**Working with Game Engines**

**Report**

**Declan Wylie**

**Matriculation Number: S1429200**

# **Scene 1: Voxel Scene**

Within this first scene there are multiple scripts that contribute to the creation and handling of the dynamic terrain created using voxels, the VoxelChunk and VoxelGenerator are used to create the terrain and to update it if any changes happen.

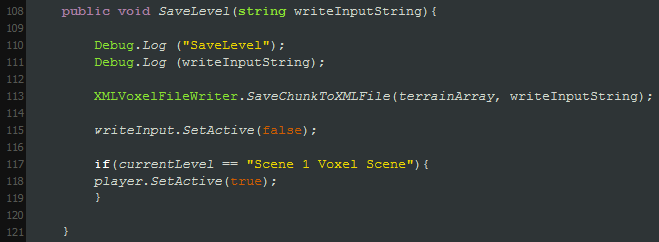
The VoxelGenerator script is capable of creating each voxel by creating each face using lists generated which contain the triangles, vertices and UV coordinates. The script has separate methods that create a face for the voxel for each side, a dictionary with a string key and values stored as a Vector 2, this dictionary is used to split the single texture into 4 “different” textures that can be used separately on each block.

The VoxelGenerator only contains methods and doesn’t actually call any on it’s own, the VoxelChunk script is what calls each method stored within the VoxelGenerator.

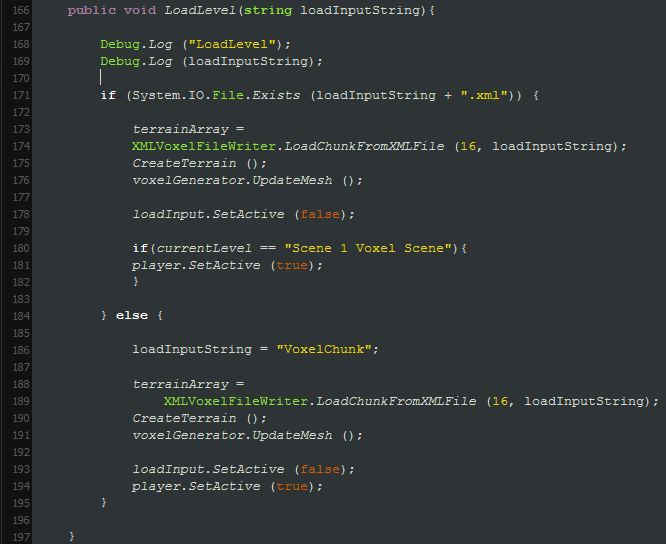
The Voxels are created by adding vertexes to a list, the vertexes are dependent on location and what direction the plane will face, after the vertexes are added to a list a method is called that will add the respective triangles to a list. After the triangle list is created the UV Coordinates are then added to a list to allow texturing, this is all contained within a CreateVoxel method that will take in coordinates and a vector 2 that will represent the UV coordinates for the texture used.

The VoxelChunk script is responsible for using methods contained within VoxelGenerator and it’s own methods to create the VoxelChunk object (The terrain). When an instance of VoxelChunk is loaded it first grabs the current loaded level and stores it in currentLevel, this is used to allows the script to perform actions dependant on the scene which is useful for dodging nullreference errors when in scenes without a player object. The VoxelGenerator is then found and the terrain array is created then the terrain is initialized and updated, a check is made to see if this is a prefab instantiated (meaning it’s created in scene 3), in this case the “AssessmentChunk1.xml” is loaded to change the terrain to the specified terrain to be used in the networking scene.

If the F1 key is pressed then the player object is set inactive and the input field for saving the level is set active, when this input field is set active the user will input a string that will then create a new file with the entered string that contains data that when read will create the respective geometry. If the F2 key is pressed the player object is then again set inactive and an input field will be set active, if the user enters a file name that is contained within the game folder that file if it is a valid XML file will be loaded and the terrain will be updated. A screenshot of the method setup for saving is shown below.



