



# Fengine

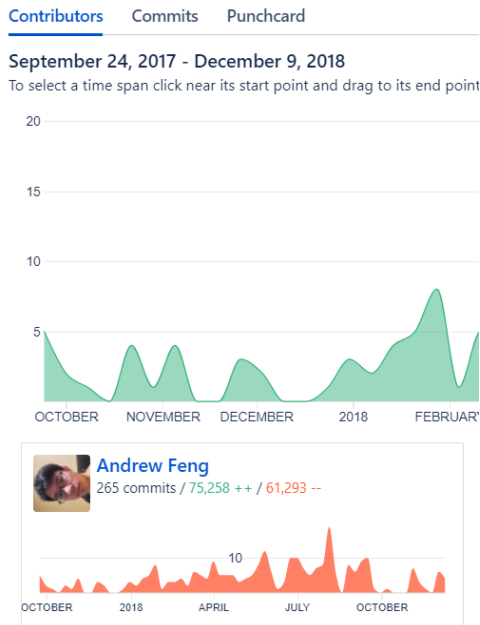
A 2D GAME ENGINE MADE IN C++

Andrew Feng | MIT Maker Portfolio | 2018 - 2019

# I. Introduction

This documentation of Fengine is divided into 6 sections that each corresponds to a major component of the engine. The components are presented in the chronological order in which they are added to the project, starting from the rendering system written in the summer after grade 10. However, they are constantly maintained/updated by me and my friends.

The engine contains over 70 classes & data types and is still expanding, which means it's impossible to address the entire engine in one single PDF document.



Still, I would love to present to you the most interesting parts of Fengine. If you are under time constraint reading this doc, I suggest you to read the following parts (ranked by importance):

1. [What I learned from Making Fengine](#) (~5 min)
2. [Scripting System](#), especially the [Script](#) class (~15 min)
3. [Physics System](#) (~15 min)
4. [Rendering System](#), especially the [Animation](#) class and the [Flashlight](#) class (~10 min)

However, I highly suggest reading this documentation in order if you have enough time (~60 min).

If you are interested in this project after reading this document, or are curious about the implementation details, you can find important links about this project below.

Repositories:

- **Fengine:** <https://bitbucket.org/milkyway2017/fengine/src/master/>
- **Designated Stim Area (a demo game project & the entry point):** <https://bitbucket.org/milkyway2017/fengine/src/master/>
- **Dependency – Windows:** <https://bitbucket.org/milkyway2017/dependency/src/master/>
- **Dependency – Ubuntu:** [https://bitbucket.org/milkyway2017/dependency\\_linux/src/master/](https://bitbucket.org/milkyway2017/dependency_linux/src/master/)

Development Blog:

- <https://milkyway-project.ooowebhostapp.com/>

Doxygen Documentation:

- <https://milkywaytest.ooowebhostapp.com/annotated.html>

Many of the feature in the engine may seem too specific or not useful to other games, because the engine was originally made for a game called *Designated Stim Area*, but the development focus has shifted to the game engine.

## II. What I learned from Making Fengine

Growing up as a gamer, I had a natural desire to make a game of my own, more specifically, a game that could bring people the joy gaming had brought me. My plan was simple: I would code the program and my friends would design the art. I quickly gathered a group of interested friends to join.

To program a game, I had two choices: use an existing game engine like Unity, or program everything from scratch with C++. The former choice allowed game development with minimum programming, but I chose the second option, thinking, “what can go wrong in going deeper and learning a bit more programming?” Well, as it turned out, using C++ to program a game from scratch is the second worst decision one can make next to making an app using assembly code. For example, it took me a full day of coding to display an image on screen, something that could be done with just one line of C# using Unity. I spent over two months, reading over technical tutorial blogs and devising software architecture, only to program a rendering system. As a result, team meetings became a place where I passionately showcased how I made the sliding door open when a character approaches while my friends puzzled over my excitement – even they could do that using a few lines of code in Unity! They didn’t understand that what they saw was the reward for my many nights of frustration and hardcore debugging. Worse, it was frustrating for me to see the confusion among my friends: I assumed they knew the effort behind the project and had no idea that I, in fact, failed to deliver robust software for the game. I became so obsessed with programming that I forgot to think about things that really matter to a game, such as the theme and the plot. I oriented the project around the software, the

exact opposite of the correct approach. I left my artistic friends with no clear instructions on what to create and rendered a blurry picture of the project’s future. Gradually, most of them quietly withdrew from the project while I was still worrying about the efficiency of my collision detection algorithm.

My game project failed miserably, and it didn’t take me long to see the reasons for my failure: a project is destined to fail if the leader cannot provide a blueprint for it and engage the members to work toward clear goals. A leader’s job is not to say “you guys work on whatever while I get this bug fixed!” Even though this realization arrived too late to save the project, there’s a bright side to my failure: I discovered how much I love software engineering. After all, I was too focused on the “art” of programming to care about the content creating process; I found more pleasure in engineering tools to help game developers than in using existing tools to create games. So, I continued the project as a game engine – Fengine.

Truly driven by passion, I dedicated hours every day to either create new features, or to optimize the performance of existing features. I acquired necessary skills and knowledge along the way. For example, I learned linear algebra to program the rendering system, took courses on edX to learn about algorithms and data structures, and studied proper software engineering techniques to improve code quality. Surprisingly, my dedication attracted two of my techie friends – I was again given another opportunity to be a leader! This time, I planned out development schedules for the team and gave clear instructions to the members.

Issues (1–25 of 34)

| Title  | T | P | Status   | Votes | Assignee      | Created    |
|--|---|---|----------|-------|---------------|------------|
| #28: [BUGS/OPTIMIZATION] Flashlight                                  |   |   | NEW      |       |               | 2018-07-04 |
| #35: [GAMEPLAY]: displaying inventory                                |   |   | NEW      |       | Stas Kalynych | 2018-08-09 |
| #25: [MAP]: object health bar  |   |   | RESOLVED |       | Stas Kalynych | 2018-06-29 |
| #20: [GAMEPLAY]: allow player to attack                              |   |   | RESOLVED |       | Stas Kalynych | 2018-05-25 |
| #17: [GAMEPLAY]: implement UpdateStats() for Player                  |   |   | RESOLVED |       | Stas Kalynych | 2018-05-20 |
| #13: [Map]: port map class to load from json file                    |   |   | RESOLVED |       | Stas Kalynych | 2018-04-20 |
| #11: [Engine]: add custom exceptions                                 |   |   | RESOLVED |       | Stas Kalynych | 2018-04-03 |
| #29: [ENGINE]: ParseScript function                                  |   |   | RESOLVED |       | Filz Pillz    | 2018-07-05 |
| #21: [GAMEPLAY]: create lycan range of vision                        |   |   | RESOLVED |       | Filz Pillz    | 2018-05-31 |
| #19: [GAMEPLAY]: implement CanAttack() and ReceiveAttack() for Lycan |   |   | RESOLVED |       | Filz Pillz    | 2018-05-25 |
| #16: [GAMEPLAY]: create pure virtual function Object::UpdateStats()  |   |   | RESOLVED |       | Filz Pillz    | 2018-05-20 |

In fact, I gave hours of guidance to each of them, introducing them to different parts of the engine: Stanislav to user interface and Filip to physics. Together, we improved the system and invented interesting parts. For example, Stas worked to make sure the [engine runs on Linux](#) and created a convenient [StatusBar class](#) for tracking game stats. Filip devised plans with me to improve the inefficient [visibility polygon algorithm of Flashlight](#) and [troubleshoot the rendering system](#). My friends have since shared their insights (on how to improve usability, or on questions like: if they are the users, what kind of API they

would want to use) with me, motivating me to continue improving and adding cool features to the engine.

In the end, my project taught me not just technical skills, but also the importance of organization and teamwork.

### III. Table of Contents

|       |   |    |
|-------|---|----|
| I.    | Introduction .....  | 1  |
| II.   | What I learned from Making Fengine .....                    | 2  |
| III.  | Table of Contents .....                                     | 5  |
| IV.   | Project Setup.....  | 7  |
| V.    | Rendering System .....                                      | 8  |
|       | ➤ Purpose.....  | 8  |
|       | ➤ Showcase .....  | 8  |
|       | ➤ Technical detail .....                                    | 9  |
|       | ❖ <i>Graphics (Graphics.h)</i> .....                        | 9  |
|       | ❖ <i>Sprite (Sprite.h)</i> .....                            | 10 |
|       | ❖ <i>Animation (Animation.h)</i> .....                      | 11 |
|       | ❖ <i>GLShape and GLVertex (Primitive.h)</i> .....           | 12 |
|       | ❖ <i>Flashlight (Flashlight.h)</i> .....                    | 14 |
| VI.   | Entity System.....  | 18 |
|       | ➤ Purpose.....  | 18 |
|       | ➤ Effect.....   | 18 |
|       | ➤ Technical Detail .....                                    | 18 |
|       | ❖ <i>Object (Object.h)</i> .....                            | 18 |
|       | ❖ <i>Figure (Figure.h)</i> .....                            | 20 |
|       | ❖ <i>GameItem (GameItem.h)</i> .....                        | 21 |
| VIII. | Map System (Map.h).....                                     | 23 |
|       | ➤ Purpose.....  | 23 |
|       | ➤ Effect.....   | 23 |
|       | ➤ Technical Detail .....                                    | 23 |
|       | ❖ <i>Overview</i> .....                                     | 23 |
|       | ❖ <i>Map Loading (Map::LoadMap):</i> .....                  | 23 |
|       | ❖ <i>Entity Query System</i> .....                          | 27 |
|       | ❖ <i>Utility Functions for Objects</i> .....                | 29 |
| IX.   | Scripting System, Event System, and Functor and Timers..... | 30 |

|   |    |
|---|----|
| ➤ Purpose.....  | 30 |
| ➤ Effect.....   | 30 |
| ➤ Technical Detail .....  | 30 |
| <i>Functor (Functor.h)</i> .....  | 30 |
| ❖ <i>EventManager (EventManager.h)</i> .....                                  | 31 |
| ❖ <i>Timer (Timer.h)</i> .....  | 32 |
| ❖ <i>Script (Script.h)</i> .....  | 34 |
| X. Physics System .....   | 39 |
| ➤ Purpose.....  | 39 |
| ➤ Effect.....   | 39 |
| ➤ Technical Detail .....  | 39 |
| ❖ <i>World (Kinematics.h)</i> .....   | 39 |
| ❖ <i>Quadrant (Quadrant.h)</i> .....  | 40 |
| ❖ <i>Motion (Kinematics.h)</i> .....  | 41 |
| ❖ <i>Body (Kinematics.h)</i> .....  | 43 |
| XI. Utility.....  | 46 |
| ❖ <i>Point2F - 2D vector (Primitive.h)</i> .....                              | 46 |
| ❖ <i>Rect4F - 4D vector (Primitive.h)</i> .....                               | 46 |
| ❖ <i>Angle (Primitive.h)</i> .....  | 46 |
| ❖ <i>Other Primitives (GLColor, GLVertex, GLShape, Line, Circle...)</i> ..... | 46 |
| XII. Things to be improved.....   | 47 |

## IV. Project Setup

Language: C++14, with some libraries written in C

Code Convention: The Google C++ Style

Documentation: Doxygen

Environment Setup:

- Windows: MSVC as compiler, Visual Studio 2017 (preferred) or CLion as IDE
- Ubuntu(18.04): gcc as compiler, CLion as IDE

Build System: CMake (minimum version required: 3.0)

Dependency:

- Windows: <https://bitbucket.org/milkyway2017/dependency/src/master/>
- Ubuntu: [https://bitbucket.org/milkyway2017/dependency\\_linux/src/master/](https://bitbucket.org/milkyway2017/dependency_linux/src/master/)
- OpenGL for graphics (with [glew](#) and [glm](#))
- [SDL2](#) for cross-platform window creation and user input
- [CEGUI](#) for GUI (deprecated and will be replaced by [imgui](#) soon)
- [JSON for Modern C++](#) for JSON parsing
- [MicroPather](#) for A\* path finding
- [LibUTF++](#) for Unicode encoding



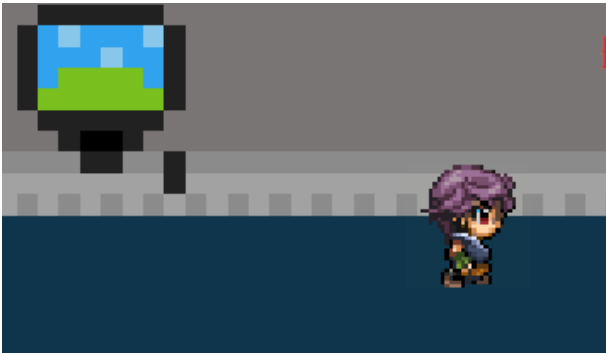
## V. Rendering System

### ➤ PURPOSE

Initially, the engine drew objects using the basic functionalities provided by the SDL library. Though engine was able to render rectangular images, do rotations, and fade, it provided too little flexibility for game development. For example, it did not even allow developers to draw basic shapes like triangles, not to mention the cool visual effects seen in many games. This is why I decided to embark on my journey to overhaul the existing rendering system and learn to integrate the graphics library OpenGL.

