Real-time Game Mechanics & Procedural Tooling with Vulkan API

(Using SDF Raymarching)

Master's Thesis

Hirad Yazdanpanah

MSc in Computer Animation and Visual Effects

National Centre for Computer Animation

Bournemouth University

August 2021

Abstract

SDF raymarching algorithms have been typically slow for real-time use until today. And no matter how optimized and efficient they are, real-time and interactive usage of these algorithms, once the CPU-GPU communication occurs, can be quite resource-intensive and expensive, which may lead to low performance and stalling the application.

This thesis describes a Vulkan-based implementation for SDF Raymarching in real-time aiming to provide a proof-of-concept, as a mean for creating efficient solutions that simplify runtime evaluation of complex data through the GPU with a low-overhead CPU to GPU abstraction.

Keywords

Vulkan, Direct3D, Metal, Pipelines, Command Buffers, Graphics Queue, SDF, Ray marching, Sphere Tracing,

Acknowledgements

I would like to thank my supervisor, Jon Macey, for his guidance, support, and patience through every step of this project.

I also, would like to thank my family and friends for their continuous support and encouragement throughout this path.

Contents

List of Abbreviations

AoS	Array of Structures
BVH	Bounding Volume Hierarchies
DCC	Digital Content Creation
DX 11/12	Direct3D/2D 11/12
GLSL	OpenGL Shading Language
kD-Tree	k-dimensional Tree
MOC	Meta Object Compiler
SDF	Signed Distance Field/Function
SPIR-V	Standard Portable Intermediate Representation - Vulkan
POD	Plain Old Data
PSO	Pipeline State Object
SDFR	SDF (Signed Distance Field/Function) Raymarching
SIMD	Single Instruction, Multiple Data
SoA	Structure of Arrays
VMA	Vulkan Memory Allocator

List of Figures

Chapter 1 Introduction

1.1 Background

Following a previous project in the personal inquiry unit, the author implemented and demonstrated dynamic transformation of 3D objects to 4D and vice versa in real-time as a game mechanic feature, using SDF Raymarching, with HLSL compute shaders, in Unity Game Engine.

The implementation, however, needed to address some more details to implement and future work which are as follows:

- It lacked correct depth buffer calculation in the shaders so that it can project Mesh-based rasterized models along with the SDF-based models (e.g., an interactive player character) in the same scene.
- It lacked certain optimizations for the SDF raymarching algorithm (e.g., scalability)
- CPU to GPU Performance overhead was automatically managed by the named Game Engine through Direct3D 11 API
- There was no SDF modelling toolset, to simplify the usability of this feature for the game artists and developers.

A screenshot of the implementation is illustrated at Appendix A1 with follow up improvements after the project delivery from Appendix A2 onwards demonstrating its performance comparisons in different Graphics APIs.

1.2 Aims & Objectives

The aim of this project is to provide a proof-of-concept solution addressing some of the above-mentioned problems using the Vulkan graphics and compute API as an optimized ecosystem enabling and allowing for any further advanced optimization techniques.

Therefore, it should be noted that the concentration of this project is to represent an optimum solution using Vulkan-based functionalities towards approaching such complex problems, rather than an algorithmically extensive approach towards SDF raymarching, though briefly discussed, in this thesis.

The Vulkan-based implementation is written in C++ using SPIR-V shaders which some are pre-compiled during cmake compilation with shell script instructions and some at runtime through GLSLang library to be consumed by the SDF Modelling Toolset.

Chapter 2 Previous Work

2.1 SDF implementations

2.1.1 Games

SDF-based raymarching algorithms and SDF-based point/surface splatting have been used amongst a few games in real-time, from amateur to industry standard, which were also named in the previous project briefly (Yazdanpanah 2021) and can be found in Appendix B. Although, their use of these algorithms and their target audience is quite limited due to technical complexities, including the one this project attempts to propose a solution for.

2.1.2 SDF Modelling Toolset

One of the most missing features in SDF Raymarched implementation is, the ability to create SDF Models easily with no requirement to understand programming. Thus, to engage with the artists allowing them to create implicit surfaces and morphed objects in real-time specially for games. This is because these types of models do not depend on rasterizing vertices and can be adopted to respond to quite unique modelling requirements (Sanchez et al. 2015; Lindborg et al. 2017).

Node-based procedural modelling has become quite popular technique for creating game assets, with efficient and quite fast results. Therefore, procedural SDF Raymarched Modelling can also provide new opportunities to create special geometries that would be, otherwise, quite difficult to create using the existing DCC and procedural modelling toolsets (Lindborg et al. 2017).

2.2 Vulkan Implementations

Vulkan API has been used across many real-time engines and games despite being a relatively new API since 2016 (Khronos 2016). A non-exhaustive list of these engines and games can be found in Appendix C.

There are also some abstractions and libraries around Vulkan SDK such as the one that author has used for this implementation, through which they require less developer time and can allow for better understanding of the graphics pipeline while can be used to provide proof of concepts and minimum viable products much faster and build further. These implementations include but not limited to (Willems et al. 2021):

- Qt Vulkan Renderer (Qt-based wrapper around Vulkan SDK)
- vk-bootstrap (C++ utility library automating Vk instance, device & Swapchain creation)
- Google's Vulkan-cpp-library (C++ 11 Vulkan abstraction library)
- V-EZ (light-weight middleware layer for Vulkan API)
- AMD's Anvil (cross-platform framework for Vulkan)
- Intrinsic Engine (Vulkan based game engine)
- Spectrum (framework and abstraction layer for Vulkan)
- MoltenVK (MacOS/iOS compatible Vulkan Interop – Metal API interface)

Chapter 3 Technical Background

3.1 OpenGL, Direct3D, Metal, Vulkan, etc.

Starting with OpenGL as to provide reasonings for further discussions on why there are other Graphics APIs like Vulkan. OpenGL as a high-level abstraction graphics API continually being developed by Khronos Group and supported by most platforms.

It is basically a large state machine that keeps track of application state, enabling its users to take advantage of availability of the software components across different contexts and platforms, while abstracting and hiding away the low-level architecture of the software by managing those resources on its own (de Vries).

These abstractions include memory management and host-device synchronization. Error handling, checking and validation is also embedded in OpenGL for both development and production environments. Shader compilation is also done at runtime. All these contribute to more performance overhead.

Vulkan, on the other hand, is a low overhead graphics and computing API also by Khronos Group, with a much more optimized abstraction to access the GPUs. Therefore, it provides better performance for the user and less error prone and surprising driver behaviours comparing to other existing graphics APIs. The advantage of Vulkan not only stems from its efficiency and productive approach, but also it is fully cross-platform across major platforms (Overvoorde 2020).

As opposed to Other APIs, such as OpenGL and Direct3D 11, Vulkan requires deeper understanding of the graphics pipeline, as the API demands for setting up every detail from scratch. Although, this verbose approach, may seem quite cumbersome at the beginning as it necessitate for significant amount of work and depth of knowledge, once the setup is made the graphics driver needs far less attention (Overvoorde 2020).

Up until recently, Direct3D 11 and below, had similar approach to OpenGL with high level of abstractions and therefore less verbose, which may have contributed to many CPU-GPU related performance bottlenecks (Sheng and Nan 2018).

Direct3D 12, however, introduced a lower level of abstraction with improved multithreaded scaling and utilizing far less CPU resources than its earlier versions. DX11 comparison with DX12 are mainly around the level of consumption for the resources between the CPU and the GPU (Sheng and Nan 2018; J'Lali 2020).

Direct3D 12 follows a similar pattern to of Vulkan's, where developers must maintain and control the resource management of the graphics application explicitly rather than offloading the burden onto the driver (Microsoft 2018b).

Concepts like Descriptor Sets and Tables, Pipeline State Objects (PSOs), Batches of Commands and Draw Calls, are quite common with narrow borderline between Vulkan and Direct3D 12, which some of these will be discussed in a bit further detail in the next chapters.

Metal API from Apple, also aimed to produce low-overhead Graphics and Compute API, combing features from OpenGL and OpenCL to improve performance by low-level access to the GPU, initially targeting mobile platforms.

This API is comparable to Vulkan and Direct3D 12, in the sense that it utilizes GPU low-level access to batch and encode commands before submitting to the GPU with asynchronous execution. This is quite comparable to Command Buffer Generation and Graphics Queues in Vulkan (Apple 2021).

Most recently, another low-level graphics API has emerged currently known as WebGPU attempting to follow the same pattern to other low level graphics APIs, addressing similar problems, while being developed by the web community engineers (Malyshau and Ninomiya 2021).

These similarities are mainly because, all these APIs attempt to take the same approach in rendering and accessing data, to the graphics hardware and to how they are built and operate internally. GPUs essentially, are asynchronous compute units which can access and process a significant amount of data. Providing lower-level access to the hardware opens a new door for enhancement and optimization.

3.2 Optimization Techniques

3.2.1 GUI and Graphics API

In optimising a Graphics application, several points need to be considered, which are mainly related to speed and memory usage, to have a performant and optimized implementation.

At this point this is an abstract overview on how basic optimizations can be applied for complex graphics applications like this one which will be further discussed in more details.

