Graphics and Game Technology

Assignment 6

A game with Box2D

Almost any computer game nowadays uses some kind of graphics, ranging from a basic 2D engine to a state-of-the-art 3D engine. In this assignment you will write your own 2D game. Similar to the game "Crayon Physics"[1] the player is presented with a puzzle, consisting of a number of shapes, a ball and a finish. The objective for the player is to get the ball to the finish. The player cannot interact with the ball directly, however, the player must solve these puzzles by drawing shapes into the world that get the ball to the finish using physics.

1 Physics: Box2D

Implementing a complete physics engine is well beyond the scope of this assignment, therefore you will use the Box2D Physics Engine.

The Box2D API is quite big and complex, and offers more than you will use for this assignment. We will go through some of the basic concepts of Box2D here, but we recommend you take a look at the manual at `http://www.box2d.org/manual.html`[2]. This explains most concepts, including common errors and code examples. You may also want to take a look at the API documentation[3] which lists all classes with their methods and arguments. Note that Box2D is implemented in C++, but don't let that put you off.

Box2D can simulate the physics, including collisions, for static and dynamic objects. In Box2D you simulate a world (`b2World`) with objects in it. Every object consists of:

- a "Shape": a 2D geometric shape, such as a circle or polygon.
- a "Body": a rigid object, i.e. an object that cannot be deformed by physics. It can only move and rotate, it is either static or dynamic and it has a position in the world.

[1] `http://www.crayonphysics.com/`

[2] Note that there are some differences between the documentation and the version that you will be using. For example, the constructor for the `b2World` objects has changed to only take a gravity vector.

[3] Online version: `http://www.few.vu.nl/~kkg370/Box2D/html/annotated.html`

- a "Fixture": this links a shape to a body. It also adds properties such as density and friction.

Box2D uses meters, kilograms and seconds as units, and is built for simulation of real-world scales. This means objects should be somewhere between 0.1 and 10 meters to get the best performance. For more details about how Box2D works, see the manual.

1.1 The framework

You received a framework that creates a window and reads a number of game levels from files. These levels are available in `main.cpp` in the global array `levels`. Every level has a starting position where the ball will spawn, and a finish position where the player must get the ball to. Every level also has a number of static bodies. For more details on how these can be accessed, see the data structures in `level.h`. As mentioned, Box2D uses meters as units. The framework sets up OpenGL in such a way that every unit in OpenGL corresponds to a unit in Box2D. Every level (and also your screen) measures 8 by 6 meters.

The framework uses GLUT, which creates a window and an OpenGL context for you. In `main.cpp` you'll find the functions `key_pressed`, `mouse_clicked`, `mouse_moved` and `draw`. The first three, as the name suggests, are called when mouse and keyboard events take place. The `draw` function will be called every 'tick'. Here you must perform both the game logic (for example, simulate the physics and check if the player has won) as well as the drawing of the game. There is already some logic in there showing the framerate (in frames-per-second, or fps) of your game and handling the swapping of buffers on which you draw. This framework is merely provided as a starting point for your game, so don't be afraid to make changes!

Warning: To run the framework on the **lab machines**, you can't simply use `./main` since this will use an incorrect Box2D library. The fix is to force your program to look in `/usr/local/lib` first when it loads shared libraries. There are two ways of doing this: use `make run` every time you want to start the program, or add the following line to your `.bashrc` file in your home directory:

```
export LD_LIBRARY_PATH=/usr/local/lib
```

If you choose for the latter option, you can simply use `./main` afterwards (after reopening your terminal). If you do not do this, you will most likely see an error related to `GLIBC_2.14`.

1.2 A falling ball

Your first step should be to start filling in the function `load_world` in `main.cpp`. Start by creating a Box2D world with gravity and a single ball. This ball is, like any Box2D object, constructed with a shape, body and fixture. To see if this actually works, add a call to the `Step` method of your `b2World` object for every

tick in the `draw` function. Then, you can print the coordinates of the ball every tick and you should see the coordinates change (the y-coordinate decreases).

Hint: If you are not sure about what classes, functions or attributes to use, take a look at the manual and API documentation.

Printing the ball's position will tell you if the ball is moving in the right direction, but you probably want to actually show the ball to the player. Replace the print with OpenGL calls to draw a circle. For example, think of an algorithm to draw (an approximation of) a circle using a `GL_TRIANGLE_FAN`.

1.3 Level objects

Now it is time to populate the Box2D world with the static objects for the current level. The `levels` data structure contains a list of objects, each with a position, list of vertices and whether they are dynamic or static. Implement the code that creates these polygons as objects in your world in the `load_world` function using a `b2PolygonShape`. The first level contains a single object: a slope from which the ball will roll down. Although you currently cannot see the objects, you should see the interactions your ball has with them (as shown in Figure 1).

To draw all the objects you could simply use the `levels` structure, but a better approach is to iterate over all the bodies in the Box2D world, since this allows you to draw the entire world at once, with the current properties for every object. For exactly this purpose, your world object has a `GetBodyList` method, and every body object has a `GetFixtureList` method. Every fixture has a shape, which can be a `b2CircleShape` or `b2PolygonShape`. These shape objects should provide you with enough information to implement the code to draw every object in the world.

2 Game mechanics

Your program is now a pretty decent physics simulator, but it's not really a game yet: it lacks player interaction and an objective.

2.1 Reaching the objective

Every level has a finish: a location that the ball must reach in order to complete the level. Implement the code that detects if the ball has reached this location, and continue to the next level if this is the case.

