|  |
| --- |
|  |
| Object Oriented Design vs Data Oriented Design |
| BSc Computer Games Programming  School of Computing  Teesside University  Middlesbrough  TS1 3BA  Final Year Project |
|  |
| **Julio Rodrigo Magraner** |
| **V8086522** |

|  |
| --- |
|  |

Contents

[Acknowledgements 4](#_Toc7608249)

[Abstract 5](#_Toc7608250)

[1 Introduction 6](#_Toc7608251)

[1.1 Project Summary and Goals 6](#_Toc7608252)

[1.2 Prodruct 7](#_Toc7608253)

[1.2.1 Overwiew 7](#_Toc7608254)

[1.2.2 Inspiration 7](#_Toc7608255)

[1.3 Methodology 8](#_Toc7608256)

[2 Research 9](#_Toc7608257)

[2.1 Vulkan 9](#_Toc7608258)

[2.2 OpenGL 10](#_Toc7608259)

[2.3 Data Oriented Design 11](#_Toc7608260)

[2.4 Introduction to hardware 12](#_Toc7608261)

[2.4.1 How cache works 12](#_Toc7608262)

[2.5 GLTF 13](#_Toc7608263)

[3. Design and Implementation 14](#_Toc7608264)

[3.1 GLTF Implementation 14](#_Toc7608265)

[3.2 Lights 15](#_Toc7608266)

[3.3 Render Passes 15](#_Toc7608267)

[3.4 Scheduler 16](#_Toc7608268)

[3.5 Memory Manager 17](#_Toc7608269)

[3.6 Input Manager 18](#_Toc7608270)

[3.7 Material 18](#_Toc7608271)

[3.8 Components 18](#_Toc7608272)

[4. Results 18](#_Toc7608273)

[5. Evaluation 18](#_Toc7608274)

[6. Bibliography 18](#_Toc7608275)

[7. Appendices 18](#_Toc7608276)

# Acknowledgements

I would like to thank all the people who have supported me, endured and encouraged throughout this trip. And especially I am thankful with my family for giving to me this great opportunity to study in a foreign country.

# Abstract

My project aims to design two engines written in two different graphic API’s, one in OpenGL and the other in Vulkan. Each engine will be developed using different programming pattern, one of them in OOD and the other in DOD.

During the initial research stage I looked into existing engines that have being programmed using “Data Oriented Design” and examples using Vulkan graphic API’s. During my research I learned a lot about how the hardware has a huge impact about how the programming code has to be written to obtain the maximum performance in your program. Another knowledge obtained was about using Vulkan API’s and how to implement its pipeline, which heavily different from OpenGL.

Testing was carried out throughout development each individual feature, ensuring that each one was working well, and then put them all together ensuring a proper work.

# 1 Introduction

## 1.1 Project Summary and Goals

During the past years the hardware has been improving continuously, both in the CPU and the GPU, to the point of having memory and cache itself in graphics cards and tri-level cache in CPU’s and nine cores. However the main programming language used in most programs has not been able to adapt properly creating a dissonance between hardware and software. This gap has been increasing during the last ten years, when the CPU started to add more cores and cache level as a result of losing performance requirements.

The core of the problem comes from the programming language “C”, C was created by Dennis Ritchie between 1969 -1972, is a a [general-purpose](https://en.wikipedia.org/wiki/General-purpose_language), [imperative](https://en.wikipedia.org/wiki/Imperative_programming) computer [programming language](https://en.wikipedia.org/wiki/Programming_language), supporting [structured programming](https://en.wikipedia.org/wiki/Structured_programming), [lexical variable scope](https://en.wikipedia.org/wiki/Lexical_variable_scope) and [recursion](https://en.wikipedia.org/wiki/Recursion_(computer_science)). However “C” was not developed for mono-thread and multi-access memory. This problem was “solved” creating some access memory variable, which avoid memory races and a new standard library for implement threads. These solutions have been working “well” during past years, but the cache problem was not fixed and it has even gotten worse. The problem was derived from a loss of knowledge, on the part of the programmer, about how the cache, compilers and memory access works. This caused a constant kill against the cache and losing performance in the absence of the proper way to program and algorithm orientation.

Currently there are extremely powerful engines in the game industry such as Unreal Engine 4, Unity, Cry Engine, Frostbite, etc... All of them with a quality and performance expected from an AAA. Despite of having such amazing features they are too complex to be understood by a junior programmer; furthermore most of them use “**Object Oriented Design**”, which creates a huge inheritance tree and complexity to look for from where the functionality comes (this problem can be seen in Unreal Engine 4).

This document aims to explain superficially about how cache works currently and a proper way to create our classes inside our program.

