Requirements

1. Introduction

This document specifies all functionality that should be supported by our system.

1.1. Purpose of the system

The systems purpose is to simulate the game world and to replicate everything relevant to other players.

1.2. Scope of the system

?

1.3. Objectives and success criteria of the project

The objective of this system is to have a stable base game simulation and network replication to easily add new weapons, enemies etc.

1.4. Definitions, acronyms, and abbreviations

Game Definitions:

- *Mage*: The mage's job is to spawn the enemies into the game world, and their goal is to eliminate the other players (*Shooters*). They see the world from a higher position and have full visibility of everything.
- *Shooters*: The shooter's job is to survive the hordes of enemies that the mages summons. The Shooters have access to a arsenal of weapons to defend themselves.
- *Mobs*: The enemies of the shooters. They get spawned by the mage and get controlled through AI.

1.5. References

?

1.6. Overview

The game is a asymmetrical survival twin-stick shooter, where one player assumes the role of the enemy spawning

The keep the networking side simple, all players are responsible for their own game logic. This means, that a client by themselves decides, whether they got hit by a projectile or dealt damage to an enemy. This keeps the networking simple and allows us to do easy unit testing.

This simple architecture is possible because of the game type. There is no direct combat between players, so they won't feel any lag or unfair hits.

2. Proposed system

2.1. Overview

As stated above, the main goal is to create a system, which acts as a foundation for all the important elements in the game. Most importantly, this means having a viable weapon architecture to easily add new content and a network foundation, which takes care of replication

2.2. Functional requirements

2.2.0 Overall Architecture

2.2.0.1 Entity Base Systems

Shooters and **Mobs** share the same parent class **Entity**. **Entity** needs to handle the following things:

2.2.0.1.1 Health Component

The **Health Component** needs to keep track of the current health of a entity. For that, it has the following properties:

- *Current Health* : stores the current health of the entity
- *Max Health* : store the maximum health of the entity

The **Health Component** needs to correctly apply healing and damage to the current health. *Current Health* can **not** exceed *Max Health*.

The **Health Component** also needs to handle specific events/signals:

- OnDamage(float amount) : Gets called when an entity gets damaged
- OnDie() : Gets called when the *Current Health* is below 0.

2.2.0.1.2 Networked Transform

The **Networked Transform** needs to replicate the entities position to the other peers. **Networked Transform** needs to have a boolean value, indicating whether the entity is owned by the local client.

- If the entity **is** owned by the local client, this component needs to take the current position and rotation of the entity and broadcast it over the network.

- If the entity **is not** owned by the local client, this component needs to receive the values sent by the other peer and apply it to the local object. The values should not be applied directly but rather the component should do some kind of interpolation (linear interpolation should be enough).

2.2.0.2 Entity Base Class

The **Entity** base class needs to contain some methods to simulate the entity over the network.

- *RPCDealDamage (amount : float)* : this method needs to be called by a client when they register a hit on this entity. It's an RPC call, that gets called on the peer owning the entity.

- *RPCPlayAnim(anim : string)* : this method gets called by the owning peer when the entity changes its animation state. This will only be used if we get to the point to add animations ¯_(ツ)_/¯

2.2.0.3 Entity Manager

The **Entity Manager** needs to keep track of all the entities in the simulation.

- *registerEntity(ent : Entity) : int* : Needs to be called when a new entity gets created (e.g. by the mage). Upon being called, it needs to add the entity to a replication group so it gets spawned on all the other clients. Returns the id of the new entity.
- *removeEntity(ent : Entity)* : Needs to be called when a entity gets removed from the simulation. Even though Godot replication can remove the nodes for us, we also need to clean the references to the entities in our dictionary.
- *getEntityByID(id : int) : Entity* : Can be called to retrieve an entity by its id (e.g. when receiving an id through an RPC).
- *getShooters() : List* : Can be called to retrieve a list of all players (helpful for enemy script, that need to check all players).
- *getLocalShooter() : Shooter* : Can be called to retrieve the local Shooter object or null if the peer is playing as mage.

The **Entity Manager** needs to have a internal dictionary for mapping entity ids to entities and a list of player objects to be able to return them with *getShooters()*.

2.2.1 The Mage

The Mages ability allows him to spawn enemies to fight the shooters. For this the mage needs mana.

