Procedurell voxel terräng i Unity Med Hjälp Utav Perlin Noise

av: Christopher Vedlund, 2022-08-22

Utvärdering

Redogör för slutresult

Problemformulering

Terräng i spel kan genereras på många olika sätt. Det kan genereras som voxel terräng, mesh terräng eller mycket annat. Det jag vill göra med detta arbetet är att lära mig hur man genererar en mesh med hjälp av kod och därefter använder den meshen för att generera olika terrängtyper.

Det som jag vill lära mig genom detta arbetet är hur man kan återskapa Minecrafts terräng generering i Unity. Jag kommer inte att implementera olika biomer eller lägga till en avancerad algoritm som kollar vilka block som ska vara vart.

Alla färdiga script och filer som kommer att behövas finns längst ned i arbetet.

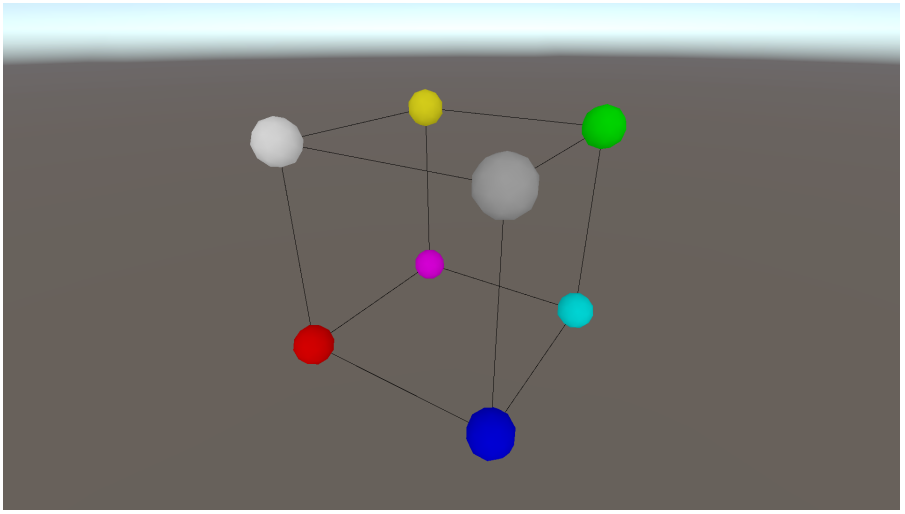
Arbetsprocess

Kod exemplerna är lite dåliga här i dokumentet, men alla finns längs ned och kan laddas ned.

Generera Perlin Noise

En mesh är en 3D-modell, ett rutnät av massa andra delar, till exempel av massa trianglar som kontrollerar hur meshens faces skall ritas mellan dess vertices. Dessa faces består därefter av tre vertices som är en punkt i världen. Så genom att manipulera dessa punkter så kan man ändra på en 3D-modell genom kod.

Genom att definiera tre punkter så kan vi skapa en egen face. Det gör vi genom att kolla på hur en kub är uppbyggd. Här nedan kommer en kub där dess hörn har fått en varsin koordinat i ett koordinatsystem.



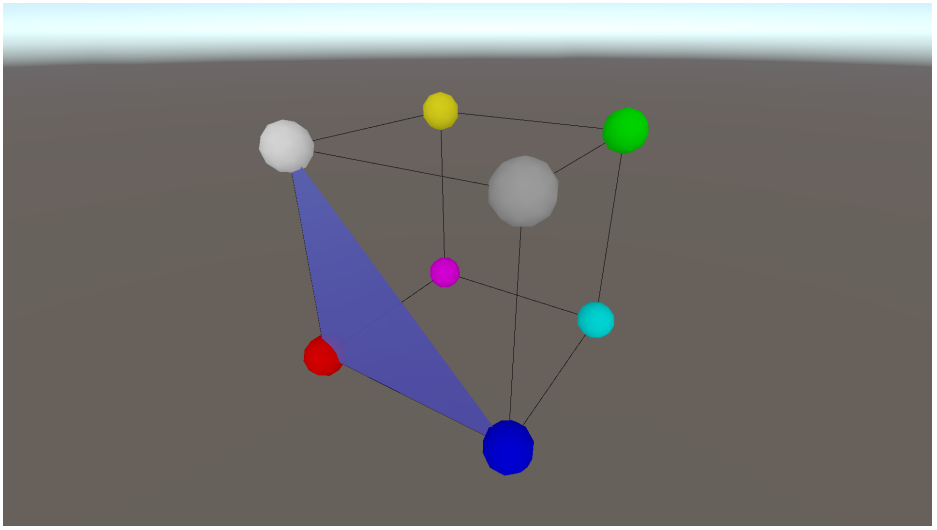
Färg	**Koordinat (X, Y, Z)**	**Index**
Blå	0, 0, 0	0
Röd	1, 0, 0	1
Lila	1, 0, 1	2
Ljusblå	0, 0, 1	3
Grå	0, 1, 0	4
Vit	1, 1, 0	5
Gul	1, 1, 1	6
Grön	0, 1, 1	7

Varje hörn har fått ett Index definierat, det är den som avgör hur en mesh blir ritad i Unity. För att skapa en mesh så måste vi nu skapa en array med varje punkt från kuben. Som du ser här nedan:

```
Vector3[] vertices = new Vector3[8]
  {
     new Vector3(0,0,0), // Index 0
     new Vector3(1,0,0), // Index 1
     new Vector3(1,0,1), // Index 2
     new Vector3(0,0,1), // Index 3
     new Vector3(0,1,0), // Index 4
     new Vector3(1,1,0), // Index 5
     new Vector3(1,1,1), // Index 6
     new Vector3(0,1,1)  // Index 7
  };
```

Denna arrayen kommer vi att använda för skapandet av våran kub. Genom att vi målar en triangel mellan punkterna 0, 1 & 5 så har vi nu fått en face. Skulle vi vända på ordningen av punkterna så hade facen bytt håll och endast blivit synlig från andra sidan.

Det vi har skapat nu är en triangel.



För att göra en kub så måste vi definiera fler trianglar. Det gör vi genom att göra en ny array där ordningen på hur varje triangel kommer att ritas. Så t.ex första facen kommer att definieras som:

0, 1, 5, 0, 5, 4

Trianglarna definieras som följande:

```
int[] triangles = new int[36]
    {
        0,1,5,0,5,4, // Front face
        3,0,4,3,4,7, // Right face
        1,2,6,1,6,5, // Left face
        4,5,6,4,6,7,  // Top face
        1,0,2,0,3,2, // Bottom face
        2,3,7,2,7,6  // Back face
    };
```

Nu har vi allt för att generera en kub med hjälp utav kod. Men vi har inget sätt att rendera kuben. Då använder vi oss utav två komponenter i unity Mesh Renderer & Mesh Filter. Så börja med att skapa en referens till dessa två komponenter i skriptet. Därefter i start funktionen så skriver du följande:

