

---

***A-Level Computer Science (7517)***

***Server Authoritative Multiplayer Game  
"Enter the Dead Zone"***

**Development Log and Documentation**

**Christopher E-Hong Teo**

The Perse Upper

9 February 2021

---

---

## 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

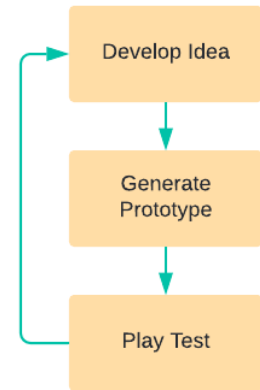
---

### 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” <sup>[1]</sup> 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 designer 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” <sup>[2]</sup> 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!



src: <https://store.steampowered.com/app/1016920/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 role 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

<sup>1</sup> <https://www.vg247.com/2019/04/18/into-the-breach-making-of/>

<https://www.gdcvault.com/play/1025772/-Into-the-Breach-Design>

<sup>2</sup> [http://twivideo01.ubm-us.net/o1/vault/GD\\_Mag\\_Archives/GDM\\_December\\_2012.pdf](http://twivideo01.ubm-us.net/o1/vault/GD_Mag_Archives/GDM_December_2012.pdf)

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.

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



src: [https://store.steampowered.com/app/251470/TowerFall\\_Ascension/](https://store.steampowered.com/app/251470/TowerFall_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.

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:

Hi Chris,

In this email I will address the questions you have provided:

**Genre of game? (RogueLike etc, Open World)?**

For replay value I think that it is best for the game to be a RogueLike / RogueLite.

**How difficult should the game be?**

I think the game should be easy to learn but hard to master. This means that there is always room for player improvement which can keep players engaged for long periods of times. Just look at Team Fortress 2, for example, which is a game that has been around for the longest time but due to its incredibly high skill ceiling it still has players playing the game to improve.

**Graphical style?**

Personally I do not think the graphical style matters but my preference would be 2D pixel art.

**What type of audience should this game aim towards? (e.g The Competitive Scene, Casual audience)**

I think the game can aim for both through the implementation of a versus mode and a coop mode. This also adds more variety to your game as there can be more to do than just beat bots.

**How should a tutorial be implemented? (Silent tutorial?)**

Silent tutorials are the best for immersion but can be fairly unclear and to make the game more accessible to a group of people I think an active tutorial would be much better.

**How should randomness effect gameplay?**

Randomness is a very important part of games, however the randomness should not make gameplay feel unfair or make a win feel unfair. This can be done by promoting input randomness over output randomness such that the player can react to randomness rather than fall victim to randomness.

**Camera handling, 3rd person, 1st person?**

Since the game will have to support multiple people sitting at one laptop screen it is best to have the camera be still and view the entire map and the players move across the screen. This way there is no confusion and the game remains readable and clear.

**Accessibility?**

The game should definitely be accessible to a large player base and the initial learning of how to play should be simple and intuitive.

**Controller support?**

I think that this is a must have to make the game accessible to multiple people easily.

**Platform support?**

Multiple platforms such as different OS should be done for the purpose of accessibility but I do not think its necessary to provide mobile / console support.

Good luck,  
Max

*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
  - a) Allow the means for 2-way communication between the server and client
  - b) A standardised packet format will be designed in the design section of the project
  - c) The server should adjust for packet loss and handle high ping / delayed packets appropriately for both incoming, and outgoing data
  - d) Each client should be assigned a unique ID
2. Authorise the clients incoming data about position and client state
  - a) 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
  - a) 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
  - b) Snapshots are sent once every server tick
4. Calculate physics of entities
  - a) Players are user-controlled entities that act on standard physics that can be controlled by commands sent via the clients
  - b) 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
  - a) 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. Read from a configuration file which allows the user to set the server port

### The client should:

7. Provide a GUI for the user
  - a) This should be usable by a naïve user
8. Host a server on the client pc with given settings / options that the user provides
9. Handle more than one player on a device and tell the server accordingly to allow for local play to work over multiplayer as well
10. Connect to a server with the IP address and port a user provides
11. Receive packets from the server
  - a) Unwrap the packets and generate the snapshot client side for the user
  - b) 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
  - c) 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

12. Display / Render the player and world onto the screen
13. Provide an interface for the user to interact with the world
  - a) 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
14. Provide a method of connecting to a given server
  - a) GUI
    - User can input the server's IP and Port and the game will attempt to connect to that server
  - b) 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
15. Send groups of player actions in "action snapshot" packets to the server
  - a) 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

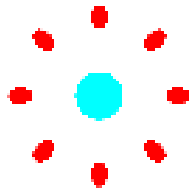
---

## 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.

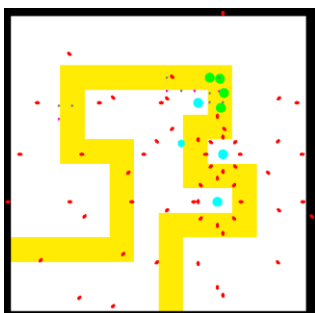
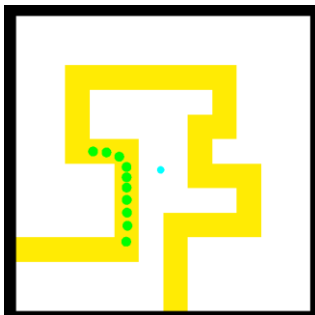
In a traditional tower defence game the towers actively target the incoming enemies, however since the game needs to factor the player into account I will make towers shoot in 8 directions around them such that the player will always be in the cross fire of the towers. Because of this, towers will also 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

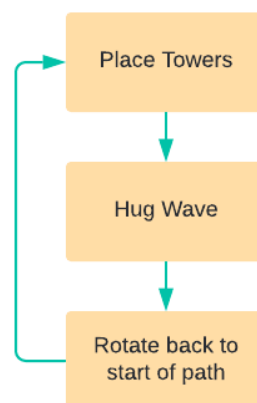
After producing a quick prototype, I sent it over to my primary user Max and spoke with him at length about the direction of this idea and how to “Follow the fun”. Below is a transcript of part of what we spoke about:

- ChrisT** Today at 11:31  
For my A-Level project I was going to make a multiplayer game for us to play during our lunch breaks. Do you have any thoughts for this ?
- MaxC** Today at 11:32  
Do you have an initial idea for what sort of game you want to make?
- ChrisT** Today at 11:33  
I was thinking of something along the lines of a tower defence shooter mixed with a bullet hell. To achieve this I plan on having towers that can also hurt the player.
- MaxC** Today at 11:35  
Will the towers specifically aim at the player? Otherwise there isn't much challenge.
- ChrisT** Today at 11:35  
Yeah, so rather than having towers aim at the player they will shoot periodically in 8 directions:
- 
- MaxC** Today at 11:36  
Surely that's too easy to exploit?  
If the towers are so predictable then the player can place them in such a way where there are "safe zones" and can afk.
- ChrisT** Today at 11:37  
I plan on fixing that by having enemy drop money on the path such that players have to pick them up to place more towers forcing them to move.
- MaxC** Today at 11:38  
Once their defense is strong enough they can still afk though. Since there is no need to collect more money.  
Maybe make player health drain over time such and enemies drop health such that regardless they have to be moving and collecting.  
What do you have for tower variation / enemy variation?
- ChrisT** Today at 11:40  
I was thinking along the lines of enemies that allow bullets to pass through them or enemies with different pathing.  
For towers I was aiming for different types like a laser tower that simply shoots a beam out periodically.
- MaxC** Today at 11:41  
For towers I think there are 2 main things:  
- Player efficiency (how easy it is for the tower to kill the player)  
- Wave efficiency (how easy it is for the tower to kill enemies)  
You need different towers that have different ratios of player efficiency vs wave efficiency

*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” (the ability to interact physically with the enemies such that they block the tower shots for you) 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 were 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.

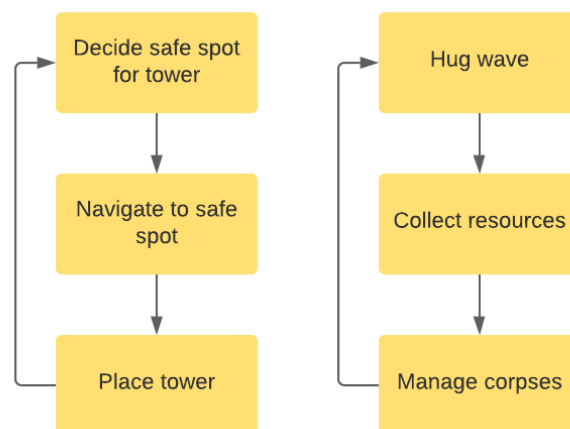
## Scaling into multiplayer / cooperative gameplay

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).

I want to make a clear distinction between these two roles; however, I also want both players to be able to interchange and swap roles on their own. Another problem is that currently, 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. Another interesting thought would be to make towers damage each other; this not only makes tower placement more important but increases the importance of the tower placing role. To make navigating waves, and thus “wave hugging” more interesting I also thought about enemies dying and creating a corpse that protects their fellow enemies from shots for a bit, and thus it is important for the corpses to be pushed out of the way of the path.

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 players that are playing.
2. Each playthrough should be procedurally generated.
3. Upon entering a level, the game should:
  - a. Display the layout of the path the enemies will take.
  - b. Allow players to move around the level.
  - c. Have enemies spawn in waves that follow the path.
  - d. Abide by standard tower defence rules:
    - i. Enemies spawn at one end of the path and upon reaching the end the player loses (either entirely or some form of health system).
    - ii. Players can kill enemies using towers.
    - iii. Enemies drop a collectable resource used in creating towers.
  - e. Tower shots can hurt the players as well.
  - f. Towers self-destruct at the end of their lifetime.

### Stage 2: Enemy variation

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 or wave formats.
  - a. Enemies that enter in a tightly packed wave.
  - b. Enemies that are spread out.
  - c. Enemies that move faster / in odd patterns.
3. Possible enemy variation through different behaviour.
  - a. Enemies may stagger and stall in "safe areas" along the path where bullets do not cross and hastily cross "dangerous areas" filled with bullets.
  - b. Enemies may move quickly in a straight line but slowly along turns.
  - c. Enemies may wait for other enemies in "safe areas" and stick together before proceeding.

### Stage 3: Quality of life:

1. Settings to customize the gameplay:
  - a. Ability to disable of certain enemy types.
  - b. Game modifiers:
    - i. Slow motion / Bullet time.
    - ii. All homing bullets.
    - iii. Enemies getting through the path instantly cause a loss.
  - c. Ability to disable permanent death.
  - d. Ability to disable different tower types.
  - e. Slow down default game speed.

### Stage 4: Polish:

1. Entity ragdolls (physics-based corpses)
2. Particles and effects.
3. Character animations.
  - a. 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.

---

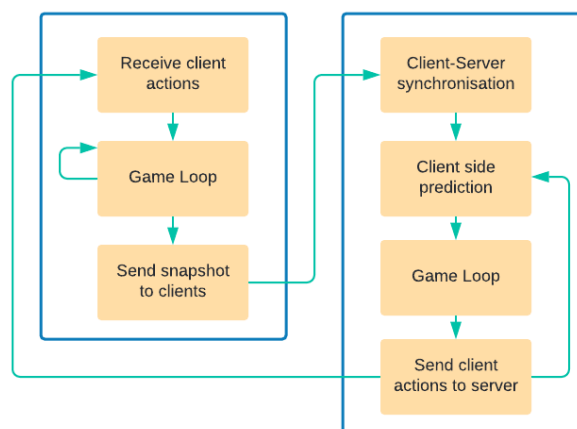
## Design – Overall System Summary

---

### 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.

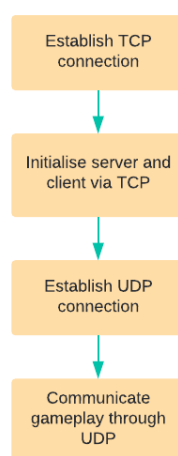


*Flow diagram showing how the server (left) and client (right) will interact with each other.*

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 <sup>[3]</sup>.

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. 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

---

<sup>3</sup> [https://www.isoc.org/inet97/proceedings/F3/F3\\_1.HTM](https://www.isoc.org/inet97/proceedings/F3/F3_1.HTM)

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 like 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 100<sup>th</sup> packet received may not be the 100<sup>th</sup> 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 am going to 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 packet’s *acknowledgment* value minus the position of the bit in the bit field. For example, if a packet of *acknowledgment* 100 is received with the 1<sup>st</sup> and 3<sup>rd</sup> 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 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.

### 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 for the most part rule out this problem.

### 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

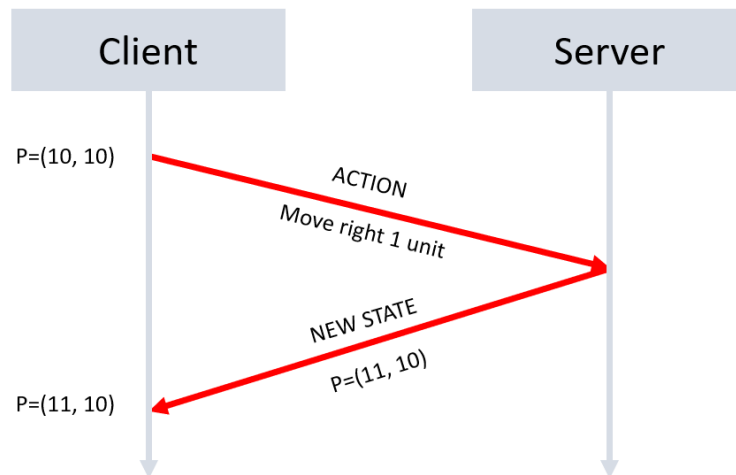
My game’s network will use server authoritative design. This means that everything happens in the server whilst the clients act as “privileged spectators” of the game. In this way the clients will send inputs such as key presses and commands to the server rather than information such as the clients player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

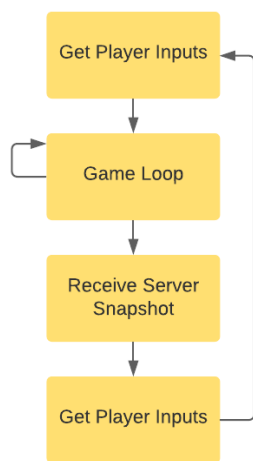
The problem with this system comes from the fact that this would mean all clients are viewing the game in the past and all player inputs will be delayed as they must be passed to the server and then wait for a server response:





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:

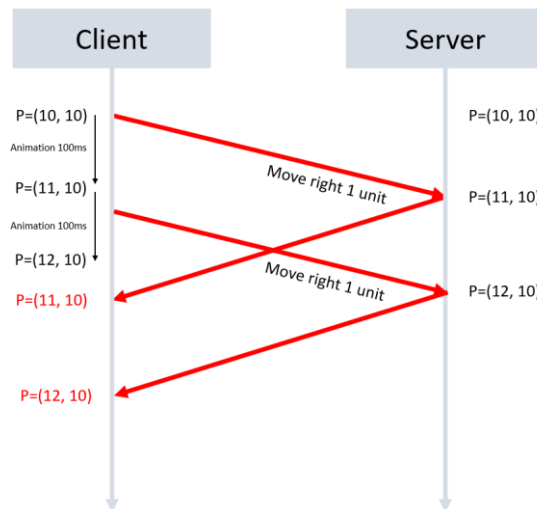


Diagram showing client-side prediction where  $p$  is the position of the player

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 simply 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.

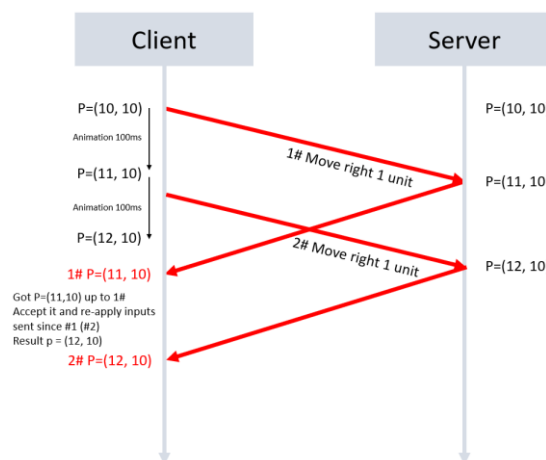


Diagram showing client-side prediction with server reconciliation where  $p$  is the position of the player

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:

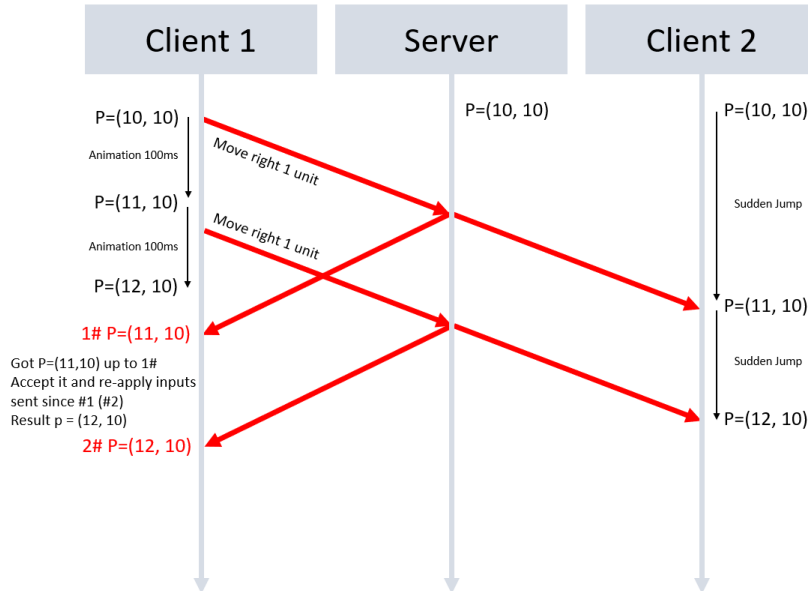


Diagram showing 2 clients receiving updates from the server where  $p$  is the position of the player

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  $t=1000\text{ms}$ , then from  $t=1000\text{ms}$  to  $t=1100\text{ms}$  the client will show the movement of the player that the server had sent from  $t=900\text{ms}$  to  $t=1000\text{ms}$ . In this way the client is always showing actual movement data except its 100ms late:

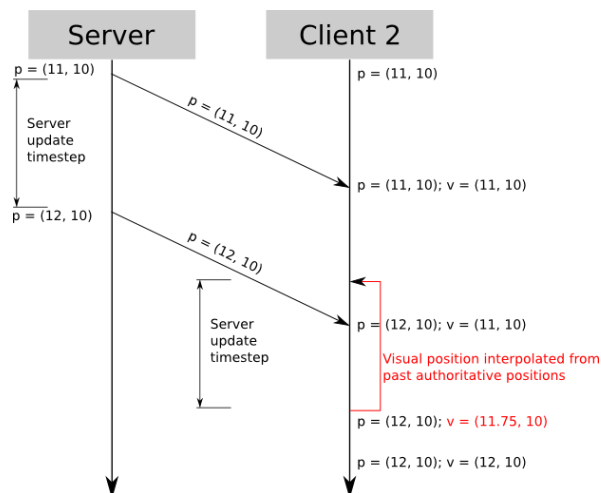


Diagram showing client 2 using the past server’s data to show entity movement in the past where  $p$  is the position sent from the server and  $v$  is the position shown by client 2

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”<sup>[4]</sup> 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 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. This method only really applies to instantaneous actions such as the shot of a gun in an first person shooter, if the game is more physics based using projectiles, roll back may not be necessary.

### 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.

Once again, since I am most likely using this on a LAN setting this is not much of an issue and I may be able to use standard compression methods such as storing states as single bits etc...

### Network Protocol

For my game I will be designing a simply protocol for how the server and client should communicate. Since my game is relatively simple, I only really need to define 3 layers:

- Initial connection
- Communication
- End of connection

#### Initial connection

The initial connection will involve a client sending the first *sync* packet to the server. Upon the server receiving this, the server will respond with a replying *sync* packet. The *sync* packet will contain all the information necessary for the client and server to sync up their base states such as what tick the server / client is on to sync timings, how many players the client will be joining with etc...

#### Communication

Once the initial connection is made the client and server can begin communicating gameplay. This involves the server sending snapshots of the world to the client which the client will unwrap and show to the player. The client will also do the same sending snapshots of the client state containing information on what keys have been pressed.

---

<sup>4</sup> [https://developer.valvesoftware.com/wiki/Source\\_Multiplayer\\_Networking#Lag\\_compensation](https://developer.valvesoftware.com/wiki/Source_Multiplayer_Networking#Lag_compensation)

### End of connection

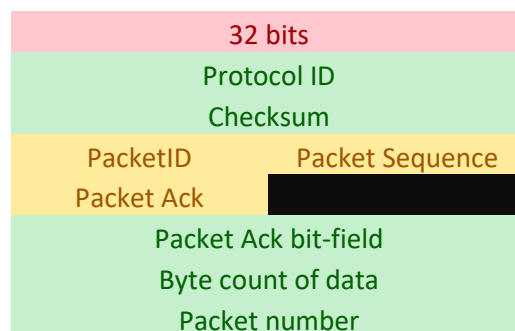
The end of a connection will be defined as the loss of communication between the server and client, in other words, when either party stops sending a stream of packets the connection is considered closed.

### Packet Format

The packet format my protocol will use will first contain a header which holds the information required for UDP as discussed previously such as the *Protocol ID* and *Sequence Number*. This also includes a checksum of the packet for verification on the receiving end. After that, a packet *Server Code* is attached which is simply a number that tells the server / client the purpose of the packet:

<b>SERVER CODE</b>	<b>NAME</b>	<b>FUNCTION</b>
<b>1</b>	SyncPlayers	This is a packet sent to synchronise the players between the client and the server, for example the information that 2 players from the client is joining the server would be sent on this packet
<b>2</b>	ClientSnapshot	This is a packet sent from the client containing a snapshot of the client's state including key presses, client tick rate, and current number of players
<b>3</b>	ServerSnapshot	This is a packet sent from the server containing a snapshot of the server's state including object / entity states and server tick rate

Finally comes the main body of data the packet is storing according to its *Server Code*. The final packet header will look something like:



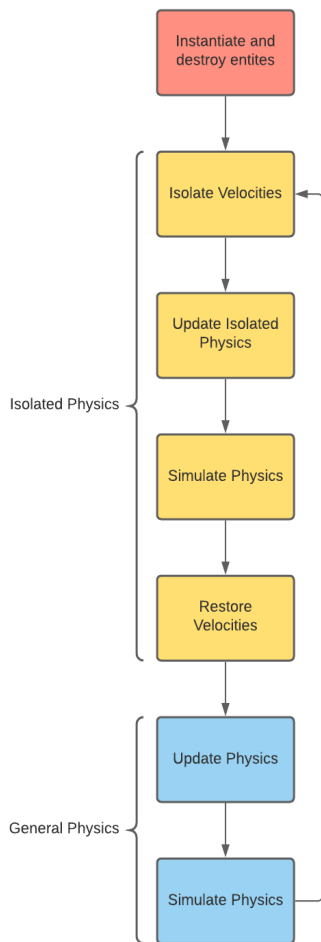
*Diagram showing the header format of a packet.  
Note: the packet number and byte count are for reconstructing split packets.*

## 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:

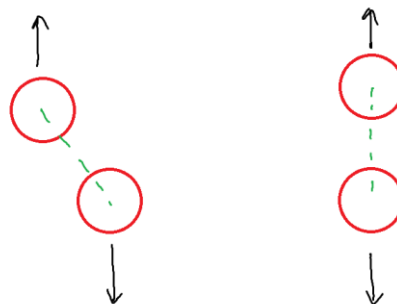


Flow chart showing how my engine interacts with Unity

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).

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:

Entity			
+	EntityID	ulong	
+	EntityType	Type	
+ {virtual}	Set	(Object obj)	void
+ {virtual}	OnDestroy	()	void
+ {virtual}	OnInstantiate	()	void
+ {virtual}	GetBytes	()	byte[]

*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. This byte format will differ from entity to entity and will be discussed when covering each entity individually later.

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.

## Entity Interfaces

Since different entities will interact with the engine differently, I am going to write different systems as interfaces, for example, if an entity uses the render loop it will implement the *IRenderer* interface. The different interfaces I will most likely need (including some of their base methods):

```
public interface _IInstantiatableDeletable
{
    void Delete();
    void Instantiate();
    object Create();
}

public interface IServerSendable : _IInstantiatableDeletable
{
    EntityID ID { get; set; }
    void ServerUpdate();
    byte[] GetBytes();
    object GetSnapshot();
    void ParseBytes(DZNetwork.Packet Data);
    void ParseSnapshot(object Data);
}

public interface IRenderer : _IInstantiatableDeletable
{
    void InitializeRenderer();

    void Render();
}

public interface IUpdatable : _IInstantiatableDeletable
{
    void Update();
}

public interface IPhysicsUpdatable : _IInstantiatableDeletable
{
    void FixedUpdate();
}

public interface IIteratableUpdatable : _IInstantiatableDeletable
{
    void PreUpdate();
    void IteratedUpdate();
}
```

Here *\_IInstantiatableDeletable* will be an interface all entities must implement since I am planning on using it for handling entity generation in my main engine. The rest of the interfaces serve their own role:

- *IServerSendable* : Implements all methods required for an item to be sent over a network where *EntityID* is a unique ID identifying this object over another. The other methods are simply for parsing byte data received from packets.
- *IRenderer* : Describes an object that is used in the render loop and will be rendered into unity.
- *IUpdate* : Describes an object that updates every frame.
- *IPhysicsUpdate* : Describes an object that updates on a fixed frame rate for deterministic physics.
- *IteratableUpdatable* : Describes an object that uses the impulse physics<sup>5</sup> engine for calculating physics.

---

<sup>5</sup> [https://box2d.org/files/ErinCatto\\_SequentialImpulses\\_GDC2006.pdf](https://box2d.org/files/ErinCatto_SequentialImpulses_GDC2006.pdf)



## Entity – Base Components

Before implementing any entities such as the player, I want to declare some entity components. These base components are abstract classes that describe specific entities, for example I am planning to implement:

- *PhysicalJoint* - Implements the interface *IIteratableUpdatable* as I am planning on using it to define physics relationships between entities and I will be using the impulse physics engine architecture to do so.
- *PhysicalObject* - Implements the interface *IPhysicsUpdate* and is being used to define any physics object that needs to be simulated.

For the most part *PhysicalObject* is used to encapsulate Unity's physics objects away from my engine. This means that it implements an interface for Unity's rigid bodies that I can use with my own system.

### DistanceJoint : PhysicalJoint

To implement basic physics into my game I am going to mainly be using distance joints that hold *body chunks* of characters together. This distance joint will be implemented using the impulse engine architecture (impulse/velocity solver) which involves solving physics constraints. In the case of a distance joint, a distance constraint is being solved. My implementation of this solver is heavily based on Erin Catto's Box2D engine's implementation.<sup>6</sup>

### BodyChunk : PhysicalObject

A body chunk will be my primary physics object. It will simply implement a basic circle physics object that I can use for most of my other entities. For example, If I want to create a player entity, it will have a "body" that will be composed of these body chunks and the player entity itself will only have to implement player logic such as movement.

### BodyChunk and DistanceJoint – Networking

The BodyChunk and DistanceJoint classes will most likely contain methods for converting their data into bytes for sending over a network, but they themselves will not implement *IServerSendable* as I plan for other classes to be composed of these.

The byte format of a body chunk and distance joint will most likely only contain their relevant physics values:

```
public struct BodyChunkPacketData
{
    public Vector2 Position;
    public float Rotation;
    public Vector2 Velocity;
    public float AngularVelocity;
    public float InvMass;
    public float InvInertia;
    public float ColliderRadius;
}
```

---

<sup>6</sup> [https://ubm-twvideo01.s3.amazonaws.com/o1/vault/gdc09/slides/04-GDC09\\_Catto\\_Erin\\_Solver.pdf](https://ubm-twvideo01.s3.amazonaws.com/o1/vault/gdc09/slides/04-GDC09_Catto_Erin_Solver.pdf)

```
public struct DistanceJointPacketData
{
    public float Distance;
    public Vector2 Anchor;
    public float ARatio;
    public float BRatio;
}
```

## Player

The player entity will most likely be a class composed of *Body Chunks* and *Distance Joints* and will provide an interface to treat them as the player. The player needs to have certain functionality implemented for my game:

1. Needs to be able to move.
2. Needs to be able to place towers.
3. Requires client-side prediction and server reconciliation.
4. Requires an interface for sending over the network.

### Player – Movement

Movement of the player will be controlled externally. I will most likely accomplish this by having a primary controller class that feeds keyboard / controller inputs into the player class via a movement Vector. This will also allow for the movement logic to be the same for the server and client, just a different primary controller would be used, such as the parsing of client snapshots for the server, and some form of input manager on the client.

### Player – Placing Towers

I will most likely have this functionality done on the server-side such and have the towers sync with the client via the network. Essentially the client performs the place tower action and that action is sent to the server which performs the action and lets the tower sync with the next snapshot delivery.

### Player – Client-Side Prediction and Server Reconciliation

Performing client-side prediction and server reconciliation would require the player, on the client side, to store all the inputs it has been performing. I could do this in a dictionary, tagging each input with an ID that the server can reply with for reconciliation, however a dictionary is not very cut out for the number of key snapshots being enqueued and dequeued. A queue implementation could work, but in the case that the connection is dropped for a couple of seconds, I would be dequeuing several key snapshots at once to clear them all. Instead, I will use a Linked List such that I can very easily queue and dequeue items as well as split the linked list and dequeue a block of items at once if needed.

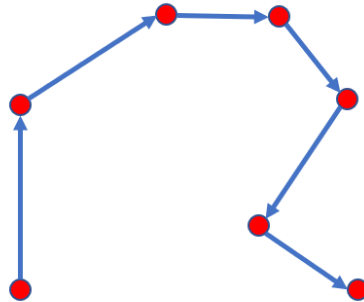
The key snapshots themselves will most likely be small structures containing the inputs in a meaningful format such as a vector to represent a joystick or arrow key input.

## Enemy

The enemy entity will most likely be implemented similarly to the player and have the primary controller be some form of algorithm to have the enemy traverse a given path. Because of this I might make the enemy and player class both inherit from another abstract class.

### Enemy – Movement

The movement of the enemy will be controlled on the server. The algorithm being used to have the enemy traverse a path is quite simple as it uses the fact that a path can be defined with waypoints such that the enemy can move from waypoint to waypoint tracing out the path:



*Image showing how an enemy may trace a path provided some waypoints (red circles).*

I would have each waypoint stored in a list and simply move to the next element when the current waypoint is reached.

## Projectiles / Bullets

Projectiles such as bullets shot from towers will also be handled primary by the server and have the client interpolate.

### Projectiles / Bullets – Collision Detection

Since the client and server will most likely be seeing very different things I will stick with authoritative design and have the server say whether a client got hit by a projectile or not. Because of this, high ping / bad connections can cause the player to get hit by a projectile client side, but not server-side as on the client they may have walked into a projectile (due to client-side prediction), but the server has yet to receive that you moved yet. Sadly, nothing can really be done about this scenario.

## Towers

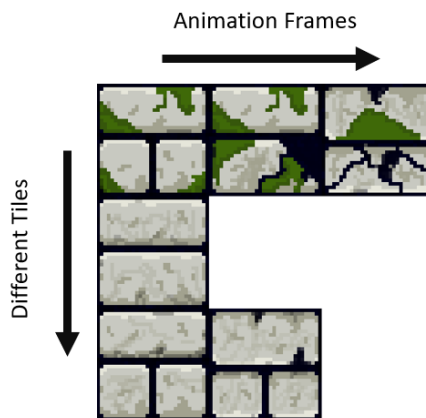
As discussed before the placement of towers is handled server-side, however, because of this there is a noticeable delay between the player placing a tower and having it appear. I do not think this delay is bad as it does not effect have that large of an impact as the responsiveness of player movement. If I were to compensate for this delay, I could very simply implement a similar system to player movement and place the towers and wait for a reconciliation response from the server.

## 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.

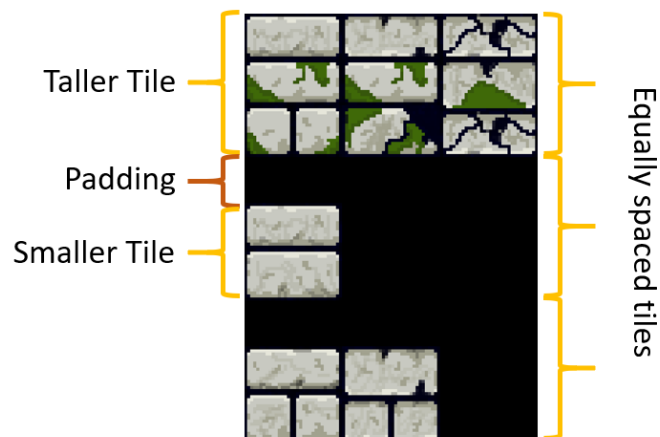
### Tile maps – Tile Pallets

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://alwilliam.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://alwilliam.itch.io/azulejo-32x32> for the tile sheet)

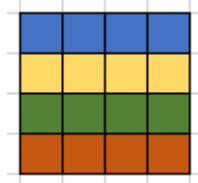
Each tile map will also have to be split into chunks of a pre-defined size such that I do not have to pass the entire world map to another client over the network but only specific chunks around each client. This is not terribly relevant however since my game will primarily be played on small maps.

## 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)*

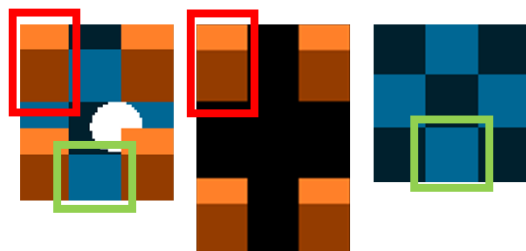
Unity, however, does not allow for different parts of a sprite to render behind or in front of another sprite, and thus the tile map will need to be split into different sprites. Because of this I can render each tile onto individual sprites and render those in Unity. These sprites can sample their textures 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*

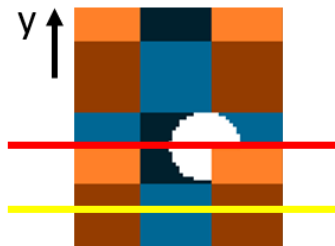


*Diagram showing how sprites would be sampled from the main textures*

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

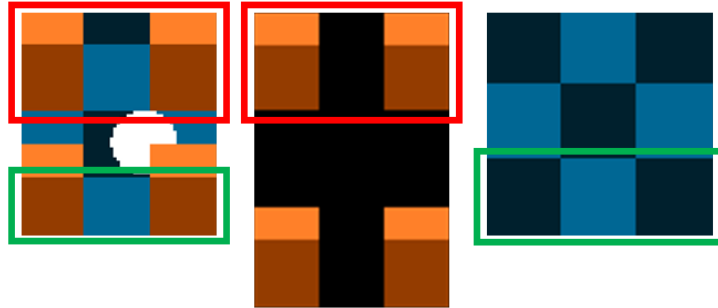
Each tile map will only transmit vital data over the network such as:

- 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.

## 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.

## Tile maps – Procedural Path Generation

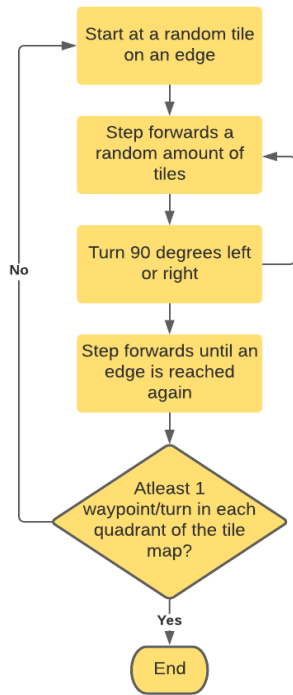
For my game I will need to procedurally generate a path for the enemies to take. This can be done in several ways; my implementation will use a simple waypoint system. This is useful since the waypoints can also be used for traversal of the path. Simply pick a starting point along the edge of the tile map. This will be the first way point. From here the path can continue in any direction, however I think it looks nicer if the paths were straighter and did not run alongside the walls, so the first way point at the edge of the tile map will always have the path point away from the edge:



*Diagram showing a yellow path leaving the edge (defined by the orange walls) of the tile map.*

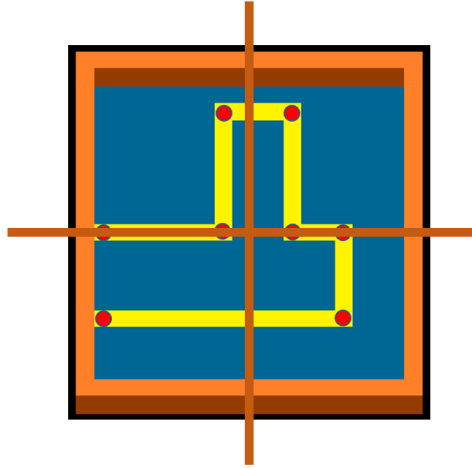
From this point the path can be extended by a random number of units before another waypoint is made. At a waypoint, the path can change direction, but I will restrict the path to only change direction at 90-degree angles. This can be repeated for as many turns as you want and then on the final turn simply extend the path until it hits the edge of the tile map again.





Flowchart of algorithm used to procedurally generate paths.

