Homework 4: Brick Out

CS 211

Spring 2023

Code Due:	May 11, 2023, 11:59 PM, Central Time
Self-Eval Due:	May 14, 2023, 11:59 PM, Central Time
Partners:	No; must be completed by yourself

Maximum allowed Gradescope submissions: 30

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.

Getting it

Download the project ZIP file to your computer[1], unzip it, and open the resulting directory in CLion. (Be careful that you open the `hw4` 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.)

[1] To complete this homework on your own computer, you need a C++14 toolchain and the SDL2 libraries. Follow these instructions to install the software you need.

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.

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 then 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.

Contents

Purpose 1
Getting it 1
Game description 1
- Physics 2
- Game configuration . . . 2

Design orientation 3
- The model 3
- The view 4
- The controller 4

Implementation hints 5
- The model: `struct Ball` and friends 5
- The model: `struct Model` 7
- The view 8
- The controller 9

Reference 9
- The GE211 geometry types 9

Deliverables & evaluation 12
Submission 12

Physics

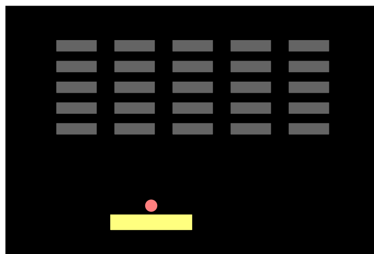
Physics in the Brick Out world is highly idealized. For the purpose of detecting collisions, we approximate the ball as its bounding box. The ball's mass is insignificant compared to every object it meets, so it rebounds fully and they never budge. Collisions with the top and sides of the screen are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

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.

Upon striking (and destroying) a brick, the ball is reflected vertically, regardless of which edge of the brick it contacts. In other words, the y component of its velocity is negated and the x component is not. In addition, the ball receives a random "boost" in the x dimension. In particular, the horizontal component of its velocity is adjusted by the addition of a random small number (balanced between negative and positive to produce a random walk with constant expectation). The potential range of that random number is determined by the game configuration.

Game configuration

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 game configuration the brick field is 10-by-10. In addition to the numbers of columns and rows of bricks, the configuration 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 the 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 a brick.

From these properties the `Game_config` 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 user interface, with varying configurations. Not all combinations are sensible, but your code should work correctly within a reasonable range.

Design orientation

The BRICK OUT game is composed of three major components:

- the *model*, which keeps track of the state of the game independent of how the user interacts with it,
- the *view*, which shows information about the state of the model to the user, and
- the *controller,* which coordinates the model's (and sometimes view's) reactions to events (such as mouse motion or key presses).

The model

The model (in src/ball.{hxx,cxx} in src/model.{hxx,cxx}) represents the game's logical state and implements its rules in a user interface–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),
- the state of the ball, including whether it's in play, and its size, location, and velocity, and
- a source of random numbers, which can be *stubbed* to return predictable values for testing.

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 (typically 1/60 s, but it can vary).

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.{hxx,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 configuration parameters (*e.g.*, the sizes of things such as bricks, the paddle, the margins, and the window), which are grouped into a `struct Game_config` (in src/game_config.{hxx,cxx}). The game configuration is passed to the `Model` constructor and is then fixed for the duration of the game.

The view

The view (in src/view.{hxx,cxx}) is responsible for producing output for the user—in the case of GE211, this means drawing to a window on the screen. The view does this by looking at the model (via a `const&`) and placing *sprites* (graphical 2-D objects, shapes, or images) on the screen.

The view's state thus defines the sprites used to portray the game entities (ball, paddle, bricks) on the screen. It declares two member functons: a drawing operation whose job is to place those sprites based on the state of the model (you'll write this), and a simple function to convey the game dimensions from the configuration to GE211 (we wrote this).

The controller

The controller (in src/controller.{hxx,cxx}) owns the view and holds a reference to the model, and otherwise has no state of its own.

In general this ownership structure can vary.

