Honours Project
Multi-Material Procedural Terrain Texturing

Author: Eric Billingsley
Supervisor: Dr. David Mould, School of Computer Science

December 11th, 2009
Carleton University

Abstract

The most common approach to real-time terrain texturing with multiple materials is to interpolate between tiling textures. This has disadvantages in that it can produce obvious tiling, and transitions between materials that do not look natural. Some solutions to these problems exist, but either take up large amounts of memory or disk space, or do not provide enough detail at close range.

The algorithm presented is a novel way of combining multiple procedural textures based on parameters like slope and incline. It renders both realistic and aesthetically pleasing terrain which contains interesting and hard-edged transitions between multiple materials. Close and distant terrain can each be rendered well with the same shader. Additionally, parallax and self-occlusion were implemented using Steep Parallax Mapping. The program runs at an acceptable frame-rate

Although the project was a success, there are some problems. Combining Steep Parallax Mapping with the procedural texturing approach greatly reduced frame-rates, and there are some visual artifacts present. Nonetheless, the shader created could very well be used in real-time applications as an alternative approach to terrain rendering.

Acknowledgements

I followed a tutorial on http://www.ziggyware.com entitled “Perlin Noise on the GPU” by Patrick McCarthy in order to get Nvidia's GPU Perlin Noise working in C#. The website is no longer in operation.

Thanks to Dr. David Mould for feedback, suggestions and guidance.

Table of Contents

List of Figures

List of Tables

1. Introduction

Rendering terrain in real time is very important for many computer applications such as games and simulations which are set in an outdoor environment. In modern games, the terrain is rendered on the graphics card as a series of textured triangles.

Typically, in games, the texturing is done by interpolating between the colour values of different textures. For example, a tiling grass texture can be used for most terrain, and interpolated with a tiling rock texture where the terrain should be rocky. Many methods exist for doing this, such as proposed by Tim Jenkz [Jenkz, 2005]. Simply interpolating between textures works, but is not ideal, as it has two problems: First, it is obvious that textures are tiling when they are a sufficient distance from the viewer. Second, the interpolation sometimes makes transitions between materials look strange, as one texture simply fades into another; there are no hard edges between textures.

The goal of this project was to implement a procedural texturing solution which solves both of these problems, and which would be combined with a parallax and self-occlusion technique to make the terrain look more realistic. It should be able to render multiple materials, such as sand, rock, and grass. For the project to be a success, it must have a realistic and aesthetically pleasing appearance at both small and large distances. The performance goal of the project is to run at 30 frames per second on an MSI GX-630 laptop, at a resolution of 1280 by 720.

Several approaches exist which address one or both of the issues presented by the texture interpolation method mentioned above. Texture Virtualization [Waveren, 2009], introduced by Id Software, present a solution to both of the problems by storing an extremely large terrain texture on the hard drive and streaming it into memory as needed. The downside to this method is that it requires a very large amount of memory and disk space.

Procedural texturing offers a solution to the tiling problem. This involves synthesizing textures based on mathematical functions such as Perlin Noise [Perlin, 2002] or Worley Noise [Worley, 1996]. Perlin noise is a continuous function which is random but which is also “smooth”, with all of its features being roughly the same size. Worley noise creates more rigid, cell-like patterns which can be used to synthesize textures such as rocks, among others.

In the game engine developed for Infinity: The Quest for Earth, a technique called Per-Pixel Procedural Texture Splatting [Brebion, 2005] was used. This method involves fading between procedurally generated textures on a per-pixel basis, using lookup textures to determine the weight given to each procedural texture. The lookup textures are indexed based on the slope and incline of a given pixel. This technique works very well for terrain viewed at a distance, but doesn't provide much detail for very close terrain.

In the following chapters, the approach taken for my project, and the results achieved are described.

2. Methodology

2.1. Base Terrain Generator

A base terrain generator was needed to generate the geometry onto which the procedural texturing shader would be applied. As the main focus of the project is on the texturing and fragment shading, this step needed to be implemented in a fast and simple way, that would create terrain with enough variety to effectively demonstrate the pixel shader.

A simple level-of-detail system was implemented so that terrain could be rendered at long distances without a large performance penalty. This was done by generating a grid of vertices, shown in Fig. 1, with the resolution of the grid doubling periodically toward the centre. The grid is centred on the camera in the X and Z dimensions each time the user moves a certain distance.

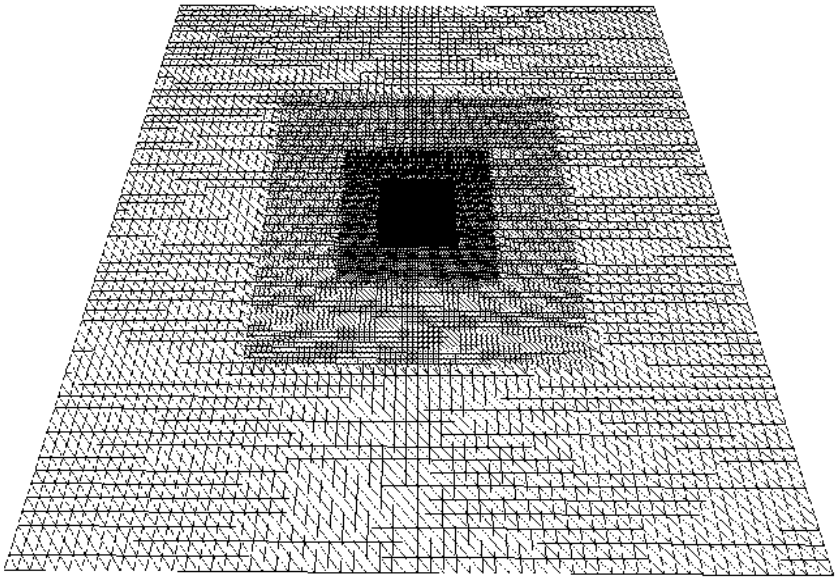

FIGURE 1: The level-of-detail grid with 4 levels of detail. The area which appears to be solid black is the highest level of detail.

