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

Project 1, 2014

Friday 29th of August, 5:00 PM
Due: Friday 12th of September, 5:00 PM
Demonstration: Workshop Week 8 (Starting Monday 15th of September)

Overview

In this project, you will create a basic graphical game engine, in the Java programming language. You will implement the graphical user interface with the Slick library[1], as introduced in Workshop 4.

This is an individual project. You may discuss it with other students, but all of the implementation must be your own work. You may use any platform and tools you wish to develop the game, but we recommend using the Eclipse IDE for Java development as this is what we will support in class.

You will be required to briefly demonstrate the game engine in a workshop. In Project 2, you will continue working on the game engine, and build a working game.

The purpose of this project is to:

- Give you experience working with an object-oriented programming language (Java);
- Introduce simple game programming concepts (2D graphics, input, simple calculations); and
- Give you experience working with a simple game API (Slick).

Figure 1 shows a screenshot from the game after completing Project 1. The game consists of navigating a character through a maze of trees, mountains and water. In this image, you can see the player in a small village, with houses.

Note that this is a "tile-based" game; the map is arranged in a grid of squares (note the blocky shapes of the stone and arrangement of the trees). This makes the game easier to program, and was a very common technique in games of the '90s.

The game has a single character, called the "player". The user controls the game with the arrow keys on the keyboard, moving the player character around the map. The player can walk on grass, stone, wood, etc., but cannot walk on trees, houses, water, etc[2].

There is no "goal" of the game, and no "enemies", at this stage – merely exploration. These will be introduced in Project 2, in which you will be required to program the complete game.

[1] The Slick library can be found at `http://slick.ninjacave.com`. See Workshop 4 for instructions on Slick.
[2] The exact terrain types which the player can/can't walk on is encoded in the map file, which is explained later.

The rest of this section details precisely how the game works. You must implement all of the features described here, unless they are marked "**optional**".

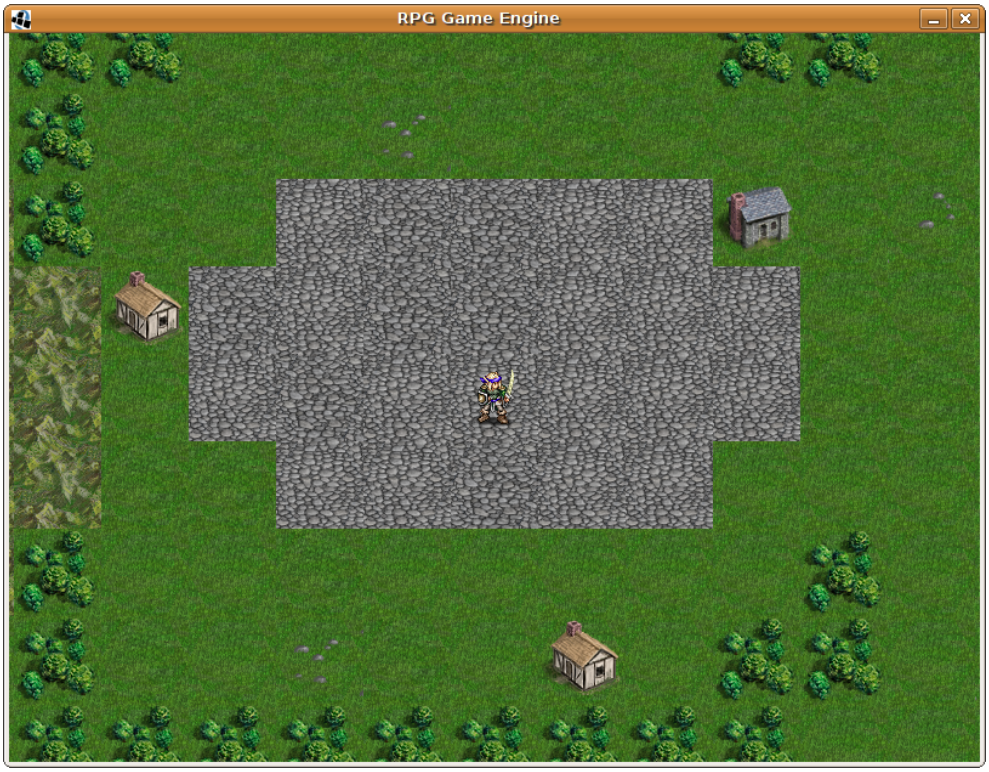

Figure 1: The opening screen of the game.

The game map

In this game, the "world" is a two-dimensional grid of *tiles*. The player is able to move freely around the world to explore the entire world (but of course, the player cannot walk on trees, mountains, water, and other types of terrain).

The *game map* is a large 2D array (96×96) of *tiles*, representing the world the player lives in. Each tile has a particular *terrain type*, such as grass, stone, mountains, water and fire. Aside from its graphical appearance, there is only one physical difference between terrain types: some tiles allow movement (such as grass and wood) while others block movement (such as mountains, water, fire and trees). Each tile is 72×72 pixels in size.

