DirectX 11 Terrain Tessellation

Iain Cantlay
icantlay@nvidia.com

January 2011

Document Change History

Version	Date	Responsible	Reason for Change
1.0	21/01/11	IAC	Initial release

Contents

Figures

NVIDIA Corporation
2701 San Tomas Expressway
Santa Clara, CA 95050
www.nvidia.com

Abstract

This sample implements a technique for rendering a terrain surface using DirectX 11 tessellation. We show how to implement a crack-free surface with LOD that is adaptive in screen-space. Tessellating the geometry on the GPU is highly efficient and rendering 1.5 million polygons at more than 100 FPS is easily possible on a GeForce GTX460.

The technique is also easy to integrate with an existing terrain rendering engine. We show how art assets from an existing engine can be reused and augmented with additional detail.

This paper assumes a general familiarity with DirectX 11 tessellation. For an introduction to tessellation see the recording of this GTC 2010 presentation (Cebenoyan, 2010) which also includes a discussion of this terrain technique.

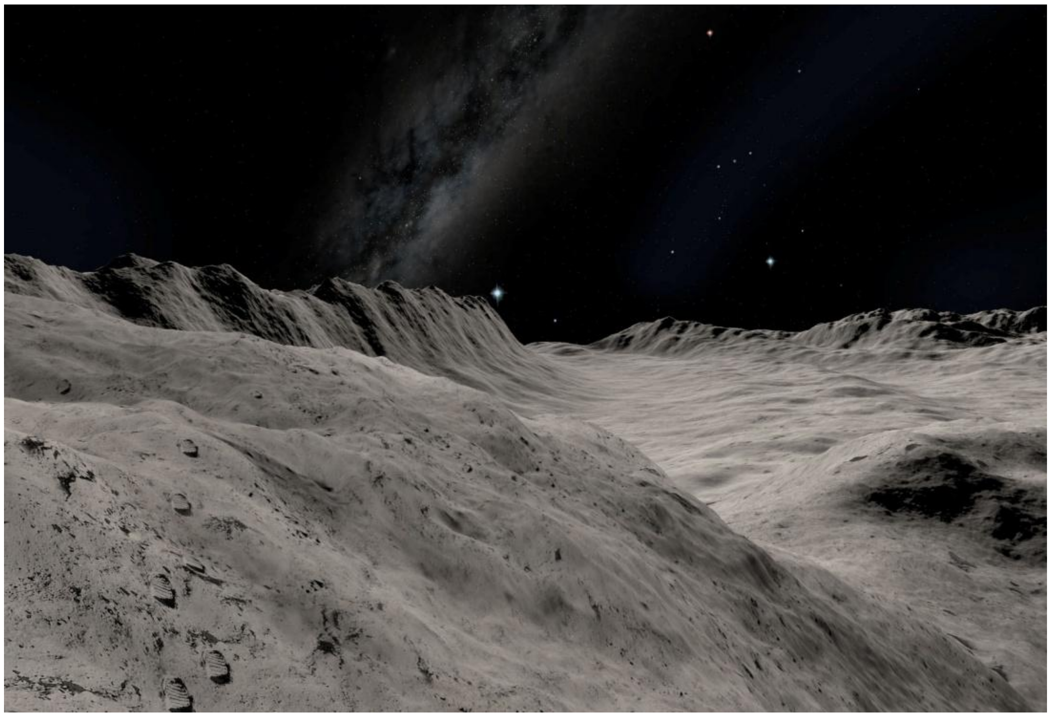


Figure 1: A typical view of our DirectX 11 sample.

Figure 2: A similar terrain technique as used in *Tom Clancy's H.A.W.X. 2*, courtesy of Ubisoft.

How Does It Work?

Mapping to the Tessellation Pipeline

Figure 3 below shows how the technique maps onto the stages of the shader pipeline. On the left are the various shader stages; on the right is an approximate sketch of the status of one patch at the corresponding pipeline stage.

