Silly Game Engine v0.1

Michael Boza

April 28, 2025

1 Introduction

This text is meant to serve as documentation for the 2D game engine I am developing. The aim of the engine itself is not to be a fun playable experience, but to provide a codebase for one to develop a game which is intuitive as possible. I will divide this document into two chapters. One for those who want to quickly set up and use the project in its current state and another for those curious as to how it works. Feel free to skip to the topics of your interest using the table of contents provided.

The engine will be programmed on C++ with SDL2 for handling graphics and sound.

Contents

2 How to Use

2.1 Installation: Debian based Linux distro

Start by downloading and installing the required libraries: SDL2 and SDL2_image.

```
sudo apt-get install libsdl2-dev libsdl2-image-dev
```

Next, clone the project from the GitHub repository. You might need to install git first. You can rename it to your own project's name in the same command.

```
git clone https://github.com/MiBoza/2D-Game-Engine.git Project_Name
```

I've included a build system in Makefile. It should allow you to simply compile the entire engine together with the most recent demo. For better performance, make sure the variable 'purpose' is set to "release". Now compile and run:

```
make all
./release.obj
```

2.2 Opening a Window

This is the simplest demonstration of the engine's workings I can think of.

For each of these examples, I've included the source code in the Tests directory and a recipe for it in the Makefile. I will do my best to ensure each test compiles and runs correctly whenever I push a new version of the engine.

Let's look at the code inside Tests/Open_Window.cpp:

```
1  #include ”Aggregate.hpp”
2
3  int main(){
4      Aggregate* game = new Aggregate(”Window␣Title”);
5
6      while(game->running){
7          game->Input_Handler();
8          game->Components();
9      }
10
11     delete game;
12
13     return 0;
14 }
```

The code above boils down to the following: Open a window called ”Window Title”, while the game is running ask if the user wants to quit. If so, quit. Draw to the window. After quitting, delete the game object and release its memory.

To run this example for yourself, first copy the code into source/Main.cpp then compile and link the code. When you're done looking at a blank window, just click the x in the corner or use Alt+F4.

```
cp Tests/Open_Window.cpp source/Main.cpp
make Main Join
./Open_Window.obj
```

There isn't much to see here. You can only take away lines 7 and 8 without errors. Delete line 7 and you'll have to forcefully close the application. Delete line 8 and you're stuck with a black screen. Delete line 11 and address_sanitizer will complain that you didn't release the memory.

2.3 Loading Images

An object is any entity you want to have on the game. It has a position and a display size. You can also give it a sprite (image) and render it to the screen or it can remain invisible. This demo will show you how to initialise objects and use them to display images.

Here is the code from Tests/Loading_Sprites.cpp. We will discuss it soon. Please forgive my programmer art as I try my best to have something mildly interesting to look at on screen.

```cpp
#include "Aggregate.hpp"

class Game : public Aggregate{
    using Aggregate::Aggregate;

    Atlas* a_square;
    Atlas* a_3D;
    Vector2 texture_res = {787, 787};
    Vector2 size = {200, 200};

    Object *square;
    Object *cube;
    Object *cone;
    Object *cylinder;

    Object* Init_Object(Atlas* atlas, int row, int column);
    public:
    void SetUp();
};

Object* Game::Init_Object(Atlas* atlas, int row, int column){
    Object* obj;

    obj = AddObject();
    atlas->Assign_Sprite(obj, row, column);
    obj->Set_Size(size);

    return obj;
}

void Game::SetUp(){
    a_square = texture_manager->Load("assets/Square.png", texture_res);

    //Loading a single sprite
    square = AddObject();
    a_square->Assign_Sprite(square);
    square->Set_Pos({25, 25});
    square->Set_Size({50, 50});

    //Loading multiple sprites from atlas
    a_3D = texture_manager->Load("assets/3D.png", texture_res, 1, 3);

    cube = Init_Object(a_3D, 0, 0);
    cylinder = Init_Object(a_3D, 0, 1);
    cone = Init_Object(a_3D, 0, 2);

    cube->Set_Pos({623, 470});
    cylinder->Set_Pos({450, 170});
    cone->Set_Pos({277, 470});
}

int main(){
    Game* game = new Game("Load␣Sprites");
    game->SetUp();
55
56      while(game->running){
57          game->Input_Handler();
58          game->Components();
59      }
60
61      delete game;
62
63      return 0;
64  }
```

