Kieran Abelen

Mell1Games

Unity Methodology 1

Contents

[Introduction 4](#_Toc54798465)

[Brief 4](#_Toc54798466)

[Unity Hub 4](#_Toc54798467)

[Working with Unity 4](#_Toc54798468)

[MoSCoW 5](#_Toc54798469)

[Must 5](#_Toc54798470)

[Should 5](#_Toc54798471)

[Could 5](#_Toc54798472)

[Would 5](#_Toc54798473)

[Naming Conventions 6](#_Toc54798474)

[Scene Assets and Code 7](#_Toc54798475)

[Brief 7](#_Toc54798476)

[Binding Assets and Code 7](#_Toc54798477)

[Clickable 7](#_Toc54798478)

[Dynamic generation 8](#_Toc54798479)

[Brief 8](#_Toc54798480)

[Grid and Tilemap 8](#_Toc54798481)

[Dynamic instantiates 8](#_Toc54798482)

[UI assets and Main script 8](#_Toc54798483)

[GameObjects into Tiles into Tilemaps 8](#_Toc54798484)

[Summary 9](#_Toc54798485)

[Loading of the level 10](#_Toc54798486)

[Brief 10](#_Toc54798487)

[Simple storage method 10](#_Toc54798488)

[Loaded map storage 10](#_Toc54798489)

[Save level format 11](#_Toc54798490)

[Result 11](#_Toc54798491)

[Load level format 11](#_Toc54798492)

[Reading the Json into the Level object 12](#_Toc54798493)

[Result 12](#_Toc54798494)

[Loading Json Format 12](#_Toc54798495)

[Result 12](#_Toc54798496)

[Selecting correct tile 13](#_Toc54798497)

[Brief 13](#_Toc54798498)

[Selected tile stored into level 13](#_Toc54798499)

[Result 13](#_Toc54798500)

[Checking tile is correct 13](#_Toc54798501)

[Result 13](#_Toc54798502)

[Changing tile sprite 14](#_Toc54798503)

[Result 14](#_Toc54798504)

[Tile type predicting 15](#_Toc54798505)

[Brief 15](#_Toc54798506)

[Correct tile count 15](#_Toc54798507)

[Type 1 15](#_Toc54798508)

[Logic 15](#_Toc54798509)

[Result 15](#_Toc54798510)

[Type 2 16](#_Toc54798511)

[Logic 16](#_Toc54798512)

[Result 16](#_Toc54798513)

[Dynamic camera 17](#_Toc54798514)

[Brief 17](#_Toc54798515)

[Grid logic 17](#_Toc54798516)

[Result 17](#_Toc54798517)

[Camera 18](#_Toc54798518)

[Result 18](#_Toc54798519)

[Screens and Popup 19](#_Toc54798520)

[Brief 19](#_Toc54798521)

[Problem? 19](#_Toc54798522)

[Migrate start and update to functions 19](#_Toc54798523)

[Menu Logic 20](#_Toc54798524)

[Active and Paused game 20](#_Toc54798525)

[State Machine 20](#_Toc54798526)

[Result 21](#_Toc54798527)

[Name and Logo 22](#_Toc54798528)

[Brief 22](#_Toc54798529)

[Name of the Game 22](#_Toc54798530)

[Logo 22](#_Toc54798531)

[Result 23](#_Toc54798532)

[Conclusion 25](#_Toc54798533)

[References 26](#_Toc54798534)

Unity

# Introduction

## Brief

This area will be added to as new knowledge as each section is acquired. I will also try a pair each introduction section with the GitHub commits. A good starting note is the Unity IDE. It is made for non-programmers since even some of the most advanced tutorials are very simple in terms of programing knowledge.

### Unity Hub

This is a tool used to develop the unity game. The most obvious change compared to other IDE is its focus on a scene and not any form of code. Most IDE have some drop and drag feature but it normally isn’t the primary way you implement things in the IDE and only works for some languages.  
In unity it’s in reverse the drop and drag UI if the primary feature and the scripts or code are the addition.