At the end of the project I will review the results obtained in the test and judge myself based on how intuitive is the API created in each engine created, and the final output obtained from both engines.

The project goals:

* Show the difference between OOD and DOD.
* Create two tiny engines with both designs.
* Learn to use Vulkan.
* Learn deeply about multi-threading.

## 1.2 Prodruct

### 1.2.1 Overwiew

For my project I am going to compare two engines I have developed, one of them have being written using OpenGL, but each of them with different code structures, one in DOD and the other in OOD. The idea of this project is create a new render engine one using Vulkan as a GPU api and making use of the DOD code structure and then compare its performance with the others.

The main idea is create a render engine using Vulkan, make profiling test and then compare it with the previous engines created. *Phantom*, the engine’s names, will give support to forward, deferred, gltf. Having enough time making use of ImGui, a tiny editor will be created, but not to with much priority. The libraries I am going to use are: Vulkan SDK, glm (for the mathematics) , gltfw (window creation) and tinygltf.

I will be making my work as modular as possible so that all the actual simulation code will be able to be reused with a different rendering system, this will allow me to use it for future games.

### 1.2.2 Inspiration

My first programming language I learned was Assembly; my father gave to me a book about the CPU commands of Commodore 64. I was only twelve years old and for me all worked using Assembly, all changed when I went to university, where I meet C, it was like a paradise for me, tones of memory, possibility to create “infinite” variable, without worries about having to read and write from memory each time, etc... But then all I have learned had its advantages, I was able to translate the code in my mind to assembly and see bottle necks quite fast. Another thing I saw was the ignorance of the operation of the processors by the programmers. I was not capable of conceiving a world where a mechanic could not handle the screwdriver; however I was in front of many. This problem inspired me to create a document, where I saw how to program using good practices and help the hardware in its job.

## 1.3 Methodology

The methodology chosen for the development of this project has been "Scrum", which allows us to improvise and make changes continuously. In addition, the sprints have been placed weekly coinciding with the tutor's meetings and where the work done will be said and what will be done.

**Scrum**

Scrum is an agile development model. Work is divided into a small phases known as sprints. Sprints will usually last 2-4 weeks, where teams will work on high priority requirements and by the end of the sprint they will have a potentially shippable product increment.

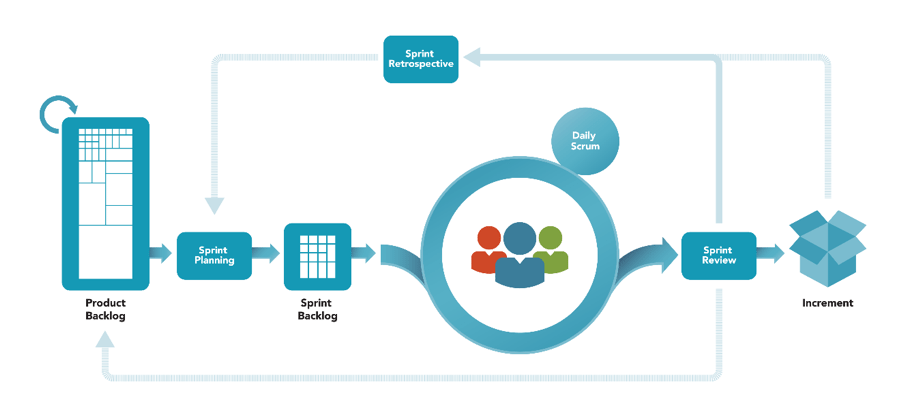
[](https://jeronimopalacios.com/scrum/)

Figure Scrum Overview

**Advantages**

Scrum can help teams complete project deliverables quickly and efficiently:

* Scrum ensures effective use of time and money.
* Large project are divide into easily manageable sprints.
* Developments are coded and tested during the sprint review.
* Works well for fast-moving development projects.
* The team gets clear visibility through scrum meetings.
* Scrum, being agile, adapts feedback from customers and stakeholders.
* Short sprints enable changes based on feedback a lot more easily
* The individual effort of each team member is visible during daily scrum meetings.

**Disadvantages**

* Lack of final date in some cases.
* The project failure is potentially linked to the individual cooperation.
* Scrum does not work well with large teams.
* Can be only successfully with experienced team members.
* Daily meetings can create some frustration.
* Member surrender can lead to a failure or produce a huge impact in the project.
* Quality is difficult to obtain, only throw many testing process is possible.

# 2 Research

## 2.1 Vulkan

Vulkan is a low level, cross-platform 3D graphics and computing API. Vulkan was created to obtain high-performance real time 3D graphics applications. Compared to OpenGL and Direct3D, Vulkan offers a higher performance and GPU/CPU connection. In addition Vulkan is intended to work well with multi-threading as a result of using command buffers. One year ago Khronos Group and GPUOpen announced project V-EZ, it aims to reduce the complexity of the IDE syntax within the Vulkan API.

