College of Engineering, Construction & Living Sciences
Bachelor of Information Technology
ID511001: Programming 2
Level 5, Credits 15

Project 2: Space Invaders

Assessment Overview

In this assessment, you will design & develop a GUI implementation of the classic arcade game **Space Invaders**.

Learning Outcomes

At the successful completion of this course, learners will be able to:

1. Build interactive, event-driven GUI applications using pre-built components.
2. Declare & implement user-defined classes using encapsulation, inheritance & polymorphism.

Assessments

Assessment	Weighting	Due Date	Learning Outcomes
Project 1: Pong	25%	17-10-2022 (Monday at 4.59 PM)	1 & 2
Project 2: Space Invaders	35%	04-11-2022 (Friday at 4.59 PM)	1 & 2
Theory Examination	30%	18-11-2022 (Friday at 4.59 PM)	1 & 2
Classroom Tasks	10%	21-10-2022 (Friday at 4.59 PM)	1 & 2

Conditions of Assessment

You will complete this assessment during your learner-managed time. However, there will be time during class to discuss the requirements & your progress on this assessment. This assessment will need to be completed by **Friday, 04 November 2022** at **4.59 PM**.

Pass Criteria

This assessment is criterion-referenced (CRA) with a cumulative pass mark of **50%** over all assessments in **ID511001: Programming 2**.

Authenticity

All parts of your submitted assessment **must** be completely your work. If you use code snippets from **GitHub**, **StackOverflow** or other online resources, you **must** reference it appropriately using **APA 7th edition**. Provide your references in the **README.md** file in your repository. Failure to do this will result in a mark of **zero** for this assessment.

Policy on Submissions, Extensions, Resubmissions & Resits

The school's process concerning submissions, extensions, resubmissions & resits complies with **Otago Polytechnic** policies. Learners can view policies on the **Otago Polytechnic** website located at https://www.op.ac.nz/about-us/governance-and-management/policies.

Submission

You **must** submit all project files via **GitHub Classroom**. Here is the URL to the repository you will use for your submission – https://classroom.github.com/a/J5gYpMTC. Create a **.gitignore** & add the ignored files in this resource - https://raw.githubusercontent.com/github/gitignore/main/VisualStudio.gitignore. The latest project files in the **master** or **main** branch will be used to mark against the **Functionality** criterion. Please test before you submit. Partial marks **will not** be given for incomplete functionality. Late submissions will incur a **10% penalty per day**, rolling over at **5:00 PM**.

Extensions

Familiarise yourself with the assessment due date. Contact the course lecturer before the due date if you need an extension. If you require more than a week's extension, you will need to provide a medical certificate or support letter from your manager.

Resubmissions

Learners may be requested to resubmit an assessment following a rework of part/s of the original assessment. Resubmissions are to be completed within a negotiable short time frame & usually **must** be completed within the timing of the course to which the assessment relates. Resubmissions will be available to learners who have made a genuine attempt at the first assessment opportunity & achieved a **D grade (40-49%)**. The maximum grade awarded for resubmission will be **C-**.

Resits

Resits & reassessments **are not** applicable in **ID511001: Programming 2**.

Instructions

You will need to submit a project & documentation that meet the following requirements:

Note: Independent research requirements are highlighted yellow.

Functionality - Learning Outcomes 1 & 2 (40%)

- Project **must** open with code or file structure modification in **Visual Studio**.
- A game of **Space Invaders must** be driven by one **Timer**.
- Mother ship
  - Move left & right in response to a key press.
  - Fire in response to a key press.
  - Have no more than 15 missiles in the air at any one time.
  - Be destroyed if hit by an enemy ship's bomb.
- Mother ship missile
  - Be fired from the top of the mother ship.
  - Move only upward & in a straight line.
  - Live for a random number of timer ticks between 1 & 70.
  - Be destroyed if they hit an enemy ship.
- Enemy ship
  - Start in a grid of four rows. Each row has ten enemy ships.
  - Move left to right in lock step (i.e. the entire grid moves) at an appropriate speed. Reverse the enemy ships' direction when the leftmost or rightmost edge of the grid reaches the edge of the screen.
  - Drop bombs at an appropriate frequency (see details below).
  - An individual enemy ship may drop a bomb only if there are no other enemy ships in front of them in its column.
  - Be destroyed when hit by a missile from the mother ship.
- Enemy ship bombs
  - Be fired by an enemy ship from the closest row to the mother ship.
  - Move only downward & in a straight line.
  - Live for a random number of timer ticks between 1 & 70.
  - Be fired probabilistically. That is, at each time interval, each enemy ship who can fire will have a 1/100 chance of doing so. The logic for this is:

```
For each enemy ship who can drop a bomb
    Select a random number between 0 & 99
        If that number is 99 then drop a bomb
```

