Individual Portfolio Document

DES310

LEE GILLAN

1701370

Role

Programmer

Team

Ellipsis

Brief

Food Science

Goals

For this project, my personal goals were:

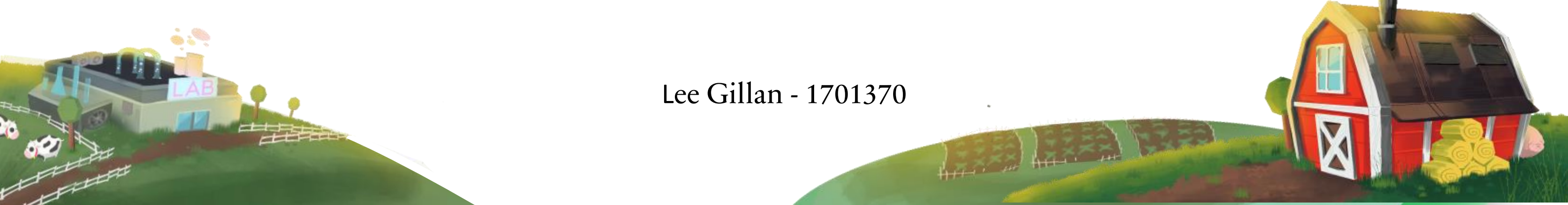
- To have the game run smoothly and function properly.
- The code to be as dynamic as possible for any changes that would have to be made.
- Meet the brief given to us by the client through implementing features that would be engaging but also educational.
- Have a working prototype that would be fun to play and have a solid basis to work from if I wanted to continue development.

Research

When deciding which game engine to develop the game in, it was a tossup between Unreal and Unity. This was mostly because I had never used either engines (neither had the other programmer) but the Artists and Designers had already had experience in Unreal. Although the group knew that Unity was less resource intensive. After a bit of research into the matter and seeing the amount of praise Unity had for the simplicity of developing a mobile game on (and the mentor, Gordon, asking why we were thinking about using Unreal for a mobile game). It was decided that Unity would be the best engine to use. Unity allows for quicker compiling and easier mobile testing. Unity allows for easy multi-platform development, meaning we could build to more than one device without hassle, which met with the target audience from the brief. Being that it would be our, programmers, first time using Unity, the mass amount of online resources also made us believe that Unity would be the better choice. (Unity Forums, no date) (Unity Answers, no date) (Medium, 2018)

I had looked at other games on the App Store (App Store, no date)and Google Play Store(Play Store, no date) that had a similar genre to the idea that team came up with from reading the brief. The research allowed for feature ideas to be suggested to the team and then we could think about how to implement them into the game. A core feature that came from this was when building a field there would be an overall value that acts as sustainability that the player must keep down or else it would affect the environment. This tied it to the brief. This research allowed for us to think about how we would implement them through code and what would be the best way to do so.

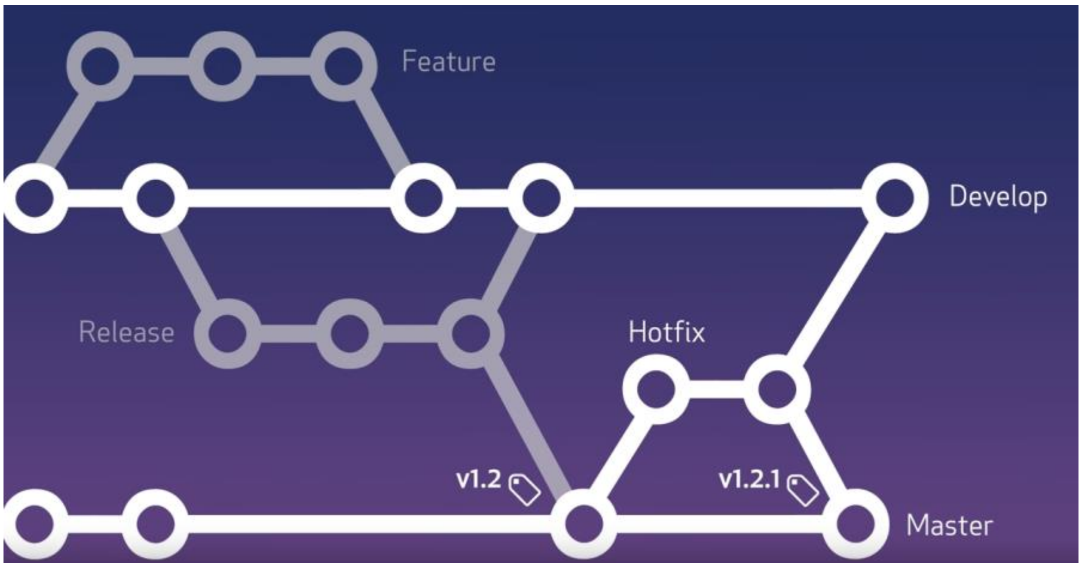
Research into mobile application building had to be done to assure the game could be played on multiple devices, including lower end. Looking into poly count and limitations was difficult as it varies so much depending on the device. Speaking to the



group mentor (Gordon) about ways to improve performance helped with understanding how to achieve our goal of a smoother experience. GPU instancing was brought up and draw calls were mentioned which was a great help when profiling any performance issue that we had.

Through looking at how to build onto mobile devices it was noted that developing for iOS meant that you would have to pay for a developer account meaning the iOS builds for the game had be halted and we would be mainly focusing on Android devices.

Learning how to use GitHub(Github, no date) worked and how to use Source Control effectively was important as we would be using it throughout the project. GitHub uses a branch-based workflow and when reading into it, David came across a good explanation of how to use branches for adding in features and then using a development branch as a working and up to date branch.



https://youtu.be/aJnFGMclhU8?t=356 - link to video

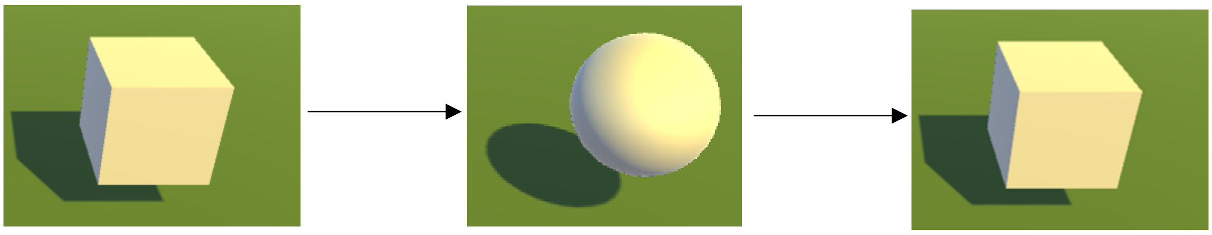
GitHub has many useful elements like projects which lets you create a list of tasks that must be done. Using this in our repo was helpful to know how far along we were with implementing features or fixing any bugs. It let us know what to do next if we had anything.

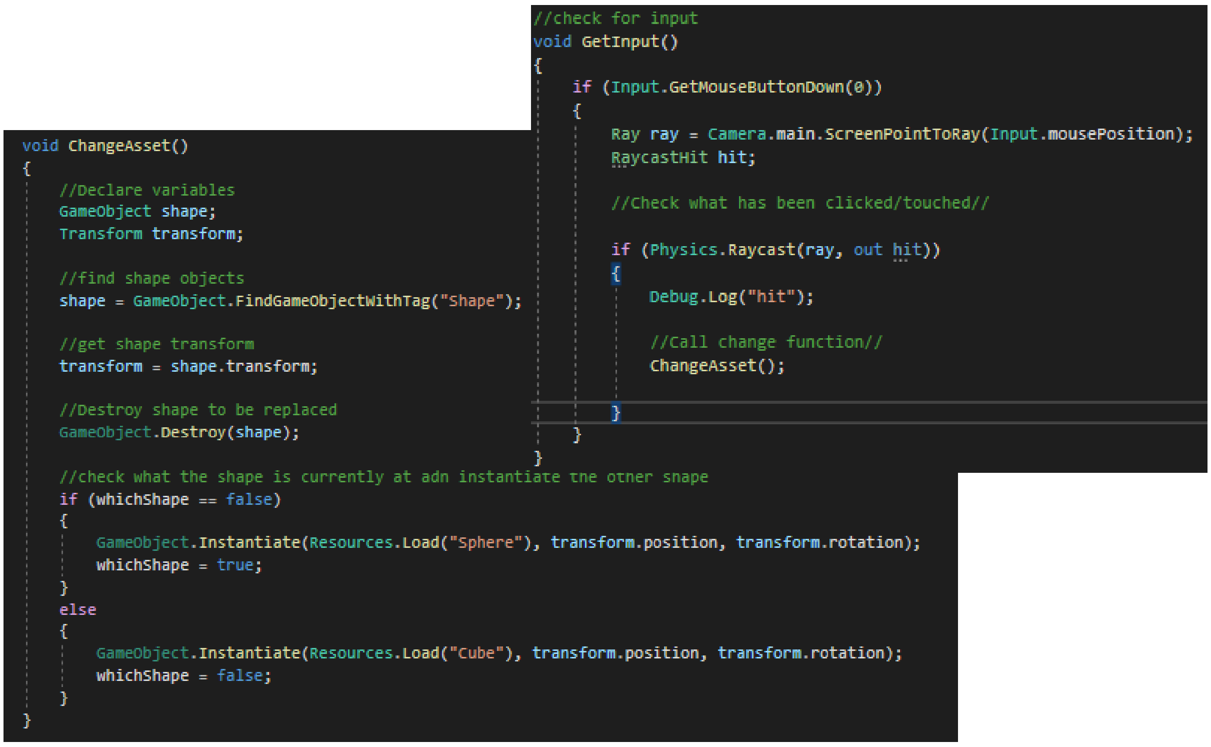
Due to not using Unity previously, using YouTube tutorials and the unity forums helped massively when not knowing how to implement something properly or when an error came up. The most helpful on YouTube was Brackeys, as the tutorials were easy to follow and described how to use many of Unity's features in a video at a time. In doing so, this

helped show any mistakes that I had made or made clear any better methods on how to implement any of the features. When getting an error that I did not know how to solve I would just search the error and for the most part it was inside the Unity forums(Unity Forums, no date) that I would find a solution. This was one of the reasons why we chose Unity as the development engine.

Project Work

Changing Asset when clicked on – To begin with we wanted to get the controls out of the way, so we decided to start with a simple task of changing a sphere to a cube and back again when it was tapped. David and I both worked on solving out how to do this as we were new to Unity.

