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

Project 2, 2013

Given: Friday 13th of September
Project 2a Due: Tuesday 24th of September, 5:00 PM
Project 2b Due: Friday 18th of October, 5:00 PM
Demonstration: Workshop, 21st–25th of October

Overview

In this project, you will create a graphical shoot 'em up game, *Shadow Wing*, 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. 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, printed or hand-written, 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 (you do not have to show the accessor (getter) and mutator (setter) methods for the instance variables). If you like, you may show the class members in a separate diagram to the hierarchy diagram, but you must use 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.

The purpose of this project is to:

- Encourage design of a system prior to implementation,
- 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),
- Give you experience managing a medium-sized project with a revision control system (Subversion).

[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 20th of September, to allow for late submissions to Project 1.

Shadow Wing

Game Manual

Breaking into the command centre of the Vaaj empire, hacking into their mainframe, downloading their top-secret military plans, and stealing a Mark IV Proton-class fighter ship was the easy part.

Now sirens are blaring, and the entire navy of the empire is headed your way, led by one of the most ruthless space invaders in the galaxy. There's only one way out. You flip the ship's thrusters into overdrive. Good thing the rebels are offering a damn fine bounty for this intel.

If you can somehow navigate a maze of twisted platforms, blast your way through waves of Vaaj fighters and survive the asteroid field, you might just be able to take down that alien commander...

Either way, it's going to be one hell of a fight.

Game overview

Shadow Wing is a space combat game where one player battles against computer-controlled enemies on a mission to escape an imperial space platform. The player controls a small ship equipped with warp thrusters, phase shields and a proton cannon. This allows a skilled pilot to make precise maneuvers, and while the ship can sustain a good deal of damage, it can also gun down obstacles and enemy forces effectively.

The goal of the game is to escape the space platform alive, and defeat the evil space invader who leads the alien forces. Figure 1 shows the game in action.

In this game, the "world" is a two-dimensional grid of *tiles*. The ship and background is constantly moving forward, and the player can also freely control the position of their ship on the screen (but of course, the player cannot fly over tiles such as metal and rock).

Gameplay

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

1. The player's ship and the "camera" automatically move forward (upward) by a small amount.
2. All game units have a chance to move and fire missiles. Enemies may move or fire automatically, depending on their attack strategy. The player moves and/or fires if the appropriate keys are being pressed.
3. Move the camera so that it is horizontally centred over the player.
4. The entire screen is "rendered", so the display on-screen reflects the new state of the world.

The camera and the player's ship both move at a rate of 0.25 pixels per millisecond in the upward direction. This means that the ship remains at the same position on the screen unless it bumps into walls or the player controls it. The camera and ship stops moving automatically once the top edge of the screen reaches y coordinate 0.

