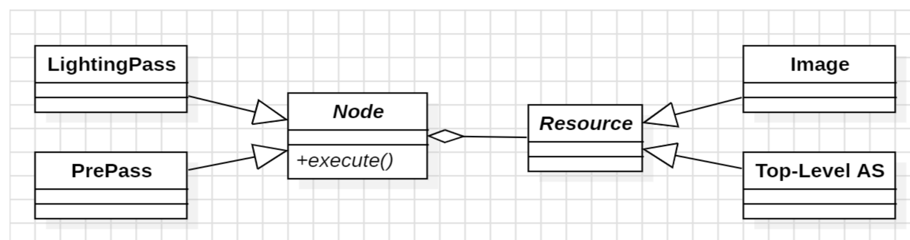

Figure 4.1: simplified class diagram of nodes, resources, and their specifications

I opted to use inheritance as it is a straightforward way of allowing for developers to create new nodes with complete control over their inner implementations. My Render Graph implementation provides eight different nodes with various functionality and support for deferred shading and ray tracing.

I included a special kind of node; the SceneProvider, whose sole job is while acting as the root node of the graph to make the resources (Objects, Cameras and when applicable Top-Level Acceleration Structure) of a scene available for nodes later down the pipeline. While this node could be removed and implicit connections could be used instead, I chose to include it so the developer can see which nodes consume these resources in an explicit manner.

The other seven nodes are all GPU workloads including a few frequently used techniques in computer graphics: Ambient Occlusion, Anti-Aliasing, Gaussian Blur, Lighting Pass, G-Buffer Pass, Ray Tracing and Present.

4.3 Setup Phase

When using a Render Graph based system there are two phases; a setup phase where the user can set up and modify the rendering pipeline, then comes the compilation

phase which is a completely autonomous procedure with no user input. The result of the compilation process is a render path, the resources, nodes and their execution order.

The user can create and set up a Render Graph using two different interfaces: in code using the Graph Builder API or using the visual graph editor.

4.3.1 Builder Interface

The Graph Builder API can be used to construct Render Graphs by code written in C++. Using the builder interface, it is possible to programmatically create Render Graphs according to our needs. This is particularly useful when we need to dynamically change the rendering pipeline (e.g.: cutscenes) or we are creating our rendering pipeline based on user configurations like in video games.

The builder API allows the developer to create nodes, configure the parameters of them and define resources. Then once ready, the graph can be compiled and used as the render path in our application.

```
auto builder = RenderGraph::Builder();
auto pass_a  = builder.create_pass(PassType::eSceneProvider);
auto pass_b  = builder.create_pass(PassType::eGBuffer);
builder.connect(pass_a, pass_b, "Camera");
builder.compile(CompilerMode::eOptimized);
```

Figure 4.2: Using the Builder Interface.

4.3.2 Editor Interface

The visual Render Graph editor is particularly useful when debugging and experimenting. The developer can quickly create and modify complex rendering pipelines using the node-based graph editor that when compiled immediately replaces the old render path with the new one. The visual editor is built using ImGui [8] and an extension of it; imnodes [9] that provides the node editor. Editor features include but are not limited to adding nodes, selecting the active scene, compiling the graph, and resetting the graph to the initial state.

Each node in the graph declares what resources it consumes on the left side and produces on the right side by name. The user can connect resources between nodes using the points next to the resources name. Both the points and connecting lines are color coded by resource type.

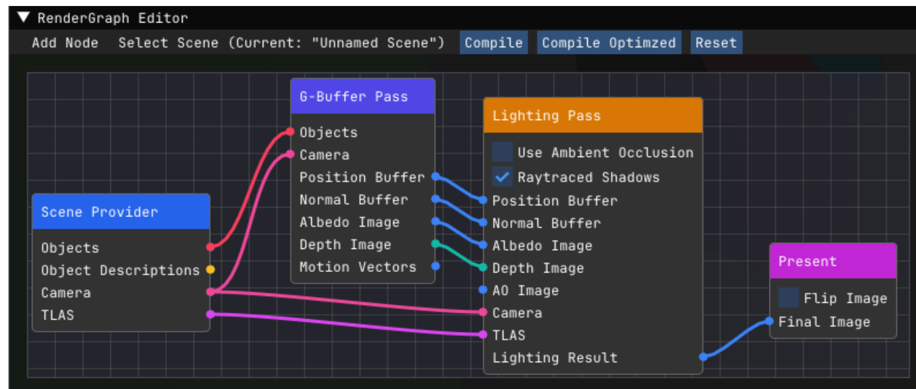

Figure 4.3: Node-based editor interface.

As seen on figure 4.3, it is also possible to expose various configuration options for a node, for example the user can enable raytraced shadows, or explicitly disable the ambient occlusion input for the lighting pass node.

If the Render Graph is successfully compiled, the rendering pipeline defined by it will replace the old one and will be used to render the selected scene. This provides the user the ability to quickly test various pipelines and verify if a resource produced by a node is as expected or not with ease without the need for debugging tools such as NVIDIA Nsight Graphics or Radeon GPU Profiler.

For example, we can view the image produced by the ambient occlusion node directly, while only using the bare minimum required nodes and connecting the image in question to the Present node.

