VAASAN AMMATTIKORKEAKOULU
VASA YRKESHÖGSKOLA
UNIVERSITY OF APPLIED SCIENCES

Topias Jokiniemi

Unity Networking

Developing a single player game into a multiplayer game

Technology and Communication

2014

VAASAN AMMATTIKORKEAKOULU
UNIVERSITY OF APPLIED SCIENCES
Tietotekniikan koulutusohjelma

ABSTRACT

Author	Topias Jokiniemi
Title	Unity Networking
Year	2014
Language	Finnish
Pages	54
Name of Supervisor	Timo Kankaanpää

This thesis is about developing a game called Servants of Aether into a networked multiplayer game. The game has been in development for eight months before this thesis, and it might take one more year to release. Originally, the game was not designed as networked multiplayer game which makes designing more challenging. Game is being developed with Unity game engine, which has been especially popular among indie developers. Network implementation was made using Unity's built-in network solution.

The most important part in developing networked multiplayer game is synchronization. Everything that happens in the game world needs to look the same for every player. Characters has to move very smoothly and all characters' features, such as attacking needs to be transmitted reliably, and it may not miss the target due latency. In addition to synchronization this thesis focuses on server logic, minimizing the network bandwidth requirement and implementing chatting possibility.

All the requirements of the were implemented, the multiplayer works well and all major error were fixed. Most difficult part of the thesis was by far the interpolation and extrapolation logic, which are also the most important aspects of achieving smooth movement. Both were successfully implemented, although there are certainly room for improvement. Also, the software structure was challenging design, as synchronization everything requires understanding the structure of the game as a whole.

Overall the thesis was extremely interesting and challenging. As a first touch of game networking, the outcome was suprisingly good.

Keywords Unity 3D, Networking, Game development, Chat

VAASAN AMMATTIKORKEAKOULU
Tietotekniikan koulutusohjelma

TIIVISTELMÄ

Tekijä	Topias Jokiniemi
Opinnäytetyön nimi	Unity verkkototeutus
Vuosi	2014
Kieli	Suomi
Sivumäärä	54
Ohjaaja	Timo Kankaanpää

Opinnäytetyössä tehdään pelistä Servants of Aether verkkomoninpeli. Peliä oli ennen tämän opinnäytetyön aloittamista kehitetty kahdeksan kuukautta ja sen valmistumiseen saattaa mennä vielä vuosi. Alun perin peliä ei suunniteltu lainkaan verkkomoninpeliksi, mikä teki toteuteksesta hieman haastavamman. Peliä tehdään Unity pelimoottorilla, joka on erityisesti indie kehittäjien suosiossa. Verkkototeutus tehtiin Unityn sisään rakennetulla verkko ratkaisulla.

Verkkomoninpelin toteutuksessa tärkeimpänä osana on kaiken synkronointi. Kaikki peli maailmassa tapahtuva pitää näkyä samanlaisena jokaiselle pelaajalle. Erityistä suunnittelua vaatii pelihahmojen synkronointi. Hahmojen täytyy liikkua erittäin sulavasti ja kaikki hahmojen ominaisuudet, kuten lyönnit pitää välittyä luotettavasti, eikä ne saa esimerkiksi vasteajan takia mennä ohi kohteesta. Opinnäytetyössä keskitytään synkronoinnin lisäksi palvelin logiikkaan, verkon kaistan käytön minimoimiseen ja keskustelu mahdollisuuden toteuttamiseen.

Opinnäytetyön kaikki tavoitteet saatiin toteutettua, moninpeli toimii hyvin, eikä suurempia virheitä jäänyt korjaamatta. Vaikeimpia asioita työssä oli ehdottomasti interpolointi ja ekstrapolointi, jotka ovat myös tärkeimpiä asioita hahmojen sulavan liikkumisen aikaansaamiseksi. Niiden logiikka saatiin hyvin toimimaan vaikka molemmissa on varmasti parantamisen varaa. Myös ohjelmiston rakenteellinen suunnittelu oli haastavaa, sillä jokaisen asian synkronointi vaatii koko pelin rakenteen ymmärtämisen kokonaisuutena.

Kaiken kaikkiaan opinnäytetyö oli erittäin mielenkiintoinen ja haastava. Ensikosketukseksi pelien verkko-ohjelmointiin työn lopputulokseen voi olla erittäin tyytyväinen.

Avainsanat Unity 3D, Networking, Game development, Chat

TABLE OF CONTENTS

ABBREVIATIONS

Prefab	Ready to use package of components
IDE	Integrated development environment
UI	User interface
GUI	Graphical user interface
P2P	Peer-to-Peer
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
RPC	Remote Procedure call
ID	Identifier
IP	Internet Protocol
VoIP	Voice over Internet Protocol
NAT	Network address translation

LIST OF FIGURES AND TABLES

1 INTRODUCTION

1.1 Motivation

Networking is one of the biggest areas in game development. It is also considered as one of the most technically challenging areas of game programming. As it is a core gameplay design for many games and is used even in many single player games. Learning game networking is big bonus for any programmer looking for job opportunities in game development field.

Networking enables long range multi player for a group of friends. As we are social creatures, being able to share your ups and downs in a game enriches the whole game experience a little.

The game this thesis will be made for is called Servants of Aether, it has multiplayer as a core design in gameplay, making the development of networking more interesting.

1.2 About the game

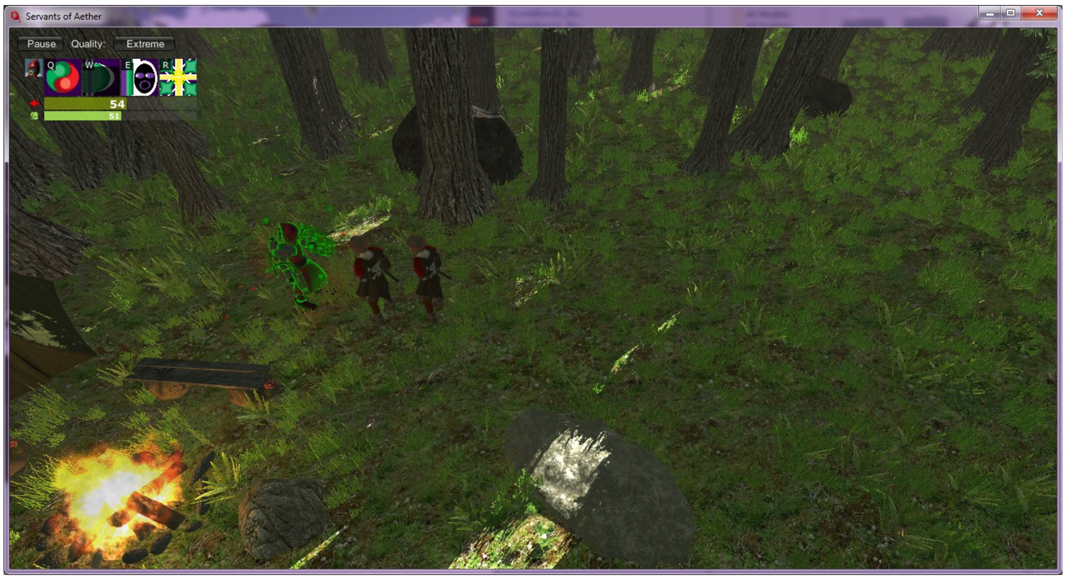


Figure 1. Servants of Aether screenshot

Servants of Aether is a brawler game that focuses on cooperative team play. Players need to slay all enemies on their way to the end of each level. All levels have one or more advanced enemy (boss) fight at the end of each level. The team needs to make preplanned strategies and communicate well to beat every boss fight as the death of a single player may make the fight impossible for the rest. All playable characters are designed to enforce team play, meaning that they all have usable abilities that can be used to support other players.

The players are a mercenary group that is formed from criminals, outcasts and more special individuals who, for some reason, want to put their lives at risk. Often that is it being their only hope for decent life. This mercenary group is specialized for suicidal missions that no one else is willing to attempt.

The story is set in a universe where the Aether, a realm of energy bleeds into the natural world. After significant Aether exposure, the exposed material begins to crystallize forming "Shards". Shards can be thought of as a magical nuclear batteries.

The game is made with Unity game engine. It is currently being developed for Windows only, but there are plans for console ports later. Team doesn't have Unity pro licenses for it which restricts some possibilities. One notable restriction is that it is only possible to use basic Unity’s networking library.

1.3 Team

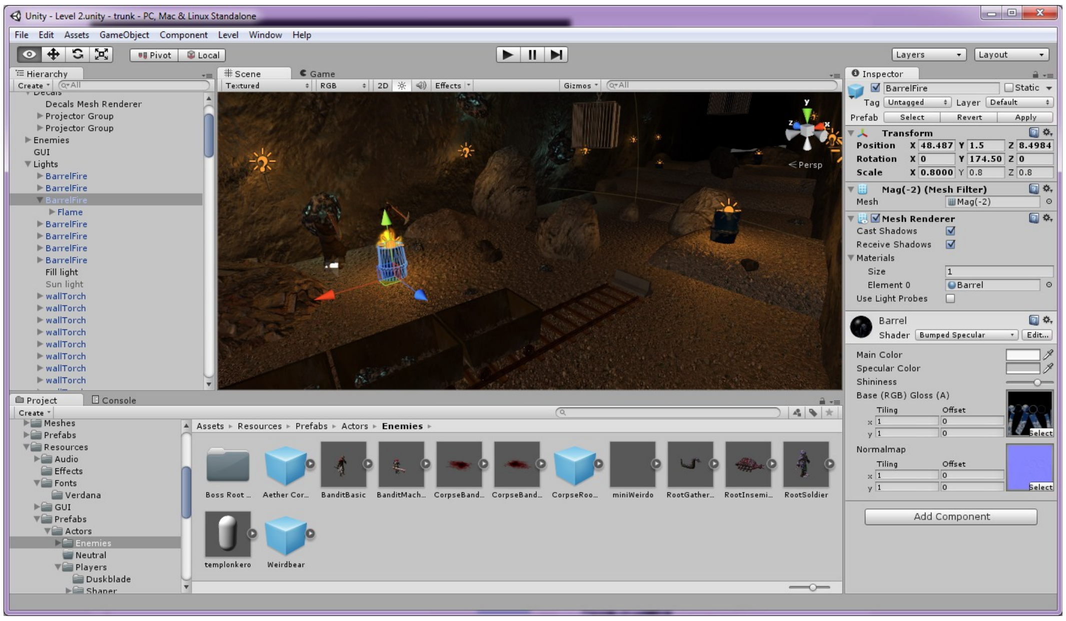
Revon Games is a group of hobbyist game developers working during their free time. Team includes seven members doing everything required for a full 3D game. Programmers, a 3D artist, a writer, a sound designer and a concept artist.

The team was formed at May of 2013. The group found out that they share similar vision and passion for game development. Revon Games is developing games they would love to play and develop further. Most of the group were making games, music or novels before the team was formed, but they know that they can develop something bigger and better as a team than alone.