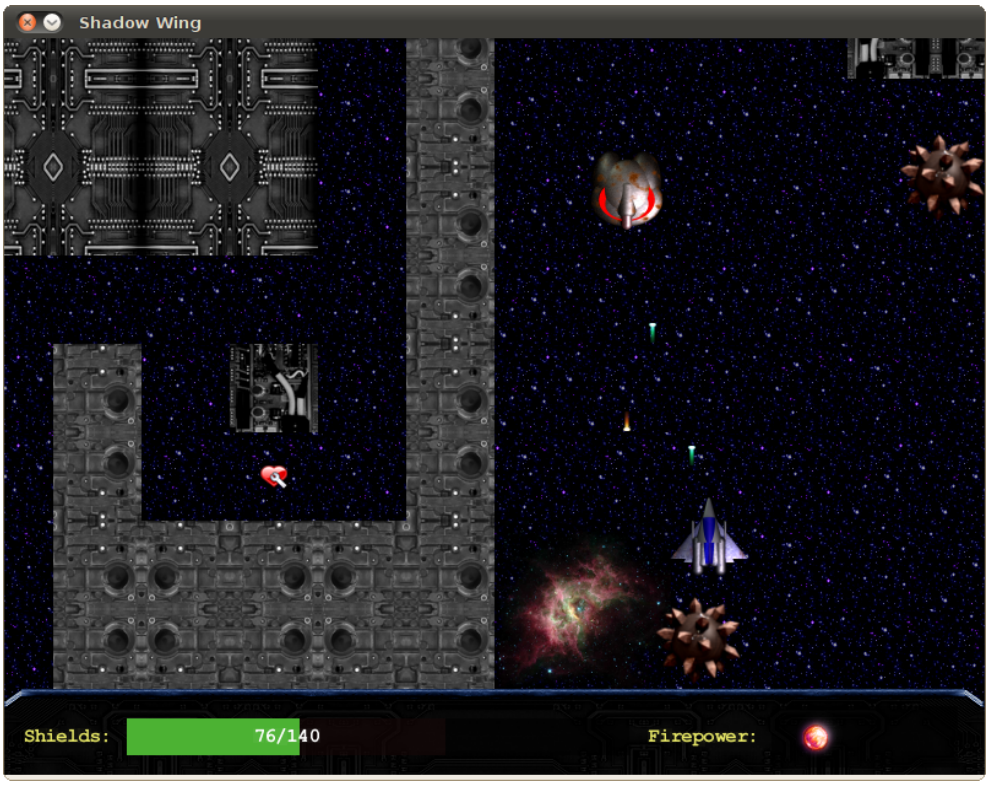

Figure 1: An intense space battle

Units and stats

A *unit* is our term for a game "character", such as a ship or asteroid. There are two general unit types in the game:

- **The player**. Controlled by you; able to fire missiles and fly about the screen.
- **Enemies**. Alien forces or ordinary asteroids; try to kill the player.

All units have a "**Shield**" attribute which is an integer value that determines the amount of damage the unit can withstand before it is destroyed. (See "Collisions and 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). Units have three such stats:

- **Full-Shield** – This stat determines the number of shield points this unit would have if fully repaired. Initially, a unit's **Shield** is equal to its **Full-Shield**.
- **Damage** – This stat determines the amount of damage the unit causes when it collides with another unit.
- **Firepower** – This stat determines how rapidly the unit can fire missiles (more firepower means less time in between shots).

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

Controls

The game is controlled entirely using the keyboard. The **left**, **right**, **up** and **down** keys move the player. The player can fire missiles by pressing the **spacebar**. Holding down **spacebar** fires the missiles continuously, as fast as the player's **Firepower** stat allows.

Enemies

There are a number of different kinds of enemies, as shown in Figure 2, sent by the alien commander to stop you from escaping with those data tapes. Each has a slightly different attack strategy. Enemies do not move or fire if they are off screen. When on-screen, all enemies move at 0.2 pixels per millisecond. Just like the player, they can't move through metal, rock, *etc.*

- **Fighters** – These Vaaj strike craft have come to shoot you down. They always move downwards, and fire their lasers continuously.
- **Drones** – Automated imperial seek-and-destroy robots. They can't shoot, but they move directly on a collision course for the player. Their movement is calculated by Algorithm 1.[2] If you have two on your tail, try crashing them into each other!
- **Asteroids** – Big chunks of rock that can punch a hole right through your ship.[3] They simply move downwards, and can't fire.
- **The Boss** – The commander of the Vaaj fleet won't let you escape easily. He moves from side-to-side, firing rapidly and continuously. He moves to the left until his x position is 288, then moves to the right until his x position is 576, and continuously repeats this process.

Optional feature: If the Boss has been defeated, the player should move automatically upward at a rate of 0.4 pixels per millisecond, flying off the screen and map to freedom. The player should only be allowed off the top edge of the map once the Boss is defeated.

Algorithm 1 Drone's movement calculation

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

[2] You are not required to implement this exact algorithm; as long as the drone moves towards the player somehow. Following this algorithm means drones make the most direct movement to the player.

[3] C-3PO claims that the possibility of successfully navigating an asteroid field is approximately 3,720 to 1.

(a) Fighter (b) Drone (c) Asteroid

Figure 2: Enemies

Combat

Some units have the ability to fire missiles. When a unit fires a missile, a new "missile" object is created. Player-fired missiles are blue and aimed upwards. Enemy-fired missiles are red and always aimed downwards. The missile's position is the same position as the firing unit, adjusted 50 pixels up or down (in the direction the missile is being fired). All missiles move at 0.7 pixels per millisecond. Missiles can collide with any other unit (even friendly units), but cannot collide with each other. When a missile collides with a unit, that unit's shield is damaged by 8 points, and the missile is destroyed. Missiles are also destroyed if they enter a blocking terrain type, or move off of the screen. There can be any number of missiles on the screen at a time.

