GPU Tessellation: We Still Have a LOD of Terrain to Cover

10

António Ramires Fernandes and Bruno Oliveira

10.1 Introduction

Terrain rendering has come a long way from the days where all data fit in the graphics memory to algorithms that deal with massive amounts of information that do not fit in the system's RAM. Nowadays, a full-blown terrain engine has to deal with out-of-core issues: a first step of the level of detail (LOD) might happen in the CPU to determine which data goes to the GPU, and a second step of LOD may be required to draw all those triangles at interactive rates.

In this chapter we are going to explore how OpenGL 4.x can boost performance in this last step. The algorithms presented will use GPU tessellation for shader-based LOD and view-frustum culling.

Although LOD can substantially reduce the amount of geometry rendered, it may also cripple the fidelity of the representation. An approach will be introduced to render heightmap-based terrains, which can be included in most of the available terrain-rendering engines, that captures in a simple process the irregularities of a terrain, maintaining a very high level of visual fidelity to the original data.

Previous knowledge on the subject of GPU tessellation is assumed. See Chapter 6 or "Programming for Real-Time Tessellation on GPU" [Tatarchuk et al. 09], for an introduction to the subject.

10.2 Rendering Terrains with OpenGL GPU Tessellation

The goal of this section is to present a heightmap-based, fully tessellated terrain rendering implementation, upon which the LOD solutions will grow (see Figure 10.1).

We assume that the heightmap is a regular grid, represented by a greyscale image loaded as a texture. However, the terrain size is not limited by the texture's size, as height values between *texels* can be sampled. The GPU has dedicated hardware for sampling, such as GLSL `texture*` functions, according to the sampler or texture sampler state. The terrain size, in terms of grid points, is, therefore, theoretically unlimited. To avoid the almost flatness of the regions represented by the sampled points, noise-based approaches can be used to provide high-frequency detail.

The terrain size, in terms of physical units, can be further parameterized by defining a grid spacing; in other words, the number of units between two consecutive points in the final grid.

To render the terrain, we use the new primitive introduced with tessellation, the `patch`. A patch can cover as many grid points as the maximum tessellation levels permitted by hardware, a square grid of 64 quads in the current OpenGL 4.0 hardware. This defines a patch as 65×65 vertices, as the edges of patches are shared between adjacent patches. To render a terrain, we define a grid of such patches. As an example, to render a terrain of $8K \times 8K$ points, a grid of 128×128 patches is required. Other patch sizes are possible, but the reported tests (shown later in Figure 10.9), show a performance penalty when using smaller patches.

Since the terrain grid is a highly regular structure, only one vertex to define a patch is needed (e.g., the lower left corner). The regular nature of the terrain's grid allows the developer to compute all other patch elements based solely on this vertex. The final vertex positions, texture coordinates, and normals will be computed in the shaders.

The patch positions are defined as if the terrain were to be drawn in a normalized square, ranging from 0 to 1. A translation and scale operation will be applied in the tessellation evaluation shader to place the terrain where needed.

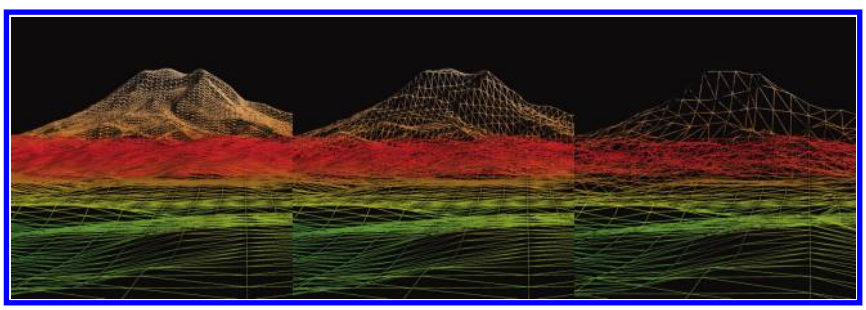

Figure 10.1. Full tessellation (left); high LOD (middle); low LOD (right).

10.2.1 GPU Tessellation Shaders

The vertex shader is a simple pass-through, as vertex transformations will be performed later in the pipeline. It receives the vertex *xz* position, and outputs it to a

```
// one vertex per patch
layout(vertices = 1) out;
// XZ position from the vertex shader
in vec2 posV[];
// XZ position for the tessellation evaluator shader
out vec2 posTC[];

void main()
{
  // Pass through the position
  posTC[gl_InvocationID] = posV[gl_InvocationID];
  // Define tessellation levels
  gl_TessLevelOuter = ivec4(64);
  gl_TessLevelInner = ivec2(64);
}
```

