

Tutorial 1: Brute Force

by Victor Saar

Content

Many of today's games use high detail and very huge terrains. To realise this terrain there are several ways. The most common ones are Real-Time Optimally Adapting Mesh (ROAM), Geo-Mipmapping and Level Of Detail (LOD). These algorithms are used to render extremely huge and complex areas. Because of the high performance of modern graphics cards it is also possible to use the brute force method, but only for smaller terrains. This means we draw every triangle no matter if it is far away or close to the viewer.

That's what we want to do in this tutorial. We will render a mesh of 64 to 64 small quads. To do that we have to load the height information first. The easiest way to do this, is to load them from a *.raw* file. The *.raw* file format is not formatted and thus very easy to use. The color values are only from a gray scale image. On the image below you can see the height map. The whiter the pixels are the higher is the terrain at this position.

I made this height map with Paint Shop Pro from Jasc. There are also trial versions of this program, which you can use to create your own height map. Just convert the image to gray scale and save it as a *.raw* file.

The main difficulty is to fill the vertex and index buffer. For the terrain we create a new class, which will include both buffers and additional functions. For that we add two files to the project. One header with the class declaration and one source code file. I will not explain all parts of the code, because the most things are already contents of the Direct3D tutorials. For more information, have a look at them.

cterrain.h

We start with the declaration of the terrain class. The two constants at the beginning define the x and y dimension of the terrain. The class includes three functions. The first is the constructor, which defines the member variables and will call the function, that creates the terrain. This is the second function. There we load the height information from the *.raw* file and fill the vertex and index buffer.

```
#define TERRAIN_X 64 //x dim
#define TERRAIN_Y 64 //y dim

class CTerrain
{
public:
    CTerrain(void); //constructor

    void    CreateTerrain(void); //load file and create buffers

    void    Draw(void); //draw the whole terrain
```

The private member variables usual. We need a vertex and an index buffer and two `DWORD`, which hold the number of vertices in the vertex buffer and the number of primitives of the terrain

```
private:
    LPDIRECT3DVERTEXBUFFER9 m_pTerrainVB; //vertex buffer
    LPDIRECT3DINDEXBUFFER9  m_pTerrainIB; //index buffer
```

```

    DWORD                m_dwTerrainVertices, //vertex count
                        m_dwTerrainPrimitives; //primitive count
};

```

cterrain.cpp

Here is the source code for the terrain class. We start with the constructor. Here we compute the number of vertices and primitives of the terrain and call the create function.

```

CTerrain::CTerrain(void)
{
    m_pTerrainVB = NULL;
    m_pTerrainIB = NULL;

    m_dwTerrainVertices = (TERRAIN_X + 1) * (TERRAIN_Y + 1);
    m_dwTerrainPrimitives = TERRAIN_X * TERRAIN_Y * 2;

    CreateTerrain();
} //CTerrain

```

Here we come to the most important function of the terrain. It loads the height information from the height map and creates the vertex and index buffer out of it. The variables we need are an arrays, which holds the height information, to pointers, that will be used to write to the buffers and a file stream object. For the file stream object we have to include the header *fstream* in *main.h*.

```

void CTerrain::CreateTerrain(void)
{
    unsigned char aHeightMap[TERRAIN_X + 1][TERRAIN_Y + 1]; //height array
    D3DVERTEX* pVertexData; //pointer to vertex
    short* pIndexData; //pointer to index
    ifstream File; //file object

```

Now we load the height values from the file *heightmap.raw*. We open the file and load the values into the array. Finally we close the file.

```

File.open("heightmap.raw", ios::binary); //open file in binary mode

for(unsigned y = 0; y < (TERRAIN_Y + 1); ++y)
{
    for(unsigned x = 0; x < (TERRAIN_X + 1); ++x)
    {
        aHeightMap[x][y] = File.get(); //get height
    }
}

File.close(); //close file

```

We start with filling the vertex buffer. First we lock it using the pointer to `D3DVERTEX` and write the vertices row by row to the buffer. The z value is taken from the height map array and the color is constantly set to white.

```

g_App.GetDevice()->CreateVertexBuffer(sizeof(D3DVERTEX)*m_dwTerrainVertices,
                                        D3DUSAGE_WRITEONLY,
                                        D3DFVF_CUSTOMVERTEX,
                                        D3DPOOL_MANAGED,
                                        &m_pTerrainVB,
                                        NULL);

m_pTerrainVB->Lock(0, 0, (void*)&pVertexData, 0);

