Y2S2 Programming Journal - Jamie Hulkes

This is my programming journey which documents my process of creating five unity packages and then creating a simple game out of packages. It contains information regarding all issues and fixes that occurred throughout the process as well as all dynamics of how the code was created, how it works and why I decided to make some development direction decisions.

To gain further instruction on how to use these packages, view the documentation included within each of the packages.

Package 1:

This first package is a challenge to myself which consists of making a simple procedural system. As I specialise within the game art field, I wanted to take a leap to gain confidence within the programming field. Using this logic, I knew that I needed to make a system that used simple algorithms in an efficient way to make it as achievable as possible (when programming you should aim for these qualities regardless in fairness). This led to the decision to create a procedural dungeon system. I chose this because its confined rooms which attracts the idea of a simple algorithm considering the nature of four walled rooms with the possibility of a door or two which led to another room containing the same content.

The goal of this package is to have a starter room prefab the developer will place in the scene and then on the beginning of run time this room will spawn out a certain number of rooms at random with connections which would in turn create a tunnel system known as a dungeon.

The way the logic initially achieved this is that there's a floor tile which has four empty gameobjects which child it. These children are located on the each of the edges of the floor tile exactly a width of the tile away from the origin of the tile. This ensures that if another new tile is to be instantiated in the place of these game objects it would be right next to the existing tiles. Then the logic for the walls was that they determine weather another tile is created or not. Within the spawner the wall already exists, and it is set active if the algorithm decides that there will be a wall there. If it is false that a wall will be there, that's when another empty gameobject is activated which is the next tile spawner which will then spawn in the next tile which will follow the same process.

The reason for the walls being set active instead of initiated in is because I had a lot of compiling errors when adding them through that way so this was the best alternative that I could find, this is because its positioning and rotation kept being reset and I tired many ways to set this through the code but after a lot of testing and research, I couldn't find a way to avoid these errors.

Due to these errors however I did find an interesting not about the behaviour of compiling errors. I found at one point in time whilst debugging my project that the number of unique errors within the debug log became overwhelming. I was fixing problems one by one by bottom up until I found out that once you fix the top error instead it tends to fix all the following errors. This occurs because, code works algorithmically in order. So, for example it reads code by line one then line two then line three and so on. If an error occurs in a piece of code that other code relies on, it can cause other issue with whatever comes next (the dependencies).

Continuing with the way the generation works, the way that the algorithm decides whether there's a wall or a next tile is the bool that sets it is immediately given a random value with some influence. (Random.value > 0.65f). This can be adjusted in order to manipulate the randomness of the system.

This system worked in terms of being random and individually the rooms worked great with the wall system working but, when the system was all together it was spawning too many walls so everywhere was blocked in despite the silhouette of the dungeon always working great and looking beautiful.

I decided to fix this I would implement an additional two scripts. The first was assigned to the walls themselves and it basically worked by destroying the wall if it was intersecting with another wall to avoid these clusters (I made the box colliders slightly smaller to avoid this destroying other walls unintentionally). This would also add a filler floor piece in order to avoid gaps and access to the void for the player. The next script added to fix this error was a destroy random walls script which was added to the walls and made them have a probability that they would destroy themselves.

Both of these scripts helped but with tweaking their values I could not find a good middle ground. I found the dungeon to be between the extremes of my previous problem and then really sparce with walls which also wasn't ideal.

In the end I did achieve a very simple procedural system that worked for the floor so I could create some sort of arena surrounded by avoid with it but the walls do work, I would just have to revisit this package in order to get them to gain a good mix.

I did realise that if I was to revisit this, I would try to tackle the core of the problem that is overlapping objects, but unfortunately, I couldn't tackle this within the time given. This said, I decided to do five packages instead of the four because of this.

