**GCSE Computer Science (7517)  
  
The Practical Project**  
  
**“Enter the Dead Zone”**

**Analysis - Overview**

# Project Background

In the last few months, I have been playing a lot of various games during my breaks and free times with a couple of friends. Most of these games feature local multiplayer or online multiplayer, however due to the restrictions of being in a school environment we are limited to local multiplayer games. As a group, we have gotten bored of many of the games we have played and thus, in this project I want to create a fun and engaging multiplayer game that will keep us entertained for a while longer.

With the production of this game I want to also solve a multitude of accessibility problems that we have had when trying to setup many of the already available games:

1. **Controller Support:** Many local games lack good controller support such as the ability to rebind buttons preventing the use of SNES USB controllers which do not have analogue sticks. This is problematic as not everyone owns the standard Xbox / PS4 controller.
2. **Lack of Controllers:** Sometimes there are not enough controllers to provide to everyone who wants to play causing people to be left out. This is mostly due to the lack of remote play support for most multiplayer games. Since everyone normally has their laptop with them the lack of controllers would not matter with remote play as one person can simply host the game and let others play via remote play on their own devices.
3. **Performance:** Since many of us do not have high-end laptops we are unable to play performance heavy games.

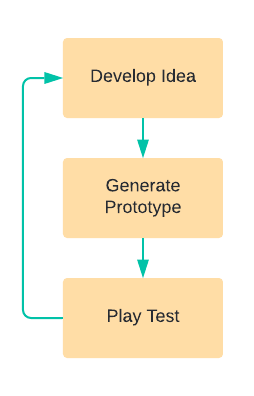
# Project Outline

As described above, I want to create a multiplayer game that fixes many of the accessibility problems of the currently existing games. In this project I also intend on improving replay ability and modding capabilities. This is because many existing games have become stale due to the lack of variation with each playthrough and the inability to change some features to make the game more interesting on a second playthrough. My clients for this project are going to be my group of friends that I play with.

**Analysis - Research**

# Secondary Research

When making a game, many developers have an initial idea or plan and they build upon this idea through progressive play testing. One way to develop on an initial idea is to *“Follow the fun”*. This is a methodology to making games where the developer should ignore their original plans and preconceived ideas and instead look to their game (first prototype) to find where the development should lead. For example, the game *Into the Breach* was originally intended to be a turn-based strategy game like *Xcom* where the player would have to manage their squad and resources to defeat their enemies. However, the designers needed something innovative as they were otherwise making a very generic tactics game. To do this, they decided to show the player the intentions of one of their enemies allowing the player to make decisions based on what that single enemy was going to do next. When play testing they found that this single mechanic was the most fun and interesting part of the game and so as Ma, one of the developers of *Into the Breach*, said in an interview, “We cut everything that didn’t inform the combat” [[[1]](#footnote-1)] thus resulting in the game revolving all around telegraphed attacks.



For this project I think this methodology of development would work very well as it will allow for me to come up with various ideas and create quick prototypes to try out to help my follow the fun. As Marc Leblanc, a designed who worked on the game *Thief: The Dark Project* and the person who originally coined the term “follow the fun”, says “Fail faster, and follow the fun” [[[2]](#footnote-2)] which briefly describes the process of quickly making a prototype of an idea as quickly as possible to see what works and what doesn’t. With each failure you can iterate and develop the original idea further into something a lot more fun / interesting saying a lot about what direction the next attempt should take. The flow diagram on the right shows the iterative process of this workflow.

# Existing Products

## Unrailed!

*Unrailed!* is a 2-4 player roguelike game in which the players must gather resources to produce rails and dig out a path for a train to reach its next station. This game is very simple by design but through its very smart systems it encourages player cooperation and creates very tense moments for a seemingly easy game.

These design choices include simple controls involving just the analogue stick for movement and a single button for interaction. This makes the game incredibly accessible for any player which is important when introducing a game to others. It also features individual tools that define player roles rather than classes / characters such that players are not restricted to one roll mid-game allowing for more dynamic gameplay as players can switch tools with each other effectively changing their identity and purpose. For example, if a player was holding the pickaxe, they were the miner but if they were holding the axe, they were the tree cutter. By allowing player roles to change throughout the game fixes the problem in other role-based games where a role may become temporarily useless due to the context that they are in; in the case of *Unrailed!* this could be the absence of trees for the tree cutter. In this way *Unrailed!* also provides each player their own unique decisions that are integral to “winning”. Each of these roles are also all equally important to beating the game, this makes every player feel like they are contributing to the greater goal giving a real sense of achievement upon completing the game which is important when it comes to cooperative gameplay. Through this, the chances of beating the *Unrailed!* massively improve the better players communicate which is integral to a cooperative game. Another important feature of *Unrailed!*’s design is how there are multiple ways to lose ranging such as running out of resources, getting blocked by the train, leaving a tool behind and not digging out a path fast enough before the train reaches the end. Through this the game is able to create high tension which keeps players engaged and on their toes constantly. *Unrailed!*,very importantly, allows player mistakes that “hurt” or effect their coop partners directly which encourage communication such as through the simple act of letting players collide with each other and get in the way. These sorts of mistakes create moments for players to have a laugh or rage which improves the fun as player interaction always adds an element of randomness and allows for unexpected moments.

src: https://store.steampowered.com/app/1016920/Unrailed/

From this I compiled a list of points to think about when developing my game if I were to revolve gameplay around cooperation:

* Each player needs to have their own roles that provide them with unique decisions that are integral to winning.
* Different roles that are all equally important.
* Dynamically changing roles to prevent the temporary exclusion of one role in various contexts of the game.
* Promote player interaction throw allowing mistakes that actively “hurt” others.
* Multiple ways of losing to increase tension.
* Direct player interaction for added randomness.
* Accessibility through the game being easy to learn and pickup.

## Towerfall Ascension | PS4 Games | PlayStationTowerfall Ascension

*Towerfall Ascension* is a 2-4 player competitive duelling game that has players shooting arrows at each other to be the last player standing. Similarly to *Unrailed!*, *Towerfall Ascension* offers very simple controls with the D-pad / analogue stick for movement two other buttons; one for shooting and another for performing a dash.

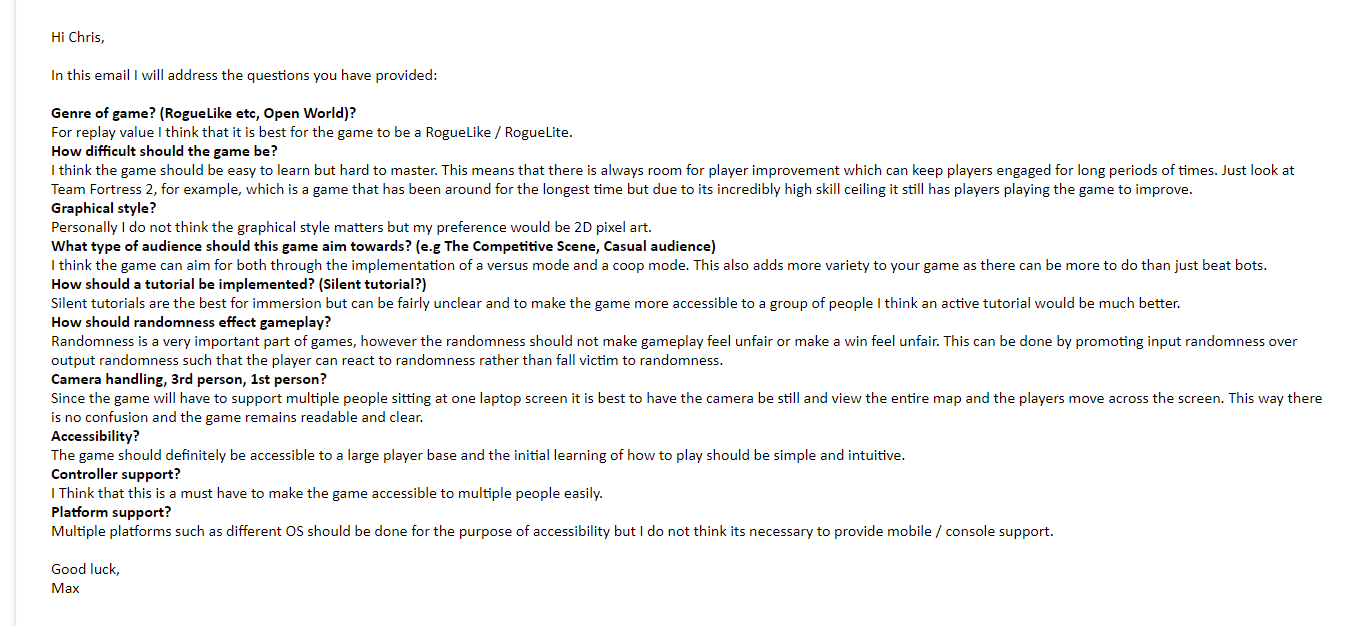
*Towerfall Ascension* offers very fair gameplay through making every player equal as every player character has the same abilities. This design choice makes each loss in game feel fair and balanced which is important when it comes to anything player versus player (PvP). *Towerfall Ascension* also makes use of powerups and varied different types of arrows such as “drill arrows” which can shoot through walls adding a surprising amount of depth to the gameplay as players have to account for different arrow types keeping gameplay fresh and interesting. The design choice to add powerups and allow for players that fall behind in points to start with a shield allows for clutch comebacks which makes losing players never feel like they really are losing keeping them playing what seems to be a lost round. For better controller support the game also allows arrows to very slightly “home” in on nearby players stopping those that play on D-pad to still land shots and not be at a disadvantage to those that play with an analogue stick. Another interesting design choice is how customizable the game is as it allows complete control on how various powerups spawn, the number of rounds players play, what powerups are banned and more. This means that the base game can be tailored to match what the players want which makes the game accessible to a wider player base.