The loading method is very similar to the writing method with the exception that it will check if the file exists, if it doesn’t then a default filename is used, in this case the default file is called “VoxelChunk”.

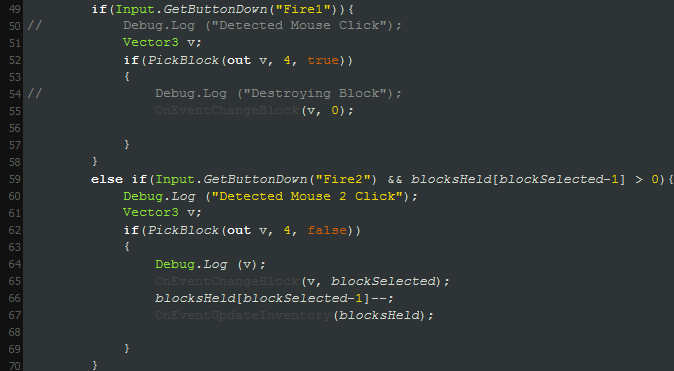


The terrain is created using a method (CreateTerrain()) contained within VoxelChunk which contains loops that will iterate through each dimension of the 3D array and check each potential face to see if it needs to be created, the voxel created is assigned a texture based on a value between 1 and 4, 1 being grass and 4 being sand.

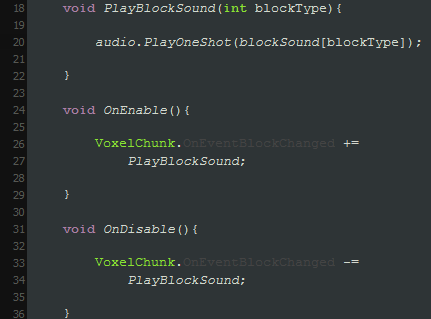
The PlayerScript is attached to the First Person Controller and responsible for destroying and placing blocks along with handling collisions with dropped block prefabs and also invokes some events related to the inventory system and the VoxelChunk script to edit terrain.

The player has the ability to select different blocks to place if they have enough in their inventory, the block selection system simply works off an integer value that is changed depending on the button pressed. The block destruction and block placement are extremely similar and actually use the same method, a Boolean value is passed into the method which is either true if the player is destroying a block or false if they are placing a block.

When the player presses a button to place or destroy a block an empty vector 3 is passed in along with a distance that determines how far the raycast will travel as well as the previously mentioned Boolean into a method. Within this method a raycast is performed from the camera to the middle of the screen and if a hit is detected the vector passed in is then either set to the hit coordinates – the normal halved if the player is destroying a block or the hit coordinates + the normal halved if a block is being placed. The method then returns true indicating that the raycast hit and a block can either be placed or destroyed otherwise false is returned. When true is returned an event (OnEventChangeBlock) is called passing in the vector from the previously mentioned method and a value for the selected block is also passed in and the amount of blocks held is reduced by one, if the player is destroying a block the passed in value is 0. The code for the input is shown below, PickBlock(Vector 3, int, bool) is the method to determine the coordinates for destruction or placing using a raycast.