There were several apparent options for generating the terrain. The first option was to generate it based on a height map loaded from an image. Another option would be to generate a texture procedurally, and use this as the height map. A third option would be to determine the height of each vertex procedurally, each frame, in the vertex shader; this is the approach that was chosen, as it was the most flexible and easy to implement. Also, this method generates terrain stretching on as far as the user could possibly travel.

To determine the height of a vertex, Ken Musgrave's “Large-Scale Model for Terrain” algorithm [Musgrave, 1993], which produces what are commonly called multifractals, was implemented in a vertex shader. This algorithm generates realistic-looking terrain based on several parameters: octaves, spectral exponent, lacunarity, offset and threshold. Their values were chosen based on trial and error, until a suitable terrain topology for showcasing the fragment shader was found. Their chosen values are, respectively, 8, 0.75, 2, 1.5 and 0.9. A rendering of terrain based on a multifractal with the above parameters is shown in Fig. 2. The multifractal algorithm requires a base noise function, and for this, Nvidia's implementation of Perlin Noise on the GPU [Green, 2005] was used. This implementation had to be modified, as it originally contained only three and four-dimensional Perlin noise, instead of the two-dimensional noise needed. This allows for the full calculation of terrain height in the vertex shader, given only the X and Z coordinates.

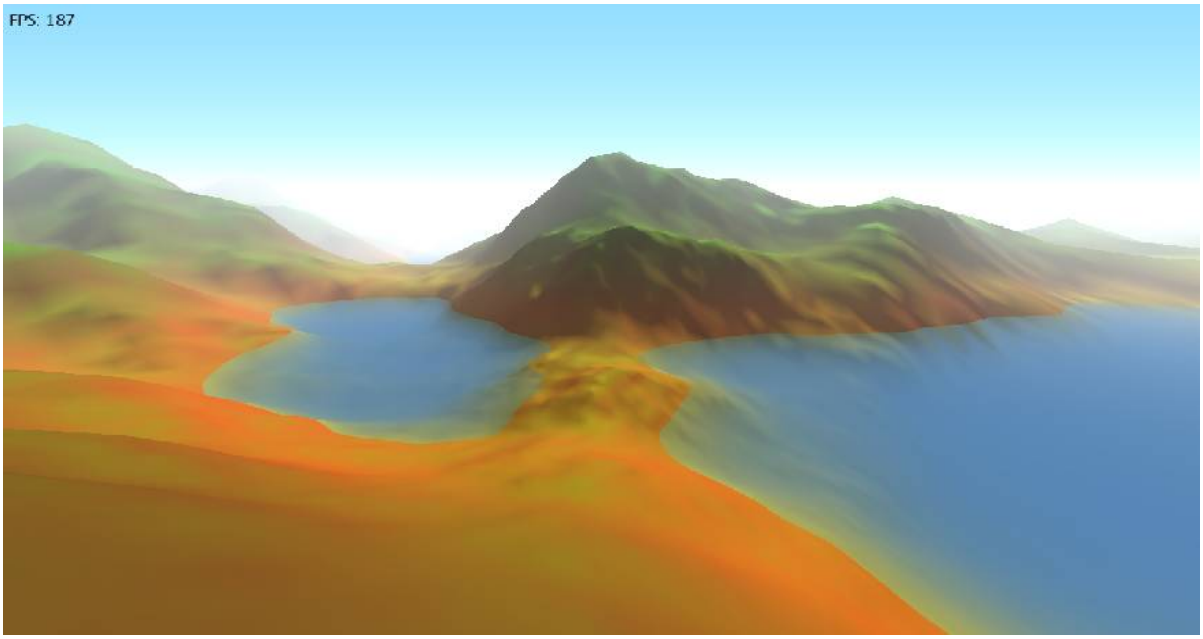

FIGURE 2: A screenshot of the project, showing multifractal terrain coloured on a per-vertex basis.

The normal, binormal and tangent vectors for each vertex are also calculated in the vertex shader. This is done by calculating the height of the terrain at two extra points. The first extra point is offset by a small amount on the X axis, while the second extra point is offset on the Z axis. The binormal is found by normalizing the difference between the first extra point and the original height, whereas the tangent is found by normalizing the difference between the second point and the original height, i.e. the binormal is the slope along the X axis, and the tangent is the slope along the Z axis. Since the normal is perpendicular to both the tangent and binormal, it is calculated by taking the cross product of these two vectors. With the normal, binormal and tangent vectors, the tangent space matrix is constructed and used to transform any vectors sent to the pixel shader into tangent space.

The final thing that the vertex shader does is calculating distance fog and water fog, the effects of which can be seen in Fig. 2. The distance fog is calculated based on the distance from the vertex to the camera, with exponential squared fall-off. A water level is specified in the program, and any point lower than this is considered to be underwater. The water fog is calculated by finding the distance which the ray from the camera to the vertex travels through the water. Exponential-squared fall-off is applied on this distance. Eq. (1) gives the distance which the ray travels through water.

$$distance = \sqrt{(w-y)^2 + \left(\sqrt{(x-CamX)^2 + (Z-CamZ)^2} \times ((w-y)/(CamY-y))\right)^2} \quad (1)$$

where w is the water level,

x, y, z are the components of the vertex's position, and

$CamX, CamY, CamZ$ are the components of the camera's position.

