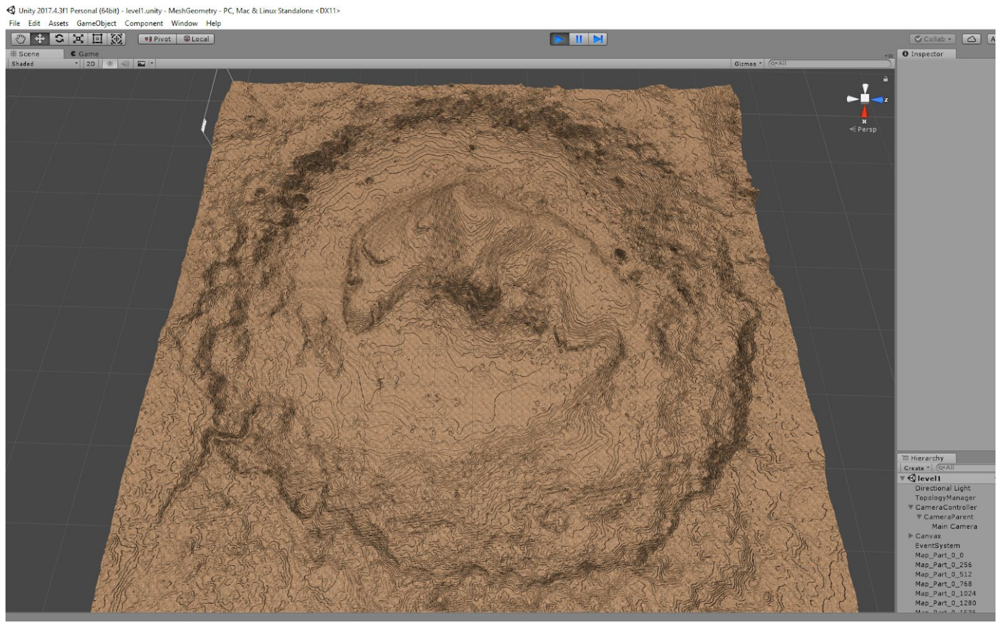
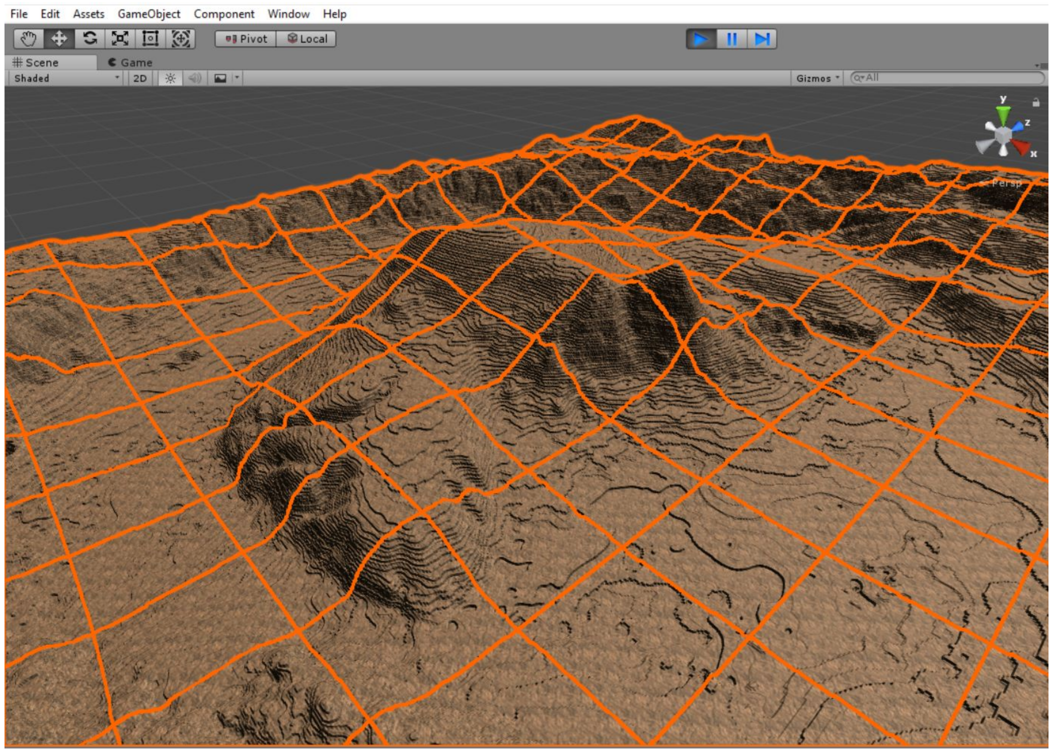
Generating 3D Map of Gale Crater Using Unity3d and C#

