Technical Design Document

Centipede | By David Flintoft

1.0 Overview

1.1 Game overview

Centipede is an old arcade game released on the Atari system in 1981. The game is simple yet quite strategic. It involves moving the player horizontally along the bottom of the screen, with the ability to fire projectiles up the screen, with the goal of hitting the closing in centipede. The centipede begins at the top of the screen and has to make it down to the player, maneuvering through a maze of mushrooms. The centipede moves horizontally until it hits either the edge of the screen, or a mushroom, in which case it moves an inch closer to the player and switches its horizontal moving direction. However, if a player's projectile hits the centipede, then it will split in half and create a new mushroom where it split in two. This results in two centipedes, with different paths, closing in on the player. This effect continues and can result in many centipedes. The player must destroy them all before they reach the bottom of the screen.

The core mechanics of Centipede is the player shooting and the centipede's movements. These two mechanics are what make the game challenging and interesting.

1.2 Project overview

My project is a combination of the aforementioned Centipede game and a collection of smaller programs for testing the functionality of various container classes & algorithms. My version of the Centipede game is a lot simpler. Set in outer-space, the player takes control of a spaceship firing lasers through an asteroid field to stop the oncoming centipede alien creature.

My project also contains some other games implemented for fun:

- Simon game
- Memory game

2.0 Development Environment

2.1 Language and IDE

- C++ in Microsoft Visual Studio Enterprise 2017 (Version 15.9.2)

2.2 Third party libraries

- AIE bootstrap
- ImGUI (https://github.com/ocornut/imgui)

2.3 Project management tools

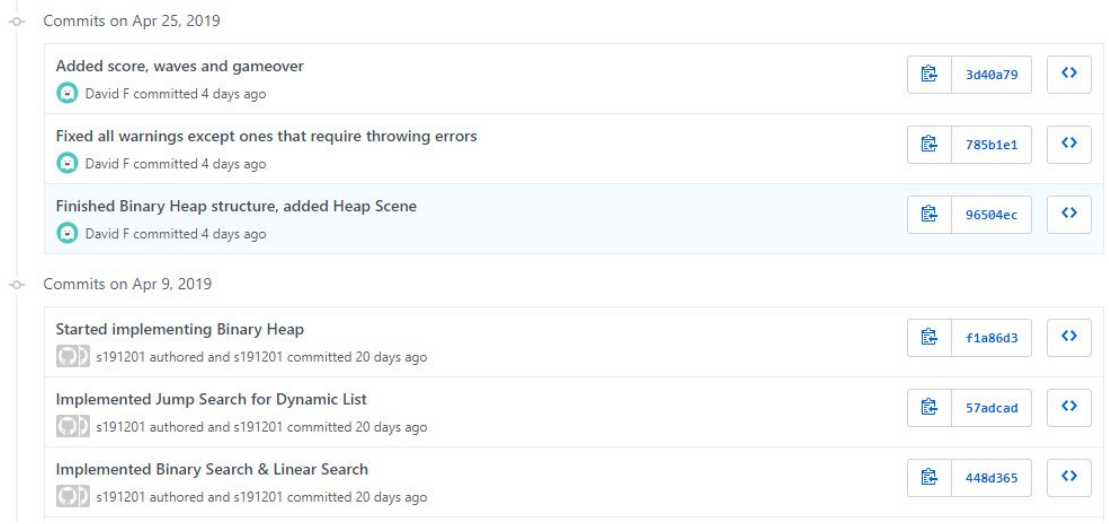
- Trello (https://trello.com/) - used for planning
- Draw.io (https://draw.io) - used for constructing UML diagrams
- Pen & paper - used for sketching and designing

2.4 Source Control procedures

GitHub was used from the beginning of the project to manage the different versions and branches of the code.

GitHub repository: https://github.com/Davinatoratoe/Centipede
Commit log: https://github.com/Davinatoratoe/Centipede/commits/master

Screenshot of commits (does not include all commits):



3.0 Timeline

3.1 Milestones

Major Milestone	Date
Alpha build ready	23/04/2019
Beta build ready	26/04/2019
Gold build ready	07/05/2019

Milestone	Date
Implement aie-bootstrap and basic game classes	29/03/2019
Implement container classes (excluding algorithms)	09/04/2019
Implement container interactive programs & menus	09/04/2019
Implement player, bullets, mushroom & centipede	23/04/2019
Implement collision functions	23/04/2019
Implement algorithms for the container classes	23/05/2019
Test menus and gameplay	06/05/2019

The play-testing sessions were completed on 06/05/2019.

