conceptual-guides

---

description: The high level design of any game

Building Blocks of a Game

Any game in Terra consists of four basic blocks - Player, GameObjects, Components and Broadcasts

Player

At the core of any game in Terra lies the "Player" , which is the graphical avatar representing players in the game world. The Player is customisable through a collection of editable features called Player Properties

GameObjects

Having a player isn't enough for a game. You need GameObjects which the the player can explore and interact with. Without GameObjects, the player would just have a big empty space. There are three main types of GameObjects in Terra Studio:

- The "Asset" GameObject - Any 3D object that is part of the game's world.
- The "SFX" GameObject - A GameObject with a Sound file attached to it
- The "Particle" GameObject - A GameObject with a visual effect or a particle effect attached to it

Components

Simply having a Player block and a GameObject block won't create an engaging game experience. Interactivity is key, and this is where Components plays a pivotal role. They are the essence of gameplay, turning a static scene into an immersive, interactive world. Components are attached to GameObjects in a way where you can program the Components to follow specific instructions, enabling interactivity in the game in precisely the ways you want.

Integrating Components into your game transforms the game into dynamic, interactive elements. Without Components, you're left with a mere navigation through static scenes.

A lot of Components for game logic are pre-built in the form of Logic Templates or Game Systems. However, you can also build your own Components using scripting.

Broadcasts

Broadcasts are messages that are used by Components and GameObjects to communicate with each other. There are a few readily available broadcasts in the editor

- **Game Win**: It is the broadcast message automatically generated when the player meets the win condition for a game
- **Game Lose**: It is the broadcast message automatically generated when the player meets the lose condition for a game
- **Custom Broadcast**: This lets you send a unique alert whenever a specific game condition happens. You can name each custom signal differently - and this is usually a string field.

Putting it all together

In a game, the player's journey is shaped by the seamless interaction of these four core components: Player, GameObjects, Components, and Broadcasts. Here's how these elements work together to create an immersive experience:

1. **Starting as the Player**
    - The journey begins with the **Player**, the avatar representing the user within the game world. As the player steps into the game, they control this avatar, navigating through the environment and interacting with the world around them.
2. **Interacting with GameObjects**
    - As the Player moves through the game, they encounter various **GameObjects**. These are the elements that fill the game world—3D assets, sound effects, and particle effects—that make the environment rich and engaging. GameObjects provide the physical and visual context within which the player's journey unfolds.
3. **Engaging with Components**
    - However, simply moving through a world of GameObjects would be a passive experience without the **Components** that bring it to life. Components are attached to GameObjects, turning them from static objects into interactive elements. For example, when the Player approaches a door (a GameObject), a Component might trigger the door to open. Similarly, picking up an item might increase the Player's score or trigger a sound. Components are the rules and logic that govern how GameObjects respond to the Player, making the game interactive and engaging.
4. **Triggering Events with Broadcasts**
    - Throughout the journey, certain actions or events need to be communicated across different parts of the game. This is where **Broadcasts** come into play. For instance, when the Player reaches the end of a level, a "Game Win" Broadcast might be sent out, triggering the display of a victory screen. Or if the Player loses all their health, a "Game Lose" Broadcast could initiate the game-over sequence. **Custom Broadcasts** allow for more specific interactions, such as triggering an alarm when the Player steps into a restricted area. Broadcasts ensure that all game elements work together, responding to the Player's actions in real-time.

Summary

Games are built using four basic blocks:

- **Player**: Represents the character avatar you control.
- **GameObjects**: Construct the environment of the game.
- **Components**: Make elements interactive, allowing you to engage with the world.
- **Broadcasts**: Messages that are used by Components and GameObjects to communicate with each other

Configuring the Player

At the core of every Terra game is the "Player," the visual character that represents users in the game world. As a game developer, the model you see in the editor is a placeholder that lets you customize all of the player's actions and animations.

However, you cannot alter the player's appearance for the user. The user has the ability to select their own custom avatar, which will follow the actions and animations you've set, but in the appearance they have chosen to play with.

Finding and Selecting the Player

You can select the player by clicking on the player's avatar that is visible in the editor mode. If you are unable to find the player, you can follow these steps:

- Click on `Layers` in the Quick Access Menu on the left
- Locate and Click on "Player".
- Press the F button

You will notice that the Player has a default child element TopDownPlayer which in turn has two more children - ModelRoot and CameraRoot.

