2nd Year

HYPERCOSMIC LITEWAVE

Game engine construction ica

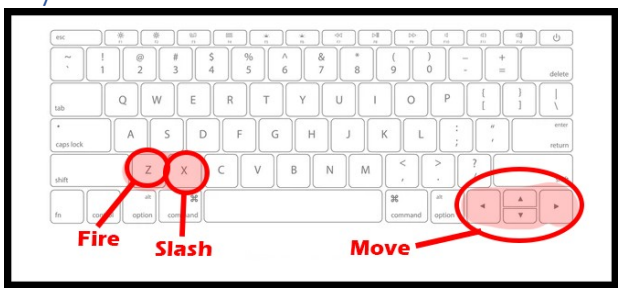
KHAN, AMIN (B1082339)

Contents

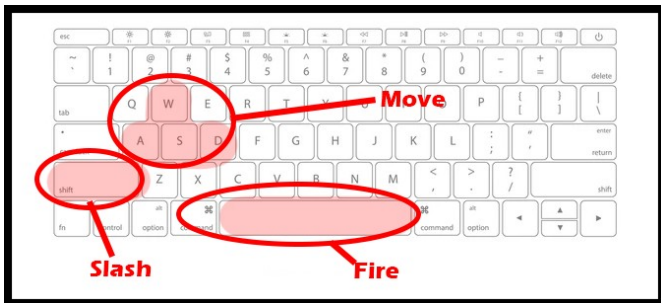
User Guide

Controls

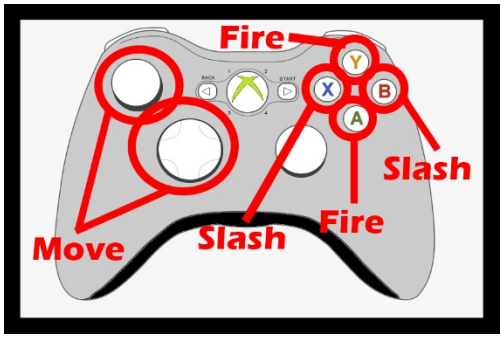
Keyboard Controls



Keyboard Controls (alternate)



Xbox 360 Controls



Keyboard:

- MOVE PLAYER: Arrow Keys or WASD
- FIRE: Hold down Z or Spacebar
- SLASH: X or LShift

Xbox360 Gamepad:

- MOVE PLAYER: Left Joystick or Dpad
- FIRE: A or Y Button
- SLASH: B or X Button

- Hold and release SLASH for HYPER SLASH
- Menus navigated via the mouse

Game rules:

This is a top-down vertical shoot-em-up game. Control your ship in all directions and dodge oncoming bullets fired from enemies. Hold down the FIRE button for a continuous rapid ranged attack that reaches across the whole screen. Press the SLASH button to perform a melee attack with high damage in a short range in front of you. Hold down the SLASH button for a while, then release to use the Hyper Saber, a wider-ranged more powerful version of a regular slash.

Some enemies fire Dark Bullets (see below).

These can be hit with your saber turn them into Light Bullets, which deflects them back at enemies. Light Bullets deal a lot of damage, and enemies destroyed with them will yield more points.

The two-player mode allows for a second person to control another ship with a connected Xbox360 controller while the first player plays with the keyboard. Alternatively, you can change the settings for which player uses which input device in the options menu.

When all players in the game lose all their lives, the game is over.

Bug Log (Listed in order of fatality, high to low)