With this knowledge I will need to sift through the UI elements and turn them into code. The UI elements are inherently hard coded and not dynamic but if I can generate UI element with code, I can break this inbuilt cycle.

### Working with Unity

The Scene and code division adds a very interesting problem especially for a puzzle game that needs dynamic generation. That problem is I don’t need the scene at all, the only potential us is for something like backgrounds or UI.

The ability to create a dynamic puzzle requires the creation of the scenes assets that can be scaled. For future note a 2d game assets would instead be duplicated many times.

Another way to look at this is each scene asset is a separate view interface. They can instance once created and manipulated; this is how normal Unity development works but on a much smaller scale.  
Pushed to the extreme one asset could serve the entire program’s needs, like let’s say like moving grass.

(Unity, 2019)

## MoSCoW

A good idea of what I need to priorities in each prototype.

### Must

1. Instanced grid asset
2. Instantiate one image into multiple grid tiles
3. UI for menu and end screen with reuse

### Should

1. Update grid with changed on from a save variable
2. Saving variable parsed into a file
3. File can be edited and is parsed into the save variable
4. More than one level with some sort of selection menu

### Could

1. Have image save file
2. Load save file into separate program to edit grid
3. Auto generated game logic based of save variable
4. Instantiate the UI assets.

### Would

1. Add some navigation
2. Hints to help discover the image

## Naming Conventions

Since assets have preset names or classifications, we need to us the same names in the script for calling the variables. Script objects should be primarily named after the unity assets.

This is the convention:

1. Asset name e.g. Grid, Map, Tile, Sprite
   1. Names should be lower case if they are a variable
   2. Should be the most unique thing if the asset so TileMap = Map.
2. Purpose name e.g. Main for script, UI for buttons
   1. Should always be capitalized
   2. What is the asset or variable being used for?
   3. Assets that have script components should have Main e.g. MainUI
   4. The same names for similar types of things
3. Unique name e.g. \_blue for blue button
   1. Lowercase
   2. Used for similar Purpose assets of variables but different function.

# Scene Assets and Code

With reference to another Nonogram game I bought I managed to come up with some core concepts.

(BZ, 2019)

## Brief

The idea is that each asset serves a view or interface, I will create all the assets I need and then manipulate them with code. The assets are the view, the code is a model and the controller in the scene itself with its main script.

Another key aspect is the use of a picture to fuel the generation of the view. This process will be reactive so during the creation of images though an in-game editor will instantly update the grid assets and therefore be re-rendered real time.

The in-game editor will have to have some logic to find out if a solution is possible and how many ways it can be solved which can then assign a difficulty, but this will be a much later addition

## Binding Assets and Code

This can be done in one of two ways, to some respects it is one-way databinding either the code generates the asset, or the script references an existing asset.

1. GameObjects can be created, components added and then instantiated.
2. Assets can be created; the script creates public variables and then the inspector can assign them to those Assets.

Some assets like Tilemaps have internal storage for each tile so once the public variable is created and assigned you can then set each tile individually on the Tilemap.

## Clickable

Buttons and other UI element have events built in already but in terms of game speed we want to use the smallest asset and rely on the most code.  
Code itself can listen to mouse inputs and retrieve the location of them, this can then be translated into events that can change let’s say the color of a map tile.

# Dynamic generation

## Brief

The real goal is how can I make game assets that are not hard coded generate during runtime. Even things like projectiles will have to be generated during run time rather than always being placed.

Unity does this by creating GameObject then instantiate it into the game files, this then can be changed or copied based on your needs. This basic tutorial has a much simpler version of the Nonogram game asset I purchased.

(Games, n.d.)

