

scripting-custom-logic-components

Creating & Using T# Scripts

About T#


The behavior of objects in the game is controlled by the Logic Templates that are attached to them. Although Terra Studio's built-in Logic Templates can be very versatile, you will soon find you need to go beyond what they can provide to implement your own gameplay features. Terra Studio allows you to create your own Logic Templates using **scripts**. These allow you to trigger game events, modify the properties of Assets over time and respond to user input in any way you like.

Terra Studio uses a programming language called T# (pronounced T-Sharp) - it has a syntax similar to C#. This was developed to provide developers who are used to writing code the flexibility to do almost anything on Terra's platforms.

A knowledge of Unity C# makes it super easy for you learn T#, but you need to take care of certain things that are different in T#. Read the section on [Unsupported Functionalities in T#](#) to know more.

Creating your own Script

You can create your own T# scripts to customize game behavior. Here's how:

1. Click the Scripts Tab in the left panel.
2. To add a new script:
 1. Click the  icon.
 2. Enter a name and press Enter.
3. Double-click the file to open it. Your script will compile and be ready.
4. The Visual Studio Code IDE will open all the scripts in the project.
5. Locate your script in the Scripts Directory.
6. You can now edit the default code to achieve your desired behavior

Adding your Script to an Asset

Scripts define how assets behave during gameplay. To enable this, attach a script to an asset. Here's how you can do it:

1. Select the asset to add the script.
2. Drag and drop the Script onto the asset.
3. Enable Advanced Mode. The Inspector Panel appears on the right.
4. Go to the Studio Machine tab in the Inspector panel. The added script will be selected by default.
5. You can change the script using the dropdown list containing all scripts in the project.

Anatomy of a Script File

When you double-click a script in Terra Studio, it will be opened in a text editor. By default, Terra Studio will use VS Code Editor:

The initial contents of the script file will look something like this:

```
using System;
using System.Collections;
using Terra.Studio;
using Terra.Studio.Exposed;
using Terra.Studio.Exposed.Layers;
using UnityEngine;
```

```

public class MyFirstScript : StudioBehaviour
{
    //Use this for initialization
    private void Start()
    {

    }

    // Update is called once per frame
    private void Update()
    {

    }

    // Use this for listening to a broadcast
    public override void OnBroadcasted(string x)
    {

    }
}

```

A script makes its connection with the internal workings of Unity by implementing a class which derives from the built-in class called `StudioBehavior`. You can think of a class as a kind of blueprint for creating a new script type that can be attached to `GameObjects`. Each time you attach a script component to a `GameObject`, it creates a new instance of the object defined by the blueprint. The name of the class is taken from the name you supplied when the file was created. The class name and file name must be the same to enable the script component to be attached to a `GameObject`.

The main things to note, however, are the three functions defined inside the class. The `Update` function is the place to put code that will handle the frame update for the `GameObject`. This might include movement, triggering actions and responding to user input, basically anything that needs to be handled over time during gameplay. To enable the `Update` function to do its work, it is often useful to be able to set up variables, read preferences and make connections with other `GameObjects` before any game action takes place. The `Start` function will be called by Terra Studio before gameplay begins (ie, before the `Update` function is called for the first time) and is an ideal place to do any initialization.

Unlike Unity, Terra Studio uses `StudioBehavior` and not `MonoBehavior`

```

public class MyFirstScript : StudioBehaviour
{
    // Enter functions here
}

```

```

// The following class does not work in Terra Studio because MonoBehavior is used
public class MyFirstScript : MonoBehaviour
{
    // Does not work since MonoBehavior is not supported
}

```

`Start()`

This function gets called at the start of the game object's lifecycle and can be used to perform actions at the start of a game

```
private void Start()
{
    // Enter code to dictate what happens at the start of the game

}
```

Update()

This runs on every frame and can be used to dynamically update interactions

```
private void Update()
{
    // Enter code to tell what should happen on each frame

}
```

OnBroadcasted()

This runs whenever a broadcasted signal is listened to by the object

```
// There is an object in the scene that generates the custom broadcast "signal" is
public override void OnBroadcasted(string x)
{
    // The code below executes doSomething when it listens to the broadcast 'signal'
    if(String.Equals(x,"signal")){
        doSomething();
    }

}
```

Adding Variables to an Asset

When you create a script in the Editor, Terra Studio automatically provides a template script for you to edit, which inherits from the `StudioBehaviour` class. Inheriting from the `StudioBehavior` class means your script can behave like a type of component, and you can attach it to GameObjects like any other component.

When your script inherits from `StudioBehaviour`, you can include properties and values in your script which you can then edit from the Editor Inspector, like you can with any other component.

Currently, in Terra Studio, you have add local variables to the Asset GameObject in the editor interface only. To add a variable,

1. Add a script to the Asset GameObject
2. The Inspector Panel that appears on the right with show you both the script and the object variables tab.
3. Go to the Object Variables tab. There are four types of custom variables you can create:
 1. String
 2. Int
 3. Float
 4. GameObject
4. Click on the + icon next to the variable you want to add. Each variable needs to have a name and a value.

In Terra Studio, variables have local scope and can only be used in the scripts of the attached GameObject where they are defined.

Here is an example snippet code to use a variable declared in the editor

```
// The following script is attached to a GameObject where myName is added as an string Object Variable
// Let's say myName is set to 'Terra'
using System;
using System.Collections;
using Terra.Studio;
using Terra.Studio.Exposed;
using Terra.Studio.Exposed.Layers;
using UnityEngine;

public class MainPlayer : StudioBehaviour
{
    private string myVar; // A strong variable created to access the strong variable created in the
editor
    void Start ()
    {
        myVar = GetStringVariable("myName") // // Accesses the string variable 'myName' created in the
editor
        Debug.Log("I am alive and my name is " + myVar);
        // The console will output "I am alive and my name is Terra"
        // Note that if myName is not added to the GameObject in the editor, this will throw up an error
    }
}
```

Accessing Variables in your script

Variables created in an asset can be referenced in all scripts that are added to that asset. Here is how you access the variables:

First, declare new variables of the same type to access those created in the editor:

```
private float myVariable1;
```

You can access these variables inside any function:

```
private void Start()
{
    myVariable1 = GetFloatVariable("a"); // Access the float variable 'a' created in the editor
    // Add code here
}
```

You can do this for all variable types:

```
public class MyFirstScript : StudioBehaviour
{

    //Declare new variables to access the variables you have created
    private float myVariable1; // To access the float variable
    private string myVariable2; // To access the String Variable
    private int myVariable3; // To access the int variable
    private GameObject myVariable4; // To access the game object variable

    private void Start()
    {
        myVariable1 = GetFloatVariable("a"); // Access the float variable 'a' created in the editor
    }
}
```

```

    myVariable2 = GetStringVariable("b"); // Access the string variable 'b'
    myVariable3 = GetIntVariable("c"); // Access the integer variable 'c'
    myVariable4 = GetGameObjectVariable("d"); // Access the game object variable 'd'
    // Add code here
}

private void Update()
{
    // Add code here
}

public override void OnBroadcasted(string x)
{
    // Add code here
}
}

```

Generating a Custom Broadcast

You can create your own custom broadcast inside any function

```

public class MyFirstScript : StudioBehaviour
{
    private void Start()
    {
        Broadcast("signal"); // Generates the broadcast "signal"
    }

    private void Update()
    {
        // Add code
    }

    public override void OnBroadcasted(string x)
    {
        // Add code
    }
}

```

Event Functions

A script in Terra Studio is not like the traditional idea of a program where the code runs continuously in a loop until it completes its task. Instead, Terra Studio passes control to a script intermittently by calling certain functions that are declared within it. Once a function has finished executing, control is passed back to Terra Studio. These functions are known as event functions since they are activated by Terra Studio in response to events that occur during gameplay.

Terra Studio uses a naming scheme to identify which function to call for a particular event. For example, you will already have seen the Update function (called before a frame update occurs) and the Start function (called just before the object's first frame update). The following are some of the most common and important events:

Regular Update Events

A game is rather like an animation where the animation frames are generated on the fly. A key concept in games programming is that of making changes to position, state and behavior of objects in the game just before each frame is rendered. The `Update` function is the main place for this kind of code in Terra Studio. `Update` is called before the frame is rendered and also before animations are calculated.

```
void Update() {
    float distance = speed * Time.deltaTime * Input.GetAxis("Horizontal");
    transform.Translate(Vector3.right * distance);
}
```

The **physics engine** also updates in discrete time steps in a similar way to the frame rendering. A separate event function called `FixedUpdate` is called just before each physics update. Since the physics updates and frame updates do not occur with the same frequency, you will get more accurate results from physics code if you place it in the `FixedUpdate` function rather than `Update`.

```
void FixedUpdate() {
    Vector3 force = transform.forward * driveForce * Input.GetAxis("Vertical");
    rigidbody.AddForce(force);
}
```

It is also useful sometimes to be able to make additional changes at a point after the `Update` and `FixedUpdate` functions have been called for all objects in the **scene** and after all animations have been calculated. An example is where a **camera** should remain trained on a target object; the adjustment to the camera's orientation must be made after the target object has moved. Another example is where the script code should override the effect of an animation (say, to make the character's head look towards a target object in the scene). The `LateUpdate` function can be used for these kinds of situations.

```
void LateUpdate() {
    Camera.main.transform.LookAt(target.transform);
}
```

\

Coroutines

A coroutine allows you to spread tasks across several frames. In Terra Studio, a coroutine is a method that can pause execution and return control to Terra Studio but then continue where it left off on the following frame.

In most situations, when you call a method, it runs to completion and then returns control to the calling method, plus any optional return values. This means that any action that takes place within a method must happen within a single frame update.

In situations where you would like to use a method call to contain a procedural animation or a sequence of events over time, you can use a coroutine.

However, it's important to remember that coroutines aren't threads. Synchronous operations that run within a coroutine still execute on the main thread. If you want to reduce the amount of CPU time spent on the main thread, it's just as important to avoid blocking operations in coroutines as in any other script code.

It's best to use coroutines if you need to deal with long asynchronous operations.

Coroutine example

As an example, consider the task of gradually reducing an object's alpha (opacity) value until it becomes invisible:

```

void Fade()
{
    Color c = renderer.material.color;
    for (float alpha = 1f; alpha >= 0; alpha -= 0.1f)
    {
        c.a = alpha;
        renderer.material.color = c;
    }
}

```

In this example, the Fade method doesn't have the effect you might expect. To make the fading visible, you must reduce the alpha of the fade over a sequence of frames to display the intermediate values that Terra Studio renders. However, this example method executes in its entirety within a single frame update. The intermediate values are never displayed, and the object disappears instantly.

