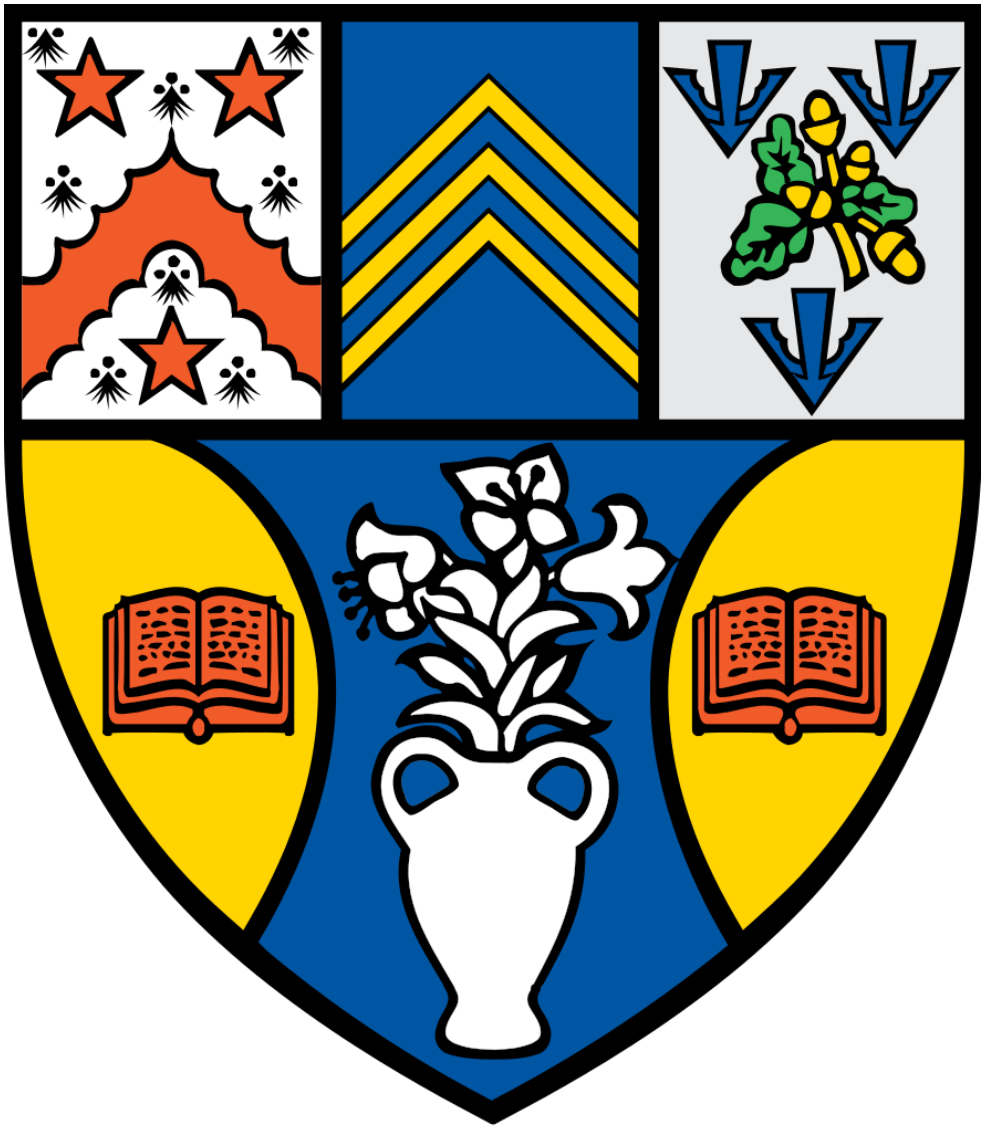


# GAMEPLAY MECHANICS DEVELOPMENT

CMP302: PROJECT REPORT



Kevin Conaghan
1500613

# Contents

# Summary

The aim of the project was to develop a dynamic spell system that allowed a designer to edit the attributes of and the look of the spells. Inspiration was taken from multiple games, such as: World of Warcraft (Blizzard, 2004); Guild Wars 2(ArenaNet, 2012) and Magicka (Paradox Interactive, 2011). The spells identified for the project were:

- Spell 1: (Fires a beam of the specified element outwards in front of the player)
    - Flamethrower spell
    - Blizzard spell
    - Lightning spell
- Spell 2: (fires a sphere of the specified element toward the enemy)
    - Fireball spell
    - Frostball spell
    - Lightning ball spell
- Spell 3: (applies a burn effect on the enemy)
    - Flame Burn spell
    - Frost Burn spell
    - Lightning Burn spell

Other Mechanics:

- Element system (allows the player to pick which element they would like to use)
- Enemy: (Basic AI with chase functionality)
- Spell 4: (applies a radial impulse on the player) *
- Spell 5: (Buffs the player with the specified element) *
- Tab Targeting: (dynamic targeting between enemies) *

The final mechanics listed and marked with an (*) were not implemented and are discussed in the methodology section of this report.

The spells are fully modular and can be edited in any way in the Unreal Editor. Each spell component has attributes such as: damage, damage rate and duration which can also be edited in the Unreal Editor.

A video demonstrating the system is available here: https://youtu.be/mM_NNHnupjM

# Requirement Specification

## Introduction:

### Purpose

In every MMORPG there are spell or special move set lists. For example, World of Warcraft uses a spell system with cast times and cooldowns. This project was designed to re-create a basic version of a spell system a MMORPG would use.

### Product Scope

The general purpose and objectives of this project are described in the CMP302 Coursework Brief. The project will have functionality to support the editing of spells and the in-game use of the spells available in the editor. There is also supporting functionality for the player character, enemy and User Interface elements to show the core functionality of the project. The intended purpose is not to provide a standalone game but a foundation for a spell system.

### Requirements and Specification References

CMP302 Coursework Specification by Matthew Bett, 2018. Available at: https://blackboard.abertay.ac.uk/webapps/blackboard/content/listContent.jsp?course_id=_7369_1&content_id=_450926_1&mode=reset

Unreal Engine Documentation by Epic Games Inc, 2017. Available from *https://docs.unrealengine.com/latest/INT/*

## Overall Description:

### Product Perspective

The core features of the finished product will be the spell system and a basic enemy AI. The supporting features such as: The User Interface and the player character which are sufficient for the demonstration but would need a significant re-work to make the spell system more modular to allow for the addition of spells.