If units could attack *every* frame, there would be far too many missiles. To slow things down, units have a "cooldown timer". The cooldown timer is measured in milliseconds, and begins at 0ms. Units can only fire when the cooldown timer is 0ms. When a unit fires a missile, its cooldown timer is set to $300 - (80 \times p)$, where p is the unit's **Firepower** stat. Every frame, the unit's cooldown timer is lowered by the number of milliseconds since the last frame, until it reaches 0ms, during which time the unit may not fire. This means the unit can only fire once every 300 milliseconds, or faster if it has upgraded firepower.

Tile Collision Detection

Continuing on from Project 1, all units should be blocked by terrain, using the blocked property of each tile as described in the tiled map. In addition, rather than just block in the centre of each object, the objects should block when their edge comes into contact with a blocking surface, using Algorithm 2.

Algorithm 2 Expanded Collision Detection

1: Let $xPlayer$, $yPlayer$ be the x and y position of the player.
2: Let $xWidth$, $yWidth$ be half of the player sprite's image *width* and *height*.
3: Calculate the positions around the edge of the sprite using a variant of $xPlayer$ +/- $xWidth$ and $yPlayer$ +/- $yWidth$.
4: For each position p, check whether that position is blocked.

Care needs to be taken to try all three possible movements of a given position. It is possible that the ship is blocked in the **y axis** stat but not on the **x axis** , as a result collision checking considers the three possibilities; new x and y coordinates, new x coordinates (vertically blocked) and new y coordinates (horizontally blocked).

A collision can occur between any two units, When two units collide, both suffer damage, and

(a) Repair (b) Shield (c) Firepower

Figure 3: Items

the weaker unit is destroyed. Collisions usually occur between the player and an enemy (usually destroying the enemy and weakening the player's shields), but enemy units can also collide with each other by accident, as well as collide with blocking tiles. When a ship collides with a block tile, it inflicts the same amount of damage it would inflict on another ship to itself, until it either stops blocking or dies.

Collisions occur when two units' sprites intersect. This can be calculated using the units' positions, and their widths and heights (the Image class has methods getWidth() and getHeight()). When two units have collided, each unit has their Shield reduced by the other unit's Damage amount. This is repeated, in the same frame, until one or both of the units' Shields reach 0 or less.

Unit death

If a unit's **Shield** reaches 0, the unit is dead. Enemies that die are removed from the game permanently (they just disappear), with the exception that they are come back if the player is reset to a checkpoint. If the player is forced off the bottom of the screen by a blocking terrain tile, the player is also dead.

There are five *checkpoints* in the game, including the start position, chosen at "safe" parts of the map. The checkpoints have the following y coordinates (in order):

$$13716,\ 9756,\ 7812,\ 5796,\ 2844$$

If the player dies, the ship is teleported so it is horizontally centred, with their x co-ordinate at 1296, and their y co-ordinate at the most recent checkpoint the player has passed, and its **Shield** is repaired to its **Full-Shield**. The camera is also adjusted such that the player's new position is 72 pixels above the bottom of the screen. In addition, all enemies are respawned to their original co-ordinates, with their original health levels. Finally, all missiles should be destroyed. The game then continues as normal.

Items

There are a handful of special *items* scattered throughout the game, as shown in Figure 3. Like units, each item has an (x, y) coordinate in pixels. Items can repair your shields, or enhance your ship's power, so you stand a better chance of completing your mission.

The player can pick up any item by moving within 50 pixels of its position. When picked up, an item will be removed from the world, and take immediate effect on the players stats.

- **Repair** – Restores your ship's **Shield** to **Full-Shield**, repairing any damage your ship has suffered.

- **Shield** – Increases your **Shield** and **Full-Shield** by 40 points each. There are numerous shield power-ups, available with no limit on how high you can increase your shield.
- **Firepower** – Increases your **Firepower** by 1. There are numerous firepower power-ups available, however you can only pick up a maximum of three, to enhance your cannon to an impressive rate of one missile every 60 milliseconds. Any additional pickups after this should be disregarded.

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

The status panel

Figure 4: The status panel

The bar at the bottom of the screen (pictured in Figure 4) is known as the *status panel*.[4] It shows the player's current stats:

- Shields: Shows the player's shield percentage as a green bar on a dark red background, overlayed with the numeric form, "**Shield**/**Full-Shield**". The shield bar is at a constant fixed width.
- Firepower: Shows the number of firepower orbs the player has collected, up to a maximum of three.