This simple algorithm does have a few issues, firstly, since the path lengths are random, the path generated may not scale across the entire provided area and may simply move in a loop in the corner. This can be easily fixed by doing the generation multiple times until a path is generated where the waypoints are evenly spread out. The method I'll use to define what "spread out" is will simply be a check that sees if a waypoint is present in all 4 corners (quadrants) of a tile map:



As can be seen here, there is at least 1 waypoint (red circle) in each quadrant of the tile map (defined by orange lines)



---

## Technical Solution

---

### Overview Guide

Group	Overview	Location
A	Use of queues for storing packets to process	ServerHandle.cs Line:39-59
A	Implementation of linked list for reducing network jitter and queuing snapshots	JitterBuffer.cs Line:9-118, Client/Game.cs Line:78-157
A	Complex client-server model	DZUDPSocket.cs, DZClient.cs, DZServer.cs, Game.cs, Packet.cs
A	Hashing	DZUDPSocket.cs Line:95-102
A	Complex user defined algorithm - optimisation on the GPU, optimisation of tilemap rendering and caching objects	TilemapComputeShader.compute, Tilemap.cs Line:316-372, 379-548
A	Advanced Matrix operations	Math2DExtensions.cs Line:14-67, DistanceJoint.cs Line:90-138
A	Complex OOP model	UpdatableAndDeletable.cs
B	Multi dimensional arrays (although stored as 1 dimensional arrays)	Tilemap.cs Line:157-165, PacketHandler.cs Line: 73-126
B	Use of Dictionaries	DZUDPSocket.cs Line:66-82
B	Reading text file	Server/Loader.cs Line:28
B	Simple user defined algorithm - Path generation	Server/Main.cs Line:107-239
C	Linear Search	Client/Main.cs Line:78-86
C	Single dimensional array	PacketHandler.cs Line: 73-126

---

## Technical Solution – Full Code

---

---

### Assets/DZNetwork/DZClient.cs

---

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Net.Sockets;
using System.Net;

namespace DZNetwork
{
    public class DZClient : DZUDPSocket
    {
```

```

    public Action<RecievePacketWrapper> PacketHandle; //Delegate that
is called on recieving a packet
    public Action DisconnectHandle;
    private bool DisconnectTrigger = true;
    public Action ConnectHandle;
    public const int ConnectionLifeTime = 150;
    public uint TicksSinceLastConnection = 0;
    public Action<SentPacketWrapper> PacketLostHandle;

    public bool Connected = false;
    public bool SocketConnected
    {
        get { return Socket.Connected; }
        private set { }
    }

    public DZClient() : base(4096)
    {
        TicksSinceLastConnection = ConnectionLifeTime;
    }

    public void FixedUpdate()
    {
        Tick();

        TicksSinceLastConnection++;
        if (TicksSinceLastConnection > ConnectionLifeTime)
        {
            Connected = false;
            if (!DisconnectTrigger)
            {
                DisconnectHandle();
            }
        }

        PacketHandler.RemoveAcknowledgement(Socket.RemoteEndPoint as IPEndPoint);
        DisconnectTrigger = true;
    }
    else
    {
        Connected = true;
        DisconnectTrigger = false;
    }
}

    public void Connect(string Address, int Port)
    {
        Socket.Connect(IPAddress.Parse(Address), Port);
        BeginReceive();
    }

    public void Send(Packet Packet, ServerCode ServerCode)
    {
        Packet.InsertServerCode(ServerCode);
        PacketHandler.PacketGroup PacketGroup =
PacketHandler.GeneratePackets(Packet, Socket.RemoteEndPoint as IPEndPoint);
        for (int i = 0; i < PacketGroup.Packets.Length; i++)
            Send((ushort)(PacketGroup.StartingPacketSequence + i),
ServerCode, PacketGroup.Packets[i]);
    }
}

```

```

Dictionary<long, PacketReconstructor> PacketsToReconstruct = new
Dictionary<long, PacketReconstructor>();
private class PacketReconstructor
{
    public int PacketByteCount;
    public int ProcessedPacketCount;
    public byte[] PacketIndex;
    public byte[] Data;
}

protected override void OnReceive(IPEndPoint ReceivedEndPoint)
{
    TicksSinceLastConnection = 0;
    if (Connected == false)
    {
        Connected = true;
        ConnectHandle();
    }
}

protected override void
OnReceiveConstructedPacket(RecievePacketWrapper Packet)
{
    PacketHandle?.Invoke(Packet);
}

protected override void OnPacketLost(SentPacketWrapper Packet)
{
    PacketLostHandle?.Invoke(Packet);
}
}
}

```

---

### *Assets/DZNetwork/DZServer.cs*

---

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Net.Sockets;
using System.Net;

namespace DZNetwork
{
    public class DZServer : DZUDPSocket
    {
        public Action<RecievePacketWrapper> PacketHandle; //Delegate that
is called on recieving a packet
        public Action<IPEndPoint> DisconnectHandle;
        public Action<IPEndPoint> ConnectHandle;
        public Action<SentPacketWrapper> PacketLostHandle;

        public const int ConnectionLifeTime = 150;
        private object DeviceUpdate = new object();
        public Dictionary<IPEndPoint, uint> ConnectedDevices;
    }
}

```

```

public DZServer() : base(4096)
{
    ConnectedDevices = new Dictionary<IPEndPoint, uint>(new
IPEndPointComparer());
}

private List<IPEndPoint> Disconnects = new List<IPEndPoint>();
public void FixedUpdate()
{
    Tick();

    Disconnects.Clear();
    List<IPEndPoint> Connections = ConnectedDevices.Keys.ToList();
    foreach (IPEndPoint EndPoint in Connections)
    {
        ConnectedDevices[EndPoint]++;
        if (ConnectedDevices[EndPoint] > ConnectionLifeTime)
        {
            Disconnects.Add(EndPoint);
        }
    }
    for (int i = 0; i < Disconnects.Count; i++)
    {
        PacketHandler.RemoveAcknowledgement(Disconnects[i]);
        ConnectedDevices.Remove(Disconnects[i]);
        DisconnectHandle(Disconnects[i]);
    }
}

public void Connect(int Port)
{
    Socket.Bind(new IPEndPoint(IPAddress.Any, Port));
    UnityEngine.Debug.Log("Server opened on port: " + Port);
    BeginReceive();
}

public void Send(Packet Packet, ServerCode ServerCode)
{
    List<IPEndPoint> Connections = ConnectedDevices.Keys.ToList();
    if (Connections.Count == 0) return;

    Packet.InsertServerCode(ServerCode);
    foreach (IPEndPoint EndPoint in Connections)
        if (EndPoint != null)
        {
            PacketHandler.PacketGroup PacketGroup =
PacketHandler.GeneratePackets(Packet, EndPoint);
            for (int i = 0; i < PacketGroup.Packets.Length; i++)
                SendTo((ushort)(PacketGroup.StartingPacketSequence
+ i), ServerCode, PacketGroup.Packets[i], EndPoint);
        }
}

public void SendTo(Packet Packet, ServerCode ServerCode, IPEndPoint
EndPoint)
{
    Packet.InsertServerCode(ServerCode);
    PacketHandler.PacketGroup PacketGroup =
PacketHandler.GeneratePackets(Packet, EndPoint);
    for (int i = 0; i < PacketGroup.Packets.Length; i++)

```

```

        SendTo((ushort) (PacketGroup.StartingPacketSequence + i),
ServerCode, PacketGroup.Packets[i], EndPoint);
    }

    protected override void OnReceive(IPEndPoint ReceivedEndPoint)
    {
        lock (DeviceUpdate)
        {
            if (!ConnectedDevices.ContainsKey(ReceivedEndPoint))
            {
                ConnectHandle(ReceivedEndPoint);
                ConnectedDevices.Add(ReceivedEndPoint, 0);
            }
            else
                ConnectedDevices[ReceivedEndPoint] = 0;
        }
    }

    protected override void
OnReceiveConstructedPacket(RecievePacketWrapper Packet)
    {
        PacketHandle?.Invoke(Packet);
    }

    protected override void OnPacketLost(SentPacketWrapper Packet)
    {
        PacketLostHandle?.Invoke(Packet);
    }
}
}

```

---

### *Assets/DZNetwork/DZUDPSocket.cs*

---

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Net.Sockets;
using System.Net;

namespace DZNetwork
{
    public class DZUDPSocket
    {
        public const int PacketLifetime = 60;

        public float RoundTripTime = 0;

        public readonly int BufferSize;
        protected readonly int BufferStride;

        private object ReceiveBufferLock = new object();
        protected byte[] ReceiveBuffer;

        protected Socket Socket;
        protected EndPoint EndPoint = new IPEndPoint(IPAddress.Any, 0);
    }
}

```

```

    public DZUDPSocket(int BufferSize, AddressFamily AddressFamily =
AddressFamily.InterNetwork)
    {
        this.BufferSize = BufferSize;
        BufferStride = BufferSize - PacketHandler.HeaderSize;
        ReceiveBuffer = new byte[BufferSize];
        Socket = new Socket(AddressFamily, SocketType.Dgram,
ProtocolType.Udp);
        Socket.SetSocketOption(SocketOptionLevel.IP,
SocketOptionName.ReuseAddress, true);
        //https://stackoverflow.com/questions/38191968/c-sharp-udp-an-
existing-connection-was-forcibly-closed-by-the-remote-host
        Socket.IOControl(
            (IOControlCode)(-1744830452),
            new byte[] { 0, 0, 0, 0 },
            null
        ); //Ignores UDP exceptions
    }

    protected void BeginReceive()
    {
        Socket.BeginReceiveFrom(ReceiveBuffer, 0, ReceiveBuffer.Length,
SocketFlags.None, ref EndPoint, ReceiveCallback, null);
    }

    public void Tick()
    {
        UpdateReconstructedPackets();
        UpdateAcknowledgedPackets();
    }

    public class RecievePacketWrapper
    {
        public int PacketID;
        public IPEndPoint Client;
        public Packet Data;
    }

    public class SentPacketWrapper
    {
        public IPEndPoint Client;
        public int Lifetime = 0;
        public ushort PacketSequence = 0;
        public ServerCode Code = ServerCode.Null;
        public long Epoch = (DateTime.Now - new DateTime(1970, 1, 1, 0,
0, 0, DateTimeKind.Utc)).Ticks / 10000;
    }

    private Dictionary<PacketIdentifier, SentPacketWrapper> SentPackets
= new Dictionary<PacketIdentifier, SentPacketWrapper>();
    private void UpdateAcknowledgedPackets()
    {
        lock (SentPackets)
        {
            List<PacketIdentifier> Keys = SentPackets.Keys.ToList();
            foreach (PacketIdentifier Key in Keys)
            {
                SentPackets[Key].Lifetime++;
                if (SentPackets[Key].Lifetime > PacketLifetime)
                {
                    OnPacketLost(SentPackets[Key]);
                    SentPackets.Remove(Key);
                }
            }
        }
    }

```



```

    }
    }
}

protected virtual void OnPacketLost (SentPacketWrapper Packet) { }

private struct PacketIdentifier
{
    public IPEndPoint Client;
    public ushort ID;

    public override bool Equals (object Obj)
    {
        return Obj is PacketIdentifier && this ==
(PacketIdentifier)Obj;
    }
    public override int GetHashCode ()
    {
        int Hash = 27;
        if (Client != null)
            Hash = (13 * Hash) + Client.GetHashCode ();
        Hash = (13 * Hash) + ID.GetHashCode ();
        return Hash;
    }
    public static bool operator ==(PacketIdentifier A,
PacketIdentifier B)
    {
        if (ReferenceEquals (A, null) && ReferenceEquals (B, null))
            return true;
        else if (ReferenceEquals (A, null) || ReferenceEquals (B,
null))
            return false;
        return A.ID == B.ID && A.Client.Equals (B.Client);
    }
    public static bool operator !=(PacketIdentifier A,
PacketIdentifier B)
    {
        if (ReferenceEquals (A, null) && ReferenceEquals (B, null))
            return false;
        else if (ReferenceEquals (A, null) || ReferenceEquals (B,
null))
            return true;
        return A.ID != B.ID || !A.Client.Equals (B.Client);
    }
}

private int ReconstructedPacketsClear = 0;
private HashSet<PacketIdentifier> ReconstructedPackets = new
HashSet<PacketIdentifier> ();
private Dictionary<PacketIdentifier, PacketReconstructor>
PacketsToReconstruct = new Dictionary<PacketIdentifier,
PacketReconstructor> ();
private void UpdateReconstructedPackets ()
{
    ReconstructedPacketsClear++;
    if (ReconstructedPacketsClear > PacketLifetime)
        ReconstructedPackets.Clear ();
    lock (PacketsToReconstruct)
    {
        List<PacketIdentifier> Keys =
PacketsToReconstruct.Keys.ToList ();

```

```

        foreach (PacketIdentifier Key in Keys)
        {
            PacketsToReconstruct[Key].Lifetime++;
            if (PacketsToReconstruct[Key].Lifetime >
PacketLifetime)
                PacketsToReconstruct.Remove(Key);
        }
    }
}

private class PacketReconstructor
{
    public int Lifetime = 0;
    public int PacketByteCount = 0;
    public int ProcessedPacketCount = 0;
    public byte[] PacketIndex;
    public byte[] Data;
}

private void ReceiveCallback(IAsyncResult Result)
{
    Packet Data = null;
    int NumBytesReceived = 0;
    IPEndPoint IPEP = null;
    lock (ReceiveBufferLock)
    {
        NumBytesReceived = Socket.EndReceiveFrom(Result, ref
EndPoint);
        IPEP = EndPoint as IPEndPoint;
        Data = new Packet(ReceiveBuffer, 0, NumBytesReceived);
    }
    Socket.BeginReceiveFrom(ReceiveBuffer, 0, ReceiveBuffer.Length,
SocketFlags.None, ref EndPoint, ReceiveCallback, null);

    int ReceivedProtocolID = Data.ReadInt();
    if (ReceivedProtocolID != PacketHandler.ProtocolID) return;

    int ReceivedChecksum = Data.ReadInt();
    int CalculatedChecksum =
PacketHandler.CalculateChecksum(Data.ReadableBuffer);
    if (ReceivedChecksum != CalculatedChecksum) return;

    ushort PacketID = Data.ReadUShort();
    PacketIdentifier PacketIdentifier = new PacketIdentifier()
    {
        Client = IPEP,
        ID = PacketID
    };

    ushort RemotePacketSequence = Data.ReadUShort();
    ushort PacketAcknowledgement = Data.ReadUShort();
    int PacketAcknowledgementBitField = Data.ReadInt();

    lock (PacketHandler.PacketAcknowledgements)
    {
        PacketHandler.Acknowledgement Ack =
PacketHandler.GetAcknowledgement(IPEP);

        //Update Acknowledgements to return
        if (RemotePacketSequence > Ack.PacketAcknowledgement)
        {

```

```

        int SkippedSequences = RemotePacketSequence -
Ack.PacketAcknowledgement;
        Ack.PacketAcknowledgement = RemotePacketSequence;
        Ack.PacketAcknowledgementBitField =
(Ack.PacketAcknowledgementBitField << SkippedSequences) | (1 <<
(SkippedSequences - 1));
    }
    else if (RemotePacketSequence < Ack.PacketAcknowledgement)
    {
        int Difference = Ack.PacketAcknowledgement -
RemotePacketSequence;
        if (Difference < ushort.MaxValue / 2)
        {
            int AcknowledgementPosition =
Ack.PacketAcknowledgement - RemotePacketSequence;
            Ack.PacketAcknowledgementBitField =
Ack.PacketAcknowledgementBitField | (1 << (AcknowledgementPosition));
        }
        else //Sequence number wrap around
        {
            int SkippedSequences = ushort.MaxValue -
Ack.PacketAcknowledgement + RemotePacketSequence;
            Ack.PacketAcknowledgement = RemotePacketSequence;
            //its different to the normal update as the skipped
sequences calculation does not include 0 so its 1 behind
            Ack.PacketAcknowledgementBitField =
(Ack.PacketAcknowledgementBitField << (SkippedSequences + 1)) | (1 <<
SkippedSequences);
        }
    }
}

//Update packet queue to check what packets were lost
lock (SentPackets)
{
    long CurrentEpoch = (DateTime.Now - new DateTime(1970, 1,
1, 0, 0, 0, DateTimeKind.Utc)).Ticks / 10000;

    PacketIdentifier Identifier = new PacketIdentifier()
    {
        ID = PacketAcknowledgement,
        Client = IPEP
    };

    if (SentPackets.ContainsKey(Identifier))
    {
        RoundTripTime += (CurrentEpoch -
SentPackets[Identifier].Epoch - RoundTripTime) * 0.1f;
        SentPackets.Remove(Identifier);
    }
    for (int i = 0; i < 32; i++)
    {
        Identifier.ID--;
        if (SentPackets.ContainsKey(Identifier) &&
((PacketAcknowledgementBitField & (1 << i)) != 0))
        {
            RoundTripTime += (CurrentEpoch -
SentPackets[Identifier].Epoch - RoundTripTime) * 0.1f;
            SentPackets.Remove(Identifier);
        }
    }
}

```

```

    }

    if (ReconstructedPackets.Contains(PacketIdentifier))
        return; //Duplicate Packet

    int PacketByteCount = Data.ReadInt();
    int PacketIndex = Data.ReadInt();

    byte[] ByteData = new byte[NumBytesReceived -
PacketHandler.HeaderSize];
    Buffer.BlockCopy(Data.ReadableBuffer, PacketHandler.HeaderSize,
ByteData, 0, ByteData.Length);

    lock (PacketsToReconstruct)
    {
        if (!PacketsToReconstruct.ContainsKey(PacketIdentifier))
        {
            PacketReconstructor Reconstructor = new
PacketReconstructor()
            {
                PacketIndex = new
byte[UnityEngine.Mathf.CeilToInt(PacketByteCount / (float)BufferSize)],
//Ensure that the client and server have the same buffer size, otherwise
this code doesnt work as the buffersize referenced here is for client
                Data = new byte[PacketByteCount]
            };

            PacketsToReconstruct.Add(PacketIdentifier,
Reconstructor);
        }
        if
(PacketsToReconstruct[PacketIdentifier].PacketIndex[PacketIndex] == 0)
        {

PacketsToReconstruct[PacketIdentifier].PacketIndex[PacketIndex] = 1;

PacketsToReconstruct[PacketIdentifier].ProcessedPacketCount += 1;
            Buffer.BlockCopy(ByteData, 0,
PacketsToReconstruct[PacketIdentifier].Data, PacketIndex * BufferStride,
ByteData.Length);

            if
(PacketsToReconstruct[PacketIdentifier].ProcessedPacketCount ==
PacketsToReconstruct[PacketIdentifier].PacketIndex.Length)
            {
                OnReceiveConstructedPacket(new
RecievePacketWrapper()
                {
                    PacketID = PacketIdentifier.ID,
                    Client = IPEP,
                    Data = new
Packet(PacketsToReconstruct[PacketIdentifier].Data, 0, PacketByteCount)
                });

                PacketsToReconstruct.Remove(PacketIdentifier);
                ReconstructedPackets.Add(PacketIdentifier);
            }
        }
    }

    OnReceive(IPEP);

```

```

    }

    protected virtual void OnReceive(IPEndPoint ReceivedEndPoint) { }

    protected virtual void
OnReceiveConstructedPacket (RecievePacketWrapper ReconstructedPacket) { }

    public void Send(ushort PacketSequence, ServerCode ServerCode,
byte[] Bytes)
    {
        SentPackets.Add(new PacketIdentifier()
        {
            ID = PacketSequence,
            Client = Socket.RemoteEndPoint as IPEndPoint
        }, new SentPacketWrapper()
        {
            PacketSequence = PacketSequence,
            Code = ServerCode
        });
        Socket.BeginSend(Bytes, 0, Bytes.Length, SocketFlags.None,
null, null);
    }

    public void SendTo(ushort PacketSequence, ServerCode ServerCode,
byte[] Bytes, IPEndPoint Destination)
    {
        SentPackets.Add(new PacketIdentifier()
        {
            ID = PacketSequence,
            Client = Destination
        }, new SentPacketWrapper()
        {
            PacketSequence = PacketSequence,
            Code = ServerCode
        });
        Socket.BeginSendTo(Bytes, 0, Bytes.Length, SocketFlags.None,
Destination, SendToCallback, null);
    }

    private void SendToCallback(IAsyncResult Result)
    {
        int NumBytesSent = Socket.EndSendTo(Result);
        OnSendTo(NumBytesSent);
    }

    protected virtual void OnSendTo(int NumBytesSent) { }

    public void Dispose()
    {
        Socket.Dispose();
        OnDispose();
    }

    protected virtual void OnDispose() { }
}
}

```

---

## Assets/DZNetwork/JitterBuffer.cs

---

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace DZNetwork
{
    public class JitterBuffer<T> where T : class
    {
        public class Key
        {
            public T Value;
            public Key Next;

            public Key(T Value)
            {
                this.Value = Value;
            }
        }

        private int _Count = 0;
        public int Count
        {
            get
            {
                return _Count;
            }
        }

        private Key Start = null;
        private Key End = null;
        public T First
        {
            get
            {
                if (Start != null)
                    return Start.Value;
                return default;
            }
        }
        public T Last
        {
            get
            {
                if (End != null)
                    return End.Value;
                return default;
            }
        }

        public Key FirstKey
        {
            get
            {
                return Start;
            }
        }
    }
}
```

```

    }
}
public Key LastKey
{
    get
    {
        return End;
    }
}

public void Add(T Value)
{
    _Count++;
    if (Start == null)
    {
        Start = new Key(Value);
        End = Start;
        return;
    }
    End.Next = new Key(Value);
    End = End.Next;
}

public void Clear()
{
    Start = null;
    End = null;
    _Count = 0;
}

public void Dequeue(int Index)
{
    for (int i = 0; i < Index; i++)
    {
        Start = Start.Next;
        _Count--;
    }
}

public void Dequeue(T From)
{
    while (!EqualityComparer<T>.Default.Equals(Start.Value, From))
    {
        if (Start.Next == null) return;
        Start = Start.Next;
        _Count--;
    }
}

public void Iterate(Action<Key> IterateOperation, Func<Key, bool>
BreakCondition = null)
{
    Key Current = Start;
    for (int i = 0; i < _Count; i++)
    {
        IterateOperation(Current);
        if (BreakCondition != null && BreakCondition(Current))
return;
        Current = Current.Next;
    }
}

```

```
}  
}
```

---

## *Assets/DZNetwork/Packet.cs*

---

```
using System;  
using System.Collections.Generic;  
using System.Linq;  
using System.Net;  
using System.Text;  
using System.Threading.Tasks;  
  
namespace DZNetwork  
{  
    public class IPEndPointComparer : IEqualityComparer<IPEndPoint>  
    {  
        public bool Equals(IPEndPoint A, IPEndPoint B)  
        {  
            return A.Equals(B);  
        }  
  
        public int GetHashCode(IPEndPoint A)  
        {  
            return A.GetHashCode();  
        }  
    }  
  
    public static class PacketHandler  
    {  
        public static int ProtocolID = 0;  
        private static int PacketHeaderSize = sizeof(int) * 2 +  
sizeof(ushort);  
        public static int HeaderSize = PacketHeaderSize + sizeof(ushort) *  
2 + sizeof(int) * 3;  
  
        public static ushort PacketID = 0;  
        public static ushort LocalPacketSequence = 0;  
  
        public class Acknowledgement  
        {  
            public ushort PacketAcknowledgement = 0;  
            public int PacketAcknowledgementBitField = 0;  
        }  
        public static Dictionary<IPEndPoint, Acknowledgement>  
PacketAcknowledgements = new Dictionary<IPEndPoint, Acknowledgement>(new  
IPEndPointComparer());  
  
        public struct PacketGroup  
        {  
            public ushort StartingPacketSequence;  
            public byte[][] Packets;  
        }  
  
        public static Acknowledgement GetAcknowledgement(IPEndPoint  
EndPoint)  
        {
```



```

        if (!PacketAcknowledgements.ContainsKey(EndPoint))
PacketAcknowledgements.Add(EndPoint, new Acknowledgement());
        return PacketAcknowledgements[EndPoint];
    }

    public static void RemoveAcknowledgement(IPEndPoint EndPoint)
    {
        if (PacketAcknowledgements.ContainsKey(EndPoint))
            PacketAcknowledgements.Remove(EndPoint);
    }

    public static int CalculateChecksum(byte[] Data, int Offset =
sizeof(int) * 2)
    {
        int Sum = 0;
        for (int i = Offset; i < Data.Length; i++)
            Sum += Data[i];
        return Sum;
    }

    public static int CalculateChecksum(List<byte> Data, int Offset =
sizeof(int) * 2)
    {
        int Sum = 0;
        for (int i = Offset; i < Data.Count; i++)
            Sum += Data[i];
        return Sum;
    }

    public static PacketGroup GeneratePackets(Packet P, IPEndPoint
EndPoint)
    {
        byte[] Data = P.GetBytes();
        float NumPacketsNoHeader =
UnityEngine.Mathf.Ceil((float)Data.Length / Loader.Socket.BufferSize);
        int NumPackets = UnityEngine.Mathf.CeilToInt((Data.Length +
NumPacketsNoHeader * HeaderSize) / Loader.Socket.BufferSize);
        byte[][] Packets = new byte[NumPackets][];

        byte[] HeaderBytes = new byte[PacketHeaderSize];
        int WriteHead = 0;
        Buffer.BlockCopy(BitConverter.GetBytes(ProtocolID), 0,
HeaderBytes, WriteHead, sizeof(int)); WriteHead += sizeof(int) * 2;
        Buffer.BlockCopy(BitConverter.GetBytes(PacketID), 0,
HeaderBytes, WriteHead, sizeof(ushort));

        PacketGroup PacketGroup = new PacketGroup()
        {
            StartingPacketSequence = LocalPacketSequence
        };

        lock (PacketAcknowledgements)
        {
            Acknowledgement Ack = GetAcknowledgement(EndPoint);

            int RemainingPacketSize = Data.Length;
            int ReadHead = 0;
            for (int i = 0; i < NumPackets; i++)
            {
                int PacketSize = Math.Min(RemainingPacketSize,
Loader.Socket.BufferSize - HeaderSize);

```

```

        Packets[i] = new byte[PacketSize + HeaderSize];
        Buffer.BlockCopy(HeaderBytes, 0, Packets[i], 0,
HeaderBytes.Length);

        int HeaderIndex = HeaderBytes.Length;

Buffer.BlockCopy(BitConverter.GetBytes(LocalPacketSequence), 0, Packets[i],
HeaderIndex, sizeof(ushort)); HeaderIndex += sizeof(ushort);

Buffer.BlockCopy(BitConverter.GetBytes(Ack.PacketAcknowledgement), 0,
Packets[i], HeaderIndex, sizeof(ushort)); HeaderIndex += sizeof(ushort);

Buffer.BlockCopy(BitConverter.GetBytes(Ack.PacketAcknowledgementBitField),
0, Packets[i], HeaderIndex, sizeof(int)); HeaderIndex += sizeof(int);
        Buffer.BlockCopy(BitConverter.GetBytes(Data.Length), 0,
Packets[i], HeaderIndex, sizeof(int)); HeaderIndex += sizeof(int);
        Buffer.BlockCopy(BitConverter.GetBytes(i), 0,
Packets[i], HeaderIndex, sizeof(int));

        Buffer.BlockCopy(Data, ReadHead, Packets[i],
HeaderSize, PacketSize);

        int CheckSum = CalculateCheckSum(Packets[i]);
        Buffer.BlockCopy(BitConverter.GetBytes(CheckSum), 0,
Packets[i], sizeof(int), sizeof(int));

        ReadHead += PacketSize;
        RemainingPacketSize -= PacketSize;

        LocalPacketSequence++;
    }
}

PacketID++;

PacketGroup.Packets = Packets;
return PacketGroup;
}
}

public class Packet
{
    public byte[] ReadableBuffer;

    private List<byte> Buffer;
    public int ReadPosition { get; private set; } = 0;

    /// <summary>
    /// Generates a blank packet
    /// </summary>
    public Packet()
    {
        Buffer = new List<byte>();
    }

    /// <summary>
    /// Generates a packet from which data can be read, making a copy
    /// </summary>
    /// <param name="Data">Bytes to be read</param>
    public Packet(byte[] Buffer, int Start, int Count)
    {

```

```

        ReadableBuffer = new byte[Count];
        System.Buffer.BlockCopy(Buffer, Start, ReadableBuffer, 0,
Count);
    }

    /// <summary>
    /// Generates a packet from which data can be read, without making
a copy
    /// </summary>
    /// <param name="Data">Bytes to be read</param>
    public Packet(byte[] Buffer)
    {
        ReadableBuffer = Buffer;
    }

    public void InsertServerCode(ServerCode Code)
    {
        Buffer.InsertRange(0, BitConverter.GetBytes((int)Code));
    }

    public void InsertChecksum(int Offset)
    {
        Buffer.InsertRange(0,
BitConverter.GetBytes(PacketHandler.CalculateChecksum(Buffer, Offset)));
    }

    public void SeekHeader()
    {
        ReadPosition = PacketHandler.HeaderSize;
    }

    /// <summary>
    /// Reset the packets buffers if passed true
    /// </summary>
    /// <param name="Reset"></param>
    public void Reset(bool Reset)
    {
        if (!Reset)
            return;

        Buffer.Clear();
        ReadableBuffer = null;
        ReadPosition = 0;
    }

    /// <summary>
    /// Converts buffer to byte array
    /// </summary>
    /// <returns></returns>
    public byte[] GetBytes()
    {
        ReadableBuffer = Buffer.ToArray();
        return ReadableBuffer;
    }

    /// <summary>
    /// Writes a range of bytes and sets it to the readable buffer
    /// </summary>
    /// <param name="Data"></param>
    public void WriteRange(byte[] Data)
    {

```

```

        Buffer.AddRange(Data);
        ReadableBuffer = Buffer.ToArray();
    }

    /// <summary>
    /// Writes a range of bytes without setting ReadableBuffer
    /// </summary>
    /// <param name="Data"></param>
    public void Write(byte[] Data)
    {
        Buffer.AddRange(Data);
    }

    /// <summary>
    /// Writes a string without setting ReadableBuffer
    /// </summary>
    /// <param name="Data"></param>
    public void Write(string Value)
    {
        Write(Value.Length); //Add length of string
        Buffer.AddRange(Encoding.ASCII.GetBytes(Value));
    }

    /// <summary>
    /// Writes a byte without setting ReadableBuffer
    /// </summary>
    /// <param name="Data"></param>
    public void Write(byte Value)
    {
        Buffer.Add(Value);
    }

    /// <summary>
    /// Writes an int without setting ReadableBuffer
    /// </summary>
    /// <param name="Data"></param>
    public void Write(int Value)
    {
        Buffer.AddRange(BitConverter.GetBytes(Value));
    }

    /// <summary>
    /// Writes an ushort without setting ReadableBuffer
    /// </summary>
    /// <param name="Data"></param>
    public void Write(ushort Value)
    {
        Buffer.AddRange(BitConverter.GetBytes(Value));
    }

    /// <summary>
    /// Writes a float without setting ReadableBuffer
    /// </summary>
    /// <param name="Data"></param>
    public void Write(float Value)
    {
        Buffer.AddRange(BitConverter.GetBytes(Value));
    }

    /// <summary>
    /// Writes a long without setting ReadableBuffer

```

```

/// </summary>
/// <param name="Data"></param>
public void Write(long Value)
{
    Buffer.AddRange(BitConverter.GetBytes(Value));
}

/// <summary>
/// Writes an ulong without setting ReadableBuffer
/// </summary>
/// <param name="Data"></param>
public void Write(ulong Value)
{
    Buffer.AddRange(BitConverter.GetBytes(Value));
}

/// <summary>
/// Writes a boolean without setting ReadableBuffer
/// </summary>
/// <param name="Data"></param>
public void Write(bool Value)
{
    Buffer.AddRange(BitConverter.GetBytes(Value));
}

/// <summary>
/// Returns the unread length of the packet
/// </summary>
/// <param name="Data"></param>
public int UnreadLength()
{
    return ReadableBuffer.Length - ReadPosition;
}

/// <summary>
/// Set the reading position for the packet
/// </summary>
/// <param name="ReadPosition"></param>
public void Seek(int ReadPosition)
{
    this.ReadPosition = ReadPosition;
}

/// <summary>
/// Skip bytes of a packet
/// </summary>
/// <param name="ReadPosition"></param>
public void Skip(int NumBytes)
{
    ReadPosition += NumBytes;
}

/// <summary>
/// Reads a byte value from the current ReadPosition of the packet
/// </summary>
/// <param name="MoveRead">Move the ReadPosition after read</param>
/// <returns></returns>
public byte ReadByte(bool MoveRead = true)
{
    if (UnreadLength() >= sizeof(byte))
    {

```

```

        byte Value = ReadableBuffer[ReadPosition];
        if (MoveRead)
            ReadPosition += sizeof(byte);
        return Value;
    }
    else
    {
        throw new Exception("Could not read Byte value");
    }
}

/// <summary>
/// Reads a boolean value from the current ReadPosition of the
packet
/// </summary>
/// <param name="MoveRead">Move the ReadPosition after read</param>
/// <returns></returns>
public bool ReadBool(bool MoveRead = true)
{
    if (UnreadLength() >= sizeof(bool))
    {
        bool Value = BitConverter.ToBoolean(ReadableBuffer,
ReadPosition);
        if (MoveRead)
            ReadPosition += sizeof(bool);
        return Value;
    }
    else
    {
        throw new Exception("Could not read Boolean value");
    }
}

/// <summary>
/// Reads a long value from the current ReadPosition of the packet
/// </summary>
/// <param name="MoveRead">Move the ReadPosition after read</param>
/// <returns></returns>
public long ReadLong(bool MoveRead = true)
{
    if (UnreadLength() >= sizeof(long))
    {
        long Value = BitConverter.ToInt64(ReadableBuffer,
ReadPosition);
        if (MoveRead)
            ReadPosition += sizeof(long);
        return Value;
    }
    else
    {
        throw new Exception("Could not read Long value");
    }
}

/// <summary>
/// Reads an ulong value from the current ReadPosition of the
packet
/// </summary>
/// <param name="MoveRead">Move the ReadPosition after read</param>
/// <returns></returns>

```

```

public ulong ReadULong (bool MoveRead = true)
{
    if (UnreadLength() >= sizeof(ulong))
    {
        ReadPosition);
        ulong Value = BitConverter.ToUInt64 (ReadableBuffer,
        if (MoveRead)
            ReadPosition += sizeof(ulong);
        return Value;
    }
    else
    {
        throw new Exception("Could not read Unsigned Long value");
    }
}

/// <summary>
/// Reads a Short value from the current ReadPosition of the packet
/// </summary>
/// <param name="MoveRead">Move the ReadPosition after read</param>
/// <returns></returns>
public short ReadShort (bool MoveRead = true)
{
    if (UnreadLength() >= sizeof(short))
    {
        ReadPosition);
        short Value = BitConverter.ToInt16 (ReadableBuffer,
        if (MoveRead)
            ReadPosition += sizeof(short);
        return Value;
    }
    else
    {
        throw new Exception("Could not read Short value");
    }
}

/// <summary>
/// Reads a ushort value from the current ReadPosition of the
packet
/// </summary>
/// <param name="MoveRead">Move the ReadPosition after read</param>
/// <returns></returns>
public ushort ReadUShort (bool MoveRead = true)
{
    if (UnreadLength() >= sizeof(ushort))
    {
        ReadPosition);
        ushort Value = BitConverter.ToUInt16 (ReadableBuffer,
        if (MoveRead)
            ReadPosition += sizeof(ushort);
        return Value;
    }
    else
    {
        throw new Exception("Could not read UShort value");
    }
}

/// <summary>
/// Reads a int value from the current ReadPosition of the packet

```

```

    /// </summary>
    /// <param name="MoveRead">Move the ReadPosition after read</param>
    /// <returns></returns>
    public int ReadInt (bool MoveRead = true)
    {
        if (UnreadLength() >= sizeof(int))
        {
            int Value = BitConverter.ToInt32 (ReadableBuffer,
ReadPosition);
            if (MoveRead)
                ReadPosition += sizeof(int);
            return Value;
        }
        else
        {
            throw new Exception ("Could not read Int value");
        }
    }

    /// <summary>
    /// Reads a uint value from the current ReadPosition of the packet
    /// </summary>
    /// <param name="MoveRead">Move the ReadPosition after read</param>
    /// <returns></returns>
    public uint ReadUInt (bool MoveRead = true)
    {
        if (UnreadLength() >= sizeof(int))
        {
            uint Value = BitConverter.ToUInt32 (ReadableBuffer,
ReadPosition);
            if (MoveRead)
                ReadPosition += sizeof(uint);
            return Value;
        }
        else
        {
            throw new Exception ("Could not read UInt value");
        }
    }

    /// <summary>
    /// Reads a float value from the current ReadPosition of the packet
    /// </summary>
    /// <param name="MoveRead">Move the ReadPosition after read</param>
    /// <returns></returns>
    public float ReadFloat (bool MoveRead = true)
    {
        if (UnreadLength() >= sizeof(float))
        {
            float Value = BitConverter.ToSingle (ReadableBuffer,
ReadPosition);
            if (MoveRead)
                ReadPosition += sizeof(float);
            return Value;
        }
        else
        {
            throw new Exception ("Could not read Int value");
        }
    }
}

```



```

    /// <summary>
    /// Reads a string value from the current ReadPosition of the
packet
    /// </summary>
    /// <param name="MoveRead">Move the ReadPosition after read</param>
    /// <returns></returns>
    public string ReadString (bool MoveRead = true)
    {
        try
        {
            int Length = ReadInt ();
            string Value = Encoding.ASCII.GetString (ReadableBuffer,
ReadPosition, Length);
            if (MoveRead && Value.Length > 0)
                ReadPosition += Length;
            return Value;
        }
        catch
        {
            throw new Exception ("Could not read String value");
        }
    }
}

```

---

### *Assets/DZNetwork/ServerHandle.cs*

---

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Net;
using System.Text;
using System.Threading.Tasks;

using UnityEngine;

namespace DZNetwork
{
    public static class ServerHandle
    {
        public static Action<DZUDPSocket.RecievePacketWrapper> PacketHandle
= null;
        public static Action<DZUDPSocket.SentPacketWrapper>
LostPacketHandle = null;

        private static readonly List<DZUDPSocket.RecievePacketWrapper>
PacketsToProcess = new List<DZUDPSocket.RecievePacketWrapper> ();
        private static readonly Queue<DZUDPSocket.RecievePacketWrapper>
PacketsProcessing = new Queue<DZUDPSocket.RecievePacketWrapper> ();

        private static readonly List<DZUDPSocket.SentPacketWrapper>
LostPacketsToProcess = new List<DZUDPSocket.SentPacketWrapper> ();
        private static readonly Queue<DZUDPSocket.SentPacketWrapper>
LostPacketsProcessing = new Queue<DZUDPSocket.SentPacketWrapper> ();
    }
}

```

```

    public static void ProcessPacket (DZUDPSocket.RecievePacketWrapper
Packet)
    {
        lock (PacketsToProcess)
        {
            PacketsToProcess.Add (Packet) ;
        }
    }

    public static void HandleLostPacket (DZUDPSocket.SentPacketWrapper
Packet)
    {
        lock (LostPacketsToProcess)
        {
            LostPacketsToProcess.Add (Packet) ;
        }
    }

    public static void FixedUpdate ()
    {
        lock (PacketsToProcess)
        {
            if (PacketsToProcess.Count > 0)
            {
                for (int i = 0; i < PacketsToProcess.Count; i++)
                    PacketsProcessing.Enqueue (PacketsToProcess [i]) ;
                PacketsToProcess.Clear () ;
            }
        }

        lock (LostPacketsToProcess)
        {
            if (LostPacketsToProcess.Count > 0)
            {
                for (int i = 0; i < LostPacketsToProcess.Count; i++)
                    LostPacketsProcessing.Enqueue (LostPacketsToProcess [i]) ;
                LostPacketsToProcess.Clear () ;
            }
        }

        while (PacketsProcessing.Count > 0)
        {
            PacketHandle?.Invoke (PacketsProcessing.Dequeue ()) ;
        }

        while (LostPacketsProcessing.Count > 0)
        {
            LostPacketHandle?.Invoke (LostPacketsProcessing.Dequeue ()) ;
        }
    }
}

```

---

## Assets/Resources/ComputeShaders/ TilemapComputeShader.compute

---