### ➤ SHOWCASE

Before the overhaul the engine was able to do basic image rendering and animation:



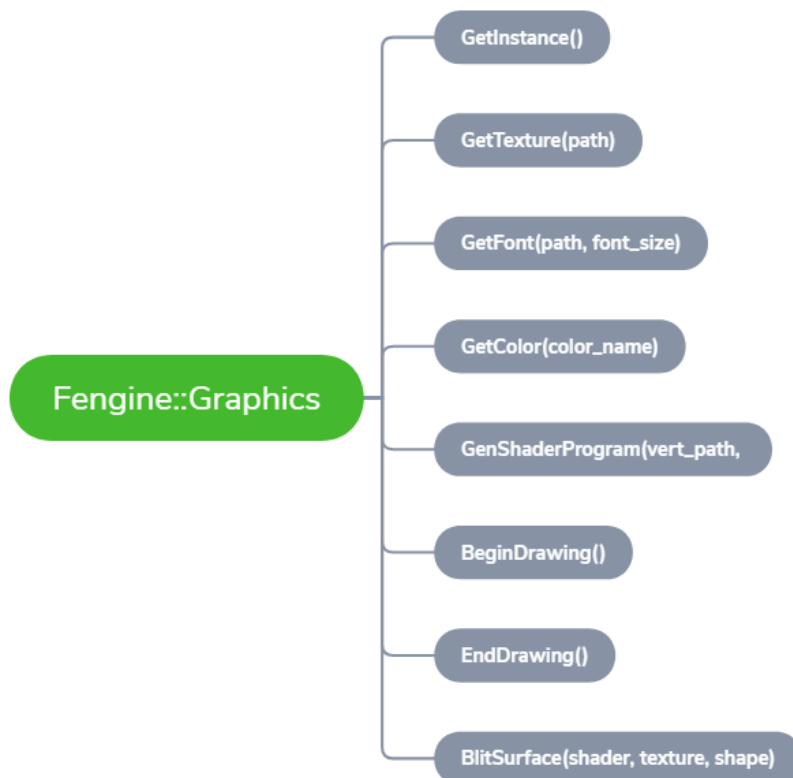
After the overhaul, the engine was able to read in any custom shader files and draw any arbitrary objects! The developer is in control of everything:



➤ TECHNICAL DETAIL

❖ Graphics (Graphics.h)

The most important class of the rendering system is the Graphics class. The class exists as a singleton<sup>1</sup> that provides low level functionalities such as shape rendering, font loading, and texture loading. Functions in Graphics are called by higher level classes. Here is a simplified diagram for Graphics:



The class acts as a wrapper around library code, abstracting away third party library such as OpenGL or GLM. This way the game developers do not need to worry about the underlying libraries. For example, `GetTexture` allows the user to load textures from image files without worrying about the underlying library function calls:

```
GLuint texture = graphics->GetTexture("image.png");
```

Image loading and displaying is further simplified by the [Sprite](#) class.

---

<sup>1</sup> All singletons in Engine are accessible by calling `___::GetInstance()`. So to get Graphics, one would call `Graphics::GetInstance()`

Most importantly, the fact that this class only contains low level features doesn't undermine its user friendliness.

For example, it can be annoying having to type out RGB values when we the developers just want a color for debugging. For this purpose, the Graphics class provides convenient features such as `GetColor(color_name)`, allowing the developers to pick from over 100 default colors given their names:

```
auto color_blue = Graphics::GetInstance()->GetColor("blue");
```

Developers can also load their own set of project specific colors by calling `LoadColor` and passing in the path to a JSON color file<sup>2</sup>.

#### ❖ Sprite (Sprite.h)

The Sprite class provides the ability to display images. Even though one can load and display textures using `Graphics::GetTexture` and `Graphics::BlitSurface`, it can get quite messy, especially when multiple sprites are to be displayed. Sprite class allows the developers to load and display a texture with only 3 lines of code:

```
Sprite sprite;

sprite.Init("image.png", {0,0,314,628});

//some other processing. . .

sprite.Draw(shader);           //shader is compiled beforehand
```

Instead of:

```
GLuint texture = graphics->GetTexture("image.png");

//setup shape for rendering

GLShape shape;

//assign positions to vertices of the shape

SubRect(shape, {0,0,314,628});

//assign texture coordinates to vertices of the shape

SubRectTextCoord(shape);

//multiple other steps to set up VAO (vertex array object) and VBO
//(vertex buffer object) for OpenGL
```

---

<sup>2</sup> An acceptable JSON color file has the following format: `{colors: [{"name": "Blue", "hex": "#0000ff", "rgb": "(0, 0, 255)"}]}`

```

//. . .
//finally display
graphics->BlitSurface(shader, texture, shape);

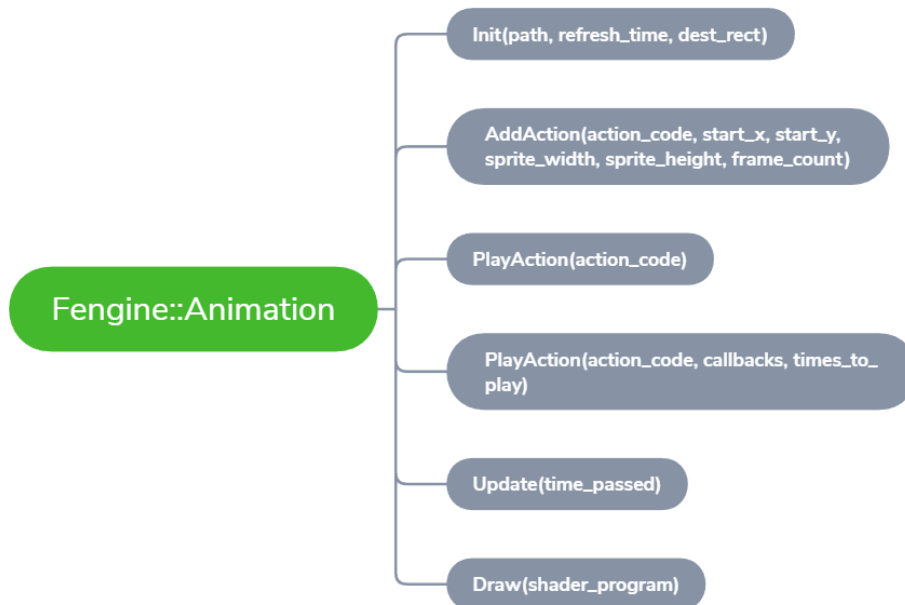
```

As you can already tell from above, the Sprite class makes it convenient to display image. This saves the developers from worrying about low level processing details such as vertex management.

Even though this is not much improvement from the SDL library's functions that also allow less-than-5-line display of textures, the process of building and maintaining this class taught me quite a bit about usability consideration. For example, a Sprite object allows the user to link it with some object by calling `SetReferenceDestinationRect`. This works as the Sprite object now owns a pointer to a "Destination Rectangle" object. Once set, the sprite object would be in "reference mode", and the user no longer needs to update the sprite's position every frame to move it; instead, the sprite "automatically" follows the destination rectangle. This is useful in game programming, where things are constantly moving around, and positions need to synchronize every frame.

#### ❖ Animation (Animation.h)

The Animation class inherits from Sprite and allows the developers to use Sprite as a sprite sheet for animation. Here is a diagram for Animation:



To create an animation, the developer can:

- Initialize an Animation object by `Animation::Init`, and pass in the path to the sprite sheet, the refresh time (not rate) for the animation, and the rectangular

position on the screen (again, one may pass in a pointer to a rectangle instead here).

- Add an action to the animation by calling `Animation::AddAction` with the following parameters (for this sample sprite sheet)



```
animation.AddAction(SLIME_WALKING, 0, 0, 20, 25, 6);
```

- Play the action added by calling `Animation::PlayAction(SLIME_WALKING)`!

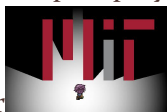
The `Animation` class even supports callbacks on specified frames (see the second `PlayAction` definition in the diagram). This allows certain actions to take place when the `Animation` gets to some frames. It's useful in many ways; for example, in the case where the character needs to fire an attack event when the attack animation finishes.

```
238 | case ATTACKING:  
239 |     //whenever status is switched to attacking, play attack animation  
240 |     lycan_actions_.PlayAction((int) LycanAnimations::ATTACK_ANIMATION,  
241 |                               {  
242 |                                   AnimationCallback(3/*trigger on the 4th frame*/,  
243 |                                                       Functor(this, &Lycan::HandleAnimationComplete))
```

When accompanied with the [Event System](#) and [Functor](#), this feature of `Animation` becomes very powerful.

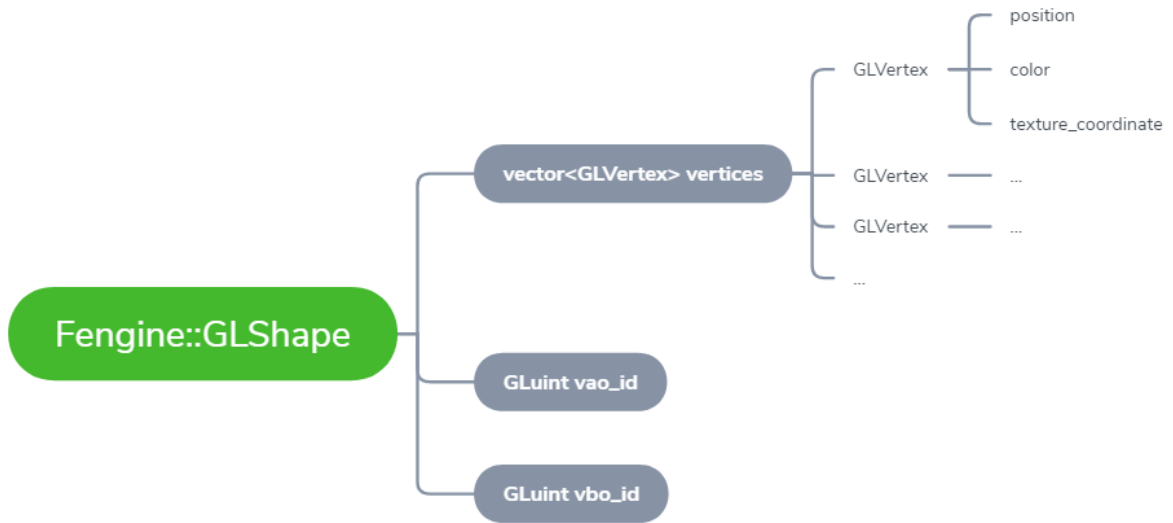
#### ❖ `GLShape` and `GLVertex` (`Primitive.h`)

`Shape` and `vertex` are the most basic structures in `Fengine`'s rendering system. They are the basis of all rendering process in the engine. From the most basic image display to the more complex polygon display (such as the visibility polygon with the MIT logo displayed



earlier), the `GLShape` and `GLVertex` support it all.

Here are the structures of these types:



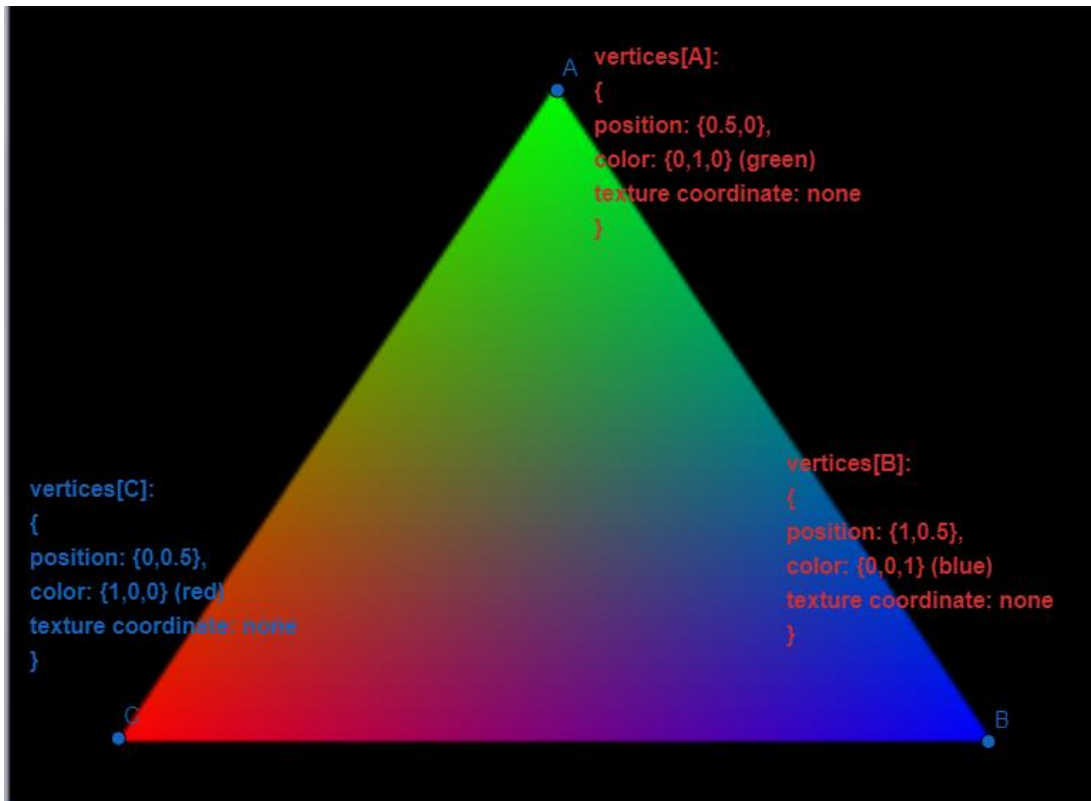
Each GLShape contains a list of GLVertex that make up the vertices of the shape. Each vertex is allowed to have a position, color, and texture coordinate.