[4]The rendering of the status panel has already been done for you – you just need to call the appropriate method. See `Panel.java` in the supplied package. Note that after adding the panel, you may need to adjust the initial camera position to ensure the player starts on-screen.

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 11.

- All the enemy and item types implemented and visible. (1)
- Units have all the necessary stats. (0.5)
- Display bottom status panel, with player's shields and firepower. (0.5)
- Collisions between units. (0.5)
- Accurate collisions with walls/tile blocking (1)
- Enemies implement their individual attack strategies. (2)
  - Shouldn't move or fire while off-screen.
  - Should be blocked by terrain just like the player.
- Players and certain enemies can fire missiles. (1)
- Missiles move properly and collide with other units. (0.5)
- Missiles are destroyed if they hit a wall or move off-screen. (0.5)
- Cooldown timer (limit firing rate based on **Firepower**). (0.5)
- Enemy and player death handled. (1)
  - Enemies die when colliding with walls.
  - Enemies disappear when killed.
  - Player gets repaired when killed.
  - Player dies if forced off the bottom of the screen.
- Player teleports to most recent checkpoint upon death. (0.5)
- Enemies are respawned and missiles destroyed when resetting to a checkpoint. (0.5)
- Player can pick up items, and they affect the player in the correct way. (1)
- Player flies away once the Boss is defeated (**optional**). (0)

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, enemies and items, map layout and game rules). You may also add any feature you can think of. However, you ***must*** implement all of the non-optional 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!

The supplied package

You will be given a package, `oosd-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.
  - `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 and missile 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 with full explanations.
  - `units.txt` – Attributes and start positions of all the units and missiles.
  - `items.txt` – Attributes and positions of all the items.
- `src/` – Various source files supplied for you.
  - `Panel.java` – A small class 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`, `panel.png`, `units/*`, `items/*`) are primarily derived from copyrighted works from the games *Chromium B.S.U.* (licensed under the Artistic License), *The Battle for Wesnoth* (licensed under the GNU General Public License, version 2 or later), and other works. You may redistribute them, but only if you agree to the license terms.

For more information, see `assets/README.txt`, included in the supplied package.

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 ICT room UG-09. (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 and use of Subversion

Project 2a must be submitted via LMS in pdf format.

For Project 2b, you must use the Subversion revision control system to manage your code. You must have an Eclipse project called "`project2`" which should be shared under the Subversion repository location:

`http://ivle-svn.informatics.unimelb.edu.au/users/`*`login-name`*`/oosd/`

The first thing you should do is copy all of the files from Project 1 (either your version or our supplied version) to the `project2` directory, make sure you can still run the game, then commit.

You should commit each substantial change you make to the code, with a meaningful log message. Not only is this good practice, and good for making incremental backups, but there will also be some marks associated with good use of revision control – see the section *marks*, below.

When your project is ready to submit, go to `http://ivle.informatics.unimelb.edu.au/`. Navigate to the "`mywork`" directory for this subject and choose "Subversion → Update" from the "More actions" menu. Select the "`project2`" directory and choose "Submit" from the "More actions" menu. You may submit as many times as you wish. **Be sure to run "Subversion → Update" before you submit, or you might submit an old version.**

Any uncommitted changes will be left out of your submission. (The icons next to each file in Eclipse tell you the file's status – you should see a small yellow box next to each file, indicating it is committed and has no uncommitted changes.) Please use the "Verify" button after you submit to make sure you have submitted the correct version, with all files.

Demonstration

You will be required to demonstrate the game for your tutor in your workshop in the week of the 21st–25th of October. You will be asked to the front of the room for a few minutes, to demonstrate various features of the game working correctly.

Late submissions

There is a penalty of 1 mark per day for late submissions without a medical certificate. IVLE will warn you if you attempt to submit after the deadline.

Marks

Project 2 is worth **20** marks out of the total 100 for the subject.

- Project 2a is worth **4** marks.
- Project 2b is worth **16** marks.
  - Features implemented correctly – **11 marks** (see *Implementation checklist* for details)
  - Code (coding style, documentation, good object-oriented principles) – **4 marks**
  - Good version control practices – **1 mark**

This game was designed by Matt Giuca.