Real-Time Ray Tracing - Creating a Vulkan Hybrid Rasterizer Ray Tracer from first principles

Oscar Mari-Jurado

b9019210@my.shu.ac.uk

Final Year Project Report

Supervised by Steve Oldacre

Second marker: Tom Sampson

Department of Computing

Sheffield Hallam University, United Kingdom

Abstract

Objective: Research and self-learning of the Khronos Group new generation graphics API, Vulkan, and from scratch design and implementation of a custom 3D rendering framework to serve as foundation for a future hybrid (rasterizer and ray tracing) real-time render. For hybrid rendering to be possible the render requires a deferred pipeline, support of render to target technique and PBR material support.

Methods and materials: The render will be developed in full C++ and GLSL (for shaders) using Visual Studio 2017 as main code editing tool. For features not related to rendering, third-party libraries and the C++ Standard Library will be used.

Results: Although the project scope has varied the basic features required as hybrid rendering base have been achieved. The research, development and abstraction of Vulkan took more time than expected due to its complexity and extension. This force the discarding of some features that were not critical. Reignite render proved to be able to render scenes with multiple geometries and materials with good quality results. Although, due to the lack of interaction tools, the development of scenes is very slow and requires to modify source code.

Conclusion: Since the base features have been achieved and the profiling data does not show any problematic performance issues it can be considered a success. However, it may require some previous preparation work before start implementing ray tracing features on it.

Keywords: C++, GLSL, 3D rendering, PBR, ray tracing

Index

1. Introduction

1.1. Overview

The main goal of this project is to design and implement the base framework for a hybrid, real-time, rendering framework/engine (rasterization and ray tracing) using Vulkan API, emphasizing learning about Vulkan, advanced rendering techniques and performance using Data-Oriented Design (DOD) and multi-thread design. With this concept, the objective is to learn about modern graphics APIs and how modern game development engines work from inside in order to achieve great quality results with a seemingly stable frame rate.

This report aims to give a global description and documentation of the design decisions and implementation of the project. Discussing initial ideas, how they have been implemented and the possible issues and design changes during development and its reasons. In the end, it will also suggest further work and possible features to implement in order to improve what has already been accomplished.

1.2. Aims and Objectives

The aims of the project are as follows:

- Learning the basics of Vulkan in order to understand how it works, its potential and being able to generalize it to a custom API.

- Create a basic HAL (Hardware Abstraction Layer) allowing abstracted window functionality, multiple components updating control and handling of graphic resources and multithreaded rendering.

- Learn and implement different rendering techniques that will support the project objectives of image quality and performance.

- Perform different demonstration scenes in order to show the final rendered results as well as their performance in machines with different specifications.

2. Research

2.1. Vulkan API

2.1.1. Overview

As other graphics APIs, Vulkan is designed as a cross-platform abstraction over GPUs. It is a relatively new API designed from scratch for modern graphic card architectures. Aside from different new features, the past decade saw an influx of powerful graphics on mobile devices. So it is, that Vulkan has been also designed to be cross-platform with mobile devices too (Alexander Overvoorde, 2016).

Vulkan solves different problems that old APIs have in exchange for a more verbose API, also allowing parallelization with multiple threads. It switches to a standardized byte format with a single compiler in order to avoid shader inconsistencies. Lastly, it acknowledges the general-purpose processing capabilities of modern graphics cards by unifying graphics (GPU) and compute (CPU) capabilities into a single API.

2.1.2. API Concepts

Vulkan has been designed with different conventions that makes easier learning to work with it. Functions have a lower case *vk* prefix. Types like enumerations and structs have *Vk* prefix and enumeration values have *VK_* prefix. Its API uses structs to provide parameters to functions. Object creation generally follows the pattern displayed in Figure 2.1.

```
VkXXXCreateInfo createInfo = {};
createInfo.sType = VK_STRUCTURE_TYPE_XXX_CREATE_INFO;
createInfo.pNext = nullptr;
createInfo.foo = ...;
createInfo.bar = ...;

VkXXX object;
if (vkCreateXXX(&createInfo, nullptr, &object) != VK_SUCCESS) {
    std::cerr << "failed to create object" << std::endl;
    return false;
}
```

Fig 2.1.- Basic creation pattern for any Vulkan object.
Retrieved from: https://vulkan-tutorial.com/Introduction

In order to control function errors, almost all functions return “VK_SUCCES” or any other error code. Is possible to find the definition of all the different error codes in the specification given by **LunarG** (2020) as well as information about functions and explanations of how work with the SDK (Software Development Kit).

It is interesting to talk about how Vulkan allows to enable extensive check with a feature called *validation layers*. The most striking is the capability to enable them during development and then completely disable them when releasing the application. LunarG provides its own standard on validation layers implementation. Because of validation layers being so extensive it is easier than with other APIs to debug and find errors and misconceptions.

2.2. Engine Programming

2.2.1. Overview

The term “*game engine”* has its origin in the mid-90s in reference to Id Software’s *Doom*, created by Id Software. Doom was architected with a well-defined separation between its core software component (3D graphics rendering system, collision system or audio system). After this, developers started to take advantage of this separations by licensing new games and retooling them into new products creating new art, weapons, characters, etc (Gregory Json, 2018).

*Fig 2.2.- Gameplay image of Id Software,s original Doom (1993) for **MS-DOS**. Retrieved from:* https://en.wikipedia.org/wiki/Doom_(1993_video_game)#cite_note-IGN100-15

Any game engine has the possibility of having different-purposed modules beyond rendering. Some of these modules could be support systems (memory management, containers, etc), debugging tools, collisions and physics, audio or even gameplay systems.

It is important to know which are the goals when developing an engine in order to focus on the main features, tools and modules that should be implemented first to head the project towards the required goals.

2.2.2. API Abstraction

Nowadays it is a common practice in modern game engines to hide methods and tool implementations to future users of the API. Developers hide external APIs and library use with custom classes that are directly related with the current framework.

Hiding function definitions allows to make changes on the implementation without modifying the current API. This allows updating and adding new features not forcing users to modify their use of the different API functionality.

Another important feature that is used in graphics programming is the ability to have different implementations of the same custom API using different graphics APIs like OpenGL, DirectX 11 or Vulkan as backends. This requires a heavy work on API design due to the many differences that graphics APIs have between them. But once is finished, it allows incredible flexibility and compatibility with many different machines and software.

The last possible objective of abstraction is to hide implementations and data information in order to maintain the functionality private so the user does not know how it works. This could be for code privatization purposes or avoiding changes in important code parts by the user.

2.2.3. Materials

Materials are an abstract concept used by artist and graphics programmers to define the different properties of how an object should be coloured and shaded. The different properties that can be applied to a material depend on the shading algorithm used (Phong, Blinn-Phong or Physically Based Shading, etc).

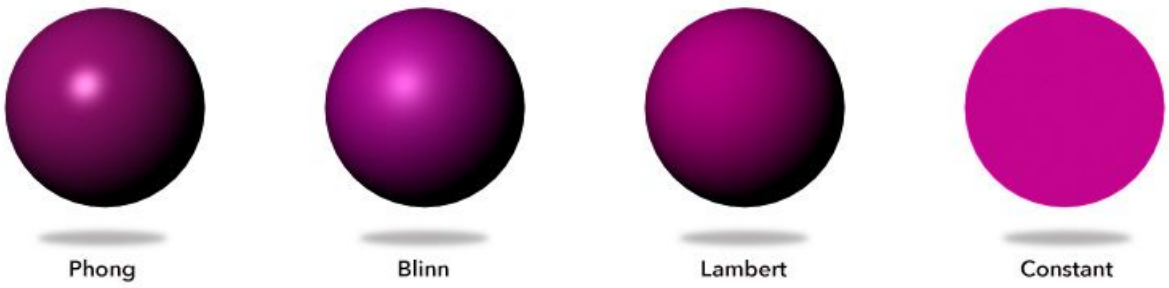

Fig 2.3.- Comparison of different lighting models and its result on the "same" material.
Retrieved from: https://docs.viromedia.com/v2.6.1/docs/3d-scene-lighting

Material instances should be entirely independent, allowing for a wide range of materials to be applied to an entity without requiring any changes in geometric data. Each time a custom shader is created, it can be used as a base archetype for multiple materials.

2.2.4. Multithreading and Parallelism

The term *parallelism* refers to any situation in which two or more distinct hardware components are operating simultaneously. However, parallel computing hardware is ubiquitous nowadays. One way to classify the various forms of parallelism in computer hardware design is considering the problem solved in each. Divided as implicit parallelism and explicit parallelism.

This project is focused on explicit parallelism due to its purpose of running more than one instruction stream simultaneously. It is designed to run **concurrent** software more efficiently than possible on a serial computing platform. Some examples of explicit parallelism are hyperthreaded CPUs, multi-core CPUs, multiprocessor computers and cloud computing. Multithreading can be performed in different ways depending on how the work is being parallelized. In this project, the chosen methodology for multithreading is the use of scheduling tools/libraries. Schedulers allow to stack different tasks, parallelize and create dependencies between them to avoid desynchronization between tasks.

CPU calculations and tasks are solved with multithreading or scheduling. However, GPU operations cannot be parallelised. Modern graphic APIs are built in order to be concurrent and thread-friendly, but this is not the case of old APIs. The graphic context on old graphic APIs is not concurrent a cannot be accessed by different threads. This makes mandatory to make all the graphic related function calls from the thread where the context was created (main thread) in order to avoid the application to break.

2.3. Physically Based Rendering

2.3.1. Basic PBR

Physically Based Rendering is a collection of render techniques that are approximately based on the same underlying theory that more closely matches that of the physical world. It generally looks more realistic compared to other lighting systems as tries to mimic light in a physically plausible way.

