4-16-2023

Voxel Generation in Real-Time

Gregory James Waldschmidt

Voxel Generation in Real-Time

by

Gregory James Waldschmidt

An Honors Capstone

submitted in partial fulfillment of the requirements

for the Honors Diploma

to

The Honors College

of

The University of Alabama in Huntsville

April 26, 2023

Honors Capstone Director: Dr. Timothy Newman
Professor of Computer Science

Gregory Waldschmidt 4/16/2023
Student (signature) Date

Timothy S. Newman 04/26/2023
Director (signature) Date

Letha Etzkorn Digitally signed by Letha Etzkorn
Date: 2023.04.27 14:31:05 -05'00'
Department Chair (signature) Date

William Wilkerson Digitally signed by William Wilkerson
Date: 2023.04.28 11:36:41 -05'00'
Honors College Dean (signature) Date

HONORS COLLEGE
THE UNIVERSITY OF ALABAMA IN HUNTSVILLE

Honors College Frank Franz

Hall

+1 (256) 824-6450 (voice)
+1 (256) 824-7339 (fax)
honors@uah.edu

Honors Thesis Copyright Permission

This form must be signed by the student and submitted with the final manuscript.
In presenting this thesis in partial fulfillment of the requirements for Honors Diploma or Certificate from The University of Alabama in Huntsville, I agree that the Library of this University shall make it freely available for inspection. I further agree that permission for extensive copying for scholarly purposes may be granted by my advisor or, in his/her absence, by the Chair of the Department, Director of the Program, or the Dean of the Honors College. It is also understood that due recognition shall be given to me and to The University of Alabama in Huntsville in any scholarly use which may be made of any material in this thesis.

Student Name (printed)

Gregory Waldschmidt

Student Signature

Gregory Waldschmidt

Date

April 26, 2023

Table of Contents

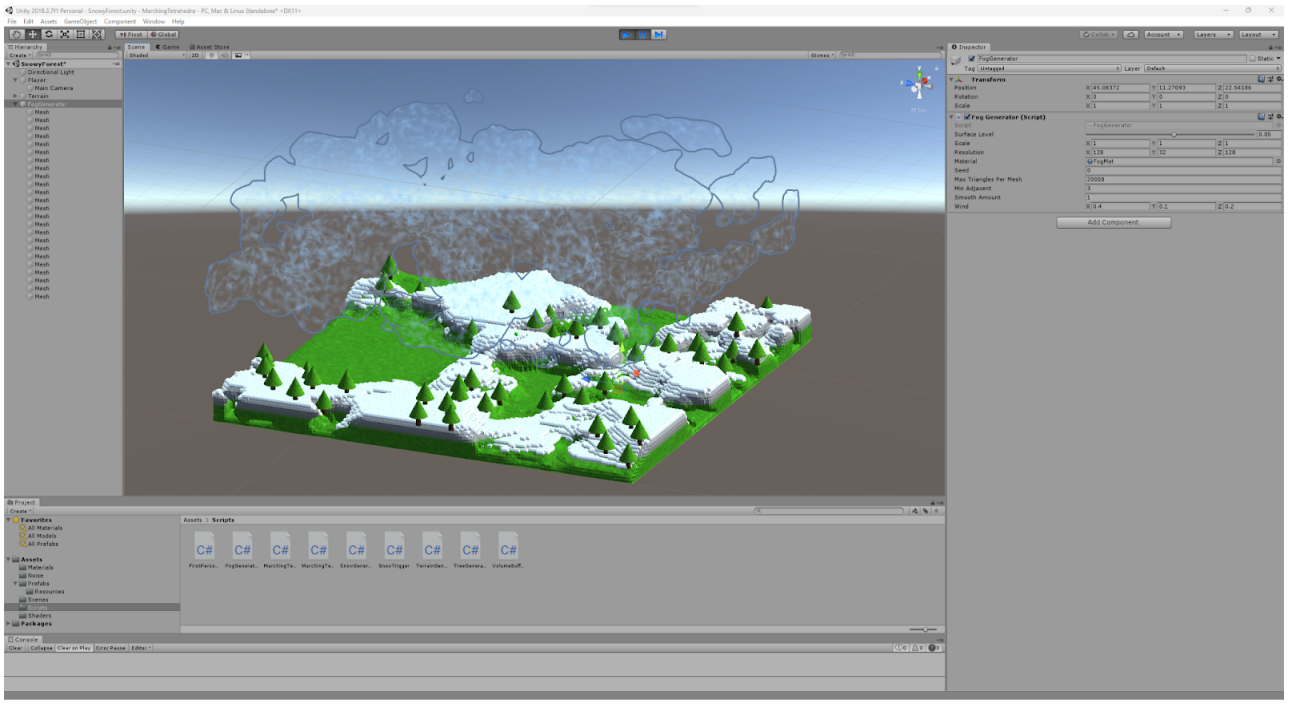
Dedication:

This project is dedicated to my friend, Amaree.

Introduction

This project uses voxels to achieve computer graphics effects in a soft real-time environment. The Marching Tetrahedra algorithm was used in Unity 2018.3.7 to generate procedural terrain, snow, trees, and fog. This project also uses noise to create procedural meshes and generate vertex colors. The complete project and video demonstration can be accessed using the following link:

https://drive.google.com/drive/folders/1BhwLqMn50cmixDQ7MymU1CmsRwehRevo?usp=share_link



(Screenshot of Unity 2018.3.7 editor during runtime. The Scene window shows procedural terrain with snow, trees, and fog.)

Chapter 1: Volume To Surface

The Marching Cubes and Marching Tetrahedra algorithm are the most widely used algorithms for converting volume data to an isosurface so it can be rendered as a 3D model. While the Marching Cubes algorithm lines up with trilinear interpolation, its original implementation can produce an isosurface with holes. The Marching Tetrahedra algorithm does not produce an isosurface with holes, is simpler to implement, and generates more triangles to better fit the volume data. Because the algorithm is being used in this project to produce computer graphics effects and not to represent scientific data, the isosurface does not need to line up with trilinear interpolation and it benefits from the added detail in Marching Tetrahedra. For the formerly stated reasons, the Marching Tetrahedra algorithm was chosen for this project.

Marching Tetrahedra in Unity

The Marching Tetrahedra algorithm is used to extract an isosurface from volume data to create procedural terrain, snow, trees, and fog in real-time within Unity 2018.3.7. Each voxel represents a point in rectilinear 3D space with an associated isovalue stored as a float. Voxels are stored in a volume buffer which is implemented as a 3D array of floats. Each volume buffer is designed to work with an arbitrary size and is configurable in the inspector window using a public Vector3 variable named *resolution*. A public variable (*surfaceLevel*) is used to specify the isosurface value. A voxel with a value below the isosurface value is considered empty space. A voxel with a value above or equal to the isosurface value is considered to be filled space. The resulting mesh is a surface representation of the volume data, effectively representing the outer shell of the volume. The volume buffer is two values bigger than the resolution in the x, y, and z direction to create extra rows and columns along

the outer edge of the volume buffer. When the volume buffer is looped over to be filled with voxel values, the first and last element is skipped in each direction; values along the outer edge of the volume buffer are never visited and are therefore set to empty space (0.0f). This ensures that the isosurface does not contain any holes along the edges of the volume buffer.

The volume buffer is divided into 2x2x2 cubes of voxels. Each cube is divided into six tetrahedra. For each tetrahedron, the MarchThroughTetrahedron method is called to generate triangles representing the isosurface. The intersection points of the isovalue with the tetrahedron edges are calculated using the *GetEdgeIntersect* method, which uses linear interpolation to determine the positions of the vertices based on their isovalues. These intersection points are stored in the *EdgeVert* array.

Triangles are generated by connecting the intersection points on the edges, and their indices are added to the index list. The vertices of these triangles are added to a vertice list. The winding order is used to determine the direction the faces are rendered. The *MarchingTetrahedraTables* class provides the necessary tables for vertex offsets, edge connections, index tables, edge intersections, and triangle generation.

The Marching Tetrahedra algorithm takes a volume buffer as an input and generates an isosurface as its output which is used to create the terrain, snow, trees, and fog. The algorithm divides each cube in the volume buffer into six tetrahedra, which is then split into triangles. The resulting geometry is added to vertex and index lists, which is used to generate meshes based on the volume data.