```

```

for(y = 0;y < (TERRAIN_Y + 1);++y)
{
    for(unsigned x = 0;x < (TERRAIN_X + 1);++x)
    {
        pVertexData[x + y * (TERRAIN_X + 1)].fX = (float)x;
        pVertexData[x + y * (TERRAIN_X + 1)].fY = (float)y;
        pVertexData[x + y * (TERRAIN_X + 1)].fZ = (float)aHeightMap[x][y]/15.0f;
        pVertexData[x + y * (TERRAIN_X + 1)].dwColor = 0xffffffff; //color
    }
}

m_pTerrainVB->Unlock();

```

The next part creates the index buffer and fills it. As you can see we need 3 indices for each triangle. In this case x is the column and $y * (TERRAIN_X + 1)$ is the row. The indices are all relative to the lower left corner of each small quad. So the first is $v1$ and the second is $v2$, which is in the second row relative to $v1$, so we have to increase x by 1. In the final analysis we go through each quad row by row. The triangles are created with a counter-clockwise order of the vertices, which makes it possible the cull backfacing triangles.

```

g_App.GetDevice()->CreateIndexBuffer(sizeof(short)*m_dwTerrainPrimitives*3,
                                     D3DUSAGE_WRITEONLY,
                                     D3DFMT_INDEX16,
                                     D3DPOOL_MANAGED,
                                     &m_pTerrainIB,
                                     NULL);

m_pTerrainIB->Lock(0,0,(void**)&pIndexData,0);

for(y = 0;y < TERRAIN_Y;++y)
{
    for(unsigned x = 0;x < TERRAIN_X;++x)
    {
        *pIndexData++ = x + y * (TERRAIN_X + 1); //v1
        *pIndexData++ = x + 1 + y * (TERRAIN_X + 1); //v2
        *pIndexData++ = x + 1 + (y + 1) * (TERRAIN_X + 1); //v4

        *pIndexData++ = x + y * (TERRAIN_X + 1); //v1
        *pIndexData++ = x + 1 + (y + 1) * (TERRAIN_X + 1); //v4
        *pIndexData++ = x + (y + 1) * (TERRAIN_X + 1); //v3
    }
}

m_pTerrainIB->Unlock();

} //CreateTerrain

```

The last function only draws the terrain. First we have to set the vertex and index buffer. Then we draw the whole mesh with a call to `DrawIndexedPrimitive()`.

```

void CTerrain::Draw(void)
{
    g_App.GetDevice()->SetStreamSource(0,m_pTerrainVB,0,sizeof(D3DVERTEX));
    g_App.GetDevice()->SetIndices(m_pTerrainIB);
    g_App.GetDevice()->DrawIndexedPrimitive(D3DPT_TRIANGLELIST,
                                           0,
                                           0,
                                           m_dwTerrainVertices,
                                           0,
                                           m_dwTerrainPrimitives);
} //Draw

```

main.cpp

Here we just have to add some small things. First we create a pointer to a `CTerrain` object. We use a pointer, because we first can create the object when Direct3D has been initialized.

```
CTerrain* pTerrain; //pointer to terrain (NEW)
```

Before the Direct3D loop we allocate memory for a `CTerrain` object by using `new`. Then we simply call `Draw()` in the loop. At the end we have to delete the object, so that we don't get a memory leak.

```
pTerrain = new CTerrain; //create terrain object (NEW)

//D3D loop
while(g_App.GetWindowStatus())
{
    if(GetMessage(&Message, NULL, 0, 0))
    {
        if(!IsDialogMessage(g_App.GetWindowHandle(), &Message))
        {
            TranslateMessage(&Message);
            DispatchMessage(&Message);
        }
    }
}

if(g_App.CheckDevice())
{
    g_App.GetDevice()->Clear(0,
                            NULL,
                            D3DCLEAR_TARGET | D3DCLEAR_ZBUFFER,
                            RGB(0,0,0),
                            1.0f,
                            0);

    g_App.GetDevice()->BeginScene();

    pTerrain->Draw(); //draw terrain (NEW)

    g_App.GetDevice()->EndScene();
    g_App.GetDevice()->Present(NULL, NULL, NULL, NULL);
}

delete pTerrain; //delete terrain object (NEW)
```

When you start the program you will look down on a wireframed terrain. Coloring and shading of the terrain will be done in the next tutorials. For more information download the source code or work through the other tutorials.