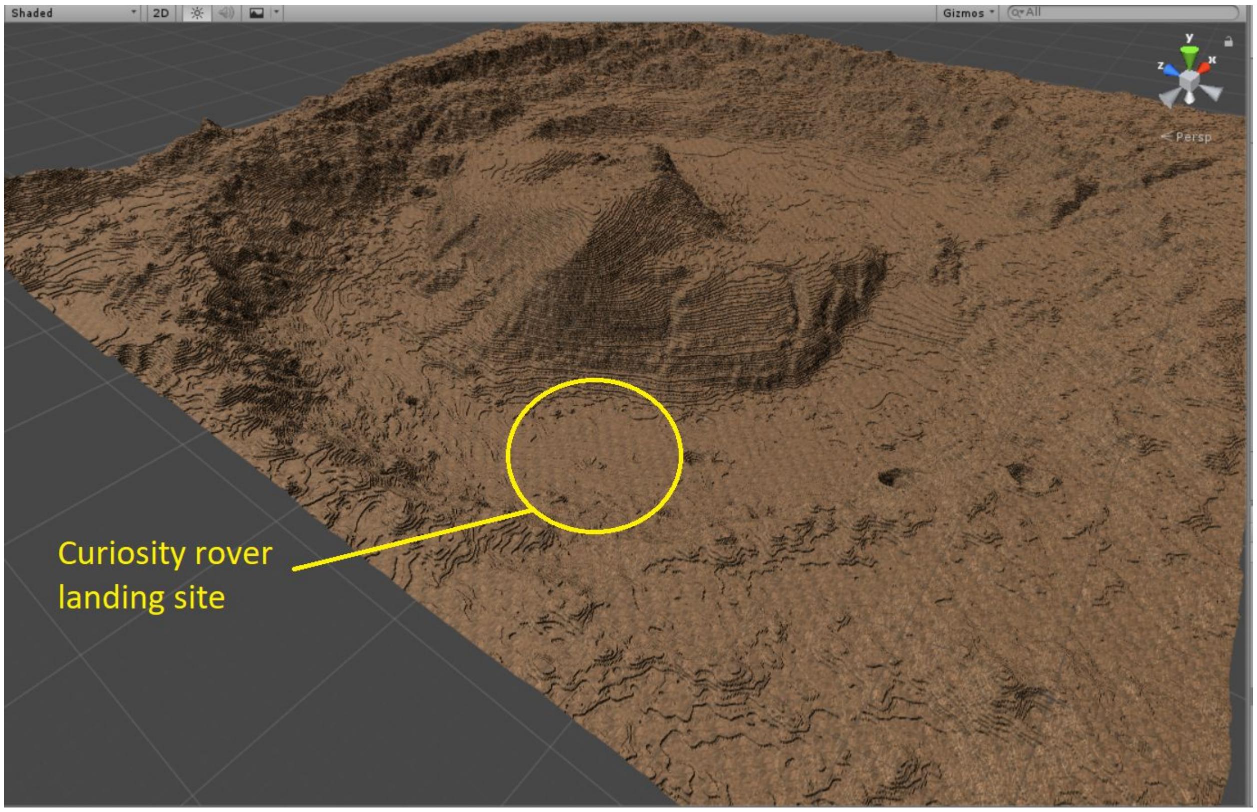
By Oguzcan Adabuk





Introduction

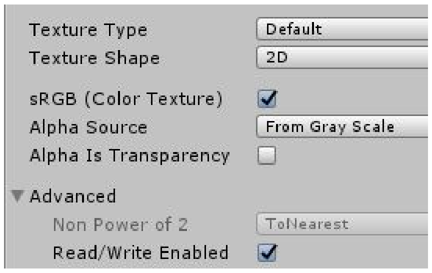
My purpose in creating this project is to backup my recent application to the 3D Graphics Software Engineer position, with a practical, hands on approach. It is also a weekend project that has been a lot of fun. The program generates the crater model during runtime. Each vertices' coordinates are calculated from a heightmap that is a PNG image and this data gets converted into a 3D model. Then the mesh triangles are calculated and visible surfaces are created. Below you will find the detailed explanation. If you have any questions please don't hesitate to contact.
Email: adabuk@gmail.com
Phone: 206 330 7253



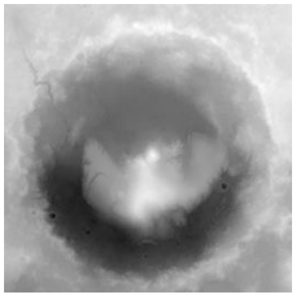
Project files can be downloaded from here.
You can watch a video of the generated map here.

1. Getting the Coordinate Data

The model of Gale Crater in the picture above was dynamically generated using Unity3d's *Mesh* class. First, I searched for a height map online. I found a good heightmap at this link. The actual image is 2883x3395 and it is a PNG file. I changed the image size to 4096x4096 in Photoshop to make the image have dimensions of power of two.



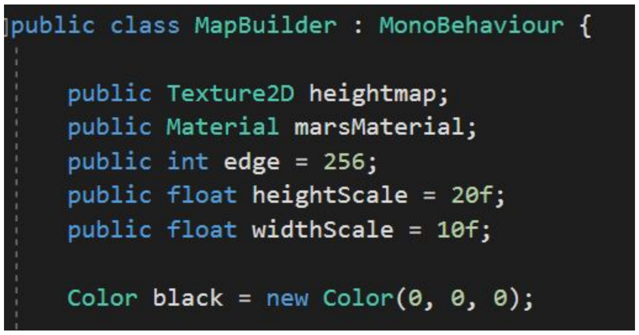
Texture Type	Default
Texture Shape	2D
sRGB (Color Texture)	✓
Alpha Source	From Gray Scale
Alpha Is Transparency	
Advanced	
Non Power of 2	ToNearest
Read/Write Enabled	✓



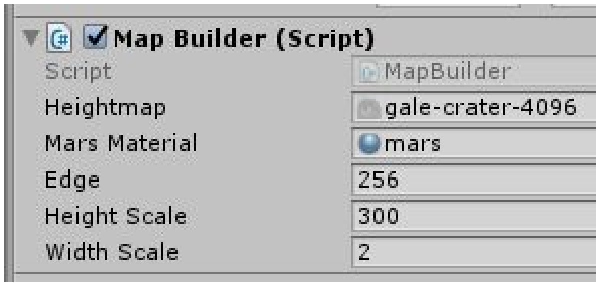
2. MapBuilder Class Variables

I created a C# script and named it *MapBuilder.cs*. This class has several public variables.

- *heightMap* is of *Texture2D* type, it contains the actual heightmap texture of Gale Crater.
- *marsMaterial* is of *Material* type, it has the material for the mesh.
- *edge* is integer type, it is the length of edge of each piece of the model.
- *heightScale* is a float, it is a constant value to change height of the model.
- *widthScale* is a float, it is a constant value to change width of the model
- *black* is of Color type, it has RGB values of the black color.



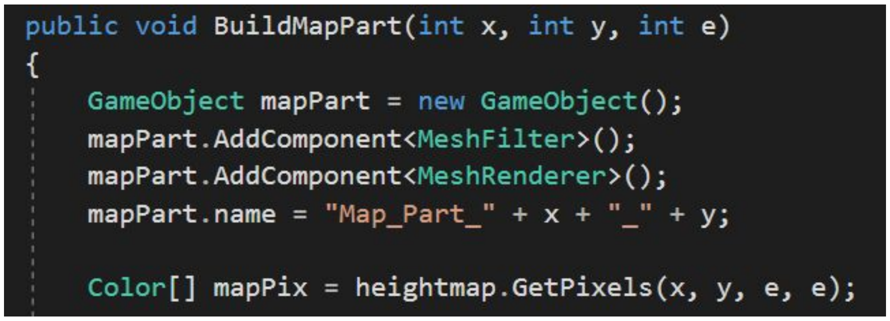
```
public class MapBuilder : MonoBehaviour {

    public Texture2D heightmap;
    public Material marsMaterial;
    public int edge = 256;
    public float heightScale = 20f;
    public float widthScale = 10f;

    Color black = new Color(0, 0, 0);
```