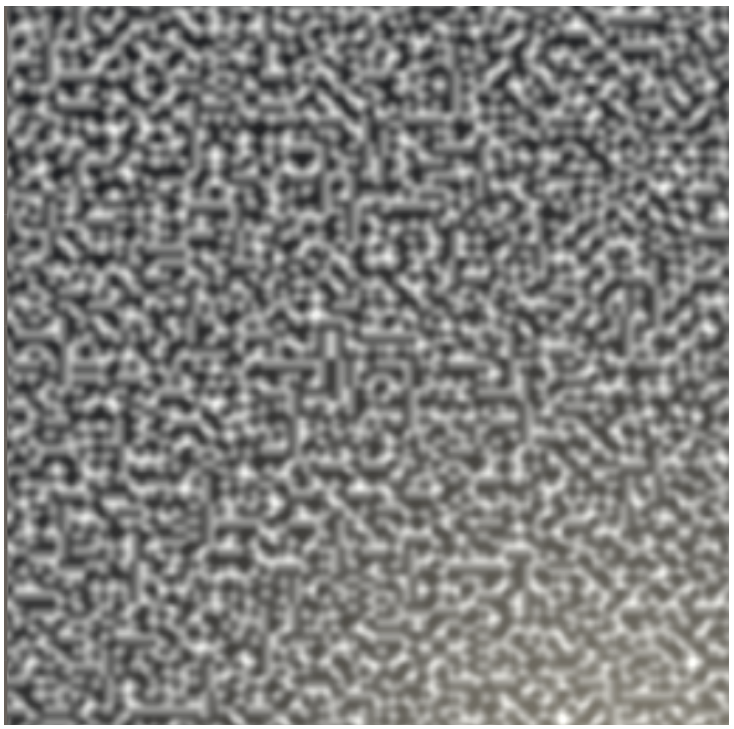
```
 private void Start()
{
    Mesh mesh = new Mesh();

    mesh.Clear();

    GenerateMesh();

    mesh.vertices = vertices;
    mesh.triangles = triangles;
    mesh.RecalculateNormals();
    mesh.RecalculateBounds();
    mesh.RecalculateTangents();
    mesh.Optimize();

    meshFilter.mesh = mesh;
}
```

Koden ovan lägger in all nödvändig data i meshen samt optimerar den.

Nu har du genererat en kub med kod. Koden kan utvecklas senare för att skapa mer procedurella saker. Jag t.ex skapade en world border som ändrar sin storlek beroende på hur stor världen är. Det var enklare att göra en world border på detta sättet då den ska skalas med storleken av terrenget.

Generera Perlin Noise

Perlin noise är en algoritm för att generera en så kallad “noise map” där man kan få världen mellan 0 & 1 där 0 är svart och 1 är vit. Bilden nedan visar hur ett färdig genererat perlin noise ser ut.



Unity har en färdig funktion i **Mathf** klassen för detta.

```
Mathf.PerlinNoise(X-Koordinat, Y-Koordinat);
```

Den ger dig ett värde i den precisa koordinaten som du angett. Så den behöver alltså loopas och kallas på flera gånger för att få något större såsom terräng.

Koden för att återskapa exemplet ovan:

```
private void Update()
    {
        // Create a new texture, the perlin noise will be projected onto this.
        Texture2D newTex = new Texture2D(256,256);

        for (int x = 0; x < height; x++)
        {
            for (int y = 0; y < height; y++)
            {
                float perlinVal = Mathf.PerlinNoise(x / width * scale, y / height * scale);
                newTex.SetPixel(x, y, new Color(perlinVal, perlinVal, perlinVal));
            }
        }
        newTex.Apply();

        text.mainTexture = newTex;
        meshRenderer.material = text;
    }
```

Det vi gör är att vi först skapar en textur där vi ska lägga in perlin noiset. Därefter så loopar vi igenom alla koordinater i X och Y led så att vi kan lägga in dess perlin värde i texturen.

Hur man får olika utseenden på perlin noise kan ibland vara lite svårt att förstå, men det är precis som en sinuskurva.

$f(X) = Amplitud \bullet sin((k \bullet Period) + förskjutning\ i\ X) + förskjutning\ i\ Y$
Amplitud är kurvans höjd medans Perioden är hur utspridd kurvan är.

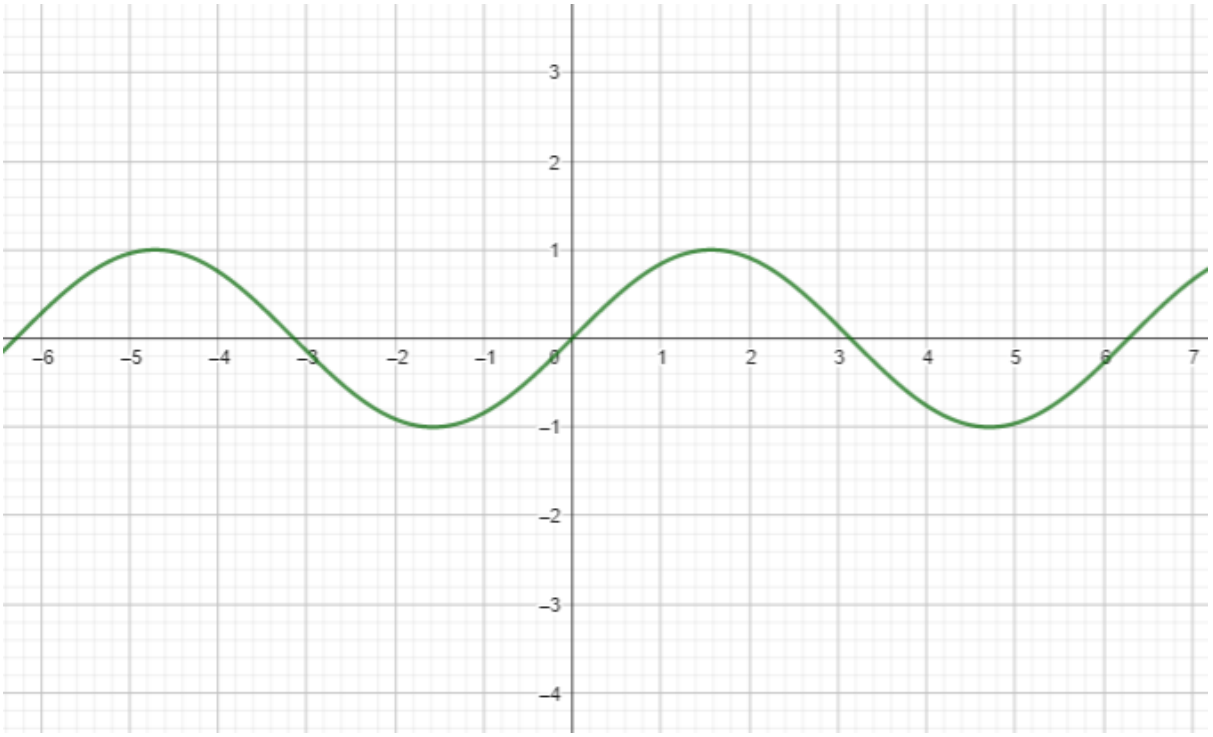
Mathf. metoden fungerar på ett liknande sätt. Om vi ska ta kod-exemplet ovan:
`float perlinVal = Mathf.PerlinNoise(x / width * scale, y / height * scale);`

Eftersom att perlin noise repeteras vid hela tal så vill vi försöka att hålla oss till decimaler. Såsom 0 till 1. Därför delar vi därför koordinaterna med bredden och höjden så när x är 256 så blir det 1. Därefter multiplicerar vi in skalan för att bestämma kurvans period.

Exemplet är gjort i Update metoden för att visa hur ljudet kan uppdateras och ändras. Detta rekommenderas inte, det är bättre att kalla på metoden en gång då terrenget bara kommer att genereras när scenen laddas in eller då spelet startas. De finns gånger då man kanske vill generera terrenget flera gånger, men då ska man aldrig göra det i Update metoden då den kallas på varje frame. Hela skriptet finns längst ner i arbetet.