Listing 10.1. Tessellation control shader (full tessellation).

```
layout(quads, fractional_even_spacing, cw) in;

// The heightmap texture sampler
uniform sampler2D heightMap;
// Scale factor for heights
uniform float heightStep;
// Units between two consecutive grid points
uniform float gridSpacing;
// Number of height samples between two consecutive texture texels
uniform int scaleFactor;
// The Projection * View * Model matrix
uniform mat4 pvm;

// Vertices XZ position from the tessellation control shader
in vec2 posTC[];
// Output texture coordinates for the fragment shader
out vec2 uvTE;

void main()
{
  ivec2 tSize = textureSize(heightMap, 0) * scaleFactor;
  vec2 div = tSize * 1.0/64.0;
  // Compute texture coordinates
  uvTE = posTC[0].xy + gl_TessCoord.st/div;
  // Compute pos (scale x and z) [0..1] -> [0..tSize * gridSpacing]
  vec4 res;
  res.xz = uvTE.st * tSize * gridSpacing;
  // Get height for the Y coordinate
  res.y = texture(heightMap, uvTE).r * heightStep;
  res.w = 1.0;
  // Transform the vertices as usual
  gl_Position = pvm * res;
}
```

Listing 10.2. Tessellation evaluator shader.

```
// the normal matrix
uniform mat3 normalMatrix;
// texUnit is the color texture sampler
uniform sampler2D texUnit, heightMap;
uniform float heightStep, gridSpacing, scaleFactor;

// Texture coordinates from the tessellation evaluator shader
in vec2 uvTE;
// Color output
out vec4 outputF;

// Function to retrieve heights
float height(float u, float v)
{
  return (texture(heightMap, vec2(u, v)).r * heightStep);
}

void main()
{
  // compute the normal for the fragment
  float delta =  1.0 / (textureSize(heightMap, 0).x * scaleFactor);
  vec3 deltaX = vec3(
    2.0 * gridSpacing,
    height(uvTE.s + delta, uvTE.t) - height(uvTE.s - delta, uvTE.t),
    0.0);

  vec3 deltaZ = vec3(
    0.0,
    height(uvTE.s, uvTE.t + delta) - height(uvTE.s, uvTE.t - delta),
    2.0 * gridSpacing);

  normalF = normalize(normalMatrix * cross(deltaZ, deltaX));
  // The light direction is hardcoded. Replace with a uniform
  float intensity = max(dot(vec3(0.577, 0.577, 0.577), normalF), 0.0);
  // Diffuse and ambient intensities - replace by uniforms
  vec4 color = texture2D(texUnit, uvTE) * vec4(0.8, 0.8, 0.8, 1.0);

  outputF =  color * intensitiy + color * vec4(0.2, 0.2, 0.2, 1.0);
}
```

Listing 10.3. Fragment shader.

`vec2` named `posV`.[1] The heights, or *y* coordinates, will be sampled in the tessellation evaluator shader.

The tessellation control shader for a fully tessellated terrain (Listing 10.1) sets all tessellation levels to maximum, as defined by the patch size, configuring the next step, the nonprogrammable tessellation primitive generator. The position from the vertex shader is passed through to the tessellation evaluator shader.

After the execution of this shader, the tessellation primitive generator has all the data it needs, in other words, the tessellation control levels. The output will be a grid of *uv* coordinates, 65×65, which will be the input of the next programmable stage, the tessellation evaluator (Listing 10.2).

[1] As a rule of thumb, in the code displayed in this chapter, all `out` variables are defined with a suffix, which represents the shader that outputs the variable, so `pos` is the input of the vertex shader, and `posV` is the output. In the tessellation control shader, `posV` will be the input, and `posTC` the output and so on.

The tessellation evaluator is responsible for the transformation of the vertex positions and computation of the texture coordinates (`uvTE`). Although normals could also be computed here, when using LOD it is advisable to compute them in the fragment shader (see Section 10.5.2).

The fragment shader (Listing 10.3) has as input the vertex position through `gl_Position`, and the texture coordinates through `uvTE`. The shader is pretty standard apart from the normal computation, which is based on the approach suggested by [Shandkel 02].

And...that's it! A full tessellated terrain can be obtained with these four simple shaders.

10.3 A Simple Approach to Dynamic Level of Detail

When using GPU tessellation, LOD naturally becomes a synonym for tessellation levels. Hence, a simple approach for LOD can be implemented by computing a tessellation level for each patch edge, the outer tessellation levels, while the inner tessellation levels can be computed as the maximum of the respective outer tessellation levels.

A criteria that has been commonly used in previous CPU LOD implementations is the projected screen size of an object's bounding box. Using this approach for the tessellation outer levels, the tessellation level of an edge becomes a function of its projected size. Therefore, adjoining patches will share the same tessellation level for the common edge, thus ensuring a crack-free geometry.

Dynamic LOD requires smooth geometry transitions as the LOD varies. OpenGL offers a tessellation approach that resembles geomorphing, using either `fractional_even_spacing` or `fractional_odd_spacing` as the output layout qualifier in the tessellation evaluator shader.

Tessellation levels are defined in the tessellation control shader, so this is where the changes happen. All other shaders remain the same. As in the previous section, the only data available to this shader is a corner of the patch.

Picking the two points that constitute an edge of the patch, one can compute its projected length [Boesch 10]. The projected length is then used to define a level of tessellation based on a single parameter: pixels per edge. For instance, if a segment has a projected size of 32 pixels and we want 4 pixels per triangle edge, then we should compute 32/4, or 8, as the respective outer tessellation level.

The main issue with this approach is that patch edges that are almost collinear with the view direction tend to have very low tessellation levels, as the projected size will be very small; however, this can be fixed with some extra parameterization or over-tessellation. Another solution, provided by [Cantlay 11], is to consider the

projected size of a sphere with diameter equal to the length of the edge of the patch. This solution deals effectively with the collinearity issue.