src: https://store.steampowered.com/app/251470/TowerFall\_Ascension/

From this I determined a couple main points I would like to consider when developing my game:

* All players can be created equal to negate the problem of balance.
* There should be no advantage from playing on a different controller.
* Use of a comeback system to keep players that are losing playing the game and not giving up.
* Use of some random powerups to allow for funny unexpected moments.
* Customizable settings for increased accessibility.

# Client Interview

As noted previously, my intended clients are my group of friends that I often play with and who will be the primary user of my product. One of member from the group, Max, had agreed to answer some questions over email:

Email response from Max, a primary user for my game, addressing some of my questions for the game.

Max also commented on the fact that when developing a game, I need to formulate an initial idea that is unique such that it stands out from all the other games that already exist. He expressed that this innovation is important to ensure that players remain interested in my game because otherwise gameplay can feel boring and stale as without some form of unique gimmick, I would be making something generic.

**Analysis – General Objectives**

Before I plan out a game idea and start prototyping, I want to define some base minimal requirements that I must have. Since my game will support both remote play and local play, I will split this into this base specification into server and client:

## The server should:

1. Establish a connection between multiple clients
   1. Allow the means for 2-way communication between the server and client
   2. A standardised packet format will be designed in the design section of the project
   3. The server should adjust for packet loss and handle high ping / delayed packets appropriately for both incoming, and outgoing data
   4. Each client should be assigned a unique ID
2. Authorise the clients incoming data about position and client state
   1. If the server disagrees with the client, the server takes priority for server-authoritative control
3. Constantly broadcast a snapshot of its current world state to its clients
   1. Snapshots should be small, and only contain relevant data towards its respective client to reduce bandwidth usage
      * Such as only the area of the world that a client can see
   2. Snapshots are sent once every server tick
4. Calculate physics of entities
   1. Players are user-controlled entities that act on standard physics that can be controlled by commands sent via the clients
   2. Each entity and entity interaction should be assigned a unique ID such that they can be referenced in snapshots sent by the server
5. Receive packets relating to player controls from clients
   1. The server should account for ping and delay from the clients when performing actions (if the player is 200ms behind, handle the packet with the world rolled back 200ms)
6. Backup current world state over a given set interval
   1. A standardised file format will be designed in the design section of the project
7. Read from a configuration file which allows the user to set the server port, world generation seed, world difficulty, generated world tiles and entity positions, disconnect AFK (away from keyboard) players, AFK timeout
8. Be able to be started automatically via the client
9. Be usable standalone with a console interface
   1. This need not be usable by a naïve user as the server is aimed to be hosted by somewhat experienced users, however the interface should still be usable by novice users with the help of documentation
   2. console should provide the following features: list of clients connected, ping of each client, errors or warnings that occur on the server, server tick rate (and warning when the tick rate falls below standard tick rate), time stamps when clients connect / disconnect, ability to type commands into server command line
10. Remember the clients that have connected and their respective in-game progress such that on disconnect and reconnect, their progress is not lost

## The client should:

1. Provide a GUI for the user
   1. This should be usable by a naïve user
2. Host a server on the client pc with given settings / options that the user provides
3. Handle more than one player on a device and tell the server accordingly to allow for local play to work over multiplayer as well
4. Connect to a server with the IP address and port a user provides
5. Receive packets from the server
   1. Unwrap the packets and generate the snapshot client side for the user
   2. The client should account for the server tick rate and interpolate between snapshots sent to ensure smooth physics client side despite the slower tick rate of the server
   3. The client should account for lost snapshots / high ping and correct for disagreements in position with the server in a smooth fashion to increase quality for the user
6. Display / Render the player and world onto the screen
7. Provide an interface for the user to interact with the world
   1. User can control their player character
      * This can be done via keyboard / controller:
        + The client should provide an interface to change these controls
        + The client should detect when a new input device is connected
          - The client should save various control schemes for already recognised controllers on the local machine
8. Provide a method of connecting to a given server
   1. GUI
      * User can input the server’s IP and Port and the game will attempt to connect to that server
   2. The client should handle time out and connection errors appropriately
      * On disconnect, the user should be made aware of the error and should be sent back to the connect GUI menu where they can attempt to reconnect
9. Send groups of player actions in “action snapshot” packets to the server
   1. These should be sent periodically (not every frame) and as a result should include a queue of all actions / button presses the player did between each packet send
10. Save player settings and options locally
    1. Different settings can be saved as profiles that can be loaded at any point
11. Provide a sufficient debug of basic statistics: Frames rate, ping, connected IP and port
12. Automatically pause the game on minimize / alt-tab action
13. Provide a text interface for players to send messages to each other
    1. Global chat (for all players)
    2. Specific message chat to talk to 1 specific player
    3. Group chat for talking to a group of players

**Analysis – Initial Game Design and Prototyping**

My plan is to create a roguelike tower defence game with a catch. The catch is going to be the fact that the towers the player places can also hit the player themselves. This essentially mixes the traditional tower defence genre with bullet hells and thus creating intense rounds as the player must dodge their own towers whilst placing down more towers to defend the oncoming waves of enemies.

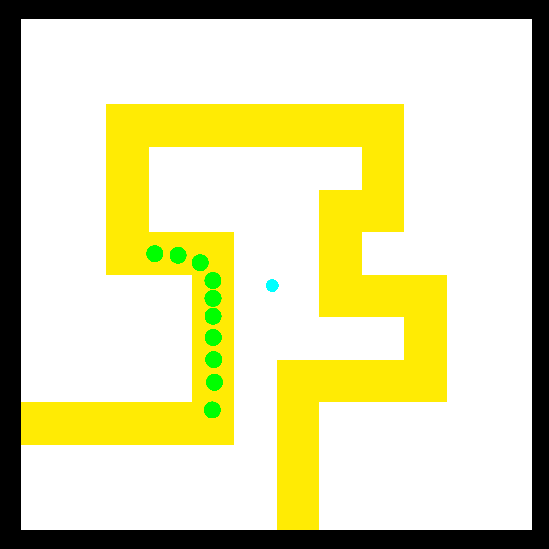
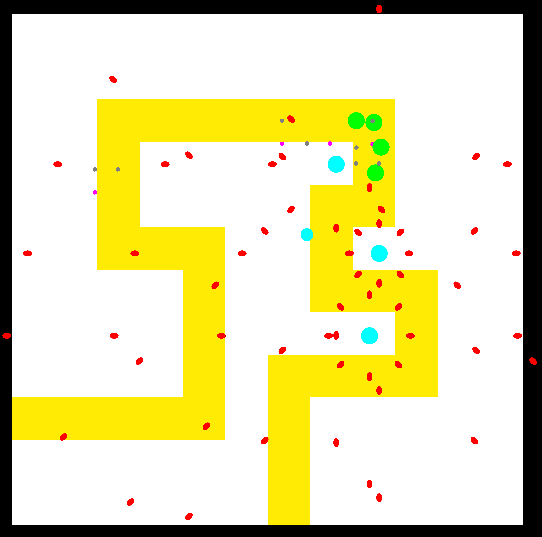
In a traditional tower defence game the towers actively target the incoming enemies, however since the game needs to factor the player into account I will make towers shoot in 8 directions around them such that the player will always be in the cross fire of the towers. Because of this, towers will also have to have infinite range such that the player cannot just rest outside of a towers range and they will have to shoot periodically to keep projectiles on the screen:



Tower (blue) shooting bullets (red) in 8 directions

Due to towers now shooting in a predictable manner it is straight forward for the player to place towers in such a way that they can sit still in a spot where the bullets will never reach them. One way to fix this is to make enemies drop the currency required to place towers as it will force the player to move to where the enemy dropped the money and pick it up so they can improve their defences.

