CMP302 Report Alessandro Loddo

1904477

Gravity Ball

Terminology

Cooldown: a period of wait that starts once the spell is cast, the spell can be cast again once the period passed.

Cast: creating, launching a spell.

Spell: a magic event that has an event once cast (moving objects, healing allies, damaging enemies), costs mana and has a cooldown.

Props: objects in the scene.

Mana: Amount of magic energy, needed to cast spells.

Summary

The mechanic chosen is based on a spellcasting system, where the player can cast different spells, with different effects. The mechanic in question takes inspiration from the Diablo III blackhole spell (Blizzard Entertainment, 2012)


Blackhole Spell, Diablo III 2012

The spellcasting system implemented includes different components:

- Multiple spells can be selected with the mouse wheel rotation, these are:
  - ‘Black hole’, a sphere that attracts everything to it for a period of time.
  - ‘Explosion matter’, an exploding sphere that recreates an explosion.
  - ‘Zero-Gravity matter’, a sphere that disables gravity for props and makes them fly.
- Cooldown system
  - Spells can be cast after a cooldown, which starts once a spell is cast.
  - Each spell has a different cooldown.
- Spell toolbar UI that shows the available spells that can be selected and the available mana.

- Mana system
    - Spells can be cast only when the amount of mana is equal to or greater than the cost of mana of the spell.
    - Each spell costs a different amount of mana.

A video that shows the game mechanics is available here:
https://www.youtube.com/watch?v=SYTzGcIqflQ

Requirement Specification

Introduction

Purpose:

The purpose of this project is not only to implement an ability system, which is very common in every game, but also to explore and investigate what the mechanic looks like in code.
The project delivered includes the spell system, cooldown, mana, a UI toolbar and a level designed for spell-testing. It is easily extendible, and more features and mechanics can be easily implemented.

Project Perspective

This project is aimed to create and implement a working magic system, where each spell has a different and specific behaviour, which leads to different results. The level features obstacles and props so that the player can orientate, move and test the different spells in the level and check each scenario where they could be useful. The level and abilities can be easily extended and improved. New features such as more complex levels, more spells and different effects of the abilities can be implemented by designers. However, to change the abilities and effects of the existing skills, the code itself must be changed.

Functions

- User can see what ability is selected through the toolbar.
- User can see how much mana is available to cast spells.
- User can switch between spells.
- User can target different elements of the scene.
- User can see if a spell is or is not in cooldown.
- The application can be extended and improved, with more abilities and spells, not necessarily with programming.

Models and resources

All the models and sounds have been obtained through purchase from the Unreal Marketplace. (Interface & Item Sounds Pack, Daydream Sound).

All the code and blueprints have been programmed by the developer, without any Unreal Engine framework assistance.

Application's features

The first spell is called 'Gravity Hole'. A floating sphere is cast forwards from the player. The sphere will float for some seconds and then, after it has reached a certain distance, it will stop and attract all the available objects within a certain distance.
After the projectile stops, a particle effect will appear.
After some seconds of attraction, it will disappear and can be cast again.
To be cast, the spell must be selected and not in cooldown.
It costs 60 mana.

The second spell is called "Explosion Projectile". A floating sphere is cast forwards from the player. The sphere will float for some seconds and then, after it has reached a certain distance, it will stop, pushing all the actors away, simulating the explosion of a bomb.
After the projectile stops, a particle effect will appear.
After some seconds the sphere will disappear and can be cast again.
To be cast, the spell must be selected and not in cooldown.
It costs 30 mana.

The third spell is called "Zero-Gravity projectile". A floating sphere is cast forwards from the player. The sphere will float for some seconds and then after it has reached a certain distance, it will stop and the gravity for some object within a certain distance is disabled. The actors will float for some seconds (8) and then will fall as the gravity is re-enabled and the sphere disappears.
To be cast, the spell must be selected and not in cooldown.
It costs 70 mana.

Finally, the character can move and jump around the scene. He can cast spells and use the toolbar to choose them.
The movement is handled with the following keyboarding keys (W/A/S/D) and Jump with the spacebar. Spells can be cast with the left mouse key. Spells can be changed with the mouse wheel.

Requirements

The application should run smoothly, without fps drops. If this is not possible and the application lags, undesired events could happen, like input lag, not-smooth animations and movements.