For a PBR lighting model to be considered physically-based it has to satisfy 3 conditions. Be based on the microfacet surface model, be energy conserving and use a physically-based BRDF (Bidirectional Reflective Distribution Function) (Morgan Kaufmann Publishers In, 2016).

Furthermore, PBR pipeline properties are commonly fed using texture maps. Using textures gives a per-fragment control over each specific surface point should react to light. Figure 2.1 shows an example of the different properties and the resultant shaded material.

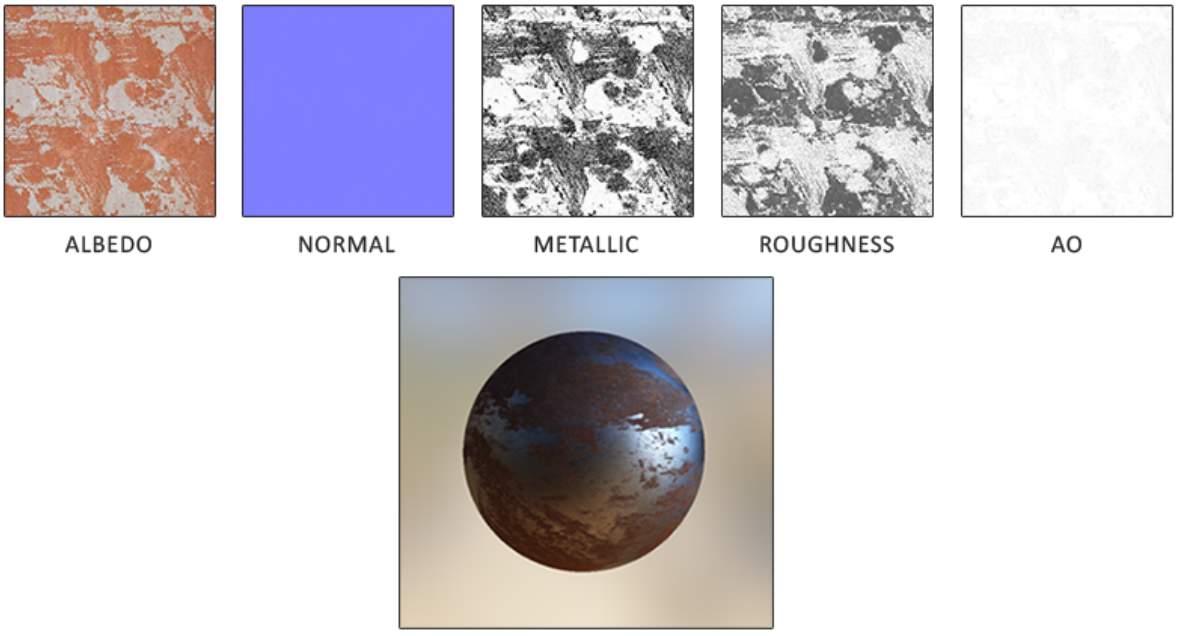

Fig 2.4.- List of textures frequently found together in PBR pipelines and visual output of it.
Retrieved from: https://learnopengl.com/PBR/Theory

2.3.2. Advanced PBR

The results of basic PBR depends only on the parameters of the material and the textures given to the pipeline for that material. These result can be improved by applying advanced PBR techniques. After this, the most common feature is Image Based-Lighting (IBL). IBL is a collection of techniques to light objects by treating the surrounding environment as one big light source. This is generally accomplished by manipulating a cube map environment map.

The objective of Image-Based Lighting is to pre-compute an irradiance map as the lighting's indirect diffuse portion. But, after this, the result does not look accurate for metallic materials because of the lack of specular reflection. After applying Specular IBL the output will be corrected for metallic materials as shown in Fig 2.2.

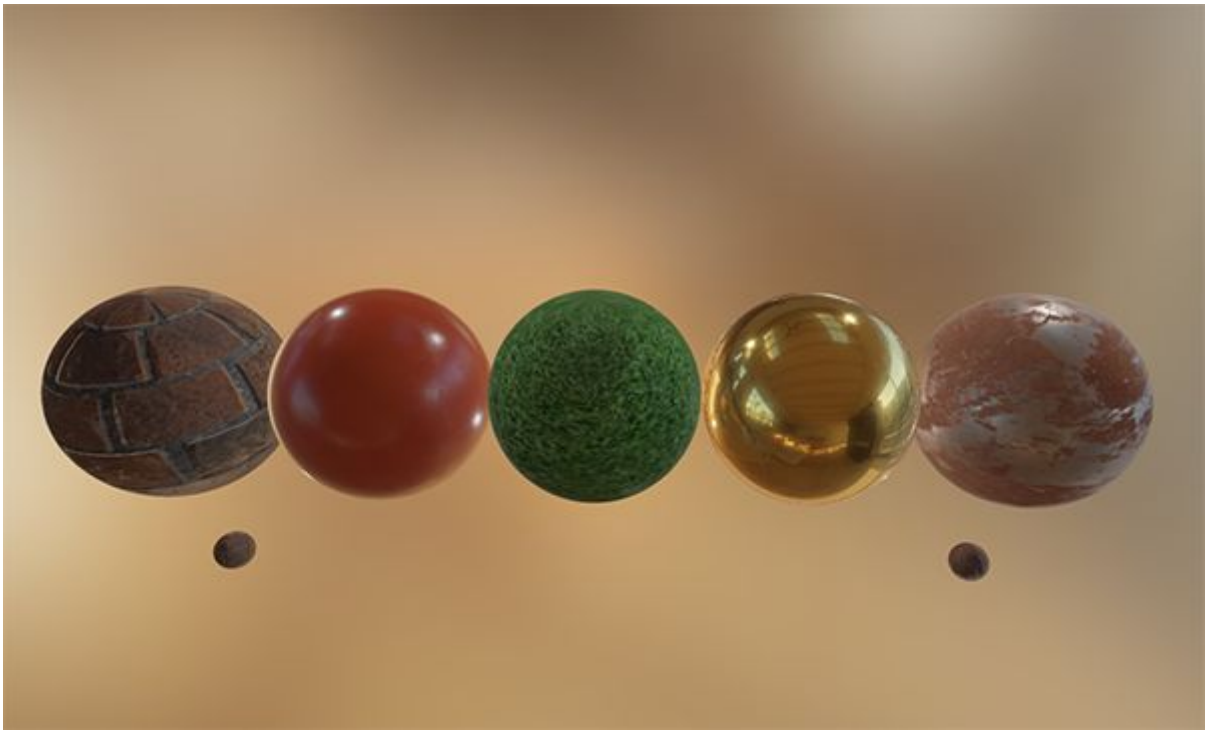

Fig 2.5.- Output of different PBR materials with diffuse and specular IBL applied to them.
Retrieved from: https://learnopengl.com/PBR/Theory

2.4. Hybrid Rendering

2.4.1. Ray Tracing

Ray tracing is a rendering technique that provides realistic lighting by simulating the physical behaviour of light. It has long been used for non-real-time rendering due to the computational cost it requires to compute the color of each pixel. Figure 2.6 shows result of offline path traced scenes based in *Ray tracing in a weekend* (2018) and *Ray tracing the next week* (2018) made by Peter Shirley.

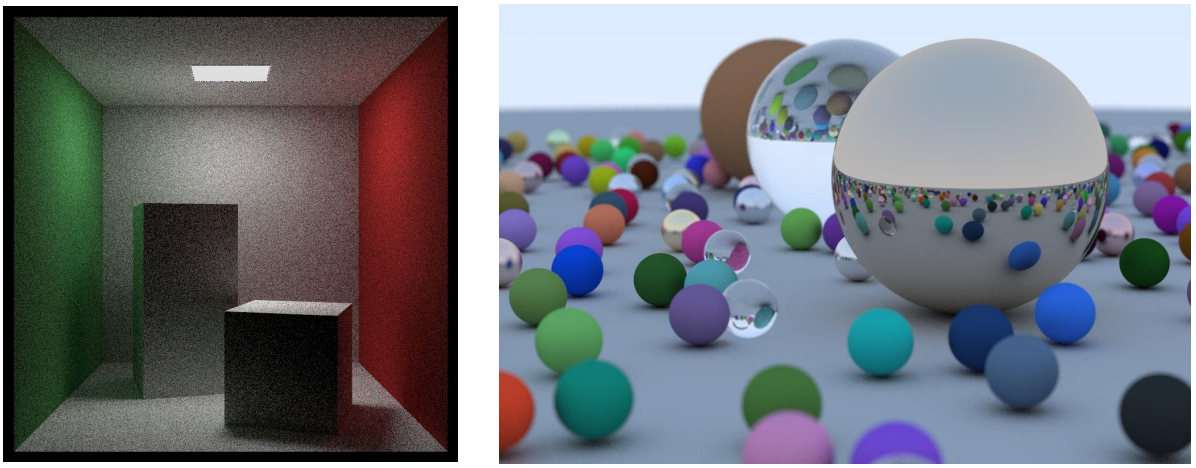

Fig 2.6.- Results of a custom path tracer for showing the quality of ray tracing techniques

The technique consists on tracing the path a light ray would take if it were to travel from the point of view of a scene. The light ray may be reflected between different objects, blocked or pass through transparent (or semi-transparent) objects. The resultant pixel color is calculated by all those interactions.

Due to the iterative process and the intensive computational operations that this technique requires, ray tracing has long been “the future” and not an affordable technique to use. Nowadays, developers are seeing the advent of consumer GPUs which have enough compute capability to do interesting ray tracing workloads in real-time.

2.4.2. Rasterization and Ray Tracing

The main objective behind hybrid rendering is to give a rasterizer render the capabilities of improved image quality results and adding ray tracing only features such as global illumination, transparency and translucency.