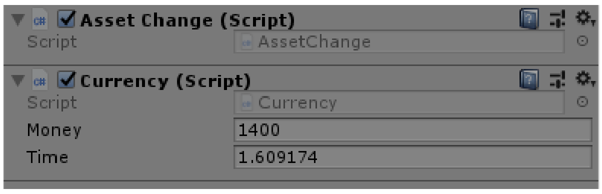




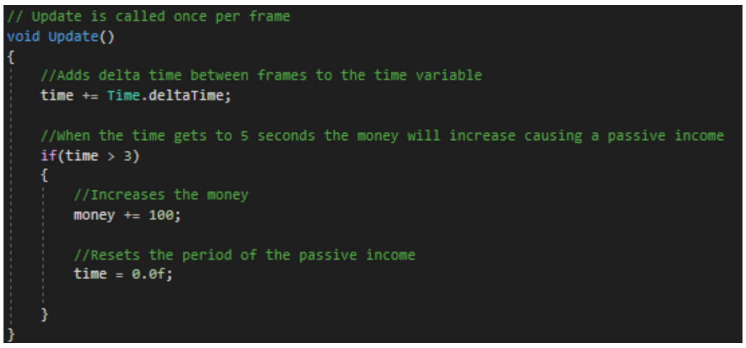
```
//check for input
void GetInput()
{
    if (Input.GetMouseButtonDown(0))
    {
        Ray ray = Camera.main.ScreenPointToRay(Input.mousePosition);
        RaycastHit hit;

        //Check what has been clicked/touched//

        if (Physics.Raycast(ray, out hit))
        {
            Debug.Log("hit");

            //Call change function//
            ChangeAsset();

        }
    }
}
```

```
void ChangeAsset()
{
    //Declare variables
    GameObject shape;
    Transform transform;

    //find shape objects
    shape = GameObject.FindGameObjectWithTag("Shape");

    //get shape transform
    transform = shape.transform;

    //Destroy shape to be replaced
    GameObject.Destroy(shape);

    //check what the shape is currently at adn instantiate the other shape
    if (whichShape == false)
    {
        GameObject.Instantiate(Resources.Load("Sphere"), transform.position, transform.rotation);
        whichShape = true;
    }
    else
    {
        GameObject.Instantiate(Resources.Load("Cube"), transform.position, transform.rotation);
        whichShape = false;
    }
}
```

Adding Buying – To implement buying into the game I created a script that would add to a money value every few seconds and, adding on top of the changing asset mechanic, a cost would be taken off the money value if the money value was greater than the cost.

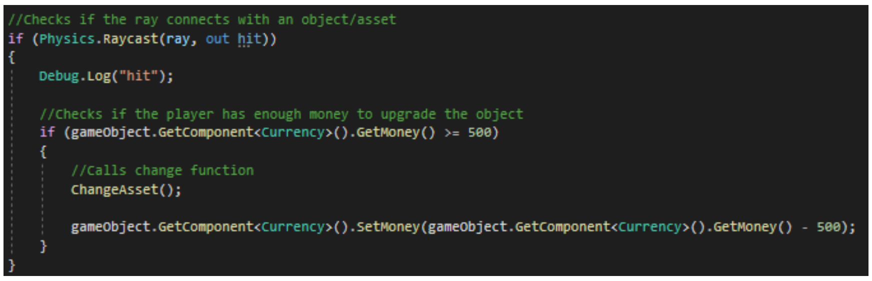


Asset Change (Script)	
Script	AssetChange
Currency (Script)	
Script	Currency
Money	1400
Time	1.609174



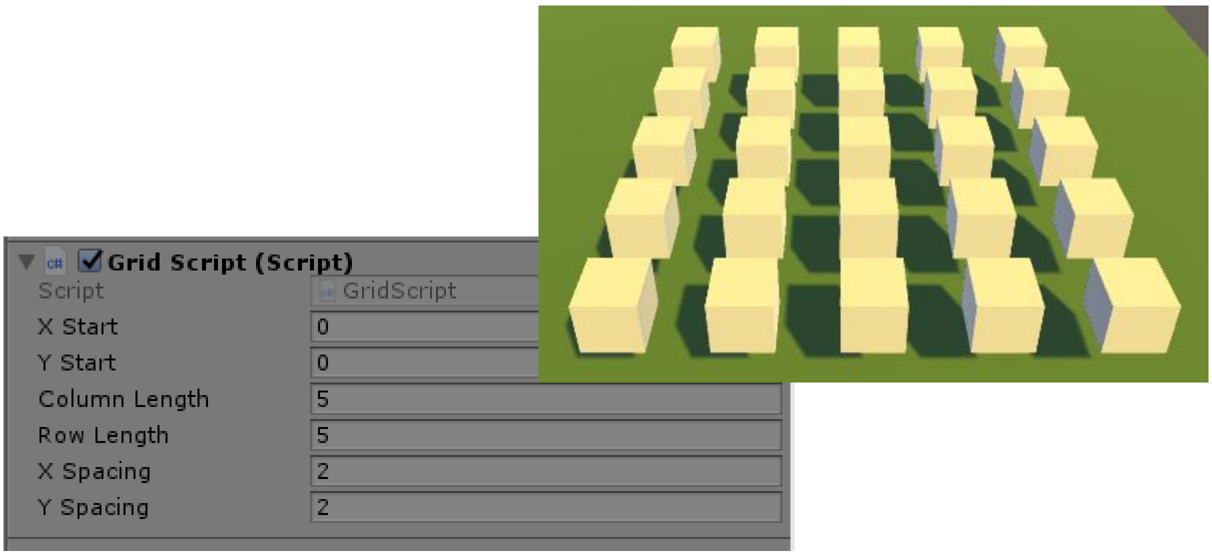
```
// Update is called once per frame
void Update()
{
    //Adds delta time between frames to the time variable
    time += Time.deltaTime;

    //When the time gets to 5 seconds the money will increase causing a passive income
    if(time > 3)
    {
        //Increases the money
        money += 100;

        //Resets the period of the passive income
        time = 0.0f;

    }
}
```

This would later be used as a guide for the food value.



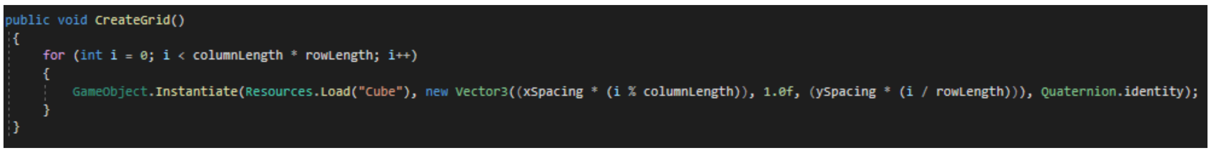
```
//Checks if the ray connects with an object/asset
if (Physics.Raycast(ray, out hit))
{
    Debug.Log("hit");

    //Checks if the player has enough money to upgrade the object
    if (gameObject.GetComponent<Currency>().GetMoney() >= 500)
    {
        //Calls change function
        ChangeAsset();

        gameObject.GetComponent<Currency>().SetMoney(gameObject.GetComponent<Currency>().GetMoney() - 500);
    }
}
```

Creating Grid – The grid would be used to place the assets on so that was also one of the first things that was set to be implemented. David and I split the tasks. I was tasked with creating the grid and having the size be modifiable so that David could then implement his part.



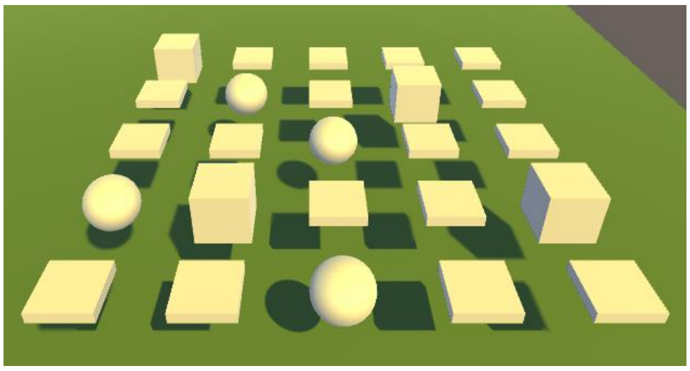
Grid Script (Script)	
Script	GridScript
X Start	0
Y Start	0
Column Length	5
Row Length	5
X Spacing	2
Y Spacing	2

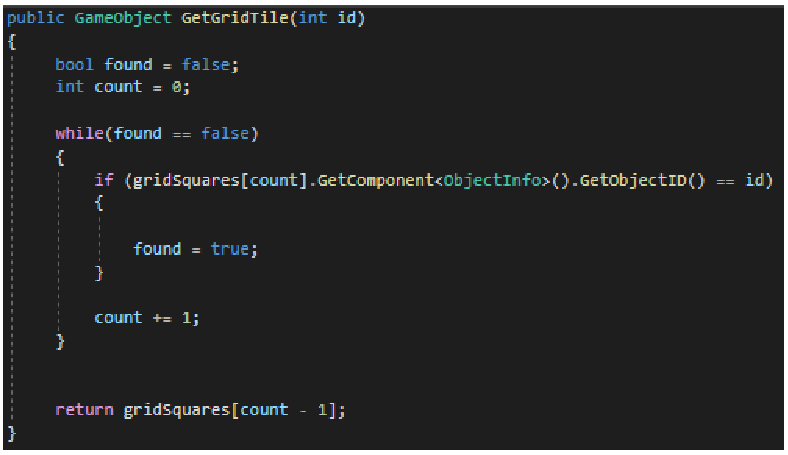
In the inspector, the size and spacing of the grid can be changed before playing.



```
public void CreateGrid()
{
    for (int i = 0; i < columnLength * rowLength; i++)
    {
        GameObject.Instantiate(Resources.Load("Cube"), new Vector3((xSpacing * (i % columnLength)), 1.0f, (ySpacing * (i / rowLength))), Quaternion.identity);
    }
}
```

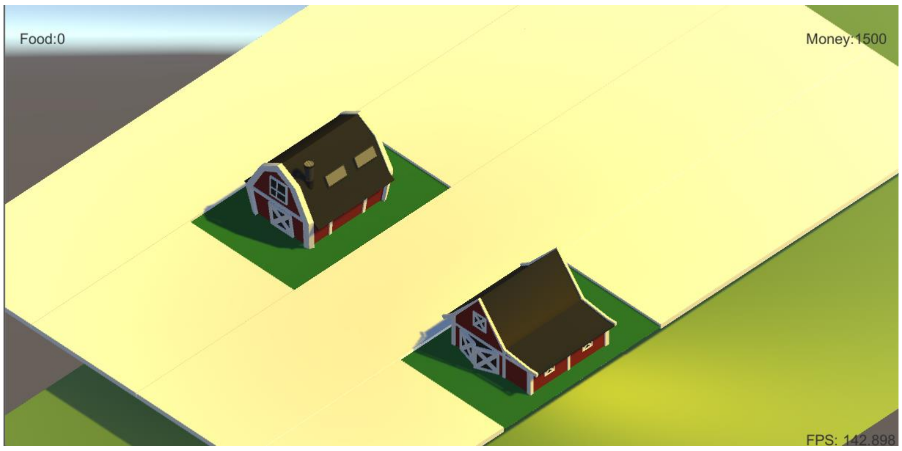
Putting what we currently had so far allowed us to then add switching each grid tile into another shape when having enough money. This would then be the basis for changing assets when the building was implemented, and assets were brought in.





```
public GameObject GetGridTile(int id)
{
    bool found = false;
    int count = 0;

    while(found == false)
    {
        if (gridSquares[count].GetComponent<ObjectInfo>().GetObjectID() == id)
        {
            found = true;
        }

        count += 1;
    }


    return gridSquares[count - 1];
}
```

Assets start being implemented – To mark the start of development in the game we wanted to make sure we had the basics of what our game would need. This included being able to import any assets made by the artists. Once we knew how bring them into Unity and place them into the grid, we could begin creating the game.



Development

Features Implemented and How

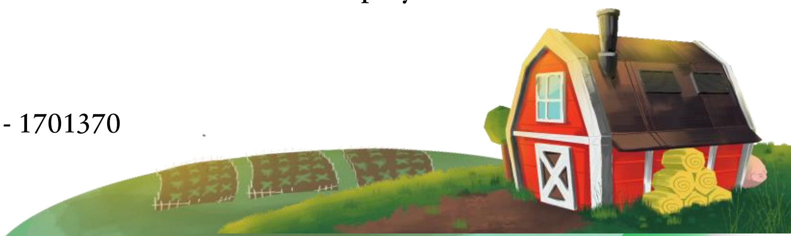
When adding in new features it was important to think about how the designers would be able to edit any values for testing. So, when adding a new feature, making the values visible in the Unity editor was on the priority list.