2.3.1 Loading a single image

Let's begin our breakdown from the bottom of the file. You'll recognise from subsection 2.2 how the main function is structured. The only notable difference is that now the game object belongs to the Game class rather than Aggregate. As you can see in line 3, Aggregate is the parent class to Game. You'll need to do this in order to create your own variables and functions that can interact with the Game class.

In fact, that's exactly what we do next. Vector2 is a type that implements some basic 2D vector arithmetic, but for now, we just use it to store 2 numbers at a time. The type Atlas is used to contain the texture we will load from a file. This texture can have one or more images, as we will see. Objects are useful for multiple reasons. For now, they display a sprite at a specific position on screen and at a certain size. All of these are mutable at runtime.

In the SetUp function, we start by loading a texture into an Atlas, a_square. It is important to give the resolution of the image we are expecting from it in addition to the path to the image. In my case, I know all my sprites are 787×787, so I defined a member Vector2 with these dimensions. If you want to load an image with a unique resolution of 300×200 pixels, you can write directly $\{300, 200\}$ into the Load function.

Next, AddObject creates an object, tells the Game class it exists and gives us a pointer to it. Assign_Sprite and Set_Pos do what they say and Set_Size scales the object up or down until it fits into the size we want to display. And that is it. That is how you load a beautiful blue square into the corner of the screen as seen in figure 1. See you in the next demo.

2.3.2 Loading an image atlas

What's that? You want to know about the other, more prominent shapes? It's not interesting, you're wasting your time here. Just more of the same. With one exception. These 3 sprites come from the same file. How, you may ask, does the engine load only a chunk of an image? Well, let's take a closer look.

The load function can take two more parameters, rows and columns. An atlas is a colection of images in the shape of a grid. If you don't specify rows and columns, it is assumed the grid has 1×1 images. In the case of assets/3D.png, we have a 1×3 grid. Again, when assigning the sprite, you can choose not to specify rows or columns like we did with the square and it will be assumed that you want the top left sprite of the file. To load any other sprite, keep in mind that the indexes start at 0. That is, the top left sprite is at position (0, 0) and there is no sprite in position (0, 3) for this 1×3 atlas.

Last point. It is slightly overkill for only 4 objects, but I wanted to show you how you can make more than the minimum necessary functions to save you trouble. For example, you may want to initialise multiple objects at the same time. Always the same few lines: AddObject, Assign_Sprite, Set_Size, etc. Go ahead, leave it to a function instead of bloating SetUp with repetition. This is how you do it. However, there are a few warnings you should keep in mind:

1. If you want to create an object in one function and use it anywhere else without passing it as a parameter, don't use the global scope. That will make you mad if you add more files. Turn it into a member variable in the Game class.

2. It is basic C++, but it got me hard. if you declare a pointer like in line 6, it points nowhere. Don't pass an empty pointer as a parameter or you'll lose the object and get extremely confused. If you see yourself writing "Init_Object(Object* obj)", go change it. Always call AddObject before you pass as a parameter.

3. Finally, if you want to make a function like Init_Object that uses AddObject or any of the other functions declared in include/Aggregate.hpp, you'll need a member function in the Game class. Otherwise, C++ doesn't know to which game the object belongs. Yes, I know you're only making one game, but C++ doesn't know what you're doing. Member functions will take care of it, trust me.

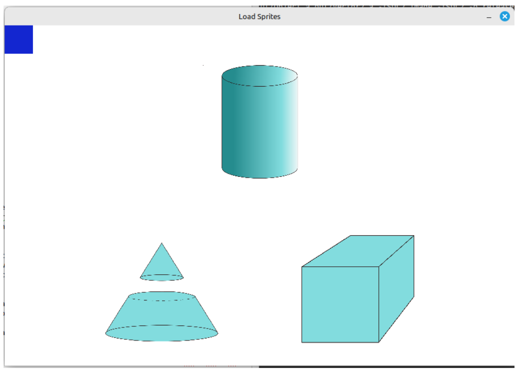

Figure 1: Lovely blue square in the corner.

