Game mechanics development project report (CMP302)

# Summary

This mechanic is an alternate ammo type mechanic that allows the player to shoot different types of bullets that has different effects on the enemies depending on the ammo type. It is similar to the mechanic seen in Call of Duty: Black Ops 3’s (Activision 2015) zombie game mode with the Double Pack a Punch feature. My feature includes multiple different bullets that can be fired Including:

* Fire Ammo: Once it hits the enemy it will spawn a lava pool below the enemies feet that damages any enemy to step in it
* Thunder Ammo: Sends the enemy flying into the sky before killing the enemy
* Turned Ammo: Sets the Enemy to target the other enemies instead of the player

A video demonstrating these systems is available here: ​

LINK VIDEO HERE

# Requirement Specification

## Introduction

### Purpose

I created this mechanic to be a very base level implementation of the alternate ammo type mechanic. The code is easy to build upon with new ammo types and different effects for the ammo so can be easily progressed and implemented into future games and projects.

## Overall Description

### Product Perspective

The Primary aim of this project was to create a working alternate ammo type system with each bullet type having different effects on hitting enemies. Other parts of the product such as the moving mannequins are there to provide testing for the player and developer.

### 

### Product Functions

1. Allow the user to pick up weapon and shoot
2. Allow the user to see what bullet was shot through the UI
3. Allow the user to target the training dummies to test the ability effects
4. Allow a designer or programmer to derive from the already created bullet types to create new effects
5. Allow a designer or programmer to create new bullet types and implement them into the system.

### User Classes & Characteristics

This project is aimed at a software engineer, or games designer who is familiar with the Unreal Engine 4 product pipeline.

A software engineer should not be required to extend on spells that have already been created, although one might be required to engineer new spells depending on their gameplay complexity.

### Design & Implementation Constraints

Some basic constraints are specified in the coursework implementation document.

The developer has decided to self-impose the constraint of not using the Unreal Gameplay Framework for the spell systems. UE4 provides a gameplay framework for damaging Actors - and actors receiving “hits”. However, the developer wanted to do this outside of the gameplay framework for the sake of extensibility and versioning safety.

Since there is no artist assigned to the project, the developer purchased a marketplace pack for the game's environment and characters. (Knight Polygon Pack, Synty Studios)

## System Features

### Spell: Fire Zone

**Description:**

A spell that can be cast by the player. Once the spell is cast, a sphere will appear in the target position, that will grow in size and damage everything it touches. The zone will do more damage to the target, the closer the target is to the spheres origin. After a set period of time, the zone will shrink back out and eventually disappear.

**Priority:** 8​

**Stimulus / response sequences**​:

For gameplay, the spell functionality is accessed via combination of keyboard and mouse.

For development, the spell blueprint will open the UE4 blueprint editor - and the C++ code will open Visual Studio.

**Functional requirements**:​

*REQ-1: Valid target*

The ‘Fire Zone’ requires a valid target to be cast. In the instance of the FireZone, this is any world actor - it does not have to be a target dummy.

*REQ-2: Spell selected*

The spell must be selected by the appropriate keybind before it can be cast.

*REQ-3: Fire input*

There must be “fire” or “trigger” input by the player for the spell to cast.

*REQ-4: Enough mana*

The player must have an appropriate amount of mana to cast the spell.

*REQ-5: Not on cooldown*

The spell must not be on cooldown to be cast.

### Spell: Magic Bullet

**Description:**

A spell that can be cast by the player. Once the magic bullet is created, a sphere will appear in the direction that the player is facing. This is a bullet that immediately homes in towards a viable spell target according to its distance from the bullet.

**Priority:**​ 8

**Stimulus / response sequences**​:

For gameplay, the spell functionality is accessed via combination of keyboard and mouse.

For development, the spell blueprint will open the UE4 blueprint editor - and the C++ code will open Visual Studio.

**Functional requirements**​:

*REQ-1: Valid target*

The magic bullet requires a valid target to be cast. In the instance of the Magic Bullet, this is any world actor - it does not have to be a target dummy.

*REQ-2: Spell selected*

The spell must be selected by the appropriate keybind before it can be cast.

*REQ-3: Fire input*

There must be “fire” or “trigger” input by the player for the spell to cast.

*REQ-4: Enough mana*

The player must have an appropriate amount of mana to cast the spell.

*REQ-5: Not on cooldown*

The spell must not be on cooldown to be cast.