To work around this situation, you could add code to the `Update` function that executes the fade on a frame-by-frame basis. However, it can be more convenient to use a coroutine for this kind of task.

In T#, you declare a coroutine like this:

```

IEnumerator Fade()
{
    Color c = renderer.material.color;
    for (float alpha = 1f; alpha >= 0; alpha -= 0.1f)
    {
        c.a = alpha;
        renderer.material.color = c;
        yield return null;
    }
}

```

A coroutine is a method that you declare with an [IEnumerator](#) return type and with a [yield](#) return statement included somewhere in the body. The `yield return null` line is the point where execution pauses and resumes in the following frame. To set a coroutine running, you need to use the [StartCoroutine](#) function:

```

void Update()
{
    if (Input.GetKeyDown("f"))
    {
        StartCoroutine(Fade());
    }
}

```

The loop counter in the Fade function maintains its correct value over the lifetime of the coroutine, and any variable or parameter is preserved between `yield` statements.

Coroutine time delay

By default, Terra Studio resumes a coroutine on the frame after a `yield` statement. If you want to introduce a time delay, use [WaitForSeconds](#):

```

IEnumerator Fade()
{
    Color c = renderer.material.color;

```

```

for (float alpha = 1f; alpha >= 0; alpha -= 0.1f)
{
    c.a = alpha;
    renderer.material.color = c;
    yield return new WaitForSeconds(.1f);
}
}

```

You can use `WaitForSeconds` to spread an effect over a period of time, and you can use it as an alternative to including the tasks in the `Update` method. Terra Studio calls the `Update` method several times per second, so if you don't need a task to be repeated quite so often, you can put it in a coroutine to get a regular update but not every single frame.

For example, you might have an alarm in your application that warns the player if an enemy is nearby with the following code:

```

bool ProximityCheck()
{
    for (int i = 0; i < enemies.Length; i++)
    {
        if (Vector3.Distance(transform.position, enemies[i].transform.position) < dangerDistance) {
            return true;
        }
    }

    return false;
}

```

If there are a lot of enemies then calling this function every frame might introduce a significant overhead. However, you could use a coroutine to call it every tenth of a second:

```

IEnumerator DoCheck()
{
    for(;;)
    {
        if (ProximityCheck())
        {
            // Perform some action here
        }
        yield return new WaitForSeconds(.1f);
    }
}

```

This reduces the number of checks that Terra Studio carries out without any noticeable effect on gameplay.

Handling MouseClick

The code below sends a debug message when a user clicks on the object to which this script is attached.

```

public class ClickHandler : StudioBehaviour
{
    // This method is called when the user clicks on the GameObject
    private void OnMouseDown()
    {
        // Initiate an action, e.g., print a message to the console
    }
}

```



```

        Debug.Log("GameObject clicked!");
        // You can add a function to do something on click as well - like the dummy function below
        // Feel free to replace it with something functional
        doSomething();

    }
}

```

Handling Collision

The code below sends a debug message when the gameObject to which the script is attached collides with another game object - a cube (whose name is declared in the object variable for the current game object as " cube ")

```

public class CollisionHandler : StudioBehaviour
{
    // Reference to the specific GameObject you want to check for collision
    private GameObject targetObject;

    // Accessing the game object variable on Game start
    private void Start()
    {
        // Assuming that the variable name for the target object was set as 'cube' on the editor
        targetObject = GetGameObjectVariable("cube");
    }

    // This method is called when the collider attached to this GameObject
    // collides with another collider.
    private void OnCollisionEnter(Collision collision)
    {
        // Check if the collision was with the specific GameObject
        if (collision.gameObject == targetObject)
        {
            // Initiate an action, e.g., print a message to the console
            Debug.Log("Collided with the specific GameObject!");

            // Enter code below to complete some other action on collision as well
        }
    }
}

```

description: This page lists everything that is currently not supported in T#

Unsupported Functionalities in T#

Default Variable Assigning

The following way of assigning default values is not supported in T#:

```
private string var1 = "hello world"; // Not allowed in T#
```

Here's what you need to do instead:

```
private string var1;

void Start() {
    var1 = "hello world"; // Allowed since you are assigning the value for the variable within a method
}
```

Generic Methods

Generic methods such as `GetComponent<T>`, `AddComponent<T>`, `FindObjectOfType<T>`, and similar methods involving generics are not supported in T#.

None of the following generic methods are supported:

```
Rigidbody rb = gameObject.GetComponent<Rigidbody>();           // Not allowed
Rigidbody[] rbs = gameObject.GetComponents<Rigidbody>();       // Not allowed
gameObject.AddComponent<Rigidbody>();                         // Not allowed
Rigidbody rb = FindObjectOfType<Rigidbody>();                 // Not allowed
Rigidbody[] rbs = FindObjectsOfType<Rigidbody>();             // Not allowed
Rigidbody rb = GetComponentInParent<Rigidbody>();             // Not allowed
Rigidbody[] rbs = GetComponentsInParent<Rigidbody>();         // Not allowed
Rigidbody rb = GetComponentInChildren<Rigidbody>();           // Not allowed
Rigidbody[] rbs = GetComponentsInChildren<Rigidbody>();       // Not allowed
```

Alternative: For most of these, non-generic alternatives should be used:

```
// Example of how to use GetComponent without generics
Rigidbody rb = (Rigidbody)gameObject.GetComponent(typeof(Rigidbody)) as Rigidbody;
```

Iteration using foreach

The following foreach loop will throw up an error as it is not Supported:

```
//This is not allowed in T#
foreach (Transform child in transform) {
    // Do something
}
```

Use a for loop instead as an alternative:

```
// This is the right way to implement what you were trying to implement above
for (int i = 0; i < transform.childCount; i++) {
    Transform child = transform.GetChild(i);
    // Do something with child
}
```

Unboxing

The following code is not supported:

```
object boxedVal = (int)3;
int failingUnboxedVal = (int)boxedVal; // Not allowed
```

Instead, you can use this alternative:

```
object boxedVal = (int)3;
int workingUnboxedVal = Convert.ToInt32(boxedVal); // Use Convert instead
```

GetComponent Instantly After Instantiate

You cannot use `GetComponent` immediately after instantiating

```
GameObject obj = Instantiate(prefab);
Rigidbody rb = obj.GetComponent(typeof(Rigidbody)); // This might not work immediately after Instantiate
```

Due to the current structure of T#, calling `GetComponent` right after `Instantiate` may not work as expected because the mono behavior might not be fully initialized yet.

SerializedFields

The following code is not supported because it uses

```
[SerializeField] private GameObject obj; // Not allowed
```

Instead, use this alternative:

```
// Use ObjectVariables component instead
ObjectVariables objVars = gameObject.GetComponent<ObjectVariables>();
objVars.SetObjectVariable("obj", someGameObject);

// Access it later
GameObject obj = objVars.GetObjectVariable("obj");
```

Coroutines in Start

This way of starting Coroutines is not supported:

```
IEnumerator Start() {
    yield return new WaitForSeconds(1);
    // Not allowed
}
```

Instead, use this alternative:

```
void Start() {
    StartCoroutine(MyCoroutine());
}

IEnumerator MyCoroutine() {
    yield return new WaitForSeconds(1);
}
```

```
// Your code here  
}
```

Dictionary

Use of Dictionary is not supported. The following way is not

```
Dictionary<string, int> myDict = new Dictionary<string, int>(); // Not allowed in T#
```

Use `TerraDictionary` as a substitute for dictionaries in T#. It provides similar functionality but is compatible with the interpreter.

```
// This is compatible with T#  
TerraDictionary<string, int> myDict = new TerraDictionary<string, int>();  
myDict.Add("key1", 1);  
int value = myDict["key1"];
```

Lists

Only certain list types, including `List<int>`, `List<string>`, `List<bool>`, `List<char>`, `List<GameObject>`, `List<Transform>`, `List<Vector3>`, `List<Vector2>`, `List<double>`, and `List<System.Single>`, are supported natively in T#. Use these directly without any issues.

For example, all of these work:

```
List<int> intList = new List<int>();  
intList.Add(5);  
int firstInt = intList[0];  
  
List<GameObject> gameObjectList = new List<GameObject>();  
gameObjectList.Add(someGameObject);  
GameObject firstObject = gameObjectList[0];
```

The following code, for instance is not supported because it is not natively supported in T#.

```
List<float> myList = new List<float>(); // Not allowed in T#  
myList.Add(1.5f);  
float firstItem = myList[0];
```

For unsupported list types such as the above, use `TerraList` instead of `List<T>`.

```
TerraList<float> myList = new TerraList<float>();  
myList.Add(1.5f);  
float firstItem = myList[0];
```

Miscellaneous

Async isn't supported

```
async Task MyAsyncMethod() {  
    await SomeTask(); // Not supported  
}
```

Task isn't supported

```
Task.Run(() => { /* Some code */ }); // Not supported
```

GetType() or typeof() isn't supported

```
Type myType = GetType(); // Not supported
```

description: >- List of the properties, events and methods wrappers available for each logic

template

T# Logic Component Template Wrappers

How to customize pre-built logic components

Terra Studio provides wrappers to access and customize pre-built logic components.

To access the wrapper, we first declare a variable that helps retrieve the logic template component using the `GetTemplate` method.

```
//This line retrieves a template of type CollectableTemplate using the GetTemplate method and then casts it to the CollectableTemplate type.  
TemplateName template = (GetTemplate(typeof(TemplateName)) as TemplateName);
```

This variable allows you to access and modify the template's properties. We then use the variable to update the properties of the template:

```
//Replace PropertyName with the property you want to change and newValue with the value you want to set.  
template.PropertyName = newValue;
```

Specific Example

The code below illustrates how to do this using the `CollectableTemplate` Wrapper, where we set the `Score` property of the `collectable` is set to `10`.

```
// Example to access the Collectable template wrapper and then set the values of one of its exposed properties  
CollectableTemplate collect = (GetTemplate(typeof(CollectableTemplate)) as CollectableTemplate);  
collect.Score = 10; // Set the score property to 10
```

List of Template Wrappers

AdvanceMoveTemplate

This template manages advanced movement for `GameObjects`, handling speed and event management for ping-pong actions. It is designed to be integrated within a `GameObject`'s movement logic. Below is how you can utilize the `AdvanceMoveTemplate` within a `StudioBehavior` class.