2 RELEVANT TECHNOLOGIES

2.1 Unity game engine

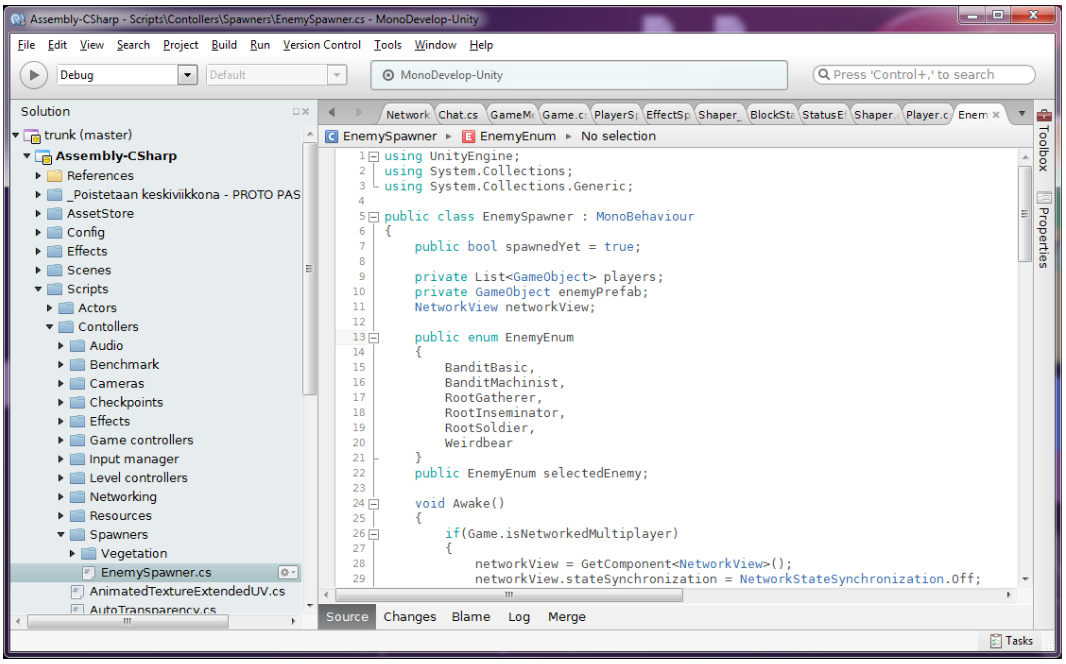
Unity is a cross-platform game engine developed by Unity Technologies. Unity supports deployment to platforms BlackBerry 10, Windows Phone, Mac, Linux, Android, iOS, Unity Web Player, Adobe Flash, PlayStation 3, Xbox 350, Wii U and Wii (Unity 3D, 2014a). Upcoming platforms are PlayStation 4 and Xbox One. (Unity 3D 2014b)



Unity - Level 2.unity - trunk - PC, Mac & Linux Standalone
File Edit Assets GameObject Component Level Window Help
Pivot
Local
Layers
Layout
Hierarchy
Scene
Game
Inspector
BarrelFire
Transform
Mesh Renderer
Project
Console
Assets ▸ Resources ▸ Prefabs ▸ Actors ▸ Enemies
Add Component

Figure 2. Unity user interface

In Unity editor one can build and modify your game. It allows one to easily drag and drop game assets into the game world and modify and edit the game assets. Unity editor can be divided into five different sections: Scene, Game, Hierarchy, Inspector and Project browser. Scene is for editing the game world. Game shows view through main camera, this view is the actual gameplay view. Hierarchy shows all game objects within current scene. Inspector shows information about currently selected object and allows you to modify it. Project browser shows all game assets in a project. (Unity 3D 2014c)


Assembly-CSharp - Scripts\Contollers\Spawners\EnemySpawner.cs - MonoDevelop-Unity
File Edit View Search Project Build Run Version Control Tools Window Help
Debug
Default
MonoDevelop-Unity
Press 'Control+,' to search
Solution
trunk (master)
Assembly-CSharp
References
_Poistetaan keskiviikkona - PROTO PAS
AssetStore
Config
Effects
Scenes
Scripts
Actors
Contollers
Audio
Benchmark
Cameras
Checkpoints
Effects
Game controllers
Input manager
Level controllers
Networking
Resources
Spawners
Vegetation
EnemySpawner.cs
AnimatedTextureExtendedUV.cs
AutoTransparency.cs
Network Chat.cs GameM Game.c PlayersS EffectSp Shaper_ BlockSta StatusE Shaper Player.c Enem
EnemySpawner EnemyEnum No selection
Source Changes Blame Log Merge
Toolbox
Properties
Tasks

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

public class EnemySpawner : MonoBehaviour
{
    public bool spawnedYet = true;

    private List<GameObject> players;
    private GameObject enemyPrefab;
    NetworkView networkView;

    public enum EnemyEnum
    {
        BanditBasic,
        BanditMachinist,
        RootGatherer,
        RootInseminator,
        RootSoldier,
        Weirdbear
    }
    public EnemyEnum selectedEnemy;