For this purpose, it is first necessary a fully functional renderer that supports render targets and deferred shading. Deferred shading takes advantage of a structure called G-Buffer. This structure contains all the necessary geometric data for rendering a scene. The values are stored into multiple render targets and are lately accessed by the fragment/pixel shader. For more information about Deferred rendering, refer to Section 4.2.3.

Ray tracing is applied after the rasterization stage, once all render targets are storing the geometric information for the scene. Each new feature requires different uses of the ray tracing techniques. The following features explain a theoretical implementation of each one as well as the required information (by SEED. PICA PICA demo).

- Shadow rays. The ray traced implementation of shadows (hard) only requires to launch a ray from the surface towards a light, and if the ray hits a mesh, the surface is in shadow. A hybrid approach would rely on a depth buffer during rasterization (G-Buffer) to reconstruct the surface's world-space position. The result serves as origin for the shadow ray.

  In addition, soft penumbra shadows are implemented by launching rays in the shape of a cone. Although soft shadow are more representative of real-world shadowing, its computational cost is also superior.

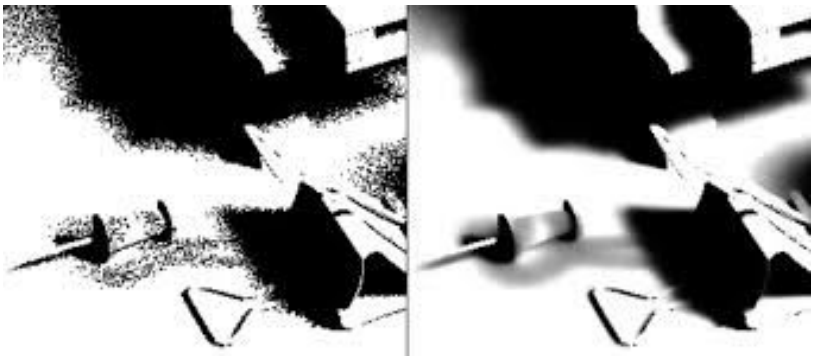

Fig 2.7.- Hybrid ray traced shadows: unfiltered (left), filtered (right).
Retrieved from: https://link.springer.com/content/pdf/10.1007%2F978-1-4842-4427-2_25.pdf

- Ambient Occlusion. Rays for AO are randomly generated with a stochastic sample function. The generation is cosine weighted in order to reduce noise generation.

Fig 2.8.- Hybrid ray traced Ambient Occlusion. Retrieved from: https://www.highperformancegraphics.org/wp-content/uploads/2018/Posters-Session/HPG2018_HybridAmbientOcclusionPoster.pdf

It is necessary to indicate that the output information of this techniques may result in generated noise. This requires the addition of an extra filter process called **denoising**.

2.5. Planning

Although the project is focused on rendering results, its implementation is related to a wider range of knowledge. In order to organise the numerous tasks that are required it is necessary to use any planning tools. This will help to improve the organisation and efficiency of the development of the project. Trello was the tool used for this project and report.

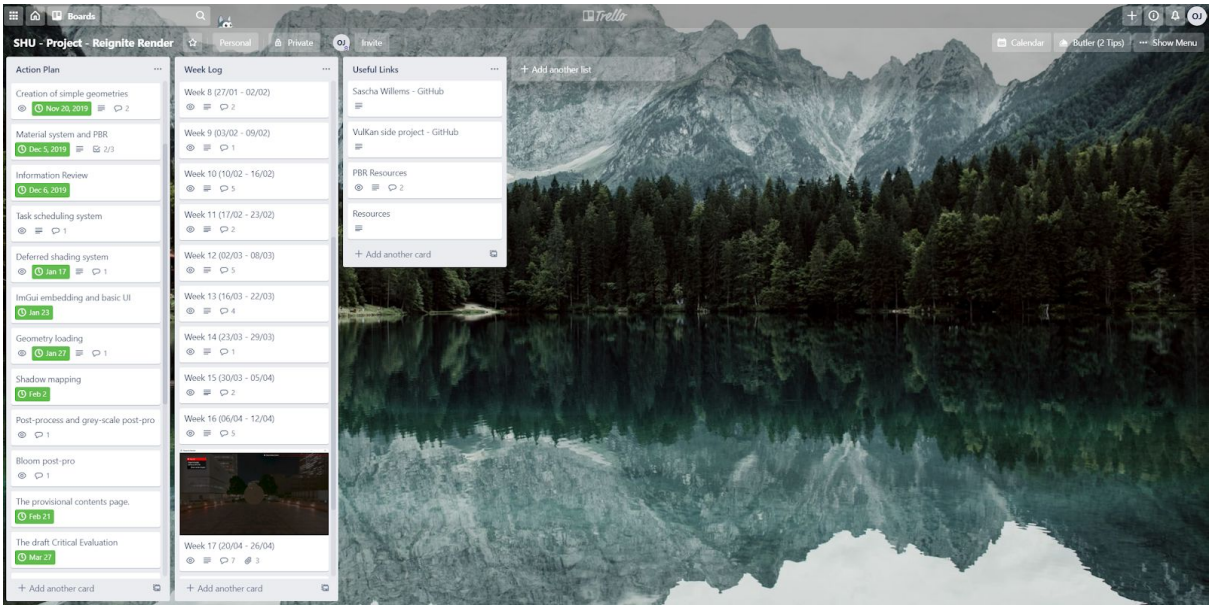

Fig 2.9.- This report's Trello board

In the Figure 2.9, a late version of this report's Trello board can be seen. The work is divided mostly in the between the "Action Plan" and "Work Log" columns.

The Action Plan displays an ordered list of cards with all the required tasks for the success of the project. Those cards have 2 important characteristics. Some tasks may be too abstract or still have a wide range of work. Because of this, the first caracteristique is the division of each main task into a check list of smaller and more accessible tasks.

The Work Log, as the name suggests, is a section in which the work done is reported per week. The overall tasks are indicated in the description of the card whilst the activity comments may reflect advancements in the development, encountered errors, solved bugs or explanations of followed processes implementing a feature.

Finally, there is also a board for useful links. This links may help the development of the project by supporting it, giving options to resources and assets or by being useful research content.

3. Design

3.1. Avoiding the Singleton Pattern

3.1.1. The Singleton Pattern

The Singleton Pattern is a programming design pattern with the objective of ensuring a class has one instance only and providing a global point of access to it. These characteristics avoid errors in classes that cannot perform correctly with more than one instance and allow any other class to access from anywhere.

```
class Singleton {
 public:

  static Singleton& instance() {
    static Singleton* instance = new Singleton();
    return *instance;
  }

 private:
  Singleton() {}
};
```

Code Snippet 3.1.- Modern implementation example of the Singleton Pattern in C++.

As other programming patterns, it has different benefits: the instance is not created if not used, It is initialized at runtime and it can be subclassed; as well as some disadvantages: it complicates error debugging, encourages to couple independent modules on the singleton (inexperienced developers), and being global allows every thread to access to its memory which is not concurrency-friendly.

3.1.2. Avoiding Singletons

Projects that depended on singleton instances tend to encounter some of the already mentioned issues as well as its benefits. In this project, one of the aims is to avoid the use of this pattern and approach different solutions to the problems solved by the Singleton Pattern.

First of all, a structure to store components and other engine data is needed. This structure will be shared between different main systems. There are multiple options in order to avoid the Singleton pattern:

Pass it in. Creating the structure as dynamic memory and moving in through the different classes as a parameter would allow me to share the instances only with the classes I want with a great control of the instance.

From the base class. Implementing the functionality and storage data members in a base class allows to move functionality easier.

From something already global. Even though this option does not discard the Singleton Pattern is a good solution for different module instances and avoiding coupling.

After comparing, the best option to choose is to pass a shared structure as a function parameter. The objective is to share memory storage overall instead of functionality. For this purpose passing the structure as a parameter when initializing the different subsystems that will need the information will work better in the project.

3.2. Entity Component System (ECS)

3.2.1. Introduction

The Entity Component System is a recently new architectural pattern used mostly in video game development. It consists of entities (any possible object inside the scene), that contain components. Its approach is often combined with data-oriented design techniques in order to maximise performance.

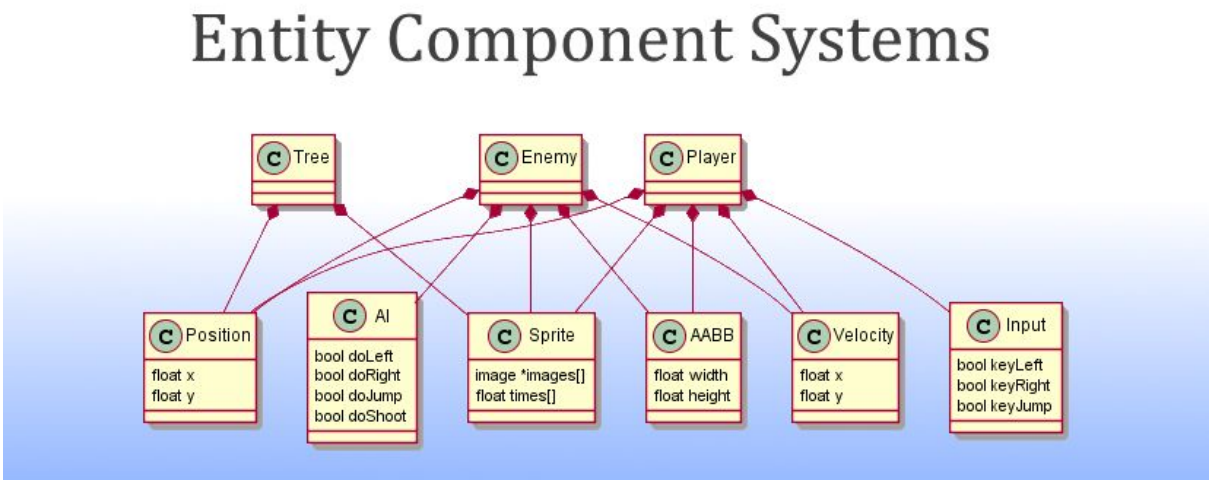

Fig 3.1.- Basic ECS scheme example. Retrieved from:
https://namekdev.net/2017/03/why-entity-component-systems-matter/

Another objective of the ECS is to avoid the complex hierarchical structures derived from Object-Oriented Design. A basic definition of ECS only involves: Entities, Components and Systems.

