Kieran Abelen

Mell1Games

Unity Methodology 2

Contents

[Introduction 2](#_Toc55141155)

[UI Development 3](#_Toc55141156)

[References 4](#_Toc55141157)

Unity

# Introduction

## Brief

This second methodology will document my learning when it comes to deploying and production a game to be posted on a website. The game is in a complete state but needs much more polish and testing to be ready for deployment.

Like before many things will be changed in the introduction as I learn new information and end up adding it to the overall methodology. Each heading under the instruction will try an summarize my goals as they evolve with each section.

Static levels should be the initial priority, but the following methodology should delve into procedural generation of levels.

# UI Development

## Brief

This section will entail Completion of the UI elements and menus. There will be three headings, one will be the wireframes that where the result of me using unity to make the menus. The second will be the testing of these UI elements, and the final third will be the look and color of all the screens.

## Wireframe

These diagrams all use a star as the anchor point for the image, so all positioning is based of the relation to that centred anchor.

The screen size is based of 800 by 600, so the menus all take about half the screen in total with exception of map selection. The screen will auto resize based on the platform either scaling up or down my initial screen size.

### Main Menu



### Select Level



### Popup Menu



## UI Testing

I only need to test the ability to move to each page at this point since I have not hosted the game yet.

### UI Functionality Test Case

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Case | Goal | Steps | Expected Result | Result |
| 1 | Main transitions | Button Start | Level started | Completed |
|  |  | Button Select Level | Transition and levels populated | Completed |
|  |  |  |  |  |
| 2 | Select Level transitions | Button Level 1 | Level started | Completed |
|  |  | Button Back | Transition to Main Menu | Completed |
|  |  |  |  |  |
| 3 | Popup transitions | Button Next | Transition to next level | Not Complete |
|  |  | Button Main Menu | Transition to Main Menu | Not Complete |
|  |  | Button Select Level | Transition and levels populated | Not Complete |
|  |  |  |  |  |
| 4 | Game transitions | Game completed | Transition to Popup | Completed |
|  |  | Pause game | Transition to Popup (no next level button) | Not Complete |

### Popup transitions Fix

The main issue is the game screen does hide itself once started; this has led to hiding the underlying menu with a white screen. I need the ability to turn off the game screen and, in the process, resetting it to a before launched screen.

#### Solution

It turns out if you have screen being turn off outside the main game loop it won’t work, so the solution is placing it in levelUpdate function. By turning off the mapClick game object here it will unload the game.

if (level.tileSelected.SequenceEqual(level.tileCorrect))

{

// Debug.Log("Won");

levelActive\_state = false;

UIMenuPopup.gameObject.SetActive(true);

GameTilemap\_clickable.gameObject.SetActive(false);

UIButtonGametoMain.gameObject.SetActive(false);

}

### Game transitions (pause game) Fix

This one is a little trickier since the real issue it with the game states. We have an active game state which will let the update have access to the LevelUpdate function but what we need a way to exit like when completing a level.

#### Solution

I need another state for the game itself and not the level (loaded or not). The pause game will just provide exit from the LevelUpdate function while it is running. It will be a hard exit so closing the game Tilemap but it needs to directly tied to a state so it can be changed with a button.

if (level.tileSelected.SequenceEqual(level.tileCorrect))

{

// Debug.Log("Won");

levelActive\_state = false;

UIMenuPopup.gameObject.SetActive(true);

GameTilemap\_clickable.gameObject.SetActive(false);

UIButtonGametoMain.gameObject.SetActive(false);

}

else if (gameActive\_state == false)

{

levelActive\_state = false;

UIMenuMain.gameObject.SetActive(true);

GameTilemap\_clickable.gameObject.SetActive(false);

UIButtonGametoMain.gameObject.SetActive(false);

}

Then a button can just flick the state as if it completed the level.

public void StateGame\_pause()

{

gameActive\_state = false;

}

# New Levels and Fixes

## Brief

This section will detail the first steps that will be taken to make at least 10 levels that can be in published version. These need to develop inside separate json files and be correctly format, they will number from map one to ten.

Each one should be play tested to see if it can be completed but this will be done as I record as bugs occur.

Any updates should also be listed under the table and can be from anywhere during implementation of the new levels.

## Json Assets

The json files need to convert to text assets until I can place them on a database or remove them entirely to replace with random generation. Even then segments of random generation need to be assets since I want symmetrical patterns to for the tile types and correct tiles.

