Bouncy Pumpkin Tutorial

2020 10/14
Kohei Nakajima

Bouncy Pumpkin is a demo using Unity Tiny. A player collects potions scattered around a spooky village. Unity Tiny provides faster loading time and faster gameplay compared to standard Unity Engine. While it's still in Preview Mode, Tiny(DOTS) already has proven to be the next trend in Game Development. The following is a tutorial mainly on programming development.

1. Installation

The following applications and dev kits are required for building a package for web and .NET Builds. (In this case, .NET Build is used for Debug and Testing.)

Unity Hub (any version)
Unity 2020.1.2f1 or above
Visual Studio 2019 Community modify with .NET desktop development and Desktop development with C++
Windows 10 SDK [https://developer.microsoft.com/en-us/windows/downloads/windows-10-sdk]
TinyPhysics and TinyRacing from [https://github.com/Unity-Technologies/ProjectTinySamples]

(Recommended)
Rider and Unity Support Plugin *Note: Enable Unity Support Plugin after installation.

2. Scene Setup

Copy TinyPhysics Project folder and rename it to TinyPhysicsTutorial. In **Unity Hub**, select the **Projects** menu button and add the TinyPhysicsTutorial folder. Once you open the project, navigate to Assets / Scenes folder in **Project** Tab. Select TinyPhysics and check **Hierarchy** Tab. Click a triangle icon next to TinyPhysics and select Subscene and load **7. Advanced Physics**. Check the box right next to this subscene item. Go to Unity Asset Store on any web browsers, and download the following assets.

Bretwalda Halloween - Free
https://assetstore.unity.com/packages/3d/props/food/bretwalda-halloween-74177

Medieval Cartoon Village Pack - VR/Mobile - $10
https://assetstore.unity.com/packages/3d/environments/fantasy/medieval-cartoon-village-pack-vr-mobile-16200

Potions, Coin And Box of Pandora Pack - Free
https://assetstore.unity.com/packages/3d/props/potions-coin-and-box-of-pandora-pack-71778

FANTASTIC - Halloween Pack - Free
https://assetstore.unity.com/packages/3d/environments/fantasy/fantastic-halloween-pack-154321

When you import assets, many Prehab and game objects have colliders script attached. All colliders need to be replaced by Physics Shape in order to compile. For Particle System assets, you can either remove them or minimize the effect types and include **Unity.Tiny.Particles**. reference in the Assembly Definitions (**.asmdef**) in **Scripts** folder. **Unity.Tiny.Particles** has limited effects compared to the standard Particle System.
https://docs.unity3d.com/Manual/ScriptCompilationAssemblyDefinitionFiles.html

3. Intro to Entities.ForEach

In Object Oriented Programming (**OOP**), a standard way to add movement and interactivity to game objects is to write codes inside **MonoBehavior.Update()**.
https://docs.unity3d.com/ScriptReference/MonoBehaviour.Update.html
In this tutorial, we are going to use multithreaded Data-Oriented Technology Stack (**DOTS**). In DOTS, the equivalent to **MonoBehavior.Update()** is **protected override void OnUpdate()**
As you will find in the Tiny Physics's **Assets/Scripts** folder, there are **Systems** and **Components** folders, most of the gameplay codes you write will be organised there.
SystemBase is a class that replaces MonoBehavior in general. The following is a simple example on how to move an entity or a game object.

SystemBase Example

```
public struct Position : IComponentData
{
    public float3 Value;
}

public struct Velocity : IComponentData
{
    public float3 Value;
}

public class ECSSystem : SystemBase
{
    protected override void OnUpdate()
    {
        // Local variable captured in ForEach
        float dT = Time.DeltaTime;
```

```
        Entities
            .WithName("Update_Displacement")
            .ForEach((
            ref Position position,
            in Velocity velocity
            ) => {
                position = new Position()
                {
                    Value = position.Value + velocity.Value * dT
                };
            }
        )
        .ScheduleParallel();
    }
}
```

https://docs.unity3d.com/Packages/com.unity.entities@0.11/api/Unity.Entities.SystemBase.html#Unity_Entities_SystemBase_Entities

Entities.ForEach is a ForEachLambdaJobDescriptionMethod method which constructs a LambdaJob (a block of codes) for each entity, and sends it to system queries. a list of **ref**'s and **in**'s finds matching game objects in the scene with all the items present in **Inspector**. In the case above, all the game objects with **Transform** and **Physics Body** active in **Inspector** will have their **position**'s affected by their own **velocity** values.
https://docs.unity3d.com/Packages/com.unity.entities@0.11/manual/ecs_entities_foreach.html

4. Character Movement / Jump Setup

Copy/Paste Worldplacement Transform value of Pumpkin GameObject to the character GameObject and parent it to the character GameObject.

In **Jumper.cs** Component, add a new variable, timer. Check the **Character** gameobject in **Inspector** to see the update.

```
using Unity.Entities;

namespace TinyPhysics
{
    [GenerateAuthoringComponent]
    public struct Jumper : IComponentData
    {
        public float jumpImpulse;
        public float timer;
    }
}
```

In the **JumpSystem.cs**, modify the existing JumpSystem by applying constant Y-Axis impulse. To move the player vertically, we use an **extension method** of **Unity.Physics** called,

ApplyLinearImpulse(ref PhysicsVelocity, PhysicsMass, float3) which is a Component Extension of Unity.Physics.Extensions. What is an extension method? Here is an explanation. https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/extension-methods

Unity.PhysicsVelocity.ApplyLinearImpulse

```
public static void ApplyLinearImpulse(this ref PhysicsVelocity velocityData,
PhysicsMass massData, float3 impulse)
```

https://docs.unity3d.com/Packages/com.unity.physics@0.0/api/Unity.Physics.Extensions.ComponentExtensions.html

Here is the modified **JumpSystem.cs**.

```
using Unity.Entities;
using Unity.Jobs;
using Unity.Mathematics;
using Unity.Physics;
using Unity.Physics.Extensions;

namespace TinyPhysics.Systems
{
    public class JumpSystem : SystemBase
    {
        protected override void OnUpdate()
        {
            var deltaTime = Time.DeltaTime;

            Entities
                .ForEach((
                ref PhysicsVelocity velocity,
                ref PhysicsMass mass,
                ref Jumper jumper
                ) => {
                    jumper.timer += deltaTime;

                    if (jumper.timer > 1f)
                    {
                        // Jump by applying an impulse on y Axis
                        velocity.ApplyLinearImpulse(mass, new float3(0,
                        jumper.jumpImpulse, 0));

                        // Reset timer
                        jumper.timer = 0f;
                    }
                })
                .WithoutBurst()
                .Run();
        }
    }
}
```

The timer regulates the interval of the impulses. Otherwise the player will fly up.

Let's work on the movement now.

