|  |
| --- |
| CMP302: Gameplay Mechanics Development |
| R-nader |
| Project Report |

|  |
| --- |
| LEWIS THOMSON  1601828 |

# Project Summary

The aim for the project was to create a gameplay mechanic which enabled the player(s) to interact with the world using their arsenal of specialised grenade types (R-nades) and their radar scan ability. The core of this design is based off of the systems used in 2009’s Manga Series ‘Btooom!’ (Junya Inoue, 1971). The core mechanics identified for this project were:

* Different R-nade types:
  + Impact Detonation.
  + Timed Detonation.
  + Pressure Detonation.
  + Tripwire Detonation. <Not Implemented>
  + Remote Detonation. <Not Implemented>
* Radar / Sonar Scan <Partial Implementation>
* R-nade swap system <Partial Implementation>

The main mechanics to be developed lies within the R-nades themselves. With each one having different, unique detonation conditions and different explosion sizes / effects.

# Requirements Specification

## Purpose

The gameplay I am aiming to produce is a first-person shooter style system where all firearms are replace with physics enabled grenade-type explosives (named, R-nades). The main objective for this project is to implement a variety of interesting weapon and tracking mechanics.

## Audience

In a real-world scenario, the target audience would be teens/adults for casual or competitive experiences. However for this being used as only a coursework submission, the audience I am targeting will be assessors and examiners who are familiar with the use of Unreal Engine and C-based coding.

## Scope

The project’s scope is already outlined in the brief provided as part of the CMP02 Coursework, thus I will not re-explain what is already written there.

However, my individual submission will focus primarily on combat mechanics with the use of explosive projectiles, as mentioned above in the summary. Aside from the mechanics and ‘weapons’ being implemented, there will also be a ‘testing range’ where it will be possible to demonstrate all of their functionality.

It is worth noting, that it is not a game that I am developing, but a foundation with functional mechanics that can be added to and built upon.

## Overview

The main submission will consist of the ~~5~~ 3 different types of functioning ‘R-nades’ (including basic models and visualisations), Radar / Scan system for gameplay mechanics regarding locating other players (though there will be no multiplayer functionality). There will also be NPCs with extremely basic pathfinding AI in the ‘test range’ to demonstrate weapon functionality. As previously stated, this is aiming to be a foundation for demonstrating the mechanics, rather than a full feature application.

All artistic-based objects in the application can be changed within engine without any prior knowledge of code. Each type of R-nade has been created in C++ through Unreal Engine, with a UE Generated Blueprint for designer use to change and tweak balancing values. All types of R-nades currently effect dynamic physics objects as well as effect enemy characters in terms of removing health.

## Development Environment

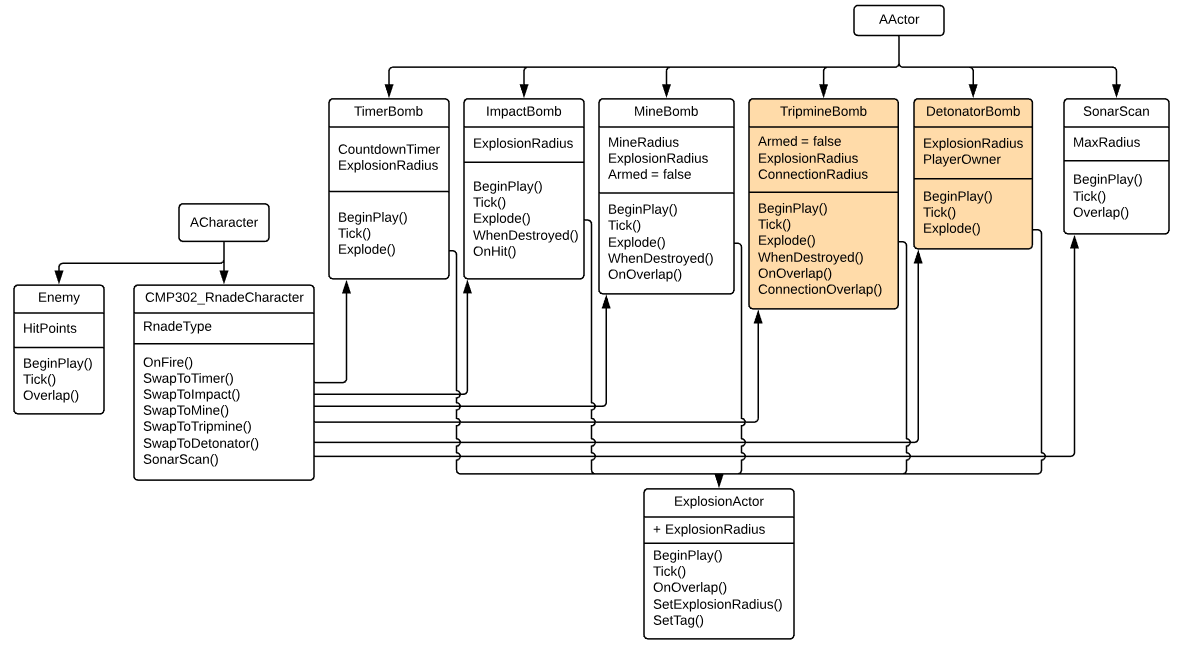
The application was created and intended to be used within Unreal Engine v4.22, however it can be opened in future generation, however it may jeopardise functionality. All code was created in Visual Studio 2017.