2.2. Multi-Material Procedural Texturing

When deciding how to implement procedural texturing, an important aspect to consider was which base noise function to use, and how it should be calculated. While Perlin noise can be used to synthesize to a wide variety of materials, Worley noise can give better results for certain materials such as rock. Another consideration was whether the noise function should be calculated in the pixel shader, or whether it should be sampled from a pre-rendered texture; using a texture would be a more efficient approach. In the end, Perlin noise was chosen, as it could be rendered with very high frequency into a texture, taking advantage of the GPU's built-in linear interpolation to smooth it out. This way, the texture could be scaled to be very large, and tiling would be unnoticeable. Representing Worley noise in a texture would not work as well, as it contains straight lines and would need to be represented with a much lower frequency. Reading from a texture instead of calculating the noise values on the GPU was chosen for increased performance. In addition to the Perlin noise texture, a uniform noise texture, where the value of each pixel is completely random, was used to represent finer detail. The Perlin noise and uniform noise images are shown in Fig. 3.

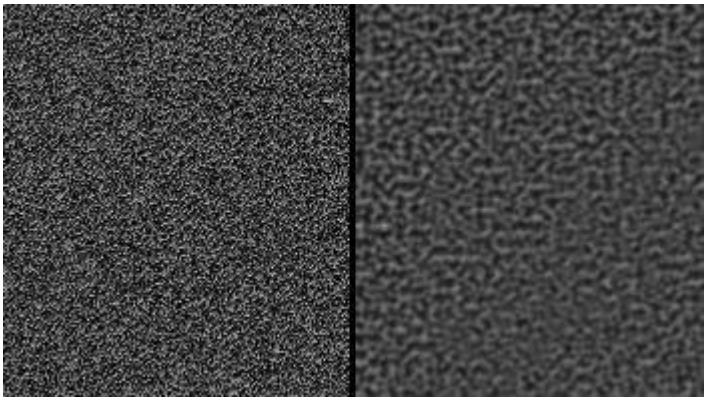

FIGURE 3: A section of each of the procedural noise textures. Only the channel representing height is shown here. Uniform noise is shown on the left, and Perlin noise is shown on the right.

One of the main goals of the procedural texturing was to be able to render multiple materials in a single pixel shader. To determine which material would be rendered at a given pixel, each material needed to be given a different weight. This was done using a two dimensional lookup texture, shown in Fig. 4, which I will refer to as the material mapping texture. The texture would be indexed based on the elevation and incline of the point. The different colours of the texture represent the weights of each material for a given elevation and incline.

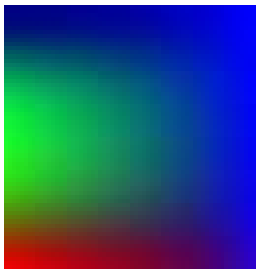

FIGURE 4: The material mapping texture. The x-axis represents incline and the y-axis represents elevation. Red is sand, green is grass, and blue is rock.

The algorithm to determine the material to be rendered at a given point consists of figuring out the height that each material (dirt, sand, grass or rock) would be at for the given point, and selecting the material which is highest. This algorithm is an original creation, and not based on anything I have previously seen. Three texture samples are taken: two of the Perlin noise texture at different scales, and

one of the uniform noise texture. A material's height is determined based on a number of factors. The important factors are the texture multiplier, offset, and scaling parameters. Tables 1 and 2 show the parameter values as defined in the current implementation.

Material	Perlin 1 Multiplier	Perlin 2 Multiplier	Uniform Multiplier	Offset
Dirt	0.3	0.2	0.03	1
Sand	0.5	0.1	0.025	0
Grass	0	0	2	-0.7
Rock	0.8	2	0.06	-1.5

TABLE 1: The material function matrix.

Material	Scaling
Dirt	1
Sand	1
Grass	0.5
Rock	3

TABLE 2: The scaling values for each material.

First, the three texture samples are combined together based on the material's “material function”. This is a 4-dimensional vector with the first three entries corresponding to the weights applied to the three different texture samples. The last entry represents an offset, which is added to the result. The reason that this is represented as a 4-dimensional vector is so that the vectors for all four materials can be combined into a 4x4 matrix, as shown in Table 1. The matrix can then be multiplied with a vector containing the texture samples to give a 4-dimensional vector containing the raw height of each material.

Once we have the raw height for a given material, it needs to be adjusted based on the weight retrieved from the material mapping texture. This is done by multiplying this weight by the material's

scaling value, and adding this to the height. This is the material's actual height. Eq. (2) encompasses all of the previous steps. Beyond that, certain material's heights are adjusted based on extra factors, to give different effects such as ridges.

$$height = (V_1 \cdot M_1 + V_2 \cdot M_2 + V_3 \cdot M_3 + Offset) + Weight \times Scaling \quad (2)$$

where V_1, V_2, V_3 are the three sampled texture values,
M_1, M_2, M_3 are the material's texture multipliers,
Weight is the value retrieved from the material mapping texture, and
Scaling is the material's scaling value.

Now that we have the height for all of the materials, the material with the maximum height is chosen to be rendered. Each material has an associated colour, which is taken and then adjusted based on the current height and the value of the uniform texture, to give a slightly noisy look. For certain materials, such as dirt and sand, the colours are blended based on the material function. The normals are determined in the same way as the vertex shader: by sampling the height of the material at two extra points, and taking the cross product of the resulting vectors. Lighting calculations are then done based on the resulting normal. Fig. 5 shows terrain rendered with the approach described above. Because the normals are calculated, a normal map is not needed, so any texture could be substituted for the noise textures.

