Computing Project

Max Rose | OCR Computer Science

Contents

Analysis

STAKEHOLDERS

The target audience for the game will be children aged 10-15 as this will give me freedom to give the game a lot of depth whilst being simple on the surface. As this will be a PC game this means that due to the age range the hardware of the PC's will probably be rather large and will probably have to run on all types of hardware from the very low end to somewhere in the high end. The game could also be useful for more than just recreation as the questing system and crafting system could be designed to teach the younger players. The needs of humans within this age range can vary. However, they will mainly need to be entertained as in the earlier ranges they could have limited computer use time decided by the larger humans. So, the game should not take hundreds of hours to complete, or if it does then you should be able to complete it in small bursts of for example 30 minutes to 1 hour, so a quest system comprised of short entertaining quests could help this problem. In the older age range the humans will not be as limited so they will problem be looking for something that you can play for many hours meaning that if a quests system was used then the quests should be varied to keep the player interested. Games are also a good thing as they can improve hand-eye coordination as it will teach the younger children as it has been proven that games help this by many different research papers. Playing games also helps with strategic thinking and helps healthy brain development.

Research

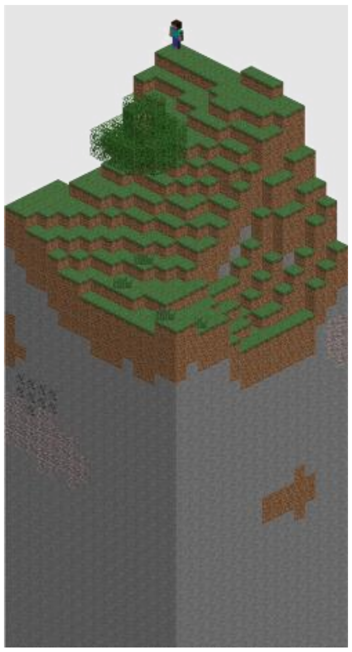
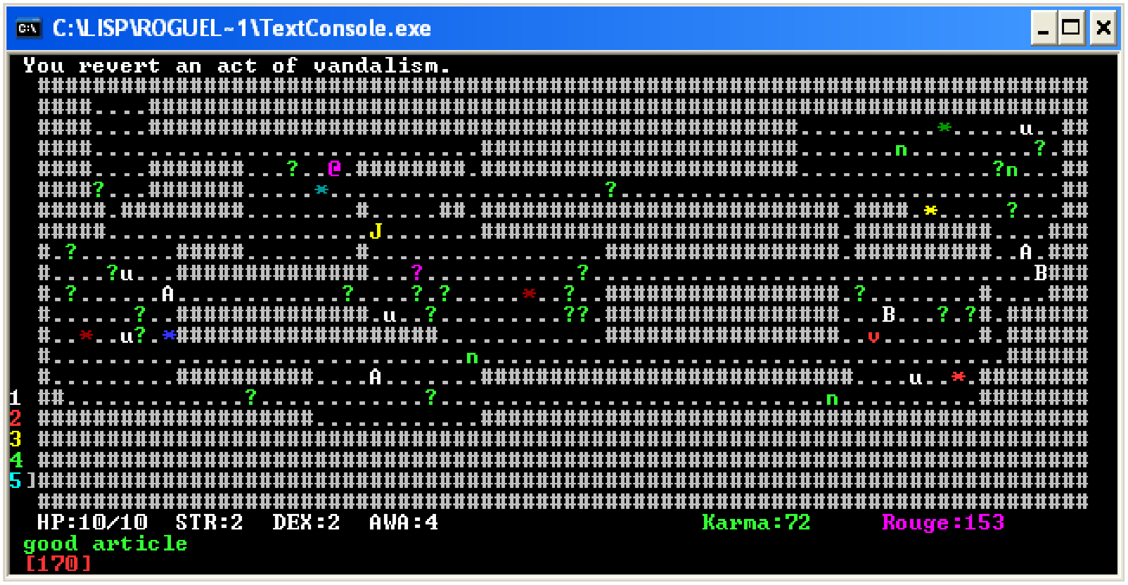


Figure 1 – Minecraft Chunk Example

Minecraft uses procedural terrain generation; however, this is not the reason that I feel that this game is popular. The reason that I feel that it appeals too so many different people is the near infinite number of possibilities that can be had when playing from simply building house to making complex machines, or if none of that appeals you can add "mods", like IC2 or Build Craft, to the game to expand its content far beyond that base "vanilla" version of the game. This game uses 16x16 chunks that can be generated individually, loaded, and unloaded one at a time. This reduces strain on the computer as the areas that do not need to be seen can be unloaded improving performance. Also blocks that are not visible are not rendered and the back faces if blocks are not rendered further improving performance. As a result, I will take a similar approach to the terrain generation for my game where I will break each section down into say a 16x16 chunk and then further section it into a 16x16x16 cube to further help with performance as this will mean that the chunks can be more quickly generated in a secondary thread. The crafting system for Minecraft is also very intuitive as it is a highly visual system. This means that even if you do not already know how to make something you can work it out. This is the kind of visual system that I would like to use in my project. The ability to add mods to the game that can change it in very drastic ways is also something that is very unique and if possible I would also like to be able to allow mods in my project, to at least some capacity as this will increase the appeal of the game as the people that want to just play the game can but the people that would like to add to the game and change their experience or use it as a platform to learn about game development through making a mod can also use it.

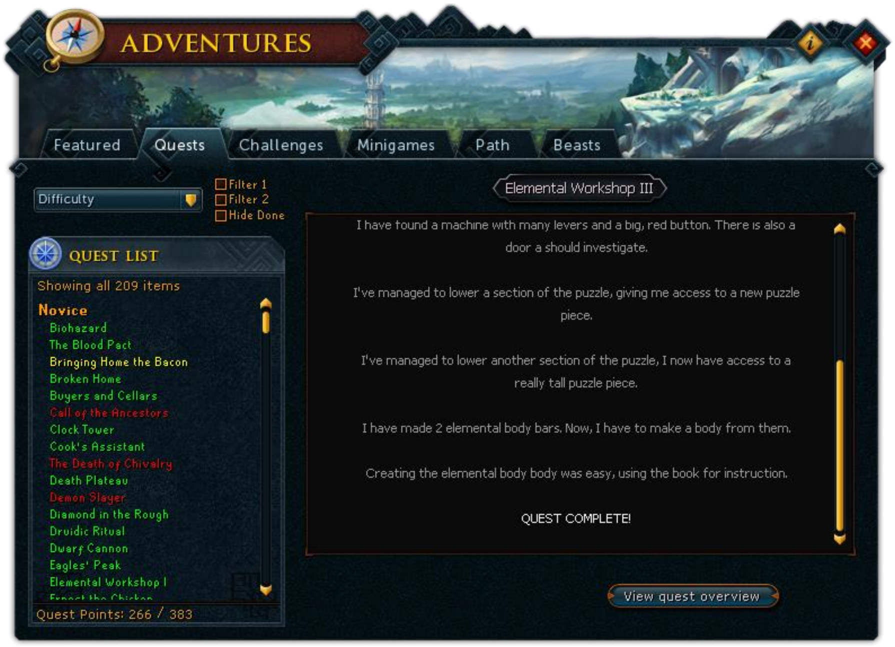


```
You revert an act of vandalism.

HP:10/10  STR:2  DEX:2  AWA:4                 Karma:72     Rouge:153
good article
[170]
```

Figure 2 – Original "Rouge" Game

Another currently popular approach to level design is the "Rouge Like/Lite" method. With this the level is generated using rooms with the player progressing through each room with one room containing some stairs or like take you to the next level. I could use a similar system in my game with the player gaining money by some system then using the money to buy new rooms possibly with extra functionality helping the player to gain money more quickly, this would help the player enjoy the game more as it would give the player some feeling of progression making them want to continue to play the game, if the players just stopped playing the game after their first time playing it.



Showing all 209 items

Novice
Biohazard
The Blood Pact
Bringing Home the Bacon
Broken Home
Buyers and Cellars
Call of the Ancestors
Clock Tower
Cook's Assistant
The Death of Chivalry
Death Plateau
Demon Slayer
Diamond in the Rough
Druidic Ritual
Dwarf Cannon
Eagles' Peak
Elemental Workshop I

Quest Points: 266 / 383

Elemental Workshop III

I have found a machine with many levers and a big, red button. There is also a door a should investigate.

I've managed to lower a section of the puzzle, giving me access to a new puzzle piece.

I've managed to lower another section of the puzzle, I now have access to a really tall puzzle piece.

I have made 2 elemental body bars. Now, I have to make a body from them.

Creating the elemental body body was easy, using the book for instruction.

QUEST COMPLETE!

Figure 3 – RuneScape Quest UI

RuneScape has a very good questing system that is engaging as unlike most the quests have context with a storyline. This helps the player remain engaged with the game and what they are doing, this is the kind of thing I would like to incorporate into my game as it will mean the player will want to keep playing the game.

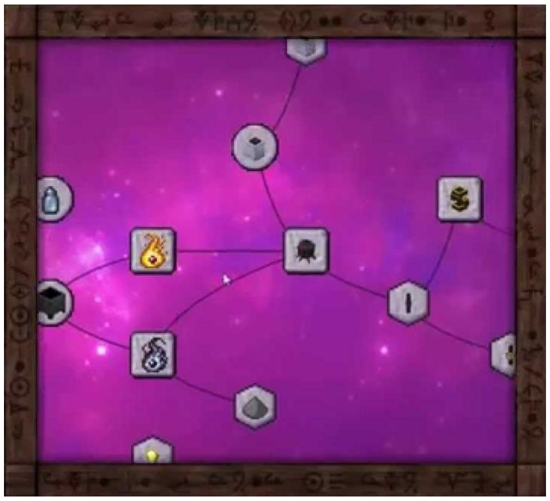


Figure 4 – Thaumcraft Quest UI

However, the UI for the RuneScape quest system is very dull, contains a lot of text, and is rather static. So, I would prefer to create a UI for the quest system as in the picture above where each quest is an icon with a background that moves as it is scrolled around. Each quest would either be greyed out or invisible when unreachable, during completion it would flash and once completed the icon would remain lit. To get the information for a quest the player would click on the icon taking them to another screen with the quest details and the background story.

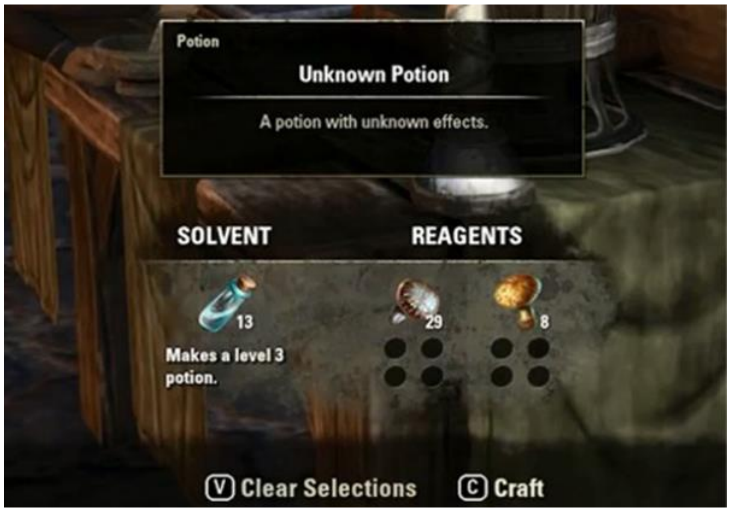


Figure 5 – Skyrim' s Potion Crafting System

Another approach to the crafting system is "Skyrim's" potion crafting system. In this system, you collect the crafting ingredients from around the world however the effects that the ingredients will have on the potion are not known. The player then needs to use the potion on order to discover the effects that it has. This is an element of the game that I would also like to attempt to incorporate into my game in some way, for example a system where possibly some animal breeding could make randomized stats and the player must analyze them for discover the effects of then animal. However, the extent to which the system can be implemented will most likely heavily depend on the story line/quests that the game ends up having.

Computational Methods

With this project, I can use all aspects of computational thinking as it will help to visualize/abstract each method so that only the most necessary parts of the method are included, as this is a game all performance increases are valuable so each line of unnecessary code that is removed is valuable. This is because although 1 or 2 lines being removed will not affect much it will quickly add up as more are removed.

I will also need to think ahead for example with the item system I will need to think about giving it enough flexibility to do all the features that I have planned and will possibly creep in as I am writing the code, but also exclude enough so that it is not filled with lots of unnecessary code bloating the items and increasing the time taken to do calculations. If done properly pattern matching cold be used on items making the code much more manageable, this would also help as it will help anomalies be detected making bugs quicker to find. It is also possible that pipelining (multi-threading) can be used of the game is designed properly for example the code to generate the terrain can run on one thread with another being used to display the currently created terrain, this would improve performance as the complex time-consuming calculations would not be done on the same thread that the user can see improving performance as multiple things can be done at once.

Both concepts will require thinking procedurally/decomposition, logically, and concurrently. As each part will require the part that is currently being worked on to be broken down and worked on one part at a time. Each part will also need to be thought about logically so that it will work and be as efficient as possible. This will also need to be done whilst thinking about how this will be used and effect the bigger scope of the rest of the game, so that it can be as general as possible whilst still being useful and to make sure that it will not break another part of the code that it shouldn't.

ESSENTIAL FEATURES AND LIMITATIONS

The essential features of the game would be:

- Terrain Generation
- Questing System
- Crafting System
- Item System
- Terrain Modification
- World Saving/Loading
- Some Modding Capabilities

I think that these features are essential as in my opinion they can make a very engaging experience and will also show my ability to solve complex problems and make that work in an optimized way.

The limitations of my project will probably be my ability to optimize the game as Unity does not allow objects in the game to be modified from within another thread. This means that any modifications make to the mesh of the chunk will need to be done from within the main thread once it has been generated. The creation of the mesh also cannot be done from within another thread as Unity does not allow its data structures to be edited from another thread so the creation of the collision mesh which is one of the most taxing parts of the mesh creation can only be done in the main thread this will again pose a problem when optimizing the game.

An item system will probably not be too difficult as inheritance can be used to create a base item class then subsequent items can then be build off that. Building the items from a single base class will allow a very easily coded inventory system as the items can just be stored as the base class then the specialist methods of each items can be called as necessary. However, in C#, thee language I will be using to program the game, classes are passed by reference, so I may need to either consider custom = method to transfer the item classes. A custom == will also be needed as classes are compared by reference alone, so this will need to be changed for my purposes. Something like a struct can be used to solve this however that will have its own problems, so I will need to test the two systems against each other to find the system that I will use.

Saving and Loading will be a challenge as I will have to make my own data structure separate from Unity as it does not support runtime serialization. So most likely only the changes to chunks will be stored and when the chunk is remade on loading the changes will be applied. This will not only have the effect of allowing me to save the changes it will also be more efficient than saving the whole chunk every time the chunk is modified, as it will reduce file size reducing saving time.

The modding component of the project will be most likely my weakest area as I do not know if I have the skills and/or foresight to see the challenges that would be presented. As I will be using C#

as the language to make my game I do know that I can use reflection to more easily load external code at runtime but the implementation of that is something that I may not be able to-do, so this feature may be something that eventually gets scraped so for a start I will aim to get a bare bones system working that only allows the addition of new blocks and crafting recipes.

One of the major limitations for the project especially will be the UI and Storyline elements as I am not the greatest at making good looking assets and writing a coherent story. So, I will most likely use UI assets that can be found on the internet that are free to use and possibly only making a bare bones story to convey the idea of the intended story or ask a friend for help writing it.

SUCCESS CRITERIA

- Run without game breaking bugs/errors
- Target audience or another audience enjoy playing it
- Meeting most of the essential features
- Run on specified hardware

If the game was filled with game breaking bugs/errors that would negatively affect the user experience. By game breaking bugs I refer to things such as memory leaks, random crashes, and other such fun and interactive elements. I can test this by looking at the memory usage for the program to ensure that it does not keep consuming memory by using the tools in visual studio as pictured in Figure 6 – Visual Studio Debug Tools.

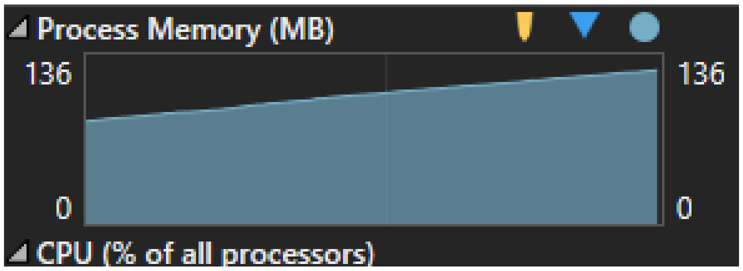


Figure 6 – Visual Studio Debug Tools

If the target audience did not enjoy playing the game, then there would be no point in making the game. I can measure this by involving my target audience in the development stage and asking for feedback on functionality and usability. This mean that I can take into account the feedback received and ensure that the game fully meets the user requirements and expectations.

If most of the features are not met, then the game will feel overly simplistic and will become be very boring meaning that users are unlikely to play the game again. This would drive away the player in my opinion making the game a failure. So as part of the development I fill test each prototype to ensure that the requirements are being met and if not, I can program the features into the net prototype.

To ensure that the game runs on the specified hardware I will test he prototypes on desktop PC's in school.

HARDWARE & SOFTWARE REQUIREMENTS

The operating system will need to be at least Windows 7 or Mac OS X 10.9 as anything prior is either not recommended or not supported by unity which is what I am using to compile the game. The user will also require a keyboard and mouse to play the game as all the menus and inventory's will be mouse driven so the user will need a mouse to use them. The user will also need a control the character with a keyboard as this will be a PC game I will not add controller support, so a keyboard will be required to control the character. A processor greater than an I3 and more than 1GB of RAM will be needed as a minimum for the game to run. I have chosen this spec as it will give me room to work with if I cannot optimize the game as much as I would like, also more computers now have a spec greater than or equal to this meaning this the game will run on most modern computers.

Design

BREAKING DOWN THE PROBLEM

The computational method that I will use initially to solve my problem is “breaking down the problem” or “decomposition” using this I have identified 6 areas that I will need to implement, these 6 are: Game Start, Render Initial Chunk, Break Block, Place Block, Saving and Loading, and Quests

GAME START

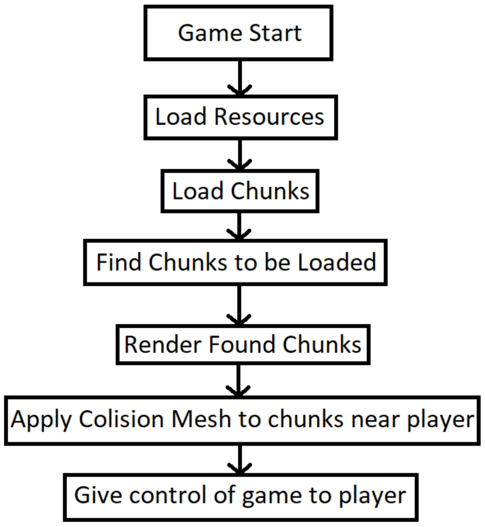


Figure 7 – Game Start Process

When the game is started the essential resources are loaded e.g. if the world has been used before the modified blocks are loaded. Then chunks are loaded, the chunks to be loaded will be in an area around the player, this will reduce the amount of work the CPU and GPU must do when the game is loading thus improving performance. The chunks are then rendered without a collision mesh, as with one there are constant collision calculations and applying a collision mesh to an object in Unity causes a large amount of lag, therefore reducing the number of collision meshes that need to be applied will greatly improve performance. The chunks that are near the player then have their collision meshes applied as they will be in direct contact with the player so need one. As a player will move unpredictably, at least a 3x3 area will be required for collision meshes around the player. Once all of this is done control is the game is given to player. The game will start in this way so as it will prevent corruption caused by the player doing something that they shouldn’t during the process of loading the game.

Game Start Pseudo Code

```
START

        Serializtion.LOAD(“Player”)

        LoadChunksAroundPlayer()
        ApplyCollisionMeshes()

        PLAYER.freezecontrols = false


        CLASS World

                Vector3 chunksAroundPlayer = [] //An array of static positions relative to
the player and when chunks are within the range of this they are rendered

                Vector3 collisionMeshChunks = [] //an array of a 3x3 grid to apply meshes to chunks

                LIST<CHUNK> chunksToMake = NEW LIST<CHUNK>()

                METHOD LoadChuksAroundPlayer()

                        FOR i = 0 TO LEN(chunksAroundPlayer)

                                Vector3 newChunkPos = PLAYER.POSITION +
chunksAroundPlayer[i]

                                CHUNK newChunk = GetChunk(newChunkPos)

                                IF(newChunk != NULL)

                                        CONTINUE

                                ENDIF

                                newChunk = NEW CHUNK(newChunkPos)

                                chunksToMake.ADD(newChunk)

                        ENDFOR


                        RenderChunks()

                ENDMETHOD

                METHOD RenderChunks()

                        FOREACH chunk in chunksToMake

                                chunk.RenderChunk()

                        ENDFOREACH

                        ChunksToMake.CLEAR()

                ENDMETHOD

                METHOD GetChunk (newChunkPos)

                        IF(GAMEOBJECTEXISTSAT(newChunkPos)

                                RETURN(GAMEOBJECTAT(newChunkPos))

                        ENDIF

                        RETURN NULL
        ENDMETHOD

    ENDCLASS

END
```

RENDER INITIAL CHUNK

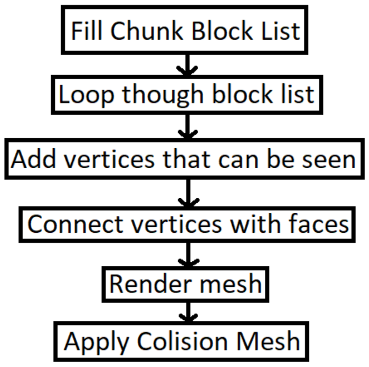


Figure 8 – Chunk Loading Process

The chunk's block array is filled so that the chunk knows what blocks it is supposed to contain, this is done by a noise semi-random noise function like Perlin Noise then edited blocks from when the world was previously loaded are added to the chunk. Then the block list can be looped though with each visible vertex added to the chunk mesh, having one mesh for the whole chunk instead of one for each block will improve performance as the more meshes, the worse the performance. The only vertices that are added are also only one ones that can be seen again to improve performance. Only thee faces that can be seen will be rendered this will improve performance as less objects are trying to be rendered. After the face and vertices lists are made they will be passed to the Unity render pathway to be rendered. The mesh is not rendered as it is created so that it is not continually re-rendered during creation this improves performance as Unity optimises a mesh before it is rendered and this effects performance negatively. The collision mesh is then applied to the chunk but only if it is near a player, this reduces the number of collision checks that need to be performed every frame thereby improving performance. The chunk block array must be public so that when blocks are added to the chunk they can add their own data to the array if they are given the chunk. It may be better to add the block via a method which would allow some data validation however leaving the block array exposed gives more freedom to add different blocks.