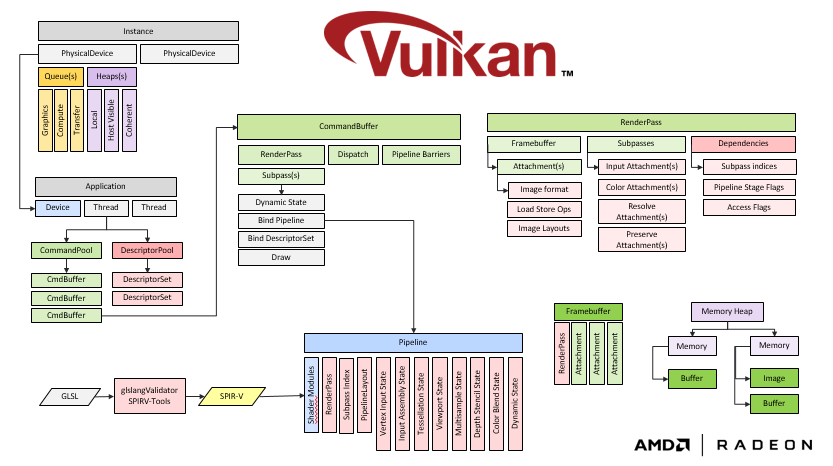
[](https://gpuopen.com/v-ez-brings-easy-mode-vulkan/)

Figure 2: Vulkan pipeline

How to create a pipeline:

1. Create an instance.
2. Select how many physical device you are going to use.
3. Create default frame buffer and swap chain.
4. Create the command pool.
5. Create the render pass.
6. Create the pipeline (material), with the uniforms, shaders is going to be used.
7. Create the mesh data (vertex, index, etc...)
8. Create the command buffers.
9. Add the commands to the command buffers.

This system allow the code to be multi-thread and each thread can have its own command buffer, then at the end of the frame, submit the command buffer to the main thread and this one submit the info to the GPU with all the synchronization objects needed.

## 2.2 OpenGL

Opposite to Vulkan, OpenGL works as a state machine and a lot of the states have a default state, this does not happen in Vulkan. OpenGL offers an easier API and quite simpler than Vukan.

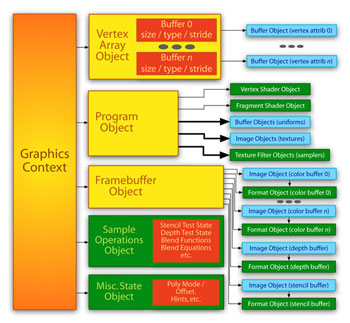
[](https://www.opengl.org/pipeline/article/vol004_4/)

Figure 3: OpenGL pipeline

How to create a pipeline:

1. Create a program.
2. Create the shaders and attach them.
3. Create mesh data (Vertex, index, etc...)
4. Bind the buffers and draw.

As you can see OpenGL requires fewer steps to render something in your screen, however the programmer lose some control about how the memory is treated and there is not possibility to set the render pass before start to draw. The render pass is created along all the render calls you are doing and which Frame Buffer Object you are binding. This absence of pipeline pre-assembly reduces the performance of the GPU and reduces the number all features, which Vulkan offers to the user.

## 2.3 Data Oriented Design

Data oriented design is a program optimization motivated by efficient usage of the CPU cache used in video game development. The approach is to focus in the data layout and how the code involves the memory access. To achieve this, the data is separated and sorted in different fields according to when they are accessed. This is easier to see with an example:

Imagine you have 6 bots aiming to you:

void updateAim(Vec3 target) {

m\_aimDirection = dot3(m\_position, target) \* m\_mod;

}

The example is a code implemented inside a class and then having an AI manager, each agent is called to aim the player; it will create this memory access and cached data:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| iCache ~ 600 | m\_position ~ 600 | m\_mod ~ 600 | Dot ~ 20 | aimDir ~ 100 |
| Total: | | | | 1920 cycles |
| Total with 6 bots: | | | | 11.520 cycles |

As you can see we are using many resources only to aim the player, now imagine computing if you have to shoot, the path finding, etc... And the fastest answer is always doing it multi-thread, but with the new design that is no necessary at all.

void updateAims(float\* aimDir, const AimingData\* aim, Vec3 target, uint count) {

for (uint i = 0; i < count; ++i)

{

aimDir[i] = dot3(aim->positions[i],target) \* aim->mod[i];

}

}

The code in this case is quite similar that the other one, but now we have the same data type saved in an array, creating a continual memory access.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| iCache ~ 600 | m\_position ~ 600 | m\_mod ~ 600 | Dot ~ 20 | aimDir ~ 100 |
| Total: | | | | 1920 cycles |
| Total with 6 bots: | | | | 1920 cycles |

Now we are using only 1920 cycles and optimizing 9.600 cycles, which can be used in other tasks.