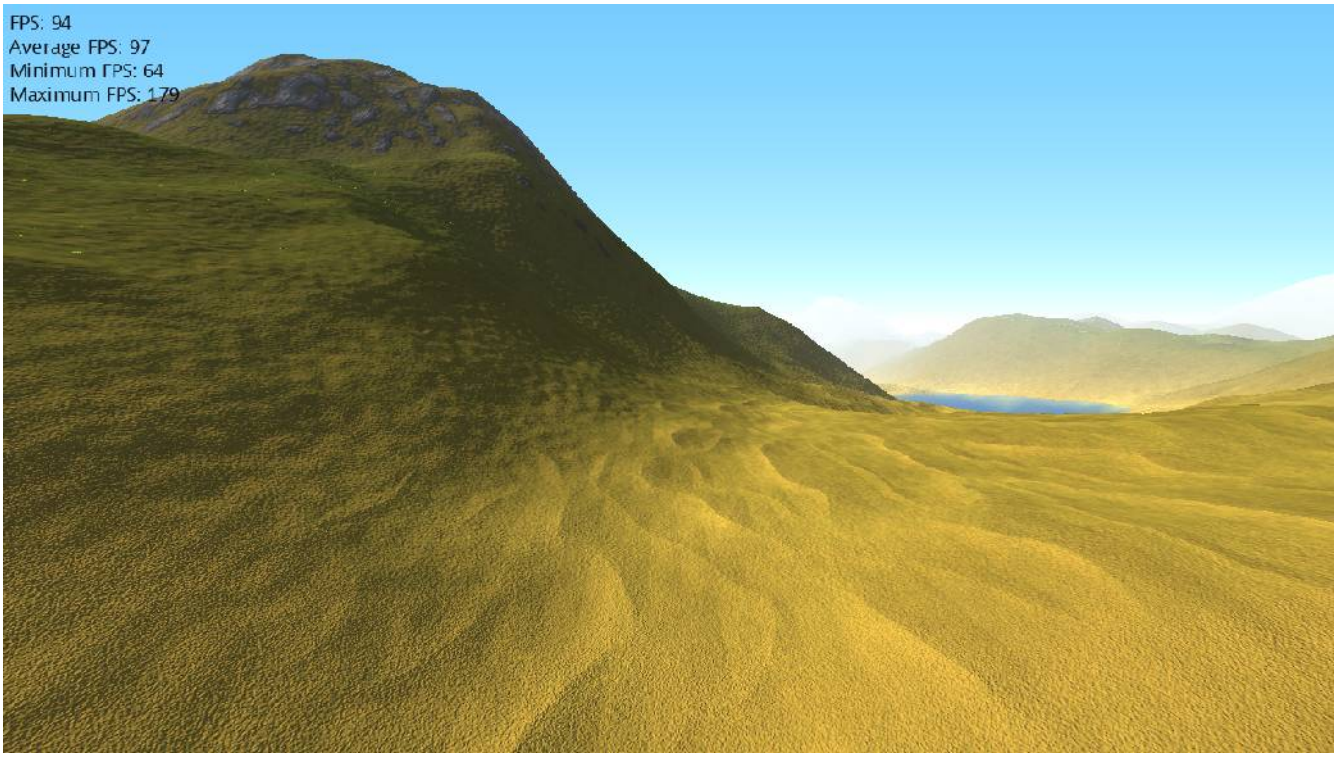

FIGURE 5: A screenshot of the project showing terrain rendered with the Multi-Material Procedural Texturing shader.

An important aspect to consider is mip-mapping, which is the use of lower resolution textures at a distance to reduce aliasing. Because the uniform texture is sampled at a high frequency, terrain which is far away looks extremely noisy if there is no mip-mapping. To correct this, mip-maps were generated for the uniform noise texture. However, if the mip-maps are generated by linearly scaling down the texture, the terrain can look quite strange; this is because the heights of a material are averaged as the texture is scaled down, so the peaks of that material may be removed and will no longer be visible at a distance. This is an especially serious problem with rendering the grass, as its height is based solely on the uniform noise texture. The solution to this, which was used for the uniform texture in the project, was to use nearest-neighbour scaling on mip-maps. In this way, the ratio between chosen materials is preserved.

2.3. Self-Occlusion and Parallax Mapping

Several approaches exist for achieving parallax and self-occlusion on a textured triangle. This makes rendered surfaces appear more three-dimensional, as features of the surface seem to move properly as the eye moves, unlike a traditional texture-mapped triangle. One of the ways to achieve this is through a multi-pass technique, such as the one described by [Lengyel et al., 2001]. Using this method, multiple "shell" textures are rendered at different elevations, and "fin" textures are rendered at the geometry's edges. This technique produces correct silhouettes and works easily with curved surfaces, but requires extra vertex processing because the entire scene must be drawn multiple times. Additionally, it requires extra geometry and texture information to render the "fin" textures.

Another way to achieve this effect is through a single-pass technique which traces a ray against a height map. Steep Parallax Mapping [McGuire and McGuire, 2005], Parallax Occlusion Mapping [Tatarchuk, 2005], and Relief Mapping [Oliveira et al., 2000] are three similar techniques which take this approach. The downside to these approaches is that curved surfaces are not rendered accurately. More advanced techniques, such as the one proposed by Oliveira and Policarpo [Oliveira and Policarpo, 2005], render curved surfaces correctly, but only for convex geometry.

Steep Parallax Mapping was the chosen technique for this project. It was chosen over the multi-pass technique because a lot of heavy processing is done in the vertex shader, as described in chapter 2.1, and transforming all of the vertices multiple times would be very expensive. Additionally, a way of rendering a procedural "fin" texture was not clear. Steep Parallax Mapping was chosen over the other single-pass techniques because it is simple, efficient, and was designed with fur rendering in mind, which is very similar to the desired effect for the grass. Existing methods for correctly rendering curved surfaces with a single-pass technique would not have worked correctly, as the terrain geometry is not entirely convex.