The Simon and Memory game were implemented after all milestones were completed.

4.0 Game Architecture

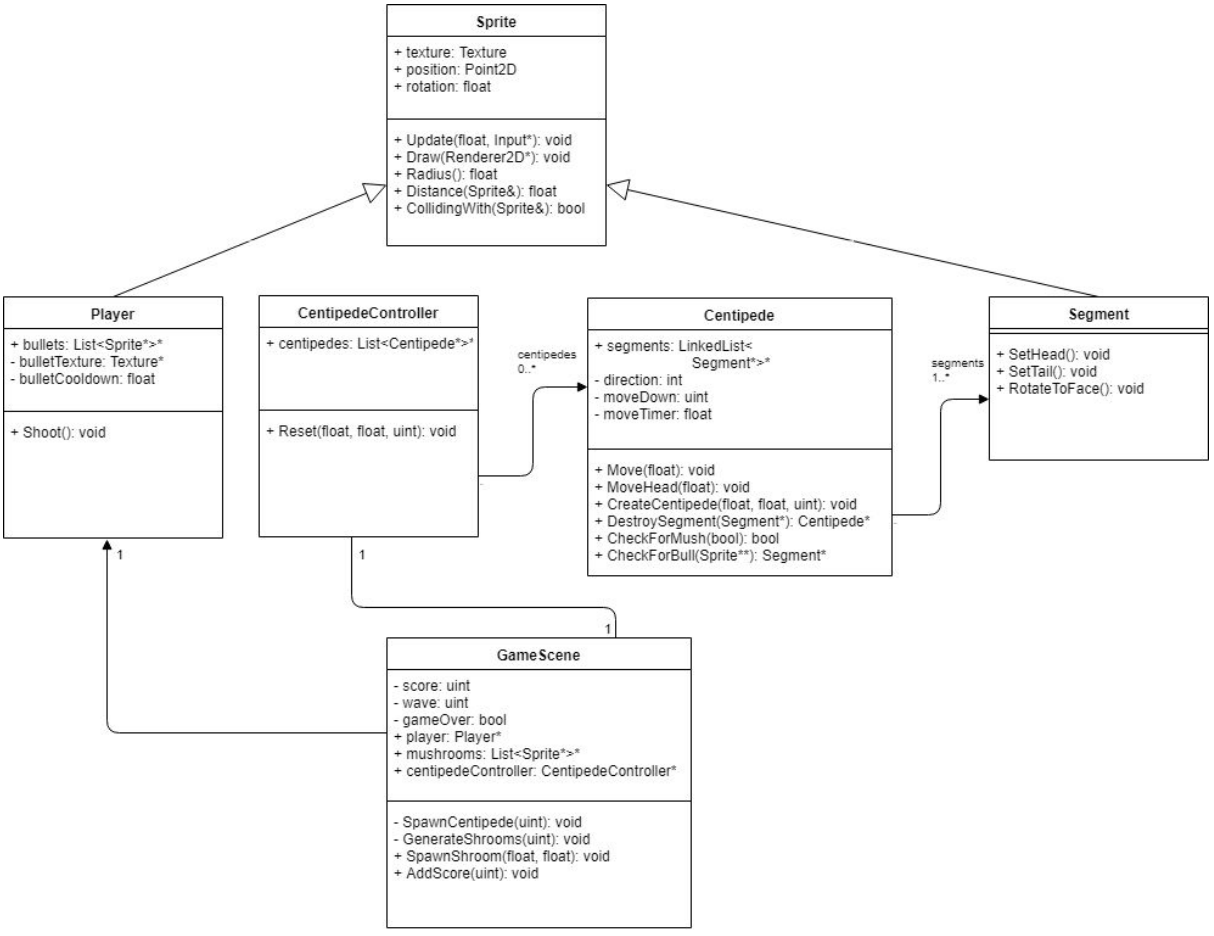
Core data structures, classes and algorithms implemented in the project.

