Sheffield Hallam University

Faculty of Science, Technology and Arts

Department of Computing Project (Technical Computing) [55-604708] 2019/20

Author:	Jorge Virgos Castejón
Student ID:	29021655
Year Submitted:	2020
Supervisor:	Aaron MacDougall
Second Marker:	
Degree Course:	BSc Computer Science for Games
Title of Project:	Photorealistic lighting and effects study on a Vulkan 3D engine

Confidentiality Required?

NO X

YES ☐

Contents

1. Introduction 4
1.1 Overview 4
1.2 Aims and Objectives 4
2. Research 5
2.1. Vulkan API implementation 5
2.1.1 Rendering Pipelines 6
2.1.2 Data Oriented Design 6
2.1.3 Entity Component System 6
2.1.4 Engine Agnosticism 7
2.2 Geometrical Techniques 8
2.2.1 Skybox 8
2.2.2 Water Simulation 8
2.3 Lighting Effects 9
2.3.1 Real Time Global Illumination 9
2.3.2 Color Spaces 9
2.3.3 Shadows 9
2.3.4 HDR and Real Time Tone Mapping Algorithms 10
3. Development 11
3.1 Engine Class Diagram and Analysis 11
3.2 Initial Implementation of the Vulkan API 12
3.2.1 Factorization of tutorial code into Render Context 12
3.3 GPU Resources 13
3.3.1 Buffers 13
3.3.2 Textures 14
3.3.3 Materials 14
3.4 Shader Techniques 14
3.4.1 Lighting 14
3.4.2 Shadow mapping 15
3.4.3 Render to Texture 15
3.4.5 Skybox 15
3.4.6 Gerstner Waves 16
3.5 Templated Entity Component System 16
Managers 17
Hierarchy context 17
3.6 Overview of the Rendering Process 17

3.6.1 Depth Pass 17
3.6.2 Scene Pass 17
3.6.3 Quad Pass 18
3.7 User Interface 18
4. Critical Evaluation 19
Project's Success 19
Project Improvements 19
5. References 20
6. Appendix A: Project Specification 21
7. Appendix B: Term Glossary 30
8. Appendix C: Engine Class Diagram 31

1. Introduction

1.1 Overview

This project aims are double pronged: first to research and understand big-budget games (commonly referred as AAA) graphical pipelines and photorealistic effects and techniques currently used; the second is to research and implement 3D real time engine in the Vulkan graphical API.

This report is structured as a reflection of the dichotomy of the project's premise, firstly focusing on the research done to get the resources and knowledge needed to implement the Vulkan API and the research done on the different graphical pipelines and effects, and secondly explaining the implementation achieved, both of the engine's structure and the graphic techniques. Thirdly, there will be a discussion on the overall challenges and results of the project, along with hypothetical improvements and prospects on the engine's development.

The research done on the engine is mostly secondary sources focused on the practical implementation of the engine, with additional chapters included to discuss common practises and programming patterns to take a more holistic and complete approach to the task of creating a modern engine. The graphical techniques and effects research are a mix of secondary sources discussing implementation of previous research and primary sources discussing novel algorithms and advancements on previous techniques.

Implementation of the engine will be accompanied by code snippets and graphs to better understand its final internal structure. Discarded ideas and challenges will also be discussed in these chapters.

1.2 Aims and Objectives

The aims of the projects then are:

1. Research the Vulkan API and the AAA games pipelines and graphical techniques implementations.
2. Development of a real time Vulkan 3D engine implementing the results of the research.

2. Research

2.1. Vulkan API implementation

The research on the graphical API Vulkan was focused on obtaining practical knowledge key to getting the environment started quickly, while simultaneously gaining knowledge on the underlaying structure needed to implement abstraction. The most straightforward practical resource I found is vulkan-tutorial.com by Alexander Overvoorde (2020) which is constantly updated and puts forward the necessary code to show a triangle on screen with C++17. This was complemented with techniques described in Vulkan Cookbook by Pawel Lapinski (2017) which combine code snippets with in-depth explanations of the API processes and Learning Vulkan by Parminder Singh (2016) a more theoretical and thoroughly detailed approach to the API.

Vulkan's main characteristics are:

- Verbosity – the API's language can and is often overtly explicit about behaviours and states on every step. As compared with a traditionally more under the hood graphical API like OpenGL, Vulkan aims to give the programmer freedom to tweak and define the app's behaviour to the tiniest detail. This results in an acute curve of difficulty starting out and demands more knowledge from the programmer in mid to late stage of development.

- Extensions – Vulkan has been engineered to have a core that works stripped of even the most basic form of debugging, in order to function at its most performant once the app is released. Debugging and any additional behaviours needed are added to the app via extensions. These usually address debugging, such as error handling and tracing, but they can have more complex processes, like Google's multithread extension.

- Explicit creation and destruction - Vulkan's objects must be kept tracked of by the app, demanding explicit and orderly creation and destruction the resources.

- Creation Info structure – Related to the two previous points, creation and allocation of a Vulkan object or resource virtually always requires an info structure. The structure must be initialised to zero values and has both compulsory and optional variables.

Appendix B is a glossary of Vulkan and graphical terms particulary useful as a reference to the implementation chapters.

2.1.1 Rendering Pipelines

The first rendering technique is the most basic, Forward Rendering. The main characteristic is its relative simplicity to implement on any graphical backend, at the expense of high computational requirements per light, leading to lighting techniques like light baking to support it. The process of forward rendering is simple: in a given scene, the elements are drawn individually, having access to all information of the lights affecting it available on shader. A single final texture is produced.

Deferred Rendering, first described by Michael Deering (1988) is a technique that delays the lighting shading calculations to a later render pass, requiring both a geometry and a lighting pass. First, the objects are rendered without any lighting, storing the resulting data on three different textures: albedo, depth and normal. A secondary rendering pass then takes place, using the scene's complete lighting data and the generated textures to calculate lighting per pixel. This vastly improves performance with multiple dynamic lights, and skips superfluous lighting calculation that Forward Rendering might have had to calculate more than once per pixel.

Finally, the Forward+ technique. This a technique meant to improve Forward Rendering while maintaining its inherent flexibility and per object encapsulation. Described by Takahiro Harada et al. (2012), it adds a light culling stage before the final scene rendering, in order to avoid multiple lighting calculations per pixel.

