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

Project 2, 2016

Given: 9th of September: 5:00 p.m.
Project 2a Due: 23rd of September: 11:59 p.m.
Project 2b Due: 14th of October: 11:59 p.m.
Demonstration: Workshops Week 12.

Overview

In this project, you will create a graphical role-playing game, *Shadow Quest*, in the Java programming language, continuing on from your work on Project 1[1]. You may use any platform and tools you wish to develop the game, but we recommend the Eclipse IDE for Java development. This is an individual project. You may discuss it with other students, but all of the implementation must be your own work and we will be strictly checking this has been adhered to. Like Project 1, you will be required to briefly demonstrate the game in a workshop.

You are not required to design any aspect of the game itself – this specification should provide all information about how the game works. However, you will be required to design the classes before you implement the game.

There are two parts to the project, with different submission dates:

The first task, **Project 2a**, requires that you produce a class design, using a diagramming tool of your choice, outlining how you plan to implement the game. This should consist of a UML class diagram, showing all of the classes you plan to implement, the relationships (inheritance and association) between them, and their attributes and primary public methods. If you like, you may show the class members in a separate diagram to the hierarchy diagram, but you **must** use proper UML notation.

The second task, **Project 2b**, is to complete the implementation of the game, as described in the manual which follows. You do not have to follow your class design – it is merely there to encourage you to think about the object-oriented design before you start implementing, however if you do change it, you will be required to provide a new class diagram that shows the new relations and a short document explaining the rationale behind your changes.

The purpose of this project is to:

- Give you experience designing object-oriented software with simple class diagrams,
- Give you experience working with an object-oriented programming language (Java),
- Introduce simple game programming concepts (user interfaces, collisions, simple AI),

[1]Note: We will supply you with a complete working solution to Project 1; you may use part or all of it. It will be available on the LMS on Friday 16th of September, to allow for late submissions to Project 1.

Shadow Quest

Game Manual

It has been seven days since my father, King Lathran, came down with a terrible illness. To date, the best efforts of our court physicians and magi have been in vain. The Elven priestess, Lady Elvira suspects this is no ordinary sickness, but a magical curse.

If she is correct, then there is only one way to cure the king, and restore order to the land: the Elixir of Life, a potion rumoured to have been created centuries ago, with the power to heal any wound, and undo any curse.

Unfortunately, the potion is found in the Land of Shadows to the east, held by the Necromancer, Draelic, who commands a vile army of the undead. He will not give it up easily. I only hope that someone shall come forth, with the will to retrieve the elixir before all is lost...

— Prince Aldric

Game overview

Shadow Quest is a role-playing game where one player battles against computer-controlled enemies on a quest to retrieve the Elixir of Life. There are two areas in the game:

- **The town**, where friendly characters live. The player can walk around and talk to the villagers. Elvira is an Elven priestess, who will heal the player if he or she is wounded.
- **The wilderness**, where evil creatures live. The player can freely explore these areas, fight monsters, and collect useful items.

The goal of the game is to defeat the Necromancer Draelic, retrieve the Elixir of Life, and return it to Prince Aldric in the town.

The game map

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

Units and stats

A *unit* is our term for a game character or person. There are three general unit types in the game:

- **The player**. Controlled by you; able to talk to villagers and fight monsters.
- **Villagers**. Friendly units; do nothing until the player talks to them.

