**World Generation**

World generations is handled by GenerationScriptV2.cs, which runs at the loading of the world scene.

**Initial World Generation**

The first step takes the entered world height and width, and along with a randomised seed generates a smooth undulating terrain.

|  |
| --- |
| for (int x = 0; y < worldWidth; x++)  {  perlinNoiseSoil = Mathf.RoundToInt(Mathf.PerlinNoise(x / Smoothness, Seed) \* height / 5);  perlinNoiseSoil += height / 3;  //Debug.Log($"Soil Noise Value is {perlinNoiseSoil}");  for (int y = 0; y < perlinNoiseSoil; y++)  {  WorldMap[x, y] = 1;  }  } |

A temporary value perlineNoiseSoil is set using Mathf.PerlineNoise() to get a smoothly increasing/decreasing value for y, then using a second for loop all values of y in the 2D map[] array are set to 1. This is repeated, however with \* height/8 and += height / 4, to produce smaller y values, which all values below are then set to 2 in the map array. 1 is the value the map renderer checks for grass or soil and 2 is the value for stone.

**Adding Caves**

Caves are added using cellular automata, to start a separate 2D array cavemap[] is create with the same width and height as the world map. Running in a nested loop to cover each element, cavemap is randomly filled using a percentage fill given by the user. This is done in the nested loop by creating a random value between 0 and 100 and then comparing it to the fill percentage, if the random value is higher cavemap[x,y] is set to 1, lower cavemap[x,y] is set to 0.

|  |
| --- |
| cavemap = new int[worldWidth, worldHeight];  for (int x = 0; x < worldWidth; x++)  {  for (int y = 0; y < worldHeight; y++)  {  cavemap[x, y] = UnityEngine.Random.Range(0, 100) < RandomPercentFill ? 1 : 0;  }  } |

The randomly filled 2D array cavemap[] is then passed to a cellular automata function, which using a nested loop runs through each element checking for the number of neighbouring elements with value 1 using GetNeighbours() function, if neighbours is >4 the value of the element is set to 1, else 0. The automata is run multiple times to alter the size and length of caves.

|  |
| --- |
| private int GetNeighbours(int[,] tempmap, int xs, int ys, int neighbours)  {  for (int x = xs - 1; x <= xs + 1; x++)  {  for (int y = ys - 1; y <= ys + 1; y++)  {  if (y <= worldHeight && x <= worldWidth)  {  if (y != ys || x != xs)  {  if (tempmap[x, y] == 1)  {  neighbours++;  }  }  }  }  }  return neighbours;  } |

The cavemap[] and overall map[] arrays are then compared in nested loops, and if cavemap[x,y] is 1 and x and y are not on the edge of the map, the value within map[] will be set to 8/9. 8 is rendered as a background soil tile, 9 is rendered as a stone background.

**Adding Ores**

|  |
| --- |
| float OrePL = (Mathf.PerlinNoise((float)x \* OreSeed, (float)y \* OreSeed));    if (OrePL <= OreThreshhold)  {  map[x, y] = 3;  TestTileFG.SetTile(new Vector3Int(x, y, 0), Ore);  } |

In the same nested loop as the initial generation, Ore distribution is done. This is done by generating a float between 0 and 1 based on the Mathf.PerlinNoise() function. This is then compared to a threshold value and added to the map[]. This leads to small clusters of ores.

**Adding Trees**

Trees are added by finding a given number of positions on the surface of the world where a tree can be drawn without collisions. This is done in a for loop for the number of trees wanted. Collision avoidance is done by generating random integers between 0 and the world width, this is then checked against all elements in an array of past position. The check is done by checking if Mathf.Abs(proposedPosition - positionHistory[j]) < 6).

|  |
| --- |
| for (int i = 0; i < density; i++)  {  bool collisionAvoidance = true;  while (collisionAvoidance)  {  int holdingvariable = UnityEngine.Random.Range(2, worldWidth - 2);  collisionAvoidance = false;  for (int j = 0; j <= i; j++)  {  if (Mathf.Abs(holdingvariable - positionHistory[j]) < 6)  {  Debug.Log(Mathf.Abs(holdingvariable - positionHistory[j]));  collisionAvoidance = true;  }  }  if (!collisionAvoidance)  {  tempx = holdingvariable;  }  }  positionHistory[i] = tempx;  { |

Once a position is cleared for collision, a while loop is used to find the height of the floor below the given x position. This is done be decrementing a value of y while (map[xvalue,tempy] != 0). Trees are then drawn at the position found then the loop repeats.

**Main Function**

|  |
| --- |
| public void OptimisedTerrainGeneration(int[,] WorldMap, int width, int height, TileBase GrassSoil, TileBase Stone)  {  int perlinNoiseSoil;  int perlinNoiseStone;  for (int x = 0; x < width; x++)  {  //Making the Soil  perlinNoiseSoil = Mathf.RoundToInt(Mathf.PerlinNoise(x / Smoothness, Seed) \* height / 5);  perlinNoiseSoil += height / 3;  //Debug.Log($"Soil Noise Value is {perlinNoiseSoil}");  for (int y = 0; y < perlinNoiseSoil; y++)  {  WorldMap[x, y] = 1;  }  //Making the Stone  perlinNoiseStone = Mathf.RoundToInt(Mathf.PerlinNoise(x / (Smoothness \* 2), StoneSeed) \* height / 8);  perlinNoiseStone += height / 4;  //Debug.Log($"Stone Noise Value is {perlinNoiseStone}");  for (int y = 0; y < perlinNoiseStone; y++)  {  WorldMap[x, y] = 2;  }    //Cutting Cacves out and Rendering to Optimise the code using the same for loops  for (int y = 0; y < perlinNoiseSoil; y++)  {  float OrePL = (Mathf.PerlinNoise((float)x \* OreSeed, (float)y \* OreSeed));    if (OrePL <= OreThreshhold)  {  map[x, y] = 3;  TestTileFG.SetTile(new Vector3Int(x, y, 0), Ore);  }  }    }  } |

This is the overall function that create the initial soil and stone, as described above. This takes in the Foreground and Background TileMaps, the 2D array that stores the world, and the two dimensions for the world.