Golem Engine
Technical Design Document (TDD)

Introduction

This document is the Technical Design Document (TDD) for the Golem Engine. It gives information about the libraries used to make our Engine and why we chose them, but also other very important information such as the coding conventions we applied, details on how the special features of our engine works, special parts of our code, and many more. You will also find a section containing useful links in order to further understand the process of the design and development of our Engine in the Useful Links section.

Golem Engine was created in four months and by four people, so naturally, the scale of this TDD will be as such as the scale of our engine. We hope this TDD will explain to the best of its ability important parts of our engine's code, through detailed explanations and diagrams.

Summary

Introduction **2**
Summary **2**
Useful Links **3**
Technological Context **3**
Coding Conventions **4**
File and Folder names 4
Variables 4
Special variable types 4
Conditional statements 5
Loops 6
Classes 6
Comments 9
Headers 10
Interaction Diagram **11**
Libraries **11**
Graphic Library 11
Conclusion: (OpenGL) 12
Windowing 12
Conclusion: (GLFW) 12
User Interface 12
Conclusion: (ImGUI) 12
Model Loader 13
Conclusion: (Custom Loader) 13
Audio Library 13
Conclusion: (OpenAL) 13
Physics Library 13
Conclusion: (Jolt) 13
Image Loader 14
Conclusion: (stb_image) 14
Fonts 14
Conclusion: (FreeType) 14
Reflection 14
Conclusion: (refl-cpp) 14
Data Storing 15
Conclusion: (nlohmann JSON) 15

Tasks Chart **15**
Sprint 1 : 15
Sprint 2 : 16
Sprint 3 16
Sprint 4 : 17
Special Features **17**
Particle System 17
Map Maker 17
Toon Shading Tool 17
Critical points **18**
File Loader 18
Physics (implementation and usage) 18
Scene Manager 18
Lighting System & Shadows 18

Useful Links

TDD :
https://docs.google.com/document/d/1Jtf2k2K6IGrmmmKho3yW_sFI192x4ObjKjQ_ZNkWz0c/edit?usp=drive_link

Backlogs :
https://docs.google.com/spreadsheets/u/0/d/13i5iRtw7TFWCFSN_rtzIWthV1jdXRb6yfZKntQG_MpQ/edit

Trello :
https://trello.com/w/golem_engine

Engine UML :
https://app.diagrams.net/#G1YYfYapjmPG-OQKQSFgRh9GnygVCJJSjI

Interaction Diagram :
https://app.diagrams.net/#G1OdVxF9WFAIje_cibcuFsyLmjxvCg91QG

Technological Context

- **Operating system :** Windows 64-bit
- **IDE :** Visual Studio 2019

- **C++ project Settings :**
  - C++ version : C++ 20.
  - CPU Architecture : x64

- **Target user :** Game developers

Coding Conventions

Use The C++ 20 norm.

File and Folder names

Folder	ExampleFolderName
File	exampleFileName.h

Variables

```C/C++
// // Local variable
int localVariableExample;

// Private class variable
int m_ClassVariableExample;

// Protected class variable
int p_ClassVariableExample;


// Public class variable
int classVariableExample;
```

Special variable types

Boolean variables should be preceded by “is”, shown as is :

```C/C++
// boolean example variable
bool isVisible;
```

Enums should be written as is :

```C/C++
enum class ExampleEnum
{
	FIRST_DAY,
	SECOND_DAY,
	THIRD_DAY
};
```

Structs should be written as is :

```C/C++
struct ExampleStruct
{
	int a;
	int b;
};
```

Conditional statements

Basic successive blocks :

```C/C++
if ()
{

}
else if()
{

}
else()
{

}
```

Blocks that don’t succeed to each other :

```
C/C++
if()
{

}

if()
{

}
```

Loops

```
C/C++
for ()
{

}
```

```
C/C++
while ()
{

}
```

Classes

Class variables should be in a separate private/protected/public section than the class functions, respecting the private/protected/public order, shown as is :

```
C/C++
class Test
{
private:
	int m_var1;
	int m_var2;

protected:
	int p_var1;

public:
```

```
	char* var3;


private:
	void PrivateFunc();
public:
	void PublicExampleFunc(int _var1, int _var2);
};
```

