Final Project

CS 211

Spring 2020

Proposal due:	May 20, 2020 at 11:59 PM (CDT)	(as HW7 on GSC)
Code due:	June 3, 2020 at 11:59 PM (CDT)	(as HW8 on GSC)
Eval guide due:	June 4, 2020 at 11:59 PM (CDT)	(as HW8 self eval)
Partners:	Yes; register on GSC before submission	

Contents

1. Purpose

The goal of this assignment is to let you apply the programming skills you've acquired in service of your own creativity.

2. Getting it

While there isn't significant starter code for the final project, we've prepared a project ZIP file containing skeletons for the three files you need to submit with your proposal—PROPOSAL.md, src/model.hxx, and test/model_test.cxx—as well as all the dependencies and CMake configuration you need to get started.

3. Project requirements

For the final project, you must implement a game (or other interactive, graphical program) using C++ and GE211. There are three phases to the project: proposal, negotiation, and delivery. In the proposal phase, you write a description of your game—in the format described below—and submit it for TA approval. In the negotiation phase, the TA may approve your design or request changes, potentially more than once. Once your proposal is accepted, you begin the delivery phase, wherein you actually implement your game.

The game is expected to be of moderate complexity, perhaps twice as complex in terms of requirements as Homework 5's Brick Out or Homework 6's Reversi. We will be more precise about assessing this aspect of your proposal below.

4. Proposal format

Your proposal must have the following five sections. The first three sections must be written in the file

[1] The .md file extension indicates Markdown, which is a text-based format for lightly-styled text. If you edit it in CLion and install the Markdown plugin then it will give you syntax highlighting and a preview of the rendered styling.

PROPOSAL.md[1], and the last two have files of their own.

4.1. Synopsis

The synopsis is a brief description of the game. You may divide it into subsections, such as "Elements," "Goal," and "Game play," as I do in the example below, but you don't need to. Your purpose with this section it to communicate, concisely, what the game is all about.

(If you need a length guideline, go for 100–200 words.)

4.2. Functional requirements

This is a list of 12–16 specific, identifiable things that your program will do. These features must be observable to a player, since the TAs will play your game and use these requirements as a checklist for grading. (It's okay if some requirements are difficult for a player to reach, but you will have to justify those by reference to your code.)

It may be a bit tricky to figure out the best granularity for describing functional requirements. It would not be good, for example, to have two separate requirements: "Pressing the left arrow key moves the player to the left," and "Pressing the right arrow key moves the player to right right." Instead, that should be a single requirement, perhaps: "The player is controlled by the arrow keys." This is a matter of taste and judgment, so see the example below for guidance, and then consult with the course staff or ask on Campuswire about how to specify your particulars.

4.3. Open questions

What don't you know yet about how your game will work? List your open questions in this section. Surely you have some. Maybe the TA will have some suggestions to help you answer them.

4.4. Model sketch

We want to see a first draft of how you think you might design your model—this is the model sketch in src/model.hxx. This should include your best guesses for whatever

- classes and structs,
- private data members,
- public operations, and
- private helpers

you expect to need.

Provide a succinct "purpose statement" comment on each of the above explaining what it's for.

4.5. Example model tests

Finally, we want to see at least five interesting test cases in test/model_test.cxx. This both shows that you are thinking about how you will eventually test your model, and helps us understand what you expect your model classes and operations to do.

5. Proposal deliverables, evaluation, and submission

For the proposal, you must write the five specified sections:

1. synopsis (in PROPOSAL.md),
2. functional requirements (at least 12, in PROPOSAL.md),
3. open questions (in PROPOSAL.md),
4. model sketch (in src/model.hxx), and
5. example model tests (at least 5, in test/model_test.cxx).

Your grade will be based on:

- the comprehensibility of your synopsis,
- the completeness and preciseness of your functional requirements,
- the relevance of your open questions,
- the adequacy of your model sketch,
- how well your tests demonstrate the meanings of your model operations, and
- and *how seriously you seem to have considered the proposal.*

Homework submission and grading will use the GSC grading server, so you should upload your files on the GSC web site. For the proposal, the files you submit will include PROPOSAL.md, src/model.hxx, and test/model_test.cxx. Submit the proposal as `hw7` on GSC.

Partnerships registered for the proposal will continue for the final code, so choose your partner wisely. You must register either on the GSC website or using the `gsc partner` command **before submitting** your work.