Mesh Generation

Mesh generation produces 3D models for the terrain, snow, trees, and fog based on volume data. This mesh generation can be directly applied to a wide variety of effects and objects that use meshes. Because the mesh is generated at runtime, an empty game object must be placed in the unity scene with a script attached containing a method to generate a mesh. The resulting mesh is a child of the empty game object. Before mesh generation, an empty list is created to store vertices, indices, and normals which are used as inputs for a marching tetrahedra object. Each list is filled based on the Marching Tetrahedra algorithm. The newly created lists are divided into separate, smaller lists if the number of vertices exceeds the maximum allowed per mesh. This split is done for runtime efficiency and to avoid artifacts caused in Unity 2018.3.7 when the triangle count of an individual mesh is very high. This method improves performance and avoids mesh artifacts but requires a marginal increase in memory. Mesh bounds and normals are recalculated after the lists are applied. A new GameObject is then created for each mesh along with a *MeshFilter* and a *MeshRenderer* component. The mesh is then applied to the *MeshFilter* mesh. The scale of the mesh is applied last and can be adjusted using a public variable. Some meshes require offsets in order to line the widget of the parent object with the mesh. This offset is done on the local position of the mesh and is relative to the resolution of the volume buffer and the scale of the object.

Vertex Coloring

Vertex colors can be assigned based on noise values and volume buffer position. The *SetVertexColors()* method takes an array of vertices as input and calculates the colors for each vertex. Meshes that use vertex coloring are assigned a material with a custom shader.

Chapter 2: Effect Generation

Noise

Noise is used in this project to generate procedural meshes and color. Because noise is sampled at locations in space, the relative distance between voxels is calculated to sample the noise at each voxel. The voxel size is calculated based on the volume buffer resolution, determining the size of each voxel in world space. The position of each voxel is then calculated using its index and voxel size, with the voxel offset being subtracted to center the voxels. Each voxel has xyz values which are used to sample the noise.

Fog Effect Generation

The fog effect is achieved using the Marching Tetrahedra algorithm and noise to create a procedurally generated mesh with variable color and transparency, with fog getting more transparent the higher it is. The fog is translated in real-time based on a wind vector (public variable) that controls the speed and direction of the fog.

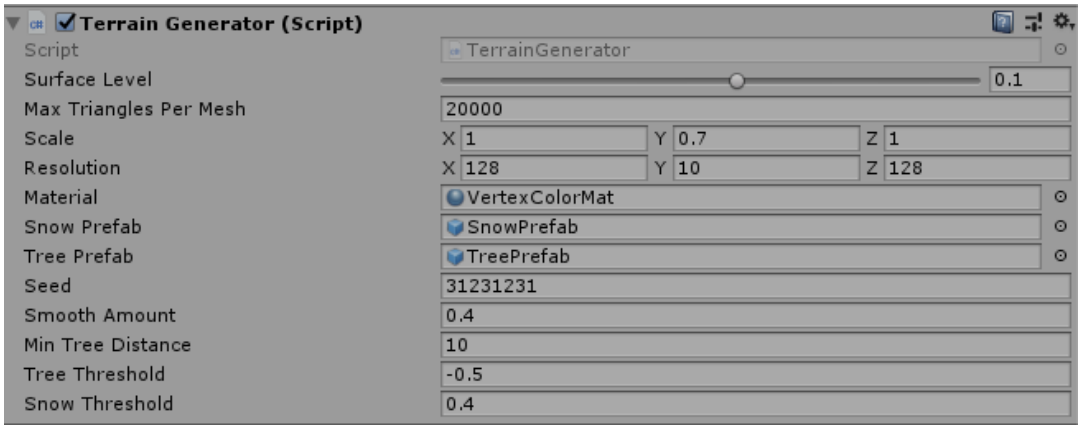
The *AddFog()* method generates 3D noise using perlin and fractal noise. After generating the fog, a Gaussian blur is applied to the volume data to smooth the fog and eliminate jagged shapes. The results of the Gaussian blur are initially stored in a separate volume buffer to avoid smoothing some areas multiple times. The kernel size is calculated based on the desired smooth amount (public variable), and the kernel is initialized and normalized so that the sum of its elements is 1. The output buffer is then combined with the kernel to generate the final blurred volume data. The *RemoveIslands()* method is used to eliminate small floating islands of fog. This method checks if a voxel has at least two adjacent voxels above the

surface level. If not, the voxel density is lowered below the surface level. The fog uses Unity's built in Standard Unlit shader with rendering mode set to Fade and Color Mode set to Color, allowing for vertex coloring with transparency.

Vertex coloring is generated based on the height of the vertex and the fog noise. A Perlin noise generator is used to create fog noise, while a Fractal noise generator is employed to calculate the height. The fog transparency is calculated by taking into account both the fog noise value and the vertex height. The fog noise value is clamped after being multiplied by 2, and a height factor is calculated by clamping the vertex height multiplied by a public variable that determines how much influence the height has on fog transparency. The fog transparency is then determined by multiplying the clamped fog noise value with (1 - height factor). The fog color's red, green, and blue components are determined using the height value. The calculated fog color, including the fog transparency, is then assigned to the corresponding index in the colors array.

Terrain Generation

The procedural terrain generation process involves generating the base terrain, placing trees and snow, smoothing the terrain, and applying vertex coloring. Public variables can be used to modify the parameters of the terrain via the Inspector window in Unity. Note that chunks can be added to create effectively infinite procedural terrain.



(Inspector window showing parameters that can be modified to customize procedural terrain)

The terrain is generated using the Marching Tetrahedra algorithm and noise to create a procedurally generated mesh with variable color and transparency. The terrain is generated using Perlin, Fractal, and Voronoi noise. The seed can be modified in the inspector window to generate terrain based on different noise. Trees and snow are placed procedurally on the terrain based on specific noise thresholds. The bottom of the volume buffer is filled with solid voxels (1.0f) to prevent holes in the terrain.

The *SmoothTerrain()* method is used to smooth the terrain by averaging the density values of neighboring voxels. The smoothing algorithm is applied to each voxel, and the new density value is set based on the average density of its neighbors. Changes to the volume buffer are initially stored in a separate volume buffer to avoid calculating some areas multiple times.

Vertex coloring is applied to the terrain to create rocks and grass. Two separate noise generators, one for rocks and one for grass, are created using Perlin noise. The grass color is calculated based on grass noise and height, while the rock color is calculated based on rock noise and height. The vertex colors are then set by lerping between the grass and rock colors based on the height of each vertex.

Tree Generation

Trees are added to the terrain procedurally using Perlin noise to determine the position of each tree. Trees are placed if the height of the terrain at a specific coordinate is greater than zero and the noise value is less than the tree threshold. Additionally, trees are placed only if they are not too close to an existing tree, as determined by the minimum tree distance (public variable).

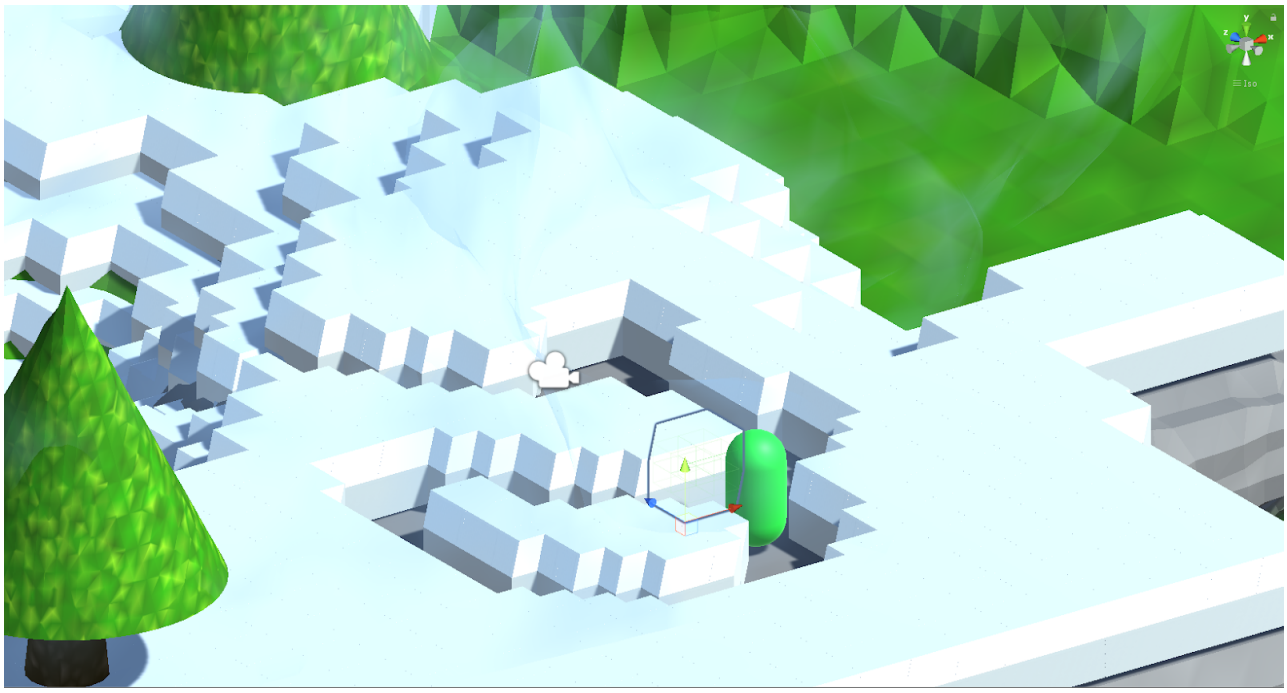
Each tree is generated using the Marching Tetrahedra algorithm. The volume buffer is split by height to determine whether it should generate the trunk or branches of the tree. The tree is created using a mathematical cylinder function while the branches use a mathematical cone function. The resulting mesh no longer lines up with the pivot point of its parent.