For the time being I think this is a good starting point and thus I decided to produce a prototype version using these core concepts. I’ll address how multiplayer will affect this design later in the design process:

In this prototype the path the enemies will follow is shown in yellow and the enemies themselves are in green. The player character is denoted by the small blue dot and the larger blue dots represent the players towers. The red dots represent the projectiles that the player must dodge and that the enemies will die from and the small grey dots represent the money dropped from dead enemies. Through early playtesting I found that the game was very challenging. In order to stay alive, I found myself moving the player opposite to the enemies such that the enemies would block the projectiles for me. This also put me in a good position to collect the money that was dropped by enemies when they died. I think that this strategy of “wave hugging” is good as it promotes player movement and is a niche strategy that works when enemies are packed closely together in a line.

# Client Opinions

After producing a quick prototype, I sent it over to my primary user Max and spoke with him at length about the direction of this idea and how to “Follow the fun”. Below is a transcript of part of what we spoke about:

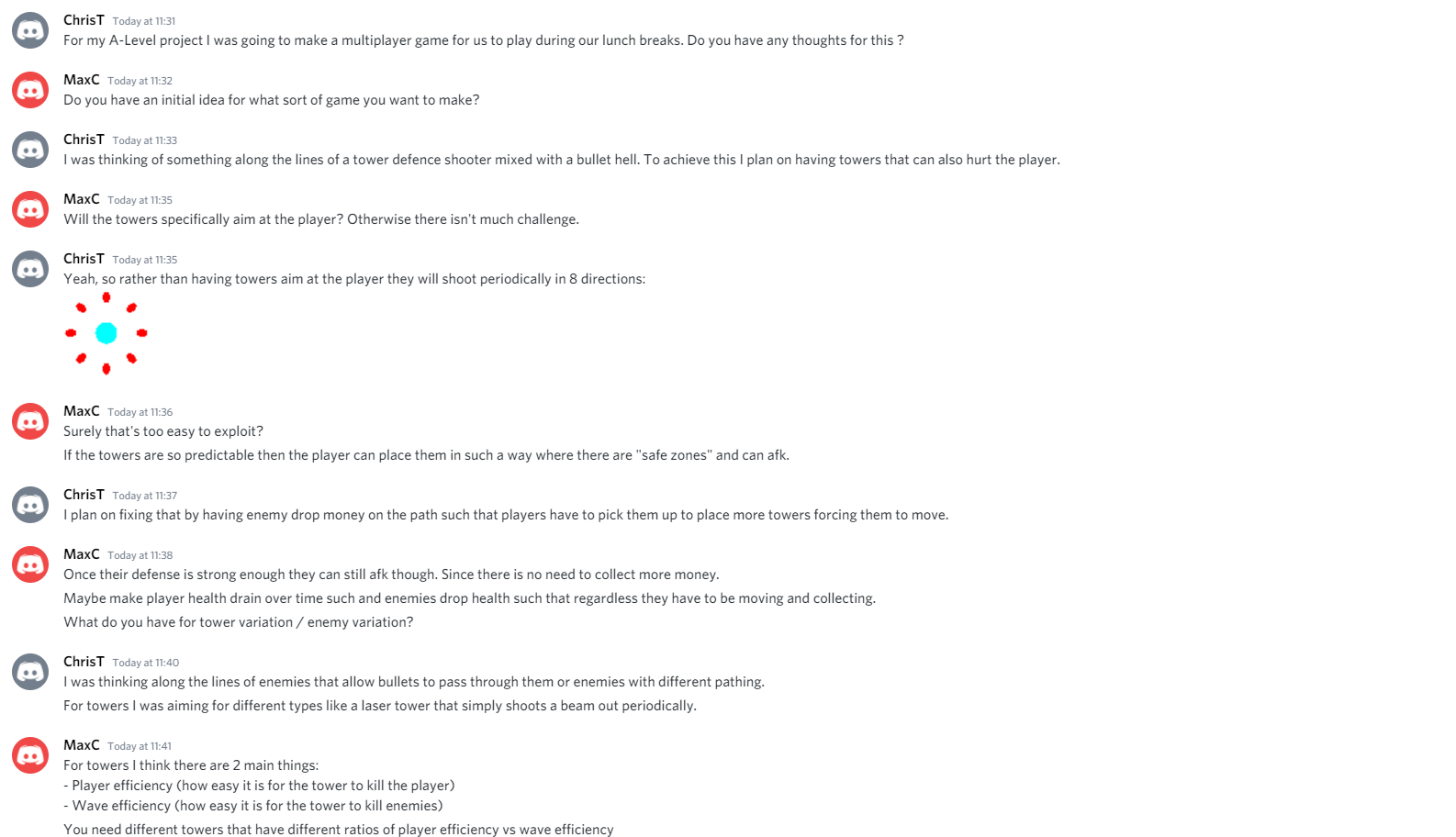
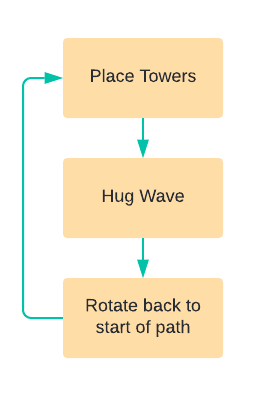


Image of conversation I had with Max

After play testing, Max and I were very happy with the game, it felt fun and fair. Whilst playing, Max commented how he found the wave manipulation such as “wave hugging” to be the most interesting and fun. Because of this we thought that a step in the right direction would be to allow the player to collide and interact with enemies such that they can form better wave states which could be very interesting and fun.

We also discussed the general gameplay loop of the game. Currently with a single infinite level to see how far the player could get, the gameplay loop was very simple:

Gameplay loop or strategy of the prototype proposed.

Max suggested that an easy way to add variety to this gameplay loop and to build upon wave manipulation would simply be to vary enemy behaviour. For example, having enemies that where more spread out reduces the amount of wave hugging the player can perform. Different enemy behaviours can also be more complex involving enemies that seem to have a personality through pausing at areas with no bullets passing by only to dash past the area filled with bullets as if they were timing their push to not get hit. This would add depth to the gameplay and give enemies character which would be fun to strategize against especially with proper player-enemy interactions.

Another problem with this prototype is that with a single infinite level there is no real end goal for the player and due to the games nature of enemies getting stronger as time goes on and how more towers to deal with said enemies will hindrance the player, the game feels unbeatable. To solve this, I am going to make the player survive for a pre-determined number of waves per level and then at the end of each level the player will be placed into some form of level selection to proceed to the next level. The game would then also have a final boss level which will be hard for the players to complete at the end, but since the game will now have an ending it no longer feels unbeatable.

Through the addition of perma-death such that when the player dies, they lose all progress and must start again at the beginning and all the levels are procedurally generated, I can add a lot of replay ability.

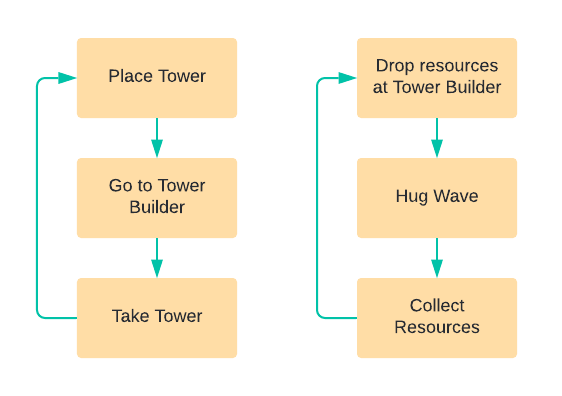
# Scaling into multiplayer / cooperative gameplay

With the core gameplay out of the way, I now need to scale the idea up to work in a multiplayer setting. For the multiplayer, I want to promote cooperative gameplay and communication and to do this I need to identify the different roles in my game:

* Tower placement.
* Resource collection (money collection).