    void Awake()
    {
        if(Game.isNetworkedMultiplayer)
        {
            networkView = GetComponent<NetworkView>();
            networkView.stateSynchronization = NetworkStateSynchronization.Off;
```

Figure 3. MonoDevelop development IDE

Scripting in Unity is built on Mono, which is open-source implementation of the .NET Framework. There are three languages programmers can use: UnityScript, C# and Boo. Unity ships with custom version of MonoDevelop for debugging scripts. (Unity 3D 2014d)

2.2 Networking frameworks for unity

Unity has built in networking possibility that uses RakNet library. There are other good frameworks that allow more possibilities, for example: Photon, Ulink, SmartFox and Tnet. Photon is the most used one of external frameworks. Only possibility for this project was using the basic Unity networking since the team does not have Unity Pro license or budget to buy from Unity Asset store. Unity Networking allows to make simple client – server based networking, which is good solution for game like this. (Unity 3D 2014e)

2.3 TCP and UDP

“Transmission Control Protocol is a connection-oriented protocol. This means that every time we want to communicate with a remote host, we must first establish a connection. Once we have established a connection, we do not have to worry about directing the messages we send to the correct place. When we are done with the connection, we must close it. TCP is also a reliable protocol. It makes sure the other end receives the messages we send, and it handles such things as duplicated packages." (Mulholland, 2004, 108)

“Unlike TPC, UDP is connectionless protocol. No actual connection is established between the two communicating hosts. UDP is not reliable, as it does not ensure that the transmission is received. UDP can be used in applications that need the best possible efficiency, but very little reliability. Computer games fall into this category. UDP can be made reliable, but this requires that we write the needed checking algorithms ourselves.” (Mulholland, 2004, 109)

“One big networking decision is selecting protocols to use and how to use them. A common trade off is between using slower TCP/IP for guaranteed delivery or faster UDP for speed.”(Lake 2011, 488) “A recommendation is to use TCP/IP for log in and authentication to guarantee communication, and then use UDP if needed for speed or bandwidth with symmetric key encryption during gameplay.” (Lake, 2011, 489)

2.4 P2P and Client-Server based network

“In a peer-to-peer network, a group of computers is connected together so that users can share resources and information. There is no central location for authenticating users, storing files, or accessing resources.” (Microsoft 2008)

“In a server-based network, the server is the central location where users share and access network resources. This dedicated computer controls the level of access that users have to shared resources. Shared data is in one location. Each computer that connects to the network is called a client computer.” (Microsoft 2008)

2.5 Sending data using Unity Networking

Unity networking uses NetworkView to send data. NetworkView is a component that makes synchronization possible. There are two possibilities to send data. First being State Synchronization. Second one being Remote procedure calls. (Unity 3D 2014f)

2.5.1 State Synchronization

"State Synchronization is the continual sharing of data across all game clients". (Unity 3D 2014h) This is especially good for anything that requires continuous update for its information. For example location, rotation, animation state and so on. Normally the NetworkView will synchronize its locational data, if one wants to add any logic to the movement of GameObject, for example interpolation, one needs to set the NetworkView to observe a script instead of only location.

2.5.2 Remote Procedure Calls

"Remote Procedure Calls (RPCs) let you call functions on a remote machine." (Unity 3D 2014g) RPCs are expecially handy for things that happen from time to time like dying, taking damage, starting an event, using light switch and so on. RPCs even allow storing states for players that connect later using RPC buffer. (Unity 3D 2014g)

3 REQUIREMENTS

3.1 Requirement specification

ID	Requirement	Priority
R1	The players can communicate with text	3
R2	Entity state synchronization	1
R3	Predict entity movement on package loss	2
R4	Connecting server selected from a server list	1
R5	Visual effect synchronization	2
R6	Keep possibility to play single player without networking	2

Table 1. Requirements

Idea is to turn single player game into networked multiplayer game and to focus on the networking aspect of it.

Player needs to be able to connect to a server. This can be made simply so that user types IP address and port number on a field and presses the connect button. Better solution would be if a player can get a list of all public server and select the one he wants to join.

Probable the biggest part of this work will be synchronization of all characters in the game and designing what logic is being run on a server and what on clients. Synchronization should look as smooth as possible.

3.2 Use-case diagram

"Use case diagrams are used to gather the requirements of a system including internal and external influences. These requirements are mostly design requirements." (Tutorialspoint 2014)

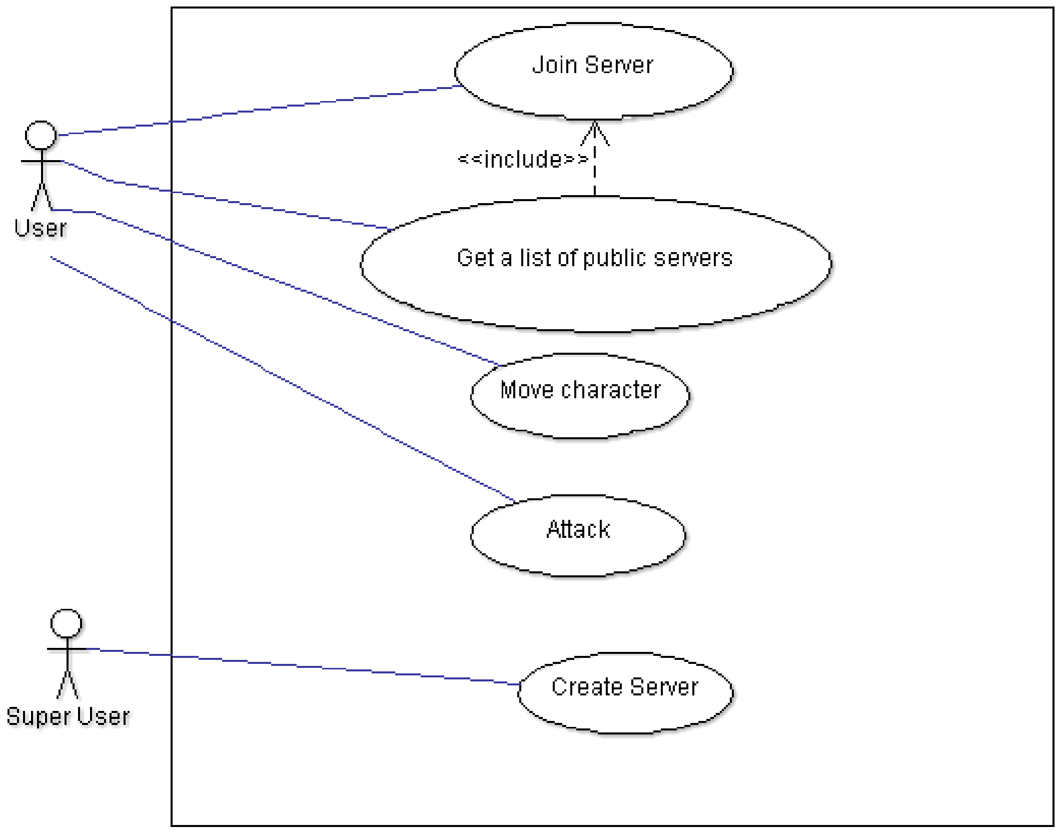


Figure 4. Use-case diagram of the multiplayer functionality

3.2.1 Join server

Player must be able to join a server. Either by joining a server selected from a list of servers or by joining directly with known IP address and port number. The list of servers is retrieved from a master server.

3.2.2 Create a server

Player should be able to create private servers for a group of friend they might have. Servers should have a possibility to choose between private and public games. Private games can be protected with a password or just choose to not be listed on master server.

3.2.3 Move character

Each player has a single playable character they are able to move. These characters have location and rotation that needs to be synched between all clients. Each character has movement animations walking, running and jumping. All of these animations needs to be synched so that other players can know what their teammates are doing.

All levels have enemy characters that also have same features as player characters. Only difference being that enemies are AI controlled. Enemy state needs to be synchronized same way as player characters.

All movement should look smooth even if client loses one or two packages. Smoothing can be made by interpolating and extrapolating locations in client.

3.2.4 Attacking

Players have a set of attacks they can use. Each attack has its own animation and possibly some graphical effects. If an attack hits any enemy character it will deal damage. Enemies have same features as player character does and also needs to be synchronized.

Synchronized animations must look as smooth as possible when transitioning between attack and movement animations. Graphical effects, that are started by a player action needs to be shown on every client. Server needs to know if an enemy character is being hit by a player and server has to synchronize that to every client.

3.3 Package Diagram of the whole game

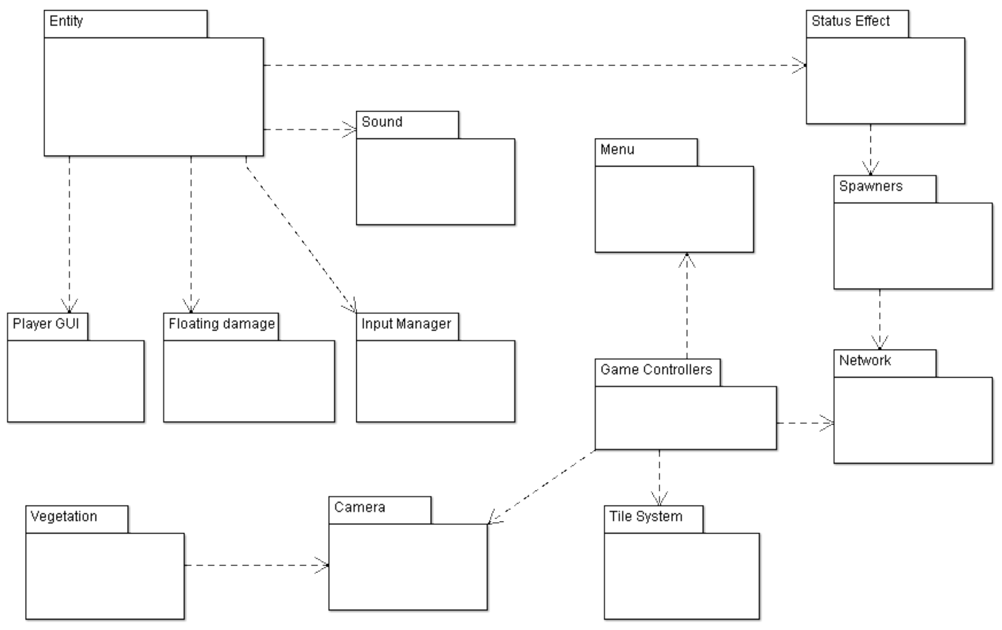


Figure 5. Package Diagram

This is diagram of all packages in the game that are used when the game is running, everything else is irrelevant for networking. All of these need to be synchronized to every player on a server so they can see the same game state.

4 DESIGN

This game wasn't originally designed to be a networked multiplayer game which makes designing the networking more challenging. This is the case for many game projects and that's why networking requires big structural changes to the whole project.

4.1 Authoritative server or non-authoritative server

Authoritative server means that the server will handle all events in a game world. Client only sends their input to the server and the server will decide if the action happens or not, and how it happens. Client continuously sends its input and server will continuously send the result. The client does not make any changes to the game world. (Unity 3D 2014h)

“A potential disadvantage with authoritative servers is the time it takes for the messages to travel over the network. If the player presses a control to move forward and it takes a tenth of a second for the response to return from the server then the delay will be perceptible to the player. One solution to this is to use so-called client-side prediction.“ (Unity 3D 2014h)

Non-authoritative server is the opposite. Client controls its own character movements and actions and sends the result to the server. Server does process what clients are sending, it just broadcasts it to other clients. (Unity 3D 2014h)

This games networking will be designed as non-authoritative server. The biggest reason for this is that it is easier to implement from design perspective, server will only broadcasts other players actions, no other processing on server is required. There is no need for prediction other that making the player movement smooth, in case server loses packages. One big advantage is collision detection, which will be pixel perfect, if a player sees that his attack will hit something, it will hit that something. (Unity 3D 2014h)

Multiplayer games, specifically all competitive ones are mostly made as server authoritative to make cheating harder. For example, clients do not have the possibility of cheating by telling the server that enemy has been killed. Clients will tell the server that they have launched an action, the server will check if it will hit or not. (Unity 3D 2014h)

As seen from new generation games like Counter Strike: Global Offensive, there will be cheaters even though networking is designed as server authoritative.

4.2 Minimizing network bandwidth

To keep required bandwidth as low as possible it is important to not synchronize every single detail trough network to make objects appear synchronized. It is better to to send as little as possible and use a lot of client side logic to make objects appear synchronized. For example, sound effects can be played after entity takes damage, instead of sending the information to play sound. Animations that are playing shouldn't be sent as strings, better idea make list of the animations and only send index of the list and as an integer. (Unity 3D 2014h)

4.3 Synchronization described visually using package diagram

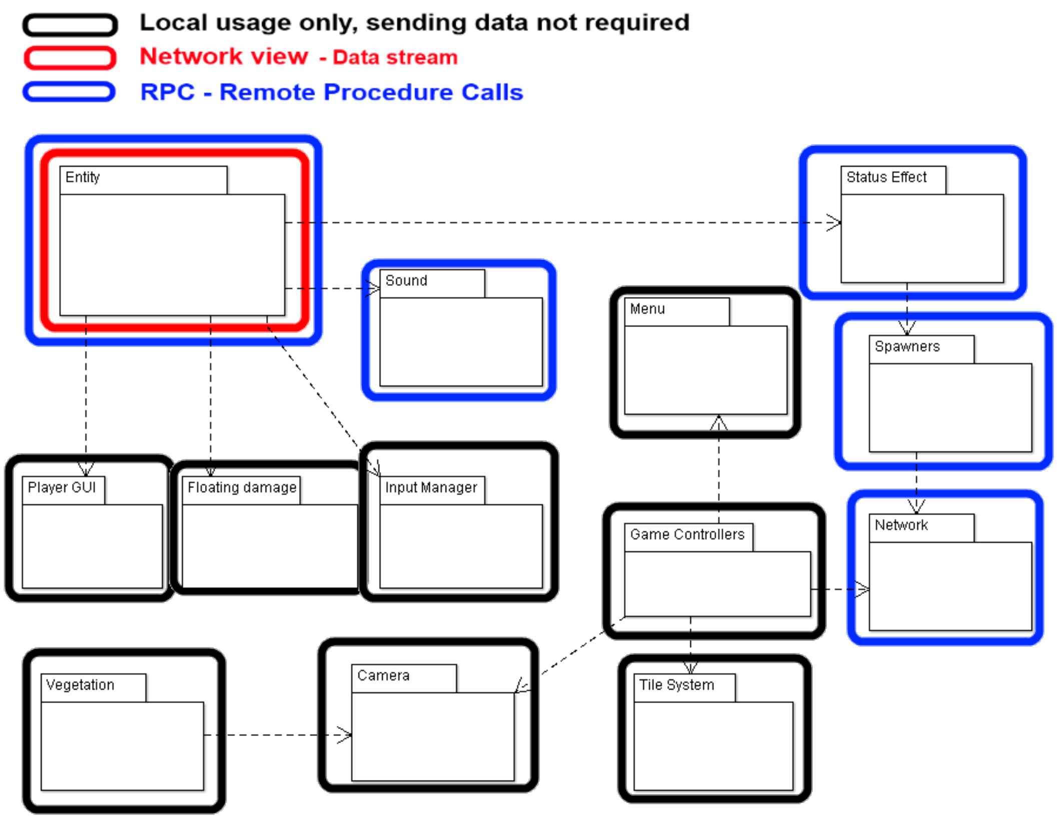


Figure 6. Synchronization design

Figure 6. shows how each component needs to be synchronized. Most of the synchronization will be made using RPCs only, but entity needs data stream for its locational changes.

4.4 Architectural design of communication

4.4.1 Entity component

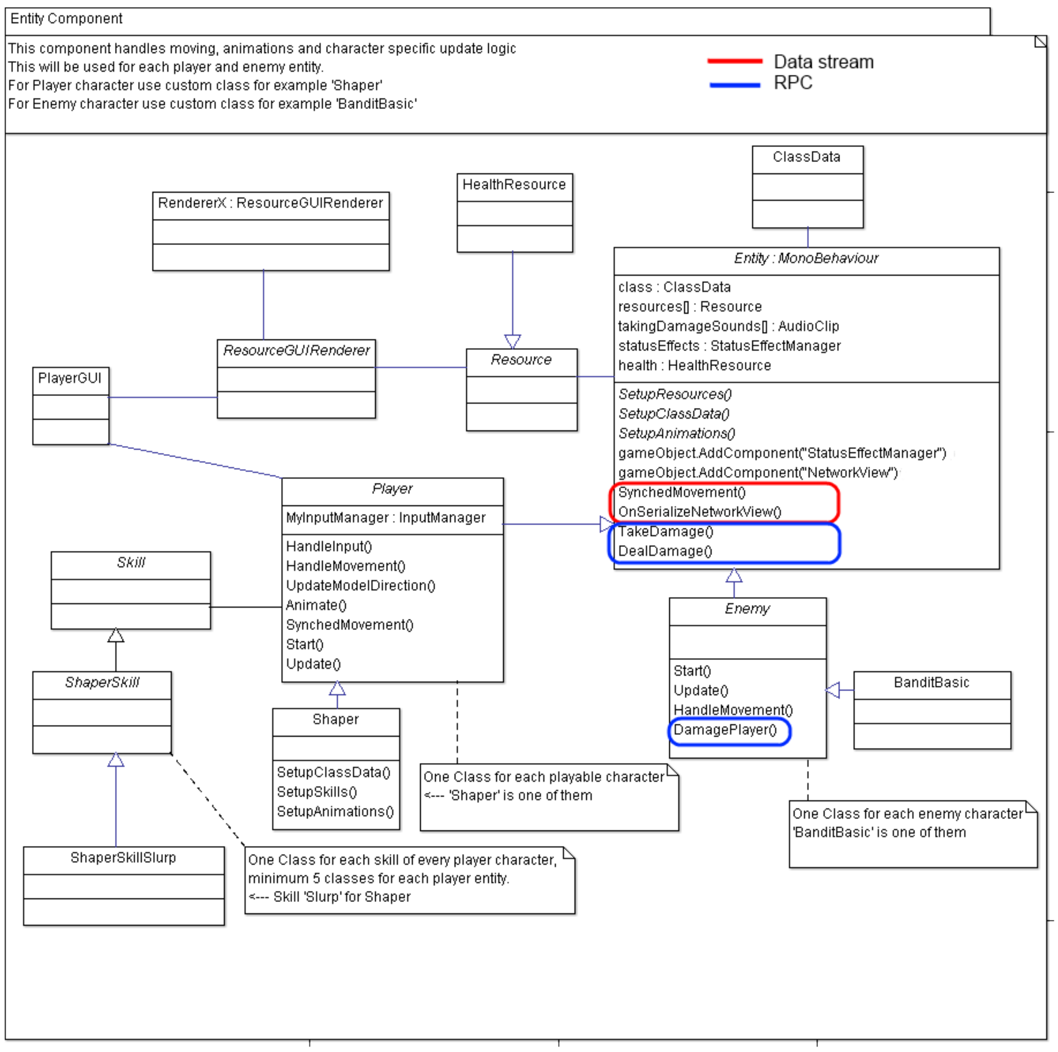

Entity Component
This component handles moving, animations and character specific update logic
This will be used for each player and enemy entity.
For Player character use custom class for example 'Shaper'
For Enemy character use custom class for example 'BanditBasic'
Data stream
RPC
ClassData
HealthResource
RendererX : ResourceGUIRenderer
Entity : MonoBehaviour
class : ClassData
resources[] : Resource
takingDamageSounds[] : AudioClip
statusEffects : StatusEffectManager
health : HealthResource
SetupResources()
SetupClassData()
SetupAnimations()
gameObject.AddComponent("StatusEffectManager")
gameObject.AddComponent("NetworkView")
SynchedMovement()
OnSerializeNetworkView()
TakeDamage()
DealDamage()
ResourceGUIRenderer
Resource
PlayerGUI
Player
MyInputManager : InputManager
HandleInput()
HandleMovement()
UpdateModelDirection()
Animate()
SynchedMovement()
Start()
Update()
Skill
Enemy
Start()
Update()
HandleMovement()
DamagePlayer()
BanditBasic
ShaperSkill
Shaper
SetupClassData()
SetupSkills()
SetupAnimations()
One Class for each playable character
<--- 'Shaper' is one of them
One Class for each enemy character
'BanditBasic' is one of them
ShaperSkillSlurp
One Class for each skill of every player character,
minimum 5 classes for each player entity.
<--- Skill 'Slurp' for Shaper

Figure 7. Entity component

Entity component is used in every character. Its purpose is calculate health points, handle animations, react to user input and surrounding. Each player and enemy character has its own class that is attached to the prefab as component. As an example: Bandit Basic and Shaper. Each player character also has its own Skill class, (for example ShaperSkill) that handles character specific base code. Each different skill has its own class for skill specific code, as in this example Slurp.

Entity is the only component that needs to transfer data streaming. Position, rotation and animations will be streamed. In addition entity needs to be able synchronize dealt damage, projectile shooting and various visual effects. This will be done with RPCs. Each enemy and player character might have some unique characteristics that will need character specific solutions.

OnSerializeNetworkView() - Used for sending own data and receiving data sent by others. This will give all the data for SyncedMovement function to use.

SyncedMovement() - Moves entity with given data, using interpolation and extrapolation logic. This is used only for the entities the player is not in control of.

Example:

```
//controlling myself
if(networkView.isMine)
{
      InputMovement();
}
//controlling other entity with given data
else
{
      SyncedMovement();
}
```

Locational data needs to be interpolated. Since server is new data 30 times per second the movement wouldn't look smooth without smoothing between old and new location. The time used for smoothing needs to be at least 1 second / 30 = 33.3ms, but its better to use this amount times two or three in case a package is lost or late.

In case a package is lost or late it is possible to extrapolate entity movement. This means predicting future, where the entity would have moved most likely, if the data was not lost. Most basic extrapolation is to assume that the entity would continue moving same direction it was moving on latest data.

4.4.2 Status effect component

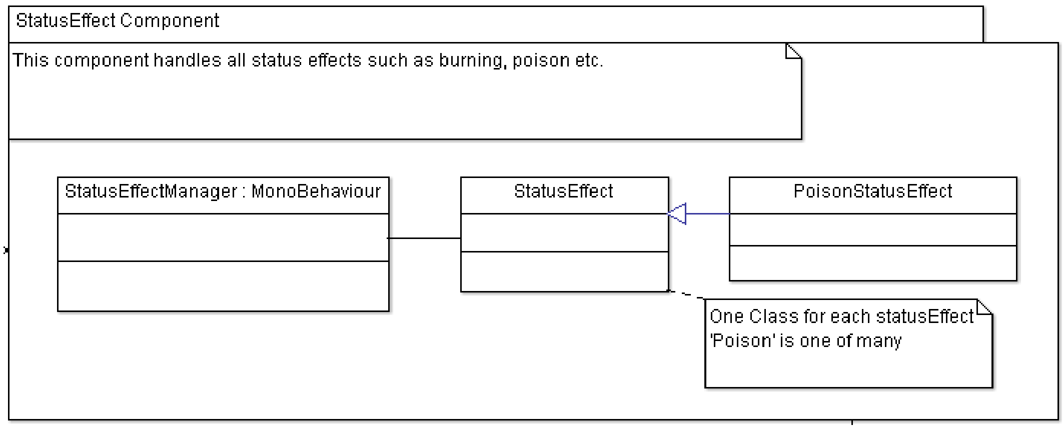

StatusEffect Component
This component handles all status effects such as burning, poison etc.
StatusEffectManager : MonoBehaviour
StatusEffect
PoisonStatusEffect
One Class for each statusEffect
'Poison' is one of many

Figure 8. Status effect component

Status effect component handles different effects that might affect the character such as burning, poisoning. Each effect has its own class that handles its own logic. StatusEffectManager manages and keeps track of each effect on. Status effect component is added into Entity component in code, so its used in every character.

In networked multiplayer this component has to send information what status effects are affecting the character. Status Effect information can be sent by RPCs. It will be enough to know starting time and the duration. Ending time can be calculated.

4.4.3 Sound components

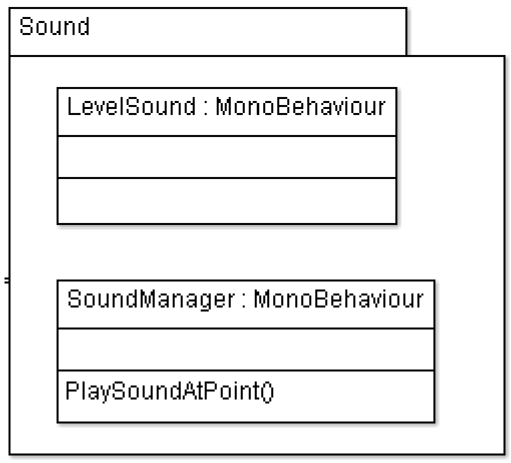


Figure 9. Sound component

LevelSound is used for location triggered sound effects. For example bird chirping near a tree or any ambient sounds of the game world. This is not used for music or character specific sounds. Main purpose for this is to create a better atmosphere and to liven up the world. This doesn't need to send anything to the server, but it should be possible, if such requirement comes in later development of the game.

SoundManager is used to play all sounds in the game. Most important function is 'PlaySoundAtPoint()' that allows to play a sound at any point in 3D space, making it possible to quiet a sound logarithmically or linearly in proportion to distance. SoundManager also allows playing of 2D sounds like music, thus sounds that don't require position in 3D space. In networked game almost every 3D sounds need to be synchronized.

Sound components do need possibility for RPCs. It might be possible to play the sounds using some kind of logic to reduce required bandwidth. Still there will be some cases where RPC possibility is required. It should be enough to mark PlaySoundAtPoint function as RPC.

4.4.4 Input system

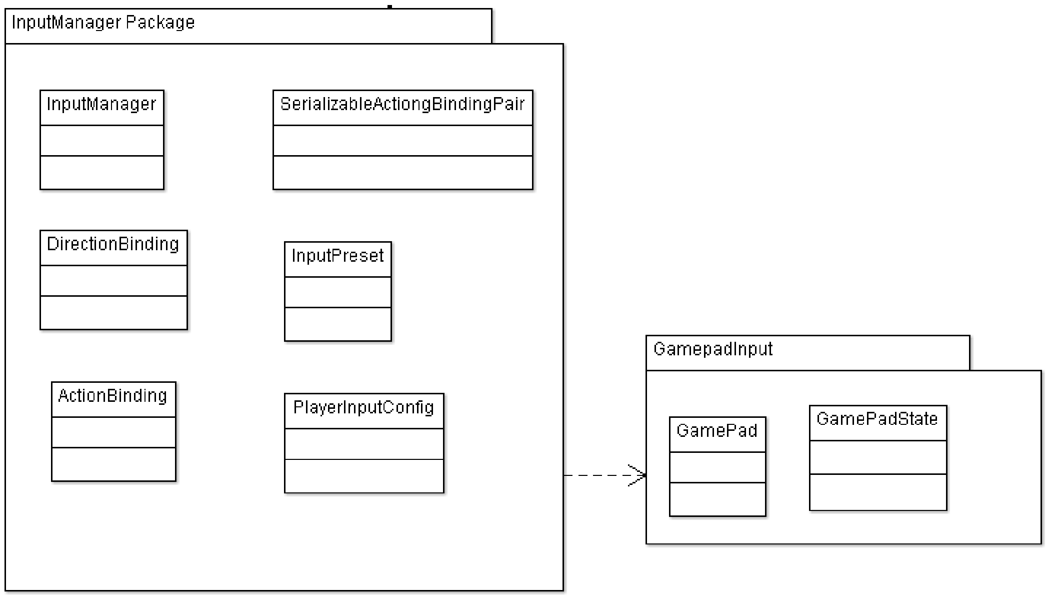


Figure 10. Input system

Input Manager handles all input reading and makes it possible for user to customize key bindings according to personal preference. GamepadInput makes it possible to use gamepads. Input system doesn't require possibility to send anything to servers since its only used for local input reading.

4.4.5 Player GUI

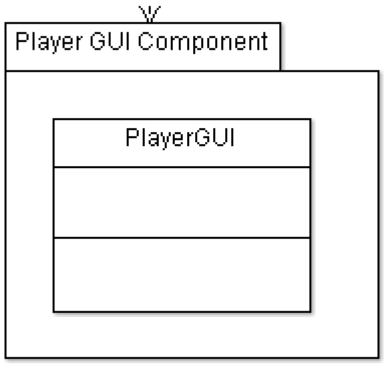


Figure 11. Player GUI component

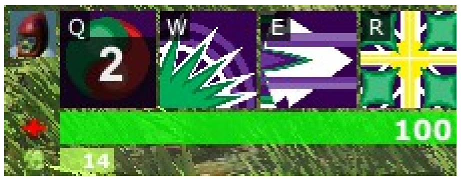


Figure 12. Player GUI in game.

Player GUI shows visually all the information player needs to know like health points and skill cool downs. It is not necessary for this component to send any data since all resource data is located in Entity and this only gives them visual presentation.

4.4.6 Floating damage text component

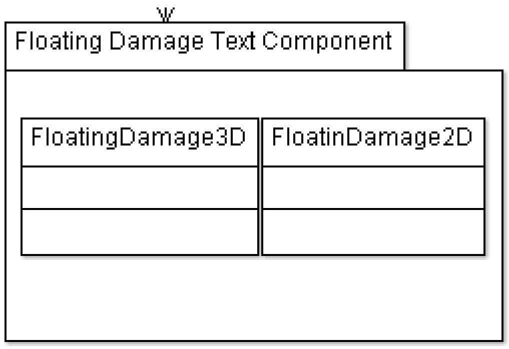