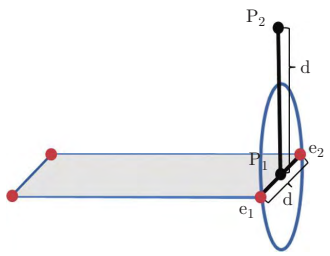

Figure 10.2. Diagram to compute the projected sphere screen size.

To compute the projected size of the sphere (Figure 10.2) we pick two corners sharing the same edge, e_1 and e_2, and compute the edge's length in world space, d. Then, we compute the midpoint of the edge, P_1, and a new point above the centre, P_2, displaced by the edge's length, d. Points P_1 and P_2 are then transformed to screen space. The distance between the transformed points provides the screen-space diameter of the enclosing sphere.

Function `screenSphereSize` (Listing 10.4) performs these computations and determines the tessellation level for the edge, based on the computed diameter divided by the parameter `pixelsPerEdge`, clamped to ensure a valid tessellation level.

Patches outside the view frustum should be discarded by setting their edge tessellation levels to zero. In order to test if a patch is inside the view frustum, we must consider the available information, the four corners of the patch, and, hence, its edges. The heights for points inside the patch are unknown at this stage, so we cannot afford to perform culling based on the y-axis information. However, it is safe

```
// Viewport dimension in pixels
uniform ivec2 viewportDim;
// LOD parameter
uniform int pixelsPerEdge;

// Sphere screen size based on segment e1-e2
float screenSphereSize(vec4 e1, vec4 e2)
{
  vec4 p1 = (e1 + e2) * 0.5;
  vec4 p2 = viewCenter;
  p2.y += distance(e1, e2);
  p1 = p1 / p1.w;
  p2 = p2 / p2.w;
  float l = length((p1.xy - p2.xy) * viewportDim * 0.5);
  return(clamp(l / pixelsPerEdge, 1.0, 64.0));
}

// determining if an edge of a patch is inside the XZ frustum
bool edgeInFrustum(vec4 p, vec4 q)
{
  return !((p.x < -p.w && q.x < -q.w) || (p.x > p.w && q.x > q.w) ||
           (p.z < -p.w && q.z < -q.w) || (p.z > p.w && q.z > q.w));
}
```

Listing 10.4. Auxiliary functions for the tessellation evaluator shader.

to perform conservative culling in clip space based on the *xz* coordinates of the transformed corners of the patch. The function `edgeInFrustum` (Listing 10.4) performs this computation. See [Ramires 07] for more details on how to perform view frustum culling in clip space.

Listing 10.5 shows the tessellation control shader. Functions in Listing 10.4 are also part of the shader's code. Initially, the remaining three corners of the patch are computed. All four corners are then transformed into clip space. Then, for each edge, we check whether it is at least partially inside the view frustum using function

```
layout(vertices = 1) out;
// ...
void main() {
  vec4 posTransV[4];
  vec2 pAux, posAux[4];

  vec2 tSize = textureSize(heightMap, 0) * scaleFactor;
  float div = 64.0 / tSize.x;
  posTC[ID] = posV[ID];
  // Compute the fours corners of the patch
  posAux[0] = posV[0];
  posAux[1] = posV[0] + vec2(0.0, div);
  posAux[2] = posV[0] + vec2(div, 0.0);
  posAux[3] = posV[0] + vec2(div, div);
  // Transform the four corners of the patch
  for (int i = 0; i < 4; ++i )
  {
    pAux = posAux[i] * tSize * gridSpacing;
    posTransV[i] = pvm * vec4(pAux[0], height(posAux[i].x,posAux[i].y), pAux[1], ↩
        1.0);
  }
  // check if a patch is inside the view frustum
  if (edgeInFrustum(posTransV[ID], posTransV[ID + 1]) ||
      edgeInFrustum(posTransV[ID], posTransV[ID + 2]) ||
      edgeInFrustum(posTransV[ID + 2], posTransV[ID + 3]) ||
      edgeInFrustum(posTransV[ID + 3], posTransV[ID + 1]))
  {
    // Compute the tess levels as function of the patch's edges
    gl_TessLevelOuter = vec4(
      screenSphereSize(posTransV[ID], posTransV[ID + 1]),
      screenSphereSize(posTransV[ID], posTransV[ID + 2]),
      screenSphereSize(posTransV[ID + 2], posTransV[ID + 3]),
      screenSphereSize(posTransV[ID + 3], posTransV[ID + 1]));
    gl_TessLevelInner = vec2(
      max(gl_TessLevelOuter[1], gl_TessLevelOuter[3]),
      max(gl_TessLevelOuter[0], gl_TessLevelOuter[2]));
  }
  else
  {
    // Discard patches by setting tessellation levels to zero
    gl_TessLevelOuter = vec4(0);
    gl_TessLevelInner = vec2(0);
  }
}
```

Listing 10.5. Tessellation control shader for simple LOD.

`edgeInFrustum`. If all edges are outside the view frustum, the tessellation levels are set to zero, and the patch is culled. Otherwise, the outer tessellation level is computed for each edge using the function `screenSphereSize`. The inner levels are set to the maximum of the respective outers to ensure a sound subdivision.

