3D PUZZLE BLOCK GAME TEMPLATE DOCUMENTATION

Game Overview

The player interacts with blocks of various types (Tetris-like) that they must place on a grid. The blocks can be dragged, rotated, and positioned to fit within predefined slots or target positions. The primary goal of the game is to correctly position these blocks in the grid, with feedback provided for incorrect placements.

Block Interaction

Each block is represented as an object in the game that can be manipulated by the player. The player can perform the following actions:

1. **Click & Drag**: Blocks can be clicked and dragged across the screen. During dragging, the blocks are moved based on the player's mouse movement. The position is updated in real time, but the block’s colliders are temporarily disabled to prevent interference with other objects.
2. **Rotation**: The block can be rotated clockwise or counterclockwise, giving the player the ability to adjust the block's orientation. The rotation is smooth, happening over time using a coroutine for a better visual experience.
3. **Error Handling**: When a block is placed incorrectly, it is returned to its previous position or a random position. Visual feedback is provided, such as changing the block’s material and applying a shake effect to indicate the error. This provides an intuitive way for the player to know when they’ve made a mistake.

Block Placement

- **Valid Placement**: The goal is to correctly place the block within the grid by aligning its tiles with the target positions. If the block is placed correctly, it locks into place on the grid and cannot be moved again unless certain conditions are met (e.g., level reset or error handling).
- **Invalid Placement**: If the player places the block incorrectly (e.g., the block doesn't fit into the required position or overlaps with another block), the block will return to its previous position or a random location. During this process, error feedback is displayed through the shake animation and a change in material to indicate the error visually.

Block Tiles & Types

Each block is made up of one or more `BlockTile` components that correspond to different sections of the block. These tiles are what the player positions in the grid. Blocks may have various types (e.g., different shapes or colors), which may influence how they interact with the grid or other blocks.

Level and Grid

The game operates on a level-based system, with a grid where blocks must be placed. The grid serves as a predefined space for the blocks, and the blocks' target positions are checked for validity as the player moves and places them. Each level may have different configurations or puzzles to solve.

Block Feedback Mechanisms

- **Visual Feedback**:
    - Blocks that are placed incorrectly trigger visual feedback. The material of the block changes to signify an error, and a shake effect is applied to show the player that they need to reposition the block.
    - Blocks also have a "bounce" effect, which adds a bit of animation for blocks that are successfully placed.
- **Feedback on Actions**: As blocks are rotated or dragged, the visual state is updated in real time, with markers or indicators showing the block's current position and potential final placement.

Gameplay Flow

1. **Start of the Level**: The player starts with a set of blocks and is presented with a grid or puzzle. The objective is to place these blocks in a valid configuration.
2. **Drag & Rotate Blocks**: The player can drag and rotate blocks to try and fit them into the grid's designated spots. If the player clicks and holds the block, it enters drag mode, allowing them to move the block around the screen.
3. **Check Placement**: After positioning and rotating the block, the game checks if the block fits within the target area in the grid. If it's placed correctly, the block locks into place.
4. **Invalid Placement**: If the placement is incorrect, the game provides feedback by shaking the block and changing its material. The block is then moved back to its previous position or to a random position.
5. **Level Completion**: The level is completed when all blocks are placed correctly in their designated spots, and the player successfully solves the puzzle.

Block Movement & Collision

- **Collisions with Other Blocks**: While dragging, the block's colliders are temporarily disabled to avoid collision detection with other blocks or grid targets. This allows the player to freely move the block across the screen without interference.
- **Snap-to-Grid Behavior**: After dragging, the block snaps into its final position if placed correctly. This snap effect ensures the block aligns precisely with the grid and target positions.

Summary of Game Mechanics:

- **Drag and Rotate**: The player can drag blocks around the grid and rotate them.
- **Snap-to-Grid**: Blocks lock into place if placed correctly on the grid.
- **Feedback**: Incorrect placements trigger error animations (shaking, material change).
- **Block Tiles**: Blocks are made up of tiles that need to be correctly positioned.
- **Level Objective**: Correctly position all blocks in a predefined grid to complete the level.

Tile Class

The `Tile` class represents a tile in the game. It handles assigning an owner to the tile and updating the tile's properties based on the owner.

Public Variables:

- `Owner` **(GameObject):**
  - The GameObject that owns the tile. Initially, it's `null`.

Methods:

- `Start()` **(void):**
  - Called when the game starts. If the tile has an owner, it sets the `isStatic` property of the owner's `BlockTile` component to `true`.
- `AddOwner(GameObject Who)` **(bool):**
  - **Parameters:** `Who` (GameObject) - The GameObject to assign as the tile's owner.
  - **Returns:** `true` if the owner is successfully added; `false` if the tile already has an owner.
  - **Description:** This method assigns a new owner to the tile if the tile doesn't already have one. It also updates the owner's `BlockTile` to point to this tile.

Creation:

1. Create an empty GameObject name it “Tile”.
2. Create a child of the tile name it “model”
3. Create a primitive gameobject(a cube) as a childe of the model.
4. Attach the script to the Tile.

BlockLib Class

Description:

The `BlockLib` class is part of the `BlockPuzzleGameTemplate` namespace. It manages a collection of block types (`BlockType`) and their corresponding `GameObject` prefabs. This class allows for easy retrieval of block prefabs based on the specified block type.

Enums:

- **`BlockType` (enum):**
  - This enum defines different types of blocks that can be used in the game. Each value corresponds to a specific block shape and size.
- **Block Types:**
  - `Square`, `MediumSquare`,
  - `RectangularHorizontal`,
  - `RectangularVertical`,
  - `LargeRectangularVertical`,
  - `LargeRectangularHorizontal`,
  - `T_shaped_Left`,
  - `T_shaped_Right`,
  - `T_shaped_Top`,
  - `T_shaped_Bottom`,
  - `L_shaped_LeftTop`,
  - `L_shaped_RightTop`,
  - `L_shaped_RightBottom`,
  - `L_shaped_LeftBottom`,
  - `Z_Shaped_LeftHorizontal`,
  - `Z_Shaped_RightHorizontal`,
  - `Z_Shaped_RightVertical`,
  - `Z_Shaped_LeftVertical`,
  - `Large_L_shaped_LeftTop`,
  - `Large_L_shaped_RightTop`,
  - `Large_L_shaped_RightBottom`,
  - `Large_L_shaped_LeftBottom`.

Classes:

- **`TheBlock` (class):**
  - A class representing a block with a specific `BlockType` and its associated prefab `BlockPref`.
- **Fields:**
  - `blockType` (BlockType): The type of the block (e.g., Square, T-shaped).
  - `BlockPref` (GameObject): The `GameObject` prefab associated with this block type.