The supplied map file, `map.tmx`, contains the tile layout information for the whole game. Slick comes with a handy class called `TiledMap` which can read `.tmx` files (see "implementation tips" for instructions).

The player

The only character in the "game" at this stage is the player. The player controls a small character with the keyboard, and is able to move him or her around the map. The player's position in the world is stored as an (x, y) coordinate in pixels[3]. This means that the player can stand at any point on the map, rather than being constrained to the tiles of the map grid. All measurements are given in pixels.

The player should start at location $(756, 684)$. This should place him/her in the middle of the village, as shown in Figure 1.

Internally, the player has no "size"; it is just a single point. When displaying the player, the image should be centred around the player's position.

Gameplay

The game takes place in *frames*. Much like a movie, many frames occur per second, so the animation is smooth. Each frame:

1. If certain keyboard keys are being pressed, the player moves by a small amount in that direction and the players position in the world is updated.
2. The "camera" is updated so that it is centred around the player's position. *This is important!* the camera should be reliant on knowing the players position, not the other way around.
3. The entire screen is "rendered", so the display on-screen reflects the new state of the world.

Display

This is a graphical game, so you will use the Slick library to draw the graphics onto the screen. We have supplied all of the graphics you will need, as PNG files (see "the supplied package").

The game's main window is 800×600 pixels. The game map is much bigger (6912×6912 pixels), so you can't see all of it at once. Instead, the game should only render a portion of the map at a time – such that the "camera" is centred around the player. This means that as the player walks around the world, the camera should follow. You will find that you will need to render 13 tiles wide and 10 tiles high, to fill up the screen.

You should *not* render the entire map, or the game may run very slowly (and we will deduct marks). Instead, you should calculate which 13×10 tiles are required to fill up the screen, and render those.

Optional feature: In the supplied image, `player.png`, the character is facing to the right. Make it so if the player moves left, (s)he faces left instead, and if the player moves right, (s)he faces right again.

[3] A pixel is a single tiny dot on the screen. You should use the `double` data type for the pixel coordinates. Even though you won't be able to visually distinguish between pixel distances less than 1.0, it is still important to store fractions of a pixel to avoid accumulating rounding errors across frames.

Controls

The game is controlled entirely using the arrow keys on the keyboard. The **left**, **right**, **up** and **down** keys move the player. Each frame, the player moves by a tiny amount in the direction of the keys being pressed, if any. It is possible to move diagonally by holding down two keys at a time (for example, move north-east by holding the **up** and **right** keys).

The player moves at a rate of a quarter of a pixel per millisecond, in each direction. For example, if the player is moving east, then every millisecond, (s)he moves 0.25 pixels to the right. If the player is moving north-east, then every millisecond, (s)he moves 0.25 pixels up and 0.25 pixels right in the world. (Note that this enables the player to move slightly faster diagonally).

Optional feature: Allow click to move style controls such that the player can be controlled by clicking on a point in the map with the mouse and the player will move towards it. This feature will require pathfinding to work, and should be considered only if you have an abundance of time and want a challenge.

Terrain blocking

Some types of terrain (such as water) should stop the player from moving, while other types should allow the player to walk. The map file itself stores whether or not each tile should block the player's movement[4]. See "implementation tips" for details on determining whether a tile should block.

If the player attempts to walk on terrain which "blocks", the player should remain in his/her current location (this will make walls feel "sticky").

Optional feature: Allow the player to "slide" along walls when moving diagonally. For example, if the player is trying to move south-east, and there are mountains to the east, move the player south, rather than not moving at all. (This may feel more natural than the "sticky" walls approach suggested above.)

Your code

Your code should consist of at least these four classes:

- **RPG** – 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 the entire game world, including the map.
- **Player** – Represents the player character.
- **Camera** – Represents the viewport, should follow the player.

[4]The `.tmx` file format allows "properties" to be associated with tiles. We associate a property `"block"` with a value of `"1"` for each tile type which should block the player. You don't need to be concerned with the details of this file format.

You will be supplied with complete code for `RPG`, and partial code for `World`. You may modify this code however you wish; it is provided merely as a guide. Your implementation of the `World`, `Player` and `Camera` classes should follow proper object-oriented design practices, such as encapsulation.

You may find that there are better ways to design the classes than just restricting yourself to these three classes and we are open to this (and in-fact encourage creative design thinking), just make sure you discuss these with your tutor before going ahead.

Implementation tips

This section presents some tips on implementing the game engine described above. You may ignore the advice here, as long as your game exhibits the required features.

The game map

You should use the `TiledMap` class to read the supplied map file, `map.tmx`. The following constructor of `TiledMap` is appropriate:

```
TiledMap(String ref, String tileSetsLocation)
```

The first argument is the location of the `map.tmx` file; the second is the location of the `assets` directory in which the tileset information used by the map.tmx file is held (when Slick opens the map.tmx file, it will look in this directory to load the accompanying tileset information - `tileset.tsx` and `tiles.png`. These files contain information about the different tiles). There are more hints about using `TiledMap` later in this section.