The parameters of a class function should always start by "_". This avoids confusion between local variables and class member variables. Shown as is :

- In .h :

```
C/C++
class Test
{
public:
	void PublicExampleFunc(int _var1, int _var2);
};
```

- In .cpp

```
C/C++
void Test::PublicExampleFunc(int _var1, int _var2)
{
	// stuff...
}
```

Empty class functions should be defined both in .cpp and .h, shown as is :

- in .h :

```
C/C++
class Test
{
public:
	Test();
	Test(const Test& copy);
	Test(Test&& move) noexcept;
	~Test();
```

```
        Test& operator=(const Test& copy) = default;
        Test& operator=(Test&& move) noexcept = default;
};
```

- In .cpp :

```
C/C++
Test::Test() {}

Test::~Test() {}
```

Class functions should respect the order of their execution (if possible) and the constructors and destructors should always be declared first, shown as is :

- In .h :

```
C/C++
class Test
{
public:
        int a;
        int b;
        int c;

public:
        Test();
        ~Test();

        int AddA(int _x, int _a);
        int AddBA(int _x, int _b);
        int AddCB(int _x, int _c);
};
```

- In .cpp :

```
C/C++
Test::Test() {}

Test::~Test() {}


int Test::AddA(int _x, int _a)
{
        return _x + _a;
```

```
}

int Test::AddBA(int _x, int _b)
{
	int X = _x + 3;
	return AddA(X, _b);		// calls AddA
}

int Test::AddCB(int _x, int _c)
{
	int X = _x + 5;
	return AddBA(X, _c);		// calls AddBA
}
```

Comments

Comments need to be used to explain complex lines of codes and functions. They can also be used to explain a whole block of code, for example loops that have a certain function.

To explain a complex line, the comment must be placed on the same line at the end of it with and indentation, shown as is :

```
C/C++
heightmap = stbi_load(_imagePath, &width, &height, &nChannel, STBI_grey);	// Load a heightmap
unsigned char heightMapValue = (heightmap + (i + width * j) * bytePerPixel)[0];	// read color value of a pixel
```

To explain an important block or loop, the comment must be placed at the top of the first line of the block, shown as is :

```
C/C++
// Make grid of vertex to make a flat surface
for (int i = 0; i < rows; i++) {
	for (int j = 0; j < columns; j++) {
		unsigned char heightMapValue = (heightmap + (i + width * j) * bytePerPixel)[0];
		vertex.pos.x = x + i * 60.0f / width;
		vertex.pos.z = z + j * 60.0f / height;
		vertex.pos.y = heightMapValue / 255.0f * 5;
	}
}
```

Comments should also be used at the beginning of complex functions to give a quick explanation about them. The comment should be written in the .h file if possible, but if not, in the .cpp file (not in both), and in either case at the top of the function, shown as is :

- .h file :

```C/C++
class Terrain {
public:
	// Generate a terrain by loading vertices on a grid and
	// assigning them their height value (y) by reading the color
	// of a loaded heightmap
	void GenerateVertexData(float _heightmapSizeMult, float _verticesSeperationDist = 0.1f);
};
```

- in .cpp file (only if not in .h file) :

```C/C++
// Generate a terrain by loading vertices on a grid and
// assigning them their height value (y) by reading the color
// of a loaded heightmap
void GenerateVertexData(float _heightmapSizeMult, float _verticesSeperationDist = 0.1f)
{
	// stuff...
}
```

Headers

The order of headers is : related file header, external libraries, standard libraries, personal files. Personal files (including file related header) should be surrounded by “ ”. Shown as is :