## 2.4 Introduction to hardware

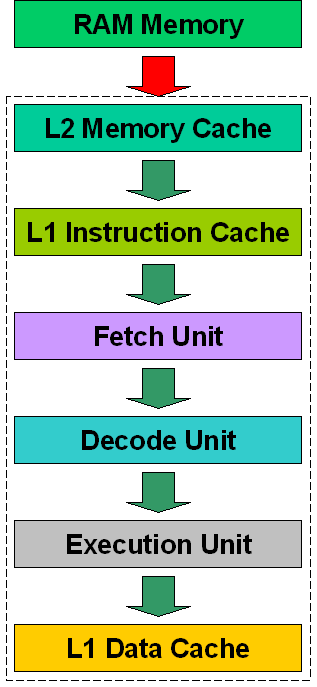
The CPU fetches programs from the RAM memory. The problem with the RAM is with the absence of power, because all the contents are deleted, that is why it is called “volatile” memory. As a result programs and data must be stored on non-volatile memory if you want to have them back after turning off your computer.

Figure : Memory Access

However the CPU does not fetch data directly from hard disk because they are too slow in comparison, this is the main reason why the CPU needs RAM to faster memory access. Furthermore there is another memory faster than RAM and it is the Cache, having even three levels each one closer to the processor. But the problem is not only the transfer rate, but also latency (“access time”) is how much time the memory delays in giving back the data the CPU asked for.

## 2.4.1 How cache works

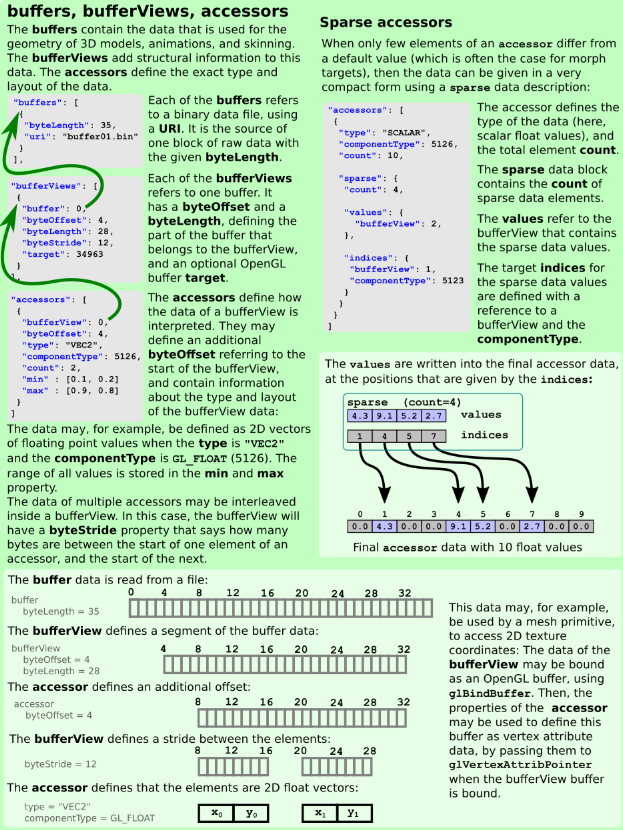
We call a “hit” when the CPU loads a required instruction or data from the cache and a “miss” if it is not there and the CPU needs to access the RAM memory directly.

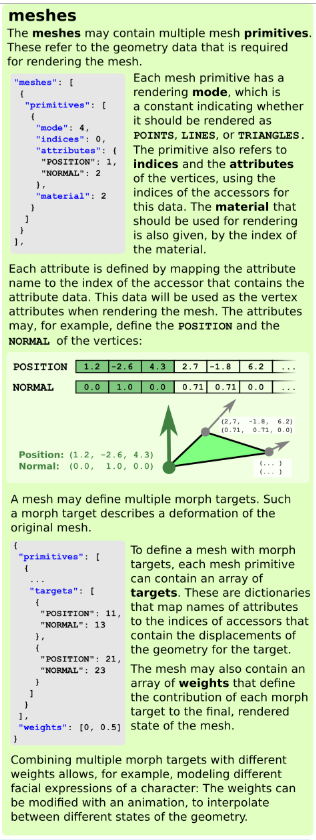
Of course when you run for the first time an application the cache will be “empty” (there is garbage in there”). But after the first instruction is loaded all the funny things start to happen.