4.1 Data structures

- Dynamic List (Used in Centipede)
- Linked List (Used in Centipede & Simon)
- Dequeue
- Stack
- Binary Tree
- Quad Tree
- Binary Heap

4.2 Core classes

Class diagram for the core classes in the project:



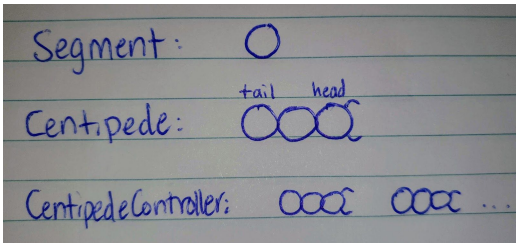
- *GameObjects*
    - Sprite - basic object that can be drawn and moved.
    - Player - controls the ship and its behaviours.
    - CentipedeController - manages the collection of centipedes on the screen.
    - Centipede - moves and checks for collisions with the segments.
    - Segment - element of the centipede.
- *Scenes*
    - Scene - abstract class used for creating, updating & drawing game states.
    - MenuScene - scene that handles the main menu.
    - GameScene - scene that handles the Centipede game.
    - SimonScene & MemoryScene - scenes that handle their respective games.
    - *Container*Scene - scene that allows manipulating various container classes.

4.3 Algorithms

- *Sorting*
    - Insertion Sort
    - Quick Sort
    - Bubble Sort & Cocktail Shaker Sort
    - Heap Sort
- *Searching*
    - Binary Search
    - Linear Search
    - Depth-First Search & Breadth-First Search
    - Fibonacci Search
    - Jump Search
    - Binary Tree Search

4.4 Decisions

4.4.1 Linked List



The Centipede object in my project is made up of a series of Segment objects. The first segment is the 'head' of the centipede and the last segment is the 'tail'. The 'head' segment is the most frequently accessed for collision checks and movement. The collection of segments is iterated over when calling the update and draw functions on each individual segment. I decided to use my LinkedList class to contain the segments. The main reason I used a linked list as opposed to any other data structure is due to how it is represented in memory and how easy it is to manipulate. Splitting the centipede into two (or more) separate centipedes requires destroying one of the segments and re-adjusting the other segments without losing order - which is a strong suit of the linked list structure. It allows fast and efficient manipulation of the segments which is what the game requires.

```
LinkedList<Segment*>* segments;     //The segments of the centipede
```

4.4.2 Dynamic List

At the beginning of the game (or a wave) there is only one centipede, but the player can shoot the centipede to split it into two (or more) centipedes. The CentipedeController class keeps track of all the centipedes in the game. I decided to use my List class, a dynamic list, to achieve this. This could also be done using a linked list but the manipulation of the elements is minimal. The only manipulation needed is removing a centipede when it is completely destroyed. Unlike the centipede and its segments, the list of centipedes is not required to stay in order - so removing one element from the list is just a simple *O*(1) task. Another reason I used a dynamic list is because the number of centipedes increases when a split occurs, so the size of the list must increase too. The number of splits depends on where the player shoots a centipede (shooting the head or tail will not result in a split). Overall, the use of a dynamic list seemed like a good fit for this collection.

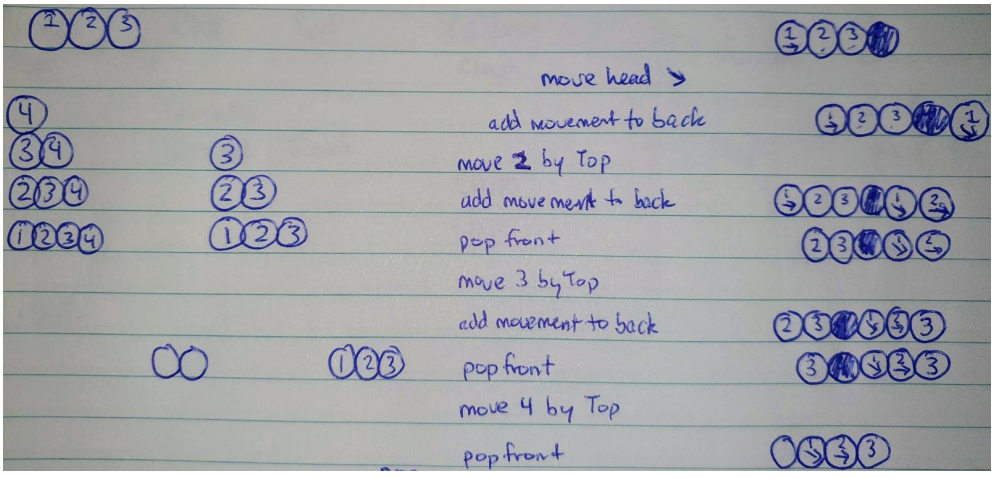
```
List<Centipede*>* centipedes;       //List of centipedes
```