- A 'random' softlock bug exists where the entire game permanently freezes during gameplay. However, during development this occurred at completely random times and could not be replicated intentionally, thus the cause of this bug is unknown.
- One time I tried exiting the game via the X button and then the game just kept stuttering and didn't close. This only happened one time so I'm not sure if it's a code problem or a one-time thing, but still worth noting.
    - Ok never mind it happens every time you try to close the game that way, however only in release mode. I tried adding destructors but this didn't help.
- Due to the game using a deltatimed update system, platforms that run the game at a slow enough FPS rate means that gameplay errors may be caused by some functions that update independent from platform capabilities. For example, a player bullet may skip over the enemy if the DT causes it to move in such a large increment that it completely misses the collider directly ahead of it.
- Dark Bullets that are hit with the hyper slash are intended to move with very high velocity, however because of this they may not bounce off the sides of the game screen as intended due to them moving so fast per update that they miss the collision check. This happens mostly when the game is running on a slower platform.
- When navigating menus, a button's event will still be activated if the player holds down the LMB when the cursor is not within the bounding box of the UI button, and then moving the cursor within the button's bounds. This causes some issues like a button being activated immediately after entering a new menu.
- Stereo panning on sound effects is noticeably 'unsmooth' and have very clear cutoff areas the make it sound like certain sounds are jumping from position to position across the screen
- If debug collisions are switched on to make colliders visible, colliders that belong to a sprite that has moved into the negative world axis will be displayed as slightly off position relative to how far the entity has moved off-screen. This is purely a display issue as collisions still work within the expected ranges.
- Explosions render above debugged colliders
- When an enemy is destroyed with a Light Bullet different coloured popup text should appear displaying the increased score as opposed to the regular pink score popup. This happens, though the regular pink popup text displaying the regular enemy score yield is also rendered below the bonus score popup.

Implementation Checklist

Adequate Rated Items (D to C)

Graphics

- A 'black boxed' graphic system is in place
- Textures can be efficiently drawn to arbitrary positions on and partially off the screen (clipped)
- Animation is implemented and working correctly

World State

- A player entity exists
- Input is recognised and can be used to alter the world state e.g. move the player entity
- The Xbox controller is supported and 'hot pluggable'

Code Quality

- Class interfaces are minimal and complete. Class function and member variable visibility is correct
- Code can be built and executed without compiler errors or warnings in debug and release configurations.
- Code is well commented
- There are no memory leaks There is good error handling throughout
-

Good Rated Items (C to B)

World State

- A world model system is in place. It is separate from other code and black boxed
- There is a game loop handling input, world update and rendering
- Bounding rectangle collisions are detected
- There are multiple world entity types

Code Quality

- Good use of object-oriented techniques e.g. polymorphism, member variable visibility
- Memory is only allocated / deallocated outside of the game loop
- Const is used correctly

AI

- Some AI routines are in place e.g. enemy entities move around the world following paths, use state machines etc.

Other

- Some sound effects are in place

Excellent Rated Items (B to A)

Graphics

- Interpolation is used to smooth entity movement (not needed due to deltatiming system)

World State

- The player entity can shoot projectiles (or equivalent functionality)
- Explosion and bullet management
- Game play is independent of platform capabilities (i.e. uses a model tick approach) (GAME USES DELTATIMING)
- Game cycling e.g. detection of win / lose conditions and restarting the game
- There is a scoring system with the score shown on screen