## Levels

All these levels are upside down since the program reads files this way.

|  |  |  |
| --- | --- | --- |
| Level | Tile Type | Tile Correct |
| 1 | 1,0,0,0,  1,0,0,0,  1,0,0,0,  0,1,1,1 | 0,0,0,0,  0,0,1,0,  0,0,0,0,  0,0,0,0 |
| 2 | 0,2,2,2,  0,2,2,2,  0,2,2,2,  0,0,0,0 | 0,0,1,1,  0,0,1,1,  0,0,0,0,  0,0,0,0 |
| 3 | 1,2,0,2,  0,0,2,0,  1,2,0,2,  0,1,0,1 | 0,0,1,0,  0,1,1,1,  0,0,1,0,  0,0,0,0 |
| 4 | 1,2,0,2,  1,2,2,2,  1,0,2,0,  0,1,1,1 | 0,1,0,1,  0,0,1,0,  0,1,0,1,  0,0,0,0 |
| 5 | 0,2,0,0,2,  1,0,2,2,0,  1,0,2,2,0,  0,2,0,0,2,  0,0,1,1,0 | 0,1,1,1,1,  0,1,0,0,1,  0,1,0,0,1,  0,1,1,1,1,  0,0,0,0,0 |
| 6 | 1,0,0,0,0,  1,2,2,2,2,  1,2,2,2,2,  1,0,0,0,0,  0,0,0,0,0 | 0,1,0,1,0,  0,1,0,1,0,  0,1,0,1,0,  0,1,0,1,0,  0,0,0,0,0 |
| 7 | 0,2,2,2,2,  1,0,2,2,0,  1,0,2,2,0,  0,2,2,2,2,  0,1,0,0,1 | 0,1,1,1,1,  0,0,0,1,0,  0,0,1,0,0,  0,1,1,1,1,  0,0,0,0,0 |
| 8 | 1,2,0,0,2,  0,2,2,2,2,  0,2,2,2,2,  1,2,0,0,2,  0,0,0,0,0 | 0,1,0,0,1,  0,1,1,1,1,  0,1,1,1,1,  0,0,1,1,0,  0,0,0,0,0 |
| 9 | 1,0,0,0,0,  0,2,2,2,2,  1,2,2,2,2,  1,2,0,0,2,  0,0,1,1,0 | 0,1,0,0,1,  0,1,1,1,1,  0,1,0,0,1,  0,1,0,0,1,  0,0,0,0,0 |
| 10 | 1,0,0,0,2,  1,0,2,2,0,  1,0,2,2,0,  1,2,0,0,0,  0,1,1,1,1 | 0,1,0,0,1,  0,0,1,0,0,  0,0,0,1,0,  0,1,0,0,1,  0,0,0,0,0 |

## Fixes

### Select Level

This was not implemented correctly as well as many other level load entries. This needed a rework to include the current level and correct encapsulation of variables for each loop.

for(int i = 0; i<maps.Length; i++)

{

// Encapsulate the level number to break dependency

int l = i;

if (maps[i].Extension.Contains("json"))

{

Debug.Log(l);

GameObject goButton = Instantiate(UIButtonSelect, UIMenuSelectPanel);

Button btn = goButton.GetComponent<Button>();

btn.GetComponentInChildren<TextMeshProUGUI>().text = l.ToString();

btn.onClick.AddListener(delegate { levelNumber\_current = l; LevelLoad(l); });

}

}

### In-game menu

The in-game menu button was selecting a tile too, this was because there were two if statements in the level update function. This meant that a tile selection could fire along with a menu selection.  
To break this asynchronous action, I added the menu if statement under the mouse click if statement. The level completion still needs to be asynchronous so it can check after the mouse click even without needing mouse click.

if (level.tileSelected.SequenceEqual(level.tileCorrect))

{

// Reset the map local save back to 0

for (int i = 0; i < level.height \* level.width; i++)

{

level.tileSelected[i] = 0;

}

string json = JsonUtility.ToJson(level);

File.WriteAllText(maps[levelNumber\_current].ToString(), json);

// Unload level and show popup menu

levelState = false;

UIMenuPopup.gameObject.SetActive(true);

GameTilemap\_clickable.gameObject.SetActive(false);

UIButtonGametoMain.gameObject.SetActive(false);

}

// Is the mouse button down and not up

if (Input.GetMouseButtonDown(0) && !Input.GetMouseButtonUp(0) || !gameState)

{

// check to see if game is paused

if (!gameState)

{

levelState = false;

UIMenuMain.gameObject.SetActive(true);

GameTilemap\_clickable.gameObject.SetActive(false);

UIButtonGametoMain.gameObject.SetActive(false);

}

else

{

string json = JsonUtility.ToJson(level);

File.WriteAllText(maps[levelNumber\_current].ToString(), json);

}

### Tile map Inconsistencies

There was an issue with Tilemaps with odd widths and heights shifting because of float rounding issues, this caused multiple issue and 2 fixes needed to take place to correct the tiles.