Stapla Vågor

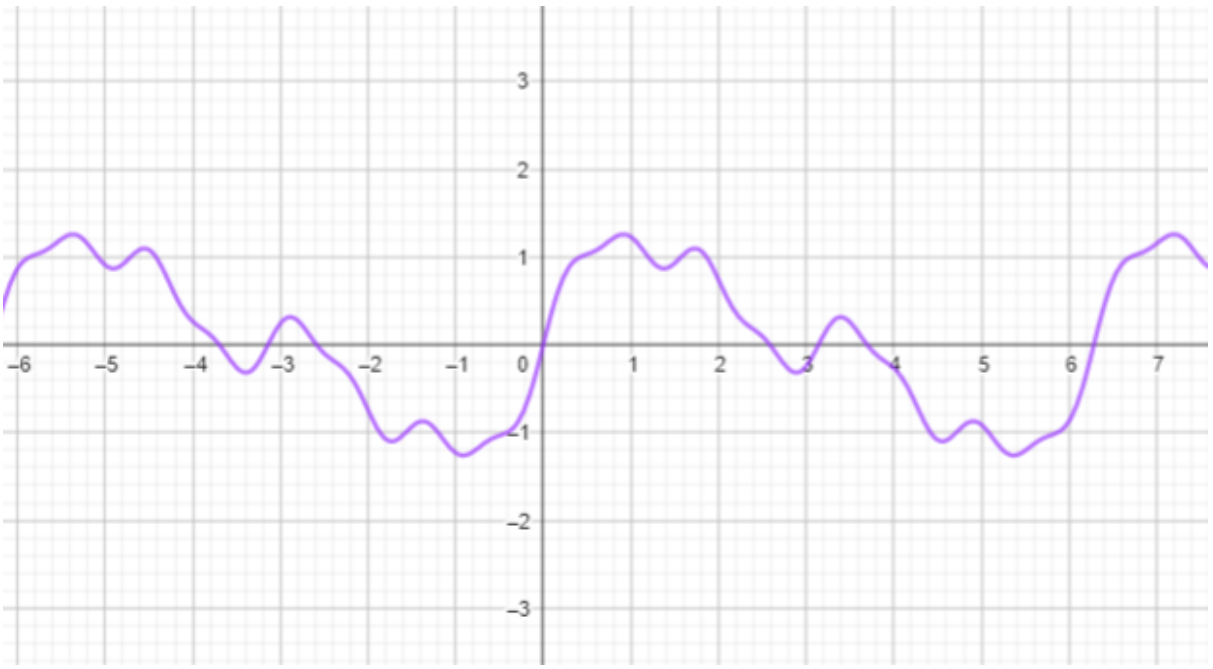
Man kan också stapla flera perlin noise värden på varandra. Man gör detta för att uppnå en mer detaljrikt terräng, det får terrenget att vara mer naturligt. När man gör detta så kan man använda en sinuskurva. En vanlig sinuskurva ser ut på detta sättet:



Ekvation till kurvan: $f(x) = \frac{sin(2^i \bullet x)}{2^i}$

Där k är 2 och i är 0. Vi kommer använda oss av denna formel när vi skapar detta exempel.

Och genom några fler sinuskurvor så kan man få ett väldigt intressant resultat. Resultatet kommer att skilja sig åt beroende på hur funktionerna ser ut. Men i detta exempel så kommer vi att använda oss av 4 sinusfunktioner. Enda nackdelen med att använda sinusfunktionen till världs generering är att de loopar. De kommer att följa samma mönster.



För att räkna ut denna linjen:

$$y_1 = \frac{sin(2^0 \bullet x)}{2^0}$$

$$y_2 = \frac{sin(2^1 \bullet x)}{2^1}$$

$$y_3 = \frac{sin(2^2 \bullet x)}{2^2}$$

$$y_4 = \frac{sin(2^3 \bullet x)}{2^3}$$

$$y(x) = y_1 + y_2 + y_3 + y_4$$

Slår du in dessa funktioner i t.ex geogebra så kommer du att få samma resultat som jag har fått. Och med ännu fler iterationer så blir terrenget endast mer detaljrik.

Du kan testa olika endimensionella vågor i itch.io länken som finns längst ned i arbetet bland de nedladdningsbara filerna.

Generera ett Mesh Terräng

Med hjälp av de kapitlen ovan så kommer ni nu att generera ett terräng med en mesh. Vi börjar med att definiera alla variabler som vi behöver ha.

```
private Mesh mesh;

private Vector3[] vertices;
private int[] triangles;

public int seed = 0;
public bool randomizeSeed = false;
public float frequenzy = 10f;
public float amplitude = 1f;

public int terrainWidth = 40;
public int terrainHeight = 40;
```

När vi har dessa variabler så är vi redo att programmera in resten av terräng-logiken. Vi börjar med att skapa en Start och Update funktion och lägger in följande kod:

```
private void Start()
{
      mesh = new Mesh();
      GetComponent<MeshFilter>().mesh = mesh;

      CreateShape();
      UpdateMesh();
}

private void Update()
{
      // CreateShape();
      // UpdateMesh();
}
```

Anledningen till att `UpdateMesh();` och `CreateShape();` finns i både Start och Update-metoden är de ska vanligtvis vara i start-metoden, men när man vill hitta bra värden för terrenget så är det bra att den uppdateras varje frame. Men det rekommenderas inte att ha med i det färdiga spelet, utan enbart för testning. För nu så behöver du inte bry dig om `UpdateMesh();` metoden, vi kommer att gå igenom den senare.

Nu så börjar vi att skapa formen som våran mesh kommer att ha. Vi kommer kalla den för `CreateShape();`.

I `CreateShape();` metoden ska vi börja med att programmera in terrängets vertice punkter genom att lägga in följande kod:

```
void CreateShape()
{
	vertices = new Vector3[(size.x + 1) * (size.y + 1)];

	for (int i = 0, z = 0; z <= size.y; z++)
	{
		for (int x = 0; x <= size.x; x++)
		{
			float y = Mathf.PerlinNoise((x + transform.position.x
+ seed) * frequency, (z + transform.position.z + seed) * frequency) *
amplitude;
			vertices[i] = new Vector3(x, y, z);
			i++;
		}
	}
}
```

Det vi gör här är att vi loopar igenom alla punkter på terrenget och genererar en höjd på det genom perlin noise. Nu när vi har definierat alla vertices av meshen så måste vi nu börja definiera i vilken ordning trianglarna ska ritas. Det koden gör är att den loopar igenom alla axlar och lägger till alla vertices som det ska målas trianglar mellan. Lägg till följande kod i samma metod som ovan:

```
	triangles = new int[size.x * size.y * 6];

	int vert = 0;
	int tris = 0;

	for (int z = 0; z < size.y; z++)
	{
		for (int x = 0; x < size.x; x++)
		{
			triangles[tris + 0] = vert + 0;
			triangles[tris + 1] = vert + size.x + 1;
			triangles[tris + 2] = vert + 1;
			triangles[tris + 3] = vert + 1;
			triangles[tris + 4] = vert + size.x + 1;
```

```
                triangles[tris + 5] = vert + size.x + 2;

                vert++;
                tris += 6;
            }
            vert++;
        }
```

Nu när vi har definierat alla vertices samt hur alla trianglar ska ritas så är det dags att lägga till UVs.

När vi har definierat alla värden till meshen så måste vi lägga till den datan i MeshFilter komponenten. Det gör vi genom att skapa en `UpdateMesh()` metod.

```
private void UpdateMesh()
{
      mesh.Clear();

      mesh.vertices = vertices;
      mesh.triangles = triangles;

      mesh.RecalculateNormals();
      mesh.RecalculateBounds();
      mesh.RecalculateTangents();
      mesh.Optimize();
}
```

Denna koden kommer att rensa den nuvarande meshen, ifall man har råkat lägga in en kub eller något, eller för att rensa sin tidigare generering. Meshen kommer att rensas automatiskt när man startar om spelet.