In order to create a clear distinction between these two roles I will make it such that the player can only carry one thing at a time, either a tower to place or resources to use in making towers such no one player can do both jobs simultaneously. Currently, however, tower placement as a role becomes much less important as time goes on as there will be enough towers on the field. To fix this I am going to give towers a form of lifetime such that over a time the tower disappears and needs to be replaced to keep tower placement relevant.

With these changes there are now two distinct roles with their own unique gameplay loops:



The left flow diagram shows the gameplay loop for the tower placement role and the right shows the gameplay loop for resource collection.

In the above diagram I mention a “Tower Builder” this is essentially a random post on the map in which players drop resources the picked up at and it drops towers for another player to pick up and place. This forces players to move around the map and not stick to distinct parts as they need to fall back towards the “Tower Builder”. It also adds slightly more complexity as the player placing towers can keep into account its location and place towers accordingly such that the bullet hell is less severe around the “Tower Builder”.

# Gameplay Objectives

With the core gameplay design figured out I can now create a specification for the game:

## Stage 1: Minimum Viable Product (Create core gameplay)

1. The game should have a lobby showing the number of players that are playing.
2. Each level should be procedurally generated.
3. The game should have a level select GUI.
4. Upon entering a level, the game should:
   1. Display the layout of the path the enemies will take.
   2. Display the location of the “Tower Builder”.
   3. Allow players to move around the level.
   4. Have enemies spawn in waves that follow the path.
   5. Abide by standard tower defence rules:
      1. Enemies spawn at one end of the path and upon reaching the end the player loses (either entirely or some form of health system).
      2. Players can kill enemies using towers.
      3. Enemies drop a collectable resource used in creating towers.
   6. Tower shots can hurt the players as well.
   7. Towers self-destruct at the end of their lifetime.

## Stage 2: Enemy variation (AI)

1. When generating each level, the game will also define what enemy types will appear in each level and for which waves.
2. Enemy variation through different game mechanics.
   1. Enemies that can only be shot from one side.
   2. Enemies that bullets can pass through.
   3. Enemies that reflect bullets upon being hit.
3. Enemy variation through different behaviour.
   1. Enemies may stagger and stall in “safe areas” along the path where bullets do not cross and hastily cross “dangerous areas” filled with bullets.
   2. Enemies may move quickly in a straight line but slowly along turns.
   3. Enemies may wait for other enemies in “safe areas” and stick together before proceeding.

## Stage 3: Tower Variation

1. Each level provides the player with three “Tower Builders”.
   1. The player can tell each “Tower Builder” what type of tower they want it to make at the start of each level. What they select is what the “Tower Builder” will make for the rest of the level.
2. Tower variation through firing:
   1. A tower that shoots in a cone towards enemies.
   2. A tower that shoots faster projectiles
3. Tower variation through different mechanics:
   1. A tower that shoots lasers.
   2. A tower that shoots homing projectiles.
4. Tower variation through upgrades:
   1. Longer lifetime
   2. Faster projectile speed

## Stage 4: Quality of life:

1. Settings to customize the gameplay:
   1. Ability to disable of certain enemy types.
   2. Game modifiers:
      1. Slow motion / Bullet time.
      2. All homing bullets.
      3. Enemies getting through the path instantly cause a loss.
   3. Ability to disable permanent death.
   4. Ability to disable different tower types.
   5. Slow down default game speed.
2. Saving the game mid-playthrough or after level completion.

## Stage 5: Polish:

1. Entity ragdolls.
2. Particles and effects.
3. Character animations.
   1. Inverse Kinematics

**Design – Programming Solution**

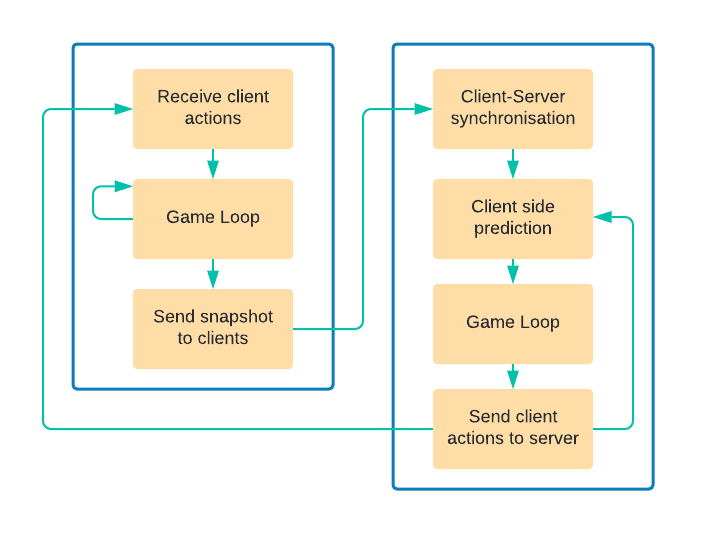
I intend to program this game in Unity 2D using C# as it provides an easy method for rendering a 2D scene as well as providing a physics engine. It is also a well-known and reliable engine for game development.

**Design – Overall System Summary**

# Local and Remote Play

## Base design

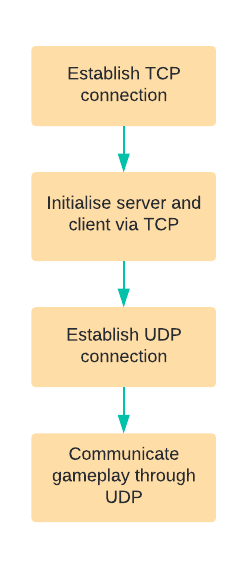
As described in the analysis, my game will support both local play and remote play. Because of this I will be developing my system with a server and client in mind and since my game is played in real time I will need to factor in synchronisation issues that come with players being on different devices. I will discuss the specifics for this synchronisation later in the design process. For my multiplayer system I will also run the server at a lower framerate as it will not perform any rendering or be required for smooth animations and game mechanics alongside its physics calculations do not require that many frames. A lower framerate will also reduce the chance that the server fails to fulfil its target framerate and skips frames which can cause all kinds of gameplay and synchronisation problems. This lower framerate should also improve the servers performance and as in my context it is very likely that the host will also be running the client on the same device, thus it will be important that the server can perform optimally whilst the client is also running. Below is a flow diagram for my basic system.



Flow diagram showing how the server (left) and client (right) will interact with each other.

To handle local play, the server considers each client as a group of players of size “n” and when sending commands between each other they refer to which player via an index.

## Networking

To establish a connection between the server and client I will be using the TCP protocol, however once the connection is established, I will be using the UDP protocol for communicating gameplay data. This is because TCP is a connection-oriented protocol meaning that TCP requires an established connection between a sender and receiver before data is sent whereas UDP is a connection-less protocol meaning that a connection does not need to be established to send data. This makes UDP simpler, faster and more efficient than TCP which is important for a real time system. TCP is still required as although UDP is faster, UDP does not care for packet loss. For a game this is not a problem as if a packet is lost, the data in that packet is soon no longer valid as it becomes “out of date” as the game time progresses, but for initially establishing a connection and sending important initialising data between the server and client, UDP is not very reliable and thus TCP must also be used. The below diagram shows how a connection would be established in this system:

Flow diagram showing the use of TCP and UDP in establishing a connection.

However, there is a key problem with using both TCP and UDP. This is mainly due to how TCP and UDP are protocols built on top of the Internet Protocol (IP) and the way they interact and affect each other is super complicated and relates to how TCP performs reliability and flow control which can cause TCP to induce packet loss in UDP packets [[[3]](#footnote-3)].

Because of this I will only use UDP but implement my own protocol on top of UDP but implement the specific features of TCP that I would need as well as other features I may need:

* Virtual Connections
* Reliability System
* Splitting data into packets
* Congestion Avoidance

## Networking UDP – Splitting up data into packets

UDP does not have a way to split up data that I want to send into packets and thus I must implement it myself. In order to do this, I simply will have a predefined byte count for the maximum number of bytes that can be sent in a singular packet. The data can then be split into packets by chunking it into the groups of bytes that can fit into a packet, using additional packets whenever there are more bytes than what I defined as the maximum. Each packet will also then be provided with a header containing information on what data the packet belongs to and which portion of the data it contains such that on receiving a packet reconstruction of the data can be done.