Properties and Methods in AdvanceMoveTemplate

Type	Name	Description
------	------	-------------

Property	Speed	Used for getting and setting the movement speed.
----------	-------	--

Method	Bla	Manages the Ping-Pong event. Allows subscription or unsubscription.
--------	-----	---

Usage Example for AdvanceMoveTemplate

```
public class MovementManager : StudioBehavior
{
    void ManageMovementAndEvents()
    {
        // Accessing the wrapper
        AdvanceMoveTemplate template = (GetTemplate(typeof(AdvanceMoveTemplate)) as AdvanceMoveTemplate);

        // Managing the Ping-Pong event
        template.Bla += OnPingPongAction; // Subscribe to the event
        template.Bla -= OnPingPongAction; // Unsubscribe from the event

        // Accessing and setting the Speed property
        float speed = template.Speed; // Getting the speed
        template.Speed = 5.0f;        // Setting the speed
    }
}
```

ChangeMagnetTemplate

This template manages the magnet range for GameObjects using the `ChangeMagnetComponent`, allowing customization of magnet range behavior within your game.

Properties and Methods in ChangeMagnetTemplate

Type	Name	Description
------	------	-------------

Property	MagnetRange	Used for getting and setting the magnet range.
----------	-------------	--

Usage Example for ChangeMagnetTemplate

```
public class MagnetManager : StudioBehavior
{
    void ManageMagnetRange()
    {
        // Accessing the wrapper
        ChangeMagnetTemplate template = (GetTemplate(typeof(ChangeMagnetTemplate)) as ChangeMagnetTemplate);

        // Accessing and setting the MagnetRange property
        float magnetRange = template.MagnetRange; // Getting the magnet range
        template.MagnetRange = 10.0f;             // Setting the magnet range
    }
}
```

ChangeMaterialPropertyTemplate

This template provides functionality to change material properties on a GameObject. It can be accessed and utilized as follows:

Properties and Methods in ChangeMaterialPropertyTemplate

Type	Name	Description
Property	Channels	Access the changed material channels array.
Property	Length	Get the number of channels.
Property	Indexer	Get or set a channel by index.
Event	OnMaterialChanged	Event triggered when material properties change.
Method	CheckIndex	Ensure the index is within the valid range of channels.
Method	GetEnumerator	Get an enumerator for iterating through the channels.

Usage Example for ChangeMaterialPropertyTemplate

```
public class MaterialManager : StudioBehavior
{
    void ManageMaterialProperties()
    {
        // Accessing the wrapper
        ChangeMaterialPropertyTemplate template = (GetTemplate(typeof(ChangeMaterialPropertyTemplate))
as ChangeMaterialPropertyTemplate);

        // Accessing and setting the Channels property
        ChangedMaterialChannel[] channels = template.Channels; // Gets the changed material channels
array

        // Accessing the Length property
        int length = template.Length; // Gets the number of channels

        // Getting a Channel by Index
        ChangedMaterialChannel channel = template[0]; // Gets the first channel

        // Setting a Channel by Index
        template[0] = new ChangedMaterialChannel(); // Sets the first channel

        // Subscribing to the OnMaterialChanged event
        template.OnMaterialChanged += HandleMaterialChanged;

        // Ensure the index is within the valid range
        template.CheckIndex(0);

        // Iterating through channels
        IEnumerator enumerator = template.GetEnumerator();
        while (enumerator.MoveNext())
        {
            ChangedMaterialChannel currentChannel = (ChangedMaterialChannel)enumerator.Current;
            // Process the channel
        }
    }
}
```

ChangePlayerSpeedTemplate

This template provides functionality to change the player speed on a GameObject. It can be accessed and utilized as follows:

Properties and Methods in ChangePlayerSpeedTemplate

Type	Name	Description
Property	Modifier	Get or set the speed modifier (increase or decrease).
Property	PlayerSpeed	Get or set the player speed.
Event	OnPlayerSpeedChanged	Event triggered when the player speed changes.

Usage Example for ChangePlayerSpeedTemplate

```
public class SpeedManager : StudioBehavior
{
    void ManagePlayerSpeed()
    {
        // Accessing the wrapper
        ChangePlayerSpeedTemplate template = (GetTemplate(typeof(ChangePlayerSpeedTemplate)) as ChangePlayerSpeedTemplate);

        // Accessing and setting the Modifier property
        ChangePlayerSpeed.Modifier modifier = template.Modifier; // Gets the modifier
        template.Modifier = ChangePlayerSpeed.Modifier.Increase; // Sets the modifier

        // Accessing and setting the PlayerSpeed property
        float playerSpeed = template.PlayerSpeed; // Gets the player speed
        template.PlayerSpeed = 10.0f; // Sets the player speed

        // Subscribing to the OnPlayerSpeedChanged event
        template.OnPlayerSpeedChanged += HandlePlayerSpeedChanged;

        // Unsubscribing from the OnPlayerSpeedChanged event
        template.OnPlayerSpeedChanged -= HandlePlayerSpeedChanged;
    }

    void HandlePlayerSpeedChanged(float newSpeed)
    {
        // Handle the player speed change
    }
}
```

CheckpointTemplate

This template provides functionality to manage checkpoint events in the game. Access it as follows:

Properties and Methods in CheckpointTemplate

Type	Name	Description
Event	OnCheckpointTouched	Event triggered when a checkpoint is touched.

Usage Example for CheckpointTemplate

```
public class CheckpointManager : StudioBehavior
{
    void ManageCheckpointEvents()
    {
        // Accessing the wrapper
        CheckpointTemplate template = (GetTemplate(typeof(CheckpointTemplate)) as CheckpointTemplate);

        // Subscribing to the OnCheckpointTouched event
        template.OnCheckpointTouched += MyMethod; // Subscribe MyMethod to the event

        // Unsubscribing from the OnCheckpointTouched event
        template.OnCheckpointTouched -= MyMethod; // Unsubscribe MyMethod from the event
    }

    void MyMethod()
    {
        // Handle the checkpoint touch event
    }
}
```

CollectableTemplate

This template provides functionality to manage collectable items in the game. Access it as follows:

Properties and Methods in CollectableTemplate

Type	Name	Description
Property	ScoreGroup	Get or set the score group associated with the collectable item.
Property	Score	Get or set the score value of the collectable item.
Event	OnCollected	Event triggered when the collectable item is collected.

Usage Example for CollectableTemplate

```
public class CollectableManager : StudioBehavior
{
    void ManageCollectableItem()
    {
        // Accessing the wrapper
        CollectableTemplate template = (GetTemplate(typeof(CollectableTemplate)) as CollectableTemplate);

        // Accessing and setting the ScoreGroup property
        string scoreGroup = template.ScoreGroup; // Getting the score group
        template.ScoreGroup = "NewScoreGroup"; // Setting the score group

        // Accessing and setting the Score property
        int score = template.Score; // Getting the score value
        template.Score = 100; // Setting the score value

        // Subscribing to the OnCollected event
        template.OnCollected += MyMethod; // Subscribe MyMethod to the event
    }
}
```

```

        // Unsubscribing from the OnCollected event
        template.OnCollected -= MyMethod; // Unsubscribe MyMethod from the event
    }

    void MyMethod()
    {
        // Handle the collectable item collection
    }
}

```

DelayBroadcastTemplate

This template manages delayed broadcast functionality in the game. Access it as follows:

Properties and Methods in DelayBroadcastTemplate

Type	Name	Description
------	------	-------------

Property	Delay	Get or set the delay time for the broadcast.
----------	-------	--

Usage Example for DelayBroadcastTemplate

```

public class BroadcastManager : StudioBehavior
{
    void ManageDelayBroadcast()
    {
        // Accessing the wrapper
        DelayBroadcastTemplate template = (GetTemplate(typeof(DelayBroadcastTemplate)) as DelayBroadcastTemplate);

        // Accessing and setting the Delay property
        float delayTime = template.Delay; // Getting the delay time
        template.Delay = 2.5f;             // Setting the delay time to 2.5 seconds
    }
}

```

DestroyOnTemplate

This template manages the behavior of objects that are destroyed after a specified delay. Access it as follows:

Properties and Methods in DestroyOnTemplate

Type	Name	Description
------	------	-------------

Property	DestroyAfter	Get or set the delay after which the object is destroyed.
----------	--------------	---

Usage Example for DestroyOnTemplate

```

public class DestructionManager : StudioBehavior
{
    void ManageDestructionDelay()
    {
        // Accessing the wrapper
        DestroyOnTemplate template = (GetTemplate(typeof(DestroyOnTemplate)) as DestroyOnTemplate);
    }
}

```

```

        // Accessing and setting the DestroyAfter property
        float destroyAfter = template.DestroyAfter; // Getting the destruction delay
        template.DestroyAfter = 5.0f;               // Setting the destruction delay to 5 seconds
    }
}

```

DestroyTemplate

This template manages the behavior of objects that are destroyed after a specified delay. Access it as follows:

Properties and Methods in DestroyTemplate

Type	Name	Description
------	------	-------------

Property	DestroyAfter	Get or set the delay after which the object is destroyed.
----------	--------------	---

Usage Example for DestroyTemplate

```

public class DestructionManager : StudioBehavior
{
    void ManageDestructionDelay()
    {
        // Accessing the wrapper
        DestroyTemplate template = (GetTemplate(typeof(DestroyTemplate)) as DestroyTemplate);

        // Accessing and setting the DestroyAfter property
        float destroyAfter = template.DestroyAfter; // Getting the destruction delay
        template.DestroyAfter = 5.0f;               // Setting the destruction delay to 5 seconds
    }
}

```

GameProgressTemplate

This template manages game progress functionality in the game. Access it as follows:

Properties and Methods in GameProgressTemplate

Type	Name	Description
------	------	-------------

Property	CurrentProgress	Get or set the current progress value.
----------	-----------------	--

Property	LerpSpeed	Get or set the lerp speed configured for the progress bar.
----------	-----------	--

Property	BroadcastOnEveryProgress	Get or set the broadcast trigger for each progress point.
----------	--------------------------	---

Property	BroadcastOnComplete	Get or set the broadcast trigger for completion.
----------	---------------------	--

Usage Example for GameProgressTemplate

```

csharpCopy codeusing UnityEngine;

public class ProgressManager : StudioBehavior
{
    void ManageGameProgress()
    {
        // Accessing the wrapper
        GameProgressTemplate template = (GetTemplate(typeof(GameProgressTemplate)) as GameProgressTemplat

```

```
e);

// Accessing and setting the CurrentProgress property
int currentProgress = template.CurrentProgress; // Getting the current progress

// Accessing and setting the LerpSpeed property
float lerpSpeed = template.LerpSpeed; // Getting the lerp speed
template.LerpSpeed = 1.0f; // Setting the lerp speed to 1.0

// Accessing and setting the BroadcastOnEveryProgress property
string broadcastOnEveryProgress = template.BroadcastOnEveryProgress; // Getting the broadcast
trigger
template.BroadcastOnEveryProgress = "progress_event"; // Setting the broadcast
trigger

// Accessing and setting the BroadcastOnComplete property
string broadcastOnComplete = template.BroadcastOnComplete; // Getting the broadcast trigger
template.BroadcastOnComplete = "completion_event"; // Setting the broadcast trigger
}
}
```

GameScoreTemplate

This template manages in-game scoring functionality. Access it as follows:

Properties and Methods in GameScoreTemplate

Type	Name	Description
Property	CurrentScore	Get or set the current score.
Property	BestScore	Get or set the best score achieved.
Property	IsScoreUIShown	Check if the score UI is currently displayed.
Property	IsBestScoreCalculated	Check if the best score calculation is enabled.
Event	OnScoreModified	Event triggered when the score is modified.