It is worth noting that by performance the author refers to efficiency of both memory usage and speed as a co-dependent and unified component rather than independent subjects. For example, if only the speed of a runtime process is optimized, that might not necessarily involve less memory usage but quite the opposite. Therefore, it would not be considered as an efficient optimization.

3.2.1.1 GUI Overhead (Qt Interface)

Naturally, using a Native API could be the most performant way to implement a GUI application. However, Qt as a wrapper around the native API has some features helping to lower the performance bottleneck:

- Qt Signal-Slot fast mechanism (statically typed and MOC slot method calls)
- Qt Multithreading (QtConcurrent & QFuture) - equivalent of C++ std::async & std::future

3.2.1.2 Graphics API Abstraction Overhead (Qt Vulkan Renderer)

As stated earlier, Vulkan already contributes directly to the optimization of the rendering engine. But the question is whether the abstraction implemented around the API may involve any overhead that requires attention and further optimizations.

As Qt framework supported Vulkan rendering since version 5.10 (Agocs 2017), the implementation is being further developed and optimized. However, the overhead of any wrapper around the Native Vulkan API weighs the same for the purpose of this project, unless the application requires quite explicit micro-optimization for certain scenarios in the future.

For instance, Memory Allocation is managed via Qt Vulkan Device Functions which has no more visible overhead than if it was managed natively or through VMA, which is a Vulkan Memory Allocation Library, simplifying the creation and allocation of resources, while giving access to Vulkan functions.

However, it is important to note that the resource management (i.e., objects creations and destruction) within the project's implementation are carefully considered for efficient usage of Vulkan functions across the application lifecycle.

Also, Shader Compilation/Loading, Pipeline Creation and Command Buffer Generations can be certainly optimized within the project scope using Multithreading, which will be discussed in the next chapter.

3.2.2 Storage Buffers to send SDF data per bounded volume

There is also a method through which the SDF related model data, such as position, scale, rotation, colour, shape type, the Boolean operation, etc., can be passed over through to the shaders (i.e., either Fragment or Compute Shaders) by what are known as Compute or Structured Buffers in Unity Engine (Unity 2021) or Direct3D (Microsoft 2018a) terms and in Vulkan as Storage Buffers (Blanco et al. 2020).

Storage Buffers are generally used as GPU Data Buffers which are typically used with compute shaders, though can be utilized with fragment shaders too. They're normally used to hold data for all the objects within a scene. And as they're unsized arrays, it is important to explicitly ensure when and how many times they need to be sent over to the GPU to avoid any performance loss (NVIDIA 2015; Blanco et al. 2020).

These essential model data or in other words object matrices can be uploaded at the beginning of the frame all at once and it is no longer needed to exchange data on every draw call (Blanco et al. 2020). Then on the GPU side they can be used to bind the raymarching algorithm with an acceleration structure algorithm such as BVH for an individual volume to cast rays for. Hence more efficient usage of GPU device by batching data in advance on the host (i.e., through the CPU), and finally sending a bundle over to the GPU to consume without major interruptions.

This method was implemented during the personal inquiry unit with Unity Engine, C# and HLSL compute shaders. As the engine has all the required abstractions available for the developer at their disposal, it was a straightforward approach.

However, when it comes to Vulkan implementation with no boilerplate code to rely on, the requirements become much larger than the scope of the project and timeframe. Therefore, the author decided to keep focusing on the Vulkan basics implementations for the purpose of this thesis.

3.2.3 SDF Raymarching

Optimizing SDF Raymarching can be done in a few different ways from SIMD instructions with Structure of Arrays (SoA), creating Uniform Buffer Objects for transferring the volumetric data, to Acceleration Structures with space subdivision methods such as BSP, kD-Tree, Octree, etc. and object subdivision methods such as BVH, and finally pre-processing.

Acceleration structures are algorithms aiding to determine, which object in the scene, a ray is more likely to intersect amongst other objects and therefore to ignore others.

Due to the wide diversity of these techniques, in this project, the author attempts to apply a basic optimization for SDF raymarching and consider the possibilities while briefly describing them in the thesis.

3.2.3.1 Space Subdivision Methods (e.g., BSP, kD-Tree, Octree, etc.)

These methods typically subdivide the space with planes recursively with no reliance on the geometry through the space. As they don't rely on the geometry, if the geometry should change (i.e., a dynamic geometry) they typically need to recreate the acceleration structure. Also, they could result in deeper recursive trees, making them inefficient in many cases (Glassner 1984; MacDonald and Booth 1990).

3.2.3.2 Object Subdivision Methods (e.g., BVH)

This method subdivides the geometry recursively into smaller pieces, until it wraps around each piece with a closely tight bounding volume. In evaluating SDF models this approach could result in a reduced complexity of the SDF evaluation, and therefore less expensive (Thrane and Simonsen 2005; Quilez 2019).

The more complex the SDF data, the deeper the Bounding Volumes recursion needs to be. There are however some efficient BVH algorithms to use for such complex SDF data which can alleviate the complexity by automatic recursion for example (Wodniok and Goesele 2017; Quilez 2019).

Both these classes of algorithms are efficient in mostly interactive rendering at their best and the main issue remains which is the real-time rendering (Schütz et al. 2020). So, why not look at this problem from a different angle, as quite often the real-time, refers to dynamic scene and geometry and therefore real-time interactions throughout, which requires more CPU usage and communication with the GPU.

Chapter 4 Implementation

4.1 Application Architecture

The architecture of the application is represented in Figure 4-1. It uses Vulkan as the graphics API, integrated with Qt as its main windowing system, and takes advantage of two more external libraries, as follows:

- ***GLSLang***, which is used for compiling GLSL shaders into SPIRV bytecodes for Vulkan
- ***Node Editor***, which is a node-based editor used for SDF Modelling (i.e., SDF Graph)

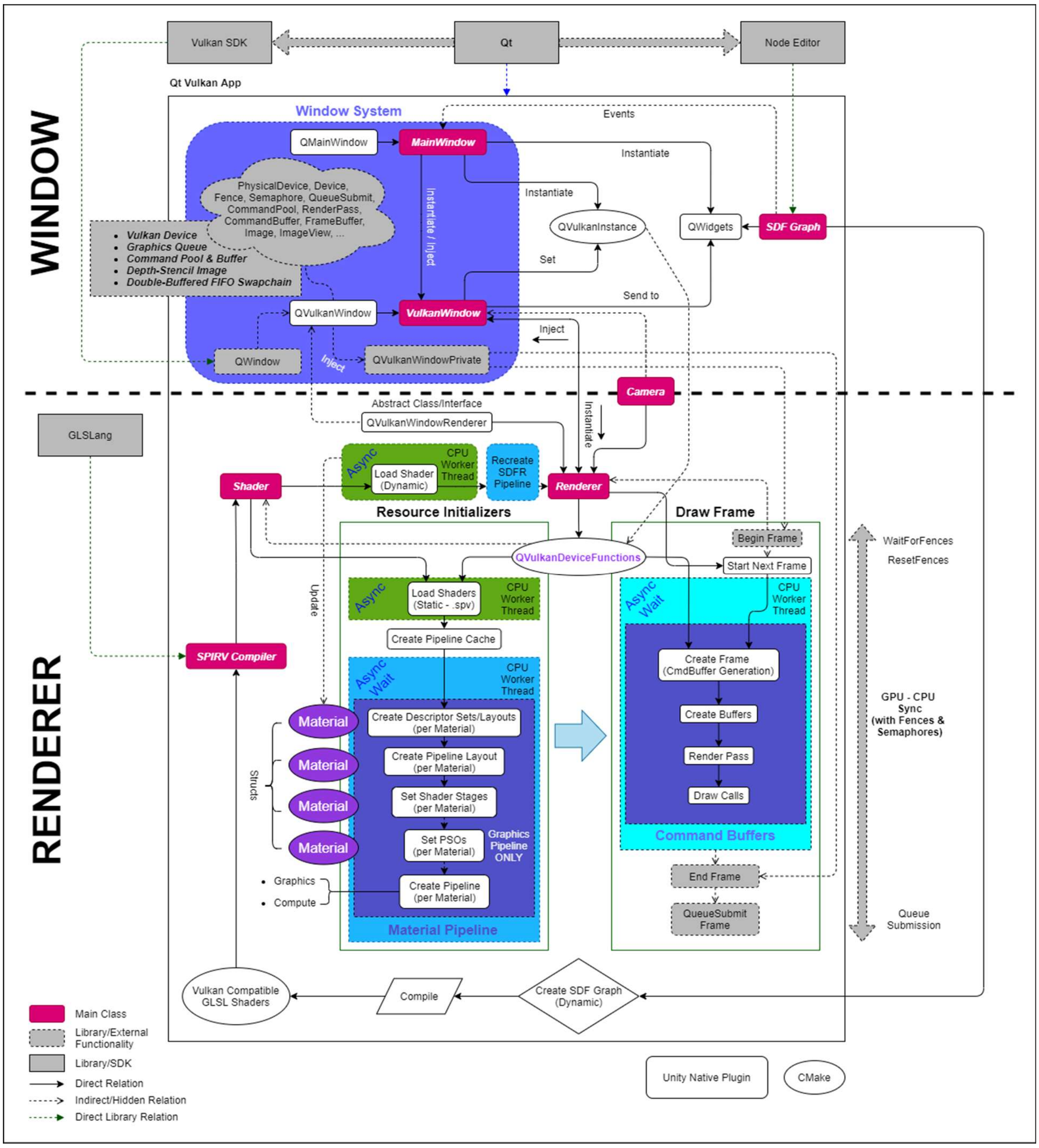


Figure 4-1 – Application High Level Architecture

4.1.1 Overview

The Qt Vulkan Implementation is basically an abstraction around the Vulkan API which exposes Vulkan instance and functions through the Qt interface, while taking care of the CPU-GPU (host-device) Synchronization, Swapchain creation and other major Vulkan functionalities. Although they're abstracted away from the user of the library, they can still be overridden by custom implementations.