10.4 Roughness: When Detail Matters

The LOD solution presented in the previous section does a great job in reducing the number of triangles, and creating a terrain that can be rendered with a high frame rate. However, there is an implicit assumption that all patches are born equal. A terrain may be considered homogeneous, when the roughness of its surface is similar at every point, or heterogeneous, where both very smooth and very rough areas can be found. An approach based on the projected size of the patch edges does not take into account the variation of the heights inside a patch or its roughness, and it will either over-tessellate flat distant patches or under-tessellate rougher patches. Hence, the previous method is more suitable for homogeneous terrains.

The goal of this section is to present an LOD solution to the aforementioned problem, considering heterogeneous terrains, by taking into account the roughness of a patch to compute its tessellation levels. To achieve this, a roughness factor for each patch is calculated to be used as a scale factor. These factors are precomputed on the CPU and submitted to the GPU along with the patch's coordinates. This information can be precalculated outside the rendering application, speeding up bootstrapping.

Due to the preprocessing stage, this approach is unsuitable for terrains with dynamic geometry. However, if only a small part of the terrain is affected, this approach can still be used for the static part of the terrain, and over-tessellation can be used for the dynamic areas.

10.4.1 Adding the Roughness Factor

To compute the roughness factor, the average patch normal is determined considering the four corners of the patch. The maximum difference between the normals of each individual vertex of the fully tessellated patch and the average normal is the value stored as the patch's roughness.

For each outer tessellation level, the roughness applied will be the maximum between the two patches that share the edge, hence ensuring crack-free geometry.

Once again, the only shader affected is the tessellation control shader. Listing 10.6 shows the changes to this shader. The function `getRoughness` fetches the roughness value for a patch, stretching it to create a wider range of values. This stresses irregularities in the terrain, which may otherwise be ignored. The function's constants are experimental, and it can be an interesting exercise to find the best scaling factor for roughness.

```
uniform sampler2D roughFactor;

float getRoughness(vec2 disp)
{
  return (pow((1.8 - texture(roughFactor, posV[0] + disp / textureSize(roughFactor↩
      ,0)).x ),  4));
}

// Place this code in the main function presented before
// replacing the outer tessellation level computation
//(...)
  vec4 rough;
  float roughForCentralP = getRoughness(vec2(0.5));
  rough[0] = max(roughForCentralP, getRoughness(vec2(-0.5, 0.5)));
  rough[1] = max(roughForCentralP, getRoughness(vec2(0.5, -0.5)));
  rough[2] = max(roughForCentralP, getRoughness(vec2(1.5, 0.5)));
  rough[3] = max(roughForCentralP, getRoughness(vec2(0.5, 1.5)));
  gl_TessLevelOuter = vec4(
    screenSphereSize(posTransV[ID], posTransV[ID + 1]) * rough[0],
    screenSphereSize(posTransV[ID + 0], posTransV[ID + 2]) * roughs[1],
    screenSphereSize(posTransV[ID + 2], posTransV[ID + 3]) * rough[2],
    screenSphereSize(posTransV[ID + 3], posTransV[ID + 1]) * roughn[3]);
//(...)
```

Listing 10.6. Snippet of the tessellation control shader for LOD with a roughness factor.

10.5 Crunching Numbers, or Is This All That Matters?

Now that the different techniques for rendering terrains with tessellation and LOD have been introduced, it is time to look at the numbers. These tests are not, however, entirely related to how many triangles, or frames-per-second, the application can score. The visual quality of what is being rendered is also important, and so image comparison tests were also conducted.

10.5.1 Test Setup

The data used for the tests is a 16-bit heightmap and its corresponding color texture (Figure 10.3), are used, for instance, in [Lindstrom and Pascucci 01]. These files report geographical data from the Puget Sound area in the USA.

In all tests, the color texture was 2K $\times$ 2K, whereas the resolution of the terrain grids tested ranged from 1K $\times$ 1K to 64K $\times$ 64K. The height data are based on a heightmap of up to 8K $\times$ 8K , with greater resolutions resorting to in-shader height sampling.

This terrain was chosen particularly for this LOD study since it is highly heterogeneous. It contains areas that are almost flat, green and blue, and areas that are very irregular, most of the red and white.

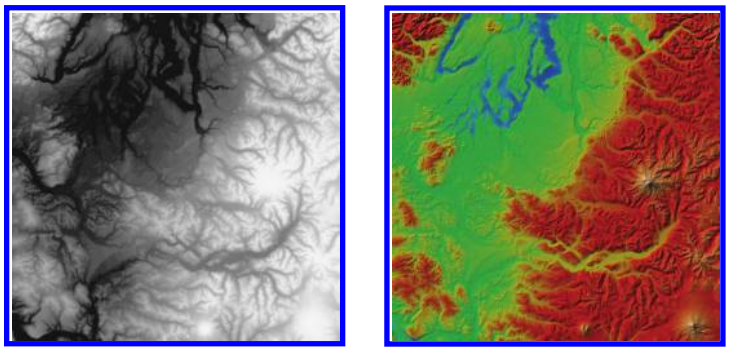

Figure 10.3. Terrain height and color maps.