3.2.2. Entities

Entities are general-purpose objects. Their implementation may defer on the different systems to be used at. It can be a structure that contains components or, as in this case, it consists of a unique id. Normally a plain integer that refers to a real entity over the scene and is related to a set of different components.

```
typedef entity_t uint32_t;

struct EngineState {

  vector<entity_t> entities; // container storing entity ids
};
```

Code Snippet 3.2.- Example pseudo-code of basic entity storing

3.2.3. Data-Oriented Components

A component contains certain information for one aspect of an object and its interaction with the world. As well as entities, its implementation may defer but is normally implemented as structs, classes or associative arrays (maps, dictionaries, etc). Components are sets of pure

data without functionality. In this case, it is structured following the SoA (Structure of Arrays) layout which is supposed to be more cache-friendly.

```
struct TransformComp {
  vector<float3> positions;
  vector<float3> rotations;
  vector<float3> scales;
  vector<mat3> local_matrices;
  vector<mat3> world_matrices;
};
```

```
struct RenderComp {
  vector<bool> is_visible;
  vector<bool> cast_shadow;
  vector<Mesh> meshes;
};
```

Code Snippet 3.3.- Basic pseudo-implementation of Data-Oriented component structs

3.2.4. Component Systems

Every system is related to an existing component. Its objective is to update the component data from its current state to its next state. This is logically operated by the processor in which each System had its own private thread.

There may be cases where some components depend on data that is owned and calculated by other components. In this case, threading should be adjusted to an order for components to update a well as the use of mutex.

3.3. Command Based Graphics

3.3.1. Overview

Although modern graphic APIs are built having multithreading in consideration it is still important to support old APIs for compatibility reasons. Therefore, it is necessary to find a way to old and new graphic APIs to work in the same environment.

It is proposed a Command System similar to what modern APIs are currently doing to manage multithreading. For this purpose, a command is a class composition of related data and functionality. A command can be initialized and later executed in order to cause the necessary changes over the render state to render an image. Commands are initialized by different threads and loaded into a new concept structure: a Display List. Finally, when all commands are stored, the frame is ready to be rendered and all the commands are

executed in the main thread. This is what allows compatibility with old graphic APIs which its context is not concurrent and can only be managed by the creation thread.

3.3.2. Required Items

Several items are needed in order to implement correctly the already named system. A base command, multiple inherited commands and the Display List object.

A BaseCommand class is necessary for further inheritance. This class will define the basic structure and functionality of all commands as an abstract class ready to inherit from.

```
class BaseCommand {
 public:

  BaseCommand() {}
  virtual ~BaseCommand() {}

  virtual void execute() = 0;
};
```

Code Snippet 3.4.- Possible pseudo implementation for an abstract BaseCommand class

New commands are classes inherited from BaseCommand. The name of each Command would refer the task to fulfil. The inherited Command would contain the necessary data to fulfil the task as well as reimplement the execute function with the necessary graphic calls for the task. Also, each new Command would implement an initializer function that would be used to initialize the data attributes of the Command.

The last required object is the Display List. This structure aims to store all the necessary commands for rendering a frame. It will allow command storage using a template container, checking if the command list is finished and submitting the list. When submitted, the main thread will iterate over the command list executing the different commands stored.

```
class BaseCommand;

class DisplayList {
 public:

  DisplayList() {}
```

```
  ~DisplayList() {}

  void uploadCommand(BaseCommand*);
  void submitDisplayList();

  void runDisplayList();

 private:
  vector<BaseCommand*> commands;
};
```

Code Snippet 3.5.- Pseudo-code example of the DisplayList class functionality and content.

4. Development

4.1. Engine and Systems

4.1.1. Application

The main application is created as a C++ class that compiles into a **DLL** file. It is developed in order to be used as a base class for inheritance to client applications. It also has a custom entry point prepared for adding multi-platform compatibility.

The client application is done by overriding an extern “CreateApplication” function that must be overwritten in order to make the application project work. This idea was taken from Yan Chernikov series of videos, called Game Engine in which he teaches how to build from scratch a game engine in C++ and using OpenGL.

Lastly, the base application class also contains the different main system that will be built as smart pointers. It will also create a state data container, also as a smart pointer, that serves as point of access between systems.

4.1.2. Hardware Abstraction Layer

A Hardware Abstraction Layer (HAL) is the group of custom API-based functions that allow clients to perform hardware-oriented operations independent of actual hardware.

First of all, in order to give users a more complete API to work with, different tool classes, as well as functions and type definitions, were implemented. Timer class, file management functions, logging functionality and window abstraction class and methods.

Instead of acquiring the different window handlers from Windows, the program runs GLFW in order to manage windows and input with a user-friendly API and cross-platform capabilities. The custom window class has its own methods implemented with GLFW functions to create the window and the graphics context as well as binding the Vulkan context to the window’s one.

4.1.3. Component System

The component system, as one of the main systems of the engine, manages the allocation, updating and destruction of all the different components. It is based on the ECS pattern modernized by Unity on 2019. On creation, it receives a data structure with the parameters that the class will use as its configuration such as maximum component size or the engine state.

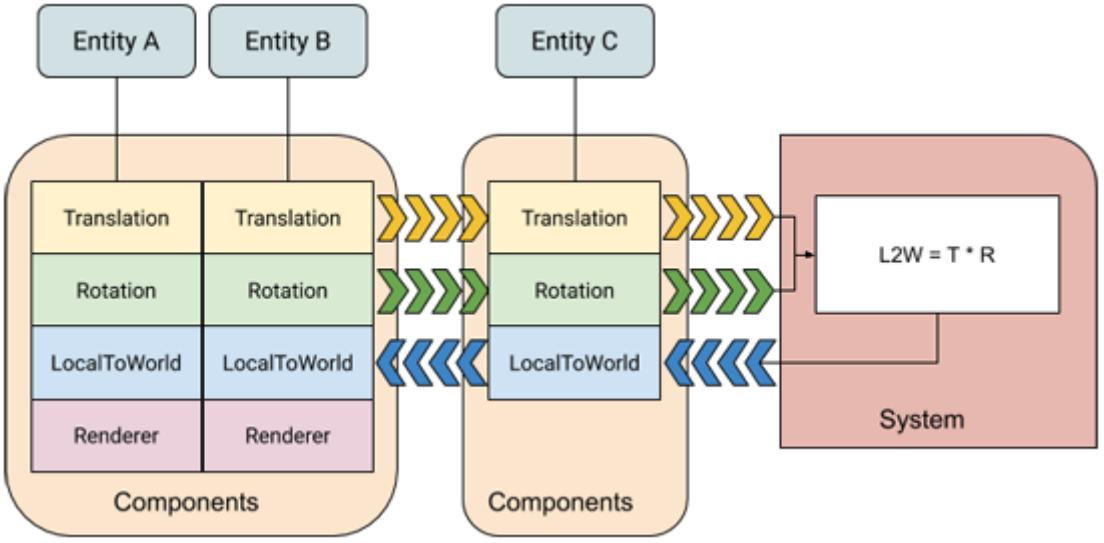

Fig 4.1.- Schematic example of an ECS following the archetype differentiation used by Unity. Retrieved from: https://docs.unity3d.com/Packages/com.unity.entities@0.10/manual/index.html

Once the component system is defined, different components are implemented as data structures that contain the necessary variables needed by each component as well as basic functionality. Each component is defined in a different file along with a set of functions for component initialization and updating. These replace the need of system classes in order to code simplification. In some cases, components may have dependencies from other components. This is why component updating occurs in a predefined order.

4.1.4. Render Context

The Render Context, as another of the main systems of the engine, manages the allocation, use and destruction of any graphic resources (related to Vulkan, in this case). Graphic resources are encapsulated inside structures or are part of or managed by the render context. As well as the Component System, the render context has access to a state pointer from which it can access to window information or entity data for rendering purposes.

When initialized, the render context contains all the necessary resources already created and prepared for rendering (geometries, materials, textures, etc). In the next section, an explanation of the state of command creation and calling will be done.

4.1.5. Command System

The development of Vulkan related code went through several changes during the whole development of the project. This complicated the implementation of abstraction commands as the information to initialize commands was continuously varying.

As there were being several complications with the implementation of rendering commands and the submitting date was too close for allowing more big changes in the current used concepts, Vulkan raw commands were left as they were.

4.1.6. Material System

Material instances are data packages composed by different Vulkan graphic resources as well as shader related information or identifiers of another resources. A material defines how the related entity is rendered on the scenes by configuring its data it is possible to modify its colors and sampled textures.

Fig 4.2.- Sample scene of multiple entities with different (PBR) material configuration in UE4. Retrieved from: https://www.unrealengine.com/marketplace/en-US/product/100-pbr-substances

In order to obtain the best quality results possible, materials also allow texture mapping in order to add irregularities and details to the created materials. This textures come in form of jpg/.png or ktx files and may represent normal maps, roughness and metallic indices and even occlusion maps.

Any material instance is stored by the render context as part of a Material Resource struct container. Entities relate to its own material by an index stored in the Render Component structure. This process is completely independent to vertices data as long as the necessary information is stored in the geometry (normal vectors, texture coordinates, tangent and bitangent vectors).

As mentioned in the previous section, Vulkan related code went to several evolution phases. These phases affected materials above all as the information contained as well as the resource creation was being continuously modified.