Vertex coloring is applied to the vertices of the tree based on its height to distinguish between the trunk and branches. Separate perlin noise is used to create color variation in the trunk and branches. The colors are interpolated between the trunk and branches sections, creating a smooth transition between the two sections. The final color of each vertex is determined by blending the trunk and branches colors using the height of the vertex in the tree model.

Snow Generation

Destructible snow is added to the terrain. Snow is placed if the height of the terrain at a specific coordinate is greater than zero, the noise value is less than the snow threshold, and the height is greater than or equal to the snow height. A prefab (premade object) is added at each voxel position in the terrain of a small 2x2x2 cube of snow. To create destructible snow,

each section of snow is the size of a terrain voxel. Each section of snow contains four snow triggers. Each snow trigger stores the voxel that it corresponds to in the snow volume. When the player enters the trigger, the surface is set to empty space (0.0f), the mesh is recalculated, and the trigger is destroyed. The snow mesh is recalculated every time the player's trigger collider overlaps with the snow trigger. Combining small sections of snow creates the effect of snow destroying without the drop in framerate caused by recalculating a large snow volume.



(A section of snow is recalculated when the player trigger overlaps with a snow trigger to create the effect of destructible snow.)

Reference List

M. Chen, A. Kaufman and R. Yagel, "Volume Graphics," in Springer-Verlag New York, Inc., 2001.

C. D. Hansen and C. R. Johnson, "The Visualization Handbook," Elsevier Inc., 2005.

W. E. Lorensen and H. E. Cline, "Marching Cubes: A High Resolution 3D Surface Construction Algorithm," in Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH '87, New York, NY, USA, 1987, pp. 163-169.

B. Lichtenbelt, Introduction to Volume Rendering, Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1998.

A. Watt, 3D Computer Graphics (3rd Edition), Addison-Wesley, 2000.

G. M. Nielson, "On marching cubes," in IEEE Transactions on Visualization and Computer Graphics, vol. 9, no. 3, pp. 283-297, July-Sept. 2003, doi: 10.1109/TVCG.2003.1207437.

A. Kaufman, "Volume graphics," IEEE Computer Graphics and Applications, vol. 13, no. 4, pp. 51-55, Jul. 1993.

Conclusion

The Marching Tetrahedra algorithm was used to achieve various computer graphics effects in a soft real-time environment within Unity 2018.3.7. The Marching Tetrahedra algorithm was used to generate procedural terrain, snow, trees, and fog. Noise was used to create procedural meshes and generate vertex colors. The Marching Tetrahedra algorithm was used to create meshes by dividing each cube in the volume buffer into six tetrahedra, which is then split into triangles to create the outer shell of the volume. Mesh generation produces 3D models for the terrain, snow, trees, and fog based on volume data. Vertex colors can be assigned based on noise values and volume buffer position. Fog, terrain, tree, and snow were generated procedurally using the Marching Tetrahedra algorithm and noise. The fog effect was achieved by generating 3D noise using perlin and fractal noise for the mesh and vertex coloring. Terrain generation involved generating the base terrain, placing trees and snow, smoothing the terrain, and applying vertex coloring using Perlin, Fractal, and Voronoi noise. Tree generation involved generating the trunk and branches of the tree and applying vertex coloring based on its height to separately color the trunk and branches. Snow generation involved placing destructible snow on the terrain using a prefab and snow triggers. When the player enters the trigger, the surface is set to empty space, the mesh is recalculated, and the trigger is destroyed. This project demonstrated the potential of using voxels in a soft real-time environment to achieve various computer graphics effects.

Appendix

The appendix includes the important scripts used to achieve each effect including terrain, snow, fog, trees, and the marching tetrahedra algorithm.

MarchingTetrehedra.cs

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

namespace MarchingTetrahedraNamespace
{
    public class MarchingTetrahedra
    {
        public float Surface;
        public float[,,] Voxels;

        private float[] Cube;
        private float[] CubeIsovalues;

        private int[] WindingOrder;

        private Vector3[] EdgeVert;
        private Vector3[] CubePosition;
        private Vector3[] TetrahedronPosition;

        // Constructor for marching tetrahedra object
        public MarchingTetrahedra(float surface, float[,,] voxels)
        {
            EdgeVert = new Vector3[6];
            CubePosition = new Vector3[8];
            TetrahedronPosition = new Vector3[4];
            CubeIsovalues = new float[4];
            Surface = surface;
            Cube = new float[8];
            WindingOrder = new int[] { 2, 1, 0 };
            Voxels = voxels;
        }

        public virtual void Generate(IList<Vector3> verts, IList<int> indices)
        {
            Vector3 resolution = new Vector3(Voxels.GetLength(0), Voxels.GetLength(1),
Voxels.GetLength(2));

            UpdateWindingOrder();

            int x, y, z, i;
            int ix, iy, iz;
```

```
        // Call March() to run the marching tetrahedra algorithm on each cube in the volume
buffer.
        for (x = 0; x < resolution.x - 1; x++)
        {
            for (y = 0; y < resolution.y - 1; y++)
            {
                for (z = 0; z < resolution.z - 1; z++)
                {
                    // Get all values of the cube and store them in the Cube array, overwriting
previous entries.
                    for (i = 0; i < 8; i++)
                    {
                        ix = x + MarchingTetrahedraTables.VertexOffsetTable[i, 0];
                        iy = y + MarchingTetrahedraTables.VertexOffsetTable[i, 1];
                        iz = z + MarchingTetrahedraTables.VertexOffsetTable[i, 2];

                        Cube[i] = Voxels[ix, iy, iz];
                    }

                    // Run marching tetrahedra on the current cube.
                    MarchThroughCube(x, y, z, Cube, verts, indices);
                }
            }
        }
    }

    // Set winding order based on Isovalue level
    // to ensure faces are in the right direction.
    private void UpdateWindingOrder()
    {
        if (Surface < 0.0f)
        {
            WindingOrder[0] = 2;
            WindingOrder[1] = 1;
            WindingOrder[2] = 0;
        }
        else
        {
            WindingOrder[0] = 0;
            WindingOrder[1] = 1;
            WindingOrder[2] = 2;
        }
    }

    // GetEdgeIntersect finds the approximate point of intersection of the Isovalue
    // between two points with the values v1 and v2
    private float GetEdgeIntersect(float vertex1, float vertex2)
```

```
    {
        return (Surface - vertex1) / (vertex2 - vertex1);
    }

    // March through one cube. There are 6 tetrahedra per cube.
    private void MarchThroughCube(float x, float y, float z, float[] cube, IList<Vector3>
vertList, IList<int> indexList)
    {
        int i, k, vertexInACube;

        for (i = 0; i < 8; i++)
        {
            CubePosition[i].x = x + MarchingTetrahedraTables.VertexOffsetTable[i, 0];
            CubePosition[i].y = y + MarchingTetrahedraTables.VertexOffsetTable[i, 1];
            CubePosition[i].z = z + MarchingTetrahedraTables.VertexOffsetTable[i, 2];
        }

        for (i = 0; i < 6; i++)
        {
            for (k = 0; k < 4; k++)
            {
                vertexInACube = MarchingTetrahedraTables.Indextable[i, k];
                TetrahedronPosition[k] = CubePosition[vertexInACube];
                CubeIsovalues[k] = cube[vertexInACube];
            }

            MarchThroughTetrahedron(vertList, indexList);
        }
    }

    // MarchThroughTetrahedron loops through a tetrahedron made up of four vertices
and generates triangles to render a 3D Isovalue
    // The generated triangles are stored as indices in indexList and the vertices are stored
in vertList
    private void MarchThroughTetrahedron(IList<Vector3> vertList, IList<int> indexList)
    {
        int i, j, vert, vert0, vert1, idx;
        int flagIndex = 0, edgeFlags;
        float offset, inverseOffset;

        for (i = 0; i < 4; i++)
        {
            if (CubeIsovalues[i] <= Surface)
            {
                flagIndex |= 1 << i;
            }
        }
```