4.4.3 Dequeue

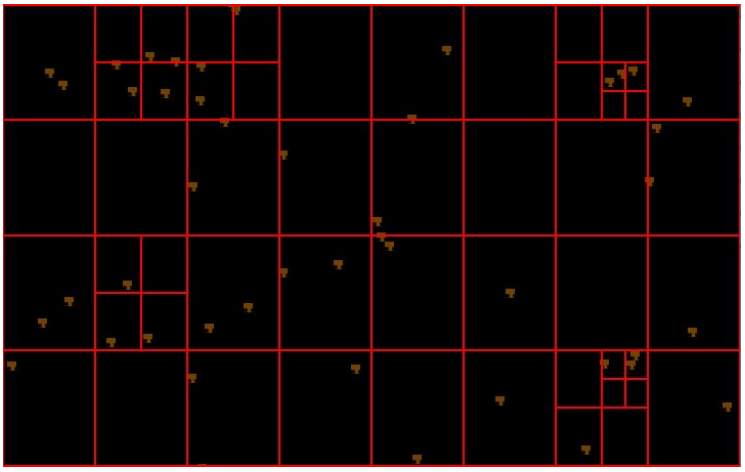
The original plan for the game was to use my Dequeue class to keep track of the positions of each segment for use in calculating movement. This method, however, failed to work and so I changed how movement works in the game to be on a timer. This is more akin to the original arcade Snake game where each segment will move a certain distance every x milliseconds. You can see the code for this below:

```
//Wait to move the segments
if (moveTimer > 0)
      moveTimer -= deltaTime;
//Move the segments
else
      Move(deltaTime);
```

Below is a version of the algorithm I came up with for movement using dequeues:



4.4.4 Quad Tree



Collisions in the game are checked using a brute-force solution. This is usually a no-no in games development because things can get laggy very quickly. In my case, the game is quite small & simple so it doesn't affect the performance unless there is an absurd number of centipedes on the screen. I could have used the quad tree structure that I had implemented - however, it is a simple implementation that does not support moving game objects, and adding it to my game would add extra difficulties which I want to avoid because the task is to make a simple game. Shown above is a visual representation of the quad tree in action. Below is a snippet of the code used for checking for collisions in my game. Notice how it uses the brute-force solution by iterating through each mushroom on the screen.

```
//Iterate over the mushrooms, check if they are colliding with the head
for (unsigned int i = 0; i < mushrooms->Size(); ++i)
{
	//Pointer to a mushroom
	Sprite* mushroom = (*mushrooms)[i];

	//If the mushroom is a nullptr or is too far away from the head,
continue
	if (mushroom == nullptr || mushroom->Distance(*head) >
mushroom->Radius() + head->Radius())
		continue;

	//Check if the mushroom is colliding with the head
	if (mushroom->CollidingWith(*head))
	{
		if (destroy)
			gameScene->mushrooms->Remove(mushroom);
		return true;
	}
}
```

5.0 Testing Strategy

5.1 Testing with Functions

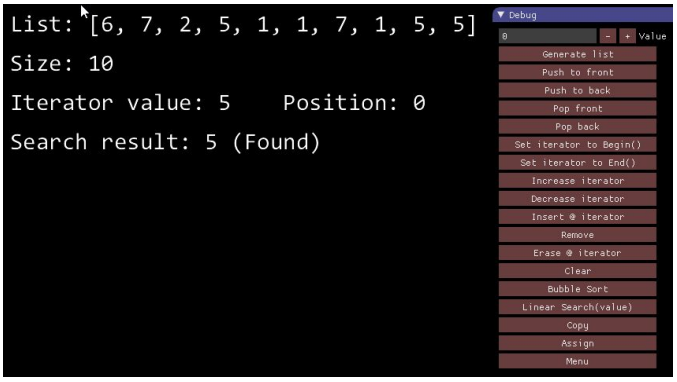
I implemented major features in separate branches using Git. Each container class was developed and tested in its own branch. When the container was complete and bug-free, it was then merged back into the master branch.