Usage Example in GameScoreTemplate

```
public class ScoreManager : StudioBehavior
{
    void ManageScore()
    {
        // Accessing the wrapper
        GameScoreTemplate template = (GetTemplate(typeof(GameScoreTemplate)) as GameScoreTemplate);

        // Accessing and setting the CurrentScore property
        int currentScore = template.CurrentScore; // Getting the current score

        // Accessing and setting the BestScore property
        int bestScore = template.BestScore; // Getting the best score

        // Checking if the score UI is shown
        bool isScoreUIShown = template.IsScoreUIShown; // Checking if the score UI is currently displayed
    }
}
```

```

        // Checking if the best score calculation is enabled
        bool isBestScoreCalculated = template.IsBestScoreCalculated; // Checking if best score
calculation is enabled

        // Subscribing to the OnScoreModified event
        template.OnScoreModified += OnScoreModifiedHandler; // Subscribe to the score modification event

        // Unsubscribing from the OnScoreModified event
        template.OnScoreModified -= OnScoreModifiedHandler; // Unsubscribe from the score modification
event
    }

    void OnScoreModifiedHandler(int modifiedScore)
    {
        // Handle score modification
    }
}

```

GrowTemplate

This template manages the growth and scale behavior of GameObjects.

Properties and Methods in GrowTemplate

Type	Name	Description
Property	ScaleTo	Get or set the scale to which the object grows.
Property	Speed	Get or set the speed at which the object grows.
Property	ShouldPingPong	Check or set whether the growth should alternate between scaling up and down.
Property	RepeatForever	Check or set whether the growth should repeat indefinitely.
Property	RepeatFor	Get or set the duration for which the growth should repeat.
Event	OnRepetitionOver	Event triggered when the repetition is over.
Event	OnGrowFinished	Event triggered when the growth is finished.

Usage Example for GrowTemplate

```

public class GrowthManager : StudioBehavior
{
    void ManageGrowth()
    {
        // Accessing the wrapper
        GrowTemplate template = (GetTemplate(typeof(GrowTemplate)) as GrowTemplate);

        // Accessing and setting the ScaleTo property
        Vector3 scaleTo = template.ScaleTo; // Getting the target scale
        template.ScaleTo = new Vector3(2f, 2f, 2f); // Setting the target scale

        // Accessing and setting the Speed property
        float speed = template.Speed; // Getting the growth speed
        template.Speed = 1.5f; // Setting the growth speed

        // Accessing and setting the ShouldPingPong property
        bool shouldPingPong = template.ShouldPingPong; // Checking if ping pong behavior is enabled
    }
}

```

```

        template.ShouldPingPong = true; // Enabling ping pong behavior

        // Accessing and setting the RepeatForever property
        bool repeatForever = template.RepeatForever; // Checking if growth repeats indefinitely
        template.RepeatForever = true; // Enabling repeat forever

        // Accessing and setting the RepeatFor property
        int repeatFor = template.RepeatFor; // Getting the repeat duration
        template.RepeatFor = 10; // Setting the repeat duration to 10 seconds

        // Subscribing to the OnRepetitionOver event
        template.OnRepetitionOver += OnRepetitionOverHandler; // Subscribe to the event

        // Unsubscribing from the OnRepetitionOver event
        template.OnRepetitionOver -= OnRepetitionOverHandler; // Unsubscribe from the event

        // Subscribing to the OnGrowFinished event
        template.OnGrowFinished += OnGrowFinishedHandler; // Subscribe to the event

        // Unsubscribing from the OnGrowFinished event
        template.OnGrowFinished -= OnGrowFinishedHandler; // Unsubscribe from the event
    }

    void OnRepetitionOverHandler()
    {
        // Handle repetition over event
    }

    void OnGrowFinishedHandler()
    {
        // Handle growth finished event
    }
}

```

InGameTimerTemplate

This template manages in-game timer functionality. Access it as follows:

Properties and Methods in InGameTimerTemplate

Type	Name	Description
Property	TimerType	Get the type of timer.
Property	CurrentTime	Get the current time from the in-game timer handler.
Property	IsUIShown	Check if the UI associated with the timer is currently shown.
Event	OnTimerUpdated	Event triggered when the timer is updated.

Usage Example for InGameTimerTemplate

```

public class TimerManager : StudioBehavior
{
    void ManageTimer()
    {
        // Accessing the wrapper
    }
}

```

```

    InGameTimerTemplate template = (GetTemplate(typeof(InGameTimerTemplate)) as InGameTimerTemplate);

    // Accessing the TimerType property
    TimerType timerType = template.TimerType; // Getting the type of timer

    // Accessing the CurrentTime property
    float currentTime = template.CurrentTime; // Getting the current time

    // Checking if the UI is shown
    bool isUIShown =
template.IsUIShown; // Checking if the UI associated with the timer is currently shown

    // Subscribing to the OnTimerUpdated event
    template.OnTimerUpdated += OnTimerUpdatedHandler; // Subscribe to the event

    // Unsubscribing from the OnTimerUpdated event
    template.OnTimerUpdated -= OnTimerUpdatedHandler; // Unsubscribe from the event
}

void OnTimerUpdatedHandler(float updatedTime)
{
    // Handle timer updated event
}
}

```

JumpPadTemplate

This template manages jump pad functionality within the game. Access it as follows:

Properties and Methods in JumpPadTemplate

Type	Name	Description
Property	JumpForce	Get or set the jump force of the jump pad.

Usage Example for JumpPadTemplate

```

public class JumpPadManager : StudioBehavior
{
    void ConfigureJumpPad()
    {
        // Accessing the wrapper
        JumpPadTemplate template = (GetTemplate(typeof(JumpPadTemplate)) as JumpPadTemplate);

        // Accessing and setting the JumpForce property
        float jumpForce = template.JumpForce; // Getting the jump force
        template.JumpForce = 10.0f;           // Setting the jump force
    }
}

```

KillPlayerTemplate

This template manages the behavior related to killing the player in the game.

Properties and Methods in KillPlayerTemplate

Type	Name	Description
Property	AnimationToPlayOnDeath	Get or set the animation to play when the player dies.
Property	Delay	Get or set the delay before respawning the player.
Property	RespawnType	Get or set the respawn type (snap or lerp).
Property	LerpTime	Get or set the lerp time for respawn.
Event	OnAnimationEnded	Event triggered when the animation ends.
Event	OnRespawned	Event triggered when the player respawns.

Usage Example for KillPlayerTemplate

```
public class PlayerManager : StudioBehavior
{
    void ConfigurePlayerBehavior()
    {
        // Accessing the wrapper
        KillPlayerTemplate template = (GetTemplate(typeof(KillPlayerTemplate)) as KillPlayerTemplate);

        // Accessing and setting the AnimationToPlayOnDeath property
        string animation = template.AnimationToPlayOnDeath; // Getting the death animation
        template.AnimationToPlayOnDeath = "death_animation"; // Setting the death animation

        // Accessing and setting the Delay property
        float delay = template.Delay; // Getting the respawn delay
        template.Delay = 3.0f;        // Setting the respawn delay

        // Accessing and setting the RespawnType property
        RespawnV2.RespawnType respawnType = template.RespawnType; // Getting the respawn type
        template.RespawnType = RespawnV2.RespawnType.Lerp;        // Setting the respawn type

        // Accessing and setting the LerpTime property
        float lerpTime = template.LerpTime; // Getting the lerp time
        template.LerpTime = 2.0f;           // Setting the lerp time

        // Subscribing to the OnAnimationEnded event
        template.OnAnimationEnded += HandleAnimationEnd; // Subscribe to the event

        // Subscribing to the OnRespawned event
        template.OnRespawned += HandlePlayerRespawn; // Subscribe to the event

        // Unsubscribing from the OnAnimationEnded event
        template.OnAnimationEnded -= HandleAnimationEnd; // Unsubscribe from the event

        // Unsubscribing from the OnRespawned event
        template.OnRespawned -= HandlePlayerRespawn; // Unsubscribe from the event
    }

    void HandleAnimationEnd()
    {
        // Handle animation end logic here
    }
}
```



```
void HandlePlayerRespawn()
{
    // Handle player respawn logic here
}
}
```

LightTemplate

This template manages the behavior related to controlling a light component in the game. Access it as follows:

Properties and Methods in LightTemplate

Type	Name	Description
Property	Color	Get or set the color of the light.
Property	Intensity	Get or set the intensity of the light.
Event	OnLightPropertiesModified	Event triggered when the light properties are modified.

Usage Example for LightTemplate

```
public class LightController : StudioBehavior
{
    void ConfigureLight()
    {
        // Accessing the wrapper
        LightTemplate template = (GetTemplate(typeof(LightTemplate)) as LightTemplate);

        // Accessing and setting the Color property
        Color color = template.Color; // Getting the light color
        template.Color = Color.red;   // Setting the light color

        // Accessing and setting the Intensity property
        float intensity = template.Intensity; // Getting the light intensity
        template.Intensity = 2.0f;           // Setting the light intensity

        // Subscribing to the OnLightPropertiesModified event
        template.OnLightPropertiesModified += HandleLightPropertiesModified;

        // Unsubscribing from the OnLightPropertiesModified event
        template.OnLightPropertiesModified -= HandleLightPropertiesModified;
    }

    void HandleLightPropertiesModified()
    {
        // Handle light properties modified logic here
    }
}
```

MoveBetweenPointsTemplate

This template facilitates movement between specified points in the game, utilizing interpolation techniques for smooth transitions. Access it as follows:

Properties and Methods in MoveBetweenPointsTemplate

Type	Name	Description
Property	Points	Access or modify the points defining the movement path.
Property	Speed	Get or set the speed of movement between points.
Property	TurnToPoints	Enable or disable rotation towards movement points.
Property	DelayAtPoint	Specify a delay upon reaching each movement point.
Property	Loop	Enable or disable looping of the movement path.
Property	DoCurve	Activate or deactivate curve interpolation for movement.
Property	LoopType	Specify the interpolation type used for movement looping.
Event	OnIntervalReached	Event triggered when an interval is reached.
Event	OnReachedEnd	Event triggered when the movement ends.
Event	OnLoopFinished	Event triggered when the movement loop is finished.