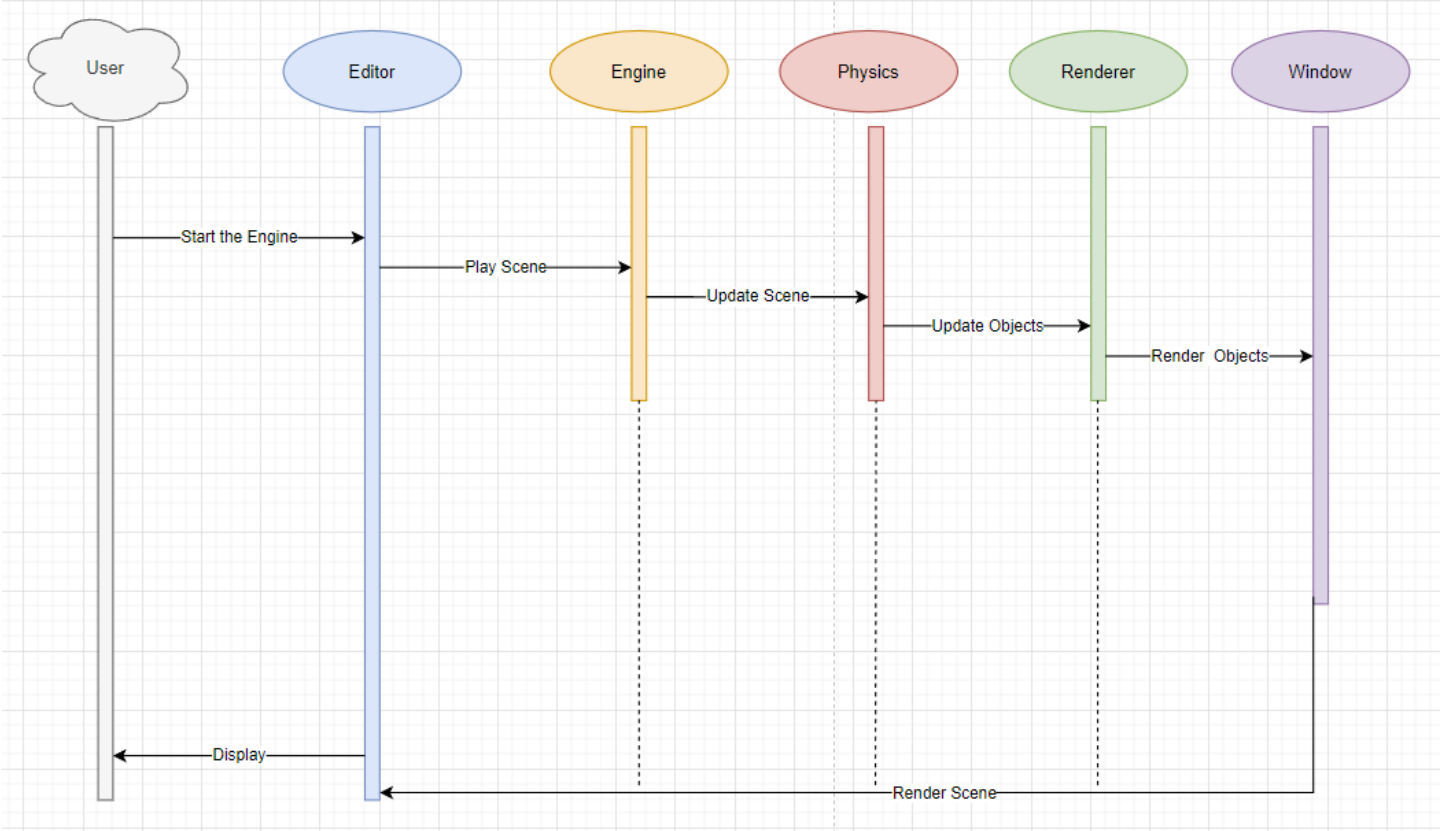
```Unset
#include "myFile.h"

#include <vulkan>
#include <bullet>

#include <iostream>
#include <vector>
#include <string>

#include "engine.h"
#include "mathsToolbox.h"
```

Interaction Diagram



Libraries

Graphic Library

Tests	OpenGL	Vulkan

Performances	Medium	High
Simplicity	High	Low
Flexibility	Low	High

Conclusion: (OpenGL)

Despite Vulkan being better overall, OpenGL was chosen because of its extreme simplicity when compared to Vulkan. Vulkan requires a large amount of setup and maintenance, and we wish to concentrate our efforts on the features we are developing, which aren't much related to rendering and don't need a deeper control over the render pipeline. Finally, the team is more familiar with OpenGL so choosing OpenGL will save us lots of work to accomplish the same result and faster.
https://history-computer.com/opengl-vs-vulkan-what-are-the-key-differences/