6. Code submission and evaluation

Your final code should be submitted as `hw8` on GSC. You need to upload all files required to run and build your game and tests. This includes your CMakeLists.txt and all files in your src/, test/, and Resources/ directories.[2] Please **do not** submit any files from the .cs211/ directory, the .idea/ directory, nor any build directory (such as cmake-build-debug/).

Note that you have a quota of 20 MB for your entire submission, but you are unlikely to reach this limit until you have a significant amount of audio among your run-time resources.

6.1. Evaluation and evaluation guide

Your proposal is worth 25% of your project grade, and the final code delivery is worth the other 75%. That 75% is further broken down into three components:

style	10%
model tests	20%
functional requirements	70%

Your project TA will assess style on their own, but for the latter two points, they will need your help in the form of the "evaluation guide" described below. You don't need to submit your evaluation guide on GSC—rather, you have 24 hours after your project is due to email your evaluation guide to the same TA who evaluated your project proposal. This is so that you 1) don't have to worry about producing the document while also trying to finish your code, and 2) can easily provide GSC line number references for the final submission.

The evaluation guide must contain the following two components.

Favorite model tests (20%). As in the proposal, we want to see five significant model tests. Choose tests that you think best characterize your design and demonstrate how your model works. For each, provide very a short description of what the test is about, along with a reference to a line number (using the numbering shown on GSC).

Functional requirement hints (70%). For the core of the evaluation, your project TA will attempt to verify that your program meets the functional requirements from your proposal. (This is why you need your TA's approval for any changes to those requirements.) For each requirement, there are three ways that they may attempt this verification:

1. By playing the game and observing the requirement, for full credit.

2. By reading a model test that demonstrates that the game meets the requirement, for full credit. (You are free to reuse a favorite test here.)

3. By looking at the code that implements the requirement, **for 80% credit.**

You must provide a numbered list matching your list of proposed and accepted functional requirements, and for each requirement, specify how the TA should attempt to check it:

1. For validation by playing, you need to provide instructions for how to play the game to a state where the requirement can be observed. If your game has multiple levels, difficulties, or modes, you may find it useful for your `main()` function to take an optional command-line argument to allow the grader to jump to a particular level. Also, if you believe there's a chance that your TA will have trouble validating a particular requirement by playing, you may also provide a test or code reference (options 2 and 3) as backup.

2. For validation by test, you need to provide GSC line numbers for the relevant test or tests, along with sufficient explanation for your TA to understand why the test you tagged is evidence that the functional requirement in question is met.

3. For validation by implementation—the least preferred method—you need to provide GSC line numbers for the relevant implementation code, along with sufficient explanation for your TA to understand why the code you tagged is evidence that the functional requirement in question is met.

A. Example proposal: *Brick Out*

In this section, we give an example proposal for the Brick Out game from Homework 5.

[2] When reconstructing your project for grading, GSC puts source files whose names begin or end with "test" in the test/ directory, other source files in the src/ directory, and files with types it doesn't recognize in the Resources/ directory. So make sure you name any files that need to be in the test/ directory appropriately.

A.1. PROPOSAL.md

The synopsis, functional requirements, and open questions must be in PROPOSAL.md:

```
# Proposal: Brick Out

## Synopsis

### Elements

My game will have three elements:

  - a stationary array of rectangular bricks at the top of the screen,

  - a rectangular paddle at the bottom that moves horizontally and is controlled by the user, and

  - a circular ball that bounces in between, destroying any bricks it collides with.

### Goal

The player's goal is to destroy the bricks without allowing the ball to reach the bottom of the screen.

### Game play

The ball starts out stuck to the top of the paddle, and the player starts the game by launching it with a mouse click or key press. Then the ball bounces between the paddle, the bricks (destroying any it hits), and the top and sides of the screen. If it reaches the bottom of the screen then it dies and returns to its initial stuck-to-the-paddle state, from which it can be launched again.

## Functional requirements

1. The bricks are placed in a grid at the top of the screen.

2. The paddle's x coordinate follows the mouse, while its y coordinate is fixed.

3. In the dead state (the ball's initial state) the ball sticks to the paddle.

4. The player can release the ball, transitioning it from dead to live state, by pressing the space bar or clicking the mouse.

5. When the ball is released, it travels upward from the paddle with some initial velocity.

6. If the ball strikes the top or side of the screen, it bounces off.

7. If the ball strikes a brick, it destroys the brick and bounces off *weirdly* (TBD).

8. If the ball strikes the paddle, it bounces off with a small, random *boost* to its velocity (TBD).

9. If the ball reaches the bottom of the screen, it transitions back to the dead state (and nothing else changes).

## Open questions

  - How should bouncing off of bricks be weird?

  - How can the random boost be generated? How can it be tested?

  - What dimensions and velocities make the game work best?
```