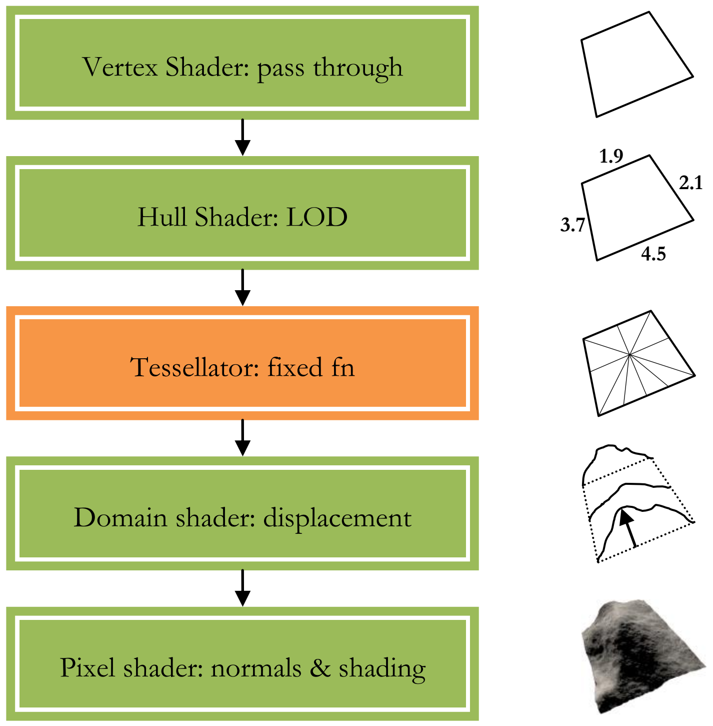

Vertex Shader: pass through
Hull Shader: LOD
Tessellator: fixed fn
Domain shader: displacement
Pixel shader: normals & shading
1.9
2.1
3.7
4.5

Figure 3: the shader pipeline stages, their tasks and a sketch of one patch, passing through the pipeline.

The primitives are axis-aligned quad patches. They are entirely flat: all the heights are zero. The vertex data defines only the 2D extents of a patch. Figure 4 shows the arrangement of patches. They are grouped into tiles for top-level culling and rendering, with 8x8 patches per tile. Patches are shown as a checkerboard pattern; whereas tiles are separated by thick green/yellow gaps. (The tessellation factors are artificially low in these screenshots for illustration purposes.)

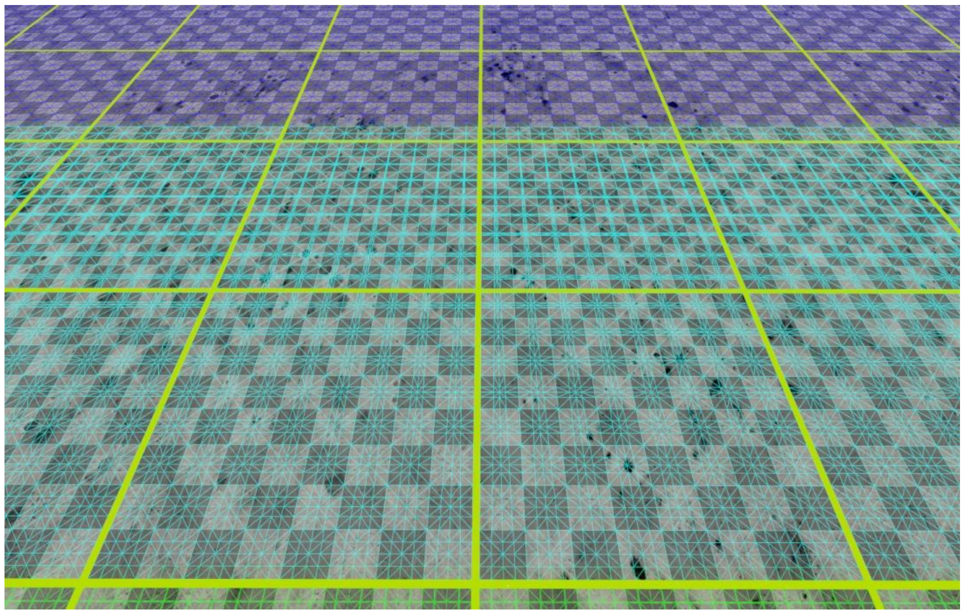


Figure 4: A basic grid of uniform patches. Terrain tiles are separated by the thick green/yellow lines; patches are shown by the checkerboard pattern (8x8 per tile).

The vertex shader is almost entirely pass-through. Its only task is to displace the quad patch corners. This correctly places the patch vertices and edges in model space, ready for subsequent LOD calculations in the next stage.

The primary task of the hull shader is to compute LOD and assign tessellation factors to the patch edges and centre. Correct choice of tessellation factors is essential for generating a crack-free surface and this task dominates the hull shader, making it the most complex part of the system.

Next, the fixed-function tessellator subdivides the patch into triangles, using the tessellation factors output by the hull shader.

The domain shader samples the displacement map(s) and offsets each vertex vertically. It then applies the world-view-projection matrix to transform the vertex into clip space. The result of a simple displacement is shown in Figure 5.

Finally, the pixel shader shades the surface in the usual manner.

