Project Documentation

1. **Overview**: what the software does, what it doesn't do? (this can be taken/updated from the project plan)

A classic Angry Birds game is developed. The game's mechanism is to throw bird objects and get points based on performance, where a higher score is given to:

- Destroy more blocks and pigs, or cause a chain reaction with a single shot
- Use fewer birds to clear the level
- Utilise birds' special abilities effectively

The data is loaded from the level file after choosing the game level from level 1 to level 3. The higher the level, the more difficult the game becomes. Higher difficulty in-game level refers to:

- More pigs to kill with fewer birds available
- More complex construction structure

Players can see their scores in real-time. Once the game is finished, the player will get a star rating (maximum 3 starts) to show how well the player played.

Apart from the basic type, two special birds are implemented: speed and explode. Speed bird will speed up towards the target if the player presses the mouse after the bird is released. The exploding bird will explode after pressing the mouse and blast away anything around it.

Currently the game has only the classic game mode.

Game features

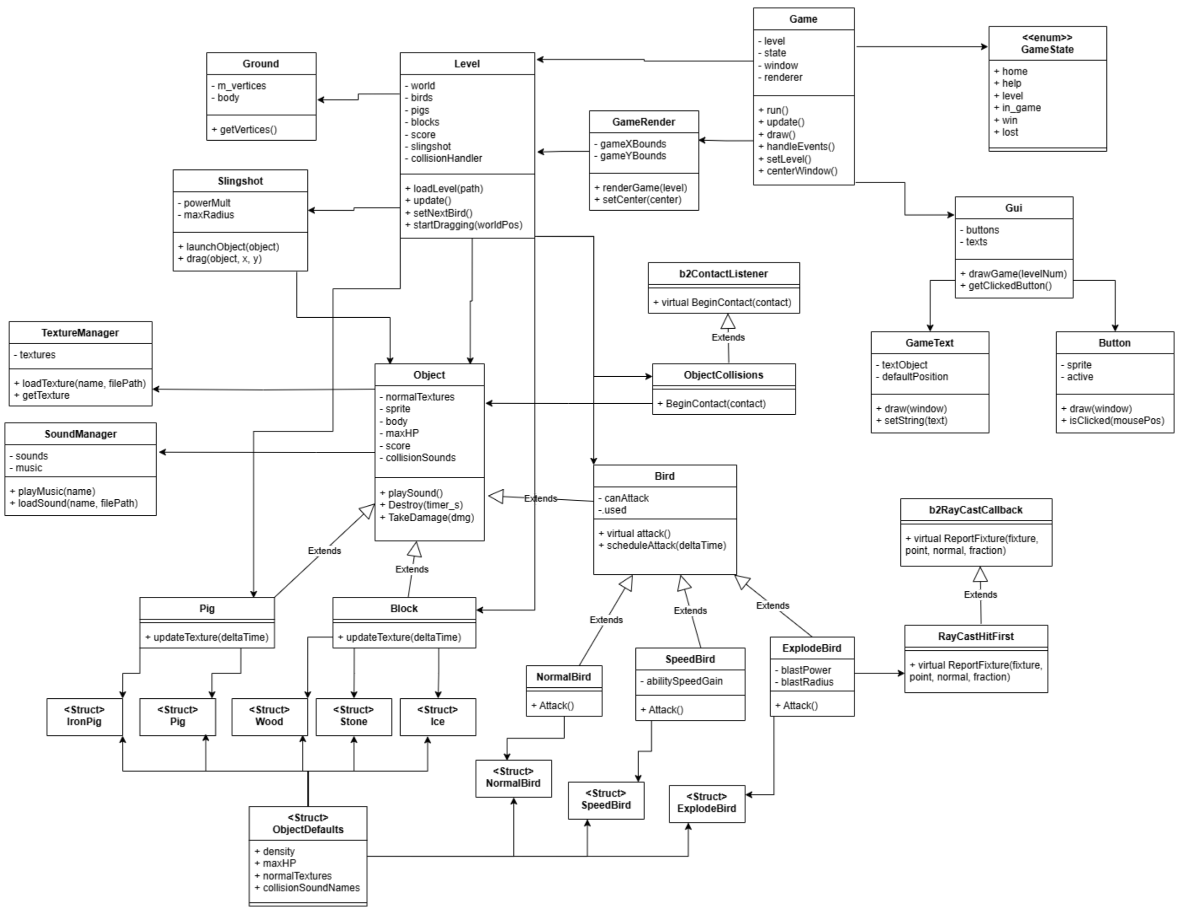
All basic features have been implemented:

- The view follows the bird as it moves sideways
- 2 birds with a special action: speed boost and explosion
- Controlling the attack with a mouse (throwing & special action)
- Game levels are loaded from files
- 3 game levels with increasing difficulty
- The user interface shows the score and number of birds left
- Physics simulation using Box2D