**position:** describes the location of the vertex on screen;

**color:** describes the color at the vertex position, areas in between vertices will have a color gradient;

**texture\_coordinate:** specifies the position in the texture this vertex represents. If a shape has no texture associated with it, texture\_coordinate will not be used when drawing.

For example, to render a colored triangle, GLShape would contain the following info:



To draw a shape, one can call `Graphics::BlitSurface` with the shader, texture, and shape.

The process of setting the position and texture coordinate of each vertex is tedious and repetitive, and this is reason the `Sprite` class exists.

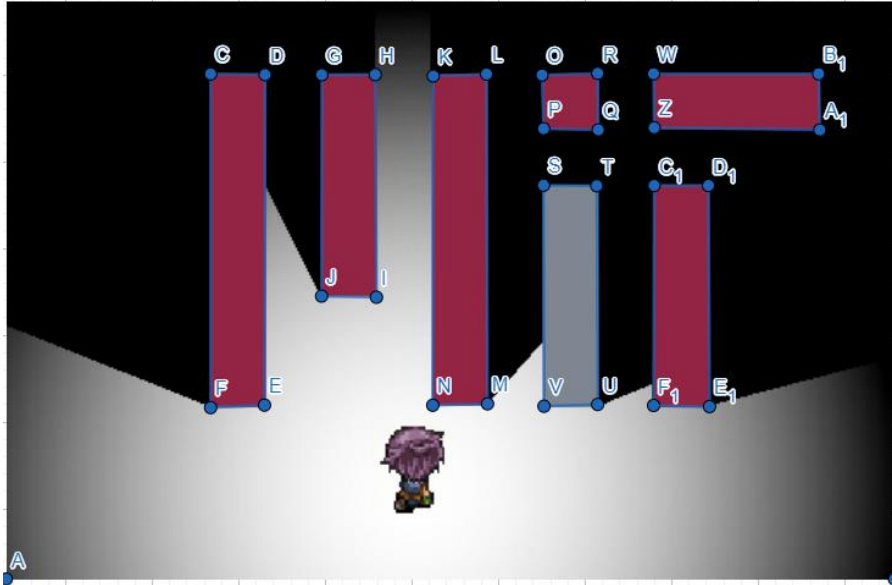
However, sometimes the flexibility is required to rendering non-tradition objects.

#### ❖ Flashlight (Flashlight.h)

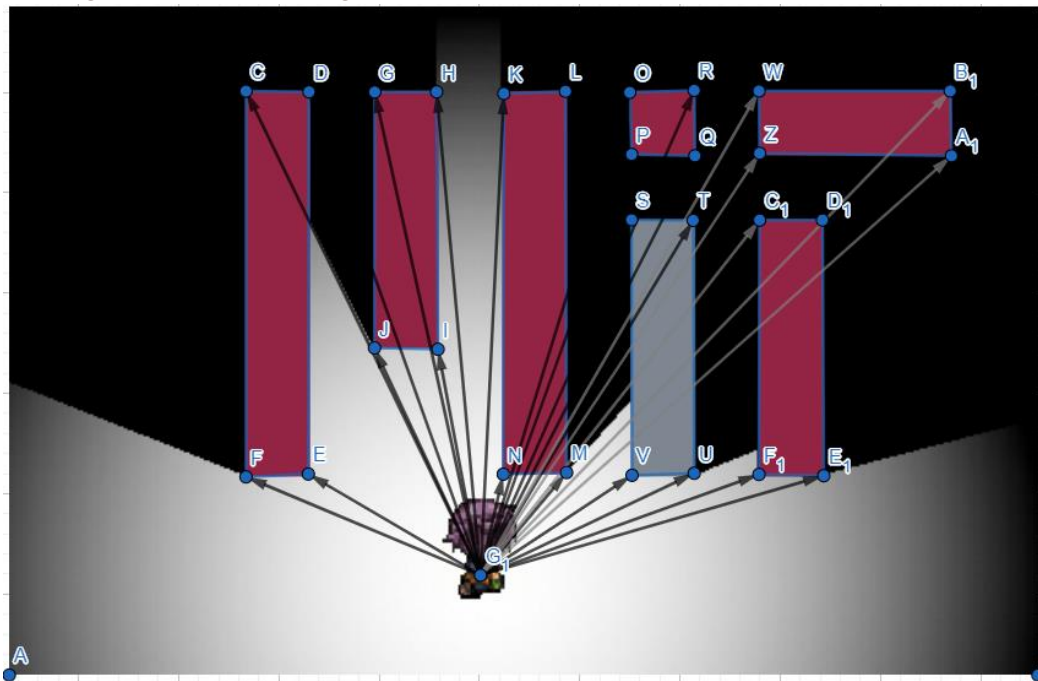
The `Flashlight` class inherits from the `GameItem` class and is an implementation of the visibility polygon algorithm. This class seem extra for a game engine to include, but I still implemented it as the algorithm is quite interesting and complex.

The following steps are taken by the algorithm to find the visibility polygon:

1. Line segments and vertices of walls in the casting region are collected:

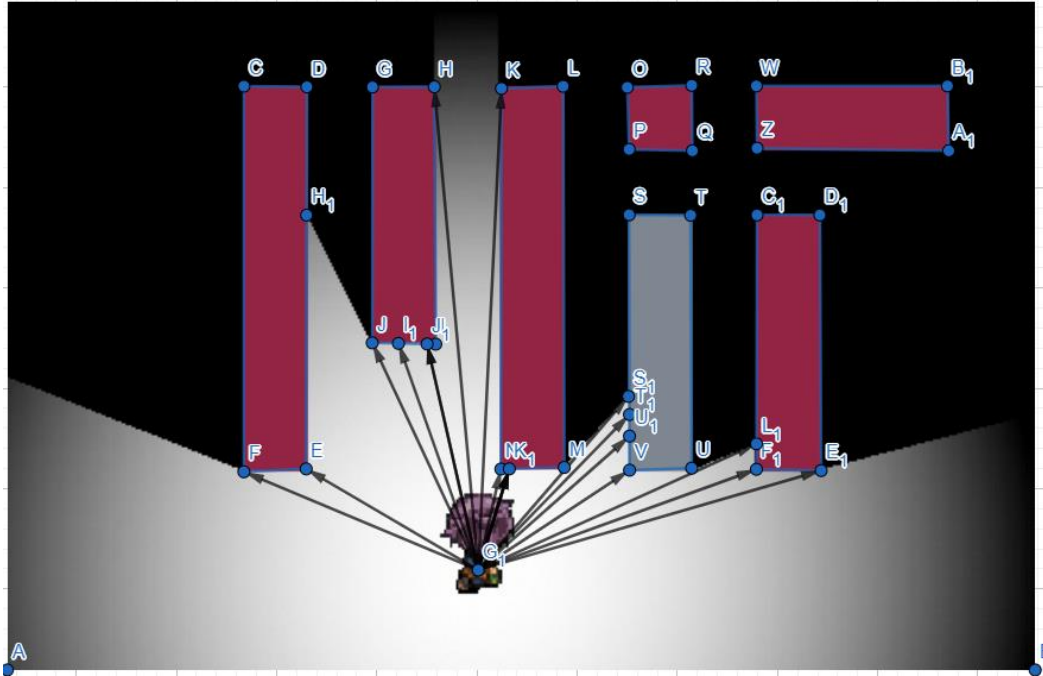


2. Imaginary “rays” are casted from the center to each vertex (by simply constructing line segments from the origin to each vertex):

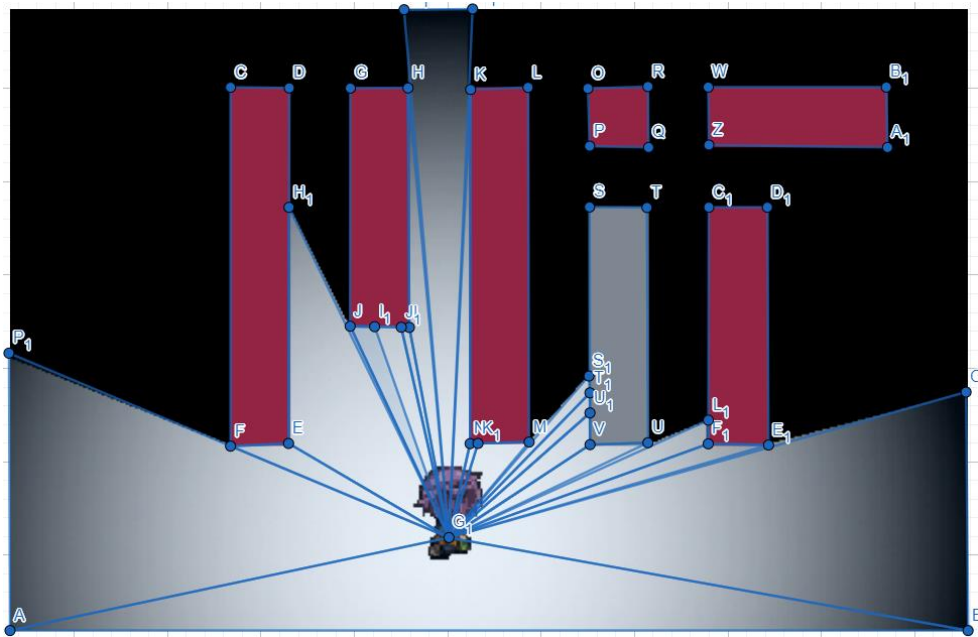


3. For each “ray”, its closest intersection with a wall is found (this is to erase the parts that “penetrated” the wall):





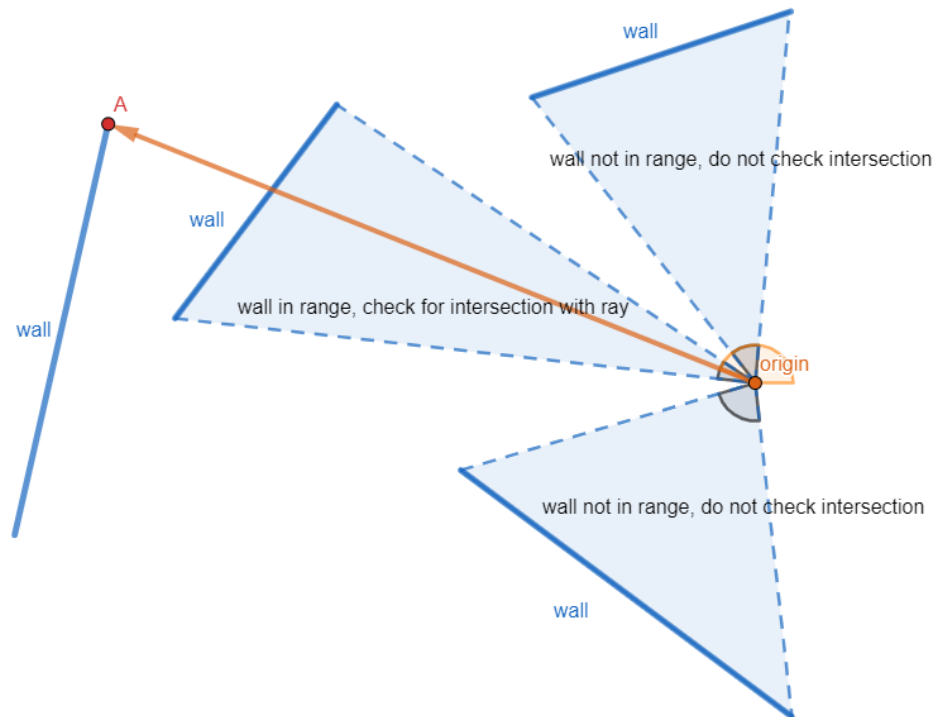
4. The intersections are sorted and triangles are constructed to generate the GLShape for rendering:



While this algorithm seems simple when described in steps, the process of coding the algorithm actually took over a month (during the beginning of grade 11), mainly because of the time taken to implement base functions such as line intersection and 2d ray casting. (I was trying to learning linear algebra at that time.)