*REQ-6: Valid target*

If the bullet does not have a valid entity that implements the ISpellTarget interface, it will instantly be destroyed after being cast.

### Spell: Thunderball

**Description:**

A spell that can be cast by the player. Once cast, the thunderball will travel in the direction that the player is facing. If a target is found within a certain radius, the ball will jump to that target and damage it. From there, it will continue to bounce to another target if it can find one in range. The ball will never jump to the same target twice.

**Priority:**​ 8

**Stimulus / response sequences**​:

For gameplay, the spell functionality is accessed via combination of keyboard and mouse.

For development, the spell blueprint will open the UE4 blueprint editor - and the C++ code will open Visual Studio.

**Functional requirements**​:

*REQ-1: Valid target*

The magic bullet requires a valid target to be cast. In the instance of the thunder ball, this is any world actor - it does not have to be a target dummy.

*REQ-2: Spell selected*

The spell must be selected by the appropriate keybind before it can be cast.

*REQ-3: Fire input*

There must be “fire” or “trigger” input by the player for the spell to cast.

*REQ-4: Enough mana*

The player must have an appropriate amount of mana to cast the spell.

*REQ-5: Not on cooldown*

The spell must not be on cooldown to be cast.

### Target Dummy

**Description:**

A target dummy is a practice target for the test environment that responds to the players spells.

**Priority:**​ 6

**Stimulus / response sequences**​:

For gameplay, the player has no direct control over the dummy. Dummies will respond to spells that the player casts.

For development, the dummy blueprint will open the UE4 blueprint editor - and the C++ code will open Visual Studio.

**Functional requirements**​:

*REQ-1: Health*

Target dummies will store a health value that can be independent from other targets.

REQ-2: Damage

Target dummies can receive damage from spells.

*REQ-3: Display health*

*Target dummies will display their current health in a health bar above their heads.*

*REQ-4: Regenerate health over time*

Target dummies health will regenerate over time.

*REQ-5*​: ​*ISpellTarget interface*

Target dummies will implement the ‘ISpellTarget’ interface, indicating that the target dummy will be affected by spells.

*REQ-6: Target applicable*

Target dummies will not be applicable for damage if they are at zero HP.

### Test Map

**Description:**

A test area for the player to explore and test their spells/abilities in.

**Priority:**​ 6

**Stimulus / response sequences**​:

For gameplay, the player will explore this area, but will not be able to leave the castle walls.

For development, the map file will open in the unreal engine editor.

**Functional requirements**​:

*REQ-1: Target Dummies*

The test area must have a set of test dummies for the player to use their spells on.

*REQ-2: Enclosed area*

The area must be closed so the player cannot leave this area, since this is a test range.

### Toolbar UI

**Description:**

The toolbar UI must display the available spells, their keybinds, active cooldowns and the players mana.

**Priority: 9**

**Stimulus / response sequences:**

For gameplay, the player will use the keybinds on the toolbar to select different spells.

Depending on what spell is currently selected, the spell background will change colour.

For development, the unreal widget editor will open.

**Functional Requirements:**

*REQ-1: Display available spells*

Displays all available spells to the player.

*REQ-2: Display current cooldowns*

Displays any active cooldowns as a “progress bar” to show the player what cooldowns are active, and how long there is until the cooldown finishes.

*REQ-3: Display appropriate keybinds*

Display the appropriate keybinds on each spell, to show the player how to select them.

*REQ-4: Change spell selection*

The toolbar must highlight the correct spell according to the players input/keybinds commands.

*REQ-5: Show the correct amount of player mana*

The toolbar UI must display the correct amount of player mana, changing the bar when the player casts and depletes their mana.

### Player character

**Description:**

The player character must be able to move around the map according to the player input. The player must also be able to cast spells.

**Priority:** ​9

**Stimulus / response sequences:**

In game, the player character will respond to the correct movement keys (W/A/S/D) to move around the scene.

In editor, the player blueprint will open in the blueprint editor.

**Functional requirements:**

*REQ-1: Respond to input*

The player character must respond to input to move and cast spells.

## Other non-functional requirements

### Performance requirements

While the game is in play, there must be a stable frame rate for the spells to function correctly. A low frame rate could cause effects like the fire zone to jump in size, when it should be a smooth animation. Low frame rates could also delay response to input actions.