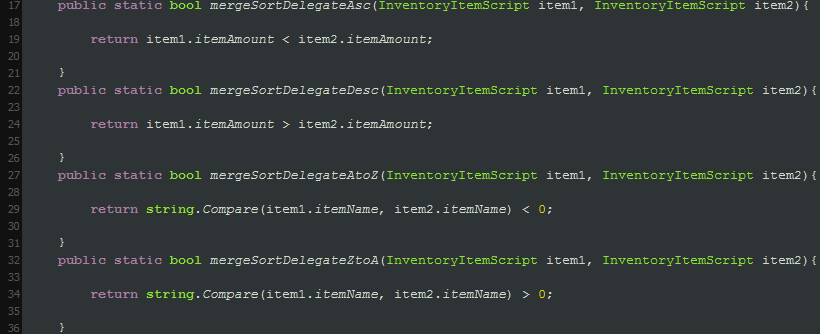


The OnEventChangeBlock() event will pass in a vector 3 and an integer, the integer represents the new block type that will replace the hit block in the terrain. This event will call the VoxelChunk script’s SetBlock method which will take the vector 3 in as coordinates or an index for the terrain 3D array. The coordinates passed in will then point to a voxel and the voxel value, the value of the block being the block type, will be changed to whatever block type was passed in from the event. The terrain is then updated and another event is invoked called OnEventBlockChanged(int blocktype) which will call a method within the AudioManager script to play a sound from an array that corresponds to the blocktype passed in. This allows the placing of blocks to play different sounds depending on the block type the code for playing audio is shown below.

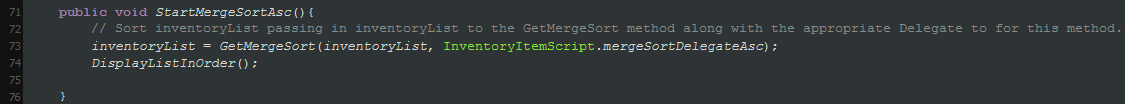


A prefab is also instantiated if a block is destroyed, the prefab instantiated is dependent yet again on the block type destroyed and it is instantiated at the destroyed block’s position however slightly higher to avoid it going into the terrain. If the player would walk into the dropped prefab then the player’s block of that type held is added onto and another event is invoked passing in the blocks held of each type, this event calls a method within the InventoryManager which will update the amount of each block held in the inventory script.

A merge sort method is implemented within the inventory system, the merge sort is a recursive sorting method which takes in a single list and in this case a delegate and splits the list into two smaller lists then recurs and when the lists have been split into the smallest they can they are then merged based on the values in each list. The delegate passed in is a comparison delegate, the comparison delegate passed in allows the sorting of the merged list in an ascending/descending fashion along with an alphabetical and reverse-alphabetical order. The different ordering methods are shown in a screenshot, notice that these are located within a different script from the script that declares the delegate.



In the InventoryManager script the merge sort is called using a method corresponding to the sorting order desired, an example of one of the methods called using buttons on scene is shown below.



Within this scene is also a button that will return the user back to the main menu, within the main menu the user can switch to either scene one, two or three.