Efforts have been taken to optimize the algorithm; for example, the times of line intersection checks are minimized. This is done by first generating an “AngleRangeList” for all walls and, when casting rays, only checking necessary

walls for intersection. In the example below, we are trying to cast a ray from the origin to point **A**, and the top and bottom walls are quickly eliminated since their “angle ranges” do not include the ray’s principal angle.



The reason this is faster is that now, to eliminate a wall, the algorithm only needs to check if the ray is within the “angle range” of that wall (and the “angle range” of all walls relative origin only needs to be calculated once!), whereas before, the algorithm needs to calculate the ray’s intersection with every wall (and this must be repeated for all rays – on all walls!).

This was able to improve the performance of the algorithm by around 30% during testing (with over 200 vertices in range), because many useless checks are avoided. However, the algorithm’s worse case remains  $O(n^2)$  where  $n$  is the edge count. In practice, the algorithm usually takes less than 1~2 millisecond per frame.

## VI. Entity System

### ➤ PURPOSE

A game, in essence, is just interactions between different entities. In Fengine, we classify entities using three classes: Object, Figure, and GameItem. Each of these classes acts like an “interface”, providing base functionality as well as virtual functions that can be implemented by game developers themselves.

However, the entity system one of my earlier works (the framework was laid during the beginning of my grade 11 year), so some parts of the entity system may not be the smartest design. If I were to program the system now, it would be vastly different 😊.

### ➤ EFFECT

All entities are updated and drawn every frame. This means that every entity will contain an Update and a Draw function.

The Update function allows the entity to update internal states. For example, a monster may choose to update its behavior according to the position of the player.

The Draw function allows the entity to render itself using the rendering system. For example, an object may simply proceed to draw its body animation.

The Update and Draw functions of all entities are called automatically by the [Map](#) class every frame once they are added to a Map.

### ➤ TECHNICAL DETAIL

#### ❖ Object (Object.h)

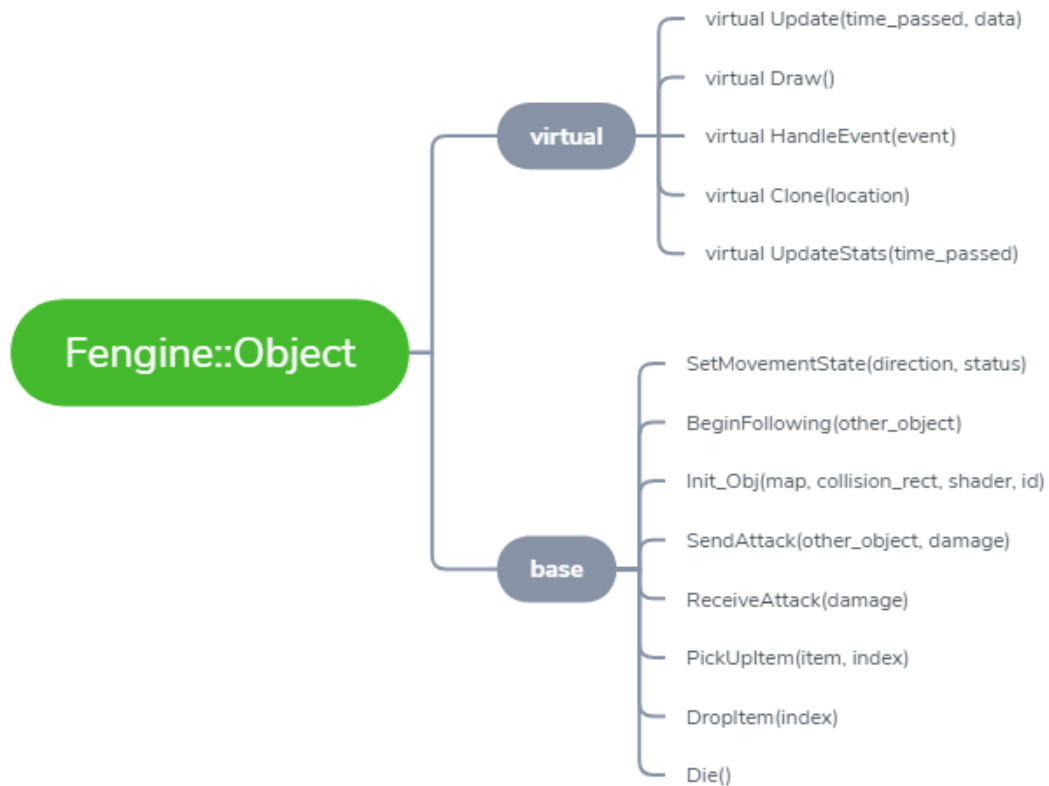
The need for an Object class emerged when more than one types of object appear. For example, it would be inefficient for a system to treat Player and Monster as totally different entities:

```
//inefficient
player_.Update();
monster_type1_.Update();
monster_type2_.Update();
//. . .
monster_typen_.Update();
```

```
//efficient
for(auto & object : objects)
    object.Update();
```

To efficiently update and render the player and the monster, it is better to have them share a base class, which I called Object. It made more sense to treat both player and monster as Object so that generic calls can be made on them. To implement this, I employed polymorphism. Similar techniques are used for Figure and GameItem.

The following functions are in my opinion the most significant ones (the declaration alone, without definition, of the Object class is 400 lines long, so this is only a part of it):



To create a new custom type of object, the developer just needs to create a new class that derives from Object.

The virtual functions shown above can be overridden by the derived class. For example, since the behaviors of different objects vary, they can override the virtual Update function differently. This is left to the game developers to decide.

Having a structure like this allows flexibility and, at the same time, regulates the standard features all game objects should possess.

A list of object utility functions is available (see “base” tree in diagram). These functions allow developers to easily prototype object behaviors without worrying much about the underlying details. For instance:

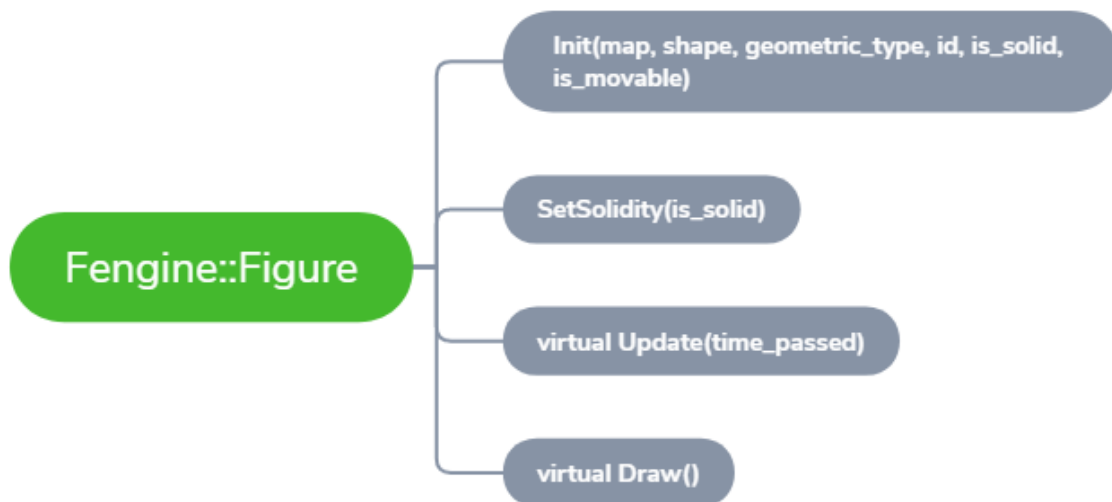
- `SetMovementState(direction, status)` allows one to control the movement of object (such as movement direction)
- `BeginFollowing(other_object)` allows one object to follow another. The A\* algorithm takes obstacles into account and generates the shortest path the object can take on the 2D tiled map to follow another object.
- `GoTo` also uses A\* to generate the shortest path, but to a fixed coordinate in map.

The interactions between objects and maps are also standardized through Object’s `SendAttack`, `ReceiveAttack`, `PickUpItem`, and `DropItem` base functions, which fire events in the [Event System](#).

#### ❖ Figure (Figure.h)

The Figure class is used to represent entities that are normally considered parts of the map, such as walls and doors. A Figure object may or may not be stationary, it completely depends on the type of Figure. For example, a `SlidingDoor` (a custom class deriving from Figure) is able to move when a certain event is fired in the [Event System](#).

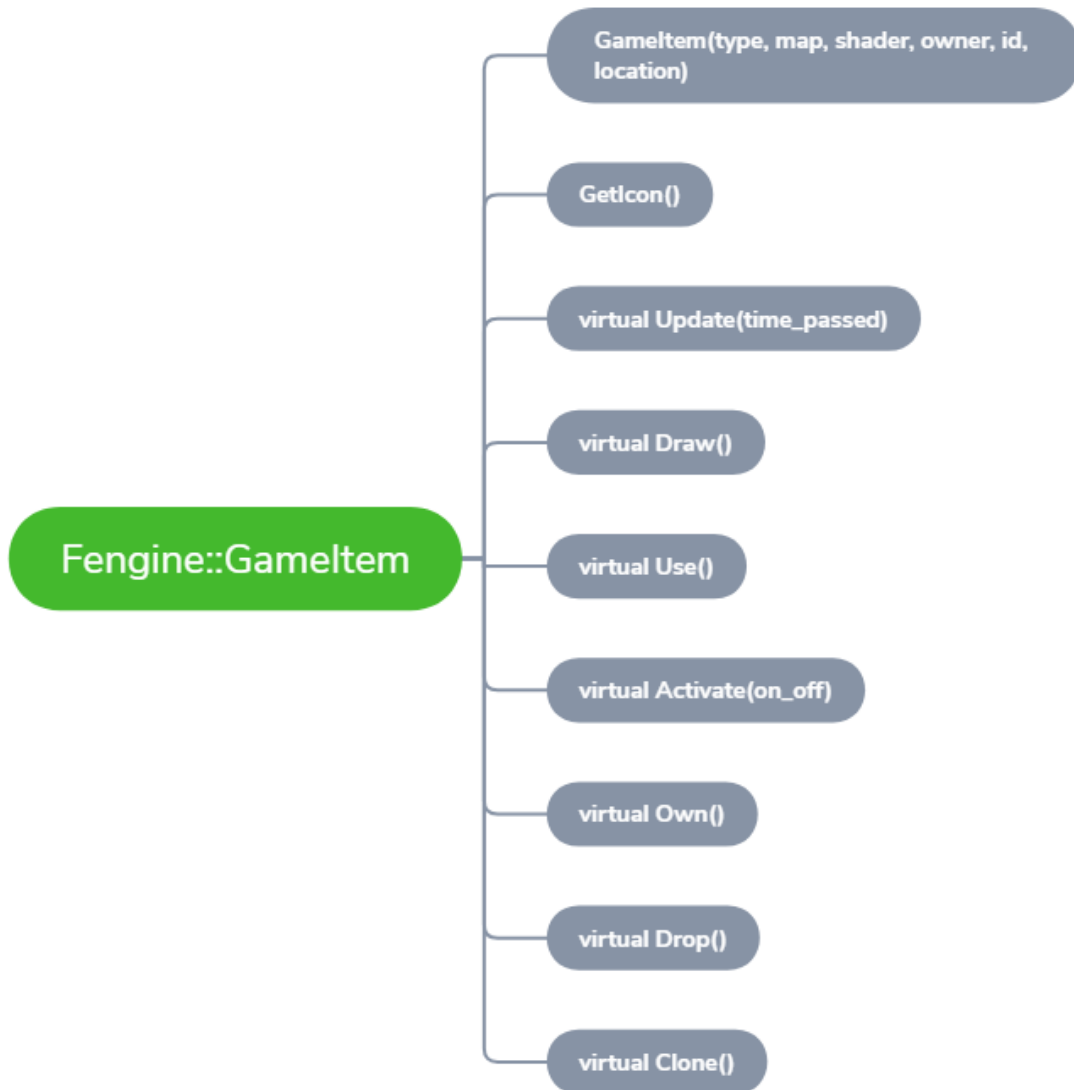
This class used to have its own collision detection algorithm against Objects, but now the [physics system](#) handles that. Similar to the Object class, the base Figure provides base functions that the derived figure classes may override:



- `Init` offers one way to initialize the figure, and another way is by using the constructor of the class. Both ways give the same result, but allow flexibility on the developer's side: they get to decide when to initialize the objects.
- `SetSolidity` function is used to set whether the body of the figure is solid (true if the figure blocks objects, false otherwise)

❖ `GameItem` (`GameItem.h`)

The following diagram shows the most important functions of the `GameItem` class:



A `GameItem` object, as the name suggests, holds a game item. It can represent anything that an `Object` can carry.

