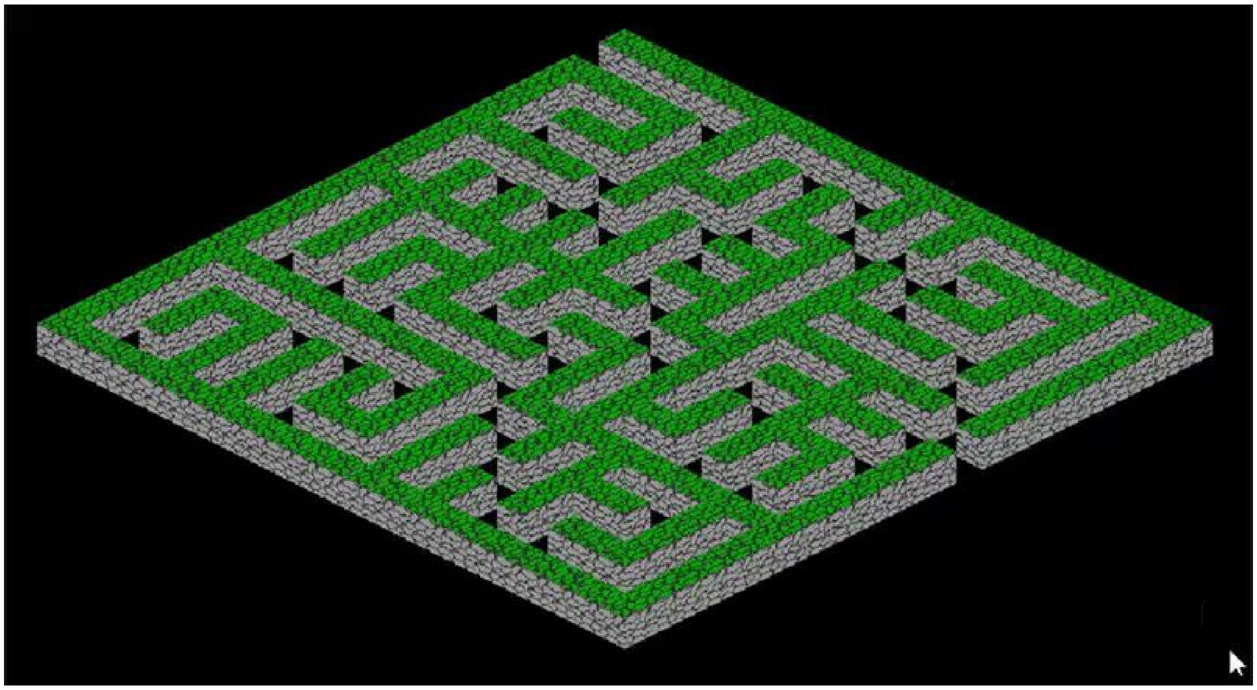
Isometric Voxel Maze Generator



The above image is what we are aiming to construct during this tutorial. Eventually we want to have a simple system that allows for a maze to be generated on the spot using voxels and a single large (collision) mesh.

Requested by : Ultimation1

1. Introduction

This project can be broken down in 3 sections of varying difficulty.

Firstly, an isometric viewpoint was requested, which should be straightforward to set up. This is quite different from the traditional (orthogonal) viewpoints. A custom controller should be developed to navigate a character in this perspective, which for now is out of the scope of this tutorial. Instead, we will setup a static camera - rig that is fixed to an isometric view.

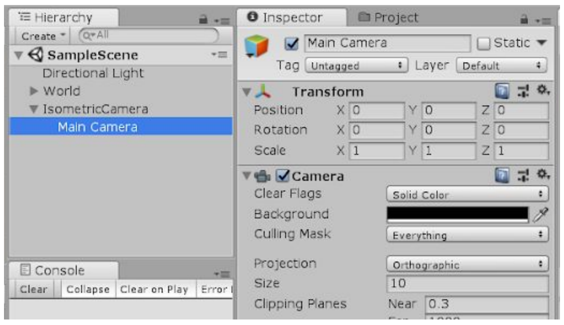
Secondly, one could easily opt to place cubes or game objects on a grid and use that as a template for a maze. Granted, this would work but has severe limitations, especially when designing very large or procedural mazes. In fact, there is a hard limit built in unity for the amount of concurrent game objects that can exist within a scene, something that can easily be avoided using voxels.

Lastly, a perfect maze generation algorithm should be implemented. In this context, perfect refers to the fact that a maze should have a single entrance point and a single exit point, in other words only a single solution. I have selected two options for this : a randomized Prim's algorithm and a recursive division algorithm. However, I found the latter to be more consistent in this case.

2. Isometric camera setup

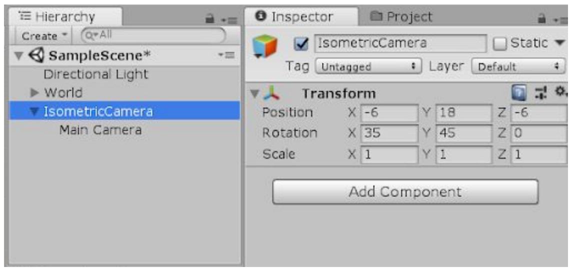
An isometric camera is in its simplest form a camera object that is directed to the center, with the angle between all axis being 120°.
In essence we start off with a camera that is aligned parallel to the floor (the x-axis in this case) orthogonal to the axis (the y-axis). Then the camera is rotated around the horizontal axis for precisely $arcsin\ 1/\sqrt{3}$ which can be approximated as 35°. Next we rotate the camera around the vertical axis for 45°.