2.1.2 Data Oriented Design

Data-oriented design is an optimization proposed to oppose the standard Object-oriented design that was common in game programming. Object-oriented design is straightforward in that it follows human logic: any defined object has characteristics, which can be modified and accessed as a part of the object. However, in a macro scale, this type of data organization means losing performance due to stack creation and cache hits: if every object is allocated on memory individually, updating them means calling a method, creating a virtual stack in the app, and requesting the data stored in a random RAM block to the cache.

Data oriented design strives to resolve these issues by abstracting the object's characteristics onto mass-allocated, continuous data blocks, and having the object defer to an element of many. When updating all these elements, a "manager" class could be used to modify the data directly, and even parallelising it. While this design comes with caveats once its most popular incarnation -- an entity-component system -- is implemented, it clearly outperforms object-oriented design on a large scale setting.

2.1.3 Entity Component System

The first introduction of Entity Component System (referred from now on as ECS) in game development is by Adam Martin (2007). It is famously at the core of Unity, a C# engine that has garnered praise and popularity in the last decade. The basic principle of ECS is based on two concepts: an Entity, empty but for basic information, and the only object permitted to be created on its own; and the Component, an object carrying data and functionality, but only created or attachable to an Entity.

Contrasting with the Object Oriented paradigm: a player, derived from a class Object, will have methods for moving, taking and inflicting damage – whereas the ECS permits abstracting these methods onto components, reusing code implementation: a player would be an entity with the movement, take damage and inflict damage component, while a moving obstacle entity would only need the movement component.

ECS is also highly compatible with data-oriented design, since all components can be created, updated and destroying consecutively, avoiding cache memory hits and permitting multithreading.

2.1.4 Engine Agnosticism

An engine is agnostic once the end user of the engine or API is unaware of the inner workings of the app, interfacing with functions that decouple from the base layer of communication with the hardware. This ensures the code to be portable, consistent through major engine changes and opens the door to multiple backends depending on compatibility or performance.

The Vulkan engine implemented follows these principles, not by necessity (since its only planned graphical backend is Vulkan) but to create clean and understandable code used for the end user.

2.2 Geometrical Techniques

2.2.1 Skybox

The skybox or cubemap technique has been around for a long time, first proposed by Ned Greene (1986). It creates a cube with six textures that is projected onto the limit view frustrum. This creates a static background to the scene and is widely adopted in games as far more performant option than real time generation of a sky and creation of geometry that is not going to be seen in any detail. Graphical API have streamlined the process to create texture arrays and sampling them in shaders.

2.2.2 Water Simulation

There has been a surprising amount of research on realistic water simulation. While many advanced techniques focus on the most realistic modelling of water, they are usually locked to offline rendering and are outside the scope of this project. Recently, previously offline techniques have been proven to work in a real time rendering setting, mainly Fast Fourier Transformation or FFT techniques, albeit at a significant performance hit. Two water simulation techniques are going to be discussed: Gerstner Waves and a recent adaptation of FFT.

The computational Gerstner Wave described by Fournier and Reeves (1968) approximates the physical behaviour of waves as a sum of multiple sine-like waves. The constructive and destructive interference between waves create chaotic looking but repeating wave patterns with far sharper crests that a pure sinus wave. As a caveat, the algorithm to create Gerstner waves can create a loop on the crest if not limited correctly, losing any resemblance to realistic water simulation in the process.

The FFT based implementation of waves described by Jerry Tessendorf (2004) implements an equation created by oceanographic statistical analysis. It is also a sum of sinus-like waves, but with a much greater number of them. FFT is a mathematical technique that evaluates the sum of these waves in a shorter timeframe. The equation is based on physical properties, and requires a wind speed, direction and gravity.

2.3 Lighting Effects

2.3.1 Real Time Global Illumination

On the subject of lighting models, the Blinn-Phong model described by James F. Blinn (1977) has been a staple on graphical engines ever since it became feasible in real time. It describes a derivation on the Phong lighting model by Bui Tuong Phong (1975), one of the first models of realistic light reflection. These models require a vertex normal, a light direction and a camera direction to be implemented. Binn-Phong additionally uses a halfway vector, calculated by normalizing the sum of the light and camera directions. These were models aimed at the recreation of specular lighting

2.3.2 Color Spaces

Two methods exist to display digital colour on 32 bit numbers, RGB and HSV. RBG is by far the most used representation method, dividing colour onto three components that mimic human eyes cone receptors. HSV, on the other hand, is specialized on image editing, and divides colour in three distinct variables: hue, saturation and value: hue describes colour, saturation describes its intensity and value (more commonly known as brightness) describing the combination of the colour with a black to white spectrum.

HSV representation and its variations are widely used on science and lighting model algorithms, since it more accurately describes human's perception and its three values are inherent properties of colour.

2.3.3 Shadows

Shadow recreation has been a contentious issue among game developers for a long time, and it is still not a solved issue within the industry. Shadows are subject to many graphical artifacts and difficult or performance intensive implementations.

The first and most widely used technique is shadow mapping. First described by Lance Williams (1978), shadow mapping renders a scene with only depth, meaning storing of object's distance to the camera in texture values ranging from 0.0 to 1.0, from the perspective of the camera. This generates a texture used to determine whether any object's fragment is in shadow or not, since the distance value sampled in the texture should be equal or less to the current fragment's distance to the light.

While a single texture is only useful for spotlights or directional lights, there are techniques to generate six shadow textures with less passes than six. One technique is even capable of reconstructing a full 360 degree space with two shadow textures, at the cost of Texel precision.

The second technique is shadow geometry, where shadow volumes are created by projecting vertices through the light's direction. These volumes are then rendered with the scene but affect only to the stencil value, an optional eight bits of memory in most depth framebuffers, such that the value increases one when entering a shadow volume, and decreases one when exiting. If the final stencil value is more than one, the fragment is rendered as shadowed. This technique's computational expense is directly proportional to the number of vertices in the scene, and so it has been abandoned in modern game development, with ever rising triangle counts.

2.3.4 HDR and Real Time Tone Mapping Algorithms

High Dynamic Range (HDR) is a photographical technique: the photographer will take multiple shots of the same scene at different exposures, and then combining the clearest and darkest colours in a process called tonemapping to create a final image with greater clarity in all parts of the scene.

