Homework 5: Brick Out

CS 211

Winter 2020

Contents

1 Purpose

2 Getting it

3 Game description
  3.1 Physics
  3.2 Geometry

4 Design orientation
  4.1 The model
  4.2 The UI

5 Implementation hints
  5.1 The ball
  5.2 The model
  5.3 The UI

6 Reference
  6.1 The GE211 geometry types

7 Deliverables and evaluation

8 Submission

1 Purpose

The primary goal of this assignment is to get you programming in C++ with member functions and std::vector. Secondarily, we want to familiarize you with the mechanics of GE211.

2 Getting it

Download the project ZIP file to your computer1, unzip it, and open the resulting directory in CLion. (Be careful that you open the hw05 directory and not some sub- or superdirectory thereof. If you do, CLion will create a bogus CMakeLists.txt that won’t be able to find SDL2.)

3 Game description

In this classic arcade game, the player seeks to destroy a field of bricks in the top portion of the screen by hitting them with a ball, while controlling a horizontally-moving paddle to prevent the ball from reaching the bottom of the screen.

---

1To complete this homework on your own computer, you need a C++14 toolchain and the SDL2 libraries. Follow these instructions to install them on your own computer, or if you need to work on a lab computer instead then see here.
When the game starts, a grid of rectangular bricks appears in the top portion of the screen, and the paddle, also a rectangle, appears at the bottom of the screen. The paddle moves horizontally with the \( x \) coordinate of the mouse pointer, but its \( y \) coordinate never changes.

Initially the ball is “dead”—rather than bouncing around, it sticks to the paddle as the paddle follows the mouse. When the player clicks the mouse or hits the space key, the ball is launched and travels upward toward the bricks. It then proceeds to bounce off of bricks, the paddle, and the top and sides of the screen, destroying each brick that it collides with, until it reaches the bottom of the screen. At that point the ball is again dead and stuck the paddle. No bricks are restored, however, and the player may launch the ball again.

3.1 Physics

Physics in the Brick Out world is highly idealized. For the purpose of detecting collisions, we approximate the ball as its bounding box\(^2\). The ball’s mass is insignificant compared to every object it meets, so it rebounds fully and they never budge. Collisions with the top and sides of the screen are perfectly elastic and perfectly conventional—the top reflects vertically and the sides reflect horizontally. But collisions with blocks, both the paddle and the bricks, are a bit weirder.

Upon striking (and destroying) a brick, the ball is reflected both vertically and horizontally, regardless of which edge of the brick it contacts. In other words, both the \( x \) and \( y \) components of its velocity are negated and it returns from whence it came.

When the ball collides with the paddle, the reflection is perfectly elastic in the \( y \) dimension, but it gets a random “boost” in the \( x \) dimension. In particular, the horizontal component of its velocity is adjusted by the addition of a random small integer (balanced between negative and positive to produce a random walk with constant expectation).

3.2 Geometry

This diagram shows a 5-by-5 field of gray bricks (at the top), the yellow paddle (at the bottom), and the red ball in its dead position:

Unlike the diagram above, in the default geometry the brick field is 10-by-10. In addition to the numbers of columns and rows of bricks, the geometry lets you control:

- the dimensions of the screen;
- the distance from the top of the screen to the top of the brick field;
- the distance from the sides of the screen to the sides of the brick field;
- the distance from the top of the screen to bottom of the brick field;
- the dimensions of the gaps between the bricks;
- the distance from the bottom of the screen to bottom of the paddle;
- the dimensions of the paddle;
- the radius of the ball;
- the initial velocity of the ball once it’s launched from the paddle; and
- the maximum absolute “boost” value for when the ball hits the paddle.

From these properties the geometry class computes the dimensions of the bricks and the initial position of the paddle, which cannot be adjusted independently.

You should test your code, both model and view, with varying geometries. Not all combinations are sensible, but your code should work correctly within a reasonable range.

\(^2\)The bounding box of a figure is the smallest rectangle enclosing it; for the ball it’s a square sharing its center whose side length is twice the radius of the ball.
4 Design orientation

The BRICK OUT game is composed of two major components: the model, which keeps track of the state of the game independent of how it is viewed or controlled, and the UI, which provides an interface to use the model by specifying how it appears on the screen and reacts to our input.