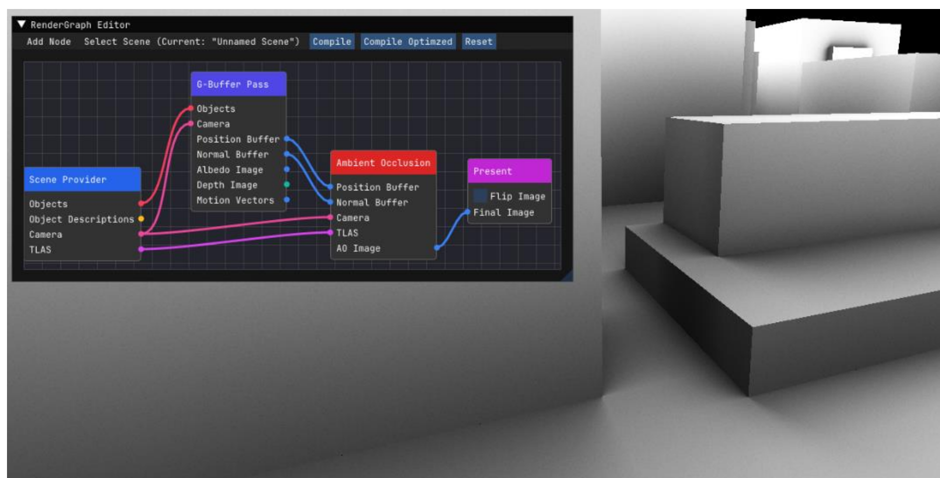

Figure 4.4: Presenting the Ambient Occlusion result.

The editor also logs compiler messages and most actions to the standard output. It also notifies the user about errors that occurred while editing or compiling the graph.

4.4 Compilation Phase

The compilation phase is a completely autonomous process in which the user has no input and where the input Render Graph is processed into an executable render path. All optimizations such as resource or memory reusing are done during this phase. A render path is the collection of nodes with their execution order and resources required to execute the rendering pipeline defined by the render graph.

The compilation phase consists of five key steps:

1. Culling unreachable nodes using BFS graph traversal started from the SceneProvider node.
2. Using topological sort to verify that the input graph is a DAG and therefore has no circular dependencies, while determining an execution order of nodes.
3. Evaluate and create the resources required by the remaining nodes.
4. Connecting the resources to nodes.
5. Compiling the execution order, nodes, and resources into a render path.

One possible optimization we can make during the compilation phase is reducing the number of resources: instead of naïvely creating all resources, we could try optimizing the required number of resources for example by resource or memory aliasing.

During compilation, the given input Render Graph is validated for correctness, checks for circular and missing required dependencies are made. It is important to note that if for whatever reason the compilation fails feedback about the error is given to the user through log files or the standard output. When using the editor interface this is particularly useful as failed compilation will not cause unexpected crashes. Regardless of the compilation result, when compiling with verbose mode on log files are produced containing log messages and a data dump of the nodes and resources of the render graph.

In my implementation making use of the strategy design pattern, I created two graph compilers: one that naïvely creates all required resources, and one that first optimizes the number of required resources before creating them. Other developers can also create their own compilation strategy by implementing the proper interface.

5 Implemented Techniques

To demonstrate the render graph system, I created various nodes that implement commonly seen techniques in computer graphics.

5.1 Deferred Shading

Deferred shading is a screen-space shading technique that aims to overcome the performance drawbacks of forward shading. With standard forward shading we render each object and light it according to all light sources in the scene. This is a performance heavy and wasteful operation as for each rendered object we iterate over all light sources for each fragment and with overlapping objects a lot of fragments get overwritten.

The primary benefit of using deferred shading is the decoupling of scene geometry and lighting making it possible to render scenes with hundreds of lights at an acceptable frame rate. Deferred shading splits the rendering process into two passes. During the first pass we gather the data required for computation such as positions, normals (rendered into texture buffers, commonly referred to as G-buffer). The second pass; the lighting pass will use the data contained within the G-buffer to compute the lighting at each pixel in screen space. The data generated during the first pass can be used by various other techniques such as ambient occlusion.

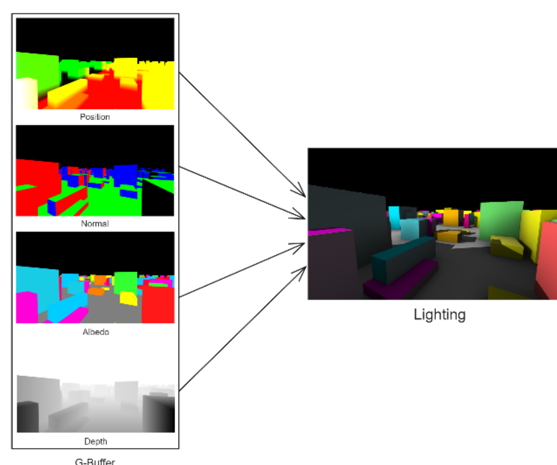

Figure 5.1: Multiple target rendering (MRT) to the G-Buffer and Lighting result.

Using deferred shading has a few serious consequences we might need to consider for our application: we lose the ability to handle transparency and hardware anti-aliasing will not produce correct results anymore since anti-aliasing the data of the G-buffer would

result in nonsensical values. This is one of the reasons why post-processing anti-aliasing is preferred over hardware anti-aliasing.