Package 2:

The second package that I decided to task myself with was a player controller. In previous projects I have relied on the unity character controller which worked really well during the prototyping whitebox stage of development and also to control characters that I have added to untiy to test if they work in engine after completing them from the 3d pbr pipeline. However, may issues have occurred once my games have begun to flesh out in content and complexity with the way that the controller interacts with the world and components. For example, there are some quirks such as not slipping down steep slopes but there are also more game breaking problems such as having a player in a lift. Using script communication in order to handle the players state during an interaction such as being in a lift is beyond difficult, whereas having my own system would bypass this. I would also have a lot more control over the feel and the whole system will be able to use a rigidbody system which is another bonus.

To further build on this package I will also create an interesting camera system which will add more challenge to the package but also give developers something more unique as there are plenty of generic third person controllers but none like what I am creating here. The idea with this is to recreate a camera system similar to Animal Crossing. This basically consists of the player being able to wall in all directions but the camera being fixed in one direction.

On first glance this may sound very easy buts its a lot harder than the standard controller because you can't just parent the camera to player and then be done with it. You have to code the player to look the direction they are walking smoothly whilst the camera just looking at them at a specific angle. Another concern is that the camera could look boring but to overcome this issue I have decided to implement Cinemachine to this as well to allow for camera breathing motions and so the camera can turn slightly ahead of the player when they walk to make the whole system be more accommodating to the players quality of life.

The system to get this to work is relatively simple but it is a lot of effort for what it is. This quality is something I like when I make packages however because it usually means that I am making something that not many people have even considered trying before. The system works by the player consistently looking at a transform called lookPoint. This is taken directly from the gameobject called playerLook which is what the Cinemachine camera looks at. Later I will explain why this is very ideal to create the best player experience.

The playerLook is teleported between eight positions in context to the player transform. These are forwards, forwards right, right, backwards right, backwards, backwards left, left and forwards left. This position is set very simply looking at the forward of the player direction through transform.forward. To then allow the player to look in the right direction visually, the player GFX is Lerped to look at this set transform. This achieves a nice smooth transition between movement directions.

Unfortunately, this whole system is based off of the players graphic rotation value. This is bad because the way Unity handles rotations isn't ideal. In fact, it caused an issue where the player would rotate back on themselves when transitioning between right and forward as well as the model not even rotating at some points. To avoid all of this all I had to do was make the starting angle 0.001f. This offset allowed the controller to understand a direction to gain context of how it should transition between states.

The way that the player themselves actually moves throughout the scene is through rigidbody.AddForce. This simply uses the controller inputs of horizontal and vertical. This adds a major advantage as it is already allows me to have not only keyboard control compatibility but also controller joystick controls.

The only other issue that I came across while creating this package was the fact that I couldn't package the Cinemachine with it. This meant that I had to create an example scene in order to show users how to set up a camera to work with the system. There are missing components, but they can be fixed by adding the Cinemachine package and then setting the look at and follow component to the player look at.

Overall, I am very happy with this result for the package as its nice and easy to add to any project and have a predefined foundation to build from. This system will also allow for easy adaptation compared to other premade controller packages provided by Unity, allowing me to always know how all the code works and how its structured as well as always preserving the maximum amount of flexibility throughout my projects.

Package 3:

The next package that I wanted to add to my collection was a dialog system. User interface is the biggest aspect of programming and Unity in general that I feel uncomfortable so I thought that the only way to avoid this being a permanent hole in my capabilities I will need to dedicate some packages to grasp a vivid idea of how to develop user interfaces for a wide range of interesting systems. This is also why for the next two packages after this package I will add at least an element of user interface to help with this building of confidence that I am trying to achieve.

The goal for this package is to create a flexible dialog system that is created using an array to allow for the developers using this package the ability to add as many text slides as they wish. To the player they will see the NPCs text animate on the screen then once all of the text has appeared a

continue button will appear which once pressed will move the dialog onto the next slide unless it has reached the end of the array in which the dialog box will disappear off of the screen.