```
#pragma kernel TilemapRender

struct Tile
{
    int NumFrames; //Total number of animation frames
    int AnimationFrame; //Which animation frame to render
    int TileIndex; //Which tile from the tile pallet
    int Blank; //Is this tile blank
    int Render; //Should this tile be rendered
};

RWTexture2D<float4>Result;
Texture2D<float4> TilePallet;
RWStructuredBuffer<Tile> Map;
int TileStride;
int TilePalletCount;
uint MapWidth;
uint MapHeight;
int TileWidth;
int TileHeight;

//Produces a tilemap texture using the provided tilemappallet
[numthreads(4,4,1)]
void TilemapRender(uint3 id : SV_DispatchThreadID)
{
    if (id.x < MapWidth && id.y < MapHeight)
    {
        Tile T = Map[id.y * MapWidth + id.x];
        if (T.Blank == 0 && T.Render == 1)
        {
            for (int x = 0; x < TileWidth; x++)
            {
                for (int y = 0; y < TileHeight; y++)
                {
                    //Calculate what pixel to render onto
                    uint2 TilePalletIndex =
                    uint2(T.AnimationFrame * TileWidth + x, (TilePalletCount - T.TileIndex - 1)
                    * TileStride + y);
                    uint2 RenderIndex = uint2(id.x *
                    TileWidth + x, (MapHeight - id.y - 1) * TileHeight + y);
                    Result[RenderIndex] =
                    TilePallet[TilePalletIndex];
                }
            }
        }
    }
}
```

---

## Client/Assets/Render/ResizeCanvas.cs

---

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;

static class CanvasExtensions
{
    public static Vector2 SizeToParent(this RawImage Image, float Padding = 0)
    {
        var Parent = Image.transform.parent.GetComponentInParent<RectTransform>();
        var Transform = Image.GetComponent<RectTransform>();
        if (!Parent) { return Transform.sizeDelta; } //if we don't have a parent, just return our current width
        Padding = 1 - Padding;
        float Ratio = Image.texture.width / (float)Image.texture.height;
        var Bounds = new Rect(0, 0, Parent.rect.width, Parent.rect.height);
        if (Mathf.RoundToInt(Transform.eulerAngles.z) % 180 == 90)
        {
            //Invert the bounds if the image is rotated
            Bounds.size = new Vector2(Bounds.height, Bounds.width);
        }
        //Size by height first
        float Height = Bounds.height * Padding;
        float Width = Height * Ratio;
        if (Width > Bounds.width * Padding)
        { //If it doesn't fit, fallback to width
            Width = Bounds.width * Padding;
            Height = Width / Ratio;
        }
        Transform.sizeDelta = new Vector2(Width, Height);
        return Transform.sizeDelta;
    }
}

public class ResizeCanvas : MonoBehaviour
{
    RawImage Render;

    // Start is called before the first frame update
    void Start()
    {
        Render = GetComponent<RawImage>();
    }

    // Update is called once per frame
    void Update()
    {
        if (Render != null) Render.SizeToParent();
    }
}
```

---

## *Assets/Scripts/Creatures/AbstractCreatures.cs*

---

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using UnityEngine;

using DeadZoneEngine;
using DeadZoneEngine.Entities;
using DeadZoneEngine.Entities.Components;

public abstract class AbstractCreature : AbstractWorldEntity, IUpdatable,
IRenderer
{
    public int SortingLayer { get; set; }

    public BodyChunk[] BodyChunks;
    public DistanceJoint[] BodyChunkConnections;

    public AbstractCreature() { }
    public AbstractCreature(ushort ID) : base(ID) { }

    public virtual void Update() { }
    public virtual void BodyPhysicsUpdate() { }
    protected override void OnDelete() { }

    public virtual void InitializeRenderer() { }
    public virtual void Render() { }
}
```

---

## *Client/Assets/Scripts/Creatures/BulletEntity.cs*

---

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

using UnityEngine;
using DeadZoneEngine.Entities;
using DeadZoneEngine.Entities.Components;
using DZNetwork;
using DeadZoneEngine;

public class BulletEntity : AbstractWorldEntity, IPhysicsUpdatable,
IRenderer, IServerSendable
{
    public int SortingLayer { get; set; }
    public int ServerObjectType { get; set; } =
(int)DZSettings.EntityType.BulletEntity;
    public bool RecentlyUpdated { get; set; } = false;
}
```

```

public bool ProtectedDeletion { get; set; } = false;

BodyChunk Bolt;
public float Speed = 100;
public Vector2 Direction = Vector2.up;

public BulletEntity(ushort ID) : base(ID)
{
    Init();
}

public BulletEntity() : base()
{
    Init();
}

public void Init()
{
    Bolt = new BodyChunk();
    Bolt.Collider.radius = 0.1f;
    Bolt.Kinematic = true;
}

public Vector2 Position
{
    get
    {
        if (Bolt != null)
            return Bolt.Position;
        return Vector2.zero;
    }
    set
    {
        if (Bolt != null)
            Bolt.Position = value;
    }
}

public void InitializeRender()
{
}

public void Render()
{
    Bolt.RenderObject.transform.localScale = new Vector2(0.2f, 0.2f);
    Bolt.RenderColor = Color.red;
}

public void ServerUpdate()
{
}

private RaycastHit2D[] RayCasts = new RaycastHit2D[6];
public void FixedUpdate()
{
    MoveBullet(Time.fixedDeltaTime);
}
private void MoveBullet(float Time)
{
}

```

```

        float ScaledSpeed = Speed * Time;
        Vector2 NormalDirection =
Vector2.Perpendicular(Direction).normalized * (Bolt.Collider.radius +
0.01f);
        RayCasts[0] = Physics2D.Raycast(Bolt.Position, Direction,
ScaledSpeed);
        RayCasts[1] = Physics2D.Raycast(Bolt.Position + NormalDirection,
Direction, ScaledSpeed);
        RayCasts[2] = Physics2D.Raycast(Bolt.Position - NormalDirection,
Direction, ScaledSpeed);
        Vector2 End = Bolt.Position + Direction * ScaledSpeed;
        RayCasts[3] = Physics2D.Raycast(End, -Direction, ScaledSpeed -
Bolt.Collider.radius - 0.1f);
        RayCasts[4] = Physics2D.Raycast(End + NormalDirection, -Direction,
ScaledSpeed);
        RayCasts[5] = Physics2D.Raycast(End - NormalDirection, -Direction,
ScaledSpeed);
        bool FoundHit = false;
        int Index = 0;
        for (int i = 0; i < RayCasts.Length; i++)
        {
            if (RayCasts[i].collider != null &&
Vector2.Dot(RayCasts[i].normal, Direction) < 0)
            {
                FoundHit = true;
                Index = i;
                break;
            }
        }
        if (FoundHit)
        {
            RaycastHit2D Hit = RayCasts[Index];
            float Distance = Mathf.Abs((Hit.point -
Bolt.Position).magnitude) - Bolt.Collider.radius;
            Vector2 NewPosition = Bolt.Position + Direction * Distance;
            AbstractWorld WorldContext =
Hit.collider.gameObject.GetComponent<AbstractWorld>();
            if (WorldContext != null)
            {
                if (WorldContext.Type ==
DZSettings.EntityType.PlayerCreature)
                {
                    PlayerCreature Player =
(PlayerCreature)WorldContext.Context;

                }
            }
            Direction = Vector2.Reflect(Direction, Hit.normal).normalized;
            Bolt.Position = NewPosition + Direction * 0.1f + Hit.normal *
0.1f;
        }
        else
        {
            Bolt.Position += Direction * ScaledSpeed;
        }
    }

    public void IsolateVelocity() { }

    public void RestoreVelocity() { }

```

```

protected override void OnDelete ()
{
    DZEngine.Destroy (Bolt);
}

public override byte [] GetBytes ()
{
    List<byte> Data = new List<byte> ();
    Data.AddRange (BitConverter.GetBytes (Speed));
    Data.AddRange (BitConverter.GetBytes (Direction.x));
    Data.AddRange (BitConverter.GetBytes (Direction.y));
    Data.AddRange (Bolt.GetBytes ());
    return Data.ToArray ();
}

public override void ParseBytes (Packet Data)
{
    ParseSnapshot (ParseBytesToSnapshot (Data));
}

public struct Data
{
    public float Speed;
    public Vector2 Direction;
    public BodyChunk.Data Bolt;
}

public override object GetSnapshot ()
{
    return new Data ()
    {
        Speed = Speed,
        Direction = Direction,
        Bolt = (BodyChunk.Data) Bolt.GetSnapshot ()
    };
}

public static object ParseBytesToSnapshot (DZNetwork.Packet Data)
{
    return new Data ()
    {
        Speed = Data.ReadFloat (),
        Direction = new Vector2 (Data.ReadFloat (), Data.ReadFloat ()),
        Bolt = (BodyChunk.Data) BodyChunk.ParseBytesToSnapshot (Data)
    };
}

public override void ParseSnapshot (object ObjectData)
{
    Data Data = (Data) ObjectData;
    Speed = Data.Speed;
    Direction = Data.Direction;
    Bolt.ParseSnapshot (Data.Bolt);
}

public override void Interpolate (object FromData, object ToData, float
Time)
{
    Data From = (Data) FromData;
    Data To = (Data) ToData;
    Direction = From.Direction;
}

```



```

        Speed = From.Speed;
        Bolt.Interpolate(From.Bolt, To.Bolt, Time);
    }

    public override void Extrapolate(object FromData, float Time)
    {
        Data From = (Data)FromData;
        Direction = From.Direction;
        Speed = From.Speed;
        MoveBullet(Time);
    }
}

```

---

### *Client/Assets/Scripts/Creatures/CoinEntity.cs*

---

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using UnityEngine;

using DeadZoneEngine;
using DeadZoneEngine.Entities;
using DeadZoneEngine.Entities.Components;
using DZNetwork;

public class CoinEntity : AbstractWorldEntity, IUpdatable, IRenderer,
IServerSendable
{
    public int SortingLayer { get; set; }
    public int ServerObjectType { get; set; } =
(int)DZSettings.EntityType.CoinEntity;
    public bool RecentlyUpdated { get; set; } = false;
    public bool ProtectedDeletion { get; set; } = false;

    public int Money = 1;
    public int Health = 0;
    public BodyChunk Coin;
    public float Decay = 10;

    public CoinEntity(ushort ID) : base(ID)
    {
        Init();
    }

    public CoinEntity() : base()
    {
        Init();
    }

    public void Init()
    {
        Coin = new BodyChunk();
        Coin.Context = this;
        Coin.ContextType = DZSettings.EntityType.CoinEntity;
    }
}

```

```

        Coin.Collider.radius = 0.01f;
        Coin.Velocity = new Vector2(UnityEngine.Random.Range(-5f, 5f),
UnityEngine.Random.Range(-5f, 5f));
    }

    public Vector2 Position
    {
        get
        {
            if (Coin != null)
                return Coin.Position;
            return Vector2.zero;
        }
        set
        {
            if (Coin != null)
                Coin.Position = value;
        }
    }

    public void InitializeRenderer()
    {
    }

    public void Render()
    {
        Coin.RenderObject.transform.localScale = new Vector2(0.2f, 0.2f);
        if (Health > 0)
            Coin.RenderColor = Color.red;
        else
            Coin.RenderColor = Color.magenta;
    }

    public void ServerUpdate()
    {
    }

    public void Update()
    {
        Decay -= Time.fixedDeltaTime;
        if (Decay < 0)
        {
            DZEngine.Destroy(this);
        }

        Collider2D[] C = Physics2D.OverlapCircleAll(Position, 1f);
        List<PlayerCreature> NearbyCreatures = new List<PlayerCreature>();
        for (int i = 0; i < C.Length; i++)
        {
            if (C[i] != null)
            {
                AbstractWorld AW = C[i].GetComponent<AbstractWorld>();
                if (AW != null && AW.Type ==
DZSettings.EntityType.PlayerCreature)
                {
                    NearbyCreatures.Add((PlayerCreature)AW.Context);
                }
            }
        }
    }

```

```

    for (int i = 0; i < NearbyCreatures.Count; i++)
    {
        Vector2 Dir = NearbyCreatures[i].Position - Coin.Position;
        if (Dir.magnitude < 0.3f)
        {
            Main.Money += Money;
            Main.GainLifeForce(Health);
            DZEngine.Destroy(this);
        }
        float Speed = 10;
        Coin.Velocity += Dir.normalized * Speed * Time.fixedDeltaTime;
    }
    Coin.Velocity *= 0.9f;
}

public void BodyPhysicsUpdate() { }

public void IsolateVelocity() { }

public void RestoreVelocity() { }

protected override void OnDestroy()
{
    DZEngine.Destroy(Coin);
}

public override byte[] GetBytes()
{
    List<byte> Data = new List<byte>();
    Data.AddRange(BitConverter.GetBytes(Money));
    Data.AddRange(BitConverter.GetBytes(Health));
    Data.AddRange(Coin.GetBytes());
    return Data.ToArray();
}

public override void ParseBytes(Packet Data)
{
    ParseSnapshot(ParseBytesToSnapshot(Data));
}

public struct Data
{
    public int Money;
    public int Health;
    public BodyChunk.Data Coin;
}

public override object GetSnapshot()
{
    return new Data()
    {
        Money = Money,
        Health = Health,
        Coin = (BodyChunk.Data)Coin.GetSnapshot()
    };
}

public static object ParseBytesToSnapshot(DZNetwork.Packet Data)
{
    return new Data()
    {

```

```

        Money = Data.ReadInt(),
        Health = Data.ReadInt(),
        Coin = (BodyChunk.Data)BodyChunk.ParseBytesToSnapshot(Data)
    };
}

public override void ParseSnapshot(object ObjectData)
{
    Data Data = (Data)ObjectData;
    Money = Data.Money;
    Health = Data.Health;
    Coin.ParseSnapshot(Data.Coin);
}

public override void Interpolate(object FromData, object ToData, float
Time)
{
    Data From = (Data)FromData;
    Data To = (Data)ToData;
    Money = From.Money;
    Health = From.Health;
    Coin.Interpolate(From.Coin, To.Coin, Time);
}
}

```

---

### *Client/Assets/Scripts/Creatures/EnemyCreature.cs*

---

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

using UnityEngine;
using DeadZoneEngine.Entities;
using DeadZoneEngine.Entities.Components;

public class EnemyCreature : AbstractCreature, IServerSendable
{
    public struct WayPoint
    {
        public int Direction;
        public Vector2Int Position;
    }

    public struct Path
    {
        public List<WayPoint> Traversal;
        public string Map;
    }

    public int ServerObjectType { get; set; } =
(int)DZSettings.EntityType.EnemyCreature;
    public bool RecentlyUpdated { get; set; } = false;
    public bool ProtectedDeletion { get; set; } = false;
}

```

```

public int CurrentWayPoint = 0;
public Path Traversal;

public int CorpseHP = 5;
public int Health = 5;
public float Speed = 1;
private float[] DynamicRunSpeed;
public BodyState State;

public enum BodyState
{
    Standing,
    Limp
}

public EnemyCreature(ushort ID) : base(ID)
{
    Initialize();
}
public EnemyCreature()
{
    Initialize();
}

public override void Render()
{
    BodyChunks[0].RenderObject.gameObject.transform.localScale = new
Vector2(0.5f, 0.5f);
    BodyChunks[1].RenderObject.gameObject.transform.localScale = new
Vector2(0.5f, 0.5f);
    if (State == BodyState.Limp) BodyColor = Color.grey;
    BodyChunks[0].RenderColor = BodyColor;
    BodyChunks[1].RenderColor = BodyColor;
}

Color BodyColor;
private void Initialize()
{
    BodyChunks = new BodyChunk[2];
    BodyChunks[0] = new BodyChunk(this);
    BodyChunks[1] = new BodyChunk(this);
    BodyChunks[0].Collider.radius = 0.25f;
    BodyChunks[1].Collider.radius = 0.25f;
    BodyChunks[0].Context = this;
    BodyChunks[0].ContextType = DZSettings.EntityType.EnemyCreature;
    BodyChunks[1].Context = this;
    BodyChunks[1].ContextType = DZSettings.EntityType.EnemyCreature;
    SetGravity(0f);

    BodyChunkConnections = new DistanceJoint[1];
    BodyChunkConnections[0] = new DistanceJoint();
    BodyChunkConnections[0].Set(new DistanceJointData(BodyChunks[0],
BodyChunks[1], 0.5f, Vector2.zero));
    BodyChunkConnections[0].Active = false;

    Physics2D.IgnoreCollision(BodyChunks[0].Collider,
BodyChunks[1].Collider, true); //Ignore collisions between body parts

    DynamicRunSpeed = new float[2];

    BodyColor = new Color(0, 1, 0);
}

```

```

        BodyChunks[0].RenderColor = BodyColor;
        BodyChunks[1].RenderColor = BodyColor;
    }

    public Vector2 Position
    {
        get
        {
            if (BodyChunks[0] != null)
                return BodyChunks[0].Position;
            return Vector2.zero;
        }
        set
        {
            if (BodyChunks[0] != null)
                BodyChunks[0].Position = value;
            if (BodyChunks[1] != null)
                BodyChunks[1].Position = value;
        }
    }

    public void ApplyVelocity(Vector2 Direction, float Force)
    {
        BodyChunks[0].Velocity += Direction * Force;
    }

    public void ApplyVelocity(Vector2 Vel)
    {
        BodyChunks[0].Velocity += Vel;
    }

    public void ServerUpdate ()
    {
    }

    public override void Update ()
    {
        UpdateBodyState ();
        UpdateMovement ();
    }

    private void UpdateBodyState ()
    {
    }

    private Vector2 MovementDirection;
    private void UpdateMovement ()
    {
        switch (State)
        {
            case BodyState.Limp:
            {
                BodyChunks[0].Velocity *= 0.3f;
                BodyChunks[1].Velocity *= 0.3f;
            }
            break;
            case BodyState.Standing:
            {
                DynamicRunSpeed[0] = 1f;
                DynamicRunSpeed[1] = 1.5f;
            }
        }
    }

```

```

        BodyChunks[0].Velocity += new Vector2(Speed *
DynamicRunSpeed[0] * MovementDirection.x, Speed * DynamicRunSpeed[0] *
MovementDirection.y);
        BodyChunks[1].Velocity += new Vector2(Speed *
DynamicRunSpeed[1] * MovementDirection.x, Speed * DynamicRunSpeed[1] *
MovementDirection.y);

        BodyChunks[0].Velocity *= 0.8f;
        BodyChunks[1].Velocity *= 0.8f;
    }
    break;
}
}

public override void BodyPhysicsUpdate ()
{
    switch (State)
    {
        case BodyState.Limp:
        {
            SetGravity(0f);
            BodyChunks[1].SpriteOffset =
Vector2.Lerp(BodyChunks[1].SpriteOffset, Vector2.zero, 4 *
Time.fixedDeltaTime);
            BodyChunkConnections[0].Active = true;
        }
        break;
        case BodyState.Standing:
        {
            SetGravity(0f);
            BodyChunks[1].SpriteOffset =
Vector2.Lerp(BodyChunks[1].SpriteOffset, new Vector2(0, 0.3f), 4 *
Time.fixedDeltaTime);
            BodyChunkConnections[0].Active = false;

            float Dist = Vector2.Distance(BodyChunks[0].Position,
BodyChunks[1].Position);
            Vector2 Dir = (BodyChunks[0].Position -
BodyChunks[1].Position).normalized;
            BodyChunks[1].Position += Dist * Dir * 0.8f;
            BodyChunks[1].Velocity += Dist * Dir * 0.8f;

            BodyChunks[0].Velocity *= 0.8f;
            BodyChunks[1].Velocity *= 0.8f;
        }
        break;
    }
}

public void SetGravity(float Gravity)
{
    BodyChunks[0].Gravity = Gravity;
    BodyChunks[1].Gravity = Gravity;
}

protected override void OnDelete ()
{
    BodyChunks[0].Delete ();
    BodyChunks[1].Delete ();
    BodyChunkConnections[0].Delete ();
}

```

```

}

public override byte[] GetBytes ()
{
    List<byte> Data = new List<byte> ();
    Data.AddRange (BitConverter.GetBytes ((int) State));
    Data.AddRange (BodyChunks [0].GetBytes ());
    Data.AddRange (BodyChunks [1].GetBytes ());
    Data.AddRange (BodyChunkConnections [0].GetBytes ());
    return Data.ToArray ();
}

public override void ParseBytes (DZNetwork.Packet Data)
{
    ParseSnapshot ((Data) ParseBytesToSnapshot (Data));
}

public struct Data
{
    public BodyState State;
    public BodyChunk.Data BodyChunk0;
    public BodyChunk.Data BodyChunk1;
    public DistanceJoint.Data BodyChunkConnections0;
}

public static object ParseBytesToSnapshot (DZNetwork.Packet Data)
{
    return new Data ()
    {
        State = (BodyState) Data.ReadInt (),
        BodyChunk0 =
        (BodyChunk.Data) BodyChunk.ParseBytesToSnapshot (Data),
        BodyChunk1 =
        (BodyChunk.Data) BodyChunk.ParseBytesToSnapshot (Data),
        BodyChunkConnections0 =
        (DistanceJoint.Data) DistanceJoint.ParseBytesToData (Data)
    };
}

public override void ParseSnapshot (object ObjectData)
{
    Data Data = (Data) ObjectData;
    State = Data.State;
    BodyChunks [0].ParseSnapshot (Data.BodyChunk0);
    BodyChunks [1].ParseSnapshot (Data.BodyChunk1);
    BodyChunkConnections [0].ParseSnapshot (Data.BodyChunkConnections0);
}

public override object GetSnapshot ()
{
    return new Data ()
    {
        State = State,
        BodyChunk0 = (BodyChunk.Data) BodyChunks [0].GetSnapshot (),
        BodyChunk1 = (BodyChunk.Data) BodyChunks [1].GetSnapshot (),
        BodyChunkConnections0 =
        (DistanceJoint.Data) BodyChunkConnections [0].GetSnapshot ()
    };
}

public override void Interpolate (object FromData, object ToData, float
Time)

```



```

    {
        Data From = (Data)FromData;
        Data To = (Data)ToData;
        State = From.State;
        BodyChunks[0].Interpolate(From.BodyChunk0, To.BodyChunk0, Time);
        BodyChunks[1].Interpolate(From.BodyChunk1, To.BodyChunk1, Time);
        BodyChunkConnections[0].Interpolate(From.BodyChunkConnections0,
To.BodyChunkConnections0, Time);
    }

    public override void Extrapolate(object FromData, float Time)
    {
        Data From = (Data)FromData;
        State = From.State;
        BodyChunks[0].Extrapolate(From.BodyChunk0, Time);
        BodyChunks[1].Extrapolate(From.BodyChunk1, Time);
        BodyChunkConnections[0].Extrapolate(From.BodyChunkConnections0,
Time);
    }
}

```

---

### *Client/Assets/Scripts/Creatures/PlayerCreature.cs*

---

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using UnityEngine;

using ClientHandle;
using DeadZoneEngine;
using DeadZoneEngine.Entities;
using DeadZoneEngine.Entities.Components;

public class PlayerCreature : AbstractCreature, IServerSendable
{
    public int ServerObjectType { get; set; } =
(int)DZSettings.EntityType.PlayerCreature;
    public bool RecentlyUpdated { get; set; } = false;
    public bool ProtectedDeletion { get; set; } = false;

    public bool Out;
    public float RunSpeed;

    public class Control
    {
        public PlayerController Owner;
        public Vector2 MovementDirection;
        public Vector2 ShieldVector;
        public float Interact;

        public struct Snapshot
        {
            public ulong InputID;
            public Vector2 MovementDirection;
        }
    }
}

```

```

public ulong InputID;

public Snapshot GetSnapshot ()
{
    return new Snapshot ()
    {
        InputID = InputID++,
        MovementDirection = MovementDirection
    };
}

public void ParseSnapshot (Snapshot Snapshot)
{
    MovementDirection = Snapshot.MovementDirection;
}
}

public Control Controller { get; private set; } //Controller for player
movement

private float[] DynamicRunSpeed; //Controls Speed of each bodychunk

public PlayerCreature (ushort ID) : base (ID)
{
    Initialize ();
}

public PlayerCreature ()
{
    Initialize ();
}

public override void Render ()
{
    BodyChunks [0].RenderObject.gameObject.transform.localScale = new
Vector2 (0.5f, 0.5f);
    BodyChunks [1].RenderObject.gameObject.transform.localScale = new
Vector2 (0.5f, 0.5f);
}

Color BodyColor;
private void Initialize ()
{
    if (DZSettings.ClientSidePrediction)
        Histogram = new DZNetwork.JitterBuffer <PlayerSnapshot> ();

    Controller = new Control ();
    RunSpeed = 2f;

    BodyChunks = new BodyChunk [2];
    BodyChunks [0] = new BodyChunk (this);
    BodyChunks [1] = new BodyChunk (this);
    BodyChunks [0].Collider.radius = 0.25f;
    BodyChunks [1].Collider.radius = 0.25f;
    BodyChunks [0].Context = this;
    BodyChunks [0].ContextType = DZSettings.EntityType.PlayerCreature;
    BodyChunks [1].Context = this;
    BodyChunks [1].ContextType = DZSettings.EntityType.PlayerCreature;
    SetGravity (0f);

    BodyChunkConnections = new DistanceJoint [1];
    BodyChunkConnections [0] = new DistanceJoint ();
}

```

```

        BodyChunkConnections[0].Set(new DistanceJointData(BodyChunks[0],
BodyChunks[1], 0.5f, Vector2.zero));
        BodyChunkConnections[0].Active = false;

        Physics2D.IgnoreCollision(BodyChunks[0].Collider,
BodyChunks[1].Collider, true); //Ignore collisions between body parts

        DynamicRunSpeed = new float[2];

        BodyColor = new Color(UnityEngine.Random.Range(0f, 1f),
UnityEngine.Random.Range(0f, 1f), UnityEngine.Random.Range(0f, 1f));
        BodyChunks[0].RenderColor = BodyColor;
        BodyChunks[1].RenderColor = BodyColor;
    }

    public Vector2 Position
    {
        get
        {
            if (BodyChunks[0] != null)
                return BodyChunks[0].Position;
            return Vector2.zero;
        }
        set
        {
            if (BodyChunks[0] != null)
                BodyChunks[0].Position = value;
            if (BodyChunks[1] != null)
                BodyChunks[1].Position = value;
        }
    }

    public void ServerUpdate()
    {
        if (Controller.Owner == null || DZSettings.ClientSidePrediction ==
false) return;

        UpdateReconcilliation();
        LerpReconcilleError();

        BodyChunks[0].PhysicallyActive = true;
        BodyChunks[1].PhysicallyActive = true;
        BodyChunkConnections[0].PhysicallyActive = true;
        BodyChunks[0].Kinematic = false;
        BodyChunks[1].Kinematic = false;
    }

    public override void Update()
    {
        UpdateBodyState();
        UpdateMovement();
    }

    private void UpdateBodyState()
    {
    }

    private void UpdateMovement()
    {
        DynamicRunSpeed[0] = 1f;
    }

```

```

        DynamicRunSpeed[1] = 2f;
        if (Controller != null)
        {
            BodyChunks[0].Velocity += new Vector2 (RunSpeed *
DynamicRunSpeed[0] * Controller.MovementDirection.x, RunSpeed *
DynamicRunSpeed[0] * Controller.MovementDirection.y);
            BodyChunks[1].Velocity += new Vector2 (RunSpeed *
DynamicRunSpeed[1] * Controller.MovementDirection.x, RunSpeed *
DynamicRunSpeed[1] * Controller.MovementDirection.y);
        }

        BodyChunks[0].Velocity *= 0.8f;
        BodyChunks[1].Velocity *= 0.8f;
    }

    public override void BodyPhysicsUpdate ()
    {
        BodyChunks[0].Kinematic = false;
        BodyChunks[1].Kinematic = false;

        SetGravity(0f);
        BodyChunks[1].SpriteOffset =
Vector2.Lerp (BodyChunks[1].SpriteOffset, new Vector2 (0, 0.3f), 4 *
Time.fixedDeltaTime);
        BodyChunkConnections[0].Active = false;

        float Dist = Vector2.Distance (BodyChunks[0].Position,
BodyChunks[1].Position);
        Vector2 Dir = (BodyChunks[0].Position -
BodyChunks[1].Position).normalized;
        BodyChunks[1].Position += Dist * Dir * 0.8f;
        BodyChunks[1].Velocity += Dist * Dir * 0.8f;

        BodyChunks[0].Velocity *= 0.8f;
        BodyChunks[1].Velocity *= 0.8f;
    }

    public void SetGravity(float Gravity)
    {
        BodyChunks[0].Gravity = Gravity;
        BodyChunks[1].Gravity = Gravity;
    }

    protected override void OnDestroy ()
    {
        BodyChunks[0].Delete ();
        BodyChunks[1].Delete ();
        BodyChunkConnections[0].Delete ();
    }

    public override byte[] GetBytes ()
    {
        List<byte> Data = new List<byte> ();
        Data.AddRange (BitConverter.GetBytes (Controller.InputID));
        Data.AddRange (BodyChunks[0].GetBytes ());
        Data.AddRange (BodyChunks[1].GetBytes ());
        Data.AddRange (BodyChunkConnections[0].GetBytes ());
        return Data.ToArray ();
    }

    public override void ParseBytes (DZNetwork.Packet Data)

```

```

{
    ParseSnapshot ((Data) ParseBytesToSnapshot (Data));
}

public struct Data
{
    public ulong InputID;
    public BodyChunk.Data BodyChunk0;
    public BodyChunk.Data BodyChunk1;
    public DistanceJoint.Data BodyChunkConnections0;
}

public static object ParseBytesToSnapshot (DZNetwork.Packet Data)
{
    return new Data ()
    {
        InputID = Data.ReadULong (),
        BodyChunk0 =
        (BodyChunk.Data) BodyChunk.ParseBytesToSnapshot (Data),
        BodyChunk1 =
        (BodyChunk.Data) BodyChunk.ParseBytesToSnapshot (Data),
        BodyChunkConnections0 =
        (DistanceJoint.Data) DistanceJoint.ParseBytesToData (Data)
    };
}

public override void ParseSnapshot (object ObjectData)
{
    if (Controller.Owner != null && DZSettings.ClientSidePrediction &&
!Reconcille)
        return;
    Data Data = (Data) ObjectData;
    BodyChunks [0].ParseSnapshot (Data.BodyChunk0);
    BodyChunks [1].ParseSnapshot (Data.BodyChunk1);
    BodyChunkConnections [0].ParseSnapshot (Data.BodyChunkConnections0);
}

public override object GetSnapshot ()
{
    return new Data ()
    {
        InputID = Controller.InputID,
        BodyChunk0 = (BodyChunk.Data) BodyChunks [0].GetSnapshot (),
        BodyChunk1 = (BodyChunk.Data) BodyChunks [1].GetSnapshot (),
        BodyChunkConnections0 =
        (DistanceJoint.Data) BodyChunkConnections [0].GetSnapshot ()
    };
}

public class PlayerSnapshot
{
    public Data Snapshot;
    public Control.Snapshot Controls;
}

public DZNetwork.JitterBuffer<PlayerSnapshot> Histogram = null;
private void UpdateReconcilliation ()
{
    Histogram.Add (new PlayerSnapshot ()
    {
        Snapshot = (Data) GetSnapshot (),
        Controls = Controller.GetSnapshot ()
    });
}

```

```

}

private void LerpReconcilleError ()
{
    if (!Loader.Socket.Connected) return;
    const float Amount = 4f;
    float Error = (ReconcilledSelf.BodyChunk0.Position -
BodyChunks[0].Position).SqrMagnitude ();
    if (Error < 1)
    {
        BodyChunks[0].Position = Vector3.Lerp(BodyChunks[0].Position,
ReconcilledSelf.BodyChunk0.Position, Amount * Time.fixedDeltaTime);
        BodyChunks[1].Position = Vector3.Lerp(BodyChunks[1].Position,
ReconcilledSelf.BodyChunk1.Position, Amount * Time.fixedDeltaTime);
    }
    else
    {
        BodyChunks[0].Position = ReconcilledSelf.BodyChunk0.Position;
        BodyChunks[1].Position = ReconcilledSelf.BodyChunk1.Position;
    }
}

private PlayerSnapshot Current = null;
public Data CurrentSelf;
private bool ValidPredictPass;
private bool FinishedPredict;
public void StartClientPrediction (Game.ServerSnapshot FromData)
{
    ValidPredictPass = FromData.Data.ContainsKey (ID);
    CurrentSelf = (Data) GetSnapshot ();
    Reconcille = true;
    if (!ValidPredictPass) return;
    Data ClientPredictBaseline = (Data) FromData.Data [ID].Data;
    if (LastReconcilled >= ClientPredictBaseline.InputID)
    {
        ValidPredictPass = false;
        return;
    }
    LastReconcilled = ClientPredictBaseline.InputID;
    Histogram.Iterate (S =>
    {
        if (S.Value.Controls.InputID >= ClientPredictBaseline.InputID)
        {
            Current = S.Value;
        }
    }, S => S.Value.Controls.InputID >= ClientPredictBaseline.InputID);
    if (Current != null)
    {
        Histogram.Dequeue (Current);
        ParseSnapshot (ClientPredictBaseline);
        FinishedPredict = false;
        CurrentKey = Histogram.FirstKey;
    }
    else
    {
        Histogram.Clear ();
        ValidPredictPass = false;
    }
}

private DZNetwork.JitterBuffer<PlayerSnapshot>.Key CurrentKey;
public void ClientPrediction ()

```

```

{
    if (!ValidPredictPass) return;
    if (CurrentKey != null)
    {
        Controller.ParseSnapshot (CurrentKey.Value.Controls);
        if (!FinishedPredict && CurrentKey.Next == null)
        {
            FinishedPredict = true;
            ReconcilledSelf = (Data)GetSnapshot ();
        }
        CurrentKey = CurrentKey.Next;
    }
}
public void EndClientPrediction ()
{
    ParseSnapshot (CurrentSelf);
    Reconcille = false;
}

private Data ReconcilledSelf;
private bool Reconcille = false;
private ulong LastReconcilled = 0;
public override void Interpolate (object FromData, object ToData, float
Time)
{
    if (Controller.Owner != null && DZSettings.ClientSidePrediction)
        return;

    Data From = (Data)FromData;
    Data To = (Data)ToData;
    BodyChunks [0].Interpolate (From.BodyChunk0, To.BodyChunk0, Time);
    BodyChunks [1].Interpolate (From.BodyChunk1, To.BodyChunk1, Time);
    BodyChunkConnections [0].Interpolate (From.BodyChunkConnections0,
To.BodyChunkConnections0, Time);
}

public override void Extrapolate (object FromData, float Time)
{
    if (Controller.Owner != null && DZSettings.ClientSidePrediction)
        return;

    Data From = (Data)FromData;
    BodyChunks [0].Extrapolate (From.BodyChunk0, Time);
    BodyChunks [1].Extrapolate (From.BodyChunk1, Time);
    BodyChunkConnections [0].Extrapolate (From.BodyChunkConnections0,
Time);
}
}

```

---

## Client/Assets/Scripts/Creatures/Turret.cs

---

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

using UnityEngine;
using DeadZoneEngine.Entities;
using DeadZoneEngine.Entities.Components;

public class Turret : AbstractCreature, IServerSendable
{
    public int ServerObjectType { get; set; } =
(int)DZSettings.EntityType.Turret;
    public bool RecentlyUpdated { get; set; } = false;
    public bool ProtectedDeletion { get; set; } = false;

    public int Timer;
    public int FireRate;

    public Turret(ushort ID) : base(ID)
    {
        Initialize();
    }
    public Turret()
    {
        Initialize();
    }

    public override void Render()
    {
        BodyChunks[0].RenderObject.gameObject.transform.localScale = new
Vector2(0.7f, 0.7f);
    }

    Color BodyColor;
    private void Initialize()
    {
        BodyChunks = new BodyChunk[1];
        BodyChunks[0] = new BodyChunk(this);
        BodyChunks[0].Collider.radius = 0.35f;
        BodyChunks[0].Kinematic = true;
        BodyChunks[0].Context = this;
        BodyChunks[0].ContextType = DZSettings.EntityType.Turret;
        SetGravity(0f);

        BodyColor = new Color(0.56f, 0.56f, 0.56f);
        BodyChunks[0].RenderColor = BodyColor;
    }

    public void SetGravity(float Gravity)
    {
        BodyChunks[0].Gravity = Gravity;
    }

    public Vector2 Position
```



```

    {
        get
        {
            if (BodyChunks[0] != null)
                return BodyChunks[0].Position;
            return Vector2.zero;
        }
        set
        {
            if (BodyChunks[0] != null)
                BodyChunks[0].Position = value;
            if (BodyChunks[1] != null)
                BodyChunks[1].Position = value;
        }
    }

    public void ServerUpdate ()
    {
    }

    public override void Update ()
    {
    }

    private void UpdateBodyState ()
    {
    }

    protected override void OnDelete ()
    {
        BodyChunks[0].Delete ();
    }

    public override byte[] GetBytes ()
    {
        List<byte> Data = new List<byte> ();
        Data.AddRange (BodyChunks[0].GetBytes ());
        return Data.ToArray ();
    }

    public override void ParseBytes (DZNetwork.Packet Data)
    {
        ParseSnapshot ((Data) ParseBytesToSnapshot (Data));
    }

    public struct Data
    {
        public BodyChunk.Data BodyChunk0;
    }

    public static object ParseBytesToSnapshot (DZNetwork.Packet Data)
    {
        return new Data ()
        {
            BodyChunk0 =
            (BodyChunk.Data) BodyChunk.ParseBytesToSnapshot (Data)
        };
    }

    public override void ParseSnapshot (object ObjectData)
    {
        Data Data = (Data) ObjectData;
    }

```

```

        BodyChunks[0].ParseSnapshot(Data.BodyChunk0);
    }

    public override object GetSnapshot()
    {
        return new Data()
        {
            BodyChunk0 = (BodyChunk.Data)BodyChunks[0].GetSnapshot();
        };
    }

    public override void Interpolate(object FromData, object ToData, float
Time)
    {
        Data From = (Data)FromData;
        Data To = (Data)ToData;
        BodyChunks[0].Interpolate(From.BodyChunk0, To.BodyChunk0, Time);
    }

    public override void Extrapolate(object FromData, float Time)
    {
        Data From = (Data)FromData;
        BodyChunks[0].Extrapolate(From.BodyChunk0, Time);
    }
}