Chunk Rendering Pseudo Code

```
CLASS Chunk

    THVector3 position

    INT chunkSize = 16

    BLOCK[,,] blocks = NEW BLOCK[chunkSize, chunkSize, chunkSize]

    BOOL shouldHaveCollisionMesh = false

    LIST<INT>() verts = NEW LIST<INT>()

    LIST<INT>() tris = NEW LIST<INT>()
```

```
      MESH mesh
      CONSTRUCTOR(int x, int y, int z)
            position = NEW THVector3(x, y, z)
            GetBlocks()
      ENDCONSTURCTOR
      METHOD GetBlocks()
            blocks = TerrainGeneration.GetBlocks(position, chunkSize)
            blocks = Serialization.LoadChunk(position.x, position.y, 
positon.z)
            RenderBlocks()
      ENDMETHOD
      METHOD PlaceBlock(Block block, Vector3 worldPosition)
            Vector3 chunkBlockPosition = worldPosition.ASCHUNKBLOCKPOSITION()
            Blocks[chunkBlockPosition.x, chunkBlockPosition.y, chunkBlockPosition.z] = block
            RenderBlocks()
            RenderChunk()
      ENDMETHOD
      METHOD RenderBlocks()
            FOR x = 0 TO chunkSize
                  FOR y = 0 TO chunkSize
                        FOR z = 0 TO chunkSize
                              blocks[x, y, z].AddToMesh(verts, tris, x, y, 
z)
                        ENDFOR
                  ENDFOR
            ENDFOR
      ENDMETHOD
      METHOD RenderChunk()
            MESH mesh = UNITY.MAKEMESH(verts, tris)
            UNITY.APPLYMESH(mesh)
            IF(shouldHaveCollisionMesh)
                  UNITY.APPLYCOLLISIONMESH(mesh)
            ENDIF
```

```
    ENDMETHOD
ENDCLASS
```

ITEMS

The items for the game will also need to be contained within a class. Each item can have its own class deriving from a base class that contains all the essential information. This would class would also contain some methods that all items will need. Having a base class will also allow the use of polymorphism when storing items meaning that it will be very easy to make inventories as the base class can be the container and if a special function is needed from an item it can be easily cast to that type. It will also allow easy filtering of inventory's as a specific item or parent item class can be specified, this will mean that only certain item types would be allowed in that inventory.

Item Base Class Pseudo Code

```
CLASS Item
    PUBLIC STRING itemName
    PUBLIC INT itemID
    PUBLIC INT stackCount
    PUBLIC INT maxStackCount
    PUBLIC SPRITE itemSprite
    CONSTRUCTOR Item(STRING itemName, INT maxStackCount = 1, INT stackCount = 1, INT id = 0)
        itemName = _itemName
        itemID = id
        stackCount = _stackCount
        maxStackCount = _maxStackCount
        GetSprite()
    ENDCONSTRUCTOR
    METHOD GetSprite()
        itemSprite = FINDSPRITE(id)
    ENDMETHOD
ENDCLASS
```

BREAK BLOCK

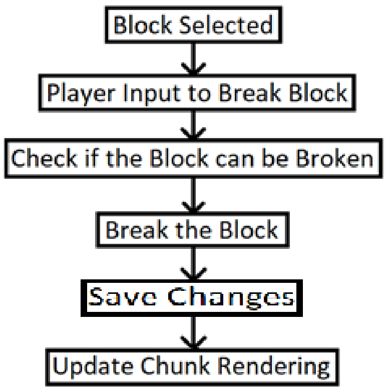


Figure 9 – Block Breaking Process

The player will look at the block that they would like to break. Then the player gives an input to the game that they would like to break the block. Once the input is processed the selected block is checked if it can be broken (is it in range? does it have a breakable check flag? etc). If the block can be broken it returns the correct item; if it cannot, nothing happens and the process breaks. Once the block is broken the changes will be saved and the chunk re-rendered. The chunk re-renderings is done last as it will not matter if the game crashes before the chunk renders as long as the chunk is saved because it will render the block when the game is re-loaded. The chunk rendering can also take some time so it is most logical to do it last as it means that the player will feel the least amount of slowdown as a result of the rendering, the game also has the highest chance of crashing as a result of rendering due to the length of time it can take therefore it also makes sense to do it last.

Break Block Pseudo Code

```
START
    IF(INPUT.BREAKBLOCK())
        Vector3 playerLookPosition = PLAYER.RAYCAST().HITPOINT
        IF(World.GetChunk(playerLookPosition) != null)
            World.GetChunk(playerLookPosition).PlaceBlock(null, playerLookPos)
        ENDIF
    ENDIF
END
```

PLACE BLOCK

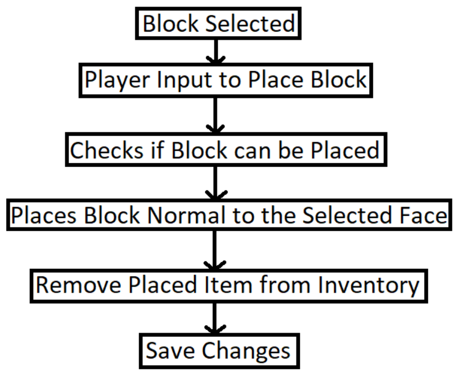

Block Selected
Player Input to Place Block
Checks if Block can be Placed
Places Block Normal to the Selected Face
Remove Placed Item from Inventory
Save Changes

Figure 10 – Block Placing Process

The face that the player would like to place the block on is selected by the player looking at the block. The player then gives an input to the game to say that they would like to place a block. The inventory would then be checked to see that there was a place able item selected and the block the player has selected is within placement range. The block is placed by giving the block to the chunk with the correct x, y, z coordinates relative to the chunk, the new block is added to the block list then the chunk is re-rendered. The placed block is removed from the player's inventory and then the inventory and the chunk are saved. The system will be done in this order so that if the game crashes whilst the block is being placed no items/data will be lost as the player will be easily able to replace the block. The whole chunk will have to be re-rendered as there is no way to reaerate part of a mesh in unity at runtime the whole thing must be updated, as far as I know.

Place Block Pseudo Code

```
START
    IF(INPUT.PLACEBLOCK())
        Vector3 playerLookPosition = PLAYER.RAYCAST().HITPOINT
        IF(World.GetChunk(playerLookPosition) != null)
            World.GetChunk(playerLookPosition).PlaceBlock(PLAYER.SelectedBlock, playerLookPos)
        ENDIF
    ENDIF
END
```

SAVING AND LOADING

Each different section will have a separate save file with each chunk also having its own file in a separate sub folder. Therefore, the file structure of a save file will be:

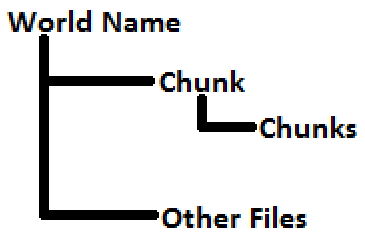


Figure 11 – Save Folder/File Structure

Each chunk file will be named with its central coordinates. A chunk will only have a save file after it has been edited. This will decrease the file size decreasing the loading time as less data is needed to be loaded into memory. This will give a smoother playing experience for player and it will also mean that the game will have lower hardware requirements as a low amount of data will need to be loaded into memory meaning that PC's with lower amounts of memory can run the game.

Saving/Loading Pseudo Code

```
STATIC CLASS Serialisation

      STRING saveFolderPath

      METHOD SetFolderPath(STRING path)

            IF(path.EXISTS())

                  saveFolderPath = path

            ENDIF

      ENDMETHOD

      METHOD SaveChunk(INT x, INT y, INT z, BLOCK[] blocks)

            STRING fileName = saveFolderPath + "Chunks/" + "Chunk@" + x + "," + y +
"," + z

            SAVE(fileName, blocks)

      ENDMETHOD

      FUNCTION LoadChunk(INT x, INT y, INT z)

            STRING fileName = saveFolderPath + "Chunks/" + "Chunk@" + x + "," + y +
"," + z

            RETURN LOAD(fileName) as BLOCK[]

      ENDFUNCTION

      FUNCTION SaveData<T>(STRING name, T data)

            STRING fileName = saveFolderPath + name

            SAVE(fileName, data)

      ENDFUNCTION
```

```
     FUNCTION LoadData<T>(STRING name)
          STRING fileName = saveFolderPath + name
          RETURN LOAD(fileName) as T
     ENDFUNCTION
END STATIC CLASS
```

QUESTS

The game will have a Quest system for this a data structure and interface methods will be needed. I think that the best way to handle this system will be to have 4 dictionaries, one for completed quests, one for completed but not handed in quests, one for currently in progress quests, and the final for currently locked quests. The dictionaries will all be private and static with helper methods so access them. The will mean that some validation can take place when accessing the dictionaries to ensure that they are used properly. This will be essential if the planned modding capabilities are added as it will give mod makers a clear way to add, remove, and use the quest system. The quests could be completed using an event system that can be accessed from anywhere in the game, using an event system will also add another layer of abstracting meaning that the system will be more robust and versatile system that could also have more chance for data validation, ensuring that the system cannot be misused. It will also give in easy interface for blocks to complete quests if they are unsure that the quest exists.

The quests name would be stored as a string with the quests type then the item/items that are required to complete it. So, for example if the player needed to pick up a dirt block the quests would be stored as “Pickup: 0”.

The data in the dictionary would be an object array. This is because in C# all data types are an object, this means that the quests can return any data type giving a very large freedom for what quests can lead to. The data type is an array because the first index will be the quest reward, which can itself be a array of items, and the other indices will be the quests that lead from the current one. The will allow a large freedom in both what the quests can give as a reward and the number of quests that lead from the current one as the object array size is not limited in the dictionary definition.

In the pseudo code an example quests are shown in the dictionaries.

Quest Pseudo Code

```
STATIC CLASS Quests
     STATIC DICTIONARY<STRING, OBJECT[]> completedQuests = NEW
DICTIONARY<STRING, OBJECT[]>()
     STATIC DICTONARY<STRING, OBJECT[]> compleatedUnclaimedQuests =
NEW DICTIONARY<STRING, OBJECT[]>()
     STATIC DICTIONARY<STRING, OBJECT[]> currentQuests = NEW
DICTIONARY<STRING, OBJECT[]>()
     {
         {“Pickup: 0”, NEW OBJECT[] { NEW DIRT(), “Craft: 1” }

     }

     STATIC DICTIONARY<STRNIG, OBJECT[]>  lockedQuests = NEW
DICTIONARY<STRING, OBJECT[]>()

     {

         {“Craft: 1”, NEW OBJECT[] { NEW STONE() }

     }


     FUNCTION ClaimQuest(STRING quest)

         VAR item = compleatedUnclaimedQuest.GETITEM(quest) //will
not return

                                                an error if the key does not
                                        exist

         IF(item == NULL)

             RETURN

         ENDIF

         compleatedQuests.ADD(quest, item)

         IF(item.LENGTH > 1)

             FOR i = 1 TO item.LENGTH

                 currentQuests.ADD(item[i],
lockedQuests[item[i]])

                 lockedQuests.REMOVE(item[i])

             ENDFOR

         ENDIF

         RETURN item[0]

     ENDFUNCTION

     FUNCTION CompleatQuest(STRING quest)

         IF(currentQuests.CONTAINS(quest))

             compleatedUnclaimedQuests.ADD(quest,
currentQuests[quest]

             currentQuests.REMOVE(quest)

         ENDIF

     ENDFUNCTION
```

```
END STATIC CLASS
```

MISCELLANEOUS

Unity Type Replacements

I will need to replace some of Unity's types as they are not serializable most notably Vector3, Vector2, and Quaternion. I will do this with a simple serializable struct and add an implicit conversion between the Unity version and my own. All the conversions are effectively the same so only the pseudo code for Vector3 is shown arithmetic and logic operators must also be added and the pseudo code will be the in the same format as said before.

For the Quaternion a forth variable would be added called w that could be calculated from x, y, and z. For the Vector2 the z variable would be removed as it is not needed. The conversion and mathematical functions would also be changed to reflect the addition and removal of variables but would otherwise remain untouched.

Vector 3 Pseudo Code

```
START

STRUCT THVector3

      FLOAT x

      FLOAT y

      FLOAT z


      CONSTRUCTOR THVector3(FLOAT _x, FLOAT _y, FLOAT _z)

            x = _x

            y = _y

            z = _z

      ENDCONSTRUCTOR

      IMPLICITCONVERSION THVector3(Vector3 vec3)

            RETURN NEW THVector3(vec3.x, vec3.y, vec3.z)

      ENDIMPLICITCONVERSION

      IMPLICITCONVERSION Vector3(THVector3 vec3)

            RETURN NEW Vector3(vec3.x, vec3.y, vec3.z)

      ENDIMPLICITCONVERSION

      ARITHMETICOPERATOR +( THVector3 a, THVector3 b)

            a.x += b.x

            a.y += b.y
```

```
        a.z += b.z
        RETURN a
    ENDARITHMETICOPERATOR
    ARITHMETICOPERATOR -( THVector3 a, THVector3 b)
        a.x -= b.x
        a.y -= b.y
        a.z -= b.z
        RETURN a
    ENDARITHMETICOPERATOR
ENDSTRUCT
END
```

Unity Input Replacement

I will also need to replace the Unity input method as it does not allow for rebinding of controls while the game is running. This is however a behaviour that I would like to include their fore I will be replacing the front end of the input method with my own. I will do his by having a dictionary of keys each assigned a name, I will use this as it will make it easy to access and use the input methods. The dictionary itself will not be accessible outside of the class as this could mean that it is possible to edit it in an improper way possibly causing a crash. To check if the input has been given the name of the input will be used and the keyboard button assigned to the name check for input by the default Unity methods.

Keyboard/Mouse Input Pseudo Code

```
STATIC CLASS Input
    DICTIONARY keyBindings = <STRING, KEYBOARDBUTTON>{"Open Inventory", k}
    METHOD AddBinding(STRING function, KEYBOARDBUTTON key)
        IF(!keyBindings.CONTAINSKEY(function))
            keyBindings.ADD(function, key)
        ENDIF
    ENDMETHOD
    METHOD ChangeBinding(STRING function, KEYBOARDBUTTON key)
        keyBindings[function] = key
    ENDMETHOD
    METHOD GetInput(STRING function)
        RETURN UNITYINPUTMETHOD(keyBindings[function])
    ENDMETHOD
ENDCLASS
```

KEY VARIABLES

Variable Name	Data type
Player Position	THVector3
Player Inventory	Item[]
Terrain Modifications	Save file outside of the game
Item Inventory's	Item[]
Completed Quests	List<Quests>

These will be key variables as many of the game components will needed to have access to the player position and inventory. The modifications to the terrain will also need to be kept track of as the player would not be pleases if they find that some of their work has not been saved between closing and reopening the game. The inventories of blocks/items will also need to be stored as again the player would not be happy if the game did not save the items that were inserted into an inventory after it has been closed and would most likely consider it a bug if the items where not saved or somehow returned to the player. Completed quests will also need to be saved and kept track of as if they weren't the player would not be able to progress through the game.

Classes

Class Name	Function	Parent Class
Item	Base class for all items and blocks	N/A
Block	Base class for all blocks	Item
Specialised Blocks	Class for a specific block	Block
Specialized Item	Class for a specific item	Item
Chunk	Holds a Chunk	MonoBehaviour
LoadChunks	Loads chunks into the game	MonoBehaviour
World	Stores active chunks	MonoBehaviour
Terrain	Interface between game and world	N/A
Player	Controls the player	MonoBehaviour
Inventory	Base inventory class	MonoBehaviour
InventorySlot	A slot in an inventory	MonoBehaviour
PlayerInventory	Players Inventory	Inventory
Block Inventory	A blocks inventory	Inventory
Crafting Inventory	An inventory that crafts things	Inventory

USABILITY FEATURES

Some usability features will need to be incorporated for the user to be able to play the game.

The game could also have different save files that can be loaded. This would be a good usability feature as it multiple people could play the game on the same computer or one user could possibly want to test something but not want to affect their own file. Having the ability to load a different world would be very useful for solving this problem as multiple players could have their own world without affecting another on the same computer/game. It could also be interesting to password protect the save files as some people may want to maliciously edit another's same file without their knowledge and a simple password would help prevent this as this will prevent 90% of users as even if it does not actually encrypt the file it will act as a strong enough deterrent.

TEST DATA

Some data that will need to be monitored is the block array for each chunk. That can be done by locking the seed in the Perlin Noise function so that it will reproduce the same data on each run. Allowing the system to be tested with the same data set multiple times.

The same world can also be loaded multiple times to test things and check that each new block/item added to the game is saved and loaded correctly.

Test Number	Description of test	Input data	Expected result	Actual result	Type
1	Save	Keypress	Data saved correctly		Iterative
2	Loading	Starting the game	Data loaded correctly		Iterative
3	Forward Movement	Keypress	Character moves forward		Iterative
4	Backward moment	Keypress	Character moves backward		Iterative
5	Left Movement	Keypress	Character moves to the left		Iterative
6	Right Moment	Keypress	Character moves to the right		Iterative
7	Terrain Loads	Noise function	Terrain loads corrects with no seams		Iterative
8	Block destruction works/saves correctly	Keypress/File	Block is destroyed correctly without causing a crash and file is loaded correctly		Iterative

9	Block placement works/saves correctly	Keypress/File	Block is placed correctly in the correct position without causing a crash and saved/loaded correctly when the game is restarted		Iterative
10	Inventories save stored items	Items inside an inventory	Items are in the same position, of the same type, and number as then input into the inventory		Iterative
11	Items can be input into an inventory	Items moved into an inventory	Item can be added to an inventory without duplication and without being able to override other items		Iterative
12	Items can be removed from an inventory	Keypress	Items can be removed form an inventory without duplication, overriding other items, and remove the correct number of items		Iterative
13	Game runs on multiple different systems	Computers	Game runs correctly on multiple		Post-Development

			different systems with different specs		
14	Save file naming	Acceptable name (no special characters)	File is named as why the user inputs		Post-Development
15	Save file naming	Erroneous data (special characters	File name is rejected, and user is notified		Post-Development

SCREEN MOCK-UPS

Player Inventory

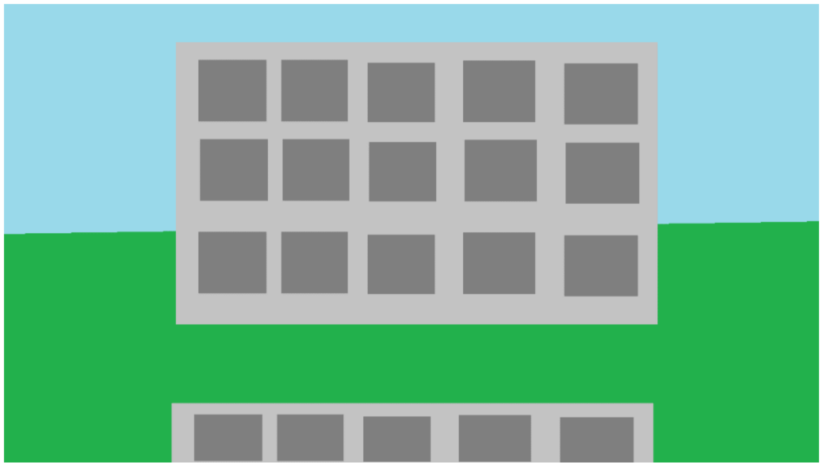


Figure 12 – Player Inventory

This is an example of the player inventory. Each dark grey square is a slot the player can insert an item into, they will also be objects that have the InventorySlot class attached to them, and then will be controlled by the PlayerInventory class attached to the player. The player will press a button to show and hide (open/close) the inventory. Each slot will show an image of the item it contains make it very user friendly as the player will not have to read lots of text can instead just quickly look for an image to find an item. The player will interact with the inventory using the mouse to select a slot and left/right mouse buttons making the interactions very natural for the player meaning that it will be easy to use. The system will also have built in error checking to ensure that items cannot be overridden meaning that the player will not have to think about that.

Chest Inventory

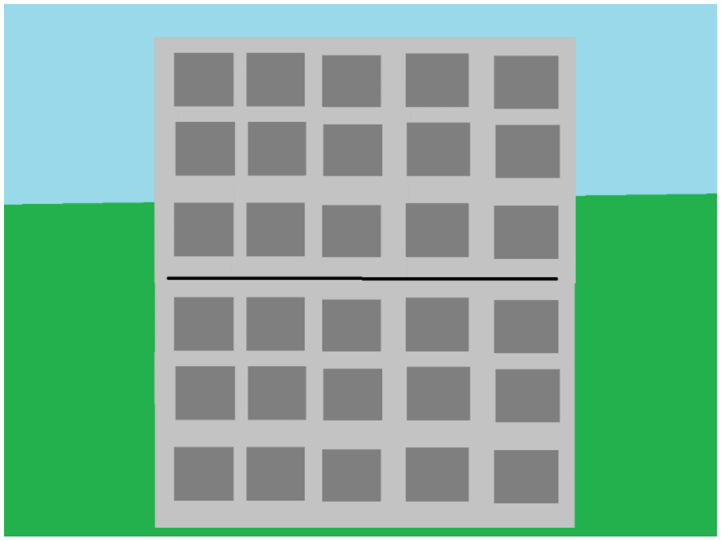
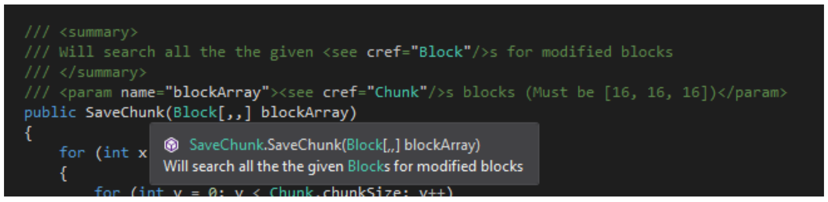


Figure 13 – Item Inventory

The chest inventory will work very similar to the player inventory. However, the inventory will be attached to a block, there for the user will first have to interact with a block to open the inventory, this will be done by the player looking at the block them pressing a button. This will make the usage of the inventory feel very natural and an easy thing to learn how to do. Another difference is that this inventory will be controlled by a class deriving from BlockInventory. The dividing line between the top and bottom inventory makes it clear to the player that the top is the block and the bottom is the player inventory. The two-different inventory will act as one when the player is using them making it very easy to use.

Development

Code comments are shown in the code snippets and a high-level explanation is contained within each section. In the code some comments are preceded by /// and contain HTML style tags, these are XML comments that the IDE (Visual Studio) can use to give information about a method, an example of an XML comment and its result is shown in Figure 14 – XML Comment Example.



```
/// <summary>
/// Will search all the the given <see cref="Block"/>s for modified blocks
/// </summary>
/// <param name="blockArray"><see cref="Chunk"/>s blocks (Must be [16, 16, 16])</param>
public SaveChunk(Block[,,] blockArray)
{
    for (int x
    {
        for (int y = 0; y < Chunk.chunkSize; y++)
```

```
SaveChunk.SaveChunk(Block[,,] blockArray)
Will search all the the given Blocks for modified blocks
```

Figure 14 – XML Comment Example

RENDERING BLOCKS

The first thing that I should ensure works is to generate the blocks, because without it the player would have nothing to interact with; this would not be very fun for the player, so I did it first.

