# TDF Game Engine

## Aim

To create a simple, reusable game engine that implements component-based architecture. The main focus is to create a modular rendering loop where shader modules can be inserted (at a given reference point) and removed, from the rendering process to create visual effects.

Note: At the time of design, the only graphics library I have encountered is OpenGL and while I wish to abstract the API from the engine’s rendering system, I am not naïve enough to believe that there will not be a need for refactoring once I learn the API of another graphical library.

## Terminology definitions

|  |  |
| --- | --- |
| Term | Description |
| Texture | Represents a 2D array of pixels. The memory required is determined by the width, height and pixel format. The texture data can be statically loaded from a file, or dynamically created via procedural algorithms and as a render target (see Frame buffers). |
| Frame buffer | A buffer that stores the frame rendered data, dependant on its attachments (stencil, colour and depth). |
| Shader Program |  |
| Shader |  |
| Shader Parameters | A simple storage class that represents different type parameters that can be sent to a shader, the parameter is paired with a string representing the identity of the shader parameter. |
| Material | A collection of shader parameters that will be sent to the shader when a render call is executed on the given material. The material associates itself with a given shader, this means that for an object to be rendered by multiple shaders, it must have multiple materials. |
|  |  |
| Mesh | A collection of vectors representing attributes of a 3D object (vertices, normals, texture co-ordinates, bi-tangents etc), which must be sent to the rendering pipeline to draw the object. |

## Initial System Requirements

The idea here is to mark out the required subsystems and their responsibilities.

### File System

* Check a file exists

### Rendering System:

* Abstract OpenGL API into wrappers:
  + Textures
  + Frame buffers
  + Programs and shaders
  + Shader parameters (uniforms, uniform buffers)
  + Mesh Rendering
* Load/manage mesh data
* Load/manage texture data
* Load/manage shaders
* Create/manage frame buffers
* Create/manage light sources

### Resource Manager:

* Manage dynamic and static resources

### Resource Loader:

* Load 3D files
* Load textures
* Load shaders

## Base type use cases

By defining the base types in terms of their usages, I can better extract the derivative classes as well as dependencies and responsibilities.

### Frame buffer:

* Depth attachment only
  + 2D texture bound to the depth attachment slot, usage example: shadow map creation for a unidirectional light source.
  + Cube map levels bound to the depth attachment slot, usage example: shadow map creation for an omnidirectional light source. There must be a rule to determine which level of the cube map should be bound at a given render pass.
* Colour attachment only, depth attachment gets a generic render buffer storage
  + 2D texture bound to the colour attachment slot, usage example: planar reflective surface.
  + Cube map levels bound to the colour attachment slot, usage example: reflective object where each axis should be rendered from the objects perspective.

Problems:

* Frame buffers, which have cube map textures attached, require a rule for specifying which level of the cube map to bind.
  1. Are these rules static or can the user create rules?
  2. Whose responsibility is it to assign the rules?
  3. Should the user be able to find out the cube map direction that is currently bound? i.e. in shader logic.

Shader Parameters

## Types and their requirements

Frame buffer usages:

Depth sampling

## Code Design

### Engine initialisation

1. OS must be defined as well as graphics library plus any other external factors
2. File System instantiated
3. Resource Manager instantiated
4. Resource loader instantiated, dependencies are file system, and resource manager
   1. Resource Loader depends on File System for locating and opening the stream.
   2. Resource Loader depends on Resource Manager for:
      1. Checking if the resource exists in memory already
      2. Adding the resource to the manager

## Project Structure

The project must be separated into parts that will be reused by the user and parts that are “invisible” to the user, e.g. engine specific.

Engine Specific:

Game Loop

Rendering Loop

Visible to the user:

Rendering resources – frame buffer, mesh etc.

Physical Directory Structure

TDF Engine/src/lib – All library dependencies are here

TDF Engine/src/src – Original code

TDF Engine/src/src/RenderResources – The primitive resources that a user can instantiate for rendering purposes, as well as the factories that make them.

## Class Definition

RenderResourceFactory – a pure abstract class that provides an interface for creation of render resources. Derived classes implement these interfaces for graphic library specific functionality.

IRender – a pure abstract class that provides an interface for rendering a given object, any object that needs to send data to a shader program must implement this interface.

Framebuffer –

FramebufferBuilder – A fr

Regarding framebuffers and whether they need a builder and director class per graphics API and per possible representation….

Resource management for static objects:

A static texture will never change, therefore, when creating a static texture, it is best to has the path and store it as a pair with a weak pointer to the object. Then when creating other static textures, it will only require a lookup of the hash, and only loading the texture from file when the map does not contain the hash (or if the weak pointer does not point to an object anymore). Similarly, when creating a dynamic texture, the texture can be copied if it exists within the map.

Material is a dictionary of pairs where the first part is a string representing the uniform name, and the second is a

Who can

A program must define its material layout. Without the program, the material has no meaning.

## Event System

Define constants for individual key inputs. For each type of context API, have an array that maps the API’s keycodes to the universal key inputs. Do similar for states.

Components can implement event interfaces, when they do, the interface must subscribe to the event object.