You may notice that the sample also includes a Geometry Shader. This is only used for rendering the debug wireframe and is not a necessary part of the terrain rendering. The "solid wireframe" technique is described in (Samuel Gateau, 2007) and (Andreas Bærentzen, 2006).

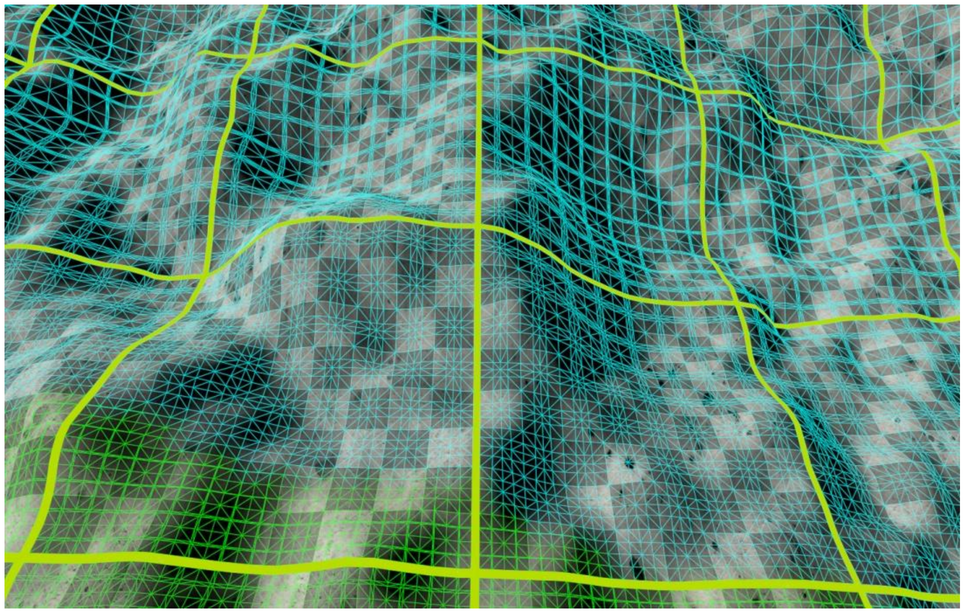


Figure 5: Patches displaced by seven octaves of fBm noise. These are the same patches as shown in Figure 4.

Hull Shader: Tessellation LOD

We will first describe a simpler version of the hull shader (HS). A more complex version that accounts for varying patch sizes will be discussed later. From now on, when we refer to the hull shader, we mean the patch constant hull shader function. The per-vertex hull shader is a trivial pass through.

Any partitioning scheme works for this simple version of the hull shader: integer, fractional, etc.

The primary task of the basic hull shader is to compute LOD or tessellation factors. The inputs from the vertex shader are four patch control points. These are also the corner vertices of the patch. Recall that they have already been displaced vertically by the vertex shader. This is important where displacements are high because the vertical offset can significantly affect the distance to the eye point. For an example, compare the tessellation factors in figures Figure 4 and Figure 5. The patches and the relative location of the view point are identical in both screenshots. However, the tessellation factors increase slightly in Figure 5 because of the displacement upwards towards the eye.

For each patch edge, the shader computes the edge length and then conceptually fits a sphere around it. The sphere is projected into screen-space and its screen-space diameter is used to compute the tessellation factor for the edge. See Figure 6 below. The algorithm targets a triangle width in pixels:

```
        tessellation factor = diameter / g_tessellatedTriSize;
```

This calculation results in a triangle size that is uniform in screen-space (boundary conditions notwithstanding). It is very scalable: it scales automatically with display resolution and the `g_tessellatedTriSize` variable can be easily be tuned to achieve the best performance/detail trade-off.

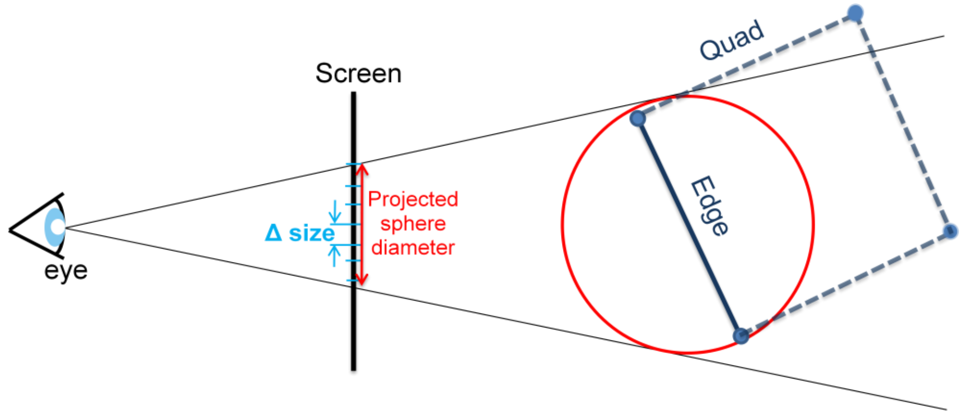


Figure 6: Screen-space-based computation of tessellation factors.