```
        // Determine which edges of the tetrahedron are intersected by the Isovalue using
the flagIndex
        edgeFlags = MarchingTetrahedraTables.EdgeIntersectTable[flagIndex];

        // If none of the edges are intersected there are no triangles to generate so return
early
        if (edgeFlags == 0)
        {
            return;
        }

        // Calculate the intersection points of the Isovalue with each intersected edge and
store them in EdgeVert
        for (i = 0; i < 6; i++)
        {
            if ((edgeFlags & (1 << i)) != 0)
            {
                vert0 = MarchingTetrahedraTables.EdgeConnectionTable[i, 0];
                vert1 = MarchingTetrahedraTables.EdgeConnectionTable[i, 1];
                offset = GetEdgeIntersect(CubeIsovalues[vert0], CubeIsovalues[vert1]);
                inverseOffset = 1.0f - offset;

                EdgeVert[i].x = inverseOffset * TetrahedronPosition[vert0].x + offset *
TetrahedronPosition[vert1].x;
                EdgeVert[i].y = inverseOffset * TetrahedronPosition[vert0].y + offset *
TetrahedronPosition[vert1].y;
                EdgeVert[i].z = inverseOffset * TetrahedronPosition[vert0].z + offset *
TetrahedronPosition[vert1].z;
            }
        }

        // Generate triangles using the intersection points and add them to indexList and
vertList
        for (i = 0; i < 2; i++)
        {
            if (MarchingTetrahedraTables.triangleTable[flagIndex, 3 * i] < 0) break;

            idx = vertList.Count;

            for (j = 0; j < 3; j++)
            {
                vert = MarchingTetrahedraTables.triangleTable[flagIndex, 3 * i + j];
                indexList.Add(idx + WindingOrder[j]);
                vertList.Add(EdgeVert[vert]);
            }
        }
    }
}
```

}

MarchingTetrehedraTables.cs

```
using UnityEngine;

namespace MarchingTetrahedraNamespace
{
    public class MarchingTetrahedraTables : MonoBehaviour
    {
        // The VertexOffsetTable stores the distance from vertex0 for each
        // of the cube's 12 vertices. Example, the first row contains {0,0,0}
        // meaning that vertex0 is 0 units away from vertex0 on the x, y, and z-axis.
        // likewise, the last row stores a vertex that is 0 units away on the x-axis,
        // 1 unit away on the y-axis, and 1 unit away on the z-axis.
        public static readonly int[,] VertexOffsetTable = new int[,]
        {
        {0, 0, 0},
        {1, 0, 0},
        {1, 1, 0},
        {0, 1, 0},
        {0, 0, 1},
        {1, 0, 1},
        {1, 1, 1},
        {0, 1, 1}
        };

        // The EdgeConnectionTable stores the index of each vertice that makes up an edge in
the cube.
        // There are two vertices for each of the cube's 6 tetrahedrons.
        public static readonly int[,] EdgeConnectionTable = new int[,]
        {
        {0,1},  {1,2},  {2,0},  {0,3},  {1,3},  {2,3}
        };

        // The EdgeConnectionTable stores the index of each vertice that makes up an edge in
the cube.
        // Each row represents a tetrahedron in a cube.
        public static readonly int[,] Indextable = new int[,]
        {
        {0,5,1,6},
        {0,1,2,6},
        {0,2,3,6},
        {0,3,7,6},
        {0,7,4,6},
        {0,4,5,6}
        };

        // The EdgeIntersectTable stores the edge intersections for each of the 16 cases.
```

```
        // Conversion of the EdgeInstersectTable from hex to binary for reference.
        // Each entry contains a list of 12 bits that correspond to the 12 edges of the cube.
        // Edges are marked as intersecting (1) or not intersecting(0).
        public static readonly int[] EdgeIntersectTable = new int[]
        {
        0x00, 0x0d, 0x13, 0x1e, 0x26, 0x2b, 0x35, 0x38,
        0x38, 0x35, 0x2b, 0x26, 0x1e, 0x13, 0x0d, 0x00
        };

