Writing a graphics API abstraction

# By Jan Kind

# Introduction

During year 2 block C I will be spending 20% of my time on a personal project. I have chosen to write a graphics API abstraction that allows you to quickly set up rendering pipelines without being exposed to any platform specific code. I will model this abstraction after the modern graphics API Direct3D12, though I will be implementing it in OpenGL due to time restrictions. I will be striving for an abstraction that would allow for a Direct3D12 implementation if I so desired.

My goal is to end up with a graphics framework that I can use to quickly set up the rendering pipelines required for future projects. I plan to continue improving this project after the block ends, and use it to create simple games and simulations.

# Intended Learning Outcomes

Starting this project, I have a fair bit of knowledge about OpenGL. I have used Direct3D12 in the first block of this year, which has given me a decent insight in how the API works. I want to expand on this knowledge by writing an interface that can apply to both API’s. There is still areas of Direct3D12 that are vague to me, and I hope to clear these up more by reading documentation and asking for information from my peers and lecturers.

I also hope to improve my software engineering skills by considering the best possible approaches to creating this abstraction. My data structures have to be optimized while also being easy to use. I will have to weigh my options and find a middle ground.

## SMART Goal

**S**. I want to study Direct3D12 and OpenGL to the point where I can confidently write an abstraction layer for both API’s that allows me to quickly set up a rendering pipeline that allows me to easily select shader programs, send data to the GPU and perform draw calls. I want to be able to render to a selectable target (for example a window or an image).

**M**. I will know that I am progressing towards this goal if I am learning new things and finding answers to questions I have. My goal has been completed when I can use my abstraction to send data to the GPU and perform draw calls.

**A**. I already have knowledge about OpenGL and Direct3D12. I have studied this topic before and been pondering over it for many months. It’s a project I’ve been wanting to do for a while so I think I am as prepared as I can be.

**R**. My original goal was to also make a tech demo with the API. I have cut that part to ensure that I don’t over-scope. Just writing the abstraction should be doable if I don’t overcomplicate it. I will set strict requirements for the project which I will follow, and I will also set features I will not support for the sake of simplicity. For example resource state tracking and multi-threaded drawing are features I will not support because they would complicate the API a lot.

**T**. I aim to have a working product as described in the requirements in week 8. I will aim for working prototypes with limited functionality early on to verify the pipeline. I will then expand on that.

# Requirements

## Included

* The project should provide a platform independent API that hides away any underlying implementation details. This should be achieved by setting up multiple projects. The main project would provide the public API and datatypes required. The sub projects contain the platform specific implementations.
* The API should give the user the possibility to load resources such as shaders, textures and meshes into GPU memory. These resources should then be usable by binding them to the pipeline.
* The pipeline should offer a render target. This could be a window or texture. The window should be provided by the API when requested with configurable parameters.
* The API should be structured after Direct3D12 in the sense that all drawing commands are deferred. A command-list like structure should be used to queue up draw commands like setting pipeline state, binding resources, setting a target and finally doing a draw call. This deferred command-list can then be executed.
* The API should provide a way to send data to GPU buffers. These buffers can be used from the shaders.
* Shaders are platform specific. Some will be GLSL and others HLSL. I will need to provide a framework that can load the correct shader file depending on the current implementation.
* Like Direct3D12’s Root Signatures and Pipeline State Objects, I will need to have a way to verify the state of the pipeline. Shaders expect a specific vertex format. This means I have to find a way to ensure that the user of the API provides the correct vertex input.   
  Furthermore, shaders have samplers that can be used to access textures. They may contain ConstantBuffers, Uniforms, Shader Storage Buffer Objects, Uniform Buffer Objects and other types of input. I have to find the similarities in these types of buffers, and provide an abstraction that allows me to use these buffers and bind resources to them.

## Excluded

* The big advantage of modern graphics API’s is that they offer multi-threaded execution. While advantageous for performance, this would beat the purpose of my API. I want to create something that has acceptable performance and is easy to use and setup. The whole point of the modern API’s is to give more control to the programmer so that very low level optimizations are made possible, at the cost of very complicated interfaces. This is why I will leave out any multi-threading from my API. This also means that I won’t have to do complicated resource tracking for all resources. That allows the API I design to remain clean and easy to use.
* For this block, I only want to implement the rendering in OpenGL. Direct3D12 is interesting but would be too time-consuming. Once my API is established more, I can start looking into implementing the more modern APIs into it.

# High Level Planning

## Week 1

* Start of the project. Start thinking about what I want to achieve, and how I will achieve it. Define my smarter goals and set a high level planning. Define the requirements of the project and define what to avoid.
* Start working on a UML detailing each of the required subsystems.

## Week 2

* Research subsystems such as buffers, textures and samplers. Figure out how I can write an abstraction that would allow me to use them across multiple graphics APIs.
* Come up with data structures that would allow me to represent the expected shader input in C++. These are equivalent to a root signature and input layout in D3D12. Every feature a shader may offer has to have an equivalent abstract interface in C++. Whether this is automatically generated for the user, or whether the user has to manually specify this, is to be determined.
* Continue working on the UML. Ask around for feedback and come up with software design choices and theoretical solutions to the issues I may face. Iterate on my design with the feedback I receive.
* Start implementing some of the structures I come up with. Run into issues and iterate. The most important ones would be the command list, resource and input layout / root signature equivalents.

## Week 3

* Iterate on last weeks tasks. I am leaving this week open because I will likely run out of time.
* Create data types and enumerations to represent platform dependent types. For example floats, unsigned integers and RGBA. These will need to be mapped to their platform dependent types.

## Week 4

* Implement the core of the project. The goal is to have an abstraction in place that will allow me to open a window, bind a simple shader and draw a triangle. This would require me to have the core data structures in place. In the prior two weeks I will have implemented most of them, so this week is all about combining these systems.

## Week 5

* Once I have something on the screen, implement the possibility to switch render targets. The ability to render to texture or a window. This is an important step as it will allow me to set up multiple render passes easily. Most graphics API’s allow the fragment shader to have multiple attachments to write to. I’ll have to come up with a design that would allow for easily attaching the render target to the pipeline with the correct datatypes and resource state.

## Week 6

* Expand the support for shader data. Allow for bulk data to be uploaded to the shader through Uniforms, SSBOs and UBOs. Abstract these in a way where the OpenGL context can set up the buffers as required by the shaders. This expands on the tasks of week 2 and 3.

## Week 7

* Set up a demo where I show off the capabilities of the API. For example try to set up a pipeline where I render some simple textured cubes instanced onto another texture. Then apply something like shadow mapping or depth of field to show the possibility for multiple render passes.

## Week 8

* The last week of the project is reserved for documentation. I do not expect to work on this personal project during that week unless time permits. If I do end up having time, I’ll spend this week polishing and revisiting areas that I think can be improved.

# Task List

## To-Do

* Set up the solution with multiple projects (API and OpenGL).
* Research resource system (Shader, Texture, Buffer). DX12 and OpenGL.
* Shader managing system (Loading the right file, possible parameters?).
* API for creating a window with parameters.
* Research pipeline state. Which data is required? How do I verify if the state matches what the shader expects?
* Research and design CommandList-like structure that takes commands, then executes them in a deferred way.

## Done

## In Progress

* Create UML detailing all these systems.

# Current State

# Log

### 06/05/2020

First day of the project. I defined my goals and SMART goal. I wrote the start of this document. I made a high level planning for the rest of the block as well as detailed requirements and what I want to learn.