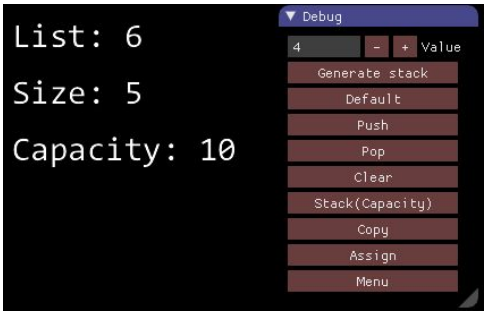
To test each container I instantiated it and tested all the different functions and combinations as unit tests to make sure it worked as intended. This also includes testing the copy constructor and assignment operator to ensure proper instantiation. Each container class was also tested for memory leaks using the CRT library. Below is an example of the function used to test the Stack class and the results in the console:

```
Stack<int> stack;              //Default constructor
stack.Push(20);                //Push to top of stack
stack.Push(40);
stack.Push(60);
stack.Push(80);
stack.Pop();
stack.PrintDetails();          //Print details

Stack<int> stack2(stack);      //Copy constructor
stack2.Pop();
stack2.PrintDetails();

stack2.Clear();                //Clear
stack2.PrintDetails();

stack2 = stack;                //= (assignment) operator
stack2.Push(100);
stack2.PrintDetails();

Stack<int> stack3(2);          //Overloaded constructor
stack3.Push(100);
stack3.Push(200);
stack3.PrintDetails();

cout << "List: " << stack << endl;  //<< operator
```

```
---Testing Stack
Size: 3   Capacity: 10   60
Size: 2   Capacity: 10   40
Size: 0   Capacity: 10   Empty
Size: 4   Capacity: 10   100
Size: 2   Capacity: 2   200
List: 60
---Finished testing Stack
```

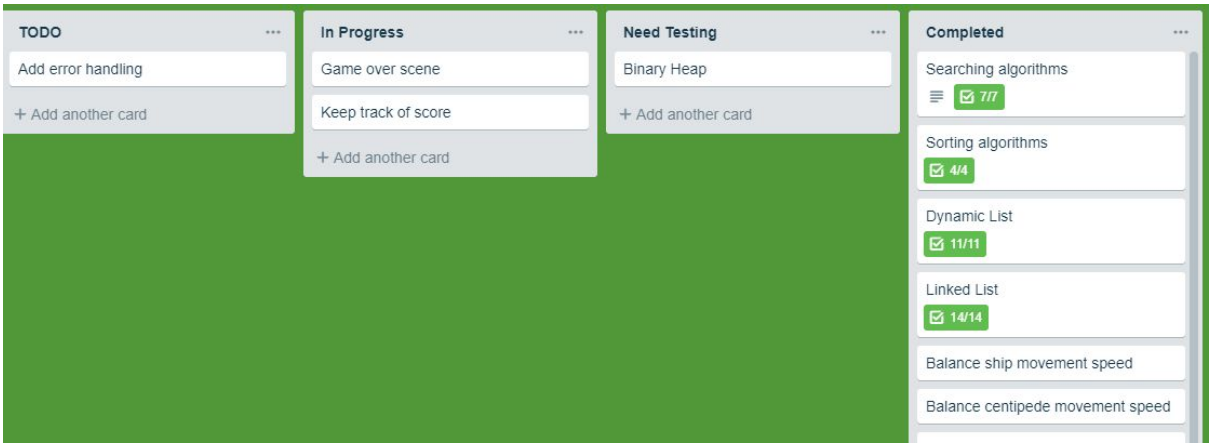
After implementing the Scene class, I was able to test any container more easily through a graphical user-interface, as seen below:





5.2 Trello Board

The main testing strategy for the game was to plan out all features and manage them using a Trello board. The board was used to develop ideas and to keep things organised as the project was completed.



5.3 Play-Testing

A document was designed detailing tests for peers to undergo. These were designed to test the important aspects of the game, i.e. making sure the game opens, can the game be lost, etc. Feedback and bugs discovered:

- Main menu “Centipede” title is bugged and will incorrectly display the text.
- Instructions text or an instructions menu is recommended.
- The centipede can get stuck at the beginning of a wave - with no solid way to reproduce the bug.
- Main menu buttons are off-screen if the window is resized before playing Centipede and then returning to the main menu - unable to reproduce the bug.

6.0 User-Interface Design

The UI mainly utilises the ImGUI library present in the AIE bootstrap project to easily create buttons and input fields. As shown in the images, the idea of the user-interface was planned to be simple - as it is not the main focus of the project. The final product is very similar to the mock-up.