In my implementation the first pass aside from position, normal and color data also produces motion vectors that can be used later in the pipeline for motion blurring or temporal anti-aliasing.

5.2 Ambient Occlusion

Ambient occlusion (AO), in 3D computer graphics is a commonly used global technique to approximate full global illumination. It simulates the soft shadows that occur when surfaces are close to each other by approximating how much ambient light can hit a point on a surface. For this reason, AO is used to add more realism to the rendered image by visually highlighting the separation of objects. There are various methods for real-time ambient occlusion simulation, with ray traced ambient occlusion joining the list of viable methods since the release of real-time ray tracing capable hardware (NVIDIA RTX 20 series GPUs, 2018).

Ambient occlusion is calculated by constructing a hemisphere of rays originating from a point in all directions then checking for ray intersections. Mathematically, we can define the ambient occlusion at a point p as the integral of the visibility function of a point p and its normal n over the normal-oriented hemisphere with respect to the projected solid angle ω.

$$AO(p,n) = \frac{1}{\pi}\int_{\Omega} V_{p,\omega}(n \cdot \omega)d\omega$$

To most straight-forward way of approximating this integral in practice is using the Monte Carlo method: we take samples by casting rays in random directions within the hemisphere from point p and testing for intersection with other geometry. This method also requires some form of denoising as we can only take a limited number of samples before significantly affecting performance.

The two traditional methods used by game engines are screen space ambient occlusion (SSAO) [10] and horizon-based ambient occlusion (HBAO) [11]. Both techniques were developed by NVIDIA. SSAO works well enough, with the obvious limitation of it being that it is a screen space technique. This means that geometry not visible to the camera will not affect ambient occlusion.

A recently popular method is ray traced ambient occlusion (RTAO). Its main benefit is that it produces higher quality, physically correct results compared to existing techniques. The drawbacks of using RTAO in real-time applications is its considerable impact to performance and that denoising becomes mandatory, otherwise the resulting image even at a high sample count (64) would be significantly noisy. This method also requires a GPU capable of hardware accelerated ray tracing. Various game engines support this technique; examples include Unreal Engine and the Luminous engine developed by Luminous Productions.

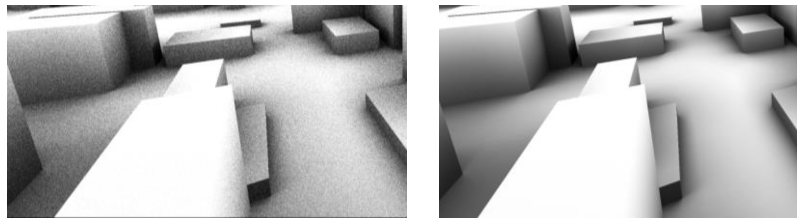

Figure 5.2: RTAO with sixteen samples in a single frame vs. samples accumulated over time.

In my implementation I chose ray traced ambient occlusion using ray queries. The ambient occlusion parameters: radius, power and sample count are configurable, with support for over time sample accumulation when the camera is stationary. However, the node could be expanded with other techniques by creating different ambient occlusion strategies which the node can use.

At 2560x1440 (2k) resolution, my implementation of ray traced ambient occlusion takes sixteen milliseconds to calculate, which is a non-negligible impact on performance. For this reason, it is common to use a lower resolution for ambient occlusion than the rendering resolution.

5.3 Anti-Aliasing

In computer graphics anti-aliasing algorithms are used to remove the aliasing effect from rendered images. Aliasing in 2D images commonly manifests as pixelated edges and the moiré pattern and is a result of undersampling. Real-world objects consist of continuous curves and lines while on a computer screen, lines can only be displayed as a collection of pixels. Therefore lines, unless they are perfectly vertical or horizontal, will appear jagged.

To achieve perfect elimination of aliasing, spatial sampling should be done at the Nyquist rate or higher after applying an anti-aliasing filter. We can classify these anti-aliasing techniques into two broad categories: spatial and post process anti-aliasing.

5.3.1 Spatial Anti-Aliasing Techniques

The most naïve method of combating the aliasing effect is increasing the sample rate, in this case the resolution of our image. This is commonly known as supersampling, which is a spatial anti-aliasing method that aims to reduce the aliasing effect by rendering the image at a much higher resolution than the one being displayed. By downsampling the high-resolution image, we get smoother transitions between pixels along the edges of objects.

However, the simplicity of supersampling comes at a cost: it is computationally expensive as the required GPU memory and memory bandwidth becomes several times larger. A less expensive and special case of supersampling is multisample anti-aliasing (MSAA), which is hard and inefficient to use in combination with deferred shading.

5.3.2 Post-Processing Anti-Aliasing Techniques

The performance drawbacks of using spatial anti-aliasing led to the development of post processing anti-aliasing techniques such as FXAA (fast approximate anti-aliasing) [12] and MLAA (morphological anti-aliasing) [15].

Early post-processing-based techniques (such as FXAA) tend towards a lower performance impact at a cost of accuracy. More recent techniques rely on temporal data from previously rendered frames to produce an anti-aliased image. Such methods are referred to as temporal anti-aliasing (TAA). Popular TAA implementations include deep learning supersampling (DLSS) by NVIDIA and AMD's FidelityFX Super Resolution (commonly referred to as FSR).