## Product Functions

The core functions allow the user to:

- Create spells using the derive Blueprint function.
- Edit spells attributes and visuals using the Blueprint Editor.
- Add more visuals to the UI (User Interface) using the Blueprint Editor.
- Edit the enemy using the Blueprint Editor.
- Test functionality using the play button in the Unreal Editor.
    - Cast Times.
    - Spell Damage, damage rate and cooldown time.
    - The enemy's reaction to collisions with spells.
    - The visuals on the UI.

## User Classes & Characteristics

The main target user for this product is a game designer who has a basic knowledge of how the Unreal Editor and the Blueprint editor works. Programmers using this product would only be required to extend, maintain or add functionality to the project and a working knowledge of the Unreal Engine blueprints and C++ development is necessary. Any artists using the product would require a basic knowledge of the Unreal Engine blueprints and material editors, as well as the Cascade particle system editor.

## Operating Environment

The project is intended for use exclusively with Unreal Engine v4.17 and editing C++ code using Microsoft Visual Studio 2015 (Community).

Hardware Requirements:

- Desktop PC or Mac.
- Windows 7 64-bit or Mac OS X 10.9.2 or later.
- Quad-core Intel or AMD processor, 2.5 GHz or faster.
- NVIDIA GeForce 470 GTX or AMD Radeon 6870 HD series card or higher.
- 8 GB RAM.

## Design & Implementation Constraints

The coursework specification states that the product must have unique functionality: it cannot be a small modification to an existing object or system from Unreal. The core functionality must be based in C++. Can have blueprints or a blueprint featured interface intended for use by a designer. You cannot create your C++ by simply creating a complex blueprint and using Unreal's nativiser. It is expected that all blueprints and C++ follow good coding practices for design, naming conventions and encapsulation.

Since there is no artist assigned to the project, all visual representation, except from the UI, will be limited to the assets the Unreal Starter Content pack provides. The UI icons are all placeholders that will be acquired from the internet. Some particle effects will be created as placeholders for visual representation.

## User Documentation

See Appendix A for System User Guide.

# External Interface Requirements

## User Interfaces

The main user interface for this product will be the Unreal Engine blueprint editor. The user will have access to edit any of the objects' visuals, as well as edit attributes of spells which affect the balance of the game such as: damage, damage rate, enemy health and cast times. Components such as spells can be added and removed using the standard method in the Unreal Editor.

## Hardware Interfaces

The product will be developed on Windows PC and will only be available to use on this platform. During game playback the primary control support is for keyboard.

## Software Interfaces

- Unreal Engine v4.17
- Microsoft Visual Studio 2015 (Community)

# System Features

## Element System

- *Description*: Upon button press, the element currently active is set to the desired element which changes the spell set to the respecting element.
- *Stimulus/Response Sequences*: For development, the element system is a part of the character blueprint and C++ class which can be opened via Unreal blueprint editor or Microsoft Visual Studio respectively. For gameplay, the element system selection is accessed through keyboard input.
- *Functional Requirements*:
    - REQ – 1: Change Spell Sets
        - It must be able to change the spells according to the specified element, for example lightning must produce lightning related spells.

## Spells:

### *Spell One: -*

- Description: Upon button press, a timer is initiated to create "casting" effect. Once the timer has finished it spawns a particle effect representing the element currently active and damages the enemy over time if they are within the collision box. After a specified duration it is then destroyed.
- Stimulus/Response Sequences: For development, the spell one blueprint or C++ code will be opened via Unreal blueprint. For gameplay, spell one selection and triggering is accessed through keyboard input.

- Functional Requirements: -
    - REQ – 1: Modular Object:
        - Spell one must be represented as a module that can be added or removed from the player.
    - REQ – 2: Cast:
        - Upon selecting spell one, a timer must appear displaying the current progress of the cast time which is defined via the blueprint editor.
    - REQ – 3: Fire:
        - After the timer has finished it must spawn the actual spell which will damage the enemy if within range.
    - REQ – 4: Damage over time:
        - Once the enemy is within range it must apply damage over time.
    - REQ – 5: Cooldown:
        - Once destroyed it is will apply a global cooldown period so other spells cannot be instantly cast.
    - REQ – 6: User editing, balancing attributes:
        - It should be possible for the user to adjust the attributes via the blueprint editor for balancing purposes.
    - REQ – 7: Visual Representation:
        - The spell must have a visual representation associated with the corresponding element.

### *Spell Two: -*

- Description: Upon button press, a timer is initiated to create “casting” effect. Once the timer had finished it spawns a static mesh and particle effect representing the element currently active and acts as a projectile, moving towards the enemy. Once it collides with the enemy it will damage the enemy and use an “exploding” particle effect and destroy itself.
- Stimulus/Response Sequences: For development, the spell two blueprint or C++ code will be opened via Unreal blueprint. For gameplay, spell two selection and triggering is accessed through keyboard input.
- Functional Requirements: -
    - REQ – 1: Modular Object:
        - Spell two must be represented as a module that can be added or removed from the player.
    - REQ – 2: Cast:
        - Upon selecting spell two, a timer must appear displaying the current progress of the cast time which is defined via the blueprint editor.
    - REQ – 3: Projectile:
        - Spell two will require a projectile object with appropriate speed and functionality to be able to damage enemy characters upon colliding with them.
    - REQ – 4: User editing, balancing attributes:
        - It should be possible for the user to adjust the attributes via the blueprint editor for balancing purposes.
    - REQ – 5: Cooldown:
        - Once destroyed it is will apply a global cooldown period so other spells cannot be instantly cast.
    - REQ- 6: Visual Representation:

        - The spell must have a visual representation associated with the corresponding element.
    - REQ -7: Destruction:
        - If the projectile cannot find the target or is taking too long to reach the target it must be destroyed after a length of time.