- Objects can pick up and drop off items by calling the `Object::PickupItem` and `Object::DropItem` base functions. The two base functions of the `Object` class in turn make calls to the item's `GameItem::Own` and `GameItem::Drop` functions.
- Objects can use its item by calling the item's `GameItem::Use` and `GameItem::Activate` functions. Other convenient alternatives are also provided, such as `GameItem::Toggle`. Note that these item functions are pure virtual, meaning that the developers must implement them if they are creating a new item (because a `Handgun::Use` would be very different from a `MachineGun::Use`).

The interaction between `Object` and `GameItem` is designed in such a way that once an object picks up an item by calling its own `Object::PickupItem` function, it is allowed to call the member functions of the `GameItem`. It may seem bizarre that sometimes we need to call the `Object`'s functions to operate an item, while at other times we call the item's functions. But, this logic makes sense to our team because, in real life, once one **picks up** an item (**by calling his/her own "Hand" function**), he/she is able to **use** the item (**by calling the item's "Use" function**).

## VIII. Map System (Map.h)

### ➤ PURPOSE

Maps are an integral part of a game. Unlike many small scale games, where maps can be hard-coded into the source code, a game engine must be able to load map files, because modifying an external file (such as JSON or XML) is much more efficient than changing and recompiling the source code. I designed Fengine to be compatible with [Tiled Map Editor](#) so that levels can be created without changing or recompiling any source code. In addition, Fengine provides the ability to read in customized configuration files called “ObjDef” for different game entities.

Map is also the “container” of all entities introduced previously; all entities are contained in Map objects and are able to make queries on the environment using utility functions provided by the Map.

### ➤ EFFECT

- Game developers are able to create, modify, and update maps/levels using the Tiled map editor, without changing any source code.
- Behaviors/stats of objects can be altered using ObjDef files

### ➤ TECHNICAL DETAIL

#### ❖ Overview

Essentially, all a game is doing is **update** and **render**. Therefore, the map operates by its two most important functions: `Map::Update` and `Map::Draw`.

`Map::Update` makes calls to the `Update` functions of all entities it contains:

```
//simplified version
void Update(UINT32 time_passed) {
    //update entities
    for(auto & entity : entities) {
        entity->Update(time_passed);
    }
}
```

`Map::Draw` functions similarly to this.

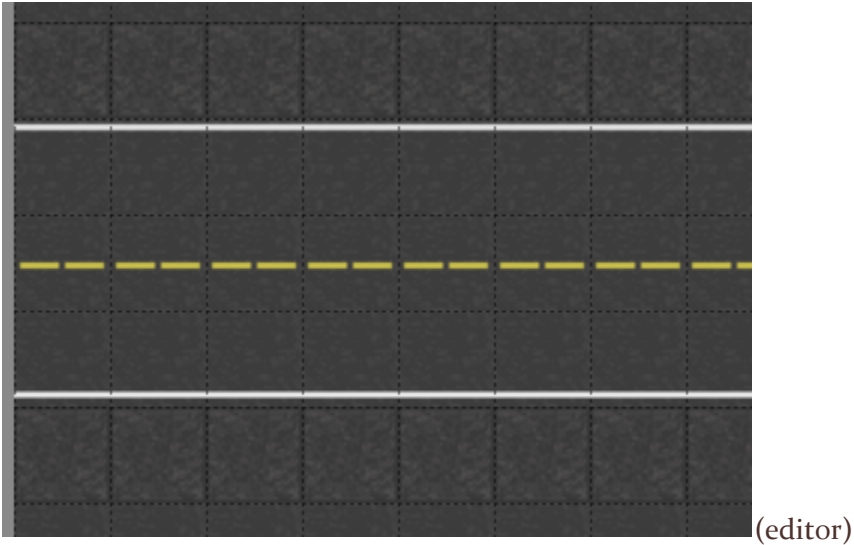
#### ❖ Map Loading (`Map::LoadMap`):

In the level editor Fengine supports, there are mainly two things that can be done:



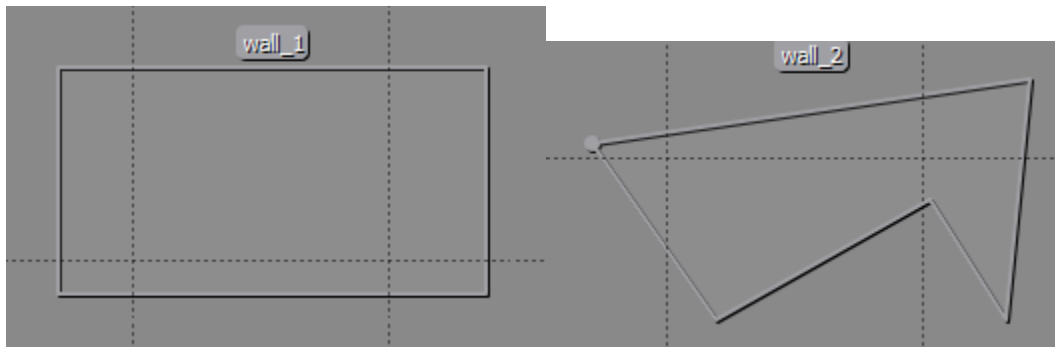
### 1. Creating tile layers

This process is like installing floor tiles in real life, developers can “paste” the floor textures from a file called “tileset”:



### 2. Creating objects:

Wall entities in Fengine can be created with a rectangle or a polygon;



(rectangular and polygonal walls)

To create other more complex objects, developers will need to:

1. Create the appropriate shape in the editor. In the case of an Object, a point will suffice:



2. Set the type of the shape to something descriptive – such as Player, Flashlight, or SlidingDoor. Fengine will need this to find the appropriate “ObjDef” file and constructor at runtime.

|          |                                     |
|----------|-------------------------------------|
| ID       | 10                                  |
| Template |                                     |
| Name     | player                              |
| Type     | Player                              |
| Visible  | <input checked="" type="checkbox"/> |
| X        | 48.00                               |
| Y        | 79.00                               |
| Rotation | 0.00                                |

3. Specify an ObjDef file

An ObjDef is a separate file that specifies the properties of an entity class in Fengine. It is also created using Tiled.

In this way, game developers don’t need to change any source code in order to change, say, the default speed of Player objects, or color of Flashlight objects; they just need to edit the ObjDef file, zero compilation required:

| Custom Properties    |        |
|----------------------|--------|
| ObjType              | PLAYER |
| SKILL_POINT_Agility  | 1      |
| SKILL_POINT_Stealth  | 1      |
| SKILL_POINT_Strength | 1      |
| STAT_Health          | 100    |
| STAT_HealthRegen     | 2      |
| STAT_Speed           | 150    |
| STAT_Stamina         | 100    |
| STAT_StaminaRegen    | 5      |

(in editor)

```

33         static float def_Health;
34         static float def_HealthRegen;
35         static float def_Stamina;
36         static float def_StaminaRegen;
37         static float def_Speed;
38         static float def_Agility;
39         static float def_Stealth;
40         static float def_Strength;

```

*(Player.h) - where the stats will be*

loaded into)

How does this happen?

First, the type specified in the map editor (“Player”, in the previous example) will be used to load the ObjDef file of that type:

(in editor)



*(Map.cpp) – code that takes the Type and find the corresponding ObjDef file path*

```

1046                                     //append "Def.json" and get the ObjDef file this object uses, Load it
1047                                     std::string defPath = "Resources/Game/ObjDef/"+type+"Def.json";
1048                                     LoadObjDef(defPath,logger_system_);

```

(in the PlayerDef.json ObjDef file)



*(Map.cpp)*

```

1223  ☐                                     else if (type=="PLAYER")
1224                                     Player::LoadDef(DefPath);

```

Finally, the static `Player::LoadDef` function, which is implemented by the developers, will be called to populate the attributes of the `Player` class, and every `Player` object created will have identical initial stats.

🔧 *How I plan to improve:*

However, as you can see, the process of mapping a string to a class (“PLAYER” to `Player`) is hardcoded and nested in an “obscure” if-statement in `Map.cpp`. This means that whenever the developers add a subclass of `Object`, they need to modify the code in `Map.cpp`. Very inconvenient and not at all logical. One way to improve this is to have a separate configuration file that developers can modify. So instead of this:

```

1223  ☐                                     else if (type=="PLAYER")
1224                                     Player::LoadDef(DefPath);

```

We have something like:

```

TestEntry g_testEntries[] =
{
    {"Shape Cast", ShapeCast::Create },
    {"Time of Impact", TimeOfImpact::Create},
    {"Character Collision", CharacterCollision::Create},
}

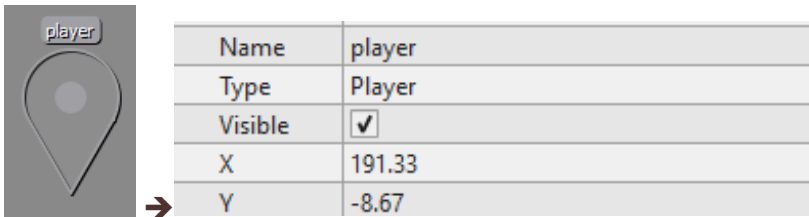
```

[<https://github.com/erincatto/Box2D/blob/master/Testbed/Tests/TestEntries.cpp>].

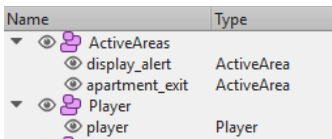
This allows mapping with only one line of code addition.

### ❖ Entity Query System

As seen previously, in the map editor, every entity appears as a shape with an in-game type specified:



They can even be put into layers with descriptive names:



, which encourage the content creators to be organized.

However, as the JSON map file gets loaded into its runtime twin – `Fengine::Map`, these well-organized objects will be distributed into simple lists, because they are now optimized for machines, not humans:

```

437         std::vector<Obj_ptr> objects_;           //contains all the objects
438         std::vector<Object*> dead_objects_;     //!< the object that died in
439         std::vector<Figure_ptr> figures_;       //contains all the figures
440         std::vector<Item_ptr> items_;          //contains all the items of
441         std::vector<Area_ptr> areas_;          //contains all active areas

```

This means that the debugging process for game developers becomes difficult. For example, it would be difficult to track what is happening to “that Player object named ‘player’ in the editor”, since now they are stored in different lists. To resolve this, I developed the Entity Query System, which is responsible for storing the name, id, type, and the pointer to the runtime object of each entity that’s loaded from map file:

```

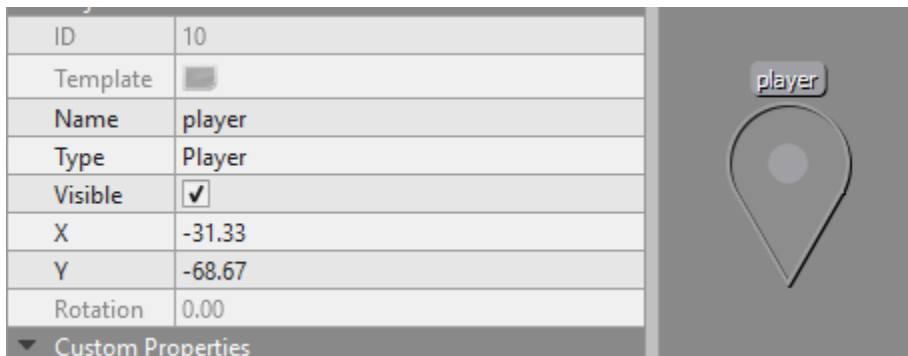
69  /**
70   * every object in Tiled translates into an EntityInfo, which remains constant throughout the Map's life.
71   * this structure can help map query information.
72   */
73  struct EntityInfo
74  {
75      std::string name;           //not unique
76      int id;                    //unique across all entities
77      std::string type;         //type of this entity
78      void* object;             //pointer to Fengine object that represents this entity
79  };
--

```

The most useful query functions are:

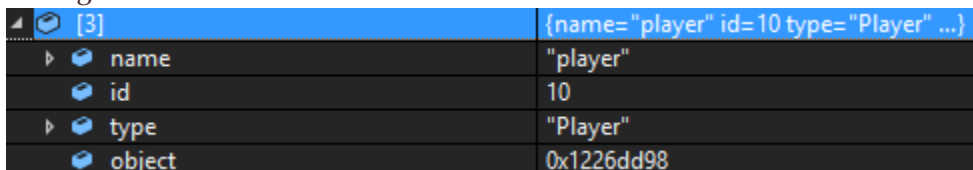
- GetEntitiesByName
- GetEntityByID
- GetObjectByID
- GetItemByID
- GetFigureByID
- GetActiveAreaByID

When accompanied with query functions shown above, this forms a robust lookup table that allows easier debugging. For example, an entity called “player” in the JSON map file -



- can be found

during runtime:



(in debugger)

Most importantly, this will open a door to scripting, because now entities can be reached by identifications that are known before runtime, such as the name or id in the Tiled editor. See more about scripting in the next chapter.