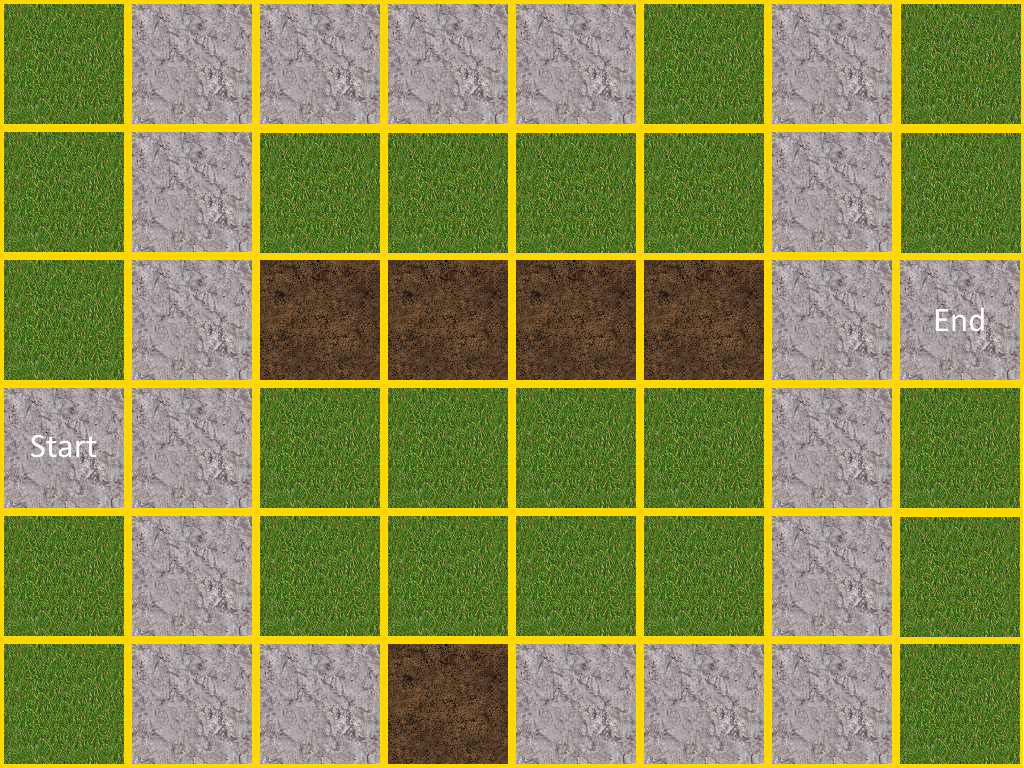
# **Scene 2: Pathfinding Scene**

Within this scene AssessmentChunk1.xml and AssessmentChunk2.xml are both loaded using a method which will check for the start element “Voxel” and parse the integers contained within each voxel element into x, y and z along with reading another value at the end of each element which is the block type, the voxel array dictionary is then set equal to [x, y, z] = block type.

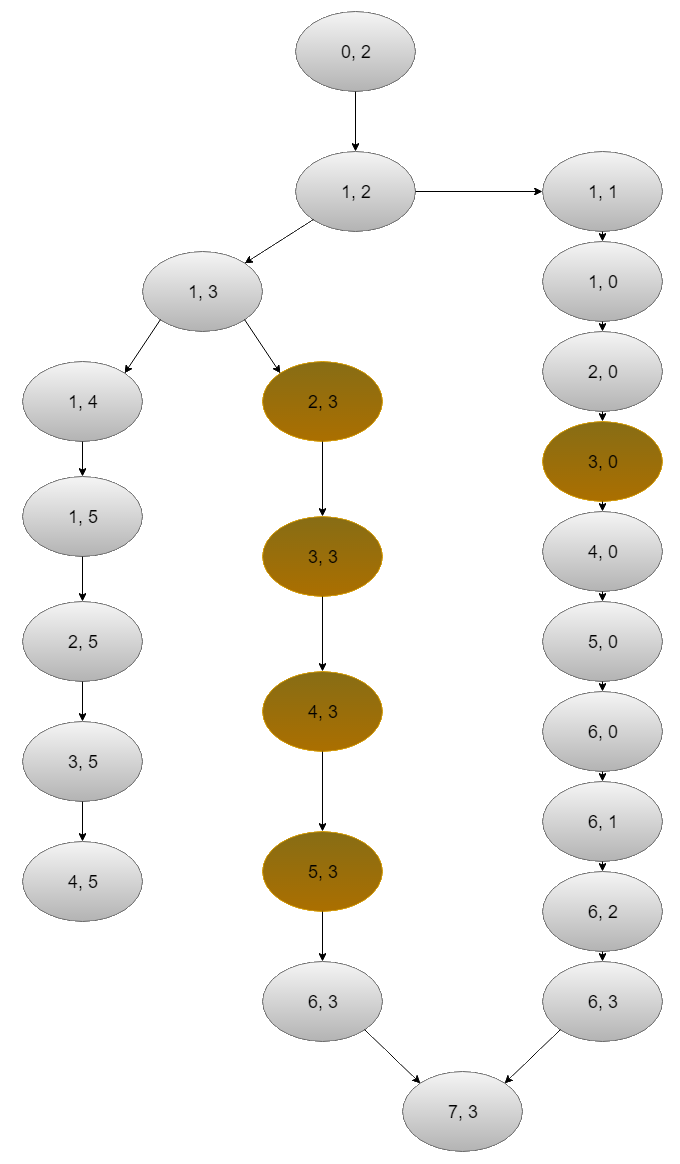
A diagram showing the DOM for the file is shown below.