Steep Parallax Mapping was implemented based on the example shader provided with the original paper [McGuire and McGuire, 2005]. It works by sampling a height-field multiple times along

the eye vector, until an intersection point is found. However, it needed to be integrated with the existing procedural texturing shader, as Steep Parallax Mapping normally operates on a traditional texture and a height-map. This was done by adding the material mapping calculations to the beginning of the Steep Parallax Mapping shader, and then substituting each height-map lookup with the entire algorithm for finding the current material height, described in chapter 2.2. Because of the way the lookups were performed, the mip-map levels did not work properly with Steep Parallax Mapping, and so they were calculated manually in the shader, based on the logarithm of distance from the camera. Once an intersection is found, the colour is calculated in the same way as usual.

For increased performance, the number of iterations, and thus required texture lookups and calculations, is adjusted based on the viewing angle of the surface. Additionally, the number of iterations is reduced with distance from the camera, and past a certain distance only 1 iteration is performed, as in the original procedural texturing shader. Reducing the number of iterations for certain fragments improves the overall performance, since dynamic flow control is supported in Shader Model 3.0, which was used for this project. Fig. 6 shows terrain rendered with the fully integrated shader. The pixel shader itself can be found in the Appendix.

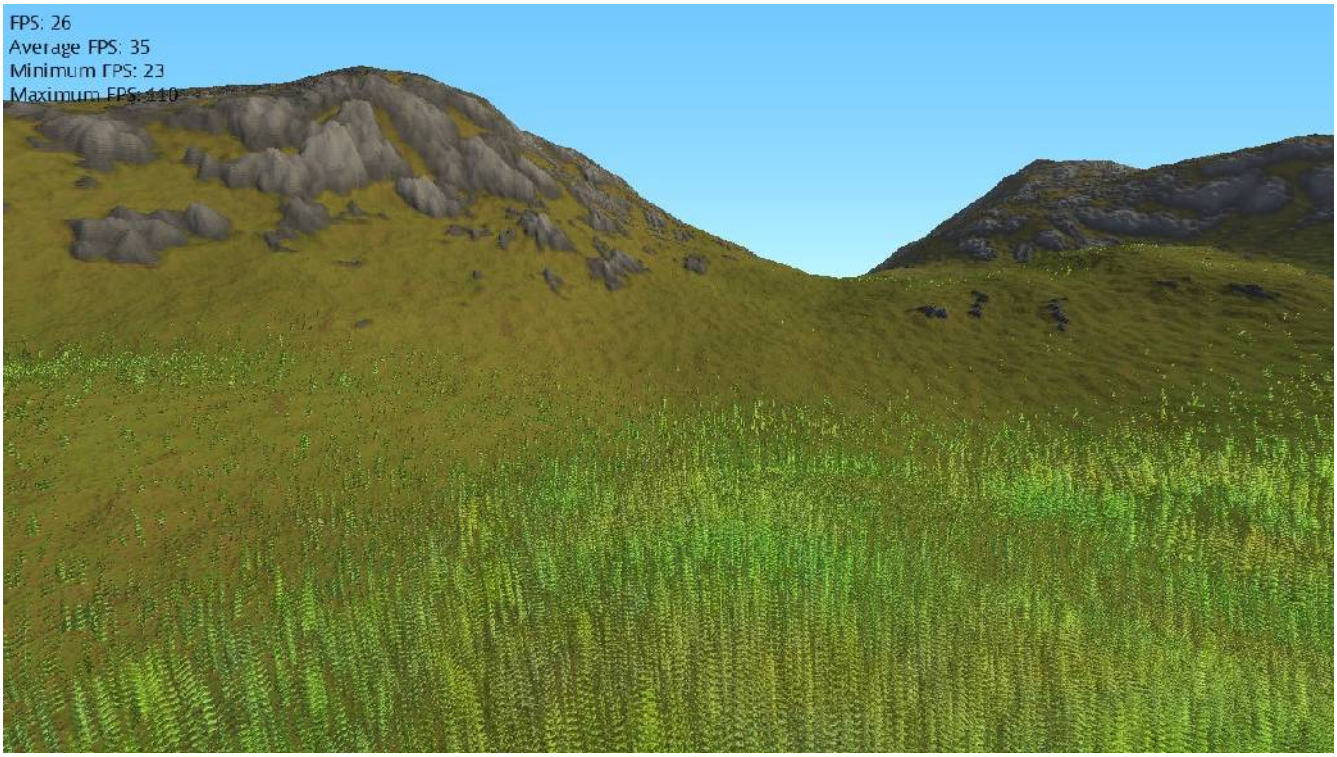

FIGURE 6: A screenshot of the project showing terrain rendered with the combined Multi-Material Procedural Texturing and Steep Parallax Mapping shader.

3. Results

3.1. Appearance

The completed procedural texturing system succeeds in meeting the appearance-related goals of the project. The terrain is aesthetically pleasing and looks quite detailed and realistic. Four different materials can be rendered using the same shader, giving the terrain a varied look. The transitions between them are smooth when necessary, such as between dirt and sand, and hard-edged for materials like rock. There is no obvious texture tiling present, due to the procedural texturing. The shader works for terrain that is close, as well as terrain that is far away, due to the mip-mapping on the uniform noise texture. Steep Parallax Mapping introduces parallax and self-occlusion, causing details such as grass and rocks to appear three-dimensional.

However, there are some problems. The level-of-detail system used is extremely simplistic. Because of this, some noticeable “popping” occurs as the level of detail changes. Because the procedural texturing is tied to the geometry of the terrain (since it is affect by altitude and incline), the level-of-detail changes in the terrain also cause the textures to change. If more time had been invested into this system, better results could have been achieved, but this was not the main focus of the project.

