Introduction to C++

|  |  |
| --- | --- |
| **Assessment Task Number:** Part 4 – Make a Retro Game | |
| **Unit Code(s):** | **Unit Title(s):** |
| ICTPRG443 | Apply intermediate programming skills in different languages |
| ICTICT449 | Use version control systems in development environments |
| **Instructions to Learners:** | |

Program a retro game of your choosing. Acceptable games include:

* Pong
* Arkanoid
* Snake
* Asteroids
* Space Invaders

Be conscious of the time you have available to program this game. Creating a very simple game is perfectly acceptable for this assessment. If you are unsure which game to make, Pong would be a good choice (either 2 player keyboard, or 1 player with a simple computer player).

Document the design of your game, including the data structures and algorithms used, in a brief design document (1-2 pages)

Use version control to create and manage a repository for your code.

|  |  |  |
| --- | --- | --- |
| **Task** | | **Evidence Criteria** |
| 1. | Design Document | A 1-2 page design document of your game, including a description of the data structures and algorithms used. |
| 2. | C++ Game Project | A game project written in C++. |
| 3. | Version Control Used | A link and screenshot of the version control repository containing your game project.  Place your evidence inside a MS Word or PDF document. |
| **Submission Requirements:** | | |
| You will need to submit the following:   * A Release build of each application that can execute as a stand-alone program * Your complete Visual Studio project * A link and screenshot of your version control repository in a MS Word or PDF document   Be sure to remove any temporary build folders (i.e., the Debug and Release folders). Only project files, source code files, and any resource files used should be included in your submission.  Package all files in a single compressed archive file (.zip, .7z, or .rar) | | |

Design document

**Project**: Space Invaders.

**Author**: Zora Jane Kerr.

**Objective**: Make a retro game in C++.

**Due date**: 26/05/2023.

**Link to version control repository commit history**: [Link](https://github.com/ZoraJayKay/Introduction-to-CPP-Assessment-4/commits).

Table of Contents

[1. Planning stage 3](#_Toc134369350)

[1.1. Planning: Expectation setting 3](#_Toc134369351)

[1.2. Planning: Milestone-setting (Indicative schedule) 3](#_Toc134369352)

[1.3. Planning: Discern minimum viable product requirements 4](#_Toc134369353)

[1.4. Planning: Document possible optional extras 5](#_Toc134369354)

[1.5. Planning: Draft code structure (minimum viable product) 7](#_Toc134369355)

[2. Project stage 8](#_Toc134369356)

[2.1. Project notes: Minimum viable product 8](#_Toc134369357)

[2.1.1. Program 8](#_Toc134369358)

[2.1.2. Game 8](#_Toc134369359)

[2.1.3. Initialise 8](#_Toc134369360)

[2.1.4. Object 9](#_Toc134369361)

[2.1.5. Sprite 10](#_Toc134369362)

[2.1.6. Player 10](#_Toc134369363)

[2.1.7. Enemy 10](#_Toc134369364)

[2.1.8. Weapon 10](#_Toc134369365)

[2.1.9. Controller 11](#_Toc134369366)

[2.2. Project notes: Optional extras 11](#_Toc134369367)

[2.3. Project: Research undertaken 11](#_Toc134369368)

[3. Completion stage 12](#_Toc134369369)

[3.1. Lessons learnt 12](#_Toc134369370)

[3.1.1. C++ language 12](#_Toc134369371)

[3.1.2. Classes 12](#_Toc134369372)

[3.1.3. Functions / methods 12](#_Toc134369373)

[3.2. Performance against expectation 12](#_Toc134369374)

[3.2.1. Minimum viable product 12](#_Toc134369375)

[3.2.2. Optional extras 12](#_Toc134369376)

[3.3. Conclusion 12](#_Toc134369377)

1. Planning stage
   1. Planning: Expectation setting

* The single biggest danger on this project is being too ambitious for the timeframe given. I need to clearly identify the critical path and consistently work to it FIRST, before implementing 'nice to have' features - MINIMUMS FIRST.
  1. Planning: Milestone-setting (Indicative schedule)
* I set the following schedule for myself as a starting point to flexibly work around while completing the project.

