# Technical Design Document

Gravity Combat

A game by Floating Toast Studios

# Gravity Combat

## Technical Design Document

# Naming Convention

## Folder Naming:

**We use UpperCamelCase.** That means all words are written together, each word start with a capitalized letter.

Examples:

- Material
- Textures
- Scenes
- MapDatabase

## Asset Naming:

We use **Snail_Case** for Assets. All words are separated with an underscore(_) and the first letter of a word is always capitalized.
Except scripts, they are using **UpperCamelCase.**

Examples:

- VFX_Heal
- Texture_Heal_Effect
- Weapon_Pistol_Texture
- Map_Deep_Space

Scripts: (based on code architecture)

- PlayerMovementController
- PlayerMovementView
- MainMenu

Tip / Hint:

→ Descriptive_Class_Variation
→ Descriptive_Class_SubClass_Variation
→ Class_Descriptive_Variation
→ Class_Descriptive_SubClass_Variation

Use general descriptive names. This also improves the file search.
→ VFX_  for Effect
→ Mat_ for Materials
→ Texture_ for Texture
→ Mesh_ for Meshes

# Version control

## General

Version control will be done with Mercurial and TortoiseHg as graphical interface over the platform BitBucket.

The repository can be used to push all directly related project things. A good structured repository will improve the teams' efficiency.

## Commits

There are certain things you have to follow before you push a Commit !

- No **unnecessary file changes** e.g personal settings, changes to Game Objects you don't need for this commit
- check out your file size, big files waste our repository storage
- every pushed file will be in the repository, don't **push carelessly**
- consider the correct **naming convention**
- use the **correct branch**
- encountering a **merge conflict**, talk to the person / department that is responsible for it

## Commit Messages

There are 3 headlines Added, Updated and **done**.

**added**
Is for things you just put in the project, things you just started to working on.
**updated**
For things that maybe changed frequently, restructuring folder, renamed files, script changes, prefab changes.
**done**
Everything that you consider as done or completed.

```
added:
- background music
- weapon model xx

updated:
- player movement speed
- player turn speed
- Lobby Room Panel, can now be used as a button

done:
- Pistol Projectile VFX
- background music volume
- player movement script
```

## Branches

Branches should be named accordingly to the feature you are working on. **Snail_Case** is our naming convention.

Consider **splitting your working directory** to put **each feature on its own Branch**.

Be aware of **Merge Conflicts** if two or more people **working on the same file,** Scene, Prefab,, Material and so on.

Example :
VFX
VFX_Audio
Map_Blockout
Score_Mechanic

Branches can be merged if a feature is done and complete, consider to ask for a second opinion to check if your branch is cleaned up before you **merge with the default branch.**

# Folder Structure

Declaration:

- Assets - Unity's Root Project Folder
  - Folder - Class Root Folder
    - SubFolder

- Assets
  - Import
    - SCT - Scriptable Text
    - Photon
  - Materials
    - Player
    - Weapon
    - Environment
  - Textures
    - Player
    - Weapon
    - Environment
  - Prefabs
    - Player
    - Weapon
    - Environment
  - Scenes
  - Scripts
    - Player
    - Menu
      - Login
      - Lobby
      - Options
  - UI
    - Button
    - Panel
    - LoadingBar
  - MapDatabase
  - Editor
  - Plugin

# Code Structure

```
using System;
using UnityEngine;

Scripting component   Oliver Sradnick *
public class CodeConvention : MonoBehaviour
{
    public static float StaticVariable = 1;
    public const string ConstString = "Words";

    #region Events

    public Action<int> OnEventName;

    #endregion

    public GameObject GameObjectVariable = null;
    public int IntVariable = 1;
    public float Property { get; set; } = 0;

    [SerializeField] private float FloatVariable2 = 0.0f;
    [SerializeField] private float FloatVariable = 0.0f;
    [SerializeField] private bool BoolVariable = false;

    private float m_localFloat = 1.0f;

    Event function   Oliver Sradnick
    private void Start()
    {
    }

    Oliver Sradnick *
    public void DoSomething()
    {
        for (var i = 0; i < 10; i++)
        {
            OnEventName?.Invoke(i);
        }
    }

    Oliver Sradnick
    private void DoSomethingRPC()
    {
    }
}
```

# A (deep) Dive into the Code Base

Let's start with the mostly used pattern in this project.
MVC , *Model View Control* Pattern. For those who are not familiar with it i'll  explain how i implemented it.