```

---

### *Assets/DZEngine/Controllers/InputMapping.cs*

---

```

using System;
using System.Collections.ObjectModel;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

using UnityEngine.InputSystem;
using UnityEngine.InputSystem.LowLevel;
using UnityEngine.InputSystem.Controls;
using UnityEngine;
using static DeadZoneEngine.Controllers.InputMapping;
using DZNetwork;
using ClientHandle;

namespace DeadZoneEngine.Controllers
{
    public enum ControllerType
    {
        PlayerController
    }

    public static class InputMapping
    {
        public static Action<InputDevice> OnDeviceAdd;
        public static Action<InputDevice> OnDeviceReconnect;
        public static Action<InputDevice> OnDeviceRemove;
        public static Action<InputDevice> OnDeviceDisconnect;
    }
}

```

```

public class DeviceController
{
    public bool Enabled { get; private set; } = false;
    public List<Controller> Controllers = new List<Controller>();

    public byte[] GetBytes ()
    {
        List<byte> Data = new List<byte> ();
        Data.AddRange (BitConverter.GetBytes (Controllers.Count));
        for (int i = 0; i < Controllers.Count; i++)
        {
            Data.AddRange (BitConverter.GetBytes ((int) Controllers[i].Type));
            Data.AddRange (Controllers[i].GetBytes ());
        }
        return Data.ToArray ();
    }

    public void Tick ()
    {
        for (int i = 0; i < Controllers.Count; i++)
        {
            Controllers[i].Tick ();
        }
    }

    public void OnInput (ButtonControl Control)
    {
        for (int i = 0; i < Controllers.Count; i++)
            Controllers[i].OnInput (Control);
    }

    public void Enable ()
    {
        Enabled = true;
        for (int i = 0; i < Controllers.Count; i++)
            Controllers[i].Enable ();
    }

    public void Disable ()
    {
        Enabled = false;
        for (int i = 0; i < Controllers.Count; i++)
            Controllers[i].Disable ();
    }
}

private class DeviceComparer : IEqualityComparer<InputDevice>
{
    public bool Equals (InputDevice A, InputDevice B)
    {
        return A.Equals (B);
    }

    public int GetHashCode (InputDevice A)
    {
        return A.GetHashCode ();
    }
}

```

```

    public static Dictionary<InputDevice, DeviceController> Devices =
new Dictionary<InputDevice, DeviceController>(new DeviceComparer());

    public static void Initialize()
    {
        //Detects any key press https://forum.unity.com/threads/check-
if-any-key-is-pressed.763751/
        InputSystem.onEvent += (Event, Device) =>
        {
            if (!Devices.ContainsKey(Device)) return;
            if (!Event.IsA<StateEvent>() &&
!Event.IsA<DeltaStateEvent>()) return;
            var Controls = Device.allControls;
            float ButtonPressPoint =
InputSystem.settings.defaultButtonPressPoint;
            for (var i = 0; i < Controls.Count; ++i)
            {
                ButtonControl Control = Controls[i] as ButtonControl;
                if (Control == null || Control.synthetic ||
Control.noisy)
                    continue;
                if (Control.ReadValueFromEvent(Event, out var Value) &&
Value >= ButtonPressPoint)
                {
                    Devices[Device].OnInput(Control);
                    break;
                }
            }
        };
        InputSystem.onDeviceChange += OnDeviceChange;
        for (int i = 0; i < InputSystem.devices.Count; i++)
        {
            if (!(InputSystem.devices[i] is Mouse))
            {
                DeviceController DC = new DeviceController();
                Devices.Add(InputSystem.devices[i], DC);
                OnDeviceAdd?.Invoke(InputSystem.devices[i]);
            }
        }
    }

    public static void Tick()
    {
        List<DeviceController> Controllers = Devices.Values.ToList();
        foreach (DeviceController DC in Controllers)
        {
            DC.Tick();
        }
    }

    public static void ParseBytes(Packet Packet, Client Client)
    {
        int NumControllers = Packet.ReadInt();
        for (int i = 0; i < NumControllers; i++)
        {
            bool Enabled = Packet.ReadBool();
            if (!Enabled) continue;
            int NumControls = Packet.ReadInt();
            for (int j = 0; j < NumControls; j++)
            {

```



```

        Devices.Add(Device, new DeviceController());
        OnDeviceReconnect?.Invoke(Device);
        Devices[Device].Enable();
        break;
    }
}

public static void Rebind(InputAction Action, InputDevice Device)
{
}

}

public abstract class Controller
{
    public DeviceController DC;
    public ControllerType Type;
    private InputDevice Device;
    protected bool IsKeyboard { get; private set; } = true;
    protected InputActionMap ActionMap = new
InputActionMap("Controller");

    public Controller()
    {
        SetType();
    }

    public Controller(InputDevice Device, DeviceController DC)
    {
        this.DC = DC;
        this.Device = Device;
        IsKeyboard = Device is Keyboard;
        SetType();
    }

    protected abstract void SetType();

    //Triggered when any button on the device is pressed
    public virtual void OnInput(ButtonControl Control)
    {
    }

    public void Enable()
    {
        ActionMap.Enable();
    }

    public void Disable()
    {
        ActionMap.Disable();
    }

    public void RebindAll()
    {
        for (int i = 0; i < ActionMap.actions.Count; i++)
        {
            InputMapping.Rebind(ActionMap.actions[i], Device);
        }
    }

    public void Reset()

```

```

    {
        ActionMap.RemoveAllBindingOverrides();
    }

    public void Reset(string ActionName)
    {
        ActionMap.FindAction(ActionName).RemoveAllBindingOverrides();
    }

    public void Reset(int Index)
    {
        ActionMap.actions[Index].RemoveAllBindingOverrides();
    }

    public void Rebind(string ActionName)
    {
        InputMapping.Rebind(ActionMap.FindAction(ActionName), Device);
    }

    public void Rebind(int Index)
    {
        InputMapping.Rebind(ActionMap.actions[Index], Device);
    }

    public void Rebind(InputAction Action)
    {
        InputMapping.Rebind(Action, Device);
    }

    public virtual void Tick()
    {
    }

    public abstract void ParseBytes(Packet Data);
    public abstract byte[] GetBytes();
}
}

```

---

### *Assets/DZEngine/Entities/Components/BodyChunk.cs*

---

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using UnityEngine;

namespace DeadZoneEngine.Entities.Components
{
    public class BodyChunk : PhysicalObject, IRenderer<SpriteRenderer>
    {
        private AbstractWorld Info;
        public object Context
        {
            get

```

```

    {
        if (Info != null)
            return Info.Context;
        else
            return null;
    }
    set
    {
        Info.Context = value;
    }
}
public DZSettings.EntityType ContextType
{
    get
    {
        if (Info != null)
            return Info.Type;
        else
            return DZSettings.EntityType.Null;
    }
    set
    {
        Info.Type = value;
    }
}

public int SortingLayer { get; set; }
public SpriteRenderer RenderObject { get; set; }
private GameObject RenderObj;
public Vector3 SpriteOffset = Vector3.zero;
public Color RenderColor = new Color(1, 1, 1);
public virtual void InitializeRenderer()
{
    RenderObj = new GameObject();
    RenderObj.transform.parent = Self.transform;
    RenderObj.transform.localPosition = Vector3.zero;
    RenderObject = RenderObj.AddComponent<SpriteRenderer>();
    RenderObject.sprite = Resources.Load<Sprite>("Sprites/Circle");
}
public virtual void Render()
{
    RenderObj.transform.localPosition = SpriteOffset;
    RenderObject.color = RenderColor;
}

public CircleCollider2D Collider { get; private set; }
public ContactPoint2D[] Contacts = new ContactPoint2D[10];
public float Height;

public BodyChunk()
{
    Init();
}
public BodyChunk(AbstractWorldEntity Parent)
{
    this.Parent = Parent;
    Init();
}

public BodyChunk(ushort ID) : base(ID)
{

```



```

        Init();
    }

    private void Init()
    {
        Info = Self.AddComponent<AbstractWorld>();
        Info.Self = this;
        Collider = Self.AddComponent<CircleCollider2D>();
        Collider.radius = 0.5f;

        InvInertia = 1;
        InvMass = 1;
        Gravity = 0;
    }

    /// <summary>
    /// Update Contacts Array => If no contacts are found, the array
will not be updated (past values will persist)
    /// </summary>
    /// <returns>Number of contacts recieved</returns>
    public int GetContacts()
    {
        return Collider.GetContacts(Contacts);
    }

    public override byte[] GetBytes()
    {
        List<byte> Data = new List<byte>();
        Data.AddRange(BitConverter.GetBytes(Position.x));
        Data.AddRange(BitConverter.GetBytes(Position.y));
        Data.AddRange(BitConverter.GetBytes(Rotation));
        Data.AddRange(BitConverter.GetBytes(Velocity.x));
        Data.AddRange(BitConverter.GetBytes(Velocity.y));
        Data.AddRange(BitConverter.GetBytes(AngularVelocity));
        Data.AddRange(BitConverter.GetBytes(InvMass));
        Data.AddRange(BitConverter.GetBytes(InvInertia));
        Data.AddRange(BitConverter.GetBytes(Collider.radius));
        return Data.ToArray();
    }

    public override void ParseBytes(DZNetwork.Packet Data)
    {
        Data D = (Data)ParseBytesToSnapshot(Data);
        ParseSnapshot(D);
    }

    public struct Data
    {
        public Vector2 Position;
        public float Rotation;
        public Vector2 Velocity;
        public float AngularVelocity;
        public float InvMass;
        public float InvInertia;
        public float ColliderRadius;
    }

    public override object GetSnapshot()
    {
        Data D = new Data()
        {

```

```

        Position = new Vector2 (Position.x, Position.y),
        Rotation = Rotation,
        Velocity = new Vector2 (Velocity.x, Velocity.y),
        AngularVelocity = AngularVelocity,
        InvMass = InvMass,
        InvInertia = InvInertia,
        ColliderRadius = Collider.radius
    };
    return D;
}

public static object ParseBytesToSnapshot (DZNetwork.Packet Data)
{
    return new Data ()
    {
        Position = new Vector2 (Data.ReadFloat (), Data.ReadFloat ()),
        Rotation = Data.ReadFloat (),
        Velocity = new Vector2 (Data.ReadFloat (), Data.ReadFloat ()),
        AngularVelocity = Data.ReadFloat (),
        InvMass = Data.ReadFloat (),
        InvInertia = Data.ReadFloat (),
        ColliderRadius = Data.ReadFloat ()
    };
}

public override void ParseSnapshot (object ObjectData)
{
    Data Data = (Data)ObjectData;
    Position = Data.Position;
    Rotation = Data.Rotation;
    Velocity = Data.Velocity;
    AngularVelocity = Data.AngularVelocity;
    InvMass = Data.InvMass;
    InvInertia = Data.InvInertia;
    Collider.radius = Data.ColliderRadius;
}

public override void Interpolate (object FromData, object ToData,
float Time)
{
    Data From = (Data)FromData;
    Data To = (Data)ToData;
    Position = From.Position + (To.Position - From.Position) *
Time;
    Rotation = From.Rotation + (To.Rotation - From.Rotation) *
Time;
    Velocity = From.Velocity + (To.Velocity - From.Velocity) *
Time;
    AngularVelocity = From.AngularVelocity + (To.AngularVelocity -
From.AngularVelocity) * Time;
}

public override void Extrapolate (object FromData, float Time)
{
    Data From = (Data)FromData;
    Position = From.Position + From.Velocity * Time;
    Rotation = From.Rotation + From.AngularVelocity * Time;
}
}
}

```

---

## Assets/DZEngine/Entities/Components/DistanceJoint.cs

---

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

using UnityEngine;

namespace DeadZoneEngine.Entities.Components
{
    public struct DistanceJointData
    {
        public PhysicalObject A;
        public PhysicalObject B;
        public float Distance;
        public Vector2 Anchor;

        public DistanceJointData(PhysicalObject A, PhysicalObject B, float
Distance, Vector2 Anchor)
        {
            this.A = A;
            this.B = B;
            this.Distance = Distance;
            this.Anchor = Anchor;
        }
    }

    public class DistanceJoint : PhysicalJoint
    {
        PhysicalObject A;
        PhysicalObject B;

        Mat22 M; //Rotation Matrix
        Vector2 LocalAnchorA; //Anchor Points
        Vector2 LocalAnchorB;
        Vector2 RA; //Relative Anchor position on Body A
        Vector2 RB; //Relative Anchor position on Body B
        Vector2 Bias; //Bias in Impulse equation
        Vector2 AccumulatedImpulse; //stores accumulated Impulse

        public float Relaxation = 1f;

        //Strength of pull on object A and B
        public float ARatio = 1;
        public float BRatio = 1;

        float Distance;
        Vector2 Anchor;

        public DistanceJoint()
        {
        }

        public DistanceJoint(ushort ID) : base(ID)
        {
        }
    }
}
```

```

    }

    public void Set(object Data)
    {
        DistanceJointData DistanceJointWrapper =
(DistanceJointData)Data;
        Distance = DistanceJointWrapper.Distance;
        Anchor = DistanceJointWrapper.Anchor;
        A = DistanceJointWrapper.A;
        B = DistanceJointWrapper.B;

        //Compute Anchor information (rotation matrices)
        Mat22 RotA = new Mat22(0);
        Mat22 RotB = new Mat22(0);
        Mat22 RotAT = RotA.Transpose();
        Mat22 RotBT = RotB.Transpose();

        LocalAnchorA = RotAT * (Anchor);
        LocalAnchorB = RotBT * (Anchor - new Vector2(Distance, 0));

        Relaxation = 1.0f;
    }

    public void SetDistance(float Distance)
    {
        this.Distance = Distance;
        Mat22 RotA = new Mat22(0);
        Mat22 RotB = new Mat22(0);
        Mat22 RotAT = RotA.Transpose();
        Mat22 RotBT = RotB.Transpose();

        LocalAnchorA = RotAT * (Anchor);
        LocalAnchorB = RotBT * (Anchor - new Vector2(Distance, 0));
    }

    public override void PreUpdate()
    {
        //Pre-compute anchors, mass matrix, and bias =>
http://twvideo01.ubm-us.net/ol/vault/gdc09/slides/04-GDC09\_Catto\_Erin\_Solver.pdf
        if (A.Position == B.Position)
            A.Position += new Vector2(0.01f, 0);

        //Same as using atan2(A.Position - B.Position) however faster
        as skips atan2 math => this is just getting the current angle between the
        two objects A and B
        Mat22 RotA = new Mat22((A.Position - B.Position).normalized);
        Mat22 RotB = new Mat22((B.Position - A.Position).normalized);
        /*Mat22 RotA = new Mat22(A.Rotation); // This is for conserving
rotation of connected blocks
        Mat22 RotB = new Mat22(B.Rotation);*/

        float AInvMass = A.InvMass * ARatio;
        float BInvMass = B.InvMass * BRatio;
        float AInvInertia = A.InvInertia * ARatio;
        float BInvInertia = B.InvInertia * BRatio;

        RA = RotA * LocalAnchorA;
        RB = RotB * LocalAnchorB;

        Mat22 K1;

```

```

        K1.Col1.x = AInvMass + BInvMass; K1.Col2.x = 0.0f;
        K1.Col1.y = 0.0f; K1.Col2.y = AInvMass + BInvMass;

        Mat22 K2;
        K2.Col1.x = AInvInertia * RA.y * RA.y; K2.Col2.x = -AInvInertia
* RA.x * RA.y;
        K2.Col1.y = -AInvInertia * RA.x * RA.y; K2.Col2.y = AInvInertia
* RA.x * RA.x;

        Mat22 K3;
        K3.Col1.x = BInvInertia * RB.y * RB.y; K3.Col2.x = -BInvInertia
* RB.x * RB.y;
        K3.Col1.y = -BInvInertia * RB.x * RB.y; K3.Col2.y = BInvInertia
* RB.x * RB.x;

        Mat22 K = K1 + K2 + K3;
        M = K.Invert();

        Vector2 p1 = A.Position + RA;
        Vector2 p2 = B.Position + RB;
        Vector2 dp = p2 - p1;
        Bias = -0.1f * DZEngine.InvDeltaTime * dp;

        //Apply accumulated impulse
        AccumulatedImpulse *= Relaxation;

        A.Velocity -= AInvMass * AccumulatedImpulse;
        A.AngularVelocity -= AInvInertia * Math2D.Cross(RA,
AccumulatedImpulse);

        B.Velocity += BInvMass * AccumulatedImpulse;
        B.AngularVelocity += BInvInertia * Math2D.Cross(RB,
AccumulatedImpulse);
    }

    public override void IteratedUpdate()
    {
        Vector2 RelativeDeltaVelocity = B.Velocity +
Math2D.Cross(B.AngularVelocity, RB) - A.Velocity -
Math2D.Cross(A.AngularVelocity, RA);
        Vector2 Impulse = M * (-RelativeDeltaVelocity + Bias);

        A.Velocity -= A.InvMass * ARatio * Impulse;
        A.AngularVelocity -= A.InvInertia * ARatio * Math2D.Cross(RA,
Impulse);

        B.Velocity += B.InvMass * BRatio * Impulse;
        B.AngularVelocity += B.InvInertia * BRatio * Math2D.Cross(RB,
Impulse);

        AccumulatedImpulse += Impulse;
    }

    public override byte[] GetBytes()
    {
        List<byte> Data = new List<byte>();
        Data.AddRange(BitConverter.GetBytes(Distance));
        Data.AddRange(BitConverter.GetBytes(Anchor.x));
        Data.AddRange(BitConverter.GetBytes(Anchor.y));
        Data.AddRange(BitConverter.GetBytes(ARatio));
        Data.AddRange(BitConverter.GetBytes(BRatio));
    }

```

```

        return Data.ToArray();
    }

    public override void ParseBytes(DZNetwork.Packet Data)
    {
        Data D = (Data)ParseBytesToData(Data);
        ParseSnapshot(D);
    }

    public struct Data
    {
        public float Distance;
        public Vector2 Anchor;
        public float ARatio;
        public float BRatio;
    }

    public override object GetSnapshot()
    {
        return new Data()
        {
            Distance = Distance,
            Anchor = Anchor,
            ARatio = ARatio,
            BRatio = BRatio
        };
    }

    public static object ParseBytesToData(DZNetwork.Packet Data)
    {
        return new Data()
        {
            Distance = Data.ReadFloat(),
            Anchor = new Vector2(Data.ReadFloat(), Data.ReadFloat()),
            ARatio = Data.ReadFloat(),
            BRatio = Data.ReadFloat()
        };
    }

    public override void ParseSnapshot(object ObjectData)
    {
        Data Data = (Data)ObjectData;
        Distance = Data.Distance;
        Anchor = Data.Anchor;
        ARatio = Data.ARatio;
        BRatio = Data.BRatio;
        SetDistance(Distance);
    }

    public override void Interpolate(object FromData, object ToData,
float Time)
    {
        Data From = (Data)FromData;
        Data To = (Data)ToData;
        Distance = From.Distance + (To.Distance - From.Distance) *
Time;

        ARatio = From.ARatio + (To.ARatio - From.ARatio) * Time;
        BRatio = From.BRatio + (To.BRatio - From.BRatio) * Time;
        SetDistance(Distance);
    }

```

```

public override void Extrapolate(object FromData, float Time)
{
    Data From = (Data)FromData;
    Distance = From.Distance;
    ARatio = From.ARatio;
    BRatio = From.BRatio;
    SetDistance(Distance);
}
}
}

```

---

## *Assets/DZEngine/Entities/Components/ AbstractWorldEntity.cs*

---

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

using UnityEngine;

namespace DeadZoneEngine.Entities
{
    public class EntityID
    {
        public static Dictionary<ushort, _IInstantiatableDeletable>
IDToObject = new Dictionary<ushort, _IInstantiatableDeletable>();

        public static ushort StaticID = 0;

        public AbstractWorldEntity Self { get; private set; }
        public ushort Value { get; private set; }
        public EntityID(AbstractWorldEntity Self)
        {
            this.Self = Self;
            AssignNewID();
        }
        public EntityID(AbstractWorldEntity Self, ushort ID)
        {
            this.Self = Self;
            if (IDToObject.ContainsKey(ID))
            {
                Debug.LogError("EntityID(ulong ID) => ID " + ID + " already
exists!");
                return;
            }
            Value = ID;
            IDToObject.Add(Value, Self);
        }

        private void AssignNewID()
        {
            ushort Next = StaticID++;
            if (IDToObject.Count >= ushort.MaxValue - 100)
            {

```

```

        Debug.LogError("No more IDs to give!");
        return;
    }
    while (IDToObject.ContainsKey(Next))
    {
        Next = StaticID++;
    }
    Value = Next;
    IDToObject.Add(Value, Self);
}

public void ChangeID()
{
    Remove(this);
    AssignNewID();
}

public void ChangeID(ushort New, bool Replace = false)
{
    if (IDToObject.ContainsKey(New))
    {
        if (IDToObject[New] != Self)
        {
            if (Replace)
            {
                DZEngine.Destroy(IDToObject[New]);
                IDToObject[New] = Self;
            }
            else
            {
                Debug.LogError("Could not change ID as an object at
that ID already exists...");
            }
        }
    }
    else
    {
        IDToObject.Add(New, IDToObject[Value]);
        Remove(this);
    }
    Value = New;
}

public static void Remove(EntityID ID)
{
    if (IDToObject.ContainsKey(ID))
    {
        IDToObject.Remove(ID);
    }
    else
        Debug.LogError("EntityID.Remove(EntityID ID) => ID " + ID +
" does not exist!");
}

public static _IInstantiatableDeletable GetObject(ushort ID)
{
    if (IDToObject.ContainsKey(ID))
        return IDToObject[ID];
    return null;
}

```



```

public static bool Exists(ushort ID)
{
    return IDToObject.ContainsKey(ID);
}

public static implicit operator ushort(EntityID ID)
{
    return ID.Value;
}

public override bool Equals(object Obj)
{
    return Obj is EntityID && this == (EntityID)Obj;
}

public override int GetHashCode()
{
    return Value.GetHashCode();
}

public static bool operator ==(EntityID A, EntityID B)
{
    if (ReferenceEquals(A, null) && ReferenceEquals(B, null))
        return true;
    else if (ReferenceEquals(A, null) || ReferenceEquals(B, null))
        return false;
    return A.Value == B.Value;
}

public static bool operator !=(EntityID A, EntityID B)
{
    if (ReferenceEquals(A, null) && ReferenceEquals(B, null))
        return false;
    else if (ReferenceEquals(A, null) || ReferenceEquals(B, null))
        return true;
    return A.Value != B.Value;
}
}

public abstract class AbstractWorldEntity : _IInstantiatableDeletable
{
    public EntityID ID { get; set; } = null;
    public AbstractWorldEntity()
    {
        if (this is IServerSendable)
            ID = new EntityID(this);
        DZEngine.Instantiate(this);
    }
    public AbstractWorldEntity(ushort ID)
    {
        if (this is IServerSendable)
            this.ID = new EntityID(this, ID);
        DZEngine.Instantiate(this);
    }

    public bool Active { get; set; } = true;
    public bool PhysicallyActive { get; set; } = !DZSettings.Client;
    public bool FlaggedToDelete { get; set; } = false;
    public bool Disposed { get; set; } = false;

    public virtual void Instantiate() { }
    public object Create()
    {
        OnCreate();
    }
}

```

```

        return this;
    }
    protected virtual void OnCreate () { }

    public void Delete ()
    {
        FlaggedToDelete = true;
        if (ID != null)
            EntityID.Remove (ID);
        OnDelete ();
    }
    protected virtual void OnDelete () { }

    public abstract byte [] GetBytes ();
    public abstract object GetSnapshot ();
    public abstract void ParseBytes (DZNetwork.Packet Data);
    public abstract void ParseSnapshot (object ObjectData);
    public abstract void Interpolate (object FromData, object ToData,
float Time);
    public abstract void Extrapolate (object FromData, float Time);
    }
}

```

---

***Assets/DZEngine/Entities/Components/  
PhysicalJoint.cs***

---

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace DeadZoneEngine.Entities
{
    public abstract class PhysicalJoint : AbstractWorldEntity,
IIterableUpdatable
    {
        public PhysicalJoint () { }

        public PhysicalJoint (ushort ID) : base (ID) { }

        public virtual void PreUpdate () { }
        public virtual void IteratedUpdate () { }
    }
}

```

---

**Assets/DZEngine/Entities/Components/  
PhysicalObject.cs**

---

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

using UnityEngine;

namespace DeadZoneEngine.Entities
{
    public abstract class PhysicalObject : AbstractWorldEntity,
    IPhysicsUpdatable
    {
        public GameObject Self;
        public AbstractWorldEntity Parent;
        protected Rigidbody2D RB;

        public PhysicalObject()
        {
            Self = new GameObject();
            RB = Self.AddComponent<Rigidbody2D>();
            RB.angularDrag = 0;
            RB.drag = 0;
            RB.gravityScale = 0;
            RB.sharedMaterial =
Resources.Load<PhysicsMaterial2D>("PhysicsMaterial/Zero");
            RB.interpolation = RigidbodyInterpolation2D.None;
            RB.isKinematic = DZSettings.Client;
        }

        public PhysicalObject(ushort ID) : base(ID)
        {
            Self = new GameObject();
            RB = Self.AddComponent<Rigidbody2D>();
            RB.angularDrag = 0;
            RB.drag = 0;
            RB.gravityScale = 0;
            RB.sharedMaterial =
Resources.Load<PhysicsMaterial2D>("PhysicsMaterial/Zero");
            RB.interpolation = RigidbodyInterpolation2D.None;
            RB.isKinematic = DZSettings.Client;
        }

        protected override void OnDelete()
        {
            GameObject.Destroy(Self);
        }

        public void PhysicsUpdate(float DeltaTime)
        {
            Self.transform.position += (Vector3)Velocity * DeltaTime;
        }
    }
}
```

```

        Self.transform.eulerAngles += new Vector3(0, 0,
AngularVelocity) * DeltaTime;
    }

    public virtual void FixedUpdate() { }
    public virtual void Update() { }
    public virtual void BodyPhysicsUpdate() { }

    private Vector2 PreVelocity;
    public void IsolateVelocity()
    {
        Vector2 Temp = PreVelocity;
        PreVelocity = Velocity;
        Velocity = Temp;
    }

    public void RestoreVelocity()
    {
        Vector2 Temp = Velocity;
        Velocity = PreVelocity;
        PreVelocity = Temp;
    }

    public bool Kinematic { get { return RB.isKinematic; } set {
RB.isKinematic = value; } }
    public int CollisionLayer { get { return Self.layer; } set {
Self.layer = value; } }
    public Vector2 Position { get { return Self.transform.position; }
set { Self.transform.position = value; } }
    public Vector2 Velocity { get { return RB.velocity; } set {
RB.velocity = value; } }
    public float Rotation { get { return Self.transform.eulerAngles.z *
Mathf.Deg2Rad; } set { Self.transform.eulerAngles = new Vector3(0, 0, value
* Mathf.Rad2Deg); } }
    public float AngularVelocity { get { return RB.angularVelocity *
Mathf.Deg2Rad; } set { RB.angularVelocity = value * Mathf.Rad2Deg; } }

    public float _InvMass = 0;
    public float InvMass { get { return _InvMass; } set { _InvMass =
value; if (_InvMass == 0) RB.constraints |=
RigidbodyConstraints2D.FreezePosition; else { RB.mass = 1 / _InvMass;
RB.constraints &= ~RigidbodyConstraints2D.FreezePosition; } } }

    public float _InvInertia = 0;
    public float InvInertia { get { return _InvInertia; } set {
_InvInertia = value; if (_InvInertia == 0) RB.constraints |=
RigidbodyConstraints2D.FreezeRotation; else { RB.inertia = 1 / _InvInertia;
RB.constraints &= ~RigidbodyConstraints2D.FreezeRotation; } } }

    public float Gravity = 1;
}
}

```

---

*Assets/DZEngine/Entities/Components/  
UpdatableAndDeletable.cs*

---

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace DeadZoneEngine.Entities
{
    public interface _IInstantiatableDeletable
    {
        bool Active { get; set; }
        bool FlaggedToDelete { get; set; }
        bool Disposed { get; set; }
        void Delete();
        void Instantiate();
        object Create();
    }

    public interface IServerSendable : _IInstantiatableDeletable
    {
        EntityID ID { get; set; }
        int ServerObjectType { get; set; }
        bool RecentlyUpdated { get; set; }
        bool ProtectedDeletion { get; set; }
        void ServerUpdate();
        byte[] GetBytes();
        object GetSnapshot();
        void ParseBytes(DZNetwork.Packet Data);
        void ParseSnapshot(object Data);
        void Interpolate(object FromData, object ToData, float Time);
        void Extrapolate(object FromData, float Time);
    }

    public interface IRenderer : _IInstantiatableDeletable
    {
        int SortingLayer { get; set; }

        void InitializeRenderer();

        void Render();
    }

    public interface IRenderer<T> : _IInstantiatableDeletable, IRenderer
where T : class
    {
        T RenderObject { get; set; }
    }

    public interface IUpdatable : _IInstantiatableDeletable
    {
        void Update();
    }
}
```

```

        void BodyPhysicsUpdate ();
    }

    public interface IPhysicsUpdatable : _IInstantiatableDeletable
    {
        bool PhysicallyActive { get; set; }
        void FixedUpdate ();
        void IsolateVelocity ();
        void RestoreVelocity ();
    }

    public interface IIterableUpdatable : _IInstantiatableDeletable
    {
        bool PhysicallyActive { get; set; }
        void PreUpdate ();
        void IteratedUpdate ();
    }
}

```

---

### *Assets/DZEngine/DZEngine.cs*

---

```

using System;
using System.Collections;
using System.Collections.Generic;
using System.Collections.ObjectModel;
using System.Linq;
using System.Reflection;
using UnityEngine;

using DeadZoneEngine.Entities;
using DeadZoneEngine.Entities.Components;

namespace DeadZoneEngine
{
    public static class DZEngine
    {
        public static float InvDeltaTime = 0; // 1 / DeltaTime of frame

        /// <summary>
        /// Called on startup
        /// </summary>
        public static void Initialize()
        {
        }

        /// <summary>
        /// Adds a given entity to a DZEngine.ManagedList if its of the
correct type
        /// </summary>
        /// <typeparam name="T">ManagedList Type</typeparam>
        /// <param name="List">List to append to</param>
        /// <param name="Entity">Entity to append</param>
        private static void AddIfContainsInterface<T>(this List<T> List,
object Entity) where T : class
        {

```

```

        T Interface = Entity as T;
        if (Interface != null)
            List.Add(Interface);
    }

    /// <summary>
    /// Calls relevant initialization functions and adds entity to
DZEngine
    /// </summary>
    /// <param name="Entity"></param>
    private static void InstantiateAndAddEntity(object Entity)
    {
        _IInstantiatableDeletable Instantiatable = Entity as
_IInstantiatableDeletable;
        Instantiatable?.Instantiate();

        if (DZSettings.ActiveRenderers)
        {
            IRenderer Renderer = Entity as IRenderer;
            Renderer?.InitializeRenderer();
        }

        _AbstractWorldEntities.AddIfContainsInterface(Entity);
        _UpdatableObjects.AddIfContainsInterface(Entity);
        _PhysicsUpdatableObjects.AddIfContainsInterface(Entity);
        _IterableUpdatableObjects.AddIfContainsInterface(Entity);
        _RenderableObjects.AddIfContainsInterface(Entity);
        _ServerSendableObjects.AddIfContainsInterface(Entity);
    }

    #region DZEngine.ManagedList

        private static HashSet<Type> ManagedListTypes = new
HashSet<Type>(); //Contains all created managed lists
        private static Dictionary<(Type, Type), Delegate> GetInvokeCache =
new Dictionary<(Type, Type), Delegate>(); //Caching delegate functions used
for calling relevant add functions
        private static Dictionary<Type, (Delegate, Delegate)>
GetManagedInvokeCache = new Dictionary<Type, (Delegate, Delegate)>();
//Caching delegate functions used for clearing and updating managed lists

        /// <summary>
        /// Invokes a given delegate on every item that contains the
provided interface
        /// </summary>
        /// <typeparam name="SearchType">Interface the item should
contain</typeparam>
        /// <typeparam name="ListType">Type of list being
checked</typeparam>
        /// <param name="Method">Delegate to call</param>
        /// <param name="List">List being checked</param>
        private static void InvokeIfContainsInterface<SearchType,
ListType>(Delegate Method, List<ListType> List) where SearchType : class
    {
        for (int i = 0; i < List.Count; i++)
        {
            SearchType Item = List[i] as SearchType;
            if (Item != null)
            {
                ((Action<SearchType>)Method)(Item);
            }
        }
    }

```

```

    }
}

/// <summary>
/// Returns the given delegate to add an item to a managed list
/// </summary>
/// <typeparam name="InvokeType">Type of list being invoked
onto</typeparam>
/// <param name="T">Type of item</param>
/// <returns></returns>
private static Action<Delegate, List<InvokeType>>
GetInvokeFromType<InvokeType>(Type T)
{
    var Label = (T, typeof(InvokeType)); //Key for cache dictionary
    if (GetInvokeCache.ContainsKey(Label)) //Check if it does not
already exist
        return (Action<Delegate,
List<InvokeType>>)GetInvokeCache[Label]; //If so return cached delegate
    //Find the right method to generate delegate from
    MethodInfo Method =
typeof(DZEngine).GetMethods(BindingFlags.NonPublic |
BindingFlags.Static).Single(I => I.Name ==
nameof(DZEngine.InvokeIfContainsInterface));
    //Generate delegate from method
    Action<Delegate, List<InvokeType>> DelegateAction =
(Action<Delegate,
List<InvokeType>>)Delegate.CreateDelegate(typeof(Action<Delegate,
List<InvokeType>>), Method.MakeGenericMethod(T, typeof(InvokeType)));
    //Add delegate to cache
    GetInvokeCache.Add(Label, DelegateAction);
    return DelegateAction;
}

private static void UpdateManagedLists ()
{
    foreach (Type T in ManagedListTypes) //Loop through all
existing managed lists
    {
        //Find the appropriate update and clear delegates for
handling the lists
        Delegate UpdateListMethod = null;
        Delegate ClearListMethod = null;
        if (GetManagedInvokeCache.ContainsKey(T))
        {
            UpdateListMethod = GetManagedInvokeCache[T].Item1;
            ClearListMethod = GetManagedInvokeCache[T].Item2;
        }
        else
        {
            Type ActionGeneric =
typeof(Action<>).MakeGenericType(T);
            Type Generic =
typeof(ManagedList<>).MakeGenericType(T);
            MethodInfo UpdateListMethodInfo =
Generic.GetMethod(nameof(ManagedList<object>.UpdateExistingLists));
            MethodInfo ClearListMethodInfo =
Generic.GetMethod(nameof(ManagedList<object>.ClearExistingLists));
            UpdateListMethod =
Delegate.CreateDelegate(ActionGeneric, UpdateListMethodInfo);
            ClearListMethod =
Delegate.CreateDelegate(typeof(Action), ClearListMethodInfo);

```



```

        GetManagedInvokeCache.Add(T, (UpdateListMethod,
ClearListMethod));
    }

    ((Action)ClearListMethod)(); //Clear the managed list of
items
    //Update the managed list with new items
    GetInvokeFromType<AbstractWorldEntity>(T) (UpdateListMethod,
_AbstractWorldEntities);
    GetInvokeFromType<IPhysicsUpdatable>(T) (UpdateListMethod,
_PhysicsUpdatableObjects);
    GetInvokeFromType<IUpdatable>(T) (UpdateListMethod,
_UpdatableObjects);

    GetInvokeFromType<IIteratableUpdatable>(T) (UpdateListMethod,
_IteratableUpdatableObjects);
    GetInvokeFromType<IRenderer>(T) (UpdateListMethod,
_RenderableObjects);
    GetInvokeFromType<IServerSendable>(T) (UpdateListMethod,
_ServerSendableObjects);
    }
}

/// <summary>
/// Defines a list that is automatically updated to contain all
entities of type T assigned to DZEngine
/// This is useful for simply getting a list of specific IRender<>
/// </summary>
/// <typeparam name="T">Type of entity</typeparam>
public class ManagedList<T> : HashSet<T> where T : class
{
    public static List<WeakReference> ExistingLists = new
List<WeakReference>();

    public ManagedList()
    {
        //Keep track of all existing managed lists with weak
references to allow Garbage Collection (GC)
        ExistingLists.Add(new WeakReference(this));
        //Add Type that is being managed to type list
        ManagedListTypes.Add(typeof(T));
    }

    /// <summary>
    /// Removes lists that have been cleaned up by GC
    /// </summary>
    public static void ClearExistingLists()
    {
        ExistingLists.RemoveAll(I =>
        {
            if (I.IsAlive)
            {
                ManagedList<T> L = (ManagedList<T>)I.Target;
                L.Clear();
                return false;
            }
            return true;
        });
    }

    /// <summary>

```

```

    /// Adds a new item to all lists of type T
    /// </summary>
    /// <param name="Item"></param>
    public static void UpdateExistingLists(T Item)
    {
        for (int i = 0; i < ExistingLists.Count; i++)
        {
            ManagedList<T> L =
(M ManagedList<T>) ExistingLists[i].Target;
            L.Add(Item);
        }
    }
}

#endregion

public static void Instantiate(object Item)
{
    EntitiesToPush.Add(Item);
}

private static List<object> EntitiesToPush = new List<object>();
//List of entities to push to DZEngine

//Lists of entity interfaces that define DZEngine
private static List<IServerSendable> _ServerSendableObjects = new
List<IServerSendable>(); //All entities that are sendable across the server
and client
private static List<AbstractWorldEntity> _AbstractWorldEntities =
new List<AbstractWorldEntity>(); //All abstract world entities
private static List<IPhysicsUpdatable> _PhysicsUpdatableObjects =
new List<IPhysicsUpdatable>(); //All objects that use the separate
(isolated) physics loop
private static List<IUpdatable> _UpdatableObjects = new
List<IUpdatable>(); //All objects that use the standard update loop
private static List<IIteratableUpdatable>
_IteratableUpdatableObjects = new List<IIteratableUpdatable>(); //All
objects that use the impulse engine
private static List<IRenderer> _RenderableObjects = new
List<IRenderer>(); //All objects that have a renderer

    /// <summary>
    /// Destroys an entity
    /// </summary>
    /// <param name="Item"></param>
    public static void Destroy(_IInstantiatableDeletable Item)
    {
        Item.FlaggedToDelete = true;
    }

    /// <summary>
    /// Adds an unmanaged entity (not of type AbstractWorldEntity),
these are called components
    /// </summary>
    /// <param name="Component"></param>
    public static void AddComponent(_IInstantiatableDeletable
Component)
    {
        EntitiesToPush.Add(Component);
    }

//public getters for lists