|  |  |  |
| --- | --- | --- |
| Item | Indicative due date | Hours allocated |
| 1: Minimum viable product | 19/05/2023 | 36 |
| 1: a) Planning | 04/05/2023 | 4 |
| 1: b) Coding | 12/05/2023 | 20 |
| 1: c) Troubleshooting | 19/05/2023 | 8 |
| 2: Optional extras | 26/05/2023 | 12 |
| 2: a) Coding | 26/05/2023 | 8 |
| 2: b) Troubleshooting | 26/05/2023 | 4 |
| Submission | 26/05/2023 | 48 |

* 1. Planning: Discern minimum viable product requirements
* Use 2D RayLib C++
* 1 scene / level
* 1 Player
  + The player has 1 weapon type.
    - Projectiles despawn on collision or leaving game scene
  + The player has 1 life.
  + Player can move side to side in the scene, constrained to the play area.
    - Game recognises both keyboard and mouse.
  + Player is a parentless object in the scene
    - Player object has a child (sprite texture / image)
* 1 wave of enemies
  + 1 type of enemy fighter.
  + Enemies move side to side and descend toward the player when the outermost enemy collides with the edge of the screen
  + Enemies fire projectiles which damage buildings and kill the player
    - Projectiles are fired at random times
    - Projectiles despawn on collision or leaving game scene
  + Enemies are a parentless object in the scene.
    - Enemy has a sprite texture / image child.
* The game tracks player score for 1 game.
* Win criteria: All enemies reduced to 0 lives.
* Loss criteria: Player reduced to 0 lives.
* Continue criteria: Both player and enemies exceed 0 lives.
  1. Planning: Document possible optional extras

1. Program
   * Pre-game lobby menu
   * Loading screen
   * Pause function
   * In-game pause menu
   * Save function
   * Load function
   * Resume (last game) function
2. Quality of life
   * Instructions / tutorial
   * Menu audio
   * In-game audio
   * Menu animations
   * In-game animations
   * Animated background
   * Change keybindings
   * Change the volume level
   * High score record
3. Player
   * Multiple lives
   * Base upgrades / powerups
   * Ship upgrades / powerups
   * Ship weapon upgrades
   * Multiple player ship types
   * Multiple player weapons
4. Enemies
   * Multiple enemy waves
   * Multiple enemy ship types
   * Multiple enemy weapon types
   * Boss encounters
   * Environmental hazards
   1. Planning: Draft code structure (minimum viable product)

Program class (called “Assessment 4”)

* Start the game.
* End the game.

Game class

* The scene for a play session
* Draw scene to screen
* Update scene calculations

Initialise class

* Set up initial conditions for a play session

Object class

* Default game object

Sprite class

* Hold a texture for a game object

Player class

* A derived class of Object for the player

Enemy class

* A derived class of Object for the enemy

Weapon class

* A class for player and enemy weapon types

Controller class

* Calculate all game changes arising from player inputs

Matrix3 class

* Maths helper class

Vector3 class

* Maths helper class

1. Project stage
   1. Project notes: Minimum viable product
      1. Program

What should the program class do?

* Start the game
  + Instantiate a Game class
* End the game
  + Destroy the Game class
  + Run shutdown on program
    1. Game

What should the game class do?

* Run initialisation
  + Instantiate initialisation class
* Update the timer
* Update relationships
  + Add objects from the scene
    - Add new objects to a list to be added to the scene on next update
  + Remove objects from the scene
    - Remove new objects from the list of items to be added to the scene on next update
* Update calculations
  + Run Update function
    - Parents calculate themselves and their children
* Update the Draw
  + Scene UI draws itself
  + List of parents Draw themselves and their children
* Detect game end
  + 1. Initialise

What should the initialise do?

* Set the parameters for the game to run after starting and before ending
  + Set game-wide parameters
    - Create the visible play area
    - Create a timer
    - Create and set player score = 0
    - Set target FPS
* Set the win and lose conditions
  + Win if enemy lives = 0
  + Lose if player lives = 0
* Instantiate all of the objects
  + Instantiate the player object
    - Set initial player position
    - Offset object position to middle of object rather than top-left
      * Set size of the offset for collision detection?
    - Set lives = 1
    - Instantiate a player sprite child object
  + Instantiate enemy
    - Set initial enemy position
    - Offset object position to middle of object rather than top-left
      * Set size of the offset for collision detection?
    - Set lives = 1
    - Instantiate an enemy sprite child object
  + Instantiate the base
    - Set position
    - Lives = 1
    - Instantiate base sprite child objects
    1. Object

What should the object class do?