## ❖ Utility Functions for Objects

NPC objects often require “intelligence” so that they can make informed decisions. For this, the Map class provides a set of utility functions that allows callers to make queries about the environment:

(Map.h)

```
132 //Get list of objects at a location (sprite rect is used to determine)
133 std::vector<Object*> GetObjectAtLoc(Point2F location);

144 //return a list of Items in range, returns maximum of itemcount of items
145 std::vector<GameItem*> GetItemInRange(Point2F location, float range, int itemcount);

147 //return a list of Objects in range, returns maximum of objectcount
148 std::vector<Object*> GetObjectInRange(Point2F location, float range, int objectcount, Object* exclude = nullptr);
149

229 /**
230  * get the player
231  * @return the pointer to the player
232  */
233 Object* GetPlayer();
```

There are many more (some of them aren't documented with Doxygen because they are used by the Map class itself).

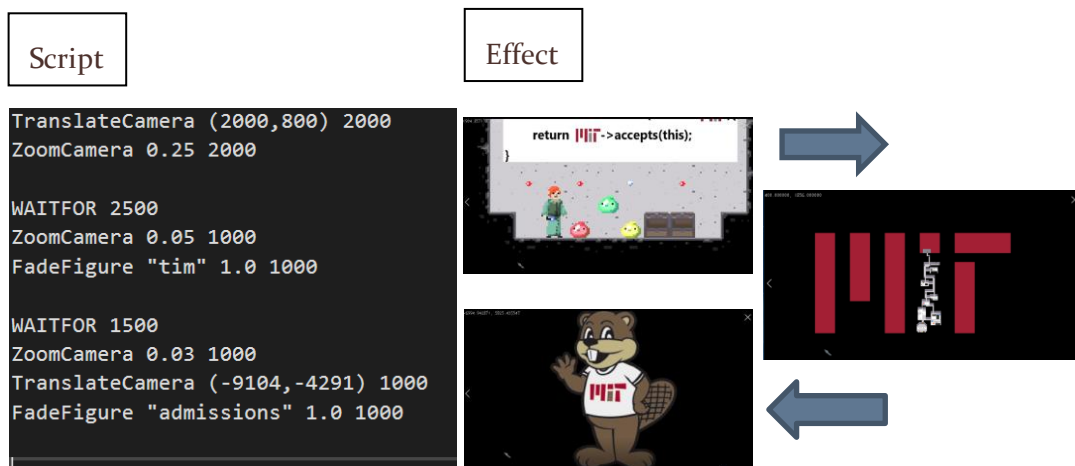
## IX. Scripting System, Event System, and Functor and Timers

### ➤ PURPOSE

Fengine's scripting system allows content creators to modify the gameplay without changing any source code. And this is with the help from multiple other systems including the event system, the functors, and the timers.

### ➤ EFFECT

The maker video's camera movement is scripted, and the script of the zoom out animation, for example, is this:



In the video, every camera movement is scripted using the ZoomCamera and TranslateCamera commands.

The scripting system also supports blocking commands like WaitFor that blocks the execution of the script, but not the thread the game is running on (this is crucial!).

And it is only possible to have all this functionality with the help of Functor, EventManager, and Timer classes.

### ➤ TECHNICAL DETAIL

#### Functor (Functor.h)

Functor are objects that are "callable". They work by storing function pointers that can be called later as needed, providing a means of callback in Fengine's event driven components.

Even though the C++ standard library already has the convenient std::function class, Functor serves as a wrapper class around it and regulates the standard callback function type. It requires functions to be in the form of: void (void\*) - "accepts void\*" and returns

void". In Engine, calling a Functor is just calling a function and passing in a void\* argument:

```
//assuming functor has been constructed already  
  
int argument = 5;  
  
functor(&argument);    //here, a pointer to variable argument is  
                       //passed as a void*
```

Here are some real usages of Functor:

- Using member function as callback

(Map.cpp)

```
1323 //subscribe to ATTACK events  
1324 event_manager_.SubscribeEvent((int)MapEventManagerEventType::ATTACK, Functor(this, &Map::HandleAttackEvent));
```

Here, the map subscribes to attack events using its Map::HandleAttackEvent.

- Using lambda expression as callback

```
427 /**  
428  * translate the camera from current position to a new position  
429  * @param position the new position  
430  * @param time the time taken to finish translating  
431  */  
432 void TranslateCamera(const Point2F & position, int time);  
433 Functor translate_camera_ = Functor([this](void* v)  
434 {  
435     auto param_list = static_cast<std::vector<ScriptCommandParam>*>(v);  
436  
437     this->TranslateCamera((Point2F)param_list->at(0), (int)param_list->at(1));  
438 });
```

Here, the function TranslateCamera is wrapped into Functor translate\_camera\_ using a lambda expression. (more about this in the [Script](#) section)

#### ❖ EventManager (EventManager.h)

The event system allows entities to subscribe to and fire events. Through the event system, map entities are no longer communicating directly with each other in a chaotic way, but in a centralized, broadcasting way. Event subscription requires an event type and a Functor as callback.

Example for subscribing to events:

(Map.cpp) – subscribing to an attack event

```
1323 //subscribe to ATTACK events  
1324 event_manager_.SubscribeEvent((int)MapEventManagerEventType::ATTACK, Functor(this, &Map::HandleAttackEvent));
```



(*SlidingDoor.cpp*) – subscribing to active area events so that the door knows to move when an entity enters

```
297 //subscribe to all activearea events
298 event_manager->SubscribeEvent(int(MapEventManagerEventType::ACTIVEAREA_EMPTIED), Functor(this, &SlidingDoor::HandleAreaEvent));
299 event_manager->SubscribeEvent(int(MapEventManagerEventType::ACTIVEAREA_FILLED), Functor(this, &SlidingDoor::HandleAreaEvent));
```

Example for firing events:

(*Map.cpp*) – firing key press events

```
520 //fire key press event
521 if(input->is_key_pressed_)
522 {
523     event_manager_.FireEvent(Event(MAP_EVENTMANAGER, (int)MapEventManagerEventType::KEY_PRESSED, "Map", &input_));
524 }
```

(*Object.cpp*) – firing item pick up / drop off events

```
67 //when picking up an item, fire an event
68 map->GetEventManager()->FireEvent(Event(MAP_EVENTMANAGER, int(MapEventManagerEventType::ITEM_PICKEDUP), "Object", this));
```

```
110 //when dropping an item, fire an event
111 map->GetEventManager()->FireEvent(Event(MAP_EVENTMANAGER,
112     int(MapEventManagerEventType::ITEM_DROPPED),
113     "Object",
114     this)
115 );
```

EventManager's centralized subscription model greatly simplifies not only the complex game logic, but also the debugging process. During debugging, developers can easily see who subscribed to what just by viewing the subscriber table at runtime.

You may also have noticed the Event class, but I won't bore you with it as it is rather complicated and boring. Just know that it's a structure that carries information about the events happened.

#### ❖ Timer (Timer.h)

As I programmed the event system, I was shocked that no existing C++ library supports object oriented timers. Most libraries, including the C++ standard library, provided only `std::this_thread::sleep_for` function, which blocks the whole thread. Moreover, the best function available, SDL's `SDL_AddTimer`, took in a plain static function as callback, which is insufficient for OOP event systems where the callbacks should be **member functions of objects**. The Timer class was my invention that serves as the base of the scripting and event systems.

In Fengine, Timer objects are created using Timer's static function `AddTimer`:

(*Timer.h*)

```

33 |     /**
34 |     * add a timer
35 |     * @param time time in ms
36 |     * @param f functor to call when time is up
37 |     * @param param the param that is passed to f when the time is up
38 |     * @return a pointer to Timer object, which can be used to remove timer with Timer::RemoveTimer
39 |     * @note timer is automatically deleted after calling the functor
40 |     */
41 |     static Timer* AddTimer(Uint32 time, Functor f, void* param);

```

AddTimer will return a handle to the timer created, allowing the user to pause, resume, delete, and make queries about the timer (such as getting the remaining time). The class is designed this way so that every timer created is safely stored in a static container and managed by the Timer class itself. This way the user won't need to worry about deleting the timer because, after all, the Timer class is responsible for deleting each Time object it created. Of course, the users are also given the freedom to delete a timer before its due time (using Timer::DeleteTimer).

The internal of Timer makes use of SDL's timer utility – [SDL\\_AddTimer](#), which takes in basically the same parameters as Timer::AddTimer does and returns an integer “id”, but the callback must be a static function.

The way Timer class utilizes SDL's timer utility is rather complicated to explain, so please be ready to “parse” the rest of this section.

The Timer class uses SDL\_AddTimer to register timers by passing in the timer duration, the static callback (Timer::SDLTimerCallback), and the pointer to the new Timer's timer\_id. The SDL\_AddTimer will return an id for the Timer to store.

In C++, this is expressed in one line of code:

*(Timer.cpp)*

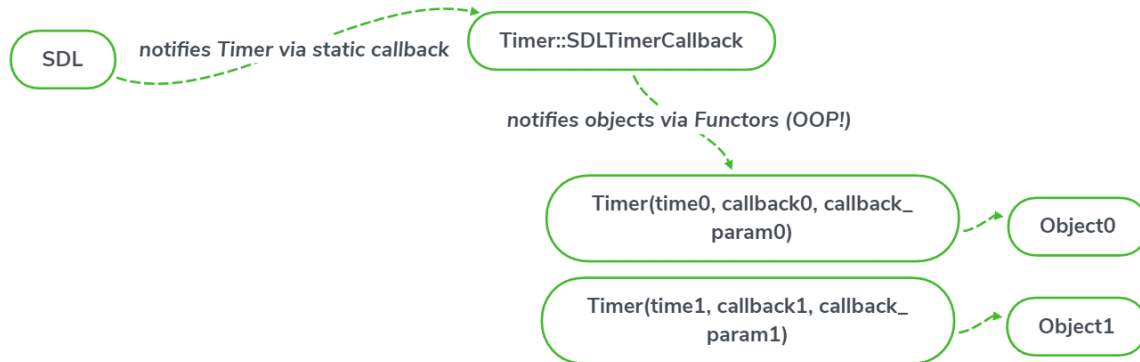
```

21 |     //add timer to SDL with timer_id_set to value returned by SDL_AddTimer
22 |     timers_.back().first->timer_id_ = SDL_AddTimer(time, &SDLTimerCallback, &timers_.back().first->timer_id_);

```

This means that, when the time is up, SDL will call the second argument (the static function Timer::SDLTimerCallback) and pass in the third argument (a pointer to the timer\_id), which will be used by Timer::SDLTimerCallback to find the due Timer objects. In other words, Timer::SDLTimerCallback maps static callback to object oriented callback:

## How does Fengine::Timer Work?



I hope you get the idea. The Timer class's implementation itself isn't complex but it's difficult to explain in words...

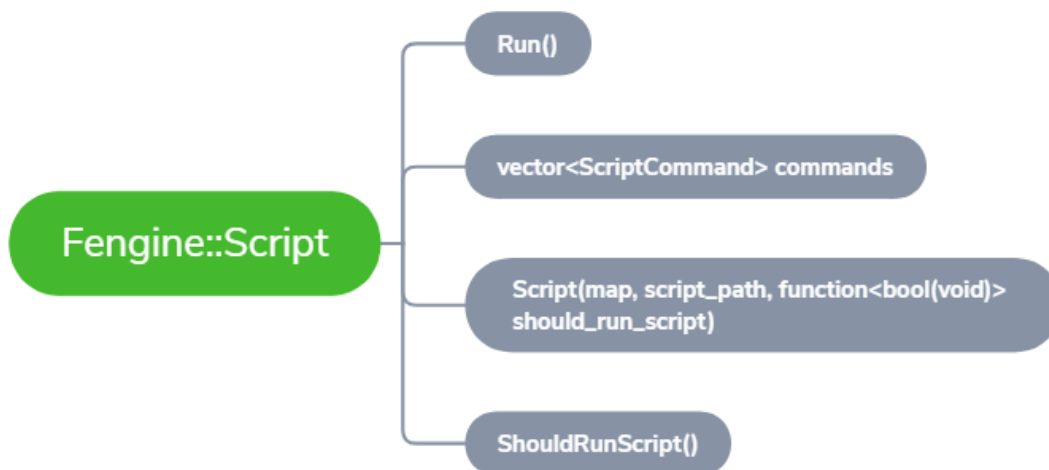
In the end, Fengine's Timer class allows for event scheduling in an object oriented fashion.