Taking the benefits and drawbacks of various anti-aliasing techniques into consideration I chose to include fast approximate anti-aliasing (FXAA) in my rendering engine.

5.3.2.1 Fast Approximate Anti-Aliasing

FXAA is a screen-space anti-aliasing algorithm created by Timothy Lottes at Nvidia [12]. Its main benefit is its considerably lower performance impact compared to

spatial anti-aliasing techniques. It achieves this by smoothing jagged edges according to how they appear on-screen while accounting for lighting, unlike MSAA. To this day FXAA is a commonly seen method in game development.

The downside of using FXAA is that it will smooth not only edges between triangles, but also edges inside textures requiring it to be applied before rendering HUD elements.

Figure 5.3: On the left: without anti-aliasing, on the right: with FXAA.

5.4 Gaussian Blur

One of the most used blur operations in image processing is the Gaussian blur. Mathematically, by applying a Gaussian blur filter to an image, we convolve it with a Gaussian function. In implementation it is best to take advantage of the Gaussian blur's separable property. With two passes, we can use a one-dimensional kernel to first blur the image in either horizontal or vertical direction, then in the other direction. The resulting effect is the same as when using a two-dimensional kernel, but computationally less expensive.

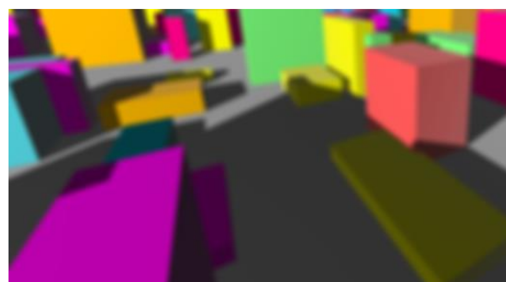

Figure 5.4: Gaussian blur applied to the rendered image.

I chose to implement a two-pass Gaussian blur using a compute shader. Using push constants, the shader can be configured to blur the image horizontally or vertically.

5.5 Ray Tracing

With the advent of GPUs capable of hardware accelerated ray tracing; interest in real-time ray tracing and path tracing has skyrocketed. All major vendors (AMD, Intel, NVIDIA) latest GPUs now support ray tracing to some capacity, even Apple joining this list with the release of its latest SoCs (System-on-a-Chip) the A17 Pro and M3 (2023).

Both DirectX 12 with DirectX Raytracing (DXR) and Vulkan with the Ray Tracing Pipeline extension support hardware accelerated ray tracing features. A critical use case for these features is real-time ray tracing in video games, typically combining rasterized scenes with ray traced aspects. Some examples include using ray tracing for ambient occlusion, shadow map generation. DXR and Vulkan Ray Tracing can also be used to accelerate offline rendering and even non-rendering techniques.

In the context of the Vulkan API, the ray tracing functionality consists of various Vulkan, SPIR-V and GLSL extensions. The primary Vulkan extensions are acceleration structure building and management, ray tracing pipelines and shader stages and ray query intrinsics for all shader stages.

Forming the backbone of high-performance ray tracing are acceleration structures (AS), an optimized data structure built on the scene information organized in a two-level hierarchy. Ray intersections are then performed against the acceleration structures. The bottom-level acceleration structures (BLAS) contain either the triangles or axis-aligned bounding boxes (AABBs) of some geometry in the scene. The top-level acceleration structure (TLAS) is made of references to bottom-level acceleration structures paired with transform and shading information. How the building of either type of acceleration structure is performed is up to the driver implementation. The BLAS is only used by reference from the TLAS. The TLAS can be accessed from shaders as a descriptor binding or device address.

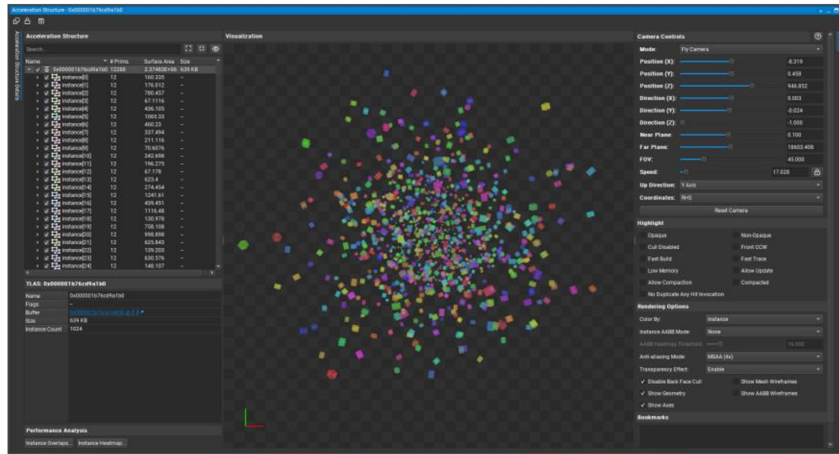

Figure 5.5: Inspecting a TLAS with 1024 BLAS instances using NVIDIA Nsight.

The ray tracing pipeline provides five new shader stages that developers can use to define pipelines of varying complexity: ray generation, closest hit, any hit, intersection and callable shaders. Ray traversal begins when a ray generation shader calls the shader language appropriate ray tracing function (`traceRaysEXT` in GLSL). During traversal, intersection and any hit shaders have the ability to control how traversal proceeds, after traversal is complete either a miss or closest hit shader is invoked. The ray tracing shader stages can communicate parameters between all stages of traversal using ray payload structures.

