Technical Design Document

LASER BEAT VR

Introduction

Synopsis

This document outlines the technical design of the game ‘laser beat’ - it does not include game design as laid out in the GDD. Laser Beat is a VR first person high energy shooter game, where targets spawn around you, and the player has to shoot them with high skill and precision.

This document can be used for reference when developing the game’s code, it outlines the technologies being used, outlines the desired coding practices throughout the project, and It presents the relationships between the different classes and objects in the game.

Liminal’s Technical Requirements

The game is being developed for a company in Australia, called ‘Liminal VR’, which means in order to publish on their platform, not only are there general technical virtual reality requirements but Liminal SDK requirements.

Technical Liminal requirements:

- “The only version of Unity we currently support is **2019.1.10f1”**
- “The only Scripting Runtime we currently support is .NET 4.x running on Mono as the Scripting Backend”
- “Experiences should consist of a single Unity scene”
- “Do not change Unity's global state from within your scripts”
- “Ensure your sound files have been normalised. This will assist your sound levels in being similar to those within the platform.”
- “To move the player, move VRAvatar”
- “These classes need to have a constructor without an argument if they use constructors.”
- “[Serializable] classes that reference a scriptable object need the scriptable object to have the [PreferBinarySerialization] attribute”

Game Liminal requirements:

- Highly optimised game for VR
- Targeted for the Oculus Quest 2
- Target constant 72Hz (Fps)

Architecture

Targets

Target Spawning System

This contains an all in one Spawner Script which deals with:

- Shape of the spawn area
- Pool size
- Spawn rate behaviour

Because all three of these functions are closely related under the definition of “Spawner” we will keep them all in one script. Although this breaks the SOLID principles mentioned later, we can separate them into different scripts if we have time.

Target Object

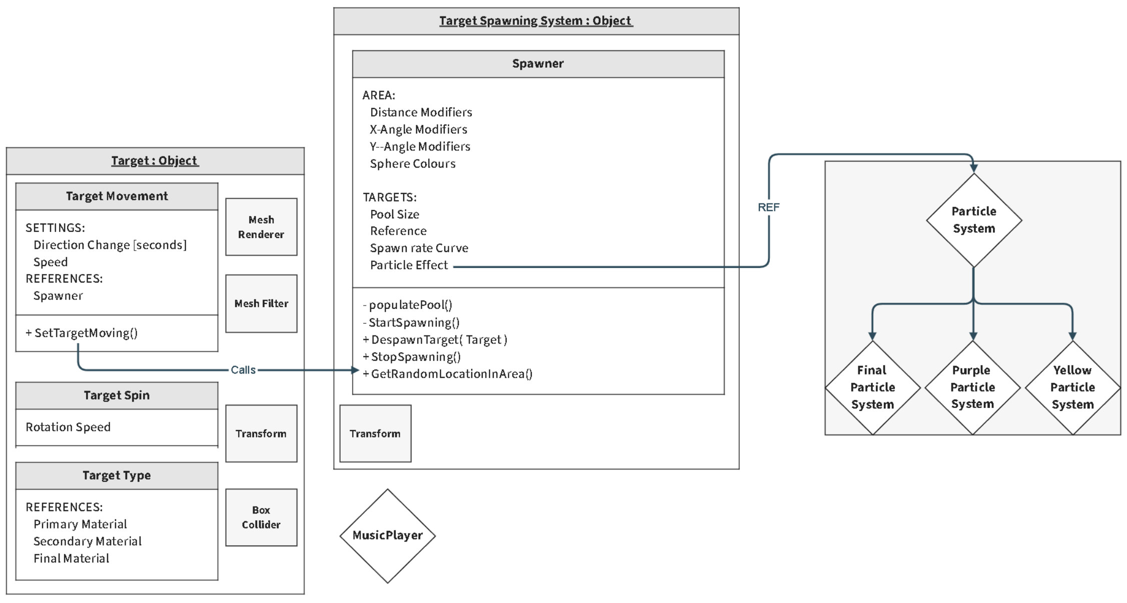
The target will have three custom made scripts:

- Target Movement: Deals with the movement behaviour
- Target Type: Replacement for tags, because we can’t use tags due to the Liminal SDK
- Target Spin: Deals with spin behaviour (Later to be refactored to baked animations to improve performance)