## *Spell Three: -*

- Description: Upon button press, a particle effect will spawn on the enemy and apply damage over time. After a specified duration it is then destroyed.
- Stimulus/Response Sequences: For development, the spell three blueprint or C++ code will be opened via Unreal blueprint. For gameplay, spell three selection and triggering is accessed through keyboard input.
- Functional Requirements: -
    - REQ – 1: Modular Object:
        - Spell three must be represented as a module that can be added or removed from the player
    - REQ – 2: Cast:
        - Upon selecting spell three it must be an instant cast time.
    - REQ – 3: Fire:
        - Once activated it must spawn the actual spell on which will inflict damage once applied to the enemy.
    - REQ – 4: Damage over time:
        - Once applied to the enemy it must apply damage over time.
    - REQ – 5: Cooldown:
        - Once destroyed it is will apply a global cooldown period so other spells cannot be instantly cast.
    - REQ – 6: User editing, balancing attributes:
        - It should be possible for the user to adjust the attributes via the blueprint editor for balancing purposes.
    - REQ – 7: Visual Representation:
        - The spell must have a visual representation associated with the corresponding element.

## *Spell Four: -*

- Description: Upon button press, a timer is initiated to create “casting” effect. Once the timer had finished it spawns a particle effect representing the element currently active then pushes all objects away from the player.
- Stimulus/Response Sequences: For development, the spell four blueprint or C++ code will be opened via Unreal blueprint. For gameplay, spell four selection and triggering is accessed through keyboard input.
- Functional Requirements: -
    - REQ – 1: Modular Object:
        - Spell four must be represented as a module that can be added or removed from the player.
    - REQ – 2: Cast:
        - Upon selecting spell one, a timer must appear displaying the current progress of the cast time which is defined via the blueprint editor.

- REQ – 3: Fire:
  - Once activated, it must apply a radial impulse at the player's location.
- REQ – 4: User editing, balancing attributes:
  - It should be possible for the user to adjust the attributes via blueprint editor for balancing purposes.
- REQ – 5: Cooldown:
  - Once destroyed it is will apply a global cooldown period so other spells cannot be instantly cast.
- REQ – 6: Visual Representation:
  - The spell must have a visual representation associated with the corresponding element.

### *Spell Five: -*

- Description: Upon button press, a particle effect will spawn on the player and apply a double damage attribute to other spells. After a specified duration it is then destroyed.
- Stimulus/Response Sequences: For development, the spell four blueprint or C++ code will be opened via the Unreal blueprint. For gameplay, spell four selection and triggering is accessed through keyboard input.
- Functional Requirements:
  - REQ – 1: Modular Object:
    - Spell four must be represented as a module that can be added or removed from the player.
  - REQ – 2: Cast:
    - Upon selecting spell five it must be an instant cast time.
  - REQ – 3: Fire:
    - Once it is activated, it must spawn the actual spell on the object which will apply double damage to all other spells.
  - REQ – 4: Cooldown:
    - Once destroyed it is will apply a global cooldown period so other spells cannot be instantly cast.
  - REQ – 5: User editing, balancing attributes:
    - It should be possible for the user to adjust the attributes via the blueprint editor for balancing purposes.
  - REQ – 6: Visual Representation:
    - The spell must have a visual representation associated with the corresponding element.

### *Enemy: -*

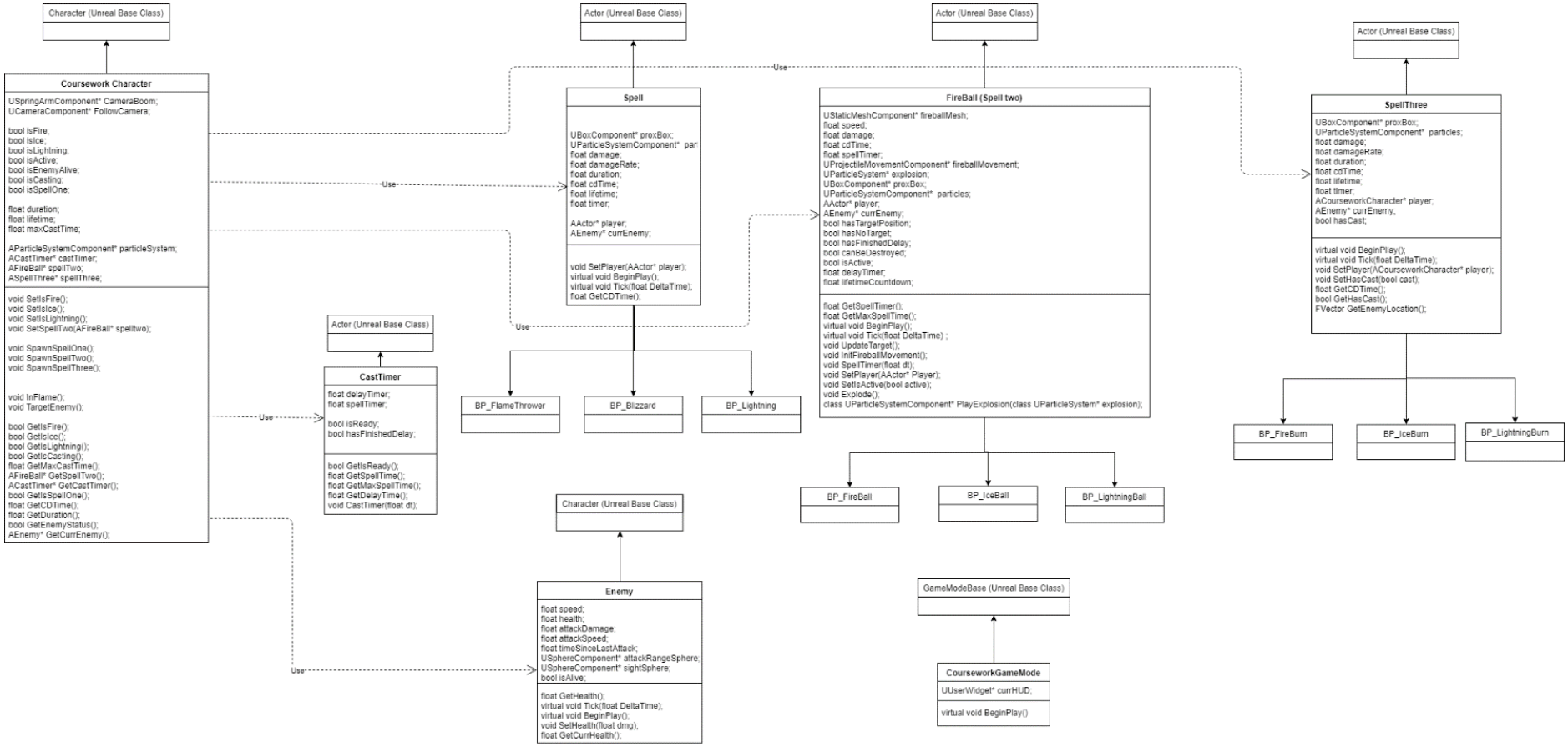
- Description: A standard basic enemy object that can take damage and chase the player when they are within range.
- Stimulus/Response Sequences: For development, the enemy blueprint or C++ code will be opened via Unreal blueprint.
- Functional Requirements:
  - REQ – 1: Modular Object:
    - Enemy must be represented as a module that can be added or removed in the scene.
  - REQ – 2: User editing, balancing attributes:

        - It should be possible for the user to adjust the attributes via blueprint editor for balancing purposes.
    - REQ -3 Chase:
        - Once the player is within a user-specified range the enemy must be able to chase the player until they are out of range.
    - REQ – 4: Visual Representation:
        - The spell must have a visual representation associated with the corresponding element.

### *Tab Targeting: -*

- Description: a dynamic targeting system that allows the player to target enemies in the scene.
- Stimulus/Response Sequences: For development, the enemy blueprint or C++ code will be opened via the Unreal Blueprint. For gameplay, tab targeting triggering is accessed through keyboard input.
- Functional Requirements: -
    - REQ – 1: Target:
        - It must allow the player to target an enemy and switch between enemies when pressed.

# UML DIAGRAM


Character (Unreal Base Class)
Coursework Character
USpringArmComponent* CameraBoom;
UCameraComponent* FollowCamera;
bool isFire;
bool isIce;
bool isLightning;
bool isActive;
bool isEnemyAlive;
bool isCasting;
bool isSpellOne;
float duration;
float lifetime;
float maxCastTime;
AParticleSystemComponent* particleSystem;
ACastTimer* castTimer;
AFireBall* spellTwo;
ASpellThree* spellThree;
void SetIsFire();
void SetIsIce();
void SetIsLightning();
void SetSpellTwo(AFireBall* spelltwo);
void SpawnSpellOne();
void SpawnSpellTwo();
void SpawnSpellThree();
void InFlame();
void TargetEnemy();
bool GetIsFire();
bool GetIsIce();
bool GetIsLightning();
bool GetIsCasting();
float GetMaxCastTime();
AFireBall* GetSpellTwo();
ACastTimer* GetCastTimer();
bool GetIsSpellOne();
float GetCDTime();
float GetDuration();
bool GetEnemyStatus();
AEnemy* GetCurrEnemy();
Actor (Unreal Base Class)
Spell
UBoxComponent* proxBox;
UParticleSystemComponent* par
float damage;
float damageRate;
float duration;
float cdTime;
float lifetime;
float timer;
AActor* player;
AEnemy* currEnemy;
void SetPlayer(AActor* player);
virtual void BeginPlay();
virtual void Tick(float DeltaTime);
float GetCDTime();
Actor (Unreal Base Class)
FireBall (Spell two)
UStaticMeshComponent* fireballMesh;
float speed;
float damage;
float cdTime;
float spellTimer;
UProjectileMovementComponent* fireballMovement;
UParticleSystem* explosion;
UBoxComponent* proxBox;
UParticleSystemComponent* particles;
AActor* player;
AEnemy* currEnemy;
bool hasTargetPosition;
bool hasNoTarget;
bool hasFinishedDelay;
bool canBeDestroyed;
bool isActive;
float delayTimer;
float lifetimeCountdown;
float GetSpellTimer();
float GetMaxSpellTime();
virtual void BeginPlay();
virtual void Tick(float DeltaTime) ;
void UpdateTarget();
void InitFireballMovement();
void SpellTimer(float dt);
void SetPlayer(AActor* Player);
void SetIsActive(bool active);
void Explode();
class UParticleSystemComponent* PlayExplosion(class UParticleSystem* explosion);
Actor (Unreal Base Class)
SpellThree
UBoxComponent* proxBox;
UParticleSystemComponent* particles;
float damage;
float damageRate;
float duration;
float cdTime;
float lifetime;
float timer;
ACourseworkCharacter* player;
AEnemy* currEnemy;
bool hasCast;
virtual void BeginPlay();
virtual void Tick(float DeltaTime);
void SetPlayer(ACourseworkCharacter* player);
void SetHasCast(bool cast);
float GetCDTime();
bool GetHasCast();
FVector GetEnemyLocation();
Actor (Unreal Base Class)
CastTimer
float delayTimer;
float spellTimer;
bool isReady;
bool hasFinishedDelay;
bool GetIsReady();
float GetSpellTime();
float GetMaxSpellTime();
float GetDelayTime();
void CastTimer(float dt);
BP_FlameThrower
BP_Blizzard
BP_Lightning
BP_FireBall
BP_IceBall
BP_LightningBall
BP_FireBurn
BP_IceBurn
BP_LightningBurn
Character (Unreal Base Class)
Enemy
float speed;
float health;
float attackDamage;
float attackSpeed;
float timeSinceLastAttack;
USphereComponent* attackRangeSphere;
USphereComponent* sightSphere;
bool isAlive;
float GetHealth();
virtual void Tick(float DeltaTime);
virtual void BeginPlay();
void SetHealth(float dmg);
float GetCurrHealth();
GameModeBase (Unreal Base Class)
CourseworkGameMode
UUserWidget* currHUD;
virtual void BeginPlay()
Use
Use
Use
Use
Use

# Methodology

## Element System:

One of the initial requirements is to allow the player to change spell sets with the relevant element. Since this system relies on player input it would be best to implement this within the player class. The most effective way to create this system is to create Boolean values for each element and a setter function, which would set the element Boolean to true when the player inputs that element. For example: Player presses Z which represents fire; use the fire setter function to set fire Boolean to true and all other element Booleans to false. The Boolean values would then be used to compare which type of element spell the player will use.