Computational HDR does not require multiple shots, but a careful consideration of the light colour values emitted by the scene's lights. If the magnitudes of these colours exceed one, they can be stored in floating point values and then tonemapped into Low Dynamic Range from zero to one, using simple algorithms to increase colour depth throughout the scene. This float point value system works well with the previously described HSV colour display, since the Value is directly related to the light's magnitude.

Adaptive tonemapping takes this a step further and, by calculating a global brightness value of the scene, it can change the resulting coluor balance across time, approximating real physical phenomena like steady eye adaptation to contrast light environments.

3. Development

3.1 Engine Class Diagram and Analysis

Appendix C is the full engine class diagram, showing the different classes on the engine and their relationship to each other. As an introduction to the implementation chapters, a short description of each class will follow.

The three pillars of the engine can be found on the center of the diagram: the RenderContext, a class tasked with everything related to graphic rendering and the Vulkan API; the Window, the class tasked with input and window creation through GLFW; and the HierarchyContext, containing everything related to the entity-component system that carry all the data objects need to be positioned in space and rendered. The inner workings of each of these classes will be discussed at length in the coming chapters.

There are two top-level classes meant to have the least dependencies possible to be included in nearly every other class: platform types, a series of redefinition of data types to ensure the stability of their sizes when used in the code; and constants, a collection of enumerations and structures that must be used in so many places that it’s easier if they are globally available.

The camera is a reused class from previous projects, and only minor changes have been implemented to function in Vulkan. It is capable of calculating view and projection matrixes if provided with a delta time, and its input was already compatible with GLFW. Since its development predates this project, the class will not be detail further.

The Base Resource and its derivate classes – Material, Texture and Buffer – are divided into a structure and an Internal version of the resource. The idea behind this separation is that the end user will only interact with the base resource, containing a simple handle, and only accesses the information requesting it from the RenderContext. This keeps all resource data in continuous memory blocks, in compliance with data-oriented design.

The Base Component is very similar to the Internal resources from RenderContext, albeit a level of abstraction above since all of its derivates – Transform and Render – contain Buffers, Textures and Materials. These components are stored and accessed by Managers, which act as a Pool of the corresponding derivative components and can apply actions to all components at once. Moreover, the managers are templated, meaning that they adapt to be created with any number of component types. These managers are then stored by the HierarchyContext.

Entity is the counterpart to the Component classes. The components are added to the entity, which stores them in a list, and provide the entity with any number of characteristics. The HierarchyContext contains a pool of entities.

Geometry Primitives is a collection of static functions that modify components of existing entities to create geometrical primitives like cubes, spheres, quads or heightmaps.

3.2 Initial Implementation of the Vulkan API

The first step on the implementation of this engine was the completion of the Vulkan Tutorial in order to understand the basics. Since the engine naturally grew from this stage, many of the structures and overall configuration of the basic Vulkan objects – Instance, Extensions, Devices – are still very similar to those provided in the tutorial's code.

The initial implementation key feature is the dichotomy between object creation methods meant to be called once in the initialization stage of the engine, and helper methods abstracted to be used multiple times. The creation of the first type of methods is not strictly necessary, but it lends itself very useful to the tutorial format, dividing the initialization process into discrete steps, and foments a deeper understanding of the overall requirements.

Objects created in this first stage were stored as global variables, with no advanced containers such as smart pointers, class vectors, or pools. The storage of Vulkan's resources is very important, since an explicit destruction is required to avoid VRAM memory leaks.

The engine's window and required Vulkan Surface were managed by GLFW, which has not changed in the final deliverable.

At the end of this first implementation, the overall initialization process of the engine was clear, a basic monochrome triangle showing on screen. Additionally, the destruction of all Vulkan resources was managed correctly, and minimization and resizing was supported.

3.2.1 Factorization of tutorial code into Render Context

As previously discussed, the engine at this stage was lacking an abstraction layer, necessary to provide an agnostic class to be used by the end user. I implemented a pseudo singleton pattern class with RenderContext: while it is a class containing all rendering processes and methods, it is not a global variable, and multiple render contexts could be technically possible in a single app instance. This means that all objects that need to use RenderContext require, at some point in their lifetime, to be provided the RenderContext object itself, usually by pointer. This also makes it obvious whether the RenderContext is being accessed, which would be hidden if it was a global variable.

3.3 GPU Resources

Continuing with the development process, a series of classes were needed to encapsulate individual GPU resources, providing an agnostic method of creation and modification. The first step as the creation of the Base Resource and Internal Base Resource classes.

To understand the implementation of these classes, knowledge of the Pool programming pattern is required: Pools take care of creation, assignation and recycling of limited resources. They first allocate a series of objects, and when requested, provide an unused one. Once an object is stopped being used, the object can be reused. Pools avoid the overhead of spontaneous memory allocation, and keep the data continuous and reusable. Pools can also expand automatically when the number of elements used reaches the limit.

Internal Base Resource is the class that all other internal resource classes derive from, and is meant to contain data and methods that are capable of creation, allocation and destruction of Vulkan resources. Its counterpart, the Base Resources class, contains only a handle -- a number corresponding to the position of the element inside the pool.

Pools of each resource are allocated in RenderContext, and the user uses a templated method, GetResource<>, to get an unused element of the corresponding Pool. However, the user only gets the handle Base Resource, keeping the data inside the RenderContext at all times. The user then can access the internal counterpart of the resource with the method GetInternalRsc<>, this time returning reference to the actual object inside the pool.

3.3.1 Buffers

The buffer class implemented derives from Base Internal Resource, and has methods to initialize, reset, upload data and get the buffers characteristics. The buffer types supported are: Vertex Buffer, storing a number of VertexData objects to provide per vertex info in the shader; Index Buffer, storing a series of unsigned shorts indicating which vertices to draw; the staging buffer, a buffer used to upload buffer and image data to their optimal data in the GPU, not accessible by the CPU; Uniform Buffer, storing data provided to the shaders independent from vertex information.

Creation of buffers is simple: size of an individual item, number of items to be updated and the target buffer type are provided. A second method is used to upload the data and can be used multiple times. The behaviour inside the upload process is different depending on the type: while the vertex and index buffers require their info to first be uploaded to RenderContext’s staging buffer, the uniform buffer memory is uploaded directly – since its data needs to be accessible from CPU.