An initial version of the screen-space algorithm projected the quad edges themselves into screen-space and then applied a similar calculation. However, this fails when a quad with significant displacement is viewed edge-on, as shown in Figure 7. The displaced quad edge is almost parallel to the view vector and thus has almost zero length in screen-space. The result is a minimum tessellation factor and severe aliasing. Projected spheres work far better. The edge-based code has been left in the HS source and is commented-out; it could be enabled for illustration purposes.

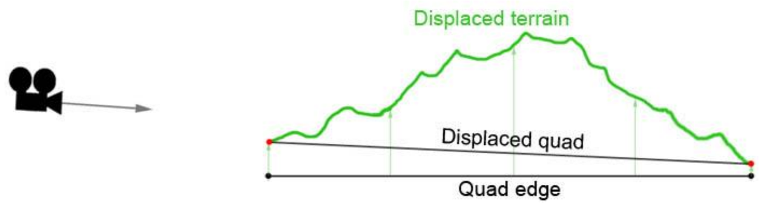


Figure 7: Screen-space-based tessellation factors for a quad seen edge-on.

The hull shader also performs view frustum culling on a per-patch basis. Patches are culled by setting their tessellation factors to -1.

Domain Shader: Displacement Mapping

The domain shader does straightforward UV parameter interpolation – it's a square quad patch. It sample two displacement maps at different scales. The detail displacement is scaled by the inverse of the coarse displacement, in the usual ridge noise manner after (F. Kenton Musgrave, 2002).

Pixel Shader & Normals

The pixel shader unremarkably samples colour textures and lights the scene. However, it is important to note that the pixel shader *alone* computes surface normals in our sample.

Normals can be computed in the domain shader and this might be more efficient than per-pixel normals. But we prefer fractional_even partitioning because it leads to smooth transitions between tessellation factors. Fractional portioning gives tessellated geometry that moves continuously. If geometric normals are computed from the tessellated polygons, the shading aliases significantly.

The tessellation can be designed such that the vertices and normals move less – see the geo-morphing ideas in (Cantlay, 2008). However, this would impose further constraints on an already-complex hull shader. We prefer to limit the hull and domain shaders to geometry and LOD and place the shading in the pixel shader. Decoupling the tasks simplifies the whole.

Crack-Free Tessellation

For patches of uniform size, a crack-free surface is achieved by computing tessellation factors purely as a function of quad patch edges. Since edges are shared, each patch arrives at a result that agrees with its neighbors' edges. This is implemented by the function `SphereToScreenSpaceTessellation`; note that it only takes the edge vertices and edge diameter as arguments.

It is quite easy to break the crack-free surface and create holes (especially the more complex version below). Careful testing is essential during development – see Debug Options below.

(It is not *strictly* true to say that edges are shared between adjacent patches. Patch vertices are constructed in the shader from a combination of the patch center and the system value `VertexId`. See the `ReconstructPosition` shader function. Theoretically, the values might differ. It would be easy to convert this to vertices that are truly shared by placing vertex positions in a vertex buffer and indexing them with an index buffer. However, it works in practice and far greater liberties are taken with adjacent patches when dealing with non-uniform patch sizes.)

Non-Uniform Patch Sizes

Motivation – Range of Scales

Terrain engines often support large world sizes (or level sizes in games). For example, *Tom Clancy's H.A.W.X. 2* uses levels that are an impressive 128x128km. To render a world of this size and maintain a roughly constant triangle size in screen-space requires a huge range of triangle sizes in world space. Triangles closest to the view-point may be only a few centimeters wide; the most distant triangles may be several kilometers across. Figure 8 below is typical – the foreground polygons are smaller than footprints, say 5cm, but the distant polygons are maybe hundreds of meters across. And the view distance in this image is fairly short.

DirectX 11 limits tessellation factors to the range 1 to 64. This is not nearly sufficient to represent the range of scales required for a large, detailed world. To continue the example above, the smallest polygons are 5cm. If they are implemented with the finest tessellation factor of 64, the largest polygon can only be 64 x 5 = 320cm. Whereas terrain may require a range of scales like 5cm to 1km or 20,000 to 1.