Usage Example for MoveBetweenPointsTemplate

```
public class MovementController : StudioBehavior
{
    void ConfigureMovement()
    {
        // Accessing the wrapper
        MoveBetweenPointsTemplate template = (GetTemplate(typeof(MoveBetweenPointsTemplate)) as MoveBetweenPointsTemplate);

        // Accessing and setting the Points property
        List<Vector3> points = template.Points; // Getting the movement points
        template.Points = new List<Vector3> { /* Define new points here */ }; // Setting new movement points

        // Accessing and setting the Speed property
        float speed = template.Speed; // Getting the movement speed
        template.Speed = 5.0f; // Setting the movement speed

        // Accessing and setting the TurnToPoints property
        bool turnToPoints = template.TurnToPoints; // Getting rotation towards points
        template.TurnToPoints = true; // Enabling rotation towards points

        // Accessing and setting the DelayAtPoint property
        float delay = template.DelayAtPoint; // Getting delay at points
        template.DelayAtPoint = 1.5f; // Setting delay at points

        // Accessing and setting the Loop property
        bool loop = template.Loop; // Getting loop status
        template.Loop = true; // Enabling loop

        // Accessing and setting the DoCurve property
        bool doCurve = template.DoCurve; // Getting curve interpolation status
        template.DoCurve = true; // Enabling curve interpolation

        // Accessing and setting the LoopType property
        InterpolateType loopType = template.LoopType; // Getting loop interpolation type
```

```

        template.LoopType = InterpolateType.Linear; // Setting loop interpolation type

        // Subscribing to events
        template.OnIntervalReached += HandleIntervalReached;
        template.OnReachedEnd += HandleMovementEnd;
        template.OnLoopFinished += HandleLoopFinished;

        // Unsubscribing from events
        template.OnIntervalReached -= HandleIntervalReached;
        template.OnReachedEnd -= HandleMovementEnd;
        template.OnLoopFinished -= HandleLoopFinished;
    }

    void HandleIntervalReached()
    {
        // Handle interval reached logic here
    }

    void HandleMovementEnd()
    {
        // Handle movement end logic here
    }

    void HandleLoopFinished()
    {
        // Handle loop finished logic here
    }
}

```

MoveTemplate

This template facilitates movement functionality within the game, focusing on translating entities using specified parameters. Access it as follows:

Properties and Methods in MoveTemplate

Type	Name	Description
Property	Speed	Get or set the speed of movement.
Property	Loop	Get or set whether the movement should loop.
Property	Interval	Get or set the pause interval during movement.
Property	Point	Get or set the target position for movement.
Event	OnStopped	Event triggered when the movement is stopped.
Event	OnResumed	Event triggered when the movement is resumed.

Usage Example for MoveTemplate

```

public class MovementController : StudioBehavior
{
    void ConfigureMovement()
    {
        // Accessing the wrapper
        MoveTemplate template = (GetTemplate(typeof(MoveTemplate)) as MoveTemplate);
    }
}

```

```

// Accessing and setting the Speed property
float speed = template.Speed; // Getting the movement speed
template.Speed = 5.0f;        // Setting the movement speed

// Accessing and setting the Loop property
bool loop = template.Loop; // Getting loop status
template.Loop = true;      // Enabling looping movement

// Accessing and setting the Interval property
float interval = template.Interval; // Getting pause interval
template.Interval = 1.5f; // Setting pause interval

// Accessing and setting the Point property
Vector3 point = template.Point; // Getting target position
template.Point = new Vector3(10f, 0f, 5f); // Setting target position

// Subscribing to events
template.OnStopped += HandleStopped;
template.OnResumed += HandleResumed;

// Unsubscribing from events
template.OnStopped -= HandleStopped;
template.OnResumed -= HandleResumed;
}

void HandleStopped()
{
    // Handle stopped logic here
}

void HandleResumed()
{
    // Handle resumed logic here
}
}

```

MoveToPlayerTemplate

This template facilitates movement towards a player entity within the game, utilizing specified parameters. Access it as follows:

Properties and Methods in MoveToPlayerTemplate

Type	Name	Description
Property	Speed	Get or set the speed of movement towards the player.
Property	Offset	Get or set the offset relative to the player position.
Property	CancelType	Get or set the type of cancellation for movement.
Event	OnCancelled	Event triggered when the movement is canceled.
Event	OnReachedDestination	Event triggered when the destination is reached.
Event	OnReachedInitialPosition	Event triggered when the initial position is reached.

Usage Example for MoveToPlayerTemplate

```
public class PlayerMovementController : StudioBehavior
{
    void ConfigurePlayerMovement()
    {
        // Accessing the wrapper
        MoveToPlayerTemplate template = (GetTemplate(typeof(MoveToPlayerTemplate)) as MoveToPlayerTemplate);

        // Accessing and setting the Speed property
        float speed = template.Speed; // Getting movement speed towards player
        template.Speed = 7.0f;        // Setting movement speed towards player

        // Accessing and setting the Offset property
        Vector3 offset = template.Offset; // Getting movement offset
        template.Offset = new Vector3(0f, 1f, 0f); // Setting movement offset

        // Accessing and setting the CancelType property
        MoveToPlayer.CancelType cancelType = template.CancelType; // Getting cancellation type
        template.CancelType = MoveToPlayer.CancelType.Immediate; // Setting cancellation type

        // Subscribing to events
        template.OnCancelled += HandleCancelled;
        template.OnReachedDestination += HandleReachedDestination;
        template.OnReachedInitialPosition += HandleReachedInitialPosition;

        // Unsubscribing from events
        template.OnCancelled -= HandleCancelled;
        template.OnReachedDestination -= HandleReachedDestination;
        template.OnReachedInitialPosition -= HandleReachedInitialPosition;
    }

    void HandleCancelled()
    {
        // Handle cancellation logic here
    }

    void HandleReachedDestination()
    {
        // Handle logic when reaching the player destination here
    }

    void HandleReachedInitialPosition()
    {
        // Handle logic when reaching initial position here
    }
}
```

ParticleEffectTemplate

This template manages particle effects within the game, offering flexibility in configuration and event handling. Access it as follows:

Properties and Methods in ParticleEffectTemplate

Type	Name	Description
Property	RepeatCount	Get or set the number of times the particle effect repeats.
Property	PlayForever	Get or set whether continuous playback is enabled.
Property	Duration	Get or set the duration of the particle effect.
Property	Delay	Get or set the delay between repetitions of the effect.
Event	OnParticlePlayingCompleted	Event triggered when the particle effect playback completes.

Usage Example for ParticleEffectTemplate

```
public class ParticleEffectController : StudioBehavior
{
    void ConfigureParticleEffect()
    {
        // Accessing the wrapper
        ParticleEffectTemplate template = (GetTemplate(typeof(ParticleEffectTemplate)) as ParticleEffectTemplate);

        // Accessing and setting the RepeatCount property
        int repeatCount = template.RepeatCount; // Getting repeat count
        template.RepeatCount = 3; // Setting repeat count

        // Accessing and setting the PlayForever property
        bool playForever = template.PlayForever; // Getting continuous playback status
        template.PlayForever = true; // Enabling continuous playback

        // Accessing and setting the Duration property
        float duration = template.Duration; // Getting effect duration
        template.Duration = 5.0f; // Setting effect duration

        // Accessing and setting the Delay property
        int delay = template.Delay; // Getting delay value
        template.Delay = 2; // Setting delay between repetitions

        // Subscribing to events
        template.OnParticlePlayingCompleted += HandleParticlePlayingCompleted;

        // Unsubscribing from events
        template.OnParticlePlayingCompleted -= HandleParticlePlayingCompleted;
    }

    void HandleParticlePlayingCompleted()
    {
        // Handle logic when particle effect playback completes here
    }
}
```

PlayAnimationTemplate

This template facilitates the management and execution of animations within the game environment, offering seamless integration and customization options. Access it as follows:

Properties and Methods in PlayAnimationTemplate

Type	Name	Description
Property	CurrentAnimation	Retrieve the currently playing animation.
Property	DefaultAnimation	Set the default animation for the component.
Property	AvailableAnimations	Access the list of available animations.
Method	PlayAnimationOverride	Play a specified animation with optional looping.
Method	StopAnimationOverride	Stop the currently playing animation.
Event	OnAnimationCompleted	Event triggered when an animation completes.

Usage Example for PlayAnimationTemplate

```
public class AnimationController : StudioBehavior
{
    void ConfigureAnimation()
    {
        // Accessing the wrapper
        PlayAnimationTemplate template = (GetTemplate(typeof(PlayAnimationTemplate)) as PlayAnimationTemp
late);

        // Accessing and setting the DefaultAnimation property
        string defaultAnimation = template.DefaultAnimation; // Getting default animation
        template.DefaultAnimation = "Idle"; // Setting default animation

        // Accessing AvailableAnimations property
        TerraList availableAnimations = template.AvailableAnimations; // Getting available animations

        // Controlling animation playback
        template.PlayAnimationOverride("Run", true); // Playing animation with looping
        template.StopAnimationOverride(); // Stopping the current animation

        // Subscribing to events
        template.OnAnimationCompleted += HandleAnimationCompleted;

        // Unsubscribing from events
        template.OnAnimationCompleted -= HandleAnimationCompleted;
    }

    void HandleAnimationCompleted(string animationName)
    {
        // Handle logic when animation completes here
    }
}
```

PlayerAnimationControlTemplate

The PlayerAnimationControlTemplate class facilitates the control and management of player animations within the game environment. Access it as follows:

Properties and Methods in PlayerAnimationControlTemplate

Type	Name	Description
Property	AnimationName	Get or set the current animation name.
Property	ResetAnimationAutomatically	Enable or disable automatic animation reset.
Event	OnAnimationCompleted	Event triggered when an animation completes.