3.3.2 Textures

The texture class is more complex. An internal load method takes care of loading disk data using stb image library. The texture can also be created directly from memory. Two initialization methods are provided. Additionally, the texture class has two static methods: one to load disk data to a block of memory, and a second to reset the default sampler, stored as a static variable of the class.

The Internal Texture class creates the required VkImage and VkImageView. It also stores the image's characteristics, like size, format and type. The texture type supported are: Sampler, an image to be used for sampling in shaders; Color Attachment, the type of swaphcain images that can be shown on-screen; Depth attachment, an image that tracks the depth values of objects during rendering; and Cubemap, six textures in one that create a background for rendering.

The default VkSampler required to use textures in shader is created as a static variable, and is used by every texture in case an additional one is not created.

3.3.3 Materials

The material class creates the pipeline that the object is going to be render through. It has more options than any other resource class, and the initialization method requires an additional Material Info structure to be provided. The material also needs to be provided with the texture that the object is going to be rendered with. Overall, this is the class that is responsible for how the rendered object is going to look.

The material info options are: shader names, loaded from disk in the Resources/Shaders folder; viewport, determining the size of the window that the object is going to be rendered on; scissor, which culls the object's vertices to avoid rendering on the provided space; culling mode, which controls what side the triangles rendered must be culled, back, front or none; pipeline type, determining on which rendering pass the object is going to be rendered in.

The material creates the specified VkPipelineLayout and VkPipeline. The material stores the Material Info provided, and has a method that recreates the VkPipeline object to support real time tweaking of shaders.

3.4 Shader Techniques

3.4.1 Lighting

The first technique that was implemented was Blinn-Phong lighting. This lighting model produces great results with specular lighting, as is straightforward to implement on a global directional illumination system. Three components of light intensity have to be determined: ambient, diffuse and specular.

Ambient lighting is a base light intensity. It is easy to think about as background lighting affecting equally to all rendered objects.

The Diffuse component is calculated with the result of a dot product between the fragment's normal and the light direction vector. Since the system uses a single directional light, the light direction is static and is available in the shader as a normalized uniform vector.

Specular light in Blinn-Phong requires two more variables: a shininess exponent that will limit the radius of effect of the resulting light and a halfway direction between the camera's vector to the fragment and the light's direction. This halfway vector is then used in a dot product against the fragment's normal, raising the result to the shininess exponent. This creates a column of bright light along the light's direction with a sharp decrease in intensity.

These three light intensity components are then added together and multiplied by the light's colour and the texture colour to create the final lighting result.

3.4.2 Shadow mapping

The implemented version of shadow mapping is based on a global directional light illumination system, affecting every object on the scene. Since the light is directional, the view frustum to render the scene's depth values from the light's perspective is orthogonal, and its near and far values are adjusted to give precision and avoid z-fighting. The result is an 8192X8192 texture that is passed, along the lighting transformation matrix, to every object in the scene that needs to determine shading.

To test whether any given fragment should be shadowed, the vertex position is multiplied by the object's model, the light's view and projection matrices and a bias matrix, which centres values to the middle of the frustrum. This position is then normalized and used to sample the shadow texture with the horizontal and vertical components. The depth component of the position is then compared to the texture value, and if it is lesser, the fragment is shadowed is stripped of diffuse and specular lighting.

3.4.3 Render to Texture

Rendering to texture is essential to apply postprocessing to the image. At this stage in implementation, any number of rendering passes could be added to the engine. Two changes were necessary: to defer the results of the scene's rendering to a secondary texture instead of storing the results on the final swapchain image; and to create a shader that could take the image, apply postprocessing, and render it at fullscreen.

Firstly, a second render pass was created, storing the results on a previously allocated texture object corresping in size and and format to the swapchain image. This means that every time the window is resized, this image must be recreated too. This changes little about the implemented rendering process. The first render pass was unchanged, targeting the swapchain image rendering a single object: a full screen quad.

To render this fullscreen quad, the process described by Sascha Willems (2016) was implemented: a technique to create a full screen spanning quad to render the scene's texture exclusively on shader, avoiding use of unnecessary resources.

3.4.5 Skybox

The skybox technique uses the vertex position to sample a samplerCube texture on the fragment shader. The position needed is stripped from translation, in order to create the illusion of an infinitely far plane that still rotates relative to the camera. The rendition of the skybox also draws a dynamic sun based on the light's direction and color.

3.4.6 Gerstner Waves
In order to create Gerstner Waves, the position and normal of vertices have to be recalculated in separate shader methods. The current implementation is a sum of four waves, changing in amplitude, frequency, and direction. A final factor shared by all four waves controls the “sharpness” of the resulting wave, ranging from a pure sinus wave at 0 to a mirrored exponential wave at 1. The waves are dependant on time to move in real-time.

The final position is a result of the wave’s amplitude multiplied by a cosinus in horizontal and depth positions and sinus in vertical position. The values inside these are the frequency of the wave times the result of the dot product between the wave’s direction and the vertex’s position, plus the phase value of the wave times time to move the waves in real time. Each wave has a positional value, and are added together to generate the final position. Normal follows a similar calculation but changes the dot product to be between the wave’s direction and the final position.

These calculations are per vertex, so the number of calculations is directly proportional to number of vertices in the used height map.

3.5 Templated Entity Component System
The engine follows an entity – component system. The entity can have any number of components added, and carries no information of its own apart from a name. Pointers to the components added are stored in a list for future access. The components carry all the information, and can be specialized in any number of ways. This paradigm lets the end user have any number of entities with component functionality, while preserving the components data-oriented design.

All components derive from the Base Component class, which has only two methods: reset and initialize. These methods are virtual, meaning that they can be replaced by the derived class implementation. Two variables are also present: an idetifier, planned to be used for advanced component searches, and the Boolean in use, to keep track of their usage after the pool has added them to an entity.

The first derived component is Transform Component, carrying transformation data, the local and global matrixes of the object, and storing a std function variable to be called during the component’s update tick (more detail in Managers section). The components give writing access to position, rotation and scale of the object. Additionally, a Buffer handle corresponding to the Internal Buffer created during initialization is stored.

The second derived component is Render Component, tasked with carrying all the data necessary to render the object. This means it carries a vertex buffer handle, an index buffer handle, and a material handle. All three are created during the initialization process, filled with default information. The component has methods to access the Internal counterpart of these variables directly. An extra Recreate Shader method is present to refresh shaders in real time, recreating the material and therefore the pipeline of the object.