Once you find and select the player in the editor, the Inspector Panel on the right displays a list of customisable Player Properties.

Transforming Initial Player Configurations

To do this, you need to select the topmost parent element in the Layers , named Player.

You will notice that can do three basic transformations of the player's initial configuration- `Move` , `Rotate` and `Scale`. When an object is selected, the Gizmo containing 3 handles will appear around it, each representing an axis along which you can manipulate the object. Alternatively, you can also manually change the values in the Inspector panel to get the exact initial configuration of the player you want.

Selecting a Gameplay Controller

Gameplay Controller in the context of mobile or tablet gaming is a system that interprets the player's input on the touchscreen into actions and movements within the game.

To change the player controller, you need to select the topmost parent element in the Layers , named Player.

When you select Player, you will see a dropdown on the inspector panel named Controller which shows the TopDown controller by default.

You can also import other controller packages from the Main Toolbar on the top of the interface by going to `File` > `Import`

Controller behavior can be customized in T# using the `StudioController` class that helps you access the controller for the player in the game. It provides some functionalities to interact with and manipulate the player character's properties and behaviors, which are listed below:

Properties and Methods in StudioController

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

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);
    }
}
```

Editing Player Motion Properties

You can alter how players move through the game space. To edit these properties, you need to select the TopDown Player element (child of the Player element). Once you select it, you can by tweak these properties:

- **Lock to Look Sideways**: Toggle button to lock or unlock sideways view.
- **Scale**: Adjust the player's size (1 = Default).
- **Gravity**: Numerical field to set gravity strength. Higher values make jumps shorter and falls faster, affecting game difficulty.
- **Jump Height**: Set the maximum height for a player's jump. This influences gameplay difficulty and accessibility.
- **Jump Lock**: Toggle to disable jumping via a controller. Useful for creating specific challenges or controlling movement in certain areas.
- **Max Speed**: Define the player's maximum achievable speed.

Editing Player Animation

The Player's animation can be edited by selecting the `ModelRoot` element under the `TopDownPlayer` element in Layers. The following animations can be edited by choosing from a list of pre-existing animations:

- Idle
- Forward Running
- Back Waling
- Walking Forward
- Walk to Right
- Walk Left
- On Jump Start
- While in Air
- On Jump End
- Bump from back
- Bump from front
- Bump from left
- Bump from right

Editing Player Camera Controls

Terra Studio allows creators to edit the following camera control parameters during game play and define how users experience their game visually. You can change this by selecting the Camera `Root` element under the `TopDownPlayer` element in Layers.

- **Camera Arc X Rot**: A numerical field through which you can define the camera's tilt, allowing for vertical adjustments. Players can aim upwards or downwards, enhancing interactions with objects or enemies located above or below.
- **Spherical Camera Positioning**:
    - **Camera Arc X**: Manages the camera's horizontal positioning, facilitating left-right movements around a focal point.
    - **Camera Arc Y**: Governs the camera's vertical placement for up-down movements, enabling varied angles and perspectives on the game environment.
- **Field of View (FoV) / Size**: A numerical field through which you can specify the visible game world area. A narrow FoV brings the world closer, ideal for precision tasks like sniping, while a wide FoV expands the view for exploratory gameplay, although excessive width can lead to edge distortion.
- **Camera Distance**: A numerical field through which you can adjusting the camera's proximity to focus points or characters, influencing the player's depth perception and detail awareness. Closer settings heighten immersion, whereas greater distances offer expansive views of the surroundings.
- **Camera Type**: Terra Studio supports both **perspective** and **orthographic** cameras, with each type significantly affecting how players perceive and navigate the game world.

Configuring GameObjects

Once you have the Player configured, you then proceed to set up your interactive world - using GameObjects. There are three types of GameObjects we typically use to set up the game scene:

- **Asset GameObjects:** 3D elements, items, and structures, that players can see and interact with in the game world.
- **SFX GameObjects:** Auditory elements and sound effects
- **Particle GameObjects:** Visual effects like smoke, fire, and magical effects, adding depth and realism to the game's visual presentation.

Here is how you can set up each of the above blocks in your scene:

Assets

Viewing and Adding New Assets to the Scene

Terra Studio has a thousands of pre-made assets for creators to use directly in their game without having to bother about creating their assets. Here are the steps you need to follow:

- Click on "Asset" in the Quick Access Menu.
- Select the "Library View" in the Asset Panel that shows up. You will be shown a library of all assets and their preview. You can search for your desired asset by
    - Typing Keywords in the Asset Search Bar
    - Filtering Assets by theme or category tags
- Once you find your desired asset, drag and drop it into the game workspace.

{% hint style="info" %} :dart: Currently you cannot import your own assets in the game. However, in an upcoming release cycle, you'll be able to upload your assets directly into the Terra Studio environment, making it easier than ever to personalise your games and bring your unique visions to life. {% endhint %}

Editing Asset Properties

Once an Asset is added to the scene, you can click on it and view a list of its editable properties in the Inspector Panel. You can edit the following properties of the asset:

Editing Asset Initial Configuration

You can adjust the initial position, initial rotation and initial scale of the Asset by changing the values manually in the inspector panel or by adjusting the Configuration Gizmo next to the Asset

Editing Asset Appearance

You can edit the asset's visual appearance by editing its material properties:

- Color: Pick from the color wheel or set RGB values
- Emission Color: Pick from the color wheel or set RGB values
- Texture: Select from a list of available textures
- Metallic: Specify a numerical value between 0 to 1
- Smoothness: Specify a numerical value between 0 to 1
- Shader: Select from a list of available shaders
- Tiling: Specify the X and Y tiling

Specifying Asset Collider

A Collider defines the physical boundaries of an object for collision detection. Colliders help determine when one object makes contact with another within a virtual space.

For any object in Terra Studio, you select from one of four types of collider shapes from the Inspector Panel dropdown:

- **Capsule:** Wraps the object in a capsule shape. Touching anywhere on the capsule counts as touching the object.
- **Sphere:** Surrounds the object with a spherical barrier. Touching the sphere means you've touched the object.
- **Box:** Encases the object in a box shape. Any contact with the box is like touching the object itself.
- **Mesh:** Follows the exact shape of the object. You have to touch the actual surface of the object to register a touch.

SFX

To infuse your game with more life and interactivity, sound effects (SFX) play a crucial role. Here's how to seamlessly integrate them into your game.

Selecting and Adding SFX

- **Navigate to the Builder Menu:** Start by opening the Quick Access Menu and selecting the `SFX` option. This will reveal a catalog of available sound effects suitable for your game's atmosphere and mechanics.
- **Preview SFX:** Ensure your selected SFX aligns with your vision by using the play button for a quick preview. This step is essential before setting the sound effect in stone.
- **Add Sound Effects:** Spot the `+` icon next to your chosen SFX in the list. Clicking this will automatically add the sound effect into your game's environment.

Configuring SFX

Once an SFX is added, it appears in the Layers panel. Here, you can select it to view all its configurable properties in the Inspector Panel. These are the properties you can change in the panel to get the sound effect you want :

- **SoundFx When:** Decide when the sound effect starts playing, either at the beginning of the game or after a specific event.
- **Pause on/Un-Pause on:** This dropdown helps you specify the game events (e.g., Game Start, Game Lose, Game Win) that will pause or resume the SFX playback.
- **Can Loop:** This toggle option for the sound to play continuously in a loop.
- **AudioPitch:** Alter the sound's pitch.
- **Volume of Sound:** Control the sound intensity at the source.
- **Is 3D Audio:** Toggle this to determine if the sound's properties change with distance, mimicking natural sound behavior. Disabling this makes the sound's volume and pitch uniform regardless of the listener's location relative to the source. Once this is active, you can specify
    - **Max Distance:** This helps you set the furthest distance at which the sound can be heard from its source.
    - **Minimum Distance:** Establishes the closest distance to the source needed to hear the sound, applicable when 3D audio is enabled.

You can also choose to customize sound effects through the SoundFxTemplate T# wrapper which manages sound effects within the game, allowing customization of volume, pitch, 3D audio settings, distance ranges, looping capabilities, and pause/resume events.

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(typeof(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
    }
}
```

Particles

Particles allow you to add moving elements such as fire, beams, aurora lights etc within the game environment.

Selecting and Adding Particles

- **Selecting the Particles:** Start by opening the Quick Access Menu and selecting the Particles option. This will reveal a catalog of available particle effects with their preview suitable for your game's atmosphere and mechanics.

- **Adding Particles :**Click on the effect you want and drag and drop it to the desired location in your game.