Usage Example for PlayerAnimationControlTemplate

```
public class PlayerAnimationController : StudioBehavior
{
    void ConfigurePlayerAnimation()
    {
        // Accessing the wrapper
        PlayerAnimationControlTemplate template = (GetTemplate(typeof(PlayerAnimationControlTemplate))
as PlayerAnimationControlTemplate);

        // Accessing and setting the AnimationName property
        string animationName = template.AnimationName; // Getting current animation name
        template.AnimationName = "Run"; // Setting animation name

        // Accessing and setting the ResetAnimationAutomatically property
        bool resetAutomatically = template.ResetAnimationAutomatically; // Getting auto-reset status
        template.ResetAnimationAutomatically = true; // Enabling automatic reset

        // Subscribing to events
        template.OnAnimationCompleted += HandleAnimationCompleted;

        // Unsubscribing from events
        template.OnAnimationCompleted -= HandleAnimationCompleted;
    }

    void HandleAnimationCompleted(bool success, string animationName)
    {
        // Handle logic when animation completes here
    }
}
```

PlayPlayersAnimationTemplate

The PlayPlayersAnimationTemplate class is designed to manage and control player animations within your game environment. Access it as follows:

Properties and Methods in PlayPlayersAnimationTemplate

Type	Name	Description
Property	AnimationName	Get or set the current animation name.
Property	ResetAnimationAutomatically	Enable or disable automatic animation reset.

Usage Example for PlayPlayersAnimationTemplate

```
public class PlayerAnimationController : StudioBehavior
{
    void ConfigurePlayerAnimation()
    {
        // Accessing the wrapper
        PlayPlayersAnimationTemplate template = (GetTemplate(typeof(PlayPlayersAnimationTemplate)) as PlayPlayersAnimationTemplate);

        // Accessing and setting the AnimationName property
        string animationName = template.AnimationName; // Getting current animation name
        template.AnimationName = "Jump"; // Setting animation name

        // Accessing and setting the ResetAnimationAutomatically property
        bool resetAutomatically = template.ResetAnimationAutomatically; // Getting auto-reset status
        template.ResetAnimationAutomatically = true; // Enabling automatic reset
    }
}
```

PushTemplate

The PushTemplate class facilitates control over push components within your game, offering methods to manage resistance properties effectively. Access it as follows:

Properties and Methods in PushTemplate

Type	Name	Description
------	------	-------------

Property	Resistance	Get or set the resistance of the push component.
----------	------------	--

Usage Example for PushTemplate

```
public class PushController : StudioBehavior
{
    void ConfigurePush()
    {
        // Accessing the wrapper
        PushTemplate template = (GetTemplate(typeof(PushTemplate)) as PushTemplate);

        // Accessing and setting the Resistance property
        float resistance = template.Resistance; // Getting resistance value
        template.Resistance = 2.5f; // Setting resistance value
    }
}
```

RandomBroadcastTemplate

This template manages random broadcasting functionality within the game, utilizing specified parameters to handle broadcast strings. Access it as follows:

Properties, Methods, and Indexer in RandomBroadcastTemplate

Type	Name	Description
Property	GetAllBroadcastingStrings	Retrieve all broadcasting strings.
Indexer	[index]	Access or modify a specific broadcasting string by index.
Method	GetEnumerator	Iterate through all broadcasting strings.
Method	CheckIndex	Validate an index before accessing or modifying broadcasting strings.
Method	UpdateConditionBroadcasts	Update a specific broadcasting string.

Usage Example for RandomBroadcastTemplate

```
public class BroadcastController : StudioBehavior
{
    void ConfigureBroadcast()
    {
        // Accessing the wrapper
        RandomBroadcastTemplate template = (GetTemplate(typeof(RandomBroadcastTemplate)) as RandomBroadcastTemplate);

        // Accessing all broadcasting strings
        List<string> broadcastingStrings = template.GetAllBroadcastingStrings; // Retrieving all strings

        // Accessing a specific broadcasting string
        string broadcast = template[0]; // Getting broadcast at index 0
        template[0] = "New Broadcast"; // Updating broadcast at index 0

        // Enumerating through broadcasting strings
        IEnumerator enumerator = template.GetEnumerator();
        while (enumerator.MoveNext())
        {
            string broadcastItem = (string)enumerator.Current;
            // Process each broadcast here
        }

        // Validating index
        try
        {
            template.CheckIndex(1);
            // Proceed with index-valid operations
        }
        catch (ArgumentOutOfRangeException ex)
        {
            // Handle index out of range exception
        }

        // Updating a broadcasting string
        template.UpdateConditionBroadcasts(1, "Updated Broadcast");
    }
}
```

ResetScoreTemplate

This template manages the resetting of scores within the game, utilizing specified parameters to handle score groups. Access it as follows:

Properties in ResetScoreTemplate

Type	Name	Description
Property	ScoreGroup	Get or set the score group.

Usage Example for ResetScoreTemplate

```
public class ScoreManager : StudioBehavior
{
    void ConfigureScore()
    {
        // Accessing the wrapper
        ResetScoreTemplate template = (GetTemplate(typeof(ResetScoreTemplate)) as ResetScoreTemplate);

        // Accessing and setting the ScoreGroup property
        string scoreGroup = template.ScoreGroup; // Getting score group
        template.ScoreGroup = "NewScoreGroup"; // Updating score group
    }
}
```

RotateOscillateTemplate

This template manages the rotation oscillation behavior within the game. Access it as follows:

Properties and Methods in RotateOscillateTemplate

Type	Name	Description
Property	RotationAxis	Get or set the rotation axis.
Property	Degrees	Get or set the degrees of rotation.
Property	Direction	Get or set the rotation direction.
Property	RepeatCount	Get or set the repeat count for the rotation.
Property	Loop	Check if the rotation is looping.
Property	StopOn	Get or set the condition to stop the rotation.
Property	ResumeOn	Get or set the condition to resume the rotation.
Event	OnRepetitionCompleted	Event triggered when a repetition completes.
Event	OnRotationCompleted	Event triggered when rotation completes.
Event	OnRotationPaused	Event triggered when rotation is paused.
Event	OnRotationResumed	Event triggered when rotation resumes.

Usage Example for RotateOscillateTemplate

```
public class RotationController : StudioBehavior
{
    void ConfigureRotation()
    {
```

```

// Accessing the wrapper
RotateOscillateTemplate template = (GetTemplate(typeof(RotateOscillateTemplate)) as RotateOscillateTemplate);

// Accessing and setting properties
Axis rotationAxis = template.RotationAxis; // Getting rotation axis
template.RotationAxis = Axis.Y; // Setting rotation axis

int degrees = template.Degrees; // Getting degrees
template.Degrees = 90; // Setting degrees

Direction direction = template.Direction; // Getting direction
template.Direction = Direction.Clockwise; // Setting direction

int repeatCount = template.RepeatCount; // Getting repeat count
template.RepeatCount = 5; // Setting repeat count

bool isLooping = template.Loop; // Checking if looping
template.StopOn = "SomeCondition"; // Setting stop condition
template.ResumeOn = "SomeOtherCondition"; // Setting resume condition

// Subscribing to events
template.OnRepetitionCompleted += HandleRepetitionCompleted;
template.OnRotationCompleted += HandleRotationCompleted;
template.OnRotationPaused += HandleRotationPaused;
template.OnRotationResumed += HandleRotationResumed;

// Unsubscribing from events
template.OnRepetitionCompleted -= HandleRepetitionCompleted;
template.OnRotationCompleted -= HandleRotationCompleted;
template.OnRotationPaused -= HandleRotationPaused;
template.OnRotationResumed -= HandleRotationResumed;
}

void HandleRepetitionCompleted()
{
    // Handle repetition completed logic here
}

void HandleRotationCompleted()
{
    // Handle rotation completed logic here
}

void HandleRotationPaused()
{
    // Handle rotation paused logic here
}

void HandleRotationResumed()
{
    // Handle rotation resumed logic here
}
}

```

RotateTemplate

This template manages rotation behavior within the game. Access it as follows:

Properties in RotateTemplate

Type	Name	Description
Property	RotationAxis	Get or set the rotation axis.
Property	Speed	Get or set the speed of rotation.
Property	Direction	Get or set the rotation direction.

Usage Example for RotateTemplate

```
public class RotationController : StudioBehavior
{
    void ConfigureRotation()
    {
        // Accessing the wrapper
        RotateTemplate template = (GetTemplate(typeof(RotateTemplate)) as RotateTemplate);

        // Accessing and setting properties
        Axis rotationAxis = template.RotationAxis; // Getting rotation axis
        template.RotationAxis = Axis.Y; // Setting rotation axis

        float speed = template.Speed; // Getting speed
        template.Speed = 5.0f; // Setting speed

        Direction direction = template.Direction; // Getting direction
        template.Direction = Direction.Clockwise; // Setting direction
    }
}
```

SetPositionTemplate

This template is used to set the position of an object within the game engine. Access it as follows:

Properties and Methods in SetPositionTemplate

Type	Name	Description
Property	TargetPosition	Get or set the target position of the object.
Event	OnPositionUpdated	Event triggered when the position is updated.

Usage Example for SetPositionTemplate

```
public class PositionController : StudioBehavior
{
    void ConfigurePosition()
    {
        // Accessing the wrapper
        SetPositionTemplate positionTemplate = (GetTemplate(typeof(SetPositionTemplate)) as SetPositionTemplate);
    }
}
```

```

        // Accessing and setting TargetPosition property
        Vector3 targetPosition = positionTemplate.TargetPosition; // Getting target position
        positionTemplate.TargetPosition = new Vector3(10, 20, 30); // Setting target position

        // Subscribing to events
        positionTemplate.OnPositionUpdated += HandlePositionUpdated;

        // Unsubscribing from events
        positionTemplate.OnPositionUpdated -= HandlePositionUpdated;
    }

    void HandlePositionUpdated()
    {
        // Custom logic to execute when the position is updated
        Debug.Log("Position updated");
    }
}

```

ShowUITemplate

This template manages the display of UI elements within the game. Access it as follows:

Properties

Type	Name	Description
Property	AnimationType	Get or set the UI animation type.
Property	ScreenPosition	Get or set the screen position.
Property	UIToShow	Get or set the UI element to show.
Property	Icon1Name	Get or set the name of the first icon.
Property	Icon2Name	Get or set the name of the second icon.
Property	Text1	Get or set the first text.
Property	Text2	Get or set the second text.
Property	Text3	Get or set the third text.
Property	AnimationDuration	Get or set the animation duration.
Property	UIDuration	Get or set the UI display duration.