The reconstruction of the data from each packet can just as easily be done by queuing all received packets until all the packets required for a given block of data has been received. Once received the information in the header of each packet can be used to correctly order the data.

## Networking UDP – Virtual Connections

I will define a virtual connection as two devices exchanging packets at some given rate and thus if both devices are receiving packets, I can consider them to be virtually connected. With this the inverse is also true, if a device is not receiving a flow of packets it can consider itself as disconnected.

With this system my software needs to distinguish between packets that are received using my UDP protocol over other protocols that I may use in the future. To do this I will provide each packet with a *protocol ID* which will be some predefined unique integer. This way, when a packet is received, the first 4 bytes are inspected and if they do not match the *protocol ID* they will be processed differently. If they do match, then the packet can be processed using my protocol.

## Networking UDP – Reliability

For reliability I will implement something similar to TCP using *sequence numbers*. This number acts like a “packet id” such that each packet sent will have a unique ID. This can be implemented by having a value as the *sequence number* and then incrementing it with every packet sent such that the first packet sent is “packet 0” and the second is “packet 1” etc… This *sequence number* is important since it allows the receiver to identify what each packet is as UDP does not guarantee the order of packets so the 100th packet received may not be the 100th packet sent. There is still a problem with this system which is that the *sequence number* can top out if it increments over its maximum value. This can simply be fixed by allowing it to overflow back to 0 once its maximum value is reached and this can be detected on the client side by checking if the difference in the received sequence number and the previous sequence number is very large, so large that it is very unlikely due to a packet being received late.

Next is to reply with acknowledgements such that the server knows what packets the client has received and vice versa. To do this the packets will also include an *acknowledgement* value which corresponds to which packet has been received via use of the *sequence number*. This introduces another problem, such as what happens if the server and client are on different flow rates such that the server sends a packet 30 times a second and the client sends only 10 times a second. Since only 1 acknowledgement is sent, the client will only be able to acknowledge 10 of the 30 packets. To solve this I can simply send more than 1 acknowledgement per packet. I’ll use 32 acknowledgements per packet for convenience as it can be stored using an integer value treated as a bit field such that each bit in the bit field represents another acknowledgement of the packets *acknowledgment* value minus the position of the bit in the bit field. For example, if a packet of *acknowledgment* 100 is received with the 1st and 3rd bit of the acknowledgement bitfield being set then the client has received packet 100, 99 and 97.

This system also means that each acknowledgement is sent an additional 32 times as each packet contains its acknowledgement and the previous 32 acknowledgements which may overlap. This is actually okay as with this redundancy, even if a few packets are lost, the server still has hopes of receiving the acknowledgement due to this redundancy and if the server does not receive an acknowledgement within a certain time frame it is incredibly likely that the packet was lost. For example if the server sends 30 packets per second and acknowledgements are sent 32 additional times, then after 1 second it is incredibly likely the packet was lost.

TODO:: insert diagram

## Networking UDP – Congestion Avoidance

TCP has a very robust congestion avoidance algorithm, but UDP does not have any form of congestion avoidance. If packets are just sent without any flow control, then there is a risk of flooding a connection and gaining severe latency. This happens as routers try very hard to deliver all packets they receive and may buffer up packets in a queue before they drop them.

Since my game will, for the most part, be run mainly in a LAN setting, I can use a very simple algorithm for congestion avoidance. This simply involves storing when each packet was sent and upon getting an acknowledgement get the time delay between sending that packet and receiving an acknowledgement. This is called the round-trip-time (RTT) and I will update it according to each incoming packet to get an average RTT. This will most likely be implemented using an algorithm called “exponentially smoothed moving average” which involves updating the RTT by moving its value by 10% of the difference between the current RTT and the RTT time for a given packet. All packets that are not acknowledged after some predefined amount of time will be dropped from this queue (as described before, since after 1 second it is likely the packet was lost, this time can also be 1 second). Using RTT, if it increases above some threshold, I drop the number of packets sent per second and upon RTT decreasing I can increase the packets sent per second back to normal. This system is very basic but should work out in a LAN setting.

## Network Structure

I have two options when it comes to designing the structure of my network:

* Peer to Peer network
* Server Client network

For a peer to peer network I would still need one device to be allocated as the host as I want the game to follow server-authoritative design which means that the server is assumed to be right and will make decisions as to how the clients should act. Because of this the only real benefit of a peer to peer network is the fact that the bandwidth would be spread over multiple devices rather than all connections being made to one device as it is in a server-client based network. In my context of a small group of friends playing this game, it is unlikely that the benefit from this will make any difference at all. A peer to peer network will also be less efficient as data from the server has to be “trickled” down the network to reach players that are not directly connected to the server. For these reasons I will be using a server-client design.

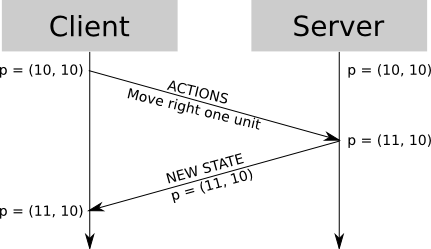
## Network synchronisation and handling bad connections

My game’s network will use server authoritative design. This means that everything happens in the server whilst the clients act as “privileged spectators” of the game. In this way the clients will send inputs such as key presses and commands to the server rather than information such as the clients player position.

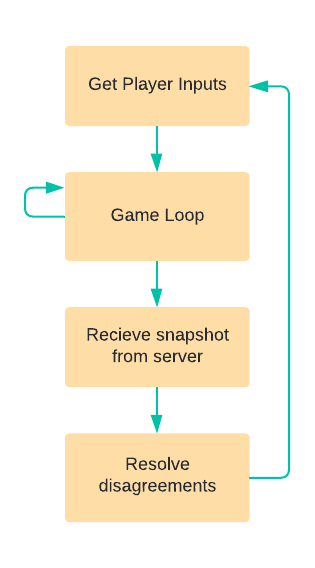
This design has the advantage of using an authoritative server that can be trusted over trusting the clients to be correct / truthful. However, this design comes with a few problems:

### Problem 1: Input Lag

The problem with this system comes from the fact that this would mean all clients are viewing the game in the past and all player inputs will be delayed as they must be passed to the server and then wait for a server response:



Simple client-server interaction showing how the client inputs are handled over time where p is the position of the player

For fast connections (low ping), this delay will be mostly unnoticeable, but for slower connections (high ping) it can ruin the player’s experience. One way to fix this is to use client-side prediction:

In this way, the client will have the same update loops as the server such that it can run the same physics and game loop to update the player and other objects. This would allow for the player to update immediately locally as according to new input commands and other objects can update as according to physics. When it comes to predicting the movement of other players, the game loop will just use their previous input commands from the last server snapshot / response and assume movement in the same direction using “Entity Interpolation”.

Once the client finally receives an update from the server it can resolve the disagreements with its prediction and with what the server had sent. This resolving cannot be as simple as just moving everything to match the server snapshot since due to the lag, the server snapshot would be in the past whilst the client-side prediction is resolving for the present.

Flow diagram showing client- side prediction

Below shows a diagram of a client using client-side prediction and what happens when it receives the past snapshots from the server after inputting a move right command twice:

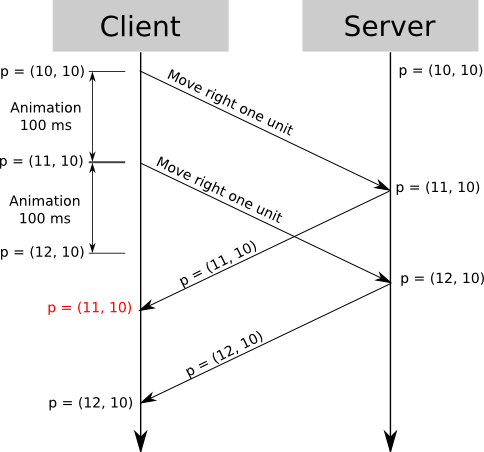


Diagram showing client-side prediction where p is the position of the player

Here it can be seen that the client correctly predicts the player movement in the first 200ms but due to the lag the server confirms this movement after the prediction. If resolving conflicts simple moved the player to the position noted by the server, the player would teleport back after the prediction which should not happen.