First, I needed to extend the border so they could access the odd Tilemap size.

// A number to remove the top and left border of the grid from tile selection

int borderY = 1;

int borderX = 1;

if (level.height % 2 == 1)

{

borderY = (int)0.5;

borderX = (int)1.5;

}

// Make sure it's in the game screen but also exclude the left column and top row from selection, it is adjusted centering it like a graph

if (tilePos.x >= level.width / 2 + borderX || tilePos.y >= level.height / 2 - borderY || tilePos.x < -level.width / 2 + borderX || tilePos.y < -level.height / 2)

{

// An unreachable tile

tilePos = new Vector3Int(0, 0, -1);

}

Second is recentring the Tilemap due to it not being able to use 0.5 lengths, the solutions is moving the grid that Tilemap is the child of.

UIGrid.localPosition = new Vector3(-400, -320, 0);

if (level.width % 2 == 1)

{

UIGrid.localPosition = new Vector3(-400.5f, -320.5f, 0);

}

### Camera and menu scaling

There are many issues around my game screen being far too small or too big. This is due to the screen scaling based on ideal pixel numbers, this becomes more an issue when you have different aspect ratios. A normal game menu will become very small on a portraited phone due to it scaling to the nearest 1000 pixels, in this case the small width of the phone.

CameraMain.orthographicSize = (float)level.height \* 5 / 6;

Debug.Log(ScreenMain.GetComponent<RectTransform>().rect.width);

Debug.Log(ScreenMain.GetComponent<RectTransform>().rect.height);

UIGrid.localPosition = new Vector3(-ScreenMain.GetComponent<RectTransform>().rect.width / 2, -ScreenMain.GetComponent<RectTransform>().rect.height / 2, 0);

if (level.width % 2 == 1)

{

UIGrid.localPosition = new Vector3(-ScreenMain.GetComponent<RectTransform>().rect.width / 2 - 0.5f, -ScreenMain.GetComponent<RectTransform>().rect.height / 2 - 0.5f, 0);

}

## Selected tile error

The out of bounds return of 0,0,-1 was triggering the tile 0,0,0 to be selected so I added an AND to make sure it wasn’t used.

// Set color to red if tile is white else color is white

if (GameTilemap\_clickable.GetColor(tileVector\_int) == Color.white && tileVector\_int.z == 0)

{

SetTileColour(Color.red, tileVector\_int);

level.tileSelected[LevelArrayIndex\_create(tileVector\_int)] = 1;

// Debug.Log(tileVector\_int);

}

else

{

SetTileColour(Color.white, tileVector\_int);

level.tileSelected[LevelArrayIndex\_create(tileVector\_int)] = 0;

// Debug.Log(tileVector\_int);

}

# Tile Theme

## Brief

This section will detail the tile theme and general UI improvements. The new look will have distinct colours, so it doesn’t confuse the user. Then the colours will then have a bloom effect to give it a more techno retro look. The entire game will have a black background a neon glow.

The final improvement is some general UI improvements to help the user figure out the game and the correct way to play it.

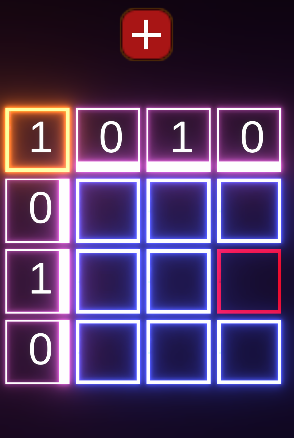
The game will be formally named Dead Pixel so each tile should represent a potential dead pixel. The number act as an overlay provided by the game narrator or avatar.

Several Tilemaps need to be added eventually but for now the Clickable and Text will be the priority.