Därefter så lägger vi till värdena som vi har genererat tidigare. Därefter så beräknar vi meshens normal, tangent och bounds. Efter det så optimerar vi meshen för att förbättra renderings prestanda.

Generera Voxel Terräng

En voxel är en 3D pixel. För att skapa ett terräng av dem (likt Minecraft) så måste vi först definiera varje hörn av voxeln, dess trianglar samt dess normal. Det gör vi genom att skapa en struct där alla dessa definieras. Denna structen finns att ladda ned längst ned i arbetet. Eller kopiera in all kod i ett script som du döper till **VoxelMeshData**

Nu behöver vi också definiera lite data angående världen. T.ex hur stor den ska vara (både i kuber och chunks). Detta scriptet definierar världens storlek, den ska inte ändras under runtime. Kopiera koden från scriptet **WorldData** som finns längst ned i arbetet. En anledning till att jag skapar det som en egen klass är eftersom att jag i framtiden kan spara ned alla data i en fil. Men just nu så kommer vi inte att spara någonting, så därför kan vi göra en struct istället.

Om du väljer att döpa om dessa två script till något annat så är det viktigt att du ändrar i andra script också. Då de används genom allt.

När vi har skapat de två scripten så är det dags att börja generera världen. Jag börjar med att skapa ett **Chunk** script. Detta scriptet kommer att hantera alla block som finns i chunken, samt dess interna generering. Börja med att lägga in följande kod i scriptet.

```
using System.Collections.Generic;
using UnityEngine;

public class Chunk : MonoBehaviour
{
    private byte[,,] voxelMap = new byte[WorldData.ChunkWidthInVoxels,
WorldData.ChunkHeightInVoxels, WorldData.ChunkWidthInVoxels];
    private Vector3Int chunkPosition;

    private MeshFilter meshFilter;

    // The required variables to generate the mesh.
    private List<Vector3> verticies = new List<Vector3>();
    private List<int> triangles = new List<int>();
```

```
    private List<Vector2> uvs = new List<Vector2>();
    private int vertexIndex = 0;

    private void Awake() => meshFilter = GetComponent<MeshFilter>();

    private void Start()
    {
        // Get the chunk position.
        chunkPosition = Vector3Int.FloorToInt(transform.position /
WorldData.ChunkWidthInVoxels);

        // Generate the world.
        GenerateTerrain();
        GenerateVoxelMesh();
    }
/// <summary>
    /// Generates the terrain. (It will check what kind of block will
be in a certain position)
    /// </summary>
    public void GenerateTerrain()
    {

    }

    /// <summary>
    /// Generates the voxel mesh.
    /// </summary>
    private void GenerateVoxelMesh()
    {

    }
}
```

Det jag gör i detta scriptet är att jag gör i detta scriptet är att jag förbereder inför chunk-genereringen. **VoxelMap** är en array där alla voxels i världen kommer att sparas. Men de viktigaste funktionerna är helt tomma. Vilket gör att vi inte kan få något terräng.

Vi börjar med **GenerateTerrain** funktionen. I denna så kommer vi att endast loopa igenom alla punkter i världen och definera vilken typ av block som kommer att vara vart.

```
public void GenerateTerrain()
    {
```

```
        // Loop through every chunk position.
        for (int x = 0; x < WorldData.ChunkWidthInVoxels; x++)
        {
            for (int y = 0; y < WorldData.ChunkHeightInVoxels; y++)
            {
                for (int z = 0; z < WorldData.ChunkWidthInVoxels; z++)
                {
                    // Populate the blocks. Will check what kind of
voxel it will be from the GetVoxel method in the world.
                    voxelMap[x, y, z] = World.Instance.GetVoxel(new
Vector3Int(x, y, z) +
Vector3Int.FloorToInt(transform.position)).blockID;
                }
            }
        }
    }
```

Eftersom att vi inte har skapat ett **World** script än så kommer du att få ett error. Ignorera det för tillfället, du kan ju ändå inte generera något än.

När funktionen är klar så är det dags att generera själva meshen. Det gör genom att återigen loopa igenom varje punkt i världen.

```
private void GenerateVoxelMesh()
    {
        // Loop through every chunk position.
        for (int x = 0; x < WorldData.ChunkWidthInVoxels; x++)
        {
            for (int y = 0; y < WorldData.ChunkHeightInVoxels; y++)
            {
                for (int z = 0; z < WorldData.ChunkWidthInVoxels; z++)
                {
                    // If the block id is 0 (Air) then create a voxel.
                    if (voxelMap[x, y, z] != 0) CreateVoxel(new
Vector3Int(x, y, z));
                }
            }
        }

        // Add the mesh data to the mesh object.
        Mesh mesh = new Mesh();
        mesh.subMeshCount = 2;
        mesh.vertices = verticies.ToArray();
        mesh.SetTriangles(triangles.ToArray(), 0);
        // mesh.SetTriangles(subTriangles.ToArray(), 1);
```

```
        mesh.uv = uvs.ToArray();

        mesh.RecalculateNormals();

        // Add the mesh object to the chunk mesh filter.
        meshFilter.mesh = mesh;
    }
```

Nu skapar vi en ny funktion. **CreateVoxel** denna kommer att kolla ifall blocket har en face mot en annan, så att den inte skapar onödiga faces som man inte ser.

```
private void CreateVoxel(Vector3Int position)
    {
        // Loops through every voxel face.
        for (int face = 0; face < 6; face++)
        {
            // Calculate the neighboring positions.
            Vector3Int neighborPosition =
Vector3Int.FloorToInt(VoxelMeshData.normal[face] + position);
            Vector3Int worldNeighborPos = neighborPosition +
chunkPosition * WorldData.ChunkWidthInVoxels;


            if (World.Instance.IsVoxelInWorld(worldNeighborPos) &&
!World.Instance.GetVoxel(worldNeighborPos).isSolid)
            {
                // Calculate the vertices.
                verticies.Add(position +
VoxelMeshData.vertices[VoxelMeshData.triangles[face, 0]]);
                verticies.Add(position +
VoxelMeshData.vertices[VoxelMeshData.triangles[face, 1]]);
                verticies.Add(position +
VoxelMeshData.vertices[VoxelMeshData.triangles[face, 2]]);
                verticies.Add(position +
VoxelMeshData.vertices[VoxelMeshData.triangles[face, 3]]);


AddTexture(World.Instance.blockTypes[voxelMap[position.x, position.y,
position.z]].GetBlockTexture(face));

                // Calculate the triangles.
                triangles.Add(vertexIndex);
                triangles.Add(vertexIndex + 1);
                triangles.Add(vertexIndex + 2);
                triangles.Add(vertexIndex + 2);
```

```
                triangles.Add(vertexIndex + 1);
                triangles.Add(vertexIndex + 3);

                vertexIndex += 4;
            }
        }
    }
```

Det enda som blocket saknar nu är en textur. Därför skapar vi en **AddTexture** funktion där facens uv kommer att bli uträknad.

```
void AddTexture(int textureID)
    {
        float textureSizeInBlocks = 1f / 4f;

        // Get the position of the (textureID) from texture.
        float y = textureID / 4;
        float x = textureID - (y * 4);
        x *= textureSizeInBlocks;
        y *= textureSizeInBlocks;
        y = 1f - y - textureSizeInBlocks;

        // Add the calculated uvs to script.
        uvs.Add(new Vector2(x, y));
        uvs.Add(new Vector2(x, y + textureSizeInBlocks));
        uvs.Add(new Vector2(x + textureSizeInBlocks, y));
        uvs.Add(new Vector2(x + textureSizeInBlocks, y +
textureSizeInBlocks));
    }
```

Nu är kuberna redo att genereras. Så det du ska göra nu är att du skapar ett nytt gameobejct och lägger in tre komponenter på den. Dessa komponenter är **[MeshFilter, MeshRenderer & Chunk]**. Därefter så gör du en prefab utav detta objektet. Den kommer att användas senare.