The following additional features are implemented:

- A star rating (1 to 3) shows how well the player played.
- Sound effects
- Excellent graphics (textures, animations, camera animations)

2. **Software structure**: overall architecture, class relationships diagrams, interfaces to external libraries



https://app.diagrams.net/#G1oC8OIDdc-8rlwZuSGTP_Yggn0FMqK1sa#%7B%22pageId%22%3A%22C5RBs43oDa-KdzZeNtuy%22%7D
(also found in the project repository)

Object class represents almost all game objects apart from Slingshot and Ground. Block, Pig and Bird classes are all derived from Object. Out of the classes, Object and Bird are abstract. We used struct for different types of pigs and blocks (ice, wood, stone). ExplodeBird, SpeedBird and NormalBird classes are derived from the

Bird class for special birds. The reason is that different birds have different attack abilities and different animations. Having sub-classes enables polymorphism, which makes the whole structure clean and organised. As for blocks, the difference lies only in density, textures, sound effects, shape and hp values. Apart from those, blocks all function the same. Therefore using a struct carrying values for initialisation is the most intuitive approach. Moreover, since we used all original textures from AngryBird loaded, it makes more sense to implement default values for birds, pigs and blocks, which are mainly maxHp, density, shape, sounds and textures.

Ground was made into its own class since it was considered to be different enough from blocks, namely by shape and textures.

Due to the large amount of textures and sounds used, we centralised resource management by having two classes: TextureManager and SoundManager. TextureManager takes care of loading all textures and retrieving textures. Similarly, SoundManager takes care of all sound loading and retrieving.

In order to make the program structure clearer, we separated the logic of the User interface (buttons, game texts) from game object rendering. We also separated all level-specific logic and operations from the Game class such that the Game class can function as a central manager, calls the GUI class to draw the window, buttons, and texts, the GameRender class to render game objects, Level class to load data of a specific level from the file. We implemented GameText and Button classes, which are called in the GUI class to be drawn. Having GameText and Button classes help to add more customisation and makes the code cleaner, as we can encapsulate some verbose namespaces, and logics inside the classes.

The level class was made to contain functionality related to game levels and game logic in a single place. Having it makes managing levels easy. Levels include an instance of the Slingshot class for launching birds. Similarly, the Slingshot class contained methods for launching objects and fields for rendering and sounds. This makes the code cleaner and easier to understand and manage.

All classes dealing with physics are connected to box2d, and all classes dealing with image and audio are connected to SFML.

3. **Instructions for building and using the software**
   a. How to compile the program ('make' should be sufficient), as taken from the git repository. If external libraries are needed, describe the requirements here
   b. How to use the software: a *basic user guide*

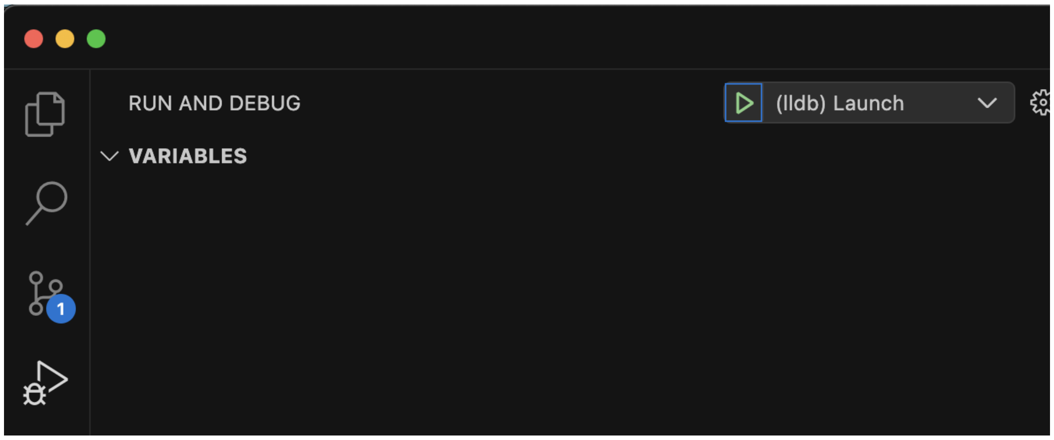
Install Required Libraries

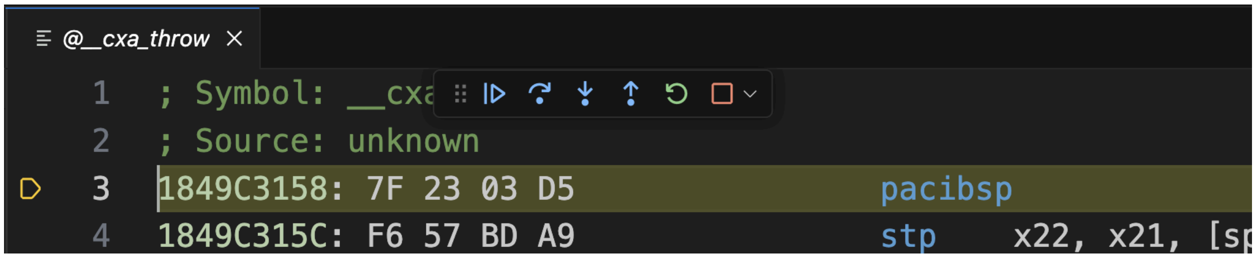
Box2D (version 2.4.2 or higher): Install via a package manager, such as:

- Brew install box2d (for mac)
- Sudo apt install libbox2d-dev (for linux)

SFML (version 2.5 or higher): Install via a package manager, such as:

- brew install sfml (for mac)
- sudo apt install libsfml-dev (for linux)





```
; Symbol: __cxa
; Source: unknown
1849C3158: 7F 23 03 D5                pacibsp
1849C315C: F6 57 BD A9                stp     x22, x21, [s
```

To run the program:

- On mac
  - Build by RunAndDebug button in VScode and then click (lldb) Launch. Afterwards, you will see @_cxa_throw. Click Step Out to start the Game.
- On other platforms
  - Click (gdb) Launch. Build the project and run.

Memory is managed mainly by the two asset managers, TextureManager and SoundManager, and by Level for Objects.

The game menus can be navigated by clicking the buttons to select levels and move to another menu. The game is played by dragging the bird in the slingshot and releasing it.

The birds can perform a special attack when the mouse is clicked. Everything else is done automatically.

4. **Testing**: how the different modules in software were tested, and a description of the methods and outcomes.

Unit testing with GoogleTest is used, and the program passes all tests.

- Test for Block module:

Block module represents destructible game objects: ice, wood and stone. Tests validate its construction, interactions(damage), and destruction.

The create method validates that a Block is initialised correctly in the game world.

The take-damage method ensures that taking damage reduces the block's health points (HP).

Destroy method validates that a block is added to the Object::destroyList when destroyed.

- Test for Bird Module:

The Bird module encapsulates the behaviour of birds that destroy objects in the game. Tests verify initialisation and interaction functionality.

The create method ensures that a bird is initialised correctly in the game world.

The attack method ensures the attack mechanic works and the bird cannot attack again immediately.

- Test for Level Module:

The Level module handles the game levels, including objects' arrangement and state. Tests ensure proper loading and initial conditions.

The load method ensures that the level loading system correctly initializes all objects and the level state.

Assertions: Validate expected conditions using Google Test assertions such as:

- EXPECT_EQ for exact matches.
- EXPECT_NE to ensure non-null values or differing results.
- ASSERT_NE for critical checks that halt testing upon failure.

In addition, the software was run after most changes to confirm correct functionality. If bugs or errors were found, new changes would be made, and the process would be repeated. This was considered to be convenient, efficient and intuitive.

5. **Work log**: a detailed description of the division of work and everyone's responsibilities. For each sprint, describe the tasks that were planned and what was completed, along with a summary of which tasks each team member contributed during the sprint and how much time was used.

Initially, the group had 5 people. The sprint plan was re-made after One person quit in the first week. Due to the inactivity of two team members, the group was split, and the sprint plan was unsuitable again. After splitting, the group had only 2 people, which made communication easy. Since we found it difficult to estimate how much time is needed, we adopted a flexible schedule: we will just do whatever we feel like doing and notify the other team members. This approach has worked very well and made written plans practically unnecessary. The team also worked together on campus very often, making communication very efficient. Additionally, each member reviewed commits that were made by others.

Work division and time:

- Sprint 1 (18.10. - 1.11.): (~5-10h each)
- Xin Lin: Original project plan
- Jaakko Rautapää: Project plan updates

- Sprint 2 (1.11. - 15.11.): (~10-20h each)
- Xin Lin: Started on GUI implementation
- Jaakko Rautapää: Initial implementations for Object and its subclasses, initial unit tests

- Sprint 3 (15.11. - 29.11.): (30h+ each)
- Xin Lin: GUI and UI elements, Game initial implementation, texture assets, texture manager, refactoring Defaults and other updates
- Jaakko Rautapää: Refactoring, sound effects, sound assets, level loading, slingshot, more blocks, collisions, damage and destroying, more birds, dragging and other updates

- Sprint 4 (29.11. - 13.12.): (40h+ each)
- Xin Lin: Updates on GUI, Game and textures, drag events, Ground, animation, more textures, refactoring, other fixes and updates
- Jaakko Rautapää: Music, score, more tests, game rendering, camera animation, launching, setting next bird, level design, refactoring, other fixes and updates

More details can be found in the project commits.