Customizing Particle Properties

Once you have added a particle effect to the game, select it from the Layers Panel . You will be shown a list of customizable properties in the Inspector Panel.

- **Particle VFX When:** Determines the condition under which the particle effect is activated. It can be triggered by a broadcast event or at the game's start.
- **Pause/Un-Pause on:** This dropdown allows you to select specific game events that will pause or resume the Sound Effects (SFX) playback. Options include events like Game Start, Game Lose, and Game Win.
- **Duration:** Sets the duration for which the particle effect will remain visible.
- **Delay Between:** Defines the delay interval between consecutive appearances of the particle effect.
- **Repeat Forever:** This Toggle, If enabled, the particle Visual Effects (VFX) will repeat continuously throughout the game.
- **Repeat Count:** Specifies the number of times the particle VFX will repeat.

You can also choose to customize particle effects through **ParticleEffectTemplate** which manages particle effects within the game, offering flexibility in configuration and event handling.

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 ParticleEffectT
emplate);

        // 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
    }
}
```

Game Environment

This feature allows you to modify the visual aspects of the game's environment, providing tools to enhance the aesthetic and mood of your game world. You can alter the following parameters of the skybox by selecting Essentials from the Quick Access Menu and the selecting GlobalRenderSettings in the Quick Access Panel. You will see the following editable parameters in the Inspector Panel.

Property	Description
`Skybox`	Adjusts the game's backdrop to give the illusion of a more expansive space. It's like changing the scenery outside a window to make a room feel bigger.
`Bloom`	Creates a glow effect around bright areas in the game, simulating the way light behaves in the real world. It adds a touch of realism by making lights and reflections seem to spill over their boundaries.
`Vignette`	Applies a shading around the screen's edges, drawing the player's focus to the center. This effect can give your game a more dramatic and cinematic feel, as if you're peering through a lens.
`Fog`	Introduces a misty overlay that can be adjusted for depth and density, perfect for setting a mysterious or eerie atmosphere. It's like adding a thin veil over the game world that can make far-off objects seem obscured and distant.

Adding Logic Components to Asset Game Objects

To make our game interactive, we introduce Components, which brings the game to life and changes a simple, static game into an engaging world that reacts to what the player does.

There are two ways to use Components in Terra Studio

1. Using pre-built Logic Components
2. Scripting custom logic components

Using pre-built Logic Components

To accelerate the time to create games, Terra Creator Studio also provides pre-built Logic Template components. These Templates represent various logical operations, conditions, and actions, and can be easily selected from the Editor and dragged and dropped onto Assets.

What is a pre-built logic component?

A pre-built logic component is always added to an Asset GameObject. It contains pre-built instructions on how the Player and the other GameObjects in the game should behave. Here are some important things to remember about templates

1. A Logic Template Component is always added to an Asset GameObject.

2. Execution of a Logic Template Component is always triggered by a Start Event
3. A Logic Template Component can affect not only the Asset GameObject to which it is attached, but also other GameObjects

Start Events for Logic Components

Logic Templates in Terra Studio start executing any of the five events listed below occur:

Start Event Name	Logic Template Component is executed when
Game Start	The game starts
Mouse Click	You click the mouse
Player Touch / Player Collide	The Player touches or collides with the Asset's collider.
Other Object Touch	The other Asset's collider touches the collider of the asset to which the Logic Template is attached
Broadcast Listened	A specified game signal is generated in the game. E.g - If the Start Event is set to Broadcast Listened - 'level_finish', the logic template will execute only when the game signal 'level_finish' is generated during the game.

How to add pre-built Logic Components

To add a logic template to an asset and customise it, follow these steps:

1. Select the Asset GameObject in the scene editor or through the `Layers` Panel.
2. Click the `Logic` tab in the Quick Access Menu on the left
3. You'll see a Logic Selector with all the possible logic templates
4. Choose the logic template you want and drag and drop into the Asset
5. The logic component has been added to the asset game object.
6. You can configure the logic component's properties by selecting the Advanced Mode toggle button and editing the various accessible fields.