When the material pipeline seemed to be finished a problem was encountered. A misunderstanding on how per-object data was given to shaders in Vulkan provoked a last modification of material data members. Vulkan requires per-object data to be bound to a separate descriptor set and to update the given descriptor set each new rendered object. This new composition was achieved but it cost time that could be spent in abstraction work.

4.2. Vulkan API

4.2.1. External Demo

Before start developing the final year project, a “Hello Triangle” demonstration with Vulkan was being developed in order to start learning about the basics of new and modern graphics APIs. Although the tutorial is very complete, it only allows a generic view of the Vulkan features.

After developing further features over the demonstration, spending some time rechecking the code in order to remember the basics of it was necessary. Then, further features were implemented following the Vulkan-Tutorial web page by Alexander Overvoorde. This allowed to have an overall-contact with the API in order to start working.

Fig 4.3.- Result of Vulkan-Tutorial web demonstration with different implemented features

During the tutorial, several issues were encountered due to pipeline creation and image layout transition errors when creating textures. But these errors did not arrive to be real complications thanks to the Vulkan validation layers which offer a good error analysis.

4.2.2. Encapsulating Vulkan

Working with the extensive API of Vulkan complicates development due to the large time spent configuring basic structures, copying or rewriting code in order to add new features.

In order to avoid to spend too much time, the solution is to encapsulate Vulkan objects within new custom C++ classes which its attributes ought to be related objects and structures. Also, a set of initializer functions for Vulkan structures help for agile development. This is not an easy task to do when starting with Vulkan and is recommended to be done only after acquiring some experience with the graphic API.

The purpose of each class depends on the objects encapsulated and its implemented functionality and follows the patterns used by Sascha Willems in its Vulkan repository.

- VulkanState. Encapsulates Vulkan device and physical device as well as their properties, loaded on creation. Also contains functionality for memory allocation of graphic resources, pipeline information and image presentation.

- Swapchain. Contains the surface to present on the different stored buffers of images rendered and presented. It manages window creation and image presentation.

- Texture. Stores different attributes for image and descriptor configuration. Also implements multiple classes for different types of textures (1D, 2D, Cubemaps, etc) and overrides its functionality when needed.

- Buffer. Encapsulates a Vulkan buffer object and the properties and attributes needed for configuration and managing. Depends on a VulkanState object for creation.

- Framebuffer. Encapsulates a Vulkan Framebuffer. Manages creation and destruction functionality and dynamic number of attachments of it.

- Overlay. Its implementation is based on ImGui, a third-party UI library. The class encapsulates the creation of the different resources needed as well as the libraries own resource initialization and configuration.

Each encapsulated class helps on quicker development and avoid continuously copying similar fragments of code.

4.2.3. Deferred Rendering

Deferred rendering or deferred shading is a rendering technique in which the shading of the elements of the scenes is posted to a second stage render pass. On the first pass, all the necessary information for shading (positions, normals, depth, etc) is gathered in multiple framebuffers. Each of these framebuffers are rendered into a Geometry Buffer using the render to “texture technique”. After that, light is computed in a pixel shader over the resultant texture.

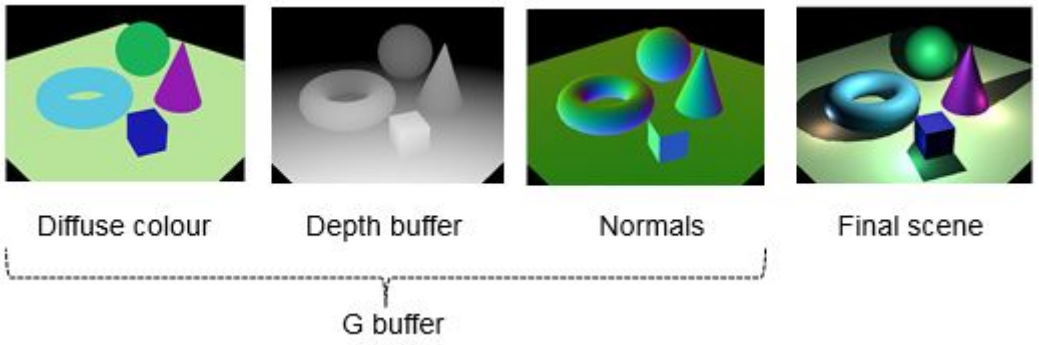

Fig 4.4.- Basic rendering layout of a deferred render. G-buffer (1st pass), Shading (2nd pass). Retrieved from: https://knowww.eu/nodes/59b8e93cd54a862e9d7e41d0

For this implementations it is necessary to implement two different pipelines. The first pipeline and its shader will manage the rendering of multiple framebuffers as framebuffer attachments to result on a G-Buffer that will be rendered into a texture. The second pipeline and its shader will shade the scene using the data obtained from the G-Buffer and paint the resultant texture on a full screen quad facing the camera. For this purpose, Vulkan requires two different command buffers to do the multiple rendering passes.

In addition to the two main vulkan pipelines that this technique requires, another one has been added for debugging purposes. The commands related to this pipeline are scoped inside a boolean flag. When activating the flag, command buffers are rebuilt in order to allow the new view to be rendered showing the basic content of the G-Buffer.

4.2.4. Physically Based Rendering

There is no real difference between implementing PBR in a forward rendering pipeline and a deferred one. In the deferred pipeline, instead of calculating the fragment color based on vertex data, the information is retrieved from the different G-Buffer attachments per pixel.

The information required by the PBR pipeline is passed to the shader using different Vulkan uniform buffer objects with multiple descriptor sets. Each descriptor set contains categorized information. Camera data (projection matrix and view matrix), light data (light view matrix, position, direction, etc) and per-object data (model matrix). The information related to each material is also passed to the shader but using Vulkan's "push constant" feature instead of ubos.

The mathematiques used for PBR are more complex than the ones used for Blinn-phong. All these calculations are packed in the BRDF function and the result is the final color of each fragment of the resultant shaded image.

4.2.5. Cubemap

A cubemap is a texture type composed by 6 2D individual textures the represent the six sides of a cube. This textures can be used both for simulate a landscape view (skybox) but also rendering reflection effects on PBR materials (IBL).

The pipeline with the aim of rendering a skybox is rendered before any other entity over the scene and ignores depth values as it will always be in the background of the rendered frame. Moreover, the geometry where the cubemap texture is sampled (a simple cube) is rendered counter-clockwise unlike other entities as the point of view is inside the geometry. The results of the cubemap object are rendered directly on the albedo attachment of the G-Buffer.

There is also the possibility of implementing **HDR** skyboxes. Those have a considerable augment in image quality for reflective materials but require a slightly different approach as HDR skybox textures are sampled as spheres and its color values have more precision.

4.2.6. User Interface

In order to facilitate the slight interaction with the scene the render requires some user interface overlay. But, to implement a user interface from scratch is a workload that could be considered a project itself. For this purpose, the render will have ImGui implemented as solution for user interface tasks.

Dear ImGui is a **bloat**-free graphical user interface library for C++. ImGui is compatible with multiple graphics API and platforms and is widely used and know by the software development community. This makes it the best option in order to resolve the lack of user interface.

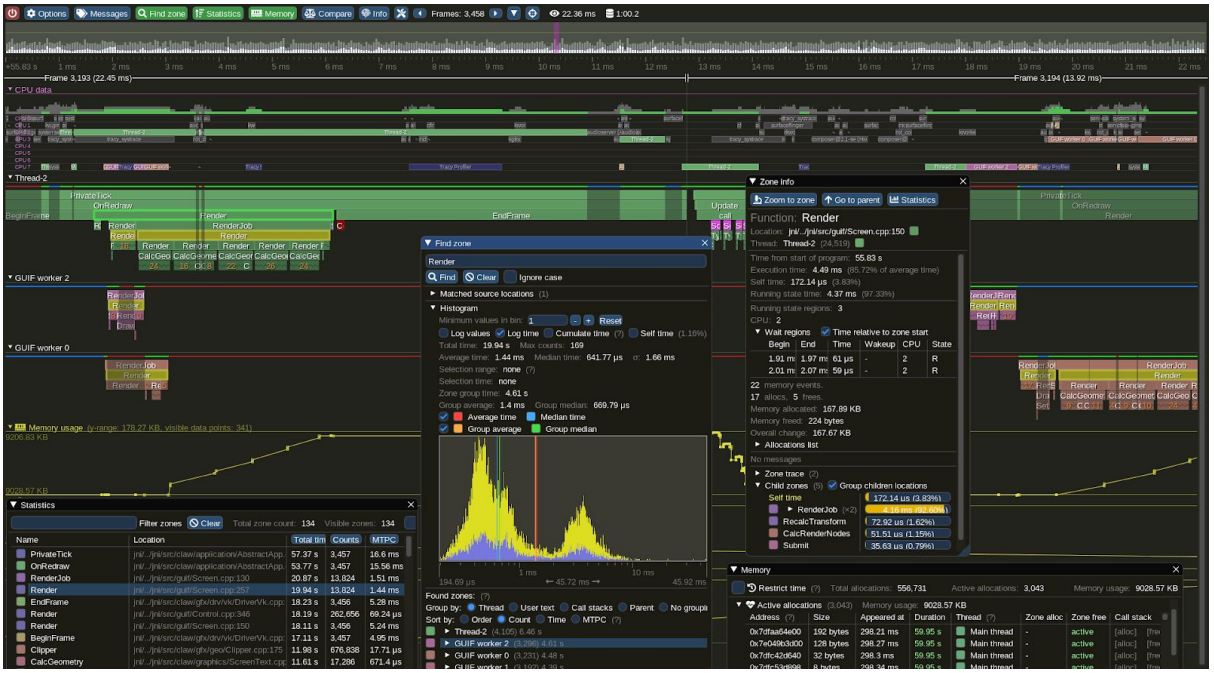

Fig 4.5.- Reference image from the Dear ImGui GitHub that show the capabilities of the library. Retrieved from: https://github.com/ocornut/imgui

Reignite render takes advantage of Dear ImGui in order to alternate between debug modes, small modifications of scene structuring and showing profile information in real time. In the future, it could also be used to display entity data to make it modifiable in at the moment alike modern engines like Unity or Unreal Engine 4.