Men som jag nämde ovan så har du fortfarande ett felmeddelande. Det ska vi nu ta hand om. Så skapa en ny klass och döp den till **World**. Den kommer att ta hand om hela världen, spara chunksen i en array, räkna ut vilket block som ska vara vart mm.

```
using UnityEngine;

public class World : MonoBehaviour
{
      public static World Instance;

      [Header("World Settings")]
      public int seed;
      public bool randomizeSeed = false;

      [Header("Caves")]
      [Range(0f, 0.7f)]
      public float caveFrequens = 0.01f;
      [Range(0f, 1f)]
      public float caveDensity = 0.5f;

      public AnimationCurve caves;

      [Header("Terrain")]
      [Range(0f, 1f)]
      public float noiseFrequenz = 0.05f;
      public AnimationCurve terrainHeight;

      public Chunk[,] chunks = new Chunk[WorldData.WorldSizeInChunks,
WorldData.WorldSizeInChunks];
      public GameObject chunkPrefab;

      [Space]
      public BlockType[] blockTypes;

      private void Awake() => Instance = this;
      private void Start()
      {
```

```
            Random.InitState(seed = randomizeSeed ? Random.Range(0,
999999) : seed);
        GenerateWorld();
      }

      private void GenerateWorld()
      {

      }

      public BlockType GetVoxel(Vector3Int position)
      {

      }

      public int GetTerrainHeight(int x, int z)
      {
            // Calculate and evaluate the terrain height according to a
curve.
            float preEvaluatedTerrainHeight = PerlinNoise.Perlin2D(x, z,
seed, noiseFrequenz);
            float evaluatedTerrainHeight =
terrainHeight.Evaluate(preEvaluatedTerrainHeight);

            return Mathf.FloorToInt(evaluatedTerrainHeight *
WorldData.ChunkHeightInVoxels);
      }

      public bool GenerateGrass(Vector3Int position)
      {
            int height = GetTerrainHeight(position.x, position.z);
            bool generateGrass = false;

            if (position.y == height + 1) generateGrass =
Random.Range(0, 4) == 0 ? true : false;
            return generateGrass;

      }

      public bool GenerateBedrock(Vector3Int position)
      {
            float value = (float)1 / (float)position.y;
```

```
            float randomValue = PerlinNoise.Perlin2D(position.x,
position.z, seed, 1.35f); // Random.Range(0.0f, 1.1f);

            if (randomValue <= value) return true;
            else return false;

      }

      public int GetDirtHeight(Vector3Int position)
      {
            float dirtHeight = PerlinNoise.Perlin2D(position.x,
position.z, seed, 14.1f, 14); // Random.Range(0.0f, 1.1f)
            return Mathf.FloorToInt(dirtHeight);
      }

      public bool IsVoxelInWorld(Vector3Int pos)
      {
            return (pos.x >= 0 && pos.x < WorldData.WorldSizeInVoxels &&
pos.y >= 0 && pos.y < WorldData.ChunkHeightInVoxels && pos.z >= 0 &&
pos.z < WorldData.WorldSizeInVoxels) ? true : false;
      }
}
```

Det som händer i scriptet ovan är att jag definierar alla variabler jag kommer att behöva, därefter så slumpar jag ett seed och lägger in det i unitys **Random** klass.

Nu kan vi bland annat räkna ut terrängets höjd i en punkt eller hur högt jorden ska gå på ett ställe innan det byts ut mot sten. Men vi saknar två funktioner. **GenerateWorld** och **GetVoxel**. Utan dessa två så kommer världen att inte genereras.

I **GetVoxel** funktionen så lägger vi in alla regler för hur blocken ska genereras.

```
public BlockType GetVoxel(Vector3Int position)
{
      // If the voxel is outside of the world the return a value that
is not in use by any block.
      if (!IsVoxelInWorld(position)) return blockTypes[0];

      // Calculate if there should be a cave in this voxel or not.
      float caveValue = PerlinNoise.Perlin3D((position.x + seed) *
caveFrequens, (position.y + seed) * caveFrequens, (position.z + seed) *
caveFrequens) * caves.Evaluate((float)position.y /
(float)WorldData.ChunkHeightInVoxels);
```

```
      int terrainHeight = GetTerrainHeight(position.x, position.z); //
Get the terrainheight.

      // Check if voxel is supposed to be a cave.
      if (caveValue >= caveDensity) return blockTypes[0];

      // Returns the block type for the voxel.
      if (position.y == terrainHeight + 1 && GenerateGrass(position) &&
GetVoxel(new Vector3Int(position.x, terrainHeight, position.z)).blockID
!= 0) return blockTypes[5]; // Grass
      if (position.y <= 4 && GenerateBedrock(position)) return
blockTypes[4]; // Bedrock
      if (position.y == terrainHeight) return blockTypes[1]; // Grass
Block
      if (position.y <= terrainHeight - GetDirtHeight(position) &&
position.y < terrainHeight) return blockTypes[3]; // Stone Block
      if (position.y < terrainHeight) return blockTypes[2]; // Dirt
Block

        return blockTypes[0];
    }
```

Detta exemplet blev lite väl dåligt. Men kom ihåg att koden alltid finns längst ned i dokumentet. Men det denna koden gör är att den kollar vad som ska vara vart utifrån en del regler. T.ex. så om dess y-position är lika mycket som terräng höjden så ska det vara gräs. Eller om y-positionen är lika med noll så ska det vara bedrock.

Den enda funktionen som jag inte har gott igenom än är **GenerateWorld** funktionen, och det enda som denna kommer att göra är att den loopar igenom varje chunk position i 2D arrayen och skapar ett nytt gameobject från en prefaben som jag har skapat sedan tidigare.

```
private void GenerateWorld()
{
      // Loop through the chunk positions in 2D.
      for (int x = 0; x < WorldData.WorldSizeInChunks; x++)
      {
            for (int z = 0; z < WorldData.WorldSizeInChunks; z++)
            {
                  // Create a new chunk object.
                  GameObject chunk = Instantiate(chunkPrefab, new
Vector3(x * WorldData.ChunkWidthInVoxels, 0, z *
WorldData.ChunkWidthInVoxels), Quaternion.identity, transform);
```

```
                chunk.name = $"Chunk: {x}, {z}";
                chunks[x, z] = chunk.GetComponent<Chunk>();
            }
        }
}
```

Du har nu gjort all logik för att generera världen, men du saknar ytterligare några saker. Vi har inte än lagt till olika block eller perlin bullret.