## Spells: Implementing Time systems

Two spells out of the five inflict damage over time. Damage over time manages the amount of damage a spell can do over a certain period. Both spells have a damage rate float variable which is editable in the blueprint editor. This is done using a timer system and is reset after inflicting damage to the enemy. A similar timer system was used to determine how long the spell is active for and is reset once the timer reaches the duration which can be set in the blueprint editor. The cast timer works in exactly the same way, only it is its own object that can be edited in the blueprint editor and spawned into the scene. Without this object spell one will not work.

## Spells: Inflicting Damage

All spells inflict damage to the enemy (except spell five which was not implemented), this was achieved using a for loop that iterates over all the actors in the scene and finds an actor of type AEnemy. After it has found the enemy, it subtracts the user defined damage variable using a setter function named SetHealth. It will only apply the damage if the for loop returns a valid pointer.

## Spell Two: Movement & Destruction

Spell two (AFireball) uses the UProjectileMovementComponent class to move. It is moved by calculating the distance between the enemy and the player character and adds that multiplied by the user specified speed variable. If the enemy is moving then the distance between the enemy and the player character is multiplied by the enemy's current velocity. Unlike spell one, spell two spawns into the scene while casting, that means even if there is no target it will still spawn. Spell two has a timer system on it that if it reaches the designated time will destroy the object. This means if the enemy is far away or if there is no enemy in the scene and the spell spawns, it will still explode.

## User Interface: Display

The User Interface is important for this genre as it displays information the player would need to know such as: cooldown times, cast time and spells available. The player's HUD is accessible via the blueprint editor and is dynamic as the user can add in new images for the spells relatively easily. All the user must do is add in a texture variable with the desired image and edit Get_Brush_0 to return that image instead of the placeholders provided. The user interface relies heavily on getter functions (GetIsIce, GetIsFire and GetIsLightning) for display as it returns the current element which will determine the type of spell that will be used.

## Enemy: Movement

The enemy has basic AI functionality, when the player gets within a user specified range then it will chase the player. This is implemented using the sphere component class. Once a sphere component is attached to the enemy in the Tick() function checks every frame to see if the distance from the player to the enemy is greater than the sphere component's radius, if it is then do nothing. However, if the player is within range then the enemy will move towards the player using the pre-defined function AddMovementInput().

## Spell Four: Radial Impulse (NOT IMPLEMENTED)

Spell four was not implemented due to time constraints. The plan for spell four was that it would apply a radial impulse at the player's location, pushing all other objects back. The spell would get all actors that collide with the collision sphere and apply an impulse to the objects in the opposite direction relative to the player. It would make use of Unreal's pre-defined function AddRadialImpulse() and the spell would also have a particle effect visually representing the sphere.

**Pseudocode for ApplyImpulse()**

***1.1 On Input***
***1.2 Spawn collision sphere at player's location***
***1.3 For all objects overlapping the collision sphere***
    ***1.3.1 Object-> apply radial impulse()***
***1.4 End loop***

## Spell Five: Buff (NOT IMPLEMENTED)

Spell five was partly implemented but due to time constraints was not finished. However, the planned approach to implementing the buff spell is described here. The spell would visually be represented by a particle effect on the player which could be set via the blueprint editor. Once activated the player would pass a double multiplier variable to the other spell classes and when they apply damage it would be multiplied by the multiplier variable.

**Pseudocode for Buff()**

***1.1 Check to see If the spell is active***
***1.2 If the spell is active:***
    ***1.2.1 Spell one -> SetIsBuffed(true)***
***1.3 In Spell One class***
***1.4 If buff is not active:***
    ***1.4.1 Enemy-> SetHealth(damage)***
***1.5 else:***
    ***1.5.1 Enemy ->SetHealth( damage * multiplier)***
*1.6* ***End***

## Tab Targeting: Target (NOT IMPLEMENTED)

The targeting system was not implemented due to time constraints. However, the planned approach can be found here. The targeting system was intended to be a significant part of the project however due to difficulties trying to apply it, it was not implemented. The target is chosen by finding the enemy closest to the player and if pressed again it would find the next nearest enemy and target that. Once the target has been decided it would feed that to all the other spells so that the spells did not have to find the enemy themselves.

# Development

The Waterfall Development Process was used as it was the most flexible and once the foundations of the project had been developed it would just be the case of adding more spells.

| Week No | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |
|---|---|---|---|---|---|---|---|---|---|
| | Project Planning | | | | | | | | |
| | | Foundations | | | | | | | |
| | | | Targeting | | | | | | |
| | | | | Spell One | | | | | |
| | | | | | Spell Two | | | | |
| | | | | | | Spell Three | | | |
| | | | | | | Spell Five | | | |
| | | | | | | | Spell Four | | |
| | | | | | | | | Documentation | |
| | | | | | | | | | |

During the project planning section, a prototype was developed for spell three to see how manageable the project would be and the schedule was adapted accordingly. Having become more familiar with Unreal by researching different ideas for spells and completing tutorials relating to the research, the spells were all chosen. Once the C++ classes were ready, blueprints were derived from them to set the attributes and create the visuals for them. The schedule compensated for setbacks that would come with the project, for example Unreal crashing after compiling.

## Source Control

Github was used to manage the project and make sure everything was backed up. It also meant that with a remote repository work could be done from anywhere. This helped keep to schedule and created a fall back for if something went majorly wrong with the local repository. A link to the repository can be found in the References section.

# Conclusion

The project failed to contain all the features set out in the requirement specification. Spells four and five were not implemented in time: this was because the project fell behind on trying to implement the targeting system (which was another component the project failed to deliver on). Looking back on the project, not enough time was given to fully develop the key features of the project which, if completed, would have made the rest of the project easier to develop and make more modular. A constraint that was overlooked was Unreal Engine's C++ projects compilation times. An initial compile after pulling from the Github repository could take up to 20 minutes. Another oversight was when Unreal crashed at runtime. The problem with this was that it did not provide a description of why it crashed. Very late on in the project schedule it was found that the description of why the application crashes could be found in the logs folder. Had this been found sooner it would have reduced debugging time significantly.

There was an issue during the development of the targeting class which was left unresolved and work arounds had to be implemented to compensate for this.

The enemy was going to have basic attack functionality, but not enough planning went into implementing the player's attributes. Had there been more time this would have been a simple task.