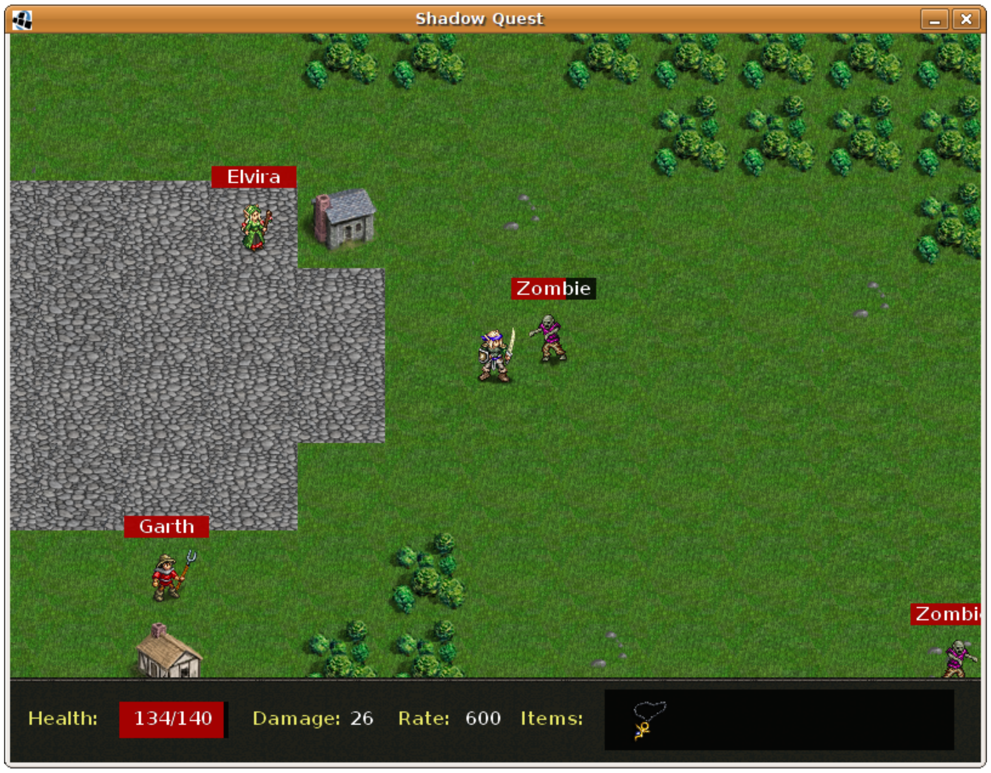

Figure 1: A battle with a Zombie

- **Monsters**. Enemy units; either passive or aggressive. Passive monsters wander around the map, and run away when attacked. Aggressive monsters try to kill the player.

Each unit's position in the world is stored as an (x, y) coordinate in pixels. This means that units 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.

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

All units have a "**HP**" (short for "hit points") field which is an integer value that determines their current level of health. Units who are wounded in battle lose HP, and units who are "healed" gain HP. Units who lose all of their HP are "dead". (See "Combat" later).

A *stat* (or attribute) is an integer value which determines how powerful a unit is in some way. A unit's "stats" never change (except in the case of the player picking up items; see "Items" later).

- **Max-HP** – (Short for "maximum hit points"). This stat determines the maximum amount of HP the unit has when fully healed.
- **Damage** – This stat determines the maximum amount of damage the unit can do when attacking (see "Combat").
- **Cooldown** – This stat determines the minimum length of time the character has to wait between attacks, in milliseconds (see "Combat").

Note that units' health is often shown as a percentage. This can be calculated as $\frac{\text{HP}}{\text{Max-HP}}$. For example, if the player's HP is 15 and their Max-HP is 20, that player's "health" is at 75%.

As shown in Figure 1, all units *except the player* have a health/name bar floating above their heads. This bar shows the unit's health percentage as a red bar on a black background, overlayed with the unit's name in white text.

See the data file `data/units.txt` for details on the stats and locations of the units.

Gameplay

This is a *real-time* game, meaning that events happen continuously, even if you do not press any keys. For example, monsters may move or attack the player even if the player stands still.

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

1. All game units have a chance to move. Monsters move automatically towards or away from the player if they are nearby. The player moves if the keyboard is being pressed.
2. The "camera" is updated so that it is centered around the player's position.
3. The entire screen is "rendered", so the display on-screen reflects the new state of the world.

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 one quarter of a pixel per millisecond, in each direction.

You can attack a monster simply by moving close to it (within 50 pixels) and holding **A**. Similarly, you can talk to a villager by moving close (within 50 pixels) and pressing **T**.

Villagers and dialogue

In the town you will find a number of friendly villagers who will help you out. When the player moves within 50 pixels of a villager, some action *may* occur (depending on the villager). There are three villagers:

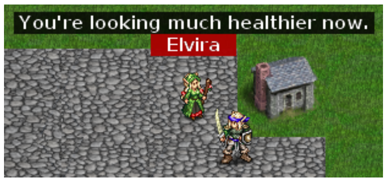

Figure 2: A dialogue bar

- **Prince Aldric** – When the player interacts with Prince Aldric, nothing happens, unless the player is carrying the Elixir, in which case the Elixir is removed from the player's inventory.