All development is being performed by a sole developer, therefore there will be limited -if any- artistic creation for the project, and heavy use of Unreal Engine’s Sample Content packs.

# Design

## UML

Attached is a UML Diagram for the developed system that covers all the C++ classes created. The coloured classes were not implemented in time, but for proof of concept in design have been included.



In this section I will cover each class shown in the UML above, each representing a component of the gameplay system created.

## R-nade Mechanics

As previously stated, there are five different R-nades that have been designed, and only 3 fully implemented at the time of submission. The reason for two being omitted from the submission was due to time and knowledge constraints, however upon a resubmission, the further two would be implemented and expanded upon.

The current list has linked priorities. Some may have multiple priorities, this is for extended features. Priority scales from low to high. The lower the number, the higher the priority.

### Impact Bomb

The impact bomb was designed to do as it sounds. Be thrown, and upon impact with any surface, create an explosion that damages and will knock back. Currently, the impact bomb can be thrown at a standard speed/distance, and will create a constant blast radius of whatever is defined in the blueprint. The constant blast means that no matter where in the sphere you are, you will take damage and –if a physics object- get knocked back.

Priority - 2

### Timer Bomb

The timer bomb is created to be thrown and bounce / roll around for a pre-set amount of time. The default designed time is 10 seconds. This means that after the R-nade leaves the players hand, it will start counting down, upon detonation, it will create a blast radius that is pre-set in the blueprint, again creating a constant blast radius. The timer bomb is designed to withstand other explosives, only to be knocked around, creating a strategic element to their use, and counter-use. As they are cubes, they can be quite unpredictable in the way they interact with the surrounding environment.

Extended use would allow the player to activate the countdown timer while holding the device. Meaning they could throw the device with however much time remaining that they desire. This would also allow the player to sacrifice themselves in an attempt to close distance and secure an elimination.

Priority – 1

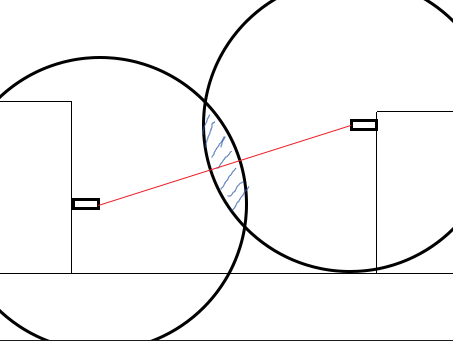
Priority – 6 (Extended use)

### Mine Bomb

Rather aptly named, the mine bomb is a puck-shaped explosive that when thrown, will travel and roll until it comes to a complete stop. Once the device lands stationary, it will arm itself, creating a sensor sphere that will only detect players. Note that this device CAN detect the player that has thrown it, so caution must be taken when traversing mined areas. The device itself can be blown and destroyed by any other explosive, meaning players can ‘defuse’ the explosive by alternative methods. The sensor-radius is defined in the blueprints, therefore can be fine-tuned by designers. Once detonated, it will create an explosion that is also pre-defined in the blueprint. It is worth noting, that if the mine is moved in any way, it will trigger its detonation. Meaning if you collide another non-player physics object into it, or move an object that it is placed on, the detonation will occur.