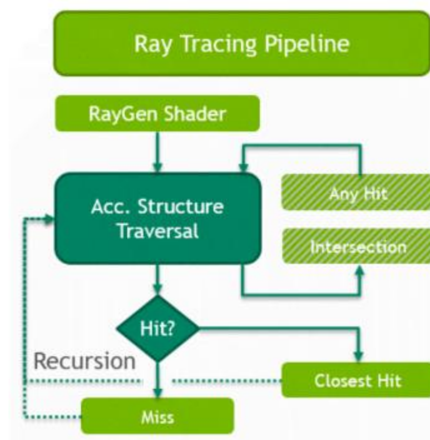

Figure 5.6: Ray tracing pipeline flow diagram (Source: [13])

The ray tracing features also include ray queries. They can be used in all non-ray tracing shader stages to perform ray intersections against the top-level acceleration structure.

6 Optimizing the Rendering Pipeline

Organizing the rendering pipeline as a graph makes several automatic optimizations possible, that otherwise would make the renderer code deeply coupled and hard to maintain. Usually, a Render Graph system is expected to handle task ordering, memory barriers and much more automatically. These expectations also include various optimizations it can make in the background, reducing the required GPU memory through resource or memory aliasing.

6.1 Task Ordering

Depending on how our renderer is structured the room for optimizing the execution varies a lot. When dispatching workloads only to a single queue, we restrict our renderer to run tasks sequentially. This makes organizing GPU workloads simple and straightforward, however can leave a significant amount of performance on the table.

6.1.1 Sequential Execution of Tasks

When executing tasks sequentially, the only restriction imposed on the ordering is defined by the dependencies between the tasks. These tasks and dependencies form a directed acyclic graph with a root node as its starting point and a sink node with its end point. Between these two points we can determine an execution order of tasks by running an algorithm such as Kahn's algorithm on our graph that topologically sorts its vertices.

The main drawback of executing tasks sequentially is that we are not utilizing the GPU to its fullest potential.

6.1.2 Parallelizing Execution Using Multiple Queues

The latest GPU architectures were designed with parallel execution of workloads in mind at a hardware level. Developers have the option to dispatch workloads to various queue families supported by the GPU which get executed in parallel. However, this introduces a completely new set of problems we need to solve. We also need to determine which tasks can be executed in parallel and now we must take care of the synchronization of workloads with fences. A Render Graph based system would be expected to do all this automatically, without any input from the user. A commonly seen use for multiple queues

that modern game engines make use of is running compute tasks asynchronously while rendering on a dedicated graphics queue.

In my Render Graph implementation, I choose sequential task execution, as it vastly simplifies the renderer architecture and compilation of the graph. Using Kahn's algorithm to form the execution order of tasks between a SceneProvider and Present node.

6.2 Resource aliasing

As mentioned earlier, it may be possible to reduce the amount of GPU memory used by a rendering pipeline by resource or memory aliasing. This effectively creates a system where resources are transient; they only exist for a certain section of the rendering pipeline.

Resource aliasing occurs when a resource is used for multiple purposes during the rendering process. While slightly easier to implement than memory aliasing, it brings several restrictions with itself. As opposed to memory aliasing, where multiple resources share the same allocated memory, resource aliasing makes use of a single, already allocated resource. In the context of GPU image resources, this would mean that for example images of different formats are incompatible with each other. This means that resource properties like image format also need to be tracked.

This optimization can be done algorithmically, for which I propose the following algorithm:

6.2.1 Algorithm for Resource Usage Range-based Memory Usage Optimization

As input, the algorithm takes in a directed acyclic graph, and a valid topological ordering of its vertices. The graph's vertices are nodes of the Render Graph, and edges represent the mapping of resources between nodes. To preserve these resource mappings while running the algorithm some bookkeeping is required.

The idea behind the algorithm is that using the input graph, we can determine for each resource which nodes will either produce or consume them. Using this knowledge then we can construct "resource usage ranges" that represent the effective lifetime of each resource. A resource usage range is defined by the earliest and latest point it is used in the execution order of nodes.

This information can be generated from the input as a preparation step. Once we have this information ready, we can run the algorithm on the evaluated resource requirements.

Pseudocode:

```
R <-- List of resource requirements with usage ranges
X <-- Empty list, will contain the optimizer generated resources
for each ResourceRequirement "rr" in R do:
     if X is empty:
          create a new optimizer resource "or"
          add the usage points of "rr" to "or"
          add "or" to X
     else:
          was_inserted <-- false
          for each OptimizerResource "or" in X do:
               if "or" is compatible with "rr":
                    add the usage points of "rr" to "or"
                    was_inserted <-- true
          if not was_inserted do:
               create a new optimizer resource "or"
               add the usage points of "rr" to "or"
               add "or" to X
     return X
```

This algorithm essentially delays extra resource creation until it is necessary. It does so by attempting to repurpose resources when their usage ranges allow for it. This is the most important constraint, as for a defined resource usage range, the data contained within a resource is expected to be constant by its users, therefore overlaps of ranges cannot be allowed under any circumstances. In other words, resources can only be aliased if they are disjoint in time. However, this means that after or before said range, it is possible to use the resource for a different purpose.