Display

The `TiledMap.render` method (in Slick) can be used to draw a portion of the map to the screen. You should use the version with six arguments:

```
void render(int x, int y, int sx, int sy, int width, int height)
```

Hint: You can pass negative numbers for the `x` and `y` parameters to `TiledMap.render`, so the top-left tile is rendered a little above and to-the-left-of the top-left corner of the screen.

You will also need to display the player. You can use the `Image.draw` method (in Slick) to render an image onto the background at a particular location. You should use the version with two arguments:

```
void draw(float x, float y)
```

If you choose to implement the optional "facing left" feature, you can use the `Image.getFlippedCopy` method to create a version of the player's image facing left.

Controls

The `delta` argument passed to the `RPG.update` method by Slick indicates the number of milliseconds passed since the last frame. You should use this to determine the number of pixels to move the player each frame. The distance moved should depend on this delta value, so that your game can be independent of the frame rate. Otherwise your game will move slower on slow computers, but faster on faster computers (and you will loose marks for this).

Terrain blocking

Finding out whether any given tile should block the player is a two-step process:

1. First, call the `TiledMap.getTileId` method. This gives an integer representing the terrain type (eg. grass = 10, tree = 7, house = 12).
2. Then, call the `TiledMap.getTileProperty` method, looking up the property name `"block"` for the given tile ID, with the default value of `"0"`. A result of `"1"` means block the player; `"0"` means allow the player to walk freely.

The above instructions use the following methods of `TiledMap`:

```
int getTileId(int x, int y, int layerIndex)
String getTileProperty(int tileID, String propertyName, String def)
```

Detailed documentation for the classes used in this project, such as TiledMap and Image can be found at `http://slick.ninjacave.com` and may be useful when trying to figure out how these classes and their methods work.

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. Loading the map.
2. Defining the `Camera` class
3. Rendering the map (center the `Camera` around position $(756, 684)$, for now).
4. Defining the `Player` class.
5. Loading the player graphics.
6. Rendering the player.
7. Movement of the player.
8. Centering the `Camera` around the player.
9. Terrain blocking.

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.
  - `RPG.java` – A complete class which starts up the game and handles input and rendering.
  - `World.java` – A file with stubs for you to fill in.
  - `Camera.java` – A file with stubs for you to fill in.
- `assets/` – The game graphics and map files.
  - `map.tmx` – The map file. You should pass this as input to Slick's `TiledMap` constructor.
  - `tileset.tsx` – The tile-set file; automatically included by `map.tmx`.
  - `tiles.png` – The terrain graphics; automatically included by `tileset.tsx`.
  - `units/` – Contains the characters' image files.
    - `player.png` – The image file to use for the player.

Legal notice

The graphics included with the package (`tiles.png`, `units/*`) are derived from copyrighted works from the game *The Battle for Wesnoth*, licensed under the GNU General Public License, version 2 or later. You may redistribute them, but only if you agree to the license terms.

For more information, see `http://www.wesnoth.org/wiki/Wesnoth:Copyrights`.

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 graphics library we have provided.
- The program must compile and run in Eclipse on the Windows machines in the labs. (This is a practical requirement, as you will be giving a demo in this room; you may develop the program with any tools you wish, as long as you make sure it runs in this environment.)

Submission

Submission will take place through the LMS. You will have to zip your project folder and submit this zip file in its entirety. 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 often a difficult objective to decide. More often than not, what is considered good style is dependent on the company you work for, the aims of the project and the budget involved. For the purposes of this project we are looking for the following:

- 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, secure use of visibility modifiers. Unless you have a very good reason for it, all instance variables should be private. You should consider proper information hiding practice at all times.
- Any constant should be defined as a static final variable, you should not leave any constants in the code base, it makes it difficult to change paths when necessary and contributes to hard to manage code.
- Think about the modifiability of your code, remember that the second portion of the project has not been released yet, but it should be clear that there will be other characters and you should consider how you can generalize this functionality to make things easier for yourself in the future.
- Think about code delegation, you should ensure that each class is responsible for its own functionality and that each class encloses its own functionality fully. For example, all player update should be contained in one `update()` method with the appropriate arguments.

Demonstration

You will be required to demonstrate the game engine for your tutor in your workshop in the week of the Workshop Week 8 (Starting Monday 15th of September). Your tutor will check out the submitted version of your code (i.e., the checked-in version of the code as of the deadline), then ask you to the front of the room for a few minutes. You will be asked to demonstrate various features of the engine working correctly. Not attending the demonstration will incur a 1 mark penalty.

Late submissions

There is a penalty of 1 mark per day for late submissions without a medical certificate. Late submissions will be handled by Sarah Erfani, a tutor for the subject. If you submit late, you *must* email Sarah at `sarah.erfani@unimleb.edu.au` with your login name and the zip file you wish

for us to consider when marking (if you don't, we will simply mark the latest version you have submitted by the deadline).

Marks

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

- Features implemented correctly – **4 marks**
  - Display of terrain and player – **1 marks**
  - Player movement and camera – **2 marks**
  - Terrain blocking – **1 marks**
- Code (coding style, documentation, good object-oriented principles) – **4 marks**

Acknowledgement

This game was designed by Matt Giuca and updated by Mathew Blair.