Naming conventions

All variables and references' names created in code have "m_" before the name of the variable. I.e: "m_projectile_component, m_maxDistance". All classes in code have the prefix "U", I.e: "UGravity_Projectile".

Method and Technical Aspect

Spell selection and casting

Changing spells is one of the bases of this application. It is achieved thanks to an integer value in the FirstPersonCharacter class that is changed every time the mouse wheel moves. If the mouse wheel goes up, the "`m_changeBulletVar`" will be increased by one, if the mouse wheel goes down, it will be decreased by one. Where 0 selects the "Gravity-hole", 1 selects the "Explosion projectile" and 2 selects the "Zero-Gravity projectile".
The value of the m_changeBulletVar can not go over 2 and below 0, to avoid problems with projectile selections. Once the player changes the type of projectile through the mouse wheel, this will be made obvious in the toolbar in the game. Before firing a projectile, the code, through an if-statement checks if the projectiles are spawnable (cooldown based), and then, if there is enough mana, the player fires the right projectile based on the "m_changeBulletVar".

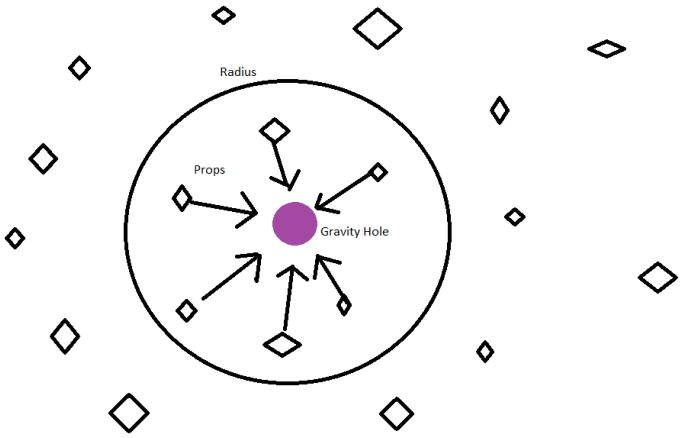
Spell data

To check if the projectile is fireable, there is a Boolean to track if the timer is over. If the timer is over, then the projectiles are fireable. The value is called "`projectileSpawnable`"
If the projectile has just been fired, the projectileSpawnable variable is set to false, once a timer of a certain number of seconds is over, the variable is set to true again and the player can fire another projectile. If the "projectileSpawnable" is true, another check makes sure that the player has enough mana to cast the spell. Finally, if the player has enough mana and the spell is not in cooldown, the player can cast it.
Each projectile has a different cooldown and a different mana cost; they can be changed in the blueprint. All these values can be found in the FirstPersonCharacter class.
This is useful for designers as it can be easily changed without touching the code. It's important to say that the different spells and the firstPersonCharacter are fully implemented in C++. Almost all the variables are blueprintable.

Spell: Gravity Hole

Gravity hole is a spell that attracts all the objects around the spell within a certain distance. Once the spell is cast, it will move in the forward vector, after a few seconds, it

stops and starts the attraction.
This spell's functionality is outlined in the .cpp files, where the spell is initialized and set. In the blueprint, only the function to cast the hole is called.
Apart from the main UE4 functions, like "Tick" and "Begin Play" (which have been modified), there are two more functions. One is to cast the attraction to the objects and one is to stop the bullet.
Once the spell is cast, "Tick" checks if the bullet has reached a certain distance, if it has, then the bool to allow the projectile to stop and activate is set to true. "stopBullet" function is then called, and checks for the bool value, if it is true, then the velocity is set to zero, and the "castAttractionToObjects" function is called.