```

```

        public static ReadOnlyCollection<IServerSendable>
ServerSendableObjects { get { return _ServerSendableObjects.AsReadOnly(); }
}

        public static ReadOnlyCollection<AbstractWorldEntity>
AbstractWorldEntities { get { return _AbstractWorldEntities.AsReadOnly(); }
}

        public static ReadOnlyCollection<IPhysicsUpdatable>
PhysicsUpdatableObjects { get { return
_PhysicsUpdatableObjects.AsReadOnly(); } }
        public static ReadOnlyCollection<IUpdatable> UpdatableObjects { get
{ return _UpdatableObjects.AsReadOnly(); } }
        public static ReadOnlyCollection<IIteratableUpdatable>
IteratableUpdatableObjects { get { return
_IteratableUpdatableObjects.AsReadOnly(); } }
        public static ReadOnlyCollection<IRenderer> RenderableObjects { get
{ return _RenderableObjects.AsReadOnly(); } }

        /// <summary>
        /// Releases and disposes of all entities and their managed
resources
        /// </summary>
public static void ReleaseResources ()
{
    EntitiesToPush.RemoveAll (I =>
    {
        InstantiateAndAddEntity (I);
        return true;
    });

    _ServerSendableObjects.RemoveAll (I =>
    {
        return DeleteHandle (I, true);
    });

    _AbstractWorldEntities.RemoveAll (I =>
    {
        return DeleteHandle (I, true);
    });

    _PhysicsUpdatableObjects.RemoveAll (I =>
    {
        return DeleteHandle (I, true);
    });

    _UpdatableObjects.RemoveAll (I =>
    {
        return DeleteHandle (I, true);
    });

    _IteratableUpdatableObjects.RemoveAll (I =>
    {
        return DeleteHandle (I, true);
    });

    _RenderableObjects.RemoveAll (I =>
    {
        return DeleteHandle (I, true);
    });
}

        /// <summary>

```

```

    /// Checks the deletion of an object
    /// </summary>
    /// <param name="DeletableObject">Object to delete</param>
    /// <param name="ForceDelete">Force delete the object
regardless</param>
    /// <returns>true if object was disposed, otherwise false</returns>
    private static bool DeleteHandle(_IInstantiatableDeletable
DeletableObject, bool ForceDelete = false)
    {
        if (DeletableObject.FlaggedToDelete || ForceDelete) //Check if
the object is flagged to delete or is forced to delete
        {
            if (!DeletableObject.Disposed) //If the object has not
already been disposed the perform delete
            {
                DeletableObject.Disposed = true;
                DeletableObject.Delete();
            }
            return true;
        }
        return false;
    }

    /// <summary>
    /// Gets the bytes of a given entity
    /// </summary>
    /// <param name="Item"></param>
    /// <returns></returns>
    public static byte[] GetBytes(IServerSendable Item)
    {
        //Header Contents => EntityID, FlaggedToDelete, EntityType
        List<byte> Data = new List<byte>();
        Data.AddRange(BitConverter.GetBytes(Item.ID));
        Data.AddRange(BitConverter.GetBytes(Item.FlaggedToDelete));
        //Provide actual data if the item is not about to be deleted
        //otherwise this data is redundant
        if (!Item.FlaggedToDelete)
        {

Data.AddRange(BitConverter.GetBytes(Item.ServerObjectType));
            Data.AddRange(Item.GetBytes());
        }
        return Data.ToArray();
    }

    public static void NonPhysicsUpdate()
    {
        //Push entites into DZEngine
        EntitiesToPush.RemoveAll(I =>
        {
            InstantiateAndAddEntity(I);
            return true;
        });

        //Remove deleted AbstractWorldEntities
        _AbstractWorldEntities.RemoveAll(I =>
        {
            return DeleteHandle(I);
        });

        //Remove deleted server entites
    }

```

```

        _ServerSendableObjects.RemoveAll(I =>
        {
            I.RecentlyUpdated = false;
            I.ServerUpdate();
            return DeleteHandle(I);
        });

        //Isolate the general physics updates from creature body
physics -> this is specific for maintaining physic objects inside of
creature bodies
        //(the creature body is updated relative to itself without the
need to worry about countering general physics (its isolated from general
physics))
        _PhysicsUpdatableObjects.RemoveAll(I =>
        {
            return DeleteHandle(I);
        });

        _UpdatableObjects.RemoveAll(I =>
        {
            return DeleteHandle(I);
        });

        //Check and resolve physics constraints from impulse engine
        _IteratableUpdatableObjects.RemoveAll(I =>
        {
            return DeleteHandle(I);
        });

        //Render renderable entities
        _RenderableObjects.RemoveAll(I =>
        {
            if (!I.FlaggedToDelete && I.Active)
            {
                if (DZSettings.ActiveRenderers)
                    I.Render();
            }
            return DeleteHandle(I);
        });

        UpdateManagedLists(); //Update DZEngine.ManagedLists
    }

    public static void PhysicsUpdate()
    {
        InvDeltaTime = Game.ServerTickRate;

        //Isolate the general physics updates from creature body
physics -> this is specific for maintaining physic objects inside of
creature bodies
        //(the creature body is updated relative to itself without the
need to worry about countering general physics (its isolated from general
physics))
        for (int i = 0; i < _PhysicsUpdatableObjects.Count; i++)
        {
            if (_PhysicsUpdatableObjects[i].Active &&
_PhysicsUpdatableObjects[i].PhysicallyActive)
                _PhysicsUpdatableObjects[i].IsolateVelocity();
        }
    }

```

```

    for (int i = 0; i < _UpdatableObjects.Count; i++)
    {
        if (_UpdatableObjects[i].Active)
            _UpdatableObjects[i].BodyPhysicsUpdate(); //This is
specific to entites mainly to update self-righting bodies or other body
animation specific physics
//its
seperated and run in a seperate physics operation to prevent self-righting
body physics from being counteracted from normal physics (such as gravity).
//In other
words this simply isolates the body physics from the standard physics
    }

    //Check and resolve physics constraints from impulse engine
    for (int i = 0; i < _IteratableUpdatableObjects.Count; i++)
    {
        if (!_IteratableUpdatableObjects[i].FlaggedToDelete &&
        _IteratableUpdatableObjects[i].Active &&
        _IteratableUpdatableObjects[i].PhysicallyActive)
        {
            _IteratableUpdatableObjects[i].PreUpdate();
        }
    }
    for (int j = 0; j < DZSettings.NumPhysicsIterations; j++)
    {
        for (int i = 0; i < _IteratableUpdatableObjects.Count; i++)
        {
            if (_IteratableUpdatableObjects[i].Active &&
            _IteratableUpdatableObjects[i].PhysicallyActive)
                _IteratableUpdatableObjects[i].IteratedUpdate();
        }
    }

    Physics2D.Simulate(Time.fixedDeltaTime / 2);

    //Restore the velocities back to normal, we are no longer
considering the entity in an isolated system
    for (int i = 0; i < _PhysicsUpdatableObjects.Count; i++)
    {
        if (_PhysicsUpdatableObjects[i].Active &&
        _PhysicsUpdatableObjects[i].PhysicallyActive)
        {
            _PhysicsUpdatableObjects[i].RestoreVelocity();
            _PhysicsUpdatableObjects[i].FixedUpdate();
        }
    }

    //Update updatable entities
    for (int i = 0; i < _UpdatableObjects.Count; i++)
    {
        if (_UpdatableObjects[i].Active)
            _UpdatableObjects[i].Update();
    }

    //Check and resolve physics constraints from impulse engine
    for (int i = 0; i < _IteratableUpdatableObjects.Count; i++)
    {
        if (_IteratableUpdatableObjects[i].Active &&
        _IteratableUpdatableObjects[i].PhysicallyActive)
            _IteratableUpdatableObjects[i].PreUpdate();
    }

```

```

    for (int j = 0; j < DZSettings.NumPhysicsIterations; j++)
    {
        for (int i = 0; i < _IterableUpdatableObjects.Count; i++)
        {
            if (_IterableUpdatableObjects[i].Active &&
                _IterableUpdatableObjects[i].PhysicallyActive)
                _IterableUpdatableObjects[i].IteratedUpdate();
        }

        Physics2D.Simulate(Time.fixedDeltaTime / 2);
    }

    /// <summary>
    /// Called once per frame
    /// </summary>
    public static void FixedUpdate ()
    {
        InvDeltaTime = Game.ServerTickRate;

        //Push entites into DZEngine
        EntitiesToPush.RemoveAll(I =>
        {
            InstantiateAndAddEntity(I);
            return true;
        });

        //Remove deleted AbstractWorldEntities
        _AbstractWorldEntities.RemoveAll(I =>
        {
            return DeleteHandle(I);
        });

        //Remove deleted server entites
        _ServerSendableObjects.RemoveAll(I =>
        {
            I.RecentlyUpdated = false;
            I.ServerUpdate();
            return DeleteHandle(I);
        });

        //Isolate the general physics updates from creature body
        physics -> this is specific for maintaining physic objects inside of
        creature bodies
        //(the creature body is updated relative to itself without the
        need to worry about countering general physics (its isolated from general
        physics))
        _PhysicsUpdatableObjects.RemoveAll(I =>
        {
            if (!I.FlaggedToDelete && I.Active && I.PhysicallyActive)
            {
                I.IsolateVelocity();
            }
            return DeleteHandle(I);
        });

        _UpdatableObjects.RemoveAll(I =>
        {
            if (!I.FlaggedToDelete && I.Active)
            {

```

```

        I.BodyPhysicsUpdate(); //This is specific to entites
mainly to update self-righting bodies or other body animation specific
physics
//its seperated and run in a
seperate physics operation to prevent self-righting body physics from being
counteracted from normal physics (such as gravity).
//In other words this simply
isolates the body physics from the standard physics
    }
    return DeleteHandle(I);
});

//Check and resolve physics constraints from impulse engine
_IteratableUpdatableObjects.RemoveAll(I =>
{
    if (!I.FlagedToDelete && I.Active && I.PhysicallyActive)
    {
        I.PreUpdate();
    }
    return DeleteHandle(I);
});
for (int j = 0; j < DZSettings.NumPhysicsIterations; j++)
{
    for (int i = 0; i < _IteratableUpdatableObjects.Count; i++)
    {
        if (_IteratableUpdatableObjects[i].Active &&
_IteratableUpdatableObjects[i].PhysicallyActive)
            _IteratableUpdatableObjects[i].IteratedUpdate();
    }
}

Physics2D.Simulate(Time.fixedDeltaTime / 2f);

//Restore the velocities back to normal, we are no longer
considering the entity in an isolated system
for (int i = 0; i < _PhysicsUpdatableObjects.Count; i++)
{
    if (_PhysicsUpdatableObjects[i].Active &&
_PhysicsUpdatableObjects[i].PhysicallyActive)
    {
        _PhysicsUpdatableObjects[i].RestoreVelocity();
        _PhysicsUpdatableObjects[i].FixedUpdate();
    }
}

//Update updatable entities
for (int i = 0; i < _UpdatableObjects.Count; i++)
{
    if (_UpdatableObjects[i].Active)
        _UpdatableObjects[i].Update();
}

//Check and resolve physics constraints from impulse engine
for (int i = 0; i < _IteratableUpdatableObjects.Count; i++)
{
    if (_IteratableUpdatableObjects[i].Active &&
_IteratableUpdatableObjects[i].PhysicallyActive)
        _IteratableUpdatableObjects[i].PreUpdate();
}
for (int j = 0; j < DZSettings.NumPhysicsIterations; j++)
{

```



```

        for (int i = 0; i < _IterableUpdatableObjects.Count; i++)
        {
            if (_IterableUpdatableObjects[i].Active &&
                _IterableUpdatableObjects[i].PhysicallyActive)
                _IterableUpdatableObjects[i].IteratedUpdate();
        }

        Physics2D.Simulate(Time.fixedDeltaTime / 2f);

        //Render renderable entities
        _RenderableObjects.RemoveAll(I =>
        {
            if (!I.FlaggedToDelete && I.Active)
            {
                if (DZSettings.ActiveRenderers)
                    I.Render();
            }
            return DeleteHandle(I);
        });

        UpdateManagedLists(); //Update DZEngine.ManagedLists
    }
}

```

---

### *Assets/DZEngine/Math2DExtensions.cs*

---

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

using UnityEngine;

namespace DeadZoneEngine
{
    /// <summary>
    /// Defines a 2x2 Matrix for transformations
    /// </summary>
    public struct Mat22
    {
        public Vector2 Col1;
        public Vector2 Col2;

        public Mat22(float Angle)
        {
            float C = Mathf.Cos(Angle);
            float S = Mathf.Sin(Angle);

            Col1.x = C; Col2.x = -S;
            Col1.y = S; Col2.y = C;
        }

        public Mat22(Vector2 DirVector) //Dir Vector should be normalized

```

```

    {
        float C = DirVector.x;
        float S = DirVector.y;

        Col1.x = C; Col2.x = -S;
        Col1.y = S; Col2.y = C;
    }

    public Mat22(Vector2 Col1, Vector2 Col2)
    {
        this.Col1 = Col1;
        this.Col2 = Col2;
    }

    public Mat22 Transpose()
    {
        return new Mat22(new Vector2(Col1.x, Col2.x), new
Vector2(Col1.y, Col2.y));
    }

    public Mat22 Invert()
    {
        float a = Col1.x, b = Col2.x, c = Col1.y, d = Col2.y;
        Mat22 B;
        float det = a * d - b * c;
        det = 1.0f / det;
        B.Col1.x = det * d; B.Col2.x = -det * b;
        B.Col1.y = -det * c; B.Col2.y = det * a;
        return B;
    }

    public static Vector2 operator *(Mat22 A, Vector2 B)
    {
        return new Vector2(A.Col1.x * B.x + A.Col2.x * B.y, A.Col1.y *
B.x + A.Col2.y * B.y);
    }

    public static Mat22 operator +(Mat22 A, Mat22 B)
    {
        return new Mat22(A.Col1 + B.Col1, A.Col2 + B.Col2);
    }
}

/// <summary>
/// Defines additional Math functions
/// </summary>
public class Math2D
{
    public static Vector2 Cross(Vector2 A, float B)
    {
        return new Vector2(B * A.y, -B * A.x);
    }

    public static Vector2 Cross(float A, Vector2 B)
    {
        return new Vector2(-A * B.y, A * B.x);
    }

    public static float Cross(Vector2 A, Vector2 B)
    {
        return A.x * B.y - A.y * B.x;
    }
}

```

```

    }
}

```

---

## Client/Assets/Templates/ClientHandle.cs

---

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Net;
using System.Text;
using System.Threading.Tasks;

using UnityEngine;
using UnityEngine.InputSystem;

using DeadZoneEngine;
using DZNetwork;
using DeadZoneEngine.Entities;
using DeadZoneEngine.Controllers;

namespace ClientHandle
{
    public class ClientID
    {
        public static Dictionary<ushort, Client> ConnectedClients = new
Dictionary<ushort, Client>();
        public static Dictionary<IPEndPoint, ushort> EndPointToID = new
Dictionary<IPEndPoint, ushort>(new IPEndPointComparer());
        private static ushort StaticID = 0;
        public Client Self { get; private set; }
        public IPEndPoint EndPoint { get; private set; }
        private ushort Value;
        public ushort ID
        {
            get
            {
                return Value;
            }
            private set
            {
                Value = value;
                if (EndPoint != null)
                    if (EndPointToID.ContainsKey(EndPoint))
                        EndPointToID[EndPoint] = Value;
                    else
                        EndPointToID.Add(EndPoint, Value);
            }
        }

        public ClientID(Client Self, IPEndPoint EndPoint)
        {
            this.Self = Self;
            this.EndPoint = EndPoint;
            AssignNewID();
        }
    }
}

```

```

public static Client GetClient(IPEndPoint EndPoint)
{
    if (EndPoint == null)
        return null;
    if (EndPointToID.ContainsKey(EndPoint))
        if (ConnectedClients.ContainsKey(EndPointToID[EndPoint]))
            return ConnectedClients[EndPointToID[EndPoint]];
    return null;
}

public static Client GetClient(ushort ID)
{
    if (ConnectedClients.ContainsKey(ID))
        return ConnectedClients[ID];
    return null;
}

private void AssignNewID()
{
    ushort Next = StaticID++;
    if (ConnectedClients.Count >= ushort.MaxValue - 100)
    {
        Debug.LogError("No more IDs to give!");
        return;
    }
    while (ConnectedClients.ContainsKey(Next))
    {
        Next = StaticID++;
    }
    ID = Next;
    ConnectedClients.Add(ID, Self);
}

public void ChangeID()
{
    Remove(ID);
    AssignNewID();
}

public void ChangeID(ushort New, bool Replace = false)
{
    if (ConnectedClients.ContainsKey(New))
    {
        if (ConnectedClients[New] != Self)
        {
            if (Replace)
                ConnectedClients[New] = Self;
            else
                Debug.LogError("Could not change ClientID as an
object at that ClientID already exists...");
        }
    }
    else
    {
        ConnectedClients.Add(New, Self);
        Remove(ID);
    }
    ID = New;
}

```

```

public static implicit operator ushort(ClientID ID)
{
    return ID.ID;
}

public static void Remove(ushort ID)
{
    if (ConnectedClients.ContainsKey(ID))
    {
        if (ConnectedClients[ID].EndPoint != null)
            EndPointToID.Remove(ConnectedClients[ID].EndPoint);
        ConnectedClients.Remove(ID);
    }
    else
        Debug.LogError("ClientID.Remove(ClientID ID) => ID " + ID +
" does not exist!");
}

public static void Remove(IPEndPoint EndPoint)
{
    if (EndPointToID.ContainsKey(EndPoint))
    {
        Remove(EndPointToID[EndPoint]);
    }
    else
        Debug.LogError("ClientID.Remove(EndPoint EndPoint) =>
EndPoint does not exist!");
}
}

/// <summary>
/// Contains information on each player and which client they refer to
/// </summary>
public class Client
{
    public static int MaxNumPlayers = 8;
    public const int TicksToTimeout = 60;

    public ClientID ID;

    public IPEndPoint EndPoint;
    public Player[] Players;
    public byte NumPlayers { get; private set; }

    public bool LostConnection = false;
    public int TicksSinceConnectionLoss = 0;

    private Client(IPEndPoint EndPoint = null)
    {
        this.EndPoint = EndPoint;
        ID = new ClientID(this, EndPoint);
        Players = new Player[MaxNumPlayers];
    }

    public static Client GetClient(IPEndPoint EndPoint = null)
    {
        Client Client = ClientID.GetClient(EndPoint);
        if (Client == null)
        {
            Client = new Client(EndPoint);
        }
    }
}

```

```

        return Client;
    }

    public Player AddPlayer()
    {
        for (byte i = 0; i < Players.Length; i++)
        {
            if (Players[i] == null)
            {
                Players[i] = new Player(i);
                NumPlayers++;
                return Players[i];
            }
        }
        Debug.LogError("Max number of players reached");
        return null;
    }

    public void RemovePlayer(int PlayerID)
    {
        if (Players[PlayerID] == null)
        {
            Debug.LogError("Player does not exist");
            return;
        }

        Players[PlayerID].Destroy();
        Players[PlayerID] = null;
        NumPlayers--;
    }

    public void RemoveAllPlayers()
    {
        for (int i = 0; i < Players.Length; i++)
        {
            Players[i].Destroy();
            Players[i] = null;
        }
        NumPlayers = 0;
    }

    public void Destroy()
    {
        ClientID.Remove(EndPoint);
        for (int i = 0; i < Players.Length; i++)
            if (Players[i] != null)
                Players[i].Destroy();
    }
}

public class Player
{
    public byte ID { get; private set; }

    private PlayerController _Controller;
    public PlayerController Controller
    {
        get
        {
            return _Controller;
        }
    }
}

```

```

        set
        {
            _Controller = value;
            _Controller.Owner = this;
            if (Entity != null)
            {
                _Controller.PlayerControl = Entity.Controller;
                Entity.Controller.Owner = _Controller;
            }
        }
    }

    public PlayerCreature Entity;

    public Player(byte ID)
    {
        Entity = new PlayerCreature();
        Entity.ProtectedDeletion = true;
        this.ID = ID;
    }

    public void Destroy()
    {
        DZEngine.Destroy(Entity);
        if (_Controller != null)
            _Controller.Disable();
    }

    public byte[] GetBytes()
    {
        List<byte> Data = new List<byte>();
        Data.Add(ID);
        Data.AddRange(BitConverter.GetBytes(Entity.ID));
        return Data.ToArray();
    }
}

```

---

### *Assets/Utility/TriggerPlate.cs*

---

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

using UnityEngine;

using DeadZoneEngine;
using DeadZoneEngine.Entities;
using DZNetwork;
using ClientHandle;

public class TriggerPlate : AbstractWorldEntity, IUpdatable, IRenderer,
IServerSendable
{

```

```

    public int ServerObjectType { get; set; } =
(int)DZSettings.EntityType.TriggerPlate;
    public bool RecentlyUpdated { get; set; } = false;
    public bool ProtectedDeletion { get; set; } = false;

    public int SortingLayer { get; set; }

    private GameObject Self;
    private SpriteRenderer Renderer;

    private GameObject Bar;
    private SpriteRenderer BarRenderer;

    public Vector2 Size = new Vector2(1, 1);
    public float Value = 0;
    public float Incrementer = 0;
    public float MaxValue = 10;

    public int RequiredNumPlayers = 0;

    public Action OnTrigger = null;
    public Action<Player> OnInteract = null;

    public TriggerPlate(ushort ID) : base(ID)
    {
        Init(new Vector2(0, 0));
    }

    public TriggerPlate(Vector2 Size, Vector2 Position, int
RequiredNumPlayers = 0, float MaxValue = 10)
    {
        this.Size = Size;
        this.MaxValue = MaxValue;
        this.RequiredNumPlayers = RequiredNumPlayers;
        Init(Position);
    }

    private void Init(Vector2 Position)
    {
        Self = new GameObject();
        Self.transform.localScale = Size;
        Self.transform.position = Position;

        Bar = new GameObject();
        Bar.transform.parent = Self.transform;
        Bar.transform.localPosition = new Vector2(-1, 0);
    }

    public void InitializeRenderer()
    {
        Renderer = Self.AddComponent<SpriteRenderer>();
        Renderer.sortingLayerName = "TriggerPlates";
        Renderer.sortingOrder = 0;
        Renderer.sprite = Resources.Load<Sprite>("Sprites/Square");

        BarRenderer = Bar.AddComponent<SpriteRenderer>();
        BarRenderer.sortingLayerName = "TriggerPlates";
        BarRenderer.sortingOrder = 1;
        BarRenderer.sprite = Resources.Load<Sprite>("Sprites/Square");
        BarRenderer.color = new Color(0, 0.64f, 0.9f);
    }

```



```

    public void Render()
    {
        BarRenderer.transform.localPosition = new Vector2(-0.5f + Value /
2, 0);
        BarRenderer.transform.localScale = new Vector2(Value, 0.9f);
    }

    public void ServerUpdate()
    {
    }

    private bool CheckBound(Vector2 Pos)
    {
        return Pos.x > Self.transform.position.x -
Self.transform.localScale.x / 2 &&
            Pos.x < Self.transform.position.x +
Self.transform.localScale.x / 2 &&
            Pos.y > Self.transform.position.y -
Self.transform.localScale.y / 2 &&
            Pos.y < Self.transform.position.y +
Self.transform.localScale.y / 2;
    }

    private bool CheckInBounds(Vector2 Position, float Margin)
    {
        return CheckBound(new Vector2(Position.x - Margin, Position.y -
Margin)) ||
            CheckBound(new Vector2(Position.x + Margin, Position.y +
Margin)) ||
            CheckBound(new Vector2(Position.x - Margin, Position.y +
Margin)) ||
            CheckBound(new Vector2(Position.x + Margin, Position.y -
Margin));
    }

    private bool Triggered = false;
    private Player LastPlayerTrigger = null;
    private Client LastClientTrigger = null;
    public void Update()
    {
        List<Client> Clients = ClientID.ConnectedClients.Values.ToList();
        int Count = 0;
        int TotalNumPlayers = 0;
        foreach (Client C in Clients)
        {
            if (C == null) continue;
            TotalNumPlayers += C.NumPlayers;
            for (int i = 0; i < C.Players.Length; i++)
            {
                if (C == LastClientTrigger && LastPlayerTrigger != null &&
i == LastPlayerTrigger.ID && C.Players[i] == null) LastPlayerTrigger =
null;
                if (C.Players[i] == null || C.Players[i].Entity == null)
continue;

                PlayerCreature Entity = C.Players[i].Entity;

                if (CheckInBounds(Entity.Position, 0.5f))
                {
                    Count++;
                }
            }
        }
    }

```

```

        == null)
            if (Entity.Controller.Interact > 0 && LastPlayerTrigger
                {
                    LastPlayerTrigger = C.Players[i];
                    LastClientTrigger = C;
                    OnInteract?.Invoke(C.Players[i]);
                    if (RequiredNumPlayers < 0)
                    {
                        Incrementer ++;
                        if (Incrementer > MaxValue)
                        {
                            Incrementer = 0;
                        }
                        Value = (float)Incrementer / MaxValue;
                    }
                }
            else if (Entity.Controller.Interact <= 0.1f &&
C.Players[i] == LastPlayerTrigger)
            {
                LastPlayerTrigger = null;
            }
        }
    }
    if (TotalNumPlayers > 0)
    {
        if (RequiredNumPlayers == 0 && Count == TotalNumPlayers)
        {
            Value += Time.fixedDeltaTime / 3;
        }
        else if (RequiredNumPlayers > 0 && Count >= RequiredNumPlayers)
        {
            Value += Time.fixedDeltaTime / 3;
        }
        else if (RequiredNumPlayers >= 0)
        {
            Value -= Time.fixedDeltaTime * 2;
            Triggered = false;
        }
    }

    if (Value >= 1 && Triggered == false)
    {
        Triggered = true;
        Value = 1;
        OnTrigger?.Invoke();
    }
    else if (Value < 0)
        Value = 0;
}

public void BodyPhysicsUpdate ()
{
}

protected override void OnDelete ()
{
    GameObject.Destroy(Self);
    GameObject.Destroy(Bar);
}

```

```

public override byte[] GetBytes ()
{
    List<byte> Data = new List<byte> ();
    Data.AddRange (BitConverter.GetBytes (Self.transform.position.x));
    Data.AddRange (BitConverter.GetBytes (Self.transform.position.y));
    Data.AddRange (BitConverter.GetBytes (Size.x));
    Data.AddRange (BitConverter.GetBytes (Size.y));
    Data.AddRange (BitConverter.GetBytes (Mathf.Min (Value, 1)));
    return Data.ToArray ();
}

public override void ParseBytes (Packet Data)
{
    ParseSnapshot (ParseBytesToSnapshot (Data));
}

public struct Data
{
    public Vector2 Position;
    public Vector2 Size;
    public float Value;
}

public override object GetSnapshot ()
{
    return new Data ()
    {
        Position = Self.transform.position,
        Size = Size,
        Value = Value
    };
}

public static object ParseBytesToSnapshot (DZNetwork.Packet Data)
{
    return new Data ()
    {
        Position = new Vector2 (Data.ReadFloat (), Data.ReadFloat ()),
        Size = new Vector2 (Data.ReadFloat (), Data.ReadFloat ()),
        Value = Data.ReadFloat ()
    };
}

public override void ParseSnapshot (object ObjectData)
{
    Data Data = (Data)ObjectData;
    Self.transform.position = Data.Position;
    Size = Data.Size;
    Value = Data.Value;
}

public override void Interpolate (object FromData, object ToData, float
Time)
{
    Data From = (Data)FromData;
    Data To = (Data)ToData;
    Self.transform.position = From.Position + (To.Position -
From.Position) * Time;
    Size = From.Size;
    Value = From.Value + (To.Value - From.Value) * Time;
}

```

```
}
```

---

## *Assets/Utility/Tilemap.cs*

---

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

using UnityEngine;
using UnityEngine.UI;

using DeadZoneEngine;
using DeadZoneEngine.Entities;
using System.Text.RegularExpressions;

/// <summary>
/// Defines a tile of a tilemap
/// </summary>
public struct Tile
{
    public int NumFrames; //Number of animation frames
    public int AnimationFrame; //Animation frame index
    public int TileIndex; //Which tile from tile pallet
    public int Blank; //Is this tile blank?
    public int Render; //Is this tile being rendered?

    public Tile(int Blank = 0, int TileIndex = 0, int NumFrames = 1, int
AnimationFrame = 0, int Render = 1)
    {
        this.TileIndex = TileIndex;
        this.NumFrames = NumFrames;
        this.AnimationFrame = AnimationFrame;
        this.Blank = Blank;
        this.Render = Render;
    }
}

/// <summary>
/// Defines a tilepallet for a tilemap
/// </summary>
public class TilePallet
{
    public int NumTiles; //Number of different tiles
    public Texture2D Pallet; //Pallet
    public int TileStride; //Number of pixels between tiles
    public int[] FrameCount; //Number of animation frames for each tile
}

public class Tilemap : AbstractWorldEntity, IUpdatable, IRenderer,
IServerSendable
{
    public int ServerObjectType { get; set; } =
(int)DZSettings.EntityType.Tilemap;
    public bool RecentlyUpdated { get; set; } = false;
}
```

```

public bool ProtectedDeletion { get; set; } = false;

public int SortingLayer { get; set; }

//ComputeShaders for GPU for rendering wall and floor tilemaps
private ComputeShader WallCompute;
private ComputeShader FloorCompute;
private int ComputeKernel; //Kernel index of GPU function

protected GameObject Self;

public Tile[] FloorMap;
public Tile[] WallMap;
// Map format is height followed by width such that (0, 0) tile is the
top left of the tilemap
//
//           x
//           ----->
//           | [ 0 0 0 ]
//           y | [ 0 0 0 ]
//           v [ 0 0 0 ]
//
// This is to make rendering easier in the compute shader

//Buffers for wall and floor for passing tilemaps to GPU
ComputeBuffer WallBuffer;
ComputeBuffer FloorBuffer;

//Components for rendering to Unity
private RawImage FloorRender;
private RawImage[] Rows;
private RenderTexture WallRenderTexture;
private RenderTexture FloorRenderTexture;

//Tile pallets for wall and floor
public TilePallet WallTilePalletData;
public TilePallet FloorTilePalletData;

private float TilesPerUnit = 1; //Size of tiles in respect to a unity
unit
private int TileDimension = 32; //Pixel width and height of tile
private int WallTileHeight = 32; //Pixel height of a wall tile
private Vector2Int TilemapSize; //Size of tilemap
private Vector2 TilemapWorldSize; //Size of tilemap in unity units

private const float ColliderMargin = 0.02f; //Margin for colliders
(shrinks collider size by this value on each side)
private CompositeCollider2D CompositeCollider; //Composite collider for
optimizing collisions
private BoxCollider2D[] ColliderMap; //Layout of colliders on tilemap
private List<BoxCollider2D> ColliderList; //List of colliders in use
private Rigidbody2D RB; //Rigidbody of tilemap

public Tilemap(int TileDimension, int WallTileHeight, Vector2Int
TilemapSize, float TilesPerUnit = 1)
{
    this.TileDimension = TileDimension;
    this.WallTileHeight = WallTileHeight;
    this.TilemapSize = TilemapSize;
    this.TilesPerUnit = TilesPerUnit;
    FloorMap = new Tile[TilemapSize.x * TilemapSize.y];
    WallMap = new Tile[TilemapSize.x * TilemapSize.y];
}

```

```

        Initialize();
    }
    public Tilemap(ushort ID) : base(ID)
    {
        TileDimension = 32;
        WallTileHeight = 32;
        TilemapSize = Vector2Int.zero;
        TilesPerUnit = 1;
        FloorMap = new Tile[TilemapSize.x * TilemapSize.y];
        WallMap = new Tile[TilemapSize.x * TilemapSize.y];
        Initialize();
    }
    public Tilemap(int TileDimension, int WallTileHeight, Vector2Int
TilemapSize, Tile[] FloorMap, Tile[] WallMap, float TilesPerUnit = 1)
    {
        this.TileDimension = TileDimension;
        this.WallTileHeight = WallTileHeight;
        this.TilemapSize = TilemapSize;
        this.TilesPerUnit = TilesPerUnit;
        this.FloorMap = FloorMap;
        this.WallMap = WallMap;
        Initialize();
    }
    public Tilemap(ushort ID, int TileDimension, int WallTileHeight,
Vector2Int TilemapSize, Tile[] FloorMap, Tile[] WallMap, float TilesPerUnit
= 1) : base(ID)
    {
        this.TileDimension = TileDimension;
        this.WallTileHeight = WallTileHeight;
        this.TilemapSize = TilemapSize;
        this.TilesPerUnit = TilesPerUnit;
        this.FloorMap = FloorMap;
        this.WallMap = WallMap;
        Initialize();
    }
}

    public Vector2 TilemapToWorldPosition(Vector2Int Position)
    {
        return new Vector2
        (
            Self.transform.position.x - TilemapSize.x / 2f / TilesPerUnit +
Position.x / TilesPerUnit + TilesPerUnit / 2,
            Self.transform.position.y + TilemapSize.y / 2f / TilesPerUnit -
Position.y / TilesPerUnit - TilesPerUnit / 2
        );
    }

    public Vector2Int WorldPositionToTilemap(Vector2 Position)
    {
        return new Vector2Int
        (
            Mathf.RoundToInt(TilesPerUnit * (Position.x -
Self.transform.position.x + TilemapSize.x / 2f / TilesPerUnit -
TilesPerUnit / 2)),
            Mathf.RoundToInt(TilesPerUnit * (Self.transform.position.y +
TilemapSize.y / 2f / TilesPerUnit - Position.y - TilesPerUnit / 2))
        );
    }

    public Tile GetFloorTileAtPosition(Vector2Int Position)
    {

```

```

        return FloorMap[Position.y * TilemapSize.x + Position.x];
    }

    public Tile GetWallTileAtPosition(Vector2Int Position)
    {
        return FloorMap[Position.y * TilemapSize.x + Position.x];
    }

    public static Tile[] TilesFromString(string TilesToParse)
    {
        List<Tile> Tiles = new List<Tile>();

        string Formatted = Regex.Replace(TilesToParse, @"[ \n\r\t]", "");
        string[] TileArray = TilesToParse.Split('/');
        for (int i = 0; i < TileArray.Length; i++)
        {
            string[] Componenets = TileArray[i].Split(',');
            Tiles.Add(new Tile()
            {
                TileIndex = int.Parse(Componenets[0]),
                AnimationFrame = int.Parse(Componenets[1]),
                Blank = int.Parse(Componenets[2]),
                Render = int.Parse(Componenets[3])
            });
        }
        return Tiles.ToArray();
    }

    /// <summary>
    /// Resize tilemap to fit a new floor and wall map
    /// </summary>
    public void Resize(Vector2Int NewTilemapSize, Tile[] FloorMap, Tile[]
WallMap)
    {
        UpdateResizeOverNetwork++;
        this.FloorMap = FloorMap;
        this.WallMap = WallMap;
        bool SizeChange = NewTilemapSize != TilemapSize;
        if (SizeChange) TilemapSize = NewTilemapSize;
        if (DZSettings.ActiveRenderers && SizeChange)
            GenerateRenderers();
        TilemapSize = NewTilemapSize;
        GenerateColliders();
    }

    /// <summary>
    /// Releases unused memory as tilemap caches textures and objects for
reuse
    /// </summary>
    public void ReleaseUnusedResources()
    {
        ColliderList.RemoveAll(I =>
        {
            if (!I.enabled)
            {
                GameObject.Destroy(I);
                return true;
            }
            return false;
        });
        if (Rows != null)

```

```

        if (Rows.Length > TilemapSize.y)
        {
            for (int i = TilemapSize.y; i < Rows.Length; i++)
            {
                GameObject.Destroy(Rows[i].gameObject);
            }
            RawImage[] Temp = Rows;
            Rows = new RawImage[TilemapSize.y];
            System.Buffer.BlockCopy(Temp, 0, Rows, 0, TilemapSize.y);
        }
        Resources.UnloadUnusedAssets();
    }

    /// <summary>
    /// Initializes base tilemap
    /// </summary>
    private void Initialize()
    {
        TilemapWorldSize = (Vector2)TilemapSize / TilesPerUnit;

        ColliderMap = new BoxCollider2D[TilemapSize.x * TilemapSize.y];
        ColliderList = new List<BoxCollider2D>();

        //Initialize GameObject
        Self = new GameObject();
        Self.name = "Tilemap";

        //Initialize Collider objects
        RB = Self.AddComponent<Rigidbody2D>();
        RB.isKinematic = true;

        CompositeCollider = Self.AddComponent<CompositeCollider2D>();
        CompositeCollider.generationType =
CompositeCollider2D.GenerationType.Manual;
    }

    public void InitializeRenderer() { }

    protected override void OnCreate()
    {
        //Initialize colliders
        GenerateColliders();
        //Initialize renders
        GenerateRenders();
    }

    public void ServerUpdate()
    {
    }

    public void Update() { }

    public void BodyPhysicsUpdate() { }

    private Vector3 PrevTilePosition; //Store previous position of tilemap
    private void UpdateRenderSortingLayers() //Updates the render sorting
layer of each tile row
    {
        if (Rows == null) return;
        PrevTilePosition = Self.transform.position;
    }