Grid – Based on the prototype for the grid creation the grid system was updated to have default positions of buildings for when it was loaded. This was also used in the tutorial to spawn fields and things to show off to the player what could be done in the game. It was created by looping through row and column values placing the blocks at each position, which was based on a spacing value for between each new asset. By doing it this way, it allowed for changing the grid to look the way we wanted it to. Making this one of the priority features at the beginning, gave us an idea of how our game would look when flying over the top of it.

Field Fillers – The system to instantiate what goes inside the fields, such as corn, cabbages and animals was created to stop the artists from having to create a field model for every type of filler. If this was not added, then there would be essentially 15 of the same models but with a different crop type inside. To stop this from being an issue we decided to use one field model for each field level and have the fills as separate assets. I created a script that would attach to the field asset that would be getting filled - i.e. the field/animal tiles - that holds multiple types of fillers, like Sunflowers, Wheat and Chicken. When attached, to select one of the fill types a drop down list of them is shown. The fill that is needed would be selected (or set via code). When changing/building a new asset, there is a switch statement to see what fill is attached to the field and depending on what it is, it will instantiate, from the resources, the correct fill as a child of and to the position of the field. Making it a child of the field was necessary so that when the asset is destroyed or when getting upgraded/changed the fill also gets destroyed with it.

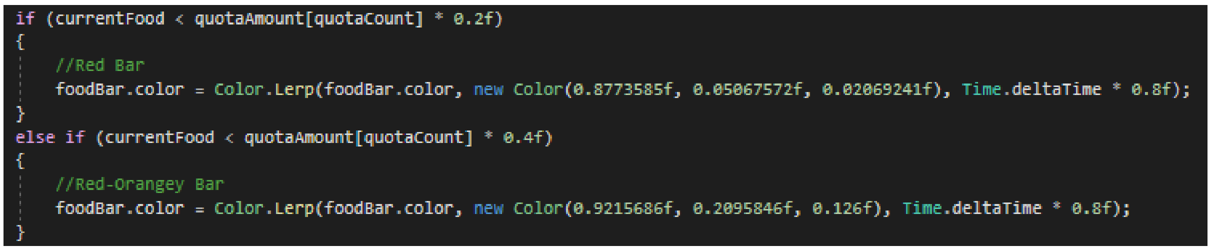
For the animal fills, instead of just instantiating at the position of the field, they are modified to be placed at set locations and rotated. The rotation the animal had to be converted into quaternions as that's how Unity instantiates prefabs.

Money and Food – Added the passive income to the game to use as a basis for the buying mechanic that we would soon implement. The food script used got dramatically edited to allow for the implementation of the quota system and to affect the whole game. Inside the food script it checks, after the quota time, if the amount of food the player has is



greater than what is being asked of them. Depending on what they have it will either go to the next amount, essentially progressing through the game, or it will add to a warning count. If the warning count was too high, then the game would load the previous save (changed to the end game in final build).

Quota System – A quota that needs to be filled within a designated time frame. An array was used to allow for the designers to add a custom number of quotas and timers into the game. A quota bar was added that fills whenever a field produces food and if the player earns enough food, after a time limit, then the quota amount changes to the next value in the array allowing for game to progress in a linear fashion. If the player earns over the quota amount, they gain money depending on how much they are over by. If the player fails, they get a warning. Depending on the value compared to the quota amount the bar changes colour to tell the player how close they are to the quota.



```
if (currentFood < quotaAmount[quotaCount] * 0.2f)
{
    //Red Bar
    foodBar.color = Color.Lerp(foodBar.color, new Color(0.8773585f, 0.05067572f, 0.02069241f), Time.deltaTime * 0.8f);
}
else if (currentFood < quotaAmount[quotaCount] * 0.4f)
{
    //Red-Orangey Bar
    foodBar.color = Color.Lerp(foodBar.color, new Color(0.9215686f, 0.2095846f, 0.126f), Time.deltaTime * 0.8f);
}
```

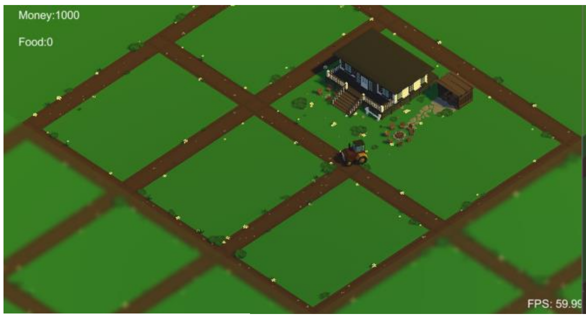
Touch Controls – There were some touch controls before I went in and changed them to make them feel more fluid and dynamic. Previously the touch controls did not have any zoom and the panning felt off. After watching a few tutorials and reading some forums(Unity Forums, no date) about how to best implement touch controls I came across a script that could be modified (GibsS, 2019). This allowed for the camera to be zoomed to the centre of the pinch motion using orthographic size of the camera. I only used parts needed in the script and made to sure to check if using the script was okay.

It used the Unity Event system to allow for touch to be recognised. If the player taps the screen with one finger and is not over any UI, then it gets where the player tapped and sends a ray to the position from the camera and if it intersects with a tile then the desired action will happen. If the player keeps their finger on the screen for over a specified time then it no longer will recognise it as a tap and will understand the press as a swipe and will move the camera to the new location. The location is specified by taking away the delta position of where the player is pressing from the current position of the camera.

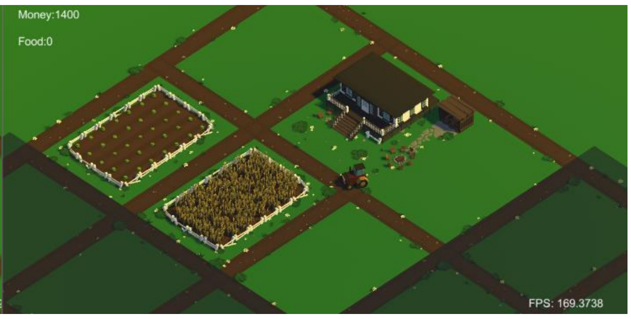
If the player is using 2 fingers then the camera get both positions and finds the distance between the two and depending on if the value is greater or lower when the fingers position moves, changes the orthographic size, acting as a zoom to the player.

There was a bug where you could tap through some menus which was fixed by adding a plane in the back of most UI screens. Another that took a while to get rid of was when you wanted to close the radial menu but the button was over a field it would open the close the current and immediately open another for the field under the button. This caused great confusion as to begin with we believed that it was the amount of time that would be used to recognise a tap. But upon changing the time it would not fix itself. It was only until after I implemented the tutorial and implementing coroutines, to wait until and action had been made, that I created the solution to have a coroutine that runs in the background that acts a timer to be able to open another menu after closing another. This solved the bug of being stuck in the loop of never-ending radial menus.

Locked Areas – For the locked areas there was a lot of debating about how to create an area that was out of bounds until later in the game. The original idea that I suggested was to have the assets be a different colour, maybe greyed out, to show that they were not available to use. They would also have construction tape over it. When trying to select this asset it would show a popup saying that it was a locked area. This idea did not come to fruition as when changing the material, it did not fit in with the game and looked wrong due to the way the grid was set up. It was at the end of a long day that I came up with the idea that there could be two planes over the top of the area that could be unlocked and to put a blur shader material on them so that it would create intrigue as to what was behind it. The way it is set up is that when changing the asset, there is a check to see if the type of tile that is being changed is the farmhouse and if so, checks what level the farmhouse is currently. Depending on the level of the farmhouse the plane that correlated to the levels area is destroyed, effectively unlocking a new area to the player, and giving them the barn to upgrade and progress further. The same again for the next level. The plane also acts as a barrier between selecting anything in those areas and saved time in writing code to not allow selection. In the end the blur got changed and a transparent material replaced it.

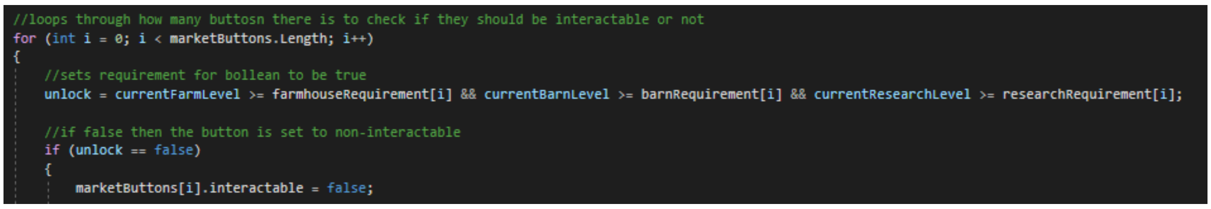


Blurred Areas



Non -Blurred Areas

Marketplace Unlocking – The team agreed that the game needed some progression so so one of the ideas was to lock what could be built behind the building levels. Due to the marketplace mostly being buttons, I decided to create a script that took those buttons and set requirements, that could be accessed by the designers, for them to be interactable. This script attached to the marketplace and when spawned checks the building levels and runs through all the buttons and checks the requirements for them to be shown to the player. If the requirements were not met, then the button would be set to be non-interactable.

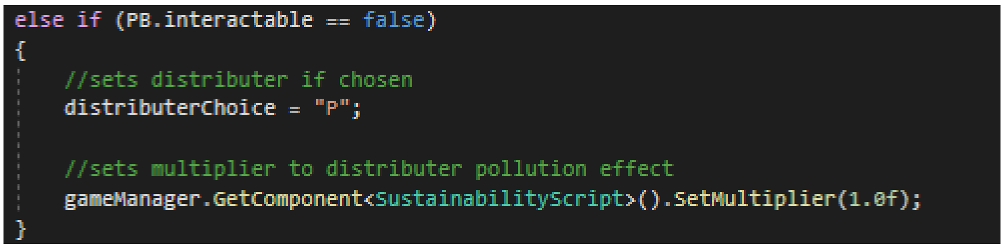


```
//loops through how many buttosn there is to check if they should be interactable or not
for (int i = 0; i < marketButtons.Length; i++)
{
    //sets requirement for bollean to be true
    unlock = currentFarmLevel >= farmhouseRequirement[i] && currentBarnLevel >= barnRequirement[i] && currentResearchLevel >= researchRequirement[i];

    //if false then the button is set to non-interactable
    if (unlock == false)
    {
        marketButtons[i].interactable = false;
```

Spawning a padlock image on to locked items was added through code after the system was in place. It is spawned as the child of the locked item and placed into the bottom right corner.

For the tutorial, these checks are overwritten so that the buttons available are only for the current events.

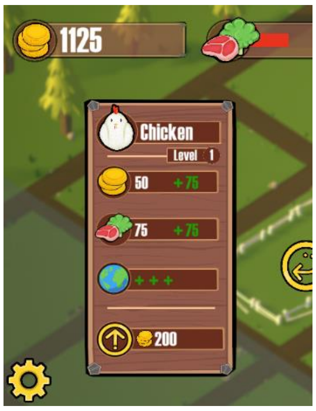
Distribution Options – When thinking about how to implement the distribution aspect of the brief it was decided throughout the group that there would be a menu to choose who to sell your food to and depending on the choice it would have either a good or bad effect on the farm. During development, this feature was slowly pushed to the back as key features took priority. After discussing how to implement the menu into the game with Andrew (Artist) a mock-up was made by him and he asked if it was possible to have it sliding in from the side. He created a quick animation for me to re-create in Unity. After adding the distribution UI, and the animation for it, I added the functionality to affect the food and money income. I made it so that if the button to choose a distributer was selected then that button would be set to not-interactable. In the code when the button is pressed it checks what buttons are interactable, if one is false (the selected one) then it sets the distributer.