* Create a blueprint for any object with a transform in the game
  + Have a parent object (optional)
  + Have a list of children
  + Have a transform (vector of ints?)
    - 3D matrix for local transform
    - 3D matrix for global transform
  + Offer types (enum?)
    - Player
    - Enemy
    - Base
    - Weapon / projectile
  + Lives integer
  + Instantiate a weapon for this object
  + Functions for use by a controller / movement pattern
    - SetPosition
    - Translate
    - CopyTransform (set local to global)
  + Create virtual OnUpdate function for use by separate types
    - Player
    - Enemy
    - Base
    - Weapon / projectile
  + Draw this object and its children
    - Have override OnDraw function
    1. Sprite

Sprite class

* Load a texture
* Set the sprite’s height equal to its texture height
* Set the sprite’s width equal to its texture width
* Run OnDraw method
  + Draw the texture
    1. Player

What should the player class do?

* Be a container for all of the parameters that collectively represent the player
  + Set the object type to ‘player’
  + Parent a sprite object to hold a texture
  + Lives
    - Receive default from initialisation
    - Receive information from the weapon class of enemies
  + Override parent OnUpdate function
    - Transform
      * Receive default from initialisation
      * Receive information from the controller
  + Weapon class object
    - Receive default from object type
  + Instantiate a copy of the controller class
    1. Enemy

What should the enemy class do?

* Be a container for all of the parameters that collectively represent an enemy
  + Set the object type to ‘enemy’
  + Lives
    - Receive information from the weapon class of the player
  + Transform
    - Execute a pattern
  + Parent a sprite object to hold a texture
    1. Weapon

What should the weapon class do?

* Distinguish between whether it is a player or enemy weapon
  + Binary option like “is this the player’s weapon?” maybe
* Receive inputs from the controller class
  + Shoot
* Receive inputs from the enemy class for periodicity of instantiation
  + Options for pattern of attack
    - Predictable
    - Random
* Send outputs to the screen (draw)
  + Transform
  + Despawn
* Calculate updates
  + Transform
    1. Controller

What should the controller class do?

* Everything to do with keystrokes
  + Movement
    - Send information to the player class
      * Move the transform of the player
  + Attacks
    - Send information to the weapon class of the player
      * Instantiate attacks for the player
  1. Project notes: Optional extras

1. Pause function

I implemented a pause function initially because I wanted a simpler way to pause my debugging than pausing the running of the game, but it illustrated to me that I wasn’t using delta time calculations properly. That led to me looking up how to build a timer from scratch for the game, which was too complex for me to follow in full honesty, though I did find an excellent tutorial online for how to create one by following along.

Debugging the pause function illustrated to me that it is vital for there to be a single source of truth for all time arithmetic in the game, and to **know unequivocally** where that is, which I had taken for granted in previous assessments where the concept was fuzzy.

1. Weapon effects audio

Weapon audio effects were, unsurprisingly, more complex than anticipated, but provided a useful lesson in how to access files external to the program source code. The return on simple QOL improvements cannot be overstated in terms of simplicity of code, ease of implementation, and low barrier to conceptual understanding.

* 1. Project: Research undertaken

1. Timers in C++
2. Enums in C++
3. Arrays in C++
4. Raylib Audio devices
5. Raylib Music devices
6. Raylib drawing
7. Randomness
8. Pointers
9. Pointer ownership for purposes of deletion (warrants its own entry separate from pointers)
10. Cyclic dependencies
11. Vectors of classes
12. Alignment of game changes with the timing of the update cycle
13. Const char pointers and concatenating various string object types in C++
14. Collision detection
15. Acceleration
16. Encapsulation