### ❖ Script (Script.h)

Last but not least, the Script class is the most interesting feature in Fengine. Just like how the map system uses map files to avoid hardcoded levels, the scripting system uses script files to avoid hardcoded game flow.

### 🚦 Script Parsing

Given a script file, the Script constructor will parse it into a list of ScriptCommands, which will be executed one by one when Script::Run is called.



Each line in the script corresponds to a `ScriptCommand` in `Fengine`. Each line must contain a command type and a list of required parameters, take the first line of the sample script:

```
TranslateCamera (2000,800) 2000
```

Here, `TranslateCamera` is the command type. `(2000,800)` is the first argument, specifying a position in the game's world. And `2000` is the second argument, specifying the total time the camera should take to move.

There are dozens of examples, here's another one:

```
ShowHintText "Enter the store..(not implemented)"
```

Again, `ShowHintText` is the command type, followed by a single string argument.

Each `ScriptCommand` parsed from each line will contain a type, a list of arguments, and a `Functor` containing the "task". The `Functor` will be called when the command is running, and the list of arguments will be passed into that `Functor` as a `void*`.

This script introduces two challenges, a) we cannot simply split the line by spaces, because each token may contain spaces, like the string argument above b) we cannot store the arguments in a list, because they have different types: for `TranslateCamera`, the parameter types are `{point, int}`, whereas for `ShowHintText`, the type is `{string}`.

This means that I cannot simply "split" or use a for loop to parse the parameters of each line. It also means that something fancy like a recursive descent parser is not necessary – all I need is the ability to read in the command type and the list of arguments.

I solved issue b) by introducing the `ScriptCommandParam` structure (I know, the name is long and ugly, but necessary). This structure is able to store any "primitive" data types including string, int, float, bool, `rect`(rectangle), and point.

*(Script.h) – data types supported by `ScriptCommandParam`<sup>3</sup>*

```
91 enum ScriptCommandParamType
92 {
93     PARAMTYPE_STRING,
94     PARAMTYPE_INT,
95     PARAMTYPE_FLOAT,
96     PARAMTYPE_BOOL,
97     PARAMTYPE_POINTER,
98     PARAMTYPE_RECT,
99     PARAMTYPE_POINT
100 };
```

---

<sup>3</sup> `PARAMTYPE_POINTER` is not fully supported because pointers only exist during runtime

By storing a list of ScriptCommandParam, we are essentially storing a list of any data types, thus solving the problem of multiple data types in one list. If you are interested in how this structure works, I'll give you a brief intro here. The structure contains an anonymous union:

(Script.h)

```
190 union
191 {
192     std::string string_;
193     int integer_;
194     float decimal_;
195     bool boolean_;
196     Rect4F rect_;
197     void* pointer_;
198 };
```

And an enum that stores the type of data stored in the union:

```
217     ScriptCommandParamType type;
```

It also has constructors and casting operators for all the “primitive” types:

```
118     ScriptCommandParam(const std::str 158     operator std::string() const;
119                                     159
120     /**                               160     /**
121     * create an integer command param 161     * casting operator to integer,
122     */                               162     */
123     ScriptCommandParam(int integer); 163     operator int() const;
124                                     164
125     /**                               165     /**
126     * create a decimal command param 166     * casting operator to float, w
127     */                               167     */
128     ScriptCommandParam(float decimal) 168     operator float() const;
129                                     169
130     /**                               170     /**
131     * create a boolean command param 169     */
132     */                               171     * casting operator to bool
133     ScriptCommandParam(bool boolean);
```

Now issue b) is solved, and we need to solve issue a). The solution is to have different parsing methods for different types. First, the parser retrieves the substring before the first space, because we know that the command types (ZoomCamera, TranslateCamera) do not contain spaces.

(Script.cpp)

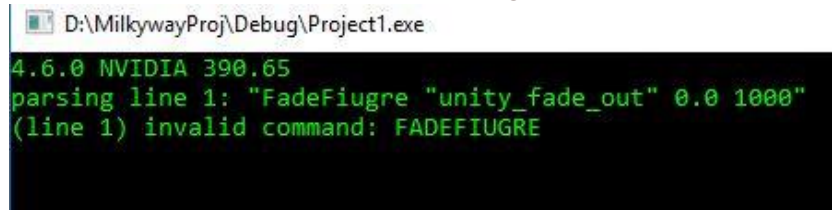
```
33     //get the first token containing the command type
34     std::string command_token = GetSubStrBeforeChar(line, '');
```

Then, the parser will call ScriptCommandParam::ParseParamFromText:

(Script.h)

```
208 /**  
209  * given a string of parameters, this function attempts to parse the string  
210  * @param input the string to parse, this string should not contain any leading/trailing space  
211  * @param output the vector to receive the output of this function  
212  * @param types list of resulting types  
213  * @return error message, if successful, this string will be empty  
214  */  
215 static std::string ParseParamFromText(std::string input, std::vector<ScriptCommandParam>* output, std::initializer_list<ScriptCommandParamType> types);
```

This function takes in a string and a list of target types as inputs and parses out a list of ScriptCommandParam. An error message will be returned if anything goes wrong:



```
D:\MilkywayProj\Debug\Project1.exe  
4.6.0 NVIDIA 390.65  
parsing line 1: "FadeFiugre "unity_fade_out" 0.0 1000"  
(line 1) invalid command: FADEFIUGRE
```

The function parses by using two key utility functions: GetEnclosedSubStrings and SplitStringByDelim (both implemented in Utility.cpp).

- GetEnclosedSubStrings finds what's enclosed in a pair of characters:

```
//parse out data enclosed by ""  
auto result = GetEnclosedSubStrings(input, "\"\"", 1);  
  
auto coord_string = GetEnclosedSubStrings(input, "()", 1).at(0);
```

- SplitStringByDelim splits the line by a delimiter (usually space):

```
content = SplitStringByDelim(input, ' ')[0];
```

So, given a list of target types `{ PARAMTYPE_STRING, PARAMTYPE_POINT }`

, ScriptCommandParam::ParseParamFromText employs different parsing methods:

- For string: find content enclosed by ""
- For numerical types (int, float, bool): split input by spaces.
- For Point or Rect: find content enclosed by () and split content by comma to get x,y,w,h

In the end, the function is able to output a list a ScriptCommandParam from an input string.

#### Script Execution

When Script::Run is called, one would think that commands are executed one by one using something like a for loop... However, the problem is that Script::Run must return

immediately to prevent blocking the game thread, but some commands are blocking statements that require some time to finish, such as the `WaitFor` command. So running commands one by one would just block the game's update and rendering thread.

I tackled this problem by treating commands differently. The script calls `Script::RunNext` recursively, finishing all non-blocking statements. Then, when a blocking statement appears, the Script will create a **Timer** with a callback to `Script::RunNext` and return immediately. After the specified amount of time, the script will be running once again. This simulates a blocking statement (in script execution) but without blocking the actual game thread.

## X. Physics System

### ➤ PURPOSE

Originally, the engine did not really have a separate physics system, because the collision detection and response is handled by the Figure class. Every update, each Figure object will loop through all other Figure and Object entities and detect any collision. However, this design quickly broke down. For example, when newer types of entities are introduced to the game, such as GameItem, it must inherit from Figure just to obtain the collision detection functionality. Not to mention the collision detection algorithm got increasingly slow as the entity count grew (it was  $O(n^2)$  where  $n$  is the entity count).

There must be a generic way of solving collisions without considering the actual type (such as Object, Figure, etc) of the entity. And this is why the Composition over Inheritance design principle is the best solution. If an entity type needs physics, then it can own a Body and add it to a World. Every entity can choose to obtain different physical properties by configuring their Body component.

In Fengine, only kinematics (physics without force) is implemented.

### ➤ EFFECT

- Any entity types are allowed to own a Body component and add it to the World.
- Uniform acceleration and uniform motion are supported
- Collisions between polygons and circles are supported

### ➤ TECHNICAL DETAIL

#### ❖ World (Kinematics.h)

- Each Map contains a World object, which can be obtained using `Map::GetWorld`.
- As the top-most level in a physics engine, World stores Body pointers in a `std::vector` but also inserts each Body into a quadtree structure – `Fengine::Quadrant` – for fast collision detection.
- Any entity can add its Body to a World by calling `World::AddBody`. Figure's and Object's Body are added by default.

*(Kinematics.h) – what a World contains*

```
205 | class World
206 | {
207 |     std::vector<Body*> bodies_;           //!< all the bodies in th
208 |     ...                                  //!< @note the world does
209 |     ...
210 |     std::unique_ptr<Quadrant> quadtree_; //!< stores elements of b
```

- Every `World::Update` call:
  - All Bodies are updated via `Body::Update`
  - Bodies that moved are recorded and removed from the quadtree:



(Kinematics.cpp) – how bodies updates get handled

```
530 auto moved = body->Update(time_passed);
531
532 //if the body moved, add this to bodies
533 if (moved)
534 {
535     bodies_moved.push_back(body);
536     quadtree_->Remove(body);
537 }
```

- A list of possible colliders of each moved Body is checked:

```
543 //retrive a list of bodies this body can collide with
544 auto colliders = quadtree_->Retrieve(body->GetBound());
```

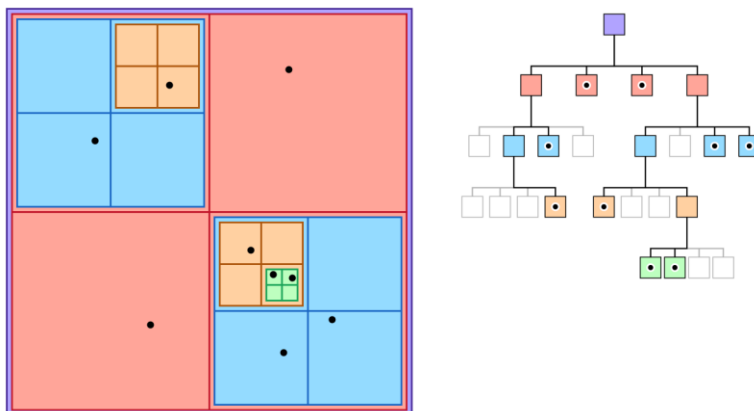
```
//resolve all collisions
for (auto i : colliders)
{
    body->ResolveCollision(*i);
    //i->DrawDebugShape();
}
```

- Removed Bodies are re-inserted into the quadtree structure:

```
554 quadtree_->Insert(body);
```

#### ❖ Quadrant (Quadrant.h)

The Quadrant class is Fengine’s implementation of the quadtree structure. As seen from the previous section, it is used to efficiently find possible colliders. It is a tree structure that splits itself into 4 smaller “quadrants” if the entity count exceeds some number (Quadrant::max\_body\_count).



This structure’s main advantage over brute force is its retrieval time:

- Its Retrieve function easily narrows down the possible colliders given a rectangular bound, because all Bodies in a Quadrant are eliminated if the rectangular bound is not in the Quadrant. If the bound is in the quadrant, then the quadrant may return all the Bodies it contains. However, if the current quadrant is splitted into 4 more child quadrants, it will call the children's Retrieve functions. This function is  $O(\log n)$  where  $n$  is the depth of the tree. See below for the code of `Quadrant::Retrieve`:

```

480 std::vector<Body*> Quadrant::Retrieve(const Rect4F& rect)
481 {
482     //if the rect is not in this quadrant at all, return empty
483     if (!RectCollide(rect, bound_))
484     {
485         return {};
486     }
487     //if the quadrant is not split, get all bodies in this quadrant
488     if (quadrants_.empty())
489     {
490         return bodies_;
491     }
492     //if the quadrant is split, just retrieve from sub-quadrants
493     std::vector<Body*> return_bodies;
494     for (auto & quadrant : quadrants_)
495     {
496         auto bodies = quadrant.Retrieve(rect);
497         return_bodies.insert(return_bodies.end(), bodies.begin(), bodies.end());
498     }
499     return return_bodies;
500 }
501

```

However, the downside is that insert and delete are no longer constant time. They are now  $O(\log n)$ . Still, it is now MUCH faster than the brute force method before with  $O(n^2)$  where  $n$  is the Body count.

