Sapphire Engine

Design Document

# Rendering

## High Level Architecture

### Render Graph

Is a collection of Render Passes and all the dependencies between them. It will define how each Pass is connected to one another, what is the final resolution, and so on. The Render Graph definition could be read from an XML file, which will add a great flexibility to the Renderer.

### Render Pass

The basic unit of rendering is called the Render Pass. Render Pass have at least one Render Target, into which it will produce its output. It can have some input Render Targets from other Render Passes. It will have access to the Game Object tree, so it can filter out the things it wants to process. A special case of Render Pass is a full screen pass, which only renders one triangle that covers the entire screen.

There are some specific things to each Render Pass, like a set of pipeline properties or a set of shaders. I’m considering having a list of Child Render Passes (or Render Sub-Pass), where than can share some things, but have it’s own set of properties or different shaders for instance.

### Game Objects Tree

It represents the scene you want to render. It should contain the data like geometry, a pack of textures or material properties. It can also have some meta data, so each Render Pass can filter out things it doesn’t want. For example, you can have a Light instance – you don’t want to render it, but you do want to have some information about it, like its color or position.

### Render Context

It will contain the data and functionality that is common for every Render Pass. Things like GPU capabilities, settings or maybe Resource Management.

### API Specific Backend

This layer will allow us to extend the engine to support other APIs, like OpenGL, Vulkan or even consoles.

# Sapphire Engine Programming

## Introduction

Programming Sapphire Engine is very primitive at this stage. At the very basics, you create some number of Game Objects and you register each one of them in the Engine. After this part is done, you can start the Game Loop of the Engine.

In general, you should create a new class representing your game. This class should have Sapphire::Engine member that would let you control your application and create necessary resources. On top of that, you should create a bunch of Game Objects (or classes that inherit from it) and register them, prior to Game Loop execution.

## Game Objects

Each Game Object keeps a pointer to the raw geometry data, which include vertex position in a 3D space and a list of indices of those vertices. Prior to registration, those pointers have to point at valid geometry data.

The geometry should be provided in a Model Space. The Engine will not apply any World transformations. The Engine has one camera at fixed position (0.0f, 0.0f, -8.0f), looking at the origin of the coordinate system. The camera has fixed POV and fixed width to height ratio. The perspective transformation is applied for every vertex.

## Geometry Asset Pipeline

First, you have to collect the geometry data. You can generate it or it can be read from a file from 3D software. Eventually, this data will be translated into Vertex Buffer and Index Buffer. Sapphire Engine supports only one topology, which is a Triangle List.

## Rendering

### Resources

In DirectX 12 there is a lot to be done when it comes to the resources.

### Uploading Textures

In order to properly use a texture, it has to be loaded from file on the disc (or generated automatically) and it has to end up on the GPU memory, preferably on a heap that is inaccessible to the CPU, but frequently read by the GPU. In DirectX 12, this is the Default heap.

|  |
| --- |
| [D3D12\_HEAP\_TYPE enumeration (d3d12.h)](https://docs.microsoft.com/en-us/windows/win32/api/d3d12/ne-d3d12-d3d12_heap_type) |
| The Default heap type experiences the most bandwidth for the GPU, but cannot provide CPU access. The GPU can read and write to the memory from this pool, and resource transition barriers may be changed. The majority of heaps and resources are expected to be located here, and are typically populated through resources in upload heaps. |

The creation and upload textures to the GPU happen in Render Context, during the Create Resource phase. For every Game Object we allocate SRV descriptor and we create new DX12Texture, with underlying DX12Resrouce, with a fixed size resolution.

Then, also for each such Game Object, we calculate the required size for the upload buffer, and then we create the Upload DX12Resource. Then we upload raw texture data by using the Upload function, which is effectively the Map/Unmap API functions.

Now we have both source data Resource and the destination empty Resource, we have to add a Copy Texture Region to the Command List as well as the transition of the newly uploaded texture to PIXEL\_SHADER\_RESOURCE state.

