CS256 — Homework 5

February 22, 2016

Due: Friday, March 11, 2016 before midnight (160 points)

Like homework 4, you may work with a partner for this assignment. Make sure to place comments at the top of the source code for each file listing both names and add the second person as a member on your project on `codebank.xyz`.

Preparing the Project

1. Go to https://codebank.xyz and you should see a project in the CS256 group named `CS256-HW5`. Fork this project for your own user.
2. Next, clone the repository so you have a local copy:

   ```
   $ git clone https://codebank.xyz/username/CS256-HW5.git
   ```

3. The repository should have several files: `main.cpp`, `Game.h`, `Game.cpp`, `particle.bmp`, and `Constants.h`.

Description

In this assignment we will simulate gravity and elastic collisions on solid objects and draw them using the Simple DirectMedia Layer 2 (SDL2) library. I will provide most of the code that interacts with SDL2 itself so you should focus on implementing the physics.

In order to use SDL2, you must have the library installed. You can obtain it from https://www.libsdl.org/.

Note: in some of the provided code I have used C++11 features. If you do not have access to C++11, please contact me and I will show you what needs to be modified. If you have been compiling on Cal Poly's servers, you will need a way of compiling locally. I can provide a virtual machine image that you can use to compile and test the program if you need it.

Simple Game Engine

We will be implementing a simple game engine but for this assignment it is not necessary to handle any user input. Extra credit will be given to students that add more than is required for this project such as handling user input and adding game features like keeping player score, etc.

In general, game engines follow a structure that we can describe in pseudo-code as:

```
initialize();
while (running)
{
    handleEvents();
```

```
    update();
    render();
}
cleanup();
```

Each function will be described below:

1. The `initialize()` function will do any setup required by the game. For example, in our assignment one step here is to randomly generate particles for the game. We also need to initialize the SDL2 library.
2. The `running` condition should stay `true` while the game is still being played. When the game should exit for whatever reason switch this condition to allow for proper cleanup.
3. The `handleEvents()` function will handle any user input events such as moving a character or maybe even network events for a multiplayer game.
4. The `update()` function has all of the game logic updates. For example, all physics calculations for our assignment should be handled here.
5. The `render()` function renders the game objects to the screen. For example, in our assignment this will render the list of particles.
6. The `cleanup()` function handles cleaning up any resources allocated for the game. In our case, this is used to cleanup the SDL2 library.

I have provided the code for the `initialize()`, `render()`, and `cleanup()` functions because they deal mostly with the SDL2 library. Our `handleEvents()` function will be mostly empty, but I have provided handling for closing the game window. This "game loop" already exists in the `operator()` function in the `Game` class.

Your main task is to implement the `update()` function, which will handle the physics calculations between particles for a single fraem. You are passed the amount of time in seconds since the last frame as a parameter to the `update` function. You must also implement a `Particle` class and `Point` class.

The `Point` class

The `Point` class should be a class for a two dimensional point and have two `double` instance variables named `x` and `y`. We can use this class for both the position and velocity (in which case, we're really using this as a vector, but we'll still call it `Point` so we don't get it confused with `std::vector`). Include getters for `x` and `y`.

The `Particle` class

Your `Particle` class must have a two dimensional position and velocity (x and y dimensions), radius, and mass. The position and velocity should be stored using `Point` objects and the mass and radius should be `double`s. Include getters for all four instance variables.

The `Game` class

Most of this class has already been provided. You must implement the `update` function to handle updating `Particle` positions and velocities based on Newton's law of universal gravitation. When two particles collide, you should use the physics of elastic collisions to handle the collision.

For extra credit, you may also include event handling or any other additional features. Credit will be awarded based on creativity and functionality.

