Milestone 1 **(due end of Week 9, 11:59pm Sunday 25th September)**

Question 1 (12 marks) – Super Speed Sprint Racer

In p5.play, sprites provide some additional functionality over normal images. Using p5.play, create a simple top-down racing game similar to Super Sprint (https://en.wikipedia.org/wiki/Super_Sprint) but **simpler**, with the following specifications:

- The program should read a track in from a file called 'track.txt'. 0 should indicate grass, 1 should indicate track and 2 should indicate start/finish line. You should find/draw appropriate images to use as tiles to represent these components, and it is recommended that you load them in as sprites. Each 'tile' should have dimensions controlled by variables, and be positioned according to the input file. If the input file is 10x10 numbers, you should draw this many tiles. The track is static, so this could be drawn in the setup phase, but you will need access to this data for detecting the car leaving the track (**3 marks**)
- The program should draw a car as a sprite on the start finish line, facing one of the adjacent road tiles. You could find a suitable image to use. The car size should be controlled by a variable (but always smaller than the tile size) and be placed in the middle of the road. **(2 marks)**
- The program should allow for 4 inputs – up/down and left/right on the keyboard. Up and down should modify the velocity of the car (you should impose limits so it cannot go too fast or backwards). Left and right should rotate the angle it is facing and moving. This should be done in a smooth way so the car cannot execute instant 90 degree turns. (**4 marks**)
- If the car leaves the track and hits the grass, the game should reset to the start position (simply move the car back to its initial position) (**3 marks**)

For this question, we are not assessing your ability to write a full game (Milestone 2 covers this), but are more interested in demonstrating your ability to use p5.play and other features.

Question 2 (18 marks) – Travelling Salesperson Visualiser

The travelling salesperson problem (TSP) is as follows; "Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city". The problem was first formulated in 1930 and is one of the most intensively studied problems in optimization. Even though the problem is computationally difficult, a large number of heuristics and algorithms are known, so that some instances with tens of thousands of cities can be solved completely and even problems with millions of cities can be approximated within a small fraction of 1% optimal distance.

This question has 3 parts:

a) loadTSP(filename) (**4 marks**). Write this p5.js function that will read a TSP problem from a .tsp file. For this assignment, we will only consider Euclidean distance problems. The information you need to load in includes the problem name, total number of cities, the id of each city, and the co-ordinates of those cities. You could store the city information as 3 parallel arrays (ids, xcords, ycoords) or an array of objects with an id, x and y value. You can find the TSPLIB documentation and the dataset with Euclidean only problems on the course website. Your function should take the name of a .tsp file to load. This can be called from the preload() function.

   Hint*: Reading the TSPLIB.pdf file can be a little daunting. Have a look at some of the actual .tsp files (berlin52.tsp, a280.tsp). It should make the problem seem a lot easier.*

b) showLoadedTSP() (**5 marks**). Write this function that will visualise a loaded problem. You will need to find a way to scale the loaded problem to fit correctly on the canvas. You may choose what shapes/colours to use to visualise the problem. You should make sure you display the problem name and number of cities somewhere on the canvas.

c) showSolution() (**4 marks**). Write this function that will visualise a solution to a loaded problem from a file. You may assume that this function will only be used AFTER showLoadedTSP() has already been run. You will need to load the .sol file in preload(). The first line of a .sol file contains the problem file name (you should check this matches the loaded TSP). The second line contains the tour length, and the rest of the file contains a list of ID's (one on each line) that represents the order that the cities should be connected to form the shortest tour. You should display the solution tour distance on the canvas. Some sample .sol files have been provided on the course website.

