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

Project 1, 2015

Given: Friday 28th of August
Due: Friday 11th of September, 5:00 PM
Demonstration: Workshop, 14th–18th 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] which was 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 the Eclipse IDE for Java development.

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),
- Give you experience working with a game API (Slick),

Figure 1 shows a screenshot from the game after completing Project 1. The game consists of driving a kart (a small car) through a race course. In this image, you can see the player's kart driving around a bendy road.

Note that this is a "tile-based" game; the map is arranged in a grid of squares (note the blocky shapes of the platform). 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, accelerating and rotating the player's kart around the track. Each tile type has a different *coefficient of friction*; tile types with low friction such as the road allow the player to drive much faster than tile types with high friction such as grass and the striped "edge" tiles. Some tiles, such as mountains and water, block the player's movement entirely.[2]

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

The rest of this section details precisely how the game works. You must implement all of the features described here.

[1] The Slick library can be found at `http://slick.ninjacave.com/`. See Workshop 4 for instructions on Slick.
[2] The friction and blocking of each terrain type is encoded in the map file, which is explained later.

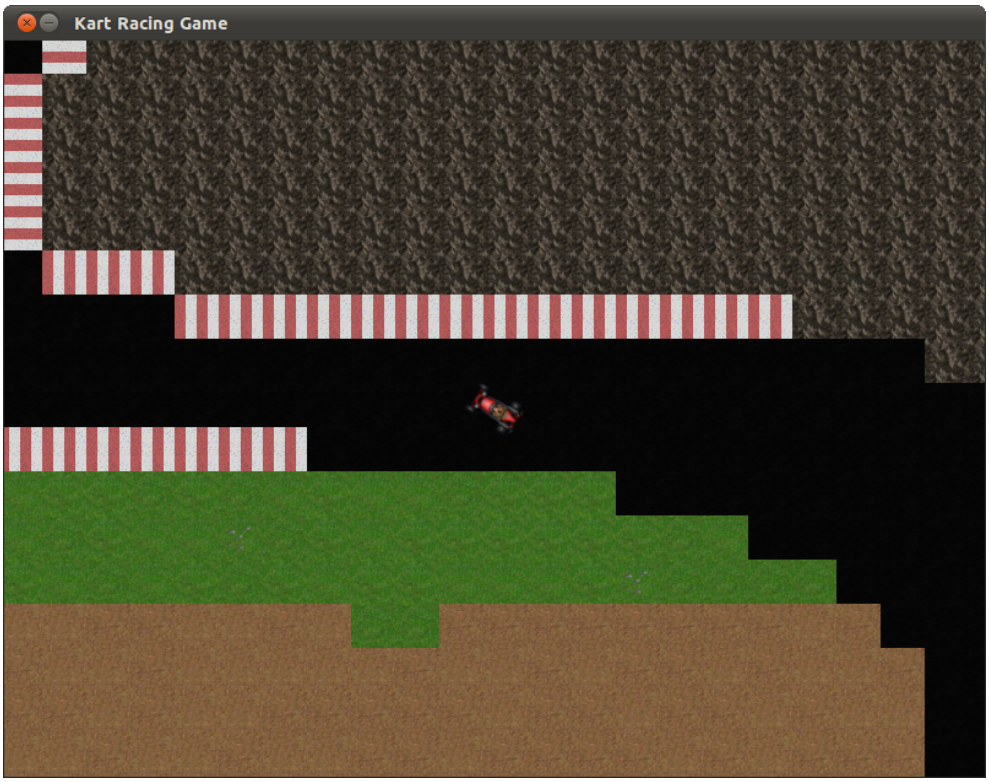

Figure 1: Near the beginning of the game.

The game map

In this game, the "world" is a two-dimensional grid of *tiles*. While the map is designed as a narrow track, the player is able to drive freely around the world (but of course, the player cannot drive on trees, mountains, water, and other types of terrain).