    Note: You do not have to use 99 here. Using any single value will achieve the same result
- A scoring system. The user wins when the mother ship destroys all enemy ships. The user loses when the mother ships is hit by an enemy ship's bomb, or if an enemy ship collides with the bottom of the screen. Appropriate feedback **must** be displayed to the user, i.e., **"You win!"** or **"You lose!"**.
- **Independent Research:**
  - A highscore system. When the game is over, the user's score is saved, i.e., written to a text file. Read the user's scores from the text file and display the last five to the user.
  - Play a sound when:
    - The mother ship fires a missile.
    - A missile destroys an enemy ship.
    - An enemy ship detroys the mother ship.

    Note: These sounds **must** be unique.
  - An ability to play a new game, restart a current game & pause a current game.

Code Elegance - Learning Outcomes 1 & 2 (45%)

- Adhere to the four principles of **OO**, i.e., encapsulation, abstraction, inheritance & polymorphism.
- Use of intermediate variables, constants & enumerations.
- Idiomatic use of control flow, data structures & in-built functions.
- Efficient algorithmic approach.
- Sufficient modularity.
- Each method & class **must** have a header comment located immediately before its declaration.
- In-line comments where required.
- Project files, i.e., **.cs** files are fomatted.
- No dead or unused code.

Documentation & Git Usage - Learning Outcomes 1 & 2 (15%)

- Provide the following in your repository **README.md** file:
  - Your project's UML diagram.
  - References to used code snippets from **GitHub**, **StackOverflow** or other online resources.
  - Known bugs if applicable.
- Commit at least **20** times per week.
- Commit messages **must** reflect the context of each functional requirement change & formatted using the naming conventions dicussed in **Week 1**.

Additional Information

- **Do not** rewrite your **Git** history. It is important that the course lecturer can see how you worked on your assessment over time.
- You **must** declare an appropriate class structure for the game objects, i.e., mother ship, missiles, enemy ship & bombs. All game objects will need to know where they are on the screen, what image they are displaying, whether or not they are still alive, how to draw, how to move & how to tell if something has collided with them. They will also need to be able to create instances of themselves & free up their memory at the end of the game.
- Individual game objects will also need to know other specialised things. For example, an enemy ship **must** know if they are in the front of the formation to drop a bomb. Missiles & bombs need to know their lifespan & how many ticks they have been alive.
- To create the enemy ship formation, do not simply use a global list or array of enemy ships. It would be a poor OO design. Instead, create a fleet object so you can have a fleet of enemy ships, a fleet of missiles & a fleet of bombs. A fleet object can contain a field that is either a list or a one or two dimensional array.
- Your game should be controlled by a Controller class, which updates the state of each game object by looping through the appropriate arrays.
- You should drive the entire game off of a single-timer. At each tick, call the Controller to run another iteration of the game.
- It is important to keep careful track of whether each object is alive or not. All enemy ships start the game alive. When a missile hits them, they are no longer alive & never regenerated. As discussed below, missiles & bombs are created & destroyed repeatedly throughout the game. To display the correct state of the game, use your timer event to move & draw only those entities who are alive.

- To launch a missile from the mother ship, you will need to ensure that the mother ship does not currently have its limit of 15 missiles in the air. When you wish to launch a missile, you can search the fleet object for a missile that is not currently alive. You can mark this missile as alive & set its coordinates to where the mother ship is. When a missile hits a target, or reaches the end of its lifespan, mark it as not alive. In the Timer, you draw all missiles which are alive.
- You may use a similar approach to that described above to manage the bombs. However, technically, there is no limit to the number of bombs that may be in the air at any time. Practically, the probabilistic firing behaviour of the enemy ships will keep the number of bombs fairly low.
- You **must** determine when a missile hits an enemy ship & when a bomb hits the mother ship. It is called collision detection, & there are many interesting algorithms for implementing it. There is a method that belongs to the Rectangle class called IntersectsWith().