To fix this I will implement “Server reconciliation”. This involves having the client save each given input from the player with a “request” number such that, in our case, the first move right input is request #1 and the second is request #2. The server will then send snapshots with the players input request such that the client can see that the server got a given position after resolving input request #1. Assuming the client keeps a copy of the requests it sends to the server, it knows that the server has just resolved request #1 so it can apply client side prediction of request #2 from the server’s position provided from its response to request #1 and update the player’s present position for request #2. Request #1 on the client can then be discarded as it has been confirmed by the server. This can then be repeated once the client receives request #2 from the server.

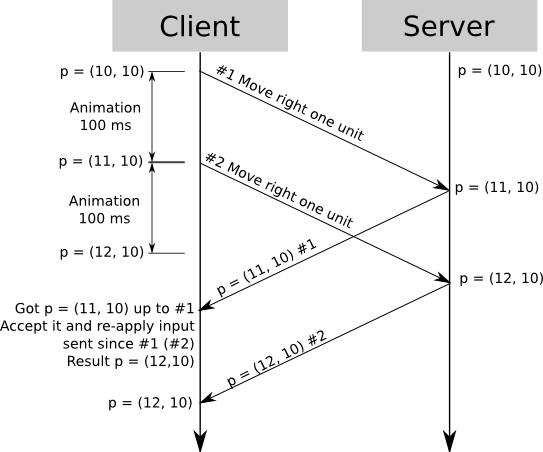


Diagram showing client-side prediction with server reconciliation where p is the position of the player

Although this example uses movement, this system can be applied to almost anything else required synchronizing.

### Problem 2: Low Frequency Updates (Low frame rate server)

The system described above works for a single client on the server, but with multiple clients, once the server receives a request, it needs to relay this to all other clients. If the server uses low-frequency updates this can create choppy movement on other clients as one will perceive the other “jumping” between positions:

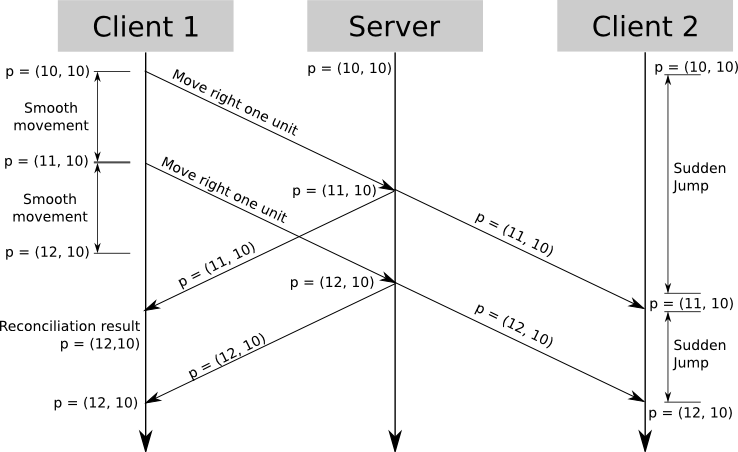


Diagram showing 2 clients receiving updates from the server where p is the position of the player

To handle this, I plan on implementing “Entity Interpolation” [3]. This involves showing the client player as in the present but other players as in the past. For example, if the server sent updates out every 100ms and the current time was t=1000ms, then from t=1000ms to t=1100ms the client will show the movement of the player that the server had sent from t=900ms to t=1000ms. In this way the client is always showing actual movement data except its 100ms late:

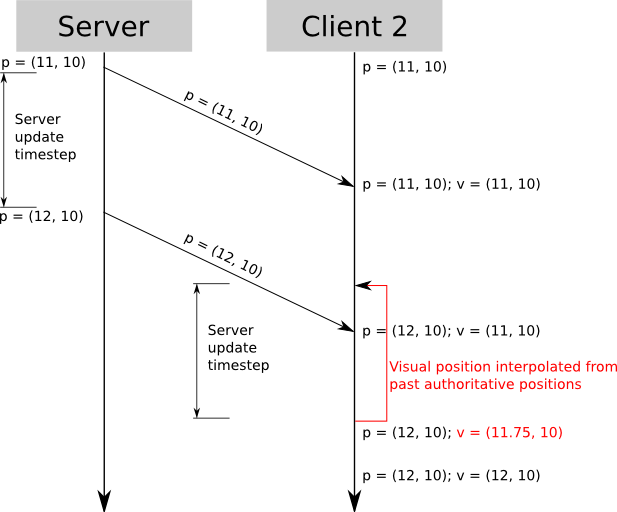


Diagram showing client 2 using the past server’s data to show entity movement in the past where p is the position sent from the server and v is the position shown by client 2

Using this form of entity interpolation comes with a problem since the players will see themselves in the present but others in the past. This becomes problematic for very time and space sensitive events, such as shooting another player / interacting with them.

This can be solved using “server rollback” [[[4]](#footnote-4)] which involves having the client send all inputs / events with timesteps such that the server can “rollback” the world to the past where the client supposedly performed its action and process it at that point in time before updating the other clients. This rollback will need to be capped as if the a client 1 has a ping of 2000ms and they shoot client 2 who has a ping of 50ms, with non-capped rollback, client 2 can be shot despite standing behind a wall (from client 2’s perspective) as client 1 is 2000ms in the past which was when client 2 was not behind a wall. Typically, rollback would be capped at ~250ms.

### Problem 3: Bandwidth usage

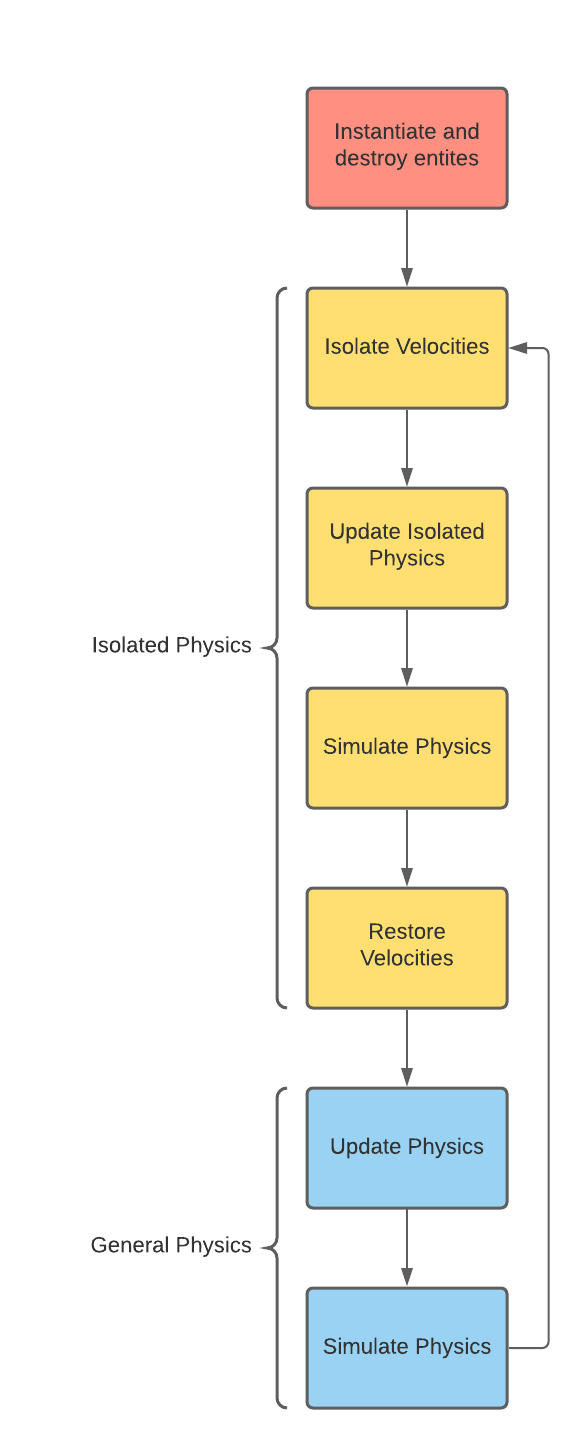
With this current system the server sends the client a world state which represents how the world should look. The problem is that these world states can be very large and consume a lot of bandwidth which is not good. To solve this, I am planning to use “Delta Compression”. This involves sending world states relative to a baseline. For example, if a world state involves a cube, the server can send to the client “The cube has not moved” which can be represented in 1 byte. Of course, the problem with this is that it requires the client to have a baseline and the server must also know this baseline. Not only this, but the baseline may be different per client. Baselines being different between clients is not a big deal as the server can store multiple baseline snapshots for each client but synchronizing the baseline between the server and client is a challenge as the client needs to send some form of acknowledgement for a snapshot to be used as a baseline. An easy fix for this would be to have the server send a full world state at a slow rate and use delta world states to fill in between the full world states. The client would then send a response saying which full world state they are using for a baseline, and update accordingly for new full world states they receive.