To start this dialog box is built up off of four core components in the hierarchy if you don't include the empty game object that holds all of these objects and the script. These components are the name text, the dialog box, the dialog text and the continue button. The name text is some simple text component that you can edit with the game of the NPC that the dialog box is for; this has to be done within the inspector for this component. The dialog box is simply the graphics that goes behind the text that displays the box for all of the user interface components, it is simply just an image game object. The dialog text is what the script generates and reloads when moving from one slide to another. Finally, the continue button allows the player to move through the dialog and end it once it is done.

So now the script is what manipulates the text that the NPC is typing. On the start a coroutine is started. A coroutine allows a function to start, the reason for using it in this situation is I need it to generate the text over time using "yield return new WaitForSeconds();". Within these brackets I can also set the typing speed. This speed will define how quickly the text appears on the screen as I have coded a fake typed out text system. Within this IEnumerator that the coroutine has activated it adds each character of the string within the sentence the array is on into an array. It with then add a letter then wait the amount of time that the typing speed declares as I established before. It does this through a for loop so it will continue to do this until the whole sentence is generated.

In the update function I constantly check to see if the text equals the sentences index and if it does, that means that the sentence has finished being typed in. So, so far, the text is animated in with flexibility given to the developer over how much time it takes to generate. This check will then activate the continue button to become active in the scene so the player can press on it. This button uses Unity's event system. This event system ensures that once the button is pressed, a function within script will be run.

The function that after this is called next sentence and as named it continues through the array to clear the existing sentence and then generate the next one. It also deactivates the continue button straight away to avoid any glitch that the player may reveal regarding skipping sentences or generation generating multiple sentences at once which creates broken text as they both run at once. For example, the sentences "hello" and "codes" would generate as "hceoldleos" which is unrecognizable. This then goes on in a loop until the check on the next sentences finds that an end has come to the sentences array. In this situation the dialog system will destroy itself as it has ended. Originally I forgot that arrays start from zero so, for this check I needed to check that it was sentences.Length – 1 and not just sentences.Length.

Finally, I the way that the developer can type in as many sentences as they want was achieved within a single line of code. The sentences string is a serializable field and an array which means that it shows up within the inspector of the dialog manager and gives them the customisability options they need by default. These options are the length of the array (the number of sentences you want the dialog to contain) and then within them the text that you want them strings to contain for them sentences.

Overall, this system did go quite smoothly. I think this was simpily because I took time to plan out how to achieve this package before I even touched Unity. Originally, I wanted to animate the dialog box with the text until I planned out how to achieve it and realised that the flexibility of that system would not work at all because developers would have to animate each sentence. Whereas now you can make edits during run time, it is that flexible.

Package 4:

For the penultimate package I decided to take on more of a challenge. This meant a package that had more content than the others and a system more complex. I decided that this had to be a crafting manager. Not only was this a challenge that I wanted but it would also be a very useful concept to get a grasp of considering my advanced game project will need a similar system implemented into it. Due to its complexity, this system uses four scripts. Two of which are used to assign to newly implemented assets to quickly get them integrated within the system in the quickest and most efficient way possible. Another script is to control the cursor which will later allow you to visually see yourself pick up the items and then drag them into the crafting system. The final script in this system is obviously the crafting system itself which hold the majority of the code involved.

To start I will talk about the first two scripts. Their purpose is to allow the developer to assign data to an object and then allow the crafting manager to access this data to use when said object is being interacted with by the player with the crafting system. The first of the two scripts are the item script which is assigned to any item in the game that a player can drag into the crafting menu. It simply holds the information of the item name in the form of a string. This string is very important to remember for when the system checks for a recipe result. The other script is added to the crafting slots and so is called slot. This holds the current item name of whatever item is in this slot (if there is one) and then the index of this slot. Once again, this index is the important variable here because it creates the order of which the items are in for the recipe to work. So, if there's wood then gold for example and then gold then wood, they can create two separate recipe results because of the order. These index's are assigned by the user but I give eight to start with (nine technically because unity starts counting from zero).