4.2.7. Shadow mapping

When rendering objects in 3D scenes it is difficult to know the exact location of an object above another. Because of this, 3D renders tend to implement shadows in order to reference correctly the position of objects as well as for realism. The technique used for rendering shadows in 3D environments is called shadow mapping.

Shadow mapping techniques have a higher computational cost compared to other basic rendering techniques. Rendering the scene is required as many times as source of light in the current scene. This is done by behaving the different light sources as if they were cameras (points of view towards the scene). The resultant “shadow map” is used after the final render pass and mixed the resultant shaded scene (cascade shadow map).

Reignite render follows a different approach. The first render pass renders the scene for each light in light space coordinates. The rendered information is stored in a new render

target attachment of the deferred G-Buffer. The resultant attachment will be used by the deferred pipeline to know if the calculated pixel needs to be shaded or shadowed.

After several days of testing no shadows were being drawn in the scene. After debugging different frames of the scene using RenderDoc the shadow map attachment was being drawn correctly with the light perspective and this attachment was being passed correctly to the deferred shader pipeline. Even after debugging, the issue was not encountered and had to be left as it was.

4.3. API Abstraction

4.3.1. Commands

Commands are designed and implemented as the main abstraction layer between the render and any of the different graphics APIs that could be implemented (Vulkan, in this case). Any command created, would have as many implementations as different APIs are compatible with the engine. But, its interface must not be modified to be adapted to any of those graphics APIs.

In order to facilitate the storing of multiple commands with different functionalities, those commands will be stored as pointers to a parent class inside an STL container. Any command which purpose is to be used in the render must inherit from the BaseCommand class.

```
class riCommandBase {
 public:

  virtual void execute() = 0;
};
```

Code Snippet 4.1.- Implementation of base command as abstract class in Reignite Render

Commands functionality may vary between modifying the rendering state, allocating graphic resources used by the render or drawing multiple entities of a scene. The new-purpose commands override the execute function in order to implement the functionality that each needs to achieve.

It is common that a command will also need extra information to work. This information may vary between identifiers (ids) or pointers to different structures containing API information. For this purpose, the command will also implement an initialization function that will receive and load the information for the command to work later on the execution of the program.

```
class riCommandNewPurpose : public riCommandBase {
 public:
  void init(/*Info*/) {
    // Initialization of attributes
  }

  virtual void execute() override {
    // API calls...
  }

 protected:
  // API resources
  // Not API resources
};
```

Code Snippet 4.2.- Generic implementation of a new-purpose command in Reignite Render

To make the commands be called, it is necessary to upload them to a Display List once the command has been initialized. The next section explains what a Display List is and how is used.

4.3.2. Display List

Display lists are the objects entrusted of managing commands and execute sets of commands. The basic functionality of a display list can be resumed as uploading commands, submitting/executing commands and flushing stored commands. Although this functionality may vary depending on how the display list is used.

```
class DisplayList {
 public:
  void uploadCommand(riCommandBase* cmd) {
    // Upload cmd to container
  }

  void submit() {
    // Iterates container
```

```
    // Call execute for each cmd
  }

 protected:
  container<riCommandBase*> commandList;
};
```

Code Snippet 4.3.- Pseudo implementation of a Display List object and its basic functionality

4.4. Tools and Third-party Libraries

Git is, nowadays, the most widely used modern version control system in the world. The decision for this tool was clear as it is free and allows great working flexibility. This project is uploaded to GitHub as the chosen git distributor.

Visual Studio 2017 has been the main tool used for the development of this project. Visual Studio is the preferred code editor for software development in windows platforms among developers create by Microsoft. Due to the nature of this project the decision clear.

GENie is a project generator tool, created by Bkaradzic, allowing compatibility with projects in multiple platforms and softwares. Although its use was not planned at first it helped during development a lot. This tool will allow the future multiplatform development of the render.

RenderDoc is a free, stand-alone graphics debugger that allow quick and easy single-frame capture and detailed introspection of any application using the compatible APIs. RenderDoc is now a dow one of the most used tools by the graphics programming community.

GLFW is an open source, multi-platform library compatible for OpenGL and Vulkan development. It provides a simple API for creating windows, contexts and surfaces, receiving input and events. Between the different options, GLFW was the best choice as it allowed a fast implementation and abstraction to start working with Vulkan.

GLM is a header only C++ mathematics library for graphics software based on GLSL. GLM is the most known math library abroad the graphics programming community. The basic types and functions that provides save lots of development time.

SPDLog is a very fast, header-only, C++ logging library by Gabime. It allows very dynamic and agile logging functionalities. Moreover, its implementation its multi platform which fits perfectly with this project.

TinyObjLoader is a single file wavefront obj loader written in C++. Reignite render takes advantage of this library to populate its scenes with different obj models loaded from file.

Volk is a meta-loader for Vulkan. It allows to dynamically load entry points required to use Vulkan without linking to vulkan-1.dll or statically linking Vulkan loader. This library was discovered when developing the first Vulkan test demonstration.

STB Image is a public-domain, single-header, image loader belonging to a set of public domain libraries for C/C++ by Sean Barrett.

KTX is a lightweight file format textures for OpenGL and Vulkan of the Khronos Group. KTX files contain all the parameters needed for texture loading.

5. Results

5.1. State of the project

The project in its last version after submitting provides all the basic features required for a second development phase in which ray tracing techniques start to be implemented and applied. Alongside the project, there is a compressed folder with several gathered resources that the engine needs to render scenes.

Reignite render allows to create scenes with varying content of loaded model (obj) files, different PBR materials (simple and textured) and multiple light sources. It also allows clients to move around the scene to have a detailed look of the content of the scene with a FPS fly camera. The render basic implementation also allows to debug multiple features related to rendering techniques.

There are also some issues alongside the project. Creating scenes (as well as materials, objects, etc) requires a heavy interaction with the source code of the project as it lacks of any wide-range interaction system (UI editor or scripting). Also, the very-detailed configuration of the material pipelines (Vulkan resources) puts some extra work when trying to create new materials. These slows development times and makes complicated to work on the render.

Lastly, some features had to be rolled out to give more development time to other more important features. This is the case for the task scheduling and post processing features that had to be discarded. However, those features are not completely discarded and may be implemented in the future

5.2. Final features

- Ready to use HAL. Reginite render allows several functionalities with its Hardware Abstraction Layer. This includes time profiling, logging information to console and loading resource tools that will be used later by the engine.

- Deferred shading pipeline and debug pipeline. Scenes in Reignite render are rendered by using deferred shading. This requires two different pipelines for G-Buffer generation and shading the result scene. Reignite also allows to draw the different basic attachments of its G-Buffer for debugging purposes.

- PBR and textured PBR. Materials in Reignite render support PBR shaded objects. This may be done with basic PBR to textured PBR and allowing to modify the different color and maps channels.

- Multiple light source. Multiple light sources and types (directional and point light) are also supported by Reigite render.

- Cubemap. Basic cubemaps can be loaded to Reignite render from ktx or jpg/png image fies.

- Interactive overlay. Scenes in Reignite render can be slightly interacted with thanks to a basic overlay implemented using ImGui.

- Porting ready. Although it still requires some work on some features, Reignite render is ready to be ported to different platforms as it does not have any Windows dependencies and the third-party libraries are compatible with multiple platforms as well.

6. Evaluation

6.1. Critical Reflection

As I expected when decided which would be the aims of the project, I have learned a lot about Vulkan in a deeper way and understood how modern graphics API work in order to improve performance results. But, also learned about programming techniques, new code libraries, rendering techniques, software design, development iteration and how to organize myself in a project with a large scope.

Specifying a bit more, this project allowed me to learn about the differences between graphics APIs and designing an abstraction compatible with different back-ends. Also, Physically Based Shading, which I know feel to understand its concepts and how the different mathematical approaches work.

The most common issue I encountered during the development was to solve the different validation layer errors when adding new content to the Vulkan pipeline. Moreover, the verbose code base of Vulkan is a great barrier at first. But, in the end, I got used to it and was really usable and understandable even though it takes some time to get new features to work. So it is that in the end, I started to encapsulate Vulkan objects into custom classes with functionalities that implied generic and parameterized methods to work faster.

Another issue I had encountered is the abstraction and hide of the general engine state object. This object is shared between the application and the base classes of the render containing data necessary to all of them. At the end, I did not find a more comfortable way to hide its content without needing to copy the structure definition between various cpp files.

Finally, time managing at first was a complication due to the magnitude of the features I had to research, implement and test before each milestone. Because of this, I had to discard different tasks that were not critical in order to achieve the expected result.

6.2. Future Improvements

The first improvement for CPU performance would be taking advantage of the command based design to apply task-scheduled multithreading to the application. For this purpose, it will be needed to modify commands in order to create the graphics resources in a deferred way. Alternately, with some extra research about Vulkan multithreading, it would be possible to check if the current system is compatible with it and design a way to implement it over the application.

Also, with the current material system, it would be possible to implement new material pipelines that take advantage of the current features and also add new to the render. Some of these features could be rendering techniques such as Parallax Mapping, PBR Image Based Lighting (IBL), HDR skyboxes and more.

In order to populate scenes with more variate content, a quality improvement would be the compatibility with GLtf models. This would allow to render new objects compatible with PBR pipelines as well as whole scenes.