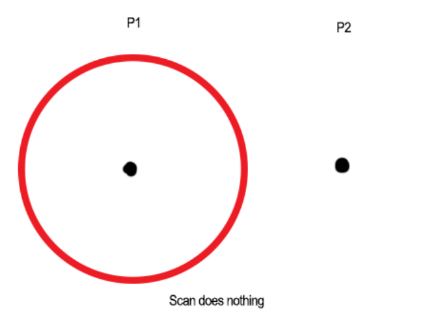
Priority - 3

### Tripmine Bomb

The tripmine bomb functions similarly to classic traps, in that you will need to throw two of them to create an actual case for detonation. The way this cylindrical trap works, is that one device will be thrown, and search for other devices in a blueprint-defined radius. If there Is another object in the area, it will create a link between the two objects, that if any object crosses, both traps will trigger. Ideally, the explosion should be large enough to cover any ground in between the two traps, this creating a well laid-trap. Multiple more devices can connect to create a mesh of traps, however only the latest trap will attempt to connect to ones in its radius.

Priority – 5

### Detonator Bomb

The detonator bomb is similar to a remote explosive, in that once thrown, can be detonated again by the press of a button. These pyramid-shaped bombs can only be detonated by the user, and function similarly to the timer bomb, in that they are effect by physics, thus allowing clever level manipulation. Once caveat with these, is that they can only be detonated if the user is within connection range, which will be pre-defined in the blueprints. Once detonated, will create an explosion like every other device.

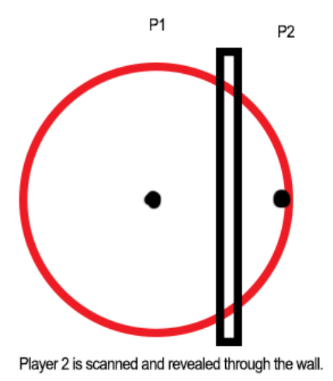
Priority - 7

## Other Mechanics

There are other mechanics that are planned, however not fully implemented. These will be listed here, though some are subject to change in functionality.

### https://i.gyazo.com/987b40d94c6087180eb1565c379dfc9c.pngSonar Scan

The sonar scan functions as a locating method for players to find enemies. Once the player activates their sonar, it will send out a pulse up to a maximum range (which again, is defined in the blueprint for editing). If an enemy is within the range, then the player will be alerted by an auditory tone, as well as gain a visible indication as to where the enemy is through the environment, in the form of a spherical dot. However, if the players scan is met with another players scan, both are cancelled out and will be placed on a cool down period. Both players will be alerted that they have been ‘counter-scanned’, but will receive no indication of where the scan originated. If a player is scanned, they will also be alerted they have BEEN scanned, but again, will not receive any information as to where from.

The scan also does not follow the player once used. The origin of the pulse will be the location the player was at when the ability was activated.

Priority - 5

### Weapon Swapping

It was designed that the player will start with only one type of R-nade, and must collect the other versions and their limited ammunition by eliminating other players, or by finding them laid about the map. Currently for demoing purposes, the player may swap weapons at any time and has an unlimited supply of them.

Extended use implies the rules of having limited R-nade types and spawning with only one type.

Priority – 4

Priority - 8 (Extended Use)

### Melee

There is also a melee function planned for when / if the player runs out of R-nades, meaning they can still attempt to take out other players, however less efficiently.

Priority – 10

# Development

Here I will go over my development method, discussing how each function works and how they could be further developed and carried forward, my main approach to development was to work things out and learn new things as I went forward. None of it was created first in Blueprints as proof of concept, and was all originally created in C++.

## Characters

### Player

For the player character, I started with using the First Person Unreal template and from there modifying and removing unnecessary code. My modifications included creating new functions to act as setters for changing R-nade types and adding a Sonar Scan function. I also used the Unreal Engine input properties to bind using the keys 1-3 as the values for swapping (1 = Timer Bomb, 2 = Impact Bomb, 3 = Mine Bomb). The Q key is also bound to call the Sonar function. The Fire function (which has a simple switch statement for the R-nade type) and the Sonar function are fundamentally similar, in that they both spawn objects, the only main difference being that the projectiles are spawned from the barrel of the ‘gun’. The gun is placeholder from the template, and is only used as a representation of how the R-nade will be fired. The scan, however, is spawned from the centre of the player.

