CS105C: Computer Programming C++
Programming Assignment #2

Adrian Trejo Nuñez
atrejo@cs.utexas.edu

Due: 11:59pm September 28, 2018

The purpose of this assignment is to familiarize yourself with implementing C++ classes and the standard template library.

1 Description

Arkanoid is an old-school arcade video game. The player controls a paddle to prevent a bouncing ball from reaching the bottom of the screen, while directing the ball to hit and eliminate as many bricks as possible. You will implement such a game for this assignment.

Video games like Arkanoid are usually implemented with the game loop pattern. The main function of the program consists of a simple loop that looks like this:

```
while (!gameOver) {
  playerInput = readInput();
  world.update(playerInput);
  render(world);
}
```

Every spin of the loop consists of three steps: (1) read the player's input, (2) update the game world according to the input, and (3) render the game world onto the screen. Writing the full game by implementing all three steps is an interesting project but also an excessive amount of work for a programming assignment. Our focus is solely the *update* logic of the game, i.e. implementing step two in the game loop. Adding support for player interaction and graphical rendering to the game is beyond the scope of this exercise and is left as an exercise to the reader.

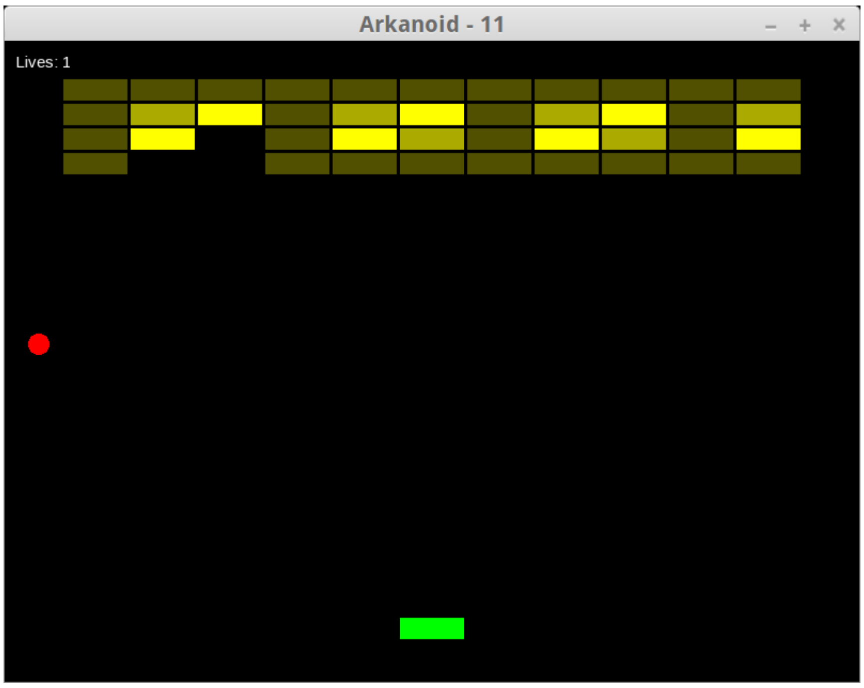

Figure 1: Arkanoid Game

2 Directions and Hints

2.1 Download the skeleton project

Download the the assignment file `assignment2.tar.gz` to the lab computers (physically or using `scp`). Unpack the tarball:

```
> tar -xvf assignment2.tar.gz --one-top-level=assignment2
```

This will create a new directory called `assignment2` in the current directory with the tarball unpacked under it. Take a look at the skeleton code. You will be modifying the following classes:

- `Point` represents a 2D point on the screen[1].
- `Velocity` represents a 2D vector that indicates the direction and speed of movement.
- `Ball` represents the bouncing ball. It has a circle shape and thus can be located by a `Point` that represents its center and an integer that represents its radius. We also need to associate a `Velocity` to the ball since it can move freely in the game world.

[1]In our coordinate system, origin point $(0, 0)$ is located on the top-left corner of the screen

- `Paddle` represents the paddle that the player controls. It has a rectangle shape and thus can be represented by a `Point` for the location of its upper-left corner and a pair (w, h) for its width and height. The paddle is also movable but can only do so horizontally, so it is sufficient to represent its velocity with a single integer.
- `Brick` represents a brick that the player has to destroy. Like the paddle, it also has a rectangle shape so we represent it in the same way, but without a velocity.
- `World` represents the entire game world. It is a rectangle area with $(0, 0)$ as its top-left corner and (w, h) as its lower-right corner, where w and h are its width and height. This rectangle defines the area of the entire coordinate system that is visible to the player.

Additionally, there is a `main.cpp` that serves as the driver for the project and a `Makefile` that builds the binary. The skeleton project will fail to compile at first, since none of the member functions declared in the various `.h` files will be implemented. Your job is to provide the implementations for all of these missing functions.

2.2 Library

Complete the member functions listed below. The code you write in this section is going to serve as base utilities for the next section.