Methods:

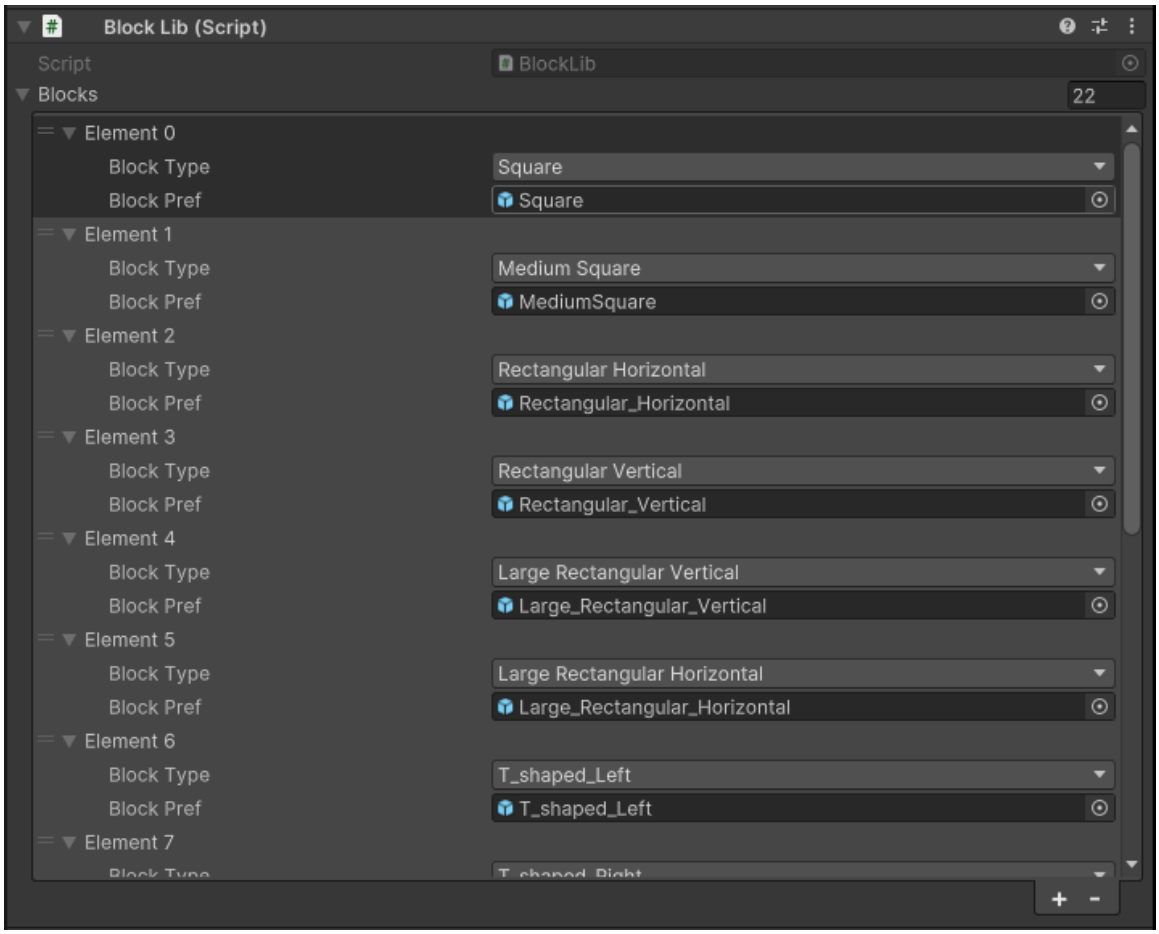
- **`GetBlock(BlockType blockType)` (TheBlock):**
  - **Parameters:**
    - `blockType` (BlockType): The block type to search for.
  - **Returns:**
    - `TheBlock`: The `TheBlock` object is associated with the specified `BlockType`. If no matching block type is found, it returns `null`.
  - **Description:**
    - This method searches the `_blocks` array for the specified `BlockType` and returns the corresponding `TheBlock` object.
- **`GetBlockPref(BlockType blockType)` (GameObject):**
  - **Parameters:**
    - `blockType` (BlockType): The block type for which to retrieve the prefab.
  - **Returns:**
    - `GameObject`: The `GameObject` prefab is associated with the block type. If no matching block is found, it returns `null`.
  - **Description:**
    - This method calls `GetBlock` to retrieve the block associated with the specified type and then returns the `BlockPref` (prefab) of that block. If no block is found, it returns `null`.

Notes:

- The `BlockLib` class maintains an array of `TheBlock` objects, which map each `BlockType` to a corresponding `GameObject` prefab.
- If the `_blocks` array is not populated or the requested `BlockType` is not found, the methods will return `null`.

Usage

1. Create a gameObject name it “Block lib”
2. Attach the BlockLib.cs script
3. On the blocks array on the BlockLib.cs ion the inspector add blocks and specify the enum for the block and attach the appropriate block to the block pref variable in the inspector.



4. To modify and create your block pref check Under BlockClass
5. To modify and add your own block type:
    a. Open the BlockLib.cs
    b. Add the name of your block to the enum
    c. Save the changes
    d. Add to the array

e. Assign the enum and block pref properly

```
public enum BlockType
{
    Square,
    MediumSquare,
    RectangularHorizontal,
    RectangularVertical,
    LargeRectangularVertical,
    LargeRectangularHorizontal,
    T_shaped_Left,
    T_shaped_Right,
    T_shaped_Top,
    T_shaped_Bottom,
    L_shaped_LeftTop,
    L_shaped_RightTop,
    L_shaped_RightBottom,
    L_shaped_LeftBottom,
    Z_Shaped_LeftHorizontal,
    Z_Shaped_RightHorizontal,
    Z_Shaped_RightVertical,
    Z_Shaped_LeftVertical,
    Large_L_shaped_LeftTop,
    Large_L_shaped_RightTop,
    Large_L_shaped_RightBottom,
    Large_L_shaped_LeftBottom,
}
```

ColorLib Class

Description:

The `ColorLib` class is part of the `BlockPuzzleGameTemplate` namespace. It manages a collection of colors and their associated materials (`Material`). This class allows for easy retrieval of materials based on the specified color.

Enums:

- `Colors` **(enum):**
  - Defines various color options used in the game. Each value represents a specific color.
- **Color Options:**
  - `Default`, `Yellow`, `Blue`, `Pink`, `Green`, `Orange`, `Red`, `Purple`.

Classes:

- `TheColor` **(class):**
  - Represents a color with its associated material (`Material`).
- **Fields:**
  - `whichColor` (Colors): The color represented by this object (e.g., Yellow, Blue).
  - `BlockMat` (Material): The `Material` that corresponds to the color.

Methods:

- `GetColorLib(Colors whichColor)` **(TheColor):**
  - **Parameters:**
    - `whichColor` (Colors): The color to search for.
  - **Returns:**
    - `TheColor`: The `TheColor` object that corresponds to the specified color. If no matching color is found, it returns `null`.
  - **Description:**
    - This method searches the `_Colors` array for the specified color and returns the corresponding `TheColor` object.
- `GetMaterial(Colors which)` **(Material):**
  - **Parameters:**