Figure 13. Floating damage text component

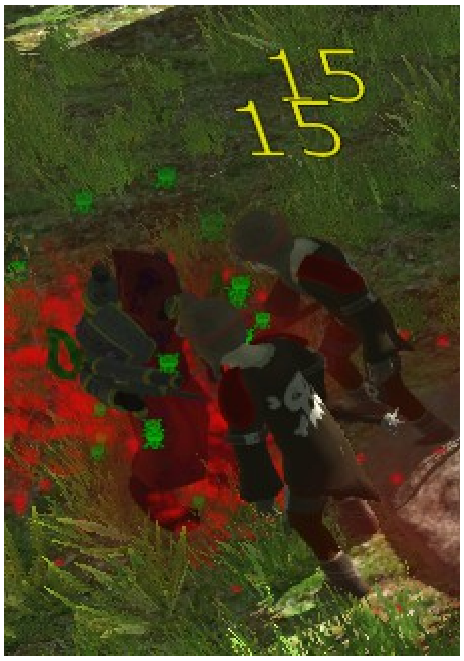


Figure 14. Floating damage text in game

Floating Damage Text Component is used to show dealt damage visually. This component does not require sending of data since deal and receive damage functions are located in Entity component and this can be used locally.

4.4.7 Tile system

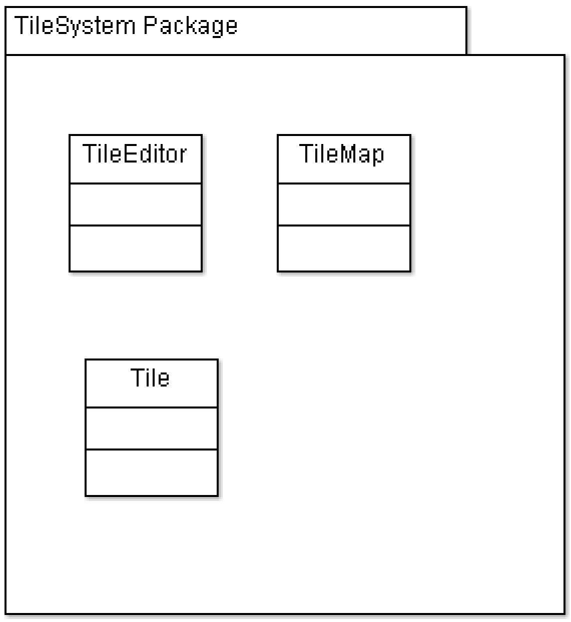


Figure 15. Tile system

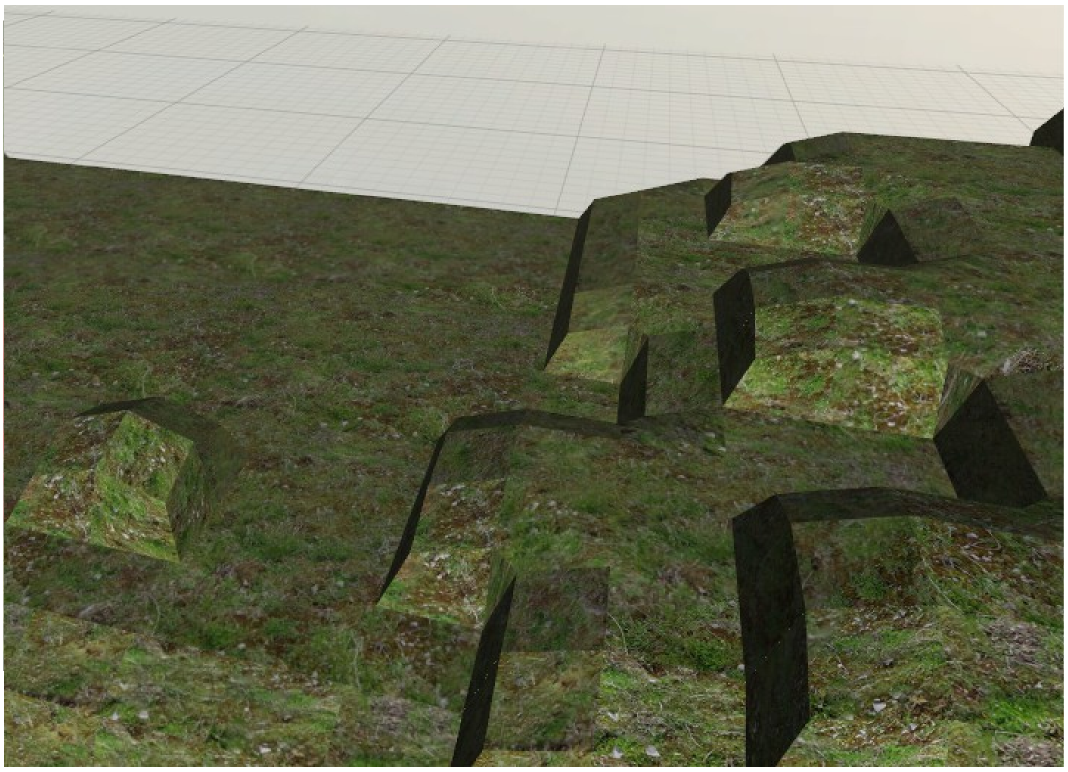


Figure 16. Tile system in game

Tile system is used to create the base of the level using 1 x 1 x 1 large cubes. This doesn't require any networking since it is loaded locally for each player.

4.4.8 Camera package

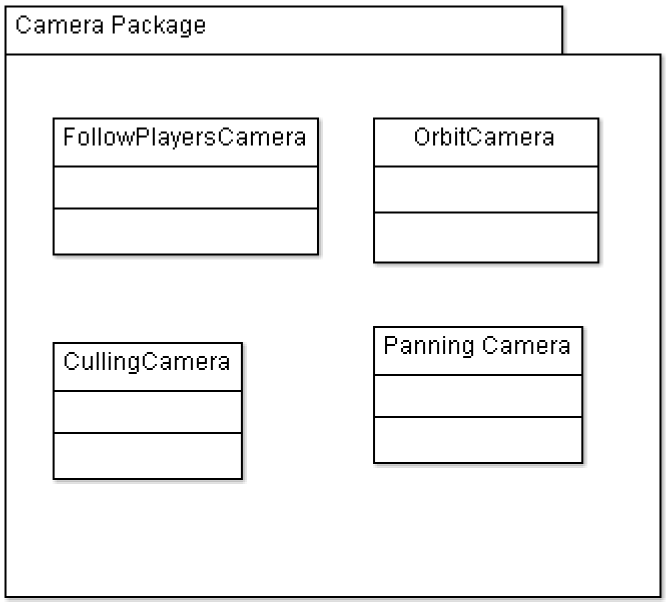


Figure 17. Camera Package

Cameras are the devices that capture and display the world to the player. One of these scripts is attached to the main camera in a scene. CullingCamera is the script used during normal gameplay others currently not in use. Camera scripts control how the main camera is moved around. Camera is run only locally.

4.4.9 Game controllers

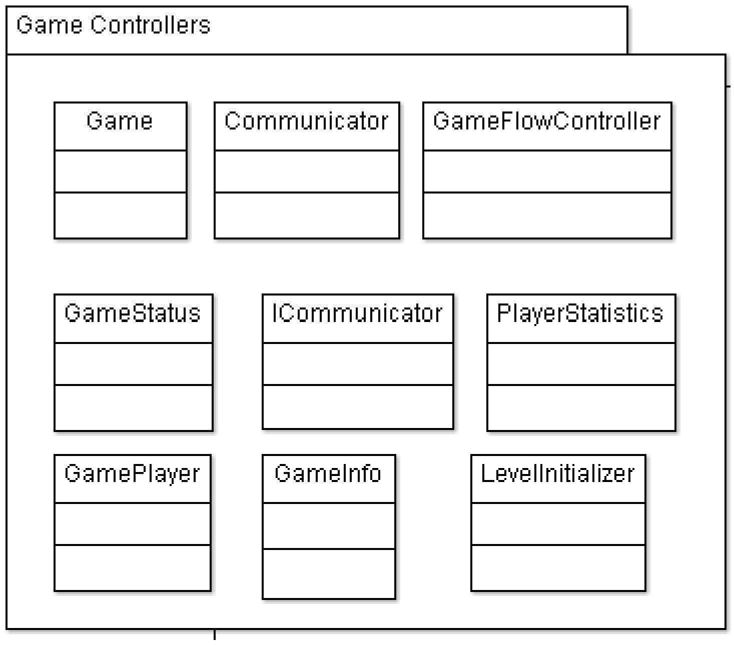


Figure 18. Game Controllers

Game controllers handle all game logic for example changing levels and keeps track of everything. Game controllers needs to know if the game is networked multiplayer or not and control the game accordingly. LevelInitialization logic will be changed to add network component when the game is networked.

4.4.10 Menu package

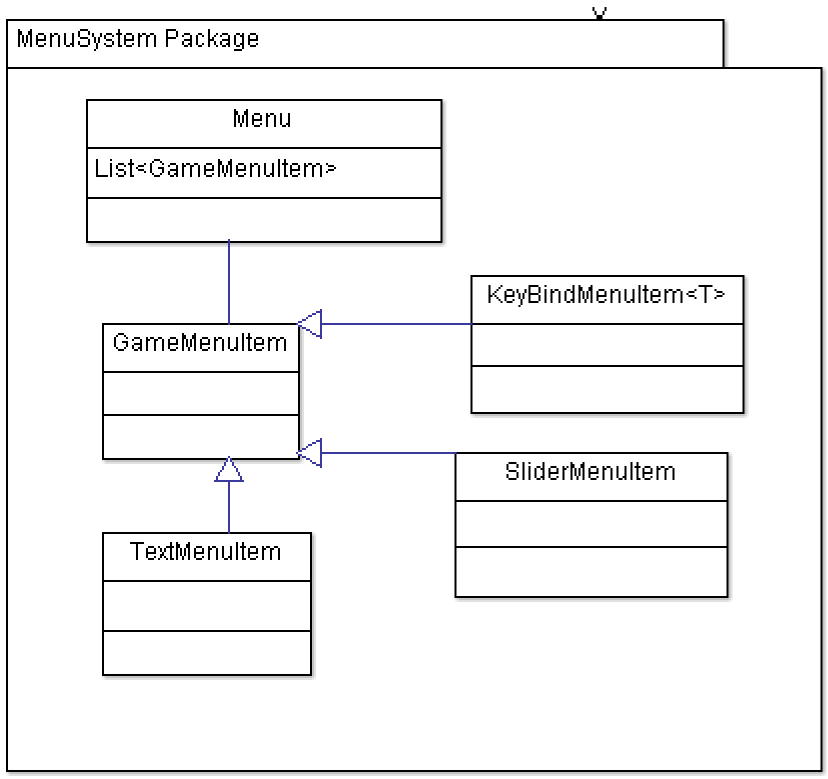


Figure 19. Menu System

Menu system handles logic behind menus. Menus are used to choose between single player and multiplayer game. If multiplayer game is chosen this will be used to either host or join a game server. Game servers will be listed visually and players can choose the game that fits their needs the best.

4.4.11 Vegetation generator

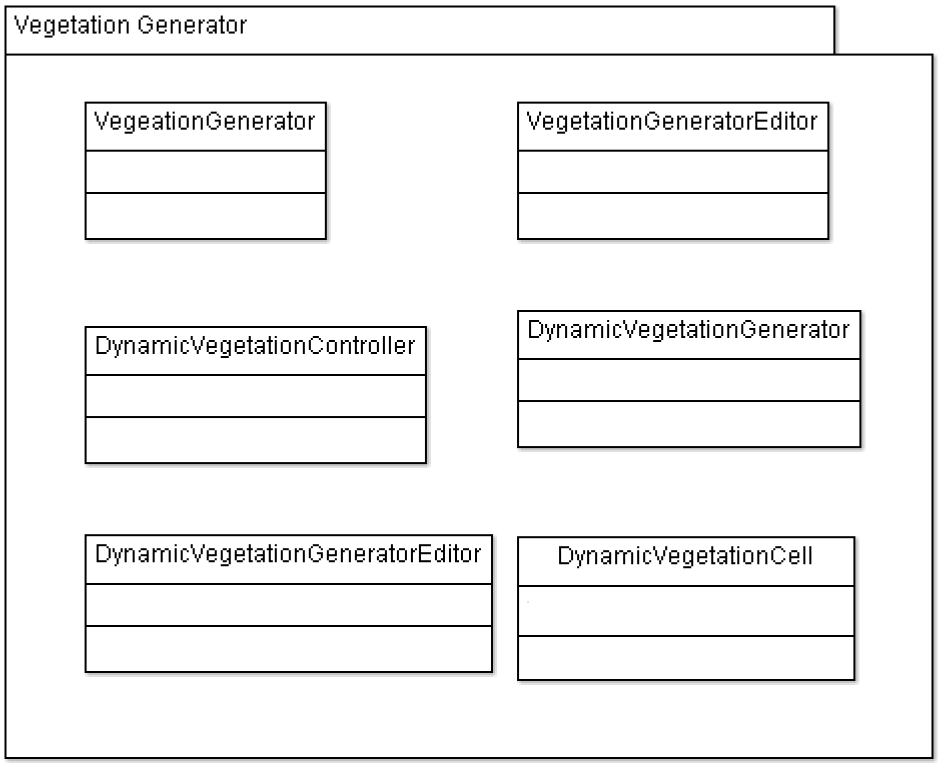

Vegetation Generator
VegeationGenerator
VegetationGeneratorEditor
DynamicVegetationController
DynamicVegetationGenerator
DynamicVegetationGeneratorEditor
DynamicVegetationCell

Figure 20. Vegetation generator

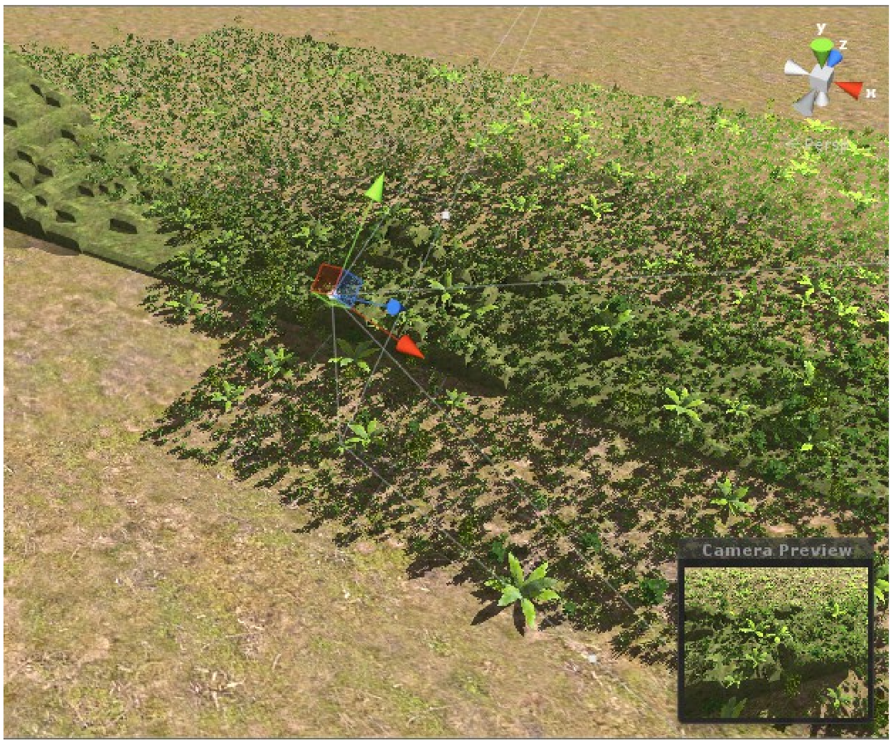


Figure 21. Vegetation generator in game

Vegetation Generator makes possible to place plants randomly.

DynamicVegetationGenerator adds plants automatically around player and deletes them when player leaves the area. This doesn't require any networking logic.

4.5 New components

Most of the code will be done in already existing components, but couple new ones are required.

4.5.1 Network Manager

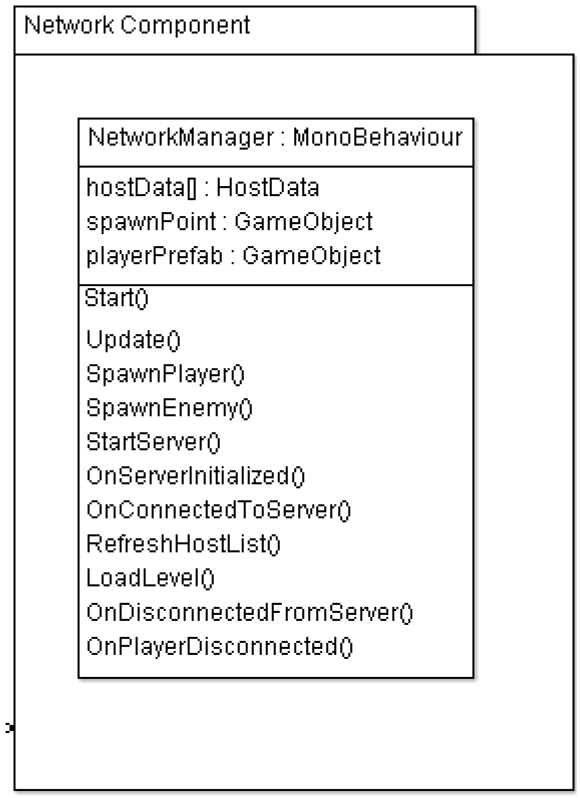


Figure 22. Network Manager design

The purpose of this component is to start server, add players to the server, handle lost connections and to take care that the players have downloaded all files needed for the level. NetworkManager will only be loaded when player wants to play networked game. This information will be taken from user input in main menu.

NetworkManager will inherit MonoBehaviour, which will provide Update() and Start() functions. This will also meant that NetworkManager must be added into GameObject as Component.

- SpawnPlayer() - Adds a player character into the server and adds NetworkView to the player. Making the GameObject send stream of locational data that other clients can receive.

- SpawnEnemy() - Adds an enemy into the server. Server will take the control of character and sends locational data to every client.

- StartServer() - Initialized server and registers it to the master server.

- RefreshHostList() - used to get server list from master server.

4.5.2 Enemy Spawner

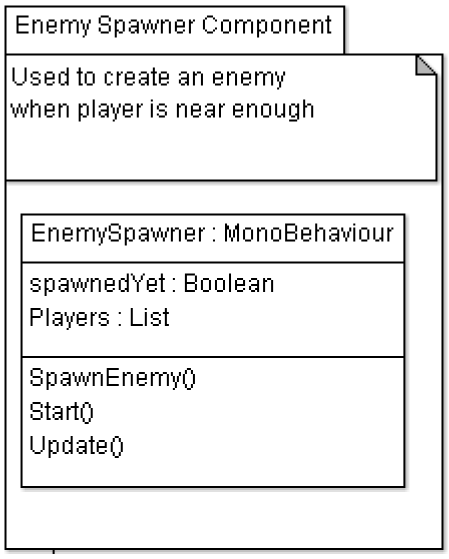


Figure 23. Enemy Spawner component design