Particle System

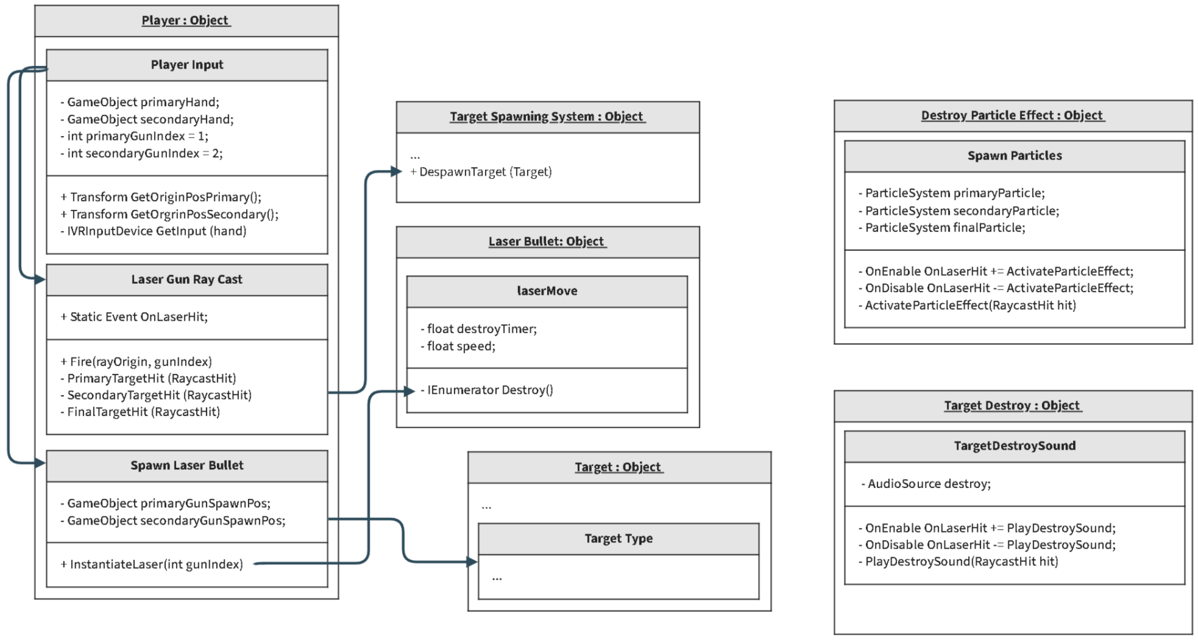
The particle system is the parent game object to three separate child variants:

1. Final Particle System - for the final target’s destruction (this has since been replaced)
2. Purple Particle System - for the purple target’s destruction
3. Yellow Particle System - for the yellow target’s destruction



Guns

The player game object deals with both player gun inputs and the firing mechanism:


Player : Object
Player Input
- GameObject primaryHand;
- GameObject secondaryHand;
- int primaryGunIndex = 1;
- int secondaryGunIndex = 2;
+ Transform GetOriginPosPrimary();
+ Transform GetOrgrinPosSecondary();
- IVRInputDevice GetInput (hand)
Laser Gun Ray Cast
+ Static Event OnLaserHit;
+ Fire(rayOrigin, gunIndex)
- PrimaryTargetHit (RaycastHit)
- SecondaryTargetHit (RaycastHit)
- FinalTargetHit (RaycastHit)
Spawn Laser Bullet
- GameObject primaryGunSpawnPos;
- GameObject secondaryGunSpawnPos;
+ InstantiateLaser(int gunIndex)
Target Spawning System : Object
...
+ DespawnTarget (Target)
Laser Bullet: Object
laserMove
- float destroyTimer;
- float speed;
- IEnumerator Destroy()
Target : Object
...
Target Type
...
Destroy Particle Effect : Object
Spawn Particles
- ParticleSystem primaryParticle;
- ParticleSystem secondaryParticle;
- ParticleSystem finalParticle;
- OnEnable OnLaserHit += ActivateParticleEffect;
- OnDisable OnLaserHit -= ActivateParticleEffect;
- ActivateParticleEffect(RaycastHit hit)
Target Destroy : Object
TargetDestroySound
- AudioSource destroy;
- OnEnable OnLaserHit += PlayDestroySound;
- OnDisable OnLaserHit -= PlayDestroySound;
- PlayDestroySound(RaycastHit hit)