2.2.1.1 Mana System

The Mana System keeps track of the current Mana the Mage has. It has the following properties:

- Maximum Mana : The maximum amount of mana the mage can accumulate
- Current Mana : The amount of mana the mage has at this moment
- Mana Regen : The amount of mana the Mage regenerates per second

As stated above, the mana systems needs to increment *current mana* by *mana regen* mana per second, until it hits *maximum mana*.

The System also needs the ability to check for the current mana amount and remove a specific amount of mana, when the mage casts an action.

2.2.1.2 Spawn System

The mage must be able to spawn enemies. For this, multiple things need to be done.

2.2.1.2.1 Spawn Logic

When the mage decides to spawn an enemy, the game needs to check the **Mana Manager** whether the mage has enough mana to do the action.

If so, the game needs to calculate the spawn position for the new mob, by raycasting from the current mouse position and converting it to a place in the world. If the spot is not valid (outside of playable area, position is blocked by other object), the spawn action is canceled.

If the last step succeded, the game creates an enemy from the choosen prefab at the calculated position and calls the register method on the **Entity Manager**. This replicates the spawned mob to all other players.

2.2.1.2.2 Spawn UI

To be able to spawn mobs, the mage needs specific functionality in their UI.

We need buttons for the different mob types, which show if the **Mana Manager** currently holds enough mana to spawn this mob type.

The UI should also include a upgrade button for every mob type, so the mage can easily upgrade them.

2.2.1.3 Blood System

To upgrade their mobs, the mage has the currency blood. Blood has the following characteristics:

- Blood is a object in the game world.
- Blood gets dropped by **Shooters**, when they receive damage by mobs. The blood should be spawned at the position of the **Shooter** when it was it.
- Blood is only visible to the mage.
- Blood can be collected by the mage by moving their mouse cursor near to the blood. For this, an invisible object is glued to the mouse cursor of the mage, which collects blood objects in it's perimeter.
- The mage class stores the current amount of collected blood.
- The mages UI has a spot to show the current amount of blood.

2.2.2 The Shooters

The Shooters need multiple requirements fullfilled.

2.2.2.1 Movement

Shooters movement is controlled by either the WASD keys on a keyboard or the left analog stick on a controller.

Shooters have a **Movement Speed** property, which is the amount of distance they can travel per second. When a user presses a button, the **shooter** should be moved for the pressed time in the specified direction (e.g. when user holds w for 0,5 seconds, the player should be moved up by *movementSpeed*0.5* units).

2.2.2.2 Aiming

The **Shooters** can aim their weapon either by moving their mouse in the direction they intend to shoot or by using the right analog stick on a controller:

- When the user is using a mouse, we need to raycast from the camera and find the spot, where our raycast hits the x-z-plane. This is where the player must look.

- When the user is using a controller, we just use his right joystick input as his rotation.

2.2.2.3 Weapon Manager

Each **Shooter** has a **Weapon Manager** node as a child, which keeps track of the weapons the shooter is currently carrying.

When a **Weapon** should be given to a player, the add function gets called on the **Weapon Manager** node. The node needs to create the corresponding weapon prefab as a child of itself.

The **Weapon Manager** needs to do the following things:

- **Handle the Weapon Prefabs**: Each weapon the Shooter picks up is a child node in its node hirarchy. The **Weapon Managers** job is to have only the currently selected weapon active.
- **Handle Weapon Switch**: When the user moves their mouse wheel or presses the upper buttons on the back of the controller, the weapon manager should cycle through the weapons, enabling and disabling the prefabs as it goes.
- **Forward Shoot Action**: When the user decides to shoot, the weapon manager must forward this to the currently active weapon prefab.

2.2.2.4 Shooting

When pressing the left mouse button or the right back button on their controller, the shoot function must be called on the currently selected weapon. (More on that in **Weapon**)

2.2.2.5 Weapon & Bullet System

Every **Weapon** is a prefab, which inherits from the **Abstract Weapon** class.

Every weapon prefab needs to override the *OnDisable* and *OnEnable* functions, which get called by the **WeaponManager** to disable and enable the spawned weapon prefabs.

To simplify the creation of generic, fully automated weapons, the helper class **BasicWeapon** can be used.

To handle networked bullets, the weapons & bullets systen needs to do the following.