### Software quality

The software implementation must be kept at a high quality and concise standard. The developer strongly believes in the KISS principle (​*KISS*​) for development. All C++ code will be written with proper object-orientation in mind, but also towards the Unreal Engine 4 online coding standard. (​*Coding Standard*​)

While the development of this project was in a waterfall manner, testing and iteration was performed regularly to assure the validity and safety of the code base.

# Method

## Spell selection

A core requirement of the project is to allow the player to able to switch between the different available spells. Each spell is linked to a keybind, in this case, keys 1 to 4 allow selection of spells 1-3.

The input actions are set up in the Unreal Editor as ‘SelectOne’, ‘SelectTwo’ and ‘SelectThree’.

In turn, the player character blueprint listens for these input action events, and responds by changing the current selected spell id, an enum that represents each spell in the game. Finally, the player toolbar widget is informed that the current spell selection has changed - which updates the visual so the player can see the selection change.

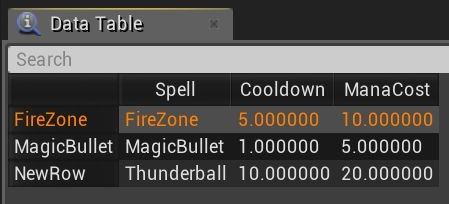
## Spell data

When a spell wants to be cast, the spell casting system must first check if there is an appropriate amount of mana - and that the spell is not on cooldown. These are variables that are specific to the spell, but they are required before the spell is cast.

There could be multiple ways to get this data, such as using a switch statement inside of two functions. For example: ‘GetSpellManaCost(Spell)’ and ‘GetSpellCooldown(Spell)’. However, this implementation would come with obvious issues. Namely, it would require recompiling a blueprint or C++ file whenever those variables were changed, which is neither programmer or designer friendly - and would become less and less maintainable as the number of spells would increase.

Instead, the developer decided to use a data table, which is a Unreal Engine feature similar to that of a simplistic database, where the elements of each table row are that of a C++ structs member variables.

In this case, the struct is ‘FSpellData’ and is formed of ‘Spell ID’, ‘Cooldown’ and ‘Mana Cost’. The spell ID is the enum that represents each spell in the game, and acts as the data tables primary key.



This is much more designer and programmer friendly, and as the data is outlined in the editor, it does not require recompilation and can be edited at runtime.

## Spell casting

The handling of the spell casting has been encapsulated and abstracted to a spell casting component (SpellCasterComponent). This is a C++ component that is used by the player character blueprint.

The spell caster component tracks the current player mana, restores that mana over time, handles the spell cooldowns, and has control over what spells can be cast. It reads the spell data table variables at runtime to get the cooldown and mana cost of each spell, and then inserts that data into a TMap, where the key is the spell id and the values are the cooldown and mana cost. This removes any linear lookup time that might have been necessary.

When the player requests that a spell is cast, the player blueprint first checks with the spell casting component if it is valid to cast that spell. The conditions for a spell being valid to cast are that the player has enough mana, and that the spell is not on cooldown.

After a spell is cast, the spell caster is informed. This depletes the players mana by the appropriate amount, and starts the cooldown for that spell.

The spell caster component reduces the active cooldowns of each spell by delta time, each frame.

## Spell: Fire Zone

The Fire Zone is a area of effect spell that is cast by the player. It is created in the position that the players camera is looking at, and increases in size over time - damaging everything in it’s zone. The closer an object is to the center of the zone, the higher the damage applied will be.

The fire zones core functionality is outlined in a CPP actor, and is structured to be overridden by a blueprint to change its visual effect, properties and functionality.

The zone has three states, ‘Grow’, ‘Still’ and ‘Shrink’. It contains protected properties for the growth time, still time and shrink time, which are set in the deriving blueprint.

The initial state of the zone is to grow, using the growth time and delta time to move a ‘size’ scalar from 0 to 1. Once the zone finishes growing, it enters the ‘still’ state.

In the still state, the zone does not change size. It will simply sit, and count down the ‘still’ time until it reaches zero. When zero is reached, it enters the ‘shrink’ state.

The shrink state is functionally the opposite of the grow state, shrinking the zone ‘size’ back from 1 to 0.