## Grid and Tilemap

These are amazing GameObject’s that can first be set up with preferences inside the grid that will launch a script on start-up and then a Tilemaps can be laid out on top. These Tilemaps act as layers in photoshop.  
Both the Grid and Tilemap need to be created before the launch of the script and a public variable will hold a reference to them inside the script which will be assigned with the inspector (where the script is in unity)

## Dynamic instantiates

The real question where the dynamic generation should go?

The example we have obtained places it inside the game main script at the start. This is okay but what if we want to reuse the code once, we what if we want to reset or change the level without restarting the scene(program).

### UI assets and Main script

Asset can simulate some sort of script hierarchy of inheritance even if unity doesn’t require it since all starts and updates are executed on launch. The way to do this tie scripts to UI, each UI script will act as a controller starting and launching the main model script.

The main model script in this case the grid script component which has public variables linking to all the assets in the scene, the View. The UI script should act like hooks and have entries inside the running model code to change it.

An example is a reset script should have a Hook method that resets the main script with some special piece of code and then destroy itself after the start is called. The advantage of this is the reset could be started anywhere, even from outside the UI like a fail state.

The MVC approach should let the screen reset the puzzle from anywhere in the program.

## GameObjects into Tiles into Tilemaps

This is the key to puzzle game since it will entail adding numbers and shaped over top of a colored Tile for the game. Tilemap tiles can add many things to them in particular a GameObject.

Prefabricated GameObject that has a child Text Mesh can be added to the tile to be rendered. By editing this child, you can set its text to the number you want to render.

TextMesh number = tileNumber.GetComponentInChildren<TextMesh>();

number.text = "1";

This is amazing since that number is changed inside the tileNumber GameObject.

tileBasic.gameObject = tileNumber;

This means when we add our basic tile to the Tilemap it will contain the number all packaged inside the tile. The same tileBasic added to the Tilemap could contain multiple things inside out prefabricated GameObject basically like tile layers.

## Summary

That is, it for the dynamic generation with colors and numbers/shapes on top. No more generation is needed, the result is relatively simple. The trick will be integrating purposeful generation of a map and logic to check for a win/loss.

# Loading of the level

## Brief

Now that we have a generating map with clickable tiles, the next step will be how to use that to generate a hard-coded puzzle. There needs to be a simple storage method of the puzzle and a rendered version.

## Simple storage method

The most obvious method to have a stored puzzle is simply a string inside a database system like firebase, this way potentially sharing maps is very easy.  
Then the Question is what size should each character be?

A sensible option is to use a base multiple since that could be compressed and manipulated more easily, (conversation into just numbers). Most games use item totals of 128 or 64 and 64 being much simpler to store as a string.

I need a string total of 64:

1. Numbers 10
2. Lowercase letters 26
3. Uppercase letter 26
   1. Total Being 62
4. “\_” makes since Its wont conflict with variable naming conventions
5. “-“ does conflict but not all languages naming conventions

A huge advantage to the 64 size is when used for a map, each tile can be divided very evenly into different tiles of 9. 7 tiles of 9 is 63 and 1 extra tile for the number column.

I may never use this, the main application of it would be a fast databasing system to query things especially for user generated content.

## Loaded map storage

There are many ways of storing variables, Static classes with protected variables, XML, JSON or even txt files. The real issue is memory, since I don’t want the loaded map to hold both the original values and the changed values.

A static class might work nice, but Unity doesn’t have true constants or services like in react. The best practice for storing something like a save game is to use file.

Unity itself has built in JSON serialization so this would be the easiest form of local storage rather than loading them into variables. One local variable should hold the json file only reading it for something like a map reset.

(Royy212, n.d.)

## Save level format

Before outputting into json I need to lay out what a save will look like. The first thing that needs to be stored is the map size, after that the map tiles itself. Then finally any extra information like what the name is or other variables, this will have to each start with a string or variables size since on translated to a single string needs a record of when it starts and stops.

2 things need to be kept in mind when design:

1. Json accepts objects which each should have a name.
2. The 64-character length means some limitations might apply to size.