7. Once you make changes, click the `Save` button in the main toolbar.

These components can either be executed simultaneously (in parallel) or in a sequence (one after the other). The key to controlling this execution order lies in using Broadcasts.

How It Works

To coordinate the execution of different logic components, we can use Broadcasts to link them together. Here's how it works:

Sequential Execution Using Broadcasts: Imagine you have two logic components: Logic Component A and Logic Component B. You want Component B to start only after Component A has finished executing. To achieve this, you set up Component **A** to generate a Broadcast once it completes its task. This Broadcast acts as a signal. Component B is configured to listen for this specific Broadcast. When it detects the signal, it knows that Component A has finished, and it begins its execution.

Example: Component A could be responsible for opening a door in the game. Once the door is fully open, it generates a Broadcast called "DoorOpened." **Component B**, which controls the sound of the door creaking, is set to listen for the "DoorOpened" Broadcast. As soon as it receives this signal, it triggers the door creaking sound.

Parallel Execution Without Waiting: If you want multiple components to execute at the same time, you don't need to link them with a Broadcast.

Example: You could have **Component C** controlling the player's movement and **Component D** controlling background music. These can run in parallel, with Component D playing the music while Component C handles player movement.

Why Use Broadcasts?

Broadcasts are especially useful when you need precise control over the order of events in your game. By coupling logic components with Broadcasts, you can create complex sequences of actions that unfold in the exact order you intend. For instance, you can ensure that a door opens before a sound plays, or that an enemy appears only after the player reaches a certain point.

This system allows you to build more sophisticated and engaging gameplay experiences by coordinating multiple actions across different components.

List of Available Logic Template Components

Terra Studio has a wide selection of logic template components for you to choose from. A logic template component can be added to any Asset GameObject and configured to elicit the interactivity you want in the game. The table below shows a list of logic template and a short description of what they do. A detailed description of each logic template is given in the respective logic template page.

Scene Management

Logic Template	Description
Checkpoint	Restarts the game from a specific point if you fail a challenge or lose a life
Update Timer	Updates the timer to a new specified value
Reset Timer	Resets the timer to zero
Load Scene	Loads a New Scene
Random Level Selector	Loads a random new scene on game start instead of the default scene

Mechanics

Logic Template	Description
Collectable	Enables an object to be collected by the player and update the game score. Used in Power-ups.
Teleport Player	Instantly spawns the Player in a new specified position
Jump Pad	Creates a jump enhancement for the player upon contact
Carryable	Enables an Asset to be carried by the Player. The Asset will now move with the Player
Deposit	Enables the Player to transfer the Carriable Asset and deposit it to a new Asset which is a storage
Modify Carryable	Modifies the number of carryables you have
Kill Player	Respawns the player to the start of the level
Hinge Joint	Enables assets to rotate about a defined hinge like a dore
Explosive Force	Applies a force / impulse on a radius
Add Force	Applies a force on an object and allows it to follow physics
Treadmill	Enables treadmill-like motion on contact
Multipoint Move	Shifts the Asset from its starting spot through a path of straight or curved points as needed.
Attach Object	Parents an object to another object

Actions

Logic Template	Description
Destroy	Destroys the Asset from the scene
Set Position	Changes the Asset's position
Advance Instantiate	Spawns an instance of the player (with advanced settings)
Grow / Shrink	Increases or decreases the size of the Asset
Move	Moves the Asset in a straight line path to a specified new position from its starting point.
Rotate	Rotates the Asset about a chosen axis
MoveTo Player	Moves the Asset to the Player
Rotate Oscillate	Oscilates the Asset about a specified axis within a specified rotation about the initial position
Basic Instantiate	Spawns an instance of the player
Bump	Bounce back when you run into it

Conditionals

Logic Template	Description
Switch	Helps activate or deactivate behaviors depending on the triggers associated with each action.
OR Gate	Acts as a gate that sends out a broadcast signal only after any one of the required conditions are met. These conditions are broadcast signals from various sources.