AI

- Several different enemies with differing behaviours

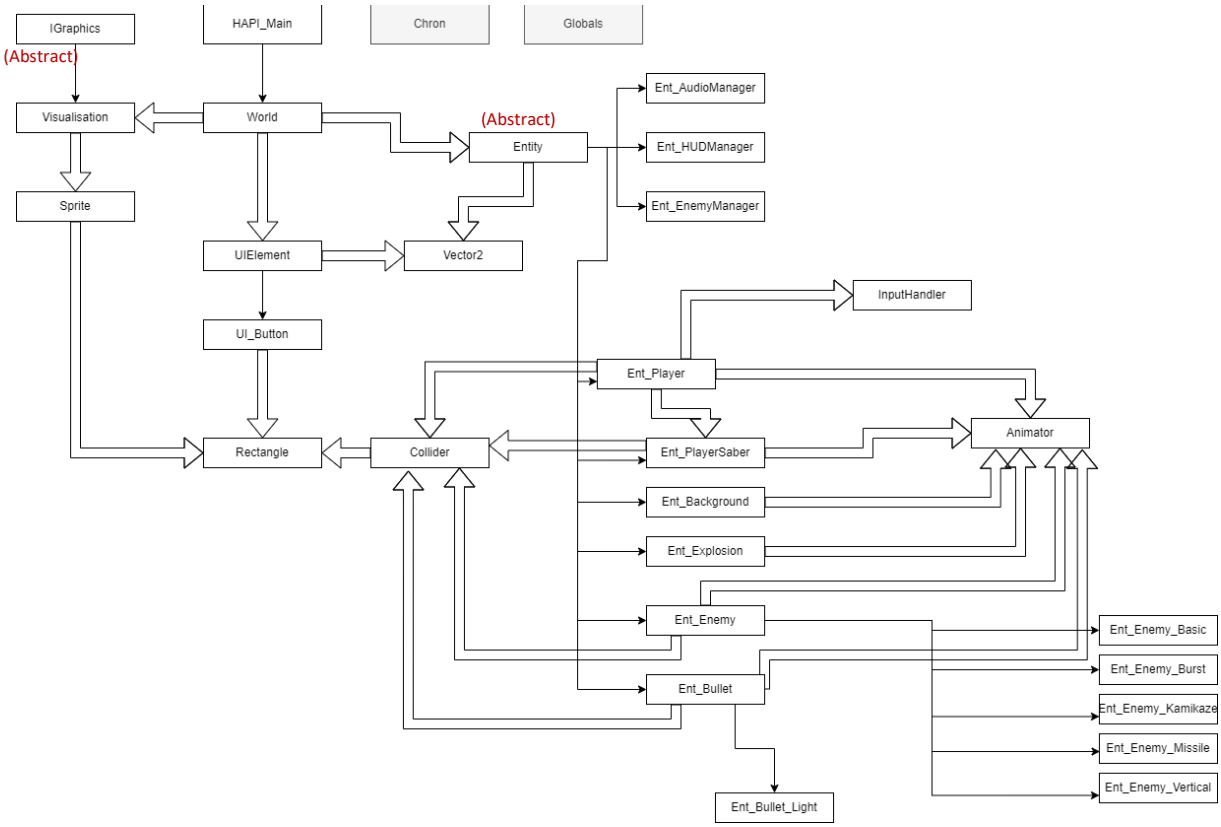
Extra Marks (Examples)

- Mapping of world space on to screen space
- Other graphics techniques have been implemented e.g. background scrolling, blending modes etc.
- Level data is loaded from a file
- Difficulty levels.
- More advanced C++ e.g., use of namespaces, STL, C++ 11 and patterns (Don't know)
- 'Intelligent' enemy behaviour (Not majorly complex state machines)
- There are sound effects for collisions, explosions and firing
- Additional black box systems have been implemented e.g. for AI, Sound
- HUD features beyond simple text e.g. health bars, mini maps etc. (Button event system)
- Other features, please list below:
    - Full UI system with working buttons
    - Sprite effects and filters (I.e, negative, transparent, boosted rgb channels)
    - Timescaling system for hitstop/slowdown effects

- Fleshed out animation system using XML
- Debug mode for viewing hitboxes
- Music
- Stereo audio

Maintenance Guide

Full UML



Black arrows show inheritance, white arrows show inclusion/ownership.

As seen here, Entity is the main class that World includes a vector of. Entity itself is abstract but all of it's derived child classes are what make up the main game. Some Entities have further children, such as the Enemy class and its many deriving enemy types.
Derived children follow naming conventions, usually a prefix based off the parent class name.

There are cases where some entities would need to store references to others, but this would only ever be done through pointers assigned in the World setup function. Entities are only ever created in World, never in an Entity class itself.

Other classes without a naming prefix (Animator, InputHandler, Collider, etc.) are component classes that store data used by Entities.

This isn't shown properly, but InputHandler is technically owned by World and not Player, as this is where inputs are updated from. The diagram displays this because this is not how it should be, a better and more encapsulated improvement would have been for the player to own its own InputHandler and update it from within it's own Update function.

Chron and Globals are singletons and store things needed by all classes like global timescale and game difficulty level.