- **Elvira** – When the player interacts with Elvira, they are healed fully (their HP is brought up to their Max-HP).
- **Garth** – While he appears to be a simple farmer, Garth knows many things and will help you to find powerful items.

The game employs a simple dialogue system. Units may display a single line of dialogue above their heads (as shown in Figure 2) for 4 seconds. See the data file `data/dialogue.txt` for the exact dialogue to be displayed.

Monsters

There are a number of species of creatures which roam the wilderness, some are passive and harmless, while others are aggressive and dangerous. If you aren't careful, you'll find them ending your quest early.

Monsters have a very simple AI (artificial intelligence), which will differ depending on whether the monster is passive or aggressive.

- **Passive Monsters**

  A passive monster will wander around the map randomly by default. Passive monsters are slow moving, and move at one fifth of a pixel per millisecond. This means that players can always move a little bit faster than passive monsters to chase them down.

  Every 3 seconds the monster should pick one of 9 directions to move in (Up, Down, Left, Right, Up-Right, Down-Right, Down-Left, Up-Left, or no direction) and move in that direction. Thus a monster will change it's direction of movement once every 3 seconds. Monsters proceed to move in this way until a player attacks it. (See "Combat" later).

  Passive monsters do not attack players. They simply wander the world. However, if a passive monster is attacked by a player, it will try and run away from the player (at the same speed of one fifth of a pixel per millisecond) as dictated by Algorithm 1[2]. If 5 seconds pass, and the passive monster has not been attacked by the player again, the passive monster will know it's safe, and will resume wandering the map. Just like the player, monsters can't move through trees, water, *etc.*

- **Aggressive Monsters**

  An aggressive monster will do nothing by default. If the player is within 150 pixels of the monster, it will move towards the player. If the player is within 50 pixels of the monster, it will not move, and instead attack the player.

  Aggressive monsters move at one quarter of a pixel per millisecond. While the player can move faster diagonally, monsters move at the same speed in all directions, as calculated by Algorithm 1[2].

[2]You are not required to implement this exact algorithm; as long as the monster moves towards (or away from) the player somehow. Following this algorithm means monsters make the most direct movement to (or away from) the player.

Algorithm 1 Monster's movement calculation

```
1: Let $dist_x$, $dist_y$ be the $x$ and $y$ distance from the monster to the player, in pixels.
2: Let $amount$ be the total distance the monster may move this frame, in pixels.
3: $dist_{total} = \sqrt{{dist_x}^2 + {dist_y}^2}$
4: $d_x = \frac{dist_x}{dist_{total}} \times amount$
5: $d_y = \frac{dist_y}{dist_{total}} \times amount$
6: $d_x$ and $d_y$ are the $x$ and $y$ distance to move this frame, in pixels.
```

- **Giant Bats, Passive** – Large looking and threatening, but surprisingly harmless, especially during the day.
- **Zombies, Aggressive** – Slow-moving but dangerous, these undead beings have returned from the grave to feast on your brains.
- **Bandits, Aggressive** – These outlaws have taken to robbing passers-by for gold. You'll have to defend yourself against their vicious attacks.
- **Skeletons, Aggressive** – Servants of Draelic, these fierce warriors will stop at nothing to slice you up.
- **Draelic, Aggressive** – Lord of the Land of Shadows, the Necromancer guards the elixir with his powerful magic.

Combat

Combat occurs between the player and a monster. If the player tries to move, and the destination coordinate is within 50 pixels of a monster, the player will be able to then attack. The player continues moving as normal, but if the user presses **A** the player attacks all monsters within range. The same applies for aggressive monsters attacking players.

To make things more interesting, attacks do a random amount of damage. On each attack, the attacker randomly generates an integer between 0 and the attacker's **Max-Damage** stat, inclusive, which is the amount of damage the attack will do. This integer is subtracted from the target's HP.

If units could attack *every* frame, battles would be over very quickly. To slow things down, units have a "cooldown timer". The cooldown timer is measured in milliseconds, and begins at 0ms. Units can only attack when the cooldown timer is 0ms. When a unit makes an attack, the cooldown timer is set to the unit's **Cooldown** stat. Every frame, the cooldown timer is lowered by the number of milliseconds since the last frame, during this time the unit may not attack. When the "cooldown timer" reaches 0ms, the unit may attack again. This means the unit can only attack once every **Cooldown** milliseconds.