2.2.2.5.1 Weapon Component

When shooting with a weapon, depending on whether the current client is the shooting one, we need to do different things.

- if the current client is shooting, we spawn the bullet projectile, turn it's *locally simulated* boolean to true and add it to the simulation. We now need to call the **RPCSpawnBullet** function on all clients, while using the bullets position and velocity as parameter.

- if the current client is not shooting, bullets by other shooters only get spawned through the **RPCSpawnBullet** RPC. This time, the bullet keeps its *locally simulated* boolean at false and uses the parameters it received from the RPC to position the bullet and give it the necessary velocity.

2.2.2.5.1.1 BasicWeapon

BasicWeapon is a helper class, extending **Abstract Weapon**. It is used for basic, machine gun type weapons, so there is not as much code repetition.

It needs to have the following components:

- *Shoot Delay* : The amount of time to wait between each bullet
- *Bullet Amount* : The amount of bullets this weapon still has
- *Ejection Point* : The position from which Projectiles are to be emitted
- *Damage Per Bullet* : The damage each projectile should inflict

BasicWeapon should emit a projectile, if the shoot button is currently being held down, *Bullet Amount* is higher than 0 and the last Shot was atleast *Shoot Delay* seconds ago.

2.2.2.5.2 Bullet Component

Bullets contain a **damage** value, which is the amount of damage it inflicts upon colliding with something.

When being spawned into the world, bullets need to be placed on the right layer, so they dont collide with objects, they are not supposed to collide with.

After being spawned into the simulation, a bullet simply flies in a specific direction, directed by its velocity. Upon colliding with something, we do the following:

- If the bullet collided with something, that is not an Entity (i.e. a wall), it simply deletes itself from the simulation.

- If the bullet hits something, that is an entity (e.g. mob or shooter), we need to check, whether it is locally simulated.

  - if the bullet **is not** *locally simulated*, we simply remove it from the simulation.
  - if the bullet **is** *locally simulated*, we call the RPCDealDamage function on the hit entity with the specified damage value. After that, we delete it from the simulation

2.2.2.6 Money System

To buy new equipment, the Shooters can earn **Money**. Money has the following characteristics:

- **Money** can exist as an object in the game world. This object contains a value *Money Amount*.
- Each **Shooter** has a property *money*, which stores their current money
- Upon getting into contact with a Shooter, the **Money** object gets removed from the game world and the **Shooter** gets *Money Amount* added to their *money* property.
- TODO: ADD MONEY SPENDING METHOD.

2.2.3 The Mobs

Mobs get controlled by the computer and their behavior is dependant on their type:

2.2.3.1 The Zombie

The **Zombie** is a very simple enemy. Upon spawning, it selects the nearest player as its target.

Zombies have different properties:

- *Speed* : Speed describes the speed of the zombie

- *Damage* : Damage is the amount of damage the zombie deals when touching a player

The AI of the Zombie is very simple. It uses the Godot Pathfinding system to find the nearest path to its selected target and deals damage to it upon beeing in a specific distance of it.

2.3 Nonfunctional requirements

2.3.1. User interface and human factors

The biggest part of the users vision should be the game world, so they can have the most vision of the action.

On small UI widgets, the player can see their current currencies, their weapons and other useful information.

2.3.2. Documentation

There should be documentation for commonly used functionality, e.g. the weapons interface. Also docs for the network API.

2.3.3. Hardware considerations

The user should be able to decide between keyboard and mouse and a controller

2.3.4. Performance characteristics

The game should run with at least 60 fps. There should be no stuttering during the action.

2.3.5. Error handling and extreme conditions

There should be Unit tests, which test the most commenly operations. This should be achievable through the simple networking.

2.3.6. Quality issues

?

2.3.7. System modifications

?

2.3.8. Physical environment

?

2.3.9. Security issues

?

2.3.10. Resource issues

?

2.4. Pseudo requirements

The game should be fun.

2.5. System models

The models can be found here

2.5.1. Scenarios

Either all shooters die and the mage wins or the endboss gets beaten and the shooters win

2.5.2. Use case model

2.5.3. Object model

2.5.3.1. Data dictionary

2.5.3.2. Class diagrams

2.5.4. Dynamic models

2.5.5. User-interface -- navigational paths and screen mock-ups

The UI should look similar to the one in League of Legends

3. Glossary