Visualisation System

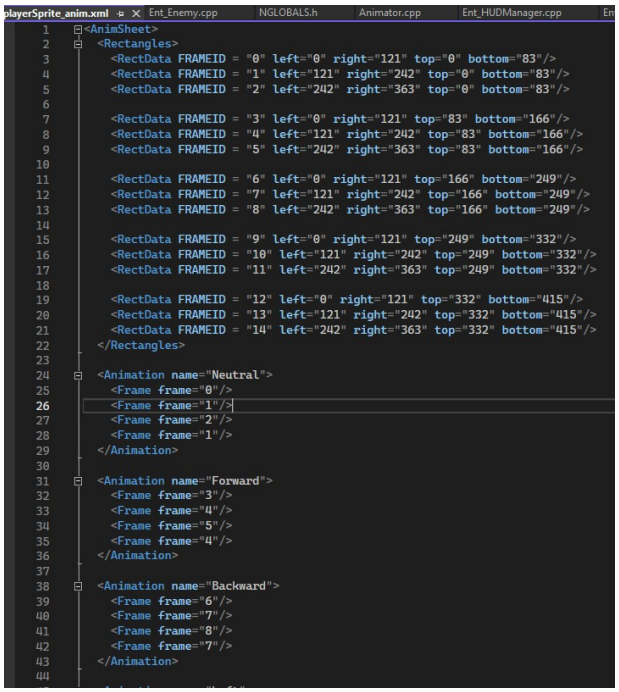
The visualisaton system in this project is completely blackboxed and separate from the World Model. The main Visualisation system derives from the IGraphics interface, which allows other Visualisation classes to be created based off other graphical APIs. It contains the code for loading texture data as sprites, then drawing them with parameters such as position, animation frame, and special sprite rendering effect. Sprites are stored in a map with a string key, making them simple to obtain from outside the class.

The world model only ever needs to have a reference to one of these systems at any time, and is only stored elsewhere via pointer in some entities that need to set their own animations. Because there is no way for this to be done without a reference to the visualisation system, this makes the class slightly less encapsulated. The addition of an animation management class may have helped with this.

Additionally, there are other special blitting functions, like for rendering scrolling textures and drawing sprites with no transparency quickly with less calculations.

Some features of this system are limited to this game only, as the enumerators used for things like effects and scrolling are defined in a global header the other classes need to reference. These enums would need to be transferred to other projects.

When creating a sprite in the visualisation system, you can optionally load in animations for this sprite by passing in the file path for an XML spritesheet file. This XML file contains data for the rectangles that define the different frames of the sprites, and additional nodes that contain which groups of frames make up different animations for that sprite.



(Though these spritesheets can be created manually, there exists dummy code in main.cpp that is commented out, but was used to automatically generate spritesheets with a large number of frames to save on time)

World/Entity Model

This system is the main meat of the engine's architecture. All data used during gameplay is stored within classes that derive from the abstract Entity class. The World class contains a vector of every entity in the game (this is the object pool), and loops through them each frame to update and render them.

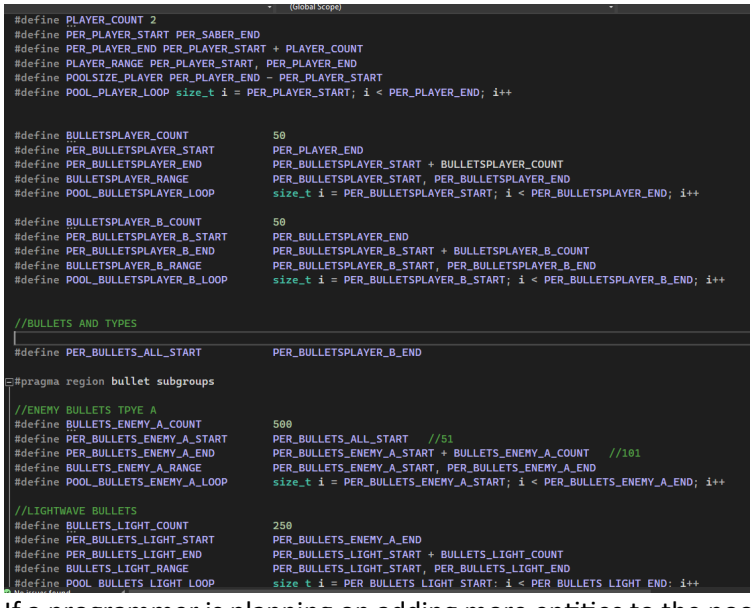
All entities have Load, Update, Render and LateUpdate functions that must be defined in children classes. 'Update' is where the entitiy's active code during gameplay is. 'Render' simply calls the graphics system to draw the entity's visual, and only needs to be overridden if an entity has some sort of special graphical modification. 'LateUpdate' is similar to Update but is called after the render function, so it is used when an Entity needs to do something on top of the current buffer (e.g. Drawing bounding boxes in debug mode)

Load() was intended to be called to set up an Entity when created, but went almost completely unused in favour for putting most of this code in constructors and manual setters, due to Load() not allowing any custom parameters.