AND Gate	Acts as a gate that sends out a broadcast signal only after all required conditions are met. These conditions are broadcast signals from various sources. It won't activate until every condition is satisfied.
Tick	Generates a broadcast at a pre-defined time or time-intervals

Triggers

Logic Template	Description
Collide	Uses contact of collider of the player as a trigger and allows you to generate a broadcast
Click	Uses mouse click as a trigger and allows you to generate broadcast
Delay	Introduces a delay of a specified time

Effects

Logic Template	Description
Stop Rotate	Stops Rotation
ShowUI	Displays a UI on the screen
Stop Animation	Stops Animation
Play Player's Animation	Plays animation of the player

PlayerStats

Logic Template	Description
Update Score	Updates a specific score group to a new specified value
Reset Score	Resets the specified score group to zero
Increase Player HP	Increases the player Health value by the specific amount
Decrease Player HP	Decreases the player Health value by a specific amount
Reset Player Health	Resets the player Health value to zero
Level Up	Guides the Level Mapper on how to increase a property's level to the next tier.
Update Magnet	Changes the magnet range for the player's collection
Stop Player Movement	Stops or starts the player movements
Change Player Speed	Changes the speed of movement of the player

Scripting your own component in T#

Terra Creator Studio enables experienced game developers to implement custom logic using a scripting language called **T# (T-Sharp)**. T# is very similar to Unity's C#, making it easy for Unity developers to learn. The links below provide detailed guidance on writing T-Sharp code:

Basics of Scripting in T#

Unsupported Functionalities in T#

Hybrid Approach - Logic Template Component + Custom Script Components

Logic Components are useful for both beginners and experienced developers. For beginners, they reduce the need for extensive coding knowledge. For senior developers, they eliminate the need to write code from scratch for simple game interactions and save time and effort.

The limitation of Logic Template Components is the inflexibility in customizing interactions, as only exposed properties are editable from the editor. We address this by providing wrappers for Logic Templates, customizable via T-Sharp code. These wrappers expose all editable properties to the developer, allowing for more complex interactions than the editor interface permits.

You can read more about how to access the Logic Templates through T-Sharp code in the documentation for Logic Component Template Wrappers.

Adding Global Logic Components

Global Logic Components affect the overall game and not just individual GameObjects. These components run in the background but significantly influence the entire gameplay experience. All these components are accessible by clicking on the Essentials button in the Quick Access Menu. The Builder Panel then shows a list of all Game systems.

Some global logic components that are present in Terra Studio are listed below:

Game Timer

The Game Timer component is a critical system that manages time-related aspects within the game. It functions as a timer that integrates with game logic.This component is particularly useful in games involving countdowns or time-limited challenges, ensuring precise control over gameplay duration.

Adding a Game Timer

To add a game timer, follow these steps:

- Select the joystick icon in the Main Tool bar
- Click on Game Timer

You can now see the Game Timer's properties in the Inspector Panel on the right. The game timer also shows up on the Essentials Tab in the Quick Access Menu

Configuring the Game Timer

You can configure the following properties of the game timer in the Inspector Panel:

Parameter	Description
`Duration Input`	Enter the time in seconds for how long the timer should run.
`Timer Type`	Choose between counting up towards a target time (Count Up) or counting down from a set time (Count Down).
`Generate Broadcast on Completion`	Generate either a Game Win signal, a Game Lose signal, or a Custom Broadcast Signal once the timer is complete.
`Advanced`	Configure a broadcast signal based on either a specific time or a repeat interval: • Under Advanced Options, choose "Broadcast Type" and select either "At" for a specific time or "Every" for a repeat interval. • If you select "At," input the time in seconds for when the signal should occur (e.g., "35" for the 35th second). • If you select "Every," enter the interval in seconds for the repeat (e.g., "5" for a signal every 5 seconds).

Parameter	Description
	• Choose your broadcast "Game Win," "Game Lose," or a Custom signal.
`Show UI Toggle`	Choose to display the timer on-screen or keep it running silently in the background.

Customizing Game Timer Behavior using T#

You can also customize the game timer by accessing it's T# wrapper - InGameTimerTemplate This template manages in-game timer functionality.

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
    }
}
```