        // The triangleTable stores the triangle list for each case,
        // one for each row. Each column corresponds to a vertex in the cube with the
exception of the final column.
        // The final column is always -1 to indicate that there are no more vertices to extract.
        // -1 is an "invalid" value and indicates that the cube has finished extracting vertices.
        public static readonly int[,] triangleTable = new int[,]
        {
        {-1, -1, -1, -1, -1, -1, -1},
        { 0,  3,  2, -1, -1, -1, -1},
        { 0,  1,  4, -1, -1, -1, -1},
        { 1,  4,  2,  2,  4,  3, -1},
        { 1,  2,  5, -1, -1, -1, -1},
        { 0,  3,  5,  0,  5,  1, -1},
        { 0,  2,  5,  0,  5,  4, -1},
        { 5,  4,  3, -1, -1, -1, -1},
        { 3,  4,  5, -1, -1, -1, -1},
        { 4,  5,  0,  5,  2,  0, -1},
        { 1,  5,  0,  5,  3,  0, -1},
        { 5,  2,  1, -1, -1, -1, -1},
        { 3,  4,  2,  2,  4,  1, -1},
        { 4,  1,  0, -1, -1, -1, -1},
        { 2,  3,  0, -1, -1, -1, -1},
        {-1, -1, -1, -1, -1, -1, -1}
        };
    }
}
```

TerrainGenerator.cs

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

using ProceduralNoiseProject;

namespace MarchingTetrahedraNamespace
{
    public class TerrainGenerator : MonoBehaviour
    {
        [Range(-1.0f, 1.0f)]
```

```
    public float surfaceLevel = 0.0F;
    public int maxTrianglesPerMesh = 20000;
    public Vector3 scale = new Vector3(1.0f, 1.0f, 1.0f);
    public Vector3Int resolution = new Vector3Int(32, 32, 32);
    public Material material;
    public VolumeBuffer voxels;
    public GameObject snowPrefab;
    public GameObject treePrefab;
    public int seed = 0;
    public float smoothAmount = 1.0f;
    public float minTreeDistance = 10.0f;
    public float treeThreshold = -0.5f;
    public float snowThreshold = 0.5f;

    private List<GameObject> meshes = new List<GameObject>();
    private List<Vector3> treePositions = new List<Vector3>();

    void Start()
    {
        // Create a volume buffer. The size is resolution + 2 because the outer edges of the
        // volume buffer will always be empty to prevent holes.
        voxels = new VolumeBuffer(new Vector3Int(resolution.x + 2, resolution.y + 2,
resolution.z + 2));

        AddTerrain();
        AddTrees();
        SmoothTerrain();
        AddSnow();
        Generate();
    }

    private void AddTerrain()
    {
        INoise perlin = new PerlinNoise(seed, 1.0f, 1.0f);
        INoise voronoi = new VoronoiNoise(seed, 1.0f, 1.0f);

        FractalNoise fractal = new FractalNoise(perlin, 4, 3.0f, 0.5f);
        FractalNoise fractal2 = new FractalNoise(voronoi, 3, 2.0f, 0.25f);

        // Calculate the position of each voxel in world space
        float voxelSize = 1f / (resolution.x - 1);

        Vector3 voxelOffset = new Vector3(0.5f, 0.5f, 0.5f) * voxelSize;
        for (int x = 1; x < resolution.x + 1; x++)
        {
            for (int y = 1; y < resolution.y + 1; y++)
            {
                for (int z = 1; z < resolution.z + 1; z++)
```

```
            {
                Vector3 pos = new Vector3(x, y, z) * voxelSize - voxelOffset;

                // Use the noise functions to generate terrain features
                float height = fractal.Sample3D(pos.x, pos.y, pos.z);
                height += fractal2.Sample3D(pos.x, pos.y, pos.z) * 0.5f;

                float density = pos.y - height;

                if (density >= 0)
                {
                    voxels[x, y, z] = 0;
                }
                else
                {
                    voxels[x, y, z] = 1;
                }
            }
        }
    }

    // Fill the bottom of the volume buffer with solid voxels
    for (int x = 1; x < resolution.x + 1; x++)
    {
        for (int z = 1; z < resolution.z + 1; z++)
        {
            voxels[x, 0, z] = 1;
        }
    }
}

private void AddTrees()
{
    INoise treePerlin = new PerlinNoise(seed + 2, 5.0f, 2.0f);

    // Calculate the position of each voxel in world space
    float voxelSize = 1f / (resolution.x - 1);

    Vector3 voxelOffset = new Vector3(0.5f, 0.5f, 0.5f) * voxelSize;
    for (int x = 1; x < resolution.x + 1; x++)
    {
        for (int z = 1; z < resolution.z + 1; z++)
        {
            // Determine the height of the terrain at this x,z coordinate
            int height = 0;
            for (int y = resolution.y + 1; y > 1; y--)
            {
                if (voxels[x, y, z] == 1)
```

```
            {
                height = y;
                break;
            }
        }

        // Get the noise value at this x,z coordinate
        Vector3 pos = new Vector3(x, 0.0f, z) * voxelSize - voxelOffset;
        float treeNoiseValue = treePerlin.Sample2D(pos.x, pos.z);

        // Add trees if the noise value is greater than the tree threshold
        if (height > 0 && treeNoiseValue < treeThreshold)
        {
            // Check the distance to the existing tree positions
            Vector3 newTreePos = new Vector3(x, height, z);
            bool isCloseToExistingTree = false;
            foreach (Vector3 treePos in treePositions)
            {
                if (Vector3.Distance(newTreePos, treePos) < minTreeDistance)
                {
                    isCloseToExistingTree = true;
                    break;
                }
            }

            // Add the new tree if it is not too close to an existing tree
            if (!isCloseToExistingTree)
            {
                PlaceTree(x, height, z);
            }
        }
    }
  }
}

private void AddSnow()
{
    INoise snowPerlin = new PerlinNoise(seed + 1, 5.0f, 0.3f);
    int snowHeight = (int)(resolution.y / 3);
    float voxelSize = 1f / (resolution.x - 1);
    Vector3 voxelOffset = new Vector3(0.5f, 0.5f, 0.5f) * voxelSize;

    for (int x = 1; x < resolution.x + 1; x++)
    {
        for (int z = 1; z < resolution.z + 1; z++)
        {
            // Determine the height of the terrain at this x,z coordinate
            int height = 0;
```

```
            for (int y = resolution.y + 1; y > 1; y--)
            {
                if (voxels[x, y, z] >= 0.3)
                {
                    height = y;
                    break;
                }
            }

            // Get the noise value at this x,z coordinate
            Vector3 pos = new Vector3(x, 0.0f, z) * voxelSize - voxelOffset;
            float snowNoiseValue = snowPerlin.Sample2D(pos.x, pos.z);

            // Add snow if the noise value is greater than the snow threshold
            if (height > 0 && snowNoiseValue < snowThreshold && height >= snowHeight)
            {
                PlaceSnow(x, height, z);
            }
        }
    }
}

// Create a trigger collider for the snow to detect player collision at a voxel
private void PlaceSnow(int x, int y, int z)
{
    GameObject snow;
    // Specify the local position for the current voxel relative to its parent (fogGenerator).
    Vector3 localPos = new Vector3(x * scale.x, y * scale.y, z * scale.z);

    snow = Instantiate(snowPrefab, Vector3.zero, Quaternion.identity, transform);

    snow.transform.parent = transform;
    snow.transform.position = snow.transform.parent.transform.position;
    snow.transform.localPosition = localPos + new Vector3(0.0f, scale.y * 0.9f, 0.0f);
    snow.transform.localScale = scale;
}

// Create a trigger collider for the snow to detect player collision at a voxel
private void PlaceTree(int x, int y, int z)
{
    GameObject tree;
    // Specify the local position for the current voxel relative to its parent (fogGenerator).
    Vector3 localPos = new Vector3(x * scale.x, y * scale.y, z * scale.z);

    tree = Instantiate(treePrefab, Vector3.zero, Quaternion.identity, transform);

    // Add the new tree position to the list
    treePositions.Add(new Vector3(x, y, z));
```

```
    tree.transform.parent = transform;
    tree.transform.position = tree.transform.parent.transform.position;
    tree.transform.localPosition = localPos + new Vector3(0.0f, scale.y, 0.0f);
    tree.transform.localScale = scale;
}

private void SmoothTerrain()
{
    VolumeBuffer output = voxels;

    // Apply the smoothing algorithm to each voxel
    for (int x = 1; x < resolution.x + 1; x++)
    {
        for (int y = 1; y < resolution.y + 1; y++)
        {
            for (int z = 1; z < resolution.z + 1; z++)
            {
                // Calculate the average density of neighboring voxels
                float sum = 0;
                int count = 0;

                for (int dx = -1; dx <= 1; dx++)
                {
                    for (int dy = -1; dy <= 1; dy++)
                    {
                        for (int dz = -1; dz <= 1; dz++)
                        {
                            sum += voxels[x + dx, y + dy, z + dz];
                            count++;
                        }
                    }
                }

                float avg = sum / count;

                // Set the new density value for the current voxel
                output[x, y, z] = avg;
            }
        }
    }

    // Copy the new voxel values back into the original buffer
    voxels = output;
}

// Generate updates the mesh and deletes any old meshes.
public void Generate()
```

```
    {
        MarchingTetrahedra myMarchingTetrahedra = new
MarchingTetrahedra(surfaceLevel, voxels.Voxels);

        List<Vector3> vertices = new List<Vector3>();
        List<int> indices = new List<int>();
        List<Vector3> normals = new List<Vector3>();

        myMarchingTetrahedra.Generate(vertices, indices);

        // Line up mesh to be in the center of its parent's pivot point.
        var position = new Vector3(-resolution.x / 2, -resolution.y / 2, -resolution.z / 2);
        GenerateMesh(vertices, indices, normals);
    }

    // GenerateMesh generates a mesh by splitting a list of vertices, indices and normals
into multiple meshes if there are too many vertices to fit in a single mesh.
    // It sets the vertices, normals, and triangles for each mesh and assigns a color to each
vertex.
    // It creates a new GameObject for each mesh and adds a MeshFilter, MeshRenderer,
and assigns a material to the renderer.
    // GenerateMesh also sets the generated mesh to be the MeshFilter's mesh and applies
a scale to the object.
    public void GenerateMesh(List<Vector3> verts, List<int> indices, List<Vector3>
normals)
    {
        int maxVertsPerMesh = maxTrianglesPerMesh * 3;
        int numMeshes = verts.Count / maxVertsPerMesh + 1;

        for (int i = 0; i < numMeshes; i++)
        {
            List<Vector3> currentVerts = new List<Vector3>();
            List<Vector3> currentNormals = new List<Vector3>();
            List<int> currentIndices = new List<int>();

            for (int j = 0; j < maxVertsPerMesh; j++)
            {
                int idx = i * maxVertsPerMesh + j;

                if (idx < verts.Count)
                {
                    currentVerts.Add(verts[idx]);
                    currentIndices.Add(j);

                    if (normals.Count != 0)
                    {
                        currentNormals.Add(normals[idx]);
                    }
```

```
    }
}

if (currentVerts.Count == 0)
{
    continue;
}

Mesh mesh = new Mesh();

// Set the mesh normals
if (currentNormals.Count <= 0)
{
    mesh.RecalculateNormals();
}

else
{
    mesh.SetNormals(currentNormals);
}

// Set the mesh vertices
mesh.SetVertices(currentVerts);

Vector3[] vertices = mesh.vertices;

// assign the array of colors to the Mesh.
mesh.colors = this.SetVertexColors(vertices);

// Set the mesh triangles using the meshIndices list
mesh.SetTriangles(currentIndices, 0);

// Recalculate the normals of the mesh
mesh.RecalculateNormals();

// Recalculate the bounds of the mesh
mesh.RecalculateBounds();

GameObject myMesh = new GameObject("Mesh")
{
    tag = "Mesh"
};

// Set the parent of the new GameObject to be the current transform
myMesh.transform.parent = transform;

myMesh.AddComponent<MeshFilter>();
myMesh.AddComponent<MeshRenderer>();
```

```
            myMesh.GetComponent<Renderer>().material = material;

            // Set the MeshFilter component to use the generated mesh"
            myMesh.GetComponent<MeshFilter>().mesh = mesh;

            // Apply the scale of the object
            myMesh.transform.localScale = scale;

            myMesh.AddComponent<MeshCollider>();

            meshes.Add(myMesh);
        }
    }

    private Color[] SetVertexColors(Vector3[] vertices)
    {
        // Create noise generators for rocks and grass
        INoise rockNoise = new PerlinNoise(seed + 3, 1.0f, 0.5f);
        INoise grassNoise = new PerlinNoise(seed + 4, 0.1f, 0.5f);

        // Create fractal noise generator for height
        FractalNoise fractal = new FractalNoise(rockNoise, 4, 1.0f, 0.5f);

        Color[] colors = new Color[vertices.Length];

        for (int i = 0; i < vertices.Length; i++)
        {
            float height = fractal.Sample3D(vertices[i].x, vertices[i].y, vertices[i].z);

            // Calculate grass color based on grass noise and height
            float grassNoiseValue = grassNoise.Sample3D(vertices[i].x * 10.0f, vertices[i].y *
10.0f, vertices[i].z * 10.0f);
            Color grassColor = new Color(0.2f, 0.5f + grassNoiseValue * 0.3f, 0.1f);
            grassColor = Color.Lerp(grassColor, new Color(0.7f, 0.9f, 0.2f),
Mathf.Clamp01(height * 2.0f));

            // Calculate rock color based on rock noise and height
            float rockNoiseValue = rockNoise.Sample3D(vertices[i].x * 5.0f, vertices[i].y * 5.0f,
vertices[i].z * 5.0f);
            Color rockColor = new Color(0.5f, 0.5f, 0.5f);
            rockColor = Color.Lerp(rockColor, new Color(0.7f, 0.7f, 0.7f), rockNoiseValue *
0.5f);
            rockColor = Color.Lerp(rockColor, new Color(0.3f, 0.3f, 0.3f), Mathf.Clamp01(height
* 0.5f));

            colors[i] = Color.Lerp(grassColor, rockColor, vertices[i].y - 6.5f);
        }
```

```
            return colors;
        }
    }
}
```