If a unit's HP reaches 0, the unit is dead. Monsters who die are removed from the game permanently (they just disappear).

If the player dies, the game does not end (this would be too cruel). Instead, the player is teleported back to position $(738, 549)$, with full health.

Items and inventory

There are a handful of special *items* scattered throughout the game. Like units, each item has an (x, y) coordinate in pixels.

The player has an *inventory*, which is simply the collection of all items the player has collected. The inventory appears as a bar along the bottom of the screen, showing which items have been collected.

The player can pick up any item by moving within 50 pixels of it's position. When picked up, an item will be removed from the world, and added to the player's inventory.

Each item has a special effect on the player:

- **Amulet of Vitality** – A magic necklace. Increases the player's max health.
- **Sword of Strength** – A flaming sword. Increases the player's max damage.
- **Tome of Agility** – An enchanted spellbook. Increases the player's attack speed (reduces the cooldown).
- **Elixir of Life** – No special effect; the goal of the game. Find it and take it back to Prince Aldric to win the game.

See the data file `data/items.txt` for details on the stats and locations of the items.

The status panel

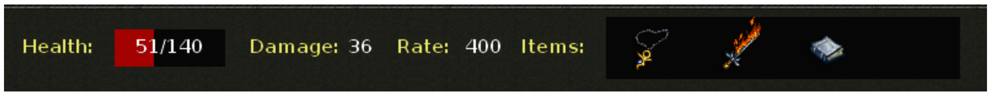

Figure 3: The status panel

The bar at the bottom of the screen (pictured in Figure 3) is known as the *status panel*. It shows the player's current health, stats and items. There are four sections:

- Health: Shows the player's health percentage as a red bar on a black background, overlayed with the numeric form, "**HP/Max-HP**".
- Damage: Shows the player's **Damage**.
- Rate: Shows the player's **Cooldown**.
- Items: Shows the image for each item in the player's inventory (except keys).

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. Next to each item is the number of marks that particular feature is worth, for a total of 6 .

- Villagers, just stand around. All the villagers in the game. (0.5)
- Monsters, just stand around. All the monster types in the game. (0.5)
- Display bottom status panel, with player's health, damage and cooldown[3]. (0.5)
- Health for all units, display health bars. (0.5)
- Player can attack monsters. (1)
- Aggressive monsters chase and attack player, according to Algorithm 1. (1)
- Passive monsters wander the map, and run from the player when attacked. (0.5)
- Cooldown (limit to one attack per n milliseconds). (0.5)
- Monster and player death handled. (1)
- Items, in the world. (0.5)
- Player can pick up items, has inventory. Display inventory in bottom status panel. (1)
- Items affect the player in the correct way. (1)
- Dialogue bar display. (0.5)

Customisation

Optional: The purpose of this project is to encourage creativity. While we have tried to provide every detail of the game design, *if* you wish, you may customise *any* part of the game (including the graphics, names of characters and items, map layout and game rules). You may also add any feature you can think of. However, you ***must*** implement all of the features in the above checklist to be eligible for full marks.

To encourage students with too much free time, we will hold a competition for "best game extension or modification". It will be judged by the lecturer and tutors, based on the demonstrations, and three prizes will be offered. We look forward to seeing what you come up with!

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.

[3]This is mostly done for you – see `renderpanel.txt` in the supplied package.

Text, rectangles and colours

In Slick, you use the provided `Graphics` object to draw things other than images, such as text and rectangles. Two useful methods are:

```
void fillRect(float x1, float y1, float width, float height)
void drawString(String str, float x, float y)
```

Both of these methods draw with the "current colour", so before you draw anything, you should set the colour you want it to be, using this method:

```
void setColor(Color color)
```

You can create `Color` objects for any colour you like with this constructor:

```
Color(float r, float g, float b, float a)
```

That lets you provide the **r**ed, **g**reen, **b**lue, and **a**lpha components, respectively. Setting *alpha* to less than 1.0 causes partial transparency, which is nice for health bar backgrounds.