- We are using Raycasts for the shooting logic - for performance
- VR Avatar parent game object, contains all the logic for the hand and head tracking
- As the player's existence and gun game objects are anchored to the VR avatar so the player input has been placed here
- There may be a case to move the laser gun ray cast and spawn laser bullet script to a seperate game object, but due to keeping the project simpler and having all the player associated logic within one place to easily maintain.

Technical Practices

Practices

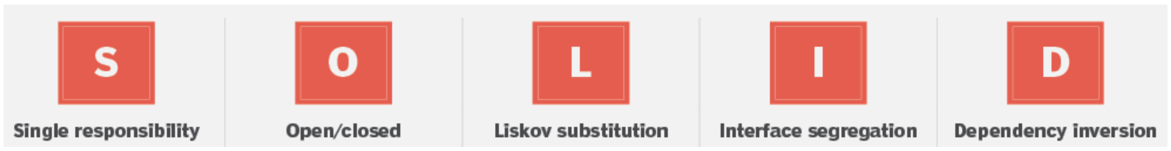
It is important to implement good coding practices throughout the process of coding, so that the code is readable for maintainability purposes. Except for the obvious practices in which we've already learnt, such as using PascalCase within the right context, here are some practices which we'd like to pick up :

- Always use access level modifiers.
- Only use a single declaration per line.
- Preface interfaces with an 'I'
- Don't let spawned objects clutter your hierarchy in runtime
- Keep well organised file a structure throughout the project, not retrospectively

These are only a few examples of conventional coding practices, but it is also important to follow coherent and up-to-date coding principles, such as SOLID…

The SOLID Principles

In this project we want to stick as close as possible to the SOLID convention. The SOLID principles are only applicable for object oriented design, and are widely regarded as standard within the OOP coding industry.



All five principles involves its own scope of concepts and practices, but can all be realised as a part of the project as we're using C#, a language which requires OOP design.

Single Responsibility

Single responsibility principle (SRP), the most simple principle, can be understood via two definitions, both of which have the same practice to implement, and describe the problem in two similar, but separate ways. The first definition that the single responsibility principle dictates is that an object class should only be responsible for one function/purpose.The second is that this principle should have only one reason to change. These two definitions are interlinked because if the class or method only has one purpose, then it can only have one reason to change, and that is to change the purpose of the class.

The benefits of this, although can take some getting used to, outweighs the cons. It reduces the complexity of the code, allowing it to be more readable which should lead to reduced time waste in both debugging and maintenance. An example for our game would be to separate the gun's raycast and the line gizmos into at the least separate methods, however, ideally, into separate classes in separate scripts.

Note: It also goes in tandem with the DRY (Don't repeat yourself) principle, which is not a part of SOLID, but it's useful to understand the practical use for.

Open/Closed

This principle is defined so that developers should be able to add new features / functions / extensions to an existing class without having to modify it. Or in other words, classes should be open to extension, but closed to modification. This is to avoid the issue regarding when the developer wants to add additional functionality to a class, they will have to change the class, which could cause issues which rely on the class being without change.

The implementation of this principle is hard to do correctly, but can be done through the use of abstractions and the Liskov substitution principle.

Liskov Substitution

The Liskov Substitution principle states that objects which are contained within a subclass must exhibit the same behaviour as the higher-level superclass it depends on. To avoid breaking the rule one must ensure that all children classes inherit from the parent class and do not break the functionality of that parent class.

If this principle is ignored, one's code is prone to run time error or exception or its functionality might not work as expected. Interfaces are commonly used among game developers in order to satisfy this principle.

Interface Segregation

The Interface Segregation principle states that one must create a separate client interface for each class, or in other words, "A client should never be forced to implement an interface that it doesn't use" - says Digital Ocean. This leads to a clearer, cleaner and more maintainable codebase, as it reduces the coupling time between two classes as it ensures that interfaces are tailored to the clients requirements.