### Result

level.width = 10;

level.height = 10;

level.tileArray = new int[level.width, level.height];

level.name = "";

## Load level format

How do I load or render the level?

The real issue is how to break equals and asset dependency. I need a new asset to inherit the original one but not change the original. In a normal game where sprites need to be duplicated then destroyed there is a very simple function called destroy, this is the key to getting this to work.

Create a new asset inside the game to break its reference to the origin

tileBasic.gameObject = Instantiate(tileNumber, transform);

Prefab can now be instanced without changing the other tiles prefabs

TextMesh number = tileBasic.gameObject.GetComponentInChildren<TextMesh>();

Simple logic since 0 is default in array

if (level.tileArray[y, x] != 0)

{

number.text = level.tileArray[y, x].ToString();

}

This is another instantiate, but all the changes need be in place before creation.

mapClick.SetTile(new Vector3Int(x, y, 0), tileBasic);

Destroy the original object after the launch

Destroy(tileBasic.gameObject, 1);

This is a very simple implementation of duplicating each number tile and distributing it to the Tilemap.  
A big improvement would be trying to destroy the object outside of the loop that way we wouldn’t need to delay it but right now it will not work.

### Reading the Json into the Level object

Its very simple, any object can be turned into a json string, then this can be outputted into json or text file to then be read back.

string json = JsonUtility.ToJson(myObject);

Then to convert it back

myObject = JsonUtility.FromJson<MyClass>(json);

(Unity, 2020)

### Result

{"width":10,"height":10,"name":""}

This simple json code can be converted from a text file into a level object.

level = JsonUtility.FromJson<Level>(File.ReadAllText(Application.dataPath + "/Scripts/Map1.json"));

## Loading Json Format

The issue is any complex array or object wont save to json, but we can use this issue as a strength of code that means out level object could have packed map that is just and array and an unpacked version within the instance of itself.

public int[] tileArray\_packed;

this means it won’t save 2 maps just one to json.

We can pack andupack the map when we set the tiles inside start with the same loops.

level.tileArray[y, x] = level.tileArray\_packed[y \* 10 + x];

The reverse equation can be used to save the file. The only issue with loading is any variables that is not created due to lack of saved variables in the json need to be created after the level is loaded.

### Result

I will leave this here, but I could just use the packed map for now, but this may change if I must have more complex maps that do need to be packed or unpacked.

The correct y and x coordaining in the array are the max width times the height plus the width since for everything max width you will get a new row.

if (level.tileArray\_packed[y \* level.width + x] != 0)

{number.text = level.tileArray\_packed[y \* level.width + x].ToString();}

# Selecting correct tile

## Brief

We need some game logic to first detect where the correct tile is selected or not. Then compare the correct tiles to the selected one to win the game. Some logic with sprites is added to make the game more clear

## Selected tile stored into level

First thing that is needed is another two level int arrays:

1. The correct selection of tiles
2. The current selection of tiles

When a tile is selected it is then added to the current selection and once per turn/selection it will be checked against the correct tiles.

### Result

The json storage is expanded.

private class Level

{

public int width;

public int height;

public int[] tileType;

public int[] tileCorrect;

public int[] tileSelected;

public string name;

}

## Checking tile is correct

The issue is we have two arrays and then need to be compared, we could loop each array comparing each index and counting the amount of equivalence until we reach the count of width time hight of the map.

There is a simple C# function called SequenceEqual that can do this without having to loop through the array. Once you have won the game needs to end and a UI element need to popup to continue if the game isn’t complete then save the to file and then continue.

### Result

The else was moved here from the existing code since we don’t want a saved map to be correct. The dialog box should be dynamically filled with content.