While debugging, it was found that Unreal crashes upon trying to access a null pointer. These were resolved quickly when referencing the crash log. These issues wasted much time before the crash log was found.

If this project was to be taken on further, there is plenty of work that could be added on such as properly implementing spells four and five. A significant amount of time would have to be put into re-working some of the code to allow for the implementation of targeting. Due to the targeting being a key feature, this would be the first thing to try and add into the project.

The features of the requirements and specifications that were delivered were finished completely and the spells that were developed can be edited completely in the blueprint editor. The basic functionality of a spell system was fulfilled and would be useful in creating the foundations of a MMORPG.

Prior to undertaking this project there was little knowledge on Unreal Editor and its blueprint system but having completed the project, the understanding of how they work have been significantly improved.

# References

Blizzard(2004), *World Of Warcraft – PC [Video Game].* Blizzard.

ArenaNet(2012), *Guild Wars 2 – PC [Video Game].* ArenaNet.

Paradox Interactive(2011), *Magicka – PC [Viedo Game].* Paradox Interactive.

Sherif, W (2015) *Learning C++ by Creating Games with UE4*. Packt Publishing Ltd.

Youtube (2017) *Unreal Engine C++ Tutorial: Inputs and Collision. Available at: https://www.youtube.com/watch?v=MFZ51eHFB_A&t=1318s*

*(Accessed April 18th 2018)*

*Youtube(2017) C++ Missiles in Unreal Engine – Tutorial [Intermediate/Advanced]. Avialable at:*

https://www.youtube.com/watch?v=w1nCGV5zcPI

*(Accessed April 18th 2018)*

*Youtube(2011) Harrison McGuire. Available at:*

https://www.youtube.com/channel/UC3QmKYux59jdGJWgMopzWTw/videos

*(Accessed April 18th 2018)*

## Image References:

Wikimedia Commons (2013) *Fire close up texture[image]. Available at:*

https://commons.wikimedia.org/wiki/File:Fire_close_up_texture.jpg *(Accessed April 22nd 2018)*

Deviant Art (2013) *Frostbite [Illus]. Available at:*

https://cloaked-nightingale.deviantart.com/journal/magic-419086325 *(Accessed April 22nd 2018)*

*Wikimedia Commons (2009) Lightning in Zdolbuniv [image]. Available at:*

https://commons.wikimedia.org/wiki/File:Lightning_in_Zdolbuniv.jpg *(Accessed April 22nd 2018)*

The Elder Scrolls Wiki (no date) *fireball [Illus]. Available at:*

http://elderscrolls.wikia.com/wiki/Fireball_(Skyrim) *(Accessed April 22nd 2018)*

Magic the Gathering (2015) *Abbot of Keral Keep [Illus]. Available at:*

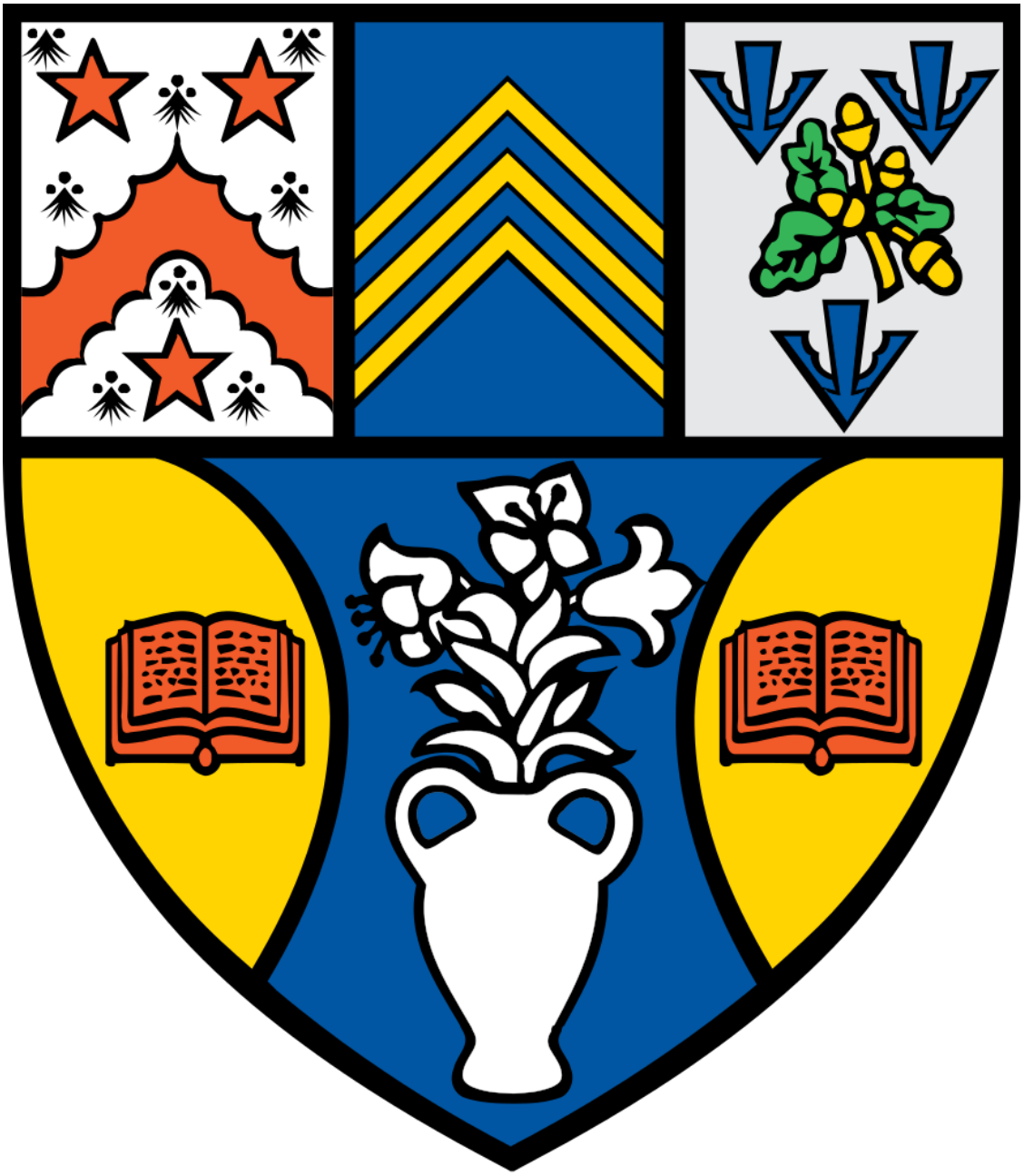
https://magic.wizards.com/en/articles/archive/magic-story/offers-fire-2015-08-05 *(Accessed April 22nd 2018)*