10.5.2 Evaluating the Quality of the LOD Solutions

To evaluate an LOD solution, we must take several factors into account, as LOD is not only about performance. Using LOD causes the geometry to change as the

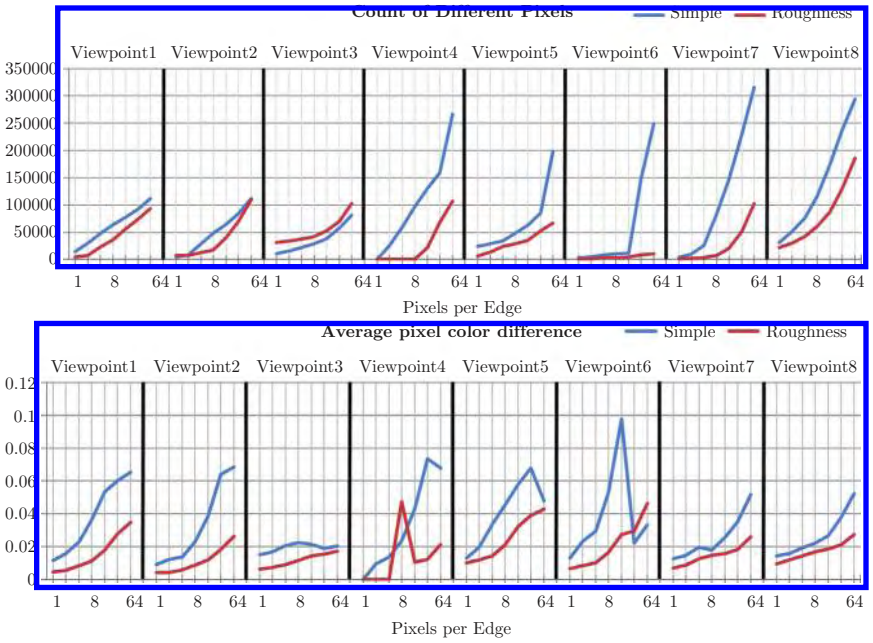

Figure 10.4. Differences between the two LOD approaches and the full tessellated solution computed from eight viewpoints. The total number of different pixels (top); the average color difference per pixel (bottom).

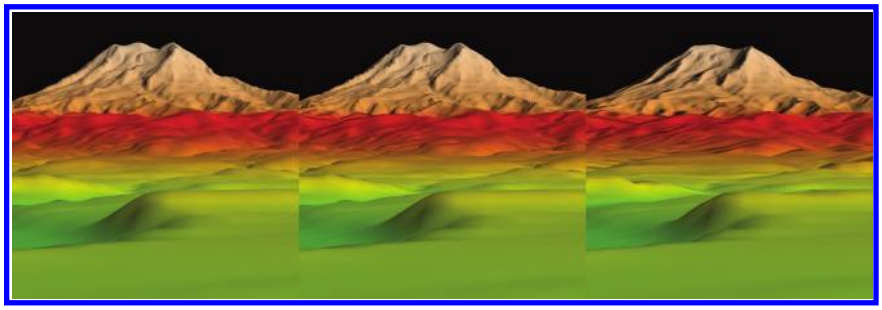

Figure 10.5. Close-ups for full tessellation (left), roughness approach (center), and simple method (right), with 16 pixels per edge.

camera moves in the scene, triggering variations in tessellation, and this may lead to visual artifacts. Another issue is related to the similarity to the original model. An LOD solution can be high performing without significant visual artifacts; still there may exist meaningful differences when compared to the original model.

The first test relates to the visual quality of the LOD solutions. The method used was to take the framebuffer from a fully tessellated terrain and compare it to the results of both LOD solutions, as presented in Sections 10.3 and 10.4. In both cases, there is only one parameter to control the LOD, the number of pixels per edge. For each comparison, two differences were computed: the number of different pixels and a pixelwise color difference. This test was performed from eight different viewpoints. The chart in Figure 10.4 provides the results.

Regarding the LOD parameter, pixels per edge ($\{1, 2, 4, 8, 16, 32, 64\}$), the methods behave as expected. As the number of pixels per edge increases, so does the amount of different pixels. In general, both the count of different pixels and the average color difference are lower for the roughness approach.

A clearer perspective can be obtained by looking at the actual images. Figure 10.5 shows close-ups of snapshots taken from viewpoint 1 for both the LOD methods and the fully tessellated geometry. The figure shows that the simple method tends to oversimplify patches that are further away, and it does alter the shape of the distant mountains. The LOD with a roughness factor, on the other hand, provides a nearly perfect contour.

Figure 10.6 was also built looking from the first viewpoint. The top row shows that the simple method clearly is more prone to misrepresenting the contour of the distant irregular geometry. The bottom row shows that, although the number of pixel differences is relatively the same for both methods, these correspond to very small color differences when using the roughness factor. For instance, considering the roughness factor with 16 pixels per edge, the differences are barely noticeable.