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

{

Debug.Log("Won");

dialogBox.gameObject.SetActive(true);

}

else

{

string json = JsonUtility.ToJson(level);

File.WriteAllText(UnityEngine.Application.dataPath + "/Scripts/Map1.json", json);

Debug.Log("Not Won");

}

## Changing tile sprite

A thing I added to make it more clear on how to win the game was to stick a game boarder around the play screen so x=0 and y=max is shown to be not selectable. This requires 2 more sprites and some logic on deciding what sprite goes where before the tile is set.

### Result

A bit crude but simple, a better one could be formulated based on the level json storage. I will use level.sprite and index it with each type.

if (x == 0 && y != level.height - 1)

{

tileBasic.sprite = square2;

}else if(y == level.height-1 && x != 0)

{

tileBasic.sprite = square3;

}else

{

tileBasic.sprite = square1;

}

# Tile type predicting

## Brief

I want to lay the groundwork for the game to predict things so first I need to predict which number should go on the tile based on the type. The tile storage will still be an int but 1 will be column and row number and 2 will be tiles inside the game window. This would then give a lot more reusability of out save storage since each int will be different type of a tile and not a just a different number.

## Correct tile count

The easiest way to predict the number on a 1 or 2 type tile is to count the correct tiles inside the level.tileCorrect array. This can be done by looping the length of a row or column for tile type 1 or by looping 9 times for each correct tile around tile type 2.

## Type 1

This will evaluate each column or row for correct tiles so there will multiple if statement and for inside each other. This will be the order of logic:

1. If tile type 1
   1. For each tile in the row
      1. If inside array boundaries
         1. If tile is correct inside array
            1. Add 1 to tile number
   2. For each tile in the column
      1. If inside array boundaries
         1. If tile is correct inside array
            1. Add 1 to tile number

I could add more if’s when I need to save time on loading but right now simple is best.

### Logic

I will need a way to translate x and y coordinates into an array index. The x coordinate is simple since it will just be additive to the index. The Y coordinate is more complex and requires the multiplication of the x max width and since it is from the top of the grid it will have to minus the current index.

### Result

if (level.tileType[levelCell] == 1)

{

int tileNumberCorrect = 0;

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

{

int logic = levelCell + xmax;

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

{

//Debug.Log(logic);

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

}

}

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

{

int logic = levelCell - ymax \* level.width;

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

{

//Debug.Log(logic);

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

}

}

number.text = tileNumberCorrect.ToString();

}

## Type 2

This will need less code but more complex code since I need to check a 9 by 9 tile around the selected tile. The easier ways are to use a for loop that starts at -1 and ends at 1, then just add it to the existing tile index.

This will be the order of logic:

1. If tile type 2
   1. For X loop starting at -1 and ends at 1 && For Y loop starting at -1 and ends at 1
      * 1. If inside array boundaries
           1. If tile is correct inside array

Add 1 to tile number

### Logic

We are checking a virtual 9 by 9 grids for correct tiles. The logic will add -1, 0 and 1 to the existing index, x coordinate is the current index but why requires the multiplication of the x width to select a different y coordinate.

### Result

if (level.tileType[levelCell] == 2)

{

int tileNumberCorrect = 0;

for (int ydif = -1; ydif < 2; ydif++)

{

for (int xdif = -1; xdif < 2; xdif++)

{

int logic = (y + ydif) \* level.width + x + xdif;

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

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

}

}

}

number.text = tileNumberCorrect.ToString();

}

# Dynamic camera

## Brief

The camera currently is static has a fixed size, some global variables and logic need to be set up to control this. The size needs to adjust based on the grid size, but UI elements should not change in size but adjust when the camera zooms out. This means a global unit variable needs to be set up updated when the camera scales the UI scales too.

This ground work will also help with different devices screen sizes and be the backbone of a camera settings in an options menu.

## Grid logic

First, I need to centre the grid so if it and the camera change in size they won’t me misaligned. This will Centre the tiles rather than using the top right quadrant.