Så det du kan göra nu är att du skapar en ny klass som du döper till **BlockType** i denna klassen kommer vi att ha all viktig information för att ett block ska kunna få dess textur. Kopiera in den följande koden i scriptet.

```
using UnityEngine;

[System.Serializable]
public class BlockType
{
    public string blockName = string.Empty;
    public byte blockID = 0;
    public bool isSolid = false;

    // En array så att man kan ha flera texturer.
    public int[] backFaceTexture;
    public int[] frontFaceTexture;
    public int[] topFaceTexture;
    public int[] bottomFaceTexture;
    public int[] leftFaceTexture;
    public int[] rightFaceTexture;

    public int GetBlockTexture(int textureID)
    {
         // Den slumpar så att man kan ha ett block med flera olika
utseenden, till exempel Minecraft bedrock.
        switch (textureID)
        {
            case 0:
                if (backFaceTexture.Length > 1)
                    return backFaceTexture[Random.Range(0,
backFaceTexture.Length)];
                else return backFaceTexture[0];

            case 1:
                if (frontFaceTexture.Length > 1)
```

```
                    return frontFaceTexture[Random.Range(0,
frontFaceTexture.Length)];
                else return frontFaceTexture[0];

            case 2:
                if (topFaceTexture.Length > 1)
                    return topFaceTexture[Random.Range(0,
topFaceTexture.Length)];
                else return topFaceTexture[0];

            case 3:
                if (bottomFaceTexture.Length > 1)
                    return bottomFaceTexture[Random.Range(0,
bottomFaceTexture.Length)];
                else return bottomFaceTexture[0];

            case 4:
                if (leftFaceTexture.Length > 1)
                    return leftFaceTexture[Random.Range(0,
leftFaceTexture.Length)];
                else return leftFaceTexture[0];

            case 5:
                if (rightFaceTexture.Length > 1)
                    return rightFaceTexture[Random.Range(0,
rightFaceTexture.Length)];
                else return rightFaceTexture[0];
            default:
                return 0;
        }
    }
}
```

Det enda som finns kvar i kod väg nu är att göra perlin buller. Så skapa en ny statisk klass som du döper till **PerliNoise**. Därefter kopierar du in följande koden.

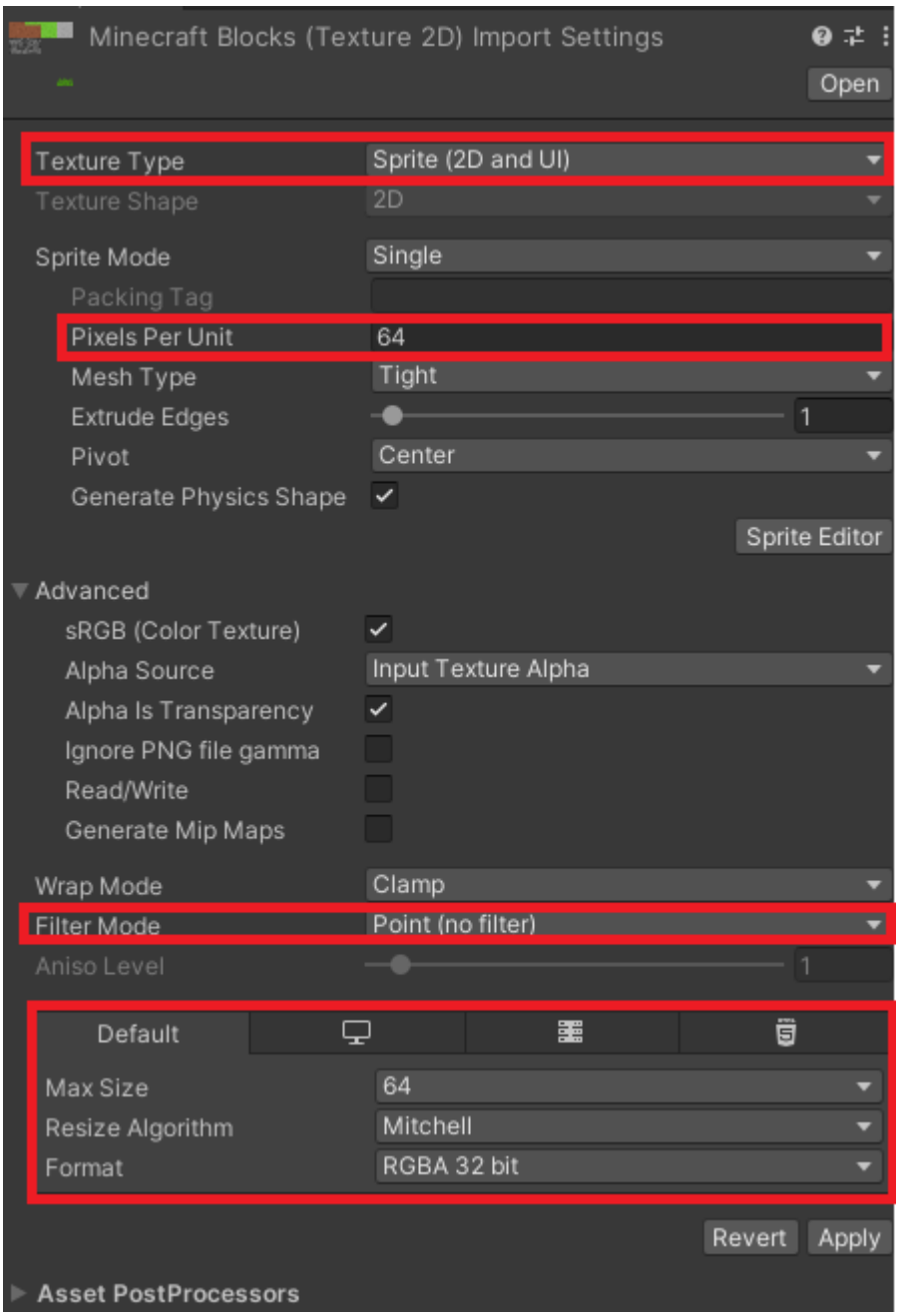
```
using UnityEngine;

public static class PerlinNoise
{
    public static float Perlin3D(float x, float y, float z)
    {
        float ab = Mathf.PerlinNoise(x, y);
        float bc = Mathf.PerlinNoise(y, z);
```

```
        float ac = Mathf.PerlinNoise(x, z);

        float ba = Mathf.PerlinNoise(y, x);
        float cb = Mathf.PerlinNoise(z, y);
        float ca = Mathf.PerlinNoise(z, x);

        float abc = ab + bc + ac + ba + cb + ca;
        return abc / 6f;
    }

    public static float Perlin2D(float x, float z, float seed, float
frequenze, float amplitude = 1, int octaves = 1)
    {
        if (amplitude <= 0) amplitude = 1;

        return Mathf.PerlinNoise((x + seed) * frequenze, (z + seed) *
frequenze) * amplitude;
    }
}
```

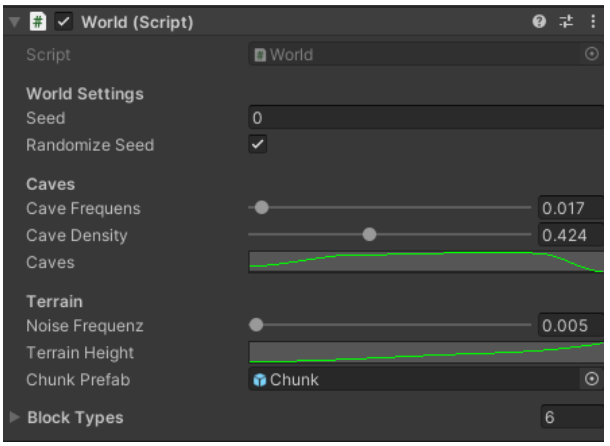
Den fungerar på samma sätt som jag har förklarat i tidigare delar av arbetet, utom att vi har lagt till en ny dimension. Det jag gör för att ta perlin värdet ur ett tredimensionellt synvinkel är att ta medelvärdet mellan pelrin bullret mellan alla värden. Det vill säga medelvärdet mellan:
$x + y,\ x + z\ därefter\ y + x,\ y + z\ och\ slutligen\ z + x, z + y$

Nu har vi gjort klar all logik inför världs genereringen. Men den fungerar inte riktigt än, vi behöver fortfarande definiera vissa variabler i unity editor.