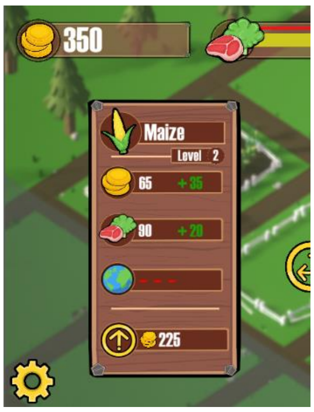


```
else if (PB.interactable == false)
{
    //sets distributer if chosen
    distributerChoice = "P";

    //sets multiplier to distributer pollution effect
    gameManager.GetComponent<SustainabilityScript>().SetMultiplier(1.0f);
}
```

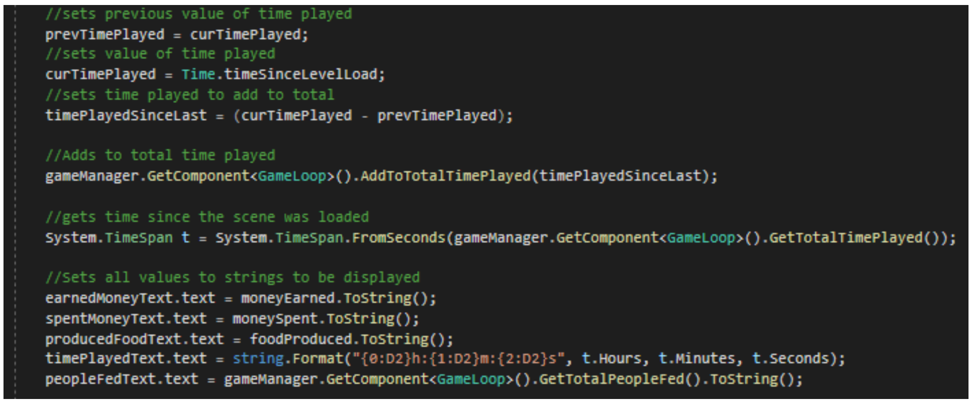
Then inside the food script, where the income is changed, the distributer affects are then applied. So, if the player chooses "Big Farma" (the conglomerate company) then the money and food income is increased. Choosing the greener option of "Go Green" will make the player's sustainability level go up but their money and food income go down. The sustainability affects were added by David (programmer) to make the pollution value be bigger or smaller depending on the choice.

Statistics of Selected Tile – When the original radial menu would come up to choose what to do to the selected tile, it was only the buttons that would show, I felt that it was too empty and that the player needed some feedback. I asked the UI artist (Dan) to create a statistic pop up to show information like Name, Upgrade Cost, Sustainability, and what values would change if it were to be upgraded. After it was created, I implemented it into the game to pop-up when the radial menu is shown. I then created a script to gather the data to show the player. It takes the values gathered and replaces text boxes attached to the script for that specific selected tile. I feel as though this greatly increases the player feedback as before, this was created and implemented, there was no way of knowing what the cost of the different levels of the tile were and the impact the tile in the game.



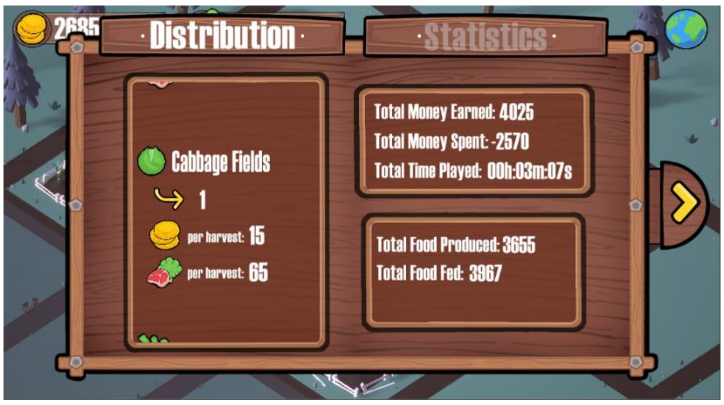


Statistics of Farm – Along the same lines of the statistics of the selected tile, I felt as though there was no way of knowing what was happening on your farm. So, I suggested that there should be a statistics page and it could possibly be in the same place as the distribution menu. As the swiping of that menu felt more mobile gamey than going into different menus and so there was more that specific area. It was suggested to me from an artist (Dan) that a part of the menu could be a scrolling menu to allow for the information to be bigger but still have all the relevant information to show off. I looked at quick video that showed me how to implement the scrolling menu and then went ahead and added it along with gathering all the stats. The gathering of the stats is called when the player opens up the distribution menu as to save performance with not always trying to get new stats when something is updated. When the player presses the button, a function is called which calls all the gets needed. The values are then parsed into strings so that they can be set.



```
//sets previous value of time played
prevTimePlayed = curTimePlayed;
//sets value of time played
curTimePlayed = Time.timeSinceLevelLoad;
//sets time played to add to total
timePlayedSinceLast = (curTimePlayed - prevTimePlayed);

//Adds to total time played
gameManager.GetComponent<GameLoop>().AddToTotalTimePlayed(timePlayedSinceLast);

//gets time since the scene was loaded
System.TimeSpan t = System.TimeSpan.FromSeconds(gameManager.GetComponent<GameLoop>().GetTotalTimePlayed());

//Sets all values to strings to be displayed
earnedMoneyText.text = moneyEarned.ToString();
spentMoneyText.text = moneySpent.ToString();
producedFoodText.text = foodProduced.ToString();
timePlayedText.text = string.Format("{0:D2}h:{1:D2}m:{2:D2}s", t.Hours, t.Minutes, t.Seconds);
peopleFedText.text = gameManager.GetComponent<GameLoop>().GetTotalPeopleFed().ToString();
```

Snippet of code showing the parsing of the values to show in the menu



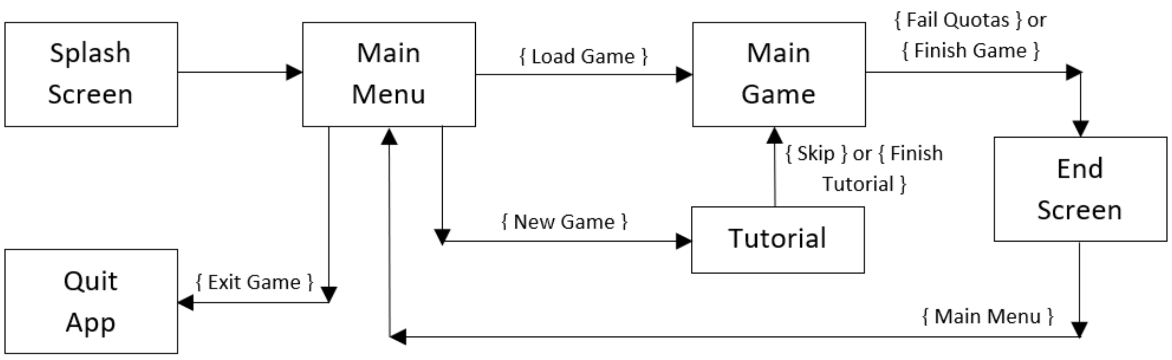
Total Money Earned: 4025
Total Money Spent: -2570
Total Time Played: 00h:03m:07s

Total Food Produced: 3655
Total Food Fed: 3967

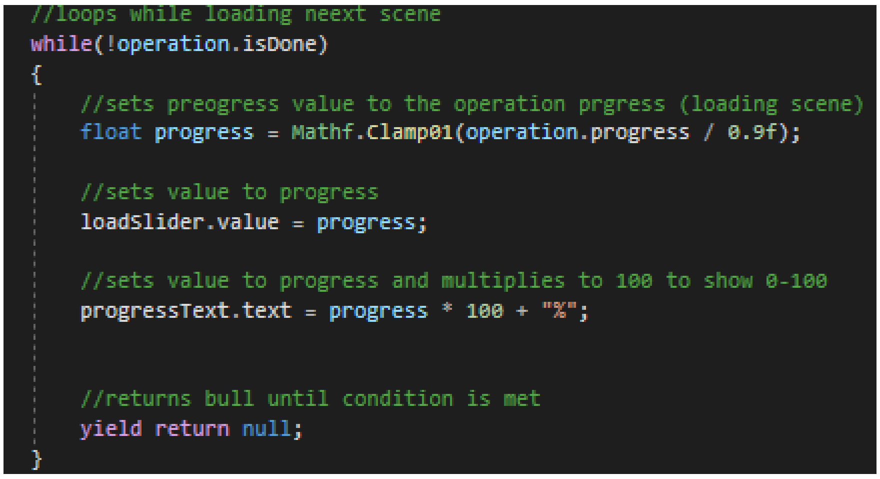
Stats in the final menu

These values are then passed into a function that stores them to show in the end screen so that the end screen is up to date when the player gets to it.

Switching Scenes – There are a few ways that the player changes the scene. Below is a diagram of the ways this can happen. This shows the loop of the game also.



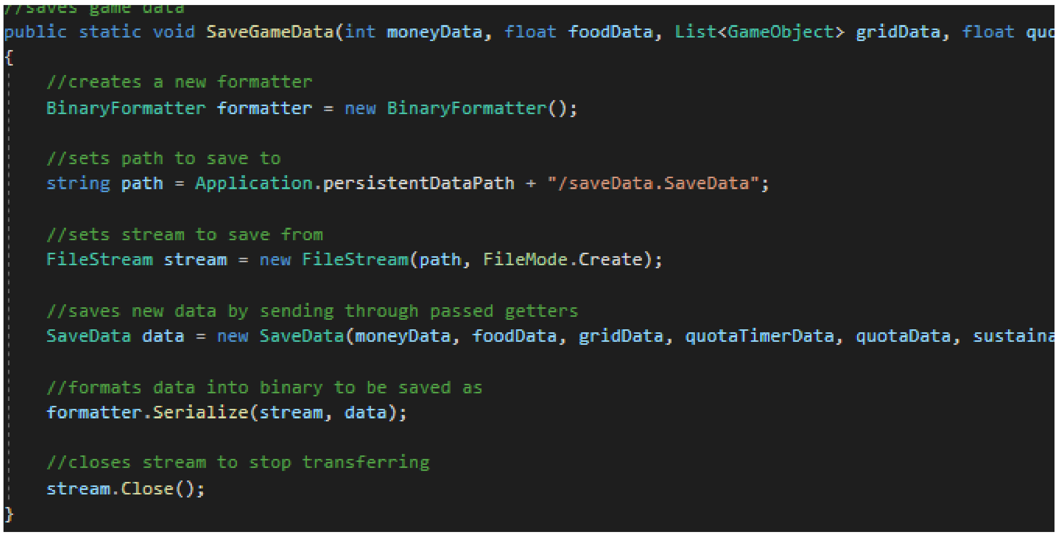
Loading Screen – Adding to the switching scenes section the loading screen was created by me. The way it works is that it takes an operation (loading and unloading scenes) and updates a bar and value depending on how far the operation is through.