Unity does come with its own cube object that has its own collider and such, however each side of the block cannot be rendered individually. This would mean that when 2 blocks are next to each other, 4 whole extra triangles are being rendered. This would not be a problem with only 2 blocks, however with a maximum of 16^3 blocks each having 2 triangles per face, that means there would be an extra 46,080 faces rendered. This poses a large performance impact on the final product; therefore, I will be rendering my own blocks that I can turn on and off faces when needed to save on rendering. This means I can also use a custom collision mesh for the chunk, reducing the number of collision calculations needed.

The Block Class

```
namespace TerrainGeneration
{
    public class Block
    {
//the % of pixels of the image that an image takes up in the texture atlas
        const float tileSize = 0.10f;

        public virtual Tile TexturePosition(BlockDirection direction)
        {
//the x, y position of the texture in the texture atlas so 1,0 would be a stone texture
            return new Tile() { x = 0, y = 0 };
        }
//UV = texture
        public virtual THVector2[] FaceUVs(BlockDirection direction)
        {
            THVector2[] uVs = new THVector2[4];

            Tile tilepos = TexturePosition(direction);
//gets the texture for each corner of the face added in a clockwise order
```

```
            uVs[0] = new THVector2(tileSize * tilepos.x + tileSize, tileSize *
tilepos.y);
 uVs[1] = new THVector2(tileSize * tilepos.x + tileSize, tileSize * tilepos.y +
tileSize);
            uVs[2] = new THVector2(tileSize * tilepos.x, tileSize * tilepos.y +
tileSize);
            uVs[3] = new THVector2(tileSize * tilepos.x, tileSize * tilepos.y);

            return uVs;
        }

        public virtual MeshData BlockMeshData(Chunk chunk, int x, int y, int z,
MeshData meshData)
        {
//does the block above me have a solid face if no add this face to the chunk
mesh
            if(!chunk.GetBlock(x, y + 1, z).IsSolid(BlockDirection.DOWN))
                meshData = FaceDataUp(chunk, x, y, z, meshData);
            other faces checked but is not shown
//return the mesh to the whatever called this method with this blocks data added
            return meshData;
        }
//add each vertex of this face into the mesh
        protected virtual MeshData FaceDataUp(Chunk chunk, int x, int y, int z,
MeshData meshData)
        {
//Added in a clockwise order, x is the blocks position in the chunk then +- the
vertex offset from that position
            meshData.verts.Add(new Vector3(x - 0.5f, y + 0.5f, z + 0.5f));
            meshData.verts.Add(new Vector3(x + 0.5f, y + 0.5f, z + 0.5f));
            meshData.verts.Add(new Vector3(x + 0.5f, y + 0.5f, z - 0.5f));
            meshData.verts.Add(new Vector3(x - 0.5f, y + 0.5f, z - 0.5f));
//make triangles out of the added vertices
            meshData.AddQuadTriangles();
//add uv's to the meshdata
            meshData.uv.AddRange(FaceUVs(BlockDirection.UP).ToList());
            return meshData;
        }
methods to add other faces omitted as they are all the same just with different
numbers

//return true if the given face is a solid face, in this case all faces are
solid so return true, can be easily changed to a switch statement if different
answers are needed for different faces
        public virtual bool IsSolid(BlockDirection direction)
        {
            return true;
        }
    }

    public struct Tile {public int x; public int y; }
}
```

Code 1 – Base Block Class

Support Classes and Assets

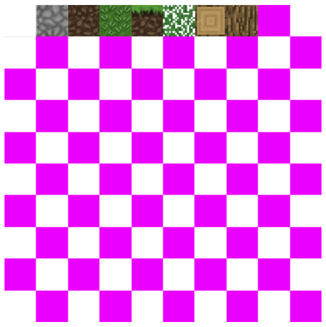


Figure 15 – Block Texture Atlas

```
namespace TerrainGeneration
{
    public class MeshData
    {
//stores the vertices in the mesh
        public List<THVector3> verts = new List<THVector3>();
//stores the triangles in the mesh
        public List<int> tris = new List<int>();
//stores the texture positions of each vertex in the mesh
        public List<THVector2> uv = new List<THVector2>();
//stores the collider vertices
        public List<THVector3> colVerts = new List<THVector3>();
//stores the collider triangles
        public List<int> colTris = new List<int>();
//collider verts and tris are split from normal verts and tris ad some blocks
may have a different collider to the model that is rendered therefore it makes
sense to split them allowing for this kind of functionality, but is not
currently used

        public void AddQuadTriangles()
        {
//makes 2 triangles from the last 4 verts given. Assumes that vertices are given
in the correct order
            tris.Add(verts.Count - 4);
            tris.Add(verts.Count - 3);
            tris.Add(verts.Count - 2);
            tris.Add(verts.Count - 4);
            tris.Add(verts.Count - 2);
            tris.Add(verts.Count - 1);
        }
}
```

Code 2 – Mesh Data Class

```
namespace BeeGame
{
    [System.Serializable]
    public struct THVector3
    {
        public float x;
        public float y;
        public float z;

        //constructors allowing construction from different data types
        public THVector3(Vector3 vector)
        {
            x = vector.x;
            y = vector.y;
            z = vector.z;
        }
        public THVector3(float _x, float _y, float _z)
        {
            x = _x;
            y = _y;
            z = _z;
        }

equality operators, ==, !=, and .Equals() as cannot use standard because default
will check for same hashcode not actual variables

addition, subtraction, multiplication, and division of THVector3 by other
THVector3's

allowing addition, subtraction, multiplication, and division of THVector3's by
floats

//implicit conversation between THVector3 and Unity Vector3
        public static implicit operator THVector3(Vector3 vec3)
        {
            return new THVector3(vec3);
        }
        public static implicit operator Vector3(THVector3 vec3)
        {
            return vec3.ToUnityVector3();
        }
    }
}
```

Code 3 – THVector3 Class

Chunk

```
namespace TerrainGeneration
{
    //components required to be on the game object for the script to work (when
the script is added these 3 things will be added automatically)
    [RequireComponent(typeof(MeshFilter))]
    [RequireComponent(typeof(MeshRenderer))]
    [RequireComponent(typeof(MeshCollider))]
    public class Chunk : MonoBehaviour
    {
//array of blocks in the chunk
        private Block[,,] blocks;
//length with height of the chunk as it will be a square
        public static int chunkSize = 16;
//should the chunk be updated? (will be used when blocks are added/removed)
        public bool update = true;
        private MeshFilter mfilter;
        private MeshCollider mcollider;

        void Start()
        {
//get the Unity mesh filter component of the game object for use
            mfilter = gameObject.GetComponent<MeshFilter>();
//get the Unity mesh collider component of the game object for use
            mcollider = gameObject.GetComponent<MeshCollider>() as MeshCollider;

            blocks = new Block[chunkSize, chunkSize, chunkSize];

//fill the block array with Air blocks so that each item in the array contains a
block
            for (int x = 0; x < chunkSize; x++)
            {
                for (int y = 0; y < chunkSize; y++)
                {
                    for (int z = 0; z < chunkSize; z++)
                    {
                        blocks[x, y, z] = new Blocks.Air();
                    }
                }
            }
//set a block in the array to a block for testing
            blocks[0, 0, 0] = new Block();

            UpdateChunk();
        }

        public Block GetBlock(int x, int y, int z)
        {
            if(x >= 0 && y >= 0 && z >= 0)
            {
                return blocks[x, y, z];
            }
            else
            {
                return new Air();
            }
        }

//makes the chunk mesh
        void UpdateChunk()
        {
            MeshData mesh = new MeshData();
//goes through each block and adds it mesh data to the chunks mesh
            for (int x = 0; x < chunkSize; x++)
            {
                for (int y = 0; y < chunkSize; y++)
                {
                    for (int z = 0; z < chunkSize; z++)
                    {
                        mesh = blocks[x, y, z].BlockMeshData(this, x, y, z,
mesh);
```

```
                    }
                }
            }
//render and make the collider
            RenderMesh(mesh);
            ColliderMesh(mesh);
        }

        void RenderMesh(MeshData mesh)
        {
//clean out the pervious mesh
            mfilter.mesh.Clear();
//convert the verts and tris to the correct format and put it into the mesh
filter variable
            mfilter.mesh.vertices = mesh.verts.ToArray().ToUnityVector3Array();
            mfilter.mesh.triangles = mesh.tris.ToArray();
//apply the textures
            mfilter.mesh.uv = mesh.uv.ToArray().ToUnityVector2Array();
//make light do the correct things however they should already be correct
            mfilter.mesh.RecalculateNormals();
        }

        void ColliderMesh(MeshData mesh)
        {
//for now just apply the rendered mesh as the collider
            mcollider.sharedMesh = mfilter.mesh;
        }
    }
}
```

Code 4 – Chunk Class

The Block class, show in Code 1 – Base Block Class, is a base class for all other blocks in the game, and will most likely be the child of the item class in the future, therefore I needed it to be easily extendable. I will most likely go back to it in the future and add or remove methods that other blocks will need but are currently not needed.

The helper class, Code 2 – Mesh Data Class, is what stores the mesh whilst it is being created this is because when a mesh is made each block is added individually this a slow process so in the future I will attempt some experimentation with threading this process, however the unity Mesh class is tagged as “not thread safe” this means that I must use some form of intermediate data structure to store the mesh whilst it is being made, then transfer that data to the Unity mesh data type. The Unity mesh class stores vertices and triangles are stored as an array as the number of vertices and triangles is not actually known before all the chunks blocks are checked, so using a List mesh data class means that the vertices and triangles are much easier to work with because the List can be resized at any time.

The helper class, Code 3 – THVector3 Class, is my implementation of the Unity Vector3 class. I have also implemented a version of the Unity Vector2 named THVector2 but us not shown as it is almost the same as THVector3. The reason this I did this is because the Unity Vector class cannot be saved at runtime this is a functionality that I would like to have so I implemented the class in a way that is serializable at runtime.

The Chunk class, Code 4 – Chunk Class, is what will contain the blocks. In the final game their will, be multiple chunks building up the terrain.

The textures for the blocks, Figure 15 – Block Texture Atlas, were taken from a Minecraft texture pack then resized, merged into one texture atlas, and then some blurring some was don't to make the textures smoother. A texture atlas is used so that Unity can load all the textures as one material file improving performance as opposed to the other option which is each block loading its own texture from a file.

The result of this test is shown in Figure 16 – Rendered Block, and as can be seen the code works.

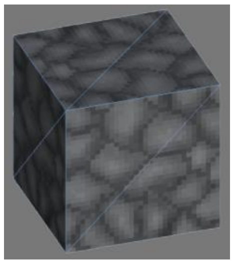


Figure 16 – Rendered Block

RENDERING MULTIPLE BLOCKS

After this I worked on rendering multiple blocks and from that multiple chunks. To render multiple blocks, I changed the code shown in Code 4 so that instead of filling the block array with Air it is filled with Blocks (see Figure 16 for image). This actually worked fine as if the block is at the edge of the chunk the GetBlock() function shown in Code 4 will return an Air Block meaning that there will be no error.

The next step was to render multiple chunks. For this the method in Code 4 – Chunk Class, GetBlock() needed to be updated so that the blocks in the edge of one chunk but next to another can interact with each other. For this I decided to implement another class, Code 5 – World Class, that contains a list of all the chunks in the world. Then the GetBlock() method would access this to get the neighboring chunk using this array that the block in the specific location in the chunk would be returned. If the chunk however did not exist, then an air block would be returned. The changes to the GetBlock() from the chunk class are shown in Code 6 – Get Block Method in Chunk Class.

To store the chunks, I decided to use a dictionary as it gives a logical method to access the chunk because the center coordinates of the chunk can be used as the key then any given x, y, z coordinates can be translated to a specific chunk, providing that it exists.

```
namespace TerrainGeneration
{
    public class World : MonoBehaviour
    {
//the dictionary of the made chunks
```

```
        public Dictionary<THVector3, Chunk> chunks = new Dictionary<THVector3, C
hunk>();

        public GameObject chunkPrefab;

        public void CreateChunk(int x, int y, int z)
        {
// coordinates of the chunk on the world
            THVector3 worldPos = new THVector3(x, y, z);

//makes the chunk in the world
            GameObject newChunkGameObject = Instantiate(chunkPrefab, worldPos, Q
uaternion.identity);

//gets the Chunk component so that we can do things to it
            Chunk newChunk = newChunkGameObject.GetComponent<Chunk>();

//Set the chunks position and world
            newChunk.worldPos = worldPos;
            newChunk.world = this;

//Adds the new chunk to the dictionary
            chunks.Add(worldPos, newChunk);

            var terrainGen = new TerrainGenerator();
            newChunk = terrainGen.ChunkGen(newChunk);
        }

        public Chunk GetChunk(int x, int y, int z)
        {
//coordinates of the chunk in the world
            THVector3 vec3 = new THVector3(x, y, z);

            float multiple = Chunk.chunkSize;

// Divides given coord as a float then tiems by the chunk size so that the world
coordinates is set to the correct chunk, e.g. world coord is (16, 1, 0) the chun
k coord would be (16, 0, 0) (dividing float by 0 is 0) as chunks are laid out in
a grid
            vec3.x = (int)Math.Floor(x / multiple) * multiple;
            vec3.y = (int)Math.Floor(y / multiple) * multiple;
            vec3.z = (int)Math.Floor(z / multiple) * multiple;
            chunks.TryGetValue(vec3, out Chunk containter);

            return containter;
        }

        public Block GetBlock(int x, int y, int z)
        {
// Gets the correct chunk the block is in
            Chunk chunk = GetChunk(x, y, z);

            if(chunk != null)
            {
//return the block in the chunk after converting the given coordinates to chunk
coordinates from world coordinates
                return chunk.GetBlock(x - (int)chunk.worldPos.x, y -
(int)chunk.worldPos.y, z - (int)chunk.worldPos.z);
            }
            else
            {
//if the chunk cannot be found then return an air block
                return new Blocks.Air();
            }
        }
    }
}
```

Code 5 – World Class

```
public Block GetBlock(int x, int y, int z)
{
//checks that the given x, y, z is actually in this chunk, of it is not check
the correct chunk
    if(InRange(x) && InRange(y) && InRange(z))
    {
        return blocks[x, y, z];
    }
    else
    {
// looks at block in the neighboring chunk
        return world.GetBlock((int)worldPos.x + x, (int)worldPos.y + y,
(int)worldPos.z + z);
    }
}

bool InRange(int i)
{
    if(i >= 0 && i < chunkSize)
        return true;

    return false;
}
```

Code 6 – Get Block Method in Chunk Class

Using these changes, I managed to successfully generate multiple chunks, this can be seen in Figure 17 – Multiple Chunks

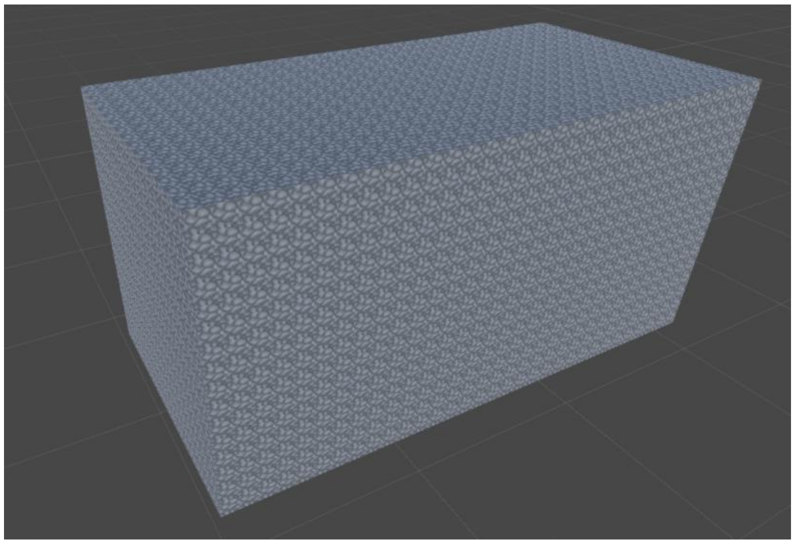


Figure 17 – Multiple Chunks

The blocks inside the chunk do not render faces that cannot be seen by the player making this a very efficient system, with most of the rest of the work left to the unity rendering system, which is far too complex for me to even imagine messing with. Now the final step of the generation system is to generate a terrain rather than just 2 boring chunks.

TERRAIN GENERATION

To generate a more Minecraft looking terrain a noise function will be needed. After looking at Wikipedia for both Perlin[1] and Simplex Noise[2] and having no idea what the math's equations meant I when looking for an open code sample in C# that I could use. Turns out that there is a lot of implementations out there, however I did have some criteria that allowed me to narrow down my search more easily. Firstly, it had to be fast as it would be called many time. Secondly, it must have some way of adding a random seed so that each time a new save file is created the games environment looks different, and finally it must have a license allowing me to use it. After some time researching this I decided to use a version by Heikki Törmälä which can be found on GitHub using this link: https://goo.gl/Q6arG8. Now with this all that is needed to be done is add it into the code and start generating the terrain.

To generate the terrain each chunk is generated in vertical strips with each block in the strip having a block placed in it. The code for this is shown in Figure 18 – Generated Terrain, and the results are shown in Figure 18 – Generated Terrain.

```
// Generates the terrain for the game
    public class TerrainGeneration
    {
```

[1] https://en.wikipedia.org/wiki/Perlin_noise
[2] https://en.wikipedia.org/wiki/Simplex_noise

```
// Base height of stone
        private float stoneBaseHeight = -24;
// Base noise of stone
        private float stoneBaseNoise = 0.05f;
// Base noise height for stone
        private float stoneBaseNoiseHeight = 6;

// Base height for a mountain
        private float stoneMountainHeight = 30;
// Frequency of mountains (larger value = more choppy terrain)
        private float stoneMountainFrequency = 0.008f; //0.008f;
// Minimum height for stone
        private float stoneMinHeight = -12;

// generates a chunk in a new thread
        public Chunk ChunkGen(Chunk chunk)
        {
            Chunk outChunk = chunk;
            lock (chunk)
            {
                Thread thread = new Thread(() => ChunkGenThread(chunk, out
outChunk)) { Name = $"Generate Chunk Thread @ {chunk.chunkWorldPos}"};

                thread.Start();
                return outChunk;
            }
        }

/// <summary>
/// Generates a new <see cref="Chunk"/>
/// </summary>
/// <param name="chunk"><see cref="Chunk"/> to be generated</param>
/// <param name="outChunk">Generated <see cref="Chunk"/> to return</param>
        public void ChunkGenThread(Chunk chunk, out Chunk outChunk)
        {
// for each x and z position in teh chunk
            for (int x = chunk.chunkWorldPos.x-3; x < chunk.chunkWorldPos.x +
Chunk.chunkSize + 3; x++)
            {
                for (int z = chunk.chunkWorldPos.z-3; z < chunk.chunkWorldPos.z
+ Chunk.chunkSize + 3; z++)
                {
                    chunk = GenChunkColum(chunk, x, z);
                }
            }

            chunk.SetBlocksUnmodified();
            outChunk = chunk;
        }

//generates a column of the chunk
        public Chunk GenChunkColum(Chunk chunk, int x, int z)
        {
// the height of the mountain
            int stoneHeight = Mathf.FloorToInt(stoneBaseHeight);
            stoneHeight += GetNoise(-x, 0, z, stoneMountainFrequency,
Mathf.FloorToInt(stoneMountainHeight));

// if the column is currently to low make it not so low
            if (stoneHeight < stoneMinHeight)
                stoneHeight = Mathf.FloorToInt(stoneMinHeight);

// add the height of normal stone on to the mountain
            stoneHeight += GetNoise(x, 0, -z, stoneBaseNoise,
Mathf.RoundToInt(stoneBaseNoiseHeight));

            //* set the colum to the correct blocks
            for (int y = chunk.chunkWorldPos.y - 8; y < chunk.chunkWorldPos.y +
Chunk.chunkSize; y ++)
            {
// puts a layer of bedrock at the bottom the world
                if (y <= (chunk.chunkWorldPos.y) && chunk.chunkWorldPos.y == -
16)
                {
                    SetBlock(x, y, z, new Blocks.Bedrock(), chunk);
                }
                else if (y <= stoneHeight)
                {
                    SetBlock(x, y, z, new Blocks.Block(), chunk);
                }
                else
                {
                    SetBlock(x, y, z, new Blocks.Air(), chunk);
                }
            }

                return chunk;
        }

/// <summary>
/// Get a noise value
/// </summary>
/// <param name="x">X pos of the noise</param>
/// <param name="y">Y pos of the noise</param>
/// <param name="z">Z pos of the noise</param>
/// <param name="scale">What the step shout be from the last x, y, z</param>
/// <param name="max">Max value of the noise</param>
/// <returns>A noise value as an int</returns>
        public static int GetNoise(int x, int y, int z, float scale, int max)
        {
            return Mathf.FloorToInt((SimplexNoise.Generate(x * scale, y * scale,
z * scale) + 1f) * (max / 2f));
        }

/// <summary>
/// Sets a <see cref="Block"/> in the given position
/// </summary>
/// <param name="x">X pos of the block</param>
/// <param name="y">Y pos of the block</param>
/// <param name="z">Z pos of the block</param>
/// <param name="block"><see cref="Block"/> to set</param>
/// <param name="chunk"><see cref="Chunk"/> to set the block in</param>
/// <param name="replacesBlocks">Can a block that exists be replaced?</param>
        public static void SetBlock(int x, int y, int z, Blocks.Block block,
Chunk chunk, bool replacesBlocks = false)
        {
// corrects the x, y, z pos of the so that the block is placed in the correct
position
            x -= chunk.chunkWorldPos.x;
            y -= chunk.chunkWorldPos.y;
            z -= chunk.chunkWorldPos.z;

// checks that the block is in the chunk and that no block is already their then
sets it
            if (Chunk.InRange(x) && Chunk.InRange(y) && Chunk.InRange(z))
                if (replacesBlocks || chunk.blocks[x, y, z] == null)
                    chunk.SetBlock(x, y, z, block, false);
        }
    }
```

Code 7 – Terrain Generation

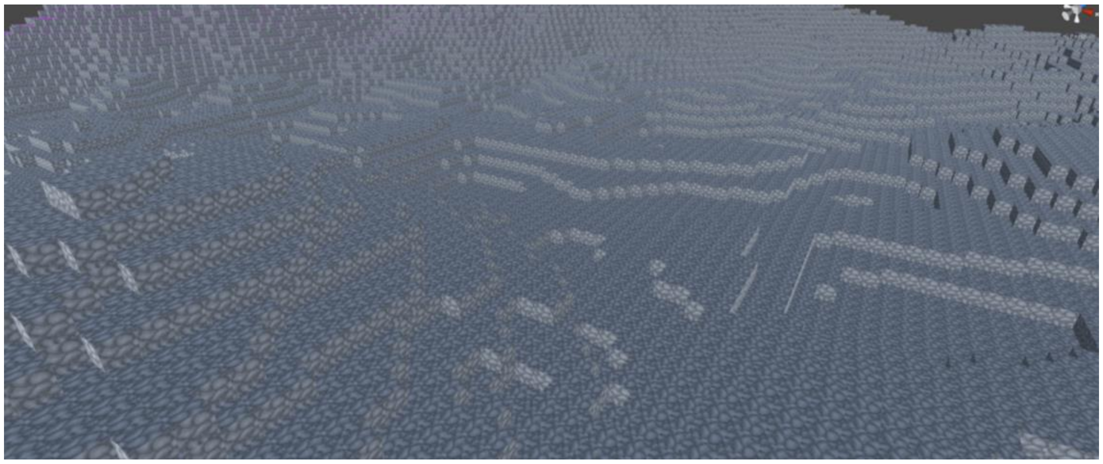


Figure 18 – Generated Terrain

After some experimentation with the performance impact of threading the terrain I decided to attempt to use it, however I am unsure if it will introduce some unexpected behavior, so I will leave it in until it does. With this new system I also generate the terrain around the player and delete the chunks that are to far away from the player. This is done with an array of THVector's that are translated to the players position at runtime.