Ideally, I would have liked to improve the firing mechanic, and potentially moving the entire code for launching the projectile (and mesh) over to the R-nades themselves, possible using a parent class from Actor, and adding all functionality to the R-nades in custom components that would be called dependant on which type was selected. This could also have solved some issues to do with device count (Ammo) and the Timer / Detonator Bomb extensions / implementations. The firing mechanic would likely be improved by changing the trajectory and force of the projectile, based on how long the fire key is pressed for. Also potentially adding an alternative fire mode that allows for short, under-armed throws, similar to the game Counter Strike: Global Offensive’s grenade throwing system.

### Enemies

In all reality, even the enemies provided in this system should be either inherited, or just also be the player class, since in a real-world situation, they would be player controlled opponents. However right now, they only serve as target dummies and offer no offensive or defensive capabilities. Since they were required to test the Mine (and non-implemented Tripmine) functionality, they use a simple nav-mesh component, along with blueprinted patrol paths. The enemies themselves are C++ created character classes that have a Hit Point variable included, which is adjustable from within the blueprints. The default for the HP is 100, and since each explosion deals 50 points of damage (more on that later) It will always take two explosions to kill.

It may also be nice to have enemies be able to throw certain types of R-nades, however this would more be for a better demonstration in a firing range on how the health system works. This is not required at all by the module and thus, is of little importance. However, having an NPC with the ability to send out a periodic scan, would have been a nice addition as it would have better allowed me to show how the Sonar Scan actually works in a real scenario.

## R-nades

### Timer Bomb

For the Timer Bomb, I first created a new actor class in C++ and then added a mesh component to give it the Cube mesh. I then created and attached a ‘Projectile Movement Component’ which allows it to be given an initial launch. If as stated above I was to add varying trajectories and throw powers, this would be where I would set them. I would likely pass values into the object spawned in the character class and use a ‘setter’ type function to make sure they get applied on the begin play. Inside the Tick() function, I’ve created a basic timer that runs off of DeltaTime and removes what is relative to 1 second from a countdown value each real-world second. After the timer hits 0, a function called ‘Explode()’ is then called to destroy the object and create a blast radius in its place. The method used for spawning the explosive is similar to the method used for spawning the R-nade in the character class, except after it is spawned, it then uses ‘setter’ methods to set the type of the explosive (currently not functioning as intended) and then setting the size of the blast sphere.

### Impact Bomb

This Impact Bomb was likely the easiest to implement and explain. For the basis of it, I created an Actor class and essentially used the FPS Sample code to implement it, as it was the first R-nade created. The way the device functions, it that it is spawned from the Character class, and then will travel until its mesh hits another collision component. This is called when the Collision Components OnHit function is called, that allows for comparing the different actors and components that have been struck. Here, the only logic is to destroy the projectile if the other actor in the world isn’t empty or the projectile itself. The explosion function is called from the ‘OnDestroyed()’ function. This allows the unique case of if an explosion hits the projectile, then it will detonate and destroy the projectile itself. Thus creating almost a chain of explosions.

### Mine Bomb

The Mine Bomb is similar in design to the Timer Bomb, in that it is comprised of a Mesh and a projectile component, however it also has a Sphere Component applied to it that serves as the sensor for the mine. The Sphere Component starts with an initial radius of 0 and his its own Overlap function, however, different from the Impact Bomb’s OnHit function, this is an OnOverlap function, that uses the ‘OnComponentBeginOverlap’ functionality. This acts as a check similar to on OnHit, except it will check not just the outer bounds for a hit, but the entirety of inside the sphere. This means it will trigger the function anytime there is another mesh or component inside the area. In the Tick() function, there is a check to see if the velocity of the R-nade is 0 or lower. If this condition is met. A bool of ‘Armed’ is toggled to true. While armed, the radius of the sphere component will be set to the value designated for design in the blueprint, and now the overlap function will come into effect. As a side note, if the velocity of the object is increased above 0 while the device is armed, a detonate function will be triggered. This means if any object attempts to move the device, it will activate.

While the device is armed, if any player character (meaning either the player themselves, or an enemy) walk into the sphere, the device will trigger due to the characters presence. This is caused by only checking if the components that are overlapping have the tag “Enemy” or “Player”.

### Tripmine Bomb

The tripmine bomb had minimal implementation at the time of writing this report, so it was best to omit the class from the final submission. However I have a good idea as to how this would be created.