The solution is of course to employ patches of differing sizes. Terrain engines commonly implement LOD with tiles of differing sizes, for example (Ulrich, 2002). We apply this to our patch sizes, as shown in Figure 9. Our `TileRing` C++ class implements the varying size. (Ring is not an entirely accurate term – they are concentric squares.) Recall that a tile is 8x8 patches. Each successive `TileRing` increases the tile size by 2x; the 2x scaling considerably simplifies the crack-free algorithm below.

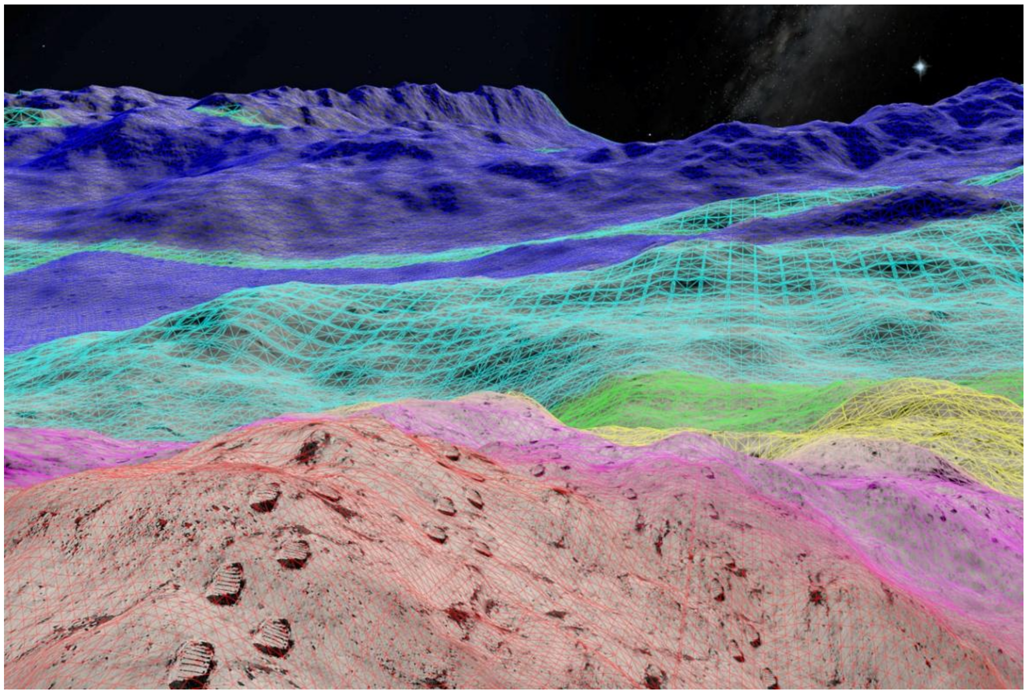


Figure 8: The wide range of polygon sizes in a typical engine. Note how the screen-space size is approximately constant.

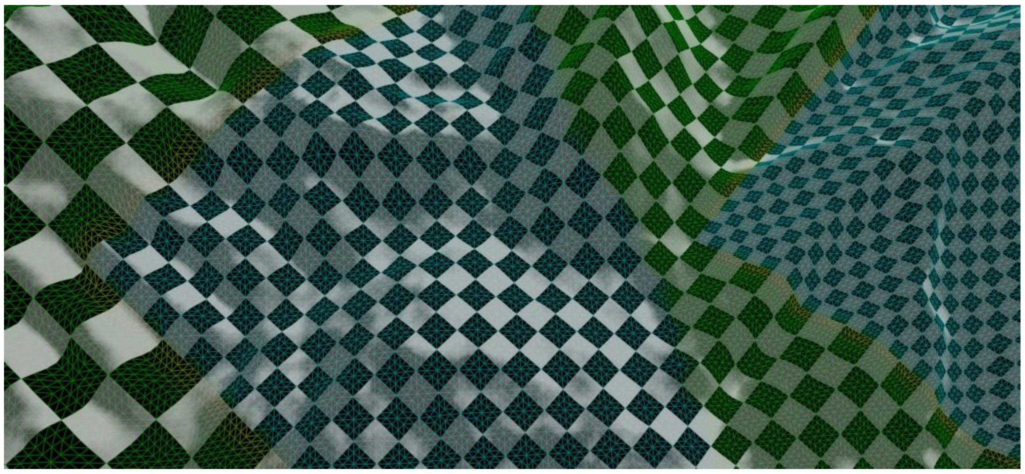


Figure 9: Non-uniform patch sizes.

Implementation – Adjacency of Different-Sized Patches

The crack-free technique described above does not work for patches of different sizes. The simpler version of the algorithm assumes that patches share edges. But this is clearly not the case in Figure 9.