The reason for choosing Qt as opposed to other GUI systems was that, firstly the Node Editor library adopted for this implementation depends on it extensively. And secondly, the focus of this project is not the fundamentals of entire Vulkan ecosystem and therefore many of the complexities within, can be hidden away to allow for implementation of a basic proof of concept using Vulkan with SDF Raymarching, with near efficient results.

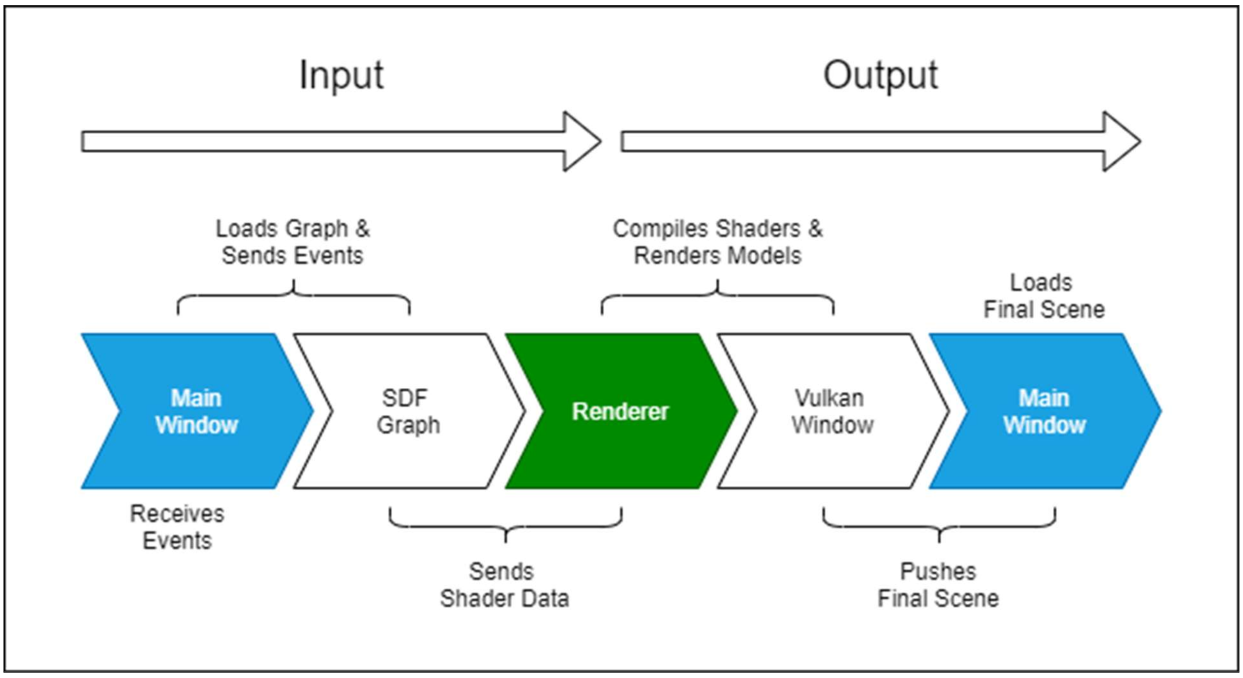


Figure 4-2 – Abstract IO Control Flow

The application is split between a Windowing system and a Renderer. The Windowing system controls the Visual Scenes and the Widgets visible to the end user, while the Renderer integrates the Vulkan Rendering Pipelines to the Scenes, which both will be discussed further in this chapter. Figure 4-2 illustrates an abstract overview of the Input/Output control flow of the Windowing Systems and the Renderer.

The Windowing System itself is split between Vulkan Window and Main Window, each having their own input event management mechanism depending on whether the input is from either the Node Editor (i.e., SDF Graph) or the actual Scene. The Vulkan window is injected into the Main Window to push the events and rendered scenes through from the Renderer to the final view.

The Renderer inherits Vulkan functions to create and execute the pipelines and interact with the scenes, while integrating with Shader related functionality. It also integrates with the SDF

Graph that allows for compiling node-based shader graph into SPIR-V shaders at runtime, to be consumable for the renderer in the pipeline.

4.1.2 Build and Configuration

CMake build system was used to build the project with UNITY_BUILD activated in for faster builds.

Shaders were split between statically loaded shaders which are compiled at compile-time and converted to SPIRV files using shell scripts to detect and concatenate if needed. Dynamically loaded shaders are the ones to be used by the SDF Graph and are in plain GLSL format. Although some are configured into separate shader files for each shader language so that they can be maintained easily for the user and then at runtime compile they're concatenated and serialized correctly to form the final SPIRV format.

4.1.3 Debugging

As the application utilizes graphics implementation with shaders and GPU related functionality, not only there was a need for C++ debugging which the development editor could already provide, but a combination of other debugging applications was needed in different scenarios to debug per frame shader data and other graphics and rendering related functions.

So, NSight from NVIDIA and RenderDoc an MIT Licensed Open-Source graphics debugger along with Vulkan Validation Layers were used providing very useful information in quite complex situations.

4.1.4 Windowing System

4.1.4.1 SDF Graph Viewport

As the name suggests, this viewport manages editing the node-based graph for SDF Raymarched models and pushing them through to the renderer to compile and load on the actual scene viewport (Vulkan Window).

The structure that makes this modelling system possible is illustrated in Figure 4-3. It basically uses a Node tree to recursively traverse through, and serialize the shader instructions into a single, final, and valid SPIRV-compatible GLSL shader and then compile.

Each Node in the tree represents a specific shader instruction/function which can take a course to connect to other relevant intermediary nodes (i.e., 3D Primitive Geometries and Boolean Operations), specifying the shader functions execution control flow.

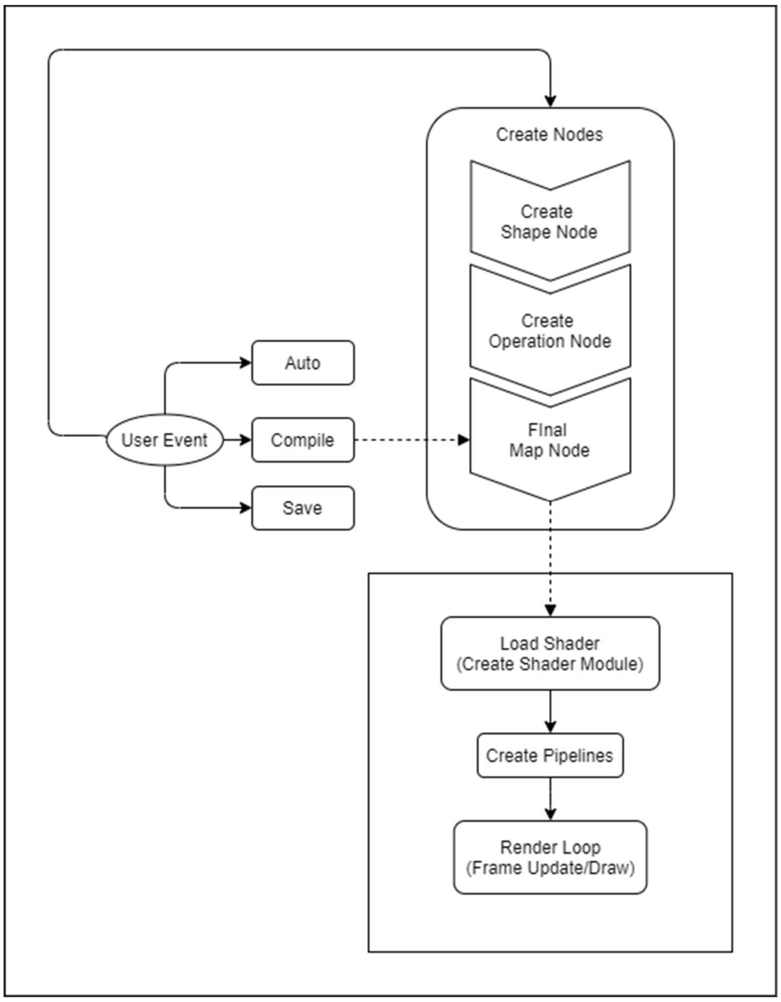


Figure 4-3 – SDF Graph Basic Architecture

Once this execution control flow specified, it needs to be finalized by connecting to the final output node which represents a map function in the shader, that its result is the SDF Model based on the Primitive Geometries combined with the specified Boolean Operations.