Another problem is related to aliasing, and is the most obvious when looking at the grass. Since a given pixel can correspond only to a single material, this leads to obvious aliasing along the edges of materials. Additionally, the steep parallax mapping implemented also either intersects a point or does not; this also creates aliasing problems. However, at a high resolution, this is less noticeable.

The most obvious visual artifacts are related to Steep Parallax Mapping. First, even though details such as grass appear to be three-dimensional, they do not stick out from the terrain's silhouette. Also, there is a great deal of “texture swimming”, or distortion, which happens when viewing the terrain at steep angles. As the camera moves, the textures appear to swim around, as if the geometry

were changing. These artifacts are shown in Fig, 7. Both the swimming and silhouetting problems are caused by the fact that Steep Parallax Mapping is not accurate for curved surfaces. In the shader, all calculations are done in tangent-space, and the ray marches through the height-field in a straight line. In other words, at each pixel, the surface is assumed to be planar, even though it is curved. This causes the shader to find an intersection point which is not correct. A possible solution to these problems would be, instead of using Parallax Mapping, to use a sort of displacement mapping post processing effect, where single pixels are projected as lines to the point where they should extend to. This could be very difficult to implement, and possibly impractical on current hardware.

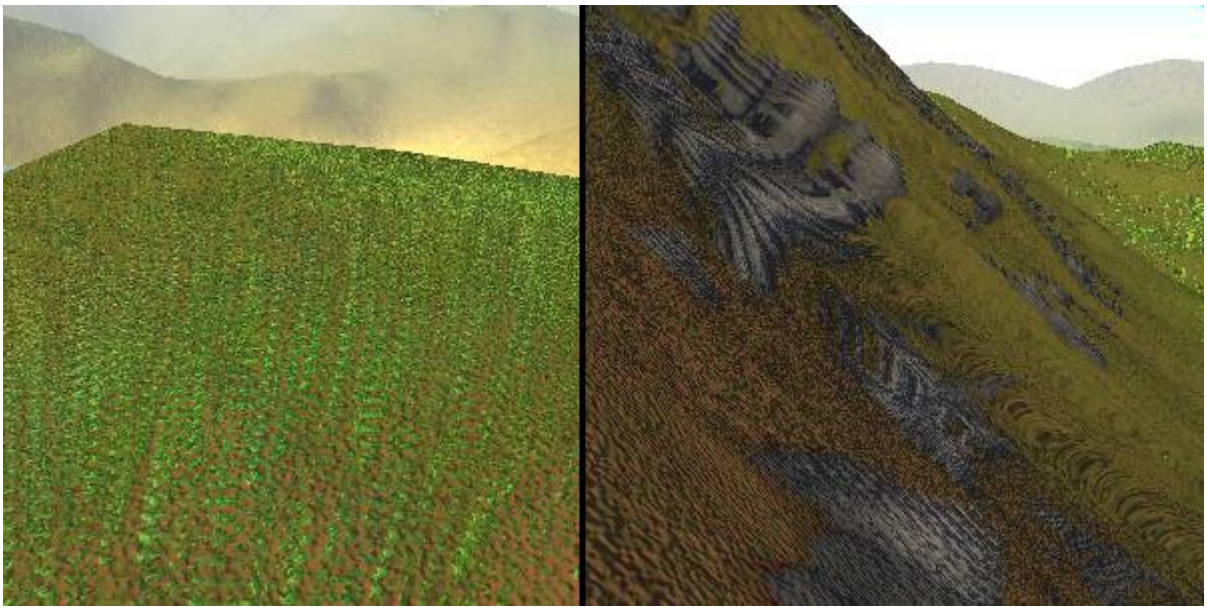

FIGURE 7: Visual artifacts related to Steep Parallax Mapping. On the left is the silhouetting problem, and on the right is the texture swimming problem.

3.2. Performance

To test the performance of the different shaders created during this project, a standard test was needed, so that they could be easily compared. To do this, a trajectory was chosen which is close to the ground, looks straight ahead, passes near every terrain type, and contains at least 50% terrain on the screen. This was chosen as it tests the kind of things a player would be looking at most of the time while playing a game in an outdoor environment, although it should be noted that the frame-rate would be lower if the player were looking straight down. The frame-rate was recorded over a 10 second as the

camera traversed the trajectory, and the average, minimum, and maximum frames per second were recorded. This was done for a very simple pixel shader, for the procedurally textured pixel shader, and for the procedurally textured pixel shader with Steep Parallax Mapping. The results are shown in Table 3.

	Average FPS	Maximum FPS	Minimum FPS
Per-Vertex Coloured Terrain	178	216	114
Procedurally Textured Terrain	102	112	77
Full Terrain Shader	37	44	31

TABLE 3: Results from the performance test for each shader.

The average frame-rate is well within the project goal of 30 frames per second, so the project has met the goal. Even the minimum frame-rate is within the limit, although if the camera were pointed down it would not be. The average frame-rate would likely be well above 30 frames per second for most playing sessions, if a player were to walk around in the world.

However, although the finished project has met the goal, there is an extremely large performance gap between the procedural texturing shaders with and without Steep Parallax Mapping. The reason for this is that for each sample taken along the ray in Steep Parallax Mapping, a large part of the procedural texturing shader needs to be iterated; each iteration includes a matrix multiplication and three texture lookups, so it is expensive. This section of the shader is iterated between 10 and 30 times, depending on the viewing angle. Moving this work into a post processing effect, could actually increase performance because of this; the procedural texturing calculations would only need to be performed once if the displacement was done as a post-processing effect, although, as stated in the previous chapter, this may not be easy or practical.