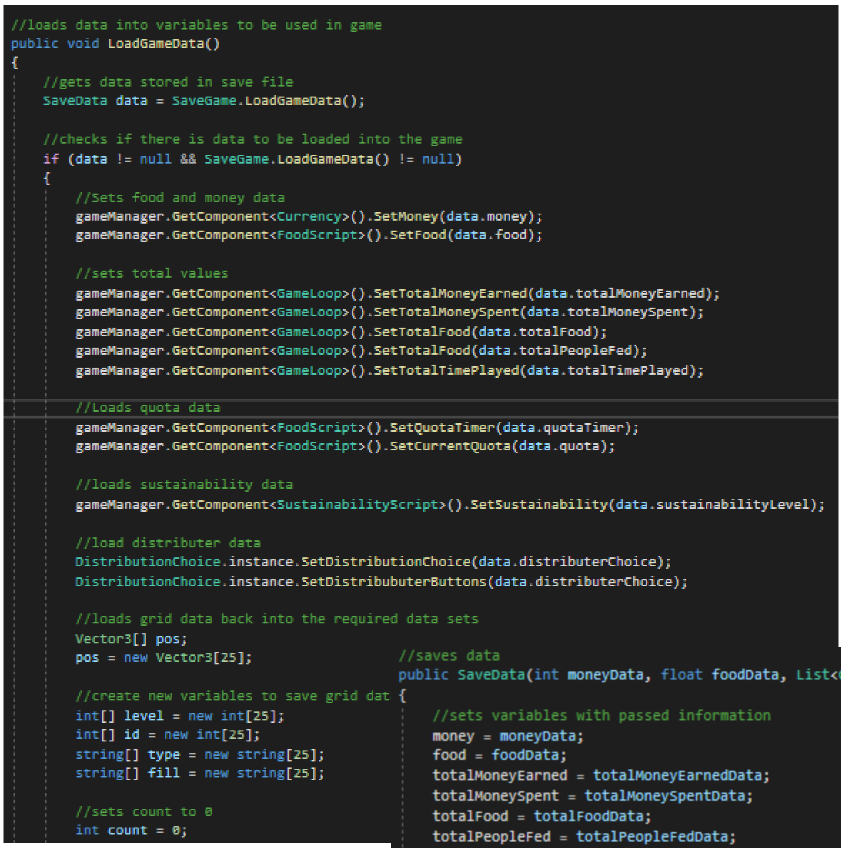
```
//loops while loading neext scene
while(!operation.isDone)
{
    //sets preogress value to the operation prgress (loading scene)
    float progress = Mathf.Clamp01(operation.progress / 0.9f);

    //sets value to progress
    loadSlider.value = progress;

    //sets value to progress and multiplies to 100 to show 0-100
    progressText.text = progress * 100 + "%";


    //returns bull until condition is met
    yield return null;
}
```

Saving/Loading Game – David and I, in passing, were talking about methods of how to save the game data and were thinking about just saving it to a text file. When looking into this I realised that it would be a lot of tedious code writing, so I searched for a better way. I looked at a few videos (mainly Brackeys) on how to best set this up. It was clear that to save on multiple devices (like PC and Mobile) we would have to create our own save file using Binary Formatting. Unity uses binary serialisation (converts an object into a long stream of 1's and 0's) to convert game data into a binary file to save.

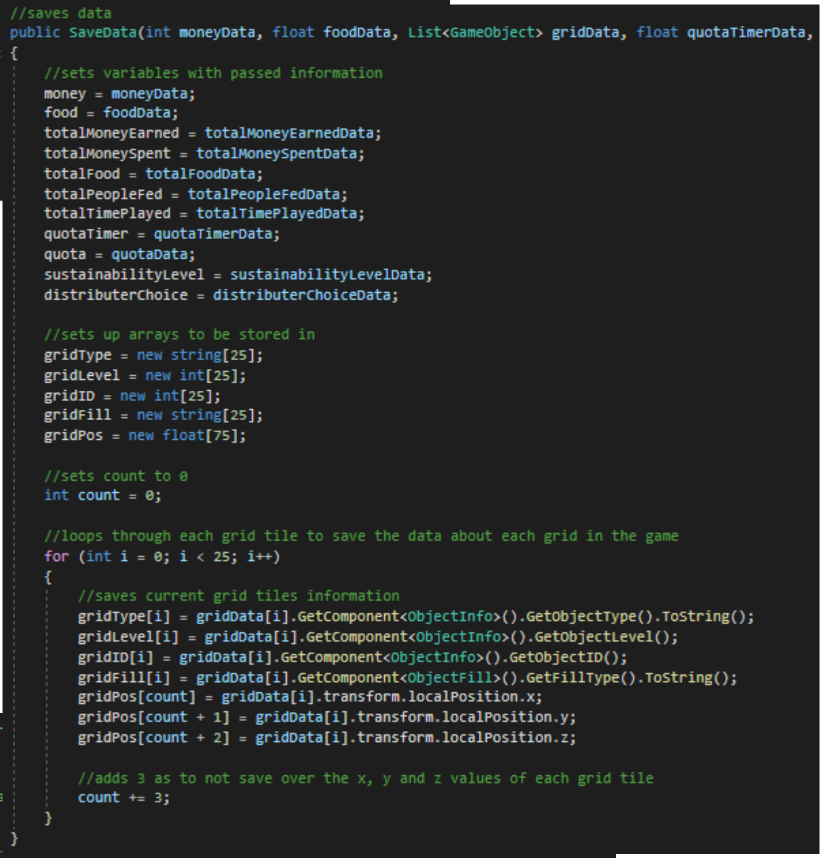
When loading, Unity converts back into a normal format to be used in game. This allows for the game to be closed and then when started back up lets the user load their progress and continue.



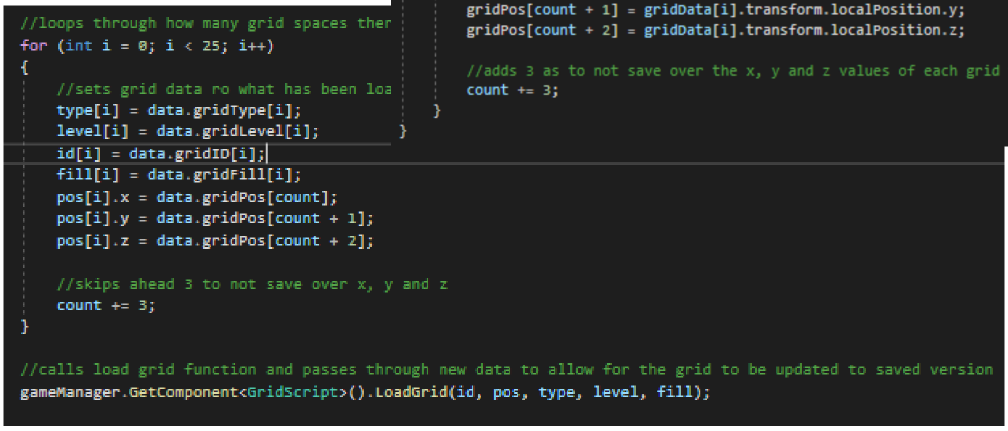
```
//saves game data
public static void SaveGameData(int moneyData, float foodData, List<GameObject> gridData, float quo
{
    //creates a new formatter
    BinaryFormatter formatter = new BinaryFormatter();

    //sets path to save to
    string path = Application.persistentDataPath + "/saveData.SaveData";

    //sets stream to save from
    FileStream stream = new FileStream(path, FileMode.Create);

    //saves new data by sending through passed getters
    SaveData data = new SaveData(moneyData, foodData, gridData, quotaTimerData, quotaData, sustaina

    //formats data into binary to be saved as
    formatter.Serialize(stream, data);

    //closes stream to stop transferring
    stream.Close();
}
```



```
//loads data into variables to be used in game
public void LoadGameData()
{
    //gets data stored in save file
    SaveData data = SaveGame.LoadGameData();

    //checks if there is data to be loaded into the game
    if (data != null && SaveGame.LoadGameData() != null)
    {
        //Sets food and money data
        gameManager.GetComponent<Currency>().SetMoney(data.money);
        gameManager.GetComponent<FoodScript>().SetFood(data.food);

        //sets total values
        gameManager.GetComponent<GameLoop>().SetTotalMoneyEarned(data.totalMoneyEarned);
        gameManager.GetComponent<GameLoop>().SetTotalMoneySpent(data.totalMoneySpent);
        gameManager.GetComponent<GameLoop>().SetTotalFood(data.totalFood);
        gameManager.GetComponent<GameLoop>().SetTotalFood(data.totalPeopleFed);
        gameManager.GetComponent<GameLoop>().SetTotalTimePlayed(data.totalTimePlayed);

        //Loads quota data
        gameManager.GetComponent<FoodScript>().SetQuotaTimer(data.quotaTimer);
        gameManager.GetComponent<FoodScript>().SetCurrentQuota(data.quota);

        //loads sustainability data
        gameManager.GetComponent<SustainabilityScript>().SetSustainability(data.sustainabilityLevel);

        //load distributer data
        DistributionChoice.instance.SetDistributionChoice(data.distributerChoice);
        DistributionChoice.instance.SetDistribubuterButtons(data.distributerChoice);

        //loads grid data back into the required data sets
        Vector3[] pos;
        pos = new Vector3[25];

        //create new variables to save grid dat
        int[] level = new int[25];
        int[] id = new int[25];
        string[] type = new string[25];
        string[] fill = new string[25];

        //sets count to 0
        int count = 0;
```



```
//saves data
public SaveData(int moneyData, float foodData, List<GameObject> gridData, float quotaTimerData,
{
    //sets variables with passed information
    money = moneyData;
    food = foodData;
    totalMoneyEarned = totalMoneyEarnedData;
    totalMoneySpent = totalMoneySpentData;
    totalFood = totalFoodData;
    totalPeopleFed = totalPeopleFedData;
    totalTimePlayed = totalTimePlayedData;
    quotaTimer = quotaTimerData;
    quota = quotaData;
    sustainabilityLevel = sustainabilityLevelData;
    distributerChoice = distributerChoiceData;

    //sets up arrays to be stored in
    gridType = new string[25];
    gridLevel = new int[25];
    gridID = new int[25];
    gridFill = new string[25];
    gridPos = new float[75];

    //sets count to 0
    int count = 0;

    //loops through each grid tile to save the data about each grid in the game
    for (int i = 0; i < 25; i++)
    {
        //saves current grid tiles information
        gridType[i] = gridData[i].GetComponent<ObjectInfo>().GetObjectType().ToString();
        gridLevel[i] = gridData[i].GetComponent<ObjectInfo>().GetObjectLevel();
        gridID[i] = gridData[i].GetComponent<ObjectInfo>().GetObjectID();
        gridFill[i] = gridData[i].GetComponent<ObjectFill>().GetFillType().ToString();
        gridPos[count] = gridData[i].transform.localPosition.x;
        gridPos[count + 1] = gridData[i].transform.localPosition.y;
        gridPos[count + 2] = gridData[i].transform.localPosition.z;

        //adds 3 as to not save over the x, y and z values of each grid tile
        count += 3;
    }
}
```



```
//loops through how many grid spaces ther
for (int i = 0; i < 25; i++)
{
    //sets grid data ro what has been loa
    type[i] = data.gridType[i];
    level[i] = data.gridLevel[i];
    id[i] = data.gridID[i];
    fill[i] = data.gridFill[i];
    pos[i].x = data.gridPos[count];
    pos[i].y = data.gridPos[count + 1];
    pos[i].z = data.gridPos[count + 2];

    //skips ahead 3 to not save over x, y and z
    count += 3;
}

//calls load grid function and passes through new data to allow for the grid to be updated to saved version
gameManager.GetComponent<GridScript>().LoadGrid(id, pos, type, level, fill);
```

The money and food values were tested first to make sure the saving and loading worked and when they did, I started on getting the entire grid to save. This included the type of tile, fill, ID, level, and position. One of the problems with Unity serialisation is that some Unity data types cannot be formatted so they had to be converted into a formattable data type. This meant a new script had to be created to be an in-between script before the formatting takes place. Throughout development the saving and loading scripts were updated to include other values to be saved.