This will use the level width and height divided by two too centre the grid. That means when placing the tiles on the grid there needs to a translation from the positive positions to negative ones. This is also the same for a selected tile, which again needs to be translated from negative positions to positive ones.

level.width / 2;

level.height / 2;

Both will be used anywhere I need to interact with the grid vs level.

### Result

This is to centre the grid at the start

0 int xCentre = x - level.width / 2;

int yCentre = y - level.height / 2;

Then I need to add code to translate between the grid which can have negative tile positions and the level that only has positive(top left quadrant).

The first one is an updated getCell and now has correct naming and excludes sections on the grid in the negative quadrants.

Vector3Int TileVector\_get(Vector3 levelPos)

{

// Return the tile that is clicked

Vector3Int tilePos = mapClick.WorldToCell(levelPos);

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

if (tilePos.x >= level.width / 2 || tilePos.y >= level.height / 2 - 1 || tilePos.x < -level.width / 2 + 1 || tilePos.y < -level.height / 2)

{

// An unreachable tile

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

}

// Return it as an int vector, this is needed to change colors

return tilePos;

}

The second one just stores the tile positions in the level correctly so there are no negative positions (moving the grid into the positive quadrant).

int LevelArrayIndex(Vector3 tilePos)

{

// Turning all negative tile positions into positive

tilePos.x += level.width / 2;

tilePos.y = (tilePos.y + level.height / 2) \* level.width;

int levelIndex = (int)(tilePos.x + tilePos.y);

// Returning the index of the level

return levelIndex;

}

### Camera

Very simple since I just need to set the camera size to the height / 2 like with grid logic.

### Result

view.orthographicSize = level.height / 2;

# Screens and Popup

## Brief

This section requires a lot of thinking and explaining since one this will set a precedent for the entire game going ford. Second this will directly impact how the UI will interact with the main game script.

## Problem?

The real question is do I separate the menus into its own scripts and or scenes?

The simplest solution would be to keep everything together, but it would conflict with the start and update inside my main game script. The grid when launching the game should not be there until a level is selected.

The next option is to separate the scripts but not the scene. This is a good option since we can enable the script in the scene at any time, and a pause the game which will have the same menu. The downside is that we will have two-way dependencies.

The last option is that we separate all logic into their own script and scene. This would mean we would have to make another in game menu or force the player to exit to the main menu to change the level.

I believe for now the best option is the simplest keeping everything in the same script. This would let me expand the functions of the base script and ruse many functions. A select level button could be using the same function that is used to load a level from a previous save

## Migrate start and update to functions

I need to pull the game out start and update and place them into their own functions, in some sense it is refactor. First 2 new private functions need to be created load and update the level, then a global variable to control the state of the game(running or not)

Once a button is pressed it will LevelLoad, then switch the levelStart\_state to true and then the level will LevelUpdate.

## Menu Logic

I need 3 sub menus, start screen, a popup menu and another to select the level. They should each use an image as background and buttons in the foreground, the select level menu will use a grid layout to position the buttons inside.

There is a single canvas and having a control script is a possibility but why would I need to have a separate script to control the UI, when I can set its state directly and use a C# subscript to control it in the future.

### Active and Paused game

The main thing I need is a state to pause the game, due to complexity of the game logic a state machine is needed to outline interactions.

levelActive\_state = false;

### State Machine



## Result

The level can’t be active when the main menu is loaded, I then need to add the load level function to the main menu. The load level should turn off all other screens after that, removing the start menu.  
Then each button should transition to another screen or if the start button is pressed load level on and start the game.