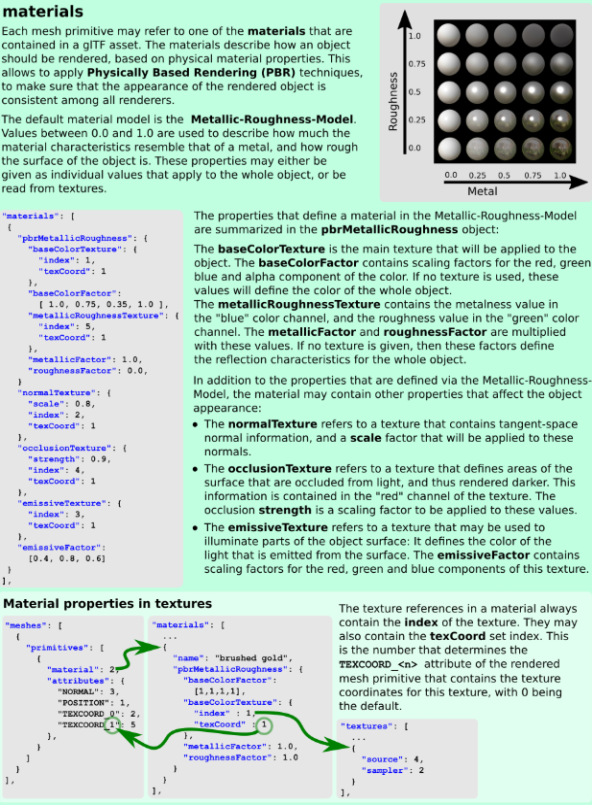
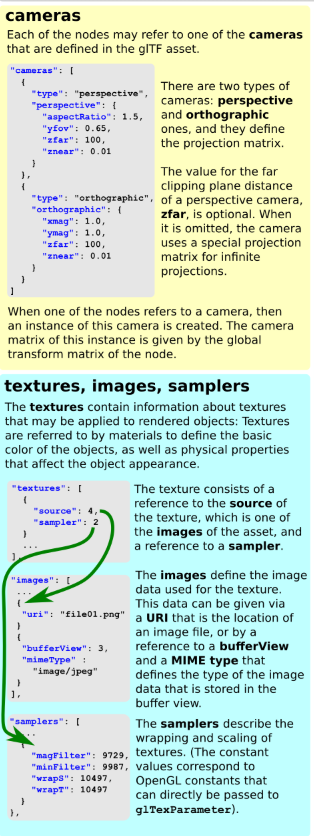
“[When the CPU loads an instruction from a certain memory position, a circuit called memory cache controller loads into the memory cache a small block of data below the current position that the CPU has just loaded. Since usually programs flow in a sequential way, the next memory position the CPU will request will probably be the position immediately below the memory position that it has just loaded. Since the memory cache controller already loaded some data below the first memory position read by the CPU, the next data will probably be inside the memory cache, so the CPU doesn’t need to go outside to grab the data: it is already loaded inside in the memory cache embedded in the CPU, which it can access at its internal clock rate](https://en.wikipedia.org/wiki/Cache_(computing)).”

This amount of data is called word size or line and is usually 64 bytes long. Even being such a tiny amount of data, the memory controller is all the time trying to guess what the CPU will ask next. The pre-fetcher will load more data located after these first 64 bytes from RAM into the memory cache. If the program continues to load instruction or read data from memory positions close or in a sequential way, the memory access or load instruction will require less cycles.

## 2.5 GLTF

[](https://raw.githubusercontent.com/KhronosGroup/glTF/master/specification/2.0/figures/gltfOverview-2.0.0b.png)“glTF™ (GL Transmission Format) is a royalty-free specification for the efficient transmission and loading of 3D scenes and models by applications. glTF minimizes both the size of 3D assets, and the runtime processing needed to unpack and use those assets. glTF defines an extensible, common publishing format for 3D content tools and services that streamlines authoring workflows and enables interoperable use of content across the industry.”

[](https://raw.githubusercontent.com/KhronosGroup/glTF/master/specification/2.0/figures/gltfOverview-2.0.0b.png)

[](https://raw.githubusercontent.com/KhronosGroup/glTF/master/specification/2.0/figures/gltfOverview-2.0.0b.png)

# 3. Design and Implementation

Throughout the development of the engine the design has been constantly changing, however a clear structure has remained: the management system. The engine is initialized creating the "Phanton" class, which the other managers continue with:

* RenderManager: manages everything related to the creation of graphics, drawing and initialization of the API.
* AssetManager: allows loading the GLTF files and obtaining the meshes and materials in the document.
* ThreadScheduler: manage all multi-thread system and allows async-task creation.
* InputManager: register and check user inputs.