Windowing

Tests	GLFW	Windows API
Simplicity	High	Low
Flexibility	Medium	High
Packages	Yes	No

Conclusion: (GLFW)

Windows API has a lot of benefits, like its flexibility due to its low-level programming, or the fact that it needs no packages to use. But it requires more experience to use, compared to GLFW that is easier to use, and the team being already familiar with it. Classic windows will be acceptable.
https://www.reddit.com/r/GraphicsProgramming/comments/tddoqi/why_would_anyone_use_the_native_libraries/

User Interface

Tests	ImGUI	QT
Simplicity	High	Low
Performances	Medium	High

Conclusion: (ImGUI)

QT is more adjustable than ImGUI as it's more complex, and so, harder to learn. It also has better performances as it doesn't redraw the full screen all the time. But the whole team used ImGUI for a while ImGUI, and we are not looking for performances, so ImGUI was the best choice.
https://forum.qt.io/topic/153317/dear-imgui/6

Model Loader

Tests	Assimp	Custom Loader
Simplicity	High	High
Packages	Yes	No

Conclusion: (Custom Loader)

As we want to load .obj files, Assimp is excessive while it permits to load several types of 3D model files. The team's loader will be reasonable and also more flexible.
https://assimp.org/

Audio Library

Tests	OpenAL	AudioPort	SFML
Performances	Best Performances	Worst Performances	Good Performances
Simplicity	High	Low	High
Features	Some	Much	Much

Conclusion: (OpenAL)

OpenAL is a cross-platform 3D audio API appropriate for use with gaming applications and many other types of audio applications. AudioPort has many powerful functions, but the code is complicated, SFML is easy to use but has more code. OpenAL is the most basic and intuitive music loading.
https://www.openal.org/documentation/

Physics Library

Tests	Jolt	PhysX	Bullet
Performances	Best performances	Good performances	Worst performances
Integration	Easy	Hard	Easy
Features	Some	Much	Some

Conclusion: (Jolt)

Jolt has better performance than Bullet and is pretty similar to PhysX. In the performance test linked below, Jolt seems to have better performances overall than Bullet, PhysX and other physic engines when dealing with multiple bodies. We don't aim to have the most complex or feature-advanced physics library, so we prioritize performance over features. With that in mind, Jolt seemed like the best choice for us.
https://jrouwe.nl/jolt/JoltPhysicsMulticoreScaling.pdf

Image Loader

Tests	stb_image	SOIL2
Performances	Good performances	Best performances
Integration	Easiest *(only header file to include)*	Good
Features	Some	Much *(more OpenGL centered and more features)*

Conclusion: (stb_image)

SOIL2 is a great library for an OpenGL project but the team stuck with stb_image being more familiar with that one which would allow us to not waste any time. On top of that, SOIL2 adds more features that aren't necessarily all needed and stb_image remains flexible for multithreading which is better in case we want to add parts of our code in other threads.

Fonts

Tests	FreeType	SFML
Performances	Good performances	Bad performances
Simplicity	Low	High
Flexibility	High	Low

Conclusion: (FreeType)

As SFML isn't a library specialized in fonts, it's too much to include it only for fonts. FreeType is a better choice because it has better performances and better flexibility due to its specialization in fonts.

Reflection

Tests	refl-cpp	Refureku
Integration	Easy	Medium
Documentation	Much	Not much

Conclusion: (refl-cpp)

We had some trouble testing Refureku while refl-cpp is a header-only library. Furthermore, Refureku isn't really popular, so it's hard to find some good references. refl-cpp is way more famous and has a lot of good sources to learn.

Tests	XML	YAML	nlohmann JSON	JSONcpp
Performances	Best performances	Worst performance	Medium	Good performance
Simplicity and Integration	Medium	Medium	Easiest	Easy
Readability	Hardest	Easiest	Easy	Easy

Conclusion: (nlohmann JSON)

There are a bunch of different data storing libraries, and each of them possess themselves are a bunch of different custom libraries. The first choice we had to make was which type of library should we pick between XML, YAML and JSON. XML turned out to be the most complicated to read the data and XML libraries are focused on increasing loading and storing performances, which we don't especially need. We were then in favor of YAML due to its best readability, however the integration of YAML was quite complicated so was the code which was quite surprising. We ended up going for JSON which is a good balance between performance and readability and on top of that, the nlohmann JSON library seemed the easiest to understand and just needed a header file for integration.