Entities have an important 'isActive' public variable. When this is false, it is skipped over in the game loop so it is not rendered and the Update function is not run, saving memory.

Some entities need to store a pointer to the main object pool if it needs to spawn/activate other entities (i.e. Players spawning bullets or enemies spawning explosion effects). When a certain entity class needs to be obtained,

NGLOBALS.H stores macros that define all the information for the locations of different entities within the pool. These range values make it easier to obtain an entity of a specific class.



If a programmer is planning on adding more entities to the pool, the adequate range macros must be added into this header file. Also to note, if inserting the macros between groups of macros for other entities, the group below must have the definition of their '_START' macro changed to the '_END' macro of the new group, as you'd be changing the start index of the entities higher up in the pool.

It is also extremely important that during the memory creation stage, all entities are created in the exact same order that they are listed within the NGLOBALS file, and the number created are the same as the '_COUNT' macro for that class (it is recommended to do this in a for loop when possible).

```
for (size_t i = 0; i < BULLETSPLAYER_COUNT; i++)
{
    Ent_Bullet* newBullet = new Ent_Bullet("bullet_player_A");

    newBullet->Load();
    newBullet->SetColliderRect(rect_bullet_player_A);

    m_entities.push_back(newBullet);

}
```

The Entity base class has a function that takes 2 integers that act as a range to loop through the object pool, and when combined with these macros it makes obtaining entities simpler.

```
Ent_Bullet* bullet = CAST_BULLET(RetrieveEntity(all_entities, BULLETS_ENEMY_A_RANGE, true));
bullet->PrimeBullet(m_posX + fireOffset.x, m_posY + fireOffset.y, 0, 1, 600);
```

This part of the system is fairly delicate due to there being little room for error in the NGLOBALS header. However, when used correctly it acts as the main cornerstone for the architecture of this game engine due to how it adds so much simplicity to coding gameplay. It also rids of the uncertainty of whether casts will fail and prevents the need for multiple pools. Although the macro definitions are very exclusive to this game demo specifically, when the time is taken to redefine them for other games the advantages of this pooling system still carry over.

Animator System

The Animator class is a component added on entities that use sprites with animations. Every entity with an Animator updates the animation timer every loop. The Animator's 'frame' and 'CurrentAnimatonName' variables and functions are used in the Entity's render overload to display a certain frame from their graphics spritesheet. This happens independent of the game's timescale as it divides the deltatime value by the game speed when passed in.

The component is quite linked to the visualisation system, as it relies on the way that animations are stored for the Sprite class to correctly update animation frames. This keeps the class small and simple for this game demo but may make it less compatible with other graphic APIs.

Input System

The InputHandler class is a very flexible, completely blackboxed system that allows you to define 'Input Maps' for different actions (i.e, left, right, jump, attack), and map various keyboard or controller definitions to each one manually in a private array or vector. It has signals for the frame an input button is pressed down, the frame it's released, and during the time it's held down. Each one also has a rumble feature that's used if the player is using a compatible controller.

Entities that have an InputHandler reference, such as the player, can check for these signals in their own update function, as well as call for rumble functions from the handler.

InputHandlers are stored by the world class and are handled before the entities update.

This system is transferrable as long as it uses HAPI, otherwise the macros that define hotkey values and the way controller/keyboard data is obtained/used need to be replaced.

Collision System

This is a component that can be prompted to check for overlaps with other instances of itself. A collider is defined by a pointer to a Rectangle class.

Collision checks are called from the world loop after the entities update and render.