This structure has been maintained to provide the user with totally free access to the different managers and greater usability of the different tools available.

## 3.1 GLTF Implementation

The AssetManager has been implemented in a way that was simple and intuitive, with this principle in mind a series of functions have been created:

* void loadFromGLTF(std::*string* file\_name):it allows to load gltf files and creates the mesh information and materials.
* std::*shared\_ptr*<Mesh> mesh(std::*string* name): Allows to obtain a mesh created by name. If the name was empty, then the name will be “mesh\_” + position in gltf file.
* std::*shared\_ptr*<graphics::Material> material(std::*string* name): Allows to obtain a material created by name. If the name was empty, then the name will be “default\_” + position in gltf file.

This Manager was designed following the same pattern present in the [Godot Engine](https://godotengine.org/), which is extremely simple and intuitive to the user. These guideline have been the basis a long all the engine development.

## 3.2 Lights

The light creation is quite simple; you only have to add a light data to the RenderManager and specify the light type. During the draw it will select between deferred or forward and will do the renders.

phantom::LightData data;

data.direction = glm::vec3(0.0, 0.0, -1.0f);

data.intensity = 3.0f;

data.position = glm::vec3(-100.0, 0.0, 0.0f);

engine.render\_manager\_->addLight(data, LightType::k\_Dir, material);

By this way there is not a class creation and the light is only a uniform block, which will be updated during the draw phase. The most important thing is add the uniform information to the material; this is explained in the material section.

## 3.3 Render Passes

In Vulkan a render pass represent a collection of attachments, subpasses and dependencies between the subpasses and describes how the attachments are used over the course of the subpasses. The use of a render pass in a command buffer is a render pass instance. Parts of a render pass:

* Attachments: describes the properties of an attachment including its format, sample count, and how it contents are treated at the beginning and end of each render pass instance.
* Subpass: uses an attachment if the attachment is a color, depth/stencil, resolve or input attachment for that subpass.
* Subpass Description: describes ordering restrictions between pairs of subpasses. If no dependencies are specified, implementation may reorder or overlap portions of the execution of subpasses. Dependencies limit the extent of overlap or reordering, and are defined using mask of pipeline stages and memory access types.
* Subpass dependencies chain: is a sequence of subpasses dependencies in a render pass, where the source subpass of each subpass dependency equals the destination subpass of the previous dependency.

## 3.4 Scheduler

This is the icing on the cake; the scheduler is one of the most complicated code I have ever implemented. It is inspired by the scheduler developed by cry engine.

The scheduler works as a pool thread, when an async-task is added, the scheduler creates a task, if it does not exist or push back the task to the current active task list in that thread. This current task list is thread\_local, so each thread has its own list. However other thread can add task to another task list in another thread, having to create mutex or guards sometimes.

In some case we will want to wait the current thread (not the main) to finish all current tasks, in this case we need to call sync; this function will yield the thread or execute the current task list until it is empty. If we want to sync all the thread with the main thread then calling flush will do that, flush() will yield the thread until all the threads have consumed all the tasks in their lists.

// Run pre update in sequence

{

scheduler\_.schedule(std::*bind*(&Phantom::runScenePreUpdate,this)).sync ();

}

// Run fixed updates in parallel

{

const auto dt = phantom::Time::fixedDeltaTime;

phantom::Time::elapsedTime += phantom::Time::deltaTime;

phantom::Time::remainingSimulationTime += phantom::Time::deltaTime;

const auto count = (uint32\_t)(phantom::Time::remainingSimulationTime / dt);

phantom::Time::remainingSimulationTime -= count \* dt;

scheduler\_.schedule(std::*bind*(&Phantom::runSceneFixedUpdate, this, count));

}

// Tell the scheduler to wait for all the fixed updates to finish

scheduler\_.sync();

input\_manager\_.update();

// Run updates in parallel

{

scheduler\_.schedule(std::*bind*(&Phantom::runSceneUpdate, this), true);

}

// Tell the scheduler to wait for all the updates to finish

scheduler\_.sync();

// Run post update in sequence

{

scheduler\_.schedule(std::*bind*(&Phantom::runScenePostUpdate, this)).sync();