In order to easily adjust the perspective and still be able to position the camera to the scene, create an empty game object and set the camera as it child. Be sure to set all the position and rotation values to 0, the camera size to 10 and the "Projection" type to Orthographic:



Main Camera
Tag Untagged Layer Default
Transform
Position X 0 Y 0 Z 0
Rotation X 0 Y 0 Z 0
Scale X 1 Y 1 Z 1
Camera
Clear Flags Solid Color
Background
Culling Mask Everything
Projection Orthographic
Size 10
Clipping Planes Near 0.3

Rather than directly rotating the camera object, we can now manipulate the transform parameters of its parent. Set the position to {-6 , 18 , -6} and the rotation to { 35 , 45 , 0 } :



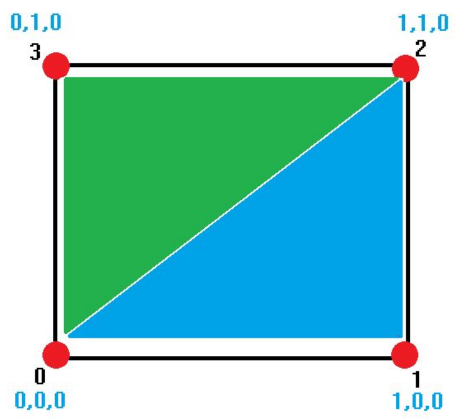
IsometricCamera
Tag Untagged Layer Default
Transform
Position X -6 Y 18 Z -6
Rotation X 35 Y 45 Z 0
Scale X 1 Y 1 Z 1

3. Voxels and Voxel World

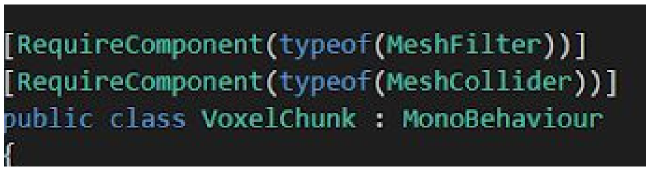
Before diving into the definition of voxels and how to use them, I would like to discuss on why they are almost mandatory in bigger projects. The best known example of a voxel game would be Minecraft. Intuitively when building such a world, one could place cube objects and stack them on top of each other to achieve the same effect. This would only work to a certain amount however, since every cube is a unique object with a unique drawing call, update loop etc. This will not work for larger projects and will cause Unity to stop working. This is where voxels come into play.

A voxel object is in essence not a gameobject, but a collection of vertices, triangles and uvFaces. In a way, they are just directions to tell your game where to draw a certain face. Let's consider the simplest voxel element : a minecraft cube.

The vertices of this cube are its corners (the red dots). Then the triangles are determined by those vertices : the blue triangle { 0 , 1 , 3 } and the green triangle { 1 , 2 , 3 }. This means that each one of the 6 sides of a cube has 2 of them defined by the corners of a square.

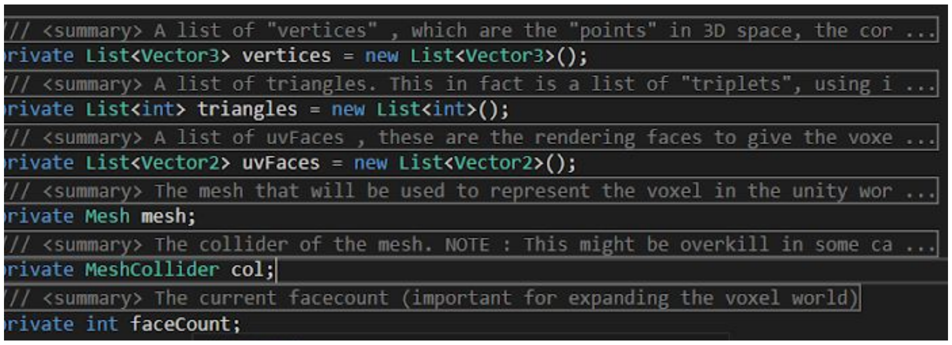


Now let's have a look at how we can achieve this data structure in code. First off, start by making a VoxelChunk monobehavior script. It should always have access to both a meshfilter and a meshcollider. We can enforce this by using the "**RequireComponent**" annotation. This will make sure that when attaching this script to a gameobject, Unity will automatically add both a MeshFilter and a MeshCollider component, should they be missing.



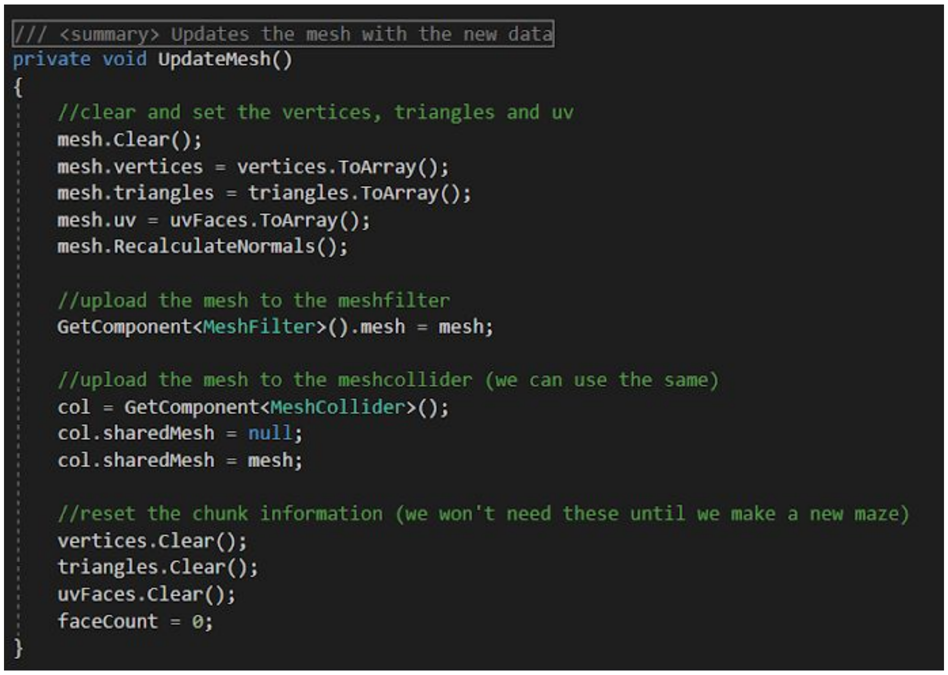
```
[RequireComponent(typeof(MeshFilter))]
[RequireComponent(typeof(MeshCollider))]
public class VoxelChunk : MonoBehaviour
{
```