The next step is to add layers to the terrain with dirt and grass. this is easily done with a simple if statement and some more variables to control the noise scale and such. The result is shown in Figure 19 – Terrain with Grass and Dirt, as can be seen I also added trees, to make the trees I find the first trunk block then use then generate the trunk and the leaves using its own function as the tree spans multiple chunks and multiple vertical slices of the chunk therefore it is easiest for it to have its own function this function is shown in Code 8 – Tree Generation Code.

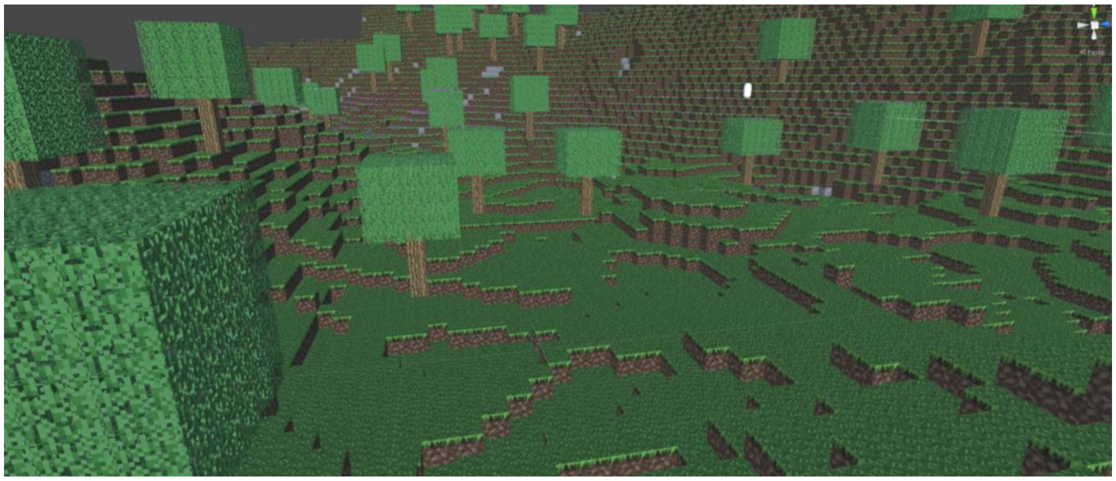


Figure 19 – Terrain with Grass and Dirt

```
void CreateTree(int x, int y, int z, Chunk chunk)
{
// makes the leaves of the tree
    for (int xi = -2; xi <= 2; xi++)
    {
        for (int yi = 4; yi <= 8; yi++)
        {
            for (int zi = -2; zi <= 2; zi++)
            {
                SetBlock(xi + x, yi + y, zi + z, new Blocks.Leaves(), chunk,
true);
            }
        }
    }
// makes the trunk of the tree
    for (int i = 0; i < 6; i++)
    {
        SetBlock(x, y + i, z, new Blocks.Wood(), chunk, true);
    }
}
```

Code 8 – Tree Generation Code

The Given x, y, z in the function is the position of the base trunk block.

BLOCK DESTRUCTION PLACEMENT AND SAVING

The final part of the game to do with the terrain so for the player to be able to destroy and place blocks and have the changes to the terrain be saved between sessions. However, to be able to do this efficiently the saving/loading cannot be done constantly as this could cause a performance dip if large files are constantly written to the disk. Also, saving between delays could also cause problems as It is possible that a modified chunk could become unloaded before it’s changes are saved, this would annoy the player and cause a large performance dip at regular intervals making the game annoying to play. So, I came up with a solution to make the files small and for the game to only save when changes have been made. This was done by adding a new variable to the block class that sates weather or not the block has been placed by the player. This also meant that block placing, and destruction were the same thing as destroying a block is effectively placing and Air block. But, the placement and destruction still needed 2 separate functions for this, so I annoyingly didn't cut my work down at all. The player also needed some way to see which block was being selected by them, so I added the placement, destruction, and selector all in one class as both functions rely on the position of the selector making it logical for them to be in the same place.

Breaking and Placing

```
namespace Player
{
/// <summary>
/// Moves the <see cref="Block"/> selector
/// </summary>
    public class Selector : MonoBehaviour
    {
/// <summary>
/// Selector
/// </summary>
        public GameObject selector;

/// <summary>
/// Layers for the selector to look at
/// </summary>
        public LayerMask layers;
/// <summary>
/// Where the raycast hit
/// </summary>
        private RaycastHit hit;

// Make the selector
        void Awake()
        {
            selector = Instantiate(selector);
        }

// Updates the selector <TODO> stop updating if an inventory is open
        void FixedUpdate()
        {
             UpdateSelector();
        }

// Breaks and places a Block
        void Update()
        {
uses custom input class meaning key bindings can be updated at runtime
             if (GetButtonDown("Break Block"))
                 BreakBlock();
             if (GetButtonDown("Place"))
                 PlaceBlock();
        }

/// <summary>
/// Updates the selectors position
/// </summary>
```

```
        public virtual void UpdateSelector()
        {
            if (Physics.Raycast(transform.position, transform.forward, out hit,
15, layers))
            {
                selector.SetActive(true);
                selector.transform.position = GetBlockPos(hit);
            }
            else
            {
                selector.SetActive(false);
            }
        }

/// <summary>
/// Breaks the <see cref="Block"/> in the selectors postion
/// </summary>
        public virtual void BreakBlock()
        {
            Chunk chunk = GetChunk(selector.transform.position);

            Block block =
chunk.world.GetBlock((int)selector.transform.position.x,
(int)selector.transform.position.y, (int)selector.transform.position.z);

            if (!block.breakable)
                return;

            chunk.world.SetBlock((int)selector.transform.position.x,
(int)selector.transform.position.y, (int)selector.transform.position.z, new
Air(), true);
// set to changed so when block is placed down again it will be saved
            block.changed = true;
            block.BreakBlock(selector.transform.position);
        }

/// <summary>
/// Places s <see cref="Block"/> in the selector postion
/// </summary>
        public virtual void PlaceBlock()
        {
            Chunk chunk = GetChunk(selector.transform.position);

            if (chunk == null)
                return;

            item = new Block()

            if (item != null)
            {
                    chunk.world.SetBlock((int)(selector.transform.position.x +
hit.normal.x), (int)(selector.transform.position.y + hit.normal.y),
(int)(selector.transform.position.z + hit.normal.z), item, true);
            }
        }
    }
```

```
}
```

Code 9 – Selector Class

```
namespace Core
{
/// <summary>
/// My implementation of the unity input system. Acts as a buffer layer to the
unity system so that the input keys can be changed at runtime
/// </summary>
    public static class THInput
    {
/// <summary>
/// Button identifiers and <see cref="KeyCode"/>
/// </summary>
        private static Dictionary<string, object> inputButtons = new
Dictionary<string, object>()
        {
            {"Forward", KeyCode.W },
            {"Backward", KeyCode.S },
            {"Right", KeyCode.D },
            {"Left", KeyCode.A },
            {"Place", KeyCode.Mouse1 },
            {"Break Block", KeyCode.Mouse0 },
            {"Jump", KeyCode.Space }
        };

/// <summary>
/// Has the given button been pressed this update
/// </summary>
/// <param name="button">The button name eg "Inventory"</param>
/// <returns>true if the given button has been pressed this update</returns>
        public static bool GetButtonDown(string button)
        {
            if (!inputButtons.ContainsKey(button))
            {
                throw new InputException($"Key input name not defined:
{button}");
            }

            switch (inputButtons[button])
            {
                case KeyCode[] arry:
// for each possible key, check if it was pressed and if it was return that it
was; if none of them was pressed, return false
                    foreach (var item in arry)
                    {
                        if (Input.GetKeyDown(item))
                        {
                            return true;
                        }
                    }

                    return false;
                default:
                    return Input.GetKeyDown((KeyCode)inputButtons[button]);
            }
        }

/// <summary>
/// Is the given button currently being held down
/// </summary>
/// <param name="button">The button name e.g. "Forward"</param>
/// <returns>true if the given button is currently being held down</returns>
        public static bool GetButton(string button)
        {
            if (!inputButtons.ContainsKey(button))
            {
                throw new InputException($"Key input name not defined:
{button}");
            }

            switch (inputButtons[button])
            {
                case KeyCode[] arry:
// for each possible key, check if it was pressed and if it was return that it
was; if none of them was pressed return false
                    foreach (var item in arry)
                    {
                        if (Input.GetKey(item))
                        {
                            return true;
                        }
                    }

                    return false;
                default:
                    return Input.GetKey((KeyCode)inputButtons[button]);
            }
        }

/// <summary>
/// Has the given button been released this update
/// </summary>
/// <param name="button">Button name e.g. "Inventory"</param>
/// <returns>true if the button has been relaesed during this update</returns>
        public static bool GetButtonUp(string button)
        {
            if (!inputButtons.ContainsKey(button))
            {
                throw new InputException($"Key input name not defined:
{button}");
            }

            switch (inputButtons[button])
            {
                case KeyCode[] arry:
// for each possible key, check if it was pressed and if it was return that it
was; if none of them was pressed return false
                    foreach (var item in arry)
                    {
                        if (Input.GetKeyUp(item))
                        {
                            return true;
                        }
                    }

                    return false;
                default:
                    return Input.GetKeyUp((KeyCode)inputButtons[button]);
            }
        }

/// <summary>
/// Gets the axis of a button press
/// </summary>
/// <param name="axis">Axis to check, Horizontal or Vertical</param>
/// <returns>+1 or -1</returns>
        public static int GetAxis(string axis)
        {
            int returnAxis = 0;

            if (axis == "Horizontal")
            {
                if (GetButton("Right"))
                {
                    returnAxis += 1;
                }

                if (GetButton("Left"))
                {
                    returnAxis -= 1;
                }
            }
            else if (axis == "Vertical")
            {
                if (GetButton("Forward"))
                {
                    returnAxis += 1;
                }

                if (GetButton("Backward"))
                {
                    returnAxis -= 1;
                }
            }

            return returnAxis;
        }
    }
}
```

Code 10 – Unity Input Interface

Code 10 – Unity Input Interface is an interface that I made for the Unity input class, this allows me to more easily change key bindings at runtime by storing all the needed keys in a dictionary and altering the value associated with the key to change the function of that key. This is necessary as the default unity key bindings cannot be altered at runtime through code. This also allows me more freedom as it means that multiple keys can be assigned to the same button if needed, which is the reason for the switch statements and foreach loops in the input checking.

For testing the player will just place a Block that looks like stone, however when the inventory is implemented then the block that is selected by the player will be selected. To make it clear to the player what is selected a green cube is drawn as shown in Figure 20 – Selector in Game.

To move the selector, method seen in Code 9 – Selector Class: UpdateSelector(), each frame a ray is fired from the center of the players view. The position that the ray hits is then converted into the center of a block by the GetBlockPos() method, which is a static method accessible by all classes that converts arbitrary <x, y, z> coordinates into a blocks coordinates.

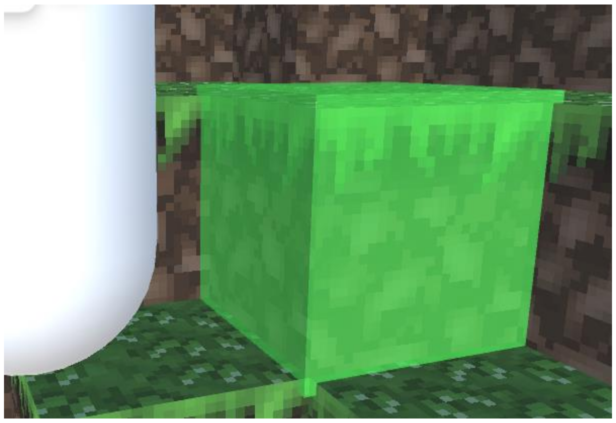


Figure 20 – Selector in Game

To place a block, method seen in Code 9 – Selector Class: PlaceBlock(), uses a new public function in the world class that will get the given coordinates, new method shown in Code 11 – New World Class Methods, the block is placed in the position that the selector is + the face normal of where the raycast hit. So, for example if the player is looking at the top of a block then the block will be placed above the one that is currently being looked at. There is no need to check if a block is already in that position as there is (should) be no way to select the space between 2 blocks that are next to each other.

```
/// <summary>
/// Sets a <see cref="Block"/> at the given position
/// </summary>
/// <param name="x">X pos of the block</param>
/// <param name="y">Y pos of the block</param>
/// <param name="z">Z pos of the block</param>
/// <param name="block"><see cref="Block"/> to be placed</param>
public void SetBlock(int x, int y, int z, Block block, bool saveChunk = false)
{
//*gets the chunk for the block to be placed in
    Chunk chunk = GetChunk(x, y, z);

//*if the chunk is not null and the block trying to be replaced is replaceable,
replace it
    if(chunk != null && chunk.blocks[x - chunk.chunkWorldPos.x, y -
chunk.chunkWorldPos.y, z - chunk.chunkWorldPos.z].breakable)
    {
        chunk.SetBlock(x - chunk.chunkWorldPos.x, y - chunk.chunkWorldPos.y, z - 
chunk.chunkWorldPos.z, block);
        chunk.update = true;

//*updates the nebouring chunks as when a block is broken it may be in the edge 
of the chunk so their meshes also need to be updated
//*only updates chunks that need to be updated as not every chunk will need to 
be and sometimes none of them will need to be

//*checks if the block changed is in the edge if the x value for the chunk
        UpdateIfEqual(x - chunk.chunkWorldPos.x, 0, new ChunkWorldPos(x - 1, y, 
z));
        UpdateIfEqual(x - chunk.chunkWorldPos.x, Chunk.chunkSize - 1, new 
ChunkWorldPos(x + 1, y, z));

//*checks if the block changed is in the edge if the y value for the chunk
        UpdateIfEqual(y - chunk.chunkWorldPos.y, 0, new ChunkWorldPos(x, y - 1, 
z));
        UpdateIfEqual(y - chunk.chunkWorldPos.y, Chunk.chunkSize - 1, new 
ChunkWorldPos(x, y + 1, z));

//*checks if the block changed is in the edge if the z value for the chunk
        UpdateIfEqual(z - chunk.chunkWorldPos.z, 0, new ChunkWorldPos(x, y, z - 
1));
        UpdateIfEqual(z - chunk.chunkWorldPos.z, Chunk.chunkSize - 1, new 
ChunkWorldPos(x, y, z + 1));
    }
}

/// <summary>
/// Gets a <see cref="Block"/> at the given position
/// </summary>
/// <param name="x">X pos of the block</param>
/// <param name="y">Y pos of the block</param>
/// <param name="z">Z pos of the block</param>
/// <returns><see cref="Block"/> at given x, y, z position</returns>
public Block GetBlock(int x, int y, int z)
{
//* gets the chunk that the block is in
    Chunk chunk = GetChunk(x, y, z);
    if(chunk != null)
    {
//* gets the block in the chunk
        return chunk.GetBlock(x - chunk.chunkWorldPos.x, y - 
chunk.chunkWorldPos.y, z - chunk.chunkWorldPos.z) ?? new Air();
    }
//* returns an empty block is the chunk was not found
    return new Air();
}
```

Code 11 – New World Class Methods

To break a block is it a very similar process, method shown in Code 9 – Selector Class: BreakBlock(). Firstly, the block at the selectors position is retrieved, then the block that is tyring to be broken is checked if it can actually be broken, next the block is replaced with Air if it can be broken, finally the BreakBlock() method in the block class is called, when items are implemented this will handle any special functions that breaking the block should trigger, for example a grass block may return a dirt block instead of grass, the position of the selector is also given in this method as the block does not know were in the world it is so the item returned when broken can be placed at the selectors position.

Saving Changes

Now that the player has a method to edit the world it would be nice of the changes were saved between running's of the game as this would give the player a sense of progress, rather than the futile attempts to change the world but with nothing they do changing anything (much like real life however I am not going for realism in this game). To do this efficiently I decided that the blocks should have a flag as to whether they have been edited by the player. Then a function can iterate through all the blocks in the chunk, create a dictionary out of the blocks that have been changed then save the dictionary in to a file that can be reloaded. This will keep file size down and improve performance as the block dictionary can be created in a separate thread so that the overall performance of the game is not affected, however I will need to be careful with this as it is possible that the player could make changes to quickly meaning that threads could complete out of order producing…interesting results.

```
namespace Serialization
{
    /// <summary>
    /// Serializes and DeSerialises things
    /// </summary>
    /// <remarks>
    /// Serialization is SLOW try to only serialize only what is absolutely 
necessary
    /// </remarks>
    public static class Serialization
    {
        #region Data
        /// <summary>
        /// Name of the world
        /// </summary>
        public static string worldName = "World";
        /// <summary>
        /// Save folder name
        /// </summary>
        public static string saveFolderName = "Saves";
        /// <summary>
        /// Path to save things
        /// </summary>
        private static string savePath;
        #endregion

        /// <summary>
        /// Set the save folder name
        /// </summary>
        /// <param name="saveFolder"></param>
        public static void SetSaveFolderName(string saveFolder)
        {
            saveFolderName = saveFolder;
        }

        /// <summary>
        /// Sets the paths for the save files
        /// </summary>
        public static void MakeDirectorys()
        {
            savePath = $"{Application.dataPath}/{saveFolderName}/{worldName}";

            if (!(Directory.Exists(savePath)))
                Directory.CreateDirectory(savePath);
        }

        /// <summary>
        /// Deletes the given file if it exists, Starts in <see
cref="Application.dataPath"/>
        /// </summary>
        /// <param name="fileName">File to delete, can be a file path that
starts in <see cref="Application.dataPath"/></param>
        public static void DeleteFile(string fileName)
        {
            string[] file = Directory.GetFiles(Application.dataPath + "/Saves",
"*.dat", SearchOption.AllDirectories);

            string[] splitCharacters = { "/", "\\", ".dat" };

            for (int i = 0; i < file.Length; i++)
            {
                string[] temp = file[i].Split(splitCharacters,
System.StringSplitOptions.RemoveEmptyEntries);

                if(temp[temp.Length - 1] == fileName)
                {
                    File.Delete(file[i]);

                    return;
                }
            }
        }

        /// <summary>
        /// Saves the player positon, rotation, and scale
        /// </summary>
        /// <param name="positon">Transform to get the data from</param>
        public static void SavePlayerPosition(Transform positon)
        {
            THVector3[] playerTransform = new THVector3[3];

            playerTransform[0] = positon.position;
            playerTransform[1] = positon.rotation.eulerAngles;
            playerTransform[2] = positon.localScale;

            string playerPosSavePath = $"{savePath}/player.dat";

            SaveFile(playerTransform, playerPosSavePath);
        }
        /// <summary>
        /// Loads the players positon, roatation, and scale if it has previously
been saved
        /// </summary>
        /// <param name="playerTransfom">Transform to apply the data to</param>
        public static void LoadPlayerPosition(Transform playerTransfom)
        {
            string playerPosSavePath = $"{savePath}/player.dat";

            if (!File.Exists(playerPosSavePath))
                return;

            THVector3[] pos = (THVector3[])LoadFile(playerPosSavePath);

            playerTransfom.position = pos[0];
            playerTransfom.rotation = (Quaternion)pos[1];
            playerTransfom.localScale = pos[2];
        }

        /// <summary>
        /// Saves a given <see cref="Chunk"/> if a block in it has been changed
        /// </summary>
        /// <param name="chunk"></param>
        public static void SaveChunk(Chunk chunk)
        {
            //* saves the blocks
            SaveChunk save = new SaveChunk(chunk.blocks);

            //* if no block was changed return early
            if (save.blocks.Count == 0)
                return;

            //* otherwise save the file
            string saveFile = $"{savePath}/{FileName(chunk.chunkWorldPos)}.dat";

            SaveFile(save, saveFile);
        }

        /// <summary>
        /// Load a <see cref="Chunk"/>
        /// </summary>
        /// <param name="chunk"></param>
        /// <returns></returns>
        public static bool LoadChunk(Chunk chunk)
        {
            //* gets the save file
            string saveFile = $"{savePath}/{FileName(chunk.chunkWorldPos)}.dat";

            //* if the file does not exist return false
            if (!File.Exists(saveFile))
                return false;

            //* set all of the changed blocks in the chunk
            SaveChunk save = (SaveChunk)LoadFile(saveFile);
            foreach (var block in save.blocks)
            {
                chunk.blocks[block.Key.x, block.Key.y, block.Key.z] =
block.Value;
            }

            return true;
        }

        /// <summary>
        /// Sets the file name of the <see cref="Chunk"/>
        /// </summary>
        /// <param name="pos">Position of teh <see cref="Chunk"/></param>
        /// <returns>The string of pos</returns>
        public static string FileName(ChunkWorldPos pos)
        {
            return $"{pos.x}, {pos.y}, {pos.z}";
        }

        /// <summary>
        /// Saves the given data in the given file
        /// </summary>
        /// <param name="obj">Object to save</param>
        /// <param name="file">File path to save to</param>
        private static void SaveFile(object obj, string file)
        {
            BinaryFormatter bf = new BinaryFormatter();
            FileStream fs = new FileStream(file, FileMode.OpenOrCreate);

            try
            {
                bf.Serialize(fs, obj);
            }
            catch(SerializationException e)
            {
                Debug.Log($"Serialization Exception: {e}");
                throw new SerializationException();
            }
            finally
            {
                fs.Close();
            }
        }

        /// <summary>
        /// Loads the file at the given path
        /// </summary>
        /// <param name="file">File to load</param>
        /// <returns>returns the loaded file as an object</returns>
        private static object LoadFile(string file)
        {
            BinaryFormatter bf = new BinaryFormatter();
            FileStream fs = new FileStream(file, FileMode.Open);

            try
            {
                return bf.Deserialize(fs);
            }
            catch(SerializationException e)
            {
                Debug.Log($"Deserialization Exception {e}");
                throw new SerializationException();
            }
            finally
            {
                fs.Close();
            }
        }
    }
}
```

Code 12 – Serialization Class

To save the player, method shown in Code 12 – Serialization Class: SavePlayerPosition(), a unity function call the method in another class every few seconds as there is no real metric that can be used other than if the player is moving and that would cause lots of file writes every frame, adversely affecting performance. Therefore, the player position is just saved every few seconds. However, when the players inventory is implemented that can have a metric for saving which will be each time it is modified it will save, I also assume that all other inventories will work in this way.

```
namespace Terrain.Chunks
{
/// <summary>
/// Saves a <see cref="Chunk"/>s modified <see cref="Block"/>s for save
optimisation
/// </summary>
    [Serializable]
    public class SaveChunk
    {
/// <summary>
/// <see cref="Block"/>s to be saved
/// </summary>
        public Dictionary<ChunkWorldPos, Block> blocks = new
Dictionary<ChunkWorldPos, Block>();

/// <summary>
/// Will search all the the given <see cref="Block"/>s for modified blocks
/// </summary>
/// <param name="blockArray"><see cref="Chunk"/>s blocks (Must be [16, 16,
16])</param>
        public SaveChunk(Block[,,] blockArray)
        {
            for (int x = 0; x < Chunk.chunkSize; x++)
            {
                for (int y = 0; y < Chunk.chunkSize; y++)
                {
                    for (int z = 0; z < Chunk.chunkSize; z++)
                    {
// if the block has changed save it
                        if (blockArray[x, y, z].changed)
                            blocks.Add(new ChunkWorldPos(x, y, z), blockArray[x,
y, z]);
```

```
                    }
                }
            }
        }
    }
}
```

Code 13 – SaveChunk Class

To save and load blocks an intermediary data structure is used, this is the SaveChunk Class, shown in Code 13 – SaveChunk Class. This is used instead of saving the whole chunk, instead the chunk is given to the class by the SaveChunk method in the Serialization class, seen in Code 12 – Serialization Class: SaveChunk(). This is done by the SaveChunk class iterating through each block in the given chunk then adding it to a dictionary if it was placed by a player or other means, this was originally intended to be done in a separate thread, however the performance gain for the added complexity was not worth it, it also has the possibility to introduce some interesting behavior if the chunk was modified twice with the later alterations thread finishing before the earlier alteration. Once this is done then the SaveChunk class is saved ueing binary serializarion, method for this is can be seen in Code 12 – Serialization Class: SaveFile(). Each chunk has a save file with its position as the name of the file.