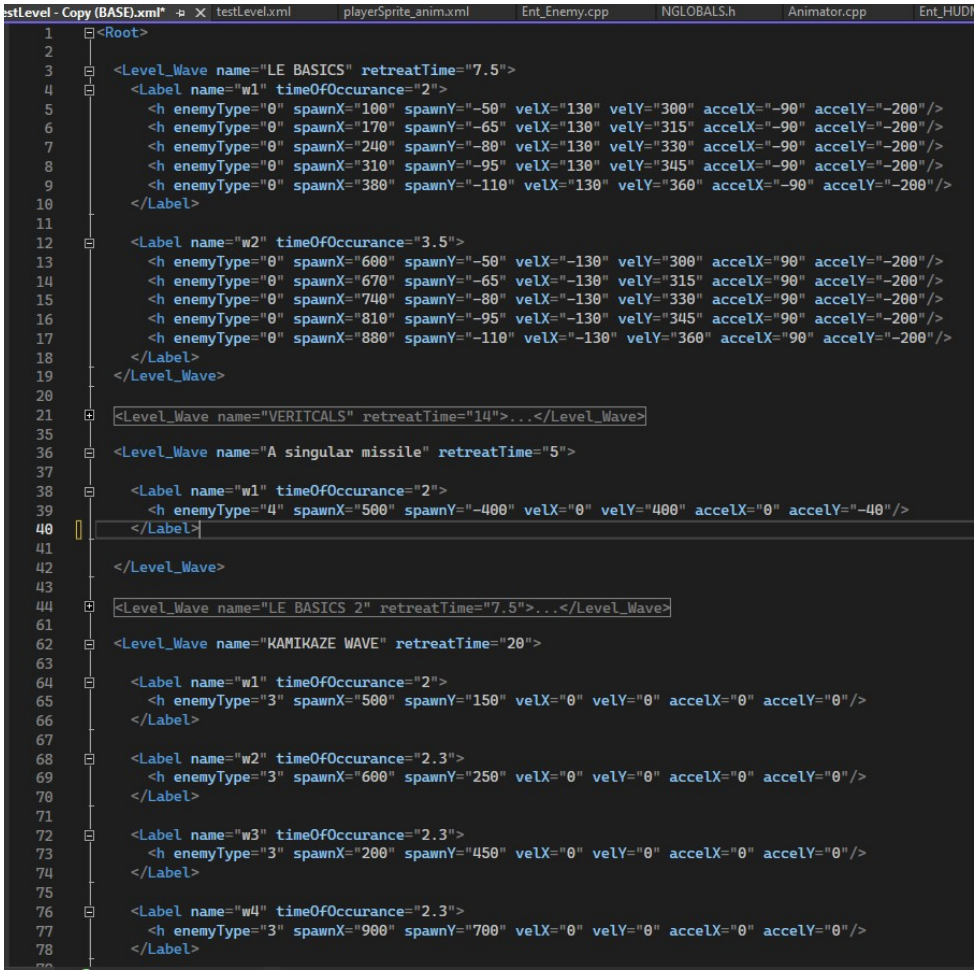
It is recommended to use the pool ranges defined in NGLOBALS.h to check collisions against a certain type of entity during a collision check function.

This system is completely transferrable to other C++ projects due to all of its calculations being fully encapsulated and never referencing any class other than itself, sans from the code that allows it to draw to the screen when debugging is on.

Level Loading/Enemy Manager System

The game demo loads in waves of enemies via a level XML file. The nodes in this file are grouped into 'waves' which contains nodes of enemies to be spawned in at a certain time during gameplay. Each node defines the IDs of which type of enemy to be spawned, their start position, as well as velocity and acceleration to define their movement pattern.

This makes it very easy to create, modify, and test levels for this game and others similar to it, but may require some modification of the spawn parameters for games that aren't in of the shoot-em-up style.



UI System

This system is completely separate from a majority of the game as it used mostly in the main menu. It is made up of 'UIElements', which are similar to Entities in that they both have update and render functions. UIElements and children of the class follow a parent hierarchy system, where they store pointers to other UIElements and have positions relative to them. This makes it much simpler to group and move around groups of buttons and images. UI_Button is a child of UI Element, and all UI_Buttons are stored in a vector during the main menu loop, which is used to check for mouse inputs. An integer is used to store the 'id' of the button clicked during a frame. A switch statement in the menu loop determines what function to carry out based on this id.

Buttons currently don't store any data relating to the game or menus within them meaning that this requires the switch statement in the menu loop to be very long and hard to read, but since the UI classes are separate from the main game, they can be easily recycled for other projects.

Conclusions

The work I created during this module, despite not usually the type of thing I like to make, is some of the work that I'm the most proud of during my course so far. Although I initially had some disdain for working with the really low levels of C++, such as when coding the graphics system, once that was all out of the way I found the flow of working with a system I created in its entirety to be quite satisfying and enjoyable on a level comparable to creating a game in a contemporary engine (even when going back to modify the graphics, I managed it with ease due to knowing how it worked).