An item doesn't only need the item script assigned to it but it also needs an image assigned to it within the image component of the game object, this will visually allow the player to see what they are clicking. It is also very important because this will also be called by the crafting manager to be set as the empty image component to the cursor once the item has been clicked on to give the visual of the item being dragged. This will also be assigned to the empty slot once the item has been dropped into the crafting system. Speaking of the cursor, its script simply is to constantly set its transform to that of the mouse allowing for this seamless picking up of the items.

Now for the crafting manager, the system starts by asking if the mouse has been released. This is because after you drag the item, you let go of the mouse. Obviously sometimes you release without using wanting to use the crafting manager so in that case it asks if the item is null or if there is an item, it will only proceed if there is an item. It will then deactivate the cursor so and then check for the nearest slot to the mouse and activate that slot as the image. This creates the output of a seamless drop of an item into the crafting system. After this code that creates the visuals, the crafting manager then finds the index of the slot and then puts the string of the item and places it there.

It then checks for if the recipe matches any of the recipes that the player decides to create a new object (recipe result). This is one of a pair of arrays that the player can control. The first array has the recipe strings which basically list the item names in the correct orders. Then the second array is the item results. So, if the fifth recipe in the array makes a wood object, then the fifth object in the second array has to be that result for it to work. In order to check for the correct result what the system will do is get the string from the current data of the items and their order and then compare it to all of the recipes in the array. If one matches, then it will get the index of the array and compare it to the other array to then assign the correct result slot item. This is done in the same fashion as

the slots where it's an empty inactive image that then has an image assigned to it depending on the result.

Obviously, most recipes don't use all nine slots so for blank slots all the user has to write for that order is null. This will tell the system that this slot is supposed to stay empty in order for the recipe to work rather than just checking for what slots are active with items.

Overall, I'm very happy with the flexibility of this system. Especially being able to rearrange slots to make all sorts of interesting looking crafting systems but also how easy it is to expand the number of items in the game and recipes. This will allow me to be able to easily implement a big system with loads of items impressively quickly which is the most important goal to me with a system like this.

Package 5:

For the final package I decided to create a time and date manager. This system is basically a user interface which displays the games current time and date. I want to continue to create flexible systems in which not only can I drag and drop into my future projects but also adapt on the spot. Especially with a concept such as time where different countries and cultures display have an alternative way of conveying this information. For example, in America the day and the month is in the opposite order to the United Kingdom. This means I can allow for an options menu or alternatively other developers that may use my packages to change this package around their preferences. This package will contain two options regarding the time and four options regarding the date. The two options for the time will be a twelve-hour clock and a twenty-four-hour clock. Of course, the twelve-hour clock will have an additional piece of user interface which indicates weather the time is am or pm. To display this information, this package will also have to create values as to what the time and date is. I will additionally allow for another editable feature of this package which is the number of seconds in a minute. This float variable will determine how quickly time will go within the game. The default value that I have given this variable is one. This basically means that for every one second in real life, a second in game passes. With this value it will continue to mean that every minute is an hour in game, therefore every twenty-four minutes will be an entire day within the game. I chose this value as it's a nice balance between fast and slow and as it's a low value its easy to work out key moments such as when a full day will go by. Obviously the next most used value for this that I predict will be sixty. If the value of this variable is sixty then the time will run to real time which will be especially handy to strategy and simulator games, or even simple applications made within Unity.

So, how it works. To start, I made a timer variable that in update had Time.delta time added to it. What this will do is go up in value every second, this is normalised because I am using deltaTime which is ideal for this type of system. I then use the variable that I was discussing earlier which is the seconds per a minute float and I compare this to the value of the timer. More specifically I ask if the timers value is higher than the seconds per a minute. If this is true then I will run a check that adds a minute but also then asks for the following values to change in a similar manor, hours, days, month. So, using the exact same process as adding a minute to the time it will then ask if the minutes has exceeded the maxMinutes. This value is sixty as that is the number of minutes in an hour. Once again, if this value is above the max then the next check will be activated which is the exact same but for adding a day. Obviously, the only difference being that the value for maxHours is twenty-four because there will always be the number of hours in a day. This process will then continue for months and years. For months the maxDays value is thirty as that's the average amount of days in a year and it is only a small detail which I didn't think was worth impacting this simple system over. After maxMonths which is twelve there is no cap on the amount of years in game there will be as

there is a very low chance that a player will play for years in order to show break make the user interface leave the screen.