}

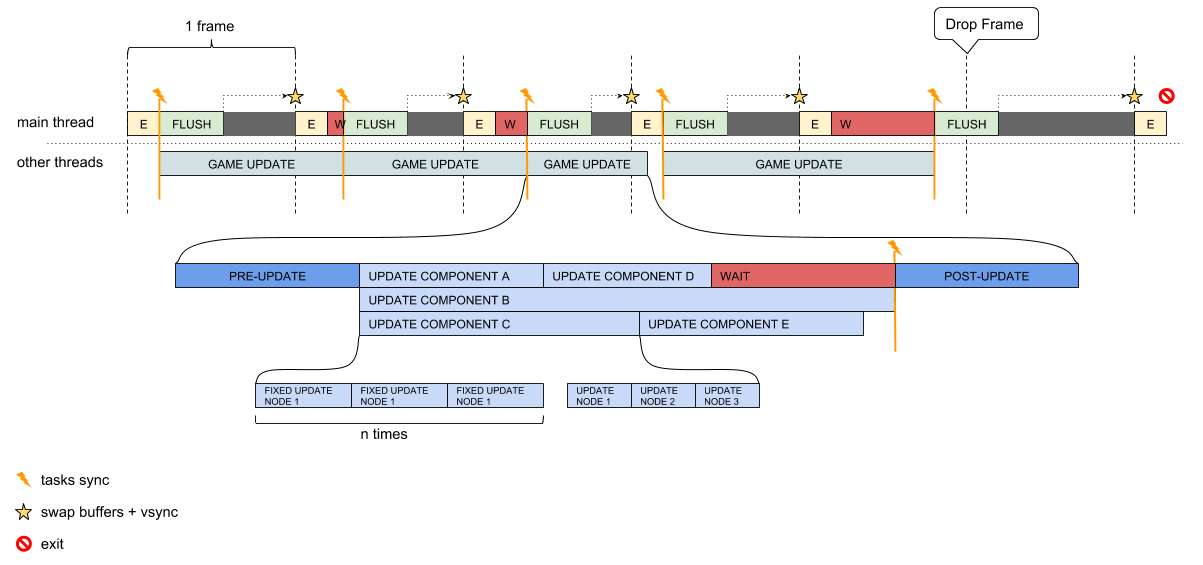


Figure Threaded Scheduler Pipeline

The only difference between both schedulers in each engine is when the main thread does flush(), one is before the update and the other after the update.

## 3.5 Memory Manager

During the research about how DOD was implemented in an engine, I reached a conclusion about it; it was too much spaghetti code. Trying to find an adaptable solution to this problem I decided to create a memory manager, which could handle groups of the same component and whose memory would be contiguous in memory. In this way I would obtain a higher performance, even if it is not entirely DOD.

The memory manager reserve chunks of memory, whose size is 100. This memory is connected to the next chunk when you overpass the max size. To create this new memory system I had to learn about how c++ uses *iterator* and how override some functionality in them and create my own. However I had to learn templates more deeply, until now I had only written some simple templates, but this time I will override something from the standard (something I have ever done). After talking with some of my previous teacher and reading some books about template programming, I created my own c++ *iterator* and doing this my own memory manager using the standard library.

This system is in memory\_region.hpp and memory\_manager.hpp

## 3.6 Input Manager

It manages all the inputs generated by the user and they are collected through the use of glfw. The user only needs to map an axis or button to be stored in a map inside the input manager when it makes update. The value will be able to be consulted when necessary and will have the minimum and maximum values seted during the mapping.

inputManager.addAxisKeyInput(translate\_input\_name, GLFW\_KEY\_A, -1.0f, <dimension>);

inputManager.addButtonMouseInput(rotate\_mouse\_button\_input\_name, GLFW\_MOUSE\_BUTTON\_RIGHT);

## 3.7 Material

To render a mesh we need to create a material and in this engine we need to add all the attribute, uniform and texture info.

To create a material all the attribute information otherwise it will throw an error and the material will not initialized. This system was chosen as been quite close to the Vulkan shader system. It reduces the code and user complexity.

struct AttributeInfo

{

ui32 binding = 0; //binding position of the buffer data

ui32 location = 0; //location inside the sahder

ui32 offset = 0;

ui32 stride = 0;

Format format = Format::k\_UNDEFINED;

InputVertexRate rate = InputVertexRate::k\_Vertex;

BufferType usage = BufferType::k\_None; //Uniform, Vertex, Index, Storage

};

When a material is loaded from gltf the location is the same as the one it occupies in the gltf primitive.

struct UniformBlockInfo

{

uint32\_t location = 0;

uint32\_t count = 0;

ShaderType type = ShaderType::k\_None;

*size\_t* bytes = 0;

std::*string* name;

};

Another thing you can add to your materials are the uniforms, which can be updated using the name when you added them. The variables are quite the same as the AttributeInfo.

Finally we have the textures only adding a std::*shared\_ptr*<Texture> with the function will do it. The only thing is the location inside the shader, it has to be after all the uniform in the shader.

