Project Report - First assignment - GAME ENGINES

Andrea Distler

28 November, 2012

Contents

1 Introduction

This is the project report for the first assignment of the Game Engines course.

A short description of the source code and the used libraries can be found in section 2. The files are available on GitHub. The libraries are also made available on GitHub for easier installation.

Section 3 describes the outcome of the project.

Section4 gives an overview of the used mechanics and points out the key features of the project.

Section 5 gives a detailed description of the important classes and their relations to each other. It also points out challenging parts of the code.

I have invested an unnecessarily big amount of time into this small assignment, and I want to elaborate on my workflow in section 6.

2 The Files

All files can be found on GitHub.
`https://github.com/JungleJinn/GE---SimplePlatformerEngine`.

2.1 Used Libraries

The project was written in C++, using additional libraries:

- Included in the repository
  - SDL image
- External
  - openGL. Installed manually. Files in zip
  - glm. Installed manually. Files in zip
  - SDL 1.2.15. `http://www.libsdl.org/download-1.2.php`
    Used environment variable named SDK_SDL

3 Outcome

The project's result is a little game in which the player can control a bomb. The player cannot directly control the bomb's position. However, it is possible to change the bomb's mass and therefore indirectly possible to steer the bomb. The goal of the game is to survive as long as possible, which is made hard by other bombs which present themselves as obstacles. Depending on the player's mass, the enemy bomb will affect the player's further movement. If the player moves backwards, the game is over. For a screenshot of the game see figure 1.

There is a short explanation of the controls and rules of the game in the beginning. Once a player loses, there is also a gameover screen, telling the player that the game is now over.

The assignment was to create a simple physics engine which can be used for a platformer. The game features:

- 2D physics
- openGL rendering
- an intro screen and an outro screen
- a dragon with a hat
- sprites and texture atlases
- collision detection and response
- bounciness
- very intuitive controls

4 Overview

This section gives an overview over used mechanics.

4.1 Game Loop

Listing 1: The game loop

```
1 lastUpdateTime = clock();
2 Init();
3
4 quit = false;
5 while (!quit)
6 {
7   Play();
8 }
```

First the game is being set up, then the game loop starts. The game loop is quite basic. While the game has not been quit, it is playing.

4.2 Messaging

The Input is realized by using SDL events. The first thing done in the Play method is to fetch all pending input events. These events are then sorted by what is important, what is to be used in the game, and what not. Then the newly written message system starts to create and forward messages.

The messenger can send messages of different types to receivers of type IMessageReceiver. All objects of this type automatically register and unregister at the messenger at creation and deletion. The receivers decide which kinds of messages they want to process, but each receiver gets all sent messages.

Listing 2: Messaging

```
1 SDL_Event sdlEvent = SDL_Event();
2 while (SDL_PollEvent(&sdlEvent))
3 {
4   // user closes window
5   if (sdlEvent.type == SDL_QUIT)
6   {
7     quit = true;
8     break;
```

```
9    }
10   else if (sdlEvent.type == SDL_KEYDOWN)
11   {
12     messenger->SendMessage(InputMessage(InputMessage::KEY_DOWN
         , sdlEvent.key.keysym.sym));
13   }
14   else if (sdlEvent.type == SDL_KEYUP)
15   {
16     messenger->SendMessage(InputMessage(InputMessage::KEY_UP,
         sdlEvent.key.keysym.sym));
17   }
18 }
```

4.3 Physics

The physics are calculated in two dimensions. The physics system is responsible for movement and for collisions as well as for collision response. Bodies are implemented as simple rigid bodies. Forces can be applied. There are two basic collision shapes: Circles and Rectangles. The collision detection happens in two states, first the broad phase, which only checks the radius of the objects, and the narrow phase, in which the collision normal is calculated. The normal is then used to determine the resulting forces acting on both colliders. Bodies have a coefficient of restitution which adds to the collision response. The combination method is to use the average coefficient of restitution of both bodies.

For integrating forces, the integration method receives a value named deltaTime. The delta time is calculated like in the following listing:

Listing 3: deltaTime

```
1 int now = clock();
2 deltaTime = (float)((now - lastUpdateTime) / CLOCKS_PER_SEC) /
      timeScale;
3 lastUpdateTime = now;
```

The whole game execution can be slowed or fastened by setting different timeScale values. At a value of 1.0 it is executed in real-time, and other values stretch or squish the perceived time.

4.4 Sprites

Sprites are used for the player and for the enemy bombs. They are stored in a spritesheet and their texture coordinates are adapted to fit certain sprite frames.

