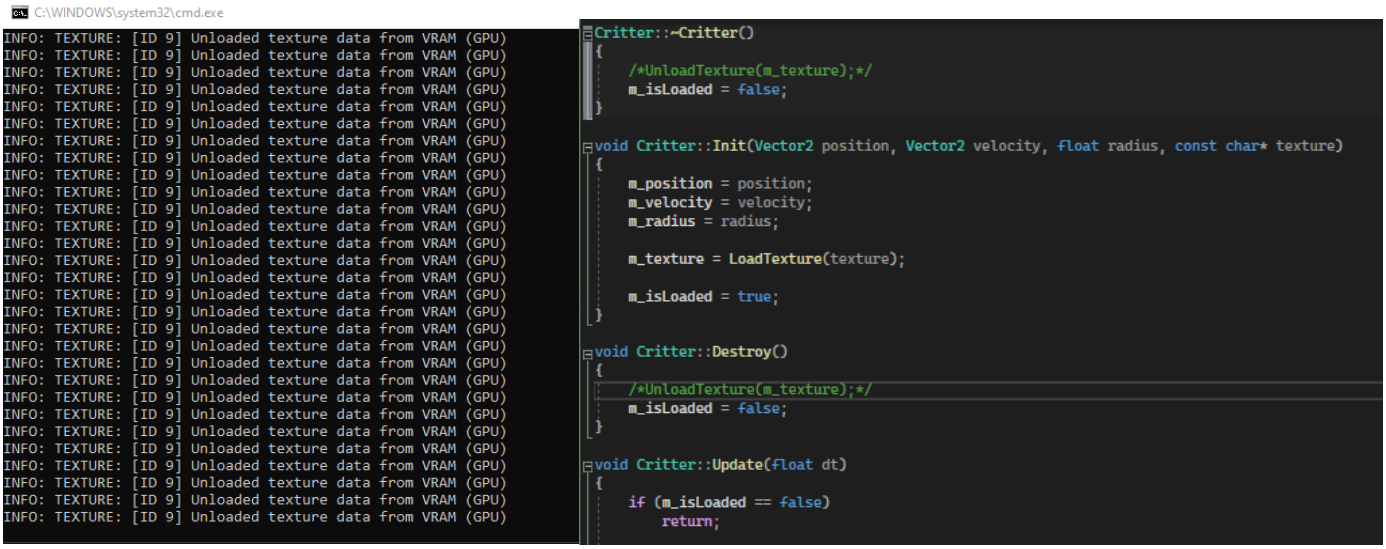
Task 1: Design Game Optimisations

Unloading Textures

Every time a critter collides with the destroyer it unloads the texture a bunch of times in the command prompt window. Commenting out these “UnloadTexture(m_texture);” functions stops the program from unloading the texture from the GPU heaps of times which makes the program more efficient.



Respawning Critters

```
        // kill any critter touching the destroyer
        // simple circle-to-circle collision check
        float dist = Vector2Distance(critters[i].GetPosition(), destroyer.GetPosition());
        if (dist < critters[i].GetRadius() + destroyer.GetRadius())
        {
            critters[i].Destroy();
            // this would be the perfect time to put the critter into an object pool
        }
    }
```

In the code above it checks if the critters are colliding with the destroyer by checking the distance between the radius of the destroyer and every critter. Instead of destroying the critter if it is collided with you could create an object pool to store the destroyed critter then return the critter back to the game after the respawn timer has finished. In the respawning code(below) it inefficiently reloads the texture again while spitting it out behind the destroyer. If we had an object pool set up this could be avoided. You could also set up the program to only load in the image once instead of re loading the image for each individual critter.

```
// find any dead critters and spit them out (respawn)
for (int i = 0; i < CRITTER_COUNT; i++)
{
    if (critters[i].IsDead())
    {
        Vector2 normal = Vector2Normalize(destroyer.GetVelocity());

        // get a position behind the destroyer, and far enough away that the critter won't bump into it again
        Vector2 pos = destroyer.GetPosition();
        pos = Vector2Add(pos, Vector2Scale(normal, -50));
        // its pretty ineficient to keep reloading textures. ...if only there was something else we could do
        critters[i].Init(pos, Vector2Scale(normal, -MAX_VELOCITY), 12, "res/10.png");
        break;
    }
}
nextSpawnPos = destroyer.GetPosition();
```