A sort of recurring motif for me during this module was doing more with less – or, creating more complex systems with less complex code. I chose to make a space shooter game over something like a platformer as I knew it would be very simple, but would allow me more opportunity to safely play with adding features.

Early on in the module I had planned on trying to implement rotations, and I got half way through this before having to give up due to various bugs. Eventually when I started implementing animation, I'd realised that this was something that was a good thing to ditch as it would've added much complexity to defining things like spritesheet frames and collision boxes. Realistically, I could've either had one or the other, and I'm glad that I picked animated sprites over rotated ones as I could do more things with that in less time – even fake rotations with animations).

Since the part I enjoy about game development is designing gameplay and aesthetics, I spent a lot of time adding mechanics that contribute to good game design and making the game feel satisfying to play. For example, when the player deflects a bullet I programmed the game to slow down for a short period to let the the moment be absorbed by the player. I also modified the graphics system to allow sprites to be rendered brighter so enemies could flash upon taking damage, made a haptic system for controllers, and added many more smaller features to provide feedback for gameplay.

I believe that the primary strength of the final product is how I handled the object pool and looping through entities with macros. This took away a lot of the tedium and convolution that would've come entities having to search through the pools with arbitrary values that'd need to be changed if the pool order was modified.

The blackboxing technique we were taught was crucial to the creation of my game – not just with the visualisation, but making sure to encapsulate other systems like collision and input ensured that bugs were contained. A good example of this is when I realised there was a bug in the animation system that caused all entities with the same sprite to use the same frame and not the frame of their individual animators. I knew that I only needed to change something in the Animator class. Blackboxing as much as possible made the coding process miles simpler, and it's also added some longevity to what I've created, as these systems can be transferred to other projects easily.

I found learning about the low-level graphics system tricky at first, but with help from my lecturers I managed to understand how bytes were used in the context of writing data to the screen. As a result, this gave me a deeper understanding of graphics programming and how to create interesting effects for the sprites, such as the negative colour filter I added.

The project fails slightly when it comes to code architecture. There are a few circular dependencies in certain classes where I forgot to use prototyping and have my code in the cpp files instead of including header files in other header files, visible in the UML diagram that VS generates. This is particularly dangerous with some classes that included the Visualisation header, as the graphics system is intended to be entirely blackboxed. This is visible in the Animator class, where it needs to be passed a Visualisation pointer from an entity to set an animation. A better way to handle that would be to have a single animation manager with a Visualisation reference.

I also neglected memory management and deletion throughout the whole project, which was a huge mistake as by the time I started getting occasional fatal crashes from the game (possibly caused by leaks), the codebase was too massive to have the time to go through. Adding destructors in classes is something that I should've considered more, especially since they're such a big part of C++.

For improvements, I definitely wish I could've added a more fleshed out level with more enemy types and patterns, and implement a win game state so the player has something to work towards rather than just survive. A highscore leaderboard would've been a nice way to add replayability to the game too, making it feel less like just a prototype. Something a bit more complicated that would be a good addition would have been a render order list, because currently entities that are further in the object pool have their sprites drawn above those that are lower. This would get rid of the need for certain entities to be added in a specific order.

In conclusion, I enjoyed the game engine construction module much more than I initially expected, especially when I got to the point where everything was up to me. It was a very unique experience to be entirely in control of an engine. Whenever an error came up, instead of going to the internet to find out how to solve a problem, I knew that I never needed to do that at any point because I was entirely responsible for everything in the code – meaning I already knew what the answers to any problem that came up would be. Despite working with some complicated code, I think I've definitely made progress on one of my personal goals to not overcomplicate things, as evidenced by me opting to abandon certain complex additions in favour for having several smaller polished features that add up to equal value. This is a characteristic that will strengthen my effectiveness as a creator and help greatly in future professional and personal projects.

Asset References

https://untiedgames.itch.io/five-free-pixel-explosions
https://www.youtube.com/watch?v=ZQo8YaG1hhs

Other than these two things, all assets in the game were created by myself (Regular explosion animations were created by an explosion texture generator XNA project contained on a disc drive with a book I found in the library. I don't know how to cite that but technically I don't need to)