The node editor library provides a data model class for any type of node data, which are used here to represent the 3D Primitive Data Models and the Operation Data Models.

The Object-oriented architecture for this graph is naturally following the same pattern of the library itself as it uses inheritance with abstract classes which at certain points serve as pure interfaces that force the subclass to implement the function of their superclass owner.

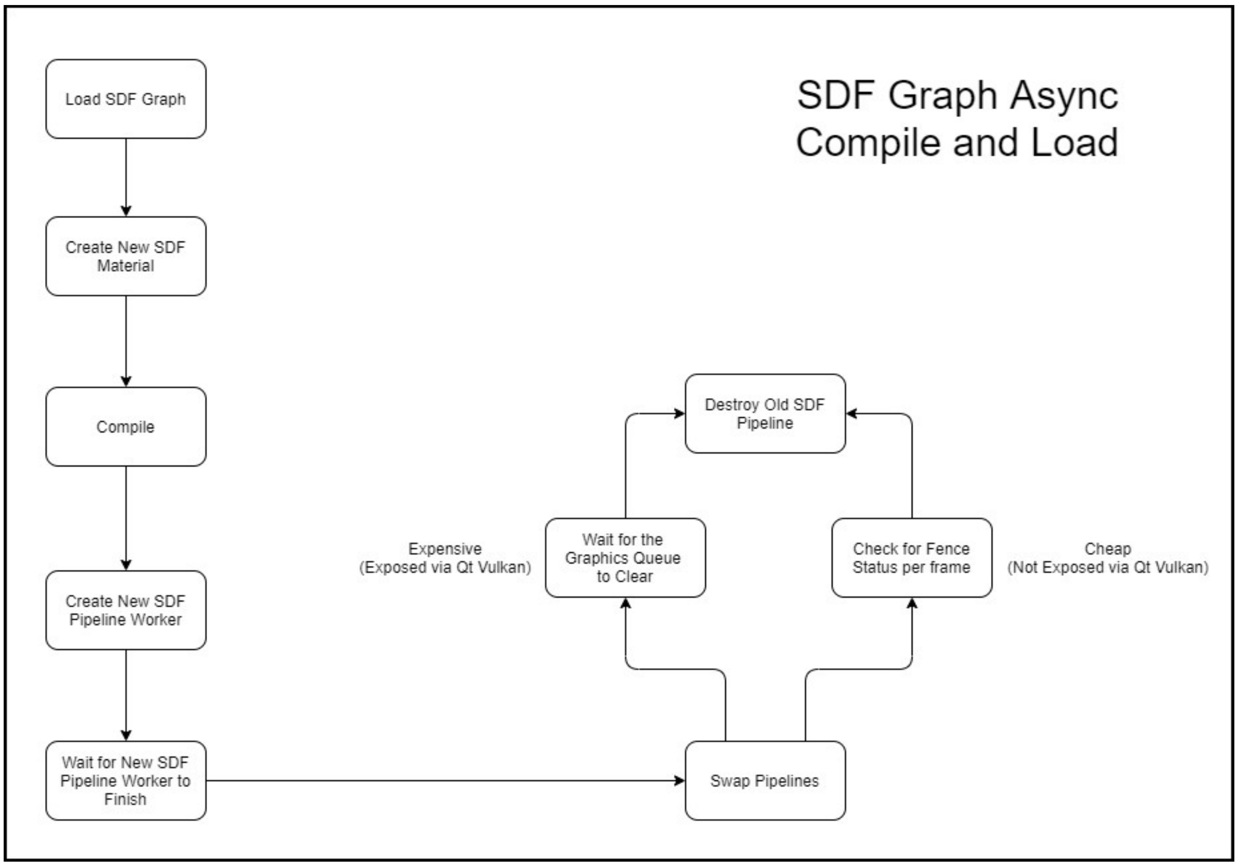

SDF Graph Async Compile and Load
Load SDF Graph
Create New SDF Material
Compile
Create New SDF Pipeline Worker
Wait for New SDF Pipeline Worker to Finish
Swap Pipelines
Wait for the Graphics Queue to Clear
Expensive (Exposed via Qt Vulkan)
Check for Fence Status per frame
Cheap (Not Exposed via Qt Vulkan)
Destroy Old SDF Pipeline

Figure 4-4 – SDF Graph Async Data Flow Model using Vulkan Pipelines

The graph needs to push shader data to the scene once compile them through nodes. It is a relatively complex and delicate data flow to achieve an efficient runtime shader load through the Vulkan pipelines while being aware of running command buffers per frame.

This process is illustrated in Figure 4-4, which represents an asynchronous multithreaded data flow while creating new pipelines and destroying the olds ones when the command buffers finished using them.

4.1.4.2 Scene Viewport (Vulkan Window)

This viewport is the glue between the Main Window and the Renderer, which, as mentioned briefly earlier, it pushes the rendered scenes and events through to the Main Window after they're processed.

As with the main Window, this Window also uses a signal-slot event system to handle widget-based events and allow for interactivity through the session.

4.1.5 Renderer

This is the core of the application, where the rendering engine operates. In an abstract view, the Renderer consists of Resource Initializers and per-frame Command Buffers Generation & Draw Calls. As shown in Figure 4-1, Resource Initializers handle the Pipeline and Swapchain resources at the start of the application and before the frame loop, while Command Buffers & Draw Calls manage the Render Pass and any Sub-Passes if available, after the resource initialization and per frame.

4.1.5.1 Pipelines

The entire workflow of the Renderer as with any Vulkan implementation is through creation and execution of the pipelines as it was briefly mentioned earlier. So, the core structure of the application must be optimized to respond to this requirement as to manage multiple pipelines efficiently and to allow for introducing new pipelines as and when needed.

Therefore, by introducing structured generic pipeline data containers that can be reused and customized, the application's complexity can be reduced to an abstracted reusable container which holds data for each specific pipeline and with attributes that correspond to main components of render-able 2D or 3D objects. These components form what most Game Engines and DCC tools call a Material.

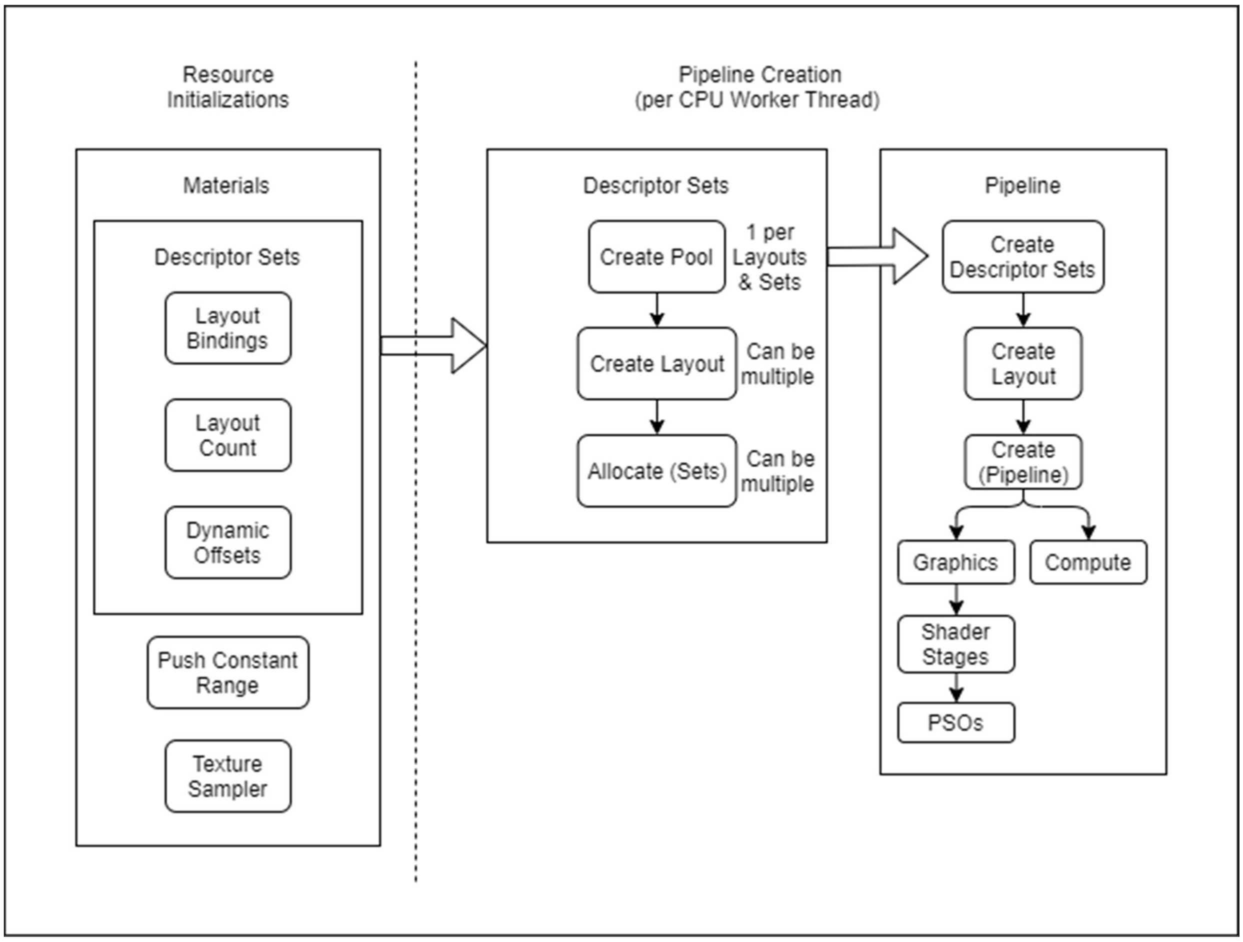


Figure 4-5 – Vulkan Pipeline Architecture

Pipelines use these materials to carry over their required data throughout the entire lifecycle of the Vulkan application and execute their command buffers and draw calls by these data containers. As materials need to expose data instantly to the Vulkan functions, the best approach is to make them as simple structs serving a similar purpose to PODs (though with constructors that are explicitly required to create shaders and textures).