Critter-On-Critter Collisions

There is for loop inside a for loop checking for critter-on-critter collisions constantly while the program is running. This is operating at a (O)n^2 (quadratic) order which means that the more critters in the game the slower this function will run. This is considered a very slow algorithm. You could slightly improve this by setting the second for loop to “int j = i;” which would half the number of searches, but it would still be very inefficient. Essentially operating at a (O)n^2 * 0.5 order. If thousands of critters are inserted in the game a better method would still be required.

```
// check for critter-on-critter collisions
for (int i = 0; i < CRITTER_COUNT; i++)
{
    for (int j = 0; j < CRITTER_COUNT; j++){
        if (i == j || critters[i].IsDirty()) // note: the other critter (j) could be dirty - that's OK
            continue;
        // check every critter against every other critter

        // use of sqrtf() could eliminate
        float r2 = critters[i].GetRadius() + critters[j].GetRadius();
        r2 = r2 * r2;

        Vector2 diff = Vector2Subtract(critters[i].GetPosition(), critters[j].GetPosition());
        float dist = Vector2DotProduct(diff, diff);

        /*float dist = Vector2Distance(critters[i].GetPosition(), critters[j].GetPosition());*/
        if (dist < critters[i].GetRadius() + critters[j].GetRadius())
        {
            // collision!
            // do math to get critters bouncing
            Vector2 normal = Vector2Normalize( Vector2Subtract(critters[j].GetPosition(), critters[i].GetPosition()));

            // not even close to real physics, but fine for our needs
            critters[i].SetVelocity(Vector2Scale(normal, -MAX_VELOCITY));
            // set the critter to *dirty* so we know not to process any more collisions on it
            critters[i].SetDirty();

            // we still want to check for collisions in the case where 1 critter is dirty - so we need a check
            // to make sure the other critter is clean before we do the collision response
            if (!critters[j].IsDirty()) {
                critters[j].SetVelocity(Vector2Scale(normal, MAX_VELOCITY));
                critters[j].SetDirty();
            }
            break;
        }
    }
}
```

To improve this, we could use a technique called spatial hashing which gets objects from a 2D space and project them into a 1D hash table which allows for very fast queries on the objects. Objects are hashed every frame for real-time applications. The spatial hashing technique works in linear time O(n) which will search through critter collision much quicker than the current quadratic solution that is implemented. If we add more critters to the game, then the query time will still increase but not exponentially like it is currently. A diagram with spatial hashing shown below to show how it stores data in the table.

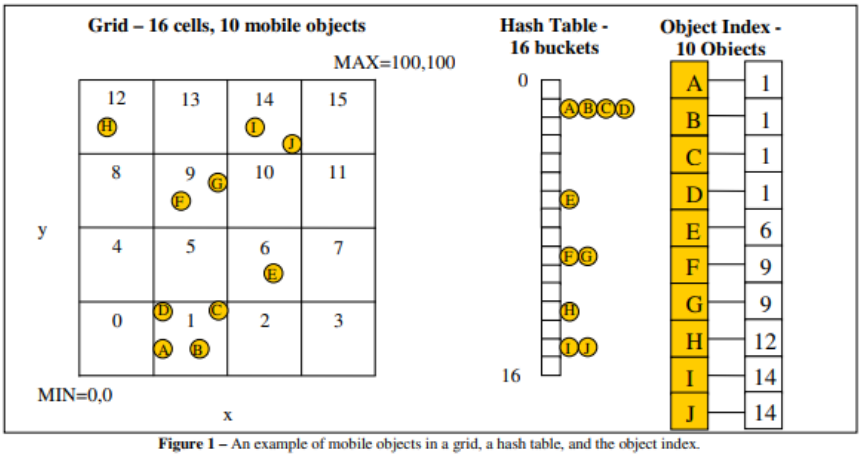


Figure 1 – An example of mobile objects in a grid, a hash table, and the object index.