# System User Guide

## CMP302: Gameplay Mechanics Development

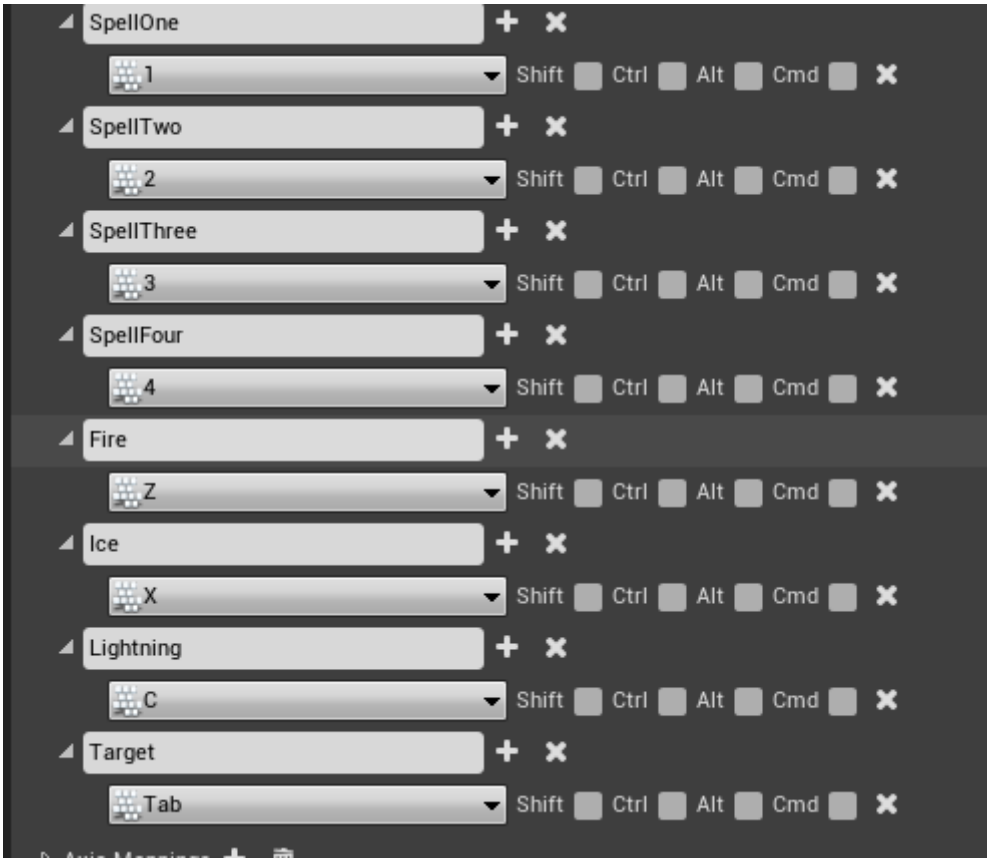
Kevin Conaghan

1500613



# Project Settings: Input

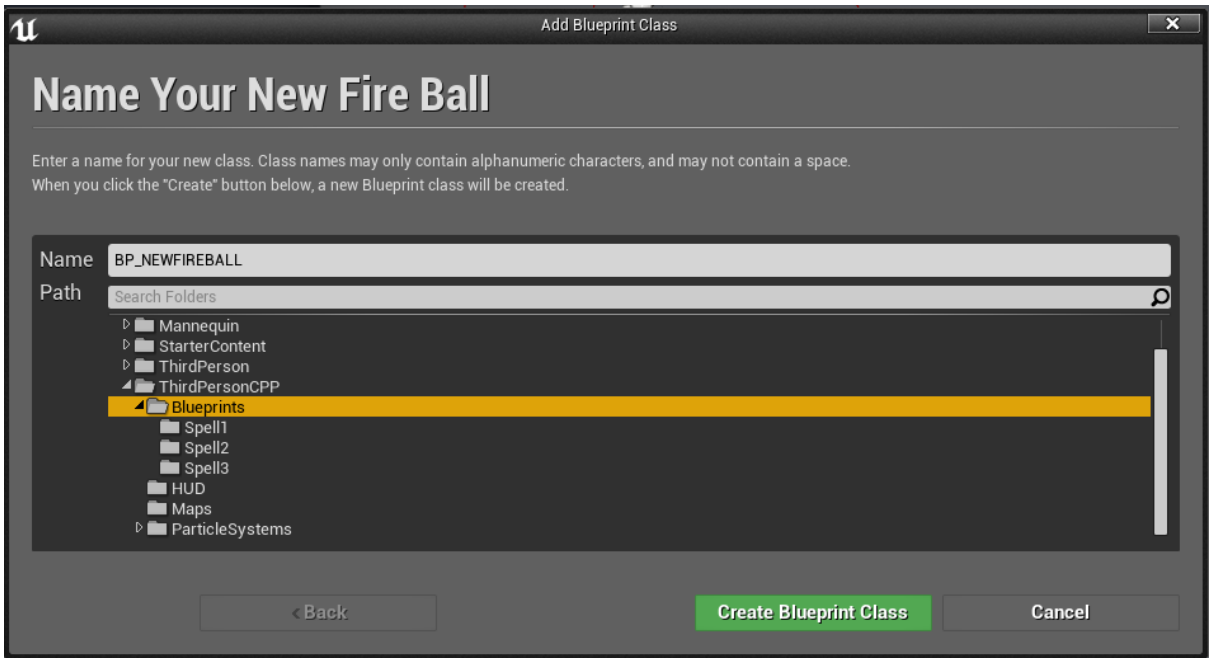
1. Control of the spells and the player has been set up for input action and axis mappings. To access these settings in your own project, go to go to Edit >> Project Settings >> Input.
2. Set up these action mappings to control which spell to use. Here is a screenshot of the development settings:



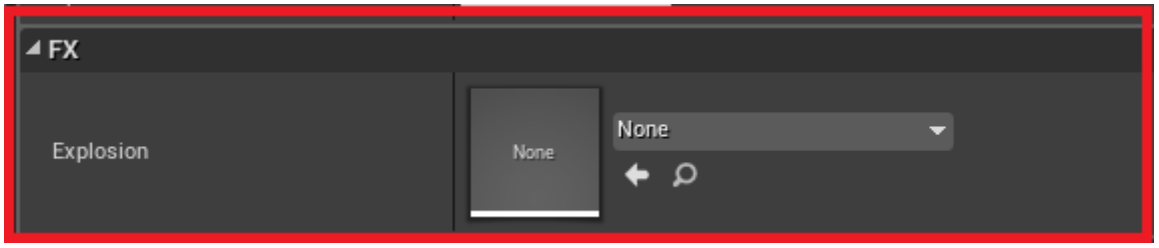

# Creating a new spell

1. Go to C++ classes >> Coursework and right click any of the spells labelled: SpellOne, FireBall and spellThree.
2. In the C++ Actions Menu that opens, select “Create blueprint class based on [Insert spell here].”
3. Name your new class and select ThirdPersonCPP >> Blueprints >> [Inset spell here] the example shown is for creating a new spell of type two (Fireball spell).

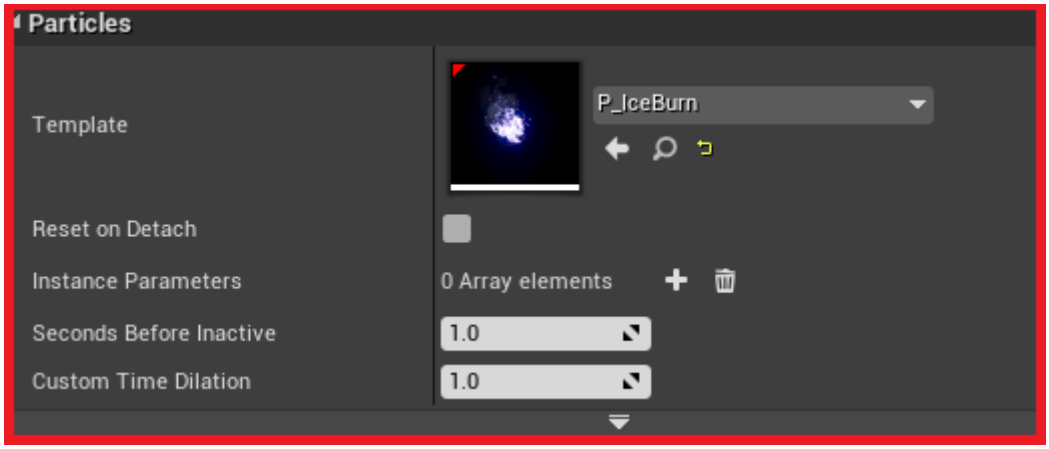
4. Click on Create Blueprint Class.



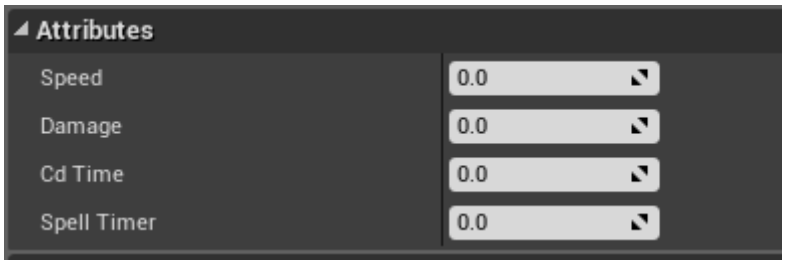

5. Open your new Blueprint Class.
6. Select an Explosion particle effect.




7. In the components tab select “particles”.
8. Select a particle effect to display on the spell.



9. Select the root component.
10. Modify attributes of spell.

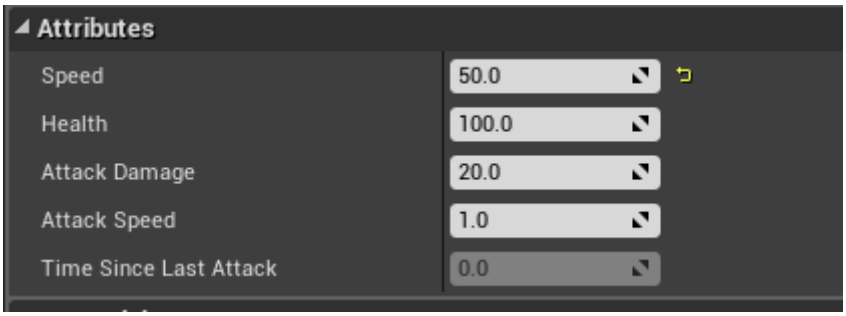



11. To test your new spell you will have to attach it to the player character. Go to your player character blueprint. Located Content >> Blueprints int eh content browser and open “ThirdPersonCharacter”
12. Once in the player character blueprint select the root component and attach to the relevant spell. For example: if you created a spell two blueprint with ice particles attach it to the ice spell in the category spell two.

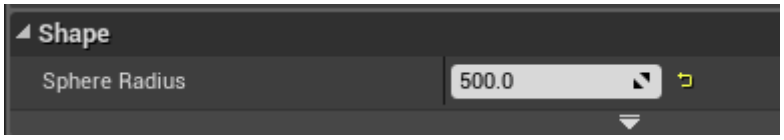

# Creating an Enemy

1. Go to C++ Classes >> Coursework and right click “Enemy”
2. In the C++ Actions Menu that opens, select “Create blueprint class based on ”Enemy.”
3. Name your new enemy and select ThirdPersonCPP >> Blueprints
4. Click on Create New Blueprint Class
5. Open your new blueprint class

6. Set up the enemy's attributes.




7. In the components tab select "sightSphere"
8. Edit the spheres radius, this acts as the enemies sight and will chase the player if they are within the sphere.




9. To test drag your new enemy into the scene and press play.