A Material struct has these main following properties:

- Shaders (Vertex, Fragment, Compute)
- Texture
- Descriptors
- Push Constants
- PSOs
- Shader Stage Infos
- Pipeline Layout
- Pipeline
- RenderPass

4.1.5.2 SPIR-V Compiler

This functionality uses GLSLang library to compile GLSL Vulkan compatible shaders into SPIR-V bytecodes at runtime. As briefly stated earlier, it is required for the SDF Graph to Compile the generated nodes into shader instructions. The workflow and architecture of this construct is illustrated in Figure 4-6.

GLSLang basically uses the same API as the glslc compiler that is used for command line compilation of shaders to SPIRV bytecodes.

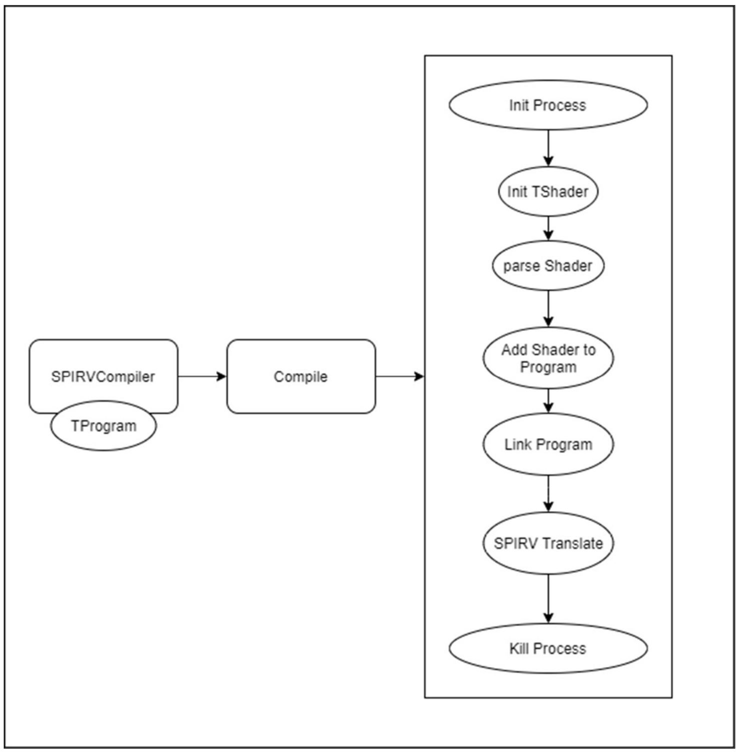


Figure 4-6 – SPIRV Compiler Architecture

4.2 Depth Buffer with SDF Raymarching

To sort the depth between the Raymarched objects and a rasterized mesh object, there must be a mechanism to calculate the depth in one render pass through the model view and project data of the vertex shader and send it over through as a texture sampler to the second pass's fragment shader, where the rasterization and SDF raymarching happen.

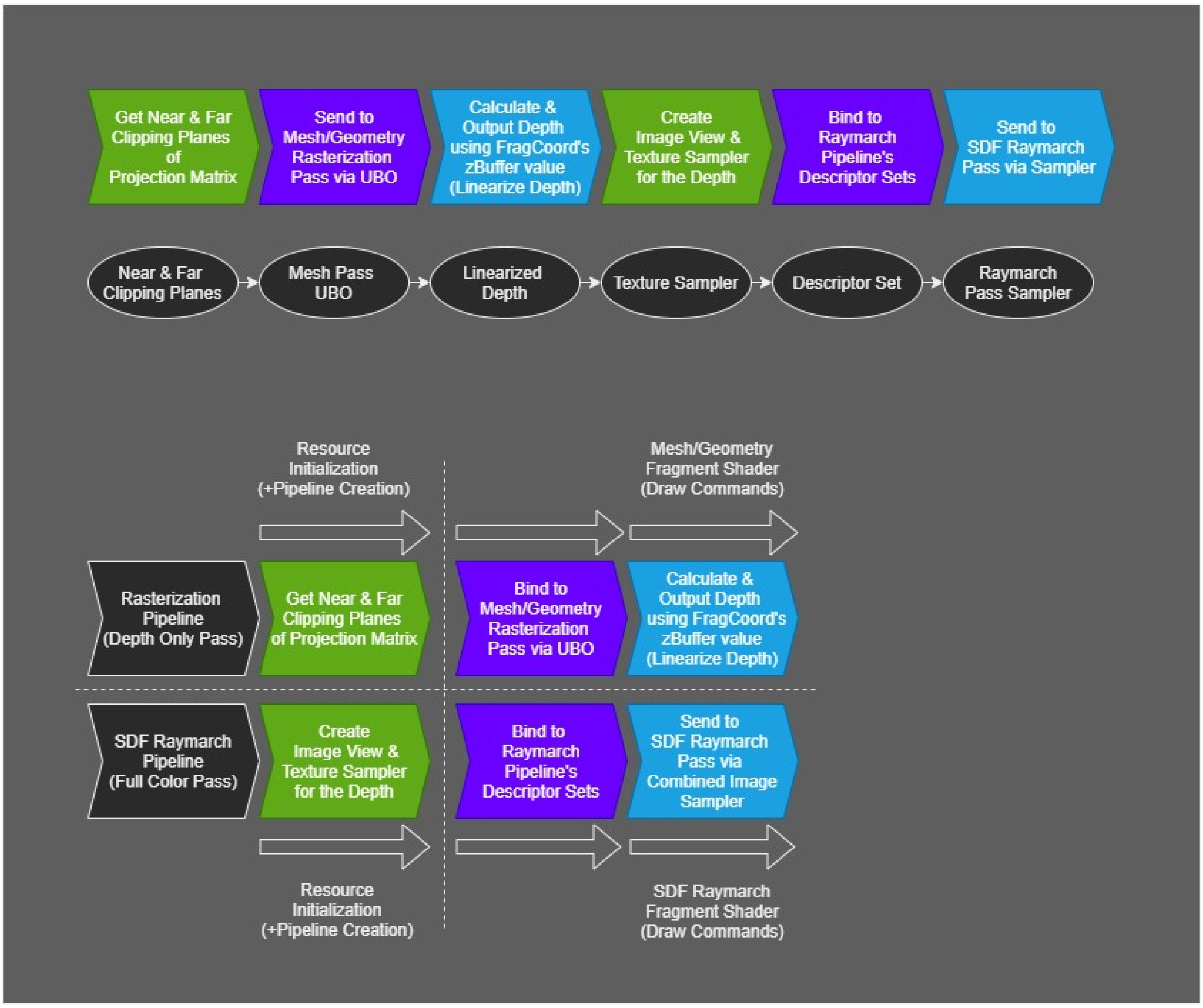

Get Near & Far Clipping Planes of Projection Matrix
Send to Mesh/Geometry Rasterization Pass via UBO
Calculate & Output Depth using FragCoord's zBuffer value (Linearize Depth)
Create Image View & Texture Sampler for the Depth
Bind to Raymarch Pipeline's Descriptor Sets
Send to SDF Raymarch Pass via Sampler
Near & Far Clipping Planes
Mesh Pass UBO
Linearized Depth
Texture Sampler
Descriptor Set
Raymarch Pass Sampler
Resource Initialization (+Pipeline Creation)
Mesh/Geometry Fragment Shader (Draw Commands)
Rasterization Pipeline (Depth Only Pass)
Get Near & Far Clipping Planes of Projection Matrix
Bind to Mesh/Geometry Rasterization Pass via UBO
Calculate & Output Depth using FragCoord's zBuffer value (Linearize Depth)
SDF Raymarch Pipeline (Full Color Pass)
Create Image View & Texture Sampler for the Depth
Bind to Raymarch Pipeline's Descriptor Sets
Send to SDF Raymarch Pass via Combined Image Sampler
Resource Initialization (+Pipeline Creation)
SDF Raymarch Fragment Shader (Draw Commands)

Figure 4-7 - Depth Texture Buffer Calculation and Multi-pass Transfer Model

This mechanism, illustrated in Figure 4-7, through Vulkan takes a journey through two pipelines. A rasterization Pipeline, known as Depth Only Pass and a SDF Raymarching Pipeline which is basically a full colour pass.