In **Movable.cs** (originally Moveable) Component, add **currentSpeed**.

```
using Unity.Entities;
using Unity.Mathematics;

namespace TinyPhysics
{
    [GenerateAuthoringComponent]
    public struct Movable : IComponentData
    {
        public float moveForce;
        public float3 moveDirection;
        public float currentSpeed;
    }
}
```

In **MovementSystem.cs**, we will modify Tiny Racing's cart movement mechanism to achieve player movement. We use an instance field of **PhysicsVelocity** to directly affect transform value. This gives a more deterministic result of player movement. For the look at rotation, we use PhysicsVelocity's extended method as in Tiny Racing. Here is the actual code. https://github.com/Unity-Technologies/ProjectTinySamples/blob/master/TinyRacing/Assets/Scripts/TinyRacing/Systems/MoveCar.cs

Unity.PhysicsVelocity.Linear

```
public float3 Linear
```

Unity.PhysicsVelocity.SetAngualrVelocityWorldSpace

```
public static void SetAngularVelocityWorldSpace(this ref PhysicsVelocity
bodyVelocity, in PhysicsMass bodyMass, in Rotation bodyOrientation, in float3
angularVelocity)
```

Here is the modified **MovementSystem.cs**.

```
using Unity.Entities;
using Unity.Mathematics;
using Unity.Physics;
using Unity.Physics.Extensions;
using Unity.Transforms;

namespace TinyPhysics.Systems
{
    /// <summary>
    ///     Use the movement direction vector to apply a force to a body
    /// </summary>
    public class MovementSystem : SystemBase
    {
        protected override void OnUpdate()
        {
```

```
            // Set delta time local variable
            var deltaTime = Time.DeltaTime;

            Entities
                .ForEach((
                    ref Movable movable,
                    ref PhysicsVelocity velocity,
                    ref PhysicsMass mass,
                    ref Translation translation,
                    ref Rotation rotation,
                    ref LocalToWorld localToWorld) =>
                {
                    if (movable.moveDirection.x != 0 ||
                        movable.moveDirection.z != 0)
                    {
                        // Move by applying a force
                         var maxSpeed = 1800f;
                        var targetSpeed = maxSpeed * movable.moveDirection.z;
                        movable.currentSpeed = math.lerp(movable.currentSpeed,
                        targetSpeed, deltaTime);
                        var currentVelocity = localToWorld.Forward *
                        movable.currentSpeed * deltaTime;
                        // Alter linear velocity
                        velocity.Linear = new float3(currentVelocity.x,
                        velocity.Linear.y, currentVelocity.z);

                        // Look where you're going
                         var rotationSpeed = 0f;
                        rotationSpeed = movable.moveDirection.x * 50f *
                        deltaTime;
                        var angular = new float3(0f, rotationSpeed, 0f);
                        var q = rotation.Value;
                        var angle = 2.0f * math.atan(q.value.y / q.value.w);
                        rotation.Value = quaternion.RotateY(angle);
                        velocity.SetAngularVelocityWorldSpace(mass, rotation,
                        angular);
                    }
                }).ScheduleParallel();
            }
        }
    }
}
```

5. Camera Follow

Drag **UpdateCameraFollow.cs** from Tiny Racing Project into **System** folder. The following is the actual code in Tiny Racing.
https://github.com/Unity-Technologies/ProjectTinySamples/blob/master/TinyRacing/Assets/Scripts/TinyRacing/Systems/UpdateCameraFollow.cs

Modify to the following so camera distance and angle fit to our gameplay. Notice they are manipulated by float value on each axis. In order to tilt down the camera, quaternion.Rotate.X was multiplied to the player's rotation and fed to camera rotation using **EntityManager.Set ComponentData**. The angle value is in Radian. (Note: 360 degree = 6.28319 radian)

qauternion.RotateX

```
public static quaternion RotateX(float angle)
```

https://docs.unity3d.com/Packages/com.unity.mathematics@0.0/api/Unity.Mathematics.quaternion.html

UpdateCameraFollow.cs will be modified to this.

```
using Unity.Entities;
using Unity.Mathematics;
using Unity.Tiny.Rendering;
using Unity.Transforms;

namespace TinyPhysics.Systems
{
    /// <summary>
    /// Update camera position and rotation to follow the player.
    /// </summary>
    [UpdateBefore(typeof(TransformSystemGroup))]
    // [UpdateAfter(typeof(MoveCar))]
    public class UpdateCameraFollow : SystemBase
    {
        private float3 DefaultCameraPosition;
        private quaternion DefaultCameraRot;
        private bool IsDefaultCameraPositionSet;

        protected override void OnUpdate()
        {
            // Get player car position and direction
            var playerPosition = float3.zero;
            var playerDirection = float3.zero;
            var playerRotation = quaternion.identity;
            var targetPosition = float3.zero;

            // Set Player's Transform local variables
            Entities
                .ForEach((
                ref Movable movable,
                in Translation translation,
                in LocalToWorld localToWorld,
                in Rotation rotation
                ) =>{
                    playerPosition = translation.Value;
                    playerDirection = localToWorld.Forward;
                    playerRotation = rotation.Value;
                }).Run();

            // Tilt the camera downward
            var tiltedplayerRotation = quaternion.identity;
```

```
            targetPosition = playerPosition + playerDirection * -40f;
            targetPosition.y = 12f;
            tiltedplayerRotation = math.mul(playerRotation,
            quaternion.RotateX(.35f));

            // TODO: Find camera with entity query once there's a pure
             component for cameras
             var cameraEntity = GetSingletonEntity<Camera>();
            var cameraPos = EntityManager
            .GetComponentData<Translation>(cameraEntity).Value;
            var cameraRot = EntityManager
            .GetComponentData<Rotation>(cameraEntity).Value;

            var deltaTime = math.clamp(Time.DeltaTime * 7f, 0, 1);

            cameraPos =
            math.lerp(cameraPos, targetPosition, deltaTime);
            cameraRot =
            math.slerp(cameraRot, tiltedplayerRotation , deltaTime);

            // Set Camera Transform Values for Update Increment
             EntityManager
            .SetComponentData(
            cameraEntity,
            new Translation {Value = cameraPos}
            );

            EntityManager
            .SetComponentData(
            cameraEntity,
            new Rotation {Value = cameraRot}
            );
        }
    }
}
```

6. Environment

Static environment assets should be set to **0:Static Environment** at **Belongs To**(**Collision Filter** Dropdown Menu item). Also set **Collides With** to **Character**. Assign **Shape Type** with simplest geometry for Smooth Gameplay. Mesh should be used on only necessary occasions such as houses with interior space for character to go in.

7.Door

Create **Door.cs** Component with following properties. Then it will be attached to **Door Prefab**.

```
using Unity.Entities;
```

```
namespace TinyPhysics
{
    public enum DoorState
    {
        Stopped,
        Opening,
        Closing
    }

    [GenerateAuthoringComponent]
    public struct Door : IComponentData
    {
        public float speed;
        public float timer;
        public float audioTimer;
        public bool isOpened;

        public DoorState DoorState { get; set; }
    }
}
```

Create **DoorSystem.cs** as follows.

```
using Unity.Entities;
using Unity.Mathematics;
using Unity.Tiny.Audio;
using Unity.Transforms;

namespace TinyPhysics.Systems
{
    /// <summary>
    /// Modify Button System from Tiny Physics
    /// </summary>
    [UpdateAfter(typeof(CollisionSystem))]
    public class DoorCollisionSystem : SystemBase
    {
        protected override void OnUpdate()
        {
            float deltaTime = Time.DeltaTime;

            Entities
            .WithAll<Door, AudioSource>()
            .ForEach((
                ref Entity entity,
                ref Door door,
                ref Collideable collideable,
                ref Rotation rotation
                ) => {

                if (collideable.CollisionEntity != Entity.Null)
                    door.DoorState = DoorState.Opening;
                else door.DoorState = DoorState.Closing;
```

```
                switch(door.DoorState)
                {
                    case DoorState.Opening:

                        if (door.isOpened) return;

                        // Set timer
                        door.timer += deltaTime;
                        // Check if Audio is playing
                        if (!GetComponent<AudioSource>(entity).isPlaying)
                        {
                            // Play Audio
                            EntityManager.
                            AddComponent<AudioSourceStart>(entity);

                            // Set Audio timer
                            door.audioTimer += deltaTime;

                            // Stop Audio
                            if (door.audioTimer > 0.01f)
                            {
                                EntityManager.
                                AddComponent<AudioSourceStop>(entity);
                            }
                        }

                        // Open the door
                        if (door.timer < 1f)
                        {
                            rotation.Value = math.mul(rotation.Value,
                            quaternion.RotateZ
                            (math.radians(door.speed * deltaTime)));
                        }

                        // Stop opening the door
                        else
                        {
                            door.DoorState = DoorState.Stopped;
                            door.isOpened = true;
                        }
                        break;

                    case DoorState.Closing:
                        //Consume Collider
                        collideable.CollisionEntity = Entity.Null;
                        break;
                    }
                }).WithStructuralChanges().Run();
        }
    }
}
```

8. Potion

Potion Prefab has 3 child GameObjects inside. 1. Potion Bottle , 2. Particle Effect, and 3. Audio.