Map Builder (Script)

Script	MapBuilder
Heightmap	gale-crater-4096
Mars Material	mars
Edge	256
Height Scale	300
Width Scale	2

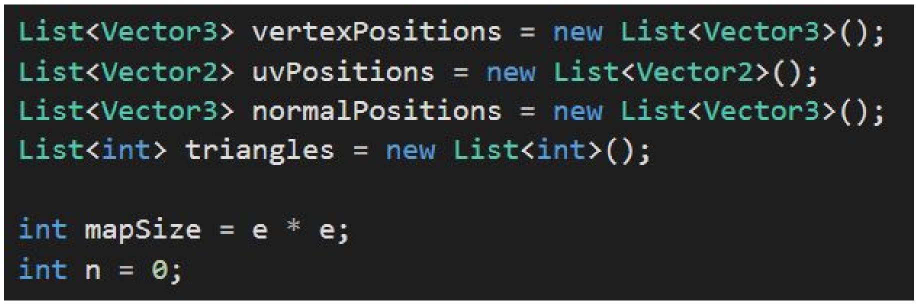
When the program starts, the *Start* method is called. *Start* method runs nested loops and *BuildMapPart* method inside the inner loop. Here the heightmap texture is divided into (edge x edge) chunks and each part of the model is created within *BuildMapPart* method.
BuildMapPart takes three parameters, these are x and y coordinates of heightmap texture region, and the edge length.
BuildMapPart starts by creating an empty *GameObject* type variable, *mapPart*. Necessary components for rendering a mesh added to the *mapPart* object. These are *MeshFilter* and *MeshRenderer* components.
Then the map part is given a unique name for easier identification inside the Unity3d editor, the name has the format *Map_Part_Xcoordinate_Ycoordinate*.
Next, using the *GetPixels* method of Unity3d's *Texture2D* class, color data of each pixel from the that chunk of height map is assigned to the *mapPix* color array. After the color data is retrieved, generating the model mesh is the next step.



```
public void BuildMapPart(int x, int y, int e)
{
    GameObject mapPart = new GameObject();
    mapPart.AddComponent<MeshFilter>();
    mapPart.AddComponent<MeshRenderer>();
    mapPart.name = "Map_Part_" + x + "_" + y;

    Color[] mapPix = heightmap.GetPixels(x, y, e, e);
```

3. Generating Vertices

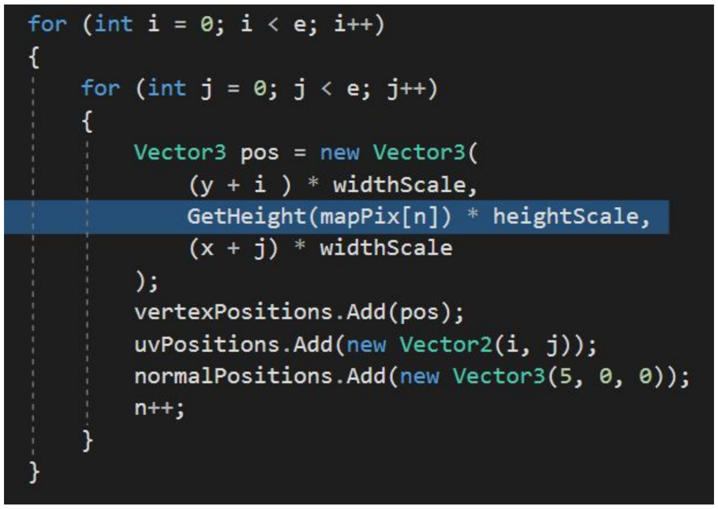
Four generic lists are created to store mesh related data. These are *vertexPositions* of *Vector3* type, *uvPositions* of *Vector2* type, *normalPositions* of *Vector3* type and *triangles* of int type. Two other integer values needed are *mapSize* and *n*.



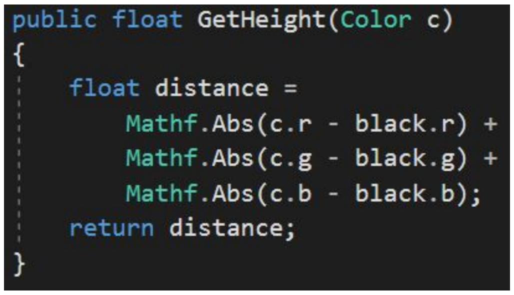
```
List<Vector3> vertexPositions = new List<Vector3>();
List<Vector2> uvPositions = new List<Vector2>();
List<Vector3> normalPositions = new List<Vector3>();
List<int> triangles = new List<int>();

int mapSize = e * e;
int n = 0;
```

vertexPositions stores positions of vertices in 3D space. These vertices will makeup meshes. *uvPositions* contains coordinates for UV Mapping, for texturing meshes. Since UV maps are 2 dimensional, *uvPositions* are of *Vector2* type. *normalPositions* contain normal data of each

vertex, normals determine how each vertex reacts to lighting in the scene. triangles contain combinations of vertices that make up surface triangles. Unity3d *Mesh* objects contain a triangles array of int type. The array contains the indices of vertices. Consecutive trios of vertices make up a triangle. Every triangle is rendered as a surface. For e.g. if the triangles array contain 0, 1, 5, this means that vertices with index 0, 1 and 5 create a triangle together and rendered as a surface. How each triangle is calculated will be explained in more detail in the next section.
Next step is to calculate positions of each vertices in the 3D space.
A nested array is used to assign x and z positions of each vertex. Each vertex is created based on each pixel from a chunk of heightmap texture. *pos* variable is used to store position of vertices before they are added to the *vertexPositions* list to be passed to mesh later. The x and z coordinates comes from pixel index on heightmap, here represented with i and j variables of nested loops. The x and z coordinates are multiplied with *widthScale* to modify how wide each vertices spread across the 3D space. A vertex's height is calculated using the *GetHeight* method and multiplying it with *heightScale*. *GetHeight* method takes color data of the relevant pixel from the heightmap and compares it to the RGB values of the black color which has RGB values all equal to 0. In a heightmap, black color represents the lowest point of the surface whereas white color represent the highest altitude. Therefore each pixel's color data is compared to black color and the difference is returned as the height.