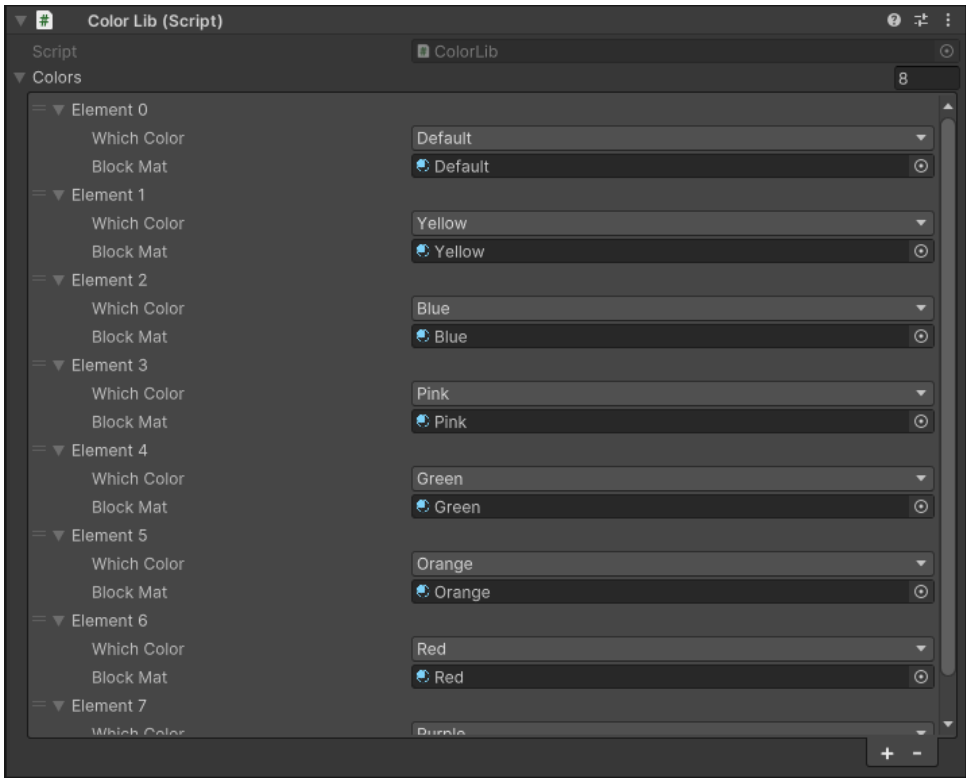
        - `which` (Colors): The color for which to retrieve the material.
    - **Returns:**
        - `Material`: The material (`BlockMat`) associated with the specified color. If no matching color is found, it returns `null`.
    - **Description:**
        - This method searches the `_Colors` array for the specified color and returns the corresponding `Material`. If no matching color is found, it returns `null`.

Notes:

- The `ColorLib` class maintains an array of `TheColor` objects, each containing a `Colors` enum and a corresponding `Material`.
- If the `_Colors` array is not populated or the requested color is not found, the methods will return `null`.
- The block types are premade for you.

Usage:

1. Create a gameObject name it “Colorlib”
2. Attach the Colorlib.cs script
3. On the ColorLib in the inspector add to the list the colors assign the correct enum to each element on the list and also the material.



4. To modify and add your own colors:
    a. Create your own material and name it
    b. Open the colorLib.cs
    c. To the colors enums add your own

```
public enum Colors
{
    Default,
    Yellow,
    Blue,
    Pink,
    Green,
    Orange,
    Red,
    Purple
}
```

d. Save the changes
e. Add to the list and assign all variables appropriately

BlockManager Class

Description:

The `BlockManager` class is responsible for managing the spawning of blocks in the game. It controls the sequential spawning of blocks, assigns colors and types to them, and animates their movement into position using coroutines. The class interacts with `BlockLib` to retrieve block prefabs and `ColorLib` to assign materials to blocks.

Classes:

- **`BlockDetails` (Serializable Class):**
  - Contains the block's type and color.
- **Fields:**
  - `Type_` (BlockType): The type of the block (e.g., Square, T-shape).
  - `Color_` (Colors): The color associated with the block.

Fields:

- **`colorLib_` (ColorLib):**
  - A reference to the `ColorLib` component, used to retrieve block materials by color.
- **`blockLib_` (BlockLib):**
  - A reference to the `BlockLib` component, used to retrieve block prefabs by type.
- **`spawnPos` (Transform):**
  - The position from which blocks will be spawned.
- **`blockTargets` (BlockTarget[]):**
  - An array of `BlockTarget` objects, which are the targets where blocks will be spawned.
- **`blocks` (List<BlockDetails>):**
  - A list of `BlockDetails` objects containing the blocks to be spawned.
- **`isSpawningBlocks` (bool):**
  - A flag that prevents spawning blocks while a spawn sequence is already in progress.

Methods:

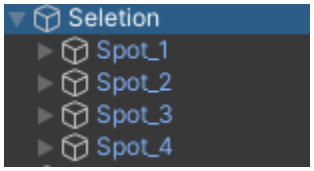
- **`Start()` (void):**
  - Initializes references to the `ColorLib` and `BlockLib` components by finding them in the scene.
- **`Update()` (void):**
  - Checks if blocks are being spawned and triggers the block spawning process if necessary.
- **`CheckAndSpawnBlocksSequentially()` (void):**
  - Calls the `SpawnBlocksSequentially` coroutine to begin spawning blocks.
- **`SpawnBlocksSequentially()` (IEnumerator):**
  - A coroutine that spawns blocks sequentially into the `blockTargets`. It checks if there are available targets and blocks in the list. It spawns blocks, assigns them a type, and then moves them to the target position with animation.
- **`moveToPosition(GameObject block, BlockTarget target)` (IEnumerator):**
  - A coroutine that moves a block to the specified target position over a duration of 0.35 seconds. The movement is animated using `Vector3.Lerp`.
- **`DoPunchEffect(GameObject block, Vector3 targetPos)` (IEnumerator):**
  - A coroutine that applies a punch effect to the block after it reaches its target position. The punch effect is a small oscillating movement to give the block a dynamic feel.
- **`MaterialSwap(GameObject block, Material Mat)` (void):**
  - Changes the material of the block's mesh renderer components to the provided material.

Notes:

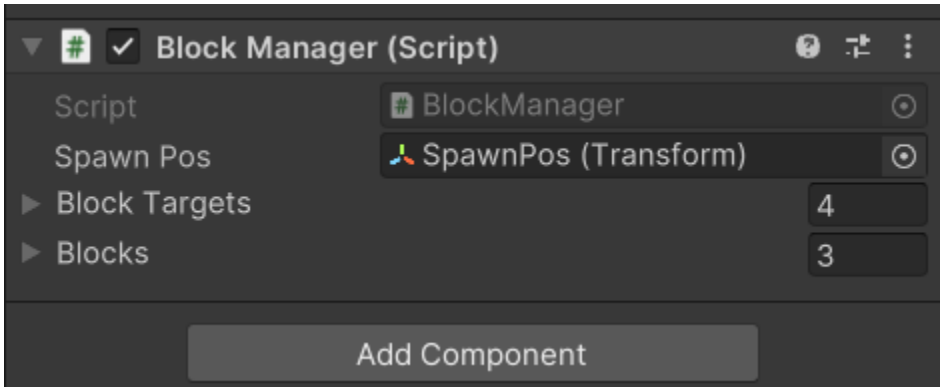
- The `BlockManager` uses the `BlockLib` to retrieve block prefabs and `ColorLib` to assign materials based on the color of each block.
- Blocks are spawned sequentially in available targets, and their movements are smoothly animated using coroutines.
- The `MaterialSwap` method ensures that each block receives the correct color material from the `ColorLib`.
- The `DoPunchEffect` gives a visual feedback after each block reaches its target position.