Potion.cs will be attached to Potion Prefab's Child GameObject Bottle..

```
using Unity.Entities;

namespace TinyPhysics
{
    [GenerateAuthoringComponent]
    public struct Potion : IComponentData
    {
        public bool IsTaken;
    }
}
```

AudioPotion.cs will be attached to the Child GameObject Audio.

```
using Unity.Entities;

[GenerateAuthoringComponent]
public struct AudioPotion : IComponentData
{
    public Entity bottle;
    public float timer;
}
```

PotionSystem.cs contains 3 **Entities.ForEach** methods for 3 Child GameObjects.

```
using Unity.Entities;
using Unity.Tiny.Audio;

namespace TinyPhysics.Systems
{
    /// <summary>
    /// Detect the collisions with Player and update potion count
    /// </summary>
    ///
    [UpdateAfter(typeof(CollisionSystem))]
    public class PotionSystem : SystemBase
    {
        protected override void OnUpdate()
        {
            // Job Description for Bottle GameObject
            Entities
                .ForEach((
                    ref Entity entity,
                    ref Potion potion,
                    ref Collideable collideable
                    ) => {
                        if (collideable.CollisionEntity != Entity.Null)
                        {
                            //Update potion status
                            potion.IsTaken = true;
```

```
                        //Disable potion visibility
                        EntityManager.AddComponent<Disabled>(entity);
                        //Consume Collider
                        collideable.CollisionEntity = Entity.Null;
                    }
            }).WithStructuralChanges().Run();

        // Job Description for AudioSource GameObject
        Entities
            .WithAll<AudioPotion, AudioSource>()
            .ForEach((
            ref Entity entity,
            ref AudioPotion audioPotion
            ) => {
                // Store Potion Entity to local variable
                Entity bottle = audioPotion.bottle;
                var isTaken =
                EntityManager
                .GetComponentData<Potion>(bottle).IsTaken;

                if (isTaken)
                {
                    //Play chime
                    if (!GetComponent<AudioSource>(entity).isPlaying)
                    {
                     EntityManager
                     .AddComponent<AudioSourceStart>(entity);
                    }
                    //Set timer
                    audioPotion.timer += Time.DeltaTime;
                    if (audioPotion.timer > 0.3f)
                    {
                        //Disable entity
                        EntityManager.AddComponent<Disabled>(entity);
                    }
                }
            }).WithStructuralChanges().Run();

        // Job Description for Particle GameObject
        Entities
            .ForEach((
                ref Entity entity,
                ref PotionStar potionStar
                ) => {
                    // Store Potion Entity to local variable
                    Entity bottle = potionStar.bottle;
                    var isTaken = EntityManager
                    .GetComponentData<Potion>(bottle).IsTaken;
                    if (isTaken)
                    {
                        //Disable potion visibility
                        EntityManager.AddComponent<Disabled>(entity);
                    }
            }).WithStructuralChanges().Run();
    }
}
```

```
}
```

9. Enemy

In this game, enemies simply follow the player when the player enters a given proximity, and stop when the player leaves the proximity.

Enemy.cs Component

```
using Unity.Entities;

namespace TinyPhysics
{
    [GenerateAuthoringComponent]
    public struct Enemy : IComponentData
    {
        public Entity currentTarget;
        public float Speed;
        public float maxDistanceSqrd;
    }
}
```

We borrowed the code from the official website to make a mechanism to get the enemies to approach the player.
https://docs.unity3d.com/Packages/com.unity.physics@0.3/manual/interacting_with_bodies.html

EnemyMovementSystem.cs

```
using Unity.Entities;
using Unity.Mathematics;
using Unity.Transforms;

namespace TinyPhysics.Systems
{
    [UpdateBefore(typeof(MovementSystem))]
    public class EnemyMovementSystem : SystemBase
    {
        protected override void OnUpdate()
        {
            Entities
                .ForEach((
                ref Entity entity,
                ref Enemy enemy,
                ref Translation translation,
                ref LocalToWorld localToWorld,
                ref Rotation rotation,
                ref Collideable collideable
                ) => {
                    // Store Target Entity to local variable
```

```
                    Entity targetEntity = enemy.currentTarget;
                    var targetTranslation = EntityManager
                    .GetComponentData<Translation>(targetEntity);
                    var targetLocalToWorld = EntityManager
                    .GetComponentData<LocalToWorld>(targetEntity);
                    var targetRotation= EntityManager
                    .GetComponentData<Rotation>(targetEntity);

                    var deltaTime = math.clamp(Time.DeltaTime , 0f, 0.001f);

                    float3 diff = targetTranslation.Value -
                    translation.Value;
                    float distSqrd = math.lengthsq(diff);
                    if (distSqrd < enemy.maxDistanceSqrd)
                    {
                        // Constraint height translation
                        translation.Value.y = 0.1f;
                        // Store a vector to target entity
                        var lookOnlook
                        = targetTranslation.Value - translation.Value;
                        // Clear
                        lookOnlook.y = translation.Value.y;
                        // Calculate rotation amount
                        quaternion rot
                        = quaternion.LookRotation
                        (lookOnlook, targetLocalToWorld.Up);
                        // Move towards target entity incrementally
                        translation.Value = math.lerp(translation.Value,
                        targetTranslation.Value, deltaTime * enemy.Speed);
                        //
                        rotation.Value = math.slerp
                        (rotation.Value, rot, deltaTime * 20f * enemy.Speed);
                    }
                }).WithoutBurst().Run();
        }
    }
}
```