4.1 The model

The model (struct Model in src/model.{h,cxx}) represents the game’s logical state and implements its rules in a UI-independent manner. For BRICK OUT, it keeps track of:

- the locations and sizes of all the bricks,
- the location and size of the paddle (the thing at the bottom that you control), and
- the state of the ball, including whether it’s in play, and its size, location, and velocity.

As far as operations, the model knows how to put a dead ball back into play, how to move the paddle to a new position (bringing a dead ball along with it), and how to update its own state for each animation frame.

Because the state and behavior of the ball account for much of the complexity of the model, the ball is factored out into its own struct Ball (in src/ball.{h,cxx}). It defines its own set of operations, mainly for detecting collisions with bricks, the paddle, and the edges of the screen.

The model is also responsible for storing the game geometry parameters (e.g., the sizes of things such as bricks, the paddle, the margins, and the window), which are grouped into a struct Geometry (in src/geometry.{h,cxx}). The geometry is passed to the Model constructor and is then fixed for the duration of the game.

4.2 The UI

The UI (struct Ui in src/ui.{h,cxx}) combines the view and controller components of the traditional model–view–controller (MVC) design, along with a reference to the model.

The view state defines the sprites used to represent the game entities on the screen, as well as two operations: a drawing operation that places those sprites based on the state of the model, and a simple function to convey the game dimensions from the model geometry to GE211.

The controller portion of the UI is stateless and defines four operations—three for reacting to user input and one for reacting to the passage of time. It reacts to two key events: it exits on q and launches a possibly-dead ball on space. It also launches the ball on mouse clicks. When the mouse moves, it tells the model to move the paddle. And with each frame (typically 1/60 s), it asks the model to update itself to reflect the passage of time.

5 Implementation hints

There is no specification in this document—instead, the functions you need to implement are specified in the header files src/ball.hxx, src/model.hxx, and src/ui.hxx, so you should read those carefully. This section provides supplementary material to help you figure out how to implement what the header comments specify.

5.1 The model: struct Ball and friends

The implementation of model logic related to the ball is in src/ball.cxx. There are seven Ball member functions and two free functions for you to complete.

```
static Position above_block(Block const&, Geometry const&)
```

This function is a helper for Ball’s constructor that computes where the ball should be when it’s dead—its bottom centered 1 pixel above the top center of the paddle.

Given block (a ge211::Rectangle representing the position and dimensions of the paddle), start at its top-left corner (Rectangle::top_left()), move to the right (Position::right_by()) by half the width of block (Rectangle::width), then move up (Position::up_by()) by 1 plus the radius of the ball (Geometry::ball_radius).

```
Ball::top_left() const
```

Returns the position at the upper-left corner of the ball’s bounding box. This is the position one ball radius to the left and one ball radius above the center of the ball.

```
Ball::hits_bottom( Geometry const&)
```

4 Design orientation
5 Implementation hints

The ball hits the bottom when the $y$ coordinate of its bottom exceeds the height of the scene.

```cpp
Ball::hits_top(
   Geometry const&)
```

The ball hits the top when the $y$ coordinate of its top is less than 0. (Note that the parameter isn’t used.)

```cpp
Ball::hits_side(
   Geometry const&)
```

The ball hits a side when the $x$ coordinate of its left side is less than 0 or the $x$ coordinate of its right side is greater than the width of the scene.

```cpp
Ball::next()
```

Recall that this is a const Ball*, and you can create a copy of a ball with the copy constructor. So to get a new Ball to return, you can write

```cpp
Ball result(*this);
```

```cpp
Ball::hits_block(Block const&)
```

As with the edge collision functions, we want to use the ball’s bounding box, which is the square whose top is `center_.y - radius_`, whose left is `center_.x - radius_`, whose bottom is `center_.y + radius_`, and whose right is `center_.x + radius_`. We use the bounding box so that we can check for the intersection of two rectangles, which is easier than checking for the intersection of a rectangle and a circle.

One way to think of that is that the rectangles don’t intersect if either of these is true:

- The right side of either rectangle is to the left of the left side of the other.
- The bottom of either rectangle is above the top of the other.