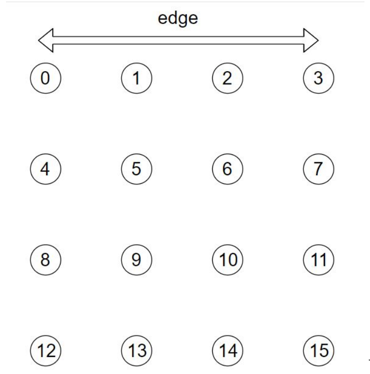
```
for (int i = 0; i < e; i++)
{
    for (int j = 0; j < e; j++)
    {
        Vector3 pos = new Vector3(
            (y + i ) * widthScale,
            GetHeight(mapPix[n]) * heightScale,
            (x + j) * widthScale
        );
        vertexPositions.Add(pos);
        uvPositions.Add(new Vector2(i, j));
        normalPositions.Add(new Vector3(5, 0, 0));
        n++;
    }
}
```



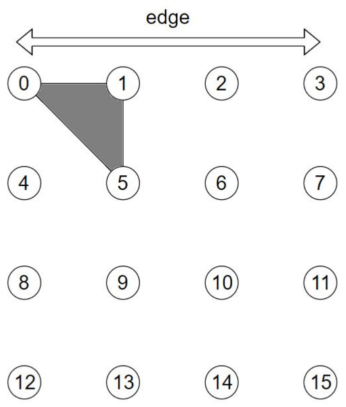
```
public float GetHeight(Color c)
{
    float distance =
        Mathf.Abs(c.r - black.r) +
        Mathf.Abs(c.g - black.g) +
        Mathf.Abs(c.b - black.b);
    return distance;
}
```

4. Triangle Mesh Processing

After vertex positions are calculated, next step is to calculate which vertices make triangles. A triangle is made of three vertices.As mentioned above, Unity3d's *Mesh* class has a vertices integer array that keeps the indices of vertices that make triangles. In order to create the surface, we need to know which vertices make a triangle. Images below and instances of vertices list show how triangles are formed.

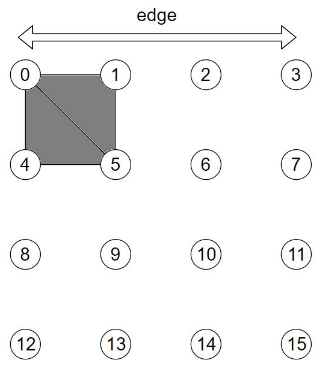


vertices = { }



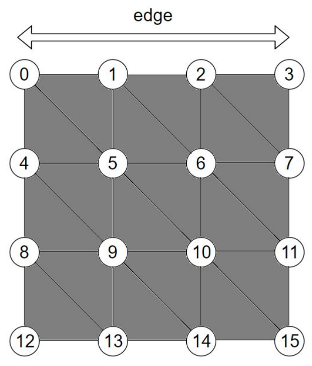
vertices = { 0, 1, 5 }

In the second picture, it can be seen that vertices with indices 0, 1 and 5 make a triangle.



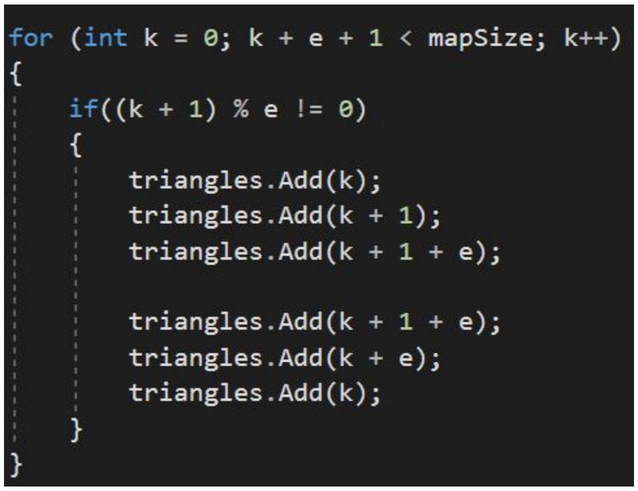
vertices = {0, 1, 5 , 0, 4, 5}

In the picture above, vertices with indices 0,1,5 make a triangle and vertices with indices 0, 4, 5 make another triangle. Together these make a quad.



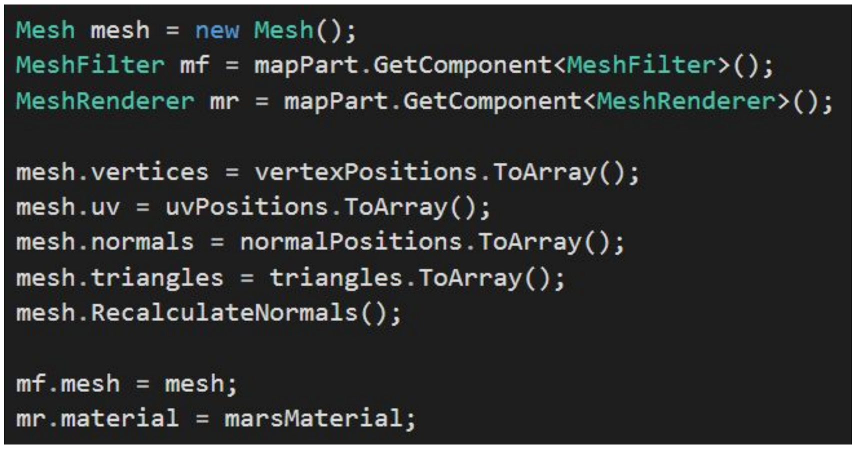
vertices = {0, 1, 5, 0, 4, 5, 1, 2, 6, 1, 5, 6, 2, 3, 7, 2, 6, 7…..}

In this picture it can be seen that a plane mesh is generated by making triangles with right vertices.
In order to create triangles for a large mesh, this can be formulated. Each first triangle is made with i_{th}, $(i+1)_{th}$ and $(i+1+edge)_{th}$ vertices. Second triangle of each quad is made with vertices i_{th}, $(i+edge)_{th}$ and $(i+edge+1)_{th}$ vertices. There should be two things checked. First, the $(i+edge+1)_{th}$ should be less than the map size to prevent overflow. Secondly, the last element in column should not be assigned as i, this will create triangles between last column vertices and first column vertices and it will look awkward and incorrect. To prevent that make sure $(i+1)_{th}$ elements mod of edge length is not equal to 0.