1. Clickable Tile
   1. Should be left white for the logic
   2. Will override the neon colour but not the base sprite image(black box)
2. Sprite
   1. Each pixel should flicker but at random to convey a potential dead pixel
   2. The border will have a bloom effect applied
3. Text
   1. Coloured and transparent
   2. Some hovering effect applied
4. Grid menu
   1. This will hold the Dividing line
   2. This will be a while glowing line that indicates where the mirror will be
   3. It will be inside a menu tile and clicking it could place it over top of the game.
5. Avatar
   1. The narrator of the game
   2. Should move between different tiles and in the story is creating the overlay.

(The main reason for this change is due to lighting, since it will mess up the Tile GameObject and Sprite order.)

## Bloom effect

Through some UX testing I found the correct colours for the level. I need the colours distinct and easy to read. The colours will be yellow, blue, purple and red. 

## Improved UI

A few things needed to be added to help with UX. Feedback from early testing found that the game was too hard to understand without thorough explanation. I feel even with a tutorial it would help to have some handy UI to explain you should be doing or aiming for.

The two additions are not only helpful for the user without increasing the difficulty but also eliminate potential bugs that could occur.

1. Tile type 3 (counter)
   1. This will offer a countdown of the correct tiles minus the selected tiles. It will be placed in the top left corner
2. Menu Panel (mirror)
   1. These will be selectable images that display some relevant information
   2. A selected image will provide and overlay to clarify to the user what the image is conveying.

## Mirror Logic added

Through some of the play testing I realized the most fun version of the game involved a mirror mechanic. This created a pattern the will would reverse on the other side of the mirror, the lead to more active thought when solving the puzzle and created a faster and harder solutions. This will lead to each player having their own ways to solve the puzzle.

This code got added to each map indicating what mirror to use.

"mirror":1,

This got added to the main script and will set the correct sprite based on the level mirror value.

UIImage.sprite = null;

//Debug.Log(level.mirror);

if (level.mirror == 1)

{

UIImage.sprite = mirror1;

}

else if(level.mirror == 2)

{

UIImage.sprite = mirror2;

}

else if(level.mirror == 3)

{

UIImage.sprite = mirror3;

}

else if(level.mirror == 4)

{

UIImage.sprite = mirror4;

}

# Random Generation of Levels

## Brief

For this section I will lay out randomly generate each puzzle level. I want to keep all the legacy code so a state will control whether I load a random or static puzzle. The random generation will have a set priority structure based on the difficulty. The static level json file will now store the potential solutions rather than a static one, a potential puzzle will be randomly selected, and mirrors based on the json file.

Random generation logic

1. Random Mirror selected.
2. Random tiles within the mirror selected.
3. Mirror multiplied to fill level.
4. Restrict Type 1 numbers for 0 and max width or height.
5. Restrict Type 2 numbers for 0 and 9.
6. Restrict Number of correct tiles for 0 and max width X max height.

Each potential selection should be inside the level json file as a setting for that difficulty. This will help set up multiple level types.

## Divide MainGame Script

This is not necessary but will help any further implementation especially around reusing classes. Right now, the level start class needs to hold the static loading and random generation. So rather than having a class inside a class in the main game I decided to break it so I can further segment it later.

* MainGame
  + LevelStart
    - LevelRandom
    - LevelLoad
  + LevelUpdate
    - LevelUpdate

I will break the level off into its own script and have a separate start and update but that will be implemented later. Level Random will reusing LevelLoad by sending it the level object.

## Difficulty level json

The new level storage will set the parameters for the level generation either the options available or the restrictions on number selection. Further down the line I could try and add reductions in numbers.

The random level will need to generate a static level. This will then follow the normal level loading and updating. This means there will always be a static level, the random level could even have saves.

|  |  |
| --- | --- |
| Static | Random |
| {"level":"static",1  "width":4,"height":4,  "tileType":[],  "tileCorrect":[],  "tileSelected":[],  "mirror":1,  "name":"Tutorial"} | {"level":"random",1  "width":4,"height":4,2  "mirrorOptions":[1],  “type1RestrictX”:[0,4],  “type1RestrictY”:[0,4],  “type2Restict”:[0,9],  “tileCorrectRestrict”:[0,16],3  "name":"Tutorial" 4} |

(1) The first thing that needs to be main clear is whether the level is static or random since I will need to execute the correct methodology to generate the map.

(2) The random generation should never include more than none map size since random generation errors will be map size specific.

(3) The 16 in this example will be divided by the respected mirror quadrants. In this case 4 since mirror 1 has four quadrants, mirror 2 has two quadrants so it would be 8.