The required data which needed to be initialized and prepared for running the pipelines are first to acquire the Near and Far clipping planes from the projection matrix. And, second to create an image view and a texture sampler to hold the depth data and its consumption.

Once the required resources initialized and the pipelines were created (in this case in parallel), the Command Buffers take over and execute the pipeline. In the buffer generation step the pipeline need to bind to rasterization pass via Uniform Buffer Objects (UBOs) and bind to Raymarching pass Descriptor Sets, so that the data can be applied to the pass accordingly.

In the Draw Calls step the depth buffer Is calculated through the fragment shader's zBuffer and to apply it to the world space, it needs to be linearized by the near and far clipping planes which were acquired by the perspective projection data earlier on. Then in the SDF Raymarching Pass, the Combined Image Sampler or texture Sampler can be sent to the fragment shader and from there on used with the texture coordinates to apply the depth per each ray casting.

The entire process within Vulkan requires a quite in-depth understanding for when and where the resources are needed to be created and consumed and understanding of the correct creation of descriptor sets and layouts to lay out the structure of the buffers to be used later in the draw calls.

This process has been implemented and available within the project. Although the depth texture is available for consumption through the raymarching pass, the calculation of the ray casting with the depth buffer has not been implemented at the time as the scope of the project was becoming larger due to the extensive requirement of handling every little detail manually, as was proposed at the beginning of this thesis. Therefore, a fine line was drawn to limit the scope of the implementation to a minimum proof of concept.

Chapter 5 Conclusion

As a result of this project, a basic proof of concept is provided demonstrating how efficient SDF raymarching can be in real-time using the right tools along with a modelling toolset that enables the user to create multiple objects in the scene while potentially can interact with different elements of the engine (Figure 5-1).

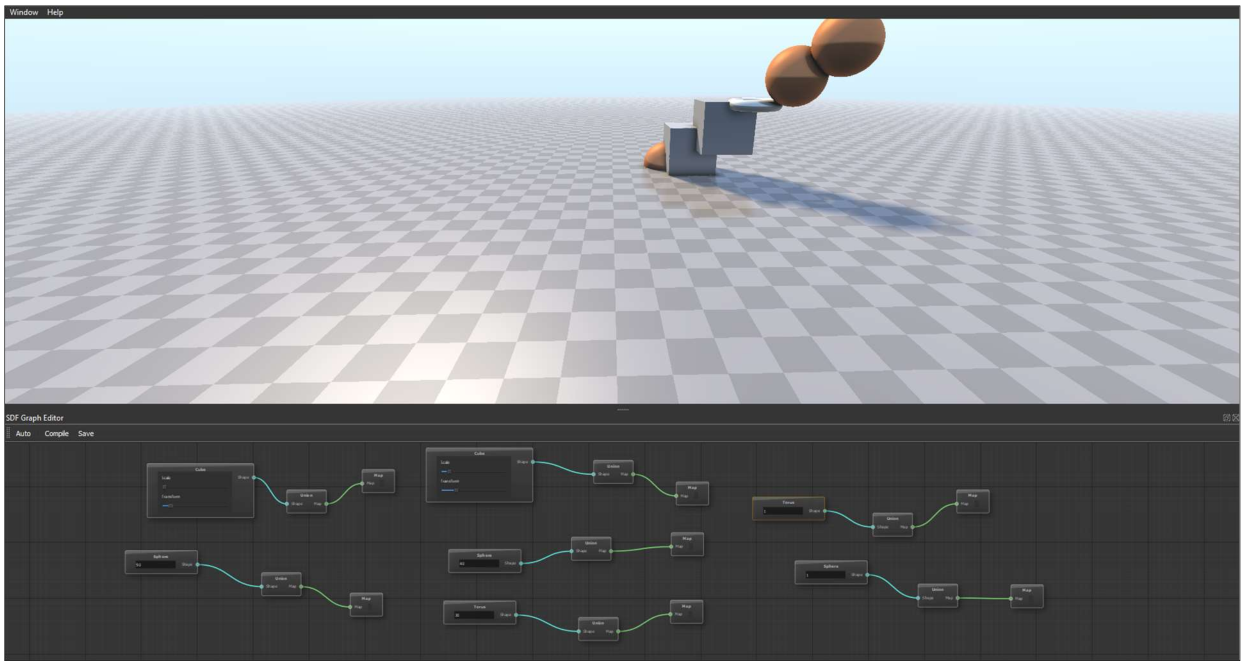


Figure 5-1 – SDFR Project Demo

Also, it is worth noting that, regardless of complexities involved with Vulkan implementation and the learning curve that it might require, the benefits of applying such implementation for complex scenarios such as real-time Ray Tracing and Raymarching are evident outside the current academic context, as the examples of which are provided in the Appendices in this thesis.

It should be noted that despite the complexity of the topic and intricacies of efficiency and performance for a real-time renderer that essentially contribute to what is known as a Game Engine, the author was able to dive deep into the subject area and manage to achieve interesting results, while some are outside the scope of this current project.

As a follow up for this project, there could be further improvements as suggested, in the earlier chapters, in using storages buffer to allow for more efficient CPU to GPU communication and implement more advanced optimization techniques for the SDF raymarching algorithm.

This project can be considered as base for game engine development as it touches the essential parts of a real-time rendering engine. Also, it is hoped that the project could be used as a motivation to adopt the fundamentals of Vulkan or a similar graphics API in the academic context's teaching agenda and their syllabus, encouraging students for better understanding of the graphics pipeline.

Bibliography

Agocs, L., 2017. Vulkan Support in Qt 5.10 - Part 1.

Apple, 2021. *Setting Up a Command Structure* [Available from: https://developer.apple.com/documentation/metal/setting_up_a_command_structure [Accessed 22 August 2021].

Blanco, V., DiGioia, D., Giessen, C. and Miller, A., 2020. Vulkan Guide - Storage Buffers.

de Vries, J., OpenGL.

Glassner, A. S., 1984. Space subdivision for fast ray tracing. *IEEE Computer Graphics and applications*, 4 (10), 15-24.

J'Lali, Y., 2020. *DirectX 12: Performance Comparison Between Single- and Multithreaded Rendering when Culling Multiple Lights* [http://urn.kb.se/resolve?urn=urn:nbn:se:bth-20201]. Student thesis (Independent thesis Basic level (degree of Bachelor)).

Khronos, 2016. *Khronos Releases Vulkan 1.0 Specification* [online]. Khronos Group. Available from: https://www.khronos.org/news/press/khronos-releases-vulkan-1-0-specification [Accessed 22 August 2021].

Lindborg, T., Gifford, P. and Fryazinov, O., 2017. Interactive parameterised heterogeneous 3D modelling with signed distance fields. *ACM SIGGRAPH 2017 Posters*, Los Angeles, California. Association for Computing Machinery. Article 6. Available from: https://doi.org/10.1145/3102163.3102246 [Accessed 22 August 2021]

MacDonald, J. D. and Booth, K. S., 1990. Heuristics for ray tracing using space subdivision. *The Visual Computer*, 6 (3), 153-166.

Malyshau, D. and Ninomiya, K., 2021. WebGPU.

Microsoft, 2018a. StructuredBuffer.

Microsoft, 2018b. What is Direct3D 12?

NVIDIA, 2015. Understanding Structured Buffer Performance.

Overvoorde, A., 2020. *Vulkan Tutorial* [Available from: https://vulkan-tutorial.com/ [Accessed 22 Aug 2021].

Quilez, I., 2019. SDF Bounding Volumes.

Sanchez, M., Fryazinov, O., Fayolle, P. A. and Pasko, A., 2015. Convolution filtering of continuous signed distance fields for polygonal meshes, *Computer Graphics Forum* (Vol. 34, pp. 277-288): Wiley Online Library.

Schütz, M., Mandlburger, G., Otepka, J. and Wimmer, M., 2020. Progressive Real-Time Rendering of One Billion Points Without Hierarchical Acceleration Structures. *Computer Graphics Forum*, 39 (2), 51-64.

Sheng, G. and Nan, M., 2018. Performance, Methods, and Practices of DirectX* 11 Multithreaded Rendering.

Thrane, N. and Simonsen, L. O., 2005. A comparison of acceleration structures for GPU assisted ray tracing.

Unity, 2021. ComputeBuffer.

Willems, S., Zhang, V. and Gil, M., 2021. *Awesome Vulkan* [online]. Available from: https://github.com/vinjn/awesome-vulkan [Accessed 22 August 2021].