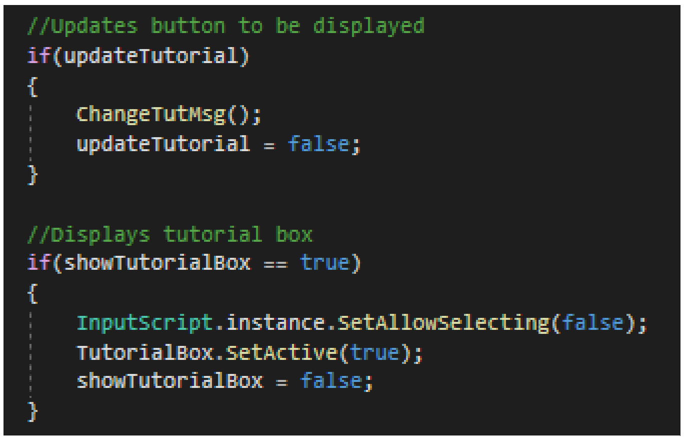
The save file is saved onto the device that the game is playing on using Unity's persistent data path method. The file can be manually deleted by the user when they go into their files. E.g. on Windows PC it gets saved in the directory:

"%userprofile%\AppData\LocalLow\Ellipsis\Greener Pastures"

Seasonal Colour Change – Our mentor mentioned about maybe changing the colour of the materials to act as a day/night cycle but as the development got underway that idea got moved to the back of the priority list. It was not until later that I decided to mess with changing the material colours as a little mess about that the idea of having different seasons in the game that would change depending on how long the player has been playing for. It started with using the lerp function in Unity to change the grass material to a lighter hue to act as summer and then into autumn by changing into an orangey-green. I then messed about with different lighting colours to achieve the right look to the season. After adding in the last two seasons (spring – yellow tint and winter – blue-green) I noticed that when it was changing from season to season that it was taking longer as it was slowing down at the end of the lerp. I wanted it to be as linear as possible. This was due to the lerp function wanting a value between 0 and 1 but I was using delta time multiplied by a speed value which would never move towards 1. The solution was to add a new variable that started at a value of 0. And would get increased by adding delta time and a speed multiplier. This would make the transition look more linear. A downside of this solution was that it was quicker than intended so to adjust the code for this the speed multiplier had to be dramatically decreased. As a small detail I made it so the trees changed colour slower than the ground

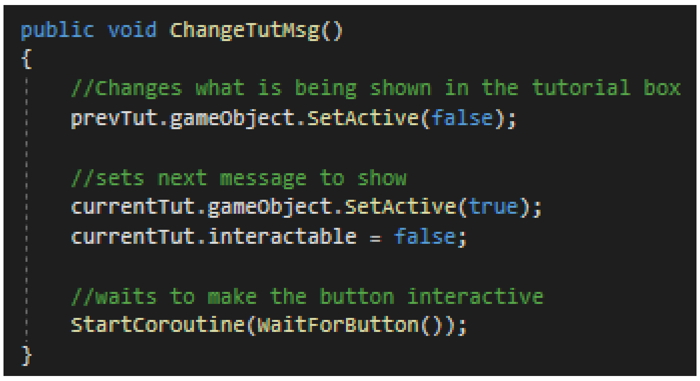
Tutorial Section of Game

The functionality of the tutorial was created solely by me. All the key features were already in the game before starting on the tutorial, so it was creating the events that took the longest amount of time and then polishing it to get rid of any bugs. I also had to go through most of the scripts and add a check to see if the game was currently in the tutorial. Adding the tutorial messages from “Mac”, the guide, to help the player understand all the components of the game was the priority when first creating the tutorial. I made a checklist of what had to be covered and started on making a manager script to deal with changing the tutorial messages.



```
//Updates button to be displayed
if(updateTutorial)
{
    ChangeTutMsg();
    updateTutorial = false;
}

//Displays tutorial box
if(showTutorialBox == true)
{
    InputScript.instance.SetAllowSelecting(false);
    TutorialBox.SetActive(true);
    showTutorialBox = false;
}
```

How the message box gets updated

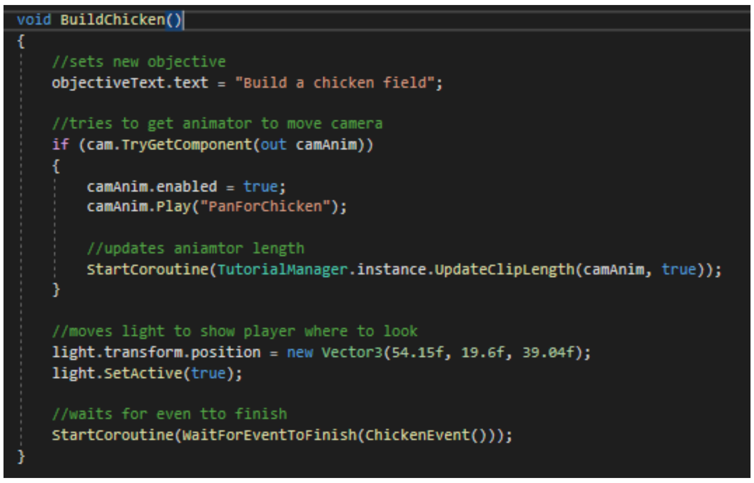


```
public void ChangeTutMsg()
{
    //Changes what is being shown in the tutorial box
    prevTut.gameObject.SetActive(false);

    //sets next message to show
    currentTut.gameObject.SetActive(true);
    currentTut.interactable = false;

    //waits to make the button interactive
    StartCoroutine(WaitForButton());
}
```

Changes message button to show

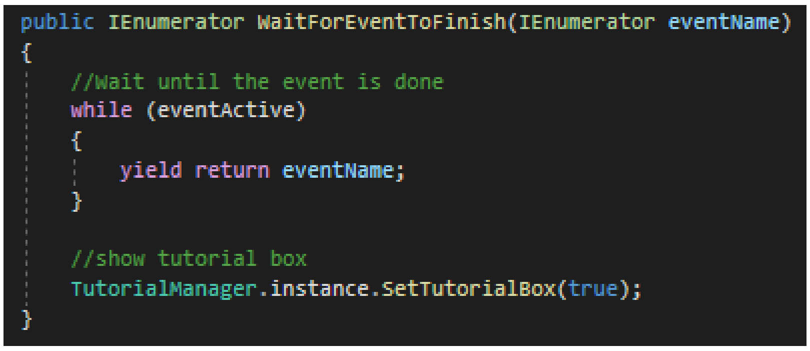
After all the text and messages worked from start to finish. I then created the tutorial grid to be instantiated and to show off what the farm could look like tot the player. This technique was taken from other games, like Need for Speed Most Wanted(Criterion Software, 2012) or Forza Horizon 4(Playground, 2018), where you would be given a super-car in the first race to familiarise you to the game and then it gets taken away and you start with a not-so-fast car and have to work your way up. I then started working on the events that player would go through to learn about the game and what can be done.

To start with, the player must build a field.



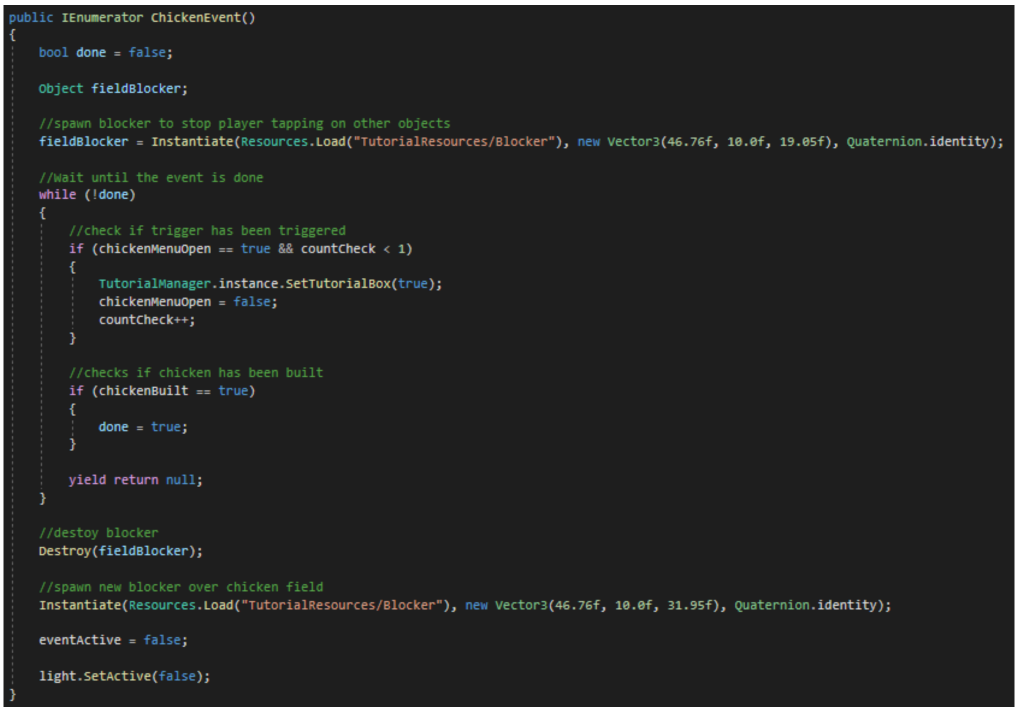
```
void BuildChicken()
{
    //sets new objective
    objectiveText.text = "Build a chicken field";

    //tries to get animator to move camera
    if (cam.TryGetComponent(out camAnim))
    {
        camAnim.enabled = true;
        camAnim.Play("PanForChicken");

        //updates aniamtor length
        StartCoroutine(TutorialManager.instance.UpdateClipLength(camAnim, true));
    }

    //moves light to show player where to look
    light.transform.position = new Vector3(54.15f, 19.6f, 39.04f);
    light.SetActive(true);

    //waits for even tto finish
    StartCoroutine(WaitForEventToFinish(ChickenEvent()));
}
```

Similar structure of code was used for other events



```
public IEnumerator WaitForEventToFinish(IEnumerator eventName)
{
    //Wait until the event is done
    while (eventActive)
    {
        yield return eventName;
    }

    //show tutorial box
    TutorialManager.instance.SetTutorialBox(true);
}
```