Next we need to declare some variables we will need to construct a voxel and manipulate its mesh into a maze.
***Note** : The code is documented, so please refer to the github page for a full and extensive explanation of what the variables are for.*



```
// <summary> A list of "vertices" , which are the "points" in 3D space, the cor ...
rivate List<Vector3> vertices = new List<Vector3>();
// <summary> A list of triangles. This in fact is a list of "triplets", using i ...
rivate List<int> triangles = new List<int>();
// <summary> A list of uvFaces , these are the rendering faces to give the voxe ...
rivate List<Vector2> uvFaces = new List<Vector2>();
// <summary> The mesh that will be used to represent the voxel in the unity wor ...
rivate Mesh mesh;
// <summary> The collider of the mesh. NOTE : This might be overkill in some ca ...
rivate MeshCollider col;
// <summary> The current facecount (important for expanding the voxel world)
rivate int faceCount;
```

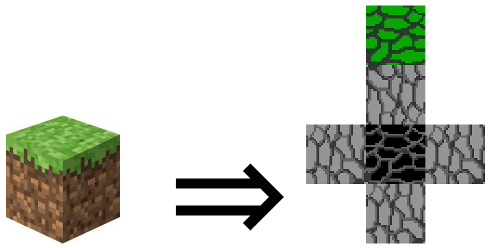
Before the actual construction of a voxel maze, let's create a method that will "upload" into the mesh object and the collider. This needs to happen to override the default that is in there.



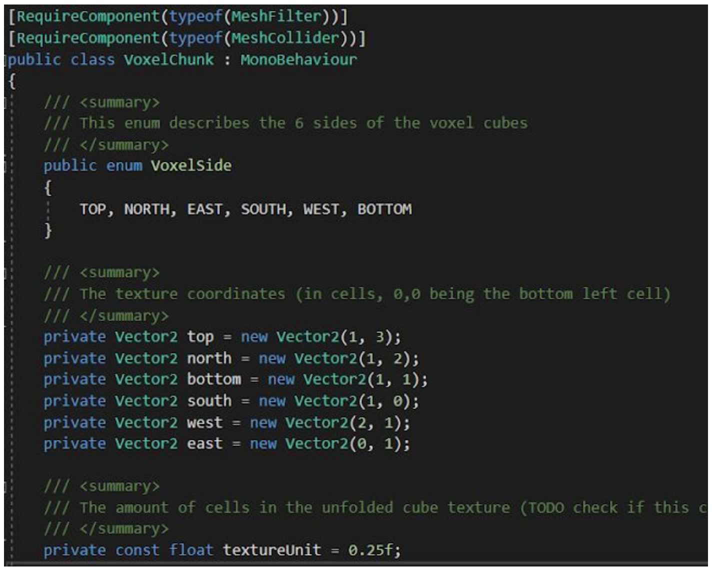
```
/// <summary> Updates the mesh with the new data
private void UpdateMesh()
{
    //clear and set the vertices, triangles and uv
    mesh.Clear();
    mesh.vertices = vertices.ToArray();
    mesh.triangles = triangles.ToArray();
    mesh.uv = uvFaces.ToArray();
    mesh.RecalculateNormals();

    //upload the mesh to the meshfilter
    GetComponent<MeshFilter>().mesh = mesh;

    //upload the mesh to the meshcollider (we can use the same)
    col = GetComponent<MeshCollider>();
    col.sharedMesh = null;
    col.sharedMesh = mesh;

    //reset the chunk information (we won't need these until we make a new maze)
    vertices.Clear();
    triangles.Clear();
    uvFaces.Clear();
    faceCount = 0;
}
```

Note that we initially clear the existing mesh. This is mainly when we want to update the existing maze, rather than creating a new one every time. This is usually a lot more efficient, especially for large worlds and chunks. Next, the vertices, triangles and UV maps. The latter is intended to assign a texture to a voxel face. The normals are then also recalculated to ensure the direction of the texture is correct. This new mesh data is then loaded into the MeshFilter as well as the MeshCollider. At the end of the method we clear everything as we will not be needing it any further until a new maze is generated. **IMPORTANT**: the face count (the total amount of faces in the voxel object) is also reset here. The code will not work if this is not performed.

When inspecting a minecraft cube, you can distinguish the top of the cube from the rest. For example, there might be grass on top.
To achieve this , we need a full texture and we need to tell the Voxel what texture it should be applying to a face. In other words, think of it as an unfolded cube :