2.2 Player interaction: creating new objects

For the user interaction part, the player should be able to create new, dynamic, bodies by drawing them with the mouse. Drawing these bodies free-form (e.g. drawing the outline of the shape using a mouse) is rather hard, since you need to extract vertices from a lot of mouse data. Another problem is that a

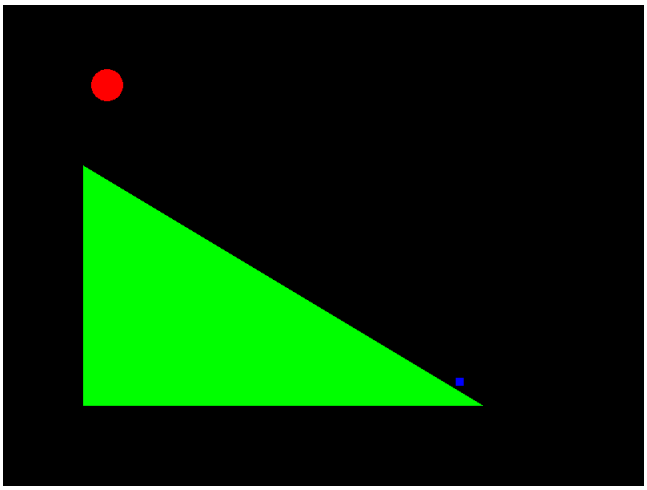

Figure 1: Level 1 as provided by the framework. Note that the blue rectangle is the level's finish.

shape can have a maximum of 8 vertices, so you need to split up the polygon in, for example, triangles.

Therefore, for this game, it is only required that you implement the drawing of polygons with a limited number of vertices (e.g. 4). You can implement this by recording each click as a vertex. Remember that the mouse event coordinates are in pixels from the top-left and your physics world is in meters from the bottom-left!

You may need to modify these vertices because of some Box2D limitations:

- Box2D does not support concave polygons.
- Box2D has a limit of 8 vertices per polygon by default.
- Box2D does not support self-intersecting polygons (when two sides intersect in some way).
- Box2D does not support polygons which are too small.
- Box2D only supports a counter-clockwise winding of the vertices.

Especially for the last item you must implement code to detect and fix this. Design an algorithm that calculates the area of a polygon. If that returns a negative number, you must reverse the list of vertices.

Once you spawn a dynamic body, the drawing code written in section 1.3 should immediately draw your dynamic bodies too. You may, however, want to alter their appearance for more clarity.

3 Joints

In games, there is often a relationship between some objects. Box2D supports this too, which it calls joints. These describe that two bodies are some way connected, depending on the type of the joint. While Box2D has support for many joints, we will only look at two of them: the revolute and pulley joints. The revolute joint means that two objects are attached to each other in some point, but can still rotate freely around that point. The pulley joint is more complex: both objects are attached to some point in the world (the *ground*), and the distance between each object and its ground must remain constant when added together. In other words, when one object moves up, the other moves down.

The bridge and pulley levels (the last two levels in the framework) use these two joints. These joints can be accessed through the same data structures as the objects themselves. The objects which are joint together are described through their index value in the level. You should therefore save these bodies created during the initialization of the level so the joints can be attached to them. The initialization of the joints should be implemented in the `load_world` function, and works similarly to the creation of fixtures.

4 Extending your game

You should now have a fully functional game, in which you can complete all the given example levels. Improve upon this in any way you like, preferably with techniques you learned during the lectures and previous assignments:

- add textures to your game[4],
- add support for free-form bodies,
- add a timer to your game that records the time to solution,
- add a high-score list,
- ...

You can also look at improving the user experience, or extending the game mechanics, or introduce something entirely new. Box2D has a lot of possibilities, and supports a wide range of features such a number of joints which allows you to create wheels, ropes, etc.

5 Grading

Correctly setting up Box2D with a falling ball, including the drawing of a ball gives you a maximum of 2 points.

[4]For loading images into your program as OpenGL textures, you can use a library such as SOIL (http://www.lonesock.net/soil.html) which can load arbitrary png and jpg files.

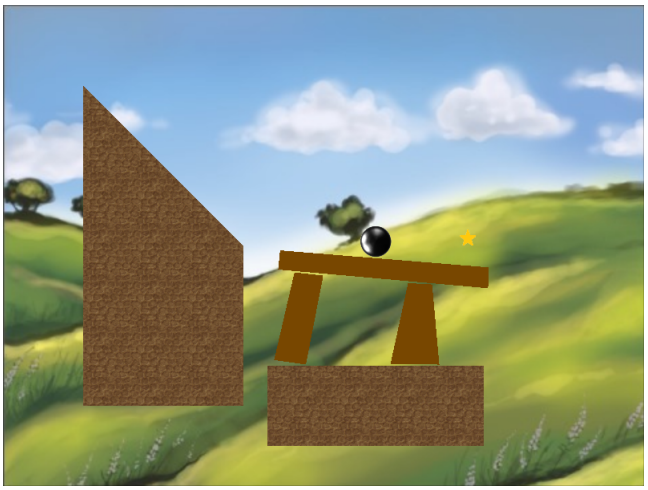

Figure 2: An example of level 3 with some basic textures.

Correctly loading all levels into Box2D and drawing the levels gives you a maximum of 3 points.

A correct implementation of the user interaction part; the drawing of polygons, detecting that the ball has reached the finish and continuing on to the next level gets you a maximum of 2 points.

Correctly creating the joints in Box2D gives you a maximum of 2 point.

You will get a maximum of 1 point for adding the extras.

Points may be deducted if your code is not well outlined or commented!