# Engine / System design

I will be using Unity 2D engine to handle most of the rendering and physics in my game and I will be designing my underlying engine to run in conjunction with Unity.

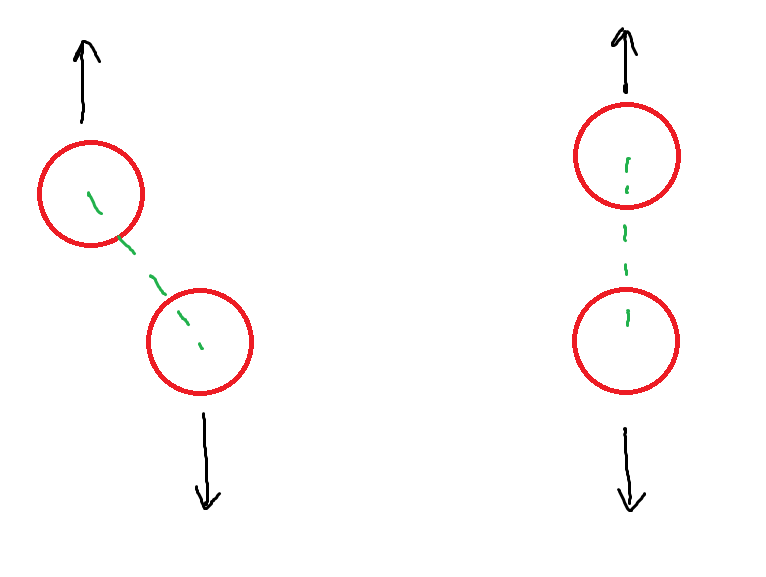
## Physics

My engine uses a Unity 2D mostly for its physics and was made to implement custom physics components as well as handle general entity behaviour. Below is a flow chart that shows how I plan my engine to interact with Unity:

This system was designed with the intention of allowing for me to implement custom physics to work with Unity. To do this, I have setup Unity to allow me to control when it performs its physics simulation as denoted by the blocks in the flow chart labelled “Simulate Physics”. I have also decided to split the physics into two parts:

* Isolated Physics
* General Physics

This was done to implement specific body physics. For example, if a character body consisted of two nodes that are joined through a distance joint and I wanted the body to right itself such that the two nodes would sit upright, I would apply an upward force to one node and an equal downward force to the lower node:



The above image shows the before and after state of a body:  
- The green dotted line represents the distance joint whilst the black arrows represent the force being applied to each node (red).

Flow chart showing how my engine interacts with Unity

The problem arises when the “righting” force is equal to the force of gravity as this would result in the top node having no net force and thus it would not move to right itself above the lower node. This is solved through the method of “Isolated Physics” as it performs character physics such as “standing upright” in an isolated setting without external forces such as gravity and simulates a physics timestep. The purpose of this functionality for my game is because I plan on implementing character bodies and ragdolls which describes a form of animation that relies on physics such as a death animation having a character fall over due to gravity and interact with other physics objects due to physic collisions.

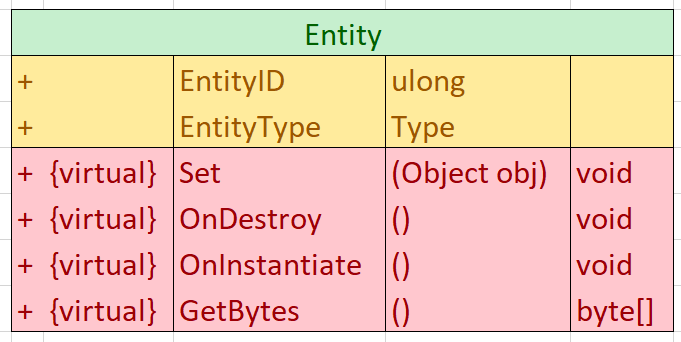
“General Physics” is used for standard physics updates such as player and enemy movement. It is also going to be used for updates not necessarily required for physics such as updating player states and enemy pathing.

## Memory Management

Since I will be using some unmanaged resources such as GPU buffers and sockets, I need to dispose of them when my program closes. To do this I will most implement a memory manager that contains a static list of all unmanaged resources and at the end of execution this list can be looped over, and all unmanaged resources can be disposed of.

# Entities

I plan for all objects and components to inherit from an abstract entity class. Below is the UML class definition:



UML diagram for abstract class ‘Entity’

“EntityID” and “EntityType” are used to identify specific entities across the server and client. “EntityID” represents the unique ID of each entity whilst “EntityType” is used to determine the type of entity that the object is supposed to be.

“Set”, “OnDestroy” and “OnInstantiate” are virtual functions specific to initialising and destroying a given entity and “GetBytes” is used to return the byte format for a given entity for storing in packets and sending it over the network.

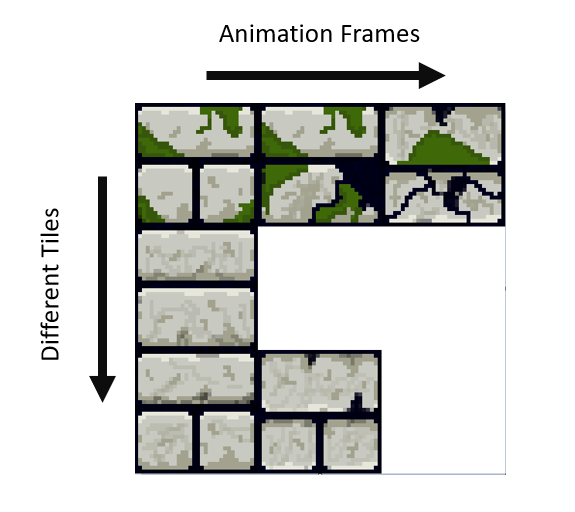
I also plan to have each entity use a different update format depending on what they are. For example, if an entity requires character body physics then they will use the isolated physics update described above. This way, entities that do not require physics updates do not get iterated over.

# Tile maps

The levels and world layout that I will be using involve tile maps. For this I will not be using Unity’s inbuilt tile map class as it is optimized for level design rather than procedurally generated maps. My tile map implementation will use the GPU to render the tiles onto a single texture that can be rendered in Unity on a singular sprite for optimal performance.

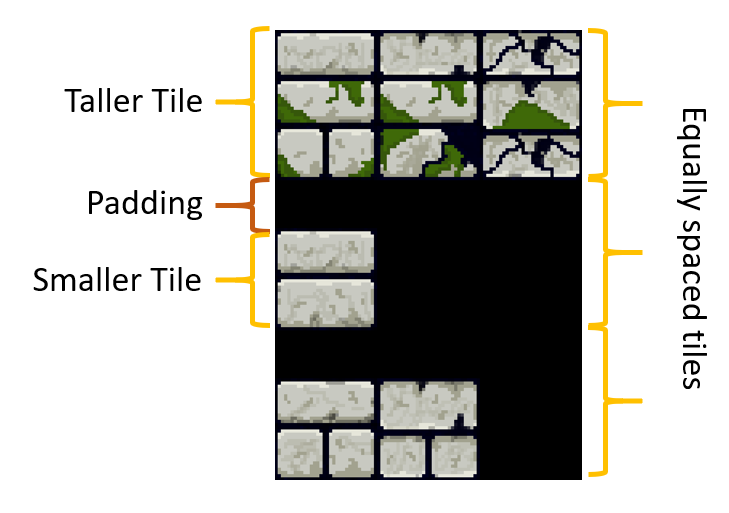
## Tile maps – Tile Pallets

Each tile map is provided with a texture which contains the tile pallet for what tiles a given tile map will use. This texture will be formatted such that textures going down represent different tiles, and textures going right are for different frames of each tile to support tile animations:



Example tile pallet for a tile sheet (courtesy of <https://alcwilliam.itch.io/azulejo-32x32> for the tile sheet)

Since I want my game to be 2.5D to give a sense of height and depth, taller wall tiles will need to be rectangular in shape meaning that the tile sheet will need padding to fit these taller tiles. The padding size will be a predefined value tailored for each tile pallet:



Example tile pallet showing rectangular tiles (courtesy of <https://alcwilliam.itch.io/azulejo-32x32> for the tile sheet)

Each tile map will also have to be split into chunks of a pre-defined size such that I do not have to pass the entire world map to another client over the network but only specific chunks around each client. This is not terribly relevant however since my game will primarily be played on small maps.

