The University of Melbourne
Department of Computer Science and Software Engineering
SWEN20003 Object Oriented Software Development

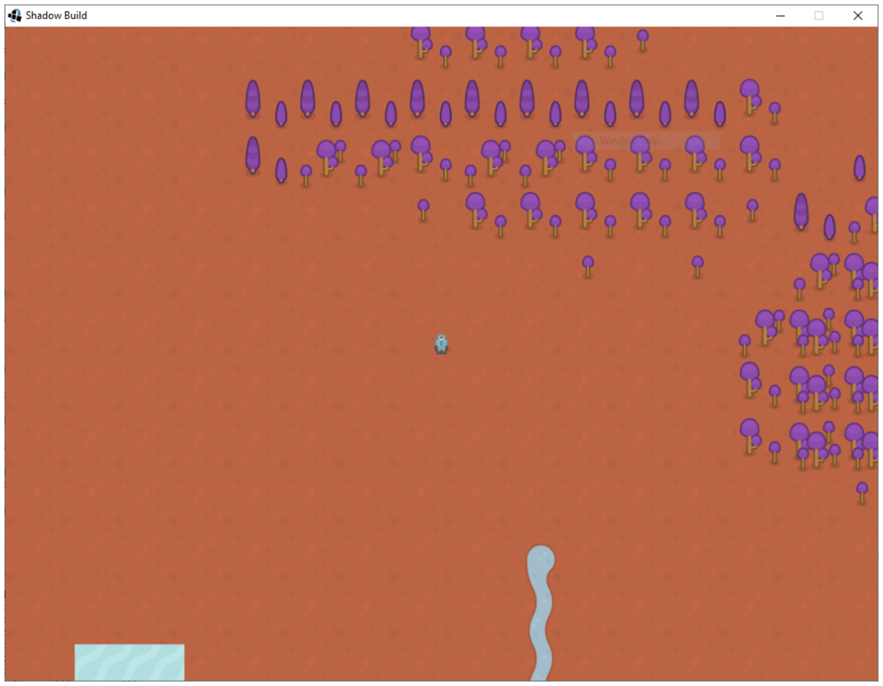
Project 1, Semester 1, 2019

Released: Friday 5th of April, 9:00am
Due: Friday 19th of April, 7:00pm

Overview

Welcome to the first project for SWEN20003! We will be using the Slick library for Java, which can be found at `http://slick.ninjacave.com/`. Week 5's workshop introduces Slick, so refer to the tutorial sheet if you have trouble getting started. This is an **individual** project. You may discuss it with other students, but all of the implementation must be your own work.

Project 2 will extend and build on Project 1 to create a complete game, so it's important to write your submission so that it can be easily extended. Below is a screenshot of the game after completing Project 1.



This early version of **Shadow Build** includes a large tile-based map to explore, and a scouting unit that can be moved around the map. Only a small portion of the map surrounding the scout is shown at any given time.

The game

Below I will outline the different game elements you need to implement.

The tile map

The game's background is **tile-based**. Each tile is 64 pixels wide and 64 pixels high, and the background is built from a collection of these tiles.

Slick has some built-in objects and methods to handle this. You should use the documentation for the `TiledMap` class to understand how to use the `assets/main.tmx` file to create and draw the background map.

Tiles can have **properties**. To find the property associated with a given coordinate, you should first find the **tile ID** corresponding to that location. Tiled maps can have several layers, but the map you will be using has only one layer (index 0). Once you have found the ID, you can find the corresponding property.

The tiled map you will be using has only one property: `solid`. This can have the value `true`, indicating that the tile should not be moved through, or `false`, indicating the tile can be moved through.

The player

The main character of the game is the **player**, represented by the image

For now, the player simply moves around the screen. When the user right clicks on the screen, the player should move in a straight line towards the location of the click at a rate of 0.25 pixels per millisecond. When the player is within 0.25 pixels of their destination, they should stop moving.

If the player would move into a solid tile, the player should instead stop moving.

Some hints for calculating movement

You will need to use some trigonometry to calculate the correct movement in each frame. We recommend using the `Math.atan2` function, which will give you the angle from (0,0) to the position described by its arguments, respecting quadrants correctly.

The camera

The tiled map is far too large to fit on the screen all at once. You should therefore only show a subset of it, defined to be 1024x768 pixels. This should be handled by an abstract "camera", which stores which part of the map is currently in view. The camera should be able to move so that the player stays in the middle of the screen, unless the player moves close to an edge of the map.

If the player moves close to an edge, the camera should stop moving once there is no more map left to view. In this way, the player should never be able to move off the map.

The "how" of this functionality is up to you. We recommend using a class to store the relevant data, and calculate transformations from **screen** coordinates to **world** (map) coordinates.

Slick concepts

This section aims to clarify some common mistaken assumptions students make about the Slick engine, as well as to outline some important concepts.