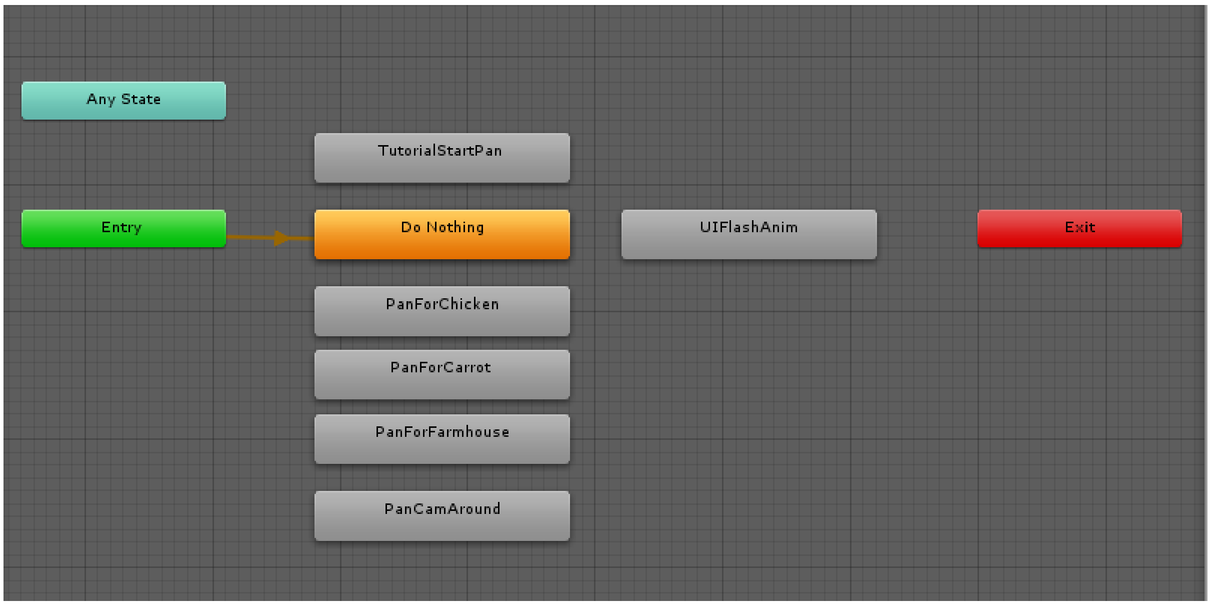
Waits for the current event to finish



```
public IEnumerator ChickenEvent()
{
    bool done = false;

    Object fieldBlocker;

    //spawn blocker to stop player tapping on other objects
    fieldBlocker = Instantiate(Resources.Load("TutorialResources/Blocker"), new Vector3(46.76f, 10.0f, 19.05f), Quaternion.identity);

    //Wait until the event is done
    while (!done)
    {
        //check if trigger has been triggered
        if (chickenMenuOpen == true && countCheck < 1)
        {
            TutorialManager.instance.SetTutorialBox(true);
            chickenMenuOpen = false;
            countCheck++;
        }

        //checks if chicken has been built
        if (chickenBuilt == true)
        {
            done = true;
        }

        yield return null;
    }

    //destoy blocker
    Destroy(fieldBlocker);

    //spawn new blocker over chicken field
    Instantiate(Resources.Load("TutorialResources/Blocker"), new Vector3(46.76f, 10.0f, 31.95f), Quaternion.identity);

    eventActive = false;

    light.SetActive(false);
}
```

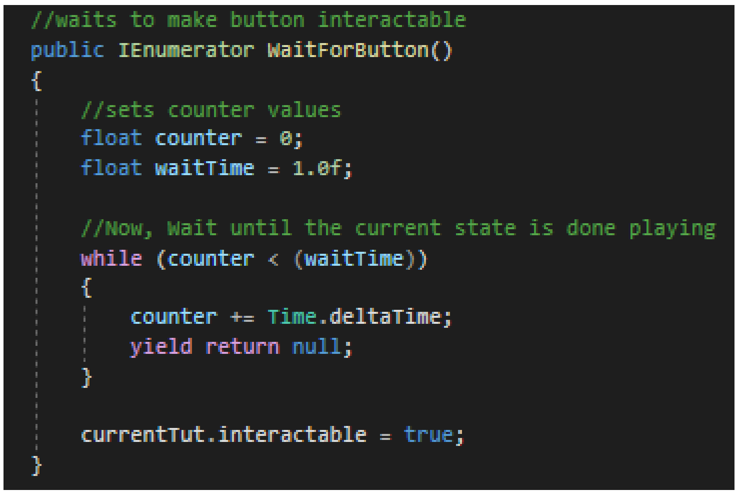
Again a similar structure of code was used for other events

This shows them the different menus in the game and how to build. Giving them the knowledge of how to do so they are then prompted to build another but without the help from before allowing them to act freely and understand what can be done. The player is then shown how to destroy a field. After telling them the basics of the game the tutorial moves onto telling them about their farm and how to unlock and upgrade their fields. The camera panning was creating using the animator in Unity and there are multiple animations in the controller that get called to play at certain events.



Tutorial Animations

To stop the player form spamming the tutorial and skipping past parts, there is a waiting function that is called when the button is changed to the next message.



```
//waits to make button interactable
public IEnumerator WaitForButton()
{
    //sets counter values
    float counter = 0;
    float waitTime = 1.0f;

    //Now, Wait until the current state is done playing
    while (counter < (waitTime))
    {
        counter += Time.deltaTime;
        yield return null;
    }

    currentTut.interactable = true;
}
```

The tutorial continues through different events showing the player the distribution and telling them about the game. It is also hinted to the player about there actions effecting the environment.

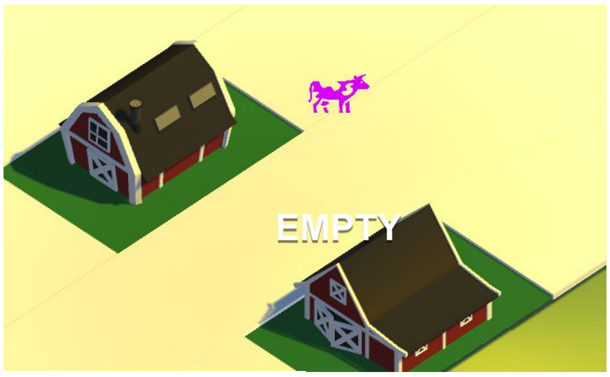
Code/Scripts

It was decided between David (other programmer) and I that we would go over coding standards such as variable names and using camelCase to make sure the source code would be clean to read and easy to understand for both sides when something was needed changed or to look at the other persons code. Other standards we agreed on was to comment frequently and to describe new code but not for easy to understand functions.

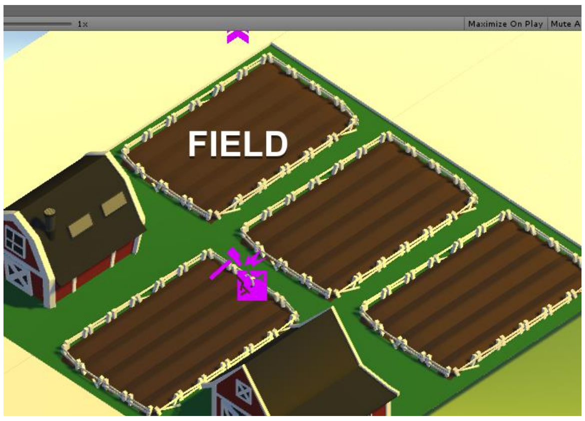
Things I helped with throughout Development

Adding Assets – When the Artists had the assets ready to bring into the engine, I helped with importing and attaching any scripts that would have to be applied to them and making sure they had the correct information to work with the scripts and game.

Radial Menu – The menu was created by Patrick (Team Lead). Before, the radial menu to show options on what to do with your selected tile would come up where you pressed/clicked, and you would have to hold down for it to stay up. This caused a few problems. One where if you pressed too high up on the screen. The menu would show as half in/half out the screen. Another was that it would stop the panning of the camera if the player were to swipe.

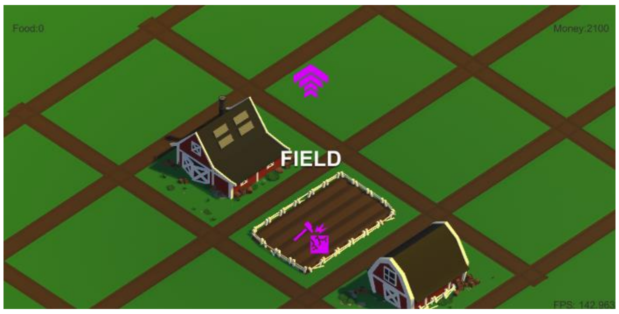


Radial Menu



Showing menu out of screen

I decided to fix the radial menu's issues after speaking to the team about the problems it had. I made it so the menu would now show up after registering a tap instead of requiring the player to hold. This fixed the issue with it overriding the swipe when on a mobile. To stop the menu going out the screen, I had to sacrifice the menu opening where the player tapped and made it, so the menu was created in the centre of the screen with the intention to go back and make it come up but with boundaries. Unfortunately, other features took priority as this was working in the final build the menu spawns in the centre.



Fixed radial menu

Sound – Added volume menu to allow for the designers to go in and add functionality. Helped with adding in destruction sound due to menu being spawned through code so had to make the sound play through code.

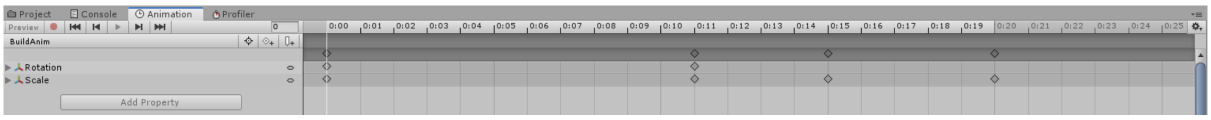
Fixing UI – Making sure the UI was correct was important. Whether it was placement on screen or adding functionality. I would go through each menu and make sure that it was not blocking any gameplay. I created the tutorial pop up and made it so that it was pressable to continue. Made scripts for gathering stats in statistical menus.

Any problems that arose from the game or technical difficulties.

In-Engine Animation

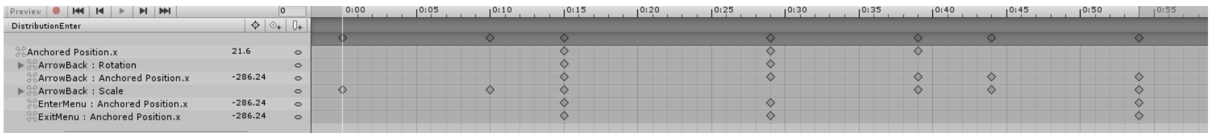
When adding the assets to the game it was unsure for part of development if animation was going to be a viable due to ongoing issues with importing animations with the assets. It was mostly agreed on that there should be animations in the game, but it was not designated as to who should do so. After some research on how to create some simple animation in Unity I decided to go ahead and add in an animation to the placeholder tractor that was currently static in the scene. This was liked by the team so was integrated into the game. So, it was decided that I would add any of the simpler animations in-engine due to previous problems with importing an asset with an animation attached. This spawned the animations for the Tractor, Building, Destroying, Field Sprayers, Distribution Swipe in/out, Light Flashing (to show where to build in the tutorial) and the camera pans (also for tutorial). These animations gave the game some more life as before it seemed desolate.

Some of the animation timelines



```
BuildAnim
Rotation
Scale
```

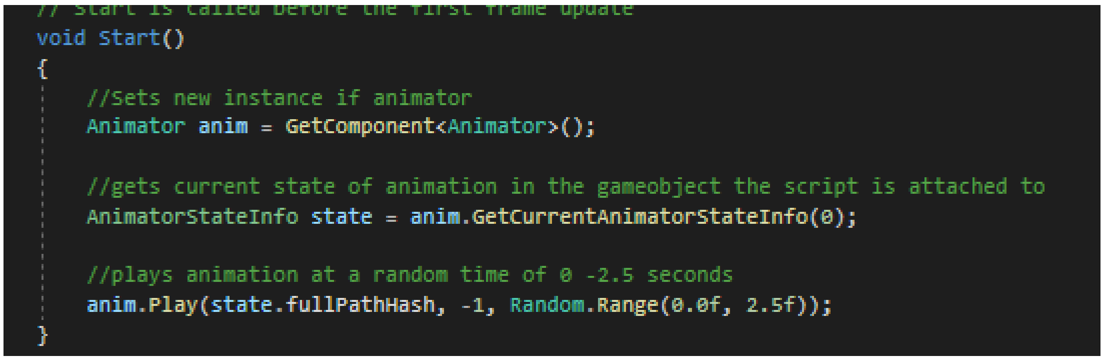
Build animation timeline



```
DistributionEnter
Anchored Position.x                  21.6
ArrowBack : Rotation
ArrowBack : Anchored Position.x      -286.24
ArrowBack : Scale
EnterMenu : Anchored Position.x      -286.24
ExitMenu : Anchored Position.x       -286.24
```

Distribution Animation

I also wanted the animations to appear random so I added a script that would attach to anything that would look repetitive, like the animals or the sprayer on the level 2 fields.