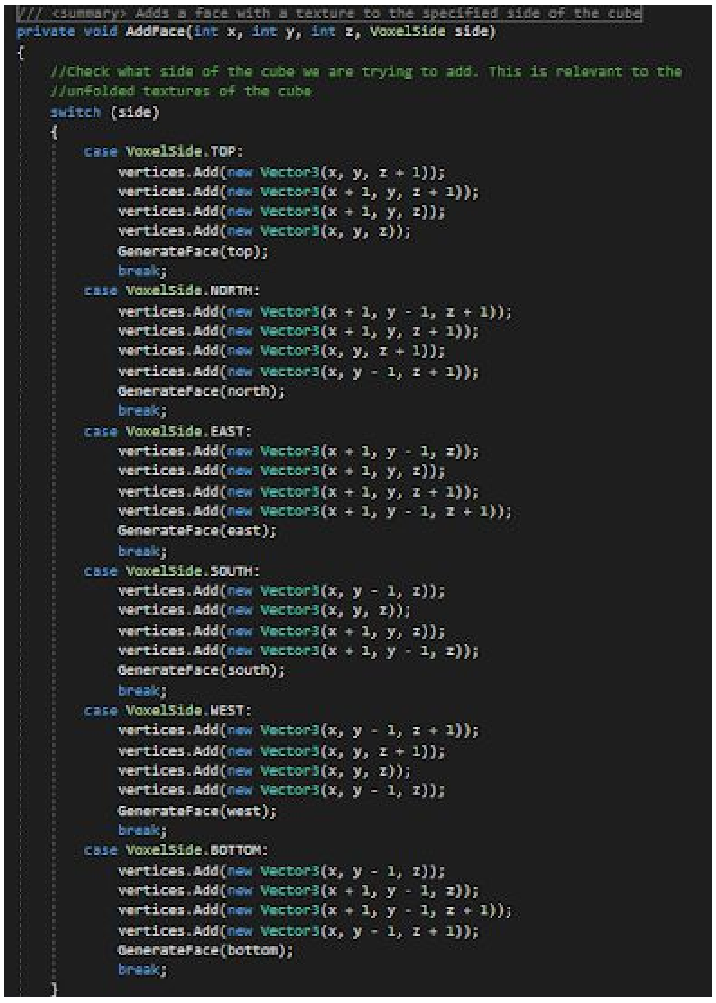


To simplify this we use a mapping, so we can identify a side of a cube and attribute a piece of the texture (unfolded cube) to it. I chose to do this using a VoxelSide enum. The coordinates (top, north, bottom, south, west, east) are the tile coordinates in the textures, starting from the bottom left (0,0). Since textures are more efficient when being in powers of 2, I decided to add an empty column of tiles to the left of the texture. The textureUnit variable reflects the relative size of a tile to the total texture size. In this case we have 4 tiles per row, so the textureUnit is ¼ or 0.25f.



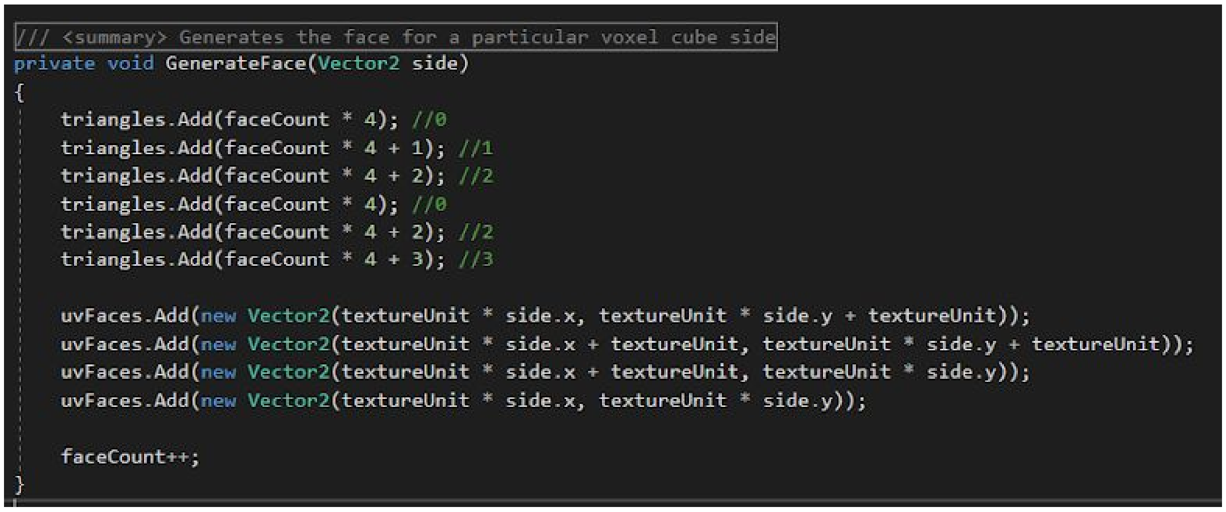
```
[RequireComponent(typeof(MeshFilter))]
[RequireComponent(typeof(MeshCollider))]
public class VoxelChunk : MonoBehaviour
{
    /// <summary>
    /// This enum describes the 6 sides of the voxel cubes
    /// </summary>
    public enum VoxelSide
    {
        TOP, NORTH, EAST, SOUTH, WEST, BOTTOM
    }

    /// <summary>
    /// The texture coordinates (in cells, 0,0 being the bottom left cell)
    /// </summary>
    private Vector2 top = new Vector2(1, 3);
    private Vector2 north = new Vector2(1, 2);
    private Vector2 bottom = new Vector2(1, 1);
    private Vector2 south = new Vector2(1, 0);
    private Vector2 west = new Vector2(2, 1);
    private Vector2 east = new Vector2(0, 1);

    /// <summary>
    /// The amount of cells in the unfolded cube texture (TODO check if this c
    /// </summary>
    private const float textureUnit = 0.25f;
```

Next we need a method to add a face to the voxel. Reminder, a face here is a “side” of a cube. To do this I’ve set up a AddFace convenience method. This method needs the x, y and z coordinates of where your cube will be on the grid AND a VoxelSide. Technically to generate a simple cube voxel, you could just call this method for all the VoxelSide values. This method handles where the vertices/triangles should be on a 3D grid and what uv textures should be applied.