Concluding this test, one can state that, even when considering higher values for the parameter pixels per edge, the results are perceptually better when using the

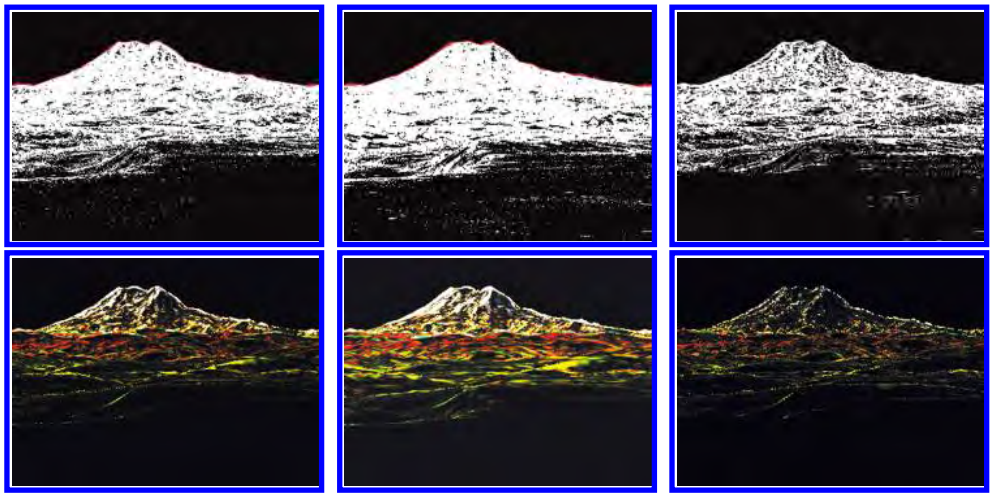

Figure 10.6. Differences between both LOD approaches and the full tessellation result. Top row: pixel difference; bottom row: five times enhanced color difference. Left column: simple method, 8 pixels per edge. Middle column: simple method, 16 pixels per edge. Right column: roughness approach, 16 pixels per edge.

roughness factor, as the shape of distant geometry seems to be significantly affected if patch roughness is not considered.

The second test relates to the expected differences when the camera moves from viewpoint $P(d)$ to viewpoint $P(d + \text{step})$, where step is the distance traveled in two consecutive frames. These differences occur during the course of camera navigation. At point $P(d)$, a feature of the terrain being observed may look differently when the camera moves a single step, due to dynamic tessellation.

Using LOD, it is to be expected that the tessellation levels vary when the camera moves, and this can cause visual artifacts that reveal the dynamic nature of the algorithm. To have zero visual artifacts, the result using tessellation computed at point $P(d)$ should be indistinguishable from the result using tessellation computed at point $P(d + \text{step})$, both being observed at point $P(d + \text{step})$.

This test calculates the differences between images generated at point $P(d+\text{step})$, using tessellation levels computed for point $P(d)$ and point $P(d + \text{step})$, for values of step in the set $\{1, 2, 4, 8, 16, 32, 64\}$ units. The chart in Figure 10.7 presents the average number of different pixels considering eight test viewpoints and using values of pixels per edge from 1 to 64. As expected, the errors reported for both LOD methods increase as the step increases. The error also grows as pixels per edge gets bigger.

The chart in Figure 10.7 clearly shows that, for the same value of pixels per edge, the differences that can be expected using the simplest method are always significantly higher than using a roughness factor.

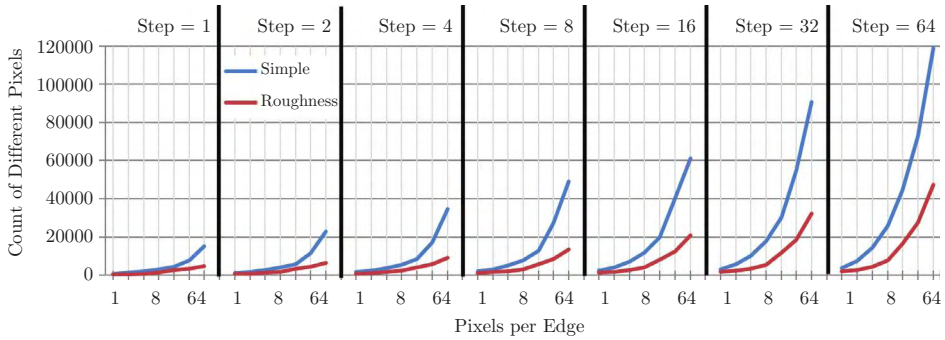

Figure 10.7. Differences in the step forward test, using both LOD methods.

10.5.3 Performance

Now it is time to see if the suggested LOD implementation using hardware tessellation pays off. It has already been shown that LOD introduces errors. These may be controlled by the number of pixels per edge, or some other more sophisticated approach, but there will be errors. So, the performance reports must be conclusive; otherwise, what is the point?

The hardware used for testing was a desktop system with a GeForce 460 GTX with 1GB of RAM and a laptop system with a Radeon 6990M with 2GB of RAM.

The results are presented for the terrain described previously, where the camera completed a full circle, advancing one degree per step, performing a total of 360 frames. The total time and the number of primitives generated, found using OpenGL queries, were recorded for each trial.

The test was performed with terrain grids ranging from 1K × 1K to 64K × 64K, varying the value of pixels per edge for both LOD methods. As a comparison, full tessellation, traditional submission of geometry (up to 4K × 4K) and instancing (up to 8K × 8K) are also presented.