```



```

        for (int i = 0; i < TilemapSize.y; i++)
        {
            float StrideHeight = ((float)WallTileHeight / TileDimension) /
TilesPerUnit;
            float BaseY = Rows[i].transform.position.y - ((StrideHeight - 1
/ TilesPerUnit) / 2);
            Rows[i].canvas.sortingOrder = Mathf.RoundToInt(-(BaseY * 10)) +
1;
        }
    }

    public void Render()
    {
        if (PrevTilePosition != Self.transform.position) //Avoid updating
all rows constantly has tilemaps can get quite large
            UpdateRenderSortingLayers(); //Update the sorting layers for
each tile row

        //Generate textures to render
        if (WallBuffer == null || FloorBuffer == null || WallCompute ==
null || FloorCompute == null)
            return;
        WallBuffer.SetData(WallMap);
        WallCompute.Dispatch(ComputeKernel, TilemapSize.x / 4 + 1,
TilemapSize.y / 4 + 1, 1);
        FloorBuffer.SetData(FloorMap);
        FloorCompute.Dispatch(ComputeKernel, TilemapSize.x / 4 + 1,
TilemapSize.y / 4 + 1, 1);
    }

    /// <summary>
    /// Assigns new tile pallets to tilemap
    /// </summary>
    public void SetTilePallet()
    {
        WallCompute.SetTexture(ComputeKernel, "TilePallet",
WallTilePalletData.Pallet);
        WallCompute.SetInt("TileStride", WallTilePalletData.TileStride);
        WallCompute.SetInt("TilePalletCount", WallTilePalletData.NumTiles);
        FloorCompute.SetTexture(ComputeKernel, "TilePallet",
FloorTilePalletData.Pallet);
        FloorCompute.SetInt("TileStride", FloorTilePalletData.TileStride);
        FloorCompute.SetInt("TilePalletCount",
FloorTilePalletData.NumTiles);
    }

    /// <summary>
    /// Generates new composite colliders for a tilemap
    /// </summary>
    /// <param name="Truncate">If true, disposes of unused
colliders</param>
    public virtual void GenerateColliders(bool Truncate = false)
    {
        //Expand collider array if needed
        if (ColliderMap.Length < TilemapSize.x * TilemapSize.y)
            ColliderMap = new BoxCollider2D[TilemapSize.x * TilemapSize.y];

        int ReuseIndex = 0; //Index of colliders to reuse
        int FinalReuseIndex = 0; //Indicates the last collider that was
reused

```

```

    int NumCurrentColliders = ColliderList.Count; //Store the number of
current colliders
    for (int i = 0; i < TilemapSize.y; i++)
    {
        for (int j = 0; j < TilemapSize.x; j++)
        {
            int Index = i * TilemapSize.x + j;
            if (WallMap[Index].Blank == 0)
            {
                BoxCollider2D Collider = null;
                if (FinalReuseIndex == 0 && ReuseIndex <
NumCurrentColliders) //If there are colliders to reuse, reuse them
                {
                    Collider = ColliderList[ReuseIndex];
                    ReuseIndex++;
                }
                else //Otherwise create new colliders
                {
                    FinalReuseIndex = ReuseIndex;
                    Collider = Self.AddComponent<BoxCollider2D>();
                    ColliderList.Add(Collider);
                }

                //Position the colliders on the tilemap
                Collider.enabled = true;
                float Dimension = 1 / TilesPerUnit;
                Collider.size = new Vector2(Dimension - ColliderMargin,
Dimension - ColliderMargin);
                Collider.offset = new Vector2(j / TilesPerUnit +
Dimension / 2 - TilemapSize.x / 2f / TilesPerUnit,
-i / TilesPerUnit -
Dimension / 2 + TilemapSize.y / 2f / TilesPerUnit);
                Collider.usedByComposite = true;

                ColliderMap[Index] = Collider;
            }
        }
    }

    if (Truncate) //If true, dispose of unused colliders
    {
        for (; ReuseIndex < NumCurrentColliders; ReuseIndex++)
        {
            GameObject.Destroy(ColliderList[ReuseIndex]);
        }
        ColliderList.RemoveRange(FinalReuseIndex, NumCurrentColliders -
FinalReuseIndex);
    }
    else //Disable unused colliders but cache to reuse
    {
        for (; ReuseIndex < NumCurrentColliders; ReuseIndex++)
        {
            ColliderList[ReuseIndex].enabled = false;
        }
    }

    CompositeCollider.GenerateGeometry();
}

/// <summary>
/// Generates new renders for rendering a different sized tilemap
/// </summary>
/// <param name="Truncate">If true, disposes of unused renders</param>

```

```

private void GenerateRenderers (bool Truncate = false)
{
    if (DZSettings.ActiveRenderers == false) return;

    //Initialize Buffers
    if (WallBuffer != null) WallBuffer.Dispose();
    if (FloorBuffer != null) FloorBuffer.Dispose();

    //Initialize compute shaders
    if (WallCompute == null)
    {
        WallCompute =
UnityEngine.Object.Instantiate (Resources.Load<ComputeShader>("ComputeShader
s/TilemapComputeShader"));
        ComputeKernel = WallCompute.FindKernel ("TilemapRender");
    }
    if (FloorCompute == null)
    {
        FloorCompute =
UnityEngine.Object.Instantiate (Resources.Load<ComputeShader>("ComputeShader
s/TilemapComputeShader"));
        ComputeKernel = WallCompute.FindKernel ("TilemapRender");
    }

    //Set Tile Dimensions
    WallCompute.SetInt ("TileWidth", TileDimension);
    WallCompute.SetInt ("TileHeight", WallTileHeight);
    FloorCompute.SetInt ("TileWidth", TileDimension);
    FloorCompute.SetInt ("TileHeight", TileDimension);
    //Set Map Dimensions
    WallCompute.SetInt ("MapWidth", TilemapSize.x);
    WallCompute.SetInt ("MapHeight", TilemapSize.y);
    FloorCompute.SetInt ("MapWidth", TilemapSize.x);
    FloorCompute.SetInt ("MapHeight", TilemapSize.y);

    //Check TilePallets
    if (WallTilePalletData == null || FloorTilePalletData == null)
    {
        Debug.LogWarning ("WallTilePalletData or FloorTilePalletData is
null, rendering default pallets...");
        GenerateDefaultTileData ();
    }

    //Set TilePallets
    SetTilePallet ();

    //Set the buffers
    if (WallBuffer != null)
        WallBuffer.Dispose ();
    WallBuffer = new ComputeBuffer (WallMap.Length, sizeof (int) * 5);
    if (FloorBuffer != null)
        FloorBuffer.Dispose ();
    FloorBuffer = new ComputeBuffer (FloorMap.Length, sizeof (int) * 5);

    WallCompute.SetBuffer (ComputeKernel, "Map", WallBuffer);
    FloorCompute.SetBuffer (ComputeKernel, "Map", FloorBuffer);

    //Create new textures if required
    if (WallRenderTexture == null)
    {

```

```

        WallRenderTexture = new RenderTexture(TileDimension *
TilemapSize.x, WallTileHeight * TilemapSize.y, 8)
        {
            enableRandomWrite = true,
            filterMode = FilterMode.Point,
            anisoLevel = 1
        };
        WallRenderTexture.Create();
        WallCompute.SetTexture(ComputeKernel, "Result",
WallRenderTexture);
    }
    else if (Truncate || WallRenderTexture.width < TilemapSize.x ||
WallRenderTexture.height < TilemapSize.y) //Truncate textures if option was
true to release more memory
    {
        WallRenderTexture.Release();
        WallRenderTexture.width = TilemapSize.x;
        WallRenderTexture.height = TilemapSize.y;
        WallRenderTexture.Create();
    }

    //Create new textures if required
    if (FloorRenderTexture == null)
    {
        FloorRenderTexture = new RenderTexture(TileDimension *
TilemapSize.x, TileDimension * TilemapSize.y, 8)
        {
            enableRandomWrite = true,
            filterMode = FilterMode.Point,
            anisoLevel = 1
        };
        FloorRenderTexture.Create();
        FloorCompute.SetTexture(ComputeKernel, "Result",
FloorRenderTexture);
    }
    else if (Truncate || FloorRenderTexture.width < TilemapSize.x ||
FloorRenderTexture.height < TilemapSize.y) //Truncate textures if option
was true to release more memory
    {
        FloorRenderTexture.Release();
        FloorRenderTexture.width = TilemapSize.x;
        FloorRenderTexture.height = TilemapSize.y;
        FloorRenderTexture.Create();
    }

    if (FloorRender == null) //Create a new render for the floor map if
needed
    {
        Canvas FloorCanvas = Self.AddComponent<Canvas>();
        FloorCanvas.renderMode = RenderMode.WorldSpace;
        FloorCanvas.sortingLayerName = "Floor";
        Self.AddComponent<CanvasScaler>();
        Self.AddComponent<GraphicRaycaster>();
        Self.GetComponent<RectTransform>().sizeDelta = Vector2.zero;
        FloorRender = Self.AddComponent<RawImage>();
    }
    //Scale the floormap render
    FloorRender.rectTransform.sizeDelta = new
Vector2((float)TilemapSize.x / TilesPerUnit, (float)TilemapSize.y /
TilesPerUnit);

```

```

        FloorRender.material =
Resources.Load<Material>("Materials/LitMaterial");
        FloorRender.texture = FloorRenderTexture;
        FloorRender.uvRect = new Rect(0, 0, (float)TileDimension *
TilemapSize.x / FloorRenderTexture.width, (float)TileDimension *
TilemapSize.y / FloorRenderTexture.height);

//Initialize row gameobjects
int CurrentLength = 0;
if (Rows == null)
    Rows = new RawImage[TilemapSize.y];
else
    CurrentLength = Rows.Length > TilemapSize.y ? TilemapSize.y :
Rows.Length; //Get the number of rows to render for a given tilemap
if (Rows.Length < TilemapSize.y) //Resize array of rows if needed
{
    RawImage[] Temp = Rows;
    Rows = new RawImage[TilemapSize.y];
    System.Buffer.BlockCopy(Temp, 0, Rows, 0, Temp.Length);
}
int i = 0;
for (; i < CurrentLength; i++) //Loop through number of rows that
can be reused
{
    Rows[i].rectTransform.sizeDelta = Vector2.zero;
    float StrideHeight = ((float)WallTileHeight / TileDimension) /
TilesPerUnit;
    Rows[i].rectTransform.position = new Vector3(0, -TilemapSize.y
/ 2f / TilesPerUnit + StrideHeight / 2 + i / TilesPerUnit);
    Rows[i].rectTransform.position += Self.transform.position;
    Rows[i].rectTransform.sizeDelta = new Vector2(TilemapSize.x /
TilesPerUnit, StrideHeight);
    Rows[i].material =
Resources.Load<Material>("Materials/LitMaterial");
    Rows[i].texture = WallRenderTexture;
    float RenderHeight = (float)WallTileHeight * TilemapSize.y /
WallRenderTexture.height;
    Rows[i].uvRect = new Rect(0, i * RenderHeight / TilemapSize.y,
(float)TileDimension * TilemapSize.x / WallRenderTexture.width,
RenderHeight / TilemapSize.y);
}
for (; i < TilemapSize.y; i++) //Loop through remainder of tilemap
and create new rows
{
    GameObject Row = new GameObject();
    Row.transform.parent = Self.transform;
    Canvas RowCanvas = Row.AddComponent<Canvas>();
    RowCanvas.renderMode = RenderMode.WorldSpace;
    RowCanvas.overrideSorting = true;
    RowCanvas.sortingOrder = 0;
    RectTransform RT = Row.GetComponent<RectTransform>();
    RawImage RowImage = Row.AddComponent<RawImage>();
    RowImage.rectTransform.sizeDelta = Vector2.zero;
    float StrideHeight = ((float)WallTileHeight / TileDimension) /
TilesPerUnit;
    RowImage.rectTransform.position = new Vector3(0, -TilemapSize.y
/ 2f + StrideHeight / 2 + i / TilesPerUnit);
    RowImage.rectTransform.position += Self.transform.position;
    RowImage.rectTransform.sizeDelta = new Vector2(TilemapSize.x /
TilesPerUnit, StrideHeight);
}

```

```

        RowImage.material =
Resources.Load<Material>("Materials/LitMaterial");
        RowImage.texture = WallRenderTexture;
        float RenderHeight = (float)WallTileHeight * TilemapSize.y /
WallRenderTexture.height;
        RowImage.uvRect = new Rect(0, i * RenderHeight / TilemapSize.y,
(float)TileDimension * TilemapSize.x / WallRenderTexture.width,
RenderHeight / TilemapSize.y);
        Rows[i] = RowImage;
    }
    for (; i < Rows.Length; i++) //Loop through remaining rows and
destroy them if truncate option was true, otherwise deactivate and cache to
reuse
    {
        if (Truncate)
            GameObject.Destroy(Rows[i].gameObject);
        else
            Rows[i].gameObject.SetActive(false);
    }
    if (Truncate && Rows.Length != TilemapSize.y) //Resize array of
rows to size of tilemap if truncate option is true
    {
        RawImage[] Temp = Rows;
        Rows = new RawImage[TilemapSize.y];
        System.Buffer.BlockCopy(Temp, 0, Rows, 0, TilemapSize.y);
    }

    UpdateRenderSortingLayers();
}

/// <summary>
/// Generate default tilepallets when none are provided
/// </summary>
private void GenerateDefaultTileData()
{
    if (WallTilePalletData == null)
    {
        WallTilePalletData = new TilePallet
        {
            Pallet =
Resources.Load<Texture2D>("TilemapPallets/Default"),
            NumTiles = 2,
            TileStride = 64, //32 + 32
            FrameCount = new int[2] { 2, 2 }
        };
    }
    if (FloorTilePalletData == null)
    {
        FloorTilePalletData = new TilePallet
        {
            Pallet =
Resources.Load<Texture2D>("TilemapPallets/Default"),
            NumTiles = 2,
            TileStride = 64, //32 + 32
            FrameCount = new int[2] { 2, 2 }
        };
    }
}

protected override void OnDelete()
{
}

```

```

//Release buffers
if (WallBuffer != null)
    WallBuffer.Dispose();
if (FloorBuffer != null)
    FloorBuffer.Dispose();
//Delete Objects
if (Rows != null)
{
    for (int i = 0; i < Rows.Length; i++)
    {
        GameObject.Destroy(Rows[i].gameObject);
    }
}
GameObject.Destroy(Self);
}

private List<byte> MapCache = new List<byte>();
private int UpdateResizeOverNetwork = 1;
private int PreviousResizeOverNetwork = 0;
public override byte[] GetBytes()
{
    if (UpdateResizeOverNetwork != PreviousResizeOverNetwork)
    {
        PreviousResizeOverNetwork = UpdateResizeOverNetwork;
        int Volume = TilemapSize.x * TilemapSize.y;
        MapCache.Clear();
        for (int i = 0; i < Volume; i++)
        {
            Tile T = FloorMap[i];
            MapCache.AddRange(BitConverter.GetBytes(T.Blank));
            if (T.Blank == 0)
            {
                MapCache.AddRange(BitConverter.GetBytes(T.Render));
                MapCache.AddRange(BitConverter.GetBytes(T.TileIndex));
            }
        }
        MapCache.AddRange(BitConverter.GetBytes(T.AnimationFrame));
    }
    for (int i = 0; i < Volume; i++)
    {
        Tile T = WallMap[i];
        MapCache.AddRange(BitConverter.GetBytes(T.Blank));
        if (T.Blank == 0)
        {
            MapCache.AddRange(BitConverter.GetBytes(T.Render));
            MapCache.AddRange(BitConverter.GetBytes(T.TileIndex));
        }
    }
    MapCache.AddRange(BitConverter.GetBytes(T.AnimationFrame));
}

List<byte> Data = new List<byte>();
Data.AddRange(BitConverter.GetBytes(UpdateResizeOverNetwork));
Data.AddRange(BitConverter.GetBytes(-1)); //Tilemap pallet
Data.AddRange(BitConverter.GetBytes(Self.transform.position.x));
//Tilemap position
Data.AddRange(BitConverter.GetBytes(Self.transform.position.y));
Data.AddRange(BitConverter.GetBytes(TilemapSize.x));
Data.AddRange(BitConverter.GetBytes(TilemapSize.y));
Data.AddRange(BitConverter.GetBytes(TilesPerUnit));

```

```

        Data.AddRange(BitConverter.GetBytes(WallTileHeight));
        Data.AddRange(BitConverter.GetBytes(MapCache.Count));
        Data.AddRange(MapCache);

        return Data.ToArray();
    }

    public override void ParseBytes(DZNetwork.Packet Data)
    {
        Data D = (Data)ParseBytesToSnapshot(Data);
        ParseSnapshot(D);
    }

    public struct Data
    {
        public int UpdateResizeOverNetwork;
        public int TilePalletIndex;
        public Vector2 Position;
        public Vector2Int TilemapSize;
        public float TilesPerUnit;
        public int WallTileHeight;
        public Tile[] FloorMap;
        public Tile[] WallMap;
    }

    public override object GetSnapshot()
    {
        Data Snapshot = new Data()
        {
            UpdateResizeOverNetwork = UpdateResizeOverNetwork,
            TilePalletIndex = -1,
            Position = Self.transform.position,
            TilemapSize = TilemapSize,
            TilesPerUnit = TilesPerUnit,
            WallTileHeight = WallTileHeight,
            FloorMap = new Tile[FloorMap.Length],
            WallMap = new Tile[WallMap.Length]
        };
        System.Buffer.BlockCopy(FloorMap, 0, Snapshot.FloorMap, 0,
FloorMap.Length);
        System.Buffer.BlockCopy(WallMap, 0, Snapshot.WallMap, 0,
WallMap.Length);
        return Snapshot;
    }

    public static object ParseBytesToSnapshot(DZNetwork.Packet Data)
    {
        Data D = new Data()
        {
            UpdateResizeOverNetwork = Data.ReadInt(),
            TilePalletIndex = Data.ReadInt(),
            Position = new Vector2(Data.ReadFloat(), Data.ReadFloat()),
            TilemapSize = new Vector2Int(Data.ReadInt(), Data.ReadInt()),
            TilesPerUnit = Data.ReadFloat(),
            WallTileHeight = Data.ReadInt()
        };

        int NumBytes = Data.ReadInt();

        int Volume = D.TilemapSize.x * D.TilemapSize.y;
    }

```



```

D.FloorMap = new Tile[Volume];
for (int i = 0; i < Volume; i++)
{
    int Blank = Data.ReadInt();
    D.FloorMap[i].Blank = Blank;
    if (Blank == 1)
        continue;
    D.FloorMap[i].Render = Data.ReadInt();
    D.FloorMap[i].TileIndex = Data.ReadInt();
    D.FloorMap[i].AnimationFrame = Data.ReadInt();
}

D.WallMap = new Tile[Volume];
for (int i = 0; i < Volume; i++)
{
    int Blank = Data.ReadInt();
    D.WallMap[i].Blank = Blank;
    if (Blank == 1)
        continue;
    D.WallMap[i].Render = Data.ReadInt();
    D.WallMap[i].TileIndex = Data.ReadInt();
    D.WallMap[i].AnimationFrame = Data.ReadInt();
}

return D;
}

public override void ParseSnapshot(object ObjectData)
{
    Data Data = (Data)ObjectData;
    UpdateResizeOverNetwork = Data.UpdateResizeOverNetwork;

    int TilePalletIndex = Data.TilePalletIndex;
    if (TilePalletIndex == -1)
        GenerateDefaultTileData();

    Self.transform.position = Data.Position;
    TilesPerUnit = Data.TilesPerUnit;
    WallTileHeight = Data.WallTileHeight;

    if (PreviousResizeOverNetwork != UpdateResizeOverNetwork)
    {
        PreviousResizeOverNetwork = UpdateResizeOverNetwork;
        Resize(Data.TilemapSize, Data.FloorMap, Data.WallMap);
    }
}

public override void Interpolate(object FromData, object ToData, float
Time)
{
    return;
}
}

```

---

## *Assets/World/AbstractWorld.cs*

---

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

using UnityEngine;
using DeadZoneEngine.Entities;

public class AbstractWorld : MonoBehaviour
{
    [SerializeField]
    public DZSettings.EntityType Type;

    public object Self;

    public object Context;
}
```

---

## *Assets/DZScript.cs*

---

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

using DZNetwork;
using DeadZoneEngine;
using DeadZoneEngine.Entities;
using DeadZoneEngine.Controllers;

//Unity Script to update DZEngine
public class DZScript : MonoBehaviour
{
    public void Start()
    {
        ServerHandler.Start();
        DZEngine.Initialize();
        Main.Start();
    }

    public void FixedUpdate()
    {
        ServerHandle.FixedUpdate();
        DZEngine.FixedUpdate();
        Game.FixedUpdate();
        Main.FixedUpdate();
    }

    private void OnApplicationQuit()
    {
    }
}
```

```

    {
        DZEngine.ReleaseResources ();
    }
}

```

---

### *Client/Assets/DZSettings.cs*

---

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

public static class DZSettings
{
    public enum EntityType
    {
        Null,
        PlayerCreature,
        Tilemap,
        TriggerPlate,
        BulletEntity,
        EnemyCreature,
        Turret,
        CoinEntity,
        CrystalEntity
    }

    public static int NumPhysicsIterations = 10;
    public static bool ActiveRenderers = true;
    public static bool ActiveControllers = true;
    public static bool ClientSidePrediction = true;
    public static bool Client = true;
}

```

---

### *Client/Assets/Game.cs*

---

```

using ClientHandle;
using DeadZoneEngine;
using DeadZoneEngine.Controllers;
using DeadZoneEngine.Entities;
using DZNetwork;
using System;
using System.Collections.Generic;
using System.Linq;
using UnityEngine;

public class Game
{
    public static Client Client;
}

```

```

    public static DZEngine.ManagedList<IServerSendable> ServerItems = new
DZEngine.ManagedList<IServerSendable>();
    public static int ServerTickRate = 60;
    public static int ClientTickRate = 60;
    public static ulong ClientTicks = 0;
    public static ulong ClientTickAsServerTick = 0;
    public static bool Initialized = false;

    public class ServerSnapshot
    {
        public struct Object
        {
            public DZSettings.EntityType Type;
            public bool FlaggedToDelete;
            public object Data;
        }

        public ulong ServerTick;
        public Dictionary<ushort, Object> Data = new Dictionary<ushort,
Object>();
    }
    public static ulong InterpolationRatio = 2;
    public static ulong TargetInterpolationRatio = 2;
    public static ulong LowerInterpolationRatioMargin = 1;
    public static ulong UpperInterpolationRatioMargin = 5;
    public static ulong IterpolationCap = 30;
    private static JitterBuffer<ServerSnapshot> Histogram = new
JitterBuffer<ServerSnapshot>();
    private static ServerSnapshot CurrentLoaded = null;

    public static void FixedUpdate ()
    {
        Loader.Socket.FixedUpdate ();

        float ServerToClientTick = (float)ClientTickRate / ServerTickRate;
        float ClientToServerTick = (float)ServerTickRate / ClientTickRate;
        ClientTickAsServerTick = (ulong) (ClientTicks * ClientToServerTick);

        if (Initialized == false && Histogram.Count > 1)
        {
            ulong NumServerTicksPassed = Histogram.Last.ServerTick -
Histogram.First.ServerTick;
            if (NumServerTicksPassed >= InterpolationRatio)
            {
                ClientTicks = (ulong) ((Histogram.Last.ServerTick -
InterpolationRatio) * ServerToClientTick);
                Initialized = true;
            }
        }

        DZEngine.FixedUpdate ();

        SendData ();

        Interp = 0;
        ClientTicks++;
    }

    private static float Interp = 0;
    public static void Update ()

```

```

{
    float ServerToClientTick = (float)ClientTickRate / ServerTickRate;
    float ClientToServerTick = (float)ServerTickRate / ClientTickRate;

    if (Initialized == true)
    {
        ServerSnapshot From = null;
        ServerSnapshot To = null;
        Histogram.Iterate(S =>
        {
            if (S.Value.ServerTick >= ClientTickAsServerTick)
            {
                From = S.Value;
                if (S.Next != null)
                    To = S.Next.Value;
            }
        }, S => S.Value.ServerTick >= ClientTickAsServerTick);
        if (From != null)
        {
            Histogram.Dequeue(From);
            if (To != null)
            {
                if (CurrentLoaded != From)
                {
                    LoadSnapshot(From, false);
                    CurrentLoaded = From;

                    int LargestHistogram = 0;
                    for (int i = 0; i < Client.Players.Length; i++)
                    {
                        if (Client.Players[i] != null)
                        {
                            Client.Players[i].Entity.StartClientPrediction(From);
                            if
                                (Client.Players[i].Entity.Histogram.Count > LargestHistogram)
                                    LargestHistogram =
                                Client.Players[i].Entity.Histogram.Count;
                        }
                    }
                    for (int j = 0; j < LargestHistogram; j++)
                    {
                        for (int i = 0; i < Client.Players.Length; i++)
                        {
                            if (Client.Players[i] != null)
                            {
                                Client.Players[i].Entity.ClientPrediction();
                            }
                        }
                    }
                    DZEngine.PhysicsUpdate();
                }
                for (int i = 0; i < Client.Players.Length; i++)
                {
                    if (Client.Players[i] != null)
                    {
                        Client.Players[i].Entity.EndClientPrediction();
                    }
                }
            }
        }
    }
}

```

```

    }
    float FromTick = (From.ServerTick *
ServerToClientTick);
    float Origin = (ClientTicks - FromTick);
    float Time = Origin / ((To.ServerTick *
ServerToClientTick) - FromTick) + Interp;
    Interpolate(From, To, Time);
    }
    else
    {
        LoadSnapshot(From, true);
        float Time = (ClientTicks - From.ServerTick *
ServerToClientTick) / ClientTickRate;
        Extrapolate(From, Time);
    }
    }
    else
    {
        Histogram.Dequeue(Histogram.Last);
    }

    if ((ulong)Histogram.Count - 1 < TargetInterpolationRatio -
LowerInterpolationRatioMargin)
    {
        InterpolationRatio += 1;
        ClientTicks = (ulong)((Histogram.Last.ServerTick -
InterpolationRatio) * ServerToClientTick);
    }
    else if ((ulong)Histogram.Count - 1 > TargetInterpolationRatio
+ UpperInterpolationRatioMargin)
    {
        if (InterpolationRatio != 0)
            InterpolationRatio -= 1;
        ClientTicks = (ulong)((Histogram.Last.ServerTick -
InterpolationRatio) * ServerToClientTick);
    }
    if (InterpolationRatio > IterpolationCap) InterpolationRatio =
IterpolationCap;

    Interp += Time.deltaTime;
    }
    }

private static void Interpolate(ServerSnapshot From, ServerSnapshot To,
float Time)
{
    List<ushort> IDs = To.Data.Keys.ToList();
    foreach (ushort ID in IDs)
    {
        if (!From.Data.ContainsKey(ID))
            continue;

        _IInstantiatableDeletable Item = EntityID.GetObject(ID);
        ServerSnapshot.Object ToData = To.Data[ID];
        ServerSnapshot.Object FromData = From.Data[ID];

        IServerSendable ServerItem = Item as IServerSendable;
        DZSettings.EntityType ToType = ToData.Type;
        DZSettings.EntityType FromType = FromData.Type;

```

```

        if (ToData.Data == null || FromData.Data == null || ToType !=
FromType || (int)ToType != ServerItem.ServerObjectType || (int)FromType !=
ServerItem.ServerObjectType)
            continue;

        if (ServerItem == null)
            continue;

        ServerItem.Interpolate(FromData.Data, ToData.Data, Time);
    }
}

private static void Extrapolate(ServerSnapshot From, float Time)
{
    List<ushort> IDs = From.Data.Keys.ToList();
    foreach (ushort ID in IDs)
    {
        if (!From.Data.ContainsKey(ID))
            continue;

        _IInstantiatableDeletable Item = EntityID.GetObject(ID);
        ServerSnapshot.Object FromData = From.Data[ID];

        IServerSendable ServerItem = Item as IServerSendable;
        DZSettings.EntityType FromType = FromData.Type;
        if (FromData.Data == null || (int)FromType !=
ServerItem.ServerObjectType)
            continue;

        if (ServerItem == null)
            continue;

        ServerItem.Extrapolate(FromData.Data, Time);
    }
}

public static void SendData()
{
    if (Loader.Socket.SocketConnected)
        SendSnapshot();

    if (!Loader.Socket.Connected)
        return;
}

private static void SendSnapshot()
{
    Packet Setup = new Packet();
    Setup.Write(Client.NumPlayers);
    Loader.Socket.Send(Setup, ServerCode.SyncPlayers);

    Packet SnapshotPacket = new Packet();
    SnapshotPacket.Write(Client.NumPlayers);
    SnapshotPacket.Write(InputMapping.GetBytes());
    Loader.Socket.Send(SnapshotPacket, ServerCode.ClientSnapshot);
}

public static void SyncClient(DZUDPSocket.RecievePacketWrapper Packet)
{
    int NumPlayers = Packet.Data.ReadByte();
    if (NumPlayers != Client.NumPlayers)

```

```

    {
        Debug.LogWarning("Sync failed due to inconsistent player
count");
        return;
    }
    ushort CID = Packet.Data.ReadUShort();
    if (Client.ID != CID)
        Client.ID.ChangeID(CID);
    for (int i = 0; i < NumPlayers; i++)
    {
        int ID = Packet.Data.ReadByte();
        if (ID == byte.MaxValue)
            continue;
        ushort PlayerEntityID = Packet.Data.ReadUShort();
        if (Client.Players[i].Entity.ID != PlayerEntityID)
            Client.Players[i].Entity.ID.ChangeID(PlayerEntityID, true);
    }
}

public static void UnWrapSnapshot (DZUDPSocket.RecievePacketWrapper
Packet)
{
    if (Client == null) return;

    int CheckSum = Packet.Data.ReadInt();
    ServerTickRate = Packet.Data.ReadInt();
    ulong ServerTick = Packet.Data.ReadULong();
    Main.LifeForce[0] = Packet.Data.ReadInt();
    Main.LifeForce[1] = Packet.Data.ReadInt();
    Main.LifeForce[2] = Packet.Data.ReadInt();
    Main.Money = Packet.Data.ReadInt();
    if (ServerTick <= (Histogram.Count == 0 ? 0 :
Histogram.Last.ServerTick))
    {
        Debug.LogWarning("Received a late packet");
        return;
    }

    ServerSnapshot Snapshot = new ServerSnapshot();
    Snapshot.ServerTick = ServerTick;

    int NumSnapshotItems = Packet.Data.ReadInt();
    for (int i = 0; i < NumSnapshotItems; i++)
    {
        ServerSnapshot.Object Object = new ServerSnapshot.Object();

        ushort ID = Packet.Data.ReadUShort();
        bool FlaggedToDelete = Packet.Data.ReadBool();
        Object.FlaggedToDelete = FlaggedToDelete;
        if (FlaggedToDelete)
        {
            Snapshot.Data.Add(ID, Object);
            continue;
        }

        DZSettings.EntityType Type =
(DZSettings.EntityType)Packet.Data.ReadInt();
        Object.Type = Type;

        object ServerItem = Parse(Type, Packet.Data);
        if (ServerItem == null)

```



```

        {
            Debug.LogWarning("Unable to Parse item from server
snapshot");
            return;
        }
        Object.Data = ServerItem;

        Snapshot.Data.Add(ID, Object);
    }

    Histogram.Add(Snapshot);
}

private static void LoadSnapshot(ServerSnapshot Snapshot, bool
ParseData = true)
{
    List<ushort> IDs = Snapshot.Data.Keys.ToList();
    foreach (ushort ID in IDs)
    {
        _IInstantiatableDeletable Item = EntityID.GetObject(ID);
        ServerSnapshot.Object Object = Snapshot.Data[ID];
        bool FlaggedToDelete = Object.FlaggedToDelete;
        if (FlaggedToDelete)
        {
            if (Item != null)
                DZEngine.Destroy(Item);
            continue;
        }

        IServerSendable ServerItem = Item as IServerSendable;
        DZSettings.EntityType Type = Object.Type;

        if (ServerItem == null)
            ServerItem = Parse(ID, Type);
        if (ServerItem == null)
        {
            Debug.LogWarning("Unable to Parse item from server
snapshot");
            return;
        }

        if ((DZSettings.EntityType)ServerItem.ServerObjectType != Type)
        {
            Debug.LogWarning("Entity Types of ID " + ID + " do not
match... (ServerID = " + Type + ", LocalID = " +
(DZSettings.EntityType)ServerItem.ServerObjectType + ") resetting IDs and
re-parsing");
            ServerItem.ID.ChangeID();

            Item = EntityID.GetObject(ID);
            ServerItem = Item as IServerSendable;
            if (ServerItem == null)
                ServerItem = Parse(ID, Type);
            if (ServerItem == null)
            {
                Debug.LogWarning("Unable to Parse item from server
snapshot");
                return;
            }
        }
    }
    if (ParseData)

```

```

        ServerItem.ParseSnapshot (Object.Data);
        ServerItem.RecentlyUpdated = true;
    }
    foreach (IServerSendable Item in ServerItems)
    {
        if (!Item.ProtectedDeletion && !Item.RecentlyUpdated)
        {
            DZEngine.Destroy (Item);
            Debug.LogWarning ("An Item was destroyed as it was not
updated by the server");
        }
    }
}

private static object Parse (DZSettings.EntityType Type, Packet Data)
{
    switch (Type)
    {
        case DZSettings.EntityType.PlayerCreature: return
PlayerCreature.ParseBytesToSnapshot (Data);
        case DZSettings.EntityType.Tilemap: return
Tilemap.ParseBytesToSnapshot (Data);
        case DZSettings.EntityType.TriggerPlate: return
TriggerPlate.ParseBytesToSnapshot (Data);
        case DZSettings.EntityType.BulletEntity: return
BulletEntity.ParseBytesToSnapshot (Data);
        case DZSettings.EntityType.EnemyCreature: return
EnemyCreature.ParseBytesToSnapshot (Data);
        case DZSettings.EntityType.Turret: return
Turret.ParseBytesToSnapshot (Data);
        case DZSettings.EntityType.CoinEntity: return
CoinEntity.ParseBytesToSnapshot (Data);
        case DZSettings.EntityType.CrystalEntity: return
CrystalEntity.ParseBytesToSnapshot (Data);
        case DZSettings.EntityType.Null: return null;
        default: Debug.LogWarning ("Parsing unknown entity type");
    }
    return null;
}

private static IServerSendable Parse (ushort ID, DZSettings.EntityType
Type)
{
    switch (Type)
    {
        case DZSettings.EntityType.PlayerCreature: return new
PlayerCreature (ID);
        case DZSettings.EntityType.Tilemap: return new Tilemap (ID);
        case DZSettings.EntityType.TriggerPlate: return new
TriggerPlate (ID);
        case DZSettings.EntityType.BulletEntity: return new
BulletEntity (ID);
        case DZSettings.EntityType.EnemyCreature: return new
EnemyCreature (ID);
        case DZSettings.EntityType.Turret: return new Turret (ID);
        case DZSettings.EntityType.CoinEntity: return new
CoinEntity (ID);
        case DZSettings.EntityType.CrystalEntity: return new
CrystalEntity (ID);
        case DZSettings.EntityType.Null: return null;
    }
}

```

```

        default: Debug.LogWarning("Parsing unknown entity type");
return null;
    }
}

public static void Connected()
{
    Debug.Log("Client Connected");
}

public static void Disconnected()
{
    Debug.Log("Client Disconnected");
    Initialized = false;
    Histogram.Clear();
}
}

```

---

### *Client/Assets/InputManager.cs*

---

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

using UnityEngine.InputSystem;
using UnityEngine;

using DeadZoneEngine.Controllers;
using ClientHandle;
using static DeadZoneEngine.Controllers.InputMapping;
using DZNetwork;

public class PlayerController : Controller
{
    public Player Owner;
    public PlayerCreature.Control PlayerControl;

    public PlayerController(Player Owner, PlayerCreature.Control
PlayerControl) : base()
    {
        this.Owner = Owner;
        this.PlayerControl = PlayerControl;
    }
    private InputAction Interact;
    public PlayerController(InputDevice Device, DeviceController DC) :
base(Device, DC)
    {
        InputAction Movement = ActionMap.AddAction("Movement",
InputActionType.PassThrough);
        Interact = ActionMap.AddAction("Interact");
        if (Device is Keyboard)
        {
            Movement.AddCompositeBinding("2DVector(mode=2)")
                .With("Up", Device.path + "/w")

```

```

        .With("Down", Device.path + "/s")
        .With("Left", Device.path + "/a")
        .With("Right", Device.path + "/d");
    Interact.AddBinding(Device.path + "/space");
}
else
{
    Movement.AddCompositeBinding("2DVector(mode=2)")
        .With("Up", Device.path + "/stick/up")
        .With("Down", Device.path + "/stick/down")
        .With("Left", Device.path + "/stick/left")
        .With("Right", Device.path + "/stick/right");
    Interact.AddBinding(Device.path + "/trigger");
}
Movement.performed += MoveAction;
}

protected override void SetType()
{
    Type = ControllerType.PlayerController;
}

public override void
OnInput(UnityEngine.InputSystem.Controls.ButtonControl Control)
{
    if (Owner != null || (IsKeyboard && Control.name != "enter"))
        return;
    Player P = Game.Client.AddPlayer();
    P.Controller = this;
    DC.Enable();
}

private Vector2 MovementDirection;
public void MoveAction(InputAction.CallbackContext Context)
{
    MovementDirection = Context.ReadValue<Vector2>();
}

public override void Tick()
{
    if (PlayerControl == null) return;
    PlayerControl.MovementDirection = MovementDirection;
    PlayerControl.Interact = Interact.ReadValue<float>();
}

public override void ParseBytes(Packet Data)
{
    PlayerControl.InputID = Data.ReadULong();
    PlayerControl.Interact = Data.ReadFloat();
    PlayerControl.MovementDirection = new Vector2(Data.ReadFloat(),
Data.ReadFloat());
}
public override byte[] GetBytes()
{
    List<byte> Data = new List<byte>();
    Data.Add(Owner.ID);
    Data.AddRange(BitConverter.GetBytes(PlayerControl.InputID));
    Data.AddRange(BitConverter.GetBytes(PlayerControl.Interact));

    Data.AddRange(BitConverter.GetBytes(PlayerControl.MovementDirection.x));
}

```

```

Data.AddRange (BitConverter.GetBytes (PlayerControl.MovementDirection.y));
    return Data.ToArray ();
}
}

public static class InputManager
{
    public static void Initialize ()
    {
        InputMapping.OnDeviceAdd += OnDeviceAdd;
        InputMapping.OnDeviceDisconnect += OnDeviceDisconnect;
        InputMapping.OnDeviceReconnect += OnDeviceReconnect;
        InputMapping.OnDeviceRemove += OnDeviceRemove;
    }

    public static void OnDeviceAdd (InputDevice Device)
    {
        InputMapping.Devices [Device].Controllers.Add (new
PlayerController (Device, InputMapping.Devices [Device]));
    }

    public static void OnDeviceDisconnect (InputDevice Device)
    {
    }

    public static void OnDeviceReconnect (InputDevice Device)
    {
    }

    public static void OnDeviceRemove (InputDevice Device)
    {
    }
}

```

---

### *Client/Assets/Loader.cs*

---

```

using System.Collections;
using System.Collections.Generic;
using System.Net;
using UnityEngine;
using UnityEngine.UI;

using DZNetwork;
using System;

public class Loader
{
    public static DZClient Socket = new DZClient ();

    public static string ServerIP = "192.168.2.26";
//"172.16.6.165"; //"192.168.2.51"; //"192.168.2.26"; //"172.16.6.165";
    public static int ServerPort = 26950;

    public static InputField IPAddressField;
    public static Text StatusText;
}