(4) If there are any random generations that create errors, there will need to be restrictions on that generation. This could be done either by outright displaying the error pattern or some other restriction method.

### Restrictions copied to Static

The type 1 and 2 restrictions will be placed in the static level, so I can reuse the logic that generates the numbers. If I did not, I would have to copy the code into the random level generation or create a common function.

## Random level logic

If I want to reuse the static level logic, I set up a generation function in the level start class to produce a static level to be passed into the level load function. That would mean a static level can either be hard coded or generate so the logic is reused.

|  |  |
| --- | --- |
| LevelGenerate(parameter) | |
| Steps | Description |
| 1 | Random level object parameter |
| 2 | Create static level object |
| 3 | Define a Tile pattern depending on the mirror set |
| 4 | Define a Random number and convert it to binary |
| 5 | Set the Tile pattern index to the random number binary |
| 6\* | Set the static Tile correct |
| 7\* | Set the static Tile type |
| 8 | Load the static Level |

\*: Expanded below the table.

### Step 6: Set Static Tile Correct

This will outline mirror 1 since that should provide coverage for all other mirrors.  
The main logic(reason this works) is due to moving the anchor of tile x=0 and y=0 to other corners of the puzzle. Doing so and reversing the X or Y means a X=1 would be X=-1 in the bottom right quadrant.

|  |  |
| --- | --- |
| Steps | Description |
| 1 | Mirror 1 |
| 2 | Loop the Y and X of the tile pattern |
| 3 | If a tile pattern is 1 |
| 4 | Set Tile correct for each:   * Bottom left quadrant is X positive and Y positive. * Bottom right quadrant is X negative and Y positive. * Top left quadrant is X positive and Y negative. * Top right quadrant is X negative and Y negative. |

### Step 7: Set Static Tile Type

This only needs to correctly set each type of number.

|  |  |
| --- | --- |
| Steps | Description |
| 1 | Loop the Y and X of the tile pattern |
| 2 | Type 1 for X == 0 or Y == max length |
| 3 | Type 3 for X ==0 and T == max length |
| 4 | Type 2 for everything else |

## Result

The Level generation will randomly create a tile pattern them duplicate it based on the mirror type.

// Randomly generate a new active level

public void LevelGenerate(MainGame.LevelRandom levelRandom)

{

// An instantiated static level

MainGame.LevelStatic levelStatic = new MainGame.LevelStatic

{

level = "static",

height = levelRandom.height,

width = levelRandom.width,

tileType = new int[levelRandom.height \* levelRandom.width],

tileCorrect = new int[levelRandom.height \* levelRandom.width],

tileSelected = new int[levelRandom.height \* levelRandom.width],

type1RestrictX = levelRandom.type1RestrictX,

type1RestrictY = levelRandom.type1RestrictY,

type2Restict = levelRandom.type2Restict,

mirror = 1,

name = "Beginner"

};

// Create a Tile Pattern array and random binary number

int[] tilePattern = new int[levelRandom.height \* levelRandom.width];

int randomNumber = UnityEngine.Random.Range(1, (int)Math.Pow(2, Mathf.Floor(levelRandom.height / 2) \* Mathf.Floor(levelRandom.width / 2)) - 2);

String randomPattern = Convert.ToString(randomNumber, 2);

// Set the Tile Pattern with the random number Pattern

for (int i = 0; i < randomPattern.Length; i++)

{

tilePattern[i] = Convert.ToInt32(randomPattern.ToCharArray()[i]) - 48;

//Debug.Log(tilePattern[i]);

}

// Mirror 1 Pattern duplication

if (levelStatic.mirror == 1)

{

for (int y = 0; y < Mathf.Floor(levelRandom.height / 2); y++)

{

for (int x = 0; x < Mathf.Floor(levelRandom.width / 2); x++)

{

int levelActiveCell = (int)(y \* Mathf.Floor(levelRandom.width / 2) + x);

// 4 Quadrant duplication

if (tilePattern[levelActiveCell] == 1)

{

// Bottom left

levelActiveCell = y \* levelRandom.width + x + 1;

levelStatic.tileCorrect[levelActiveCell] = 1;

// Bottom right

levelActiveCell = y \* levelRandom.width - x + levelRandom.width - 1;

levelStatic.tileCorrect[levelActiveCell] = 1;