It defines three operations for reacting to user input: key presses, mouse clicks, and mouse motion. When `q` is pressed, it causes the game to exit. If the ball is dead when either the mouse is clicked or spacebar is pressed, it causes the model to launch the ball. And when the mouse moves, it causes the paddle (in the model) to follow it.

In the GE211 architecture, the framework only talks to the controller, not the model and the view, so the controller is also responsible for mediating between the framework and the model and view. It does this by defining three additional member functions `Controller::on_frame`, `Controller::draw`, and `Controller::initial_window_dimensions`, each which merely forwards to `Model::on_frame`, `View::draw`, and `View::initial_window_dimensions`, respectively.

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, src/view.hxx, and src/controller.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.

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 (non-member functions) for you to complete.

```
static ge211::Posn<float>
above_block(Block const&,
            Game_config 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::Rect<int>` representing the position and dimensions of the paddle), start at its top-left corner (`Rect<int>::top_left()`), move to the right (`Posn<int>::right_by()`) by half the width of `block` (`Rect<int>::width`), then move up (`Posn<int>::up_by()`) by 1 plus the radius of the ball (`Game_config::ball_radius`).

Depending on your implementation, you might find that you get a conversion error the first time you write this code. The issue is that each block has its coordinates specified as `int` values, but the ball `Position` has coordinates specified as `float` values. You'll need to convert between the two! The easiest way to do this is to construction a new `Position` with the coordinate values of the `ge211::Posn<int>` from the block. Alternatively, the `Posn::into<TYPE>()` method will do the job.

```
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(
    Game_config const&) const
```

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

```
Ball::hits_top(Game_config const&) const
```

The ball hits the top of the scene when the y coordinate of its top is less than 0. (Note that the parameter isn't used in this case, but we include it for symmetry.)

```
Ball::hits_side(Game_config const&) const
```

The ball hits the side of the scene when either 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.

```
Ball::next(double dt) const
```

Within the members of an object, the keyword `this` is a pointer to the current object. Therefore, `this` is a `Ball const*`, and you can create a copy of a ball with the copy constructor. So to get a new `Ball` object (named `result`) to return, you can write

A copy constructor takes in an object of the same type and copies its values to the current object being created.

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

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

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

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. Replace the hit brick with the last brick in the vector (`bricks.back()`) by assigning over it. (If the hit brick *is* the last brick in the vector then this step won't do anything, but this algorithm will still work without a special case.)
2. Now the last brick in the vector is redundant, so remove it using `std::vector::pop_back()`.
3. `return true` immediately after the `pop_back()`. The loop condition won't adjust to the diminished vector size, so if you keep iterating after removing an element then you'll go out of bounds. One brick is enough.

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

This is how you will overload the `==` operator to check equality between two ball objects. It can be written as a four-way `&&` expression.

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.

```
Model::Model(Game_config const&)
```

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

This much is done for you: the `config`, `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 `config.paddle_top_left_0()` and with dimensions `config.paddle_dims`.
- The ball is initialized with the state of the paddle and the game configuration.

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

- Each brick should have dimensions `config.brick_dims()`.
- The first (top-left–most) brick should have its top left at the position `{config.side_margin, config.top_margin}`.
- 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 `config.brick_spacing`. Or in other words, the x offset is `config.brick_spacing.width + config.brick_dims().width`, and the y offset is likewise but with heights.

```
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 then it needs to follow the paddle, which is best done by constructing a new `Ball` and assigning it to `ball`.

```
Model::on_frame(double dt)
```

And with each frame (typically 1/60 s), it asks the model to update itself to reflect the passage of time.

The description in src/model.hxx is pretty detailed. You probably want to call `Ball::next(double) 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.

When the ball destroys a brick, you will need to generate a random boost. The model contains a data member, `random_boost_source`, whose type is `ge211::Random_source<float>`, which can be used the generate random `float`s. The `Model` constructor, which we've defined for you, takes a `Game_config const& config` parameter and initializes its `random_boost_source` member to generate random values between `-config.max_boost` and `config.max_boost`. Thus, you can generate a random boost from `random_boost_source` using the member function `Random_source<float>::next()`.