Returning to the Entity, the process to add or access a component is templated, accessing the correct component pool through a mapping of type ids with the correct pointer to the pool. Since all components pools can be treated as Base Components, a mapping technique is used to determine the correct pool to add from: when initializing the app, Transform Component and Render Component type ids -- a hash code unique to the C++ type -- are paired and inserted in an std map variable along with a pointer to the corresponding pool. This technique allows any component to be added to the entity, as long as its type is mapped to a valid pool.

Managers

The manager class has been templated to work with any component. It has two purposes: being a pool for any component type, and applying actions to all its components such as Update and Reset. Since the components are in continuous memory vectors, modifying them in a single batch is much more performant. Since components carry different information, each component manager has to have specialized code.

The Transform Component Manager update generates local matrixes on the transformation info already in the component, and calls the std function if it has been assigned one. This method can apply any continuous transformation effect, having access to the delta time between frames.

The Render Component Manager update can recreate shaders on all components and updates the dynamic rendering properties to be equal to the screen's current size.

Hierarchy context

The entity pool and all managers are stored in a pseudo singleton class called Hierarchy Context. Similarly to Render Context, it carries all information pertaining to the engine's object hierarchy. It is not accessible statically, and must be created at the initialization stage of the app. The hierarchy context calls the update methods of all managers, and functions as the entities pool. It also provides all necessary information of the hierarchy User Interface, to be discussed later.

3.6 Overview of the Rendering Process

The engine's rendering process uses three individual passes: a depth only pass, where we create the shadowmapping texture used in the shader technique previously discussed; a complete rendering of the scene stored in a texture; and the rendering of a full screen quad, where we render the second pass texture with postprocessing.

3.6.1 Depth Pass

The first Render Pass on the engine is distinguished by only having a depth attachment, with its format depending on the system's compatibility. Shadowmaps have already been discussed both in research and implementation at this point, so the use of the pass' resulting texture is clear. All objects during this pass use a specific simplified material, which ensures that the correct orthogonal view matrix is displays and increases performance compared to simply reusing the entity's own material. For this, the incompatible skybox geometry had to be excluded, since it would render a cube on scene's centre.

3.6.2 Scene Pass

Originally the first Render Pass, this includes both colour and depth attachment. Every entity with transform and render components is renderer on this pass, utilizing the material's pipeline, uniform data and textures. This is the Render Pass with most draw calls. The results of the rendering are stored on a previously allocated texture corresponding in resolution to the app's window.

3.6.3 Quad Pass

Lastly, a Render Pass with a single draw call is executed, with only a color attachment. Only a custom Material is needed, since the geometric data and postprocessing options are configured through shader and uniform data.

3.7 User Interface

The in-engine user interface (referred as UI from now on) was implemented with Dear Imgui, due to its proven compatibility with both GLFW and Vulkan and its relatively simple process of setting up. The UI at the current deliverable state is made up of three windows:

- Console: A window displaying C++ and API errors. Most error handling on the engine is done with exceptions, so this console is used mainly for unhandled API error callbacks.
- Rendering Options: A small window letting the user tweak global uniform values passed onto the object's rendering shaders. The position of the sun, light's color, debug lighting visualizations and tonemapping exposure are customizable.
- Hierarchy viewer: A list of all entities accounted by the engine. The current number and type of components can also be seen in a secondary column. Intended to be expanded for in-engine entity creation, destruction, parenting and access to the components values.

4. Critical Evaluation

Project's Success

Overall, going back to the project specification, the project's success is very limited. Information on big budget's graphical techniques is hard to come by due to the secretive tendencies of the industry, forcing the research to focus on secondary or overly theoretical sources. This limited the extent of the research, since understanding any given technique required a highly theoretical background.

The development of the second objective, creating a 3D Vulkan engine, was met with a lot of challenges: the process of abstracting an agnostic class for rendering, the framework and boilerplate code needed to create and customize GPU resources, the implementation of a data oriented entity-component system, and the many hours that sank in shaders just to name a few. However, the results adjust much more to the original plan, and it resulted in a functioning and optimized engine base to keep exploring techniques in the future.

While working on this project, the work load was very intermittent, resulting in poor results and overall dissatisfaction with the project's idea as a whole. With better spent time and tempered expectations from the start, results could have drastically improved, which is something to keep in mind for future projects.

Project Improvements

There are many things to improve or expand upon in the engine's current state. The current hierarchy relating Entities to their world transformation is flawed, since children are not supported. Most pools cap at a limit of elements that will not expand after running out. The user interface does not support modification of the scene's entities or their components. Finally, a serialization and storage system should be implemented to aid in testing different prepared scenes.

New ideas and systems directly related to graphics could also be explored. Dynamic lighting with a new component and a deferred rendering pipeline could reveal a lot of insight into modern lighting in games. Adding advanced screen space effects, like SSAO or god-rays, in a modular system that permits individual shaders to be incorporated into the scene. Adaptive tone mapping, which was researched but not implemented, along with changing the colour space to HSV. Shadowmapping techniques that blur the image or distort the view frustrum to add more detail where needed to the shadows could also be an interesting avenue. Every one of these ideas is feasible in the current state of the engine.

5. References

Alexander Overvoorde (2020). *Vulkan Tutorial* - https://vulkan-tutorial.com/

Pawel Lapinski (2017*). Vulkan Cookbook.* http://shop.oreilly.com/product/9781786468154.do

Parminder Singh (2016). *Learning Vulkan.* http://shop.oreilly.com/product/9781786469809.do

Michael Deering (1988). *The triangle processor and normal vector shader: a VLSI system for high performance graphics*. https://dl.acm.org/doi/10.1145/378456.378468

Takahiro Harada et al. (2012) *Forward+: Bringing Deferred Lighting to the Next Level.* https://takahiroharada.files.wordpress.com/2015/04/forward_plus.pdf

Adam Martin (2007). *Entity Systems are the future of MMOG development.* http://t-machine.org/index.php/2007/09/03/entity-systems-are-the-future-of-mmog-development-part-1/

Ned Greene (1986). *Environment mapping and other applications of world projections.* https://ieeexplore.ieee.org/document/4056759

Fournier and Reeves (1968). *A Simple Model of Ocean Waves.* https://dl.acm.org/doi/10.1145/15922.15894

Jerry Tessendorf (2004) *Simulating Ocean Water.* https://www.researchgate.net/publication/264839743