Usage:

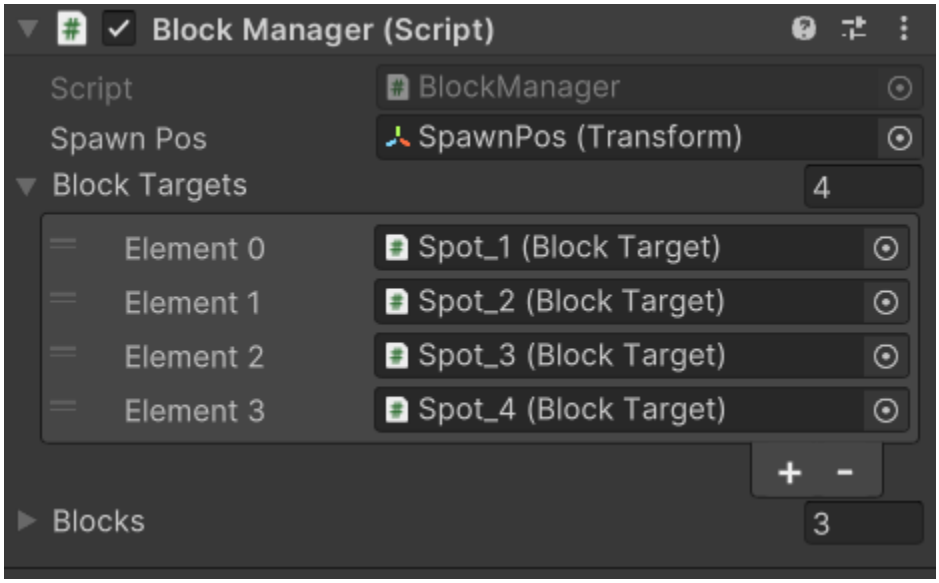
1. Create a game object and attach a BlockManager.cs
2. Create a spawn position
3. Create selection points:
    a. Create a game object
    b. Create children game object(s) and name the according to the number of positions you have i.e “pos”_{the number} Pos_1 e.t.c



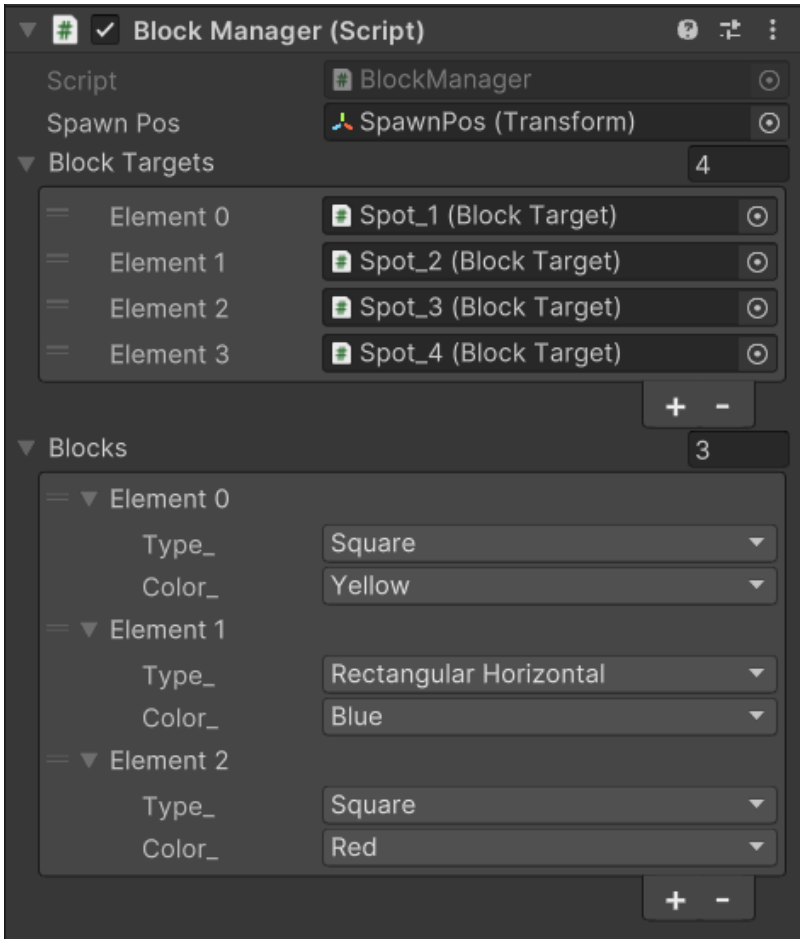
    c. On the Positions Add a script called BlockTarget.
    d. If it is a random pos and not the main positions set the is random Pos to true
4. Attach the spawnposition to the BlockManager



5. Attach the positions from 3.a to the BlockTargets array on the blockManager.cs



6. Set the order you want the blocks to spawn and how many the color and shape on the Bloks List on the blockmanager .cs



Coroutines and Animations:

- **`moveToPosition`**: This coroutine animates the block's movement to its target using linear interpolation (`Vector3.Lerp`).
- **`DoPunchEffect`**: Adds an extra punch effect where the block slightly moves left and right for a more dynamic feel.

FeatureManager Class Documentation

Description:

The `FeatureManager` class controls the enabling and disabling of rotation features in the block puzzle game. It provides flags to control whether clockwise and counterclockwise rotations are enabled. It also contains methods to check if each rotation feature is currently enabled.

Fields:

- **`EnableClockwiseRotation` (bool):**
  - A flag indicating whether clockwise rotation is enabled in the game.
- **`EnableCounterClockwiseRotation` (bool):**
  - A flag indicating whether counterclockwise rotation is enabled in the game.

Methods:

- **`CanRotateClockWise()` (bool):**
  - Returns `true` if `EnableClockwiseRotation` is set to `true`; otherwise, returns `false`.
- **`CanRotateCounterClockWise()` (bool):**
  - Returns `true` if `EnableCounterClockwiseRotation` is set to `true`; otherwise, returns `false`.

Notes:

- To enable rotation features, set `EnableClockwiseRotation` and `EnableCounterClockwiseRotation` to `true` in the `FeatureManager` inspector in the Unity Editor.
- If `EnableClockwiseRotation` is `false`, blocks will not be able to rotate clockwise.
- If `EnableCounterClockwiseRotation` is `false`, blocks will not be able to rotate counterclockwise.

Usage:

1. Create a Game object and attach the featureManager.cs to the game object.
2. Set to true whenever each feature is required

Level Class Documentation

Description:

The `Level` class is responsible for managing the game's level progression, checking if blocks are correctly placed, and handling level completion actions such as confetti effects and wave animations. It also includes logic to check success conditions and handle tile rearrangement.

Fields:

- **`Tiles` (Tile[]):**
  - An array of `Tile` objects that represents the tiles in the level. Each tile can hold a `Block` (through the `Owner` property).
- **`Rearrangedtiles` (List<BlockTile>):**
  - A list that temporarily stores rearranged `BlockTile` objects during the level completion process.
- **`Confetti` (ParticleSystem):**
  - A reference to a `ParticleSystem` for displaying a confetti effect when the level is successfully completed.

Methods:

`Start()`

- Initializes the `Tiles` array by gathering all child `Tile` objects of the current `Level`.

`OnBlockPlacement()`

- Checks if all blocks are correctly placed on the tiles. If successful, it logs a message indicating the level is completed and triggers the wave action.

`CheckSuccess()` (bool):

- Loops through all tiles to check if they have an owner (block). If all tiles are correctly filled, it triggers the success actions and returns `true`. Otherwise, it starts a restart sequence and returns `false`.