4.5 Camera

The camera is moving with the player all the time. A problem encountered with a potentially endless level was whether the floor and ceiling should always be recreated. The player can move as far as he wants to, therefore an inifinite amount of rectangular walls would be needed. The solution which avoids creating endless amounts of floor tiles was to always move the floor and ceiling with the player (not unlike the camera). The texture coordinates change to create the visual feedback of the player's velocity. The illusion that the player moves in relation to the floor is created, whereas the player and the floor have a relative velocity of zero. The background also follows the player.

4.6 Dragon

In the first phase of the game the player can choose the launch angle by rotating a dragon's head.

5 Important Classes

In figure 2 a selection of classes is depicted. The most interesting parts are described in this chapter.

5.1 Game

The game class manages the game states and the game objects. It renders the objects and updates them automatically. The game class is intended to serve as a base class for other games, although it was used as both the abstract class and the implementation in this project.

Initialization. First of all, SDL is set up. After that, openGL is prepared for 2D drawing. The projection mode used is orthogonal. After the graphics setup, the frame for the game logic is initialized. This includes the messenger (5.2), the log, and the physics (5.3). Subsequently the game objects are set up. `GameObjects` need a reference to the game and the messenger, which they use for automatically registering. `GameObjects` in this game are the playerBomb, floor, ceiling, dragon, and the background. The enemy bombs are spawned at run-time.

Play. First, all relevant events from SDL are converted to `Messages` and broadcast by the messenger. Then `deltaTime` is calculated. The game, physics and gameObjects are updated after. The render method is called afterwards. When the game objects are updated, the game loops through the list containing them. If a game object has to be deleted it cannot be done in this loop, so the object is moved to a deletion list. Deleting gameObjects is the last thing to do in the `Play` method.

Listing 4: The play method.

```
1  void Game::Play()
2  {
3    // events
4    SDL_Event sdlEvent = SDL_Event();
5    while (SDL_PollEvent(&sdlEvent))
6    {
7      // user closes window
8      if (sdlEvent.type == SDL_QUIT)
9      {
10       quit = true;
11       break;
12     }
13     else if (sdlEvent.type == SDL_KEYDOWN)
14     {
15       messenger->SendMessage(InputMessage(InputMessage::
             KEY_DOWN, sdlEvent.key.keysym.sym));
16     }
17     else if (sdlEvent.type == SDL_KEYUP)
18     {
19       messenger->SendMessage(InputMessage(InputMessage::KEY_UP
             , sdlEvent.key.keysym.sym));
20     }
21   }
22
```

```
23    int now = clock();
24    deltaTime = (float)((now - lastUpdateTime) / CLOCKS_PER_SEC)
          / timeScale;
25    lastUpdateTime = now;
26
27    // update
28    Update(deltaTime);
29    // physics update
30    physics->Update(deltaTime);
31    // call update for each game object
32    UpdateGameObjects(deltaTime);
33
34    // render the game
35    Render();
36
37    // remove game objects which have been marked for deletion
38    for (list<GameObject*>::iterator it = gameObjectsToDelete.
          begin();
39      it != gameObjectsToDelete.end();
40      it++)
41    {
42      gameObjects.remove(*it);
43    }
44
45    gameObjectsToDelete.clear();
46  }
```

Render. The render method uses standard double-buffered openGL rendering. First the buffers are cleared, matrices are pushed and popped and vertices are sent to the hardware. In the end the buffers are swapped. The render method is virtual, because the render order of objects determines their visibility. Each game implementation has different orders and objects.

Destructor. In the destructor, all managed objects are deleted. First the `GameObjects` and then the messenger and physics. Last `SDL_Quit()` is called.

Object registration and unregistration. There are two `Add` and `Remove` methods in the game. One pair is used to relay registration to the physics class. The other pair is used to register `GameObjects` for updating.

5.2 Messenger

The messaging system has three main components: the `Messenger`, the `IMessageReceiver`, and the `Message`. The messenger is used for broadcasting messages and for registering/unregistering receivers. All receivers get all messages. They decide for themselves whether they want to receive certain types of messages. The `IMessageReceiver` registers in the constructor and unregisters in the destructor. When sending a message, the messenger tells the message to deliver itself to all receivers. This is done because then the receiver can check the message's type.

5.3 Physics

The physics class holds a list of physicsObjects, updates them and relays detected collisions via the messenger.