A.2. src/model.hxx

The model sketch must be in src/model.hxx:

```
//
// Model constants:
//

const int ball_radius;
const ge211::Dimensions paddle_dims;
const ge211::Dimensions brick_dims;

//
// Model classes:
//

// The position of one brick, and
// whether it's still there.
struct Brick
{
    // The top-left corner
    ge211::Position top_left;

    // Whether the brick still exists
    bool live;
}

// The whole state of the game.
class Model
{
```

```
    //
    // PRIVATE DATA MEMBERS
    //

    // The top left of the paddle
    ge211::Position paddle_;

    // The center of the ball
    ge211::Position ball_;

    // The velocity of the ball
    // (0, 0 means dead).
    ge211::Dimensions vel_;

    // The bricks
    std::vector<Brick> bricks_;

public:
    //
    // PUBLIC FUNCTIONS
    //

    // Returns the ball's position.
    ge211::Position get_ball() const;

    // Returns the paddle's position.
    ge211::Position get_paddle() const;

    // Views the states of all the
    // bricks.
    std::vector<Brick> const&
    get_bricks() const;

    // Updates the model state for
    // one time step.
    void update();

    // Moves the x coordinate of the
    // paddle to 'x'.
    void move_paddle(int x);

    // Launches the ball if it's dead.
    void launch_ball();

private:
    //
    // POSSIBLE HELPER FUNCTIONS
    //

    // Determines whether the ball
    // hits the given object.
    bool ball_hits_top_() const;
    bool ball_hits_left_() const;
    bool ball_hits_right_() const;
    bool ball_hits_bottom_() const;
    bool ball_hits_paddle_() const;

    // Returns a pointer to a hit
    // brick, or nullptr if none.
    Brick* find_hit_brick_() const;

    // Removes the indicated brick.
    void destroy_hit_brick_(Brick&);

    // Reflects the ball from the given
    // (kind of) object.
    void reflect_ball_top_();
    void reflect_ball_sides_();
    void reflect_ball_paddle_();
    void reflect_ball_brick_();

    // Returns the ball to dead state.
    void reset_ball_();

    // Test access[3]
    friend struct Test_access;
};
```

A.3. test/model_test.cxx

The example model tests must be in test/model_-test.cxx:

```
using ge211;

TEST_CASE("initial bricks")
{
    Model m;

    std::vector<Brick> expected {
        {{100, 100}, true},
        {{250, 100}, true},
        {{400, 100}, true},
        {{550, 100}, true},
        {{100, 175}, true},
        {{250, 175}, true},
        {{400, 175}, true},
        {{550, 175}, true},
        {{100, 250}, true},
        {{250, 250}, true},
        {{400, 250}, true},
        {{550, 250}, true}
    };
```

[3] A friend declaration allows members of the friend type (in this case `Test_access`), to access private members of the declaring type (in this case `Model`). In our tests, we define a struct `Test_access` and use it to get at our `Model` class's private state.

```
    CHECK( m.get_bricks() ==
               expected );
}

struct Test_access // [4]
{
    Model& model;

    Position& paddle()
    { return model.paddle_; }

    Position& ball()
    { return model.ball_; }

    Dimensions& vel()
    { return model.vel_; }

    std::vector<Brick>& bricks()
    { return model.bricks_; }
};

TEST_CASE("left side collision")
{
    Model m;
    Test_access t{m};

    t.bricks().clear();
    t.ball() = { 13, 200 };
    t.vel() = { -10, 3 };

    m.update();

    CHECK( t.vel() ==
               Dimensions{10, 3} );
    CHECK( t.ball() ==
               Position{23, 203} );
}

///
/// NEED AT LEAST THREE MORE TEST
/// CASES FOR PROPOSAL
///
```

[4] Because the `Model` class declares that `Test_access` is its friend, members of `Test_access` can access private members of `Model`. It returns them by reference, allowing the tests below to inspect or modify the model's private state. You don't need to write a test access friend like this for your proposal, but you may find the technique useful later.