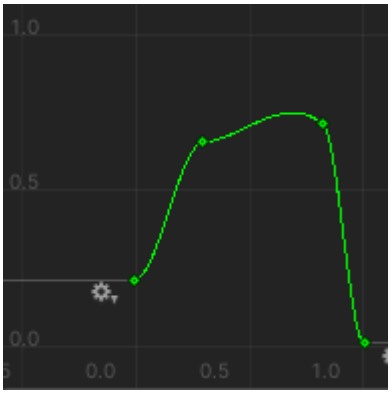
Ladda ned **MinecraftBlocks** texturen och lägg nu in det i projektet. Den följande inställningar.



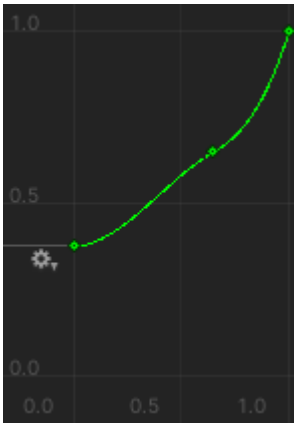
Skapa ett nytt material med och lägg in texturen på **Albedo.** Ändra därefter **Rendering Mode** från Opaque till Cutout. Så att man kan se igenom materialet. **Transparent** renderar andra objekt framför medan cutout skär bort dem. Skapa ett nytt gameobject. Denna kommer att vara själva världen, så lägg på **World** komponenten på objektet. **World** komponenten har flera inställningar som är roliga att ändra lite på, en liten ändring kan ändra hur hela världen genereras. Men jag har använt mig av dessa inställningar:



Caves kurvan kan ha följande positioner.

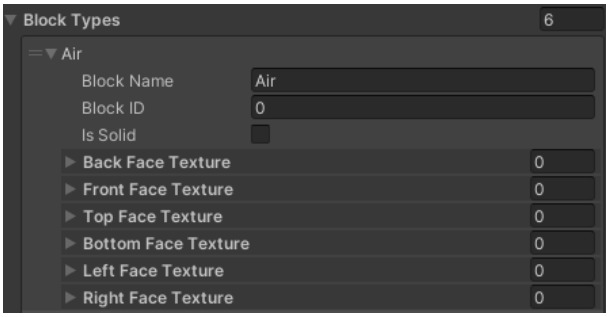


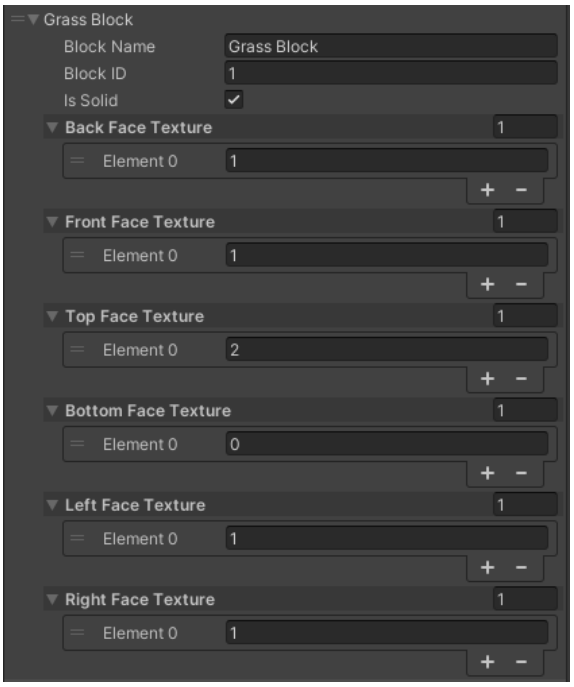
TerrainHeight kurvan kan ha följande.

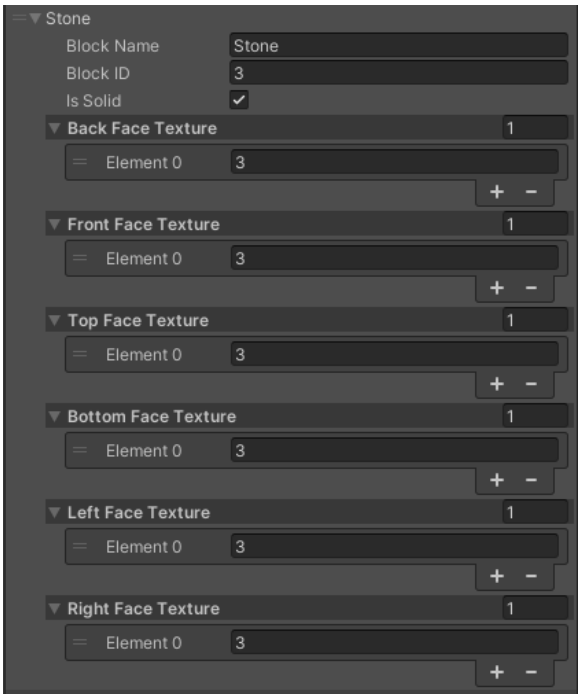


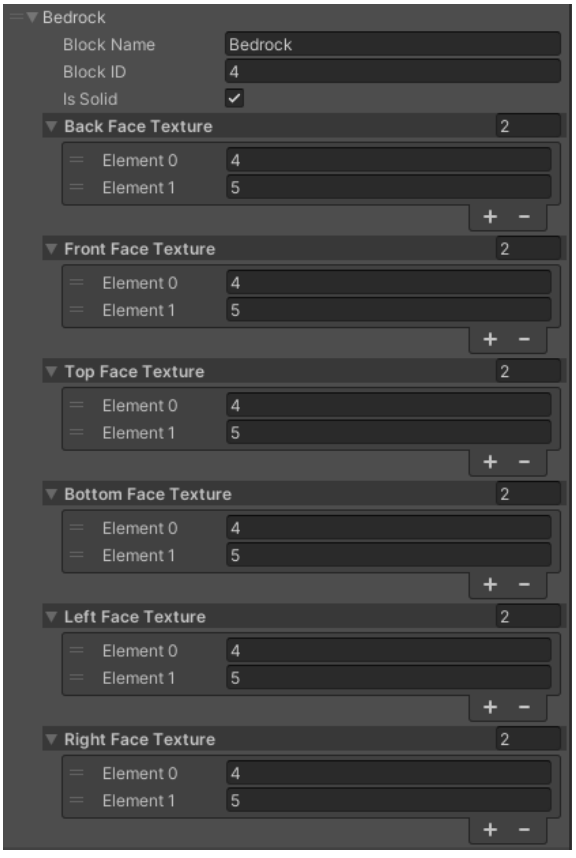
Det vi kommer att evaluera i dessa kurvor är terrängets höjd, de används för att störa höjden på perlin noiset och att göra så att det är mindre grottor desto högre upp och ned du är i landskapet.

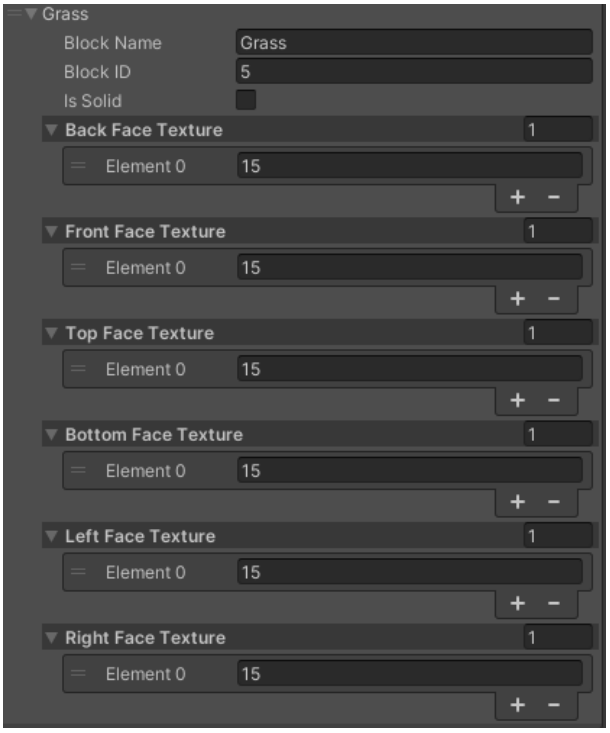
BlockTypes arrayen kommer att ha följande inställningar.