FogGenerator.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using ProceduralNoiseProject;
namespace MarchingTetrahedraNamespace
{
    public class FogGenerator : MonoBehaviour
    {
        [Range(-1.0f, 1.0f)]
        public float surfaceLevel = 0.0F;
        public Vector3 scale = new Vector3(1.0f, 1.0f, 1.0f);
        public Vector3Int resolution = new Vector3Int(16, 8, 16);
        public Material material;
        public int seed = 0;
        public int maxTrianglesPerMesh = 20000;
        public int minAdjacent = 3;
        public VolumeBuffer voxels;
        public float smoothAmount = 1.0f;
        public Vector3 Wind = new Vector3(0.4f, 0.1f, 0.2f);
        public float heightTransparencyInfluence = 0.05f;

        private GameObject player;

        private void Update()
        {
            // Move the fog based on a wind vector.
            transform.localPosition += Wind * Time.deltaTime;
        }

        private void Start()
        {
            player = GameObject.Find("Player"); // Find player.

            // Create volume buffer. The size is resolution + 2 because the outer edges of the
            // volume buffer will always be empty to prevent holes.
            voxels = new VolumeBuffer(new Vector3Int(resolution.x + 2, resolution.y + 2,
resolution.z + 2));

            AddFog();
            ApplyGaussianBlur();
            RemoveIslands();
```

```
        Generate();
    }

    // AddFog generates a 3D noise pattern using fractal noise
    // It uses PerlinNoise and FractalNoise classes to generate noise with different
parameters
    // voxelSize is calculated based on the resolution to determine the size of each voxel in
world space
    // The position of each voxel is calculated based on its index and voxelSize, and
voxelOffset is subtracted to center the voxels
    // The density value is generated for each voxel using fractal.Sample3D function and
stored in the voxels array.
    private void AddFog()
    {
        // Generate a 3D Perlin noise object with the given parameters
        INoise perlin = new PerlinNoise(seed, 3.0f, 1.0f);

        // Create a fractal noise object that combines the Perlin noise with additional layers
        FractalNoise fractal = new FractalNoise(perlin, 3, 2.0f, 0.5f);

        // Calculate the size of each voxel in world space
        float voxelSize = 1f / (resolution.x - 1);

        // Calculate the offset of the voxel grid so that the center of the first voxel is at (0, 0,
0)
        Vector3 voxelOffset = new Vector3(0.5f, 0.5f, 0.5f) * voxelSize;

        // Loop through all voxels in the grid
        for (int x = 1; x < resolution.x + 1; x++)
        {
            for (int y = 1; y < resolution.y + 1; y++)
            {
                for (int z = 1; z < resolution.z + 1; z++)
                {
                    // Calculate the position of the voxel in world space
                    Vector3 pos = new Vector3(x, y, z) * voxelSize - voxelOffset;

                    // Sample the fractal noise function to get the density value of the voxel
                    float density = fractal.Sample3D(pos.x, pos.y, pos.z);

                    // Store the density value in the voxel buffer
                    voxels[x, y, z] = density;
                }
            }
        }
    }

    private void RemoveIslands()
```

```
    {

        VolumeBuffer output = voxels;

        for (int x = 1; x < resolution.x + 1; x++)
        {
            for (int y = 1; y < resolution.y + 1; y++)
            {
                for (int z = 1; z < resolution.z + 1; z++)
                {
                    if (voxels[x, y, z] >= surfaceLevel)
                    {
                        // Check if voxel has at least 2 adjacent voxels above surface level
                        int numAboveSurfaceLevel = 0;
                        if (voxels[x - 1, y, z] >= surfaceLevel) numAboveSurfaceLevel++;
                        if (voxels[x + 1, y, z] >= surfaceLevel) numAboveSurfaceLevel++;
                        if (voxels[x, y - 1, z] >= surfaceLevel) numAboveSurfaceLevel++;
                        if (voxels[x, y + 1, z] >= surfaceLevel) numAboveSurfaceLevel++;
                        if (voxels[x, y, z - 1] >= surfaceLevel) numAboveSurfaceLevel++;
                        if (voxels[x, y, z + 1] >= surfaceLevel) numAboveSurfaceLevel++;

                        if (numAboveSurfaceLevel < minAdjacent)
                        {
                            // Lower voxel density below surface level
                            output[x, y, z] = surfaceLevel - 0.001f; // or any value slightly below
surfaceLevel
                        }
                    }
                }
            }
        }

        voxels = output;
    }

    // Apply a Gaussian blur to the volume data using a convolution kernel
    private void ApplyGaussianBlur()
    {

        // Calculate the kernel size based on the desired smooth amount
        int kernelSize = Mathf.CeilToInt(smoothAmount * 3) * 2 + 1; // Round to nearest odd
integer

        // Initialize the kernel and calculate the sum of all elements in the kernel
        float[,] kernel = new float[kernelSize, kernelSize];
        float kernelSum = 0;
        for (int i = 0; i < kernelSize; i++)
        {
```

```
            for (int j = 0; j < kernelSize; j++)
            {
                float x = i - kernelSize / 2;
                float y = j - kernelSize / 2;
                kernel[i, j] = Mathf.Exp(-(x * x + y * y) / (2 * smoothAmount * smoothAmount));
                kernelSum += kernel[i, j];
            }
        }

        // Normalize the kernel so that the sum of its elements is 1
        for (int i = 0; i < kernelSize; i++)
        {
            for (int j = 0; j < kernelSize; j++)
            {
                kernel[i, j] /= kernelSum;
            }
        }

        // Create a new VolumeBuffer object to hold the blurred volume data
        VolumeBuffer output = new VolumeBuffer(new Vector3Int(resolution.x + 2,
resolution.y + 2, resolution.z + 2));

        // Convolve the kernel with the volume data
        for (int x = 1; x < (resolution.x + 1); x++)
        {
            for (int y = 1; y < (resolution.y + 1); y++)
            {
                for (int z = 1; z < (resolution.z + 1); z++)
                {
                    float sum = 0;

                    // Apply the kernel at the current voxel position
                    for (int i = 0; i < kernelSize; i++)
                    {
                        for (int j = 0; j < kernelSize; j++)
                        {
                            int u = x + i - kernelSize / 2;
                            int v = y + j - kernelSize / 2;

                            // Make sure that the voxel position is within the volume bounds
                            if (u >= 0 && u < resolution.x && v >= 0 && v < resolution.y)
                            {
                                // combine the kernel with the voxel value at the current position
                                sum += kernel[i, j] * voxels[u, v, z];
                            }
                        }
                    }
```

```
                // Store the combined value in the output buffer
                output[x, y, z] = sum;
            }
        }
    }

    // Update the voxels with the blurred data
    voxels = output;
}

// Generate updates the mesh and deletes any old meshes.
public void Generate()
{
    MarchingTetrahedra myMarchingTetrahedra = new
MarchingTetrahedra(surfaceLevel, voxels.Voxels);

    List<Vector3> vertices = new List<Vector3>();
    List<int> indices = new List<int>();
    List<Vector3> normals = new List<Vector3>();

    myMarchingTetrahedra.Generate(vertices, indices);

    // Line up mesh to be in the center of its parent's pivot point.
    var position = new Vector3(-resolution.x / 2, -resolution.y / 2, -resolution.z / 2);

    GenerateMesh(vertices, indices, normals);
}

// GenerateMesh generates a mesh by splitting a list of vertices, indices and normals
into multiple meshes if there are too many vertices to fit in a single mesh.
// It sets the vertices, normals, and triangles for each mesh and assigns a color to each
vertex.
// It creates a new GameObject for each mesh and adds a MeshFilter, MeshRenderer,
and assigns a material to the renderer.
// GenerateMesh also sets the generated mesh to be the MeshFilter's mesh and applies
a scale to the object.
public void GenerateMesh(List<Vector3> verts, List<int> indices, List<Vector3>
normals)
{
    int maxVertsPerMesh = maxTrianglesPerMesh * 3;
    int numMeshes = verts.Count / maxVertsPerMesh + 1;

    for (int i = 0; i < numMeshes; i++)
    {
        List<Vector3> currentVerts = new List<Vector3>();
        List<Vector3> currentNormals = new List<Vector3>();
        List<int> currentIndices = new List<int>();
```