`DelayRestart()` (IEnumerator):

- Waits for 3 seconds before potentially restarting the level or performing other actions after a failure.

`WaveAction()`

- Starts the wave action, which is an animation where tiles are rearranged and their blocks interact in a specific sequence.

`ShowWave()` **(IEnumerator):**

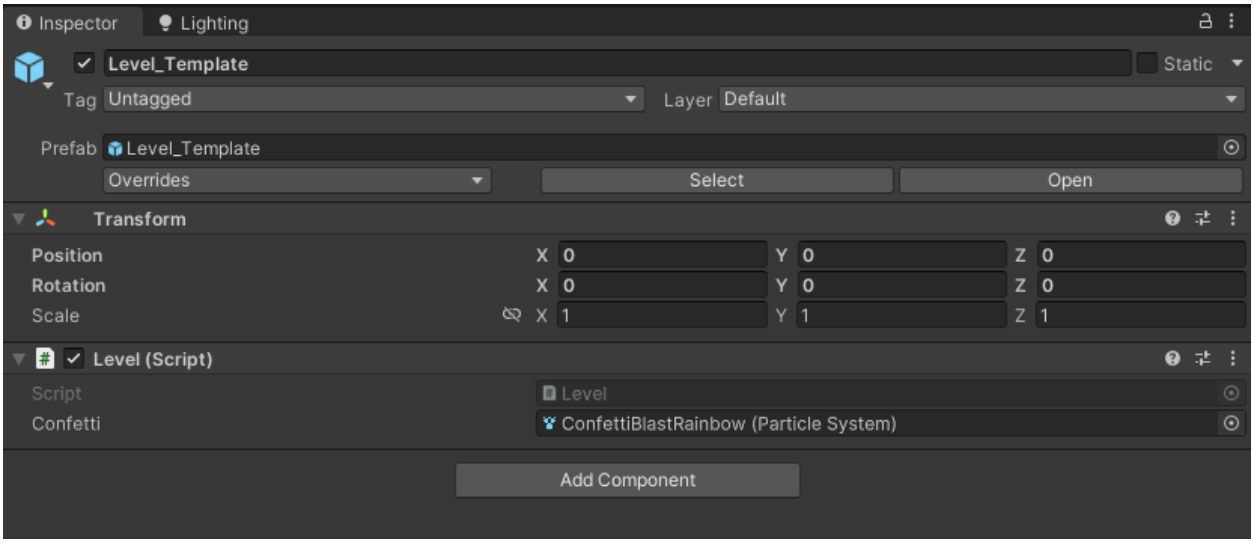
- Rearranges tiles based on the distance between their positions and animates the wave effect by iterating through the `BlockTile` objects.
- Plays a wave animation for each tile and shows the "correct" indication for blocks that were placed correctly.
- Plays the Confetti particle effect when all tiles are processed.

Notes:

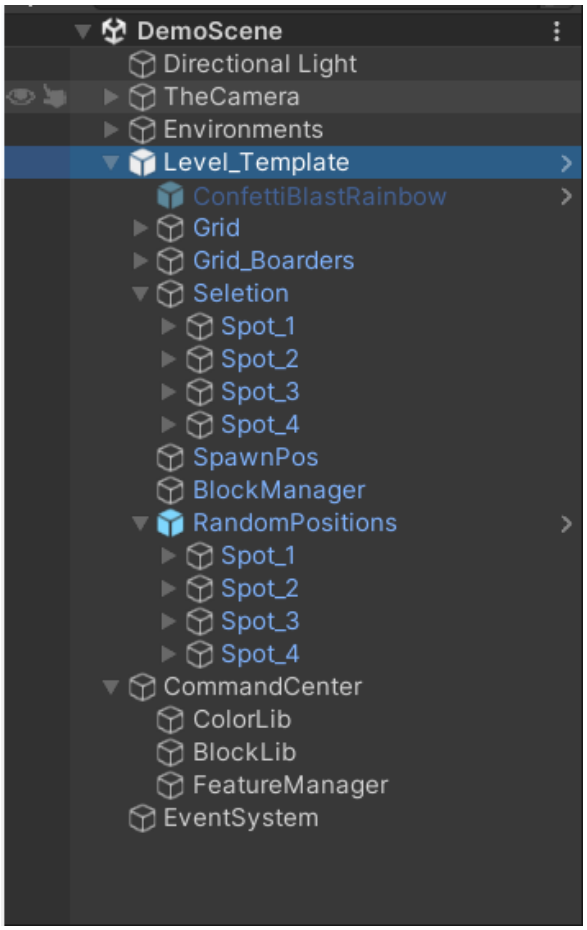
- `CheckSuccess()`: This method is responsible for determining if all blocks are placed correctly. It compares the current state of tiles with the expected correct state.
- `ShowWave()`: The wave animation is triggered when the level is completed successfully. The tiles are rearranged to animate the blocks in a specific order before showing the confetti effect.
- `Confetti`: The confetti effect is played to visually signify the successful completion of the level.
- `DelayRestart()`: The level will wait for 3 seconds after a failure before restarting or performing any additional actions.

Usage:

1. Create a gameobject attach a Level.cs script



2. The level game object should have a number of children namely
    a. Confetti
    b. The grid
    c. The environment
        i. Grid borders
    d. Selection
    e. SpawnPos
    f. BlockManager
    g. RandomPositionHolder



Assign the confetti to the level script.

`BlockTile` Class Documentation

The `BlockTile` class handles the behavior of individual block tiles in the game. It manages user interactions, such as drag, click, and rotation, along with visual feedback like animations and color changes. It also provides functionality to detect collisions with other objects and trigger various actions, including bouncing and rotation.

Fields:

- **FeatureManager_ (FeatureManager)**: A reference to the `FeatureManager` to check if features like clockwise or counterclockwise rotations are unlocked.
- **initialMousePosition (Vector3)**: Stores the initial position of the mouse when the user begins interacting with the block tile.
- **RayPoints (Transform[])**: A set of transforms used for raycasting to detect collisions beneath the block tile.
- **OccupiedTarget (GameObject)**: A reference to the target the block tile is occupying.
- **IsDragging (bool)**: Indicates if the block tile is being dragged by the user.
- **isStatic (bool)**: A flag indicating whether the block tile is static or movable.
- **IsClicked (bool)**: Flag for detecting a click on the block tile.
- **IsDoubleclicked (bool)**: Flag for detecting a double-click on the block tile.
- **ErrorMat (Material)**: Material applied to the block tile when an error is detected.
- **FlipMat (Material)**: Material used for flipping or highlighting the block tile.
- **StartPos (Vector3)**: The starting position of the block tile in local space.
- **UpPos (Vector3)**: The upward position used for some animations (not used in the script).
- **WaveTimestamp (float)**: Timestamp to control the timing of the wave animation.
- **UsedTimestamp (float)**: Timestamp used to throttle the wave action.
- **ClickResetTime (float)**: Time in seconds before a click is reset.
- **mouseDownTime (float)**: Time when the mouse button is pressed.
- **dragThreshold (float)**: The threshold distance the mouse must move to register as a drag.
- **isMouseHeld (bool)**: Flag for holding the mouse down for more than a specified time.
- **holdThreshold (float)**: The time threshold for considering the mouse down as a hold.
- **actionTriggered (bool)**: Flag to track if an action has been triggered after holding the mouse down.
- **StarBurst (GameObject)**: A particle system that is played when certain actions are triggered.
- **LandingSmoke (GameObject)**: A particle system used for landing effects (not utilized in the script).
- **clockwiserotation (bool)**: Flag for controlling the clockwise rotation action.
- **counterclockwiserotation (bool)**: Flag for controlling the counterclockwise rotation action.
- **clickCount (int)**: Tracks the number of clicks performed on the block tile.
- **clickResetDelay (float)**: Delay to reset the click count.