```

```

[RuntimeInitializeOnLoadMethod]
private static void Start()
{
    Physics2D.queriesStartInColliders = false;
    Application.quitting += Dispose;
    Time.fixedDeltaTime = 1f / Game.ClientTickRate;
    Physics2D.simulationMode = SimulationMode2D.Script;

    IPAddressField =
GameObject.FindGameObjectWithTag("IPInput").GetComponent<InputField>();
    IPAddressField.text = ServerIP + ":" + ServerPort;
    IPAddressField.onValueChanged.AddListener(delegate { Connect(); });
    StatusText =
GameObject.FindGameObjectWithTag("StatusBox").GetComponent<Text>();
    StatusText.text = "";

    //Remove later
    Socket.ConnectHandle += Game.Connected;
    Socket.DisconnectHandle += Game.Disconnected;
    Socket.PacketHandle += ServerHandle.ProcessPacket;
    Socket.PacketLostHandle += ServerHandle.HandleLostPacket;
    Socket.Connect(ServerIP, ServerPort);
}

private static void Connect()
{
    string[] IPPort = IPAddressField.text.Split(':');
    if (IPPort.Length != 2)
    {
        StatusText.text = "Please enter IP in the correct format";
    }
    try
    {
        Socket.Connect(IPPort[0], int.Parse(IPPort[1]));
    }
    catch (Exception E)
    {
        StatusText.text = "Failed to parse IP";
    }
}

private static void Dispose()
{
    Socket.Dispose();
}
}

```









```

        C.Players[i].Entity.Position = new
Vector2(UnityEngine.Random.Range(-8f, 8f), UnityEngine.Random.Range(-8f,
8f));
        C.Players[i].Entity.Out = false;
    }
}

if (Tilemap == null)
{
    Tilemap = (Tilemap)new Tilemap(32, 64, new Vector2Int(20, 20),
Tilemap.TilesFromString(MenuFloorMap),
Tilemap.TilesFromString(MenuWallMap)).Create();
}
else
{
    Tilemap.Resize(new Vector2Int(20, 20),
Tilemap.TilesFromString(MenuFloorMap),
Tilemap.TilesFromString(MenuWallMap));
    Tilemap.ReleaseUnusedResources();
}

StartPlate = new TriggerPlate(new Vector2(4, 2), new Vector2(0, -
3));
StartPlate.OnTrigger = StartGame;
}

private static EnemyCreature.Path GeneratePath(int Height, int Width)
{
    List<EnemyCreature.WayPoint> Path = new
List<EnemyCreature.WayPoint>();

    int NumTurns = UnityEngine.Random.Range(6, 10);
    string[] Tiles = null;
    bool ValidPath = false;
    while (!ValidPath)
    {
        Path.Clear();
        Tiles = MenuFloorMap.Split('/');
        Vector2Int StartPosition = new Vector2Int(1,
UnityEngine.Random.Range(1, Height - 1));
        float Chance = UnityEngine.Random.Range(0f, 1f);
        if (Chance > 0.25)
            StartPosition = new Vector2Int(Width - 2,
UnityEngine.Random.Range(1, Height - 1));
        else if (Chance > 0.5)
            StartPosition = new Vector2Int(UnityEngine.Random.Range(1,
Width - 1), 1);
        else if (Chance > 0.75)
            StartPosition = new Vector2Int(UnityEngine.Random.Range(1,
Width - 1), Height - 2);

        int Direction = 1;
        if (Chance > 0.25)
            Direction = 3;
        else if (Chance > 0.5)
            Direction = 0;
        else if (Chance > 0.75)
            Direction = 2;

        Path.Add(new EnemyCreature.WayPoint()
    {

```

```

        Direction = Direction,
        Position = StartPosition
    });

    for (int i = 0; i < NumTurns; i++)
    {
        int CurrentDirection = Direction;
        int Length = UnityEngine.Random.Range(3, 8);
        if (i == NumTurns - 1)
        {
            switch (Direction)
            {
                case 0: Length = Height - StartPosition.y; break;
                case 1: Length = Width - StartPosition.x; break;
                case 2: Length = StartPosition.y; break;
                case 3: Length = StartPosition.x; break;
            }
        }
        for (int j = 0; j < Length; j++)
        {
            Tiles[StartPosition.y * Width + StartPosition.x] =
"0,2,0,1";

            Vector2Int NewPosition = StartPosition;
            switch (Direction)
            {
                case 0: NewPosition.y++; break;
                case 1: NewPosition.x++; break;
                case 2: NewPosition.y--; break;
                case 3: NewPosition.x--; break;
            }
            if (NewPosition.x < 1 || NewPosition.x > Width - 2 ||
NewPosition.y < 1 || (NewPosition.y > Height - 3 && Direction == 0))
            {
                break;
            }
            StartPosition = NewPosition;
        }
        bool ValidDirection = true;
        do
        {
            Direction = (Direction + (UnityEngine.Random.Range(0f,
1f) > 0.5 ? 1 : -1)) % 4;
            if (Direction < 0) Direction += 4;
            Vector2Int NewPosition = StartPosition;
            switch (Direction)
            {
                case 0: NewPosition.y++; break;
                case 1: NewPosition.x++; break;
                case 2: NewPosition.y--; break;
                case 3: NewPosition.x--; break;
            }
            if (NewPosition.x < 1 || NewPosition.x > Width - 1 ||
NewPosition.y < 1 || NewPosition.y > Height - 2)
            {
                ValidDirection = false;
            }
            else if (Tiles[NewPosition.y * Width + NewPosition.x]
== "0,2,0,1")
            {
                ValidDirection = false;
            }
        }
    }
}

```

```

    }
    while (CurrentDirection == Direction && !ValidDirection);
    Path.Add(new EnemyCreature.WayPoint()
    {
        Direction = CurrentDirection,
        Position = StartPosition
    });
}

bool TopLeftQuadrant = false;
bool TopRightQuadrant = false;
bool BottomLeftQuadrant = false;
bool BottomRightQuadrant = false;
for (int i = 0; i < Path.Count; i++)
{
    if (Path[i].Position.x < Width / 2)
    {
        if (Path[i].Position.y < Height / 2)
        {
            BottomLeftQuadrant = true;
        }
        else
        {
            TopLeftQuadrant = true;
        }
    }
    else
    {
        if (Path[i].Position.y < Height / 2)
        {
            BottomRightQuadrant = true;
        }
        else
        {
            TopRightQuadrant = true;
        }
    }
}

ValidPath = TopLeftQuadrant && TopRightQuadrant &&
BottomLeftQuadrant && BottomRightQuadrant;
}

return new EnemyCreature.Path()
{
    Map = string.Join("/", Tiles),
    Traversal = Path
};
}

private static EnemyCreature.Path CurrentPath;
public static void StartGame()
{
    CurrentPath = GeneratePath(20, 20);
    Tilemap.Resize(new Vector2Int(20, 20),
    Tilemap.TilesFromString(CurrentPath.Map),
    Tilemap.TilesFromString(MenuWallMap));

    GameStarted = true;
    DZEngine.Destroy(StartPlate);
}

```

```

public static int[] LifeForce = new int[3] { 10, 10, 10 };
private static float WaveTimer = 5;
private static int WaveSize = 10;
private static int WaveMaxSize = 10;
private static int WaveHealth = 1;
private static int Wave = 0;
private static float WaveSpacing = 0.3f;
private static float WaveSpacingMax = 1;
private static int EnemiesToSpawn = 5;
private static float SpawnTimer = 0;
public static int Money = 40;
public static float Drain = 0;
private static List<EnemyCreature> Enemies = new List<EnemyCreature>();
public static List<Turret> Towers = new List<Turret>();

public static void TakeLifeForce ()
{
    for (int i = 0; i < LifeForce.Length; i++)
    {
        if (LifeForce[i] > 0)
        {
            LifeForce[i]--;
            break;
        }
    }
}

public static void GainLifeForce(int Health)
{
    for (int i = 0; i < LifeForce.Length; i++)
    {
        if (LifeForce[i] != 0)
        {
            LifeForce[i] += Health;
            if (LifeForce[i] > 10)
                LifeForce[i] = 10;
            break;
        }
    }
}

private static void Reset ()
{
    Money = 40;
    for (int i = 0; i < Enemies.Count; i++)
    {
        DZEngine.Destroy(Enemies[i]);
    }
    Enemies.Clear();
    for (int i = 0; i < Towers.Count; i++)
    {
        DZEngine.Destroy(Towers[i]);
    }
    Towers.Clear();
    LifeForce = new int[3] { 10, 10, 10 };
    GameStarted = false;
    LoadMenu();
}

// Update is called once per frame
public static void FixedUpdate ()

```

```

{
    if (DZSettings.ActiveRenderers == true)
        foreach (IRenderer<SpriteRenderer> Renderer in SpriteRenderers)
        {
            if (Renderer.SortingLayer == (int)SortingLayers.Default)
                Renderer.RenderObject.sortingOrder = Mathf.RoundToInt(-
Renderer.RenderObject.transform.parent.position.y * 10);
        }

    if (GameStarted)
    {
        if (Drain <= 0)
        {
            Drain = 5;
            TakeLifeForce();
        }
        else Drain -= Time.fixedDeltaTime;

        List<Client> Clients =
ClientID.ConnectedClients.Values.ToList();
        int TotalNumPlayers = 0;
        foreach (Client C in Clients)
        {
            if (C == null) continue;
            if (C.Players == null) continue;
            for (int i = 0; i < C.Players.Length; i++)
            {
                if (C.Players[i] == null) continue;
                if (C.Players[i].Entity == null) continue;
                TotalNumPlayers++;
            }
        }

        if (TotalNumPlayers == 0)
        {
            Reset();
        }

        bool Alive = false;
        for (int i = 0; i < LifeForce.Length; i++)
        {
            if (LifeForce[i] > 0)
                Alive = true;
        }

        if (WaveTimer > 0)
        {
            WaveTimer -= Time.fixedDeltaTime;
        }
        else
        {
            WaveTimer = Random.Range(4f, 10f);
            Wave++;
            EnemiesToSpawn = Random.Range(10, WaveMaxSize);
            WaveHealth++;
            WaveSpacing = Random.Range(0.1f, WaveSpacingMax);
            WaveSpacingMax += Random.Range(-0.5f, 0.5f);
            if (Random.Range(0f, 1f) < 0.3)
            {
                WaveSpacing = 0.3f;
            }
        }
    }
}

```



```

};

ServerHandle.LostPacketHandle = (SentPacketWrapper) =>
{
    HandleLostPacket (SentPacketWrapper.Code);
};
}

private static void HandleLostPacket (ServerCode Job)
{
    //If socket is not connected and packets are lost well.. theres a
    good reason why packets are lost
    if (!Loader.Socket.Connected)
        return;

    switch (Job)
    {
        case ServerCode.Null:
            break;

        default:
            Debug.LogWarning ("Unknown ServerCode: " + Job);
            break;
    }
}

private static void
PerformServerAction (DZUDPSocket.RecievePacketWrapper Packet, ServerCode
Job)
{
    switch (Job)
    {
        case ServerCode.SyncPlayers:
            Game.SyncClient (Packet);
            break;
        case ServerCode.ServerSnapshot:
            Game.UnWrapSnapshot (Packet);
            break;
        default:
            Debug.LogWarning ("Unknown ServerCode: " + Job);
            break;
    }
}
}
}

```

---

### *Server/Assets/Scripts/Creatures/BulletEntity.cs*

---

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

using UnityEngine;
using DeadZoneEngine.Entities;
using DeadZoneEngine.Entities.Components;
using DZNetwork;

```



```

using ClientHandle;
using DeadZoneEngine;

public class BulletEntity : AbstractWorldEntity, IPhysicsUpdatable,
IRenderer, IServerSendable
{
    public int SortingLayer { get; set; }
    public int ServerObjectType { get; set; } =
(int)DZSettings.EntityType.BulletEntity;
    public bool RecentlyUpdated { get; set; } = false;
    public bool ProtectedDeletion { get; set; } = false;

    BodyChunk Bolt;
    public float Speed = 5;
    public int NumBounces = 10;
    public Vector2 Direction = Vector2.up;

    public BulletEntity(ushort ID) : base(ID)
    {
        Init();
    }

    public BulletEntity() : base()
    {
        Init();
    }

    public void Init()
    {
        Bolt = new BodyChunk();
        Bolt.Context = this;
        Bolt.ContextType = DZSettings.EntityType.BulletEntity;
        Bolt.Collider.radius = 0.1f;
        Bolt.Kinematic = true;
    }

    public Vector2 Position
    {
        get
        {
            if (Bolt != null)
                return Bolt.Position;
            return Vector2.zero;
        }
        set
        {
            if (Bolt != null)
                Bolt.Position = value;
        }
    }

    public void InitializeRenderer()
    {
    }

    public void Render()
    {
        Bolt.RenderObject.transform.localScale = new Vector2(0.2f, 0.2f);
        Bolt.RenderObject.color = Color.red;
    }
}

```

```

public void ServerUpdate ()
{

}

private RaycastHit2D[] RayCasts = new RaycastHit2D[6];
public void FixedUpdate ()
{
    float ScaledSpeed = Speed * Time.fixedDeltaTime;
    Vector2 NormalDirection =
Vector2.Perpendicular(Direction).normalized * (Bolt.Collider.radius +
0.01f);
    RayCasts[0] = Physics2D.Raycast (Bolt.Position, Direction,
ScaledSpeed);
    RayCasts[1] = Physics2D.Raycast (Bolt.Position + NormalDirection,
Direction, ScaledSpeed);
    RayCasts[2] = Physics2D.Raycast (Bolt.Position - NormalDirection,
Direction, ScaledSpeed);
    Vector2 End = Bolt.Position + Direction * ScaledSpeed;
    RayCasts[3] = Physics2D.Raycast (End, -Direction, ScaledSpeed -
Bolt.Collider.radius - 0.1f);
    RayCasts[4] = Physics2D.Raycast (End + NormalDirection, -Direction,
ScaledSpeed);
    RayCasts[5] = Physics2D.Raycast (End - NormalDirection, -Direction,
ScaledSpeed);
    bool FoundHit = false;
    int Index = 0;
    for (int i = 0; i < RayCasts.Length; i++)
    {
        if (RayCasts[i].collider != null &&
Vector2.Dot (RayCasts[i].normal, Direction) < 0)
        {
            FoundHit = true;
            Index = i;
            break;
        }
    }
    if (FoundHit)
    {
        if (Main.GameStarted)
        {
            RaycastHit2D Hit = RayCasts[Index];
            float Distance = Mathf.Abs ((Hit.point -
Bolt.Position).magnitude) - Bolt.Collider.radius;
            Vector2 NewPosition = Bolt.Position + Direction * Distance;
            AbstractWorld WorldContext =
Hit.collider.gameObject.GetComponent<AbstractWorld>();
            Direction = Vector2.Reflect (Direction,
Hit.normal).normalized;
            CheckContext (WorldContext);
        }

        DZEngine.Destroy (this);
    }
    else
    {
        int NumContacts = Bolt.GetContacts ();
        if (NumContacts > 0)
        {
            for (int i = 0; i < NumContacts; i++)

```

```

        {
            AbstractWorld WorldContext =
Bolt.Contacts[i].otherCollider.gameObject.GetComponent<AbstractWorld>();
            Direction = Vector2.Reflect(Direction,
Vector2.Perpendicular((Vector2)Bolt.Contacts[i].otherCollider.transform.pos
ition - Position)).normalized;
            CheckContext(WorldContext);
        }
    }

    Bolt.Position += Direction * ScaledSpeed;
}

private void CheckContext(AbstractWorld WorldContext)
{
    if (WorldContext != null)
    {
        if (WorldContext.Type == DZSettings.EntityType.PlayerCreature)
        {
            PlayerCreature Player =
(PlayerCreature)WorldContext.Context;
            Main.TakeLifeForce();
        }
        else if (WorldContext.Type ==
DZSettings.EntityType.EnemyCreature)
        {
            EnemyCreature Enemy = (EnemyCreature)WorldContext.Context;
            Enemy.ApplyVelocity(-Direction, 10);
            Enemy.Health--;
            if (Enemy.State == EnemyCreature.BodyState.Limp)
            {
                if (NumBounces > 0)
                {
                    NumBounces--;
                    return;
                }
            }
        }
        else if (WorldContext.Type == DZSettings.EntityType.Turret)
        {
            ((Turret)WorldContext.Context).LifeTime -= 1;
            if (NumBounces > 0)
            {
                NumBounces--;
                return;
            }
        }
        else if (WorldContext.Type ==
DZSettings.EntityType.BulletEntity)
        {
            ((BulletEntity)WorldContext.Context).Direction = -
Direction;
            if (NumBounces > 0)
            {
                NumBounces--;
                return;
            }
        }
    }
}
}

```

```

public void IsolateVelocity() { }

public void RestoreVelocity() { }

protected override void OnDelete ()
{
    DZEngine.Destroy(Bolt);
}

public override byte[] GetBytes ()
{
    List<byte> Data = new List<byte> ();
    Data.AddRange (BitConverter.GetBytes (Speed));
    Data.AddRange (BitConverter.GetBytes (Direction.x));
    Data.AddRange (BitConverter.GetBytes (Direction.y));
    Data.AddRange (Bolt.GetBytes ());
    return Data.ToArray ();
}

public override void ParseBytes (Packet Data)
{
    ParseSnapshot (ParseBytesToSnapshot (Data));
}

public struct Data
{
    public float Speed;
    public Vector2 Direction;
    public BodyChunk.Data Bolt;
}

public override object GetSnapshot ()
{
    return new Data ()
    {
        Speed = Speed,
        Direction = Direction,
        Bolt = (BodyChunk.Data) Bolt.GetSnapshot ()
    };
}

public static object ParseBytesToSnapshot (DZNetwork.Packet Data)
{
    return new Data ()
    {
        Speed = Data.ReadFloat (),
        Direction = new Vector2 (Data.ReadFloat (), Data.ReadFloat ()),
        Bolt = (BodyChunk.Data) BodyChunk.ParseBytesToSnapshot (Data)
    };
}

public override void ParseSnapshot (object ObjectData)
{
    Data Data = (Data) ObjectData;
    Speed = Data.Speed;
    Direction = Data.Direction;
    Bolt.ParseSnapshot (Data.Bolt);
}

```

```

    public override void Interpolate(object FromData, object ToData, float
Time)
    {
        Data From = (Data)FromData;
        Data To = (Data)ToData;
        Direction = From.Direction;
        Speed = From.Speed;
        Bolt.Interpolate(From.Bolt, To.Bolt, Time);
    }
}

```

---

### *Server/Assets/Scripts/Creatures/CoinEntity.cs*

---

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using UnityEngine;

using DeadZoneEngine;
using DeadZoneEngine.Entities;
using DeadZoneEngine.Entities.Components;
using DZNetwork;

public class CoinEntity : AbstractWorldEntity, IUpdatable, IRenderer,
IServerSendable
{
    public int SortingLayer { get; set; }
    public int ServerObjectType { get; set; } =
(int)DZSettings.EntityType.CoinEntity;
    public bool RecentlyUpdated { get; set; } = false;
    public bool ProtectedDeletion { get; set; } = false;

    public int Money = 1;
    public int Health = 0;
    public BodyChunk Coin;
    public float Decay = 10;

    public CoinEntity(ushort ID) : base(ID)
    {
        Init();
    }

    public CoinEntity() : base()
    {
        Init();
    }

    public void Init()
    {
        Coin = new BodyChunk();
        Coin.Context = this;
        Coin.ContextType = DZSettings.EntityType.CoinEntity;
        Coin.Collider.radius = 0.01f;
    }
}

```

```

        Coin.Velocity = new Vector2(UnityEngine.Random.Range(-5f, 5f),
UnityEngine.Random.Range(-5f, 5f));
    }

    public Vector2 Position
    {
        get
        {
            if (Coin != null)
                return Coin.Position;
            return Vector2.zero;
        }
        set
        {
            if (Coin != null)
                Coin.Position = value;
        }
    }

    public void InitializeRenderer()
    {
    }

    public void Render()
    {
        Coin.RenderObject.transform.localScale = new Vector2(0.2f, 0.2f);
        if (Health > 0)
            Coin.RenderColor = Color.red;
        else
            Coin.RenderColor = Color.magenta;
    }

    public void ServerUpdate()
    {
    }

    public void Update()
    {
        Decay -= Time.fixedDeltaTime;
        if (Decay < 0)
        {
            DZEngine.Destroy(this);
        }

        Collider2D[] C = Physics2D.OverlapCircleAll(Position, 1f);
        List<PlayerCreature> NearbyCreatures = new List<PlayerCreature>();
        for (int i = 0; i < C.Length; i++)
        {
            if (C[i] != null)
            {
                AbstractWorld AW = C[i].GetComponent<AbstractWorld>();
                if (AW != null && AW.Type ==
DZSettings.EntityType.PlayerCreature)
                {
                    NearbyCreatures.Add((PlayerCreature)AW.Context);
                }
            }
        }
        for (int i = 0; i < NearbyCreatures.Count; i++)

```

```

    {
        Vector2 Dir = NearbyCreatures[i].Position - Coin.Position;
        if (Dir.magnitude < 0.3f)
        {
            Main.Money += Money;
            Main.GainLifeForce(Health);
            DZEngine.Destroy(this);
        }
        float Speed = 10;
        Coin.Velocity += Dir.normalized * Speed * Time.fixedDeltaTime;
    }
    Coin.Velocity *= 0.9f;
}

public void BodyPhysicsUpdate() { }

public void IsolateVelocity() { }

public void RestoreVelocity() { }

protected override void OnDestroy()
{
    DZEngine.Destroy(Coin);
}

public override byte[] GetBytes()
{
    List<byte> Data = new List<byte>();
    Data.AddRange(BitConverter.GetBytes(Money));
    Data.AddRange(BitConverter.GetBytes(Health));
    Data.AddRange(Coin.GetBytes());
    return Data.ToArray();
}

public override void ParseBytes(Packet Data)
{
    ParseSnapshot(ParseBytesToSnapshot(Data));
}

public struct Data
{
    public int Money;
    public int Health;
    public BodyChunk.Data Coin;
}

public override object GetSnapshot()
{
    return new Data()
    {
        Money = Money,
        Health = Health,
        Coin = (BodyChunk.Data) Coin.GetSnapshot()
    };
}

public static object ParseBytesToSnapshot(DZNetwork.Packet Data)
{
    return new Data()
    {
        Money = Data.ReadInt(),

```

```

        Health = Data.ReadInt(),
        Coin = (BodyChunk.Data)BodyChunk.ParseBytesToSnapshot(Data)
    };
}

public override void ParseSnapshot(object ObjectData)
{
    Data Data = (Data)ObjectData;
    Money = Data.Money;
    Health = Data.Health;
    Coin.ParseSnapshot(Data.Coin);
}

public override void Interpolate(object FromData, object ToData, float
Time)
{
    Data From = (Data)FromData;
    Data To = (Data)ToData;
    Money = From.Money;
    Health = From.Health;
    Coin.Interpolate(From.Coin, To.Coin, Time);
}
}

```

---

### *Server/Assets/Scripts/Creatures/EnemyCreature.cs*

---

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

using UnityEngine;
using DeadZoneEngine.Entities;
using DeadZoneEngine.Entities.Components;
using DeadZoneEngine;

public class EnemyCreature : AbstractCreature, IServerSendable
{
    public struct WayPoint
    {
        public int Direction;
        public Vector2Int Position;
    }

    public struct Path
    {
        public List<WayPoint> Traversal;
        public string Map;
    }

    public int ServerObjectType { get; set; } =
(int)DZSettings.EntityType.EnemyCreature;
    public bool RecentlyUpdated { get; set; } = false;
    public bool ProtectedDeletion { get; set; } = false;
}

```



```

public int CurrentWayPoint = 0;
public Path Traversal;

public float DecayTimer = 5;
public int CorpseHP = 5;
public int Health = 5;
public float Speed = 1;
private float[] DynamicRunSpeed;
public BodyState State;

public enum BodyState
{
    Standing,
    Limp
}

public EnemyCreature(ushort ID) : base(ID)
{
    Initialize();
}
public EnemyCreature()
{
    Initialize();
}

public override void Render()
{
    BodyChunks[0].RenderObject.gameObject.transform.localScale = new
Vector2(0.5f, 0.5f);
    BodyChunks[1].RenderObject.gameObject.transform.localScale = new
Vector2(0.5f, 0.5f);

    if (State == BodyState.Limp) BodyColor = Color.grey;
    BodyChunks[0].RenderColor = BodyColor;
    BodyChunks[1].RenderColor = BodyColor;
}

Color BodyColor;
private void Initialize()
{
    BodyChunks = new BodyChunk[2];
    BodyChunks[0] = new BodyChunk(this);
    BodyChunks[1] = new BodyChunk(this);
    BodyChunks[0].Collider.radius = 0.25f;
    BodyChunks[1].Collider.radius = 0.25f;
    BodyChunks[0].Context = this;
    BodyChunks[0].ContextType = DZSettings.EntityType.EnemyCreature;
    BodyChunks[1].Context = this;
    BodyChunks[1].ContextType = DZSettings.EntityType.EnemyCreature;
    SetGravity(0f);

    BodyChunkConnections = new DistanceJoint[1];
    BodyChunkConnections[0] = new DistanceJoint();
    BodyChunkConnections[0].Set(new DistanceJointData(BodyChunks[0],
BodyChunks[1], 0.5f, Vector2.zero));
    BodyChunkConnections[0].Active = false;

    Physics2D.IgnoreCollision(BodyChunks[0].Collider,
BodyChunks[1].Collider, true); //Ignore collisions between body parts

    DynamicRunSpeed = new float[2];

```

```

    BodyColor = new Color(0, 1, 0);
    BodyChunks[0].RenderColor = BodyColor;
    BodyChunks[1].RenderColor = BodyColor;
}

public Vector2 Position
{
    get
    {
        if (BodyChunks[0] != null)
            return BodyChunks[0].Position;
        return Vector2.zero;
    }
    set
    {
        if (BodyChunks[0] != null)
            BodyChunks[0].Position = value;
        if (BodyChunks[1] != null)
            BodyChunks[1].Position = value;
    }
}

public void ApplyVelocity(Vector2 Direction, float Force)
{
    BodyChunks[0].Velocity += Direction * Force;
}

public void ApplyVelocity(Vector2 Vel)
{
    BodyChunks[0].Velocity += Vel;
}

public void ServerUpdate ()
{
}

public Func<Tilemap, WayPoint, Vector2, Vector2> PathingAlgorithm =
(Map, WP, Position) =>
{
    Vector2 WPAcualPosition =
Main.Tilemap.TilemapToWorldPosition(WP.Position);
    return (WPAcualPosition - Position).normalized;
};

private bool Death = false;
public override void Update ()
{
    if (Health < 0)
    {
        State = BodyState.Limp;
        if (DecayTimer > 0)
            DecayTimer -= Time.fixedDeltaTime;
        else
        {
            DecayTimer = 5;
            Health--;
        }
    }
    if (!Death)
    {
        Death = true;
    }
}

```

```

        int Count = UnityEngine.Random.Range(1, 4);
        for (int i = 0; i < Count; i++)
        {
            CoinEntity CE = new CoinEntity();
            CE.Money = 1;
            CE.Position = Position;
            CE.Health = UnityEngine.Random.Range(0f, 1f) < 0.25f ?
1 : 0;
        }
    }
    if (Health < -CorpseHP)
    {
        DZEngine.Destroy(this);
    }

    if (CurrentWayPoint < Traversal.Traversal.Count)
    {
        WayPoint WP = Traversal.Traversal[CurrentWayPoint];
        Vector2 WPActualPosition =
Main.Tilemap.TilemapToWorldPosition(WP.Position);
        MovementDirection = PathingAlgorithm(Main.Tilemap, WP,
Position);
        if ((Position.x < WPActualPosition.x + 0.5 && Position.x >
WPActualPosition.x - 0.5) &&
            (Position.y < WPActualPosition.y + 0.5 && Position.y >
WPActualPosition.y - 0.5))
            CurrentWayPoint++;
    }
    else
    {
        Main.TakeLifeForce();
        DZEngine.Destroy(this);
    }

    UpdateBodyState();
    UpdateMovement();
}

private void UpdateBodyState()
{
}

private Vector2 MovementDirection;
private void UpdateMovement()
{
    switch (State)
    {
        case BodyState.Limp:
        {
            BodyChunks[0].Velocity *= 0.8f;
            BodyChunks[1].Velocity *= 0.8f;
        }
        break;
        case BodyState.Standing:
        {
            DynamicRunSpeed[0] = 1f;
            DynamicRunSpeed[1] = 1.5f;
        }
    }
}

```

```

        BodyChunks[0].Velocity += new Vector2(Speed *
DynamicRunSpeed[0] * MovementDirection.x, Speed * DynamicRunSpeed[0] *
MovementDirection.y);
        BodyChunks[1].Velocity += new Vector2(Speed *
DynamicRunSpeed[1] * MovementDirection.x, Speed * DynamicRunSpeed[1] *
MovementDirection.y);

        BodyChunks[0].Velocity *= 0.8f;
        BodyChunks[1].Velocity *= 0.8f;
    }
    break;
}
}

public override void BodyPhysicsUpdate ()
{
    switch (State)
    {
        case BodyState.Limp:
        {
            SetGravity(0f);
            BodyChunks[1].SpriteOffset =
Vector2.Lerp(BodyChunks[1].SpriteOffset, Vector2.zero, 4 *
Time.fixedDeltaTime);
            BodyChunkConnections[0].Active = true;
        }
        break;
        case BodyState.Standing:
        {
            SetGravity(0f);
            BodyChunks[1].SpriteOffset =
Vector2.Lerp(BodyChunks[1].SpriteOffset, new Vector2(0, 0.3f), 4 *
Time.fixedDeltaTime);
            BodyChunkConnections[0].Active = false;

            float Dist = Vector2.Distance(BodyChunks[0].Position,
BodyChunks[1].Position);
            Vector2 Dir = (BodyChunks[0].Position -
BodyChunks[1].Position).normalized;
            BodyChunks[1].Position += Dist * Dir * 0.8f;
            BodyChunks[1].Velocity += Dist * Dir * 0.8f;

            BodyChunks[0].Velocity *= 0.8f;
            BodyChunks[1].Velocity *= 0.8f;
        }
        break;
    }
}

public void SetGravity(float Gravity)
{
    BodyChunks[0].Gravity = Gravity;
    BodyChunks[1].Gravity = Gravity;
}

protected override void OnDelete ()
{
    BodyChunks[0].Delete ();
    BodyChunks[1].Delete ();
    BodyChunkConnections[0].Delete ();
}

```

```

public override byte[] GetBytes ()
{
    List<byte> Data = new List<byte> ();
    Data.AddRange (BitConverter.GetBytes ((int) State));
    Data.AddRange (BodyChunks [0].GetBytes ());
    Data.AddRange (BodyChunks [1].GetBytes ());
    Data.AddRange (BodyChunkConnections [0].GetBytes ());
    return Data.ToArray ();
}

public override void ParseBytes (DZNetwork.Packet Data)
{
    ParseSnapshot ((Data) ParseBytesToSnapshot (Data));
}

public struct Data
{
    public BodyState State;
    public BodyChunk.Data BodyChunk0;
    public BodyChunk.Data BodyChunk1;
    public DistanceJoint.Data BodyChunkConnections0;
}

public static object ParseBytesToSnapshot (DZNetwork.Packet Data)
{
    return new Data ()
    {
        State = (BodyState) Data.ReadInt (),
        BodyChunk0 =
        (BodyChunk.Data) BodyChunk.ParseBytesToSnapshot (Data),
        BodyChunk1 =
        (BodyChunk.Data) BodyChunk.ParseBytesToSnapshot (Data),
        BodyChunkConnections0 =
        (DistanceJoint.Data) DistanceJoint.ParseBytesToData (Data)
    };
}

public override void ParseSnapshot (object ObjectData)
{
    Data Data = (Data) ObjectData;
    State = Data.State;
    BodyChunks [0].ParseSnapshot (Data.BodyChunk0);
    BodyChunks [1].ParseSnapshot (Data.BodyChunk1);
    BodyChunkConnections [0].ParseSnapshot (Data.BodyChunkConnections0);
}

public override object GetSnapshot ()
{
    return new Data ()
    {
        State = State,
        BodyChunk0 = (BodyChunk.Data) BodyChunks [0].GetSnapshot (),
        BodyChunk1 = (BodyChunk.Data) BodyChunks [1].GetSnapshot (),
        BodyChunkConnections0 =
        (DistanceJoint.Data) BodyChunkConnections [0].GetSnapshot ()
    };
}

public override void Interpolate (object FromData, object ToData, float
Time)
{

```

```

        Data From = (Data)FromData;
        Data To = (Data)ToData;
        BodyChunks[0].Interpolate(From.BodyChunk0, To.BodyChunk0, Time);
        BodyChunks[1].Interpolate(From.BodyChunk1, To.BodyChunk1, Time);
        BodyChunkConnections[0].Interpolate(From.BodyChunkConnections0,
        To.BodyChunkConnections0, Time);
    }
}

```

---

## *Server/Assets/Scripts/Creatures/PlayerCreature.cs*

---

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using UnityEngine;

using ClientHandle;
using DeadZoneEngine;
using DeadZoneEngine.Entities;
using DeadZoneEngine.Entities.Components;

public class PlayerCreature : AbstractCreature, IServerSendable
{
    public int ServerObjectType { get; set; } =
(int)DZSettings.EntityType.PlayerCreature;
    public bool RecentlyUpdated { get; set; } = false;
    public bool ProtectedDeletion { get; set; } = false;

    public bool Out;
    public float RunSpeed;

    public class Control
    {
        public PlayerController Owner;
        public Vector2 MovementDirection;
        public Vector2 ShieldVector;
        public float Interact;

        public struct Snapshot
        {
            public ulong InputID;
            public Vector2 MovementDirection;
        }

        public ulong InputID;

        public Snapshot GetSnapshot()
        {
            return new Snapshot()
            {
                InputID = InputID++,
                MovementDirection = MovementDirection
            };
        }
    }
}

```

```

        public void ParseSnapshot (Snapshot Snapshot)
        {
            MovementDirection = Snapshot.MovementDirection;
        }
    }
    public Control Controller { get; private set; } //Controller for player
movement

    private float[] DynamicRunSpeed; //Controls Speed of each bodychunk

    public PlayerCreature (ushort ID) : base (ID)
    {
        Initialize ();
    }
    public PlayerCreature ()
    {
        Initialize ();
    }

    public override void Render ()
    {
        BodyChunks [0].RenderObject.gameObject.transform.localScale = new
Vector2 (0.5f, 0.5f);
        BodyChunks [1].RenderObject.gameObject.transform.localScale = new
Vector2 (0.5f, 0.5f);
    }

    Color BodyColor;
    private void Initialize ()
    {
        if (DZSettings.ClientSidePrediction)
            Histogram = new DZNetwork.JitterBuffer <PlayerSnapshot> ();

        Controller = new Control ();
        RunSpeed = 2f;

        BodyChunks = new BodyChunk [2];
        BodyChunks [0] = new BodyChunk (this);
        BodyChunks [1] = new BodyChunk (this);
        BodyChunks [0].Collider.radius = 0.25f;
        BodyChunks [1].Collider.radius = 0.25f;
        BodyChunks [0].Context = this;
        BodyChunks [0].ContextType = DZSettings.EntityType.PlayerCreature;
        BodyChunks [1].Context = this;
        BodyChunks [1].ContextType = DZSettings.EntityType.PlayerCreature;
        SetGravity (0f);

        BodyChunkConnections = new DistanceJoint [1];
        BodyChunkConnections [0] = new DistanceJoint ();
        BodyChunkConnections [0].Set (new DistanceJointData (BodyChunks [0],
BodyChunks [1], 0.5f, Vector2.zero));
        BodyChunkConnections [0].Active = false;

        Physics2D.IgnoreCollision (BodyChunks [0].Collider,
BodyChunks [1].Collider, true); //Ignore collisions between body parts

        DynamicRunSpeed = new float [2];

        BodyColor = new Color (UnityEngine.Random.Range (0f, 1f),
UnityEngine.Random.Range (0f, 1f), UnityEngine.Random.Range (0f, 1f));

```

```

        BodyChunks[0].RenderColor = BodyColor;
        BodyChunks[1].RenderColor = BodyColor;
    }

    public Vector2 Position
    {
        get
        {
            if (BodyChunks[0] != null)
                return BodyChunks[0].Position;
            return Vector2.zero;
        }
        set
        {
            if (BodyChunks[0] != null)
                BodyChunks[0].Position = value;
            if (BodyChunks[1] != null)
                BodyChunks[1].Position = value;
        }
    }

    public void ApplyVelocity(Vector2 Direction, float Force)
    {
        BodyChunks[0].Velocity += Direction * Force;
    }

    public void ApplyVelocity(Vector2 Vel)
    {
        BodyChunks[0].Velocity += Vel;
    }

    public void ServerUpdate()
    {
        if (Controller.Owner == null || DZSettings.ClientSidePrediction ==
false) return;

        UpdateReconcilliation();
        LerpReconcilleError();

        BodyChunks[0].PhysicallyActive = true;
        BodyChunks[1].PhysicallyActive = true;
        BodyChunkConnections[0].PhysicallyActive = true;
        BodyChunks[0].Kinematic = false;
        BodyChunks[1].Kinematic = false;
    }

    bool Placed = false;
    public override void Update()
    {
        if (Main.GameStarted)
        {
            if (Controller.Interact > 0 && !Placed)
            {
                Placed = true;
                Vector2Int PlacePositionBounds0 =
Main.Tilemap.WorldPositionToTilemap(Position + new Vector2(0.3f, 0.3f));
                Vector2Int PlacePositionBounds1 =
Main.Tilemap.WorldPositionToTilemap(Position + new Vector2(0.3f, -0.3f));
                Vector2Int PlacePositionBounds2 =
Main.Tilemap.WorldPositionToTilemap(Position + new Vector2(-0.3f, 0.3f));
            }
        }
    }