To load the changes to the chunk after it has been generated in the standard manner the LoadChunk method is called which will load the chunks save file if it exists then iterate through the dictionary and replace every block that has been modified to its modified form.

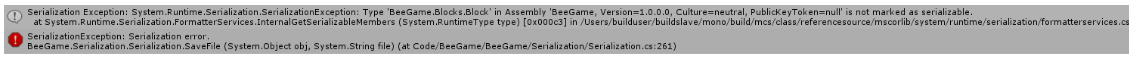
FEEDBACK FROM PROTOTYPE 1

After the Terrain Generation, Saving, Loading, and Block modification had been completed. I compiled a version to test. And compared the results against the table from earlier, the results are presented below.

Test Number	Description of Test	Input data	Expected result	Actual result	Type
1	Save	Block Edit	Data saved correctly	Data Did not save	Iterative
2	Loading	Starting the game	Data loaded correctly	Data did not load due to saving issues	Iterative
3	Forward Movement	Keypress	Character moves forward	Works as intended	Iterative
4	Backward moment	Keypress	Character moves backward	Works as intended	Iterative
5	Left Movement	Keypress	Character moves to the left	Works as intended	Iterative
6	Right Moment	Keypress	Character moves to the right	Works as intended	Iterative
7	Terrain Loads	Noise function	Terrain loads corrects with no seams	Works as intended	Iterative

8	Block destruction works/saves correctly	Keypress/File	Block is destroyed correctly without causing a crash and file is loaded correctly	Block is destroyed however not saved correctly	Iterative
9	Block placement works/saves correctly	Keypress/File	Block is placed correctly in the correct position without causing a crash and saved/loaded correctly when the game is restarted	Placement Works as intended however saving did not work	Iterative

All the failed tests were from very a very simple error which was that I forgot to add a "Serialzable" tag that C# requires for a class to be serialized to the block classes. The error message is shown in Figure 21 – Serialization Error Message, the first message being from my own error class then the unity engine recognizes the error and displays it below in red. This was a simple fix and once done a second compile of the code fixed all errors related to saving.



```
Serialization Exception: System.Runtime.Serialization.SerializationException: Type 'BeeGame.Blocks.Block' in Assembly 'BeeGame, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null' is not marked as serializable.
  at System.Runtime.Serialization.FormatterServices.InternalGetSerializableMembers (System.RuntimeType type) [0x000c3] in /Users/builduser/buildslave/mono/build/mcs/class/referencesource/mscorlib/system/runtime/serialization/formatterservices.cs
SerializationException: Serialization error.
BeeGame.Serialization.Serialization.SaveFile (System.Object obj, System.String file) (at Code/BeeGame/BeeGame/Serialization/Serialization.cs:261)
```

Figure 21 – Serialization Error Message

However, there was one other issue with the game. This, however is an issue that I cannot control as it is one due to the "Garbage Collector" (GC). This is something that automatically handles the deallocation of unused data for example when a chunk is deleted. An image of the problem is shown in Figure 22 – Unity Profiler.

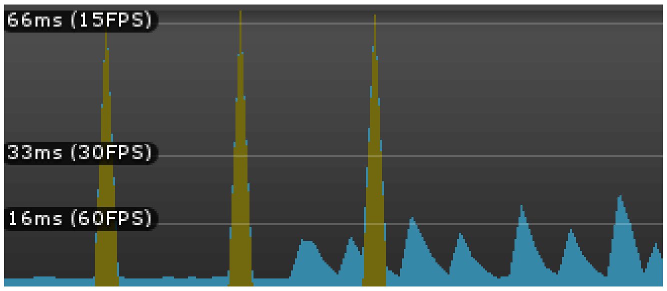


Figure 22 – Unity Profiler

In Figure 22 – Unity Profiler the Brown spike is the GC and the Blue spikes are my code. As can be seen MY code is perfect, however whenever a chunk is deleted, and the GC activates there is a large performance drop, this is very noticeable in the game, but I don't know how to fix it. At the time of creation Unity (from what I can research) is using a very old and inefficient GC that it currently in need of updating. There are some methods to optimize this however they will either not work in this case or are too complex for me to understand, meaning that this is something that I will have to live with unless Unity updates their GC, or I come up with a better way to optimize for it, until them, annoyingly, I will have to live with it.

ITEMS AND INVENTORY

Now that blocks and terrain can be generated properly, and item and inventory system can be implemented with faith that things won't change too much. It also makes sense to implement both the inventory and items at the same time as having one without the other wouldn't make much sense.

To begin with I will make a new base class for the Block that called Items as all blocks can be items but not all items can be blocks.

```
namespace Items
{
        /// <summary>
        /// Base class for all Items and Blocks in the game
        /// </summary>
    [Serializable]
    public class Item : AbstractItem, ICloneable
    {
        /// <summary>
        /// Name of the item
        /// </summary>
        internal string itemName { get; set;}
        /// <summary>
        /// Does the item use a gameobject
        /// </summary>
        public bool usesGameObject { get; set; }
        /// <summary>
        /// How big are the texture tiles in the texture map (1/tile number x)
        /// </summary>
        private const float tileSize = 0.1f;

        /// <summary>
        /// The unique ID of this item
        /// </summary>
        public static int ID => 0;

        public Item()
        {
            itemName = "TestItem";
        }

        public Item(string name)
        {
            itemName = name;
        }

        /// <summary>
        /// Returns the <see cref="GameObject"/> for the item of it has one
        /// </summary>
        /// <returns>GameObject for the item</returns>
        public virtual GameObject GetGameObject() { return null; }

        /// <summary>
        /// Returns the id for the item as a string
        /// </summary>
        /// <returns></returns>
        public override string GetItemID()
        {
            return $"{GetHashCode()}";
        }

        /// <summary>
        /// Returns the items name
        /// </summary>
        /// <returns></returns>
        public override string GetItemName()
        {
            return $"{itemName}";
        }

        /// <summary>
        /// Texture postion of the items texture
        /// </summary>
        /// <param name="direction">Direction for the texture</param>
        /// <returns>Position of the texture</returns>
        public virtual Tile TexturePosition(Direction direction)
        {
            return new Tile() { x = 1, y = 9 };
        }

        /// <summary>
        /// Returns the mesh for the item
        /// </summary>
        /// <param name="x">X pos if the item</param>
        /// <param name="y">Y pos if the item</param>
        /// <param name="z">Z pos if the item</param>
        /// <param name="meshData">data to add the mesh to</param>
        /// <returns>given <see cref="MeshData"/> with the items mesh
added</returns>
        public virtual MeshData ItemMesh(int x, int y, int z, MeshData meshData)
        {
// adds all faces of the item to the mesh as all faces could be seen at any time
            meshData = FaceDataUp(x, y, z, meshData, true, 0.25f);
            meshData = FaceDataDown(x, y, z, meshData, true, 0.25f);
            meshData = FaceDataNorth(x, y, z, meshData, true, 0.25f);
            meshData = FaceDataEast(x, y, z, meshData, true, 0.25f);
            meshData = FaceDataSouth(x, y, z, meshData, true, 0.25f);
            meshData = FaceDataWest(x, y, z, meshData, true, 0.25f);

            return meshData;
        }

        /// <summary>
        /// Sets the UVs for the given <see cref="Direction"/>
        /// </summary>
        /// <param name="direction">Direction to add the texture</param>
        /// <returns>Array of <see cref="Vector2"/> to add to the UVsreturns>
        public virtual Vector2[] FaceUVs(Direction direction)
        {
// only 4 uvs per face
            Vector2[] UVs = new Vector2[4];
            Tile tilePos = TexturePosition(direction);
// sets the UVs for each vertex
            UVs[0] = new THVector2(tileSize * tilePos.x + tileSize - 0.01f,
tileSize * tilePos.y + 0.01f);
            UVs[1] = new THVector2(tileSize * tilePos.x + tileSize - 0.01f,
tileSize * tilePos.y + tileSize - 0.01f);
            UVs[2] = new THVector2(tileSize * tilePos.x + 0.01f, tileSize *
tilePos.y + tileSize - 0.01f);
            UVs[3] = new THVector2(tileSize * tilePos.x + 0.01f, tileSize *
tilePos.y + 0.01f);

            return UVs;
        }

        /// <summary>
        /// Adds the Upwards face to the given <see cref="MeshData"/>
        /// </summary>
        /// <param name="x">X pos of the item</param>
        /// <param name="y">Y pos of the item</param>
        /// <param name="z">Z pos of the item</param>
        /// <param name="meshData"><see cref="MeshData"/> to add the face
to</param>
        /// <param name="addToRenderMesh">Should the mesh be added to the render
mesh (default true)</param>
        /// <param name="blockSize">how big is the item</param>
        /// <returns>Given <see cref="MeshData"/> with the face data
added</returns>
        protected virtual MeshData FaceDataUp(int x, int y, int z, MeshData
meshData, bool addToRenderMesh = true, float blockSize = 0.5f)
        {
// Adds vertices in a anti-clockwise order
            meshData.AddVertices(new THVector3(x - blockSize, y + blockSize, z +
blockSize), addToRenderMesh, Direction.UP);
            meshData.AddVertices(new THVector3(x + blockSize, y + blockSize, z +
blockSize), addToRenderMesh, Direction.UP);
            meshData.AddVertices(new THVector3(x + blockSize, y + blockSize, z -
blockSize), addToRenderMesh, Direction.UP);
            meshData.AddVertices(new THVector3(x - blockSize, y + blockSize, z -
blockSize), addToRenderMesh, Direction.UP);

// adds the tris for the quad
            meshData.AddQuadTriangles(addToRenderMesh);

// if the data should be added to the render mesh also add the uvs to the mesh
            if (addToRenderMesh)
                meshData.uv.AddRange(FaceUVs(Direction.UP));

            return meshData;
        }
Down, North, East, South, and West omitted as they are the same as this but in
different positions

        /// <summary>
        /// Returns a new copy of this item
        /// </summary>
        /// <returns>this as if it was a value type</returns>
        public object Clone()
        {
// Saves this to a file then reads it back so that a copy and not a reference is
passed
            BinaryFormatter bf = new BinaryFormatter();
            MemoryStream ms = new MemoryStream();

            bf.Serialize(ms, this);
            ms.Seek(0, SeekOrigin.Begin);

            return bf.Deserialize(ms);
        }

        /// <summary>
        /// Returns the item name an id formatted nicely
        /// </summary>
        /// <returns></returns>
        public override string ToString()
        {
            return $"{itemName} \nID: {GetItemID()}";
        }

        /// <summary>
        /// Returns the hashcode for the item
        /// </summary>
        /// <returns>1</returns>
        public override int GetHashCode()
        {
            return ID;
        }

        /// <summary>
        /// Checks if the item is equal to another
        /// </summary>
        /// <param name="obj">object to check against</param>
        /// <returns>true if items are the same</returns>
        public override bool Equals(object obj)
        {
            if (!(obj is Item))
                return false;

            return this == (obj as Item);
        }

        /// <summary>
        /// Overides the default == operator as different things need to be
checked
        /// </summary>
        /// <param name="a">Item</param>
        /// <param name="b">Item</param>
        /// <returns>true if <paramref name="a"/>  == <paramref
name="b"/></returns>
        public static bool operator ==(Item a, Item b)
        {
            if (ReferenceEquals(a, null) && ReferenceEquals(b, null))
                return true;
            if (ReferenceEquals(a, null) || ReferenceEquals(b, null))
                return false;

            if(a.GetItemID() == b.GetItemID())
                return true;

            return false;
        }

!= exists and just returns the inverse of ==
    }

    /// <summary>
    /// Position of the items texture
    /// </summary>
    [Serializable]
    public struct Tile
    {
        /// <summary>
        /// X pos of the texture
        /// </summary>
        public int x;
        /// <summary>
        /// Y pos of the texture
        /// </summary>
        public int y;
    }
}
```

Code 14 – Base Item Class

Shown in Code 14 – Base Item Class, is the base class for all blocks and items in the game. As this is the base class for all blocks some of the block methods have been moved to this class as they can be made use of by an item, the main methods that were moved are the one for generating a mesh, this is so that when a block is destroyed it can be rendered as a smaller version of the block on the ground, however an item can also use a mesh that is not a cube this is supported by the "usesGameObject" variable that can be seen, the GameObject is set by the GetGameObject method called by second class that handles the item when it is outside of a player inventory and not placed as a block (Code 15 – Item GameObject Interface Class), this class can be seen in and is discussed later as some changes were also made to the break block method to support this new functionality (shown here, Code 16 – Changes to the Break Block Methods).

The == operator and GetHasCode operator also needed to be overridden in this case as this is a class and by default that == only checks for the same reference however in this case 2 items can have different references but still be the same item therefore I use a custom hash code which is set manually as the unique ID for the item.

The Clone method is also overridden as using = on a class will copy a reference therefore their needs to be some way to do this by value so I implemented a method to do this.

```
namespace Items
{
/// <summary>
/// Interface between Item's and Unity GameObjects
/// </summary>
    [RequireComponent(typeof(Rigidbody))]
    [RequireComponent(typeof(MeshFilter))]
    [RequireComponent(typeof(MeshRenderer))]
    [RequireComponent(typeof(BoxCollider))]
    public class ItemGameObject : MonoBehaviour
    {
        /// <summary>
        /// Item that this gameobject represents
        /// </summary>
        public Item item;
        /// <summary>
        /// gameobject to make
        /// </summary>
        public GameObject go;

        /// <summary>
        /// Makes the mesh or instantiates the items gameobject
        /// </summary>
        private void Start()
        {
// of the item does not use a go then make a cube mesh
            if (!item.usesGameObject)
                MakeMesh();

            if (item.usesGameObject)
            {
                Instantiate(item.GetGameObject(), transform, false);
                transform.localScale = new Vector3(0.5f, 0.5f, 0.5f);
            }
        }

        /// <summary>
        /// Destroys the game object if it falls to low
        /// </summary>
        private void Update()
        {
// item can fall though the world, we don't want to happen so it is destroyed
after a while (consider making item load chunks around it)
// considered it
// and?
// too difficult
// ok
           if(transform.position.y < -100)
            {
                Destroy(gameObject);
            }
        }

        /// <summary>
        /// Makes the items mesh
        /// </summary>
        void MakeMesh()
        {
            MeshData meshData = new MeshData();
            if(item != null)
```

```
                meshData = item.ItemMesh(0, 0, 0, meshData);

            Mesh mesh = new Mesh()
            {
                vertices = meshData.verts.ToArray(),
                triangles = meshData.tris.ToArray(),
                uv = meshData.uv.ToArray()
            };

            mesh.RecalculateNormals();

            GetComponent<MeshFilter>().mesh = mesh;
        }
    }
}
```

Code 15 – Item GameObject Interface Class

In Code 15 – Item GameObject Interface Class, it is shown how I decided to make it easily make an item into an object, this is needed as the Item scripts are not able to be attached to a Unity GameObject as they do not derive from a MonoBehaviour class therefore an interface is needed. This also allows me to add some methods that would not be needed for an item in an inventory meaning the item class can be more focused.

```
void BreakBlock()
{
    Chunk chunk = GetChunk(selector.transform.position);

    Block block = chunk.world.GetBlock(selector.transform.position);

    if (!block.breakable)
        return;

    chunk.world.SetBlock(selector.transform.position, new Air(), true);
// set to changed so when block is placed down again it will be saved
    block.changed = true;
// tells the block the coordinates of where it was destroyed
    block.BreakBlock(selector.transform.position);
}
```

Code 16 – Changes to the Break Block Methods

```
public virtual void BreakBlock(THVector3 pos)
{
    GameObject go =
Object.Instantiate(UnityEngine.Resources.Load("Prefabs/ItemGameObject") as
GameObject, pos, Quaternion.identity) as GameObject;
    go.GetComponent<ItemGameObject>().item = this;
}
```

Code 17 – New BreakBlock method in Block class

The modification to the BreakBlock method in the Selector Class (Code 16 – Changes to the Break Block Methods), and the new BreakBlock method in the Block class (Code 17 – New BreakBlock method in Block class), basically just make it so that when the block is destroyed a GameObject is spawned at the position that the block is destroyed with the ItemGameObject class attached to it (Code 15 – Item GameObject Interface Class) so that a dropped item is rendered.

This new system was then tested, and the result is shown in Figure 23 – Dropped Items, with the smaller blocks being the dropped items.

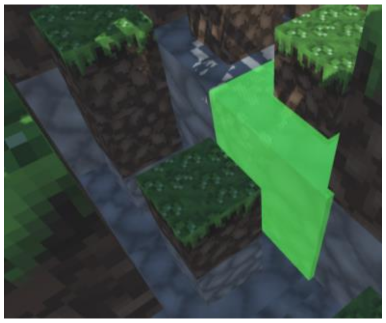


Figure 23 – Dropped Items

Now that this has been done the player needs some way to collect the items therefore the inventory needs to be implemented.

For this I decided to come up with a structure that consists of an extendable inventory controller that contains an array of item slots that shows each item in the inventory.

```
namespace Inventory
{
    /// <summary>
    /// Base class for all inventorys in the game
    /// </summary>
    public class Inventory : MonoBehaviour
    {
        /// <summary>
        /// Items in the inventory
        /// </summary>
        public ItemsInInventory items;
        /// <summary>
        /// Slots in the inventory
        /// </summary>
        public InventorySlot[] slots;
        /// <summary>
        /// <see cref="Item"/> that is currenty being moved
        /// </summary>
        internal Item floatingItem;
        /// <summary>
        /// Name of this inventory, used for serialization
        /// </summary>
        public string inventoryName = "";
        /// <summary>
        /// is this inventory open?
        /// </summary>
        protected bool thisInventoryOpen = false;

        /// <summary>
        /// Is the inventory set?
        /// </summary>
        /// <returns>true if <see cref="items"/> == null</returns>
        public bool InventorySet()
        {
            if (items == null)
                return true;

            return false;
        }

        /// <summary>
        /// Sets the inventory soze to the number of slots in the invnetory
        /// </summary>
        /// <param name="inventorySize"></param>
        public void SetInventorySize(int inventorySize)
        {
            items = new ItemsInInventory(slots.Length);
        }

        /// <summary>
        /// Things in the inventory that should be updated
        /// </summary>
        protected void UpdateBase()
        {
            PutItemsInSlots();
        }

        /// <summary>
        /// Sets an <see cref="Item"/> in the <see
cref="ItemsInInventory.itemsInInventory"/> array to a <see
cref="InventorySlot.item"/>
        /// </summary>
        public virtual void PutItemsInSlots()
        {
// goes through all of the items in the array setting then all to a slot
            for (int i = 0; i < slots.Length; i++)
            {
                slots[i].slotIndex = i;
                slots[i].myInventory = this;
                slots[i].item = items.itemsInInventory[i];
            }
        }

        /// <summary>
        /// Gets all of the items in the invntory
        /// </summary>
        /// <returns>All of the items in the inventory as <see
cref="ItemsInInventory"/></returns>
        public ItemsInInventory GetAllItems()
        {
            return items;
        }

        /// <summary>
        /// Adds the given <paramref name="item"/> to the inventory in the given
<paramref name="slotIndex"/>
```

```
        /// </summary>
        /// <param name="slotIndex">Slot to add item to</param>
        /// <param name="item">Item to add</param>
        public virtual void AddItemToSlots(int slotIndex, Item item)
        {
            items.AddItem(slotIndex, item);
        }

        /// <summary>
        /// Add an item to the inventory
        /// </summary>
        /// <param name="item">Item to add</param>
        /// <returns>true if item wasa added</returns>
        public bool AddItemToInventory(Item item)
        {
            return items.AddItem(item);
        }
    }
}
```

Code 18 – Base Inventory Class

In Code 18 – Base Inventory Class can be seen the base class for all inventory's in the game. It contains functionality that all inventories will need. The items in the inventory are stored in a separate class called "ItemsInInventory", this was done so that when inventories are saved the file size is kept to a minimum as only the necessary data is in the class, the code for this class can be seen in Code 19 – Items in Inventory Class.