It is necessary to add explicit information about a patch's neighbors. A relatively simple and concise description suffices: relative size. The relative size is one float scalar that indicates the neighbor's size, relative to ours. Figure 10 shows some example adjacency scales. In practice, each patch has four adjacency scalars – only a few are show for the sake of clarity. Setting the adjacency scalars per-tile is one of the primary jobs of the `TileRing` class. See the `AssignNeighbourSizes` function.

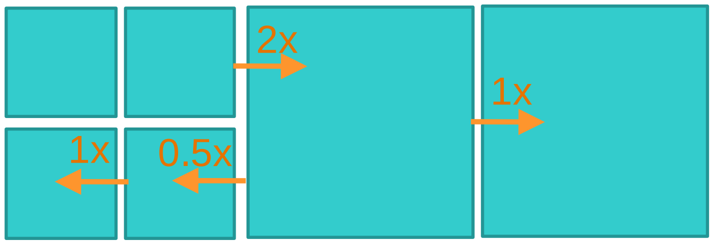


Figure 10: Adjacency size relationships between non-uniform patches.

Note that the adjacency information is per-instance vertex data; one instance is one tile. Thus all the patches and vertices within a tile share the same adjacency. Boundary patches and edges must be identified on a finer granularity. It is also necessary to know which patch is being processed within the tile/instance:

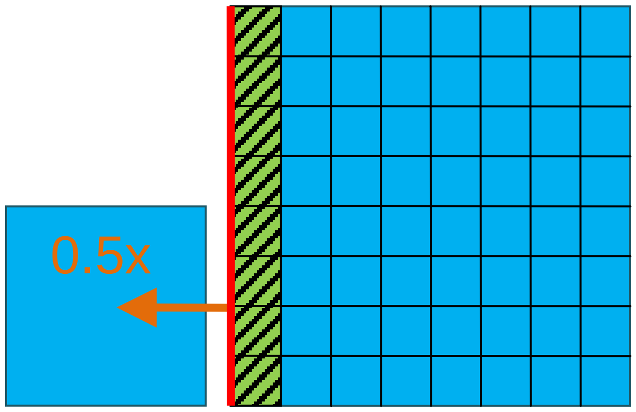


Figure 11: Identifying edges that require adjacency adjustment within a tile. The larger tile is shown divided into its 8x8 patches.

Figure 11 shows the central tile from Figure 10 in more detail. The larger tile has been subdivided into its constituent quad patches. The patches that are adjacent to smaller tiles are hatched green and, within the patches, the adjacent edges are red.

The code to detect these edges forms a large part of the hull shader – the eight if statements in `TerrainScreenspaceLODConstantHS`. The basic tessellation LOD is computed at the top of the hull shader, then the if statements detect edges that have different-sized neighbors. Note that one patch can trigger multiple if statements where it is on a tile corner.

Having detected an edge that is adjacent to a different-sized tile, it is then necessary to adjust its tessellation factor to match the neighbor. There are three main parts to this:

1. Duplicating the neighbor's LOD calculation.
2. Clamping to integer tessellations.
3. As a work-around, clamping to power-of-2 tessellations.

In more detail:

Firstly, one side of the unequal size difference must calculate its neighbor's LOD (they don't both need to adjust). We arbitrarily chose to alter the smaller patch's LOD when it has a larger neighbor; adjusting the larger patch would probably work equally well. Some simple vector math is used to reconstruct the larger neighbor's vertices:

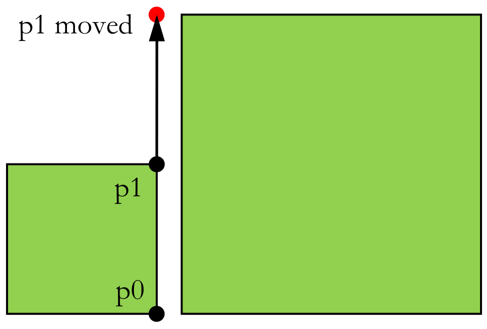


Figure 12: Reconstructing the edge of a patches larger neighbor.

The edge reconstruction of Figure 12 is implemented by the function `LargerNeighbourAdjacencyFix`. The patch corner vertices have also been displaced. So it is necessary to also fix the height of the new p1 in the function `MakeVertexHeightsAgree`. (See the source code for a discussion of a bug work-around.) The smaller tile's tessellation factor is then half of its larger neighbor's.