Task 5: Debugging

Double Linked List Unit Tests

Test 1 - pushFront

For the pushFront test I created a list, then inserted the integers 1, 2 and 3(in that order) into the list. I then checked if the first item in the list is equal to '1' by using my 'begin()' function which returns the first element of the list.

Test 2 - popFront

For the popFront test I created a list, then inserted the integers 3, 2 and 1(in that order) into the list. Finally, I call the 'popFront()' function on the current list. I then checked if the FIRST item in the list is NOT equal to '3' by using my 'begin()' function to check the first element and make sure it is removed.

Test 3 - pushBack

For the pushBack test I created a list, then inserted the integers 3, 2 and 1(in that order) into the list. I then checked if the last item in the list is equal to '1' by using my 'end()' function which returns the last element of the list.

Test 4 - popBack

For the popBack test I created a list, then inserted the integers 3, 2 and 1(in that order) into the list. Finally, I call the 'popBack()' function on the current list. I then checked if the LAST item in the list is NOT equal to '1' by using my 'end()' function to check the last element and make sure it is removed.

Test 5 - countList

For the countList test I created a list, then inserted the integers 3, 2 and 1(in that order) into the list. I then check if the number of elements in the list is equal to '3' by using my 'count()' function which returns the total number of elements of the list.

Test 6 - insertAtLocation

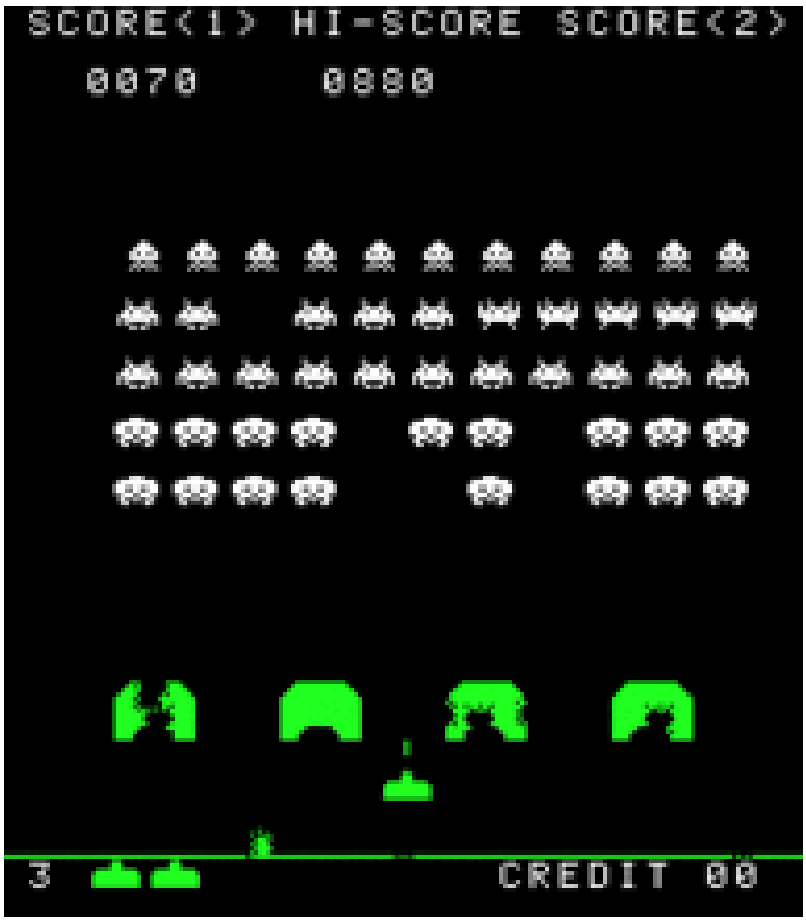
For the insertAtLocation test I created a list, then inserted the integers 3, 2 and 1(in that order) into the list. I proceed to insert '4' at node 2 in the list. I then checked if the element at node 2 in the list is equal to '4' by using my 'getAtPos()' function which returns the element at a desired position in the list.

