Identity Modular Engine

(I.D.E)

By team

Looking For Group

Filipe Antunes Da Silva, Lilian Gadeau and Gabriel Meloche

Overview

This Technical Design Document (TDD) is written for the expressed purpose of defining the basis of the work that will have to be undertaken to create a video game engine for Windows.

Our goal is to create an engine that is:

- User-friendly
- Optimised (uses as little resources as possible)
- Focused on FPS game-making

Our engine will handle:

- Rendering, including shaders
- Audio with basic 2D and 3D functions
- Compatibility with Windows
- Object loading within a scene
- Physics and collisions between objects

Systems

We will code our engine using mostly the KISS principle (Keep It Simple Stupid), meaning that the architecture of the code will be as clear as possible to our programmers as well as to outside observers. This extends to the way the user will interact with the engine as well; we want to make sure that the interface is intuitive, uncluttered and simple yet elegant.

We will assuredly have an Entity Component (EC) engine architecture, but we are aiming to make it into an Entity Component System (ECS). Game objects will have components attached to them, such as meshes or transforms, and those components will only contain pointers and/or references to their data in the memory; no raw data is held directly by the components.

Our architecture will have to be modular, meaning that modules can be added to or removed from the project without breaking the program. For example, we could run the project while deactivating the audio module without any consequences on the other modules. This will ensure that programmers working on different modules would not create errors for the other workers, thus saving a significant amount of time and headaches on debugging. Theoretically, it may also permit us to enable the user to change external dependencies to their liking, e.g. using a different physics library than the one included in our engine. However, this is not a priority, nor do we expect to implement this idea.

To make this possible, we will need to wrap all of the external dependencies' functions, methods and variables in classes of our own so that the engine doesn't directly use the API. This makes it possible to swap dependencies without compromising the rest of the engine's code.

Design

Naming conventions

General	Classes
Functions : PascalCase() *ex: DoSomething(...);*	**Namespaces** : PascalCase *ex: Core::Rendering*
Local Variables : camelCase *ex: newPosition*	**Basic Classes :** PascalCase *ex: InputManager*
Parameter variables : p_camelCase *ex: p_newPosition*	**Abstract Classes** : Capital 'A' before the name *ex: AClassName*
Static Variables : static m_ camelCase *ex: static m_numberOfInstances*	**Interfaces :** Capital 'I' before the name *ex: IClassName*
Pointers : type* m_camelCase *ex: char* m_playerName* NOTE : Prefer use of smartPointers;	**Struct** : PascalCase *ex: Vector3*
Typedef : alllower *ex : typedef*	**Methods** : PascalCase *ex: DoSomething(...)*
Macros : ALLCAPS *ex: RENDERINGMODE*	**Properties**: PascalCase *ex: Property {get(){...}; set(){...};} (might change)*
Templates : template<typename PascalCase> *ex: template<typename MyType>*	**Fields (Members)** : m_ camelCase *ex: m_memVar*

Enum : PascalCase { ALLCAPS, ALLCAPS… };
ex: Color { RED, GREEN, BLUE };
	Events : PascalCase
	ex: Start();

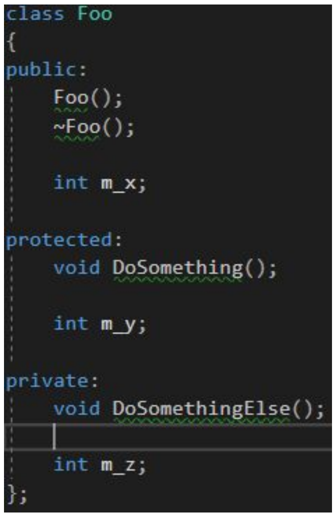
Norms

Language specification: c++17

Class and struct layout: Classes and structs will be declared in this order:

- public : methods, then variables
- protected: methods, then variables
- private: methods, then variables

For example:



```
class Foo
{
public:
    Foo();
    ~Foo();

    int m_x;

protected:
    void DoSomething();

    int m_y;

private:
    void DoSomethingElse();

    int m_z;
};
```

For primitive variable types, we will always try to use a fixed width type that takes the least memory space possible while still being able to represent maximum and minimum values that we will need to use. For example, should an int variable's max used value be less than 127, we would declare it as an int8_t rather than an int.

Organization

File Organization

All C++ classes will be part of a general namespace relative to their common context (behaviour).

For now, the engine will be installed on the user’s computer, rather than it being a simple executable. This ensures that we will have full admin rights in order to modify needed files on the computer, that all of the base resources will be at the right place on the hard drive, and that the engine’s path will be easily found by the project.

Since the engine will be installed onto the user’s PC (subject to change), there will be 2 areas where files will exist:

1. Install Directory
   - Place on the drive where the engine is installed. Will include all files of the dependencies we will use, the engine’s DLL and basic visual components (shaders and primitive objects).

2. Project Directory
   - Place where the user is working on their project.
   - Build: Folder of fully built and ready to use project.
   - Resources: Folder with user-created subfolders that hold the user’s files (ex: scripts, meshes, etc.)

When adding a new external library you need to add it to its specific folder in the Dependencies folder.

Code organisation

Our Visual Studio solution will contain 2 projects:

1. Engine

- This project will be exported into a DLL to be used by the Editor project.
- There will be 2 main folders that hold the .h and .cpp files: include and src. These two folders will contain as many subfolders as there are namespaces used in our code and the subfolders will be hierarchised in the same fashion. For example, a class that is part of the namespace Rendering::Tools will have its .cpp file in ProjectDirectory->src->Rendering->Tools and its .h file in ProjectDirectory->include->Rendering->Tools.