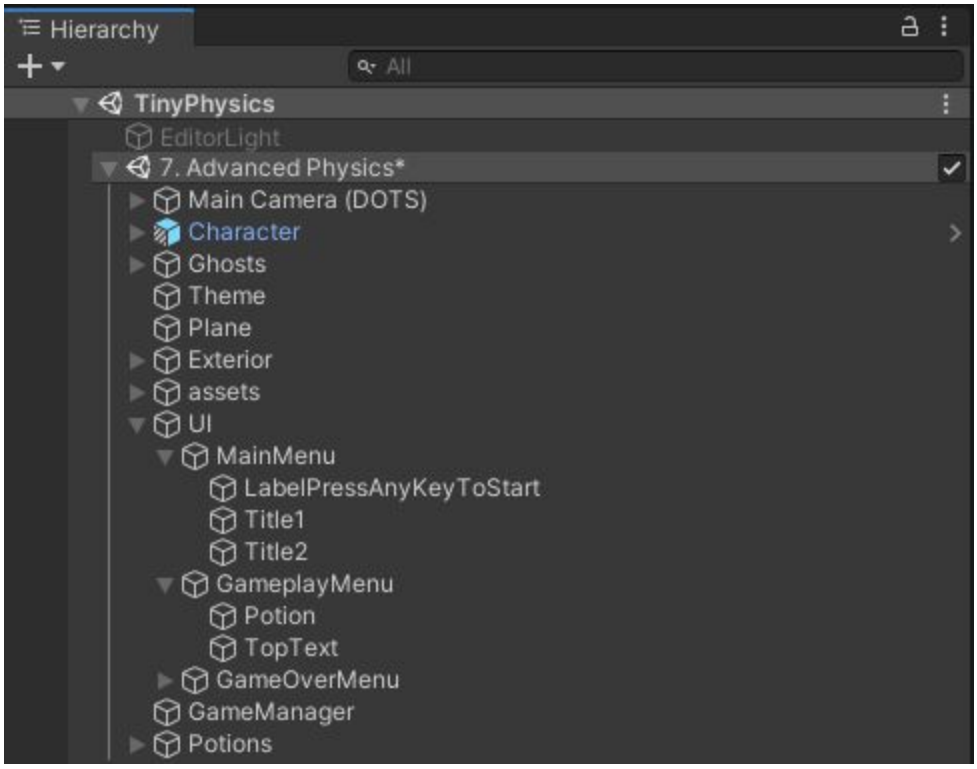
10. Game Manager & UI

GameManager GameObject stores many variables to drive the gameplay and display UI elements. GameManager.cs just needed to be attached to an empty GameOject in the scene in order to run. As it is in Tiny Racing, there is no GameManager System file. Instead, there are system files for the title UI (UpdateMainMenu.cs), for the gameplay UI (UpdateGamePlay.cs), and for the ending UI (UpdateGameOverMenu.cs). Each UI System has its own unique set of Component file and System file, and their visibilities are enabled by boolean values in GameManager.cs (System). GameManager also stores conditions whether the player is inside the building or not, and a potion count that the player has collected. Then you will need to revise

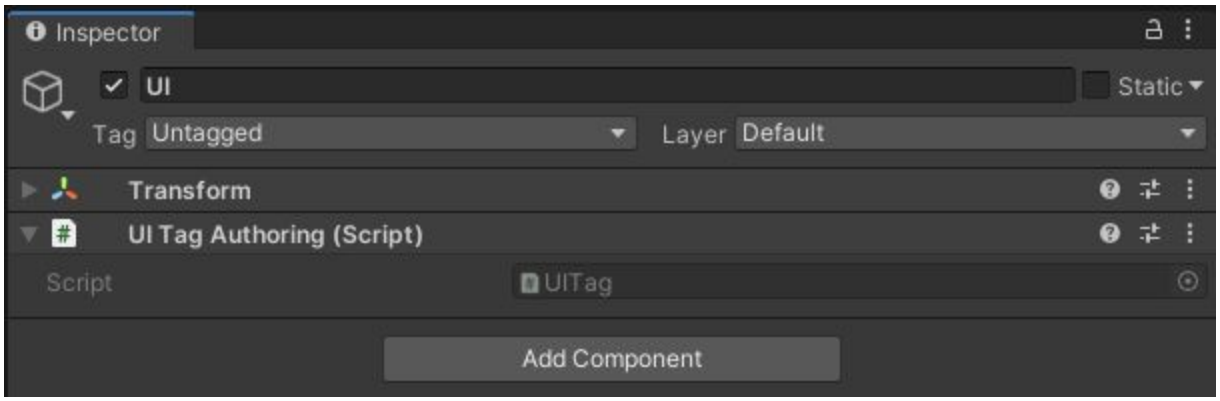
CameraFollow.cs, so the camera is lower and closer to the player while he is inside the buildings.

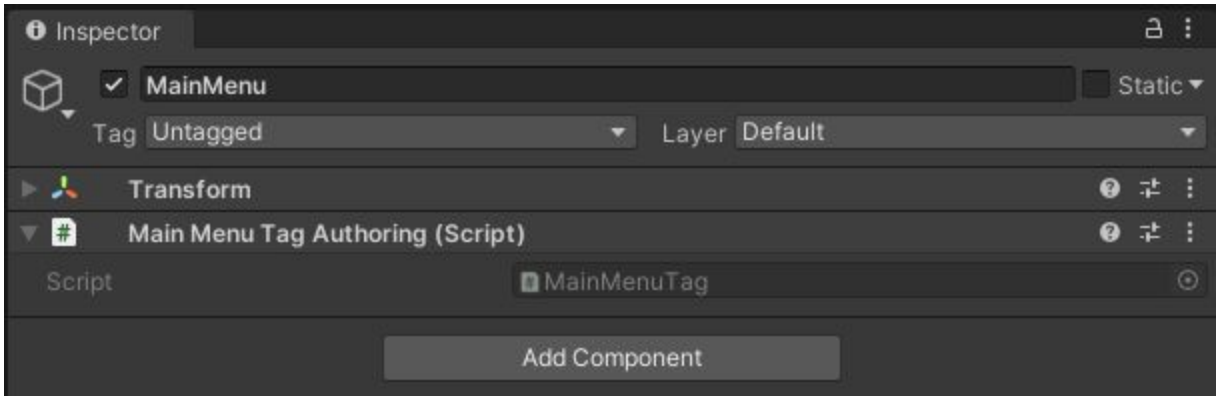
In Unity3D, UI's are set as follows.