For example, if one were to implement an audio library for their game, and call functions directly, and then want to use a different library, then you'd have to spend a lot of time rewriting code, unplugging the old library, and implementing the new one.

However if one were to create an interface called lets say 'AudioManager' which is called from multiple places throughout your script, which then calls the audio library, now when you can replace the library with contained errors within a single place, reducing the friction time it takes to replace the library calls.

Dependency Inversion

The Dependency Inversion principle is the last of the SOLID principles, and states that high-level modules should depend on abstraction, and leave the concrete implementations for the lower level inherited modules. Or from another angle, when a subclass is dependent on a higher-level class, the higher class should not be affected by any changes made to the subclass. Although these are technically two different concepts, they fall under one banner, called the 'Dependency Inversion Principle'.

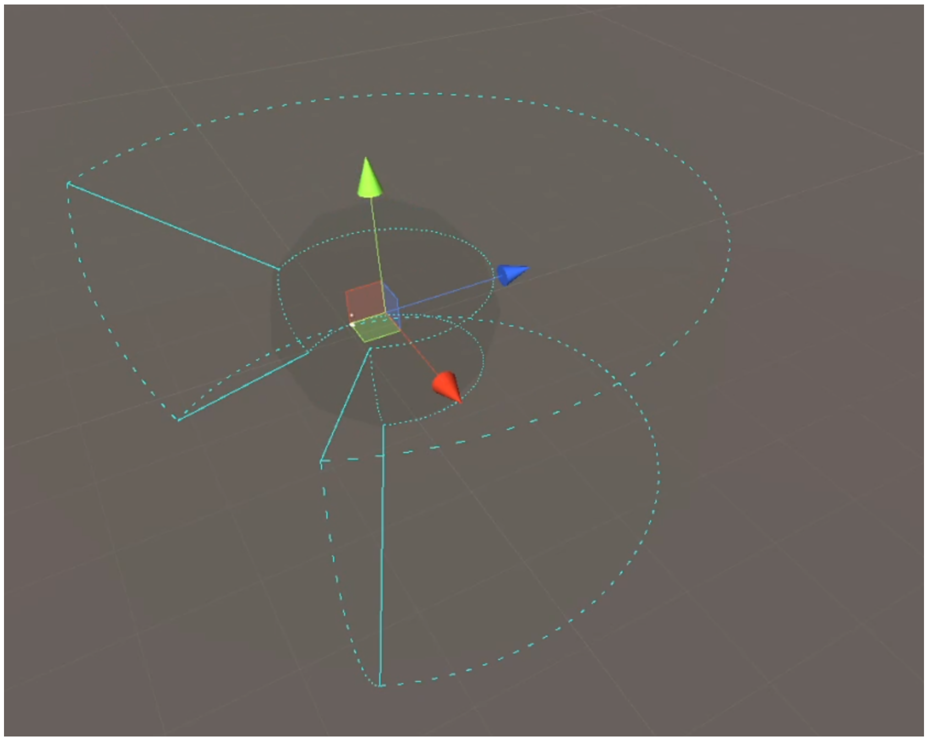
This principle utilises the advantages that comes with OOP programming, however could cause some problems so it must be used with caution. For example the testability of the modules could become difficult, as well as our ability to develop code in parallel.

Tool Creation

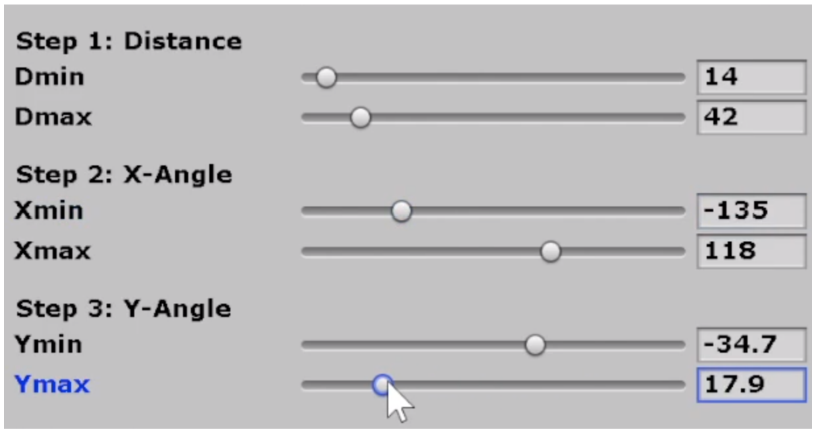
Although technical design normally solely consists of code which directly contributes to the game, other industry standard software development techniques can apply such as tool creation. Too creation aids for developers, especially for teams, to test and develop the game they're trying to create.