The *game map* is a large 2D array (72×384) of *tiles*, representing the racing track and its surroundings. Each tile has a particular *terrain type*, such as road, grass, tree or water. Each tile has a *coefficient of friction* between 0 and 1, where 0 is a frictionless surface and 1 completely blocks the player's movement. Each tile is 36×36 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 kart with the keyboard, and is able to move it 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,

[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.

rather than being constrained to the tiles of the map grid. All measurements are given in pixels. The coordinate $(0, 0)$ refers to the top-left corner of the map. The player also has an angle, which can be any value from $-\pi$ to π radians. The angle 0 points due north (upwards). Angles increase clockwise, so positive angles face east (right) while negative angles face west (left). Internally, the player has no "size"; it is just a single point, which the player's image should be centred around.

The player should start at location $(1332, 13086)$, at angle 0 radians. This should place the kart at the bottom of the map, on the starting line, facing north.

Controls

The game is controlled entirely using the arrow keys on the keyboard. The **up** key causes the player to accelerate in the forward direction; the **down** key, in the backward direction. The **left** key rotates the player in the anti-clockwise direction; the **right** key, in the clockwise direction. Unlike a real car, it is possible to rotate while stationary.

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 keys are down, the player accelerates forwards/backwards and/or rotates left/right.
2. The player's kart moves according to its current velocity and angle.
3. The "camera" is updated so that it is centered around the player's position.
4. The entire screen is "rendered," so the display on-screen reflects the new state of the world.

Steps 1 and 2 are described in more detail in Algorithm 1. The pseudocode shows how to update the player's state each frame. You **must** implement an algorithm equivalent to this in Project 1.[4]

When rotating, the player's angle changes at a rate of 0.004 radians per millisecond.

The game has a simple physics model, allowing karts to gradually accelerate and decelerate (due to friction). The kart has a variable *velocity*, measured in pixels per millisecond (px/ms). **Note:** Velocity in this game is *not a vector* – it is a single number which is positive when the kart is moving forwards, and negative when the kart is rolling backwards. Initially, the player's velocity is 0 px/ms. The player may accelerate forwards or backwards at a rate of 0.0005 px/ms^2. Conversely, the friction between the kart and the ground causes the player to *lose* velocity: every drivable terrain type has a *coefficient of friction*, μ ("mu"). The kart's velocity is multiplied by $1 - \mu$ every millisecond. For example, on grass, $\mu = 0.0025$, which means the velocity is multiplied by 0.9975 each millisecond. When the player is constantly accelerating, the kart will naturally reach a top speed when the acceleration and friction cancel each other out.[5] When the player is not accelerating, the friction will gradually bring the kart to a halt. See "implementation tips" for details on determining the coefficient of friction for each terrain type.

[4] Disclaimer for students of physics and the mechanical sciences: We are aware that real cars and real friction behave quite differently to the model presented here. However, we have chosen to favour simplicity and fun over realism for this project. To ensure fairness, we require that everybody implements the same algorithm. You will be able to implement more realistic physics in Project 2 if you wish.

[5] The top speed will be $0.0005 \times (\frac{1}{\mu} - 1)$ px/ms.

Algorithm 1 Player update algorithm, run once per frame.

Let Δ be the number of milliseconds elapsed since the previous frame.[6]
Let *rotate_dir* and *move_dir* be -1, 0 or 1, depending on the player's input.
Let μ be the coefficient of friction of the surface the player is currently over.
$angle := angle + 0.004 \times rotate\_dir \times \Delta$
$velocity := (velocity + 0.0005 \times move\_dir \times \Delta) \times (1 - \mu)^{\Delta}$
$x := x +$ X component of polar vector $(velocity \times \Delta, angle)$
$y := y +$ Y component of polar vector $(velocity \times \Delta, angle)$

Some types of terrain (such as water and tyre stacks) are impossible to drive over – these have a μ of 1. If the player crashes into such a tile, their kart should instantly stop moving. The player must not become stuck inside such tiles (see "implementation tips").

The velocity always acts in the direction the player's kart is facing. Unlike a real car (where if you turn sharply you will continue sliding in the direction you were going), in this game, a sharp turn allows you to keep driving at full speed in the new direction. (This simplification allows us to treat velocity as a scalar value and not a vector.) Every millisecond, the player's position is increased by the player's velocity in the direction of the player's angle.

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 (2592×13824 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 centered around the player. This means that as the player drives around the world, the camera should follow. You will find that you will need to render 24 tiles wide and 18 tiles high, to fill up the screen.