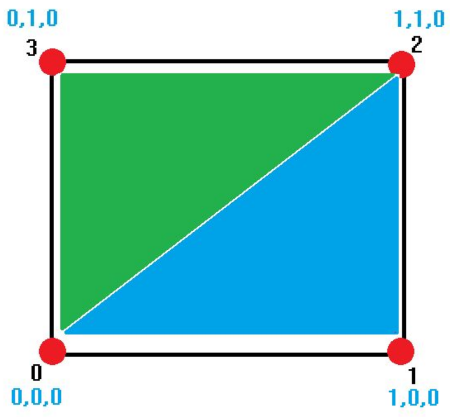
```
/// <summary> Adds a face with a texture to the specified side of the cube
private void AddFace(int x, int y, int z, VoxelSide side)
{
    //Check what side of the cube we are trying to add. This is relevant to the
    //unfolded textures of the cube
    switch (side)
    {
        case VoxelSide.TOP:
            vertices.Add(new Vector3(x, y, z + 1));
            vertices.Add(new Vector3(x + 1, y, z + 1));
            vertices.Add(new Vector3(x + 1, y, z));
            vertices.Add(new Vector3(x, y, z));
            GenerateFace(top);
            break;
        case VoxelSide.NORTH:
            vertices.Add(new Vector3(x + 1, y - 1, z + 1));
            vertices.Add(new Vector3(x + 1, y, z + 1));
            vertices.Add(new Vector3(x, y, z + 1));
            vertices.Add(new Vector3(x, y - 1, z + 1));
            GenerateFace(north);
            break;
        case VoxelSide.EAST:
            vertices.Add(new Vector3(x + 1, y - 1, z));
            vertices.Add(new Vector3(x + 1, y, z));
            vertices.Add(new Vector3(x + 1, y, z + 1));
            vertices.Add(new Vector3(x + 1, y - 1, z + 1));
            GenerateFace(east);
            break;
        case VoxelSide.SOUTH:
            vertices.Add(new Vector3(x, y - 1, z));
            vertices.Add(new Vector3(x, y, z));
            vertices.Add(new Vector3(x + 1, y, z));
            vertices.Add(new Vector3(x + 1, y - 1, z));
            GenerateFace(south);
            break;
        case VoxelSide.WEST:
            vertices.Add(new Vector3(x, y - 1, z + 1));
            vertices.Add(new Vector3(x, y, z + 1));
            vertices.Add(new Vector3(x, y, z));
            vertices.Add(new Vector3(x, y - 1, z));
            GenerateFace(west);
            break;
        case VoxelSide.BOTTOM:
            vertices.Add(new Vector3(x, y - 1, z));
            vertices.Add(new Vector3(x + 1, y - 1, z));
            vertices.Add(new Vector3(x + 1, y - 1, z + 1));
            vertices.Add(new Vector3(x, y - 1, z + 1));
            GenerateFace(bottom);
            break;
    }
```

Note that there is a GenerateFace method that returns for every side. This is simply because this code will stay the same for every side, as its task is to add the corresponding triangles to the newly added vertices. This method also accepts the texture coordinates discussed higher up.



```
/// <summary> Generates the face for a particular voxel cube side
private void GenerateFace(Vector2 side)
{
    triangles.Add(faceCount * 4); //0
    triangles.Add(faceCount * 4 + 1); //1
    triangles.Add(faceCount * 4 + 2); //2
    triangles.Add(faceCount * 4); //0
    triangles.Add(faceCount * 4 + 2); //2
    triangles.Add(faceCount * 4 + 3); //3

    uvFaces.Add(new Vector2(textureUnit * side.x, textureUnit * side.y + textureUnit));
    uvFaces.Add(new Vector2(textureUnit * side.x + textureUnit, textureUnit * side.y + textureUnit));
    uvFaces.Add(new Vector2(textureUnit * side.x + textureUnit, textureUnit * side.y));
    uvFaces.Add(new Vector2(textureUnit * side.x, textureUnit * side.y));

    faceCount++;
}
```

Let's have a closer look at this method, since it's probably the most abstract part of the code. The first thing it does is populate the triangles list. Remember that the vertices are used to construct the triangle list. In other words the triangle list does not contain vertice points, but it does contain a list of indices of vertice points in the list. To simplify it, let's consider the following image again : *Note that the z-values here are 0, so this is the front side of the cube.*



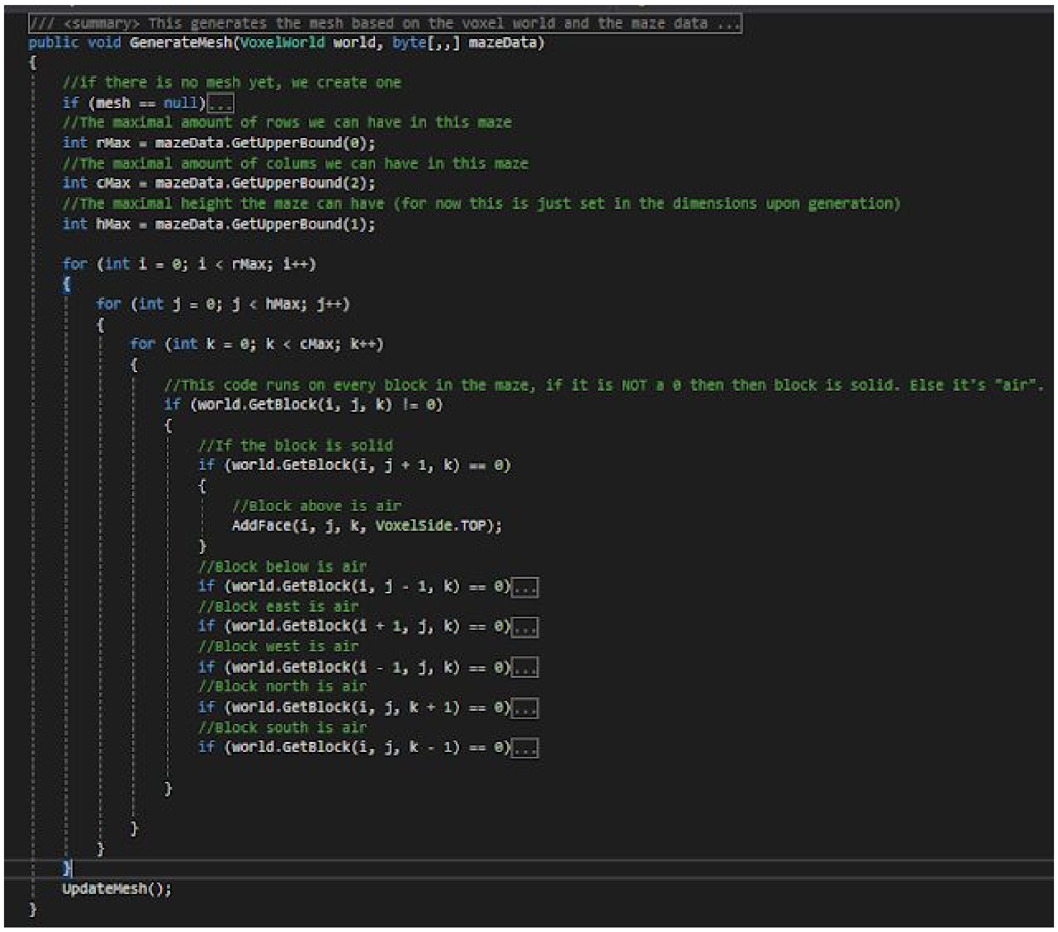
Here, the vertices would be the coordinates in blue, stored in a list of Vector3 objects. The triangle list does not contain the actual Vector3 coordinates, but it contains triplets (the 3 corners of a triangle) of indices that can be used internally to look up the Vector3 coordinates. Simply put, the triangle list contains the black number index of the vertices used. A single “quad” or face thus would add 6 integers to the triangle list.