#### ❖ Motion (Kinematics.h)

Motion handles Body's...motion. Currently, only uniform acceleration and uniform motion are supported. Developers can:

- Configure the Speed (structure that controls the terminal speed) of the motion by calling `Motion::GetSpeed`:
  - The Speed structures operates by components: a speed component with a name (string) and a magnitude (float) can be added to a Speed via `Speed::AddSpeedComponent(key, val)`. This allows the developers to do something like:

```

speed.AddSpeedComponent("base_speed", 0.3f);
speed.AddSpeedComponent("boost", 0.05f);

```

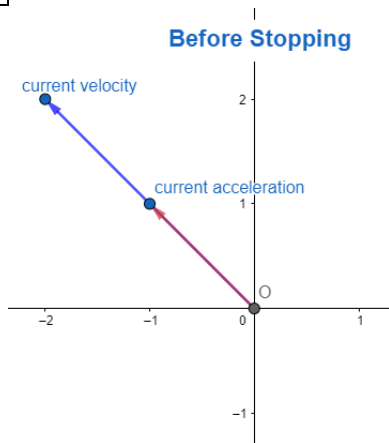
After an add component call, the structure will be marked "dirty" until...

- `Speed::GetSum` sums up the values of all added components. This function is optimized, meaning that the sum is calculated and stored locally during the first `GetSum` call after an add or delete component

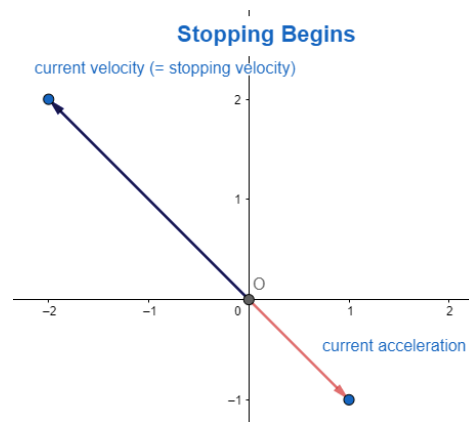
operation. The structure will be marked “clean” until another add/delete call.

- Set acceleration (2 dimensional, in pixel per second squared) by calling `Motion::SetAcceleration`. After setting the direction, the motion will simply accelerate at the specified acceleration, with the terminal speed in mind.
- Stop Motion by calling `Motion::StopWithAcceleration`. Stopping a motion smoothly is simple: just reverse the acceleration; however, **knowing when to stop stopping** is a “challenge”. It requires keeping an acceleration opposite to the current velocity for some period of time such that the current velocity becomes (0,0). It is almost impossible that the current velocity will become (0,0) due to floating point error and, if we attempt rounding, the object may never stop, because the velocity may bypass (0,0) and continue to grow in the opposite direction. Note that calculating the amount of time required using  $\Delta t = \frac{\Delta \vec{v}}{a}$  is also not possible because we cannot guarantee the time between update calls in a game. Fortunately, `Motion::StopWithAcceleration` takes care of this by recording the current velocity  $\vec{v}_{current}$  as  $\vec{v}_{stopping}$  and reversing the acceleration until  $\|\vec{v}_{current} + \vec{v}_{stopping}\| < \|\vec{v}_{stopping}\|$ . This sounds confusing, so please see the illustration below:

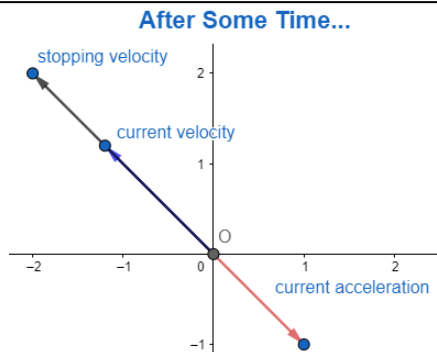
1



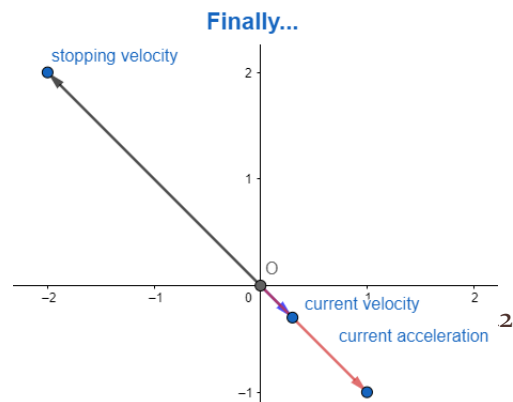
2: stopping begins, reverse acceleration



3:  $\vec{v}_{current}$  shrinks, but still:  
 $\|\vec{v}_{current} + \vec{v}_{stopping}\| > \|\vec{v}_{stopping}\|$



4: Finally,  
 $\|\vec{v}_{current} + \vec{v}_{stopping}\| < \|\vec{v}_{stopping}\|$ ,  
 go to step 5



5: Reset  $\vec{v}_{current}$  and  $\vec{a}$  to (0,0)!

❖ Body (Kinematics.h)

A Body object does 2 jobs only:

- Update its position according to its motion: (*Kinematics.cpp*)

```
259 bool Body::Update(unsigned int time_passed)
260 {
261     //update body position
262     auto displacement = motion_.GetDisplacement(time_passed);
263     if (displacement)
264     {
265         MoveCenter(displacement);
266         if (is_recording_)
267             dist_travelled_ += displacement.Norm();
268         return true;
269     }
270     return false;
271 }
```

- Resolve collision with another Body (recall that the Quadrant only collects all possible colliders, because it's Body's responsibility to actually check if collision occurs and if so, respond to it)

```
321 void Body::ResolveCollision(Body & other)
322 {
```

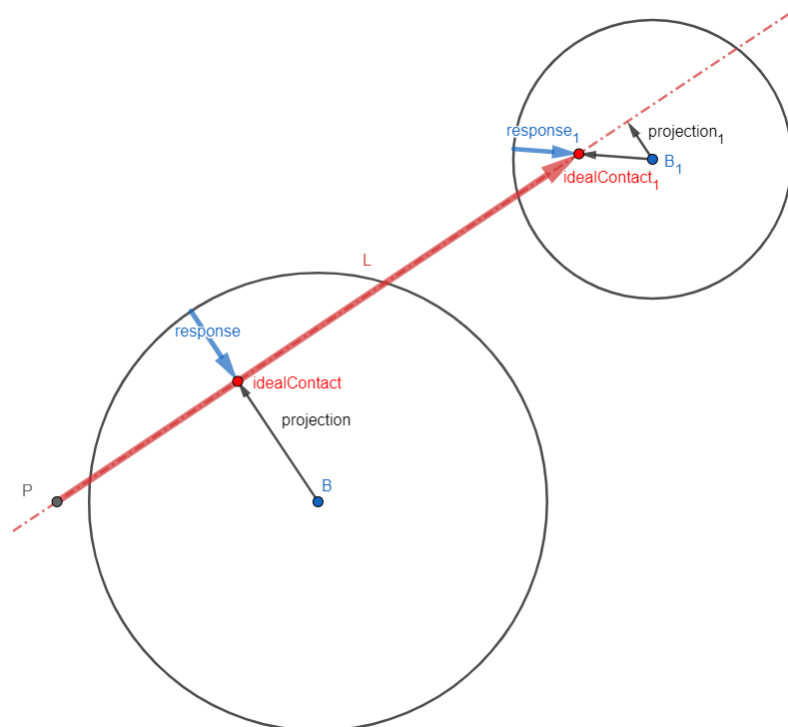
Body currently supports mainly the following types of collision response:

- Polygon – Polygon (convex only)
- Polygon – Line
- Circle – Line

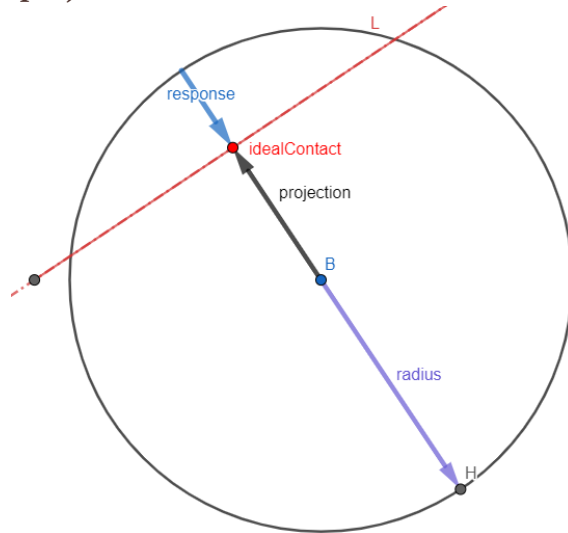
(*Kinematics.h*)

```
243 namespace Collision
244 {
245     /**
246     * functions that detect collision and return the collision response *
247     * always return how much B should be moved to resolve collision *
248     */
249     Point2F Rect_Rect(const Rect4F & A, const Rect4F & B);
250     Point2F Circle_Rect(const Circle & A, const Rect4F & B);
251     Point2F Line_Rect(const Line & A, const Rect4F & B);
252     Point2F Line_Line(const Line & A, const Line & B);
253     Point2F Line_Circle(const Line & A, const Circle & B);
254     Point2F Circle_Circle(const Circle & A, const Circle & B);
255     //an implementation of the separated axis theorem
256     Point2F Polygon_Polygon(const GLShape & A, const GLShape & B);
```

- Rect – Rect is the most common type of detection, it uses AABB bounding box check to find the collision response between two Rect4F (orthogonal rectangle).
- For Polygon – Polygon detection, the engine uses Separate Axis Theorem. It works by finding the projection of both polygons on a set of axes (2d vectors) that are diagonal to each edge of the polygon. For each axis, the algorithm projects both polygons onto the axis. In a loop, the algorithm records the axis with minimum projection overlap. If there is a time when projections have no overlap, then the algorithm ends immediately because it proves that there is no collision. On the other hand, if the projections for all axes, the polygons are colliding. Then, the algorithm will use the axis with the minimum overlap recorded to get the collision response:  $overlap * \widehat{axis}$ , where  $overlap$  is the amount of overlap, a scalar, and  $\widehat{axis}$  is the normalized axis (mag. = 1), a vector.
- Rectangle – Line detection was my intermediate (stupid) solution before understanding the Separate Axis Theorem. It treats the rectangle as lines and does Line – Line check on them. Really, this should just be treated as just Polygon-Polygon intersections.
- Line – Circle detection works the following way:
  - Naming:  $B$  is the circle (and its center);  $L$  is the line (and line segment).
  - Depending on the position of the circle relative to the line, there are 2 possibilities of collision. One possibility is demonstrated using the circle on the left. This happens when the projection of center  $B$  is on the red line segment  $L$ : the *idealContact* is simply defined as the projection of **point B** onto  $L$ . The second possibility is demonstrated on the right, where **point B**'s projection onto  $L$  is not on segment  $L$ : the *idealContact* is the endpoint of segment  $L$  that circle  $B_i$  contains.



- The “*response*” vector (the minimum translation of *B* to separate the shapes) can be obtained by adding the “*radius*” vector to the “*projection*” vector:



, where *projection* =  $idealContact - B$ , and *radius* =  $-r * \widehat{projection}$ .

## XI. Utility

Primitive.h and Primitive.cpp contain a collection of “primitive” data structures used in Engine. The two most important ones are shown below:

### ❖ Point2F – 2D vector (Primitive.h)

Point2F contains 2 floating point values: x and y. It is usually used to store a position in a 2d plane, but it can also function as a 2d vector, which is used extensively by the physics engine. Its alias is Vector2F.

The following abilities are provided:

- default, copy, and (x,y) constructors
- scalar multiplication & division (\*, /)
- scalar and vector addition & subtraction (+, -)
- dot product (\*)
- norm
- “pseudonorm” (just norm squared, used for calculation in situations where the actual length of vector does not matter, such as when comparing the lengths of 2 vectors)
- relational operators (>, <, ==)
- distance from another point
- ToString
- angle formed with another vector, measuring from the origin

### ❖ Rect4F – 4D vector (Primitive.h)

Rect4F contains 4 floating point values: x, y, w(width), and h(height). It usually stores the position of a rectangle, with the top-left at (x,y) and some width and height. However, it is also occasionally used as a “vector”. For example, it is used to describe a constant relation between 2 rectangles, answering a question like this: “what values to change in x, y, w, and h of a known rectangle to reach the other unknown rectangle?”

The following abilities are provided:

- default, copy, (x,y,w,h), (xy,wh), and (xy,w,h) constructors
- SetCenter
- GetCenter
- relational operator (==)
- vector addition and subtraction (+, -)

### ❖ Angle (Primitive.h)

The parameter type and return type for all angles in Engine. See more about [Angle](#) here.

- ❖ Other [Primitives](#) (GLColor, GLVertex, GLShape, Line, Circle...)

## XII. Things to be improved

1. Library data types should be completely abstracted away, because the developers are using Fengine and they shouldn't need to worry about the underlying libraries. (No more `GLuint` or `TTF_Font` or `nlohmann::json`)
2. Possible types of entities shouldn't be hardcoded into the map loading process, instead use a configuration source file that developers should change (Something similar to Box2D's [TestEntries.cpp](#)) (see [here](#) for a concrete example).