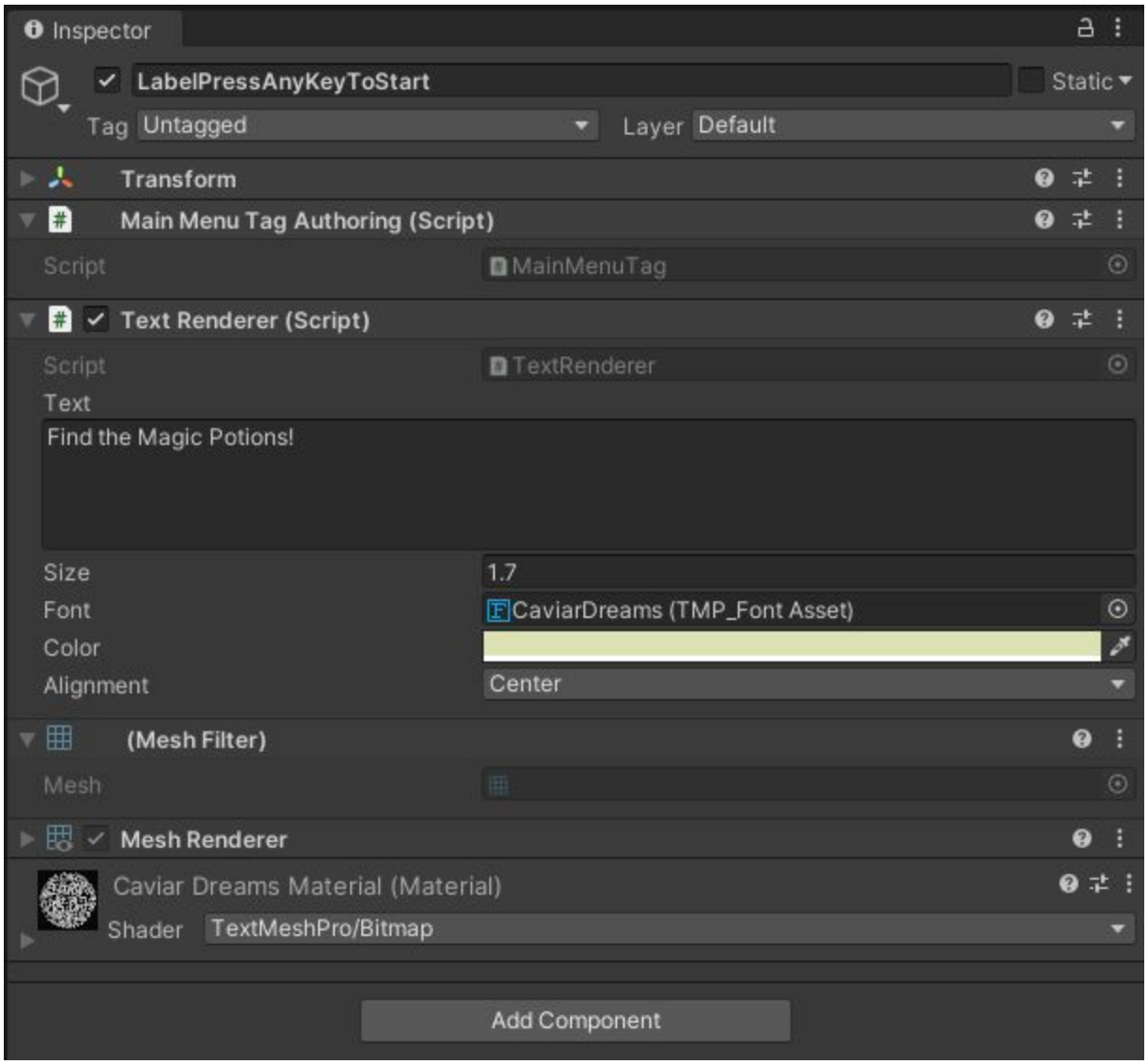
Hierarchy Tab

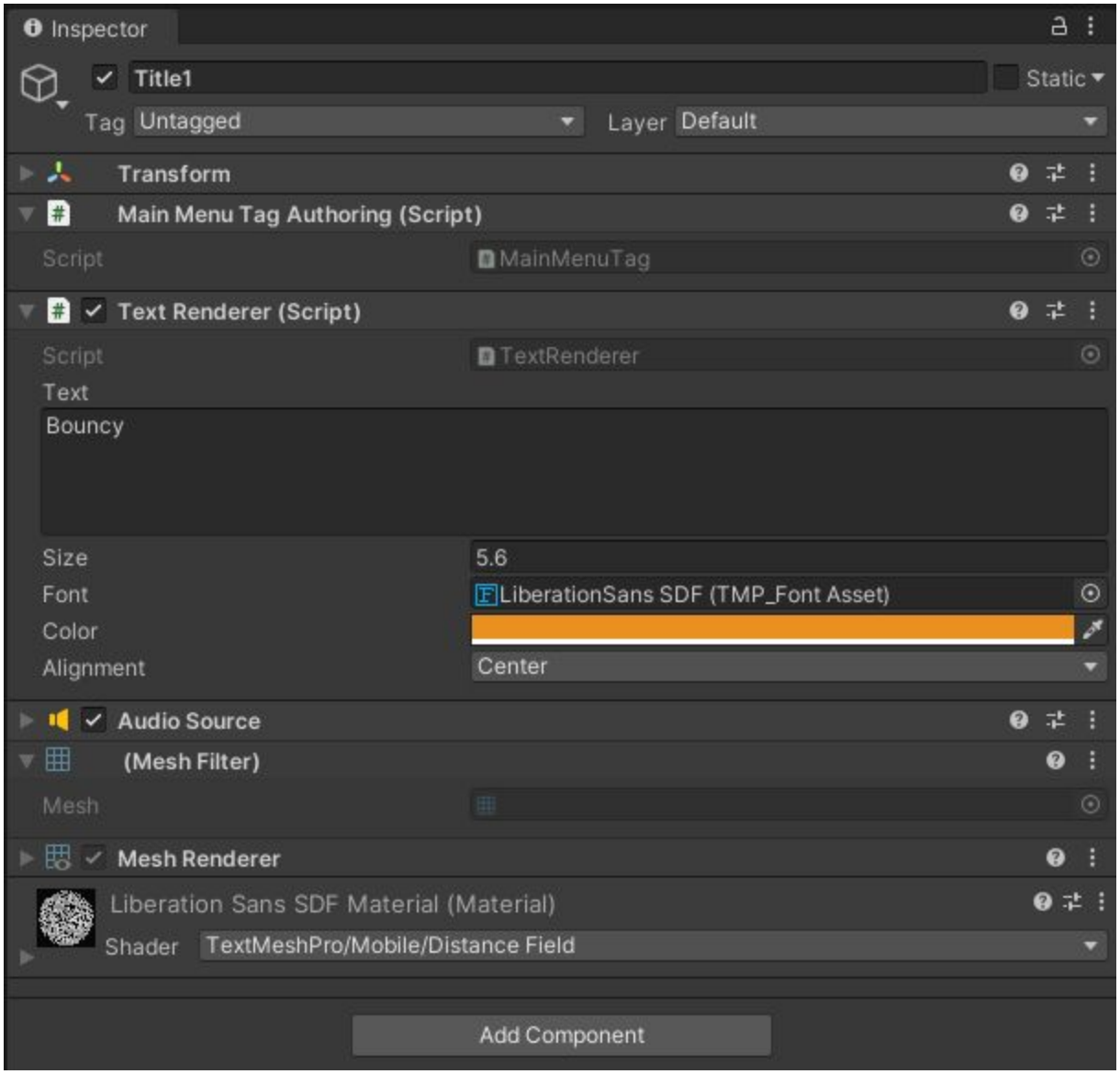


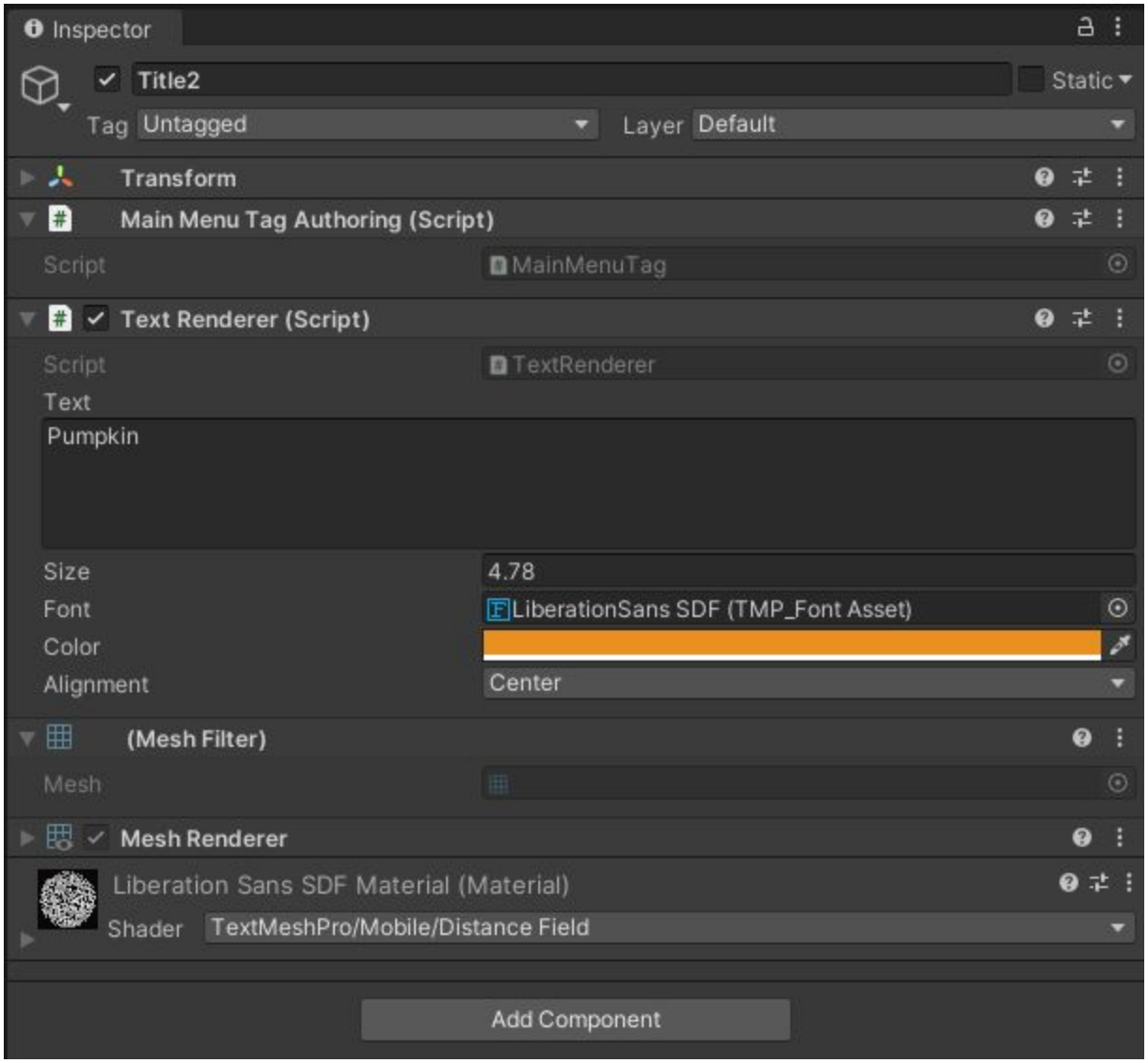
Inspector Tab

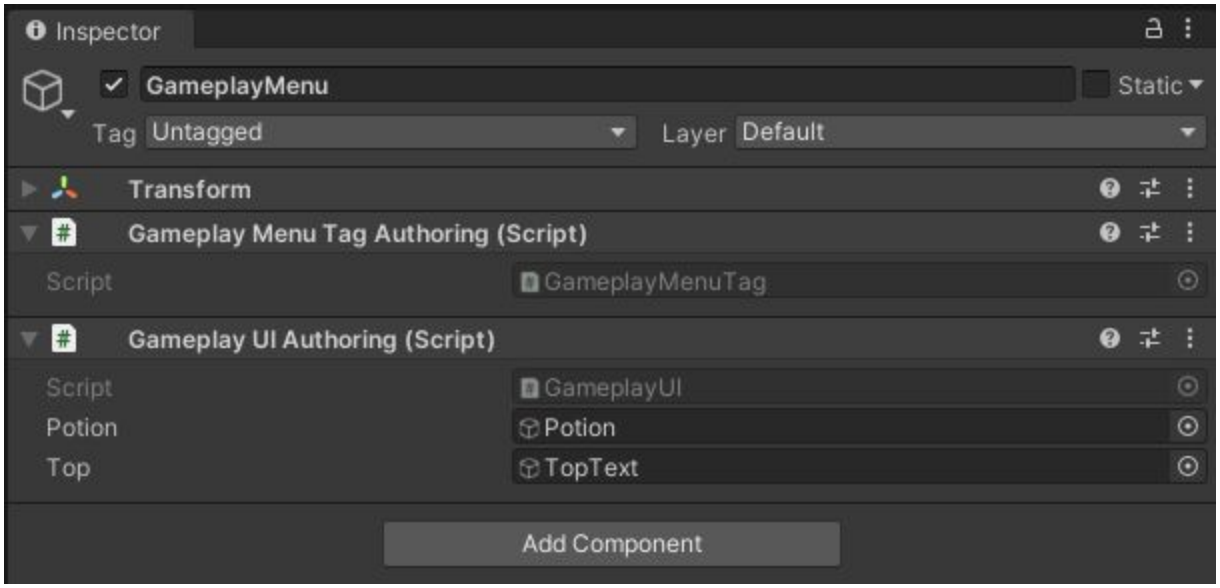