```
for (int j = 0; j < maxVertsPerMesh; j++)
{
   int idx = i * maxVertsPerMesh + j;

   if (idx < verts.Count)
   {
      currentVerts.Add(verts[idx]);
      currentIndices.Add(j);

      if (normals.Count != 0)
      {
         currentNormals.Add(normals[idx]);
      }
   }
}

if (currentVerts.Count == 0)
{
   continue;
}

Mesh mesh = new Mesh();

// Set the mesh normals
if (currentNormals.Count <= 0)
{
   mesh.RecalculateNormals();
}

else
{
   mesh.SetNormals(currentNormals);
}

// Set the mesh vertices
mesh.SetVertices(currentVerts);

Vector3[] vertices = mesh.vertices;

// assign the array of colors to the Mesh.
mesh.colors = this.SetVertexColors(vertices);

// Set the mesh triangles using the meshIndices list
mesh.SetTriangles(currentIndices, 0);

// Recalculate the normals of the mesh
mesh.RecalculateNormals();
```

```
            // Recalculate the bounds of the mesh
            mesh.RecalculateBounds();

            GameObject myMesh = new GameObject("Mesh")
            {
                tag = "Mesh"
            };

            // Set the parent of the new GameObject to be the current transform
            myMesh.transform.parent = transform;

            myMesh.AddComponent<MeshFilter>();
            myMesh.AddComponent<MeshRenderer>();
            myMesh.GetComponent<Renderer>().material = material;

            // Set the MeshFilter component to use the generated mesh"
            myMesh.GetComponent<MeshFilter>().mesh = mesh;

            // Apply the scale of the object
            myMesh.transform.localScale = scale;
            //   myMesh.transform.localPosition = Vector3.zero;
        }
    }

    private Color[] SetVertexColors(Vector3[] vertices)
    {
        INoise fogNoise = new PerlinNoise(seed + 5, 0.01f, 0.2f);

        // Create fractal noise generator for height
        FractalNoise fractal = new FractalNoise(fogNoise, 3, 0.005f, 0.3f);

        Color[] colors = new Color[vertices.Length];

        for (int i = 0; i < vertices.Length; i++)
        {
            float height = fractal.Sample3D(vertices[i].x, vertices[i].y, vertices[i].z);

            // Calculate fog color based on fog noise and height
            float fogNoiseValue = fogNoise.Sample3D(vertices[i].x * 50.0f, vertices[i].y * 50.0f,
vertices[i].z * 50.0f);
            float fogTransparencyBase = Mathf.Clamp01(fogNoiseValue * 2.0f);
            float heightFactor = Mathf.Clamp01(vertices[i].y * heightTransparencyInfluence);
// Adjust the multiplier to control the height influence on the transparency
            float fogTransparency = fogTransparencyBase * (1 - heightFactor);
            float r = 0.6f + height * 0.1f;
            float g = 0.8f + height * 0.05f;
            float b = 1.0f + height * 0.2f;
            Color fogColor = new Color(r, g, b, fogTransparency);
```

```
            colors[i] = fogColor;
        }

        return colors;
    }
  }
}
```

SnowGenerator.cs

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

namespace MarchingTetrahedraNamespace
{
  public class SnowGenerator : MonoBehaviour
  {
    [Range(-1.0f, 1.0f)]
    public float surfaceLevel = 0.0F;
    public Vector3 scale = new Vector3(1.0f, 1.0f, 1.0f);
    public Vector3Int resolution = new Vector3Int(32, 4, 32);
    public Material material;
    public int seed = 0;
    public int maxTrianglesPerMesh = 20000;
    public GameObject triggerPrefab;
    public VolumeBuffer voxels;

    private GameObject player;

    void Start()
    {
      player = GameObject.Find("Player"); // Find player.

      // Create volume buffer. The size is resolution + 2 because the outer edges of the
      // volume buffer will always be empty to prevent holes.
      voxels = new VolumeBuffer(new Vector3Int(resolution.x + 2, resolution.y + 2,
resolution.z + 2));

      FillVolumeBuffer();

      Generate();
    }

    // Fill the entire volume buffer with snow.
    // Set the scalar values for each voxel in the volume buffer
    // Note that values along the outer edge of the volume buffer are never visited and are
therefore
```

```
    // set to 0.0f. This ensures that each mesh is closed.
    private void FillVolumeBuffer()
    {
        for (int x = 1; x < resolution.x + 1; x++)
        {
            for (int y = 1; y < resolution.y + 1; y++)
            {
                for (int z = 1; z < resolution.z + 1; z++)
                {
                    voxels[x, y, z] = 1.0f;
                    CreateTrigger(x, y, z); // create trigger collider
                }
            }
        }
    }

    // Create a trigger collider for the snow to detect player collision at a voxel
    private void CreateTrigger(int x, int y, int z)
    {
        GameObject trigger;
        // Specify the local position for the current voxel relative to its parent (fogGenerator).
        Vector3 localPos = new Vector3(x * scale.x, y * scale.y, z * scale.z);
        trigger = Instantiate(triggerPrefab, Vector3.zero, Quaternion.identity, transform);
        trigger.GetComponent<SnowTrigger>().Voxel = new Vector3Int(x, y, z);
        trigger.GetComponent<SnowTrigger>().Initialize(this);
        trigger.transform.localPosition = localPos;
        trigger.transform.localScale = scale;
    }

    // Generate updates the mesh and deletes any old meshes.
    public void Generate()
    {
        MarchingTetrahedra myMarchingTetrahedra = new
MarchingTetrahedra(surfaceLevel, voxels.Voxels);

        List<Vector3> vertices = new List<Vector3>();
        List<int> indices = new List<int>();
        List<Vector3> normals = new List<Vector3>();

        myMarchingTetrahedra.Generate(vertices, indices);

        // Line up mesh to be in the center of its parent's pivot point.
        var position = new Vector3(-resolution.x / 2, -resolution.y / 2, -resolution.z / 2);

        DestroyMeshes();
        GenerateMesh(vertices, indices, normals);
    }
```

```
    // Destroy all children mesh to remove old meshes
    public void DestroyMeshes()
    {
        foreach (Transform child in transform)
        {
            if (child.gameObject.CompareTag("Mesh"))
            {
                GameObject.Destroy(child.gameObject);
            }
        }
    }

    // GenerateMesh generates a mesh by splitting a list of vertices, indices and normals into multiple meshes if there are too many vertices to fit in a single mesh.
    // It sets the vertices, normals, and triangles for each mesh and assigns a color to each vertex.
    // It creates a new GameObject for each mesh and adds a MeshFilter, MeshRenderer, and assigns a material to the renderer.
    // GenerateMesh also sets the generated mesh to be the MeshFilter's mesh and applies a scale to the object.
    public void GenerateMesh(List<Vector3> verts, List<int> indices, List<Vector3> normals)
    {
        int maxVertsPerMesh = maxTrianglesPerMesh * 3;
        int numMeshes = verts.Count / maxVertsPerMesh + 1;

        for (int i = 0; i < numMeshes; i++)
        {
            List<Vector3> currentVerts = new List<Vector3>();
            List<Vector3> currentNormals = new List<Vector3>();
            List<int> currentIndices = new List<int>();

            for (int j = 0; j < maxVertsPerMesh; j++)
            {
                int idx = i * maxVertsPerMesh + j;

                if (idx < verts.Count)
                {
                    currentVerts.Add(verts[idx]);
                    currentIndices.Add(j);

                    if (normals.Count != 0)
                    {
                        currentNormals.Add(normals[idx]);
                    }
                }
            }
```

```
if (currentVerts.Count == 0)
{
    continue;
}

Mesh mesh = new Mesh();

// Set the mesh normals
if (currentNormals.Count <= 0)
{
    mesh.RecalculateNormals();
}

else
{
    mesh.SetNormals(currentNormals);
}

// Set the mesh vertices
mesh.SetVertices(currentVerts);

Vector3[] vertices = mesh.vertices;

// assign the array of colors to the Mesh.
// mesh.colors = this.SetVertexColors(vertices);

// Set the mesh triangles using the meshIndices list
mesh.SetTriangles(currentIndices, 0);

// Recalculate the normals of the mesh
mesh.RecalculateNormals();

// Recalculate the bounds of the mesh
mesh.RecalculateBounds();

GameObject myMesh = new GameObject("Mesh")
{
    tag = "Mesh"
};

// Set the parent of the new GameObject to be the current transform
myMesh.transform.parent = transform;

myMesh.AddComponent<MeshFilter>();
myMesh.AddComponent<MeshRenderer>();
myMesh.GetComponent<Renderer>().material = material;

// Set the MeshFilter component to use the generated mesh"
```