Usage Example

```

public class UIController : StudioBehavior
{
    void ConfigureUI()
    {
        // Accessing the wrapper
        ShowUITemplate template = (GetTemplate(typeof(ShowUITemplate)) as ShowUITemplate);

        // Accessing and setting properties
        UIAnimation animationType = template.AnimationType; // Getting UI animation type
        template.AnimationType = UIAnimation.FadeIn; // Setting UI animation type

        UIPoint screenPosition = template.ScreenPosition; // Getting screen position
    }
}

```

```

        template.ScreenPosition = new UIPoint(100, 200); // Setting screen position

        string uiToShow = template.UIToShow; // Getting UI element to show
        template.UIToShow = "MainMenu"; // Setting UI element to show

        string icon1Name = template.Icon1Name; // Getting first icon name
        template.Icon1Name = "Icon1"; // Setting first icon name

        string icon2Name = template.Icon2Name; // Getting second icon name
        template.Icon2Name = "Icon2"; // Setting second icon name

        string text1 = template.Text1; // Getting first text
        template.Text1 = "Welcome"; // Setting first text

        string text2 = template.Text2; // Getting second text
        template.Text2 = "Start Game"; // Setting second text

        string text3 = template.Text3; // Getting third text
        template.Text3 = "Options"; // Setting third text

        float animationDuration = template.AnimationDuration; // Getting animation duration
        template.AnimationDuration = 1.5f; // Setting animation duration

        float uiDuration = template.UIDuration; // Getting UI display duration
        template.UIDuration = 5.0f; // Setting UI display duration
    }
}

```

SoundFxTemplate

This template manages sound effects within the game, allowing customization of volume, pitch, 3D audio settings, distance ranges, looping capabilities, and pause/resume events. Access it as follows:

Properties and Methods in SoundFxTemplate

Type	Name	Description
Property	Volume	Get or set the volume.
Property	Pitch	Get or set the pitch.
Property	Is3DAudio	Get or set whether the sound is 3D audio.
Property	MinDistance	Get or set the minimum distance for 3D audio.
Property	MaxDistance	Get or set the maximum distance for 3D audio.
Property	CanLoop	Get or set whether the sound can loop.
Property	PauseOn	Get or set the event on which the sound pauses.
Property	ResumeOn	Get or set the event on which the sound resumes.
Event	OnPaused	Event triggered when the sound is paused.
Event	OnResumed	Event triggered when the sound is resumed.

Usage Example for SoundFxTemplate

```

public class SoundManager : StudioBehavior
{

```

```

void ConfigureSound()
{
    // Accessing the wrapper
    SoundFxTemplate template = (GetTemplate(sizeof(SoundFxTemplate)) as SoundFxTemplate);

    // Accessing and setting properties
    float volume = template.Volume; // Getting volume
    template.Volume = 0.5f; // Setting volume

    float pitch = template.Pitch; // Getting pitch
    template.Pitch = 1.0f; // Setting pitch

    bool is3DAudio = template.Is3DAudio; // Checking if 3D audio
    template.Is3DAudio = true; // Enabling 3D audio

    float minDistance = template.MinDistance; // Getting minimum distance
    template.MinDistance = 1.0f; // Setting minimum distance

    float maxDistance = template.MaxDistance; // Getting maximum distance
    template.MaxDistance = 50.0f; // Setting maximum distance

    bool canLoop = template.CanLoop; // Checking if looping
    template.CanLoop = true; // Enabling looping

    string pauseOn = template.PauseOn; // Getting pause event
    template.PauseOn = "PlayerDeath"; // Setting pause event

    string resumeOn = template.ResumeOn; // Getting resume event
    template.ResumeOn = "PlayerRespawn"; // Setting resume event

    // Subscribing to events
    template.OnPaused += OnSoundPaused;
    template.OnResumed += OnSoundResumed;

    // Unsubscribing from events
    template.OnPaused -= OnSoundPaused;
    template.OnResumed -= OnSoundResumed;
}

void OnSoundPaused()
{
    // Your code here
}

void OnSoundResumed()
{
    // Your code here
}
}

```

SwitchTemplate

This template manages the switch component within the game, allowing custom actions when toggled on or off. Access it as follows:

Events in SwitchTemplate

Type	Name	Description
Event	OnToggledOn	Event triggered when the switch is toggled on.
Event	OnToggledOff	Event triggered when the switch is toggled off.

Usage Example for SwitchTemplate

```
public class SwitchController : StudioBehavior
{
    void ConfigureSwitch()
    {
        // Accessing the wrapper
        SwitchTemplate template = (GetTemplate(typeof(SwitchTemplate)) as SwitchTemplate);

        // Subscribing to events
        template.OnToggledOn += OnSwitchToggledOn;
        template.OnToggledOff += OnSwitchToggledOff;

        // Unsubscribing from events
        template.OnToggledOn -= OnSwitchToggledOn;
        template.OnToggledOff -= OnSwitchToggledOff;
    }

    void OnSwitchToggledOn()
    {
        // Your code here
    }

    void OnSwitchToggledOff()
    {
        // Your code here
    }
}
```

TeleportTemplate

This template manages the teleport component within the game, allowing custom actions when teleportation occurs. Access it as follows:

Properties and Methods in TeleportTemplate

Type	Name	Description
Property	TeleportPoint	Get or set the teleport destination point.
Event	OnTeleported	Event triggered when teleportation occurs.

Usage Example for TeleportTemplate

```
public class TeleportController : StudioBehavior
{
    void ConfigureTeleport()
    {
        // Accessing the wrapper
        TeleportTemplate template = (GetTemplate(typeof(TeleportTemplate)) as TeleportTemplate);

        // Accessing and setting TeleportPoint property
        Vector3 currentPoint = template.TeleportPoint; // Getting teleport point
        template.TeleportPoint = new Vector3(10, 20, 30); // Setting teleport point

        // Subscribing to events
        template.OnTeleported += OnTeleport;

        // Unsubscribing from events
        template.OnTeleported -= OnTeleport;
    }

    void OnTeleport()
    {
        // Your code here
    }
}
```

TickTemplate

This template manages the tick component within the game, providing access to conditions and events related to tick operations. Access it as follows:

Properties and Methods in TickTemplate

Type	Name	Description
Indexer	[index]	Get or set values by index.
Property	Length	Get the number of condition broadcasts.
Property	StartOn	Get or set the start condition.
Property	StopOn	Get or set the stop condition.
Property	ResumeOn	Get or set the resume condition.
Event	OnStarted	Event triggered when the tick starts.
Event	OnPaused	Event triggered when the tick is paused.
Event	OnResumed	Event triggered when the tick is resumed.
Event	OnTick	Event triggered for each tick.

Usage Example for TickTemplate

```
public class TickController : StudioBehavior
{
    void ConfigureTick()
    {
        // Accessing the wrapper
```

```

TickTemplate template = (GetTemplate(typeof(TickTemplate)) as TickTemplate);

// Accessing and setting properties
int length = template.Length; // Getting length
template.StartOn = "StartCondition"; // Setting start condition
template.StopOn = "StopCondition"; // Setting stop condition
template.ResumeOn = "ResumeCondition"; // Setting resume condition

// Subscribing to events
template.OnStarted += OnTickStarted;
template.OnPaused += OnTickPaused;
template.OnResumed += OnTickResumed;
template.OnTick += OnTickAction;

// Unsubscribing from events
template.OnStarted -= OnTickStarted;
template.OnPaused -= OnTickPaused;
template.OnResumed -= OnTickResumed;
template.OnTick -= OnTickAction;
}

string this[int index]
{
    get { return template[index]; } // Getting value by index
    set { template[index] = value; } // Setting value by index
}

void OnTickStarted()
{
    // Your code here
}

void OnTickPaused()
{
    // Your code here
}

void OnTickResumed()
{
    // Your code here
}

void OnTickAction()
{
    // Your code here
}
}

```

TogglePlayerMovementTemplate

This template manages the player movement component, specifically handling stun animation and shader effects, and provides events for pause and resume actions. Access it as follows:

Properties and Methods in TogglePlayerMovementTemplate

Type	Name	Description
Property	PlayStunAnimation	Get or set the stun animation state.
Property	ChangeToStunShader	Get or set the stun shader state.
Property	ResumeOn	Get or set the resume condition.
Event	OnPaused	Event triggered when the player movement is paused.
Event	OnResumed	Event triggered when the player movement is resumed.

Usage Example for TogglePlayerMovementTemplate

```
public class PlayerMovementController : StudioBehavior
{
    void ConfigureMovement()
    {
        // Accessing the wrapper
        TogglePlayerMovementTemplate template = (GetTemplate(typeof(TogglePlayerMovementTemplate)) as TogglePlayerMovementTemplate);

        // Accessing and setting properties
        bool isStunAnimationOn = template.PlayStunAnimation; // Getting stun animation state
        template.PlayStunAnimation = true; // Setting stun animation state

        bool isStunShaderOn = template.ChangeToStunShader; // Getting stun shader state
        template.ChangeToStunShader = true; // Setting stun shader state

        string resumeCondition = template.ResumeOn; // Getting resume condition
        template.ResumeOn = "newResumeCondition"; // Setting resume condition

        // Subscribing to events
        template.OnPaused += OnMovementPaused;
        template.OnResumed += OnMovementResumed;

        // Unsubscribing from events
        template.OnPaused -= OnMovementPaused;
        template.OnResumed -= OnMovementResumed;
    }

    void OnMovementPaused()
    {
        // Your code here
    }

    void OnMovementResumed()
    {
        // Your code here
    }
}
```

UpdateScoreTemplate

This template manages the score component, handling score groups, modifiers, and score values. It integrates with the `ScoreHandler` to get and set scores. Access it as follows:

Properties and Methods in UpdateScoreTemplate

Type	Name	Description
Property	ScoreGroup	Get or set the score group.
Property	Modifier	Get or set the modifier for score updates.
Property	Score	Get or set the score value.
Method	GetScore	Retrieve the current score from the <code>ScoreHandler</code> .
Method	SetScore	Set a new score using the <code>ScoreHandler</code> .