EnemySpawner component is used to instantiate enemies when players are close enough. This range should be adjustable. This component is required to make different instantiation logic for single- and multiplayer. This will attached to a Game Object and used as prefab that can be drag-and-dropped directly from Unity inspector into any scene. Most of this script will only run on server side and will be disabled on client side.

- SpawnEnemy() - Will be the RPC function that will be sent to everyone.

4.5.3 Chat Component

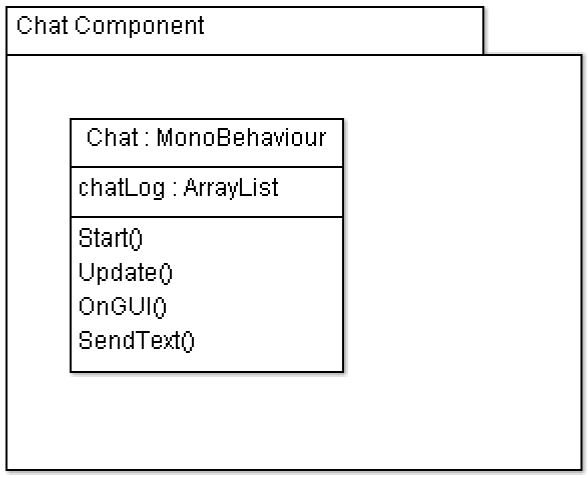


Figure 24. Chat component design

Chat component makes in game chatting possible. Chat needs to have possibility for scrolling up and down to read older messages.

- OnGui() - Handles the graphical presentation of the chat window.
- SendText() will be RPC function that is used to text to all clients.

4.5.4 Effect spawner

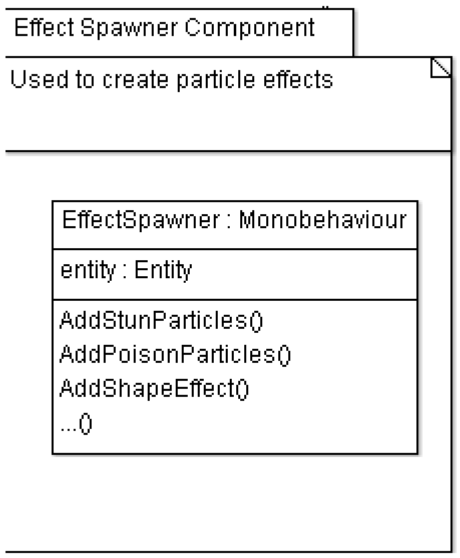


Figure 25. Effect spawner design

Effect spawner is used to instantiate all visual effects in the game. Most of the visual effects do not need its own RPC function as they can be started at the same time some other event happens.

4.6 Sequence diagrams

4.6.1 Start server sequence

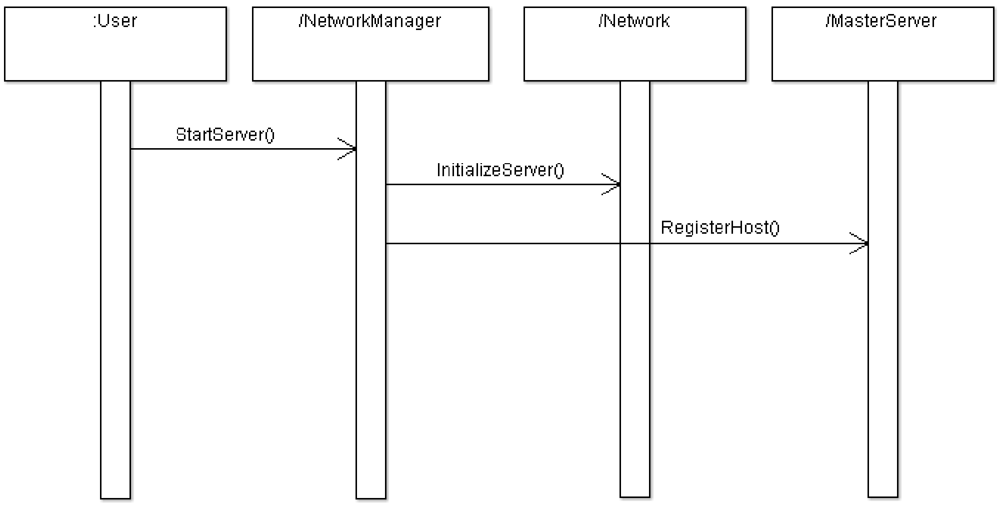


Figure 26. Start server sequence diagram

Figure 26. shows how server is started and what classes are used. User starts the sequence by pressing the start server button. Network Manager handles the logic, it used InitializeServer() function from Network to initialize the server. After initialization. Server will be registered to the master server using RegisterHost() function in MasterServer

4.6.2 Join server sequence

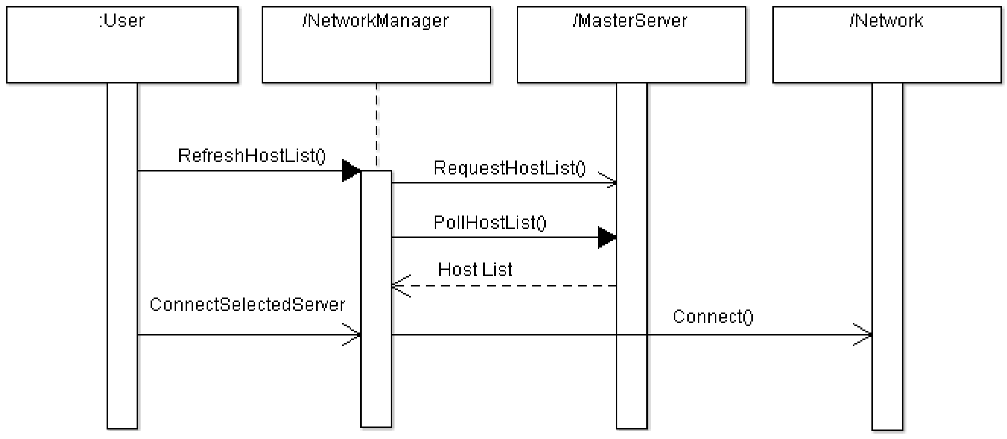


Figure 27. Join server sequence diagram

Figure 27. presents how joining to a server works. User will start it by pressing Refresh button. NetworkManager handles this and requests the list from master server. After the request master server needs to be polled until server list is sent back. User selects from available servers and connects one of them. Connecting will be done using IP address and port number taken from the selected host.

4.6.3 Update entity state sequence

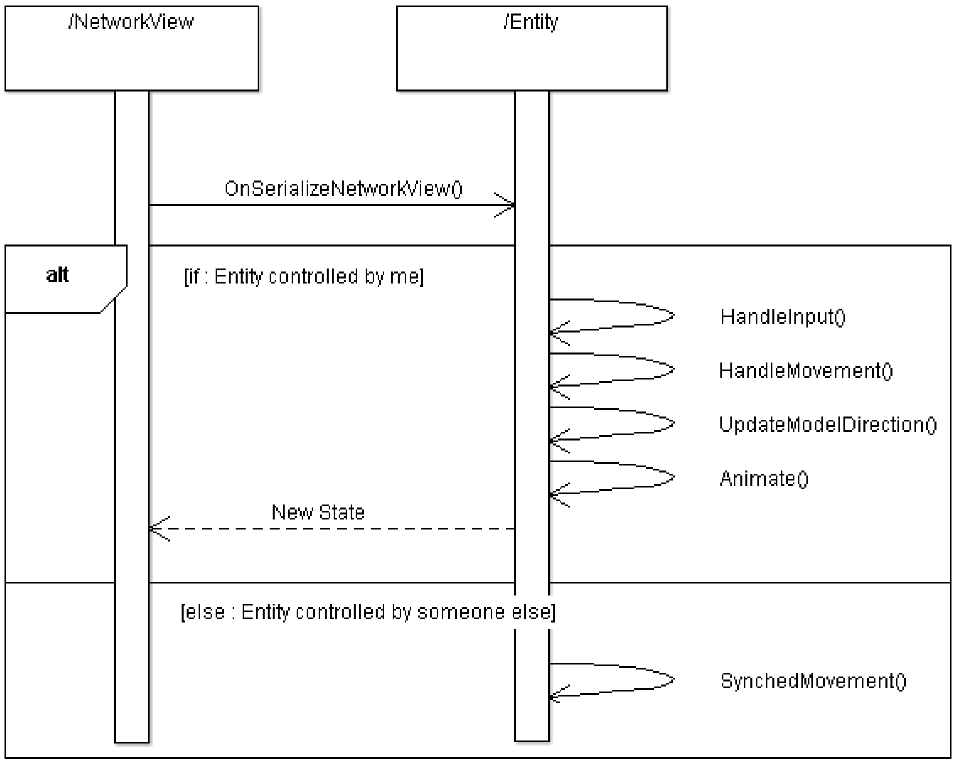


Figure 28. Update entity state sequence diagram

Figure 28. shows how entity state is being updated. Multiplayer game will always have huge amount of entities and all but one will be synchronized using SynchedMovement(). The entity that the client is controlling will be sending its state at the speed of servers send rate, that is 30 times per second in this project.

4.6.4 Instantiate entity sequence

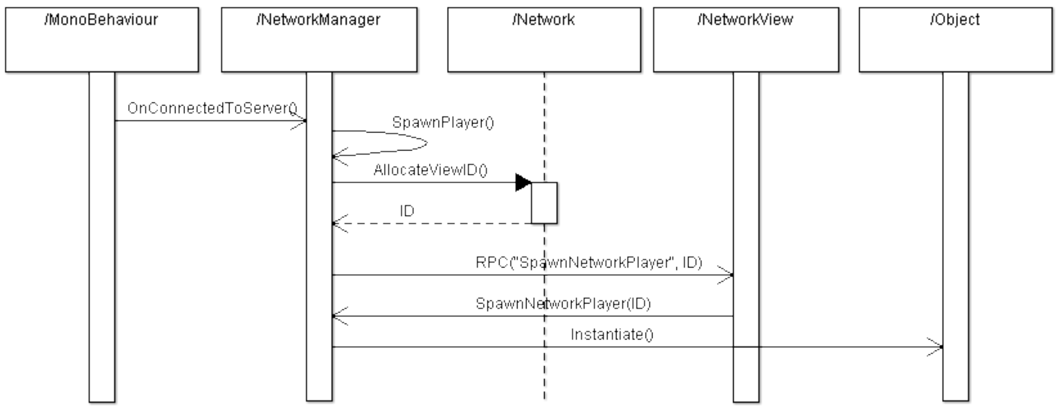


Figure 29. Instantiate entity sequence diagram

Figure 29. is an example of SpawnPlayer RPC. This RPC happens every time new player joins the server. SpawnPlayer RPC is always used by client. This allows client to take control of the entity.

Sequence is started by OnConnectedToServer event that happens on client side when connecting has completed. NetworkManager will use SpawnPlayer() function that allocates one free network ID and uses RPC function SpawnNetworkEnemy() with the ID as parameter. The needs to be sent so that every client will have same ID and synchronization can work properly.

5 IMPLEMENTATION

5.1 Network and server management

Network Manager is the main class to handle all the networking logic. It offers functions to change playable level and creating and joining into servers. Network Manager also handles all required server logic like cleaning up after player has left the server and reconnecting after losing connection to server. Network Manager is only added into a scene when multiplayer game is selected.

User will try to reconnect to the server when connection is lost to the server. Client will try reconnecting once ever 5 seconds for maximum of 10 tries.