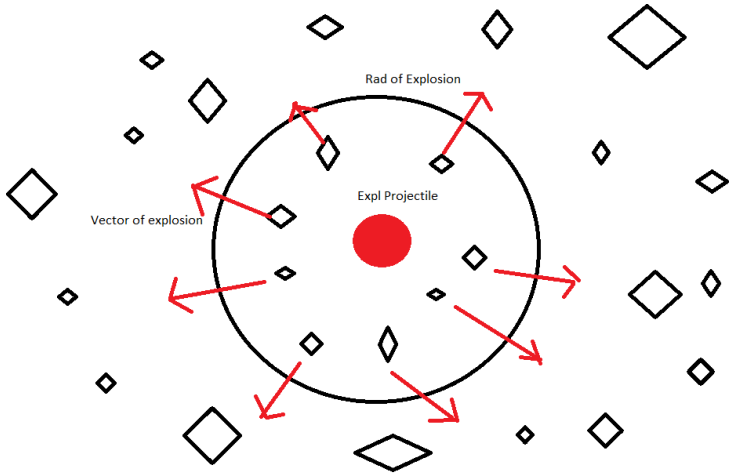
How gravity hole works

More technically, an array that points to the actors is created and used to loop through all the actors in the scene. Each distance from each actor to the player is then calculated and if it is within the distance of attraction, the objects are attracted, and particles are spawned. After the seconds of life span, the projectile is then destructed.

Spell: Explosion Projectile

The Explosion projectile is a spell that simulates an explosion and pushes all the actors away. Once the spell is cast, it will move in the forward vector, after a few seconds, it stops and explodes. This spell's functionality is outlined in the .cpp files, where the spell is initialized and set, in the blueprint, only the function to cast the hole is called.
Apart from the main UE4 functions, like "Tick" and "Begin Play" (which have been modified), there are two more functions. One is to cast the explosion to the actors and one is to stop the projectile, similar to the gravity one.
Once the spell is cast, the Tick function checks whether the projectile has reached a certain distance, if it has, then it stops and explodes.
The bool to allow the projectile to stop and activate is set to true. The "setNewVel" function is then called, and checks for the bool value, if it is true, then the velocity is set to zero, and the "castExplosionToActors" function is called.
In this function, as before, an array that points to the actors is created and used to loop through all the actors in the scene. This time, the vector from the projectile to the actors

is calculated, the direction will be the opposite of the gravity projectile one.



How Explosion projectile works.

Spell: Zero-Gravity Projectile

The zero-gravity projectile is a spell that simulates a zero-gravity environment for some seconds, making all the actors float. Once the spell is cast, it will move in the forward vector, after a few seconds, it stops and creates a zero-gravity field. This spell's functionality is outlined in the .cpp files, where the spell is initialized and set, in the blueprint, only the function to cast the hole is called.

Apart from the main UE4 functions, like "Tick" and "Begin Play" (which have been modified), there are two more functions. One is to cast the zero-gravity field to the actors, and one is to stop the projectile, similar to the previous ones.

Once the spell is cast, the Tick function checks whether the projectile has reached a certain distance, if it has, then it stops and casts the gravitational environment.

The bool to allow the projectile to stop and activate is set to true. The "setNewVel" function is then called, and checks for the bool value, if it is true, then the velocity is set to zero, and the "zeroGravity" function is called.

In this function, as before, an array that points to the actors is created and used to loop through all the actors in the scene.

The positions of the actors and the projectile are calculated and if within that range, the gravity for the actors is disabled and a small impulse is added to them to make them float. This happens only if the life span bullet is between 1 and the life span, when the bullet is going to be destructed, the gravity is set again, and the actors will fall on the scene.

FirstPersonCharacter class

The First Person character class is the one where all the projectiles are spawned from. Where the amount of mana available is declared and the cost of each spell is declared. It is almost entirely implemented in C++. After being initialized and the different inputs are set, there are 2 functions to update the variable to change the spell. One is called when the mouse wheel goes up, one is called when it goes down.

```
void AUFirstPersonCharacter::changeBullCountDown()      //If mouse wheel goes down, decrease bullet counter
{
    m_changeBulletVar--;
    if (m_changeBulletVar >= 2)     //If variable goes beyond the last projectile, the variable is stuck at the last projectile
        m_changeBulletVar = 2;

    if (m_changeBulletVar <= 0)         //If variable goes beyond the first projectile, the variable is stuck at the first projectile
        m_changeBulletVar = 0;
}
void AUFirstPersonCharacter::changeBullCountUp()        //If mouse wheel goes down, increase bullet counter
{
    m_changeBulletVar++;
    if (m_changeBulletVar > 2)          //If variable goes beyond the last projectile, the variable is stuck at the last projectile
        m_changeBulletVar = 2;
    if (m_changeBulletVar < 0)
        m_changeBulletVar = 0;              //If variable goes beyond the first projectile, the variable is stuck at the first projectile
}
```