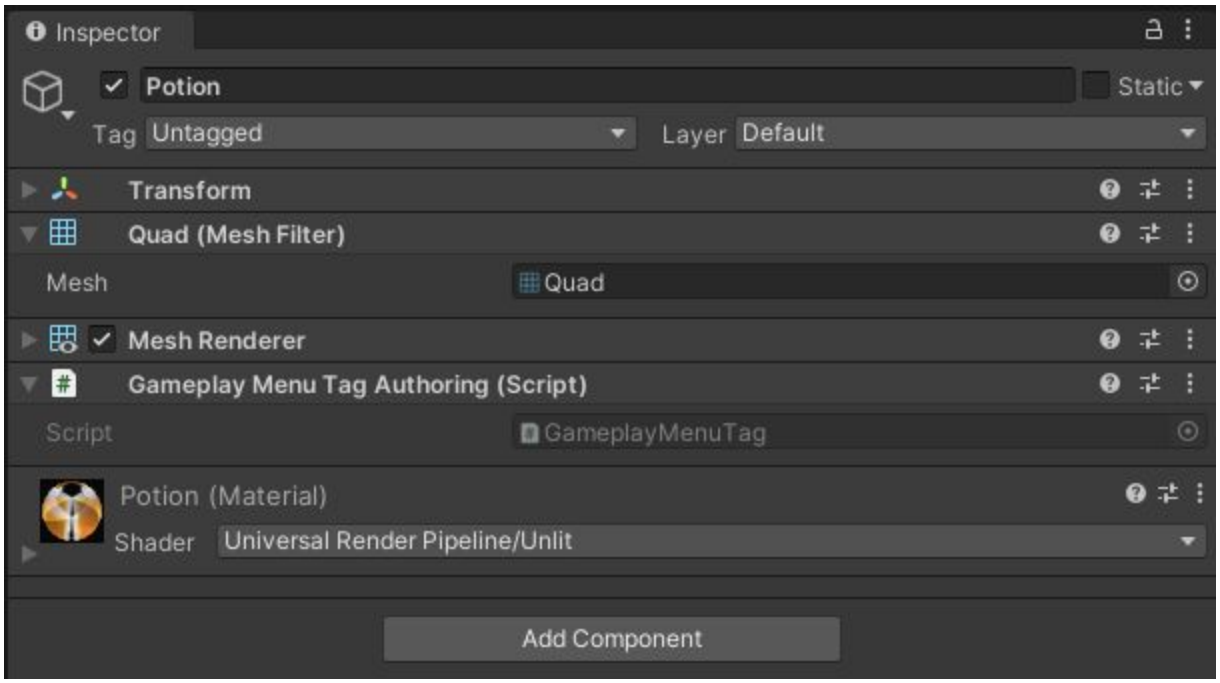

Inspector
MainMenu
Static
Tag
Untagged
Layer
Default
Transform
Main Menu Tag Authoring (Script)
Script
MainMenuTag
Add Component