Update. The update method contains a nested loop for collision checking. However, it does not check collisions twice, which would happen if it would just go through all the objects in the inner and the outer loop. The determination of which objects should be checked works like this:

All collisions can be represented as a matrix of $n*n$ elements. The matrix can be mirrored at its diagonal, meaning that all collisions below the diagonal are too much and don't need to be calculated. An x marks the test for a collision between the two objects.

	$o1$	$o2$	$o3$	$o4$	...
$o1$		x	x	x	...
$o2$			x	x	...
$o3$				x	...
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	

6 Workflow

I started working on the project quite early, because I wanted to try new things. So I tried and tried and jumped from one interesting thing to the next. I tried many graphics frameworks (openFrameworks, SDL, openGL,

freeImage) I wanted to further my openGL skills, so I set up openGL (as well as the other frameworks). I decided to use openGL. So I created the drawing context and used the openGL display/event callbacks. (Which was a mistake, since the assignment was to create the loop and event system ourselves, but I only realized that later.) I got that running, so I created the physics logic (which I mostly reused in the final project). After that, however, things became chaotic. There were many loose ends which would have to be tied to the rest of the project. I would not have enough time to do that AND finish the other projects. Two weeks ago I decided to let it rest in peace. (It rests on GitHub: `https://github.com/JungleJinn/GE---Physics-Engine`)

I took the good stuff and used it for a new, clean, and simple project, which is the one described in this report. I had a better vision of what I wanted to do from the beginning. It worked better. Even if it is only a small and simple thing, I hope the work it took to get to this point is visible.

Listings

List of Figures

Figure 1: A screenshot of the game.

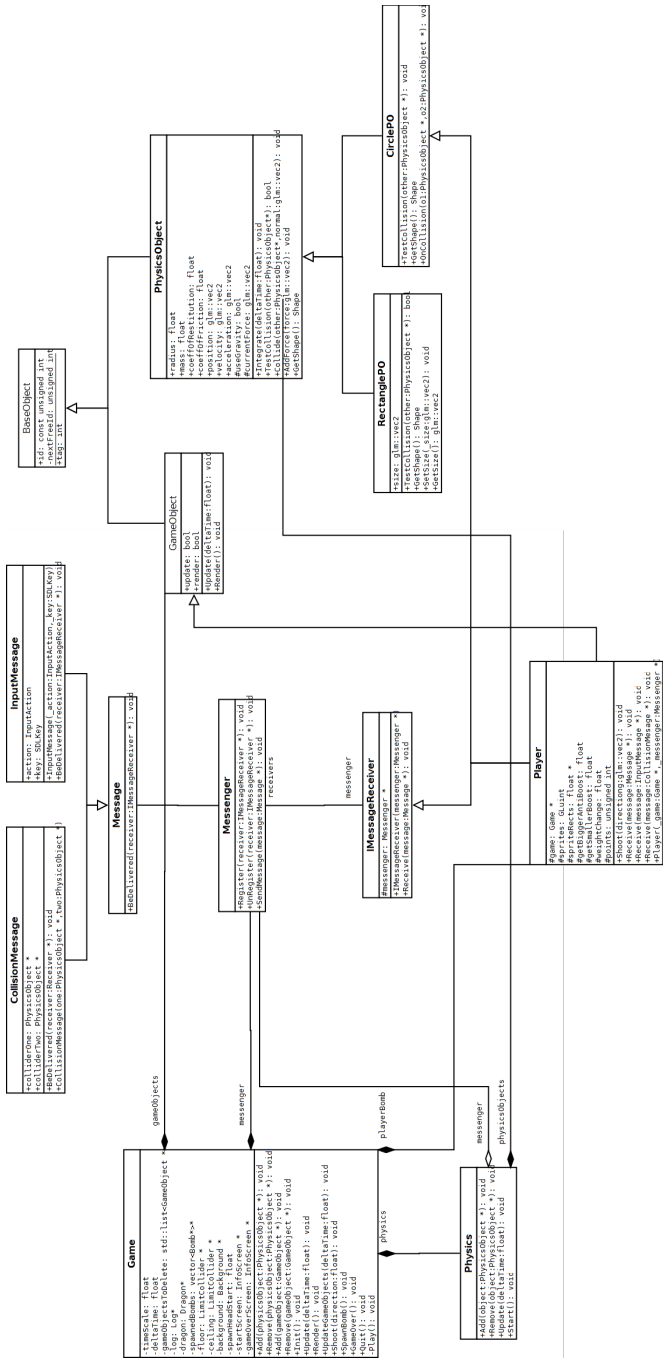

Figure 2: A sketch of the most important classes.