Custom tools allow game developers to design their workflow that specifically caters to their unique requirements, as the tools are to be tailor made.

In our game, Laser Beat, we want to be able to spawn targets within a given area which is not rectangular, but a radius around the player. We don't want the targets to be too close nor too far away, too far to our left behind us, or too far to our right behind us. Nor too high above, nor in the ground below us. And we want to be able to adjust this with visualisation on the fly.



This gizmo using tool, designed to create a custom area, will automate the otherwise repetitive processes, and allow for testing and tweaking the target area frequently, which should reduce development time and overall reduce errors.



Step 1: Distance

Dmin	14
Dmax	42

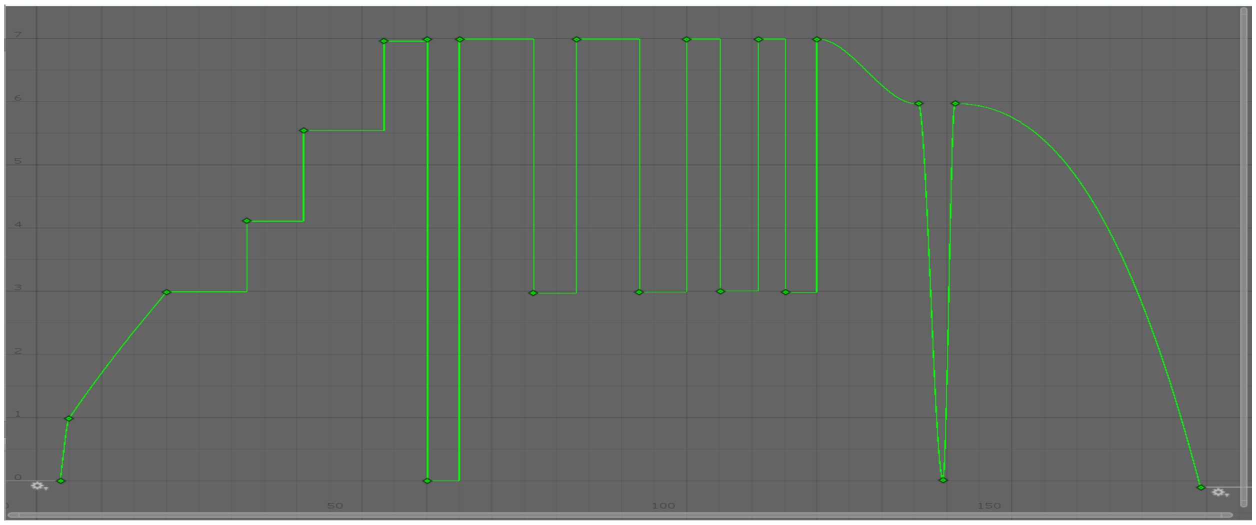
Step 2: X-Angle

Xmin	-135
Xmax	118

Step 3: Y-Angle

Ymin	-34.7
Ymax	17.9

This tool will be imperative considering the fact that Unity does not support complex shapes like this by default, however developing this custom tool will require considerable time, and time must be spent making this tool intuitive so that other members of the team can face the most lenient learning curve.