// Main menu button listeners

UIButtonMaintoStart.onClick.AddListener(delegate { LevelLoad(0); });

UIButtonMaintoSelect.onClick.AddListener(UISelect\_transition);

// Select level menu btton listeners

mapDirectory = new DirectoryInfo(Application.dataPath + "/Scripts/Maps");

fis = mapDirectory.GetFiles();

foreach (FileInfo fi in fis)

{

if (fi.Extension.Contains("json"))

{

GameObject goButton = Instantiate(UIButtonSelect);

UnityEngine.UI.Button goButton2 = gameObject.AddComponent<UnityEngine.UI.Button>();

goButton.GetComponentInChildren<Button>().onClick.AddListener(delegate { LevelLoad(0); });

goButton.transform.SetParent(UIMenuSelectPanel, false);

}

}

UIButtonLeveltoMain.onClick.AddListener(UIMainMenu\_transition);

// Popup menu button listeners

UIButtonPopuptoStart.onClick.AddListener(delegate { LevelLoad(0); });

UIButtonPopuptoMain.onClick.AddListener(UIMainMenu\_transition);

UIButtonPopuptoSelect.onClick.AddListener(UISelect\_transition);

// Transition screens

UIMenuMain.gameObject.SetActive(true);

UIMenuSelect.gameObject.SetActive(false);

UIMenuPopup.gameObject.SetActive(false);

levelActive\_state = false;

Each buttons function will just change the active screen making it a transition.

# Name and Logo

## Brief

This section will lay out my thinking regarding the name of the game and the logo/look I have in mind.

The main consideration will be the fact that I want to sell themes or looks for microtransactions, so the logo and look need to basic to allow flexible payed themes to be applied instead. This means that the core game will mostly be free outside of cosmetic themes, potentially accessing content earlier with a theme bundle might be an option in the future.

I will then demonstrate me applying the theme and trailing a level. Some wireframes in future could be generated to set a president for other themes.

## Name of the Game

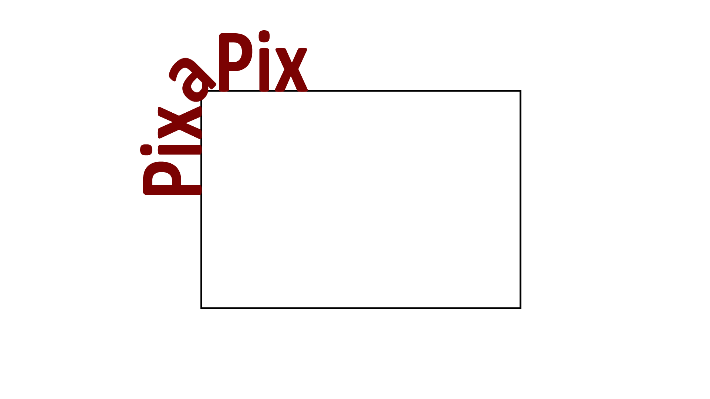
The game is called Pix-a-Pix with direct inspiration from Pic-a-Pix. These games are a more marketable version of the Japanese Nonogram game. The Pix-a-Pix name means It will draw inspiration from all similar games and make my own inspired genre of puzzle games but with a very cool theme.

## Logo

The primary game will have a balloon or party theme with a Nonogram and Minesweeper gameplay. Select balloons you think are part of the image and watch the balloons you didn’t select pop to reveal the image. This will all be animation and Other similar themes like candles or party poppers could be there as purchasable themes.

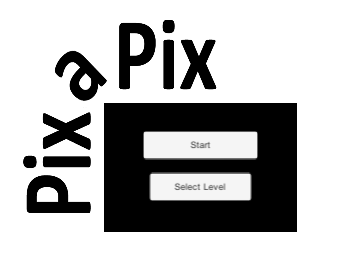
With all this in mind the logo needs to be adaptable to different themes, so I want a fun going logo that bends around each screen to then be later changed by a payed theme.