- **clickCoroutine (Coroutine)**: Reference to the running coroutine for handling clicks.

Methods:

Start()

- Initializes the block tile's starting position and obtains a reference to the `FeatureManager`.

OnMouseDown()

- Handles mouse down events.
- If the pointer is not over a UI element, it records the initial mouse position and prepares the block for dragging or clicking.

OnMouseDrag()

- Handles dragging of the block tile.
- Checks if the mouse has moved beyond the `dragThreshold` to start the dragging process.

OnMouseUp()

- Handles mouse release events.
- If the block tile was not dragged, it checks for clicks and starts the `HandleClicks` coroutine to determine if it's a single or double click.
- If the block tile was dragged, it finishes the drag process.

HandleClicks()

- Waits for a brief delay and checks the number of clicks:
    - **Single-click**: Calls the `Clicked()` method.
    - **Double-click**: Calls the `DoubleClicked()` method.
- Resets the click count and coroutine after handling the clicks.

Clicked()

- Called when a single click is detected.
- Checks if clockwise rotation is available via the `FeatureManager`. If available, triggers clockwise rotation.

DoubleClicked()

- Called when a double click is detected.

- Checks if counterclockwise rotation is available via the `FeatureManager`. If available, triggers counterclockwise rotation.

IsPointerOverUIObjectPos(Vector2 ThePos)

- Returns `true` if the pointer is over a UI element at the specified screen position.

Update()

- Called every frame.
- If the mouse is held for a specified amount of time (`holdThreshold`), triggers the action using the `TriggerAction()` method.
- Updates the position and rotation of the block tile during drag or idle state.

TriggerAction()

- Triggers a bounce action on the block, presumably for visual feedback or animation.

Wave()

- Triggers the wave animation for the block tile.
- Adjusts the tile's rotation and position, plays the wave animation, and changes the tile's material to the `FlipMat` material.
- Starts a particle system (starburst) when triggered.

ChangeColor()

- Changes the material of the block tile to `FlipMat` for a brief time during the wave animation.

ShowCorrect()

- Changes the block tile's material to `FlipMat`, signaling correctness (possibly used for visual feedback after completing an action).

GetSingleHit()

- Performs a raycast from the `RayPoints` to detect the first object hit by the ray.

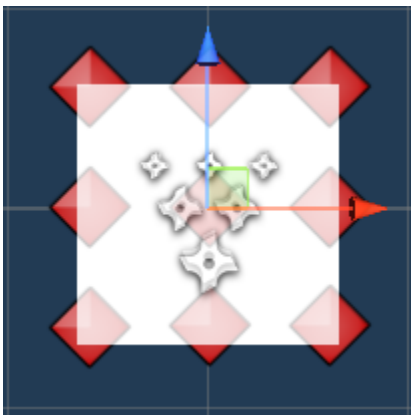
GetHit()

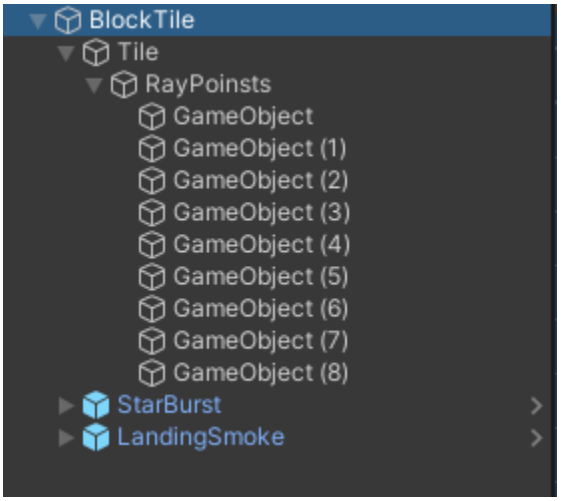
- Performs multiple raycasts from each `RayPoint` and returns a list of all objects hit by the rays.

Usage:

The `BlockTile` class is intended for use in a game where tiles or blocks are manipulated by the player. This script allows for the rotation of blocks via clicks and double-clicks, as well as drag-and-drop interactions. The `FeatureManager` is used to control whether specific actions like rotation are allowed. The `Wave()` and `ShowCorrect()` methods provide visual feedback when the block tile is part of a successful action.

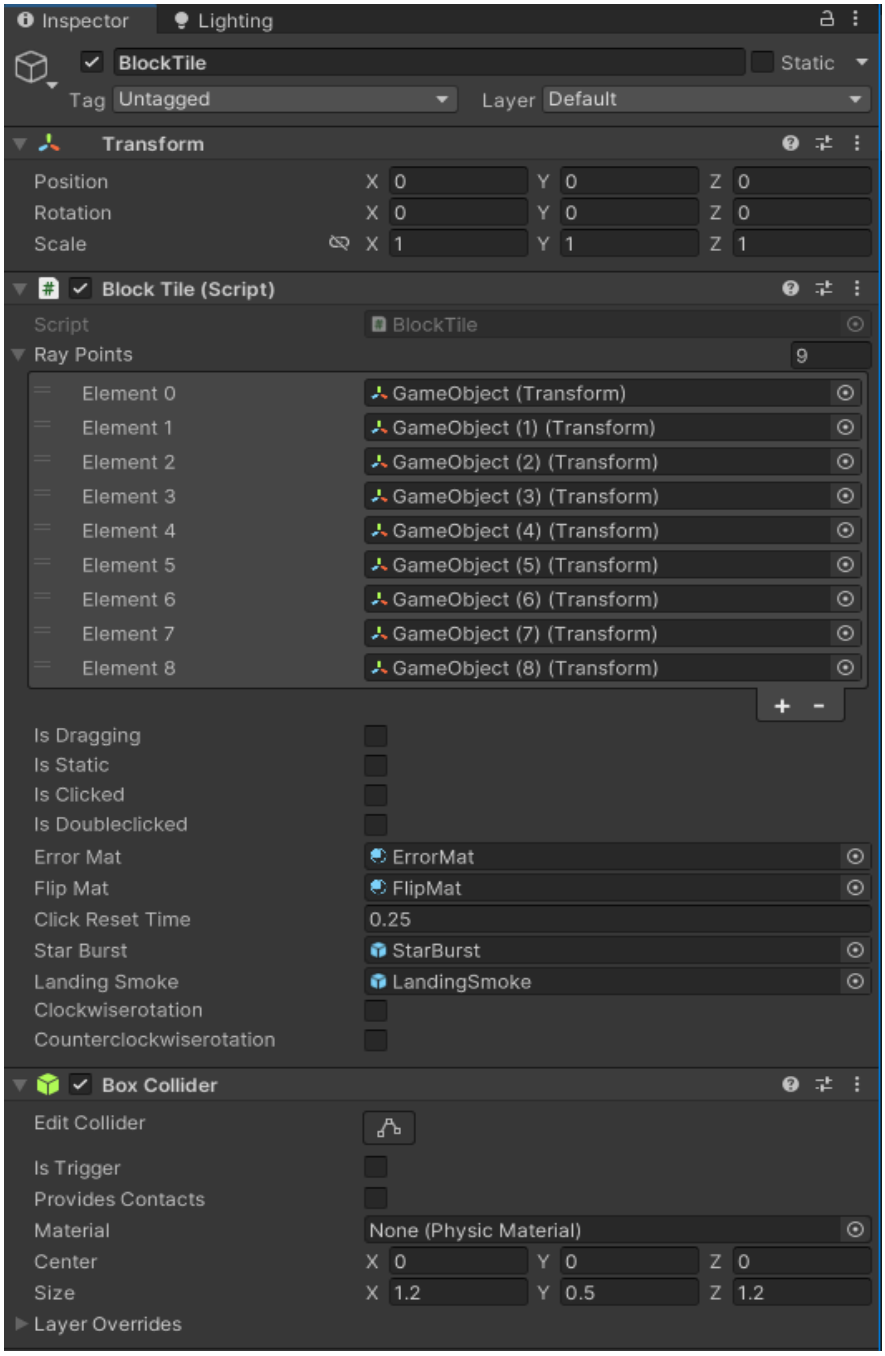
1. The block Tile is premade .
2. You can create your own by
    a. Create a gameObject and name it “BlockTile” keep the vector scale to VectorOne(1,1,1).
    b. Create a cube as a child of the block tile name it “Tile” and set the vector scale to (1.2,0.5,1.2)
    c. Create a child of the Tile and name it Raypoints keep the vector scale to VectorOne(1,1,1).
    d. For the ray points game object create children and place them on the
        i. Four corners
        ii. The centers
        iii. Between the four corners on the edge at the center