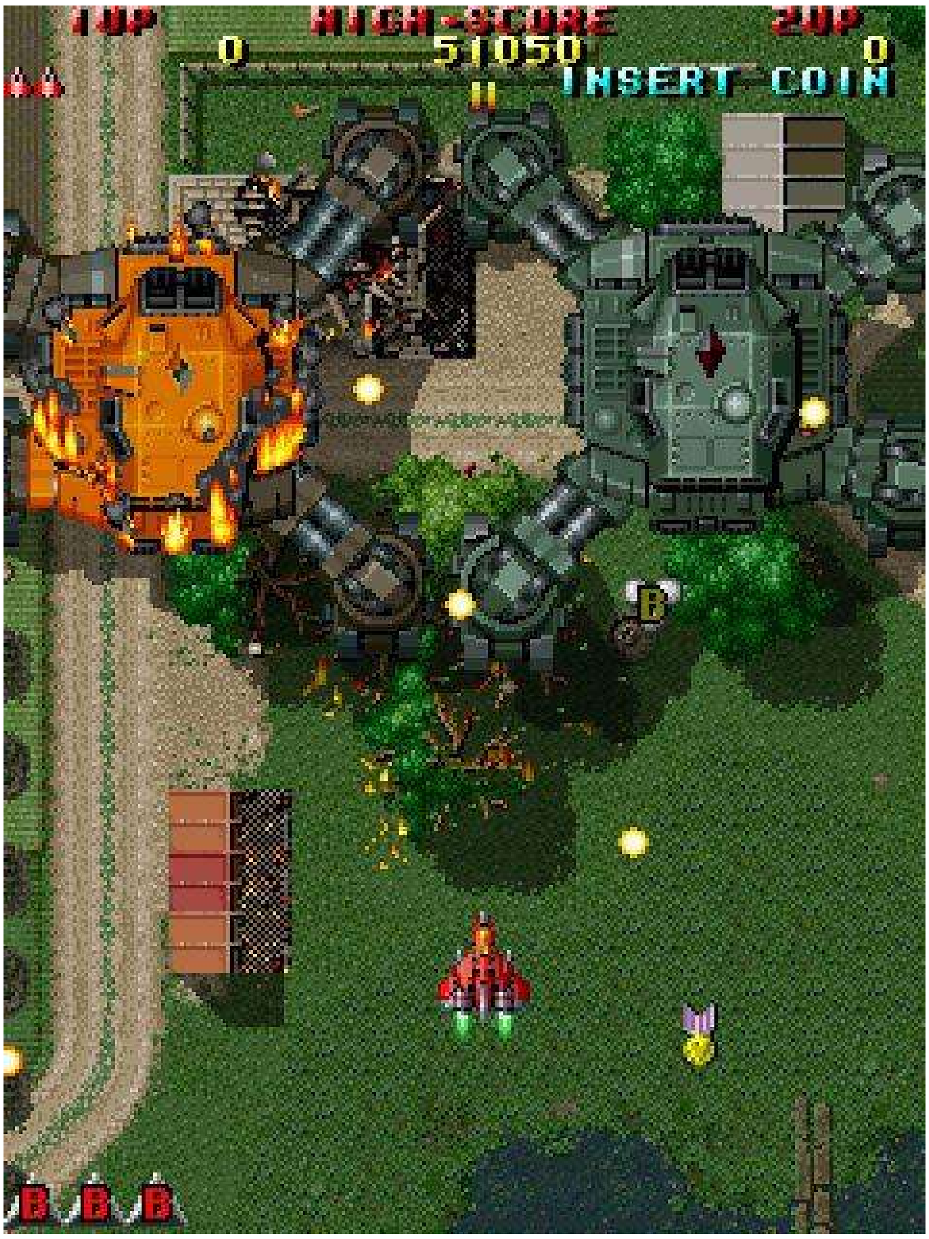
Challenge (**5 marks**): Make the program animate the tour. The program should show the salesman starting from the first city and smoothly moving between all the cities leaving the tour as a trail behind them until they reach the last city and return back to the first one. You can loop this animation if you wish.