Tasks Chart

Sprint 1 :

Story ID	Tasks	Task Score
1	Create TDD on Google drive	13
1	Update TDD	8
20	Make unit tests	5
20	Include maths in Solution	8
6	Initialize window with GLFW api	13
6	Display objects	13
2	Load model infos	8
2	Send model infos to shader	8
2	Use shader	8
19	Create a scene	13
19	Be able to navigate with camera	13
29	Include the Sound library	13
29	Include the Physic library	20

Sprint 2 :

Story ID	Tasks	Task Score
2	Render 3D models	5
2	Render texture on 3D models	5
2	Render a mesh with model, texture and shader	5
3	Create light classes	13
3	Implement lights in shader	8
28	Viewport	20
28	File browser	3
28	Inspector	3
28	IsActive window UI	13

Sprint 3

Story ID	Tasks	Task Score
5	Content Browser	20
7	World Actors	13
2	OpenGL wrapper in shader class	5
2	Create an Objects system with transforms	13
3	Review light architecture	5
4	Content Browser	20
4	Do unique resource manager	3
11	Add quaternion class to mathtoolbox	13
18	Input class	13
28	Put ui in editor	40

Sprint 4 :

Story ID	Tasks	Task Score
8	Load and save level	60
2	Render texture on 3D models	5
2	Render a mesh with model, texture and shader	5
3	Create light classes	13
3	Implement lights in shader	8
28	Viewport	20
28	File browser	3
28	Inspector	3
28	IsActive window UI	13

Special Features

Particle System

A system to create, import and modify particles that can be added to the levels.

Map Maker

A tool to create maps with hills, rivers, mesh painting, instancing…etc.
Wind simulation : Two noise maps : one for x vector and one for y vector, noise color determines the intensity of the vector. Make the map move slightly through time.)

Toon Shading Tool

A tool to easily implement the Toon Shading shader effect in levels

Critical points

File Loader

The file loading system is considered a critical point because of its decent complexity and multiple challenges. Our file loading system will, first of all, have to handle different types of files (models, sounds, textures, levels…etc). It will have to create the corresponding class depending on the type of file loaded, which makes this system deeply connected to multiple parts of our code. Finally, another important task to do will be to load the file, if wanted, inside the scene (or load the scene in case of the file being a level) at the position of the mouse.

- View project documents in real time (Efficient import that facilitates project management and file loading.)
- File identification (Check whether the file is of a type that the engine can import. If so, it can be loaded. If not, the user will be notified of the type error.)
- Drag & Drop (Users can directly drag files from the file viewer to the editor window, which improves loading efficiency and is more in line with user intuition.)

Physics (implementation and usage)

Scene Manager

The scene manager system is considered are critical point for lots of different reasons. Firstly, there are three main functions to add to the scene manager : CreateScene, SaveScene and LoadScene. Each of these functions present their own technical difficulty. Secondly, these functions will all consume a certain amount of time to finalize, not being just technically difficult but long as well. Finally, two of these features (SaveScene and LoadScene) will require JSON knowledge that we will have to learn quickly in order to continue the project as fast as possible to respect the deadlines.

- CreateScene : This first functionality represents a technical difficulty on multiple levels. It requires us to redesign completely our scene system, storing them in arrays, and accessing them differently. But it also means that all scenes must be correctly initialized with the correct components (world transform, directional lights…).
- SaveScene : This functionality makes it necessary to learn JSON language in order to store the data of the scene on .json files. Scenes also content entities, meaning loading data from pointers, whose adresse change on each load. And of course, entities also contain pointers themselves which will probably require some form of recursion, or a way to access all the data of any pointer.
- LoadScene : The load scene sets us two main challenges, the first one being similar to save scene : instead of storing data from pointers, we will now need to access that stored data and transfer it to pointers. The second big difficulty is that we will now need to find a way to correctly load all the data from files and transform them into the corresponding entities, whether they are objects, lights or audio.

Lighting System & Shadows