Note: You should *not* render the entire map, or the game may run very slowly. Instead, you should calculate which 24×18 tiles are required to fill up the screen, and render those.

Note: Your code should *not* assume that the player is always in the centre of the screen (even though in this case, that will be true, it won't be true for any other objects in the future). Instead, you should calculate the player's on-screen position properly.

Your code

Your code should consist of at least three classes:

- **Game** – 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.

[6] The Δ value is supplied as the `delta` argument to `Game.update` by Slick, and passed on to `World.update`.

- **Camera** – Represents the viewport, should follow the player.

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

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 (which is required, because Slick will automatically load the files `tileset.tsx` and `tiles.png` from this directory – 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. To display an image rotated to an arbitrary angle (in degrees), you should call the `Image.setRotation` method before drawing it:

```
void setRotation(float angle)
```

You can use the `Image.drawCentered` method (in Slick) to render an image onto the screen at a particular location. You should use the version with two arguments:

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

Angles

This game engine involves a lot of angular calculations; in particular, converting polar coordinates (the player's velocity and angle) into Cartesian (the amount to move in the X and Y axes). The second project will involve more advanced angular calculations. To help you with these calculations, we have supplied a complete class called `Angle`.

Objects of type `Angle` represent an angular measure between $-\pi$ and π radians. Methods of this class allow easy conversion between degrees and radians, as well as between polar and Cartesian coordinates. You can construct an Angle in three ways: `Angle.fromDegrees`, `Angle.fromRadians` or `Angle.fromCartesian`. You can retrieve the angle using `getDegrees` or `getRadians`. You can compute the amount to move in the X and Y axes, given an angle and a particular length, using the following methods:

```
double getXComponent(double length)
double getYComponent(double length)
```

Even though you can do these simple calculations yourself, it is strongly recommended that you represent angles using the `Angle` class, to make it easier to do the more complex angular calculations in Project 2. The full documentation for the `Angle` class is provided on the LMS.

Terrain friction and blocking

Finding out the coefficient of friction for any given tile is a two-step process:

1. First, call the `TiledMap.getTileId` method. This gives an integer representing the terrain type (eg. road = 2, tyre stack = 3, grass = 9).
2. Then, call the `TiledMap.getTileProperty` method, looking up the property name `"friction"` for the given tile ID, and convert the resulting string into a number, μ. You may pass `null` as the default value, since all tiles have an explicit friction value.

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)
```

You will find that you need a special case for tiles with a μ of 1. These tiles are "blocking tiles," and you will find that if the player hits them, they become stuck, unable to move due to the infinite friction. You should add a rule that if the player *would* have entered such a tile, they instead "crash," having their velocity immediately set to 0. The player will then be able to reverse out of the crash without becoming stuck.

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 (centre the camera around position $(1332, 13086)$, for now).
4. Defining the `Player` class.
5. Loading the player graphics.
6. Rendering the player at the correct position and angle.
7. Automatically moving the player according to velocity and angle (just assume a constant velocity of, say, 0.25 px/ms, for now).
8. Centering the camera around the player.
9. Ability to manually accelerate and rotate the player.
10. Friction according to terrain type.
11. Block the player from entering terrain with $\mu = 1$, to avoid getting stuck.

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. Its contents include:

- `src/` – The supplied source code.
  - `Game.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.
  - `Angle.java` – A complete class which helps with angular calculations.
- `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`.
  - `karts/` – Contains the kart image files.
    - `donkey.png` – The image file to use for the player.

Compiling and running this code should result in an empty black screen.

Legal notice

The graphics included with the package (`tiles.png`, `karts/*`) are primarily derived from copyrighted works from the games *SuperTuxKart* (licensed under Creative Commons Attribution-ShareAlike 3.0) and *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 `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 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.)

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.

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.

Demonstration

You will be required to demonstrate the game engine for your tutor in your workshop in the week of the 14th–18th 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 Mat Blair, the head tutor for the subject. If you submit late, you *must* email Mat Blair at `mathew.blair@unimelb.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 mark**
  - Player control and movement – **1 marks**
  - Friction and blocking – **2 marks**
- Code (coding style, documentation, good object-oriented principles) – **4 marks**

Acknowledgement

This game was designed by Matt Giuca.