The beauty of this algorithm comes from its flexibility, the compatibility constraint can be changed and used to control how exactly resources are aliased. As mentioned, the most important constraint is that usage ranges cannot overlap with each other, but other than that, we are free to define any other constraints we want.

6.2.2 Optimality

The algorithm described above is essentially a greedy coloring algorithm. Examining the problem in the context of graph coloring it is possible to prove that by applying a greedy coloring in the reverse of the perfect elimination ordering of an incompatibility graph's vertices we get the optimal solution.

Let graph $G = (V, E, R)$ be the compatibility graph of resources. Each vertex in the set V represents a resource. The conditions for compatibility between vertices can be defined by us.

6.2.2.1 Accounting for compatibility

Compatibility can be defined as a symmetric transitive homogeneous relation R on the set V. If the relation R applies to vertices u and v we add the edge (u, v) to G. Therefore, the set of edges E of graph G is composed of edges between compatible resources. Due to the nature of compatibility we can assume that the relation R is a symmetric transitive relation, therefore the following statements are true for R: $\forall u, v, w \in V: (u\ R\ v\ \wedge\ v\ R\ w) \Rightarrow u\ R\ w$ and $\forall u, v, w \in V: (u\ R\ v, u\ R\ w) \Rightarrow v\ R\ w$. Therefore if u and v, v and w are compatible, then so are u and w. From the symmetric transitive property of compatibility, we can conclude that in the compatibility graph G a set of compatible vertices will always form cliques. A clique is an induced subgraph of G that is complete.

Now if we take the graph $G' = (V, \bar{E})$, we get a graph with edges between incompatible resources. Since in graph G compatible resources formed cliques, there are no edges between compatible resources in graph G'. Let us take a maximal clique (complete subgraph) H of G, then let the set of incompatible vertices be $\overline{H}$, the complement of the set H. In the graph G' the following applies: $\forall v, w \in H: (v, w) \notin \bar{E}$ and $\forall x \in H, y \in \overline{H}: (x, y) \in \bar{E}$. This means that in graph G' every vertex of a clique (in G) will be connected to the vertices of the other cliques. Therefore, we can conclude that G' is a complete multipartite graph, where the partitions are formed from cliques in the original graph G. Without temporal constraints on compatibility the optimal number of resources would be the number of partitions in graph G'.

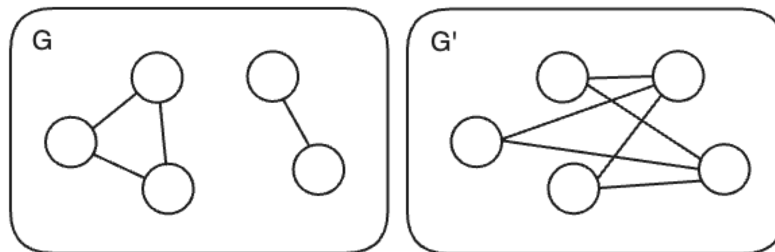

Figure 6.1: Visualizing cliques transforming into a complete multipartite graph.

6.2.2.2 Accounting for the temporal constraint

However, for the problem of resource aliasing we also need to consider the effective lifetime of resources, introducing a temporal constraint on compatibility. Using the calculated effective lifetime of resources, it is possible to construct an interval graph $I = (V, E_I)$, where vertices are connected if their effective lifetimes overlap.

After partitioning the vertices based on our non-temporal constraints into n sets, we can apply the relevant edges of I to each set, resulting in n interval graphs. Since interval graphs are a special case of chordal graphs, we can make use of the fact that chordal graphs are perfectly orderable. Therefore, an optimal coloring of its vertices may be obtained by applying a greedy coloring algorithm in the reverse of its perfect elimination ordering using unique colors across partitions. This generated coloring is the optimal solution to the resource aliasing problem.

Based on the above, we can modify the proposed algorithm to iterate over each set of compatible resources separately, and if we order R by the reverse of the G' graphs perfect elimination ordering we get an optimal solution.

6.2.3 Time complexity analysis

The time complexity of the algorithm can be described by $O(n^2)$. Let the size of the set Rgenerated during the preparation step, be $|R|$. The algorithm's outer loop runs for $|R|$ iterations. The inner loops iteration count will never exceed the size of $|R|$. Therefore, the algorithm takes quadratic time: $O(|R|^2) = O(n^2)$. The optimal version of the algorithm does not impact the time complexity.

6.2.4 Results

In my implementation the algorithm only optimizes image resources, and all other types are marked as non-optimizable. Since I am dealing with image resources, I also need to include image format as a constraint. This resulted in four constraints: no overlapping ranges, resource types and image formats must be compatible, and the resource must be marked as optimizable.