Each time an inventory is opened the PutItemsInSlots method will be called, this places all of the items stored in the "items" variable into the item slots in the inventory.

```
namespace Inventory
{
    /// <summary>
    /// Class that holds all of the items in the inventory. Can be serialized so
inventory may be saved
    /// </summary>
    [Serializable]
    public class ItemsInInventory
    {
        /// <summary>
        /// All of the items in the inventory
        /// </summary>
        public Item[] itemsInInventory;

        /// <summary>
        /// Sets the size of the inventory
        /// </summary>
        /// <param name="numberOfInventorySlots"></param>
        public ItemsInInventory(int numberOfInventorySlots)
        {
            itemsInInventory = new Item[numberOfInventorySlots];
        }

        /// <summary>
        /// Add an <see cref="Item"/> to a specific index in the inventory
        /// </summary>
        /// <param name="index">Were to add the item</param>
        /// <param name="item">What <see cref="Item"/> to put in the
inventory</param>
        public void AddItem(int index, Item item)
        {
// possibly add error checking but it should be done somewhere else
            itemsInInventory[index] = item;
        }

        /// <summary>
        /// Adds a <see cref="Item"/> to the inventory
        /// </summary>
        /// <param name="item">Item to add</param>
        /// <returns>true if <paramref name="item"/> was added to the
inventory</returns>
        public bool AddItem(Item item)
        {
            for (int i = 0; i < itemsInInventory.Length; i++)
            {
                if (itemsInInventory[i] == null)
                {
                    itemsInInventory[i] = item;
                    return true;
                }
                if (itemsInInventory[i] == item &&
itemsInInventory[i].itemStackCount + 1 <= itemsInInventory[i].maxStackCount)
                {
                    itemsInInventory[i].itemStackCount++;
                    return true;
                }
            }

            return false;
        }
    }
}
```

Code 19 – Items in Inventory Class

```
namespace Inventory
{
    public class InventorySlot : MonoBehaviour, IPointerClickHandler,
IPointerEnterHandler, IPointerExitHandler
    {
        /// <summary>
        /// The slot in the inventory this is
        /// </summary>
        internal int slotIndex;
        /// <summary>
        /// The item this slot has in it
        /// </summary>
        public Item item;
        /// <summary>
        /// The <see cref="Inventory"/> this slot is in
        /// </summary>
        public Inventory myInventory;
        /// <summary>
        /// Updates the slot
        /// </summary>
        protected void Update()
        {
            UpdateIcon();
        }

        /// <summary>
        /// Applies the correct icon to the slot depending on what is in the
slot
        /// </summary>
        internal void UpdateIcon()
        {
            if(item == null)
            {
                GetComponent<Image>().sprite = null;
            }
            else
            {
                if(!item.Equals(new Item()))
                    GetComponent<Image>().sprite = item.GetItemSprite();
            }
        }
    }
}
```

Code 20 – Inventory Slot Class

In Code 20 – Inventory Slot Class can be seen the class the user will directly interface with when accessing an inventory. This will be, as shown in the design section a grid of squares and each inventory slot class is a cell in that grid. The user will also need some way to know what item is in which slot so if there is an item in the slot it gets its sprite. This obviously needed to be supported by functions in the item class and the new method is shown in Code 21 – New Sprite Method, this is a virtual method meaning that each new item will override this method so that it will point to the correct sprite.

```
public virtual Sprite GetItemSprite()
{
    return SpriteDictionary.GetSprite("TestSprite");
}
```

Code 21 – New Sprite Method

All the sprites in the game are stored in a static dictionary that is comprised of the sprites name as the key and the value being the image/sprite itself. This dictionary is constructed when the game is loaded and the code for the dictionary is shown in , with the code to load the sprites shown in .

```
namespace Core.Dictionaries
{
    /// <summary>
    /// All of the sprites available to the game
    /// </summary>
    public static class SpriteDictionary
    {
        /// <summary>
        /// All of the sprites available to spawn in
        /// </summary>
        private static Dictionary<string, Sprite> itemSpriteDictionary = new
Dictionary<string, Sprite>();

        /// <summary>
        /// Get a sprite of the given name
        /// </summary>
        /// <param name="spriteName">Name of sprite to get</param>
        /// <returns>A sprite of the given name, null if no sprite of that name
exists</returns>
        public static Sprite GetSprite(string spriteName)
        {
            itemSpriteDictionary.TryGetValue(spriteName, out Sprite sprite);

            if (sprite == null)
                return new Sprite();

            return sprite;
        }

        /// <summary>
        /// Loads the sprites into the dictionary
        /// </summary>
        public static void LoadSprites()
        {
            itemSpriteDictionary = Resources.Resources.GetSprites();
        }
    }
}
```

Code 22 – Sprite Dictionary

```
//*-----------------------------------------------------------------------------
-
//* <auto-generated>
//*     This code was generated by a tool.
//*     Runtime Version:4.0.30319.42000
//*
//*     Changes to this file may cause incorrect behavior and will be lost if
//*     the code is regenerated.
//* </auto-generated>
//*-----------------------------------------------------------------------------
-

namespace Resources {
    /// <summary>
    ///   A strongly-typed resource class, for looking up localized strings,
etc.
    /// </summary>
    //* This class was auto-generated by the StronglyTypedResourceBuilder
    //* class via a tool like ResGen or Visual Studio.
    //* To add or remove a member, edit your .ResX file then rerun ResGen
    //* with the /str option, or rebuild your VS project.
```

```
[global::System.CodeDom.Compiler.GeneratedCodeAttribute("System.Resources.Tools.
StronglyTypedResourceBuilder", "4.0.0.0")]
    [global::System.Diagnostics.DebuggerNonUserCodeAttribute()]
    [global::System.Runtime.CompilerServices.CompilerGeneratedAttribute()]
    internal class Resources {

        private static global::System.Resources.ResourceManager resourceMan;

        private static global::System.Globalization.CultureInfo resourceCulture;


[global::System.Diagnostics.CodeAnalysis.SuppressMessageAttribute("Microsoft.Per
formance", "CA1811:AvoidUncalledPrivateCode")]
        internal Resources() {
        }

        /// <summary>
        ///   Returns the cached ResourceManager instance used by this class.
        /// </summary>

[global::System.ComponentModel.EditorBrowsableAttribute(global::System.Component
Model.EditorBrowsableState.Advanced)]
        internal static global::System.Resources.ResourceManager ResourceManager
{
            get {
                if (object.ReferenceEquals(resourceMan, null)) {
                    global::System.Resources.ResourceManager temp = new
global::System.Resources.ResourceManager("Resources.Resources",
typeof(Resources).Assembly);
                    resourceMan = temp;
                }
                return resourceMan;
            }
        }

        /// <summary>
        ///   Overrides the current thread's CurrentUICulture property for all
        ///   resource lookups using this strongly typed resource class.
        /// </summary>

[global::System.ComponentModel.EditorBrowsableAttribute(global::System.Component
Model.EditorBrowsableState.Advanced)]
        internal static global::System.Globalization.CultureInfo Culture {
            get {
                return resourceCulture;
            }
            set {
                resourceCulture = value;
            }
        }

        /// <summary>
        ///   Looks up a localized resource of type System.Byte[].
        /// </summary>
        internal static byte[] Sprites {
            get {
                object obj = ResourceManager.GetObject("Sprites",
resourceCulture);
                return ((byte[])(obj));
            }
        }
My Code Below auto generated code above included for completeness
        /// <summary>
        /// Gets the sprites from Sprites.dat file and loads them
        /// </summary>
        /// <returns>A dictionary of the <see cref="Sprite"/> with its reference
name (dictionary key)</returns>
        internal static Dictionary<string, Sprite> GetSprites()
        {
            string[] splitCharacters = new string[] { "," };
            object obj = ResourceManager.GetObject("Sprites", resourceCulture);

// gets the text from the spries.dat file
            string text = System.Text.Encoding.Default.GetString((byte[])obj);
            string lineText = "";
            string[] splitText;
            Texture2D tex;
            Dictionary<string, Sprite> sprites = new Dictionary<string,
Sprite>();

// goes through all characters in the file
            for (int i = 0; i < text.Length; i++)
            {
// when their is a new line the path for that sprite is found
                if (text[i] != '\n')
                {
                    lineText += text[i];
                }
                else
                {
                    splitText = lineText.Split(splitCharacters,
StringSplitOptions.RemoveEmptyEntries);
                    lineText = "";
                    tex = UnityEngine.Resources.Load("Sprites/" +
splitText[1].Remove(splitText[1].Length - 1, 1)) as Texture2D;
// to create a sprite from a texture 2D because Unity wont allow images to be
loaded directly as sprites at runtime...for some reason
                    sprites.Add(splitText[0], Sprite.Create(tex, new
UnityEngine.Rect(0, 0, tex.width, tex.height), Vector2.zero));
                }
            }

            splitText = lineText.Split(splitCharacters,
StringSplitOptions.RemoveEmptyEntries);
            lineText = "";
            tex = UnityEngine.Resources.Load("Sprites/" + splitText[1]) as
Texture2D;
            sprites.Add(splitText[0], Sprite.Create(tex, new UnityEngine.Rect(0,
0, tex.width, tex.height), Vector2.zero));

            return sprites;
        }
    }
}
```

Code 23 – Load Resources Class

The sprite resources are loaded at runtime form the sprites folder contained within the games files, code to load the sprites shown in Code 23 – Load Resources Class. The found images are loaded into the game, the name of the image is taken to be the sprite name and then the loaded image is converted to a sprite that unity can use. Things are done this way so that the images can be changes outside of the game without having to recompile the whole game and it also means that I don't have to manually populate the sprite dictionary.

Once this was done I took pictures of each block and cropped then to all be in the same aspect ratio, so they didn't look weird next to each other and the result is seen in Figure 24 – Item Icons in Inventory.

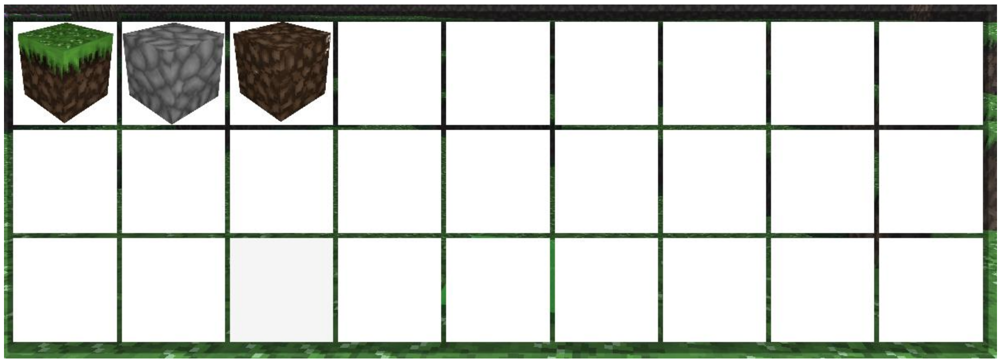


Figure 24 – Item Icons in Inventory

Now that the inventory items can be displayed they need to be able to be moved around by the and the inventory also needs to be opened and closed.

To address the second issue, I added a new class called player inventory the is attached to the player and contains their inventory. This class can be seen in .

```
namespace Inventory.Player_Inventory
{
    /// <summary>
    /// Controls the player inventory
    /// </summary>
    public class PlayerInventory : Inventory
    {
        /// <summary>
        /// Object that the inventory is
        /// </summary>
        public GameObject playerInventory;

        /// <summary>
        /// Sets all required params for the inventory and loads ant saved
versions of it
        /// </summary>
        protected void Awake()
        {
            SetPlayerInventory();
            inventoryName = "PlayerInventory";
        }

        /// <summary>
        /// Set the size of the player inventory
        /// </summary>
        public virtual void SetPlayerInventory(int size = 36)
        {
            if (!InventorySet())
                SetInventorySize(size);
        }

        /// <summary>
        /// Gives the inventory update ticks
        /// </summary>
        protected void Update()
        {
            UpdateBase();

// checks if the inventory should be opened/closed
            if ((thisInventoryOpen || !playerInventory.activeInHierarchy)
&& !THInput.chestOpen && THInput.GetButtonDown("Player Inventory"))
            {
                if (THInput.blockInventoryJustClosed)
                {
                    THInput.blockInventoryJustClosed = false;
                    return;
                }
                else
                {
                    OpenPlayerInventory();
                }
            }

// don't pickup items if the inventory is open
            if (THInput.isAnotherInventoryOpen)
                return;

// checks if somethig should be picked up and put into the inventory
            RaycastHit[] hit = Physics.SphereCastAll(transform.position, 1f,
transform.forward);

            for (int i = hit.Length - 1; i >= 0; i--)
            {
                if (hit[i].collider.GetComponent<ItemGameObject>())
                    PickupItem(hit[i].collider.GetComponent<ItemGameObject>());
            }

        }

        /// <summary>
        /// Updates the currently selected hot bar slot
        /// </summary>
        /// <param name="index">Slot that is selected</param>
        public void SelectedSlot(int index)
        {
            for (int i = 0; i < slots.Length; i++)
            {
                slots[i].selectedSlot = false;
            }

            slots[index].selectedSlot = true;
        }

        /// <summary>
        /// Gets an item from the hotbar (9 <see cref="InventorySlot"/>s at the
bottom of the screen)
        /// </summary>
        /// <param name="slotIndex">Index to get <see cref="Item"/> from</param>
        /// <param name="outItem"><see cref="Item"/> in the slot</param>
        /// <returns>true if <paramref name="outItem"/> is placeable, false if
<paramref name="outItem"/> is null or not placeable</returns>
        public bool GetItemFromHotBar(int slotIndex, out Item outItem)
        {
// get the item
            outItem = GetAllItems().itemsInInventory[slotIndex];

            if (outItem == null)
                return false;

// if the item is placeabale and is not null remove 1 from the inventory as it
is assumed it is about to be placed in the world
            if(outItem.placeable)
                RemoveItemFromInventory(slotIndex);

            return outItem.placeable;
        }

        /// <summary>
        /// Show/Hide the player inventory
        /// </summary>
        internal void OpenPlayerInventory()
        {
            if (floatingItem != null)
                return;
            thisInventoryOpen = !thisInventoryOpen;
            playerInventory.SetActive(!playerInventory.activeInHierarchy);
            THInput.isAnotherInventoryOpen = !THInput.isAnotherInventoryOpen;

// hides/shows the mouse depending on if the inventory is open or not
            if (playerInventory.activeInHierarchy)
            {
                Cursor.lockState = CursorLockMode.None;
                Cursor.visible = true;
            }
            else
            {
                Cursor.visible = false;
                Cursor.lockState = CursorLockMode.Locked;
            }
        }

        /// <summary>
        /// Removes 1 item from the given inventory index
        /// </summary>
        /// <param name="index"></param>
        public void RemoveItemFromInventory(int index)
        {
// if the item is already null nothing needs to be removed
            if (GetAllItems().itemsInInventory[index] != null)
            {
// remove 1 item and if that was the last in the stack remove the item from the
inventory
                GetAllItems().itemsInInventory[index].itemStackCount -= 1;

                if (GetAllItems().itemsInInventory[index].itemStackCount <= 0)
                    GetAllItems().itemsInInventory[index] = null;
            }
        }

        /// <summary>
        /// Pickup an item and put it into the <see cref="Inventory"/>
        /// </summary>
        /// <param name="item">Item to try to put into the inventory</param>
        void PickupItem(ItemGameObject item)
        {
            item.item.itemStackCount = 1;

// if the item can be added to the inventory do that
            if (AddItemToInventory(item.item))
            {
// if the item was added destroy its gameobject
                Destroy(item.gameObject);
            }
        }
    }
}
```

Code 24 – Player Inventory Class

To open and close the inventory the class checks for input every frame (Code 24 – Player Inventory Class: Update), and if an input is detected it is also checking if another inventory is open as 2 cannot be open at the same time. If the correct button is pressed the inventory object is shown giving the player access to the inventory.

The player also has a “hotbar”, this is will be used for the player to select the block they would like to place with the currently selected item being accessible by the GetItemFromHotBar() method, this will return if the current item can be placed and the item the player has selected. The item is returned as an out variable as when the code was written tuples were not easy to use in C# with Unity.

Using this class the player will also collect the items that are produced when a block is broken, providing that there is space in the inventory, using the PickupItem() method called form the Update() method.

The other methods in this class are support methods for the functions that the player inventory should have.

To allow the player to move items around in the inventory was a task that I found slightly more tedious. This is because I wanted to have a functionality were if the player right clicks on a slot it will give ½ if the stack to the player and the other ½ will stay in the slot, also if the player left clicks the whole stack would be given to the player, and a final check would needs to be done checking that the held item is the same as the item being removed or added to the held item and the handling of that, the solution to this is shown below in Code 25 – Additions to ItemSlot Class with this code being added to the Inventory Slot class (first shown here: Code 20 – Inventory Slot Class). There are most likely more efficient ways to do what is shown but I couldn't think of any "floatingItem" is the item currently held by the player. Also, when the player hovers over a slot the name is of item and stack count is displayed.

Whilst adding this functionality I also added the ability to save the inventory, this new code Code 26 – Inventory Saving was added to the Serialization class (first shown here: Code 12 – Serialization Class).

```
        /// <summary>
        /// Allows the player to interact with the item slot
        /// </summary>
        /// <param name="eventData">Right or Left click</param>
        /// <remarks>
        /// Called by the unity event handler when the slot is clicked on
        /// </remarks>
        public virtual void OnPointerClick(PointerEventData eventData)
        {
            if (myInventory.floatingItem != null)
            {
// Left click moves whole stacks of items
                if (eventData.button == PointerEventData.InputButton.Left)
                {
// If the item in the slot is empty put the floating item into it then clear it
and the slot can have items inserted
                    if (item == null && itemsCanBeInserted)
                    {
                        item = myInventory.floatingItem;
                        myInventory.floatingItem = null;
                        myInventory.AddItemToSlots(slotIndex, item);
                        return;
                    }
// if the items are the same
                    if(myInventory.floatingItem == item && itemsCanBeInserted)
                    {
// if the item in the inventory stack count + the floating items stack count is
less than the max stack count
                        if (myInventory.floatingItem.itemStackCount +
item.itemStackCount <= item.maxStackCount)
                        {
                            AddToSlot(myInventory.floatingItem.itemStackCount);
                            return;
                        }
// if the item stack added is larger than the max count add as many as you can
and move on
```

```
                        else
                        {
                            AddToSlot(item.maxStackCount - item.itemStackCount);
                            return;
                        }
                    }
// if the items are the same but items cannot be inserted into the slot add as
many items as you
// can from the slot to the floating item
                    else if(myInventory.floatingItem == item
&& !itemsCanBeInserted)
                    {
                        AddToFloatingItem();
                        return;
                    }
// If the items were not == swap them
                    else
                    {
// only if items can be inserted into the slot
                        if(itemsCanBeInserted)
                            SwapItems();
                        return;
                    }
                }
                else if(eventData.button == PointerEventData.InputButton.Right)
                {
// if the item in slot is null add 1 from the floating item to it
                    if(item == null && itemsCanBeInserted)
                    {
                        AddToSlot(1);
                        return;
                    }
// if the items are the same add 1 from the floating item to this item
                    else if(item == myInventory.floatingItem &&
itemsCanBeInserted)
                    {
                        AddToSlot(1);
                        return;
                    }
                }
            }
// if the floating item is null
            else
            {
// add 1/2 of the stack into the floating item if right click was pressed
                if(eventData.button == PointerEventData.InputButton.Right)
                {
                    SplitStack();
                    return;
                }

                if (item == null)
                    return;

// otherwise add the items into the floating item slot
                myInventory.floatingItem = item.Clone();
                item.itemStackCount -= item.itemStackCount;

                return;
            }

        }

        /// <summary>
        /// Add items from the slot to the <see cref="Inventory.floatingItem"/>
        /// </summary>
        void AddToFloatingItem()
        {
// if the whole stack can be added do it and move on
            if(myInventory.floatingItem.itemStackCount + item.itemStackCount <=
item.maxStackCount)
            {
                myInventory.floatingItem.itemStackCount += item.itemStackCount;

                item = null;

                myInventory.AddItemToSlots(slotIndex, item);

                return;
            }

// if the whole stack cannot be added calculate how many need to be removed from
the slots item stack
            item.itemStackCount -= (item.maxStackCount -
myInventory.floatingItem.itemStackCount);
// set the floating item to the max stack count
            myInventory.floatingItem.itemStackCount = item.maxStackCount;

            myInventory.AddItemToSlots(slotIndex, item);
        }

        /// <summary>
        /// Adds a number to items into the slot
        /// </summary>
        /// <param name="numerToAdd">Number of items to add to the slot</param>
        void AddToSlot(int numerToAdd)
        {
// if the item in the slot is null create it
            if (item == null)
            {
                item = myInventory.floatingItem.Clone();
                item.itemStackCount = 0;
            }

// add to number to add to the stack count
            item.itemStackCount += numerToAdd;

// if the stack count is now larger than it should be dont let it be
            if (item.itemStackCount > item.maxStackCount)
                item.itemStackCount = item.maxStackCount;
```

```
// remove the number of items form the floating item then check the floating
item is not null
            myInventory.floatingItem.itemStackCount -= numerToAdd;
            CheckFloatingItem();
            //* save the inventory changes
            myInventory.AddItemToSlots(slotIndex, item);
        }

        /// <summary>
        /// Halfs a <see cref="Item.itemStackCount"/> between the slot and the
<see cref="Inventory.floatingItem"/>
        /// </summary>
        /// <remarks>
        /// If the stack count is the slot is not an even number more items go
to the floating item than go to the slot. This is so that right clicking on a
slot when their is only 1 item in it actually make the item in that slot go into
the floating item
        /// </remarks>
        void SplitStack()
        {
            myInventory.floatingItem = item.Clone();
            int give = (item.itemStackCount + 1) / 2;
            myInventory.floatingItem.itemStackCount = give;
            item.itemStackCount -= give;

            if (item.itemStackCount <= 0)
                item = null;

            myInventory.AddItemToSlots(slotIndex, item);
            Destroy(itemText);
        }

        /// <summary>
        /// Swaps the <see cref="Item"/> in the <see
cref="Inventory.floatingItem"/> with the slots <see cref="item"/>
        /// </summary>
        void SwapItems()
        {
// temp copy of the item
            Item temp = myInventory.floatingItem;
// sets the floating item
            myInventory.floatingItem = item;
// sets the item that was in the floating item to the item in the the slot
            item = temp;
// Saves the changes to the inventory
            myInventory.AddItemToSlots(slotIndex, item);
// destroys the text as it is not needed anymore
            Destroy(itemText);
        }

        /// <summary>
        /// Checks if the <see cref="Inventory.floatingItem"/> should be null
        /// </summary>
        void CheckFloatingItem()
        {
            if(myInventory.floatingItem.itemStackCount <= 0)
            {
                myInventory.floatingItem = null;
            }
        }

        /// <summary>
        /// checks that the item is valid
        /// </summary>
        internal void CheckItem()
        {
            if (item != null && myInventory != null)
            {
                if (item.itemStackCount == 0 || item.itemName == "TestItem")
                {
                    myInventory.items.itemsInInventory[slotIndex] = null;
                    Destroy(itemText);
                }
            }
        }
```

Code 25 – Additions to ItemSlot Class

```
/// <summary>
/// Serializes a given <see cref="Inventory"/>
/// </summary>
/// <param name="inventory">Invenotry to Serialize</param>
/// <param name="inventoryName">Name of the inventory</param>
public static void SerializeInventory(Inventory.Inventory inventory, string
inventoryName)
{
    string inventorySavePath = $"{savePath}/Inventorys";

    if (!Directory.Exists(inventorySavePath))
        Directory.CreateDirectory(inventorySavePath);

    SaveFile(inventory.GetAllItems(),
$"{inventorySavePath}/{inventoryName}.dat");
}

/// <summary>
/// Deserializesd an <see cref="Inventory"/> from its name into a given
<paramref name="inventory"/>
/// </summary>
/// <param name="inventory">Inventory to apply the data to</param>
/// <param name="inventoryName">Inventory to deserialize</param>
public static void DeSerializeInventory(Inventory.Inventory inventory, string
inventoryName)
{
// make the path
    string inventorySavePath = $"{savePath}/Inventorys/{inventoryName}.dat";

// checks that the file exists
    if (!File.Exists(inventorySavePath))
    {
        for (int i = 0; i < inventory.items.itemsInInventory.Length; i++)
        {
            inventory.items.itemsInInventory[i] = null;
        }

        SerializeInventory(inventory, inventoryName);

        return;
    }
    inventory.SetAllItems((ItemsInInventory)LoadFile($"{inventorySavePath}"));
}
```

Code 26 – Inventory Saving

Saving the inventory was added to every place were the inventory can be edited. Which include but is not limited to, item pickup, moving the items between slots, opening and closing the inventory, and reloading the inventory.

