2D Platformer game engine

By: Niels Justesen (noju)

Introduction

The 2D platformer game engine of this project is written in C++ using the SDL library. My goal was to make a multi-directional platformer where the player can explore the world in all directions instead of only walking in horizontal (and most often right) direction all the time. Additionally, it should be noted that I am entirely new to the C++ programming language, so I am learning this as well.

Architecture

The main.cpp file of the project contains functions that can load images and blitting images to the window surface. These functions were highly influenced by the SDL tutorial on LazyFoo.net[1].

The project contains three controllers that each have a specific purpose. The level controller loads the level, the input controller registers which keys are down and the physics controller takes care of movement and collisions.

Besides the image related functions the main.cpp file also contains the game loop. The sequence of the game loop is shown on the simplified sequence diagram on Figure 1. The game loop starts by pulling all the key events that have occurred and foreach event it tells the input controller to register whether the given key is down or up.

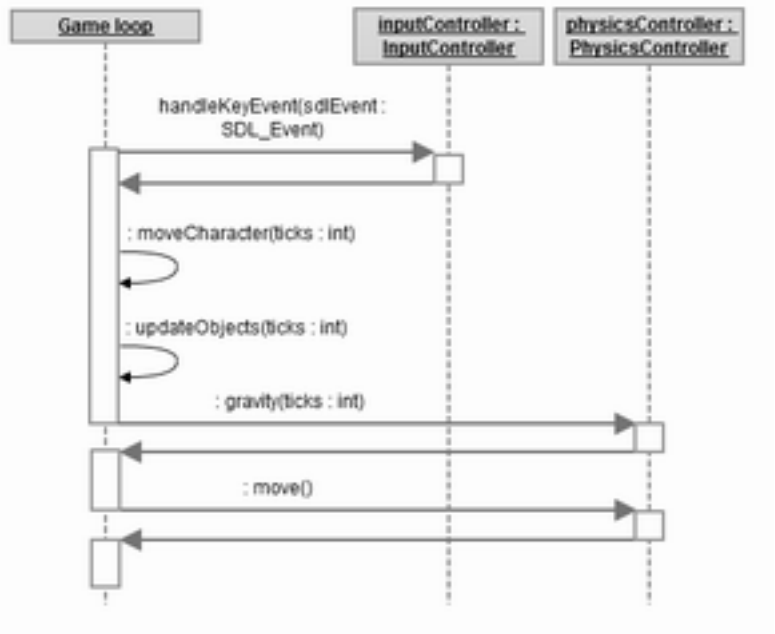

Figure 1: Simplified sequence diagram of the game loop.

[1] http://lazyfoo.net/SDL_tutorials/lesson01/windows/msvsnet2010e/index.php

After the key presses are registered the movement in both axes of the character are updated based on which keys are down, and all game objects in the game will act independently based on their type. Enemies may shoot and fireballs may disappear. Now, the game objects have acted and it is time for the physics controller to calculate the results. It starts by adding a gravitational acceleration to all objects that are not situated on the ground, whereafter it moves all game objects based on their movement. It is also the physics controllers job to detect collisions and handle any outcomes. After all these steps the coordinates of the game objects have been updated and the game objects will be drawn on the screen.

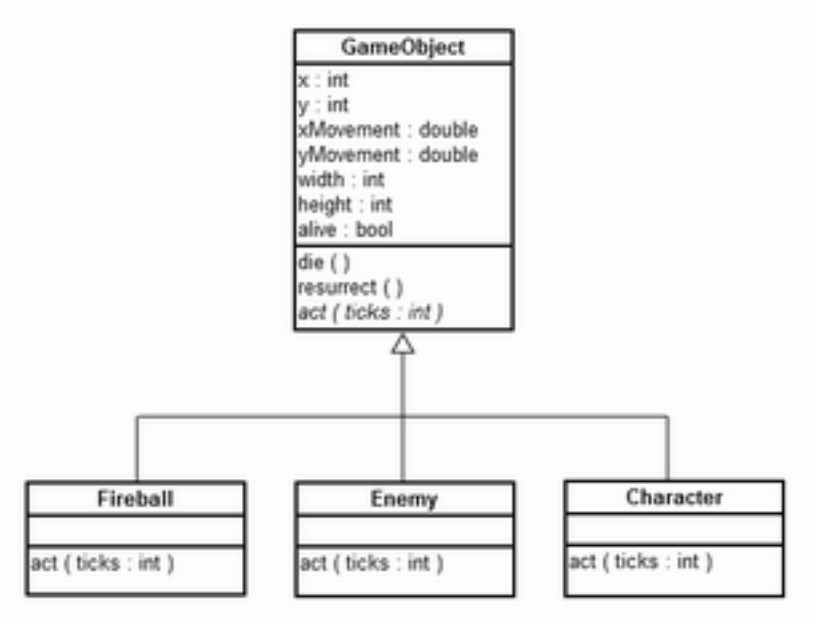

Figure 2: Game object inheritance. Only showing some methods and fields.

The game objects all share the same base class, namely GameObject, which has a number of fields and methods, most importantly the abstract method act(ticks : int).
The physics controller handle all game objects (almost) equally which means that all game objects are affected by gravity and collision detection.

Levels

The levels are tile based and are loaded from level files into the game by the level controller[2]. The level files contain a lot of characters where each character maps to a specific type of tile. E.g. ‘0’ is air, ‘1’ is solid and ‘g’ is grass. ‘c’ denotes the spawn point of the character, ‘R’ denotes enemies shooting right and ‘L’ denotes enemies shooting left. The level controller reads the file when the game is initialized and inserts the values into a two-dimensional array which makes it easy to look up tiles in the level.

[2] The level controller on the cd does not load from a file but instead reads the level from a variable. This was due to some problems when I had to build the project in release mode.

Figure 3: Map tiles

Collision detection

Bounding boxes

I started out by manually assigning bounding boxes to all game objects. Each type of game object had their own list of points that were used for collision testing. I changed this to another approach which I found much more pleasant when adding and changing the game objects in the game. I still kept the idea of a bounding boxes that were axis aligned since none of the game objects are supposed to rotate, but now the physics controller calculates the points used for collision detection in run time. It does this by using the width and height properties of the game objects.

Level collision

Game objects can collide with some of the tiles in the level. To see whether a point is inside a solid tile the coordinates of the point is divided by the width/height of a tile. The result is then used as indices to look up in the two-dimensional array. In this way, all the points of the bounding box is tested to see if it is inside a solid tile. If so, the position of the game object is corrected to align with the tile.

Stairs and slopes

Stairs works as solid tiles except they need an extra check for collision and another way of correcting the position of the game object. If a point of a bounding box is inside a stairs tile the coordinates are translated into coordinates relative to the tile. In this way, the relative coordinates are checked to see whether they are in the stairs area or in the air area. My ultimate goal was to make the player able to move the character up and down the stairs by pressing the left or right buttons. But that turned out to be quite difficult. If I had more time I would also have made slopes which would work in much the same way as stairs, but the character should slowly slide down when standing on them.

Bouncing objects

It was actually quite easy to make objects bounce on floors, walls and ceilings. When a collision is detected with a solid tile the movement of the colliding object is simply multiplied by -1. To make it more realistic I have added a bounce property to all game objects that represents how much of the force that are lost due to the bounce. Here is a code snippet that shows how:

```
object->setYMovement(-moveY*object->getBounceEffect());
```