```



```

        Vector2Int PlacePositionBounds3 =
Main.Tilemap.WorldPositionToTilemap(Position + new Vector2(-0.3f, -0.3f));
        if
(Main.Tilemap.GetFloorTileAtPosition(PlacePositionBounds0).AnimationFrame
!= 2 &&

Main.Tilemap.GetFloorTileAtPosition(PlacePositionBounds1).AnimationFrame !=
2 &&

Main.Tilemap.GetFloorTileAtPosition(PlacePositionBounds2).AnimationFrame !=
2 &&

Main.Tilemap.GetFloorTileAtPosition(PlacePositionBounds3).AnimationFrame !=
2 &&

Main.Tilemap.GetWallTileAtPosition(PlacePositionBounds0).Blank != 1 &&

Main.Tilemap.GetWallTileAtPosition(PlacePositionBounds1).Blank != 1 &&

Main.Tilemap.GetWallTileAtPosition(PlacePositionBounds2).Blank != 1 &&

Main.Tilemap.GetWallTileAtPosition(PlacePositionBounds3).Blank != 1)
        {
            Collider2D[] C = Physics2D.OverlapCircleAll(Position,
0.5f);
            bool ValidPlace = true;
            for (int i = 0; i < C.Length; i++)
            {
                if (C[i] != null)
                {
                    AbstractWorld AW =
C[i].GetComponent<AbstractWorld>();
                    if (AW == null || (AW != null && AW.Type !=
DZSettings.EntityType.PlayerCreature))
                    {
                        ValidPlace = false;
                    }
                }
            }

            if (ValidPlace && Main.Money >= 10)
            {
                Main.Money -= 10;
                Turret T = new Turret();
                T.Position = Position;
                Main.Towers.Add(T);
            }
        }
        else if (Controller.Interact <= 0)
        {
            Placed = false;
        }
    }

    UpdateBodyState();
    UpdateMovement();
}

private void UpdateBodyState()
{

```

```

}

private void UpdateMovement ()
{
    DynamicRunSpeed[0] = 1f;
    DynamicRunSpeed[1] = 1.5f;
    if (Controller != null)
    {
        BodyChunks[0].Velocity += new Vector2 (RunSpeed *
DynamicRunSpeed[0] * Controller.MovementDirection.x, RunSpeed *
DynamicRunSpeed[0] * Controller.MovementDirection.y);
        BodyChunks[1].Velocity += new Vector2 (RunSpeed *
DynamicRunSpeed[1] * Controller.MovementDirection.x, RunSpeed *
DynamicRunSpeed[1] * Controller.MovementDirection.y);
    }

    BodyChunks[0].Velocity *= 0.8f;
    BodyChunks[1].Velocity *= 0.8f;
}

public override void BodyPhysicsUpdate ()
{
    BodyChunks[0].Kinematic = false;
    BodyChunks[1].Kinematic = false;

    SetGravity (0f);
    BodyChunks[1].SpriteOffset =
Vector2.Lerp (BodyChunks[1].SpriteOffset, new Vector2 (0, 0.3f), 4 *
Time.fixedDeltaTime);
    BodyChunkConnections[0].Active = false;

    float Dist = Vector2.Distance (BodyChunks[0].Position,
BodyChunks[1].Position);
    Vector2 Dir = (BodyChunks[0].Position -
BodyChunks[1].Position).normalized;
    BodyChunks[1].Position += Dist * Dir * 0.8f;
    BodyChunks[1].Velocity += Dist * Dir * 0.8f;

    BodyChunks[0].Velocity *= 0.8f;
    BodyChunks[1].Velocity *= 0.8f;
}

public void SetGravity (float Gravity)
{
    BodyChunks[0].Gravity = Gravity;
    BodyChunks[1].Gravity = Gravity;
}

protected override void OnDestroy ()
{
    BodyChunks[0].Delete ();
    BodyChunks[1].Delete ();
    BodyChunkConnections[0].Delete ();
}

public override byte[] GetBytes ()
{
    List<byte> Data = new List<byte> ();
    Data.AddRange (BitConverter.GetBytes (Controller.InputID));
    Data.AddRange (BodyChunks[0].GetBytes ());
}

```

```

        Data.AddRange (BodyChunks [1].GetBytes ());
        Data.AddRange (BodyChunkConnections [0].GetBytes ());
        return Data.ToArray ();
    }

    public override void ParseBytes (DZNetwork.Packet Data)
    {
        ParseSnapshot ((Data) ParseBytesToSnapshot (Data));
    }

    public struct Data
    {
        public ulong InputID;
        public BodyChunk.Data BodyChunk0;
        public BodyChunk.Data BodyChunk1;
        public DistanceJoint.Data BodyChunkConnections0;
    }

    public static object ParseBytesToSnapshot (DZNetwork.Packet Data)
    {
        return new Data ()
        {
            InputID = Data.ReadULong (),
            BodyChunk0 =
                (BodyChunk.Data) BodyChunk.ParseBytesToSnapshot (Data),
            BodyChunk1 =
                (BodyChunk.Data) BodyChunk.ParseBytesToSnapshot (Data),
            BodyChunkConnections0 =
                (DistanceJoint.Data) DistanceJoint.ParseBytesToData (Data)
        };
    }

    public override void ParseSnapshot (object ObjectData)
    {
        if (Controller.Owner != null && DZSettings.ClientSidePrediction &&
            !Reconcille)
            return;
        Data Data = (Data) ObjectData;
        BodyChunks [0].ParseSnapshot (Data.BodyChunk0);
        BodyChunks [1].ParseSnapshot (Data.BodyChunk1);
        BodyChunkConnections [0].ParseSnapshot (Data.BodyChunkConnections0);
    }

    public override object GetSnapshot ()
    {
        return new Data ()
        {
            InputID = Controller.InputID,
            BodyChunk0 = (BodyChunk.Data) BodyChunks [0].GetSnapshot (),
            BodyChunk1 = (BodyChunk.Data) BodyChunks [1].GetSnapshot (),
            BodyChunkConnections0 =
                (DistanceJoint.Data) BodyChunkConnections [0].GetSnapshot ()
        };
    }

    public class PlayerSnapshot
    {
        public Data Snapshot;
        public Control.Snapshot Controls;
    }

    public DZNetwork.JitterBuffer<PlayerSnapshot> Histogram = null;
    private void UpdateReconcillation ()

```

```

{
    Histogram.Add(new PlayerSnapshot ()
    {
        Snapshot = (Data)GetSnapshot (),
        Controls = Controller.GetSnapshot ()
    });
}

private void LerpReconcilleError ()
{
    const float Amount = 4f;
    float Error = (ReconcilledSelf.BodyChunk0.Position -
BodyChunks[0].Position).SqrMagnitude ();
    if (Error < 1)
    {
        BodyChunks[0].Position = Vector3.Lerp (BodyChunks[0].Position,
ReconcilledSelf.BodyChunk0.Position, Amount * Time.fixedDeltaTime);
        BodyChunks[1].Position = Vector3.Lerp (BodyChunks[1].Position,
ReconcilledSelf.BodyChunk1.Position, Amount * Time.fixedDeltaTime);
    }
    else
    {
        BodyChunks[0].Position = ReconcilledSelf.BodyChunk0.Position;
        BodyChunks[1].Position = ReconcilledSelf.BodyChunk1.Position;
    }
}

private PlayerSnapshot Current = null;
public Data CurrentSelf;
private bool ValidPredictPass;
private bool FinishedPredict;
public void StartClientPrediction (Game.ServerSnapshot FromData)
{
    ValidPredictPass = FromData.Data.ContainsKey (ID);
    CurrentSelf = (Data)GetSnapshot ();
    Reconcille = true;
    if (!ValidPredictPass) return;
    Data ClientPredictBaseline = (Data)FromData.Data[ID].Data;
    if (LastReconcilled >= ClientPredictBaseline.InputID)
    {
        ValidPredictPass = false;
        return;
    }
    LastReconcilled = ClientPredictBaseline.InputID;
    Histogram.Iterate (S =>
    {
        if (S.Value.Controls.InputID >= ClientPredictBaseline.InputID)
        {
            Current = S.Value;
        }
    }, S => S.Value.Controls.InputID >= ClientPredictBaseline.InputID);
    if (Current != null)
    {
        Histogram.Dequeue (Current);
        ParseSnapshot (ClientPredictBaseline);
        FinishedPredict = false;
        CurrentKey = Histogram.FirstKey;
    }
    else
    {
        Histogram.Clear ();
    }
}

```

```

        ValidPredictPass = false;
    }
}
private DZNetwork.JitterBuffer<PlayerSnapshot>.Key CurrentKey;
public void ClientPrediction()
{
    if (!ValidPredictPass) return;
    if (CurrentKey != null)
    {
        Controller.ParseSnapshot(CurrentKey.Value.Controls);
        if (!FinishedPredict && CurrentKey.Next == null)
        {
            FinishedPredict = true;
            ReconcilledSelf = (Data)GetSnapshot();
        }
        CurrentKey = CurrentKey.Next;
    }
}
public void EndClientPrediction()
{
    ParseSnapshot(CurrentSelf);
    Reconcille = false;
}

private Data ReconcilledSelf;
private bool Reconcille = false;
private ulong LastReconcilled = 0;
public override void Interpolate(object FromData, object ToData, float
Time)
{
    if (Controller.Owner != null && DZSettings.ClientSidePrediction)
        return;

    Data From = (Data)FromData;
    Data To = (Data)ToData;
    BodyChunks[0].Interpolate(From.BodyChunk0, To.BodyChunk0, Time);
    BodyChunks[1].Interpolate(From.BodyChunk1, To.BodyChunk1, Time);
    BodyChunkConnections[0].Interpolate(From.BodyChunkConnections0,
To.BodyChunkConnections0, Time);
}
}

```

---

### *Server/Assets/Scripts/Creatures/Turret.cs*

---

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

using UnityEngine;
using DeadZoneEngine.Entities;
using DeadZoneEngine.Entities.Components;
using DeadZoneEngine;

public class Turret : AbstractCreature, IServerSendable

```

```

{
    public int ServerObjectType { get; set; } =
(int)DZSettings.EntityType.Turret;
    public bool RecentlyUpdated { get; set; } = false;
    public bool ProtectedDeletion { get; set; } = false;

    public float Timer = 0;
    public float FireRate = 2;
    public float LifeTime = 150;

    public Turret(ushort ID) : base(ID)
    {
        Initialize();
    }
    public Turret()
    {
        Initialize();
    }

    public override void Render()
    {
        BodyChunks[0].RenderObject.gameObject.transform.localScale = new
Vector2(0.7f, 0.7f);
    }

    Color BodyColor;
    private void Initialize()
    {
        BodyChunks = new BodyChunk[1];
        BodyChunks[0] = new BodyChunk(this);
        BodyChunks[0].Collider.radius = 0.35f;
        BodyChunks[0].Kinematic = true;
        BodyChunks[0].Context = this;
        BodyChunks[0].ContextType = DZSettings.EntityType.Turret;
        SetGravity(0f);

        BodyColor = new Color(0.56f, 0.56f, 0.56f);
        BodyChunks[0].RenderColor = BodyColor;
    }

    public void SetGravity(float Gravity)
    {
        BodyChunks[0].Gravity = Gravity;
    }

    public Vector2 Position
    {
        get
        {
            if (BodyChunks[0] != null)
                return BodyChunks[0].Position;
            return Vector2.zero;
        }
        set
        {
            if (BodyChunks[0] != null)
                BodyChunks[0].Position = value;
        }
    }

    public void ServerUpdate()

```

```

{
}

Action<Turret> PerformFireAction = (Turret Self) =>
{
    Vector2[] Directions = new Vector2[]
    {
        new Vector2(0, 1),
        new Vector2(0, -1),
        new Vector2(1, 0),
        new Vector2(-1, 0),
        new Vector2(1, 1),
        new Vector2(1, -1),
        new Vector2(-1, 1),
        new Vector2(-1, -1)
    };
    for (int i = 0; i < Directions.Length; i++)
    {
        BulletEntity Shot = new BulletEntity();
        Shot.Position = Self.Position;
        Shot.Direction = Directions[i].normalized;
        Shot.Speed = 2f;
    }
};

public override void Update ()
{
    LifeTime -= Time.fixedDeltaTime;
    if (LifeTime < 0)
    {
        DZEngine.Destroy(this);
    }

    Timer += Time.fixedDeltaTime;
    if (Timer >= FireRate)
    {
        Timer = 0;
        PerformFireAction?.Invoke(this);
    }
}

private void UpdateBodyState ()
{
}

protected override void OnDelete ()
{
    BodyChunks[0].Delete();
}

public override byte[] GetBytes ()
{
    List<byte> Data = new List<byte>();
    Data.AddRange(BodyChunks[0].GetBytes());
    return Data.ToArray();
}

public override void ParseBytes (DZNetwork.Packet Data)
{
    ParseSnapshot((Data) ParseBytesToSnapshot(Data));
}

```

```

public struct Data
{
    public BodyChunk.Data BodyChunk0;
}

public static object ParseBytesToSnapshot (DZNetwork.Packet Data)
{
    return new Data ()
    {
        BodyChunk0 =
        (BodyChunk.Data)BodyChunk.ParseBytesToSnapshot (Data)
    };
}

public override void ParseSnapshot (object ObjectData)
{
    Data Data = (Data)ObjectData;
    BodyChunks [0].ParseSnapshot (Data.BodyChunk0);
}

public override object GetSnapshot ()
{
    return new Data ()
    {
        BodyChunk0 = (BodyChunk.Data)BodyChunks [0].GetSnapshot ()
    };
}

public override void Interpolate (object FromData, object ToData, float
Time)
{
    Data From = (Data)FromData;
    Data To = (Data)ToData;
    BodyChunks [0].Interpolate (From.BodyChunk0, To.BodyChunk0, Time);
}
}

```

---

### *Server/Assets/Scripts/Creatures/DZSettings.cs*

---

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

public static class DZSettings
{
    public enum EntityType
    {
        Null,
        PlayerCreature,
        Tilemap,
        TriggerPlate,
        BulletEntity,
        EnemyCreature,
        Turret,
    }
}

```



```

        CoinEntity,
        CrystalEntity
    }

    public static int NumPhysicsIterations = 10;
    public static bool ActiveRenderers = true;
    public static bool ActiveControllers = false;
    public static bool ClientSidePrediction = false;
    public static bool Client = false;
}

```

---

### *Server/Assets/Scripts/Creatures/Game.cs*

---

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

using UnityEngine;
using DeadZoneEngine;
using ClientHandle;
using DZNetwork;
using DeadZoneEngine.Entities;
using System.Net;
using DeadZoneEngine.Controllers;

/// <summary>
/// Manages the server connection and game functionality
/// </summary>
public class Game
{
    public class ServerSnapshot
    {
        public struct Object
        {
            public DZSettings.EntityType Type;
            public bool FlaggedToDelete;
            public object Data;
        }

        public ulong ServerTick;
        public Dictionary<ushort, Object> Data = new Dictionary<ushort,
Object>();
    }

    public static ulong ServerTicks = 0;
    public static int ServerTickRate = 60;

    /// <summary>
    /// Called once a frame
    /// </summary>
    public static void FixedUpdate ()
    {
        Loader.Socket.FixedUpdate ();
    }
}

```

```

        UpdateClients();
        SendSnapshot();

        ServerTicks++;
    }

    private static void UpdateClients ()
    {
        List<Client> Clients = ClientID.ConnectedClients.Values.ToList();
        foreach (Client C in Clients)
        {
            if (C != null)
                if (C.LostConnection)
                {
                    if (C.TicksSinceConnectionLoss > Client.TicksToTimeout)
                        C.Destroy();
                    C.TicksSinceConnectionLoss++;
                }
                else
                    C.TicksSinceConnectionLoss = 0;
        }
    }

    private static void SendSnapshot () //Sends world snapshot to given
client
    {
        Packet SnapshotPacket = new Packet();
        SnapshotPacket.Write(ServerTickRate);
        SnapshotPacket.Write(ServerTicks);
        SnapshotPacket.Write(Main.LifeForce[0]);
        SnapshotPacket.Write(Main.LifeForce[1]);
        SnapshotPacket.Write(Main.LifeForce[2]);
        SnapshotPacket.Write(Main.Money);

        SnapshotPacket.Write(DZEngine.ServerSendableObjects.Count);
        for (int i = 0; i < DZEngine.ServerSendableObjects.Count; i++)
        {
            SnapshotPacket.Write(DZEngine.GetBytes(DZEngine.ServerSendableObjects[i]));
        }
        SnapshotPacket.InsertChecksum(sizeof(int) + sizeof(ulong));
        Loader.Socket.Send(SnapshotPacket, ServerCode.ServerSnapshot);
    }

    public static Client SyncPlayers (DZUDPSocket.RecievePacketWrapper
Packet)
    {
        Client C = Client.GetClient(Packet.Client);
        byte NumPlayers = Packet.Data.ReadByte();
        if (C.NumPlayers != NumPlayers)
        {
            while (C.NumPlayers < NumPlayers)
                C.AddPlayer();
        }

        Packet SyncPacket = new Packet();
        SyncPacket.Write(NumPlayers);
        SyncPacket.Write(C.ID);
        for (int i = 0; i < C.NumPlayers; i++)
        {
            if (C.Players[i] == null)

```

```

        SyncPacket.Write (byte.MaxValue);
    else
        SyncPacket.Write (C.Players [i].GetBytes ());
    }
    Loader.Socket.SendTo (SyncPacket, ServerCode.SyncPlayers,
Packet.Client);
    return C;
}

public static void UnWrapSnapshot (DZUDPSocket.RecievePacketWrapper
Packet)
{
    Client C = SyncPlayers (Packet);
    InputMapping.ParseBytes (Packet.Data, C);
}

public static void AddConnection (IPEndPoint EndPoint)
{
    Debug.Log ("Client Connected: " + EndPoint.Address + ":" +
EndPoint.Port);

    Client ConnectedClient = Client.GetClient (EndPoint);
    ConnectedClient.LostConnection = false;
}

public static void RemoveConnection (IPEndPoint EndPoint)
{
    Debug.Log ("Client Disconnected: " + EndPoint.Address + ":" +
EndPoint.Port);

    Client DisconnectedClient = Client.GetClient (EndPoint);
    DisconnectedClient.LostConnection = true;
}
}
}

```

---

### *Server/Assets/Scripts/Creatures/InputManager.cs*

---

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

using UnityEngine.InputSystem;
using UnityEngine;

using DeadZoneEngine.Controllers;
using ClientHandle;
using static DeadZoneEngine.Controllers.InputMapping;
using DZNetwork;

public class PlayerController : Controller
{
    public Player Owner;
    public PlayerCreature.Control PlayerControl;
}

```

```

    public PlayerController(Player Owner, PlayerCreature.Control
PlayerControl) : base()
    {
        this.Owner = Owner;
        this.PlayerControl = PlayerControl;
    }
    public PlayerController(InputDevice Device, DeviceController DC) :
base(Device, DC)
    {
        InputAction Movement = ActionMap.AddAction("Movement",
InputActionType.PassThrough);
        InputAction Interact = ActionMap.AddAction("Interact",
InputActionType.Button);
        if (Device is Keyboard)
        {
            Movement.AddCompositeBinding("2DVector(mode=2)")
                .With("Up", Device.path + "/w")
                .With("Down", Device.path + "/s")
                .With("Left", Device.path + "/a")
                .With("Right", Device.path + "/d");
            Interact.AddBinding(Device.path + "/space");
        }
        else
        {
            Movement.AddCompositeBinding("2DVector(mode=2)")
                .With("Up", Device.path + "/stick/up")
                .With("Down", Device.path + "/stick/down")
                .With("Left", Device.path + "/stick/left")
                .With("Right", Device.path + "/stick/right");
            Interact.AddBinding(Device.path + "/dpad/up");
        }
        Movement.performed += MoveAction;
        Interact.performed += InteractAction;
    }

    protected override void SetType()
    {
        Type = ControllerType.PlayerController;
    }

    public override void
OnInput(UnityEngine.InputSystem.Controls.ButtonControl Control)
    {
    }

    private Vector2 MovementDirection;
    private float Interact;
    public void MoveAction(InputAction.CallbackContext Context)
    {
        MovementDirection = Context.ReadValue<Vector2>();
    }

    public void InteractAction(InputAction.CallbackContext Context)
    {
        Interact = Context.ReadValue<float>();
    }

    public override void Tick()
    {
        if (PlayerControl == null) return;

```

```

        PlayerControl.MovementDirection = MovementDirection;
        PlayerControl.Interact = Interact;
    }

    public override void ParseBytes(Packet Data)
    {
        PlayerControl.InputID = Data.ReadULong();
        PlayerControl.Interact = Data.ReadFloat();
        PlayerControl.MovementDirection = new Vector2(Data.ReadFloat(),
Data.ReadFloat());
    }
    public override byte[] GetBytes()
    {
        List<byte> Data = new List<byte>();
        Data.Add(Owner.ID);
        Data.AddRange(BitConverter.GetBytes(PlayerControl.InputID));
        Data.AddRange(BitConverter.GetBytes(PlayerControl.Interact));

Data.AddRange(BitConverter.GetBytes(PlayerControl.MovementDirection.x));

Data.AddRange(BitConverter.GetBytes(PlayerControl.MovementDirection.y));
        return Data.ToArray();
    }
}

public static class InputManager
{
    public static void Initialize()
    {
        InputMapping.OnDeviceAdd += OnDeviceAdd;
        InputMapping.OnDeviceDisconnect += OnDeviceDisconnect;
        InputMapping.OnDeviceReconnect += OnDeviceReconnect;
        InputMapping.OnDeviceRemove += OnDeviceRemove;
    }

    public static void OnDeviceAdd(InputDevice Device)
    {
    }

    public static void OnDeviceDisconnect(InputDevice Device)
    {
    }

    public static void OnDeviceReconnect(InputDevice Device)
    {
    }

    public static void OnDeviceRemove(InputDevice Device)
    {
    }
}

```

---

### *Server/Assets/Scripts/Creatures/Loader.cs*

---

```

using System.Collections;
using System.Collections.Generic;

```

```

using UnityEngine;
using System.IO;

using DZNetwork;
using System;
using System.Text.RegularExpressions;

public class Loader
{
    public static DZServer Socket = new DZServer();

    [RuntimeInitializeOnLoadMethod] //Runs on application start
    private static void Start()
    {
        Physics2D.queriesStartInColliders = false;
        Application.quitting += Dispose; //Setup dispose to call when game
is closed
        Application.targetFrameRate = Game.ServerTickRate; //Limit server
tick rate / frame rate
        Time.fixedDeltaTime = 1f / Game.ServerTickRate; //Fixed physics
update rate
        QualitySettings.vSyncCount = 0; //Turn off vsync
        Physics2D.simulationMode = SimulationMode2D.Script; //My program
controls when unity updates

        Socket.ConnectHandle += Game.AddConnection;
        Socket.DisconnectHandle += Game.RemoveConnection;
        Socket.PacketHandle += ServerHandle.ProcessPacket;

        int Port = 26950;
        try
        {
            string Text = Regex.Replace(File.ReadAllText(@"Server.cfg"),
@"[\n\r\t]", "");
            Port = int.Parse(Text.Split(':')[1]);
        }
        catch (Exception E)
        {
            Port = 26950;
        }
        Socket.Connect(Port); //Startup server
    }

    private static void Dispose()
    {
        Socket.Dispose();
    }
}

```

---

### *Server/Assets/Scripts/Creatures/Main.cs*

---

```

using System.Collections;
using System.Collections.Generic;
using UnityEngine;

using DeadZoneEngine;

```









```

        Tilemap.Resize(new Vector2Int(20, 20),
Tilemap.TilesFromString(MenuFloorMap),
Tilemap.TilesFromString(MenuWallMap));
        Tilemap.ReleaseUnusedResources();
    }

    StartPlate = new TriggerPlate(new Vector2(4, 2), new Vector2(0, -
3));
    StartPlate.OnTrigger = StartGame;
}

private static EnemyCreature.Path GeneratePath(int Height, int Width)
{
    List<EnemyCreature.WayPoint> Path = new
List<EnemyCreature.WayPoint>();

    int NumTurns = UnityEngine.Random.Range(6, 10);
    string[] Tiles = null;
    bool ValidPath = false;
    while (!ValidPath)
    {
        Path.Clear();
        Tiles = MenuFloorMap.Split('/');
        Vector2Int StartPosition = new Vector2Int(1,
UnityEngine.Random.Range(1, Height - 1));
        float Chance = UnityEngine.Random.Range(0f, 1f);
        if (Chance > 0.25)
            StartPosition = new Vector2Int(Width - 2,
UnityEngine.Random.Range(1, Height - 1));
        else if (Chance > 0.5)
            StartPosition = new Vector2Int(UnityEngine.Random.Range(1,
Width - 1), 1);
        else if (Chance > 0.75)
            StartPosition = new Vector2Int(UnityEngine.Random.Range(1,
Width - 1), Height - 2);

        int Direction = 1;
        if (Chance > 0.25)
            Direction = 3;
        else if (Chance > 0.5)
            Direction = 0;
        else if (Chance > 0.75)
            Direction = 2;

        Path.Add(new EnemyCreature.WayPoint()
        {
            Direction = Direction,
            Position = StartPosition
        });

        for (int i = 0; i < NumTurns; i++)
        {
            int CurrentDirection = Direction;
            int Length = UnityEngine.Random.Range(3, 8);
            if (i == NumTurns - 1)
            {
                switch (Direction)
                {
                    case 0: Length = Height - StartPosition.y; break;
                    case 1: Length = Width - StartPosition.x; break;
                    case 2: Length = StartPosition.y; break;
                }
            }
        }
    }
}

```

```

        case 3: Length = StartPosition.x; break;
    }
}
for (int j = 0; j < Length; j++)
{
    Tiles[StartPosition.y * Width + StartPosition.x] =
"0,2,0,1";
    Vector2Int NewPosition = StartPosition;
    switch (Direction)
    {
        case 0: NewPosition.y++; break;
        case 1: NewPosition.x++; break;
        case 2: NewPosition.y--; break;
        case 3: NewPosition.x--; break;
    }
    if (NewPosition.x < 1 || NewPosition.x > Width - 2 ||
NewPosition.y < 1 || (NewPosition.y > Height - 3 && Direction == 0))
    {
        break;
    }
    StartPosition = NewPosition;
}
bool ValidDirection = true;
do
{
    Direction = (Direction + (UnityEngine.Random.Range(0f,
1f) > 0.5 ? 1 : -1)) % 4;
    if (Direction < 0) Direction += 4;
    Vector2Int NewPosition = StartPosition;
    switch (Direction)
    {
        case 0: NewPosition.y++; break;
        case 1: NewPosition.x++; break;
        case 2: NewPosition.y--; break;
        case 3: NewPosition.x--; break;
    }
    if (NewPosition.x < 1 || NewPosition.x > Width - 1 ||
NewPosition.y < 1 || NewPosition.y > Height - 2)
    {
        ValidDirection = false;
    }
    else if (Tiles[NewPosition.y * Width + NewPosition.x]
== "0,2,0,1")
    {
        ValidDirection = false;
    }
}
while (CurrentDirection == Direction && !ValidDirection);
Path.Add(new EnemyCreature.WayPoint()
{
    Direction = CurrentDirection,
    Position = StartPosition
});
}

bool TopLeftQuadrant = false;
bool TopRightQuadrant = false;
bool BottomLeftQuadrant = false;
bool BottomRightQuadrant = false;
for (int i = 0; i < Path.Count; i++)
{

```

```

        if (Path[i].Position.x < Width / 2)
        {
            if (Path[i].Position.y < Height / 2)
            {
                BottomLeftQuadrant = true;
            }
            else
            {
                TopLeftQuadrant = true;
            }
        }
        else
        {
            if (Path[i].Position.y < Height / 2)
            {
                BottomRightQuadrant = true;
            }
            else
            {
                TopRightQuadrant = true;
            }
        }
    }
    ValidPath = TopLeftQuadrant && TopRightQuadrant &&
    BottomLeftQuadrant && BottomRightQuadrant;
}

return new EnemyCreature.Path()
{
    Map = string.Join("/", Tiles),
    Traversal = Path
};
}

private static EnemyCreature.Path CurrentPath;
public static void StartGame()
{
    CurrentPath = GeneratePath(20, 20);
    Tilemap.Resize(new Vector2Int(20, 20),
    Tilemap.TilesFromString(CurrentPath.Map),
    Tilemap.TilesFromString(MenuWallMap));

    GameStarted = true;
    DZEngine.Destroy(StartPlate);
}

public static int[] LifeForce = new int[3] { 10, 10, 10 };
private static float WaveTimer = 5;
private static int WaveSize = 10;
private static int WaveMaxSize = 10;
private static int WaveHealth = 1;
private static int Wave = 0;
private static float WaveSpacing = 0.3f;
private static float WaveSpacingMax = 1;
private static int EnemiesToSpawn = 5;
private static float SpawnTimer = 0;
public static int Money = 40;
public static float Drain = 0;
private static List<EnemyCreature> Enemies = new List<EnemyCreature>();
public static List<Turret> Towers = new List<Turret>();

```

```

public static void TakeLifeForce()
{
    for (int i = 0; i < LifeForce.Length; i++)
    {
        if (LifeForce[i] > 0)
        {
            LifeForce[i]--;
            break;
        }
    }
}

public static void GainLifeForce(int Health)
{
    for (int i = 0; i < LifeForce.Length; i++)
    {
        if (LifeForce[i] != 0)
        {
            LifeForce[i] += Health;
            if (LifeForce[i] > 10)
                LifeForce[i] = 10;
            break;
        }
    }
}

private static void Reset()
{
    Money = 40;
    for (int i = 0; i < Enemies.Count; i++)
    {
        DZEngine.Destroy(Enemies[i]);
    }
    Enemies.Clear();
    for (int i = 0; i < Towers.Count; i++)
    {
        DZEngine.Destroy(Towers[i]);
    }
    Towers.Clear();
    LifeForce = new int[3] { 10, 10, 10 };
    GameStarted = false;
    LoadMenu();
}

// Update is called once per frame
public static void FixedUpdate()
{
    if (DZSettings.ActiveRenderers == true)
        foreach (IRenderer<SpriteRenderer> Renderer in SpriteRenderers)
        {
            if (Renderer.SortingLayer == (int)SortingLayers.Default)
                Renderer.RenderObject.sortingOrder = Mathf.RoundToInt(-
Renderer.RenderObject.transform.parent.position.y * 10);
        }

    if (GameStarted)
    {
        if (Drain <= 0)
        {
            Drain = 5;
            TakeLifeForce();
        }
    }
}

```

```

    }
    else Drain -= Time.fixedDeltaTime;

    List<Client> Clients =
ClientID.ConnectedClients.Values.ToList();
    int TotalNumPlayers = 0;
    foreach (Client C in Clients)
    {
        if (C == null) continue;
        if (C.Players == null) continue;
        for (int i = 0; i < C.Players.Length; i++)
        {
            if (C.Players[i] == null) continue;
            if (C.Players[i].Entity == null) continue;
            TotalNumPlayers++;
        }
    }

    if (TotalNumPlayers == 0)
    {
        Reset();
    }

    bool Alive = false;
    for (int i = 0; i < LifeForce.Length; i++)
    {
        if (LifeForce[i] > 0)
            Alive = true;
    }

    if (WaveTimer > 0)
    {
        WaveTimer -= Time.fixedDeltaTime;
    }
    else
    {
        WaveTimer = Random.Range(4f, 10f);
        Wave++;
        EnemiesToSpawn = Random.Range(10, WaveMaxSize);
        WaveHealth++;
        WaveSpacing = Random.Range(0.1f, WaveSpacingMax);
        WaveSpacingMax += Random.Range(-0.5f, 0.5f);
        if (Random.Range(0f, 1f) < 0.3)
        {
            WaveSpacing = 0.3f;
        }

        if (WaveSpacing < 0.3)
            WaveSpacing = 0.3f;
    }

    SpawnTimer += Time.fixedDeltaTime;
    if (WaveTimer <= 0 && EnemiesToSpawn > 0 && SpawnTimer >
WaveSpacing)
    {
        EnemiesToSpawn--;

        SpawnTimer = 0;

        EnemyCreature EC = new EnemyCreature();

```



```

        default:
            Debug.LogWarning("Unknown ServerCode: " + Job);
            break;
    }
}

private static void
PerformServerAction (DZUDPSocket.ReceievePacketWrapper Packet, ServerCode
Job)
{
    switch (Job)
    {
        case ServerCode.SyncPlayers:
            Game.SyncPlayers (Packet);
            break;
        case ServerCode.ClientSnapshot:
            Game.UnWrapSnapshot (Packet);
            break;
        default:
            Debug.LogWarning("Unknown ServerCode: " + Job);
            break;
    }
}
}

```

---

## Testing

---

Video of me testing out the main gameplay: <https://bit.ly/3tHSWuU>

---

## Revisiting objectives:

---

The server should:

1. Establish a connection between multiple clients
 

*The server can handle more than one connection and multiple players on one client.*

  - a) Allow the means for 2-way communication between the server and client
 

*This has been achieved clearly as shown in testing as the server and client both sync their state.*
  - b) A standardised packet format will be designed in the design section of the project
 

*As discussed above, a standard packet format for my protocol has been designed.*
  - c) The server should adjust for packet loss and handle high ping / delayed packets appropriately for both incoming, and outgoing data
 

*As shown in testing the server can adjust for packet loss and handle a high ping from the client, including packet tampering, loss and duplication.*
2. Authorise the clients incoming data about position and client state
  - a) If the server disagrees with the client, the server takes priority for server-authoritative control
 

*As shown in testing, when the client disagrees with the server the positions are snapped back as shown by the rubber banding effect.*



3. Constantly broadcast a snapshot of its current world state to its clients  
*As shown in the testing video a constant stream of world snapshots are being sent to the client in order for it to render the game world.*
  - a) 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  
  
*This is not very relevant to my context as I only need a single tile map for my game.*
  - b) Snapshots are sent once every server tick
4. Calculate physics of entities  
*As can be clearly seen in the video, entities are fully simulated and have the appropriate physics.*
  - a) Players are user-controlled entities that act on standard physics that can be controlled by commands sent via the clients  
*As can be clearly seen in the testing video, the player is a user-controlled entity and acts on normal physics*
5. Receive packets relating to player controls from clients
  - a) 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)  
*This is not very relevant for my context as the players don't perform any other time critical actions other than movement*
6. Read from a configuration file which allows the user to set the server port  
*There is a Server.cfg file of which the server reads from to open a connection from*

#### The client should:

1. Provide a GUI for the user
  - a. This should be usable by a naïve user  
  
*The GUI is self-explanatory for how to connect simply by inputting the IP address and Port into the top right. It can definitely be improved as I don't give a tutorial or any information as to how to play the game or start it.*
2. Host a server on the client pc with given settings / options that the user provides  
*The server can be hosted on the client pc with the provided options in the Server.cfg file*
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  
*As shown by the testing video more than one player can be used per client device*
4. Connect to a server with the IP address and port a user provides  
*As shown by the testing video the server can be connected to via an IP address and Port*

5. Receive packets from the server  
*As shown by the testing video the program clearly receives packets and unwraps them for the game world.*
  - a. Unwrap the packets and generate the snapshot client side for the user
  - b. 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  
*The client interpolates the server snapshots very well as shown in the testing video*
  - c. 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  
*The client accounts for lost packets and high ping and corrects for disagreements smoothly as shown by the video.*
6. Display / Render the player and world onto the screen  
*As clearly shown in the testing video the player is rendered onto the screen*

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

2. The game should have a lobby showing the players that are playing.  
*As shown at the beginning of the testing video all players are visible in the lobby and can start the game by standing on the starting square.*
3. Each playthrough should be procedurally generated.  
*Every time a new game is started a new randomly generated path is made.*
4. Upon entering a level, the game should:
  - a. Display the layout of the path the enemies will take.  
*As clearly shown in the testing video a clear path is shown.*
  - b. Allow players to move around the level.  
*As clearly shown in the testing video all the players can move around in the level*
  - c. Have enemies spawn in waves that follow the path.  
*As clearly shown in the testing video the enemies spawn in waves and follow the yellow path.*
  - d. Abide by standard tower defence rules:
    - i. Enemies spawn at one end of the path and upon reaching the end the player loses (either entirely or some form of health system).  
*As shown in the testing video, enemies spawn at one end of the path and make their way to the end. Upon reaching the end the player loses life.*
    - ii. Players can kill enemies using towers.  
*Towers can shoot the enemies as shown in the testing video.*
    - iii. Enemies drop a collectable resource used in creating towers.  
*Enemies drop coins that the player can pick up to spend on making more towers*
  - e. Tower shots can hurt the players as well.  
*As shown in the testing video getting hit by a tower does damage and can lead to a game over*
  - f. Towers self-destruct at the end of their lifetime.  
*After 150 seconds a tower will destroy itself.*

## Stage 2: Enemy variation

1. When generating each level, the game will also define what enemy types will appear in each level and for which waves.  
*Unfortunately there is only 1 enemy type that has been implemented.*
2. Enemy variation through different game mechanics or wave formats.  
*As clearly shown in the testing video there are various enemy formations with tightly packed waves to very spread out enemies.*
  - a. Enemies that enter in a tightly packed wave.
  - b. Enemies that are spread out.
  - c. Enemies that move faster / in odd patterns.
3. Possible enemy variation through different behaviour.  
*Unfortunately I never got round to implementing various enemy behaviours.*
  - a. Enemies may stagger and stall in “safe areas” along the path where bullets do not cross and hastily cross “dangerous areas” filled with bullets.
  - b. Enemies may move quickly in a straight line but slowly along turns.
  - c. Enemies may wait for other enemies in “safe areas” and stick together before proceeding.

## Stage 3: Quality of life:

1. Settings to customize the gameplay:  
*Unfortunately I never got round to implementing quality of life options.*
  - a. Ability to disable of certain enemy types.
  - b. Game modifiers:
    - i. Slow motion / Bullet time.
    - ii. All homing bullets.
    - iii. Enemies getting through the path instantly cause a loss.
  - c. Ability to disable permanent death.
  - d. Ability to disable different tower types.
  - e. Slow down default game speed.

## Stage 4: Polish:

1. Entity ragdolls (physics-based corpses)  
*As clearly shown in the testing video, upon death enemies drop dead as a corpse.*
2. Particles and effects.
3. Character animations.  
*Characters are lightly animated through physics*
  - a. Inverse Kinematics

---

## *Evaluation – Potential Extensions and Improvements*

---

I have a few suggestions for potential improvements and extensions that could be made to my project:

1. Transition into WAN networks through updating my network protocol to better handle congestion.
2. Implementation for security as currently my packets are sent completely bare and are subject for being impersonated or altered over the network. This is not really a problem as my game will not be global and is just used locally, but it would be an improvement to encrypt my packets.
3. Implementing more enemy and tower variations with new mechanics.
4. Implementing a level designer for custom.
5. Implementing a modding API.
6. Graphics overhaul, upgrading the graphics such as using better fonts and symbols for representing health and improving sprite graphics.
7. A proper game over screen rather than just reloading into the lobby.
8. Implementing a text chat system

---

## *Git Log*

---