The final thing that needs to be done in this section before a second prototype is placement of the selected block. This will not be too difficult to implement as I have already added methods to access the currently selected item an all that is needed is to attach to this in the PlaceBlock() method in the selector class (Code 9 – Selector Class), the changes to the PlaceBlock method are shown in .

```
void PlaceBlock()
{
    Chunk chunk = GetChunk(selector.transform.position);

    if (chunk == null)
        return;

// gets the currently selected item
transform.parent.GetComponentInChildren<PlayerInventory>().GetItemFromHotBar(sel
ectedHotbarSlot, out var item);

    if (item != null)
    {
// if the item can be placed place it
        if (item.placeable)
        {
            chunk.world.SetBlock(selector.transform.position + hit.normal,
(Block)item.CloneObject(), true);
        }
    }
}
```

Code 27 – Place Block Additions

FEEDBACK FROM PROTOTYPE 2

Now that player inventory and blocks breaking/placing have been implemented it is time for another test.

Test Number	Description of test	Input data	Expected result	Actual result	Type

1	Save	Keypress	Data saved correctly	Now works correctly	Iterative
2	Loading	Starting the game	Data loaded correctly	Works	Iterative
3	Forward Movement	Keypress	Character moves forward	Works	Iterative
4	Backward moment	Keypress	Character moves backward	Works	Iterative
5	Left Movement	Keypress	Character moves to the left	Works	Iterative
6	Right Moment	Keypress	Character moves to the right	Works	Iterative
7	Terrain Loads	Noise function	Terrain loads corrects with no seams	Works	Iterative
8	Block destruction works/saves correctly	Keypress/File	Block is destroyed correctly without causing a crash and file is loaded correctly	Works	Iterative
9	Block placement works/saves correctly	Keypress/File	Block is placed correctly in the correct position without causing a crash and saved/loaded correctly when the game is restarted	Works	Iterative
10	Inventories save stored items	Items inside an inventory	Items are in the same position, of the same type, and	Works however slow	Iterative

			number as then input into the inventory		
11	Items can be input into an inventory	Items moved into an inventory	Item can be added to an inventory without duplication and without being able to override other items	Works however needs work as discussed below	Iterative

The two main issues that arose from the testing was that moving items around the inventory was slow and that it can be difficult to know if an item is currently being moved or not.

For the second problem the solution was simple to draw the item currently being moved at the cursors position, the code to do this can be seen in Code 28 – Code to Draw Item at Cursor and was added to the Inventory Class

```
/// <summary>
/// Draws the <see cref="floatingItem"/>s <see cref="Item.GetItemSprite()"/> at
the mouse position
/// </summary>
private void DrawItemAtCursor()
{
    if(floatingItem != null)
    {
        if (spriteAtCursor == null)
        {
            spriteAtCursor =
Instantiate(PrefabDictionary.GetPrefab("ItemIcon"));
            spriteAtCursor.GetComponentInChildren<UnityEngine.UI.Image>().sprite
= floatingItem.GetItemSprite();
        }
// will update the sprite of an item is swapped between a slot and teh floating
item if the previous item wasnt put into a slot first
        else if(spriteAtCursor != null)
        {
            spriteAtCursor.GetComponentInChildren<UnityEngine.UI.Image>().sprite
= floatingItem.GetItemSprite();
        }

        spriteAtCursor.transform.GetChild(0).position = Input.mousePosition;
    }
    else
    {
        Destroy(spriteAtCursor);
    }
}
```

Code 28 – Code to Draw Item at Cursor

The results of this change are shown in Figure 25 – Moving Item Drawn at Cursor.

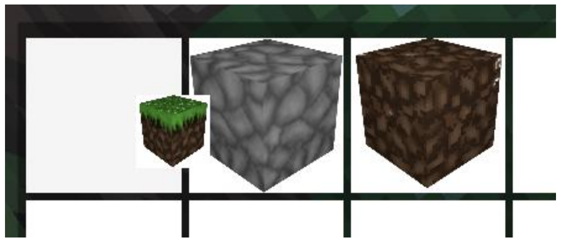


Figure 25 – Moving Item Drawn at Cursor

The second problem was slightly more difficult to solve. The issue was that the current method I use to get around the items being passed by reference was far too slow, as it involves making a copy of the item, serializing that copy, reloading that copy, and deleting the file. This is far too slow, and I needed to come up with a better solution. However, as this is a problem that could come up allow and it needed to work with all items without me having to manually add something to make it work this was no easy thing.

After some time googling I eventually found that in C# you can use reflection to do what I want. the code is shown in Code 29 – New Method to Speed Things Up.

```
/// <summary>
/// Allows the copying of a class by value useing reflection
/// </summary>
/// <param name="obj">Object to copy</param>
/// <returns>a new object with all values copyed</returns>
/// <remarks>
/// Mush faster than the serialization method however a lot more complicated
/// </remarks>
public static T CloneObject<T>(this T obj)
{
// gets the type of the given object
    Type typeSource = obj.GetType();

// makes a new object of type T
    T objTarget = (T)Activator.CreateInstance(typeSource);

// gets the properties in T
    PropertyInfo[] propertyInfo = typeSource.GetProperties(BindingFlags.Public |
BindingFlags.NonPublic | BindingFlags.Instance);

// applies the properties in T to the new type T object
    foreach (var property in propertyInfo)
    {
        if (property.CanWrite)
        {
// if the property is a value type just set it
            if (property.PropertyType.IsValueType ||
property.PropertyType.IsEnum || property.PropertyType.Equals(typeof(string)))
            {
                property.SetValue(objTarget, property.GetValue(obj, null),
null);
            }
            else
            {
// if the property is not a value type this function will need to be called
recursively as it could also have non-value type variables
                object propertyValue = property.GetValue(obj, null);

                if (propertyValue == null)
                {
                    property.SetValue(objTarget, null, null);
                }
                else
                {
                    property.SetValue(objTarget, propertyValue.CloneObject(),
null);
                }
            }
        }
    }

// gets all the fields in T
    FieldInfo[] fieldInfo = typeSource.GetFields();

// applies all the fields of T to the new object of type T in the same manor
that the properties are applied
    foreach (var field in fieldInfo)
    {
        if(field.FieldType.IsValueType || field.FieldType.IsEnum ||
field.FieldType.Equals(typeof(string)))
        {
            field.SetValue(objTarget, field.GetValue(obj));
        }
        else
        {
            object fieldValue = field.GetValue(obj);

            if(fieldValue == null)
            {
                field.SetValue(objTarget, null);
            }
            else
            {
                field.SetValue(objTarget, field.CloneObject());
            }
        }
    }
    return objTarget;
}
```

Code 29 – New Method to Speed Things Up

I made this a generic method so that it can be accessed by anything in the game because I feel like I may need it in the future.

The T in the <> brackets at the start of the function is a placeholder for a type, this is to avoid making the type into an object as this would mean that a lot of casting could happen possibly effecting performance.

Firstly, what happens is a new item of the given type (T) is created. Then all the properties are copied over to the new object. If the property is a value type (it is passed by value and not reference) then it is just immediately copied over as there is no worry other things can affect it. However, if the type is a reference type (if it passed by reference) then a new object of that type must be created so that is disconnected from the original value, this is done by recursively calling the function on that type.

Once this is done the fields of the passed object are copied over in the same manner as the fields and once that is done the copied object is returned. This returned object is a completely new object disconnected from the original given object, however, with all the same values.

All places that used the old method were then changed to use this new one. Using the new system, the performance was increased by orders of magnitude. After this the game was recompiled and the slowness was gone. Also drawing the item currently being moved at the cursor position was liked by the testers.

BLOCK INVENTORYS, CRAFTING, AND QUESTS

The next thing after a player inventory is some place to store items in the world and have something to do with the items, be it crafting or usable items.

For as it most logical to start with first will come the block inventories. Most of the hard work is already complete as this is handled in the base inventory class therefore all I need to do is make a child class that can fit into a block. This will need 3 things:

- Display the player inventory
- Allow the player to move items between the player and block inventory
- Save the changes to the player and block inventory between sessions

Firstly, I made the method to show the player items in the chest inventory as shown in Code 30 – Block Inventory. Also allow the player to open and close the inventory.

```
namespace Inventory
{
    /// <summary>
    /// Inventory for the chests
    /// </summary>
    public class ChestInventory : Inventory
    {
        /// <summary>
        /// Refernce to the players <see cref="Inventory"/> so that it can be
updated when chest is closed
        /// </summary>
        public Inventory playerinventory;
        /// <summary>
        /// The inventory gameobject that will be displayed
        /// </summary>
        public GameObject inventory;
        /// <summary>
        /// How many slots are in this <see cref="Inventory"/>
        /// </summary>
        public int inventorySize;

        /// <summary>
        /// Updates the slots and checks if the inventory should be closed
        /// </summary>
        void Update()
        {
            UpdateChestInventory();
        }

        /// <summary>
        /// The unity Update method is not called if the class is a
parent...annoyingly so need to call the base update method from the child
        /// </summary>
        public void UpdateChestInventory()
        {
// the chest should always have a player inventory when it does this but checks
just in case
            if (playerinventory != null)
                UpdateBase();

// checks if the inventory should be closed, but only if nothing is being moved
and this inv open
            if (GetButtonDown("Player Inventory") && thisInventoryOpen &&
floatingItem == null)
                ToggleInventory(playerinventory);
        }

        /// <summary>
        /// Sets the Size and name of this <see cref="Inventory"/>
        /// </summary>
        public virtual void SetChestInventory(string invName = "Chest")
        {
            SetInventorySize(inventorySize);
// sets the UI to not be seen as inventorys cannot start open
            inventory.SetActive(false);

// sets the name and postion if this inventory used during serialization and
deserialization
            inventoryName = $"{invName} @ {(ChunkWorldPos)inventoryPosition}";
        }

        /// <summary>
        /// Puts the player items into the chest
        /// </summary>
        void SetPlayerItems()
        {
            for (int i = 0; i < playerinventory.items.itemsInInventory.Length;
i++)
            {
                items.itemsInInventory[i + (inventorySize - 36)] =
playerinventory.items.itemsInInventory[i];
```

```
            }
        }

        /// <summary>
        /// Opens and closes the inventory
        /// </summary>
        /// <param name="inv"></param>
        public override void ToggleInventory(Inventory inv)
        {
// sets the player inventory
            playerinventory = inv;

            thisInventoryOpen = !thisInventoryOpen;

            isAnotherInventoryOpen = thisInventoryOpen;

            inventory.SetActive(!inventory.activeInHierarchy);

            if (inventory.activeInHierarchy)
            {
                chestOpen = true;

// puts the items into the chest
                SetPlayerItems();
// shows and unlocks the cursor
                Cursor.lockState = CursorLockMode.None;
                Cursor.visible = true;
            }
            else
            {
                chestOpen = false;

// hides and locks the cursor
                Cursor.lockState = CursorLockMode.Locked;
                Cursor.visible = false;
            }
        }
    }
}
```

Code 30 – Block Inventory

This “should” open and close the block inventory, however it won’t because at this point it is not connected to the actual placed block.

However, this will work be being given the player inventory from the selector class as it will need to be modified so that it checks for interactable blocks, it will also need to be attached to a block when it is placed so that it knows which block to refer to when the inventory saving is implemented and also so that the correct inventory is shown if there is more than one chest is in the world.

So, I made a new block class called chest shown in Code 31 – Chest Class, and also modified the selector place block function so check for interactable blocks as shown in Code 32 – Modified Place Block.

```
namespace Blocks
```

```
{
    /// <summary>
    /// Chest Block
    /// </summary>
    [Serializable]
    public class Chest : Block
    {
        /// <summary>
        /// Chest model for when it is placed
        /// </summary>
        [NonSerialized]
        private GameObject myGameobject;

        public new static int ID => 8;

        /// <summary>
        /// Makes a new chest from a parameterless constructor
        /// </summary>
        public Chest() : base("Chest")
        {
            usesGameObject = true;
        }

        /// <summary>
        /// Gets the gme object for this chest
        /// </summary>
        /// <returns>The chest game object</returns>
        public override GameObject GetGameObject()
        {
            return PrefabDictionary.GetPrefab("Chest");
        }

        /// <summary>
        /// Returns the texture for the chest <see cref="Block"/>
        /// </summary>
        /// <param name="direction"><see cref="Direction"/> of thhe desired
face</param>
        /// <returns><see cref="Tile"/> with the textture coordinates of the
<see cref="Block"/> texture</returns>
        /// <remarks>
        /// Returns a transparent texture as the chest model already has a
texture applied
        /// </remarks>
        public override Tile TexturePosition(Direction direction)
        {
            return new Tile() { x = 0, y = 9 };
        }

        /// <summary>
        /// The data that this block adds to the mesh
        /// </summary>
        /// <param name="chunk">Chunk the block is in</param>
        /// <param name="x">X pos of the block</param>
        /// <param name="y">Y pos of the block</param>
        /// <param name="z">Z pos of the block</param>
        /// <param name="meshData">meshdata to add to chunk</param>
        /// <param name="addToRenderMesh">should the block also be added to the
render mesh not just the collision mesh</param>
        /// <returns>Given <paramref name="meshData"/> with this blocks data
added to it</returns>
        /// <remarks>
        /// Only adds to the collision mesh as the model is handlled by the
unity prefab system
        /// </remarks>
        public override MeshData BlockData(Chunk chunk, int x, int y, int z,
MeshData meshData, bool addToRenderMesh = true)
        {
            if (myGameobject == null)
            {
                myGameobject =
UnityEngine.Object.Instantiate(PrefabDictionary.GetPrefab("Chest"), new
THVector3(x, y, z) + chunk.chunkWorldPos, Quaternion.identity, chunk.transform);
                myGameobject.GetComponent<ChestInventory>().inventoryPosition =
new THVector3(x, y, z) + chunk.chunkWorldPos;
                myGameobject.GetComponent<ChestInventory>().SetChestInventory();
            }
            return base.BlockData(chunk, x, y, z, meshData, true);
        }

        /// <summary>
        /// Breaks the block
        /// </summary>
        /// <param name="pos">Position of the block</param>
        public override void BreakBlock(THVector3 pos)
        {
// destroys the game object
            UnityEngine.Object.Destroy(myGameobject);
// removes the collision mesh from the chunk
            base.BreakBlock(pos);
        }

        /// <summary>
        /// Gets the Chest <see cref="Sprite"/>
        /// </summary>
        /// <returns>The Chest <see cref="Sprite"/></returns>
        public override Sprite GetItemSprite()
        {
            return SpriteDictionary.GetSprite("Chest");
        }

        /// <summary>
        /// Opens the <see cref="ChestInventory"/> when clicked on
        /// </summary>
        /// <param name="inv">Inventory that the chest is interacting
with</param>
        /// <returns>true</returns>
        public override bool InteractWithBlock(Inventory inv)
        {
            myGameobject.GetComponent<ChestInventory>().ToggleInventory(inv);
            return true;
        }
        /// <summary>
        /// Gets the ID of the <see cref="Block"/>
        /// </summary>
        /// <returns>8</returns>
        public override int GetHashCode()
        {
            return ID;
        }

        /// <summary>
        /// Returns the <see cref="Block"/> name and ID formatted nicely
        /// </summary>
        /// <returns></returns>
        public override string ToString()
        {
            return $"{itemName}\nID{GetItemID()}";
        }
    }
}
```

Code 31 – Chest Class

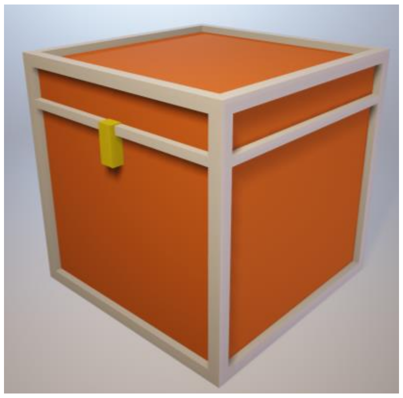


Figure 26 – Chest Model

Figure 27 – Chest Icon

```
void PlaceBlock()
{
    Chunk chunk = GetChunk(selector.transform.position);
    if (chunk == null)
        return;

    if (chunk.GetBlock((int)selector.transform.position.x -
chunk.chunkWorldPos.x, (int)selector.transform.position.y -
chunk.chunkWorldPos.y, (int)selector.transform.position.z -
chunk.chunkWorldPos.z).InteractWithBlock(playerInventory))
        return;

transform.parent.GetComponentInChildren<PlayerInventory>().GetItemFromHotBar(sel
ectedHotbarSlot, out var item);

    if (item != null)
        if (item.placeable)
            chunk.world.SetBlock(selector.transform.position + hit.normal,
(Block)item.CloneObject(), true);
 }
```

Code 32 – Modified Place Block

As can be seen this is the first block to use its own model for a block, this model can be seen in Figure 26 – Chest Model also by this point I was sick of making icons so I just found random pictures on the internet the icon for the chest is shown in Figure 27 – Chest Icon, as I had already planned for blocks to use premade models as their blocks it was not very difficult to implement into this block as it needed no other changes to the code than a normal block other than the function allowing interaction with the block, the prefab dictionary work the same way as the sprite dictionary, however with Unity prefabs instead of sprites.

This works by first the selector checks of a block is interactable by attempting to call the "InteractWithBlock" method passing the player inventory as the argument, in case there are more than one player inventories it is passed as an argument(all blocks have this function it is inherited from parent block class and later from the item class so that items can also be interactable), if the block can be interacted with then it will return true otherwise false will be returned and the placement will carry on as normal.

If the block is interactable the inventory attached to the block when it was placed will be opened, this in turn calls the method to add the players items into the inventory, the result is shown in Figure 28 – Block Inventory.

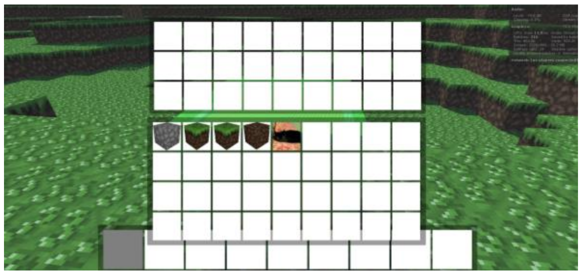


Figure 28 – Block Inventory

The next this is to allow the player to interact with the inventory by allowing movement of items between their inventory and the block inventory.

As it happens, since I make a copy of the player inventory in the block inventory to display the items it can just move the items around as if they are the same inventory, so I don't have to add anything as it is already handled by the base class, in case you wanted to see items in the chest inventory and the player inventory it is shown in Figure 29 – Items in Chest and Player Inventory.

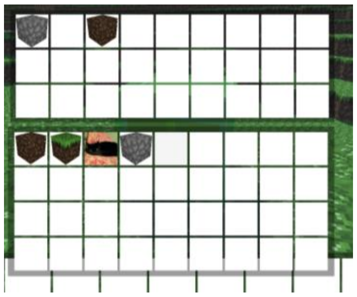


Figure 29 – Items in Chest and Player Inventory

Now all that needs to be done is applying the changes to the player inventory and saving the chest inventory to a file so that its contents persists across play sessions.

To apply the player inventory all that is done is go through each item slot in the player inventory reference the block has and apply the changes made in the block, this method is shown in Code 33 – Apply Changes to Player Inventory. After the changes are made the player inventory is saved.

```
/// <summary>
/// Applies the changes made to the <see cref="playerinventory"/> in <see
cref="this"/>
/// </summary>
void ApplyPlayerItems()
{
    for (int i = 0; i < playerinventory.items.itemsInInventory.Length; i++)
        playerinventory.items.itemsInInventory[i] = items.itemsInInventory[i +
(inventorySize - 36)];

    playerinventory.SaveInv();
}
```

Code 33 – Apply Changes to Player Inventory

To save the block inventory whenever the inventory is closed the entries contents of the chest will be saved into a file called "Chest @ X, Y, Z", where x, y, z is the blocks position in the world (I think? It "should" be at least).

Then when the block is reloaded the save file is retrieved. The code to save and load the block inventories can be reused from the earlier Code 26 – Inventory Saving, all that needs to be done is alter the save location and file name. The inventory is also already automatically saved as the base inventory class has that functionality, all that needed to be added was loading the inventory and this is done just after the inventory has been created and is shown in *Code 34 – Block Inventory Class Load Method*.

```
/// <summary>
/// Sets the Size and name of this <see cref="Inventory"/>
/// </summary>
public virtual void SetChestInventory(string invName = "Chest")
```

```
{
    SetInventorySize(inventorySize);
// sets the UI to not be seen as inventorys cannot start open
    inventory.SetActive(false);

// sets the name and postion if this inventory used during serialization and
deserialization
    inventoryName = $"{invName} @ {(ChunkWorldPos)inventoryPosition}";

// loads the inventory if it had had items put in it last time it existed
    Serialization.Serialization.DeSerializeInventory(this, inventoryName);
}
```

Code 34 – Block Inventory Class Load Method

Now that blocks can easily have inventories by extending the "ChestInventory" class blocks that allow for crafting will be much easier to implement. All that must be done is make an inventory the will extend the base chest class that checks certain slots and if the items input match a recipe.

For the crafting recipes I will use a dictionary that will store the needed items as a string which is a key to a dictionary and the resulting item will be the value for the dictionary.