Take the following Render Graph as example:

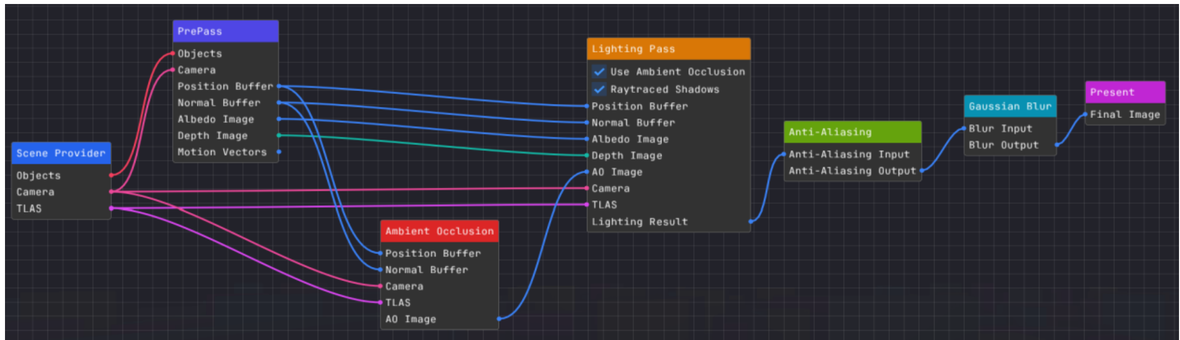

Figure 6.2: Example Render Graph with seven nodes.

After running the algorithm on this graph, the number of required image resources was reduced from 9 to 6. For the specific graph shown above, optimization resulted in a 40% reduction in memory usage at various resolutions.

The figures 6.3 and 6.4 showcase the resources allocated by the render graph compiler with and without optimization.

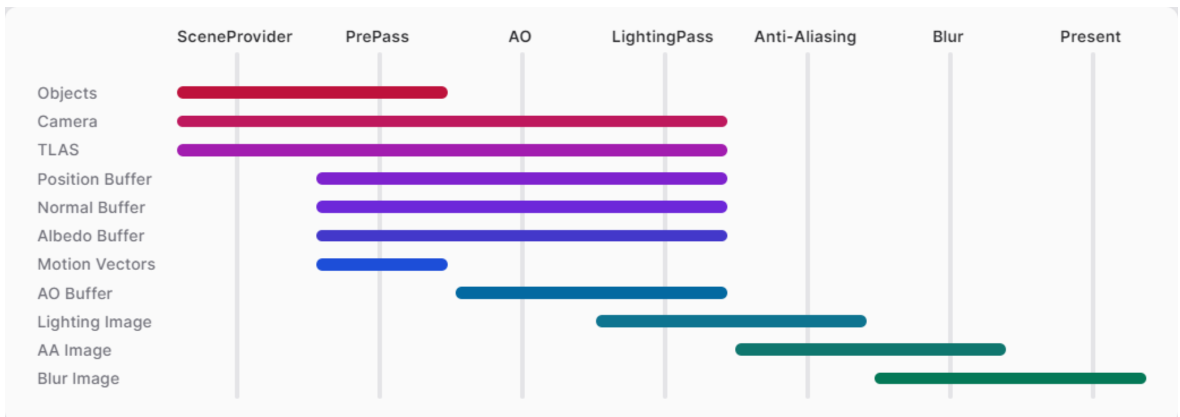

Figure 6.3: Unoptimized resource allocation. Resource allocations are exclusive, each horizontal line represents the effective lifetime of a resource.

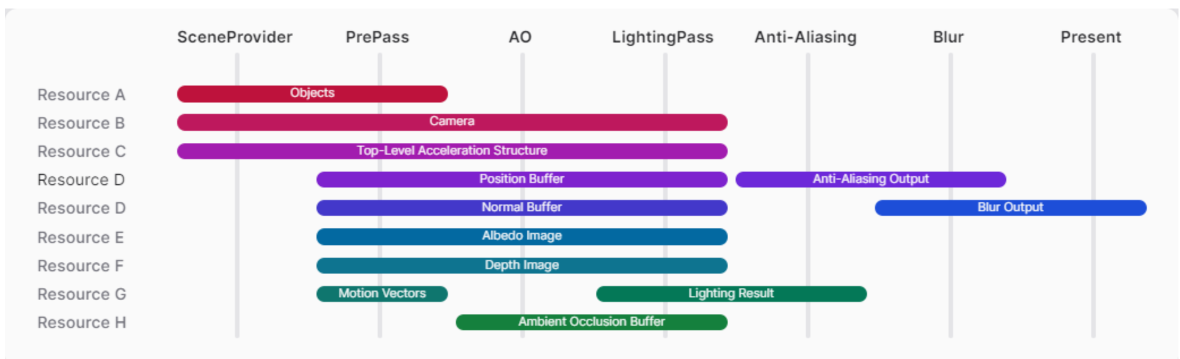

Figure 6.4: Optimized resource allocation. Resource allocations are shared, each horizontal line represents different usages of a resource.

The following tables detail the memory consumption of images before and after optimization:

Color Channels	Bits per Color	Resolution	Memory
4	32	1920x1080	34.5 MB
1	32	1920x1080	8.85 MB
4	32	2560x1440	62.9 MB
1	32	2560x1440	15.7 MB

Table 6.1: Images and their GPU memory requirements at various resolutions

Resolution	Image Count	4 Channel Images	Single Ch. Images	Σ Memory
1920x1080	9	7	2	265.5 MB
2560x1440	9	7	2	471.1 MB

Table 6.2: Pre-optimization image count and memory usage

Resolution	Image #	4Ch. Images	1Ch. Images	Σ Memory	Reduction	Reduction
1920x1080	6	4	2	159.3 MB	106.2 MB	40.0000%
2560x1440	6	4	2	283 MB	188.1 MB	40.0042%

Table 6.3: Post-optimization image count and memory usage

While for a simple graph like the example; a reduction of 188.1 megabytes may not seem like much, but for pipelines many times the size and complexity of the example it can end up saving a significant amount of memory. Applying this 40% reduction in memory usage may allow for more memory for nodes in a more complicated Render Graph, or an increased texture streaming budget which is increasingly important in an age where the use of 4k high-resolution textures is increasingly common.