The view

The implementation of the view is in src/view.cxx. The `View` constructor, which initializes its model reference and sprites, is already defined

for you, as is one of the member functions. There's one `View` member function for you to complete:

```
View::draw(ge211::Sprite_set&)
```

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

The controller

The controller is implemented in src/controller.cxx. There are three `Controller` member functions for you to complete:

```
Controller::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 space character, not an empty character.)

```
Controller::on_mouse_up(
    ge211::Mouse_button,
    ge211::Posn<int>)
```

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

```
Controller::on_mouse_move(
    ge211::Posn<int>)
```

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

Reference

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.

```
struct ge211::Posn<T>
```

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

```
struct ge211::Posn<float>
{
    float x;
    float y;
};
```

However, `ge211::Posn` is a *struct template*, which means that the coordinate type isn't fixed at `float`. You can make a `Posn<double>` whose coordinates are `double`s, a `Posn<int>` whose coordinates are `int`s, and so on. It provides a variety of member functions, such as `Posn<int>::up_by(int) const` and `Posn<int>::down_right_by(Dims<int>) const`, for computing related positions.

```
struct ge211::Dims<T>
```

For representing the width and height of a 2-D objects (such as bounding boxes), GE211 provides the `Dims<T>` struct template. As with `ge211::Posn`, the real definition is a bit more complicated, but you can think of it as:

```
struct ge211::Dims<float>
{
    float width;
    float height;
};
```

Why do we need `Dims` if we have `Posn`? Aren't these basically the same thing? Yes, each is a pair of numbers, 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 `Dims`es, or to multiply a `Dims` by a scalar, so the infix operators `+` and `*` are *overloaded* with signatures such as

- `Dims<T>::operator+(Dims<T>) const` and
- `Dims<T>::operator*(double) const`.

But it doesn't mean anything to add two `Posn`s, or to scale a `Posn`. So having separate types for `Posn` and `Dims` 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 Table 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 `Dims` to a `Posn` to get an offset `Posn`. And as the difference between two pointers is an integer, the difference between two `Posn`s is a `Dims`.

Table 1: Affine spaces

general term	memory	time	C++ STL	GE211
point	pointer	time point	iterator	Posn<T>
displacement	integer	duration	difference type	Dims<T>
span	array	time span	range	Rect<T>

```
struct ge211::Rect<T>
```

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

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

```
static Rect<int>
Rect<int>::from_top_left(
    Posn<int>,  // top left vertex
    Dims<int>)  // width and height
```

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

- `Dims<T> Rect<T>::dimensions() const` and
- `Posn<T> Rect<T>::top_left() const`

among others.

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

```
struct ge211::Rect<int>
{
    int x;
    int y;
    int width;
    int height;
};
```

Deliverables & 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 one unimplemented `View` member function in src/view.cxx.
4. Complete the three unimplemented `Controller` member functions in src/controller.cxx.
5. Add more test cases to test/ball_test.cxx and test/model_test.cxx in order to the test the functions that you wrote.

   We don't require you to write automated tests for the controller and view, but it might not be a bad idea to try.

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 for C++ (linked here).

Submission

Homework submission and grading will use Gradescope. You must include any files that you create or change. For this homework, that will include src/ball.cxx, src/model.cxx, src/view.cxx, src/controller.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.)

Per the syllabus, if you engaged in arms-length collaboration on this assignment, you must cite your sources. You may write citations either in comments on the relevant code, or in a file named

README.txt that you submit along with your code. See the syllabus for definitions and other details.

Submit your files by uploading them directly to the Gradescope website. When you click on the assignment in Gradescope for the first time, you will get a window where you can upload files. You can drag-and-drop files or browse to select them. Make sure you include all the necessary files in the src/ and test/ directories! Submitting extra files is fine. To submit additional times, select the “Resubmit” button on the bottom right.