2. Editor

- We will use this project's main() to test our functionalities, so the engine's DLL will need to be included into the project settings.
- Same file hierarchy as for the engine.

All external libraries need to be wrapped by classes that we will create so the user will only use the engine's functions directly instead of the libraries'. This also ensures that maintenance or changes of the dependencies will not need a refactoring of the whole code for the user program.

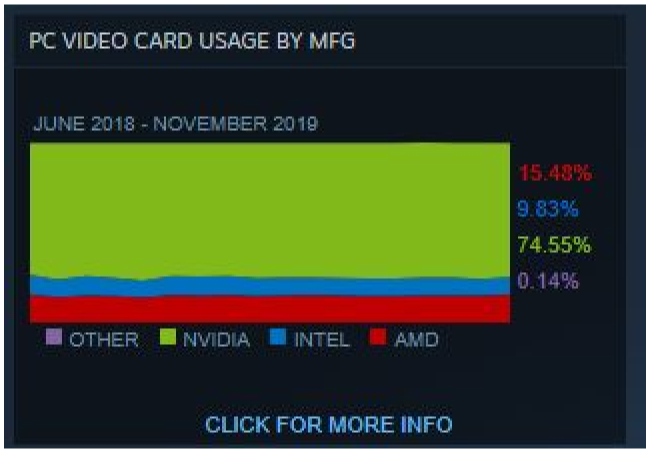
Tab indentations will consist of 4 spaces, not a full tab.

Includes will be declared using chevrons (<>), not quotes (“”).

Dependencies

Rendering: DirectX

We chose DirectX because we have never worked with it, so it would be a nice experience. It’s one of the most used APIs so it would definitely be a nice addition to our knowledge. It’s made by Microsoft and it is much more adapted to Windows than OpenGL, and it usually gets all the updates and fixes first, then they get implemented on OpenGL.
We are building an Engine for Windows. The biggest manufacturer for video cards is currently Nvidia, and Nvidia cards work better with DirectX than AMD ones. DirectX is made for Windows. Therefore, by using DirectX, our engine would be adapted to the majority of our users.



One other reason to use DirectX is that it comes with a lot more developer options (such as debugging) by default. This comes due to the fact that more developers use DirectX, so there is a bigger community, which in turn means there is more documentation and more support for it.

Physics: PhysX

PhysX belongs to NVidia, which is currently the most popular graphics card company. It can run on both GPUs and CPUs. There already is a lot of documentation available, and according to many of its users, the code is much more straightforward, less confusing and overall less messy than it would be if we use Bullet. According to Unity's dev team's stress tests, it is also more efficient for the CPU than Bullet. Furthermore, PhysX also permits us to separate the rigidbodies from the colliders, whereas Bullet only has rigidbodies. Using PhysX would permit us to easily make objects that are affected by physics while deactivating collisions.

Physics: Bullet

Given the smaller scale of our project, using Bullet would be wiser than using PhysX since Bullet generally requires less debugging. We would expect the implementation of PhysX to bog down production for a while since this has happened to a more experienced team of people we know working on a similar project before.

Math Library: GPM

We will use our homemade math library because we made it ourselves so we know exactly how it works and we can modify it whenever we want. We will use it as a static library since its code will never be modified at runtime and because accessing its functionalities will be faster and more efficient.

3D Loader: Assimp

We want to work with Assimp since it is efficient resource-wise and our team is already vaguely familiar with it. Our engine needs to be able to handle both .obj and .fbx files, which Assimp supports (among many other types). Furthermore, Assimp can "repair" or readapt broken models (ie triangulate quads, create normals if they're not present, flip the UVs…). Textures are also fairly easy to swap during run-time. It would take a long time for our team to develop such capabilities, thus using Assimp will allow us to direct fewer resources to mesh parsing (mostly debugging) and more resources to building other critical parts of the engine.

Scripting Language: [To be determined]

Audio: Irrklang

We will use Irrklang for our engine's audio because it is very easy to implement and has many low-level and high-level features. It uses its own decoders and is independent from the OS, saving us much time and effort while removing most of the potential hardware compatibility issues. Its documentation is very detailed and straight-forward. We would have liked to add it as a static library, but unfortunately that is only available for the pro version. Therefore, we will use it as a dynamic library.

Graphic Interface: ImGui

ImGui is much more efficient and easier to implement than our other alternative, Qt. ImGui is an API that is more suited to making widgets and interfaces, but more importantly, it does not run on its own update loop as opposed to Qt, to which we would have to adapt. We prefer an API that will adapt to our engine, and not the other way around.

Saving: Homemade Serialization

By the client's demand, we will save our user's projects and configurations using serialization that we will code ourselves.

Technical Risks

- Since we will attempt to run physics (and perhaps other processes as well) in a thread that is seperate from the main thread, there might be some conflicts and data races. We will need to make sure our processes are thread-safe.

- Our engine will be event-driven, so managing the event queue might pose a challenge, especially if we multithread.

- Given our team's past experiences, we expect our first iterations of the engine to run slowly. Optimizing the engine's performance will pose a challenge. It will be important to optimize frequently so as to avoid accumulating technical debt.

- Git branching might pose a problem since our team has no experience with branches. It will be important to keep track of them and to make sure that merges are carefully executed.