V2.png

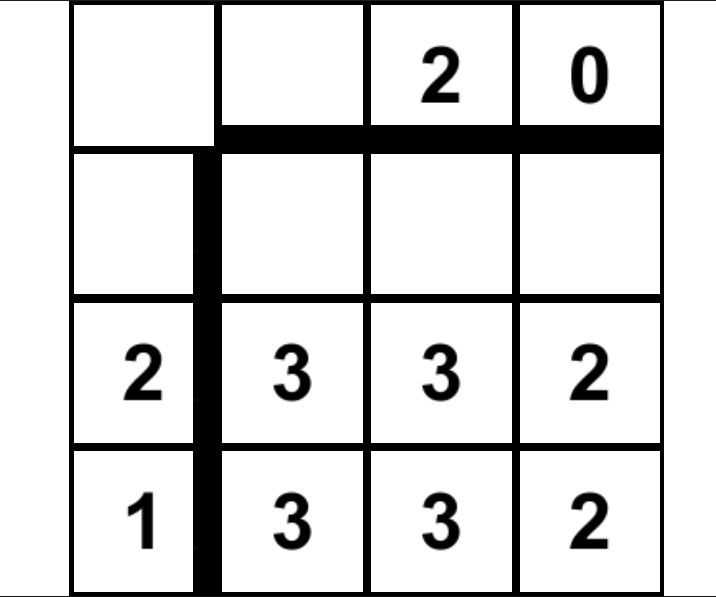


## Result

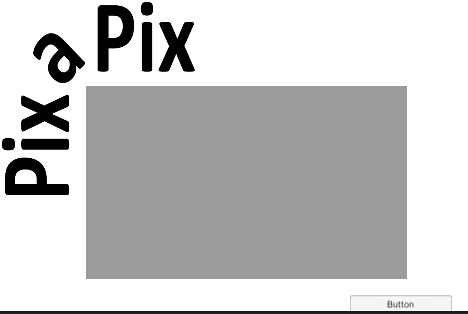
Main Menu and general menu design. I may add rounded corners to box but at the moment it is not necessary.



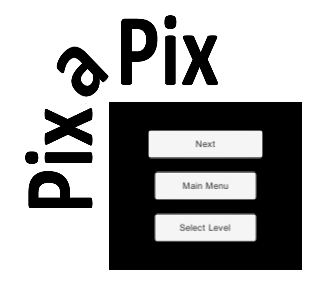
Game screen once tiles have been generated and numbered



Select level screen, buttons will be added dynamicly



End level screen.



# Conclusion

This concludes the Methodology 1 and marks the end of the initial understanding of unity, the next step will be furthering my understanding around the publishing process. An intermediate understanding of 2d unity will have to come after the completion of the publishing of the game in Methodology 2.

The core game is working as of the completion of this methodology and only production related tasks need to be completed like testing and creating more levels.   
Regarding Testing and the fixing of many UI elements will be the main priority and will require a standard manual testing framework. Testing logs or crash dumps need to implement in some capacity for during release. Automated testing can’t really be done unless I use servers to offload game processes.

Over all I am happy with this methodology and feel like I really did learn how to begin the development of a 2D unity game.

# References

BZ. (2019, July 26). *Picross - Nonograms | Packs | Unity Asset Store*. Retrieved from assetstore.unity.com: https://assetstore.unity.com/packages/templates/packs/picross-nonograms-150711

Games, L. R. (n.d.). *GameDev Tutorial: HowTo make a Dynamic Grid in Unity C# (2d sprites) - YouTube*. Retrieved from www.youtube.com: https://www.youtube.com/watch?v=u2\_O-jQDD6s

Royy212. (n.d.). *XML or JSON for Unity : Unity2D*. Retrieved from www.reddit.com: https://www.reddit.com/r/Unity2D/comments/6pldok/xml\_or\_json\_for\_unity/

Unity. (2019, April). *Unity - Manual: Unity User Manual (2019.4 LTS)*. Retrieved from docs.unity3d.com: https://docs.unity3d.com/Manual/index.html

Unity. (2020, 09 01). *Unity - Manual: JSON Serialization*. Retrieved from docs.unity3d.com: https://docs.unity3d.com/Manual/JSONSerialization.html