## Basics

*Model :*

It's the place where the logic is placed. Variables, stats, data, components like rigidbody, colliders are the correct choice.
The model don't know that the controller or the view exists, but can communicate with other models . The classes should describe the data it is working with.
Should have accessible events.

*View :*

Handle visuals, with events coming from the model. View only knows the model components like mesh renderer, sprites, images and in brackets GUI.

*Controller :*

A set of classes that handles communication from the user. In Unity also a (UI)button can be a controller because it handles the underlying mechanics on its own.
The controller is aware of a model.

## Player

Your entry point should be the player. It uses the above explained pattern.

*PlayerController.cs*

Works with PlayerMovemtModel, PlayerBootsModel and WeaponModel.
It only checks for input and calls a method on the models.

```
private void Movement()
{
    var horizontal = Input.GetAxis("Horizontal");
    var vertical = Input.GetAxis("Vertical");

    m_playerMovementModel.PlayerInput(horizontal, vertical);
}

Frequently called   1 usage   oliversradnick
private void UseBoots()
{
    if (Input.GetKeyDown(KeyCode.Space))
    {
        m_playerBootsModel.Use();
    }
}
```

*PlayerMovementModel.cs*

Proceed the input from PlayerController.
All code basically only moves the player. Around platforms, walls, corners. Rotates the player if not grounded. And some collision forces for a floaty like behaviour.

Movement

```
Frequently called   1 usage   Oliver Sradnick +1
internal void PlayerInput(float horizontal, float vertical)
{
    MoveDirection = new Vector3( x: horizontal,  y: vertical,  z: 0);
}

/// Creates direction, based on input direction and ground normal, to move the Rigidbody. ...
1 usage   Oliver Sradnick +1
private void Movement(Vector3 input)
{
    var dir = m_wasdMovement ? Camera.transform.TransformDirection(input) : transform.TransformDirection(input);

    //transform direction accordingly to the ground normal
    var projectOnPlane = Vector3.ProjectOnPlane( vector: dir, m_groundNormal);

    projectOnPlane = Vector3.ClampMagnitude(projectOnPlane,  maxLength: 1);
    Rigidbody.MovePosition(Rigidbody.position + Time.fixedDeltaTime * MovementSpeed * projectOnPlane);
}
```

*PlayerMovementView.cs*

Is on the first child(root player 3D model) of the player prefab. That's the heart of the IK system and also handles animation. The View needs the PlayerMovementModel.cs to work. It basically proceeds specific data to play the correct Animation or reacts to events(Action) that come from the PlayerMovementModel.cs

Reaction to Collision and Animation,

```
private void CollisionHit()
{
    if (CollisionHitEffect != null)
    {
        CollisionHitEffect.Play();
    }
}

Event function   OliverSradnick +1   ··· More
private void Update()
{
    if (!PlayerMovementModel.PhotonView.IsMine) return;

    var modelInput = PlayerMovementModel.MovementInput;

    if (modelInput != 0 && PlayerMovementModel.IsGrounded)
    {
        Animator.SetFloat(SpeedHash, modelInput);
    }
    else
    {
        Animator.SetFloat(SpeedHash,  value: 0);
    }
}
```

## Networking

*Lobby:*

While in Lobby the Player is able to join or host a Game. For join there is a Room Overview and a Matchmaking button.
Some options to Host a room are also available.

In this Picture you see what for Properties and Option are created for a single Room.

```
var parsedEnum = (Mode) System.Enum.Parse(typeof(Mode), GameMode.options[GameMode.value].text);

//Create a new Hashtable and store room Properties
var customRoomOption = new Hashtable()
{
    {RoomProperties.Map, m_pickedMap.GetMapName},
    {RoomProperties.Gamemode, parsedEnum},
    {RoomProperties.Hash, Random.Range(1,10000)}
};

//set Room Options and store the properties in it
//To get/set properties in the Lobby set CustomRoomPropertiesForLobby
var roomOptions = new RoomOptions()
{
    BroadcastPropsChangeToAll = true,
    IsVisible = true,
    IsOpen = true,
    PublishUserId = true,
    MaxPlayers = MaxPlayerPerRoom,
    CustomRoomPropertiesForLobby = new string[]
        {RoomProperties.Map, RoomProperties.Gamemode, RoomProperties.Ping},
    CustomRoomProperties = customRoomOption,
};
```

## Layer of Abstraction

Photon offers custom Properties for Lobby, Room and Player. This is used to store for Lobby / Room , Map , GameMode, Ping and for Player kills,death,assists, Score, ReadyState.