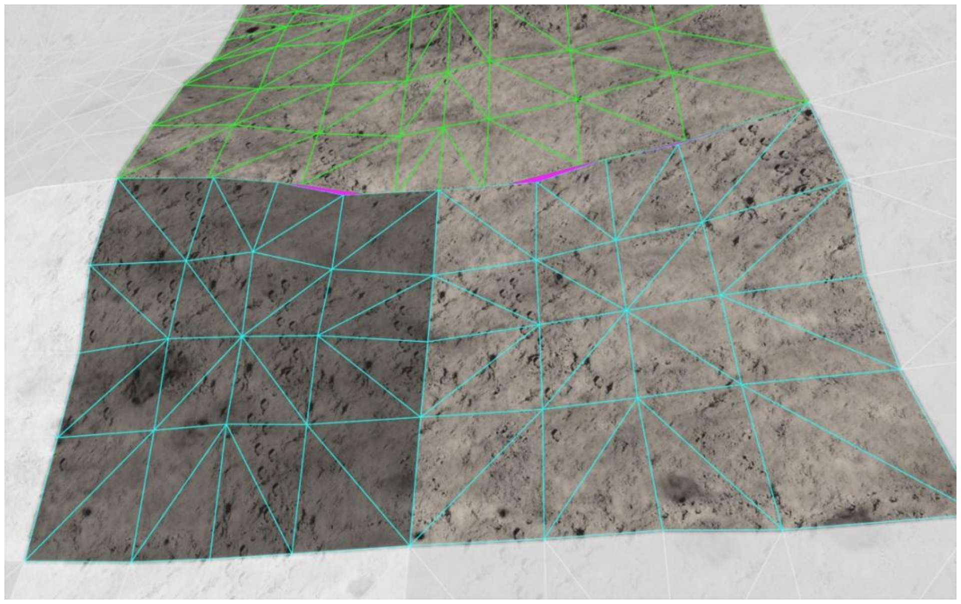


Figure 13: fractional_even partitioning across a boundary between different-sized patches.

The smaller patch's tessellation factor is now exactly one half of its larger neighbor's. However, the tessellation will not match due to our choice of fractional_even partitioning. This is best illustrated with an example showing the actual tessellation patterns - Figure 13. One large patch is shown with two smaller adjacent neighbors. The larger patch has a tessellation factor of about 6.8 and the small patches thus have 6.8/2 = 3.4. But the even partitioning divides each of these tessellation values into symmetric patterns. They do not match and purple cracks are visible.

The fix is to round each tessellation factor to an integer value and additionally ensure that the larger patch has an even value. To avoid losing detail, we round the values up. So to continue the example in Figure 13, the larger patch rounds up to the nearest even value of 8 and the small patches round up to 4. This is implemented in the functions `SmallerNeighbourAdjacencyClamp` and `LargerNeighbourAdjacencyClamp`.

These functions must also take into account that their neighbors will hit the hardware limits on tessellation factors, namely 2 and 64 (for even partitioning). So a small patch with a larger neighbor cannot exceed 64/2=32 and a larger patch cannot go below 2*2=4.

Finally, we implement a work-around. It appears that even if adjacent different-sized patches have the correct integer tessellations (say, 8 and 2x 4 to continue the above example), the tessellation patterns only seem to match in practice if the values are powers of two. We can't really explain this result, except to say that we should not strictly expect adjacent different-sized patches to work at all. The situation is analogous to T-junctions in triangle meshes.

Advisability of Adjacent, Different-sized Patches

Putting different-sized patches next to each other takes big liberties with the correctness of the crack-free result:

1. The vector math in Figure 12 is not guaranteed to produce a bit-wise accurate reconstruction of the neighbor's vertex.
2. Nothing in the specification guarantees that simply choosing the numerically correct tessellation levels will result in the exact same vertices on either side of the boundary.

The need for the power-of-2 workaround is a clear demonstration of the second point: on a purely numerical basis, one would expect and edge with 12 to match 2 edges of 6.

Our algorithm probably works only because the fixed-function tessellator stage is required to interpolate in 16-bit precision. But it does work on every implementation currently available, including the Direct3D 11 reference rasterizer.

If the algorithm breaks, say on a future Direct3D implementation, we suggest that it could be fixed by adding non-square patches at the boundary as in Figure 14.