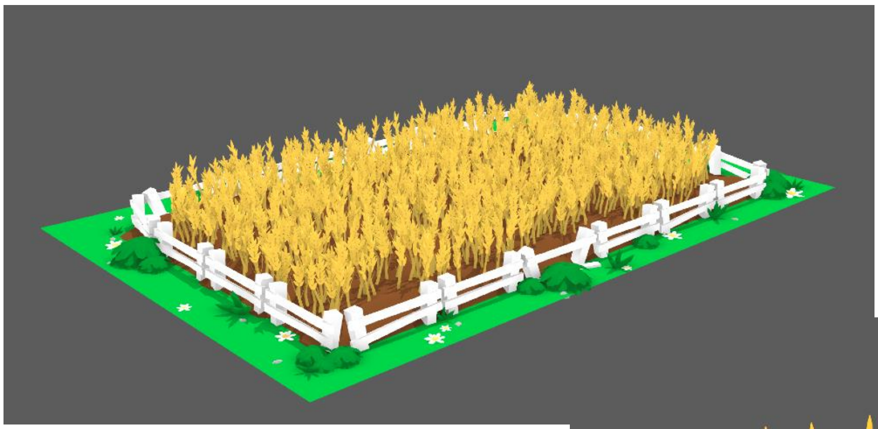
```
// Start is called before the first frame update
void Start()
{
    //Sets new instance if animator
    Animator anim = GetComponent<Animator>();

    //gets current state of animation in the gameobject the script is attached to
    AnimatorStateInfo state = anim.GetCurrentAnimatorStateInfo(0);

    //plays animation at a random time of 0 -2.5 seconds
    anim.Play(state.fullPathHash, -1, Random.Range(0.0f, 2.5f));
}
```

Profiling – Bug Fixing

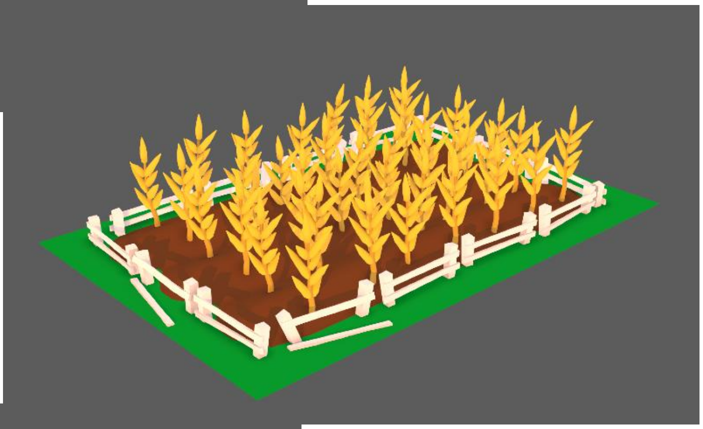
In week 7, when we had most of the assets in the game and tested the game on mobile, we ran into performance issues. When using Unity's profiler to test why, we noticed that the verts in the game were over 1 million. We tested taking different assets out and concluded that the artists would have to go back and make them have less. After some researching on how many verts should be the target, we decided to aim for 500,000. This was because in the grid we use in-game there are a max of 25 field tiles. Using the number of vertices, they already had and the number they could bring it down to and still be viable asset, we set the max number of vertices the tile could have to 20,000. This helped the performance significantly but ultimately delayed any progress due to the going back and forth testing performance until it was stable.

Using shared materials across all the assets also increased performance as it meant that we could batch the draw calls together and use GPU instancing to lower the amount of draw calls needed to render the scene. The frame debugger helped with this debugging dramatically. (In the zip folder there are a few videos showing the frame debugger going frame by frame to render the scene).

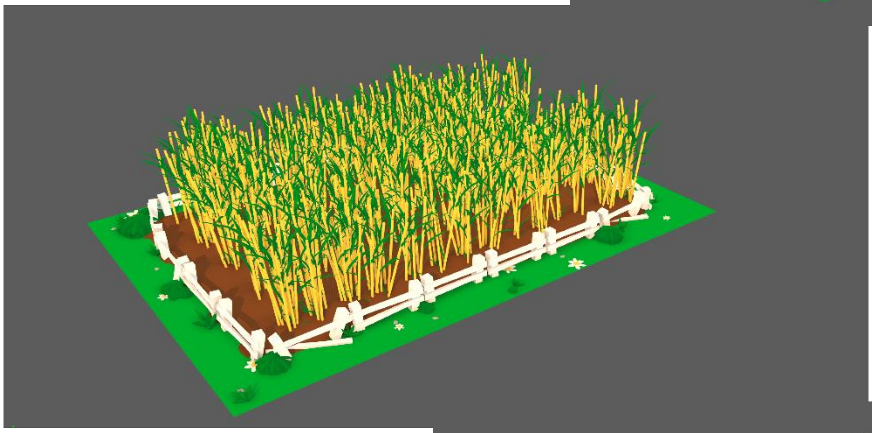
Some of the assets causing problems shown below to see difference after getting them changed.



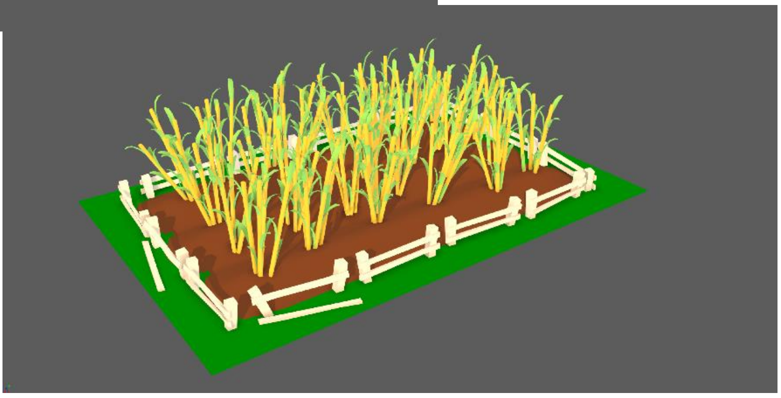
High Poly



Low Poly

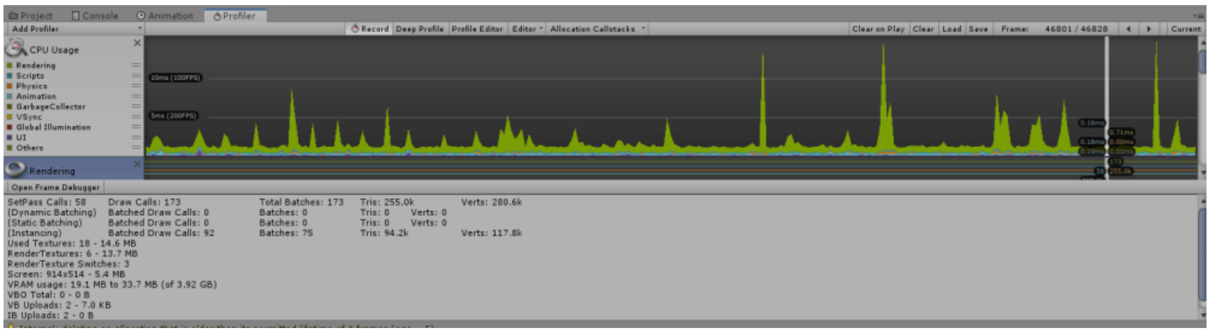


High Poly



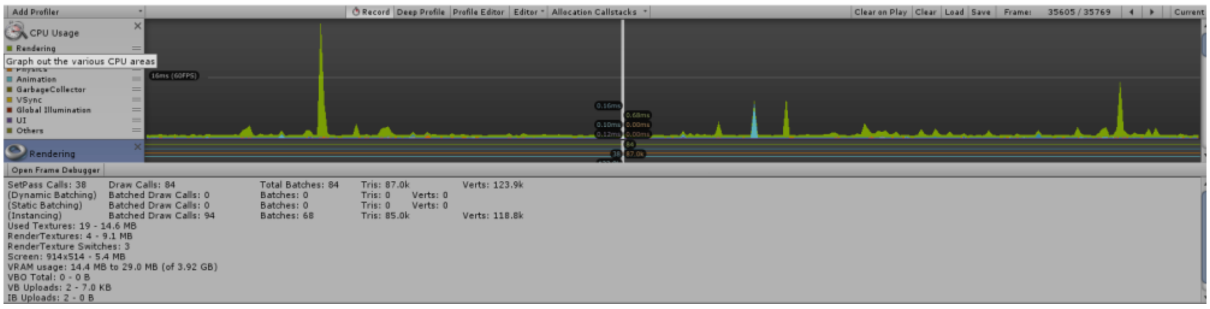
Low Poly

Some profiling screenshots that were taken when adding animals in:



```
SetPass Calls: 58      Draw Calls: 173              Total Batches: 173   Tris: 255.0k          Verts: 280.6k
(Dynamic Batching)     Batched Draw Calls: 0        Batches: 0           Tris: 0     Verts: 0
(Static Batching)      Batched Draw Calls: 0        Batches: 0           Tris: 0     Verts: 0
(Instancing)           Batched Draw Calls: 92       Batches: 75          Tris: 94.2k           Verts: 117.8k
Used Textures: 18 - 14.6 MB
RenderTextures: 6 - 13.7 MB
RenderTexture Switches: 3
Screen: 914x514 - 5.4 MB
VRAM usage: 19.1 MB to 33.7 MB (of 3.92 GB)
VBO Total: 0 - 0 B
VB Uploads: 2 - 7.0 KB
IB Uploads: 2 - 0 B
```

No Animals



```
SetPass Calls: 38      Draw Calls: 84               Total Batches: 84    Tris: 87.0k           Verts: 123.9k
(Dynamic Batching)     Batched Draw Calls: 0        Batches: 0           Tris: 0     Verts: 0
(Static Batching)      Batched Draw Calls: 0        Batches: 0           Tris: 0     Verts: 0
(Instancing)           Batched Draw Calls: 94       Batches: 68          Tris: 85.0k           Verts: 118.8k
Used Textures: 19 - 14.6 MB
RenderTextures: 4 - 9.1 MB
RenderTexture Switches: 3
Screen: 914x514 - 5.4 MB
VRAM usage: 14.4 MB to 29.0 MB (of 3.92 GB)
VBO Total: 0 - 0 B
VB Uploads: 2 - 7.0 KB
IB Uploads: 2 - 0 B
```

With animals

Technical Design Document

David would update the TDD throughout development and I would add parts to it when necessary.

See TDD in ZIP folder.

References

Unity vs. Unreal: How to Choose the Best Game Engine (2018), *Medium*, Available at: https://medium.com/@N_iX/unity-vs-unreal-how-to-choose-the-best-game-engine-d3dbb4add73c

Build software better, together, *GitHub*,
https://github.com/

GibsS/unity-pan-and-zoom, (2019), *GitHub*, Available at:
https://github.com/GibsS/unity-pan-and-zoom

Unity Forums Available at:
https://forum.unity.com/

Unity Answers Available at:
https://answers.unity.com/index.html

App Store, *Apple (United Kingdom)*, Available at:
https://www.apple.com/uk/ios/app-store/

Play.google.com, Available at:
https://play.google.com/store/apps/developer?id=The+Tor+Project&hl=en_GB

Playground Games, Turn 10 Studios (2018) Forza Horizon 4 [Video game]. Xbox Game Studios.
Criterion Software (2012) Need for Speed: Most Wanted [Video game]. Electronic Arts.

For repo and projects

https://github.com/david128/DES310_Repo

https://github.com/david128/DES310_Repo/projects/2

Attached in the ZIP folder is all the scripts referenced in here and videos.