On top of the given custom Properties I put one Layer to have 3 classes to handle properties, set-, add-, get Properties. On top of this I put another Layer to specify the Properties I like to set.

Methods used to directly Change specific Properties without knowing what's hiding behind.

```
/// <summary>
/// Assign Score to Player Properties.Will Override the current Score.
/// </summary>
/// <param name="player">Target Photon Player</param>
/// <param name="value">Score</param>
1 usage   OliverSradnick +1
public static void SetKill(this Player player, int value)
{
    player.SetPropertyValue( property: PlayerProperties.Kills, value);
}
```

Synchronization

For the Player a State Buffer Synchronization Technique is used .
Using a State Buffer Synchronization Technique seems to be a good Option.

This will buffer the incoming data (Position,Velocity, Rotation) and tries to reproduce the exact movement on all Clients, it basically interpolates between states.

If the ping is too high and the Server Timestamp different from the previous and the next state is too high , it switches for a short amount of time to Extrapolation. Be aware if the allowed Extrapolation Time is too high, the movement will not be the same on all Clients.

That's how the Interpolation is build up.

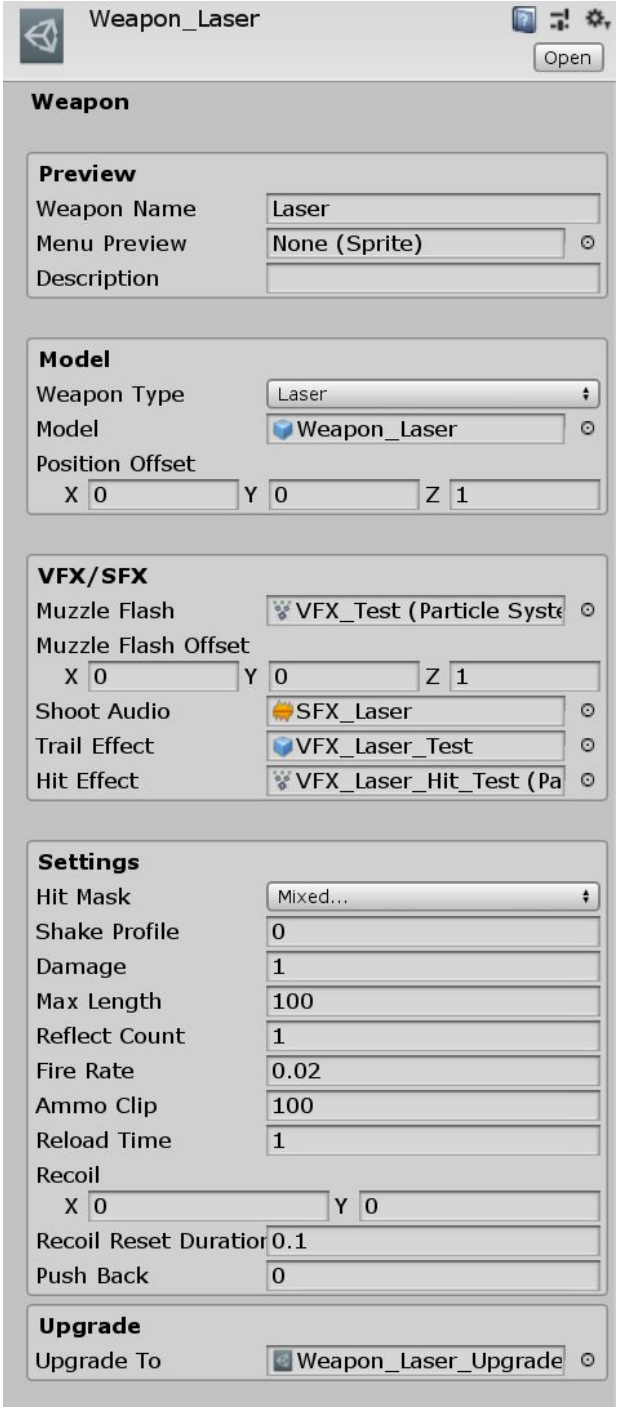
```
private void Interpolation(double interpolationTime)
{
    Debug.Log( message: "Interpolation");

    for (var i = 0; i < m_stateCount; i++)
    {
        //closest state that matches network Time or use oldest state
        if (m_stateBuffer[i].Timestamp <= interpolationTime || i == m_stateCount - 1)
        {
            //closest to Network
            var lhs = m_stateBuffer[i];
            //one newer
            var rhs = m_stateBuffer[Mathf.Max( a: i - 1,  b: 0)];
            //time between
            var length = rhs.Timestamp - lhs.Timestamp;

            var t = 0.0f;
            if (length > 0.0001f)
            {
                t = (float) ((interpolationTime - lhs.Timestamp) / length);
            }

            Rigidbody.position = Vector3.Lerp( a: lhs.Position,  b: rhs.Position, t);
            Rigidbody.rotation = Quaternion.Slerp( a: lhs.Rotation,  b: rhs.Rotation, t);
            break;
        }
    }
}
```