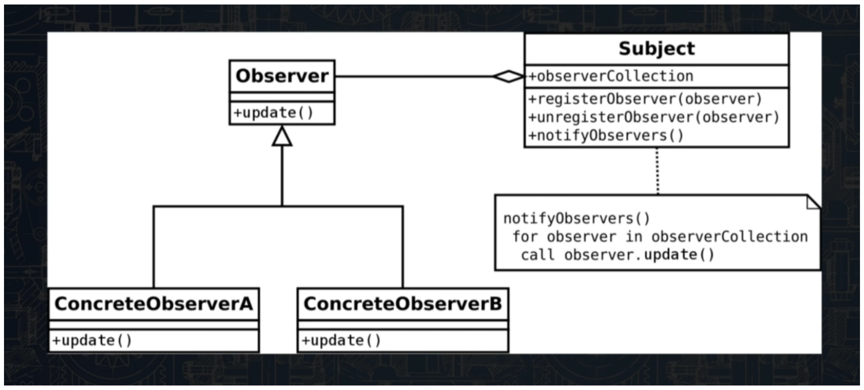


Another tool is the spawn curve, which allows the developer curate a custom spawning experience to be created, where the object pooling will spawn less when the music's at lows, and spawn more when the music is at highs.

Patterns

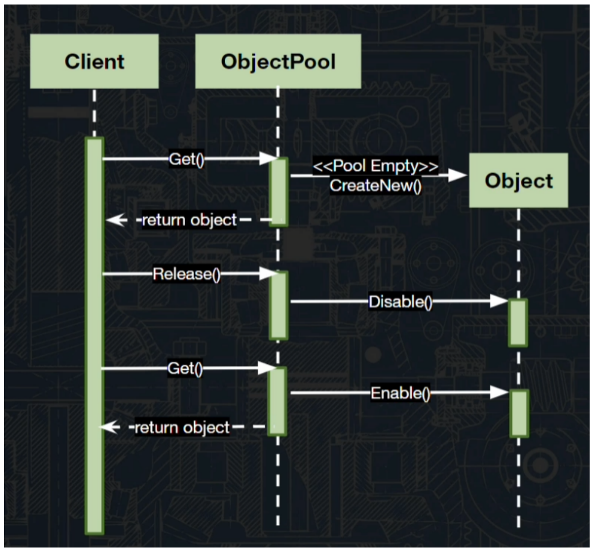
Observer pattern

The observer pattern is where an object, known as the subject, maintains a list of its dependents, called observers, that are notified of any state changes, typically by calling one of their methods.



This is useful as it enables a loosely coupled technical design, allowing the subject and observers to interact without having detailed knowledge of each other, and it improves the readability of code. This is to be implemented using Unity’s in-built observer pattern using Unity Events.

Object Pooling



This game has object pooling to improve run-time performance as it avoids having to instantiate and destroy objects during runtime, ultimately improving runtime performance and fps, which is imperative for the VR experience. Without object pooling, the low frames per second could cause motion sickness.

Object pooling will be used on the targets, which means the game will pre-instantiate targets, instead of creating and destroying targets on demand, just to re-instantiate them again.

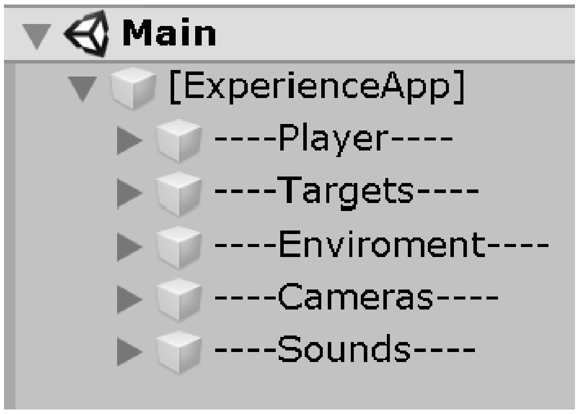
File Structure

Implementing well thought practices in place regarding the Unity project's file structure is crucial for maintaining a clean and organised codebase, making collaboration easier, and enhancing overall project maintainability.

We will do so by creating an intuitive file structure, with subfolders where appropriate, all being well named and easy to understand the contents of.

Scene Hierarchy

Same goes with the scene hierarchy, as with the file structure, well named parent objects. The Liminal SDK dictates that everything in the scene must be parented by the ExperienceApp object. This means when spawning targets we will have to spawn them in as children.



```
Main
  [ExperienceApp]
    ----Player----
    ----Targets----
    ----Enviroment----
    ----Cameras----
    ----Sounds----
```

The hierarchy organisation shown in the screenshot above is how the game objects in the scene will be organised, separated by: *player, targets, environment, cameras and sound.*