The frame-rate could likely be raised noticeably if the vertex shader and level-of-detail system were changed. Currently, significantly complicated calculations are being performed on the vertex shader to determine terrain height, tangents and binormals If the terrain heights, tangents and binormals

were calculated ahead of time and stored in textures, this would reduce the work that would need to be done at each frame. Alternatively, the level-of-detail system could be optimized. It currently does not perform any view-frustum clipping, so every vertex is processed each frame, even if it is off-screen. Adding view-frustum clipping could also greatly reduce the work the vertex shader needs to do, or alternatively, the geometric detail of the terrain could be increased while retaining the same frame-rate.

4. Conclusion

A procedural texturing system was built which combines multiple procedural materials in a novel way. A single shader was made, based on Perlin noise and uniform noise textures, which produces realistic and varied terrain. Whether the terrain is close or far from the viewer, it is aesthetically pleasing and there is no apparent tiling of textures. Grass, rock, dirt and sand can all be rendered with the single shader. Transitions between materials, which have hard edges when necessary, give a more detailed look than simple texture interpolation. In addition to the procedural texturing, parallax and self-occlusion were added in the form of Steep Parallax Mapping, giving features a more three-dimensional appearance. The performance goals of the project were also met. From the balanced test that was run, the average frame-rate with the full terrain texturing shader was thirty-seven frames per second, well above the thirty frames-per-second goal.

However, there are some down-sides to the approach taken. Because of the simplistic level-of-detail system that was implemented, performance is diminished and there is some visible texture popping. Also, due to the fact that Steep Parallax Mapping is not entirely accurate when used on curved surfaces, silhouettes are rendered incorrectly, and textures distort and “swim” at steep angles from the eye vector. In addition to this, Steep Parallax Mapping, combined with the procedural texturing shader, increases greatly in complexity, nearly lowering the frame-rate by two thirds.

Despite the problems, the project was able to meet all of its original goals, and so it was a success. With the current frame-rate, it could very well be integrated into a real-time application and run smoothly on today's hardware, providing an alternative to existing approaches to terrain texturing. Further work could be done on the level-of-detail system, or investigating an alternative to Steep Parallax Mapping.

References

Brebion, F., (2005) Infinity's planetary engine, http://www.gamedev.net/community/forums/topic.asp?topic_id=365210&whichpage=1&#2404834.

Green, S., (2005) Implementing Improved Perlin Noise, GPU Gems 2, Chapter 26.

Jenks, T., (2005) Terrain texture blending on a programmable GPU, http://www.jenkz.org/articles/terraintexture.htm.

Lengyel, J., Praun, E., Finkelstein, A., Hoppe, H., (2001) Real-time fur over arbitrary surfaces, Proceedings of the 2001 symposium on Interactive 3D graphics, pp. 227 – 232.

McGuire, M., McGuire, M., (2005) Steep Parallax Mapping, I3D 2005 Poster.

Musgrave, F.K., (1993) Methods for realistic landscape imaging, Yale University.

Oliveira, M., Bishop, G., McAllister, D., (2000) Relief Texture Mapping, Proceedings of SIGGRAPH 2000, pp.359-368.

Oliveira, M., Policarpo, F., (2005) An effective representation for surface details, Technical Report RP-351, Federal University of Rio Grande do Sul – UFRGS.

Perlin, K., (2002) Improving Noise, Proceedings of the 29th annual conference on Computer graphics and interactive techniques, pp. 681 – 682.

Tatarchuck, N., (2005) Practical dynamic parallax occlusion mapping, International Conference on Computer Graphics and Interactive Techniques, Article No. 106.

Waveren, J.M.P. van, (2009) id Tech 5 Challenges: From Texture Virtualization to Massive Parallelization, http://s09.idav.ucdavis.edu/talks/05-JP_id_Tech_5_Challenges.pdf.

Worley, S., (1996) A cellular texture basis function, Proceedings of the 23rd annual conference on Computer graphics and interactive techniques, pp. 291 – 294.