Usage Example for UpdateScoreTemplate

```
public class ScoreManager : StudioBehavior
{
    void ConfigureScore()
    {
        // Accessing the wrapper
        UpdateScoreTemplate template = (GetTemplate(typeof(UpdateScoreTemplate)) as UpdateScoreTemplate);

        // Accessing and setting properties
        string scoreGroup = template.ScoreGroup; // Getting score group
        template.ScoreGroup = "newScoreGroup"; // Setting score group

        Modifier modifier = template.Modifier; // Getting modifier
        template.Modifier = Modifier.Add; // Setting modifier

        int score = template.Score; // Getting score
        template.Score = 100; // Setting score

        // Using methods
        int currentScore = template.GetScore(); // Getting current score
        template.SetScore(150); // Setting a new score
    }
}
```

UpdateTimerTemplate

This template manages the timer component, handling modifiers, update intervals, and current time. It integrates with the `InGameTimeHandler` to get and set in-game time. Access it as follows:

Properties and Methods in UpdateTimerTemplate

Type	Name	Description
Property	Modifier	Get or set the modifier for timer updates.
Property	UpdateBy	Get or set the update interval.
Property	CurrentTime	Get or set the current time.
Method	GetTime	Retrieve the current time from the <code>InGameTimeHandler</code> .

Method `SetTime` Set a new time using the `InGameTimeHandler`.

Usage Example for UpdateTimerTemplate

```
public class TimerManager : StudioBehavior
{
    void ConfigureTimer()
    {
        // Accessing the wrapper
        UpdateTimerTemplate template = (GetTemplate(typeof(UpdateTimerTemplate)) as UpdateTimerTemplate);

        // Accessing and setting properties
        Modifier modifier = template.Modifier; // Getting modifier
        template.Modifier = Modifier.Add; // Setting modifier

        int updateBy = template.UpdateBy; // Getting update interval
        template.UpdateBy = 5; // Setting update interval

        float currentTime = template.CurrentTime; // Getting current time
        template.CurrentTime = 10.5f; // Setting current time

        // Using methods
        float time = template.GetTime(); // Getting current time
        template.SetTime(12.0f); // Setting a new time
    }
}
```

T# StudioController Wrappers

StudioController

The `StudioController` class represents a controller for the player in the game. It provides various functionalities to interact with and manipulate the player character's properties and behaviors.

Properties and Methods in StudioController

Type	Name	Description
Method	<code>GetMyController</code>	Retrieves the current player's <code>StudioController</code> .
Method	<code>CheckIfController</code>	Checks if a given <code>GameObject</code> has an associated <code>StudioController</code> .
Property	<code>IsMoving</code>	Checks if the player is moving.
Property	<code>GetPlayerPosition</code>	Gets the player's current position.
Property	<code>SetPlayerPosition</code>	Sets the player's position with optional smooth movement and camera transition.
Property	<code>GetPlayerForward</code>	Gets the forward direction of the player.
Property	<code>SetPlayerRotation</code>	Sets the player's rotation.
Property	<code>GetOverHeadLocator</code>	Gets the transform of the overhead locator.
Property	<code>GetControllerCamera</code>	Gets the camera associated with the player controller.
Property	<code>ResetCameraCurrentPosition</code>	Resets the camera to its current position.
Property	<code>GetLocator</code>	Finds and returns a transform based on a given locator name.
Property	<code>GetPlayerData</code>	Returns the underlying player data associated with this controller.
Property	<code>GetLocatorEnumBased</code>	Gets the transform of a location based on the <code>PlayerLocEnum</code> .

Usage Example

```
public class PlayerControllerExample : StudioBehavior
{
    void ManagePlayerController()
    {
        // Retrieve the current player's controller
        StudioController myController = StudioController.GetMyController();

        // Check if a GameObject has an associated StudioController
        StudioController controller = StudioController.CheckIfController(gameObject);

        // Check if the player is moving
        bool isMoving = myController.IsMoving();

        // Get the player's current position
        Vector3 playerPosition = myController.GetPlayerPosition();

        // Set the player's position with smooth movement and camera transition
        myController.SetPlayerPosition(new Vector3(0, 0, 0), smoothMove: true, smoothCamera: true);

        // Get the forward direction of the player
        Vector3 forwardDirection = myController.GetPlayerForward();

        // Set the player's rotation
        myController.SetPlayerRotation(Quaternion.identity);

        // Get the transform of the overhead locator
        Transform overHeadLocator = myController.GetOverHeadLocator();

        // Get the camera associated with the player controller
        Camera controllerCamera = myController.GetControllerCamera();

        // Reset the camera to its current position
        myController.ResetCameraCurrentPosition();

        // Find and return a transform based on a given locator name
        Transform locator = myController.GetLocator("locatorName");

        // Return the underlying player data associated with this controller
        IPlayerData playerData = myController.GetPlayerData();

        // Get the transform of a location based on the PlayerLocEnum
        Transform locatorEnumBased = myController.GetLocatorEnumBased(PlayerLocEnum.LeftLegWingLoc);
    }
}
```

Available Locators for `PlayerLocEnum`

```
PlayerLocEnum
{
    LeftLegWingLoc,
    LeftLegGreavesLoc,
```

```

RightLegWingLoc,
RightLegGreavesLoc,
LeftGlovesLoc,
LeftLowerArmLoc,
LeftShoulderLoc,
LeftShoulderArmourLoc,
HaloLoc,
EyewearLoc,
RightEarWearLoc,
HeadWearLoc,
MaskLoc,
LeftEarWearLoc,
CamLoc,
ShootIKLoc,
RightPalmLoc,
RightGlovesLoc,
RightLowerArmLoc,
RightShoulderLoc,
RightShoulderArmourLoc,
WeaponLoc,
BackLoc,
NeckLoc,
PetLoc,
LegendaryPetLoc,
ShoulderPetLoc,
BaseLoc
}

```

T# Haptics & Extensions

StudioHaptics

This class provides static methods to play different haptic feedback patterns. It uses the Lofelt Nice Vibrations library to trigger haptic feedback on supported devices.

Properties and Methods in StudioHaptics

Type	Name	Description
Method	<code>PlayHapticSelection</code>	Plays the haptic feedback for selection.
Method	<code>PlayHapticSuccess</code>	Plays the haptic feedback for success.
Method	<code>PlayHapticWarning</code>	Plays the haptic feedback for warning.
Method	<code>PlayHapticFailure</code>	Plays the haptic feedback for failure.
Method	<code>PlayHapticLightImpact</code>	Plays the haptic feedback for light impact.
Method	<code>PlayHapticMediumImpact</code>	Plays the haptic feedback for medium impact.
Method	<code>PlayHapticHeavyImpact</code>	Plays the haptic feedback for heavy impact.
Method	<code>PlayHapticRigidImpact</code>	Plays the haptic feedback for rigid impact.
Method	<code>PlayHapticSoftImpact</code>	Plays the haptic feedback for soft impact.

Usage Example

```
public class HapticFeedbackExample : StudioBehavior
{
    void TriggerHapticFeedback()
    {
        // Play haptic feedback for selection
        StudioHaptics.PlayHapticSelection();

        // Play haptic feedback for success
        StudioHaptics.PlayHapticSuccess();

        // Play haptic feedback for warning
        StudioHaptics.PlayHapticWarning();

        // Play haptic feedback for failure
        StudioHaptics.PlayHapticFailure();

        // Play haptic feedback for light impact
        StudioHaptics.PlayHapticLightImpact();

        // Play haptic feedback for medium impact
        StudioHaptics.PlayHapticMediumImpact();

        // Play haptic feedback for heavy impact
        StudioHaptics.PlayHapticHeavyImpact();

        // Play haptic feedback for rigid impact
        StudioHaptics.PlayHapticRigidImpact();

        // Play haptic feedback for soft impact
        StudioHaptics.PlayHapticSoftImpact();
    }
}
```

StudioExtensions

This class provides static methods for various utility functions within the Terra Studio environment.

Properties and Methods in StudioExtensions

Type	Name	Description
Method	SetCurrentScore	Sets the current score for a specified group.
Method	GetCurrentScore	Gets the current score for a specified group.
Method	DestroyObject	Destroys the specified GameObject. (Deprecated: Use StudioBehavior.Destroy() instead)
Method	LoadScene	Loads the specified scene by name.
Method	GetAllScenes	Returns a list of all scenes.
Method	GetCurrentScene	Returns the name of the current scene.
Method	GetColorFromHex	Converts a hex string to a Color.
Method	FindDeepChild	Finds a child transform by name, searching recursively.

Usage Example

```
public class UtilityFunctionsExample : StudioBehavior
{
    void ExecuteUtilityFunctions()
    {
        // Set the current score for a group
        StudioExtensions.SetCurrentScore("group1", 100);

        // Get the current score for a group
        int score = StudioExtensions.GetCurrentScore("group1");

        // Destroy a GameObject (deprecated)
        StudioExtensions.DestroyObject(gameObject);

        // Load a scene by name
        StudioExtensions.LoadScene("SceneName");

        // Get all scenes. Remember to use TerraList and not List, since List is not supported
        TerraList allScenes = StudioExtensions.GetAllScenes();

        // Get the current scene name
        string currentScene = StudioExtensions.GetCurrentScene();

        // Convert a hex string to a Color
        Color color = StudioExtensions.GetColorFromHex("#FFFFFF");

        // Find a child transform by name
        Transform child = StudioExtensions.FindDeepChild(parentTransform, "ChildName");
    }
}
```

Adding UI using T#

To add game UI using T#, you need to do the following

1. From the main toolbar, pick the primitives dropdown. Select `Edit UI`
2. You will see a UI Picker in the Inspector Panel. From the UI picker, choose the UI you want
3. Edit in portrait/ landscape as you wish.

Terra studio will automatically handle the spawning of the UI in landscape / portrait depending on the device specs. The UI instantiated in play-mode will be separate from the one in the editor, but maintains the same hierarchy and edits you make. To access the instantiated UI, these are the steps:

1. Add the Studio Machine Logic Component to "EditCanvas_1" parent in Layers
2. Add the following code Snippet that logs "Hello" when a button in the UI is clicked. Customize as required

```
// This script will add a button UI to the game which when clicked will log the message Hello on the
Debug panel
public class TestScript : StudioBehaviour
{
    private void Start()
    {
```

```
var UI = GetTemplate(typeof(EditUITemplate)) as EditUITemplate;
var instantiatedUI = UI.GetInstantiatedUI;
var buttonObj = StudioExtensions.FindDeepChild(instantiatedUI.transform, "[BUTTONNAMEHERE]");
var buttonComponent = buttonObj.GetComponent(typeof(Button)) as Button;
buttonComponent.onClick.AddListener(()=>{Debug.LogError("Hello");}); // Can be modified to
perform the action you want
    }
private void Update()
{
}
public override void OnBroadcasted(string x)
{
}
}
```