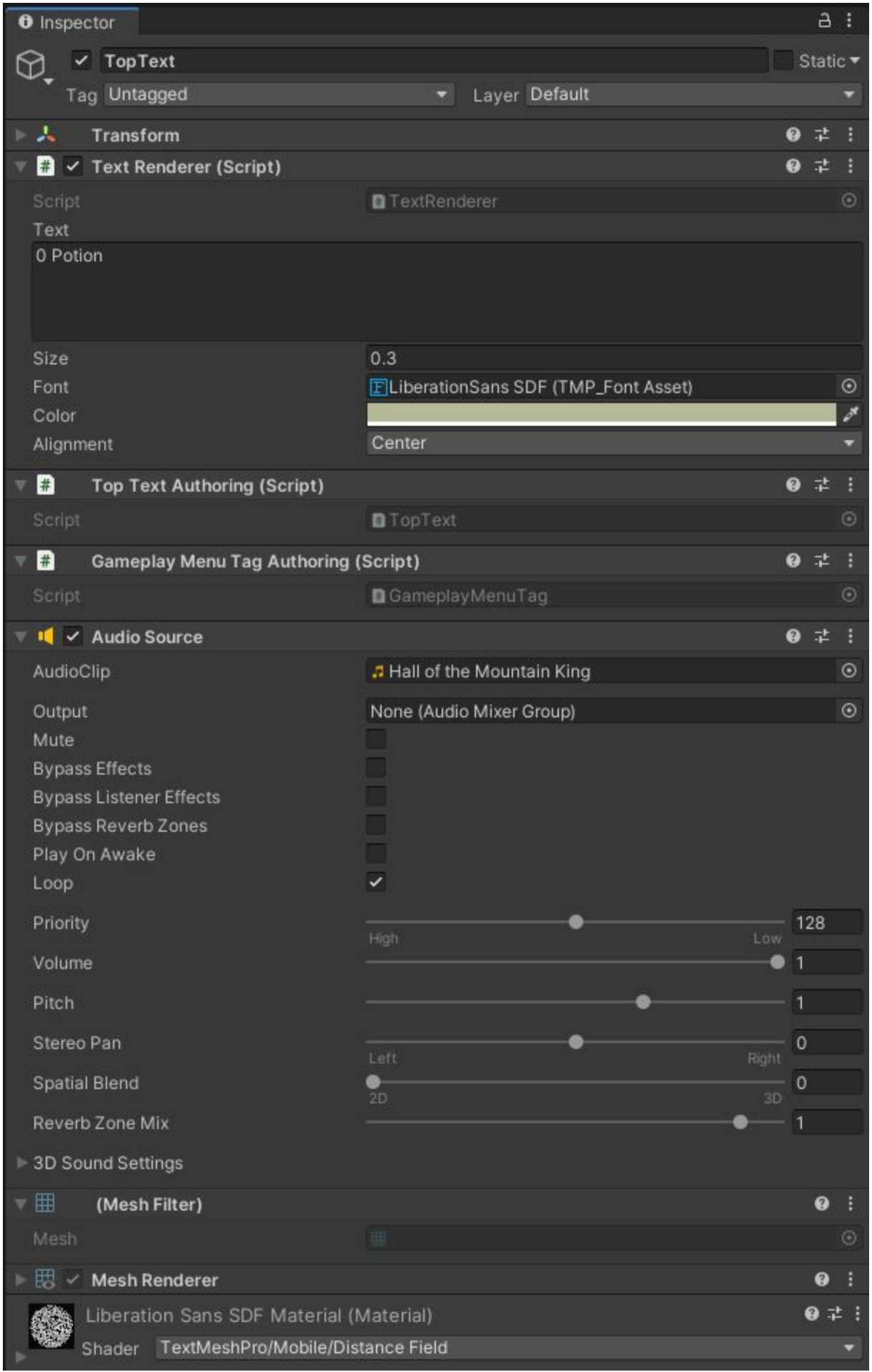

Inspector
LabelPressAnyKeyToStart
Static
Tag
Untagged
Layer
Default
Transform
Main Menu Tag Authoring (Script)
Script
MainMenuTag
Text Renderer (Script)
Script
TextRenderer
Text
Find the Magic Potions!
Size
1.7
Font
CaviarDreams (TMP_Font Asset)
Color
Alignment
Center
(Mesh Filter)
Mesh
Mesh Renderer
Caviar Dreams Material (Material)
Shader
TextMeshPro/Bitmap
Add Component


Inspector
Title1
Static
Tag Untagged
Layer Default
Transform
Main Menu Tag Authoring (Script)
Script MainMenuTag
Text Renderer (Script)
Script TextRenderer
Text
Bouncy
Size 5.6
Font LiberationSans SDF (TMP_Font Asset)
Color
Alignment Center
Audio Source
(Mesh Filter)
Mesh
Mesh Renderer
Liberation Sans SDF Material (Material)
Shader TextMeshPro/Mobile/Distance Field
Add Component


Inspector
Title2
Static
Tag Untagged
Layer Default
Transform
Main Menu Tag Authoring (Script)
Script MainMenuTag
Text Renderer (Script)
Script TextRenderer
Text
Pumpkin
Size 4.78
Font LiberationSans SDF (TMP_Font Asset)
Color
Alignment Center
(Mesh Filter)
Mesh
Mesh Renderer
Liberation Sans SDF Material (Material)
Shader TextMeshPro/Mobile/Distance Field
Add Component


Inspector
GameplayMenu
Static
Tag Untagged
Layer Default
Transform
Gameplay Menu Tag Authoring (Script)
Script GameplayMenuTag
Gameplay UI Authoring (Script)
Script GameplayUI
Potion Potion
Top TopText
Add Component


Inspector
Potion
Static
Tag Untagged
Layer Default
Transform
Quad (Mesh Filter)
Mesh Quad
Mesh Renderer
Gameplay Menu Tag Authoring (Script)
Script GameplayMenuTag
Potion (Material)
Shader Universal Render Pipeline/Unlit
Add Component


Inspector
TopText
Static
Tag Untagged
Layer Default
Transform
Text Renderer (Script)
Script TextRenderer
Text
0 Potion
Size 0.3
Font LiberationSans SDF (TMP_Font Asset)
Color
Alignment Center
Top Text Authoring (Script)
Script TopText
Gameplay Menu Tag Authoring (Script)
Script GameplayMenuTag
Audio Source
AudioClip Hall of the Mountain King
Output None (Audio Mixer Group)
Mute
Bypass Effects
Bypass Listener Effects
Bypass Reverb Zones
Play On Awake
Loop
Priority 128
High
Low
Volume 1
Pitch 1
Stereo Pan 0
Left
Right
Spatial Blend 0
2D
3D
Reverb Zone Mix 1
3D Sound Settings
(Mesh Filter)
Mesh
Mesh Renderer
Liberation Sans SDF Material (Material)
Shader TextMeshPro/Mobile/Distance Field

GameManager.cs

```
using Unity.Entities;

namespace TinyPhysics
{
    [GenerateAuthoringComponent]
    public struct GameManager : IComponentData
    {
        public int PotionCount;
        public bool IsGameStarted;
        public bool IsGameFinished;
        public bool IsInside;
        public float GameOverTimer;
    }
}
```

MainMenuTag.cs

```
using Unity.Entities;

namespace TinyPhysics
{
    [GenerateAuthoringComponent]
    public struct MainMenuTag : IComponentData
    {
    }
}
```