Task 7: Design a Classic Game – Space Invaders

Description of the Game

Gameplay in Space Invaders is relatively simple. The player controls a small ship that can only move laterally across the bottom of the screen and fires vertically. 5 rows of 11 aliens each advance slowly from one side of the screen to the other, dropping down one space and reversing direction when they reach either side. The player's task is to acquire points by eliminating enemies and to destroy all the aliens before they reach the bottom of the screen and complete their "invasion." As aliens are destroyed, the speed of the remaining enemies increase, as does the tempo of the music. Once all the enemies are destroyed, the wave resets and the difficulty increases.

The Invaders constantly shoot back at the player as they advance from side to side across the screen. To help avoid their attacks, the player can hide behind a few destructible barriers or "bunkers" near the bottom of the screen (four in the original version). Occasionally a "mystery ship" will appear near the top of the screen and move quickly from one side to the other. Destroying it rewards the player with a sizeable point bonus.



Programming Patterns

Using **command pattern** on the bullet object. The bullet object will create a bullet, check if it's a player or alien spawning the bullet then update the movement based on its type, if it is the player it will move forward, if the enemy it will move backwards on the Y-axis.

A **factory programming pattern** could be used to create the other game objects in the game e.g. player, aliens and bullet. This helps to avoid the problem of complex object instantiation by keeping it all in a single place rather than scattering it around in the code.

Flyweight programming pattern could be used to store the shared aspects of the aliens (textures and animations) away from the individual aspects (position and isAlive). More efficient as you can reuse the shared aspects instead of creating them for 55 aliens.

Data Structures

For all the enemy aliens I would store them in a double linked list. I believe this would be beneficial to remove individual invaders from the game. When the player bullet collides with an invader you could call the remove function for the double linked list and pass in that invader to be removed.

A hash map is also used when implementing flyweight programming patterns so that might be a better way to implement the enemy system rather than storing all the aliens in a double linked list. Then use a remove function to remove the alien that is collided by a player bullet.

Algorithms

Bullet Spawning – when the player presses space bar the bullet spawns and moves forward with speed multiplied by delta time, also has a small cooldown in between shooting. For the enemy the bullet will spawn randomly at one of the aliens at the front of a column then move backwards from the alien at speed multiplied by delta time.