Newton's law of universal gravitation

Newton's second law of motion for constant-mass systems says that the net force (F) experienced by an object of mass m and acceleration a is:

$$F = ma$$

Newton's law of universal gravitation says that the force due to gravity (F) between two objects is calculated as:

$$F = G\frac{m_1 m_2}{r^2}$$

where:

- G is the universal gravitational constant with value 6.674×10^{-11}
- m_1 is the mass of the first object
- m_2 is the mass of the second object
- r is the distance between the centers of the masses

If we want to determine how much one object (m_1) is accelerated due to gravity with another object (m_2), we can apply both laws:

$$\begin{aligned} F_1 &= G\frac{m_1 m_2}{r^2} && (1) \\ m_1 a &= G\frac{m_1 m_2}{r^2} && (2) \\ a &= G\frac{m_2}{r^2} && (3) \end{aligned}$$

However, we may have many different objects all applying gravitational force to each other. To simulate this, we will calculate the effect of gravity in discrete chunks (once per frame). We will do this with the following process:

1. Iterate over our list of objects:
    - For each *other* object in the list, calculate the acceleration due to gravity between our first object and this other object
    - Accelerate the first object by this amount (change its velocity based on the acceleration)
2. Iterate over our list of objects and shift each object's position based on its velocity (which has the accumulated acceleration changes from gravitational force with all other objects)

By doing it this way, we only change the position of each object after we have accounted for the effect of gravitation from every other object. If we moved an object immediately after calculating the effect of gravitation from one other object, it would change the result of the later calculations (because gravitational force is affected by the distance between the objects).

Make sure that when you apply velocity or position changes to an object, you account for the fact that each frame is not exactly one second after the previous frame. To handle this, I have included calculating the number of milliseconds between each frame and provided this value in seconds as a `double` as an argument to the `update` function.

Hitting the edges

After you move a particle based on its velocity, we also need to check if our object has gone past one of the window borders (left, right, top, or bottom). In any of these cases, we should “bounce” the object off the border so that it stays within our window.

Elastic collisions

Another task we should handle after all of the gravity calculations is object collision. Otherwise, our objects would be able to fly through each other. We will do this using elastic collisions. An elastic collision is a collision where the total kinetic energy before and after the collision does not change.

To handle collisions, we must first determine if two objects are colliding. Once this has been determined, we need to calculate the new velocity x and y values based on the collision. The following are the calculations for the first object. To calculate the values for the second object, replace all 1 subscripts with 2 and vice versa. Note: first calculate the new velocities for each object before making any changes or else your second object will not receive the correct calculations.

$$v_{1x} = \frac{v_1 \cos(\theta_1 - \phi)(m_1 - m_2) + 2m_2 v_2 \cos(\theta_2 - \phi)}{m_1 + m_2} \cos\phi + v_1 \sin(\theta_1 - \phi) \cos\left(\phi + \frac{\pi}{2}\right)$$

$$v_{1y} = \frac{v_1 \cos(\theta_1 - \phi)(m_1 - m_2) + 2m_2 v_2 \cos(\theta_2 - \phi)}{m_1 + m_2} \sin\phi + v_1 \sin(\theta_1 - \phi) \sin\left(\phi + \frac{\pi}{2}\right)$$

where:

- v_1 and v_2 are the magnitudes of the velocities of each object *before* the collision
- m_1 and m_2 are the masses of each object
- θ_1 and θ_2 are the angles of movement of each object
- ϕ is the angle of contact for the collision

Once you implement the elastic collisions, I would recommend placing the collision handling between the new velocity calculations due to gravity and the position changes, so we can update object positions once after all other physics calculations.

Compiling

We can generally use the following command to compile the program on Linux as long as we have installed SDL2:

```
clang++ -std=c++11 -o Gravity main.cpp Game.cpp Particle.cpp Point.cpp \
$(pkg-config --cflags sdl2) $(pkg-config --libs sdl2)
```

Note: the backslash above is just used to mark the end of the line. In a Unix-like terminal, you can continue a command on to a second line by placing a backslash at the end of the previous line. If you are typing the above command in to a terminal as a single line, omit the backslash.

If you are using Windows, there are many tutorials online for properly using SDL with Visual Studio or MinGW. As mentioned at the beginning of this document, I have a virtual machine image available that you can use to test your program.