Once the update function has gone through this string of if statements and added the necessary values it will reset the timer to start again. It also activates a function called SetTimeDateString. This function is made up of two sections which are setting the date and time strings and then checking the format and setting them accordingly to the data sent to the date and time strings. The date and time format values are set as enumerators. Time stores two formats which are for a twenty four hour clock and a twelve hour clock and the date stores four formats which are the (day, month, year) format, (month, day, year) format and then the same as these two but with the year value at the front if the game has a higher focus on year, like for example a strategy building game where it takes years to build up your empire. The developers chosen formats are editable within the inspector of this script and that is how these values are set.

Going back to the SetTimeDateString function, it is compiled of cases in which the format for the string to put the values in is outputted. For the twelve and twenty-four-hour clock time system however, this function also has to be minus twelve hours from its value if its above twelve for the twelve-hour time system. It is also set to pm when the value goes above twelve initially. Once the formatting has been assigned the outputs will look as follows: (date = days + "/" + months + "/" + years;), (date = months + "/" + days + "/" + years;), (date = years + "/" + days + "/" + months;) and (date = years + "/" + months + "/" + days;). The actual text itself is set as an array to account for being able to display this updated animation in real time. This is assigned through the code dateText[i].text = date;.

Now this system is completely working and all of the customisability can be edited during run time of the project however, all the values of time as of the moment are zero which means the game starts at 00:00 00/00/0000 or the equivalent for the other appropriate format types. To fix this issue I had discovered that Unity has a way to access the information of what the current hour, minute, day, month and year in real time is, which is as follows for hours for example; System.DateTime.Now.Hour;.

Overall, this system worked very well the whole way through with the exception of me realizing that as I was using the TextMeshPro package that Unity provides I didn't need to use UnityEngine.UI and instead TMPro. I think this is due to the fact that this system uses the safe system of a lot of if statements which for this purpose do work however, this isn't the most efficient way of the system running. This is simply due to the lack of experience that I have with more streamlined systems. However, that said, I am very happy with how adaptable that this system is whilst still maintaining other admirable features such as being strong and reliable – I am yet to find a problem with this system. I would also like to mention that if I was to revisit this package, I would like to update the method for the month system and check the month number in order to define how many days are in that month. The only other option with this system would be that I would also have to create a check for whether that year was a leap year so I could alter the number of days in the month of February once every four years.

Package game:

The point of this game is to showcase the fact that I can get packages to work seamlessly together. I decided to start with some of my generic packages and put them together so I could get a good foundation to work from and then from there I could decide on a game to create from there. I came to the conclusion that I should start with the procedural dungeon package and the character controller. This way I had a foundation for the player to control and play with and also a scene in

which they can do this. This was a very simple process for the dungeon generator, all I had to do was place the starting dungeon piece at 0, 0, 0 in the world space and the rest was taken care of by the package although I did make some edits to these tile prefabs later in the process. I then proceeded to add the character control. This package was made in urp so I did have to firstly update the materials to work with the standard render pipeline as at first everything was bright missing texture pink. This fix was simple compared to the next issue I came across which was that the packages camera system does rely on cinemachine, therefore I had to install that package and then implement the cinemachine brain component to the camera. After this I had to then set the values to the noise which dictates the camera shake and also where the camera looks and what it follows. This was already made in the package so all I had to do was simply drag the player look game object into these boxes.

Now that I had this basic foundation, I decided to add my time manager to the scene. I decided however to use it for a slightly different use to what it was initially designed for, this being that it served the purpose more as a timer than for a day and night cycle. This is definitely overkill for this type of game, but the intention of this project wasn't based on game design but, more to prove the point that I can use packages together. This was once again more of a simple case of implementation but that is how it was intended to be. I did however remove the code that set the date and instead let it all start from 00:00 00/00/0000 to get the look of a timer. Once again, very overkill but it still functioned the intended way.