After all Game Objects being processed, we close and execute the Command List. There are some known limitations for this procedure.

* The Texture is being created with hardcoded dimensions and format
* There is really no need to create Upload buffer for every Game Object
* The Texture has to be in hardcoded R32G32B32A32\_FLOAT format to match the internal data structure. This is very big and not usual format for rendering, need to check popular formats like JPG or BMP
* There are no place for Subresources or MIP maps
* All textures are created with the same parameters

### Render Pass

One of the basic execution units in rendering is the Render Pass. The main role of a Render Pass is to populate provided Command List with appropriate commands. On top of that, Render Pass must make sure it is in the proper state and all the necessary data is there.

Each Render Pass must have a corresponding Multi Render Target that contains at least one Render Target. Render Pass can also have a mandatory Depth Buffer. Each of those buffers will be cleared during the Render Pass Setup phase with a mandatory color value.

Render Pass has predefined Setup phase, where the common resources are being set and cleared. Then each derived Render Pass has to provide the implementation for PreRender, Render and PostRender functions.

Render Pass also owns Input Layout and at least one Pipeline State object. The first Pipeline State is being bound during the Setup phase, along with the first Render Target and the Depth Buffer.

#### Design Goals

The main design goals for this class are (in importance order):

* To easily add new Render Pass and create its mandatory resources, like Render Target
* To

### Render Target

# Engine

## UC001: Engine Startup

### Description

The main goal of this Use Case is to get the engine up and running.

### Inputs

The width and height of the display window.

### Outputs

All necessary resources are initialized, the display window is created, everything is ready to run the game loop.

### Procedure

1. Execute UC002: Create The Display Window
2. Execute UC003: Create The Render System
3. Execute UC004: Create The Raw Input System
4. Execute UC005: Create The Mouse Object
5. Execute UC006: Create The Model Loader Object

## UC002: Create The Display Window

### Description

Do all the necessary steps to create a window with given dimensions to show a windows on a desktop, on which our engine will be presenting the frames.

### Inputs

The width and height of the window’s client area

### Outputs

An empty window displayed on the screen.

### Procedure

1. Register the Window Class
2. Adjust the Window’s Client Area so it fits exactly the given window dimensions
3. Create Window Instance

# Render System

## UC003: Create Render System

### Description

This Use Case will create all the necessary DirectX 12 resources to start producing the fames.

### Inputs

The window handle, the width and height of the render area.

### Outputs

### Procedure

1. Create the DXGI Factory
2. Check system capabilities
3. Enumerate all the available adapters
4. Create DirectX 12 Device
5. Create Command Queue
6. Create Command List
7. Create Swap Chain
8. Create Descriptor Heap
9. Create two Render Targets, one for Back Buffer and one for Front Buffer. Those two resources were created along with the Swap Chain.

# Command Queue

In Sapphire Engine there is only one command queue. It’s main responsibility is to execute the Command Lists, created by the renderer. The Command Queue should be responsible for the synchronization with the GPU. The synchronization is really simple – you can’t do anything on the CPU side until the Command Queue is finish executing.

## UC0004: Create Command Queue

### Description

The engine should have just one Command Queue. It is a simple, direct queue with a normal priority.

### Inputs

### Outputs

### Procedure

1. Create and store the DirectX 12 Command Queue. It should be of the type direct, with normal priority and with no additional flags.
2. Create and store the DirectX 12 Fence. It should have the default value of zero.
3. Create an empty Windows Event. Store the handle to it.
4. Flush the GPU by executing the UC0005: Flush

## UC0005: Flush

### Description

### Inputs

### Outputs

### Procedure

## UC0006: Execute Command List

### Description

### Inputs

### Outputs

### Procedure

# Command List

## UC0007: Create Command List

### Description

### Inputs

### Outputs

### Procedure

# Render Target

## UC0008: Create Render Target

### Description

### Inputs

### Outputs

### Procedure