When any projectile is fired (so it is not in cooldown), the fire animation starts and a sound is played.
The rotation and direction of the bullet spawning are based on the character rotation and the direction that he is facing.
The main function in this class is the “spawnProj()” function.

```
void AUFirstPersonCharacter::spawnProj()
{
    if (m_projectileSpawnable == true)
    {
        const FRotator SpawnRotation = GetControlRotation();
        // MuzzleOffset is in camera space, so transform it to world space before offsetting from the character location to find the final muzzle position
        const FVector SpawnLocation = (GetWorld()->GetFirstPlayerController()->GetPawn()->GetActorLocation() + SpawnRotation.RotateVector(GunOffset));
        FActorSpawnParameters SpawnInfo;
        // try and play a firing animation if specified
        if (FireAnimation != nullptr)
        {
            // Get the animation object for the arms mesh
            UAnimInstance* AnimInstance = Mesh1P->GetAnimInstance();
            if (AnimInstance != nullptr)
            {
                AnimInstance->Montage_Play(FireAnimation, 1.f);
            }
        }
        // try and play the sound if specified
        if (FireSound != nullptr)
        {
            UGameplayStatics::PlaySoundAtLocation(this, FireSound, GetActorLocation());
        }
        if (m_changeBulletVar == 0&&m_mana>= m_GravHoleCost)        //If the gravity bullet is selected and the player has enough mana
        {
            GetWorldTimerManager().SetTimer(Cooldown, this, &AUFirstPersonCharacter::TimerFunc, m_gravCooldown, false);         //Timer for cooldown
            GetWorld()->SpawnActor<AUGravity_Projectile>(SpawnLocation, SpawnRotation, SpawnInfo);              //spawn gravity projectile
            m_projectileSpawnable = false;   //Projectile is not spawnable until cooldown is over
            m_mana -= m_GravHoleCost;       //mana minus cost for casting
        }
        if (m_changeBulletVar == 1&&m_mana >= m_ExplCost)          //If the explosion bullet is selected and the player has enough mana
        {
            GetWorldTimerManager().SetTimer(Cooldown, this, &AUFirstPersonCharacter::TimerFunc, m_explCooldown, false);         //Timer for cooldown
            GetWorld()->SpawnActor<AUExplosion_Projectile>(SpawnLocation, SpawnRotation, SpawnInfo);             //spawn explosion projectile
            m_projectileSpawnable = false;          //Projectile is not spawnable until cooldown is over
            m_mana -= m_ExplCost;       //mana minus cost for casting
        }
        if (m_changeBulletVar == 2&& m_mana >= m_zeroGravCost)          //If the zero-gravity bullet is selected and the player has enough mana
        {
            GetWorldTimerManager().SetTimer(Cooldown, this, &AUFirstPersonCharacter::TimerFunc, m_zeroGravCooldown, false);         //Timer for cooldown
            GetWorld()->SpawnActor<AUFloatingBullet>(SpawnLocation, SpawnRotation, SpawnInfo);                  //spawn zero-grav projectile
            m_projectileSpawnable = false;                                                                      //Projectile is not spawnable until cooldown is over
            m_mana -= m_zeroGravCost;       //mana minus cost for casting
        }
    }
```

When the projectile should be spawned, a check makes sure that the projectiles are not in cooldown, if they are not, the fire animation is played along with the sound. A second check makes sure that the player has enough mana to cast the spell, if it does, the spell is cast.

Toolbar Widget

The Toolbar widget is used by the toolbar UI and it contains these elements:

- Canvas
    - Size box
        - Horizontal box
            - Three Size boxes, each containing
                - An Overlay, each containing
                    - An image, material and the cost of the spell.

