

Introduction

Our project is a text-based platformer game built using ncurses. The user can select a level to play through. The program will then allow the user to maneuver through the level by controlling a player object and avoiding enemies and pitfalls. They will move the player using input from the keyboard in the form of the arrow keys, allowing them to jump and move from side to side. They can jump onto platforms without falling from them. Gravity will pull them downwards if they are in the air. The program will track the player's score based on the collectibles they collect and display it at the top of the screen. They can enter their name at the end of a level to be displayed on a leaderboard.

Assessment Concepts

1. Memory allocation from the stack and the heap

- **Arrays:** A dynamically allocated array will be used to store the different objects that need to be loaded in as the player progresses through the environment.
- **Strings:** The player can store their name as a string, which will be displayed on a leaderboard along with their score.
- **Objects:** Each aspect of the environment that needs to be rendered will be stored as an object.

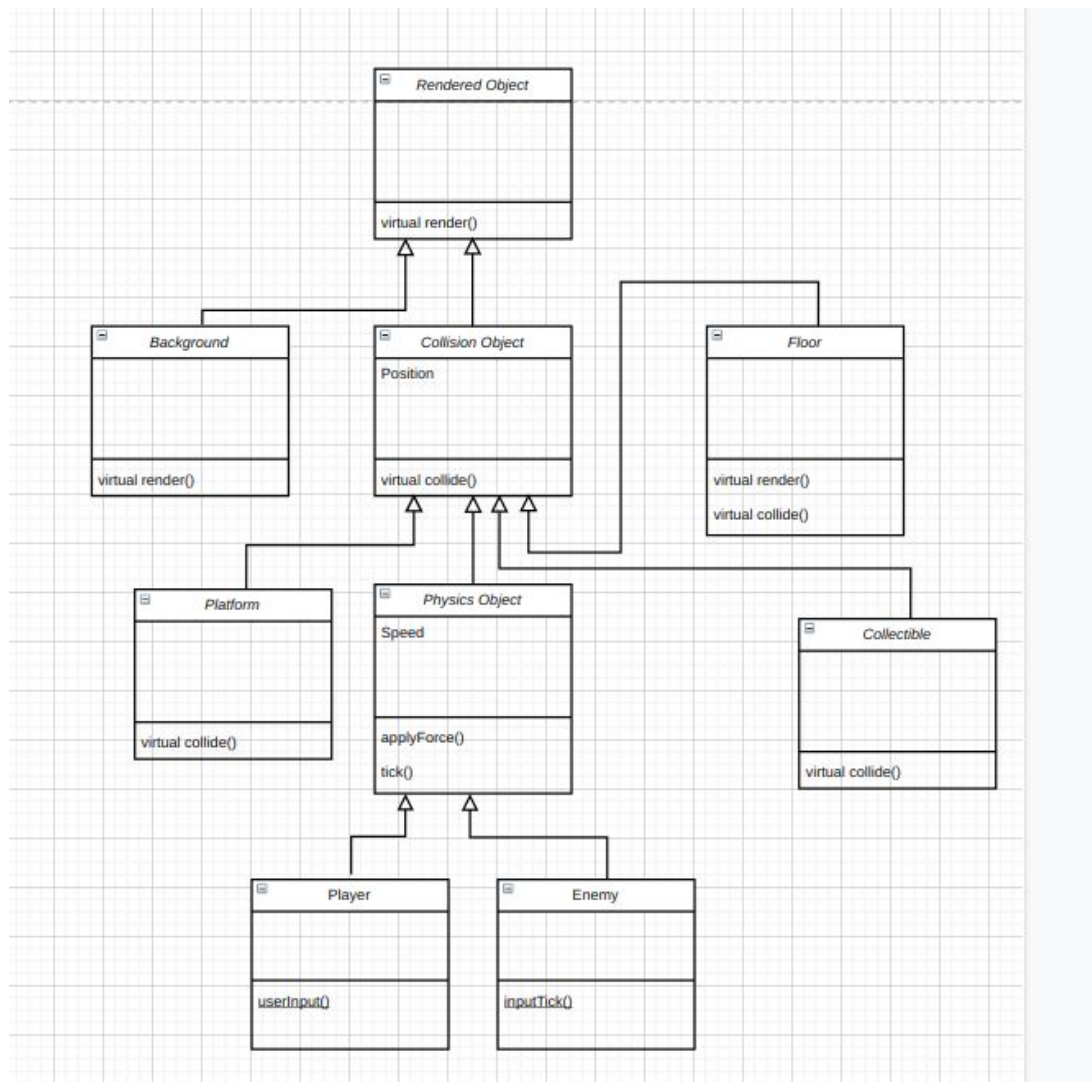
2. User Input and Output

- **I/O of different data types:** The environment will consist of characters and boxes, and will be rendered with `printw()` and similar ncurses functions. The player's score, represented as an integer, will be displayed on the screen, as well as the name's of the highest scorers on the leaderboard, stored as strings. The player controls the character with the arrow keys, which will be input as key codes / integers. The player can input their name as a string to be represented on the leaderboard.

3. Object-oriented programming and design

- **Inheritance:** A rendered object class is the ancestor to all of the other classes. These classes can then inherit from more specific classes such as one for objects that can be collided with or one for objects that have their own physics.
- **Polymorphism:** Objects to be displayed on the screen will be stored as generic rendered objects, and then polymorphic behaviour can be invoked to make use of the specific method for rendering whatever type of object they actually are.
- **Abstract Classes:** The rendered object class will be an abstract class with a generic, virtual `render()` method. The classes that inherit from it can then implement their own versions of this method.