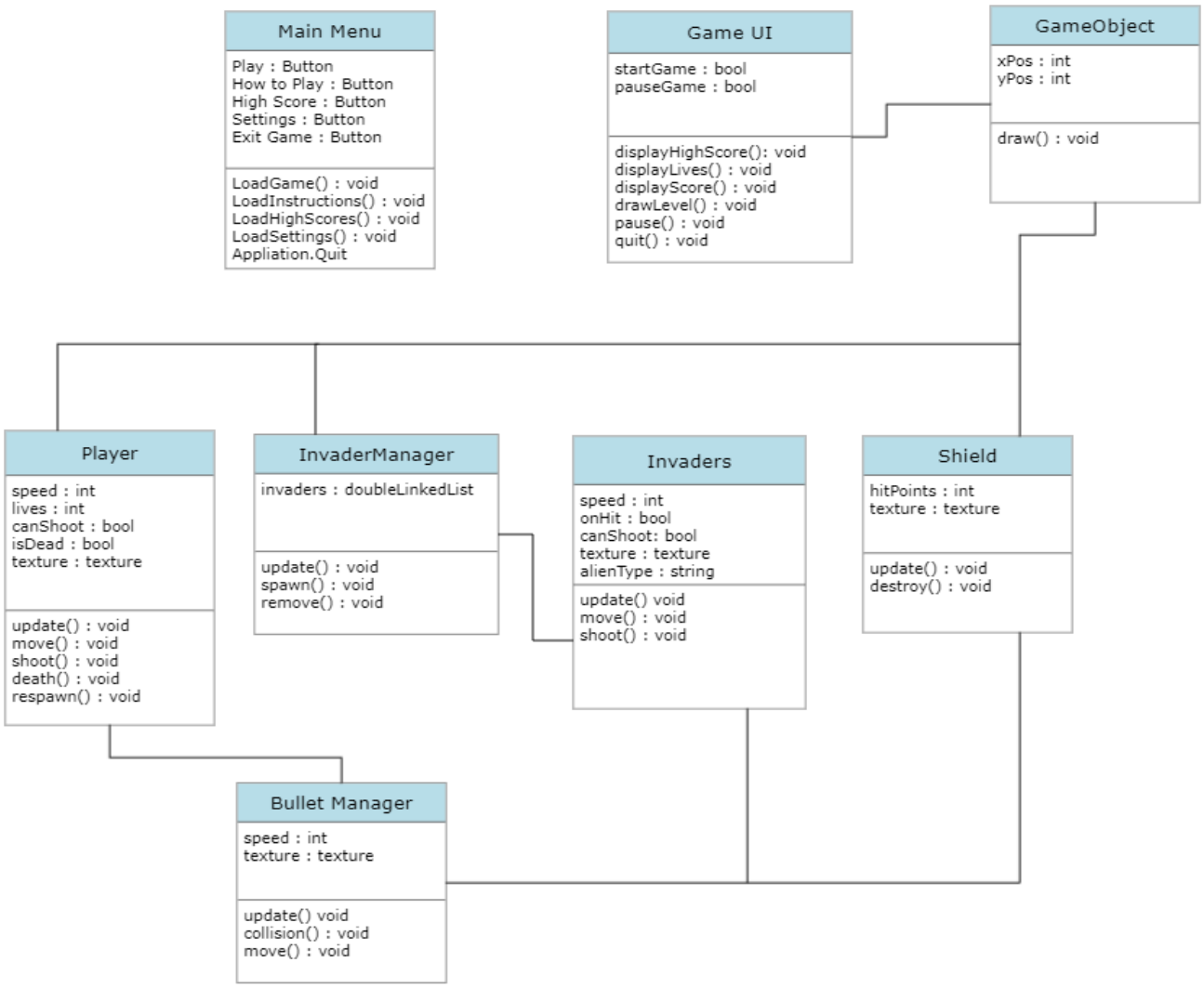
Bullet Collision – When a player bullet collides with an enemy the remove function will be called to remove that alien from the double linked list. When the enemy bullet hits the player the game pauses, checks if player has any lives, the player is destroyed and loads another player then resumes game. If the player is out of lives, it is game over. If the player or enemy bullet hits the shields, then a part of the shield is destroyed which means the player has less cover from incoming bullets.

Enemy Movement – All enemy aliens will have an x and y position and sprite that is stored in the double linked list. Will have to update the position with speed multiplied by delta time for each alien as they move side to side and down towards the player. It should get easier to update all enemy movement as the game goes on as there will be less enemies once the player starts shooting them.

Player Movement – The player will need a speed variable, X and Y position. The X position will need to be updated depending on if the right or left arrow key is pressed and bound the movement to the edges of the screen.

Score System – The score text will be updated when a player destroys an alien which will feed in an int value and add it to the score total. The score value will be saved if it's higher than the previous high score then displays in the high score text from then on until it is beaten.

Class Diagram



Testing Strategies

Functionality Testing – is done to confirm whether the end product works to the specifications. This form of testing mainly has the testers looking for generic problems within the game or its graphics and user interface e.g. game assets, stability issues, AV issues and game mechanics issues.

Performance Testing - is used to determine the application's overall performance under real-time scenarios and load. Conducting this type of testing helps to ensure whether the present infrastructure allowing the smooth functioning of the game.

Ad-Hoc Testing - is a less structured way of testing and it is randomly done on any section of the gaming application. Specifically, there are two distinct types of ad hoc testing. This kind of testing works on the technique called "error guessing" and requires no documentation or process or planning to be followed. Since Ad hoc testing aims at detecting defects or errors through a random approach, with zero documentation, errors won't be mapped to test cases.

Report and Repair – Throughout these testing procedures all bugs and errors need to be documented in a report. The development team can then sit down and come up with solutions to fix these bugs/errors. Multiple solutions might need to be implemented and tested to see which solution performs better.

- Score and high score displayed at top in text.
- Mystery ship in orange floating above other aliens.
- All invaders in green and white slowly moving side to side and down the screen.
- Shields placed between player and aliens as green domes for player to hide behind.
- Player underneath the shields as rectangle shape object.
- Line at bottom for enemy bullets to hit and be destroyed.
- Lives displayed in text and image form.