James F. Blinn (1977) *Models of light reflection for computer synthesized pictures.* https://dl.acm.org/doi/10.1145/563858.563893

Bui Tuong Phong (1975). *Illumination for computer generated pictures.* https://users.cs.northwestern.edu/~ago820/cs395/Papers/Phong_1975.pdf

Lance Williams (1978) *Casting curved shadows on curved surfaces*. http://cseweb.ucsd.edu/~ravir/274/15/papers/p270-williams.pdf

Sascha Willems (2016). *Vulkan tutorial on rendering a fullscreen quad without buffers*. https://www.saschawillems.de/blog/2016/08/13/vulkan-tutorial-on-rendering-a-fullscreen-quad-without-buffers/

6. Appendix A: Project Specification

PROJECT SPECIFICATION - Project (Technical Computing) 2019/20

Student:	**Jorge Virgos Castejón**
Date:	**24/10/2019**
Supervisor:	**Aaron MacDougall**
Degree Course:	**BSc Computer Science for Games**
Title of Project:	**Photorealistic lighting and effects study on a Vulkan 3D engine**

Elaboration

The project would be comprised of two macro tasks:

The study of Frostbite's and other AAA engine's implementation of advanced lighting techniques, such as Atmospheric Scattering and Image Based Lighting. This would be an exploration of different implementations of each technique and looking for the most performant and convincing photorealistic one.

The implementation of a 3D Vulkan engine with the selected implementation of each technique. The graphical pipeline itself (Forward, Forward +, Deferred) of the engine would be subject to the previous study, but the setup of the engine would be done parallelly to the study task.

This structure lets the project develop in a clear study / implementation divide, firstly with the techniques study and engine groundwork in parallel and later with the implementation of said techniques.

Project Aims

- Understand the graphical pipeline of in-studio AAA engines.
- Learn the Vulkan API.
- The creation of a functional engine in Vulkan supported by my own research.
- Learn about the lighting techniques and effects to be implemented both in theory and practise.

Project deliverable(s)

A study on the lighting techniques and effects theory used by AAA in-studio engines, analysing how performant and convincing are compared to other implementations. One of the approaches will be chosen for each technique to be implemented in the engine. The results of the effects' implementation will be reflected in the study, supported by images and code from the engine.
A C++ Vulkan based 3D engine, with special focus on the graphical pipeline (forward, forward +, deferred, clustered) and the implementation of the techniques concluded by the study. The engine will be a Windows executable, without sound capabilities or advanced user interfaces. The engine would have, however, a way to change between scenes that demonstrated the different techniques, and a way to change any of the input values that the techniques and effects implementation may have.

Action plan

As I laid out in the project elaboration, I will be dividing the project in two great tasks / deliverables: a study on the implementation of other engines' lighting techniques, and their implementation on my own Vulkan 3D engine. As such, the tasks will be parallel at the start of the project and will converge only once both the study is completed and the engine is in a functional state. This is projected to be just at the end of January.

ENGINE STUDY TASKS	*ESTIMATED DATE*
Determine the number and scope of the techniques and effects to implement.	1/11/2019
Gather development resources of the commercial engines to be studied. At least two implementations of each technique before moving on.	8/11/2019
Analyse the theory and the results behind the implementations of the engines.	20/12/2019
Induct a rough estimate of the processing cost of each implementation and conclude upon an implementation with the best performance / photorealism relation for each technique.	24/1/2020

ENGINE DEVELOPMENT TASKS	*ESTIMATED DATE*
Start a window within a Vulkan environment.	4/11/2019
Setup the resources and memory management in the engine.	15/11/2019
Setup the basic graphical pipeline in the engine.	25/11/2019
Setup geometry and model (obj) loading in the engine.	2/12/2019
Setup texture and texture loading in the engine.	16/12/2019
Setup basic lighting (phong) and hard shadows in the engine.	17/1/2020
Implement a saving and loading or scripting method for the engine.	24/1/2020
Implement the lighting techniques and effects concluded by the research.	20/3/2020

MILESTONE TASKS	*ESTIMATED DATE*
Deliver the Project Specification and ethics form.	25/10/2019
The Information Review.	6/12/2019
The Provisional Contents Page.	21/2/2020
The draft Critical Evaluation.	27/3/2020
Sections for a draft report.	27/3/2020
Submit the body of the project and report to Turnitin.	22/4/2020
Submit the Project Report (physical and electronic) and copies of the deliverable in the report.	23/4/2020
Demonstration of work.	To be agreed with, before 12/5/2020

BCS Code of Conduct

I confirm that I have successfully completed the BCS code of conduct on-line test with a mark of 70% or above. This is a condition of completing the Project (Technical Computing) module.
Signature: [illegible]

Publication of Work

I confirm that I understand the "Guidance on Publication Procedures" as described on the Bb site for the module.
Signature: [illegible]

GDPR

I confirm that I will use the "Participant Information Sheet" as a basis for any survey, questionnaire or participant testing materials. This form is available on the Bb site for the module.
Signature: [illegible]

Ethics

The research ethics checklist has been completed and is found below.

RESEARCH ETHICS CHECKLIST FOR STUDENTS (SHUREC 7)

This form is designed to help students and their supervisors to complete an ethical scrutiny of proposed research. The SHU Research Ethics Policy should be consulted before completing the form.

Answering the questions below will help you decide whether your proposed research requires ethical review by a Designated Research Ethics Working Group.

The final responsibility for ensuring that ethical research practices are followed rests with the supervisor for student research.

Note that students and staff are responsible for making suitable arrangements for keeping data secure and, if relevant, for keeping the identity of participants anonymous. They are also responsible for following SHU guidelines about data encryption and research data management.

The form also enables the University and Faculty to keep a record confirming that research conducted has been subjected to ethical scrutiny.

For student projects, the form may be completed by the student and the supervisor and/or module leader (as applicable). In all cases, it should be counter-signed by the supervisor and/or module leader, and kept as a record showing that ethical scrutiny has occurred. Students should retain a copy for inclusion in their research projects, and staff should keep a copy in the student file.

Please note if it may be necessary to conduct a health and safety risk assessment for the proposed research. Further information can be obtained from the Faculty Safety Co-ordinator.