// Top left

levelActiveCell = (-(1 + y - levelRandom.height) \* levelRandom.width - levelRandom.height + x + 1);

levelStatic.tileCorrect[levelActiveCell] = 1;

// Top right

levelActiveCell = (-1 + levelRandom.height \* levelRandom.width) - ((y + 1) \* levelRandom.width + x);

levelStatic.tileCorrect[levelActiveCell] = 1;

}

}

}

}

// Set grid tile type

for (int y = 0; y < levelStatic.height; y++)

{

for (int x = 0; x < levelStatic.width; x++)

{

int activeCell = y \* levelStatic.width + x;

if (x == 0 || y == levelStatic.height - 1) { levelStatic.tileType[activeCell] = 1; }

else { levelStatic.tileType[activeCell] = 2; }

}

}

LevelLoad(levelStatic);

}

## Small Code Changes

Many things had to change to accommodate a randomly generating level. I will list the comments and code of the changes.

Some logic to restriction the input so the restriction logic can work correctly.

// Logic for tile type 1

if (levelStatic.tileType[activeCell] == 1)

{

int tileNumberCorrect = 0;

if (x == 0)

{

for (int xmax = 0; xmax < levelStatic.width; xmax++)

{

int logic = activeCell + xmax;

if (0 < logic && logic < levelStatic.width \* levelStatic.height)

{

//Debug.Log(logic);

if (levelStatic.tileCorrect[logic] == 1) { tileNumberCorrect++; }

}

}

}

if (y == levelStatic.height - 1)

{

for (int ymax = 0; ymax < levelStatic.height; ymax++)

{

int logic = activeCell - ymax \* levelStatic.width;

if (0 < logic && logic <= levelStatic.width \* levelStatic.height)

{

//Debug.Log(logic);

if (levelStatic.tileCorrect[logic] == 1) { tileNumberCorrect++; }

}

}

}

// Applies the static level type 1 restrictions

if (Array.Exists<int>(levelStatic.type1RestrictX, element => element.Equals(tileNumberCorrect)) &&

Array.Exists<int>(levelStatic.type1RestrictY, element => element.Equals(tileNumberCorrect))) { tileNumberCorrect = 0; }

number\_value.text = tileNumberCorrect.ToString();

}

# Dialogue System

## Brief

This section will implement a new system for displaying text from our narrator. This will be used to display the tutorial dialogue in a readable manor. All normal dialogue conventions should apply like being able to skip dialog by clicking the screen.

The other addition will be true enumeration states that control what things get displayed on screen. Menus, Game and Dialogue should all have multiple enumerations. No longer should you enable or disable the game object, some state machine based on enumerations should control it. Each state machine should only run once in the update function so there should be a standby enumeration.

## Enumeration States

This will be in place of the Boolean variables that control functions to change the active visibility of the GameObjects. All these systems will be replaced with enumeration variables, that when changed will set the active visibility. This will be done on the condition that it is not already active or unactive to avoid duplicate execution during runtime.

There is an order the states variables and their update if statements.

1. Game States
2. Menu States
3. Mouse States
4. Run time States

### Result

## Drag Selection

A simple State was added to change based on what tile was selected. By using the built in variable GetMouseDown(0) was able to have the tile blink on and off. I just needed the simple state 0, 1, 2 to indicate the base selected tile.

if (Input.GetMouseButton(0) && !Input.GetMouseButtonUp(0) || !MainGame.gameState)

if (tileVector\_int.z == 0)

{

if (MainGame.MapClick.GetColor(tileVector\_int) == Color.white && MainGame.mouseState == 0 || MainGame.mouseState == 1)

{

MainGame.SetTileColour(new Color(200, 0, 0), tileVector\_int);

levelActive.tileSelected[MainGame.LevelArrayIndex\_create(tileVector\_int)] = 1;

// Debug.Log(tileVector\_int);

MainGame.tileCorrect\_countdown++;

MainGame.mouseState = 1;

}

else if(MainGame.mouseState == 0 || MainGame.mouseState == 2)

{

MainGame.SetTileColour(Color.white, tileVector\_int);

levelActive.tileSelected[MainGame.LevelArrayIndex\_create(tileVector\_int)] = 0;

// Debug.Log(tileVector\_int);

MainGame.tileCorrect\_countdown--;

MainGame.mouseState = 2;

}

if (Input.GetMouseButtonUp(0))

{

MainGame.mouseState = 0;

}

# Publish the Game

# References

**There are no sources in the current document.**