Appendix: HLSL Shader Code for Full Pixel Shader

```
//This is the High Level Shading Language pixel shader code for the full Multi-Material Procedural
Texturing with Steep Parallax Mapping.

struct VertexShaderOutput
{
    float4 Position : POSITION0;
    float4 Color : COLOR0;
    float4 Info : TEXCOORD0; //stores info for texture generation
    float3 TanPos : TEXCOORD1;
    float3 Normal : TEXCOORD2;
    float3 TanLightDir : TEXCOORD3;
    float3 TanCamPos : TEXCOORD4;
    float2 TexCoord : TEXCOORD5;
};

float4 FullTerrainPS(VertexShaderOutput input) : COLOR0
{
    float4 color;

    //these are the uv multipliers for the three texture samples
    const float bigMultiplier = 0.001;
    const float mediumMultiplier = 0.004;
    const float smallMultiplier = 0.032;

    //get the material mapping based on the information passed from the vertex shader. (Info.xy
represents the x and z of the normal and Info.z is the terrain height)
    float4 materialMapping = tex2D(materialMappingSampler, (float2(length(input.Info.xy),1-
(input.Info.z-WaterLevel)*0.005)));
    materialMapping = float4(0, //dirt
                             materialMapping.r,   //sand
                             materialMapping.g,   //grass
                             materialMapping.b);  //rock

    //make rocks smoother as they get rockier
    MaterialFunctions[3][1] *= 1-abs(materialMapping[3]*2-1);

    //make dirt bumpier on inclines
    MaterialFunctions[0].xyz *= length(input.Info.xy)+0.5;

    // We are at height bumpScale.  March forward until we hit a hair or the
    // base surface.

    float testHeight = 1.0;

    // Number of height divisions

    float numSteps;

    /** Texture coordinate marched forward to intersection point */
    float2 offsetCoord = input.TexCoord.xy*smallMultiplier;
    float4 NB;

    //get camera position relative to current position (eye vector is this but normalized)
    input.TanCamPos -= input.TanPos;
    float3 tsE = normalize(input.TanCamPos);

    //get the mip-map level based only on the distance from the camera
```

```
    float mipLevel = log(length(input.TanCamPos)*0.15);

    // Increase steps at oblique angles
    // Note: tsE.z = N dot V
    numSteps = lerp(30, 10, min(1,max(0,tsE.z)));

    if (mipLevel > 4)
    {
       //if we're far enough away, don't do any ray marching
       bumpScale = 0;
        numSteps = 1;
    }
    else if (mipLevel > 3)
    {
        //if we're not quite as far, fade out the steep parallax mapping
        bumpScale *= max(0, 4-mipLevel);
        numSteps = max(1, numSteps * max(0, 4-mipLevel));
    }

    float step;
    float2 delta;
    float texBias = 0.5;

    // Constant in z
    step = 1.0 / numSteps;
    delta = (float2(-tsE.y, -tsE.x) * bumpScale / (tsE.z * numSteps));
    delta.y /= 1-(input.Info.x*input.Info.x);
    delta.x /= 1-(input.Info.y*input.Info.y);

    //variables needed within the ray-marching loop
    float4 bigValue;
    float4 values;
    float4 rawHeights;
    float currentMaterial;
    float currentHeight = 0;

    //this is the main loop where ray marching is performed
    while (currentHeight-1 < testHeight + step && testHeight + step > 0) {

        //sample the three textures
        bigValue = tex2Dlod(perlinSampler, float4(offsetCoord*bigMultiplier/smallMultiplier,0,0));
        bigValue.r -= 0.5;
        values = float4(bigValue.a,
                            tex2Dlod(perlinSampler,
float4(offsetCoord*mediumMultiplier/smallMultiplier,0,0)).a,
                            tex2Dlod(uniformSampler, float4(offsetCoord,0,mipLevel)).a,
                            1);

        //get the heights for each material (without taking the material mapping into account)
        rawHeights = mul( MaterialFunctions, values);

        //adjust rock and sand to add ridges
        rawHeights[1]*= 1-abs(bigValue.r*0.75);
        rawHeights[3]*= 1-abs(bigValue.r);

        currentMaterial = 0;
        currentHeight = -10;

        //loop through each material
        for(int i =0; i < 4; i++)
        {
```

```
            //add an offset to the material based on the material mapping
            float height = rawHeights[i]+materialMapping[i]*Scaling[i];
            if (height > currentHeight)
            {
                //keep track of the highest material
                currentMaterial = i;
                currentHeight = height;
            }
        }

        testHeight -= step;
        offsetCoord += delta;
    }
    offsetCoord -= delta;

    //get neighbouring heights to calculate the normal
    float height2 = calculateHeightForMaterialLOD(offsetCoord/smallMultiplier+float2(0.04, 0),
materialMapping, currentMaterial, mipLevel);
    float height3 = calculateHeightForMaterialLOD(offsetCoord/smallMultiplier+float2(0, 0.04),
materialMapping, currentMaterial, mipLevel);
    float3 normal = normalize(-cross(float3(0, 0.02, height2-currentHeight), float3(0.02, 0,
height3-currentHeight)));

    if(currentMaterial >1)
    {
        //use the material color
        color = MaterialColors[currentMaterial];
        if (currentMaterial == 2)
        {
            //for grass, add some variance to the color based on the perlin noise textures
            color *= float4(-bigValue.a+2, bigValue.a*0.25+0.75, 1+values[1]*3,0);
        }
    }
    else
    {
        //if we have dirt or sand, blend the colours between them based on the sand's material
mapping
        color = lerp(MaterialColors[0], MaterialColors[1], materialMapping[1]);

        //darken the ground near rocks
        color *= 0.75+ 2 * min(0.125,currentHeight-(rawHeights[3]+materialMapping[3]*Scaling[3]));

        //make flat parts grass-coloured when in a grassy area
        color = lerp(color,MaterialColors[2],materialMapping[2]*(1-length(normal.xy)*5)*0.5);
    }
    if (currentMaterial == 3)
    {
        //make the edges of rocks dark
        color *= clamp((currentHeight-0.6)*0.9,0.75,1);

        //make the specular intensity vary for rocks (based on the fine noise)
        SpecularIntensity *= values[2]*2;
    }

    //add noise to the colour and make higher parts lighter
    color *= 0.5 + currentHeight*0.5*(values[2]*0.2+0.9);

    if(input.Info.z <= WaterLevel+1.2-currentHeight)
    {
        //make underwater terrain tinted blue
        color = float4(0.8,0.8,0.4,0);
```

```
    }

    //do lighting calculations
    float3 directionToCamera = normalize(input.TanCamPos - input.TanPos);
    float3 reflectionVector =  normalize(reflect(-input.TanLightDir, normal));
    float specular = pow(saturate(dot(-reflectionVector, directionToCamera)),
SpecularPower[currentMaterial]);
    float diffuse = saturate(dot(-input.TanLightDir, normal));

    if(currentMaterial == 2)
    {
        //for grass, make high parts unaffected by diffuse lighting
        diffuse = lerp( diffuse, 1, saturate(currentHeight-
rawHeights[0]+materialMapping[0]*Scaling[0]));
    }
    else
    {
        //make the lighting harder
        diffuse = pow(diffuse, 0.75);
    }

    //apply ambient, diffuse, and specular lighting
    color *= float4(0.4,0.45,0.6,1) + diffuse*float4(1,0.9,0.75,0) +
specular*SpecularIntensity[currentMaterial];

    if(input.Info.z <= WaterLevel)
    {
        //apply water fog
        color = lerp(color, float4(0.3,0.5,0.7,1), input.Color.r);
    }

    //apply distance fog
    color = lerp(color, float4(1,1.03,1.06,1), input.Color.g);
    color.a = 1;
    return color;
}
```