See `renderpanel.txt` in the supplied package for examples of drawing text and rectangles, as well as some sample colours.

Health bars and the status panel

We've given you most of the code for rendering the status panel (see `renderpanel.txt` in the supplied package), since it's pretty tricky to get right. You may simply take that code and use it in your `Player.java` (it is incomplete – note the comments there). After adding the status panel, the player won't be centered in the playable area (which will be 70 pixels shorter). To fix this, calculate the camera position using `RPG.screenheight-RPG.panelheight`.

You may wish to re-use some of this code to render the health/name bars above all of the non-player characters, since it is quite similar. A nice algorithm for calculating the width of the bar is whichever is greater out of $textwidth + 6$ and 70 (but you don't need to use this).

Using a nicer font

Optional feature: The default font that comes with Slick is pretty ugly. If you like, you can change to a nicer font, such as DejaVu Sans Bold (supplied). Loading fonts in Slick is difficult, so we have supplied a source file, `FontLoader.java` which loads fonts for you.

To use this, you'll have to import `org.newdawn.slick.Font` in RPG.java, and add a private field, `Font font`. Applying the font is a two-step process. First, in the `RPG.init` function, load the font using `FontLoader.loadFont`, with a font size of 15, and store it in the `font` field. Second, in the `RPG.render` function, override the default font, using `g.setFont(font)`.

The supplied package

You will be given a package, `project2-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:

- `assets/` – The game graphics and map files.
  - `DejaVuSans-Bold.ttf` – A font, which you may use for rendering text in the game. (You don't have to use this).
  - `map.tmx` – The map file. You should pass this as input to Slick's `TiledMap` constructor.
  - `tileset.tsx` – The tileset file; automatically included by `map.tmx`.
  - `tiles.png` – The terrain graphics; automatically included by `tileset.tsx`.
  - `panel.png` – An image file you may use for the panel at the bottom of the game.
  - `units/` – Contains all of the unit image files. See `data/units.txt`.
  - `items/` – Contains all of the item image files. See `data/items.txt`.
- `data/` – Text files with unit and item stats. These files contain tables and full descriptions of the tables.
  - `units.txt` – Attributes and start positions of all the units.
  - `items.txt` – Attributes and positions of all the items.
  - `dialogue.txt` – Villager's dialogue strings.
- `src/` – Various source files supplied for you.
  - `FontLoader.java` – A small wrapper around Slick to load fonts. (You don't have to use this).
  - `renderpanel.txt` – A small method for rendering the status panel, since it's so fiddly. (You don't have to use this).

Legal notice

The graphics included with the package (`tiles.png`, `units/*`, `items/*`) 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`.

The font included with the package (`DejaVuSans-Bold.ttf`) is from the DejaVu font family, licensed under copyright by Bitstream, Inc. You may redistribute them, but only if you agree to the license terms.
For more information, see `http://dejavu-fonts.org/wiki/index.php?title=License`.

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 be through the LMS. You will need to create a zip file of your eclipse project, along with a PDF version of your reflection. The reflection should contain a description of any changes you have made to your game design (Class diagram) in the process of implementation along with the reasons for these changes. You should also discuss any difficulties you had with the project, at least one key piece of knowledge that you have taken away from this project and finally, what you would do differently if you were to do this project again. The reflection should be no more than **1000** words in total.

We will publish submission links closer to the submission date through the LMS.

Demonstration

You will be required to demonstrate the game engine for your tutor in your workshop in the week of the Workshops Week 12.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 game working correctly. Not attending the demonstration will incur a penalty of one mark.

Late submissions

There is a penalty of 1 mark per day for late submissions without a medical certificate. Late submissions will be handled by Eleanor McMurtry the head tutor for the subject - please email Sarah at `emcmurtry@student.unimelb.edu.au` regarding late submission requests.

Marks

Project 2 is worth **22** marks out of the total 100 for the subject.

- Project 2a is worth **6** marks.
- Project 2b is worth **16** marks.

- Features implemented correctly – **9 marks** (see *Implementation checklist* for details)
- Code (coding style, documentation, good object-oriented principles) – **4 marks**
- Reflection - **2 marks**
- Documentation (javadoc) – **1 mark**

Acknowledgement

This game was designed by Matt Giuca. Amendments by Dengke Sha and Mathew Blair.