iv. Under the blockTile add the Startburst particle and landing Smoke particle system.

3. On the BlockTile Game Object
    a. add a BoxCollider
    b. Attach the BlockTile.cs
    c. On the raypoints array add the raypoints
    d. Add the error material,the flip material
    e. The clickResetTime is preset
    f. Attach the starburst
    g. Attach the LandingSmoke



Block Class

The `Block` script is responsible for managing the behavior of blocks in the puzzle game. It includes functionality for block movement, rotation, error handling, placement, and other interactions. The block can be dragged, rotated, and placed on a grid. If the block placement is incorrect, it will provide visual feedback.

Fields

- **`Level level`**
  Reference to the level object containing the grid and the block's target positions.
- **`Transform PrevPos`**
  Stores the previous position of the block before dragging or placement.
- **`bool IsDrag`**
  Indicates if the block is currently being dragged.
- **`bool IsPlaced`**
  Indicates if the block is successfully placed on the grid.
- **`BlockTile[] blockTiles`**
  Array of `BlockTile` objects that are part of the block.
- **`bool IsClicked`**
  Tracks if the block has been clicked during interaction.
- **`BlockType type`**
  Specifies the type of block (e.g., block's shape or color).
- **`GameObject marker`**
  A visual indicator for the block's position or state (e.g., during dragging or placement).
- **`bool IsStatic`**
  Determines if the block is static and cannot be moved or rotated.
- **`bool IsOnGrid`**
  Indicates if the block is currently placed on the grid.

Methods

Start()

Initializes the block's properties:

- Sets the `PrevPos` to the block's parent target position.
- Collects the block's child `BlockTile` components.
- References the `Level` object.

- Creates a visual marker for the block.

SetUp()

- Initializes the block's visual marker as a small sphere.
- The marker is parented to the `Level` object and has its scale adjusted.
- Disables the marker's collider and renderer for the time being.

Update()

Checks if the block is clicked using the `CheckClick` method and sets `IsClicked` accordingly.

Rotate(bool isClockwise)

Rotates the block either clockwise or counterclockwise:

- Starts a coroutine for rotation, depending on the direction.

IEnumerator Rotation(bool isClockwise)

Handles the rotation of the block over time:

- Calculates the start and end rotations.
- Interpolates between the two rotations using `Quaternion.Slerp`.

Drag(Vector3 Offset)

Handles dragging of the block:

- Disables block tile colliders to prevent interaction while dragging.
- Uses a raycast to move the block based on mouse position.
- Clears any tile ownership and updates the block’s position.

Bounce()

Starts the bounce effect for the block, which increases and decreases the scale over time.

IEnumerator BounceCoroutine()

Handles the bounce animation for the block:

- Scales up the block to simulate a bounce, then scales it back to its original size.

FinishedDrag()

Called when the block finishes dragging:

- Re-enables colliders on the block tiles.
- Checks if the block placement is correct using `CheckCorrect()`.
- If the placement is correct, marks the block as placed and updates its position.
- If the placement is incorrect, it clears the block's tile occupation and provides error feedback.

UnParent()

Unparents the block from any existing `BlockTarget` objects, ensuring it is no longer associated with previous targets.

IEnumerator ErrorShow(bool ShowError)

Handles visual feedback when the block placement is incorrect:

- Changes the block's materials to indicate an error.
- Initiates the shake effect for the block.
- Returns the block to its previous position or a random position based on the `ShowError` flag.

ShakeRotation()

Starts the shake rotation animation for the block, providing feedback on an incorrect placement.

IEnumerator ShakeRotationCoroutine()

Executes the shake rotation effect:

- Randomly applies small rotational offsets to simulate a shake.
- Gradually reduces the magnitude of the shake if the `fadeOut` flag is set.

CheckIfFromGrid()

Checks if the block's tiles are already placed on the grid by verifying if each tile has a corresponding `BlockTarget` owner.

IEnumerator ReturnToPosition()

Handles the return of the block to its previous position:

- Smoothly interpolates the block's position from its current location to `PrevPos`.
- After reaching the target position, the block is scaled down and marked as not on the grid.

IEnumerator ReturnToRandomPos(Vector3 offset)

Returns the block to a random position:

- The block is moved to a random target position, using interpolation over time.

Pos()

Selects a random position for the block to return to:

- Filters available `BlockTarget` objects based on whether they are occupied or random.
- Returns a suitable target position for the block.

Helper Methods

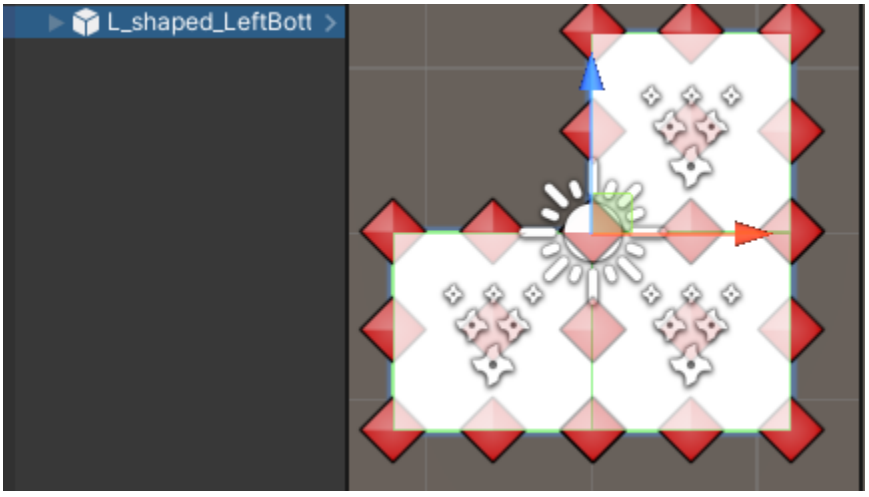
- `CheckClick()`
  Determines if the block was clicked using input checks. This can be implemented based on the project's input management system.
- `CheckCorrect()`
  Determines if the block's current position and rotation are correct (i.e., the block can be placed successfully on the grid).