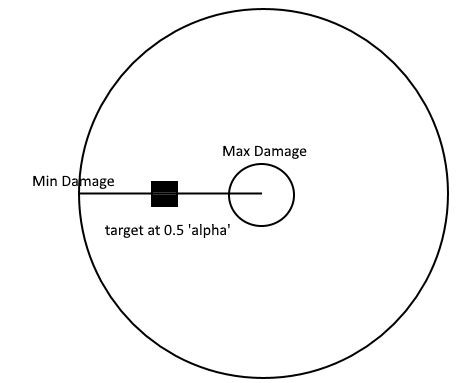
The zone uses blueprint implementable events, so that the blueprint that implements the FireZone can implement ‘ChangeSize’ and ‘GetSphereRadius’ as if they were abstract functions. This allows the blueprint to control the effect, but encapsulates the core functionality away from the blueprint.

The ‘ChangeSize’ function passes in a float value indicating the size of the sphere (0:1), allowing the zone to linearly interpolate between two custom sizes. In the instance of the fire zone blueprint, it scales from (0,0,0) to (6,6,6). Secondly, a boolean value is passed into the function, indicating if the sphere is growing in or out. This is so that the blueprint can use different easing functions to animate the spheres growth depending if it is growing or shrinking.

The ‘GetSphereRadius’ function that is overridden by the blueprint, simply returns the current radius of the sphere for the damage calculation.

As the firezone blueprint increases and decreases in size, it will overlap different actors. Whenever an actor is overlapped, or stops overlapping, the blueprint passes that actor to the base class. If that actor implements ISpellTarget, it is added or removed to a list of actors that should be affected by the zone.

Every frame, the zone will damage each target. The damage applied will be between the values ‘MinDamage’ and ‘MaxDamage’. A linear interpolation between Min:MaxDamage is applied, where the ​*alpha*​ of the linear curve is the distance of the object in relation to the spheres center and radius. All damage is applied by delta time, so it is damaged at the same rate, regardless of frame rate.



This simplistic drawing depicts the idea. If max damage was 20, and min damage was 10, the target would have 15 damage applied to them per second.

## Spell: Magic Bullet

The magic bullet is a spell that can be cast by the player. It is a sphere that homes in towards potential targets, in this case actors that implement ISpellTarget.

In the bullets begin play function, it finds every actor that implements the ISpellTarget interface through the “GetAllActorsWithInterface” function. Each actor is then iterated through, and evaluated as a potential target, usually the chosen target is the closest target - but this will not be the case if that target is at 0% health.

The magic bullet will then move towards the selected target. However, if the selected target is no longer applicable, it will search for a new target and change direction towards that instead.

The magic bullet also accelerates its movement by time. The base class contains two variables for minimum and maximum speed, and these two values are interpolated between by the actors active lifetime.

## Spell: Thunder Ball

The thunder ball is a spell that can be cast by the player. It is a sphere that moves in the direction that the players camera is facing, and jumps between targets to damage them. When no target is found, or there are no new targets to jump towards - the ball destroys itself.

The thunderball has two states, represented by an enum. ‘Travel’ and ‘Jumping’. The ball begins in the travel state, which means that it simply travels in the direction that the player was facing.

Every frame of the travel state, the thunder ball performs a check to find all actors within a spherical radius of the ball. This is done through the “SphereOverlapActors” function in the Kismet library. Originally, this system was going to perform a series of line casts around the sphere, but this was a cleaner and more performant solution. If an applicable target is found, the ball will enter the “jump” state, in order to jump to the actor.

Jumping to the actor is done through linear interpolation, however, a sin wave is applied to the z axis component of the interpolation to appear as if the ball is jumping in an arc.

After the interpolation is complete, the ball will search for a new jump target, restarting the jump state to the new target if one is found, or destroying itself if one is not.

For a target to be “applicable” for the thunder ball, it must derive from ISpellTarget, it must have health greater than zero - and must not have been visited by the ball once before. For checking that the ball has not visited a previous target, each visited target is stored in a TArray of ISpellTarget. However, searching a TArray incurs linear lookup time - so this could have been improved by using a map or hashset instead.

## Spell Widget

The spell widget is used by the toolbar ui, and is structured as so:

● Canvas

○ Border

○ Spell Icon

○ Progress Bar

The spell widgets functionality is encapsulated. The widget does not directly access any actors or components, and is just given what data it requires: the spell icon texture, percentage of the remaining cooldown, the numerical keybind and selection state.

The remaining cooldown percentage is used to change the amount of progress in the progress bar, appearing as if the ability is on “cooldown”. The selection state simply changes the border colour from the selected to not selected colour, appropriately - showing the player what spell is selected.