Otherwise, they do.

```cpp
Ball::destroy_brick(
   std::vector<Block>&)
```

Once you’ve written `Ball::hits_block`, finding an element of bricks that collides with this ball isn’t hard—use a for-each loop—but how to remove it once you find it? The more obvious solution may be to shift all the elements after it to the left, but that’s awkward, and there’s a cleaner way when the order of the elements of the vector doesn’t matter:

1. Swap the hit brick with the last brick in the vector (bricks.back()) using std::swap.
2. Now the hit brick is in the back, so you can remove it with std::vector::pop_back().
3. return true immediately after the pop_back(). The loop condition won’t adjust to the diminished vector, so if you keep iterating after removing an element then you’ll go out of bounds. One brick is enough.

```cpp
operator==(Ball const&, Ball const&)
```

This can be written as a four-way && expression.

5.2 The model: struct Model

The implementation of the remaining model logic is in src/model.cxx. There are two Model member functions and one constructor for you to complete.

```cpp
Model::Model(Geometry const&)
```

Constructs a Model from the given Geometry. Note that the Geometry is passed by const& but Model saves its own copy of it.

This much is done for you: The geometry_.paddle_, and ball_ member variables are initialized in a member initializer list, not in the body of the constructor:

- The paddle is initialized with its top-left at `geometry_.paddle_top_left()` and with dimensions `geometry_.paddle_dims_`
- The ball is initialized with the state of the paddle and the geometry.

What you need to do: In the body of the constructor, iterate through the positions of all the bricks (geometry_.brick_rows * geometry_.brick_cols of them) and push_back each into the bricks_ vector. The details:

- Each brick should have dimensions `geometry_.brick_dims_`
- The first (top-left-most) brick should have its top left at the position `{geometry_.side_margin, geometry_.top_margin}`
6 Reference

- You will need nested loops to create all the bricks in each row and column, but note that the order in the vector doesn’t matter.
- The offset between each brick and the next is given by the dimensions of each brick plus `geometry_.brick_spacing`. Or in other words, the $x$ offset is `geometry_.brick_spacing.width + geometry_.brick_dims().width`, and the $y$ offset is the same but with heights.

```cpp
Model::paddle_to(int x)
```

In addition to moving the paddle, this may need to move the ball. If the ball isn’t live then it needs to follow the paddle, which is best done by constructing a new `Ball` and assigning it to `ball_`.

```cpp
Model::update(int boost)
```

The description in `src/model.hxx` is pretty detailed, so there’s not too much else to say. You probably want to call `Ball::next()` `const` at most twice: once speculatively as soon as you know that the ball is live, and once again at the end, storing the result back to the ball for real that time.

### 5.3 The UI

The implementation of the user interface, including both drawing and reacting to input, is in `src/ui.cxx`. There are five `Ui` member functions for you to complete.

```cpp
Ui::draw(ge211::Sprite_set&)
```

Use `Sprite_set::add_sprite(Sprite&, Position)` to add each sprite to `sprites`. Note that `add_sprite` positions the sprite using the top-left corner of its bounding box, so you don’t want to position a circle by its center.

```cpp
Ui::on_key(ge211::Key)
```

The starter code already quits on `q`. To make a dead ball start moving on spacebar, you need to check for `ge211::Key::code(' ')` and call `model_.launch()` when you get it. (That’s a “visible” space character, not an underscore.)

```cpp
Ui::on_frame(double)
```

To get a random number, you should use a `ge211::Random` object, which provides the member function

```cpp
int Random::between(int min, int max);
```

which returns a random `int` from the closed interval `[min, max]`.

So how can you get access to a `ge211::Random` object? `Ui`’s base class `ge211::Abstract_game` constructs and stores one for us, granting us by-reference access via its `get_random()` member function.

```cpp
Ui::on_mouse_up(ge211::Mouse_button, ge211::Position)
```

Makes the ball live via `Model::launch()`.

```cpp
Ui::on_mouse_move(ge211::Position)
```

Informs the model of the mouse position (and thus the desired paddle position) via `Model::paddle_to(Position)`.

### 6 Reference

#### 6.1 The GE211 geometry types

The GE211 library defines three types for representing the geometry of points and rectangles. You will need to use these types to calculate the positions of game entities and place them on the screen, so read on.

```cpp
struct ge211::Position
```

For representing 2-D positions, either logical or in screen pixels, GE211 provides the `Position` struct. While the actual definition is more complicated, the basic idea can be understood as:

```cpp
struct ge211::Position
{
    int x;
    int y;
};
```

3That may look like a free function call, not a member function call, but it’s actually being called on `this`, since members of a base class (from which we derived our struct) are members of the derived struct as well.
7 Deliverables and evaluation