General Details

Name of student	Jorge Virgos Castejón
SHU email address	b9021655@my.shu.ac.uk
Course or qualification (student)	BSc Computer Science for Games
Name of supervisor	Aaron MacDougall
email address	A.MacDougall@shu.ac.uk
Title of proposed research	Photorealistic lighting and effects study on a Vulkan 3D engine

Proposed start date	24/10/2019
Proposed end date	20/3/2020
Brief outline of research to include, rationale & aims (250-500 words).	The project would be comprised of two macro tasks: The study of Frostbite's and other AAA engine's implementation of advanced lighting techniques, such as Atmospheric Scattering and Image Based Lighting. This would be an exploration of different implementations of each technique and looking for the most performant and convincing photorealistic one. The implementation of a 3D Vulkan engine with the selected implementation of each technique. The graphical pipeline itself (Forward, Forward +, Deferred) of the engine would be subject to the previous study, but the setup of the engine would be done parallelly to the study task. This structure lets the project develop in a clear study / implementation divide, firstly with the techniques study and engine groundwork in parallel and later with the implementation of said techniques. My aims with this project are: • Understand the graphical pipeline of in-studio AAA engines. • Learn the Vulkan API. • The creation of a functional engine in Vulkan supported by my own research. • Learn about the lighting techniques and effects to be implemented both in theory and practise.
Where data is collected from individuals, outline the nature of data, details of anonymisation, storage and disposal procedures if required (250-500 words).	

1. Health Related Research Involving the NHS or Social Care / Community Care or the Criminal Justice Service or with research participants unable to provide informed consent

Question	**Yes/No**
1. Does the research involve? • Patients recruited because of their past or present use of the NHS or Social Care • Relatives/carers of patients recruited because of their past or present use of the NHS or Social Care • Access to data, organs or other bodily material of past or present NHS patients • Foetal material and IVF involving NHS patients • The recently dead in NHS premises • Prisoners or others within the criminal justice system recruited for health-related research* • Police, court officials, prisoners or others within the criminal justice system* • Participants who are unable to provide informed consent due to their	No
2. Is this a research project as opposed to service evaluation or audit? *For NHS definitions please see the following website* http://www.hra.nhs.uk/documents/2013/09/defining-research.pdf	No

If you have answered **YES** to questions **1 & 2** then you **must** seek the appropriate external approvals from the NHS, Social Care or the National Offender Management Service (NOMS) under their independent Research Governance schemes. Further information is provided below.

NHS https://www.myresearchproject.org.uk/Signin.aspx

* All prison projects also need National Offender Management Service (NOMS) Approval and Governor's Approval and may need Ministry of Justice approval. Further guidance at:

http://www.hra.nhs.uk/research-community/applying-for-approvals/national-offender-management-service-noms/

NB FRECs provide Independent Scientific Review for NHS or SC research and initial scrutiny for ethics applications as required for university sponsorship of the research. Applicants can use the NHS pro-forma and submit this initially to their FREC.

2. Research with Human Participants

Question	**Yes/No**
Does the research involve human participants? This includes surveys, questionnaires, observing behaviour etc.	No

Question	Yes/No
1. *Note If YES, then please answer questions 2 to 10* *If NO, please go to Section 3*	
2. Will any of the participants be vulnerable? *Note: Vulnerable' people include children and young people, people with learning disabilities, people who may be limited by age or sickness, etc. See definition on website*	
3. Are drugs, placebos or other substances (e.g. food substances, vitamins) to be administered to the study participants or will the study involve invasive, intrusive or potentially harmful procedures of any kind?	
4. Will tissue samples (including blood) be obtained from participants?	
5. Is pain or more than mild discomfort likely to result from the study?	
6. Will the study involve prolonged or repetitive testing?	
7. Is there any reasonable and foreseeable risk of physical or emotional harm to any of the participants? *Note: Harm may be caused by distressing or intrusive interview questions, uncomfortable procedures involving the participant, invasion of privacy, topics relating to highly personal information, topics relating to illegal activity, etc.*	
8. Will anyone be taking part without giving their informed consent?	
9. Is it covert research? *Note: 'Covert research' refers to research that is conducted without the knowledge of participants.*	
10. Will the research output allow identification of any individual who has not given their express consent to be identified?	

If you answered **YES only** to question **1,** the checklist should be saved and any course procedures for submission followed. If you have answered **YES** to any of the other questions you are **required** to submit a SHUREC8A (or 8B) to the FREC. If you answered **YES** to question **8** and participants cannot provide informed consent due to their incapacity you must obtain the appropriate approvals from the NHS research governance system. Your supervisor will advise.

3. Research in Organisations

Question	Yes/No
1. Will the research involve working with/within an organisation (e.g. school, business, charity, museum, government department, international agency, etc.)?	No
2. If you answered YES to question 1, do you have granted access to conduct the research? *If YES, students please show evidence to your supervisor. PI should retain safely.*	

3. If you answered NO to question 2, is it because: A. you have not yet asked B. you have asked and not yet received an answer C. you have asked and been refused access. *Note: You will only be able to start the research when you have been granted access.*	

4. Research with Products and Artefacts

Question	**Yes/No**
1. Will the research involve working with copyrighted documents, films, broadcasts, photographs, artworks, designs, products, programmes, databases, networks, processes, existing datasets or secure data?	Yes
2. If you answered YES to question 1, are the materials you intend to use in the public domain? *Notes: 'In the public domain' does not mean the same thing as 'publicly accessible'.* • *Information which is 'in the public domain' is no longer protected by copyright (i.e. copyright has either expired or been waived) and can be used without permission.* • *Information which is 'publicly accessible' (e.g. TV broadcasts, websites, artworks, newspapers) is available for anyone to consult/view. It is still protected by copyright even if there is no copyright notice. In UK law, copyright protection is automatic and does not require a copyright statement, although it is always good practice to provide one. It is necessary to check the terms and conditions of use to find out exactly how the material may be reused etc.* *If you answered YES to question 1, be aware that you may need to consider other ethics codes. For example, when conducting Internet research, consult the code of the Association of Internet Researchers; for educational research, consult the Code of Ethics of the British*	No
3. If you answered NO to question 2, do you have explicit permission to use these materials as data? *If YES, please show evidence to your supervisor.*	Yes
4. If you answered NO to question 3, is it because: A. you have not yet asked permission B. you have asked and not yet received and answer C. you have asked and been refused access.	**A/B/C**

Adherence to SHU policy and procedures

Personal statement

I can confirm that: – I have read the Sheffield Hallam University Research Ethics Policy and Procedures	
Student	
Name: Jorge Virgos Castejón	Date: 25/10/2019
Signature: Jorge Virgos	
Supervisor or other person giving ethical sign-off	
I can confirm that completion of this form has not identified the need for ethical approval by the FREC or an NHS, Social Care or other external REC. The research will not commence until any approvals required under Sections 3 & 4 have been received.	
Name:	Date:
Signature:	

7. Appendix B: Term Glossary

Instance -- The first required object is the Vulkan Instance, a pseudo singleton object that stores the states of the API. Desired extensions must be first queried and set in the info structure. This list can contain any extension installed.