UpdateMainMenu.cs

```
using Unity.Entities;
using Unity.Transforms;
using Unity.Tiny.Input;
using Unity.Tiny.Audio;

namespace TinyPhysics.Systems
{
    /// <summary>
    /// Update the main menu UI
    /// </summary>
    [UpdateAfter(typeof(TransformSystemGroup))]
    public class UpdateMainMenu : SystemBase
    {
        protected override void OnUpdate()
        {
            // Hide main menu when user presses any key
            var Input = World.GetExistingSystem<InputSystem>();
            var startGameButtonPressed =
            Input.GetKeyDown(KeyCode.Space) ||
            Input.GetMouseButtonDown(0) ||
            Input.TouchCount() > 0;
```

```
            var gameManager = GetSingleton<GameManager>();
            if (startGameButtonPressed && !gameManager.IsGameStarted)
            {
                gameManager.IsGameStarted = true;
                SetSingleton(gameManager);
            }

            SetMenuVisibility(!gameManager.IsGameStarted);
        }

        private void SetMenuVisibility(bool isVisible)
        {
            if (isVisible)
            {
                Entities
                  .WithEntityQueryOptions(EntityQueryOptions.IncludeDisabled)
                  .WithAll<MainMenuTag, AudioSource, Disabled>()
                  .ForEach((
                  Entity entity
                  ) => {
                      EntityManager.AddComponent<AudioSourceStart>(entity);
                  }).WithStructuralChanges().Run();

                Entities
                  .WithEntityQueryOptions(EntityQueryOptions.IncludeDisabled)
                  .WithAll<MainMenuTag, Disabled>()
                  .ForEach((Entity entity
                  ) => {
                      EntityManager.RemoveComponent<Disabled>(entity);
                  }).WithStructuralChanges().Run();
            }
            else
            {
                Entities
                    .WithAll<MainMenuTag>()
                    .ForEach((
                    Entity entity
                    ) => {
                        EntityManager.AddComponent<Disabled>(entity);
                    }).WithStructuralChanges().Run();
            }
        }
    }
}
```

GameplayMenu.cs

```
using Unity.Entities;

namespace TinyPhysics
{
    [GenerateAuthoringComponent]
    public struct GameplayMenuTag : IComponentData
    {
    }
```

```
}
```

UpdateGamePlay.cs

```
using Unity.Entities;
using Unity.Tiny.Audio;
using Unity.Transforms;

namespace TinyPhysics.Systems
{
    [UpdateAfter(typeof(TransformSystemGroup))]
    public class UpdateGameplayMenu : SystemBase
    {
        protected override void OnUpdate()
        {
            var gameManager = GetSingleton<GameManager>();

            // Update gameplay menu visibility
            if (!gameManager.IsGameStarted || gameManager.IsGameFinished)
            {
                SetMenuVisibility(false);
                return;
            }

            SetMenuVisibility(true);

            var ui = GetSingleton<GameplayUI>();
        }
        private void SetMenuVisibility(bool isVisible)
        {
            if (isVisible)
            {
                Entities
                  .WithEntityQueryOptions(EntityQueryOptions.IncludeDisabled)
                  .WithAll<GameplayMenuTag, AudioSource, Disabled>()
                  .ForEach((
                  Entity entity
                  ) => {
                  EntityManager.AddComponent<AudioSourceStart>(entity);
                  }).WithStructuralChanges().Run();

                Entities
                  .WithEntityQueryOptions(EntityQueryOptions.IncludeDisabled)
                  .WithAll<GameplayMenuTag, Disabled>()
                  .ForEach((
                  Entity entity
                  ) => {
                  EntityManager.RemoveComponent<Disabled>(entity);
                  }).WithStructuralChanges().Run();
            }
            else
            {
                Entities
                .WithAll<GameplayMenuTag>()
                .ForEach((
```

```
                Entity entity
                ) => {
                EntityManager.AddComponent<Disabled>(entity);
                }).WithStructuralChanges().Run();
            }
        }
    }
}
```

GameOverMenuTag.cs

```
using Unity.Entities;

namespace TinyPhysics
{
    [GenerateAuthoringComponent]
    public struct GameOverMenuTag : IComponentData
    {
        public float timer;
    }
}
```

UpdateGameOverMenu.cs

```
using Unity.Entities;
using Unity.Transforms;
using Unity.Tiny.Audio;

namespace TinyPhysics.Systems
{
    /// <summary>
    ///     Update the ending UI menu
    /// </summary>
    [UpdateAfter(typeof(ResetGame))]
    [UpdateAfter(typeof(TransformSystemGroup))]
    public class UpdateGameOverMenu : SystemBase
    {
        protected override void OnUpdate()
        {
            var gameManager = GetSingleton<GameManager>();
            SetMenuVisibility(gameManager.IsGameFinished);
        }

        private void SetMenuVisibility(bool isVisible)
        {
            if (isVisible)
            {
                Entities
                .WithEntityQueryOptions(EntityQueryOptions.IncludeDisabled)
                .WithAll<GameOverMenuTag, AudioSource, Disabled>()
                .ForEach((
                    Entity entity
                    ) =>{
                    EntityManager.AddComponent<AudioSourceStart>(entity);
```

```
                }).WithStructuralChanges().Run();

                Entities
                .WithEntityQueryOptions(EntityQueryOptions.IncludeDisabled)
                .WithAll<GameOverMenuTag, Disabled>()
                .ForEach((
                    Entity entity
                    ) => {
                    EntityManager.RemoveComponent<Disabled>(entity);
                }).WithStructuralChanges().Run();
            }
            else
            {
                Entities
                .WithAll<GameOverMenuTag>()
                .ForEach((
                    Entity entity
                    ) => {
                    EntityManager.AddComponent<Disabled>(entity);
                }).WithStructuralChanges().Run();
            }
        }
    }
}
```

11. Text UI

In this tutorial, we will focus on one text UI. Any UIs will need an individual tag such as below.

TopText.cs

```
using Unity.Entities;

namespace TinyPhysics
{
    [GenerateAuthoringComponent]
    public struct TopText : IComponentData
    {
    }
}
```

The following code uses a standardized format for UI. Here, GameManager singleton updates the potion count.

TextSystem.cs

```
using Unity.Entities;
using Unity.Tiny.Text;

namespace TinyPhysics.Systems
```

```
{
    public class TextSystem : SystemBase
    {
        private Entity TopEntity;

        protected override void OnCreate()
        {
            RequireSingletonForUpdate<TopText>();
            base.OnCreate();
        }

        protected override void OnStartRunning()
        {
            TopEntity = GetSingletonEntity<TopText>();
            base.OnStartRunning();
        }

        protected override void OnUpdate()
        {
            var gameManager = GetSingleton<GameManager>();

            if (gameManager.PotionCount >= 1)
            {
                TextLayout
                .SetEntityTextRendererString(
                EntityManager,
                TopEntity,
                $"{gameManager.PotionCount} / 13 Potion");
            }
            else TextLayout
                 .SetEntityTextRendererString(
                 EntityManager,
                 TopEntity,
                 "0/13 Potion");
        }
    }
}
```

12. Conclusion

The main purpose of this tutorial is to understand the basic workflow of Unity Tiny development and some of the crucial mechanisms. With this knowledge, one can create pretty interesting games or interactive web contents. As Unity Tiny develops further, there will be more functions and documentations available for users. Then I believe the game dev process will be much faster and intuitive.