The first chart, Figure 10.8, compares the performance obtained for full tessellation (with and without culling) with both full triangle grid submission and instancing with a patch of 64 × 64 vertices, which is replicated to cover the full terrain. The terrain grid size varies from 1K up to 8K.

Full tessellation is only beaten by the full triangle grid submission. The instancing approach does not match the performance of either in this case. Culling, as expected, boosts the tessellation approach, making it worthwhile to include the extra bit of code in the shaders.

Considering the largest terrains, and comparing to the full tessellation with culling approach, the frame rates of the other approaches are simply to low for any practical use. Notice that to show the data, the chart was created with a logarithmic FPS scale (base 2); otherwise, some of the data would not show up meaningfully.

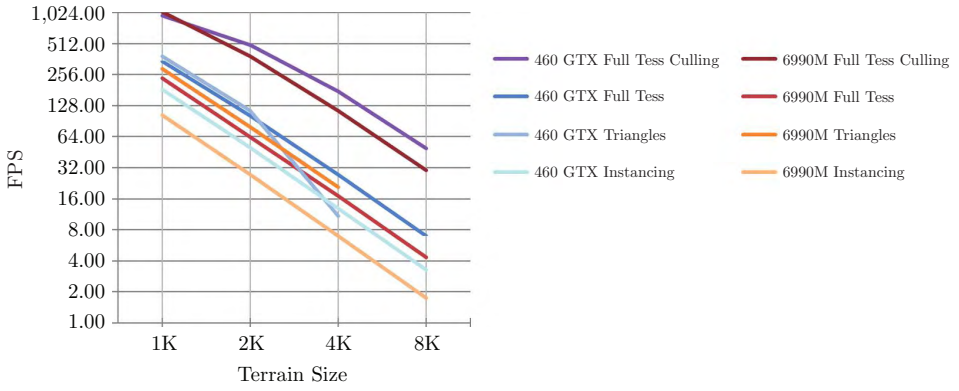

Figure 10.8. Frames-per-second performance on rendering a terrain without LOD.

The chart in Figure 10.9 reports on the performance of the roughness approach and full tessellation (with and without culling), considering three possible patch sizes on a Radeon 6990M.

In all cases, LOD introduces a very significant performance boost. In fact, the boost is so significant that, again, a logarithmic scale was used to show visible curves for every method. The patch size does influence performance, with larger sizes performing better overall; in particular, culling with larger patches is clearly more efficient. The other feature that is highlighted from the chart is the relevance of the LOD factor, pixels per edge, performancewise. The parameter works as expected, with performance increasing with the number of pixels per edge.

Now that the benefits of culling have been shown and the patch size impact on performance has been observed, all the remaining tests make use of culling and a

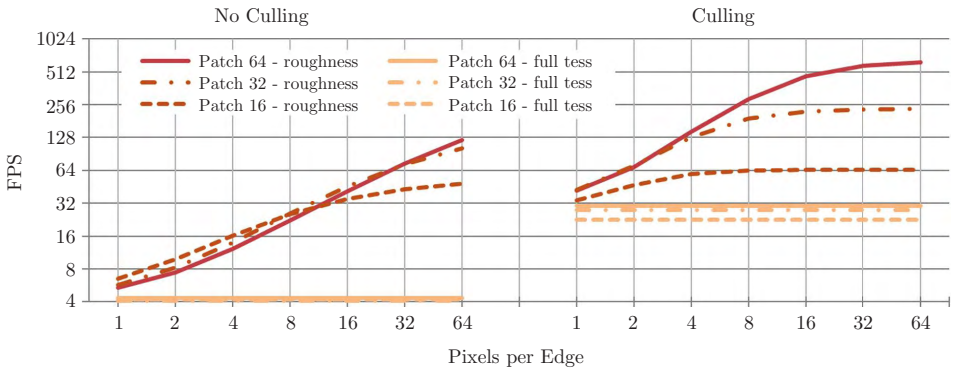

Figure 10.9. Comparison on the effect of culling in full tessellation and roughness methods.

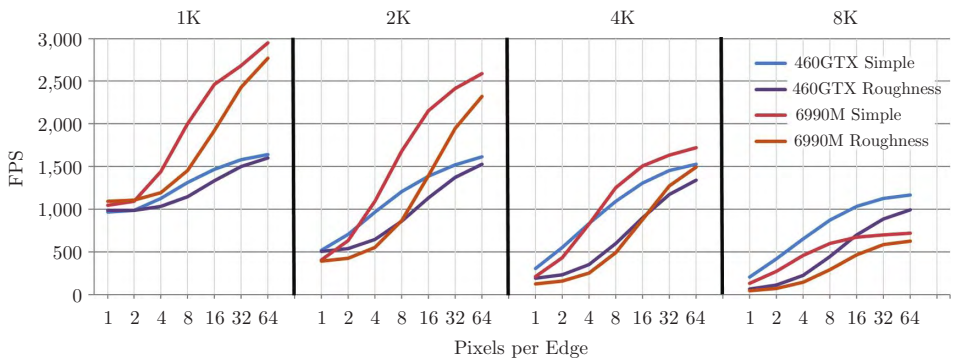

Figure 10.10. Frames per second for both LOD methods, with terrain sizes up to 8K.

patch size of 65 × 65. The goal is to test how the variation of the terrain's grid size affects performance and also to see how far one can push GPU LOD based on tessellation with OpenGL.