Once I had implemented the timer, I knew that I had to make an arcade type of game. The procedural aspect of the game never being the same each time you play it also made a lot of sense and really worked in my favour. It was at this time that I knew that I needed to have a win and lose condition and although I still didn't have an idea over what the gameplay would be I did know that these were still necessary components considering the type of game I was aiming to make. I decided that the fourth package that I would implement would be the dialog system and I would make two dialog prefabs from using the package that I would instantiate. One being on a win and one being on a lose. I wrote in the array the text that I wanted to come up for each of these conditions and I didn't have to do any more editing to the scripts within that package however I did have to use another script in order to make the dialog appear at the right times.

This leads me to the new code and scripts I had to make in order to create a functioning game with these packages. First, I came up with a game idea in order to work out what I needed. I decided that I could expand upon my procedural system and add coins that could spawn randomly on the floor for the player to collect. I still needed a lose condition to this however so I decided that the twist was that the coins would either be a coin or a mine at random, turning the game dynamic into minesweeper. This meant that the win condition could be the player collecting all of the coins and the lose condition being that the player accidently picks up a mine instead of a coin (visually there is no difference between the two).

I first started by working on the system to get the coins spawning in the game but first I had to make the coins themselves. This was relatively simple. All I had to do was create a cylinder, make a material for it, put it on its side and create a very basic rotation script which rotated the coin constantly in update. There were then two more scripts that I need to make which was the location setter for the coins and then the game manger which would handle the collection of the coins and the user interface for the player to see how many coins that they have left to collect.

Starting with the location setter for the coins, I made the script very basic at first. The coins would be in the scene from the start (I added ten) and then the script on start would just set their location to be random between a radius of the players spawn to make sure that not too many coins would

spawn off of the floor and in the void. I then added a check to the coins where they have to check that they are colliding with a roof trigger. This meant that I had to go back into my procedural dungeon prefabs and add these trigger colliders to them so that the coin would interact with them if it spawned above them for the player. If this check came back negative, then it would destroy the coin as it would be in the void. This also made it so depending on the random generator there would also be a random number of coins below ten as some would get destroyed which leads to even more procedurally random and interesting results for a player to experience. At first all the coins would destroy all of the time. I found that this problem was caused because the coins thought that they were in a void before the dungeon had generated to, I delayed this check by putting it in a coroutine so I could add a "yield return new WaitForSeconds" to ensure that the dungeon generates first. As this was the only script on the coin it also holds the Boolean for if the coin is actually a coin or if it is a mine. I then added this script to the coin prefab and also the mine prefab and ticked this box for the coin, I also added tags accordingly in preparation for the game manager.

First of all, I had to go into the player movement script and add the game manager component to it so that I could make the player colliders look out for the triggers of the coin and the mines in order to interact with them. To both collisions it will destroy the other game objects but fir the mine it will activate the lose dialog by instantiating the prefab. Everything works from start in the dialog system which allows this to work best when instantiated. For the coin however it just minuses one from the coins left for the game manager. Back in the game manager it is constantly updating the user interface for coins left which is copied from the time manger user interface to keep it consistent and then checking for when the coins left is equal to zero. Once it is equal to zero it sets it to minus one thousand to avoid the player retriggering this event and then it sets the win condition dialog.

Overall, these systems are quite simple, but this is due to me coding my packages earlier on in the semester in really durable ways which allowed me to achieve the majority of my edits for a range of uses through the inspector instead of having to rewrite code. When I did have to rewrite code, it wasn't difficult as I could create my own functions to keep the code clean and avoid confusion over the code from the package and new code although, I did try to avoid this as much as I could regardless. Overall, I am very happy with this result, and it gives me a lot of confidence to not only create packages to make developing my advanced game project easier but also create packages from my 3d props that I make which I could potentially sell on the unity market place or create for teams that I work with on future projects.