This is showing the DOM for the reading method, when a file is being written the start and end positions are not saved.

Next up is the pathfinding for Dijkstra’s Algorithm, in my case it is very almost completed however runs into bugs or errors which I can’t find a solution to, however I’ll show a diagram below on how it should work for AssessmentChunk1.xml.



In my version there is no costing or distance weighting as when I tried to implement it there were compiler errors within the methods used, here is how it would work however if costing was implemented.



With dirt weighing 3 and stone weighing 1 the path chosen or order of coordinates interpolated to would be in order of the list of coordinates to the right of the above diagram as the total cost of the middle path is 16 and the path to the right has a total weight of 14.

With no costing implemented the block would go through the middle route since all costs are 1 the middle path would only have a cost of 8 whilst the right path would cost 12.

# **Scene 3: Networking Scene**

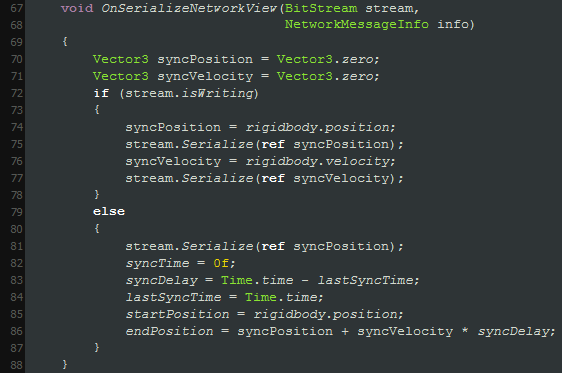
Within scene 3 a voxel chunk or “terrain” object is instantiated via the network when the server is initialized, the terrain object is instantiated by the game server and not the client.

The VoxelObject prefab has a network view attached to allow modification of the terrain and the networked cube prefab also has a network view attached to it although this is an overwritten OnSerializeNetworkView() method that will allow the cube to have latency compensation attached in and this case dead reckoning implemented.

Within this overwritten method we set the syncposition to 0 along with the syncvelocity and if a stream is being written the syncposition is set to the cube’s current position and the syncvelocity is set to the cube’s velocity both are sent.

If the stream is not writing then the syncposition is sent (it’ll be 0 at this point) along with making the syncTime equal to zero and the synchronized delay is set to the current time – the last sync time, the last sync time is then set to the current time. (Last sync time is essentially time since communication with the server) The start position is then set to the current cube’s position and the end position is set to the sync position + the sync velocity \* sync delay. This effectively predicts where the cube will be next frame.

Within the cube script we also check if the cube belongs to the current client, if it does then the components used for moving the cube such as the other script used to control movement are set active, this will effectively stop the server client or another client from controlling the movement of the cube. Within the update method we check if we own the cube and if we don’t we set the sync time to the delta time and if the sync time is less than the sync delay we interpolate the cube from the start position and end position over time sync time / sync delay. All the values used for interpolation are the output of the dead reckoning algorithm describes above, the code for the dead reckoning is shown below.



Within the update method we also call the method used to modify terrain if the mouse button is clicked, this method only allows destruction of the terrain and the raycast is modified to raycast downwards from the cube. If we return that we hit the terrain then we grab the network component located on the terrain object and if the networkview component isn’t null we use a remote procedure call. The method called using this is located in the VoxelChunk script and will take in the coordinates of the hit and the new block type. (in this case 0 for nothing)

The server has ownership of the VoxelObject (Terrain), and the client has ownership of the cube.

Once the server is initialized clients may connect, to do this the client must enter in the IP address of the master server they want to connect through, when they enter in the IP address the input field is set inactive and the client will request a host list with game names equal to the game name they have. If the master server receives a host list the client’s host list is then filled with the hosts within the master server’s list. For each game within the client’s host list a button is created and when that button is clicked it will join the respective game, when the client connects to the server the cube prefab is instantiated.