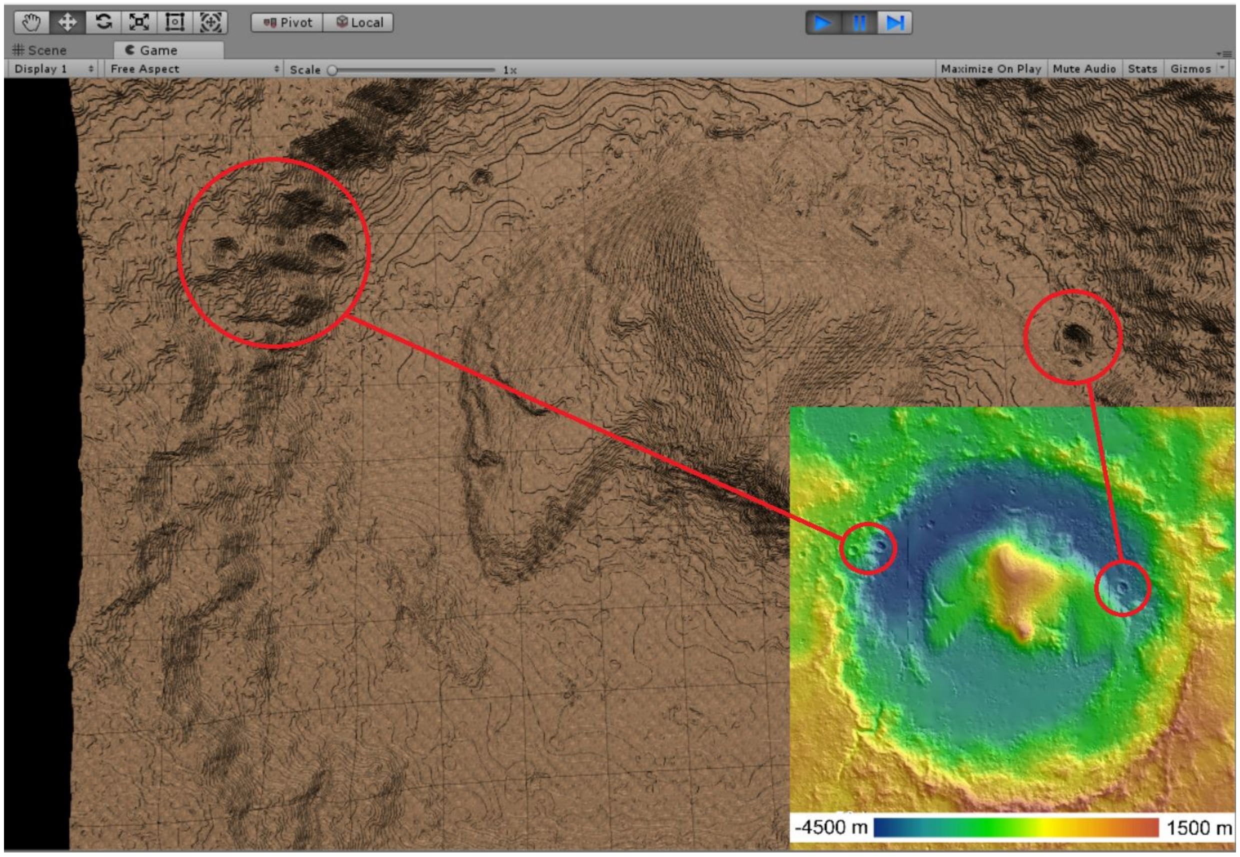
```
for (int k = 0; k + e + 1 < mapSize; k++)
{
    if((k + 1) % e != 0)
    {
        triangles.Add(k);
        triangles.Add(k + 1);
        triangles.Add(k + 1 + e);

        triangles.Add(k + 1 + e);
        triangles.Add(k + e);
        triangles.Add(k);
    }
}
```

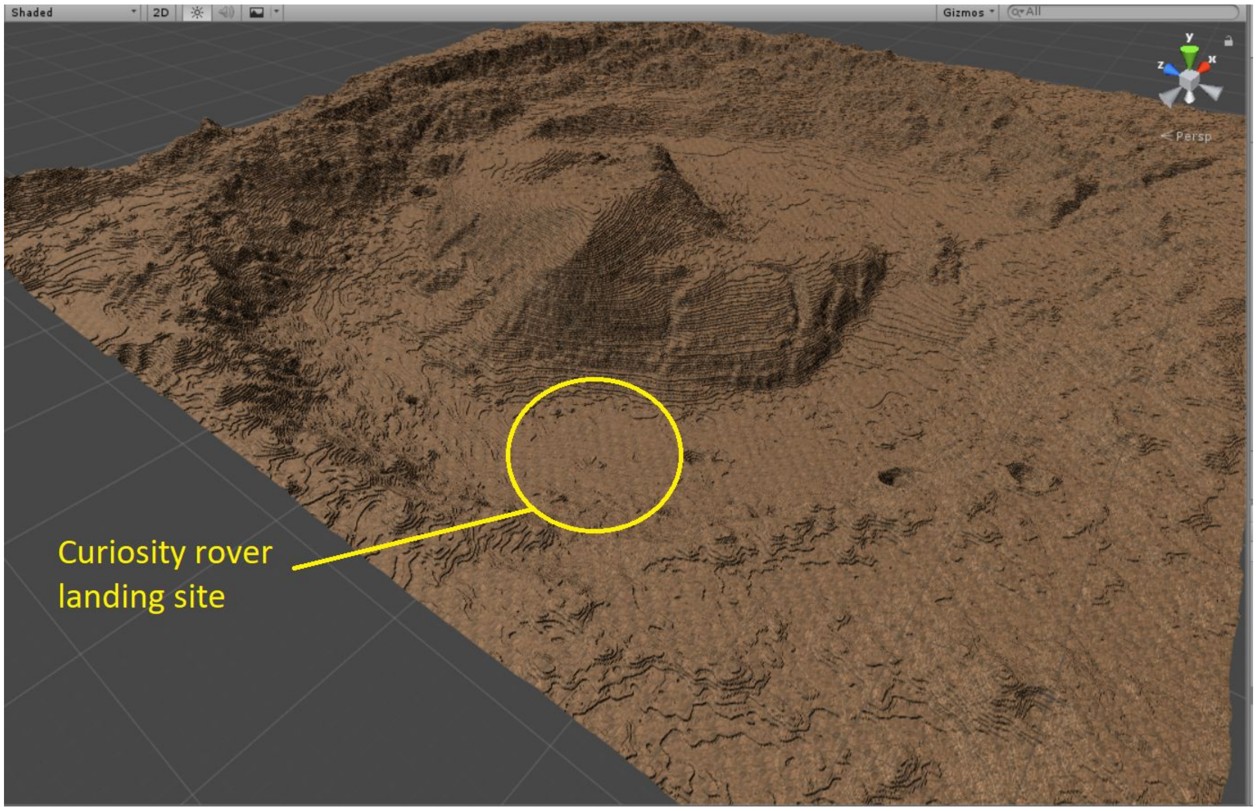
After the triangles are calculated, a mesh object is created and previously generated vertex, uv, normal, and triangle data is passed in to the mesh object. Unity3d will use these data to render the mesh. Then the material for mesh is assigned and its normals are calculated using the *Mesh.RecalculateNormals()* method to have a mesh that responds better to scene lighting.