The next thing the GenerateFace method does is assign the proper texture in the uv mapping for the mesh. Without going in too much detail, it adds a reference to the position on the texture where this side is located, so it only draws a portion of the total texture. One could theoretically put all the sides of a cube into their own files, but this would cause a major overload. From a technical standpoint, it is a lot more efficient to make a single draw call to load the image into the GPU and draw a section of a texture rather than loading yet another image.

Lastly we increase the facecount, as this is required for the composition of the entire mesh. We are adding cube faces onto a bigger chunk.

So far so good… However at this point we are still causing a lot of overhead. Consider what we have done so far. For every cube we theoretically will add 6 faces, but only a very minor part of those are actually required. We only really need the outer “shell” of the the mesh to be rendered. Only when this is achieved will we be able to harness the power and efficiency of a voxel based world !

Exactly this is handled in the GenerateMaze method. Basically we iterate every point in our voxel chunk (or every “cube”) and we check what is adjecent to it. The data grid (eventually will be generate by a maze algorithm) will contain a byte indicating the type of a block. In case the type is 0, it will be considered “empty” or “clear”. Reflecting on minecraft, you only see blocks (even more specifically, sides of blocks) that are being touched by air. Similarly, we only add the faces (using the aforementioned methods) if they are touching an empty spot.



```
/// <summary> This generates the mesh based on the voxel world and the maze data ...
public void GenerateMesh(VoxelWorld world, byte[,,] mazeData)
{
    //if there is no mesh yet, we create one
    if (mesh == null)...
    //The maximal amount of rows we can have in this maze
    int rMax = mazeData.GetUpperBound(0);
    //The maximal amount of colums we can have in this maze
    int cMax = mazeData.GetUpperBound(2);
    //The maximal height the maze can have (for now this is just set in the dimensions upon generation)
    int hMax = mazeData.GetUpperBound(1);

    for (int i = 0; i < rMax; i++)
    {
        for (int j = 0; j < hMax; j++)
        {
            for (int k = 0; k < cMax; k++)
            {
                //This code runs on every block in the maze, if it is NOT a 0 then then block is solid. Else it's "air".
                if (world.GetBlock(i, j, k) != 0)
                {
                    //If the block is solid
                    if (world.GetBlock(i, j + 1, k) == 0)
                    {
                        //Block above is air
                        AddFace(i, j, k, VoxelSide.TOP);
                    }
                    //Block below is air
                    if (world.GetBlock(i, j - 1, k) == 0)...
                    //Block east is air
                    if (world.GetBlock(i + 1, j, k) == 0)...
                    //Block west is air
                    if (world.GetBlock(i - 1, j, k) == 0)...
                    //Block north is air
                    if (world.GetBlock(i, j, k + 1) == 0)...
                    //Block south is air
                    if (world.GetBlock(i, j, k - 1) == 0)...

                }

            }
        }
    }
    UpdateMesh();
}
```

Note: I use a convenience method in the VoxelWorld class to check whether the coordinates are a. In bound and b. have a byte value (or block type) in the 3D byte array.

4. Maze generation algorithm

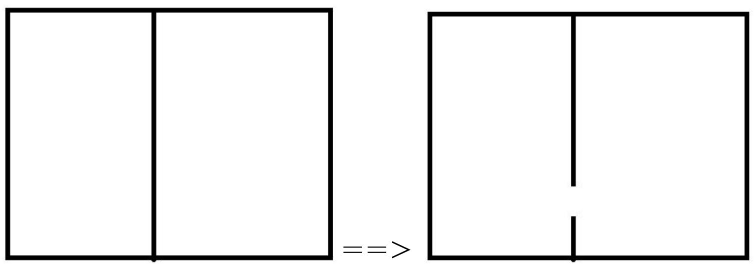
Now we have the means to render and use a 3D grid of data into a voxel mesh. You can experiment with this by just providing the data for a single cube voxel and it should render it just fine. But a single block does not really make for an interesting world / maze. So in order to form this, we need a proper datastructure.

For this tutorial I have opted for a recursive division algorithm to generate a perfect maze. You will also find a Prim's version, but I do recommend you use the conceptually simpler recursive division. Note that both of these are implentations of the abstract "**VoxelMazeGenerator**" class and are not MonoBehaviors. Again, here I extensively documented the code, and I will stick to briefly explaining in pseudocode how the algorithm works.