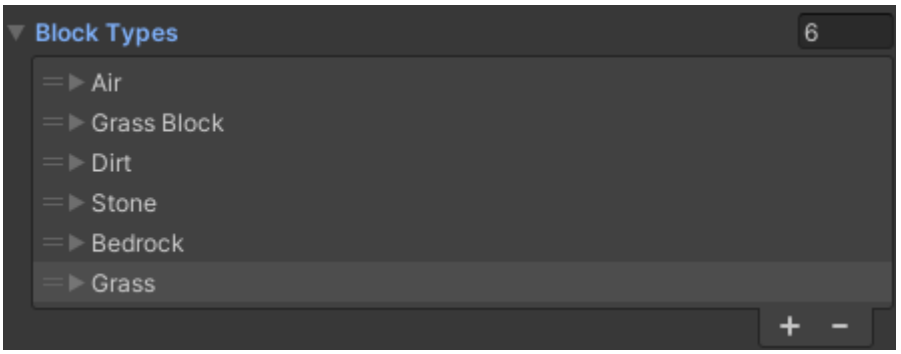

Grass Block
Block Name
Grass Block
Block ID
1
Is Solid
Back Face Texture
1
Element 0
1
Front Face Texture
1
Element 0
1
Top Face Texture
1
Element 0
2
Bottom Face Texture
1
Element 0
0
Left Face Texture
1
Element 0
1
Right Face Texture
1
Element 0
1


Stone
Block Name
Stone
Block ID
3
Is Solid
Back Face Texture
1
Element 0
3
Front Face Texture
1
Element 0
3
Top Face Texture
1
Element 0
3
Bottom Face Texture
1
Element 0
3
Left Face Texture
1
Element 0
3
Right Face Texture
1
Element 0
3

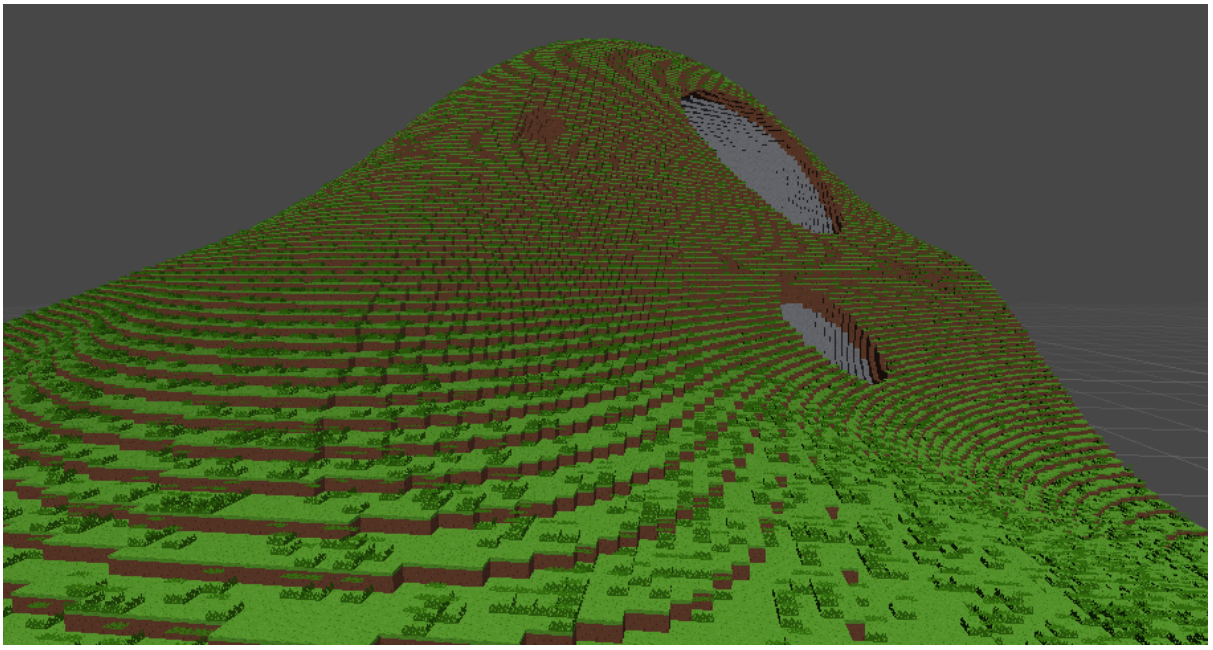

Bedrock
Block Name
Bedrock
Block ID
4
Is Solid
Back Face Texture
2
Element 0
4
Element 1
5
Front Face Texture
2
Element 0
4
Element 1
5
Top Face Texture
2
Element 0
4
Element 1
5
Bottom Face Texture
2
Element 0
4
Element 1
5
Left Face Texture
2
Element 0
4
Element 1
5
Right Face Texture
2
Element 0
4
Element 1
5


Grass
Block Name
Grass
Block ID
5
Is Solid
Back Face Texture
1
Element 0
15
Front Face Texture
1
Element 0
15
Top Face Texture
1
Element 0
15
Bottom Face Texture
1
Element 0
15
Left Face Texture
1
Element 0
15
Right Face Texture
1
Element 0
15

Det är viktigt att varje element sitter på samma index som jag har i arrayen för att få samma resultat som jag har, men testa gärna och ändra något för att se hur blocken ändrar plats. T.ex gräs istället för sten.



Mitt slutresultat.



Länkar

Mesh Terrain Generator	Google Drive
PerlinNoise	Google Drive
WorldData	Google Drive
BlockType	Google Drive
Chunk	Google Drive
VoxelMeshData	Google Drive
World	Google Drive
MinecraftBlocks.png	Google Drive
Testa olika endimensionella vågor.	Itch.io

Utvärdering

Slutresultatet av arbetet blev väldigt bra, jag är väldigt nöjd med det som jag har gjort även då jag har gjort om arbetet vid flera tillfällen. Jag är också nöjd med hur bra slumpmässigheten typ bedrock och jord blev. Samt placeringen av gräset med hjälp av ett slumpmässigt värde. Jag har fått ett resultat som liknar Minecrafts världar med Perlin noise, vilket var den grundläggande iden med arbetet. Jag blev också nöjd över att du kan skapa oändligt många världar med det som jag har gjort. Genom att ändra ett simpelt värde så får du högre berg eller större grottor.

Men några förändringsförslag hade varit att ändra sättet som jag genererar voxlarna. Jag hade velat byta ut den mot en MeshData klass där information om andra meshar kan finnas med, t.ex gräset i arbetet är genererade kuber. Det är endast där för att visa att det fungerar. Men hade man gjort en större klass med mesh-datan så hade man kunnat göra flera olika andra typer av block. Jag vill också ändra så att meshen delar upp sig i flera sub-chunkar. Så att man inte behöver ha en hel chunk aktiverad. Utan kan dela upp det ännu mer för att tänka på spelets prestanda. En sub-chunk hade i sådana fall varit 16*16*16 block stort.