You can find this in the PlayerSyncModel.cs

I used this articles https://developer.valvesoftware.com/wiki/Source_Multiplayer_Networking and https://gafferongames.com/ from Glenn Fiedler.

## Weapons

Weapon.cs creates ScriptableObjects which are just data container to store Information about weapons and allows easily to created tons of new weapons.
Each weapon should assign to the Weapon Database, this is needed to easily sync an index over network to tell all Clients which weapon is used.




WeaponModel.cs is where you could assign a new Weapon and it creates in Play Mode the Weapon.

The underlying Mechanics are added or deleted dynamically.
WeaonBase.cs is the core of each Weapon, to create new mechanics you have to inherit from it.

*WeaponBase*

Got some basic mechanics, recoil, push back, audio , reloading some of it can be overwritten to extend the default behavior. It also connects to the Weapon view.

```
public virtual void Initialize(WeaponSocket weaponSocket, Weapon weapon, WeaponHud hud, AimOrigin aimOrigin,
    HitEvent hitEvent){...}

/// <summary>Create Weapon GameObject if available.</summary>
1 usage  Oliver Sradnick +2
private void CreateWeapon(){...}

/// <summary>Creates MuzzleFlash if not assigned default Particle System will be created.</summary>
1 usage  Oliver Sradnick +1
private void CreateMuzzleFlash(){...}

5 usages  Oliver Sradnick
protected void OnFirstShot(){...}

/// <summary>Should be used to creat an Individual Shooting mechanic.</summary>
Frequently called  1 usage  5 overrides  Oliver Sradnick
public abstract void Shoot();

Frequently called  3 usages  2 overrides  Oliver Sradnick
public virtual void StopShooting(){...}

/// <summary>Decrease Current Ammo, reload if empty.</summary>
5 usages  Oliver Sradnick
protected void ReduceAmmo(){...}

/// <summary>Start IEnumerator -> DoReload.</summary>
Frequently called  2 usages  Oliver Sradnick
internal void Reload(){...}

/// Performs a delay and resets the Ammo to max. ...
Frequently called  1 usage  Oliver Sradnick
private IEnumerator DoReload(){...}

/// Used for effects. ...
[PunRPC]
5 overrides  Oliver Sradnick
protected abstract void ShootEffectRpc(Vector3 vec3);
```

GameModeBase.cs is the base class for all Game Modes.
There are just some methods that you have to override to create a new GameMode.

```
/// Should be overriden for new GameModes to add a individual Team management. ...
1 usage  5 overrides  Oliver Sradnick
protected abstract void Assign(Player player);
```

To create a individual team management,
Gamemode.cs contains as default  a Team property to do this easier.

```
/// Can be implemented for specific condition when a gameMode usually ends. ...
1 usage  3 overrides  Oliver Sradnick
protected virtual bool WinCondition(){...}
```

Each Game Mode can have a different win condition which will be check in *UpdateWinCondition()*

```
/// Should return the leading Player, Team or specific GameMode stuff. ...
1 usage  5 overrides  Oliver Sradnick
public abstract string GetLeading();
```

Should return a specific message, this will be announced and every Player will be aware of who won or lost the match.

```
/// Implementation should be in a fixed order. ...
Frequently called  1 usage  5 overrides  Oliver Sradnick
protected abstract string[] ConfigStats();
```

A array of string contains informations for for e.g UI.
There is specific order you have to parse these informations.
[0] Friendly [1] Opponent [2] Counter [3] Bar
e.g

```
0+1 usages  Oliver Sradnick
protected override string[] ConfigStats()
{
    Stats[0] = GetLocalTeamScore().ToString();
    Stats[1] = GetEnemyTeamScore().ToString();
    return Stats;
}
```

Friendly should store information like points from a team the player is in.
Counter could be for a specific amount of kills the players has to made.

The UI will just show what is inside of these array.

MatchModel.cs is responsible for a working Match. If something is not working properly while playing check it out.