6.2.5 Memory aliasing

As mentioned earlier, memory aliasing occurs when multiple resources share the same allocated memory. While not as easy to integrate as resource aliasing, it comes with the benefit of not needing to conform to an already existing resource's properties while

optimizing. Instead, we need to pay careful attention to the memory requirements of each resource.

The previously detailed algorithm with a slight modification is perfectly suitable for memory aliasing. Let our only constraint for allowing aliasing be that there must be no overlapping usage ranges. While running the algorithm we keep track of the memory requirements at each point of usage, and at the end take the largest memory requirement and use that while allocating.

6.3 Existing solutions

Similar algorithms have been described by others previously such as in an article by Pavlo Muratov titled "GPU memory aliasing" [14], in which they propose an algorithm to solve memory aliasing. They did not evaluate the optimality of their proposed algorithm, but similarly to the algorithm designed by me, it works by first evaluating the "effective lifetime" (which I referred to as usage range) of resources.

7 Conclusion and Future Work

In this work, after evaluating existing implementations I introduced a way of designing a Render Graph based rendering engine then explored methods to optimize the memory usage of the rendering pipeline. Using various commonly used techniques in computer graphics I highlighted the possibilities of such a rendering engine. The node editor provides a quick and effortless way of prototyping and testing rendering pipelines. Complementing the node editor, the builder interface provides a simple method for building complex rendering pipelines in code.

With the algorithm I introduced I saw a reduction of up to 40% in video memory usage by resource aliasing. While not yet perfect, these savings can be significant for larger, more intricate rendering pipelines. Doing memory aliasing would improve the solution, as it would eliminate restrictions like image formats for aliasing.

Moving forward, aside from implementing memory aliasing, extending my Render Graph implementation with support for parallel task execution would be the next major step in improving its capabilities.

8 Bibliography

[1] O'Donnel, Yuriy: *FrameGraph: Extensible Rendering Architecture in Frostbite*, https://www.gdcvault.com/play/1024612/FrameGraph-Extensible-Rendering-Architecture-in (2017)

[2] Rodrigues, Tiago.: *Moving to DirectX 12: Lessons Learned*, https://www.gdcvault.com/play/1024656/Advanced-Graphics-Tech-Moving-to (2017)

[3] Epic Games: *Render Dependency Graph in Unreal Engine*, https://docs.unrealengine.com/5.3/en-US/render-dependency-graph-in-unreal-engine (2023)

[4] Heslik, Chris (AMD): *New Work Graphs sample and Radeon GPU Profiler* support for GPU Work Graphs, https://gpuopen.com/learn/rgp-work-graphs/ (Aug. 2023)

[5] A. B. Kahn: *Topological sorting of large networks*, Commun. ACM 5, 11 (Nov. 1962), pp. 558–562. https://doi.org/10.1145/368996.369025

[6] Cormen, Thomas H.; et al.: *Introduction To Algorithms.* Edited by Thomas H. Cormen, MIT Press, 2009.

[7] Corneil, Derek G. (2004), "*Lexicographic breadth first search – a survey*", Graph-Theoretic Methods in Computer Science: 30th International Workshop, WG 2004, Bad Honnef, Germany, June 21-23, 2004, Revised Papers, Lecture Notes in Computer Science, vol. 3353, Springer-Verlag, pp. 1–19, doi:10.1007/978-3-540-30559-0_1.

[8] ImGui: https://github.com/ocornut/imgui

[9] Imnodes: https://github.com/Nelarius/imnodes

[10] Bavoil, Louis.; et al.: *Screen Space Ambient Occlusion* https://developer.download.nvidia.com/SDK/10.5/direct3d/Source/ScreenSpaceAO/doc/ScreenSpaceAO.pdf (Sept. 2008)

[11] Bavoil, Louis.; et al.: *Image-Space Horizon-Based Ambient Occlusion* https://developer.download.nvidia.com/presentations/2008/SIGGRAPH/HBAO_SIG08b.pdf (SIGGRAPH 2008)

[12] Lottes, Timothy (NVIDIA): *FXAA* https://developer.download.nvidia.com/assets/gamedev/files/sdk/11/FXAA_WhitePaper.pdf (Feb. 2009)

[13] Koch, Daniel.; et al.: *Ray Tracing in Vulkan* https://www.khronos.org/blog/ray-tracing-in-vulkan (March 2020)

[14] Muratov, Pavlo: *GPU Memory Aliasing* https://levelup.gitconnected.com/gpu-memory-aliasing-45933681a15e (Jul. 2020)

[15] Reshetov, Alexander: *Morphological Antialiasing,* Proceedings of the Conference on High Performance Graphics https://www.intel.com/content/dam/develop/external/us/en/documents/z-shape-arm-785403.pdf (2009)

[16] Microsoft: *D3D12 Work Graph*s https://github.com/microsoft/DirectX-Specs/blob/master/d3d/WorkGraphs.md (2023)