Devices –- The API needs objects to define the conditions and characteristics it is working on. These are the VkPhysicalDevice, an object representing the hardware GPU, and the VkDevice, representing the logical actions carried by the app onto the hardware.

Queues – Queues are used to submit actions to be carried out by the API. Queues are specialized in type, but the main ones are Graphics Queue, used to render, and Present Queue, used to submit the final frame to be seen in the app's window. The actions are carried out asynchronously unless indicated. Queues are created along the VkDevice and must be queried to the Physical Device to check whether the hardware supports them.

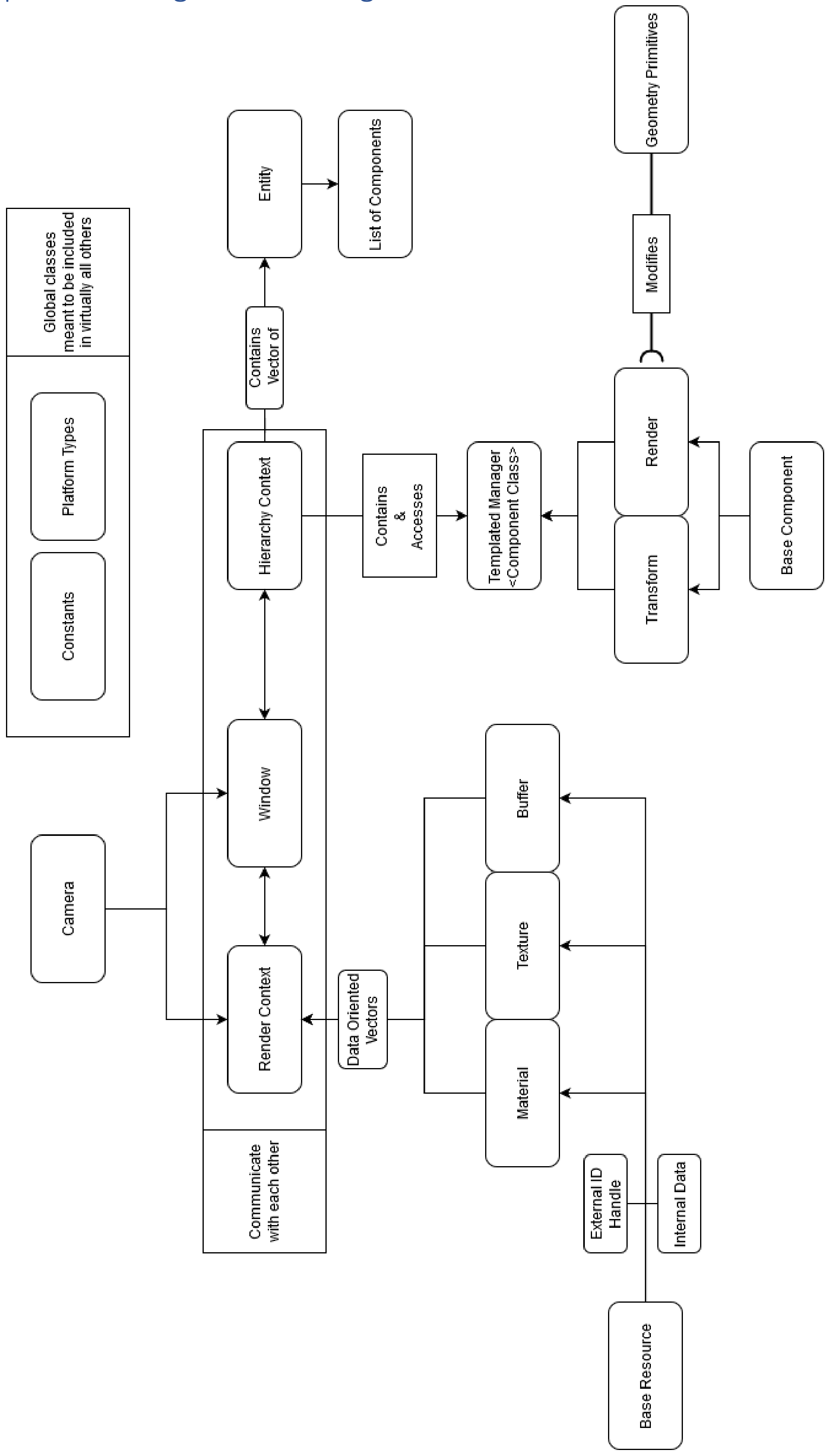
Surface – A surface is the object representing where the graphics are going to be rendered. It is optional – if the API is going to render offscreen and store the results, there is no need to create a surface.

Swapchain – Object defining the number of frames and its characteristics that are going to be used to render. The required image resources are automatically created in the swapchain's defined format, color space and size.

Commands – The commands will define the API behaviour during rendering and can be used to modify existing GPU resources asynchronously. They must be allocated from a Command Pool, recorded, and then sent to the corresponding Graphics Queue.

Render Pass – An object that determines the type and format of rendering output. It could target a colour texture, a depth swapchain image or multiple targets at once. While the object is necessary in the creation of a pipeline, indicating it the steps on the object's rendering, it can be destroyed and recreated without consequences once the pipeline has been successfully created.

8. Appendix C: Engine Class Diagram



Global classes meant to be included in virtually all others
Platform Types
Constants
Camera
Entity
List of Components
Contains Vector of
Hierarchy Context
Window
Render Context
Communicate with each other
Contains & Accesses
Templated Manager <Component Class>
Geometry Primitives
Modifies
Render
Transform
Base Component
Data Oriented Vectors
Buffer
Texture
Material
External ID Handle
Internal Data
Base Resource