```
/* This event will happen when disconnected from a server */
void OnDisconnectedFromServer(NetworkDisconnection info)
{
      if (info == NetworkDisconnection.LostConnection)
      {
            Debug.Log("Lost connection to the server");
            reconnectTries = 0;

            //Starts the reconnection
            InvokeRepeating("Reconnect",
                  reconnectInterval,reconnectInterval);
      }
}

/* tries to reconnect to the server
* and cancels reconnect progress if
* it has tried 10 times already */
void Reconnect()
{
      NetworkConnectionError connectionError;
      connectionError = Network.Connect(joinedGame);

      reconnectTries++;
      if(reconnectTries > reconnectMaxTries)
      {
            CancelInvoke("Reconnect");
      }
}
```

Cleaning after a player disconnection is done by removing all RPCs the player buffered to the server and destroying all player objects. If this wasn't done all player made objects would stay on the server.

```
/* Cleaning after a player disconnects*/
void OnPlayerDisconnected(NetworkPlayer player)
{
      Debug.Log("Clean up after player " + player);
```

```
	Network.RemoveRPCs(player);
	Network.DestroyPlayerObjects(player);
}
```

5.2 Synchronization of entity instantiation

Instantiating an entity into game server was bit tricky. There were problems with serialization of each entity. NetworkViews synchronizes ID automatically only if it is already attached to the prefab. Entities are instantiated with RPC, and the RPC had to be buffered so that everyone that joined the server late could still see everything.

One working possibility was to attach NetworkView manually to each entity prefab in whole game. Since this game is still being developed and new characters are being made weekly, this wasn't a good idea.

Better solution was to add NetworkView in code. This made another problem with serialization. All clients gave different ID for the entity making it unable to be synchronized. Solution for this problem was to send ID with RPC as a parameter.

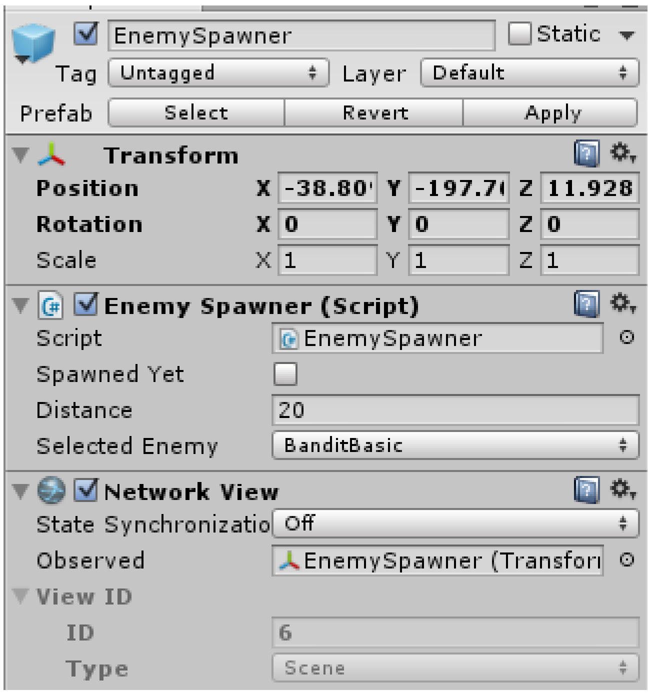
5.2.1 Instantiating players

Players had to send the RPC themselves so that they could gain control of the entity.

```
/*function used to instantiate player character */
void SpawnPlayer()
{
	NetworkViewID viewID = Network.AllocateViewID();
	this.networkView.RPC("SpawnNetworkPlayer", RPCMode.AllBuffered, 
viewID);
}

/* RPC used to instantiate player character for every client */
[RPC] void SpawnNetworkPlayer(NetworkViewID viewID)
{
	GameObject clone;
	clone = Instantiate(playerPrefab, spawnPoint.transform.position, 
Quaternion.identity) as GameObject;
	NetworkView nView;
	nView = clone.GetComponent<NetworkView>();
	nView.viewID = viewID;
}
```

5.2.2 Instantiating enemies



EnemySpawner ☐ Static
Tag Untagged Layer Default
Prefab Select Revert Apply

Transform
Position X -38.80 Y -197.7 Z 11.928
Rotation X 0 Y 0 Z 0
Scale X 1 Y 1 Z 1

Enemy Spawner (Script)
Script EnemySpawner
Spawned Yet ☐
Distance 20
Selected Enemy BanditBasic

Network View
State Synchronizatio Off
Observed EnemySpawner (Transfor
View ID
ID 6
Type Scene

Figure 30. Enemy spawner prefab inspector view

Instantiating enemies were done in new component EnemySpawner. EntitySpawner will track the distance between players and EnemySpawner. Instantiating an enemy will be done same way as player entity. Biggest difference is just that server will take the control.