All the widget does is it holds the information to understand what spell is chosen, it receives the data it requires, textures and colours to differentiate the spell. If the spells are in cooldown, they have red borders, if they are not in cooldown, they are fireable, but the player can fire them only if he has enough mana

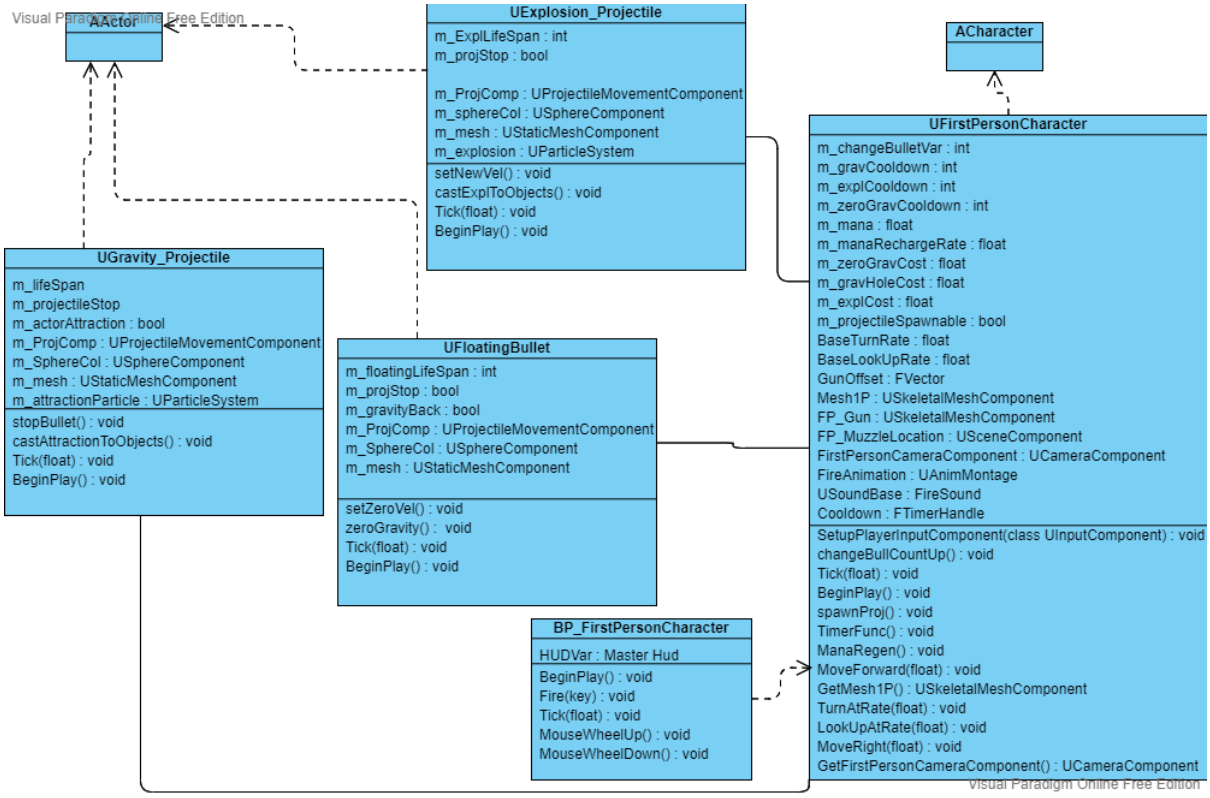
Master HUD

The Master HUD is the user interface created once the BeginPlay function is called (the FirstPersonCharacter one). Once created, it is added to the viewport. The player has then a visual reference to understand what spell is chosen.
It contains the canvas panel, toolbar Widget and mana bar.

Development

The application was developed starting in blueprint until the end, I could have chosen to start it and convert it step by step, but I found it easier to just finish it and convert it all.
I expected the code to look pretty much like the blueprint version, yet they are totally different. The code version is way more straightforward and small, more than 30 nodes are less than a hundred lines of code (in the case of the zero-gravity projectile).

UML Diagram



Conclusions

Altogether, the application works as intended and all the spells behave in the initially planned way.
I would have liked to implement more features and more projectiles but due to shortage of time, 3 were implemented. This is not a problem, as previously said, more can be easily implemented. Most of the time was spent researching a functional way to implement timers and convert all the firstPersonCharacter class in code.
Another feature I would have liked to implement is enemies and damaging spells.
This could be something I implement in the future.

References

4, U. E., n.d. *Programming with C++ Unreal Engine 4.* [Online]
Available at: https://docs.unrealengine.com/4.27/en-US/ProgrammingAndScripting/ProgrammingWithCPP/
[Accessed 2022].

Productions, D., 2020. *DPTV UE4 Minecraft Style Tutorial 8 (Toolbar Selection).* [Online]
Available at: https://www.youtube.com/watch?v=hPLHJ9wAciY
[Accessed 2022].