Extra / Bonus: (**No marks, just kudos**). Investigate algorithms to solve tsp problems. You could start with Greedy search, and then move on to something like 2-opt. Try and implement a solver to generate your own solutions to TSP problems. This is advanced work and not part of the course. Do this **AFTER** finishing everything else.

Milestone 2 **(due end of Week 11, 11:59pm Sunday 9th October)**

Game (30 marks)

You will be creating your own version of the game 'Raiden'. This is provided as inspiration, but the possibilities are vast (eg. Strikers 1945, etc.)



The player ship should be controlled by keyboard, and at a minimum you should have to fight waves of enemies that come from the top (and maybe sides) of the screen. https://www.youtube.com/watch?v=4i64jDQGZig provides some more inspiration.

Requirements

1. Multiple Scenes

The game must involve at least 4 scenes, there must be a:

- Loading / splash scene
- Main menu scene
- Main game scene
- Leaderboard scene

You should be able to navigate between the scenes

Note: each scene needs to only be navigable by at least one other scene (with the exception of the loading scene, it can be a time-oriented screen)

2. Loops

The game must use at least 3 loops which are fundamental to the operation of the program.

3. Arrays

The game must incorporate at least 3 arrays, with at least 1 of the arrays being used to manage a list of objects on the screen (such as incoming missiles).

4. Interactivity

There must exist player interaction using a keyboard. The game needs to handle at least 3 keyboard keys. Using mouse input is optional.

5. Images

There must be at least 1 image present in the final product (separate to sprites) that is drawn at an appropriate size.

Note: it is up to you which scene you incorporate the image in.

6. Sprites

The p5.play library must be incorporated with a minimum of 3 sprites used. The sprites must have collision detection (between sprites) and must move independently. Sprites must be animated (i.e. a moving object, changing size or colour)

7. Video & Sound

The game should use at least 3 sounds and 1 video. Hint: you can use sounds for shooting and video to introduce the game in the main screen.

8. GUI Input

At least one type of GUI input is required (e.g. text input, slider, button). The input should affect what is drawn. (Hint: Text input could be useful for high score entry)

9. Data

The program needs to read in data from a JSON file where each object must have at least 3 properties. This data must be presented in one of the scenes.

Hint: if your themes / ideas of the main game scene conflict/don't work well with using external data, use the data in the leaderboard scene!

10. Stability

The game must feel stable to play. Your game should not feel 'buggy'.

11. Creativity

You must produce creative work that adds complexity to your game.

Examples of creative work that add complexity to your game include:

- Having multiple types of enemies
    - In addition to incoming missiles, you may have other enemy types that enter to make the game harder
- Having a dynamic background
    - Having a scene with a day/night cycle? Or Stars (sprites or images) may traverse down the screen in a straight line, being generated at random rates & positions from the top of the screen

- Having multiple weapon types / modes of fire
    - Your ship may fire in a straight line in its unaltered state, but when picking up a 'bonus' blob (that traverses down the screen every 20 seconds, 30 seconds if missed) the ship has an altered mode of fire, such as shooting in multiple directions.
    - You could add super weapons, or other beneficial additions to the defenders (shields? Power-ups?)
- UI & Fonts
    - You may download and use custom fonts that better fit your theme.
- Score & Sound Effects
    - Play a special sound every 10 enemies destroyed and add relevant bonus scoring.
- Enemy health
    - Enemies only fall after being hit 3 times. Use an array to store the health of individual enemies and update it upon collision detection.
- Ability to change the difficulty of game
    - Could be as simple as having two difficulty options that change the number of lives available, or as creative as having increased speed of incoming fire / increased toughness of opponents. You are encouraged to create your own assets (i.e. missiles, explosions) on top of using public assets from the internet.

You are encouraged to create your own assets (i.e. opponents or laser fire) on top of using public assets from the internet.

Note: the ability to creatively expand your game is highly dependent on having basic functionality, such as interactivity.

12. Code

You must produce clean, well-written code. This encompasses indentation, meaningful naming of functions & variables, use & overuse of absolute values.