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

            // Apply the scale of the object
            myMesh.transform.localScale = scale;

            myMesh.transform.position = myMesh.transform.parent.transform.position;
            myMesh.transform.localScale = scale;
        }
    }
  }
}
```

SnowTrigger.cs

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

// Create a namespace for the script
namespace MarchingTetrahedraNamespace
{
   // Declare the class for the script
   public class SnowTrigger : MonoBehaviour
   {
      public Vector3Int Voxel { get; set; }

      private SnowGenerator snowGeneratorScript;

      // Identify the snow generator that coresponds to the snow trigger.
      // This is called from a snow generator or another class.
      public void Initialize(SnowGenerator genScript)
      {
         snowGeneratorScript = genScript;
      }

      // Detects when a collider enters the trigger
      private void OnTriggerEnter(Collider collision)
      {
         // If the collider has the SnowDestroyer tag, destroy the snow at this voxel
         if (collision.gameObject.tag == "SnowDestroyer")
         {
            // Set the density of the voxel to zero
            snowGeneratorScript.voxels[Voxel.x, Voxel.y, Voxel.z] = 0.0f;

            // Generate the mesh for the snow
            snowGeneratorScript.Generate();

            // Destroy this game object
            Destroy(gameObject);
```

```
        }
      }
   }
}
```

TreeGenerator.cs

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

using ProceduralNoiseProject;

namespace MarchingTetrahedraNamespace
{
   public class TreeGenerator : MonoBehaviour
   {
      [Range(-1.0f, 1.0f)]
      public float surfaceLevel = 0.0F;
      public Vector3 scale = new Vector3(0.25f, 0.25f, 0.25f);
      public Vector3Int resolution = new Vector3Int(32, 32, 32);
      public Material material;
      public int seed = 0;
      public int maxTrianglesPerMesh = 20000;
      public VolumeBuffer voxels;

      void Start()
      {
         // Create a volume buffer. The size is resolution + 2 because the outer edges of the
         // volume buffer will always be empty to prevent holes.
         voxels = new VolumeBuffer(new Vector3Int(resolution.x + 2, resolution.y + 2,
resolution.z + 2));
         CreateTree();
         Generate();
      }

      // Fill the volume buffer with a tree, split between trunks and branches.
      private void CreateTree()
      {
         for (int x = 1; x < resolution.x + 1; x++)
         {
            for (int y = 1; y < resolution.y + 1; y++)
            {
               for (int z = 1; z < resolution.z + 1; z++)
               {
                  float centerX = resolution.x / 2.0f;
                  float centerZ = resolution.z / 2.0f;
                  float distance = Mathf.Sqrt(Mathf.Pow(x - centerX, 2) + Mathf.Pow(z - centerZ,
2));
```

```
                float treeRadius = (resolution.y - y) / (float)resolution.y * (resolution.x / 2.0f);

                // Trunk Section
                if (y < resolution.y / 3.0f)
                {
                    voxels[x, y, z] = treeRadius / 5.0f - distance;
                }

                // Branches Section
                else
                {
                    voxels[x, y, z] = treeRadius - distance;
                }
            }
        }
    }
}

// Generate updates the mesh and deletes any old meshes.
public void Generate()
{
    MarchingTetrahedra myMarchingTetrahedra = new
MarchingTetrahedra(surfaceLevel, voxels.Voxels);

    List<Vector3> vertices = new List<Vector3>();
    List<int> indices = new List<int>();
    List<Vector3> normals = new List<Vector3>();

    myMarchingTetrahedra.Generate(vertices, indices);

    // Line up mesh to be in the center of its parent's pivot point.
    var position = new Vector3(-resolution.x / 2, -resolution.y / 2, -resolution.z / 2);
    GenerateMesh(vertices, indices, normals);
}

// GenerateMesh generates a mesh by splitting a list of vertices, indices and normals
into multiple meshes if there are too many vertices to fit in a single mesh.
// It sets the vertices, normals, and triangles for each mesh and assigns a color to each
vertex.
// It creates a new GameObject for each mesh and adds a MeshFilter, MeshRenderer,
and assigns a material to the renderer.
// GenerateMesh also sets the generated mesh to be the MeshFilter's mesh and applies
a scale to the object.
public void GenerateMesh(List<Vector3> verts, List<int> indices, List<Vector3>
normals)
{
    int maxVertsPerMesh = maxTrianglesPerMesh * 3;
    int numMeshes = verts.Count / maxVertsPerMesh + 1;
```

```
for (int i = 0; i < numMeshes; i++)
{
    List<Vector3> currentVerts = new List<Vector3>();
    List<Vector3> currentNormals = new List<Vector3>();
    List<int> currentIndices = new List<int>();

    for (int j = 0; j < maxVertsPerMesh; j++)
    {
        int idx = i * maxVertsPerMesh + j;

        if (idx < verts.Count)
        {
            currentVerts.Add(verts[idx]);
            currentIndices.Add(j);

            if (normals.Count != 0)
            {
                currentNormals.Add(normals[idx]);
            }
        }
    }

    if (currentVerts.Count == 0)
    {
        continue;
    }

    Mesh mesh = new Mesh();

    // Set the mesh normals
    if (currentNormals.Count <= 0)
    {
        mesh.RecalculateNormals();
    }

    else
    {
        mesh.SetNormals(currentNormals);
    }

    // Set the mesh vertices
    mesh.SetVertices(currentVerts);

    Vector3[] vertices = mesh.vertices;

    // assign the array of colors to the Mesh.
    mesh.colors = this.SetVertexColors(vertices);
```

```
            // Set the mesh triangles using the meshIndices list
            mesh.SetTriangles(currentIndices, 0);

            // Recalculate the normals of the mesh
            mesh.RecalculateNormals();

            // Recalculate the bounds of the mesh
            mesh.RecalculateBounds();

            GameObject myMesh = new GameObject("Mesh")
            {
               tag = "Mesh"
            };

            // Set the parent of the new GameObject to be the current transform
            myMesh.transform.parent = transform;

            myMesh.AddComponent<MeshFilter>();
            myMesh.AddComponent<MeshRenderer>();
            myMesh.GetComponent<Renderer>().material = material;

            // Set the MeshFilter component to use the generated mesh"
            myMesh.GetComponent<MeshFilter>().mesh = mesh;

            myMesh.transform.localScale = scale;
            myMesh.transform.position = myMesh.transform.parent.transform.position;
            myMesh.transform.localPosition = new Vector3(-resolution.x / 2.0f * scale.x, 0.0f,
-resolution.z /2.0f * scale.z);

            myMesh.AddComponent<MeshCollider>();

            mesh.RecalculateBounds();
         }
      }

      private Color[] SetVertexColors(Vector3[] vertices)
      {
         // Create noise generators for trunk and branches
         INoise trunkNoise = new PerlinNoise(seed + 3, 5.0f, 2.0f);
         INoise branchesNoise = new PerlinNoise(seed + 4, 0.1f, 0.5f);

         // Create fractal noise generator for height
         FractalNoise fractal = new FractalNoise(trunkNoise, 4, 1.0f, 0.5f);

         Color[] colors = new Color[vertices.Length];

         for (int i = 0; i < vertices.Length; i++)
```

```
        {
            float height = fractal.Sample3D(vertices[i].x, vertices[i].y, vertices[i].z);

            // Calculate branches color based on branches noise and height
            float branchesNoiseValue = branchesNoise.Sample3D(vertices[i].x * 10.0f,
vertices[i].y * 10.0f, vertices[i].z * 10.0f);
            Color branchesColor = new Color(0.2f, 0.5f + branchesNoiseValue * 0.3f, 0.1f);
            branchesColor = Color.Lerp(branchesColor, new Color(0.7f, 0.9f, 0.2f),
Mathf.Clamp01(height * 2.0f));

            // Calculate trunk color based on trunk noise and height
            float trunkNoiseValue = trunkNoise.Sample3D(vertices[i].x * 5.0f, vertices[i].y *
5.0f, vertices[i].z * 5.0f);
            Color trunkColor = new Color(0.2f, 0.15f, 0.05f);
            trunkColor = Color.Lerp(trunkColor, new Color(0.5f, 0.4f, 0.3f), trunkNoiseValue *
0.5f);

            colors[i] = Color.Lerp(trunkColor, branchesColor, vertices[i].y - 10f);
        }

        return colors;
    }
  }
}
```