Each slot in the crafting grid will be represented by a string of "X:X:X:X:X:X:X:X:X" where each X represents an item slot in the grid and each group of 3 X's is one row of the grid. To store the recipes each X will be replaced by the needed items specific ID. I the slots were, and item is not needed the will be represented by an empty space. Therefore, a recipe could be represented as " :1: :1:1:1: :1: ", this recipe would require the user to input a recipe using item ID 1 in and English cross shape in the crafting grid. Some recipes my also only need the items in the grid but no is specific places, these "shapeless" recipes will be stored in their own dictionary and the crafting class will have to check both when a recipe is attempted to be crafted. The class to store this data is shown in Code 35 – Crafting Recipes Storage.

```
namespace Core.Dictionaries
{
    public static class CraftingRecipies
    {
        /// <summary>
        /// Contains all crafting recipes that require a certain layout in the
crafting grid (<see cref="Blocks.CraftingTable"/>
        /// </summary>
        private static Dictionary<string, Item> shapedCraftingRecipies = new
Dictionary<string, Item>();

        /// <summary>
        /// Will add a shaped crafting recipe to the game
        /// </summary>
        /// <param name="reicpe">The desired recipe.  Layout is {"XXX", "XXX",
"XXX", "X", ItemID} where each X is a slot in the crafting grid, Each group of 3
is a row, and a "X", ItemID is the <see cref="Item"/> ID X represents (for each
new item a new symbol is required), a Sapce is no item required in that
slot</param>
        /// <param name="result">The <see cref="Item"/> that the recipe will
produce</param>
        /// <example>
        /// This example shows how to call <see cref="AddShapedRecipie(object[],
Item)"/>
        /// <code>
        /// void Main()
        /// {
        ///     CraftingRecipies.AddShapedRecipie(new object[] { " X ", "X@X", "
X ", "X", Wood.GetItemID(), "@", Stone.GetItemID() }, new Chest());
        /// }
        /// </code>
        /// </example>
        public static void AddShapedRecipie(object[] reicpe, Item result)
        {
// converts the given blocks of 3 characters to a 9 character string
            var stringRecipie = "";

            for (int i = 0; i < 3; i++)
            {
                stringRecipie += reicpe[i] as string;
            }

// gets what character represents which item
            for (int i = 3; i < reicpe.Length; i += 2)
            {
                var character = (string)reicpe[i];
                var itemID = (int)reicpe[i + 1];

// replaces the character with the items id
                stringRecipie = stringRecipie.Replace(character,
$"{itemID.ToString()}:");
            }

// converts empty sots " " into "0:"
            stringRecipie = stringRecipie.Replace(" ", "0:");

// if the recipe exists an exception is thrown as two recipies cannot be the
same
            if (shapedCraftingRecipies.ContainsKey(stringRecipie))
                throw new CraftingRecipeAdditionException($"Shaped Recipie
already exists: {stringRecipie}");

            result.itemStackCount = 1;

// adds the recipe to the dictionary
            shapedCraftingRecipies.Add(stringRecipie, result);
        }

        /// <summary>
        /// Returns an <see cref="Item"/> from the <see
cref="shapedCraftingRecipies"/> dictionary
        /// </summary>
        /// <param name="recipe">Recipe for <see cref="Item"/></param>
        /// <returns>An <see cref="Item"/> or <see cref="null"/> is recipe was
not found</returns>
        public static Item GetShapedRecipeItem(string recipe)
        {
            shapedCraftingRecipies.TryGetValue(recipe, out var item);
            if (item != null)
                item.itemStackCount = 1;

            return item;
        }

        /// <summary>
        /// All shapeless recipes
        /// </summary>
        private static Dictionary<string, Item> shaplessRecipies = new
Dictionary<string, Item>()
        {
            {"5:", new Blocks.Planks() }
        };

        /// <summary>
        /// Adds a Shapeless recipe to the dictionary
        /// </summary>
        /// <param name="recipe">Recipe to add. Format as { Item, Number of
items }</param>
        /// <param name="result">Result of the crafting recipe</param>
        /// <example>
        /// 2 Examples of adding a shapeless recipe
        /// <code>
        /// void Main()
        /// {
        ///     CraftingRecipies.AddShaplessRecipie(new object[] { new Dirt(),
2 }, new Grass());
        /// }
        /// </code>
        ///
        /// <code>
        /// void Main()
        /// {
        ///     CraftingRecipies.AddShaplessRecipie(new object[] { new Stone(),
3, new Wood(), 3 }, new Apiary());
        /// }
        /// </code>
        /// </example>
        public static void AddShaplessRecipie(object[] recipe, Item result)
        {
            var itemList = new List<int>();
            var stringRecpie = "";

            for (int i = 0; i < recipe.Length; i+=2)
            {
                for (int j = 0; j < (int)recipe[i+1]; j++)
                {
                    itemList.Add(int.Parse(((Item)recipe[i]).GetItemID()));
                }
            }

            itemList.Sort();

            for (int i = 0; i < itemList.Count; i++)
            {
                stringRecpie += $"{itemList[i]}:";
            }

            if (shaplessRecipies.ContainsKey(stringRecpie))
                throw new CraftingRecipeAdditionException($"Shaped Recipie
already exists: {stringRecpie}");

            result.itemStackCount = 1;
            shaplessRecipies.Add(stringRecpie, result);
        }

        /// <summary>
        /// Gets a shapeless recipe string from a given recipe
        /// </summary>
        /// <param name="recipe">Recipe for string</param>
        /// <returns>A string of the given shapeless recipe</returns>
        public static string GetShaplessRecipieString(Item[] recipe)
        {
            var IDList = new List<int>();
            var stringRecipe = "";

// converts the given item list to an ID list so it can be sorted
            for (int i = 0; i < recipe.Length; i++)
            {
                if(recipe[i] != null)
                    IDList.Add(recipe[i].GetHashCode());
            }

            IDList.Sort();

// converts the sorted ID list to a string so it can be used as a dictionary key
            for (int i = 0; i < IDList.Count; i++)
            {
// after each ID as it is possible for ID clashes without eg ID: 11 can be seen
as 2 * ID: 1
                stringRecipe += $"{IDList[i]}:";
            }

            return stringRecipe;
        }

        /// <summary>
        /// Tries to get a shapeless recipe
        /// </summary>
        /// <param name="recipe">Recipe to get</param>
        /// <returns><see cref="Item"/> for the recipe, null if recipe does not
exist</returns>
        public static Item GetShaplessRecipieResult(int[] recipe)
        {
            var list = recipe.ToList();
            list.Sort();

            var stringRecipe = "";

            for (int i = 0; i < list.Count; i++)
            {
                stringRecipe += $"{list[i]}:";
            }

            return GetShaplessRecipieResult(stringRecipe);
        }

        /// <summary>
        /// Tries to get a shapeless recipe
        /// </summary>
        /// <param name="recipe">Recipe to get</param>
        /// <returns><see cref="Item"/> for the recipe, null if recipe does not
exist</returns>
        public static Item GetShaplessRecipieResult(string recipe)
        {
            shaplessRecipies.TryGetValue(recipe, out var item);

            return item;
        }

        /// <summary>
        /// Tries to get a shapeless recipe
        /// </summary>
        /// <param name="recipe">Recipe to get</param>
        /// <returns><see cref="Item"/> for the recipe, null if
        /// does not exist</returns>
        public static Item GetShaplessRecipieResult(Item[] recipe)
        {
            shaplessRecipies.TryGetValue(GetShaplessRecipieString(recipe), out
var item);

            if (item != null)
                item.itemStackCount = 1;

            return item;
        }
    }
}
```

Code 35 – Crafting Recipes Storage

To add a crafting recipe to the game a method is sued so validate the input and if mods were loaded into the game then it would give the mod creator a clear method to add a crafting recipe to the game.

The “AddShapedRecipie” works by being given an object array containing at least 5 things, the first 3 elements of the array will represent each row of the crafting grid, then the next will be a character and the final element will be the item used to craft and example usage is shown in the XML comment in Code 35 – Crafting Recipes Storage above the “AddShapedRecipie” method. Each character given in the first 3 elements represents an item, the next element after indicates the meaning of each character and the final index is the item that should replace the character, each item and character come in pairs with the character that represents the item preceding the item in the array.

The reason that the recipe is stored as a string is for simplicity as by default a dictionary cannot have an array as a key however they can use a string therefore it is logical to use keys as a string instead of making my own method of searching the dictionary.

For shapeless recipes they are added by the method "AddShaplessRecipie" and this is also passed an object array however this contains item and number pairs, this is converted into an array of the item ID with the ID appearing in the array the same number of times that the number given in the array after it dictates. This array is then sorted and converted into a string like the shaped recipes.

To access a recipe the contents of the crafting grid are given to the function the array is sorted by item ID then conveted into a string and if it matches something the crafting dictionaries.

The next thing that needs to be done is the blocknto allow for crafting. This is the WorkBench and its class is shown in Code 36 – WorkBench Class.

```
namespace Blocks
{
    /// <summary>
    /// The Workbench <see cref="Block"/> class
    /// </summary>
    [Serializable]
    public class CraftingTable : Block
    {
        /// <summary>
        /// The <see cref="GameObject"/> for this block
        /// </summary>
        [NonSerialized]
        private GameObject myGameobject;

        /// <summary>
        /// This block's ID
        /// </summary>
        public new static int ID => 9;

        /// <summary>
        /// Constructor
        /// </summary>
        public CraftingTable() : base("Workbench")
        {
            usesGameObject = true;
        }

        /// <summary>
        /// Makes a shaped crafting recipe from the given items and return if it
is a recipe
        /// </summary>
        /// <param name="items">Items to make the recipe from</param>
        /// <returns>A <see cref="Item"/> if the recipe exists</returns>
        public Item ReturnShapedRecipieItem(Item[] items)
        {
            var recipe = "";

            for (int i = 0; i < items.Length; i++)
            {
                if (items[i] == null)
                {
                    recipe += "0:";
                    continue;
                }

                recipe += $"{items[i].GetItemID()}:";
            }

            return ReturnShapedRecipieItem(recipe);
        }

        public virtual Item ReturnShapelessRecipieItem(Item[] items)
        {
            return CraftingRecipies.GetShaplessRecipieResult(items);
        }

        /// <summary>
        /// Returns a crafting recipe from a given recipe
        /// </summary>
        /// <param name="recipe"></param>
        /// <returns>A <see cref="Item"/> if the recipe exists</returns>
        /// <remarks>
        /// Virtual incase needs to be overriden by a different crafting system
        /// </remarks>
        public virtual Item ReturnShapedRecipieItem(string recipe)
        {
            return CraftingRecipies.GetShapedRecipeItem(recipe);
        }

        /// <summary>
        /// Toggles the <see cref="CraftingTableInventory"/> for the block
        /// </summary>
        /// <param name="inv"></param>
        /// <returns></returns>
        public override bool InteractWithBlock(Inventory inv)
        {
            myGameobject.GetComponent<CraftingTableInventory>().myblock = this;

myGameobject.GetComponent<CraftingTableInventory>().ToggleInventory(inv);
            return true;
        }

        /// <summary>
        /// Returns this <see cref="Block"/>s game object
        /// </summary>
        /// <returns></returns>
        public override GameObject GetGameObject()
        {
            return PrefabDictionary.GetPrefab("CraftingTable");
        }

        /// <summary>
        /// The data that this block adds to the mesh
        /// </summary>
        /// <param name="chunk">Chunk the block is in</param>
        /// <param name="x">X pos of the block</param>
        /// <param name="y">Y pos of the block</param>
        /// <param name="z">Z pos of the block</param>
        /// <param name="meshData">meshdata to add to</param>
        /// <param name="addToRenderMesh">should the block also be added to the
render mesh not just the collsion mesh</param>
        /// <returns>Given <paramref name="meshData"/> with this blocks data
added to it</returns>
        /// <remarks>
        /// Only adds to the colision mesh as the model is handlled by the unity
prefab system
        /// </remarks>
        public override MeshData BlockData(Chunk chunk, int x, int y, int z,
MeshData meshData, bool addToRenderMesh = true)
        {
            if (myGameobject == null)
            {
                myGameobject =
UnityEngine.Object.Instantiate(PrefabDictionary.GetPrefab("CraftingTable"), new
THVector3(x, y, z) + chunk.chunkWorldPos, Quaternion.identity, chunk.transform);
            }
            return base.BlockData(chunk, x, y, z, meshData, true);
        }

        /// <summary>
        /// So that the game knows to render the faces of the blocks aroound the
crafting table as they can be seen
        /// </summary>
        /// <param name="direction"><see cref="Direction"/></param>
        /// <returns>false</returns>
        public override bool IsSolid(Direction direction)
        {
            return false;
        }

        /// <summary>
        /// Breaks the <see cref="Block"/>
        /// </summary>
        /// <param name="pos">Positon of the <see cref="Block"/></param>
        public override void BreakBlock(THVector3 pos)
        {
// removes the game object
            UnityEngine.Object.Destroy(myGameobject);
// removes the collision mesh from the chunk
            base.BreakBlock(pos);
        }

        /// <summary>
        /// Returns the sprite for the <see cref="Item"/>
        /// </summary>
        /// <returns><see cref="Sprite"/> for this <see cref="Item"/></returns>
        public override Sprite GetItemSprite()
        {
            return SpriteDictionary.GetSprite("CraftingTable");
        }

        /// <summary>
        /// Returns the texture for the crafting table <see cref="Block"/>
        /// </summary>
        /// <param name="direction"><see cref="Direction"/> of thhe desired
face</param>
        /// <returns><see cref="Tile"/> with the textture coordinates of the
<see cref="Block"/> texture</returns>
        /// <remarks>
        /// Returns a transparent texture as the chest model already has a
texture applied
        /// </remarks>
        public override Tile TexturePosition(Direction direction)
        {
            return new Tile() { x = 0, y = 9 };
        }

        /// <summary>
        /// Returns the ID of the Item
        /// </summary>
        /// <returns><see cref="ID"/></returns>
        public override int GetHashCode()
        {
            return ID;
        }
    }
}
```

Code 36 – WorkBench Class

In Code 36 – WorkBench Class this is the class the the chunk will contain to know that the block is there.

This will return the items that needs to be crafted using the “ReturnShapedRecipieItem”, and “ReturnShaplessRecipieItem”, theise are called by the crafting inventory class when it is open. They are given the items that are in the crafting grid and display the returned item in the output slot. This can be seen inFigure 30 – Crafting Grid .

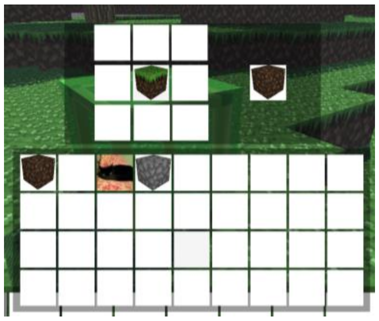


Figure 30 – Crafting Grid

For the quests it made sense to be able to interact with items, as a “Quest Book” type thing. Theirfore I implemented this. It was rather simple, as all that needed to be done was add a new function to the base item class and check if it interactable.

When the item is interacted with it will spawn an inventory of the correct type and after this it works just like the crafting inventory, however specialised to the function of the item.

Whis meant all that was done to implement quests was to make an event system that plugs into the existing crafting systems and item collection system, this checks that item that was passed and if ti was part of a quests the quest is triggered. The UI for the quests is shown in Figure 31 – Quest Book UI, with red being uncompleated quests, yellow being completed not collected quests, and greed being completed and collected quests. The quests are stored in an dictiaonry just like the crafting recipies.

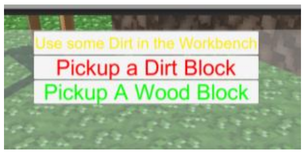


Figure 31 – Quest Book UI

After this was done I saw a Minecraft mod called Forestry which has a bee breeding system. I decided to impement it as it would be a good challenge to see if my item system could handle an item with the same primery ID however a different secondary ID as each bee has different stats however they are all technicaly the same item thefore there is is setout as “PrimaryID\SecondaryID” the secondary ID is as shown in Code 37 – Bee Secondary ID.

```
public override int GetHashCode()
{
    unchecked
    {
        var temp =
$"{(int)pSpecies}{(int)sSpecies}{(int)pLifespan}{(int)sLifespan}{(int)pFertility
}{(int)sFertility}{(int)pEffect}{(int)sEffect}{(int)pProdSpeed}{(int)sProdSpeed}
";

        var hashcode = (int)(Int64.Parse(temp) ^ (127 * 13) / 159);

        return hashcode;
    }
}
```

Code 37 – Bee Secondary ID

The bees are split into 3 catogories, Drone, Princess, and Queen. The Princess is mated with a Drone an an Apiary to form a Queen. The queen then produces items over its lifespan and will then also produce offspring with stats randomized based on the original Drone and Princess. Each ata has a primary and secondary, the primary trait is the trait that effect the bee in the currsnt generation and the secondary trait is when the next generation bees will be made from. The species would control what items are produced by the bee (I didn’t have time to implement this for now they only produce honey combs of differing colours) and the other traits are self explanatory.

The Bee Item class does all this and is shown in Code 38 – Bee Item Class.

Code 38 – Bee Item Class

To signify differens species each bee has its own colouring, as I did not want to manualy make a sprite for each colour a dictionary is used for he colour needed then the outside white area of the sprite is colourd accordingly, 2 differently coloured sprites are shown in Figure 32 – Bee Sprites. This is also how the original mod colours the bees, so I took the sprite from that mod.

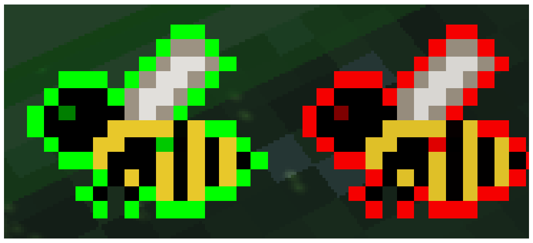


Figure 32 – Bee Sprites

The apiary then goes thorugh all the stats of the bee and produces item and offspring as shown in Figure 33 – Apiary UI.

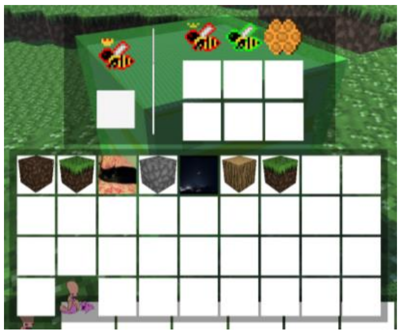
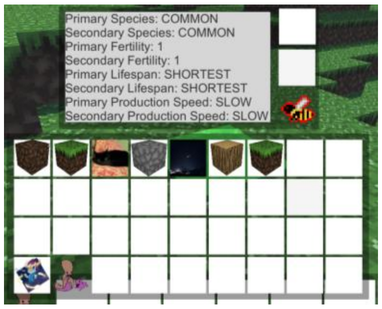


Figure 33 – Apiary UI

To see the stats of the bees I also made a new item called a "BeeAlyzer" that just displayes all the stats of the bee as text. this can be seen in Figure 34 – BeeAlyzer.



```
Primary Species: COMMON
Secondary Species: COMMON
Primary Fertility: 1
Secondary Fertility: 1
Primary Lifespan: SHORTEST
Secondary Lifespan: SHORTEST
Primary Production Speed: SLOW
Secondary Production Speed: SLOW
```

Figure 34 – BeeAlyzer

After this the 3rd prototype was compleat and ready for testing.

FEEDBACK FROM PROTOTYPE 3

After giving this to the testers I got this feedback:

Test Number	Description of test	Input data	Expected result	Actual result	Type
1	Save	Keypress	Data saved correctly	Works	Iterative
2	Loading	Starting the game	Data loaded correctly	Works	Iterative
3	Forward Movement	Keypress	Character moves forward	Works	Iterative
4	Backward moment	Keypress	Character moves backward	Works	Iterative
5	Left Movement	Keypress	Character moves to the left	Works	Iterative
6	Right Moment	Keypress	Character moves to the right	Works	Iterative
7	Terrain Loads	Noise function	Terrain loads corrects with no seams	Works	Iterative
8	Block destruction works/saves correctly	Keypress/File	Block is destroyed correctly without causing a crash and file is loaded correctly	Works	Iterative
9	Block placement works/saves correctly	Keypress/File	Block is placed correctly in the correct position without causing a crash and saved/loaded correctly when the game is restarted	Works	Iterative

10	Inventories save stored items	Items inside an inventory	Items are in the same position, of the same type, and number as then input into the inventory	Works	Iterative
11	Items can be input into an inventory	Items moved into an inventory	Item can be added to an inventory without duplication and without being able to override other items	Works	Iterative
12	Items can be removed from an inventory	Keypress	Items can be removed form an inventory without duplication, overriding other items, and remove the correct number of items	Works	Iterative

As can be seen for one I did everything fomr according to the table however their were problems, firstly items could be put into crafting output slots and also if a block like a chest was destroyed with items within it, the next time a chest is placed at the same location the items previously in the chest appeared in the new chest, this is becse the save file was not deleted.

Both were thankfuly quick fixes and all that needed to be done was add a line of code preventing items entering the crafting output slots and deleting the block's save file when it is destroyed iun the game.

POST DEVELOPMENT TESTING

Test Number	Description of test	Input data	Expected result	Actual result	Type
1	Save	Keypress	Data saved correctly	Works	Iterative
2	Loading	Starting the game	Data loaded correctly	Works	Iterative

3	Forward Movement	Keypress	Character moves forward	Works	Iterative
4	Backward moment	Keypress	Character moves backward	Works	Iterative
5	Left Movement	Keypress	Character moves to the left	Works	Iterative
6	Right Moment	Keypress	Character moves to the right	Works	Iterative
7	Terrain Loads	Noise function	Terrain loads corrects with no seams	Works	Iterative
8	Block destruction works/saves correctly	Keypress/File	Block is destroyed correctly without causing a crash and file is loaded correctly	Works	Iterative
9	Block placement works/saves correctly	Keypress/File	Block is placed correctly in the correct position without causing a crash and saved/loaded correctly when the game is restarted	Works	Iterative
10	Inventories save stored items	Items inside an inventory	Items are in the same position, of the same type, and number as then input into the inventory	Works	Iterative

11	Items can be input into an inventory	Items moved into an inventory	Item can be added to an inventory without duplication and without being able to override other items	Works	Iterative
12	Items can be removed from an inventory	Keypress	Items can be removed form an inventory without duplication, overriding other items, and remove the correct number of items	Works	Iterative
13	Game runs on multiple different systems	Computers	Game runs correctly on multiple different systems with different specs	Works, howerver slow on some machines	Post-Development
14	Save file naming	Acceptable name (no special characters)	File is named as why the user inputs	Works	Post-Development
15	Save file naming	Erroneous data (special characters	File name is rejected, and user is notified	Works	Post-Development

Now the post development testing can be done. After compiling the game for what I thought was the final time I reilised that I forgot to implement a way to name the save files, so I made a main menu that allows this, and it works.

the game does however run slowly one some less powerful machines due to inefficencites when generteing the world however there is not time to go back and optimise this further.

Evaluation

Is this all I have to now?