You can also use the UpdateTimer which manages the timer component, handling modifiers, update intervals, and current time. It integrates with the `InGameTimeHandler` to get and set in-game time.

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 `InGameTimeHandler`.
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
    }
}
```

Score

The Score system tracks how players perform in a game, showing their competition level and progress. Every game automatically includes a primary score group called the Main Score_GameScore. This group becomes active when you use certain game logic templates to change scores.

Score Groups

A score group is a system used to track various scores within a game, allowing you to monitor multiple achievements or performance metrics independently. For example, a game might have separate score groups for different objectives, such as collecting different types of items or completing various tasks. Score Groups helps in tracking specific areas of a player's performance and progress. Each metric you want to track will have a separate score group associated with it.

Every game automatically includes a primary score group called **Main Score_GameScore**. This group becomes active when certain game logic templates are used to modify scores. The value within a score group can only be altered by specific logic components:

- **Collectable Logic Components**: Changes the score when items are picked up.
- **Update Score Logic Components**: Directly modifies the score.
- **Reset Score Logic Component**: Resets the score to zero.

- **Carriable Logic Component**: Uses the score as currency for carrying certain objects.
- **Deposit Logic Component**: Uses the score to place items in a designated location.

Creating & Updating Score Groups

You can create a new score group only when you add one of the five mentioned logic components. In the editable properties of the behavior, you'll find a "Score Group" option. Here, you can choose to update the existing "Main Score_GameScore" or create a new custom score group by selecting Custom. If you select Custom Score Group, a new score group will be created, and it will appear under Essentials alongside the "Main Score_GameScore."

After this you must also decide how much to change the score when the event starts. You can pick any integer for the change.

Configuring Score Groups

You can further configure Score Groups by selecting Essentials from the Quick Access Menu and clicking on the relevant score group you want to customize. Here are the options available

- **ShowUI Toggle Button**: Shows the score on the game screen when this is turned on.
- **UI Prefab Dropdown**: This option lets you choose from a dropdown menu the UI templates where ShowUI will display the score. These templates are predetermined.
- **Persistent Toggle**: Allows you to keep the same score group total when moving from one level to the next. If not chosen, the score group starts over at zero with each new level.
- **Save Best Score Toggle**: When selected, this saves the player's highest score.
- **Show Best ScoreUI Toggle**: When you turn on the "Save Best Score" option, it displays your highest score on the game screen.
- **BestScore UI PreFab Dropdown**: This lets you choose the UI templates that should display the highest score in ShowUI.

Scores can also be configured using T# wrappers - , GetScoreTemplate, UpdateScoreTemplate and ResetScore Template.

The GetScore template manages in-game scoring functionality.

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
    }
}
```

The UpdateScore 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 `ScoreHandler`.
Method	SetScore	Set a new score using the `ScoreHandler`.

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
    }
}
```

The ResetScoreTemplate manages the resetting of scores within the game, utilizing specified parameters to handle score groups and reset them. 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
    }
}
```

Health

The Health system is a background system that tracks the player health. There are only three behavior blocks that can affect player health - the Increase Player HP, the Decrease Player HP and the Reset Player Health logic components. By default, the player health is set to a value of 100. When the player health becomes zero, the player is respawned to the last checkpoint in the game, unless configured otherwise.

To configure the Health System, click on Essentials in the Quick Access Menu and select PlayerHealth. You can then configure the following properties of the Health system:

- **Auto Heal Rate**: Turn on Auto Heal and choose how fast the Health should go up every second.
- **Generate a Broadcast when Player Health becomes zero**: You can create a broadcast for winning a game, losing a game, or a custom message.

Game Progress

The Game Progress System tracks the player's advancement in the game according to predefined progress points. It provides players with a sense of achievement and progression as they play. Multiple progress points can be added through the inspector panel, allowing for greater customization and flexibility in the game's structure.

To add a game progress system,

1. From the main toolbar at the top of the editor screen, select "GameProgress" to add the game system to the Essentials tab.
2. In the builder menu on the left side of the screen, navigate to the "Essential" tab.
3. Find the "GameProgress" system in the builder panel.
4. Click on "GameProgress" to open the inspector panel.

5. In the inspector panel, you can customize the parameters according to your needs and preferences.

Parameters	Description
Progress start	This parameter helps you define the point the game progress will be tracked. Game starts - At the start of the game Broadcast Listened -Starts tracking the progress when it listens to a broadcast from another object
Progress points	List of diffrent milestones in the game
Persistent	Game progress came be made persistent by checking the checkbox
BroadCast at points	Broadcast can be trigger when player reaches a paticular milestone
BroadCast On Completion	Broadcast can be triggered when player reaches the fine milestone.

Level Mapper

In game development, leveling up is a core feature that not only marks progression but also enriches the player's experience by offering tangible rewards and new challenges. The Level Mapper game system is designed for customizing the upgrade paths of objects within a game. This can include example use-cases like:

- **Weapon evolution**: Weapons can evolve or be upgraded as players collect certain materials or reach particular benchmarks
- **Building upgrades**: to unlock new features or improve their efficiency (upgrading a farm may increase food production rates or a barracks might allow for the training of more advanced units.)
- **Character Progression:** Accumulating enough XP results in the character gaining a level, which might increase the character's stats (like health, strength, agility) and unlock new abilities or skills.