Initially I created a similar layout to the Mine Bomb, in that there was a mesh component, projectile component and a sphere component. The object would function similarly, except that OnHit would be called upon impact with any other mesh, and then setting the rotation of the object to be that of the Normal Vector of the surface. Thus setting the Tripmine component to that of the vector. From here, the Sphere component will then be increased in size to a blueprint-set value (for designers to balance) that will check for an overlap of the same object type. Essentially checking if there is another tripmine within range. If there is, then it will create a CollisionBox from both of the actors, acting as the beam connecting them. If there is any overlap on this box, then both the tripmines will explode.

### Detonator Bomb

The Detonator Bomb would function almost identically to the Timer Bomb, in that it would be a physics enabled object, and instead of being detonated after a time as elapsed, it would instead have its detonation function bound to a control of the user. There would however be a collision sphere that would check if the owner (player who threw it) was within the sphere/range, allowing the detonation of it.

## Other

### Sonar Scan

The Sonar Scan is quite simple really. It consists of only a Sphere Component that increases in size up until a maximum range, and then destroys itself. The size is increased via DeltaTime manipulation which gives it a nice smooth increase in size. Currently it is being displayed via the engine editors ‘set visible’ function, for debugging and testing visualisation. As the sphere increased, it will be routinely checking the OnOverlap function to find if a player / enemy is inside the sphere. Currently, it only types out on screen that an enemy has been scanned. Though it is planned to change the custom render depth and applying an alternate material that allows for it to be rendered through walls. It was also planned for the scan to be stopped when another actor of the same type intersects with it, however this was not implemented due to the lack of testing conditions.

### Explosions

The explosion class is probably one of my biggest faults with the current system as it stands. This is due to my first attempt at creating the explosion effect, making it an actor instead of a component that would be called. As it currently stands, each ‘Explode()’ function, spawns the Explosion actor. However setting the explosion radius is the only thing that is easily changeable. Unfortunately there is no easy way as it currently stands for me to change the damage dealt. My first implemented attempt that sadly doesn’t function properly, was to set a Tag for each explosion when called, and that it’s damage calculation would be performed on the enemy class, checking for a collision that has the specified damage tag and then applying the numerical operation there. However, currently, the tag is applied after the only collision check, which means that the specified damage is not applied.

This could have been remedied by creating the explosion as a separate component class, and sending in the radius and damage tags from the R-nade that each component will be attached to.

# Conclusion

In conclusion, I feel as though the features developed are functional, but only to a point. Reflecting back on the work completed at the time of writing this, I am ashamed to admit that I feel as though this is a very weak submission and really has no right to be. I feel as though I could have utilised my time and resources a lot more efficiently, as well as better planned the overall design of my features. Since there was a lot of going back on previous designs to get things working in different ways after numerous different iterations.

In the future (be it either a resubmission, or my own desire to carry the development forward) I feel as though it would be best to once again, go back and rebuild from the ground up, using more personalised classes and components. These would be changing the explosion effect to a component that can be applied to any new kind of R-nade that would allow for different kinds of explosions (like lingering effects, different explosion effects, etc). As well as the creation of a new character class that would be more universal to be used as either a PC, friendly or enemy.

In short, I feel like I should have done a lot more work myself, though this is quite difficult due to the nature of Unreal Engine wanting to do a lot of complex work for you (such as collisions, etc). This may just me being hard on myself and the project itself and overlooking the work I HAVE actually done, but it feels as though I could have –and should have- implemented a lot more.

Everything is a learning experience, and I feel as though I have definitely gained a working understanding of Unreal Engine, even if that is not demonstrated in this submission and would gladly demonstrate the knowledge and skills I have gained in the future, if given the chance.

# References

Counter-Strike Wiki. (2019). *Grenades*. [online] Available at: https://counterstrike.fandom.com/wiki/Grenades [Accessed 1 May 2019].

Docs.unrealengine.com. (2019). *Unreal Engine 4 Documentation*. [online] Available at: https://docs.unrealengine.com/en-us/ [Accessed 1 May 2019].

En.wikipedia.org. (2019). *Counter-Strike: Global Offensive*. [online] Available at: https://en.wikipedia.org/wiki/Counter-Strike:\_Global\_Offensive [Accessed 1 May 2019].

Inoue, J., Dashiell, C. and Delgado, T. (2013). *Btooom!*. New York: Yen Press.

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

YouTube. (2019). *Unreal C++ Tutorial - Grenade Launcher!*. [online] Available at: https://www.youtube.com/watch?v=5XiBnL4W-mM [Accessed 1 May 2019].