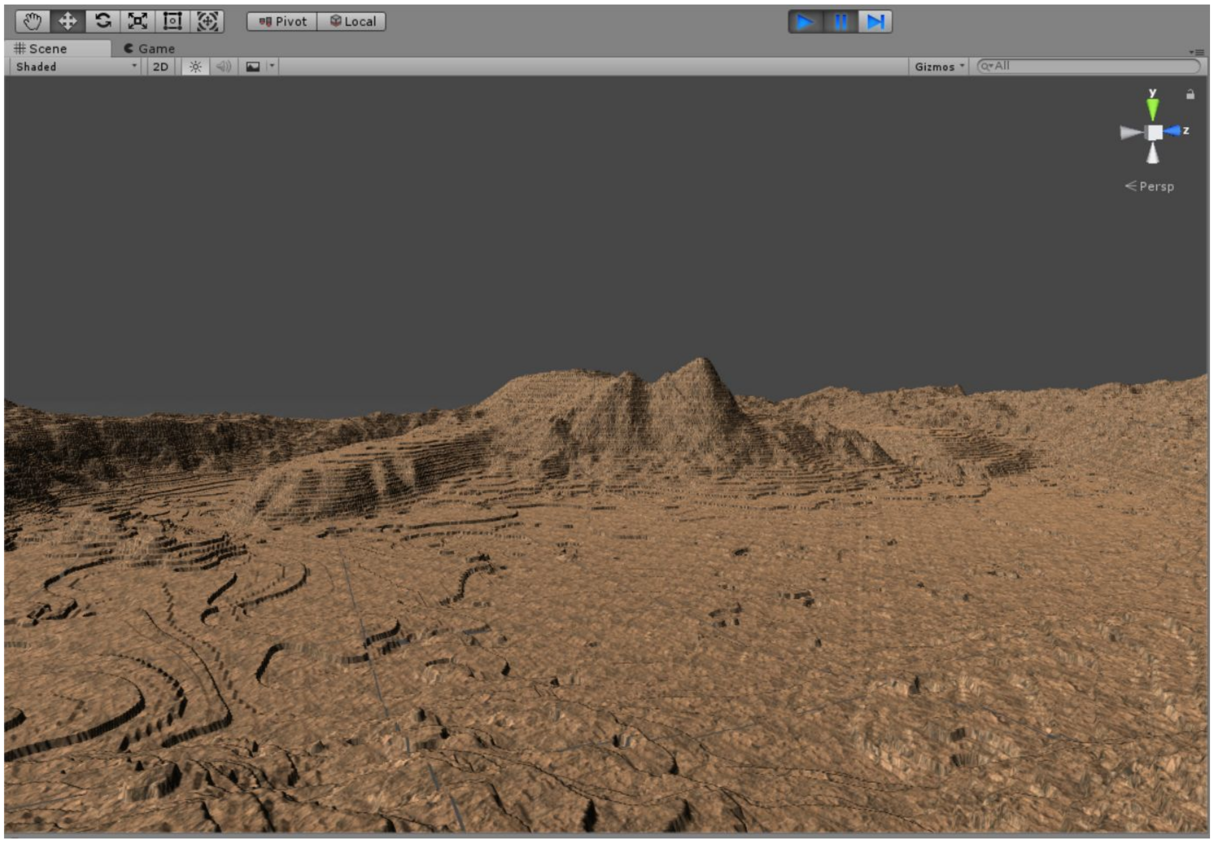


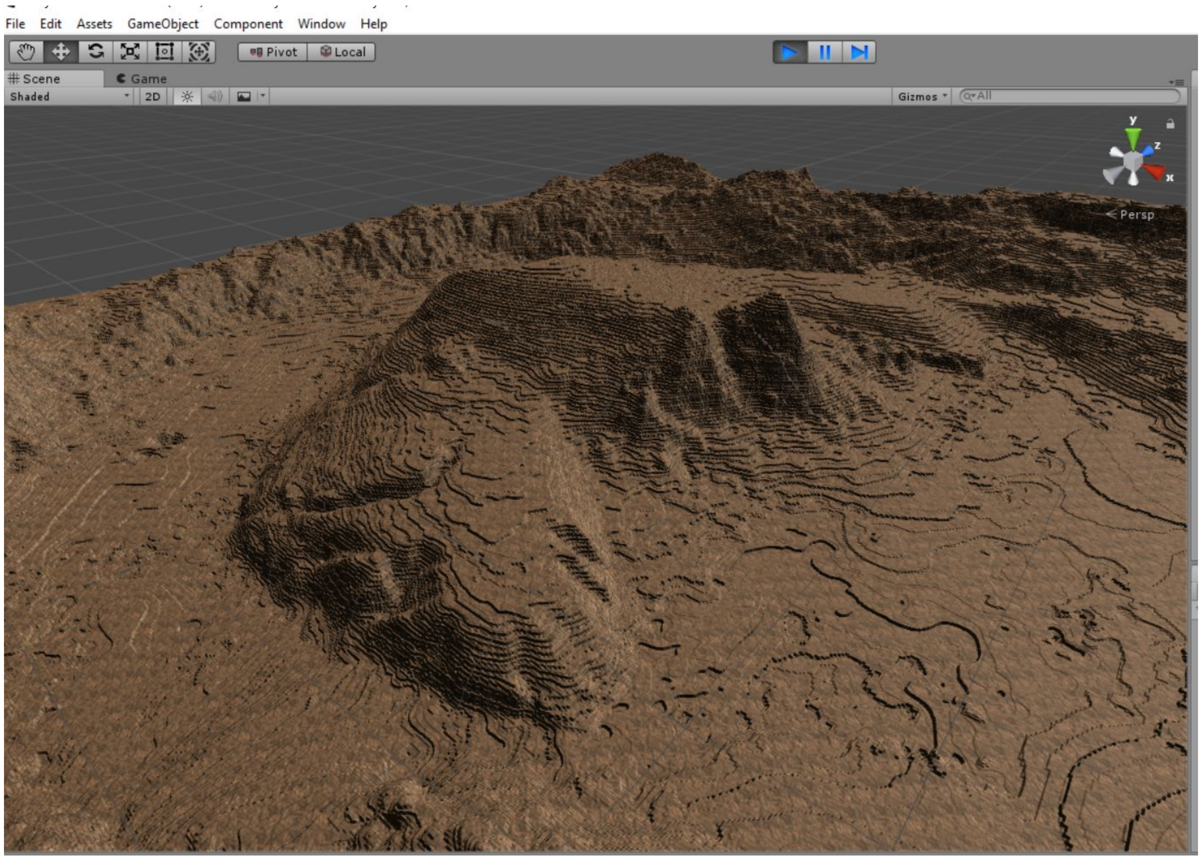
```
Mesh mesh = new Mesh();
MeshFilter mf = mapPart.GetComponent<MeshFilter>();
MeshRenderer mr = mapPart.GetComponent<MeshRenderer>();

mesh.vertices = vertexPositions.ToArray();
mesh.uv = uvPositions.ToArray();
mesh.normals = normalPositions.ToArray();
mesh.triangles = triangles.ToArray();
mesh.RecalculateNormals();

mf.mesh = mesh;
mr.material = marsMaterial;
```