Class Diagram



Class Descriptions

Rendered Object

A generic, abstract class that all objects that need to be rendered inherit from.

Background

Represents the background to be displayed at the back of the window throughout the game. Inherits directly from the rendered object class because it has no special behaviour aside from being rendered.

Collision Object

A class for objects that have behaviour upon being touched by the player object, but do not have their own physics.

Floor

A class representing the platform that the player can walk along without falling through. Can be located in the air or along the ground. When the player leaves the platform, they are marked as ungrounded and gravity is applied.

Collectible

A class for objects that the player can touch in order to gain bonuses such as a higher score or more lives.

End

Class that inherits from Collision Object and ends the game upon being touched by the player. Endpoint of the different levels.

Physics Object

An abstract class for objects that are not only rendered, but also can move according to their own physics.

Player

A class representing the object that the user controls with the arrow keys. Includes the implementation for the player's physics, including what happens when they jump and move to the left and right.

Enemy

Represents enemies that the player must avoid or else they will lose lives or score. May have different physical behaviour such as moving across the screen, jumping, and so on.

User Interface

The user interacts with the program through an ncurses window. The different levels are displayed in boxes on the screen, and they can use the arrow keys to select their preferred one. The environment of the level will then be displayed to them, and they can use the arrow keys to maneuver through it. At the end of the game, they can use the keyboard to enter a name that will be displayed on the scoreboard along with their score.

Code Style

All code should be indented properly with 4 spaces. Lines should generally not be longer than 120 characters.

Each class should be in its own file. Classes will be separated into separate interface (.h) and implementation (.cpp) files. Preprocessor include guards should be used in each header file.

Classes should be named in UpperCamelCase according to the function of the class. Variables and functions should be named in snake_case. For variables with short lives (eg, loop index variables), short names (i, n, etc) can be used for terseness and readability. Other variables should have a descriptive name according to their purpose. Class member variables should always have a descriptive name, with no affixes.

Unless a function is trivial, or the behaviour is obvious from the name (ie, getters and setters, basic calculations), a comment should be given to describe the function's behaviour. In general, comments should be written for any code that has behaviour that isn't obvious at a glance. Abstract classes should generally have all methods documented to describe the behaviour they should be implementing. Comments should be written as the code they're documenting is written.

The code should be periodically checked to ensure these standards are being followed.

Testing

Multiple different testing methods are going to be employed during our testing process, including automated testing, regression testing and white and black box testing.

Our files would be tested by executing the makefile to compile the completed components of our code. This is an example of regression testing to ensure all previous components are present when tested, and that the new code is able to behave as intended in synchronisation with the previously implemented code. Integration testing is also exhibited through this process, as the different classes and their interactions are being tested.

Automated, unit testing will be also utilised to test each class code in order to both reliably cover all the components, as well as reduce the resources and requirements per individual test trial. For example, each new class will likely be tested using automated testing through specific inputs within the test files. The test files will be executed among compiling and would consider a wide variety of inputs, such as regular expected inputs as well as edge cases.

Both white box and black box testing will also be utilised in the testing process to cover a wide boundary of what the user may do. White box testing will be employed through testing the program as intended for the user, to see how the program generally runs if any improvements are required, and black box testing would be used to test for edge cases, or specific features.

Example Tests

Gravity Test: The Player object is dropped two pixels above a floor object. The test ticks the Player object three times and calls collision object. On the first tick, the Player should go down one pixel. On the second tick, it should go down another such that it is above the platform. It should not move on the third tick.

Collision Test: A Player and Collision object are placed at different pixels in the window. They are given momentum and ticked such that they come into contact with one another. On this tick, the Scene should call the Collision Object's collide method on the Player and invoke the appropriate behaviour. For example, if it is a coin, the Player's score should be incremented.

Walking Test: The Player is placed on a Floor and moved to the right. It is ticked until it exits the Floor. On the tick that it is no longer grounded, gravity is invoked and the Player starts moving downwards on each tick. This test served as an example of regression testing, as it allowed us to identify broken behaviour where the Player would not fall after exiting the platform, which was caused by an earlier change to the physics.

Schedule

Prior to Week 9 Assessment

- Write the specification document
- Set up the code structure and code sharing
- Create a makefile to compile and run all of the code and tests
- Outline the testing strategy

Prior to Week 10 Assessment

- Code the different abstract classes
- Develop the player physics
- Create the behaviour for rendering and colliding with the basic aspects of the environment, including the floor, platforms, and background
- Create a basic level by storing these environmental aspects to be rendered in a dynamic array
- Develop and progressively run automated tests for each of these aspects of the code
- Plan how to split the workload for the rest of the project

Prior to Week 11 Assessment

Our plan to finish work is to first implement some of the non-essential features that have not been completed. These include enemies, selecting from different levels, a leaderboard, and entering a user name. We will continue to implement further tests for these new features and ensure they, along with existing tests, continue to pass. We will also improve the physics engine so that the game feels smoother and movement is more realistic. Finally, we will review code style, test coverage, and appropriate comments and ensure that these aspects meet our standards. We will then do some final manual testing of the code before submitting it.

Hamish

- Improve the physics engine
- Implement enemies class
- Check that comments are included above any ambiguous code

Jason

- Make some more levels using the available objects

- Check code style meets the requirements
- Make a leaderboard that is displayed and stored in a local file

Billy

- Check that the tests cover all aspects of the code
- Add feature to let the user enter their name
- Add feature to choose a level