In the end, this engine is half of the main aim of the project. The most important improvement once delivered is the implementation of the features to allow ray tracing over new G-Buffers containing information (diffuse, specular,z-depth and optical properties) for casted rays to improve the quality results of the render (soft shadows, visually-improved materials reflections).

6.3. Personal Reflection

When I started working on this project I was not sure how common was the use of hybrid rendering in the game industry and the graphics programming community nowadays. Moreover, the first option of this project was more related with only ray tracing instead of hybrid rendering. But, after a week of research, I found plenty of information to work with and the techniques that I would need to learn. The only thing I was clear about was the use of Vulkan API that, although it was difficult, I am very happy with the result.

My objective was to focus on the aspects a rendering engineer would do so I could focus on research and implementation of rendering techniques. But, I had to give some time for

designing and implementing the engine framework for rendering. This was not a real problem as I really liked trying (and succeeding, as I see it) to improve the engine design work I did in a previous rendering engine with OpenGL. I had the opportunity to implement deferred rendering with a new API, and learning new math applications with physically-based rendering (PBR). So, although its difficulty and workload, I personally enjoyed working on this project.

By contrast, I need to get myself used to report and document my work aside from Git commit messages. I do not think I have any problem with the language itself but I am not used to write in English for reports and that is something I need to improve. Still, I think I did not do too badly and I am happy with the result.

7. References & Bibliography

1. Akenine-Moller, Tomas, Haines, Eric, Hoffman, Naty, Pesce, Angelo, Hillaire, Sebastien & Iwanicki, Michal. (2018). *Real-Time Rendering, 4th edition.* A. K. Peters / CRC Press

2. Barret, Sean. (2020). STB. Single file public domain libraries for C++. Retrieved from: https://github.com/nothings/stb

3. Bkaradzic. (2016). GENie. Project generator tool. Retrieved from: https://github.com/bkaradzic/GENie

4. CppCOn. (2014, Sep 29). *CppCon 2014: Mike Acton "Data-Oriented Design and C++"*. Retrieved from: https://www.youtube.com/watch?v=rX0ItVEVjHc

5. Christophe. (2005 - 2020). GLM. OpenGL mathematics Retrieved from: https://glm.g-truc.net/0.9.9/index.html

6. de Vries, Joey. (2014). Learn OpenGL. Retrieved from: https://learnopengl.com

7. Fujita, Syoyo. (2016-2020). tinyObjLoader. Tiny but powerful single-file obj loader. Retrieved from: https://github.com/tinyobjloader/tinyobjloader

8. Google. (2018). Filament. Cross-platform, real-time, PBR engine. Retrieved from: https://github.com/google/filament

9. Gregory, Jason (2018). *Game Engine Architecture, 3rd ed*. A.K. Peters/ CRC Press.

10. Haines, Eric & Akenine-Moller, Tomas. (2019). *Ray Tracing Gems: High-Quality and Real-Time Rendering with DXR and Other APIs.* Apress.

11. Kapoulkine, Arseny . (2018). Volk. Meta loader for Vulkan API. Retrieve from: https://github.com/zeux/volk

12. Karlsson, Baldur. (2018). RenderDoc.
Retrieved from: https://renderdoc.org

13. Krakowiak, Daniel. (2018, Apr 4th). *Hybrid Ray Tracing - Final demo, real-time*.
Retrieved from:
https://www.youtube.com/watch?time_continue=390&v=itoMzP_KG6M&feature=emb_logo

14. Lapinski, Pawel. (2017). *Vulkan Cookbook, 1st edition*. Safari, an O'Reilly Media Company

15. LunarG. (2018). Vulkan SDK.
Retrieved from: https://www.lunarg.com

16. Melman, Gabi. (2018). SPDLog. Fast C++ logging library.
Retrieved from: https://github.com/gabime/spdlog

17. NvidiaGameWorks. (2018, Oct 17). *Exploring Real-Time Ray Tracing and Self Learning AI with the "PICA PICA" Demo.*
Retrieved from: https://www.youtube.com/watch?v=j-lCKUPIc-o

18. Nystrom, Bob. (2009). Game Programming Patterns.
Retrieved from: https://gameprogrammingpatterns.com

19. Open Source. (2019). GLFW.
Retrieved from: https://www.glfw.org

20. Overvoorde, Alexander. (2016). VulKan Tutorial.
Retrieved from: https://vulkan-tutorial.com

21. Owens, Brent. (2013). *Forward Rendering vs Deferred Rendering*. Retrieved from:
https://gamedevelopment.tutsplus.com/articles/forward-rendering-vs-deferred-rendering--gamedev-12342

22. Pharr, Matt, Humphreys, Greg & Jakob, Wenzel. (2016). *Physically Based Rendering: From theory to implementation.* Morgan Kaufmann Publishers In.

23. Seed, EA. (2018). Project Pica Pica.
Retrieved from: https://www.ea.com/seed/news/seed-project-picapica

24. Singh, Parminder. (2016). *Learning Vulkan, 1st edition*. Safari, an O'Reilly Media Company

25. The Cherno. (2018, 30th September). *Game Engine.* Retrieved from: https://www.youtube.com/playlist?list=PLlrATfBNZ98dC-V-N3m0Go4deliWHPFwT

26. The Khronos Group. (2014). A Specification (with all registered Vulkan extensions). Retrieved from: https://www.khronos.org/registry/vulkan/specs/1.1-extensions/html/vkspec.html

27. The Khronos Group. (2017). KTX. File format for storing GPU-ready texture data. Retrieved from: https://github.com/KhronosGroup/KTX-Specification

28. Willems, Sascha. (2018). Vulkan. Collection of open source C++ examples for Vulkan. Retrieved from: https://github.com/SaschaWillems/Vulkan

8. Glossary

- **Bloat:** (Software Bloat) is the process whereby successive versions of a program become slower, use more memory, disk space or processing power perceptively.
- **Concurrent:** (when related to computing) is a form of computing in which several computations are executed during overlapping time periods instead of sequentially, with one completing before the next starts.
- **Denoising**: (or noise reduction) Process of removing noise from a signal. Noise reduction techniques exist for audio and images.
- **DLL:** Dynamic-Link Library, is Microsoft's implementation of the shared library concept in the Microsoft Windows operating systems.
- **HDR:** The term refers to HDRI (High-Dynamic-Range Imaging). It is the composition and tone-mapping of images to extent the dynamic range beyond the native capability of the capturing device.
- **LunarG:** Software company specialized in device driver development for video cards. Developers of tools and infrastructure for the Vulkan graphics API.
- **MS-DOS:** Microsoft Disk Operating System, is a x86-based operating system for personal computers developed by Microsoft (1981).

9. Appendix

9.1. Appendix A - Project Specification

PROJECT SPECIFICATION - Project (Technical Computing) 2019/20

Student:	**Oscar Mari-Jurado**
Date:	**24-10-2019**
Supervisor:	**Steve Oldacre**
Degree Course:	**Computer Science for Games**
Title of Project:	**Making Real-Time Ray Tracing Possible. From scratch VulKan Hybrid Rasterizer - Ray Tracing render.**

Elaboration

In order to create real-time scenes with better quality and realism but allowing hardware setups with lower power to render them I will develop a render/engine from scratch, using VulKan API. The render will have the basic features of lighting, shadowing and materials but data-oriented and prepared for further development on ray tracing. Since then, new features could be added or even improved by adding researched ray tracing features.

Project Aims

- VulKan API development.
- Multithreading / Scheduling.
- Oriented Data Design.
- Deferred shading rendering method.
- Physically Based Rendering.
- Post-processing.
- Ray tracing.
- Different scenes showing the render features.
- Each benchmark will show the features and quantities on the scene.
- Scenes with instanced big quantities of objects with benchmarking purposes.
- Benchmarking objective: Res: 1920x1080p / FPS: 30 (minimum) - 60 (optimum).

Project deliverable(s)

- Custom, multi-threaded, rasterizer render / engine with VulKan back-end.
- Development log-book and documentation.
- Benchmarking data over different given scenes.
- Documentation over Ray Tracing implementations on the render.
- Trello. Project administration tool (https://trello.com/b/2cdS4zfb/shu-project-tech-comp).
- GitHub. Version control tool (https://github.com/marijuESAT/hybrid_render_assessment).

Action plan

Tasks	Days spent	Start date
API design and implementation (GLFW + VulKan).	10	26-10-2019
HAL basics (types definition and Timer class).	2	05-11-2019
System and components (Transform, Render, Light).	6	07-11-2019
Creation of simple geometries.	4	17-11-2019
Material system and PBR.	15	21-11-2019
Information Review	1	06-12-2019
Task scheduling system.	14	07-12-2019
Deferred shading system.	12	06-01-2020
ImGui embedding and Basic UI implementation.	6	18-01-2020
Geometry loading.	4	24-01-2020
Shadow mapping.	6	28-01-2020
Performance tests.	2	03-02-2020
Post-processing and grayscale post-pro.	10	05-02-2020
Bloom post-pro.	6	15-02-2020
The provisional contents page.	1	21-02-2020
Project documentation writing.	12	22-02-2020
Ray tracing research and impl. theory documents.	22	05-03-2020
The draft Critical Evaluation.	1	27-03-2020
Sections of a Draft Report.	1	27-03-2020
Submit the body of the project report to Turnitin.	1	22-04-2020
Submit the Project Report (physical and electronic) and copies of the deliverable in the report and on BB or the Q drive.	1	23-04-2020
Demonstration of your work.	1	12-05-2020

9.2. Appendix B - Ethics Form

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: Oscar Mari-Jurado

Publication of Work

I confirm that I understand the "Guidance on Publication Procedures" as described on the Bb site for the module.

Signature: Oscar Mari-Jurado

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: Oscar Mari-Jurado

Ethics

Complete the SHUREC 7(research ethics checklist for students) form below. If you think that your project may include ethical issues that need resolving (working with vulnerable people, testing procedures etc.)then discuss this with your supervisor as soon as possible and comment further here.