Break down :
- keeps track of joined clients
- handles pre and match - timer
- setup selected Game Mode
- updated Game Mode win condition and reacts to it

match start snippet
Basically it unloads the pre load timer, the master client send spawn event to all clients and a new Timer will be initialized.

```
private void OnPreloadTimerFinished()
{
    m_preloadTimer.Deinitialize();
    m_preloadTimer.RemoveListener(OnMatchTimerFinished);

    if (PhotonNetwork.IsMasterClient)
    {
        m_matchSpawn.SpawnAll();
    }

    if (ScriptableTextDisplay != null)
    {
        ScriptableTextDisplay.InitializeScriptableText( listPosition: 6, transform.position,  msg: "GO !");
    }

    OnTimerStarted?.Invoke(m_timer);
    OnMatchStart?.Invoke();
    m_matchStarted = true;

    m_timer.Start(PhotonNetwork.CurrentRoom.GetMatchTime());
}
```

game mode setup snippet
First lines it loads the selected game mode and gives the loaded mode to other components which then use it to act correctly.

```
private void SetupGameMode()
{
    m_currentModeBase = PhotonNetwork.CurrentRoom.GetGameMode();
    m_currentModeBase.HandlePlayer();

    if (m_currentModeBase is KillTheKing)
    {
        (m_currentModeBase as KillTheKing).SetHealth();
    }

    OnReceivedGameMode?.Invoke(m_currentModeBase);

    m_currentModeBase.OnConditionReached += OnMatchEnd;

    m_matchSpawn.SetGameMode(m_currentModeBase);
    m_matchRespawn.SetGameMode(m_currentModeBase);
}
```

If you are looking for our Character Animations you will sadly just find some half animations. The complete upper body is controlled via Inverse Kinmatic.


I'm using a layered animation controller with animations that only have keyframes for the legs.
From Hands to Hips it just follows the mouse position.

```
public void HandleShoulderRotation()
{
    RightShoulderIK.LookAt( worldPosition: AimPoint, transform.up);

    var rightShoulderPos = RightShoulderBone.TransformPoint(Vector3.zero);
    m_ikHelper.transform.position = rightShoulderPos;
    m_ikHelper.transform.parent = transform;

    RightShoulderIK.position = m_ikHelper.transform.position;
}

/// <summary>
/// Convert a Aim Position according to the Mouse position, little delayed.
/// </summary>
Frequently called   1 usage   OliverSradnick +1
public void Aim()
{
    var mousePos = Input.mousePosition;
    var ray = Camera.ScreenPointToRay(mousePos);
    var diff = -ray.origin.z / ray.direction.z;
    var target = ray.origin + ray.direction * diff;
    AimPoint = Vector3.SmoothDamp( current: AimPoint, target, ref m_aimDampingVelocity,  smoothTime: 0.05f);
}
```

Unity makes it pretty simple to control certain bones, but you need at least one empty animation to override those bones.

```
private void SetLookAtPosition()
{
    m_aimAtPosition = PlayerMovementModel.PhotonView.IsMine
        ? PlayerMovementModel.AimPoint
        : PlayerAnimSyncModel.NetworkAimPos;

    this.m_lookAtPosition = m_aimAtPosition;

    m_lookAtPosition.z = transform.position.z;

    var dist = Vector3.Distance( a: m_lookAtPosition,  b: transform.position);

    if (dist > UpdateLookPosThreshold)
    {
        m_targetPos = m_lookAtPosition;
    }

    m_ikLookPos = Vector3.SmoothDamp( current: m_ikLookPos, m_targetPos, ref m_iKVelocity, SmoothTime);

    Animator.SetLookAtPosition(m_ikLookPos);
}

1 usage   Oliver Sradnick
private void SetHandWeight()
{
    Animator.SetIKPositionWeight(AvatarIKGoal.RightHand, RightHandWeight);
    Animator.SetIKPositionWeight(AvatarIKGoal.LeftHand, LeftHandWeight);
}

1 usage   Oliver Sradnick
private void SetHandPosition()
{
    Animator.SetIKPosition( goal: AvatarIKGoal.RightHand, RightHandTarget.position);
    Animator.SetIKPosition( goal: AvatarIKGoal.LeftHand, LeftHandTarget.position);
}
```

Take a look at PlayerMovementView.cs for bone handling ,PlayerMovementModel.cs for aim position calculation and animation controller for the setup.

Putting all together gives a pretty solid result.

[Click for Video]