Unity

# Introduction

## Brief

This area will be added to as new knowledge of each section in acquired. I will also try a pair each introduction section with the GitHub commits. I good starting note is the unity development kit was almost 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 assets
2. Grid changed with image files

### Should

1. Update grid with changed on image file
2. Saving image changed back to an image
3. Click image tiles to select or unselect them

### Could

1. Progress image vs real image
2. Saving the progress image as a separate image file
3. Loading/checking for save file image

### 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 e 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 generation

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.

# The Puzzle

## 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, unpacked version 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 layout 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 wont 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 have to 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();}

# Game Logic

## Brief

We need some game logic to first detect where the correct tile is selected or not but accurately assess what number should be there. This could even go so far and predicting the correct number in a selected tile, this would then give a lot more reusability of out save storage since each int could be different types of a tile and not a different number.

## 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.