- `Ball`
  - Implement the constructor of the `Ball` class. The constructor takes three parameters: the center point of the ball, the radius of the ball, and the velocity of the ball.
  - Implement the `getCenter()`, `getRadius()`, and `getVelocity()` member functions, which returns the center point, the radius, and the velocity of the ball, respectively.
- `Paddle`
  - Implement the constructor of the `Paddle` class. The constructor takes three parameters: the upper-left corner of the paddle, the width of the paddle, the height of the paddle, and the velocity of the paddle.
  - Implement the `getUpperLeft()` and `getLowerRight()` member functions, which returns the upper-left corner and the lower-right corner of the paddle, respectively.
- `Brick`
  - A brick is like a paddle without velocity. Implement its constructor, `getUpperLeft()` and `getLowerRight()` member functions similarly to the `Paddle` class.
- `World`

  - Implement the `World` constructor. There are 5 parameters in the constructor: its width, its height, a ball, a paddle, and a vector of bricks.
  - Implement the `get()` functions for each of the 5 parameters.

2.3 Game Logic

The core of this assignment is implementing the two functions `World::isLegal()` and `World::update()`.

2.3.1 `World::isLegal()`

Users of the `World` class are able to use the class' constructor to create arbitrary game world, including worlds that contain an illegal game states. For example, there is nothing that prevents the user from constructing a world without any bricks, or a world with a ball of velocity zero. The `isLegal()` member function should detect such illegal initial game states. Specifically, it does the following check:

- There should be at least one brick in the world.
- The ball must have a nonzero velocity.
- All entities (ball, paddle, and bricks) must locate completely inside the boundaries of the world.
- Entities must not collide with one another[2]

2.3.2 `World::update()`

Given a legal `World` object, the final task is to write the update logic for the game: `World::update()`. The function takes the user input as a parameter, which is an scoped enum that has three possibilities: `Input::None` (player does not press any key), `Input::Left` (player wants to move the paddle to the left), `Input::Right` (player wants to move the paddle to the right). Based on the user input, it should update the internal state of the `World` object:

- First, the paddle moves. If the original location of the paddle is (x, y), the new location would be $(x + v, y)$, where v is the paddle's velocity. However, if the paddle would move beyond the left (or right) boundary of the game world, you need to keep it inside the world and align it to the left (or right, respectively) world boundary.
- Next, the paddle's velocity changes according to the user input.

[2]For collision detection, view all entities (including the ball) as a box, and use the simple `Axis-Aligned Bounding Box` algorithm. Note that according to the algorithm, if the bounding boxes of two game entities intersect only through an edge or through a vertex then this does not count as a collision.

  - If the input is `Input::None`, set the velocity to zero.
  - If the input is `Input::Left`, set the velocity to `-Paddle::DefaultVelocity`.
  - If the input is `Input::Right`, set the velocity to `Paddle::DefaultVelocity`.

- Then, the ball moves. If the original location of the ball is (x, y), the new location would be $(x + v_x, y + v_y)$, where v_x and v_y are the ball's horizontal velocity and vertical velocity, respectively.
  After this step, it is OK if the ball moves to an illegal position (e.g. it goes outside of the game world, or it overlaps with either the paddle or some bricks). In the next step, we will fix that by changing the ball's velocity in such a way that will move the ball out of the illegal position in the next couple of frames.

- Next, the ball's velocity changes.
  - If the ball collides with or goes beyond the left boundary of the world, set its horizontal velocity to `Ball::DefaultHorizontalVelocity`.
  - If the ball collides with or goes beyond the right boundary of the world, set its horizontal velocity to `-Ball::DefaultHorizontalVelocity`.
  - If the ball collides with or goes beyond the upper boundary of the world, set its vertical velocity to `Ball::DefaultVerticalVelocity`.
  - If the ball collides with the paddle, set its vertical velocity to `-Ball::DefaultVerticalVelocity`.
  - If the ball collides with or goes beyond the lower boundary of the world, the game is over. We handle this case by throwing an exception: `throw GameOverException();`[3]

- Finally, remove all bricks that collide with the ball.

If you have a question about various containers (e.g. vector), check cppreference.com. Don't be afraid to make a small program just to try out some code. The compiler is your friend.

3 Submission and Grading

- You will submit a single file `assignment2.tar` on Canvas containing all of the files from the assignment, except for `main.cpp`.

- Make sure your final submission compiles without any external dependencies other than the C++ standard libraries. I will compile it using the included `Makefile` with the command `make`. If your code does not compile, you will receive zero points for this assignment.

[3]We use exceptions just to make the task simple. Please be aware that this is not how C++ exceptions are supposed to be used. You would not want to do this in real-world game development.

- I will automate the grading process by running your program against several test cases.
- In particular, I will use my own main file that creates several test cases to make sure your implementation is correct. You should use the main file to add your own test cases.
- For the grading script to work, do not change any function signature or remove any existing source files, classes or methods from the skeleton project. You may add extra files, classes, or methods, as long as you modify the `Makefile` accordingly.
- A small portion of your grade is style, so please make sure you comment your file and you adhere to good coding practices. In particular, you should be consistent throughout your code.
- If you have any questions about the assignment, please ask on Piazza.