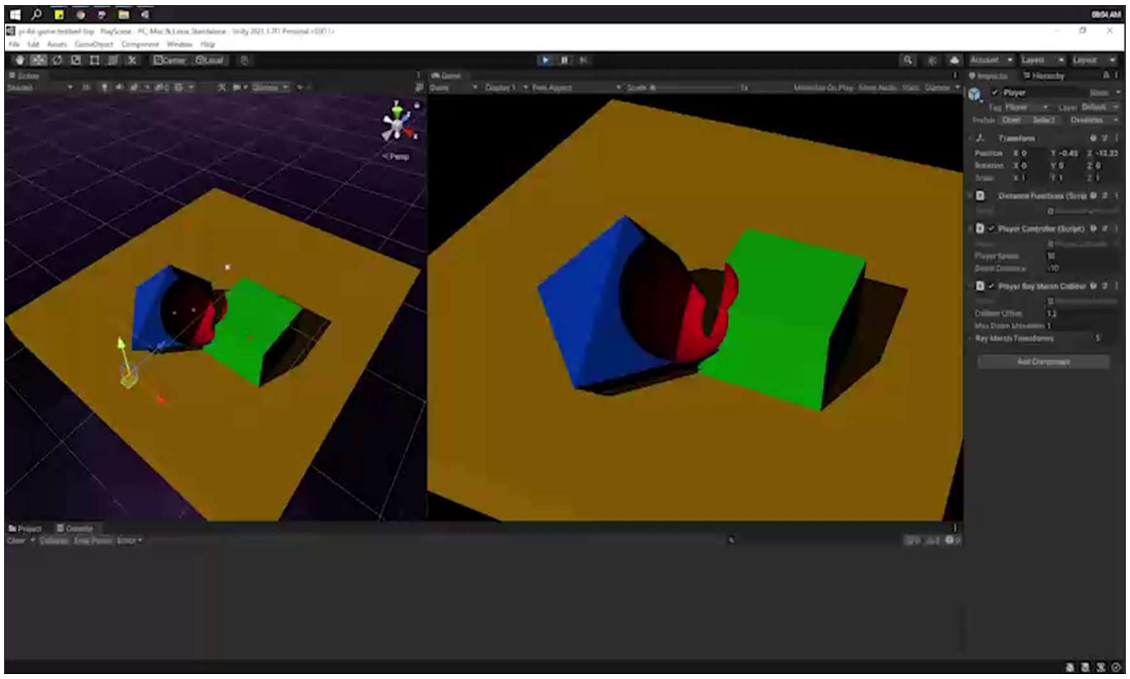
Wodniok, D. and Goesele, M., 2017. Construction of bounding volume hierarchies with SAH cost approximation on temporary subtrees. *Computers & Graphics*, 62, 41-52.

Yazdanpanah, H., 2021. *Real-time 4D Transformations for Games.* University of Bournemouth.

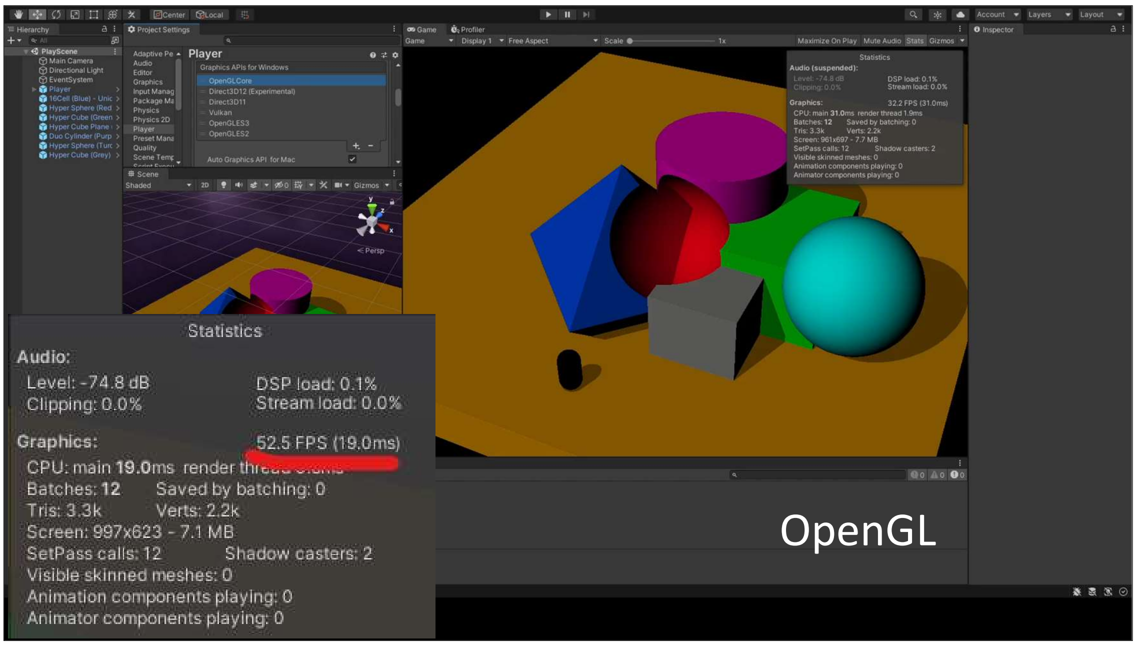
Appendices

Appendix A in-Game Graphics API Comparison with SDFR

Appendix A.1 Direct3D 11 (Personal Inquiry)



Appendix A.2 OpenGL



Project Settings — Player

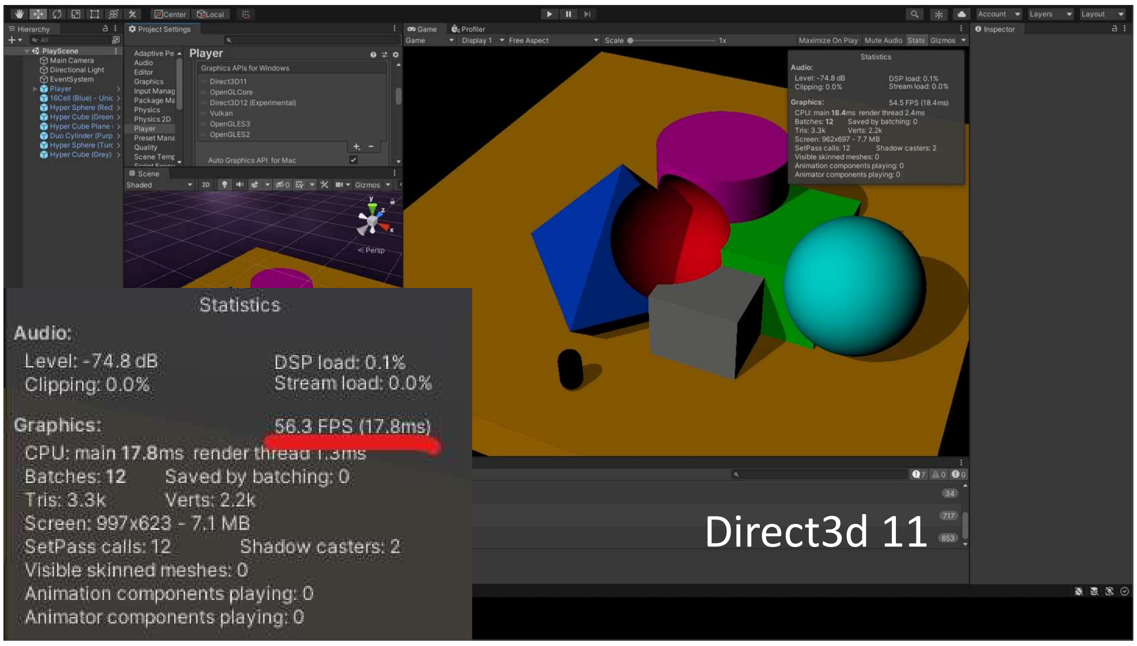
Graphics APIs for Windows
- OpenGLCore
- Direct3D12 (Experimental)
- Direct3D11
- Vulkan
- OpenGLES3
- OpenGLES2

Auto Graphics API for Mac ✓

```
Statistics
Audio (suspended):
  Level: -74.8 dB                 DSP load: 0.1%
  Clipping: 0.0%                  Stream load: 0.0%
Graphics:                         32.2 FPS (31.0ms)
  CPU: main 31.0ms  render thread 1.9ms
  Batches: 12       Saved by batching: 0
  Tris: 3.3k        Verts: 2.2k
  Screen: 961x697 - 7.7 MB
  SetPass calls: 12           Shadow casters: 2
  Visible skinned meshes: 0
  Animation components playing: 0
  Animator components playing: 0
```

```
Statistics
Audio:
  Level: -74.8 dB                 DSP load: 0.1%
  Clipping: 0.0%                  Stream load: 0.0%
Graphics:                         52.5 FPS (19.0ms)
  CPU: main 19.0ms  render threa
  Batches: 12       Saved by batching: 0
  Tris: 3.3k        Verts: 2.2k
  Screen: 997x623 - 7.1 MB
  SetPass calls: 12           Shadow casters: 2
  Visible skinned meshes: 0
  Animation components playing: 0
  Animator components playing: 0
```