The chart in Figure 10.10 reports the tests in terrains up to 8K × 8K. Performancewise, both LOD methods achieve very high frame rates. As expected, the simple method performs better than the roughness approach. However, as seen before, the errors obtained with the latter approach are lower than the errors obtained with the former method, and a global comparison should take this into consideration.

Pushing further, we have tested both LOD methods using terrains up to 64K × 64K (Figure 10.11). Such a grid has over four billion vertices, so its a huge challenge for any technique. Since, presently, there is no hardware that is even remotely capable of handling this massive requirement, optimization techniques such as culling and LOD are mandatory.

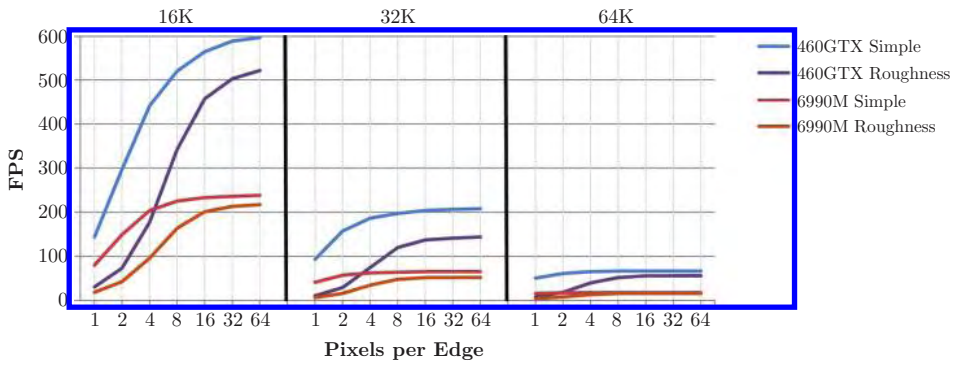

Figure 10.11. Frames per second for both LOD methods, with terrain sizes up to 64K.

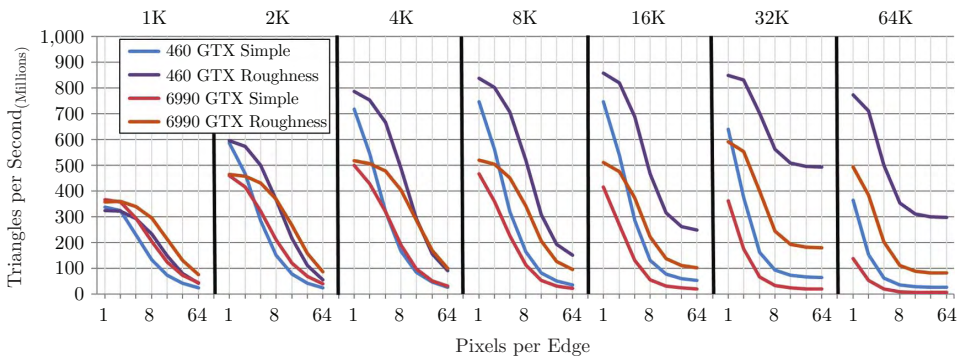

Figure 10.12. Triangles per second on both LOD methods, with terrain sizes up to 64K.

The throughput of primitives created by the tessellation primitive generator stage has much higher values when using the roughness approach, particularly as the terrain grows larger. This is the price to pay for having higher fidelity to the original model, but it also suggests that our implementation might be too conservative.

Regarding the number of triangles processed per second (Figure 10.12), the results show the GeForce 460 GTX achieving over 800 million, and the Radeon 6990M topping out at 500 million.

10.6 Conclusion

Taking advantage of the new tessellation engine and corresponding shaders, a GPU-based LOD algorithm was presented. The algorithm takes into account the roughness of the terrain to preserve a high level of fidelity to the original data, which is specially required in highly irregular, distant patches. It can be either used as a standalone method for terrains that fit in graphics memory, or, when considering larger terrains, as the final rendering stage of full-blown terrain-rendering engines, enhancing the fidelity of the rendered terrain while not compromising the frame rate.

Bibliography

[Boesch 10] Florian Boesch. "OpenGL 4 Tessellation." http://codeflow.org/entries/2010/nov/07/opengl-4-tessellation/, 2010.

[Cantlay 11] Iain Cantlay. "DirectX 11 Terrain Tessellation." Technical report, NVIDIA, 2011.

[Lindstrom and Pascucci 01] P. Lindstrom and V. Pascucci. "Visualization of Large Terrains Made Easy." In *Proceedings of the Conference on Visualization'01*, pp. 363–371. IEEE Computer Society, 2001.

[Ramires 07] António Ramires. "Clip Space Approach: Extracting the Planes." http://www.lighthouse3d.com/tutorials/view-frustum-culling/clip-space-approach-extracting-the-planes, 2007.

[Shandkel 02] Jason Shandkel. "Fast Heightfield Normal Calculation." In *Game Programming Gems 3*. Hingham, MA: Charles River Media, 2002.

[Tatarchuk et al. 09] Natalya Tatarchuk, Joshua Barczak, and Bill Bilodeau. "Programming for Real-Time Tessellation on GPU." Technical report, AMD, Inc., 2009.