<table>
<thead>
<tr>
<th>general term</th>
<th>memory</th>
<th>time</th>
<th>C++ STL</th>
<th>GE211</th>
</tr>
</thead>
<tbody>
<tr>
<td>point</td>
<td>pointer</td>
<td>time point</td>
<td>iterator</td>
<td>Position</td>
</tr>
<tr>
<td>displacement</td>
<td>integer</td>
<td>duration</td>
<td>difference type</td>
<td>Dimensions</td>
</tr>
<tr>
<td>span</td>
<td>array</td>
<td>time span</td>
<td>range</td>
<td>Rectangle</td>
</tr>
</tbody>
</table>

Figure 1: Affine spaces

The `Position` struct provides a variety of member functions, such as `Position::up_by(int) const` and `Position::down_right_by(Dimensions) const`, for shifting to related positions.

```cpp
struct ge211::Dimensions
{
    int width;
    int height;
};
```

Why do we need `Dimensions` if we have `Position`? Aren’t these basically the same thing? Yes, each is a pair of `ints`, one with a horizontal sense and the other vertical, but semantically they are different and their operations differ. For example, it makes sense to add two `Dimensions`, or to multiply a `Dimensions` by a scalar:

```cpp
operator+(Dimensions, Dimensions);
operator*(Dimensions, double);
```

Both of these operations yield a `Dimensions`. But it doesn’t mean anything to add two `Position`s, or to scale a `Position`. So having separate types for `Position` and `Dimensions` helps us keep the two concepts precise and prevents at least some kinds of nonsense.

The algebra of positions and dimensions is a two dimensional generalization of the algebra of pointers and integer offsets (see Fig. 1), which can help us understand what other operations are meaningful. Like adding an integer to a pointer in order to offset the pointer, it makes sense to add a `Dimensions` to a `Position` to get an offset `Position`. And as the difference between two pointers is an integer, the difference between two `Position`s is a `Dimensions`.

```cpp
struct ge211::Rectangle
{
    int x;
    int y;
    int width;
    int height;
};
```

In Brick Out, we use `ge211::Rectangles` to represent blocks (both bricks and the paddle), so for convenience, `src/ball.hxx` typedefs `Block` to mean `ge211::Rectangle`.

A `Rectangle` is essentially a pairing of a `Position` (its top left corner) with a `Dimensions`. You can create one from those parts and project each part back out. To create one you might use

```cpp
static Rectangle
Rectangle::from_top_left(Position, Dimensions);
```

among other static factory functions. To project, you will want member functions such as

```cpp
Dimensions Rectangle::dimensions() const;
Position Rectangle::top_left() const;
```

and others.

You can also access the data members of a `ge211::Rectangle` directly, but note that they don’t actually include a `Rectangle` and a `Dimensions`, but rather both flattened together:

```cpp
struct ge211::Rectangle
{
    int x;
    int y;
    int width;
    int height;
};
```

7 Deliverables and evaluation

For this homework you must:

1. Complete the seven unimplemented `Ball` member functions and two free functions (`above_block()` and `operator=(Ball const&, Ball const&)`) in `src/ball.cxx`.
2. Complete the unimplemented `Model` constructor and two member functions in `src/model.cxx`.
3. Complete the five unimplemented `Ui` member functions in `src/ui.cxx`.

6
4. Add more test cases to `test/ball_test.cxx` and `test/model_test.cxx` in order to test the functions that you wrote.

(We don’t have a way for you to write automated tests for the UI, but you should test it interactively.)

As usual, self evaluation will spot-check your test coverage by asking for just a few particular test cases. You certainly want to test each significant event, such as the ball hitting the paddle or the ball falling off the bottom of the screen. You can’t anticipate what other cases we may ask about, so you should try to cover everything.

Your grade will be based on:

- the correctness of your implementations with respect to the specifications,
- the presence of sufficient test cases to ensure your model code’s correctness, and
- adherence to the CS 211 Style Manual.

8 Submission

Homework submission and grading will use the GSC grading server. You must include any files that you create or change. For this homework, that will include `src/ball.cxx`, `src/model.cxx`, `src/ui.cxx`, `test/ball_test.cxx`, and `test/model_test.cxx`. (You should not need to submit a modified `CMakeLists.txt` and you must not modify any of the `.hxx` files besides `src/ui.hxx`.

You should upload your files on the GSC web site.\(^4\)

\(^4\)It’s also possible to install the command-line GSC client `gsc(1)` on your local machine.