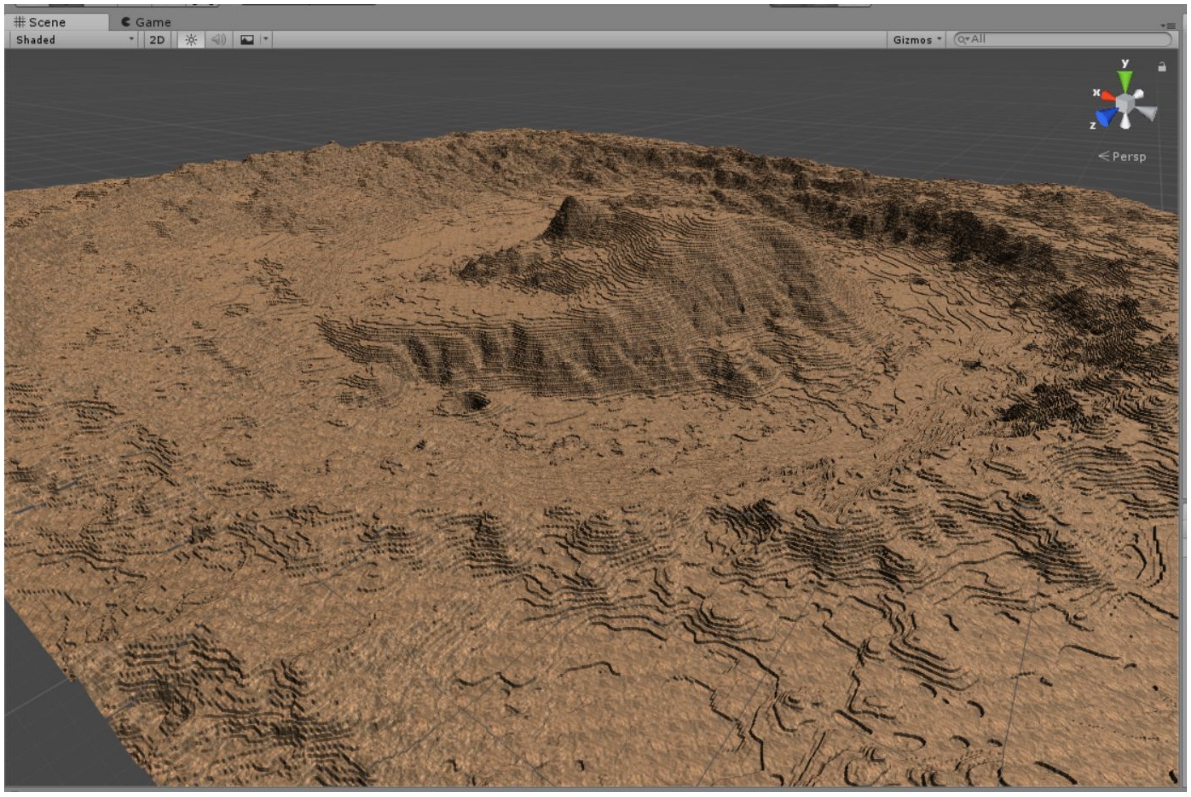
5. Results

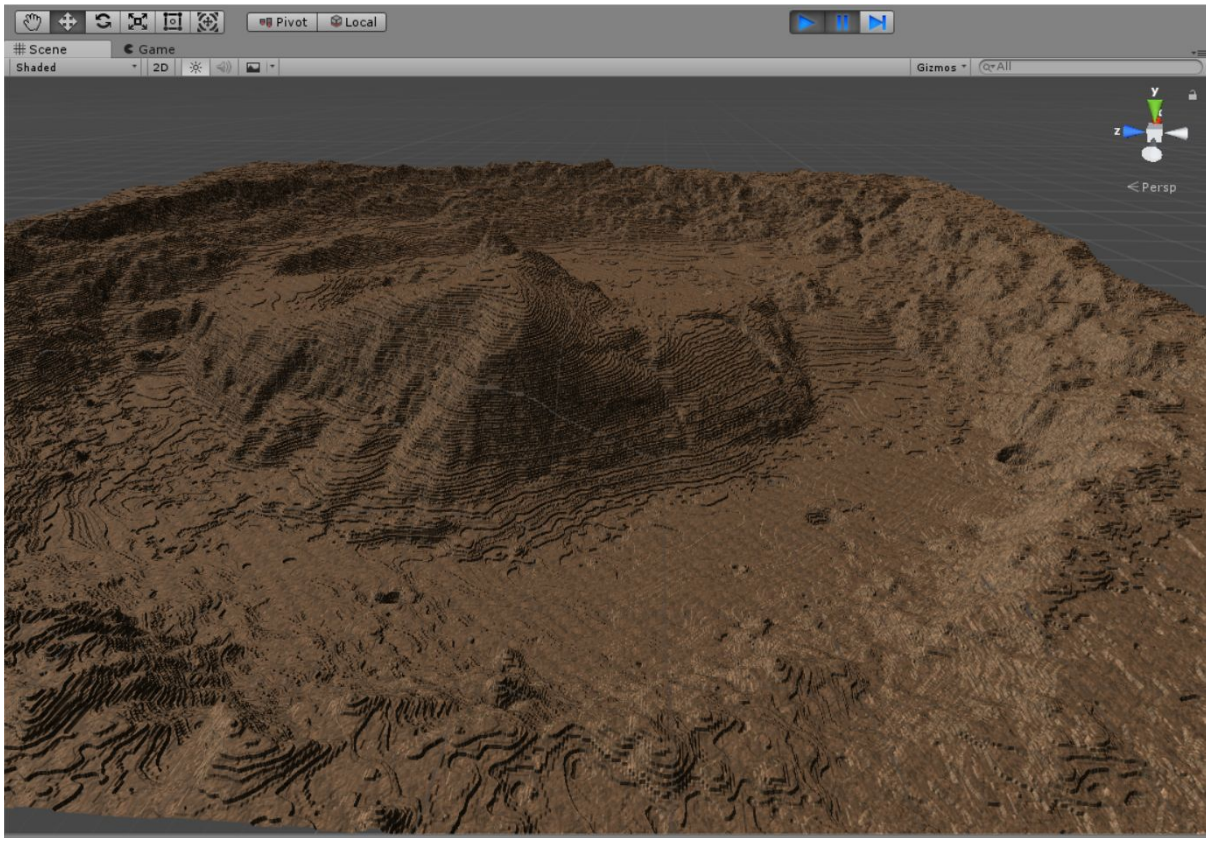

Pivot
Local
Scene
Game
Display 1
Free Aspect
Scale
1x
Maximize On Play
Mute Audio
Stats
Gizmos
-4500 m
1500 m

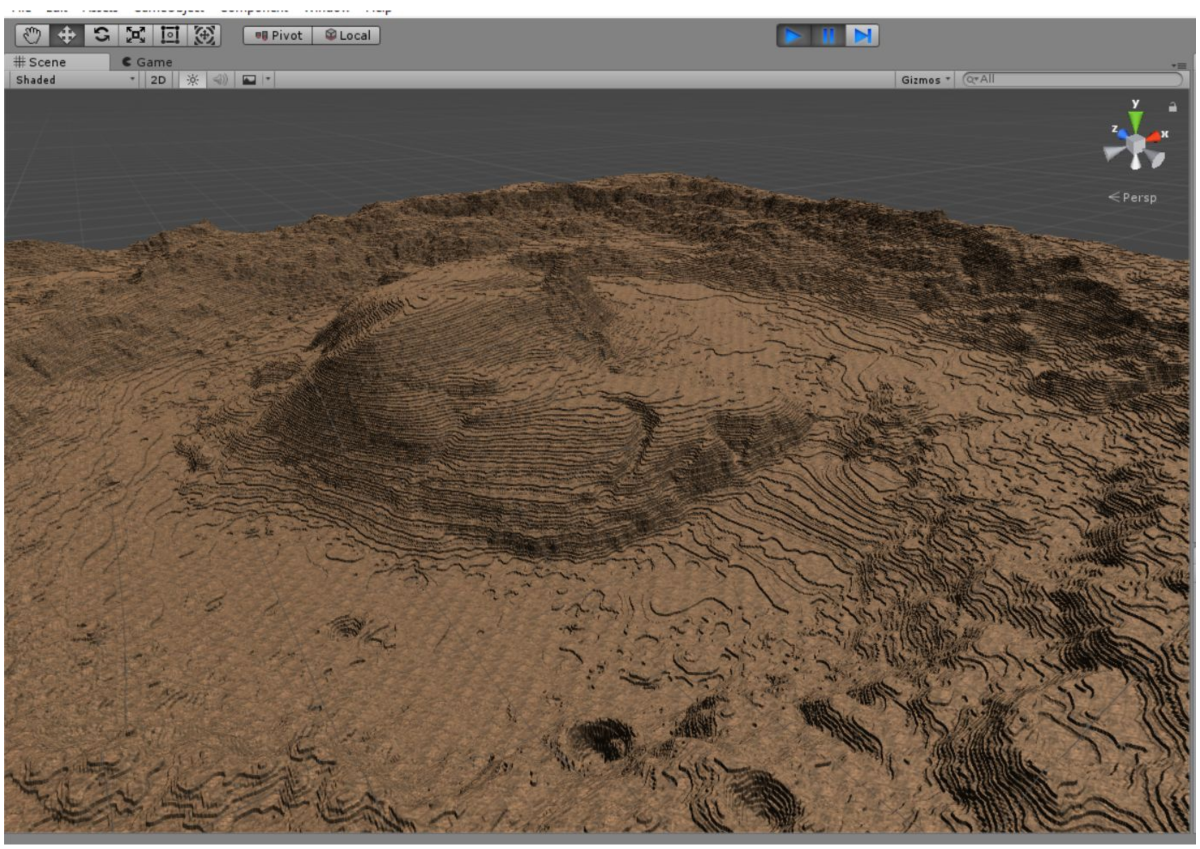

Shaded
2D
Gizmos
All
y
z
x
Persp
Curiosity rover landing site


Pivot
Local
Scene
Game
Shaded
2D
Gizmos
All
y
z
Persp


File
Edit
Assets
GameObject
Component
Window
Help
Pivot
Local
Scene
Game
Shaded
2D
Gizmos
All
y
z
x
Persp


Scene
Game
Shaded
2D
Gizmos
All
Persp


Pivot
Local
Scene
Game
Shaded
2D
Gizmos
All
Persp


Pivot
Local
Scene
Game
Shaded
2D
Gizmos
All
y
x
z
Persp