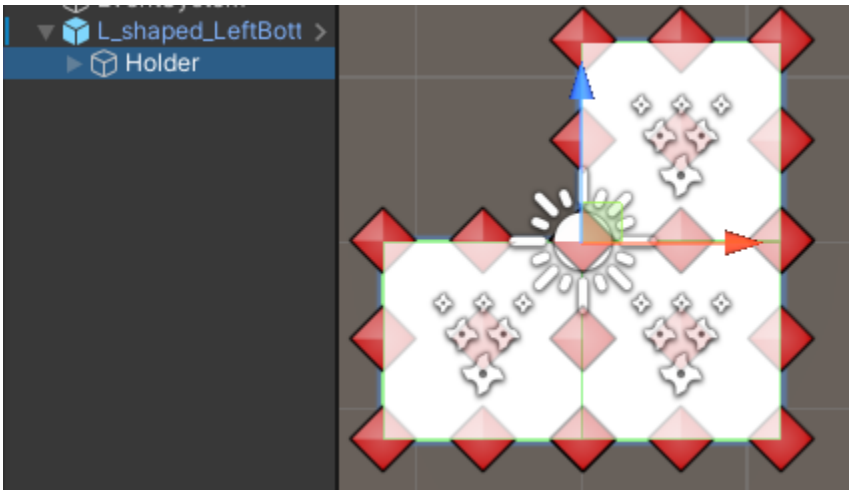
Error Handling

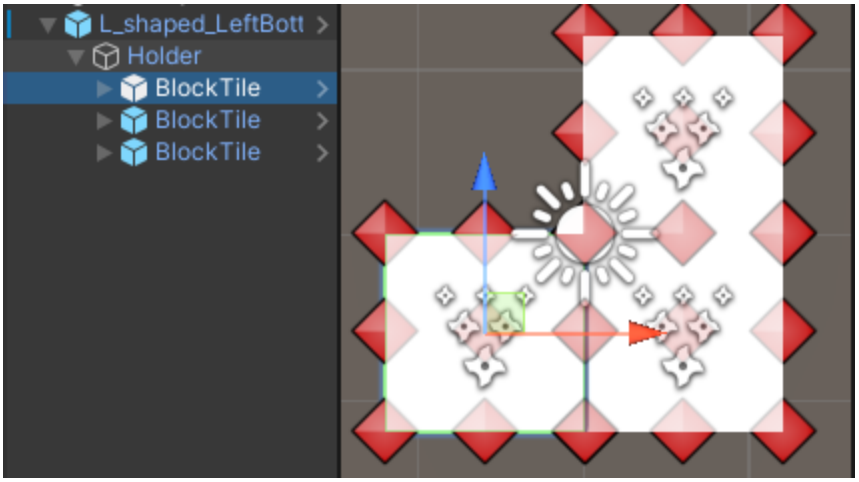
- **Error Materials**: If a block is placed incorrectly, its material is changed to indicate an error.
- **Shake Effect**: A shake effect is applied to visually highlight the error.
- **Repositioning**: After an incorrect placement, the block is returned to its previous or random position for retrying.

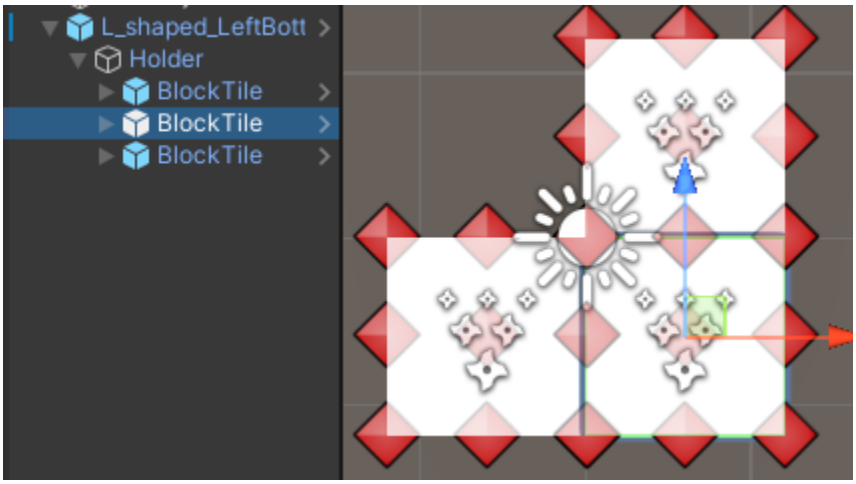
Usage

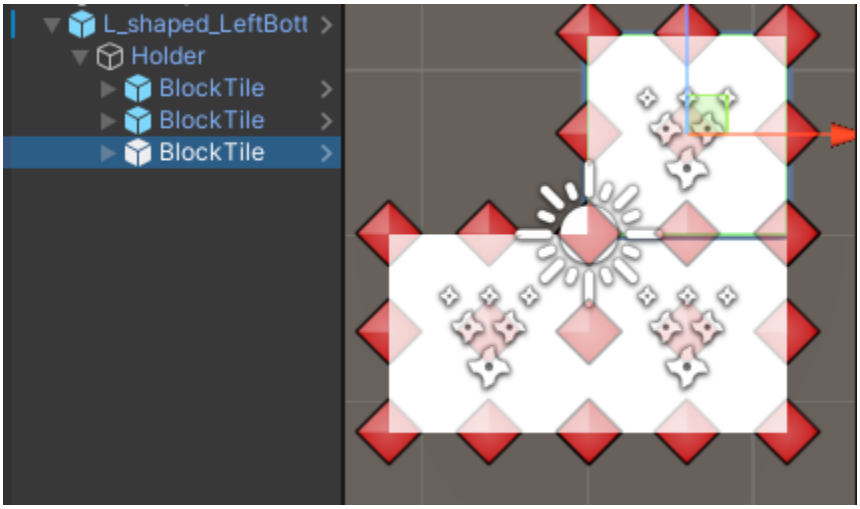
- Attach this script to block objects that the player interacts with.
- The script is designed to work within a puzzle grid where blocks can be dragged, rotated, and placed.
- The block types have been premade reference them if you want to create your own.
    - To create your own:
        - Create a gameobject name it however you wish and this is the name you add to the block lib cs enum .ie if you name it L_shaped_leftbottom.On the eneum you add the same name.
        - Create a child of the L_shaped_leftbottom name it holder .
        - Add the blockTiles ad form an L shape.
        - You can move the Blocktiles however you want but make sure the center of the block tiles is at zero.Meaning the “Holder” and the L_shaped_leftbottom are at the center


L_shaped_LeftBott


L_shaped_LeftBott
Holder







On the “L_shaped_LeftBottom” attach Block.cs .The block tiles array will populate itself on start.