## 3.8 Mesh

The mesh contains the entire vertex and index buffer data stored and created to pass to the shader and be read by the graphics pipeline.

struct MeshInfo

{

void\* data;

*size\_t* size;

BufferType usage = BufferType::k\_None;

};

As the material creation system, the mesh creation system only needs the buffer data, size and the usage

## 3.8 Components

The components are the only class you will need to inherit from if you want to customize or create your own component. Each component type will have its own memory region inside the memory manager, as explained. A node can contain only one component type, but can have many components. This limitation was added to reduce code complexity and avoid giving to me some headaches.

|  |  |  |
| --- | --- | --- |
| Transform | Movements | Camera |

node

Memory manager

|  |  |  |
| --- | --- | --- |
| Transform | - | - |

|  |  |  |
| --- | --- | --- |
| Movements | - | - |

|  |  |  |
| --- | --- | --- |
| Camera | - | - |

# 4. Results

To verify the differences between DOD and OOD a series of stress tests have been carried out. To this end, a series of limitations have been taken to improve the results obtained and to be truer. However, the results obtained will only be suitable for the computer where they were made, since depending on the type of hardware, temperature and ordering conditions, different results may be obtained. Another data to take into account the stress tests performed only affect the CPU, at no time is looking to compare the efficiency and speed of the different graphic API.

These limitations have been only the maximum frames per seconds, in this case 60 and the number of threads, three in box executions.

The only difference between both designs has been in how the components are updated in the case of DOD are grouped by systems where they are subsequently traveled contiguously in memory and more efficiently. On the other side in OOD the scene is traversed by node and is updated from top to bottom as if it were a tree, producing huge jumps in memory and continuously killing the cache.

These tests have involved the execution of expensive mathematical formulas, cos (), sin (), sqrt (), etc ... continuously and within loops to simulate complex movements and calculations within video games. Also the tools given by GLM have been used for the alignment of the memory obtaining a little improvement, although not very appreciable.

During the stress tests I was placing different amounts of "objects" until I saw a decline in the performance. This decrease begins to be noticed when you are calculating amounts close to half a million objects, this data is later written in a file whose extensions can read Excel and I can easily obtain the means.

# 5. Evaluation

Sincerely the results obtained have pleasantly surprised me even being only valid in my computer. However this shows the amount of performance that we are wasting by the mere fact of trusting blindly in the hardware and assuming that the player will have the latest computer model and in perfect conditions. Of course, in that case it would be the paradise of every programmer where we would not have to worry about anything and only write code as robots.

The industry of video games, like the c-standard, have gone by different ways, we have focused on creating super dynamic codes, when video games deal with closed environments, determined by the designers. These environments are limited by rules that we have created ourselves. Why not also do it with the code? We are constantly focused on making everything as dynamic as possible, complicating our existence and reaching a point where you do not know if you are programming a "hello world" or the best artificial intelligence in the world. We have forgotten the most important acronyms of our industry, KISS (keep it stupid simple or keep it simple stupid).

Programming is simple, relatively, but we have made it complicated we have wanted to create tools so that they can be used by as many people as possible, when at the end of it all we are going to use it ourselves.

DOD is not the industrial revolution of videogames, it is simply a return to our roots, to the real programming that we have been forgetting and that we were simply abandoning for the mere fact that it was not as dynamic as OOD, but how dynamic are our games when we are not able to get through that damn invisible wall?

# 6. Bibliography

-Overvoorde, A. (2019). *Introduction - Vulkan Tutorial*. [online] Vulkan-tutorial.com. Available at: <https://vulkan-tutorial.com> [Accessed 15 Jan. 2019].

- Khronos Group (no date) *Vulkan Reference Pages*. Available at: https://www.khronos.org/vulkan/ (Accessed: 15 January 2019).

-Gamesfromwithin.com. (2019). *Data-Oriented Design (Or Why You Might Be Shooting Yourself in The Foot With OOP) – Games from Within*. [online] Available at: [http://gamesfromwithin.com/data-oriented-design](http://gamesfromwithin.com/data-oriented-design%20) [Accessed 15 Jan. 2019].

-Dice.se. (2019). [online] Available at: <https://www.dice.se/wp-content/uploads/2014/12/Introduction_to_Data-Oriented_Design.pdf> [Accessed 15 Jan. 2019].

-YouTube. (2019). *CppCon 2018: Stoyan Nikolov “OOP Is Dead, Long Live Data-oriented Design”*. [online] Available at: <https://www.youtube.com/watch?v=yy8jQgmhbAU> [Accessed 15 Jan. 2019].

-Gregory, J. (2018). *Game engine architecture*.

-Akenine-Möller, T., Haines, E. and Hoffman, N. (2014). *Real-Time Rendering, Fourth Edition*.