By utilizing the Level Mapper, developers can set clear, trackable goals for players, encouraging engagement and offering rewarding gameplay experiences as they watch their in-game assets grow and evolve.\ The level-up behavior can be utilized to adjust the progression paths of objects based on the Level Mapper system.

How to add Level mapper?

1. From the main toolbar at the top of the editor screen, select "LevelUpgrader" to add the game system to the Essentials tab.
2. In the builder menu on the left side of the screen, navigate to the "Essential" tab.
3. Find the "LevelMapper" system in the builder panel.
4. Click on "LevelMapper" to open the inspector panel.
5. In the inspector panel, you can customize the parameters according to your needs and preferences.

Parameter	Description
Group	custom name allows the game to keep track of the variables that helps level up the game. steps: 1. Click on custom option 2. Name the variable in the field added below.
Cost Type	The parameter that helps achieve the next level There are two options available: 1. Resource (Game Score) 2. Carryable select any one based on the game requirement.
Resource Tag	This is the parameter whose value will be affected once we achieve level up.
Value	Update in the property of the group name after you achieve level upgrade
Currency	Cost of upgrading to a new level

1. By following these steps, you will have successfully designed upgrade paths for objects within the game.
2. To use this system to upgrade different in-game objects, utilize the Level Up behavior. Refer to the provided link to learn how to add the "Level Up" behavior.

Customer Manger

Customer manager is a game system used to spawn and manage the customer. This is a game system mostly used in the tycoon games.

To add this game system, follow these steps:

1. Navigate to the essentials tab from the builder menu.
2. Click on "Customer Manager".
3. In the Inspector panel, you can customize the below-mentioned parameters according to your requirements:

Parameter	Description
Customer Manager When	This dropdown list allows to specify when new customers will be spawned. It can be triggered either on game start or on any broadcast.
Broadcast data	This dropdown can be used to add any broadcast at every customer being spawned.
Delay between	This Field can be used to add a delay between spawning of each round of customer.
Level	This section is used to adjust the rate at which customers are spawned in progressive manner. you can create multiples levels and define the range of customer spawned for each level.
Defficulty	You need to define "X" and "Y' values. the number of customer spawned will lie between these values. once the number of customer spawned reaches the "Max spawn" value, the level will get upgraded and next level block will get executed
Max spawn	You can specify here the max number of customer that can be spawned in each level

Order Generator

Order Generator is a game system responsible for generating order for each customer based on the availabilty of items in the store. This is a game system mostly used in the tycoon games.

To add this game system, follow these steps:

1. Navigate to the essentials tab from the builder menu.
2. Click on "Customer Manager".
3. In the Inspector panel, you can customize the below-mentioned parameters according to your requirements:

Parameter	Description
Type	This dropdown list is used to specify the type of order. It can either be a storage or a service.
Maximum number in an order	This field can be used to specify the maximum number of items that an order can have.
Items	
Group	
Cost	
Data	This section is used to adjust the number of item per order at each specific level. you can create multiples levels and define the range of items that each order should contain at any specific level.
Difficulty	You need to define "X" and "Y' values. the number of items in the order will lie between these values. once the number of items reaches the "Threshold" value, the level will get upgraded and next level block will get executed
Max spawn	You can specify here the max number of items that can be added in order at each level.

Path finder

Path finder is a game system that is used to define the area in which the customer can move. This is a game system mostly used in the tycoon games.

To add this game system, follow these steps:

1. Navigate to the essentials tab from the builder menu.
2. Click on "Customer Manager".
3. In the Inspector panel, you can customize the below-mentioned parameters according to your requirements:

Parameter	Description
Path finder UI initializes when	
Lowest point	This coordinate is used to define the one of cordinate of the diganol of the rectangular area in which the customers can move.
Highest point	This coordinate is used to define the other coordinate of the diganol of the rectangular area in which the customers can move.
Navmesh accur	
Obstacle avoiding distance offset	This field can be used to define the distance that the player would maintain from the obstacle
Minimum height	This field is used to define the minimum height below which any structure in the player's path would be considered as obstacle
Maximum height	This field is used to define the maximum height below which any structure in the player's path would be considered as obstacle and player needs to avoid.
POI distance offset	This field is used to define the distance that player need to maintain from POI in the game.
Broadcast	Choose to enter a broadcast that can be used as a trigger for any other behavior.