Appendix A.3 Direct3D 11

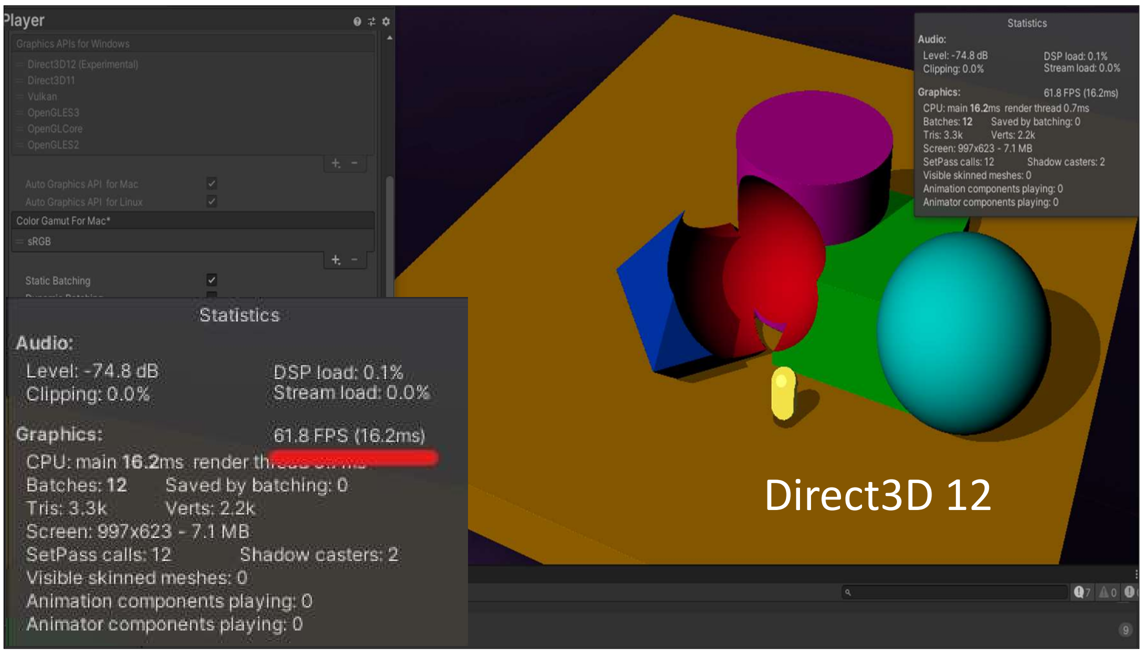


```
Player
Graphics APIs for Windows
Direct3D11
OpenGLCore
Direct3D12 (Experimental)
Vulkan
OpenGLES3
OpenGLES2
Auto Graphics API for Mac ✓
```

```
Statistics
Audio:
  Level: -74.8 dB                DSP load: 0.1%
  Clipping: 0.0%                 Stream load: 0.0%
Graphics:                        54.5 FPS (18.4ms)
  CPU: main 18.4ms  render thread 2.4ms
  Batches: 12        Saved by batching: 0
  Tris: 3.3k         Verts: 2.2k
  Screen: 962x697 - 7.7 MB
  SetPass calls: 12           Shadow casters: 2
  Visible skinned meshes: 0
  Animation components playing: 0
  Animator components playing: 0
```

```
Statistics
Audio:
  Level: -74.8 dB                DSP load: 0.1%
  Clipping: 0.0%                 Stream load: 0.0%
Graphics:                        56.3 FPS (17.8ms)
  CPU: main 17.8ms  render thread 1.3ms
  Batches: 12        Saved by batching: 0
  Tris: 3.3k         Verts: 2.2k
  Screen: 997x623 - 7.1 MB
  SetPass calls: 12           Shadow casters: 2
  Visible skinned meshes: 0
  Animation components playing: 0
  Animator components playing: 0
```

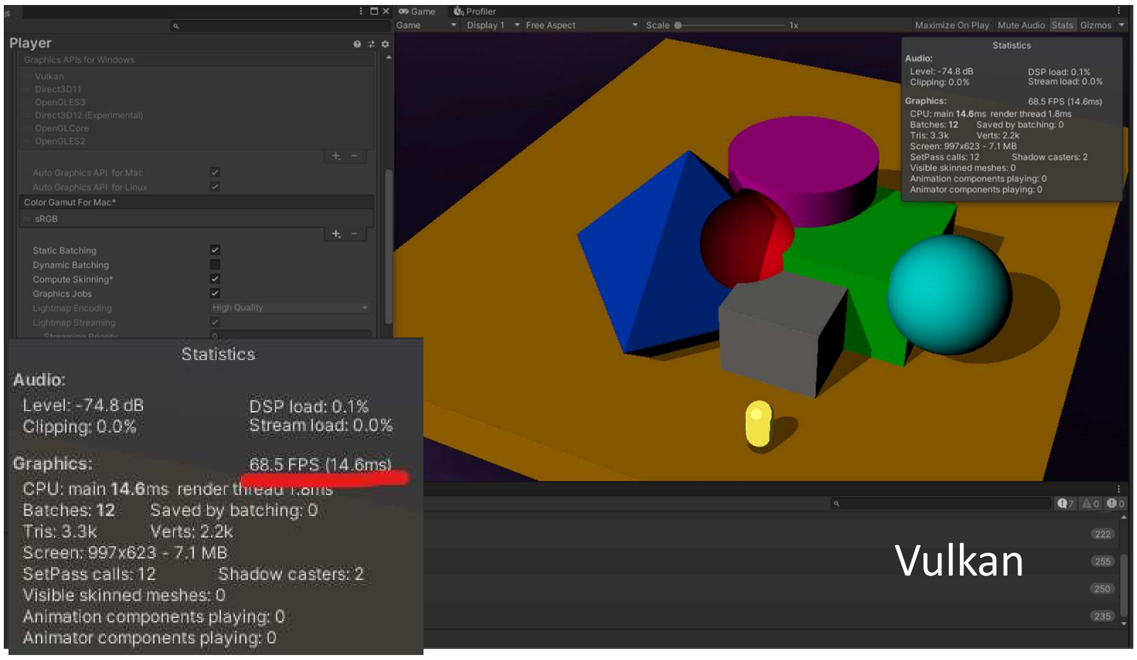
Appendix A.4 Direct3D 12



```
Player
Graphics APIs for Windows
Direct3D12 (Experimental)
Direct3D11
Vulkan
OpenGLES3
OpenGLCore
OpenGLES2
Auto Graphics API for Mac ✓
Auto Graphics API for Linux ✓
Color Gamut For Mac*
sRGB
Static Batching ✓
```

```
Statistics
Audio:
  Level: -74.8 dB                DSP load: 0.1%
  Clipping: 0.0%                 Stream load: 0.0%
Graphics:                        61.8 FPS (16.2ms)
  CPU: main 16.2ms  render thread 0.7ms
  Batches: 12        Saved by batching: 0
  Tris: 3.3k         Verts: 2.2k
  Screen: 997x623 - 7.1 MB
  SetPass calls: 12           Shadow casters: 2
  Visible skinned meshes: 0
  Animation components playing: 0
  Animator components playing: 0
```

Appendix A.5 Vulkan



```
Player
Graphics APIs for Windows
  Vulkan
  Direct3D11
  OpenGLES3
  Direct3D12 (Experimental)
  OpenGLCore
  OpenGLES2
Auto Graphics API for Mac      ✓
Auto Graphics API for Linux    ✓
Color Gamut For Mac*
  sRGB
Static Batching                ✓
Dynamic Batching
Compute Skinning*              ✓
Graphics Jobs                  ✓
Lightmap Encoding              High Quality
Lightmap Streaming             ✓
```

```
Statistics
Audio:
  Level: -74.8 dB                 DSP load: 0.1%
  Clipping: 0.0%                  Stream load: 0.0%
Graphics:                         68.5 FPS (14.6ms)
  CPU: main 14.6ms  render thread 1.8ms
  Batches: 12       Saved by batching: 0
  Tris: 3.3k        Verts: 2.2k
  Screen: 997x623 - 7.1 MB
  SetPass calls: 12         Shadow casters: 2
  Visible skinned meshes: 0
  Animation components playing: 0
  Animator components playing: 0
```

Appendix B List of SDF-based Implementations

Name	Type	Year Introduced/Published
Dreams	Game/Game Creation System	2015-2020
4D Explorer	Game (Platformer/Puzzle)	2020
MarbleMarcher	Game (Fractal Racing)	2019
Claybook	Game (Puzzle)	2017

Appendix C Non-exhaustive List of Vulkan-based Engines

Name	Type	Year Vulkan Supported
Unreal Engine 4+	Game Engine	February 2016
Unity Engine 5.6+	Game Engine	
CryEngine	Game Engine	
Source 2	Game Engine	
Valheim	Game (Sandbox/Survival)	February 2021
Star Citizen	Game	
Dota 2	Game	
Ashes of the Singularity: Escalation	Game (RTS)	Next Version
Ballistic Overkill	Game (FPS)	May 2017
Doom	Game (FPS)	July 2016
Doom 3	Game (FPS)	August 2017
F1 2017	Game (Racing)	-
Mad Max	Game (Action-Adventure)	-
Quake	Game (FPS)	July 2016
Quake III Arena	Game (FPS)	May 2017
Counter-Strike: Global Offensive	Game (FPS)	Will transfer to Vulkan (Source 2)
Rise of the Tomb Rider	Game (Adventure)	November 2017
Serious Sam VR: The Last Hope	Game (FPS)	2017
The Talos Principle	Game (Puzzle)	-
Total Ware Saga: Thrones of Britannia	Game (RTS)	2018
Warhammer 40,000: Dawn of War III	Game (RTS)	2017
Wolfenstein II The New Colossus	Game (Action)	2017