## Tile maps – Data Format

Each tile map will firstly need to store which tile pallet it is using. This can simply be done by having each tile pallet be associated with a unique ID and reference it in each tile map. The tile map also needs to store the tiles that makes up the tile map with each tile storing which tile it is rendering from the tile pallet, which frame of animation it is rendering and whether the tile is blank or not. Below is pseudo-code for how I might define a tile type:

**public** **struct** Tile

{

**public** **int** AnimationFrame; //Which animation frame from tile pallet

**public** **int** TileIndex; //Which tile from tile pallet

**public** **int** Blank; //Is this tile blank?

**public** **int** Render; //Is this tile being rendered?

}

These tiles can then be stored in a list of tiles to represent what tiles make up the tile map. I will need 2 of these such that one can represent the wall tiles and the other can represent the floor tiles. The list of tiles could be stored using a 2D array however the tile data needs to be sent to the GPU so the GPU knows how to render the tile map and the format of a 2D array cannot be easily sent to the GPU and thus I will be using a 1D array but treat it as a 2D array as I can simply treat each row of the tile map being placed next to each other:



2D representation of tiles



1D representation of tiles showing how it can be treated as a 2D representation by joining each row

This means that I will need to convert from a 2D coordinate of a tile to a index on a 1D array, this can easily be done since I know the width and height of my tile map, so given the *x* and *y* position of a tile and the *width* of the tile map, the singular index is equal to [*y \* width + x]* where x and y are both 0 based.

As the player will need to interact with each tile map through colliding with the wall tiles, the tile map will also need to store a collision map which contains all the colliders for a given tile map. This can be stored in the same format as described above. As for the colliders I will be using Unity’s in-built physics colliders.

## Tile maps – Rendering

When it comes to rendering each tile map there is a lot to consider. Firstly, the player needs to render behind the wall tiles but in front of the floor tiles. The wall tiles also need to render behind the player depending on their position:

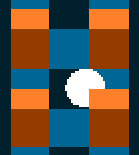


Image showing how a white ball should be rendered above the floor map (dark blue) but behind or in front of the wall tiles (orange)

Unity, however, does not allow for different parts of a sprite to render behind or in front of another sprite, and thus the tile map will need to be split into different sprites. Because of this I can render each tile onto individual sprites and render those in Unity. These sprites can sample their textures from a larger texture generated by the GPU which contains each sprite for the wall and floor maps:



Image of floor texture that is sampled for sprites

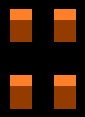


Image of wall texture that is sampled for sprites

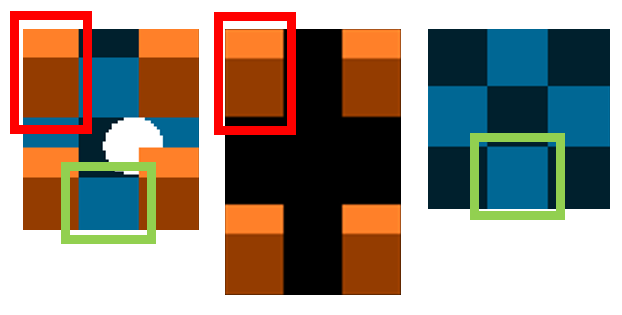


Diagram showing how sprites would be sampled from the main textures

The GPU would generate these texture maps for the sprites to sample from using the tile pallet and tile data provided. I would most likely implement it using 2 thread groups for that represent the rows and columns of a tile map. Each thread would then draw a tile onto the texture such that each the GPU is rendering all tiles of the tile map simultaneously.

For getting the wall tiles to render above a given object I will be using the *sorting order* property that Unity provides each sprite. This sorting order is an integer value that determines what gets drawn to the screen first, starting from lowest to highest. This results in the objects of highest sorting order to render above everything and vice versa. I simply need an algorithm to determine what this sorting order should be. Firstly, I will define how when an object becomes “behind” another object. Since my game is 2.5D where it is 2D but the walls seem 3D, an object becomes “behind” a wall when it has a *y* position that is smaller than the *y* position of a wall tile:

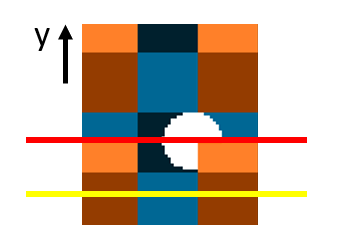


Diagram showing how the orange walls beneath the white ball have a y position (shown by yellow line) lower than the ball’s y position (shown by red line) and is thus rendered above it as the ball is considered to be “behind”

I can simply implement this by having each y position correspond to a *sorting order* value. For example, if the sorting order of all sprites was equal to its negative *y* position truncated as an int, then the sprites of lower y value would have a larger *sorting order* value and vice versa. To accommodate for the *y* values getting truncated as integers the *sorting order* for wall sprites will be their *y* position truncated plus 1. For the floor tiles to render behind I can use Unity’s *sorting layers* which simply groups sprites such that sprites in each group follow their respective *sorting order* values but will always render behind or in front of another group regardless of *sorting order* depending on how the *sorting layers* are ordered.

## Tile maps – Byte data for network transmission

Each tile map will only transmit vital data over the network such as:

* Tile pallet index referring to which tile pallet it is using
* Tile map position
* Size of tile map
* Floor Tiles
* Wall Tiles

Storing most of these properties is easy, for example the tile pallet index can simply be a single byte as I doubt that I will need more than 255 tile pallets, and for position and size I can simply use integer and float values. However, for the floor and wall tiles there are a few space optimizations I can make. For example, if a tile is blank, I do not need to send any data relating to *tile index* or *animation frame* etc… and thus for blank tiles I only need to store the fact that it is blank. Similarly, if the tile map does not change, then I do not even need to send the tile data at all, but this last part would be handled in “Delta Compression” discussed previously.

## Tile maps – Optimization

In Unity creation and destruction of objects is a very expensive process and having lots of objects in at once can tank performance. Because of this my tile map will group each row of tiles into 1 sprite and 1 object since the *sorting order* is only dependent on the *y* position:

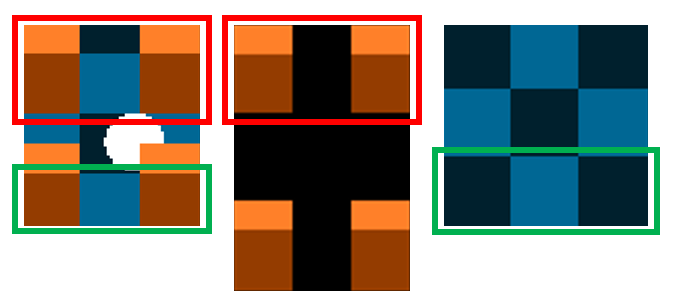


Diagram showing how sprites would be sampled where 1 sprite represents 1 row of a tile map

Whenever a tile map gets resized to be smaller, rather than destroying the unused rows I will simply disable the objects such that they can be reused. The same goes for the GPU and its buffers where I will simply reuse them and only decrease the buffer sizes when necessary.

Each tile map will also cache the byte data representing its floor and wall map such that when generating the byte data for a network transmission, it does not need to be regenerated every time.

1. <https://www.vg247.com/2019/04/18/into-the-breach-making-of/> <https://www.gdcvault.com/play/1025772/-Into-the-Breach-Design> [↑](#footnote-ref-1)
2. <http://twvideo01.ubm-us.net/o1/vault/GD_Mag_Archives/GDM_December_2012.pdf> [↑](#footnote-ref-2)
3. [https://www.isoc.org/inet97/proceedings/F3/F3\_1.HTM](https://web.archive.org/web/20160103125117/https://www.isoc.org/inet97/proceedings/F3/F3_1.HTM) [↑](#footnote-ref-3)
4. <https://developer.valvesoftware.com/wiki/Source_Multiplayer_Networking#Lag_compensation> [↑](#footnote-ref-4)