2.4 Movement and the Update function

I mentioned in subsection 2.3 how the size, and position of an object can change during runtime. You know what this means? We can move stuff around, Igor! For this, we will need the Update function. I'll explain how it works using the code in Tests/Objects_Swirl.cpp.

```
void Game::SetUp(){
    a_circle = texture_manager->Load("assets/Circle.png", texture_res);
    a_square = texture_manager->Load("assets/Square.png", texture_res);

    circle1 = Init_Object(a_circle);
    circle2 = Init_Object(a_circle);
    circle3 = Init_Object(a_circle);
    square1 = Init_Object(a_square);
    square2 = Init_Object(a_square);
    square3 = Init_Object(a_square);

    Set_Framerate(40);
}

Vector2 Swirl(float degrees){
    float angle = degrees*M_PI/180;

    Vector2 pos;

    pos.x = 450-400*cos(angle);
    pos.y = 300-100*sin(2*angle);

    return pos;
}

void Game::Update(){
    const static float frequency = 0.12;

    circle1->Set_Pos( Swirl(frequency*runtime +   0) );
    circle2->Set_Pos( Swirl(frequency*runtime +  24) );
    square1->Set_Pos( Swirl(frequency*runtime +  48) );
    square2->Set_Pos( Swirl(frequency*runtime +  72) );
    square3->Set_Pos( Swirl(frequency*runtime +  96) );
    circle3->Set_Pos( Swirl(frequency*runtime + 120) );
}

int main(){
    Game* game = new Game("Infinity Swirl");
    game->SetUp();

    while(game->running){
        game->Input_Handler();
        game->Timing();
        game->Update();
        game->Components();
    }

    delete game;

    return 0;
}
```

I don't think you'll need me to copy all of the code here any more. Especially since the demos will be getting more lengthy and complex from here on. So here are the new additions.

The SetUp function works much like it did in subsection 2.3. You load the texture and initialise the objects. This time, I only loaded one sprite per file, so I simplified the Init_Object function.

The importYou'll notice that we have 6 objects and only 2 load calls. Yes, it is possible (and highly encouraged) to load a texture only once and use it as many times as necessary.

Another change is the there are 2 new functions called in main. The first one is Game::Timing. It allows you to control the framerate. In the earlier demos, framerate did not matter so the window rendered as often as it could. Now we set it to 40 frames per second towards the end of Game::SetUp. Feel free to experiment with different framerates, this demo was programmed in such a way, everything should move at the same speed at any framerate.

Once you quit the game, you may have noticed a message on the console saying something along the lines of:

```
Game ended after 8669 ms.
Relaxed for 8327 ms (96%).
```

This tells you the total runtime of your session followed by the fraction of that time spent waiting. The lower this fraction, the more demanding your game is on your hardware and the more likely you are to drop frames.

The second function added to main is Game::Update. Like SetUp, you have to declare this function yourself in the Game class. Unlike SetUp, however; this function is meant to run every single frame which helps us make changes over time. In this case, we use it to assign the position of every object every frame.

Don't worry too much about the Swirl function. It's a little analytical geometry meant to create a fun animation. What's important here is that it takes a float and returns a Vector2 we can use to change the positions of the objects. Now look a little closer at which float I chose, especially runtime.

Runtime is a variable in the aggregate class that keeps track of how long your game has been up. It is frame independent so my code calculates where an object should be at a specific frame not based on the amount of frames so far, but on the runtime. That's how I know the frame rate doesn't impact the animation speed.

Here are some variables provided by the Aggregate class that might help you control the flow of your game. Remember, none of these work unless you call Game::Timing in main.

Variable Name	Description
runtime	Real time in miliseconds since opening SDL.
delta_time	Real time in miliseconds since the last frame
frame_number	Number of frames calculated so far.