A Slick game works according to the **game loop**. Dozens of times each second, a **frame** of the game is processed. In a frame, the following happens:

1. The game is **updated** by calling the `update()` method. The number of milliseconds since the last frame is passed as the argument `delta`; this value can be used to make sure objects move at the same speed no matter how fast the game is running.
2. The game is **rendered** by calling the `render()` method. To do this, the entire screen is cleared so it displays only black; that way, no images from the previous frame can be seen. Images can **only** be drawn inside this method.

The number of frames that are processed each second is called the **frames per second**, or FPS. Different computers will likely have a different value for FPS. Therefore, it is important to make sure your `update()` method works the same way regardless of this value.

In Slick, positions are given as pairs of (x, y) coordinates called **pixels**. Note that $(0, 0)$ is the top-left of the window; the y direction is therefore the opposite of what you may be used to from mathematics studies. Keep in mind that while only integer locations can be rendered, it may make sense to store positions as floating-point values. For the purposes of this document, positions are given **from the centre of the image**.

A terminology note: an image with a defined position in the world that can move around or perform some other action is often called a **sprite**.

Your code

The skeleton we have provided has these three classes.

- **App** – The outer layer of the game. Inherits from Slick's `BasicGame` class. Starts up the game, handles the `update` and `render` methods, and passes them along to `World`.
- **World** – Represents everything in the game, including the background and all sprites. For now, this is just the tiled map and the player.
- **Camera** – Represents the subset of the world that can actually be viewed. It should be able to follow a particular sprite, so that different sections of the world can be viewed at different times. This class may need to do some calculations on behalf of other classes.

You will likely find that creating more classes will make the project easier. This decision is up to you as a software engineer! Make sure your code follows object-oriented principles like abstraction and encapsulation.

Implementation checklist

This project may seem daunting. As there are a lot of things you need to implement, we have provided a checklist, ordered roughly in the order we think you should implement them in:

1. Draw the tiled map on the screen
2. Draw the player on the screen
3. Allow the player to move around the screen
4. Prevent the player from moving through solid tiles
5. Allow the camera to follow the player

The supplied package

You will be given a package, `oosd-project1-package.zip`, which contains all of the graphics and other files you need to build the game. You can use these in any way you want. Here is a brief summary of its contents:

- `src/` – The supplied source code.
  - `App.java` – A complete class which starts up the game and handles input and rendering.
  - `World.java` – A file with stub methods for you to fill in.
  - `Camera.java` – A file with stub methods for you to fill in.
  - `assets/` – The images for the game.
  - `credit.txt` – The source for each of the images included.
  - `main.tmx` – The tiled map data.
  - `planet.tsx` – The image data for the tiled map. You do not need to explicitly use this file.
  - `scout.png` – The image for the player character.

Submission and marking

Technical requirements

- The program must be written in the Java programming language.
- The program must not depend upon any libraries other than the Java standard library and the Slick library.

- The program must compile fully without errors.

Submission will take place through the LMS. Please zip your project folder in its entirety, and submit this `.zip` file. **Do not submit a .rar, .7z, .tar.gz, or any other type of compressed folder.** Especially do not submit one of these files that has simply been renamed to have a `.zip` extension, as then we will be unable to open your file.[1]

Ensure all your code is contained in this folder. We will provide a link on the LMS to the appropriate submission page closer to the due date.

Good Coding Style

Good coding style is a contentious issue; however, we will be marking your code based on the following criteria:

- You should *not* go back and comment your code after the fact. You should be commenting as you go.
- You should be taking care to ensure proper use of visibility modifiers. Unless you have a very good reason for it, all instance variables should be private.
- Any constant should be defined as a static final variable. Don't use magic numbers!
- Think about whether your code makes assumptions about things that are likely to change for Project 2.
- Make sure each class makes sense as a cohesive whole. A class should contain all of the data and methods relevant to its purpose.

Extensions and late submissions

If you need an extension for the project, please email Eleanor at `mcmurtrye@unimelb.edu.au` explaining your situation with some supporting documentation (medical certificate, academic adjustment plan, wedding invitation). If an extension has been granted, you may submit via the LMS as usual; please do however email Eleanor once you have submitted your project.

The project is due at **7:00pm sharp**. Any submissions received past this time (from 7:01pm onwards) will be considered late unless an extension has been granted. There will be no exceptions. There is a penalty of 1 mark for a late project, plus an additional 1 mark per 24 hours. If you submit late, you **must** email Eleanor with your student ID so that we can ensure your late submission is marked correctly.

Marks

Project 1 is worth **8** marks out of the total 100 for the subject.

- Features implemented correctly – **4 marks**
  - Map is drawn correctly: **1 mark**

[1]This may sound ridiculous, but it has happened several times in the past!

  - Player moves correctly: **1 mark**
  - Camera follows player: **1 mark**
  - Player cannot move through solid tiles: **1 mark**
- Code (coding style, documentation, good object-oriented principles) – **4 marks**
  - Delegation – breaking the code down into appropriate classes (**1 mark**)
  - Use of methods – avoiding repeated code and overly complex methods (**1 mark**)
  - Cohesion – classes are complete units that contain all their data (**1 mark**)
  - Code style – visibility modifiers, consistent indentation, lack of magic numbers, commenting (**1 mark**)