## Toolbar UI

The toolbar UI, or “player widget” is a user interface that is dynamically created by the player blueprint at the start of the games runtime, and added to the camera viewport. It is responsible for displaying the mana of the player - as well as the available spells, the selected spell, their keybinds and their active cooldowns.

The player widget is structured as so:

* Canvas
* Mana (Progress Bar) ● Canvas

○ Image (Background)

○ Spell Widget

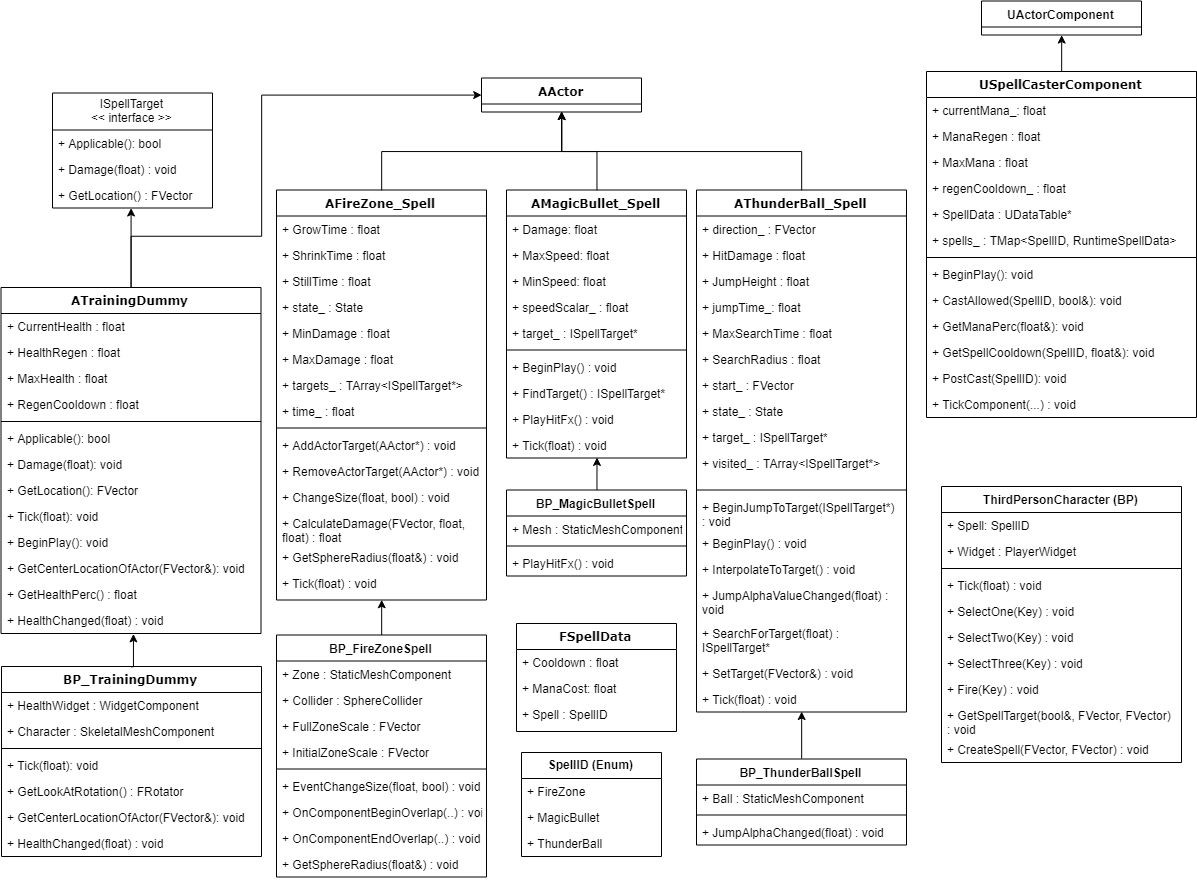
○ Spell Widget

○ Spell Widget

The toolbar UI is handed a reference to the ‘SpellCasterComponent’ during the games initialization. It uses this to get the correct spell state data to manipulate its spell widgets appropriately, as well as update the current amount of player mana. This happens every frame.

# Development

# UML Diagram



*Any functions that use the parameters ‘(...)’, are standard Unreal Engine functions that have three or more parameters, and as such were cut from the UML diagram for the sake of readability.*

# Technical Discussion:

# Development :

# Conclusion:

# References:

All Web pages, Tutorials, videos etc you used. List them here.

# 