1. Completion stage
   1. Lessons learnt
      1. C++ language

* There is no default stopwatch in C++, I have to build one or use **#include <ctime>**
* To perform arithmetic like the C# library, **#include <cmath>**
* DECLARE members in the header file and DEFINE/INITIALISE them in the source code!
* If a member function creates a pointer to an instance of an object whose existence is constrained to that scope, but I still want to access the object after its scope terminates, be sure to either allocate memory for it via a pointer to a *new* objector else return the original object somewhere by reference before its scope expires.
* Array iterators should always be of type size\_t.
  + 1. Raylib
* You must use the **InitWindow()** function before loading any images or textures.
* I can use ExportImageAsCode() in Raylib instead of loading an image and then turning it into a texture. This means that I don’t have to have separate .png files – I can have an image file and convert its pixel data into a header file.
  + 1. Classes
* I need to remember that to instantiate class (B) from class (A), the referencing class (A) needs to **#include** the header file of class (B).
* Having a timer class with methods you can call from elsewhere neatens up code implementation a lot.
* Make sure there are functions to access protected variables, and protect every variable that doesn’t explicitly warrant having public access.
* A vector of base class pointers can accept derived class pointers, but once inside the vector they will be identified as the base type, so can’t be worked with according to their derived type in reverse (they become ‘spliced’). Once I put enemies into the bucket of root objects, I can’t do things like count how many enemies are in there by class. I *can* check their other parameters, such as their enum, which records which type they are, but that may or may not be especially type-safe.
  + 1. Functions / methods
* Constructors and destructors do not have return types.
* Be sure to only call delete on pointers with memory allocations from the function that logically owns that memory allocation.
* Always create instances of classes as pointers where possible for speed of memory access.
* Give functions names that make the most sense *when they are in contextual use* not *when being read in the header file*. A neat header file naming convention does not necessarily make for source code that is easier to read.
  + 1. Collision detection
* I did not anticipate the complications arising from simultaneous collision detection. Projectiles and their AABBs are wide enough to hit two base block objects simultaneously, which causes an exception in the deletion iterator. I didn’t have time to work out how to appropriately execute a solution, so I simply made the projectiles thin enough that they cannot collide with two blocks at the same time. The compromise is that projectiles can, if lined up correctly, pass between blocks of the base, but if the user didn’t know that was a compromise, it would probably just appear intentional.
  + 1. Sprites vs no sprites
* The Player, Enemies and projectiles all use sprites for their visual appearance, but I wanted to demonstrate that I knew an alternative way to create objects. For bases, I created one Base object per overall structure, which is parent to a collection of GameObject base blocks. These blocks are dynamically created by each Base, and every base block can be individually shot and removed, which I feel is a core part of the space invaders game loop. Each base block is given a number at its creation according to the order in which it was instantiated. Each update, the base looks through all of its children, and records a collection of the creation integers that still exist, and how many total children remain. The base iterates through all of the possible base block positions, but only draws those whose position matches one of the creation integers. For example, if base blocks 7, 9 and 12 have been destroyed, then base block indices 7, 9 and 12 will be iterated through, but they will not be drawn to the screen.
  1. Performance against expectation
     1. Minimum viable product

The minimum viable game took around 75% more hours than anticipated which was disappointing but also unsurprising.

The process of migrating the work on the tank game I had created in the *Maths for Games* course into C++ was around 25% of total hours used, with around 50% devoted to learning C++ while implementing new concepts, and 25% on researching game functionalities. Data transmission between classes was the largest complicating factor, though I felt I made significant improvements in my understanding over the course of building the game. I did not have time to learn how to implement AABB collision detection during creation of the tank game, so was satisfied with its use in this project.

* + 1. Optional extras

I had little to no time to implement optional extras, due to the time needs of other programming modules.

* + 1. Project timing

The game was 90% complete before I needed to put it on hiatus at the end of the allocated time so that I could assess the time needs of *Code Design & Data Structures*. That unit took significant effort to complete, and I needed to conduct the same exercise for *Artificial Intelligence for Games* and *User Interfaces* before I had an opportunity to return to this game and complete it. After leaving the project for fully 3 months and returning to it, I was very pleased with my code commentary, function and class structure of the program, and educated guesses that I had made about how to undertake logical processes. It is important to note that I did not take as break for the purpose of polishing or finessing the game; I simply didn’t have time to complete it without falling behind on even more complex subjects.

* 1. Conclusion

I’m pleased with the minimum product I’ve made. While the game is not feature-heavy, I think the structure works well, is easy for me to follow, and importantly does not contain large sections of spaghetti code that I created on-the-fly without a plan for how it would interact with the rest of the program. After having a break of approximately 3.5 months from working on the project, I returned and don’t believe it would need real changes or re-factoring in order to expand it as a demonstration project - all that I need to flesh it out is more time to write more code.