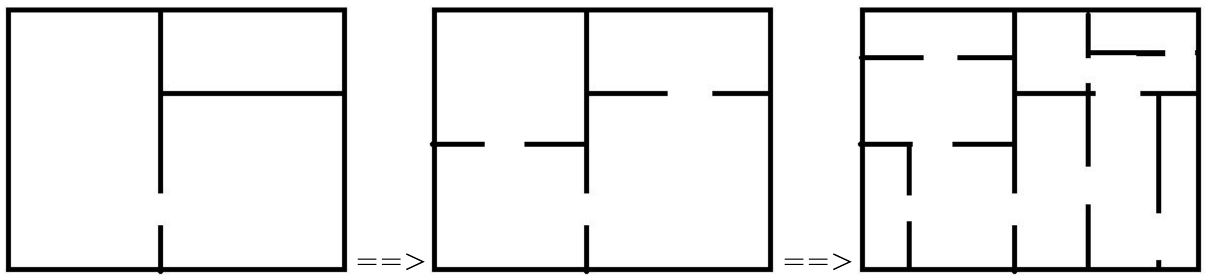
Step 1 : Create a grid of datapoints

Step 2 : Set up the walls surrounding the grid

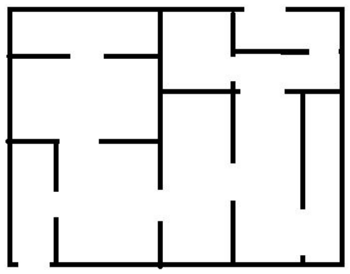
Step 3 : Randomly pick a point and divide the grid in 2 sections and create a random passegeway



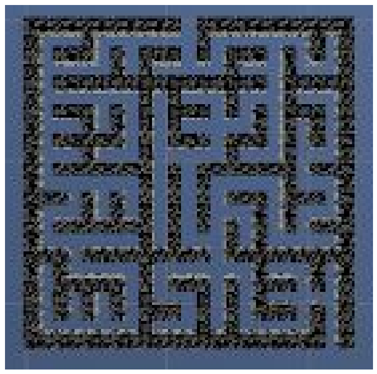
Step 4 : For each of the generated new “subfields” repeat the procedure until there is no more room for new “divisions” (that being the size of the sides of a sub-field should be larger than 2). **Note :** The best results are obtained when the division happens on the long side of the “rectangle”. For example if the rectangle is wider than tall, cut it up vertically and vice versa.



Step 5 : Create 2 random openings on opposite sides. This dungeon then inherently only has a single solution !

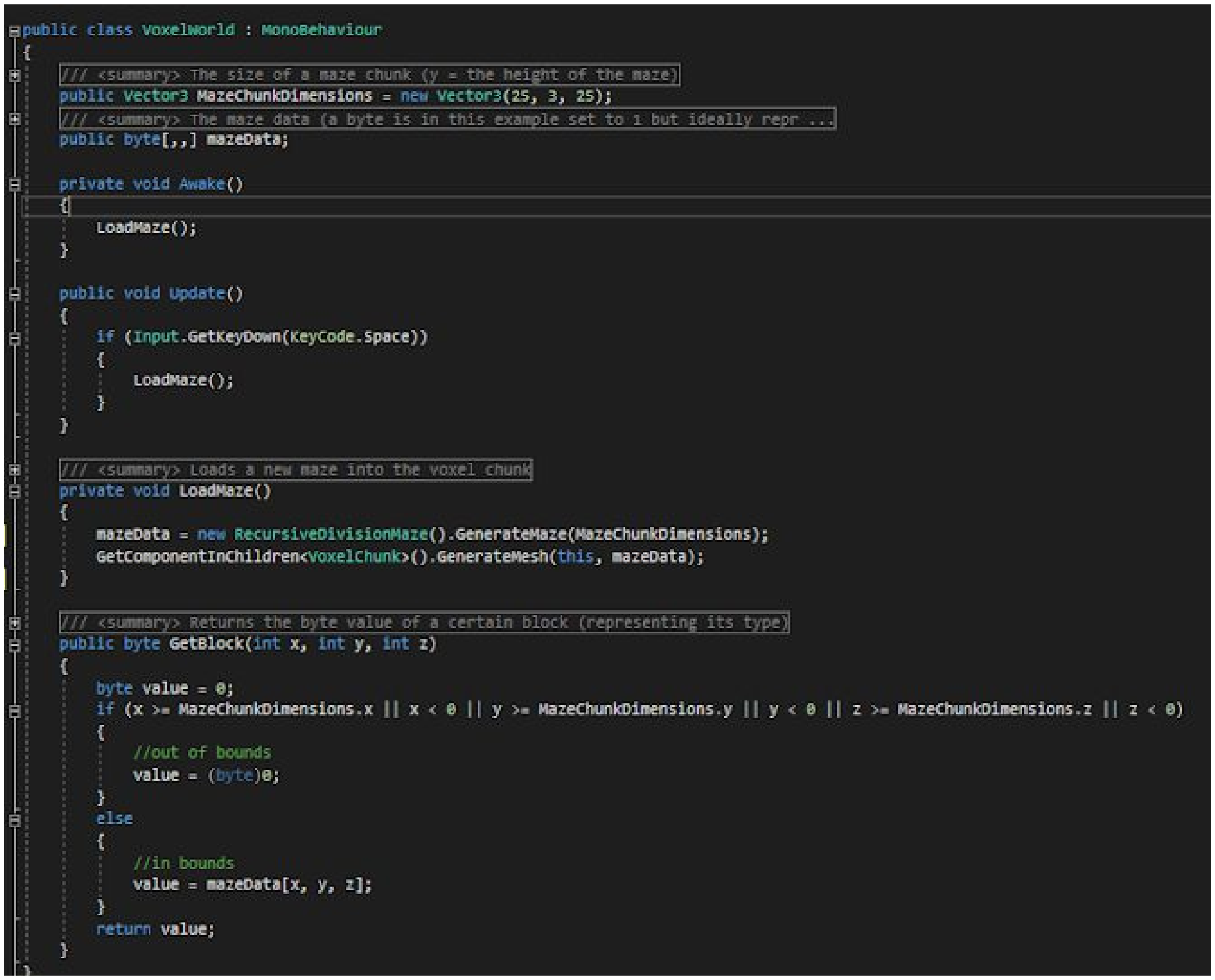


The resulting code leads to this (top down view) :