Both you and your supervisor need to sign the completed SHUREC 7 form.

Please contact the project co-ordinator if further advice is needed.

RESEARCH ETHICSCHECKLIST FOR STUDENTS (SHUREC7)

Thisformisdesignedtohelpstudents and their supervisorstocompleteanethicalscrutinyofproposed research. The SHU ResearchEthicsPolicyshouldbeconsultedbeforecompletingtheform.

Answeringthe questionsbelowwillhelpyoudecidewhetheryourproposedresearchrequires ethicalreview bya Designated Research Ethics Working Group.

The finalresponsibility forensuringthatethicalresearchpracticesare followedrestswiththe supervisor forstudent research.

Notethatstudentsandstaffareresponsibleformakingsuitablearrangementsforkeepingdata secure and, if relevant, for keeping the identity of participants anonymous. Theyare also responsiblefor followingSHU guidelinesabout dataencryption and research data management.

The formalsoenablestheUniversityandFacultyto keepa recordconfirmingthatresearch conductedhasbeensubjectedtoethical scrutiny.

Forstudentprojects,theformmaybecompletedbythestudentandthesupervisorand/or moduleleader(asapplicable).Inallcases,itshouldbecounter-signedbythesupervisor and/or module leader, and kept as a record showing that ethical scrutiny has occurred. Studentsshouldretainacopy forinclusionintheirresearchprojects,andstaffshouldkeep a copy in thestudentfile.

Pleasenote if itmaybenecessarytoconduct ahealthandsafetyriskassessmentforthe proposedresearch. Furtherinformationcanbeobtainedfrom theFacultySafety Co-ordinator.

GeneralDetails

Nameofstudent	Oscar Mari-Jurado
SHU email address	b9019210@my.shu.ac.uk
Courseorqualification (student)	Computer Science for Games
Nameofsupervisor	Steve Oldacre
email address	s.oldacre@shu.ac.uk
Titleofproposedresearch	Making Real-Time Ray Tracing Possible. From scratch VulKan Hybrid Rasterizer - Ray Tracing render.
Proposedstartdate	26-10-2019
Proposedenddate	05-03-2020
Briefoutlineofresearchto include,rationale& aims(250-500words).	In order to create real-time scenes with better quality and realism but allowing hardware setups with lower power to render them I will develop a render/engine from scratch, using VulKan API. The render will have the basic features of lighting, shadowing and materials but data-oriented and prepared for further development on ray tracing. Since then, new features could be added or even improved by adding researched ray tracing features.
Where data is collected from individuals, outline the nature of data, details of anonymisation,	

storage and disposal procedures if required (250-500 words).	

1.HealthRelated Research InvolvingtheNHSorSocial Care/ CommunityCareorthe Criminal JusticeServiceorwith researchparticipantsunabletoprovideinformedconsent

Question	Yes/No
1.Does theresearchinvolve? • Patientsrecruitedbecauseoftheir pastorpresentuseoftheNHS or Social Care • Relatives/carersofpatientsrecruitedbecauseof theirpastorpresentuse oftheNHS orSocial Care • Accesstodata, organsorother bodilymaterial of pastorpresent NHS patients • Foetalmaterial andIVFinvolvingNHS patients • TherecentlydeadinNHS premises • Prisonersorotherswithin thecriminal justicesystem recruitedfor health-relatedresearch***** • Police,court officials, prisonersorotherswithin thecriminal justicesystem***** • Participantswhoareunable toprovideinformed consentdue totheir incapacityeveniftheproject isnothealthrelated	No.
2. Isthisaresearchproject asopposedtoserviceevaluationoraudit? *ForNHS definitionspleaseseethefollowingwebsite* http://www.hra.nhs.uk/documents/2013/09/defining-research.pdf	No.

Ifyouhaveanswered**YES**toquestions**1& 2**thenyou**must** seek theappropriateexternal approvalsfrom theNHS, Social Care or the National Offender Management Service (NOMS)under their independent Research Governanceschemes.Furtherinformationisprovidedbelow.
NHS https://www.myresearchproject.org.uk/Signin.aspx

*****All prisonprojectsalsoneedNational OffenderManagementService (NOMS)Approval and Governor'sApprovalandmayneedMinistryofJusticeapproval.Further guidanceat: http://www.hra.nhs.uk/research-community/applying-for-approvals/national-offender-management-service-noms/

NBFRECsprovideIndependentScientificReviewforNHSorSCresearchandinitialscrutiny for ethicsapplicationsasrequired foruniversitysponsorshipoftheresearch.Applicantscanusethe NHS pro-formaandsubmitthis initiallytotheirFREC.

2.ResearchwithHumanParticipants

Question	Yes/No
Does theresearchinvolvehumanparticipants?Thisincludessurveys, questionnaires, observingbehaviouretc.	No.

Question	Yes/No
1. *Note If YES, thenpleaseanswer questions2to10* *If NO,pleasegotoSection3*	
2. Will anyoftheparticipantsbe vulnerable? *Note: Vulnerable'peopleinclude children and youngpeople,peoplewithlearning disabilities,people whomaybelimitedbyageor sickness,etc. See definition on website*	
3. Aredrugs, placebosorother substances(e.g. foodsubstances, vitamins) to beadministeredtothestudy participantsorwill thestudy involveinvasive,intrusiveorpotentiallyharmful proceduresofanykind?	
4. Will tissuesamples(includingblood) beobtainedfrom participants?	
5. Ispainor morethanmilddiscomfortlikelytoresult from thestudy?	
6. Will the study involveprolongedor repetitive testing?	
7. Isthereanyreasonableandforeseeable riskofphysical oremotional harm to anyoftheparticipants? *Note: Harm maybecausedbydistressingorintrusiveinterviewquestions, uncomfortableproceduresinvolving theparticipant, invasionof privacy, topics relating tohighlypersonal information, topicsrelating toillegal activity, etc.*	
8. Will anyonebetakingpart without givingtheir informedconsent?	
9. Isit covert research? *Note: 'Covert research'refers to researchthat isconductedwithout theknowledge ofparticipants.*	
10. Will theresearchoutputallowidentificationofany individual whohasnot given their expressconsent tobeidentified?	

Ifyouansered**YESonly**toquestion**1,**the checklist should be saved and any course procedures for submission followed. IfyouhaveanseredYES toanyoftheotherquestionsyouare**required** tosubmitaSHUREC8A (or8B) to the FREC.IfyouanseredYESto question**8**and participants cannotprovideinformed consentdue to theirincapacityyoumustobtaintheappropriateapprovals from theNHSresearchgovernancesystem. Your supervisor will advise.

3.ResearchinOrganisations

Question	Yes/No
1. Will theresearchinvolveworkingwith/withinanorganisation(e.g. school, business,charity, museum,governmentdepartment, international agency, etc.)?	No.
2. IfyouanseredYES toquestion1, do youhavegrantedaccesstoconduct theresearch? *If YES,studentspleaseshow evidencetoyour supervisor. PI shouldretain safely.*	

3. IfyouansweredNO toquestion2, isit because: A. youhavenotyet asked B. youhaveaskedandnotyet receivedananswer C. youhaveaskedandbeen refusedaccess. *Note: You will onlybeable to start theresearchwhen youhavebeengranted access.*	

4.ResearchwithProductsandArtefacts

Question	**Yes/No**
1.Will theresearchinvolveworkingwithcopyrighteddocuments, films, broadcasts, photographs,artworks, designs, products, programmes, databases, networks,processes, existing datasetsorsecuredata?	Yes.
2.IfyouansweredYES toquestion1, arethematerialsyouintend touseinthe publicdomain? *Notes: 'In thepublicdomain'doesnot meanthesamethingas'publiclyaccessible'.* • *Informationwhichis'inthepublicdomain' isnolongerprotectedby copyright (i.e.copyrighthaseither expiredor been waived)andcanbe usedwithout permission.* • *Informationwhichis'publiclyaccessible'(e.g. TVbroadcasts, websites, artworks, newspapers)isavailable for anyoneto consult/view. Itisstill protectedbycopyright evenif thereisnocopyrightnotice. InUK law, copyright protectionisautomaticanddoesnot requireacopyright statement, althoughit isalwaysgoodpracticetoprovideone. It is necessarytocheckthetermsandconditionsofuse tofindout exactly how thematerialmaybereusedetc.* *If youansweredYEStoquestion1, beawarethatyoumayneedtoconsider other ethicscodes.Forexample, whenconducting Internet research, consult thecodeof theAssociationof InternetResearchers; foreducational research, consult theCodeof Ethicsof theBritishEducational Research Association.*	Yes.
3. IfyouansweredNO toquestion2, do youhaveexplicitpermissiontouse thesematerialsasdata? *If YES,pleaseshowevidence toyour supervisor.*	
4.IfyouansweredNO toquestion3, isit because: A.youhavenotyet askedpermission B.youhaveaskedandnotyet receivedandanswer C.youhaveaskedandbeen refusedaccess. *Note: You will onlybeable to start theresearchwhen youhavebeengranted permissiontousethespecified material.*	**A/B/C**

Adherence toSHU policyandprocedures

Personalstatement

Icanconfirm that: – Ihave readtheSheffieldHallamUniversity ResearchEthicsPolicyandProcedures – Iagreetoabideby itsprinciples.	
Student	
Name: Oscar Mari-Jurado	Date: 23-10-2019
Signature: Oscar Mari-Jurado	
Supervisororotherpersongivingethicalsign-off	
Icanconfirm that completionofthisform hasnot identifiedtheneedforethical approval by theFREC oranNHS,Social Careor otherexternal REC. Theresearchwill notcommence until anyapprovals requiredunderSections3& 4havebeen received.	
Name: Steve Oldacre	Date: 24/10/2019
Signature: Steve Oldacre	