```
/* Used to instantiate selected enemy character*/
public void SpawnEnemy()
{
	spawnedYet = true;
	//Only instantiate to server when the game networked
	if(Game.isNetworkedMultiplayer)
	{
		//Only server starts the RPC
		if(Network.isServer)
		{
			NetworkViewID viewID = Network.AllocateViewID();
			this.networkView.RPC("SpawnNetworkEnemy", 
	RPCMode.AllBuffered, viewID);
		}
	}
	//Not multiplayer game
	else
	{
		Instantiate(enemyPrefab, this.transform.position,
Quaternion.LookRotation(Vector3.left)) as GameObject;
        }
}

/* RPC to instantiate enemy to every client */
[RPC] void SpawnNetworkEnemy(NetworkViewID viewID)
{
        GameObject clone;
        clone = Instantiate(enemyPrefab, this.transform.position,
Quaternion.LookRotation(Vector3.left)) as GameObject;
        NetworkView nView;
        nView = clone.GetComponent<NetworkView>();
        nView.viewID = viewID;
}
```

5.3 Synchronization of entity component

5.3.1 Movement

Movement will be synchronized using OnSerializeNetworkView function that is used to stream data in Unity's build in networking. Send rate is 30 times per second, new package will on average once every 33,3 milliseconds. This is not enough to make movement look smooth, characters would be warping around.

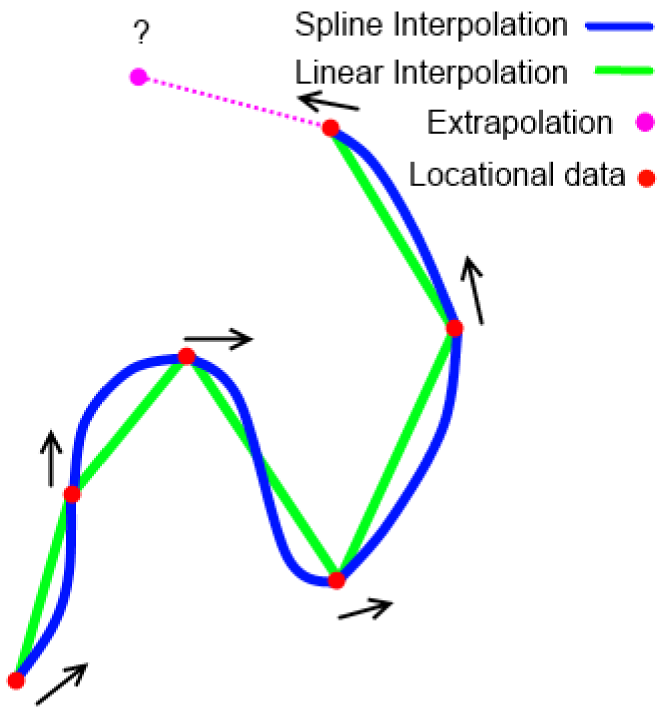


Figure 31. Visual example of interpolation and extrapolation

To fix there is a need for interpolation. Interpolation means calculating values between known points. Most basic interpolation logic is linear interpolation which draws straight lines between known points. To improve this logic the solution was to try to use character velocity and direction it is headed and round up corners using quadratic Bézier curve. After it was working it was found out that Bézier curve was not suited for this because character was using more time at early points of the curve and moving fast at the end of the curve. It was decided to use Linear interpolation instead as it was working well enough. One possible way is to use cupid splines to get better effect, this was not tried for this project and is good next step for future improvement.

Package loss will make character stop until new package has arrived, resulting as jittering and loss of smoothness. To fix this extrapolation was used. Extrapolation is the process of estimating, beyond the given range. Simplest way is to assume character continues moving towards same direction and at the same speed it was headed in last known data. It was decided that the game should look smooth even if three packages are lost. This means extrapolating for 3 * (the delay between packages) = 100 milliseconds.

5.3.2 Animations

Animations are bit tricky, since some animations are only moving upper body of a character and some animations are moving both upper and lower body. Upper body only animation are prioritized and played together with full body animations to reduce amount of animations needed. This means it is needed to send two animations on every update. For example upper body does punch animation and lower body does running animation.

Animations are played using name of the animation as a string, but sending long strings 30 times per second will take bit too much bandwidth. Instead it is better to keep animation names in a list and stream its index.

5.3.3 Dealing and receiving damage

Game was designed to have non-authoritative servers meaning collision detection will be made in client side and if attack hits something the damage and result will be sent to other clients. This will make required logic much simpler since no client side prediction is required and collision detection will be always pixel perfect.

Solution was to use remote procedure call (RPC). All the checking if character was hit or not was left unchanged. To minimize required bandwidth this was also used to start blood splatter effect and victims sound effects.

```
/* Does damage to target entity */
public virtual void DoDamage(Entity target, float amount)
{
	//Only broadcast to everyone if the game is networked
	if(Game.isNetworkedMultiplayer)
	{
		target.networkView.RPC("TakeDamage", RPCMode.All, amount);
	}
	//not networked multiplayer, not broadcasting to anyone
	else
	{
		target.TakeDamage(amount);
	}
}
/* RPC broadcast taken damage, will also instantiate blood effect,
* play sound effect and check if entity is dead or not */
[RPC] public virtual void TakeDamage(float damage)
{
	Health -= damage;
	SpawnBlood();
	PlayTakeDamageSound();
	CheckDead();
}
```

5.3.4 Special cases

Some enemy characters and all player characters have something unique on their design that needs to be synchronized as well. For example, one playable character named Shaper changes the weapon it is wielding. The solution is always the same, make a new RPC function.

5.4 Synchronization of status effects

Status effects can be synchronized in StatusEffectManager that is used to add and remove all all status effects. StatusEffect component is used directly in Entity component, This was done by turning every function that adds status effect into RPC function. If game is networked multiplayer we broadcast adding an effect to everyone. And if the game is single player we do not send anything. All status effects do have some kind of visual effect as well these can be synchronized at the same time to reduce bandwidth requirement.

Example:

```
/*Adds stun status effect and plays particle effect*/
[RPC] public void AddNetworkedStun(float duration)
{
	//Makes new stun object and adds
	//it to the list of all statuseffects
	statusEffectList.Add(new StunStatusEffect(duration));

	//starts stun particle effect
	enemy.effectSpawner.AddStunParticles(duration);
}
```

5.5 Synchronization of visual effects

Most of visual effects can be synchronized with logic as they are started by a sequence of functions. For example blood splatter effect can be played when entity takes damage. Some effects are loaded locally, like fire on torches. The remaining few visual effects need to be broadcasted to everyone using RPCs.

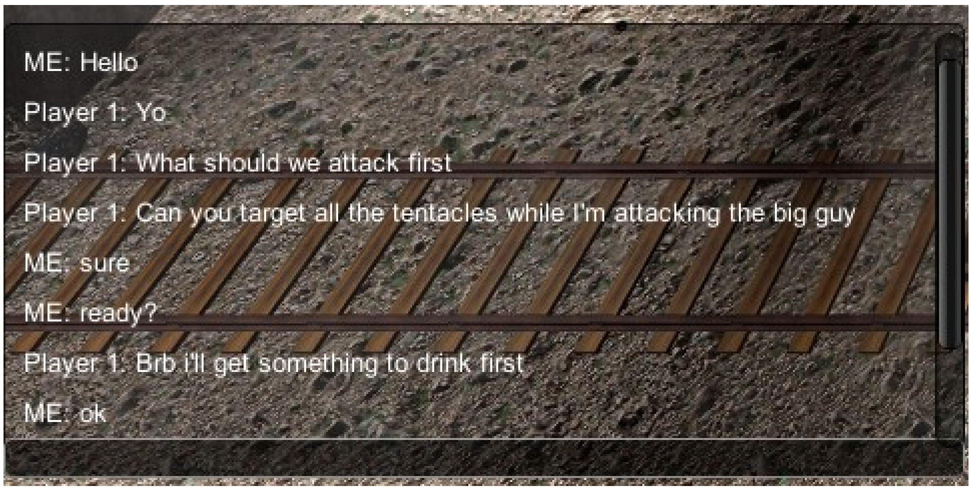
5.6 Synchronization of sounds

All sounds used in this game project so far are activated using other RPCs and other logic. For example taking damage sounds can be played in the TakeDamage function. This will reduce bandwidth requirements a little.

5.7 Chat

Chat was made so player can communicate together when they are not using some third party VoIP service. The game would be too hard, if communication was not possible as harder parts of the game require good teamwork. Chat will be loaded only when game is multiplayer game.

Visual presentation was made with UnityGUI using basic box for background labels for text, scrollview for scrolling and textfield to write text input. All the text is stored in ArrayList. It is located at bottom left side of the game window. The size will be calculated from screen height and width to make it fit on every screen resolutions.


ME: Hello

Player 1: Yo

Player 1: What should we attack first

Player 1: Can you target all the tentacles while I'm attacking the big guy

ME: sure

ME: ready?

Player 1: Brb i'll get something to drink first

ME: ok

Figure 32. Chat window in game

Pressing enter key will send written text to others using RPC. Players will have names when development of this game is further. But for now chat will use player numbers instead of names to show who is talking.

```
/* Broadcasts written message to everyone */
if(enterPressed)
{
	enterPressed = false;
	if(textfieldText != "")
	{
		networkView.RPC("SendText",RPCMode.Others, textfieldText);
		chatLog.Add("ME: "+ textfieldText);
	}
	textfieldText = "";
}

/* used to broadcast written text to everyone */
[RPC] public void SendText(string text, NetworkMessageInfo messageInfo)
{
	chatLog.Add("Player " + messageInfo.sender + ": " + text);
	if(chatLog.Count > maxLogSize)
	{
		chatLog.RemoveAt(0);
	}
}
```

6 CONCLUSION

Requirements of this thesis were completed well. Most work was required to get movement look smooth enough. Interpolation and extrapolation logic required a lot of tweaking to work properly. I did use a lot of time to improve my interpolation logic, which in the end failed and I had to use linear interpolation instead. This probably failed because I tried Bézier curve or because my math wasn't done correctly. I may try to improve interpolation again later using cupid spline instead of Bézier curve.

I did not have prior experience of network programming, meaning that I had to read a lot of tutorials and articles about the topic just to get started with this thesis. It wasn't a big problem because I find this subject really interesting and the thesis is made for my own game project.

The game wasn't originally designed as networked multiplayer game which made added its own difficulty. I've learned from this and I will design my future projects keeping this in mind.

Servants of Aether has definitely improved as a game after the implementation of networked multiplayer. Especially harder parts of the game are more fun now. The game now being developed as multiplayer being the core function of the game.

Since game is now fully networked, Revon Games can do level testing together on the same server, which makes developing and designing the game slightly more fun.

Developing the network aspects of the game does not entirely end here. Game is still in developments so some new characters and features might need to be synchronized.

REFERENCES

Mulholland, A. & Hakala, T. 2004. Programming multiplayer games. Plano. Wordfare

Lake, A. 2011. Game programming gems 8. Boston. Cengage

Unity 3D 2014a. Effortlessly unleash your game on the world's hottest platforms, Accessed 28.5.2014. http://unity3d.com/unity/multiplatform/

Unity 3D 2014b. Get going with Unity for consoles. Accessed 28.5.2014. http://unity3d.com/unity/multiplatform/consoles

Unity 3D 2014c. Get to grips with the panels, tabs and views of Unity then learn how to extend and customise it. Accessed 28.5.2014. http://unity3d.com/learn/tutorials/modules/beginner/editor

Unity 3D 2014d. Getting started with Mono Develop. Accessed 13.5.2014. http://docs.unity3d.com/Documentation/Manual/HOWTO-MonoDevelop.html

Unity 3D 2014e. Network Reference Guide. Accessed 28.5.2014. http://docs.unity3d.com/Manual/NetworkReferenceGuide.html

Unity 3D 2014f. Networking Elements in Unity. Accessed 28.5.2014. http://docs.unity3d.com/Manual/net-UnityNetworkElements.html

Unity 3D 2014g. RPC Details. Accessed 28.5.2014. http://docs.unity3d.com/Manual/net-RPCDetails.html

Unity 3D 2014h. High Level Networking Concepts. Accessed 28.5.2014. http://docs.unity3d.com/Manual/net-HighLevelOverview.html

Valvesoftware 2014. Source Multiplayer Networking. Accessed 28.5.2014. https://developer.valvesoftware.com/wiki/Source_Multiplayer_Networking

Valvesoftware 2012. Interpolation. Accessed 28.5.2014. https://developer.valvesoftware.com/wiki/Interpolation

Tutorialspoint 2014. UML Use Case Diagram Accessed 26.5.2014 http://www.tutorialspoint.com/uml/uml_use_case_diagram.htm

Microsoft 2008. Networking Basics: peer-to-peer vs. server-based networks http://technet.microsoft.com/en-us/library/cc527483%28v=ws.10%29.aspx