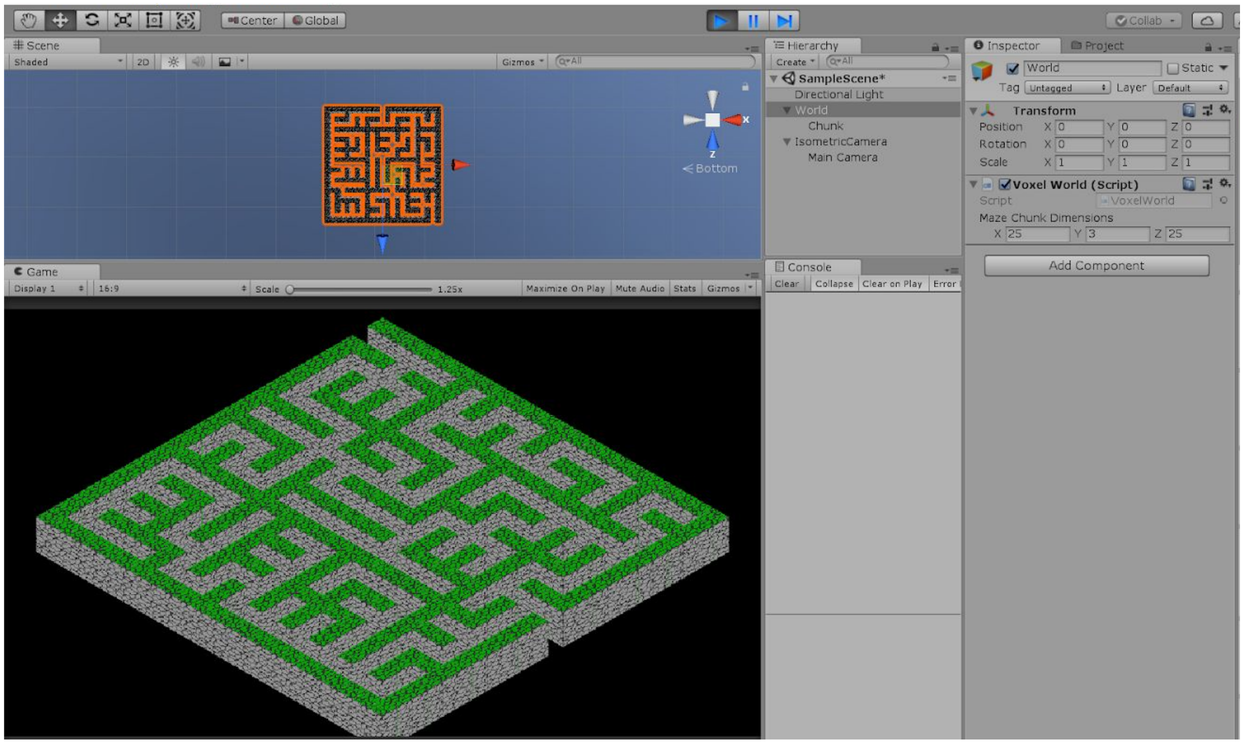
5. Setting it all up in Unity

To use all of the materials and scripts discussed above we need a central point. A “Voxel world manager” so to speak :



```
public class VoxelWorld : MonoBehaviour
{
    /// <summary> The size of a maze chunk (y = the height of the maze)
    public Vector3 MazeChunkDimensions = new Vector3(25, 3, 25);
    /// <summary> The maze data (a byte is in this example set to 1 but ideally repr ...
    public byte[,,] mazeData;

    private void Awake()
    {
        LoadMaze();
    }

    public void Update()
    {
        if (Input.GetKeyDown(KeyCode.Space))
        {
            LoadMaze();
        }
    }

    /// <summary> Loads a new maze into the voxel chunk
    private void LoadMaze()
    {
        mazeData = new RecursiveDivisionMaze().GenerateMaze(MazeChunkDimensions);
        GetComponentInChildren<VoxelChunk>().GenerateMesh(this, mazeData);
    }

    /// <summary> Returns the byte value of a certain block (representing its type)
    public byte GetBlock(int x, int y, int z)
    {
        byte value = 0;
        if (x >= MazeChunkDimensions.x || x < 0 || y >= MazeChunkDimensions.y || y < 0 || z >= MazeChunkDimensions.z || z < 0)
        {
            //out of bounds
            value = (byte)0;
        }
        else
        {
            //in bounds
            value = mazeData[x, y, z];
        }
        return value;
    }
}
```

This class is very simple, it listens for a spacebar down press and then reloads data into the mesh. Here I added the VoxelChunk object as a child of the VoxelWorld object (See example scene on GitHub and screenshot below). The only method I haven’t discussed but used is the “GetBlock” method. This is merely a convenience method to check whether a block is in bounds and if that’s the case, return its value in the 3D byte array. It was used earlier when we were checking what faces had to be added into the mesh.



World Static
Tag Untagged Layer Default

Transform
Position X 0 Y 0 Z 0
Rotation X 0 Y 0 Z 0
Scale X 1 Y 1 Z 1

Voxel World (Script)
Script VoxelWorld
Maze Chunk Dimensions
X 25 Y 3 Z 25