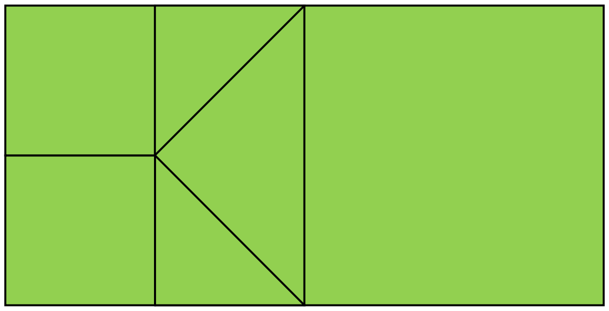
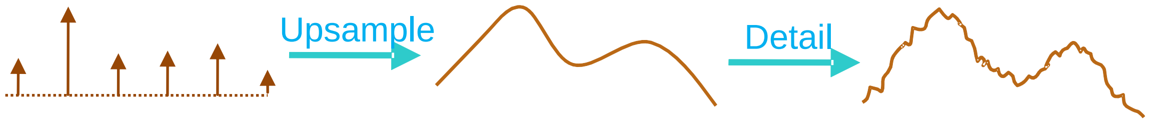


Figure 14: fixing boundaries with non-square patches.

Multiple Displacement Maps

Our technique uses two displacement maps, applied at different scales. The coarse displacement map is a 1024x1024 texture that covers the entire world. It defines the broad, overall shape of the landscape: the hills and valleys. In our sample, the coarse displacement map is generated procedurally from fractals. After (F. Kenton Musgrave, 2002), we create a "hybrid" terrain from a combination of ridge noise and fBm noise. This is rendered to a texture whenever the viewpoint moves.

The detailed displacement map is a repeating, generic map containing 5 octaves of fBm noise. The detail map was also created by a version of our sample in order to give noise characteristics that are similar to the coarse map. It was manually edited in Photoshop to make it tile. The detail map is also a 1024x1024 map in our sample and is shown in Figure 16.



The use of detail displacement was inspired by Figure 15 from (Fournier, 1982). (However, the Australia diagram shows stochastic subdivision, rather than displacement mapping.)

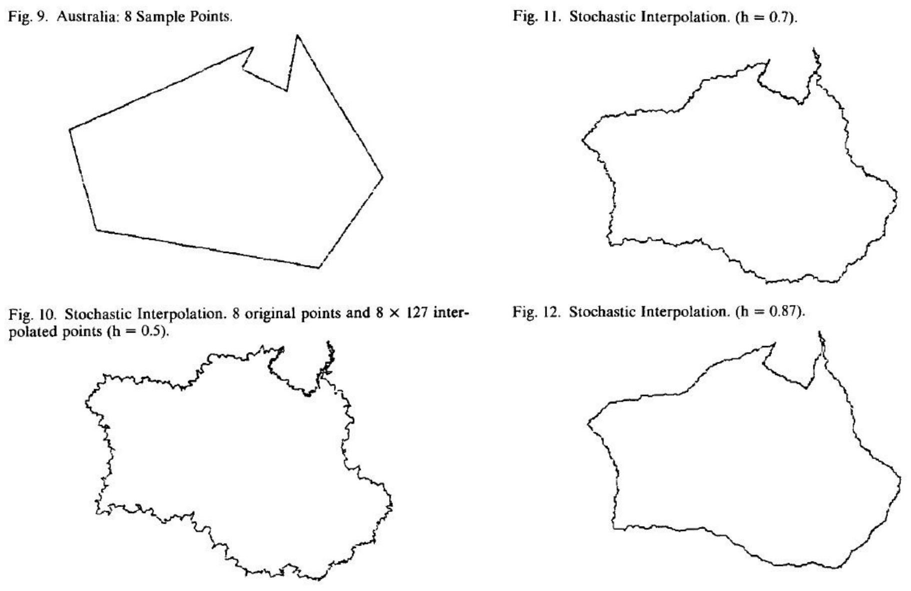


Fig. 9. Australia: 8 Sample Points.
Fig. 11. Stochastic Interpolation. (h = 0.7).
Fig. 10. Stochastic Interpolation. 8 original points and 8 × 127 interpolated points (h = 0.5).
Fig. 12. Stochastic Interpolation. (h = 0.87).

Figure 15: Adding detail to an Australia model from (Fournier, 1982).

Detail displacement mapping is very efficient. Storage requirements are minimal, but the perceived level of detail is very high. As Figure 17b shows, the most detailed displacements are only slightly larger than footprints. (The footprint is a simple 2D colour texture, not a displacement.) The shape of the distant mountains and valleys is defined by the coarse map and can be unique at each location. The limited resolution of the coarse map is shown in Figure 17a.

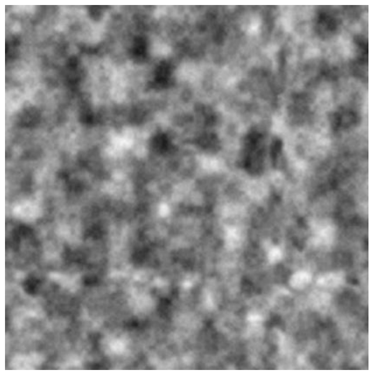
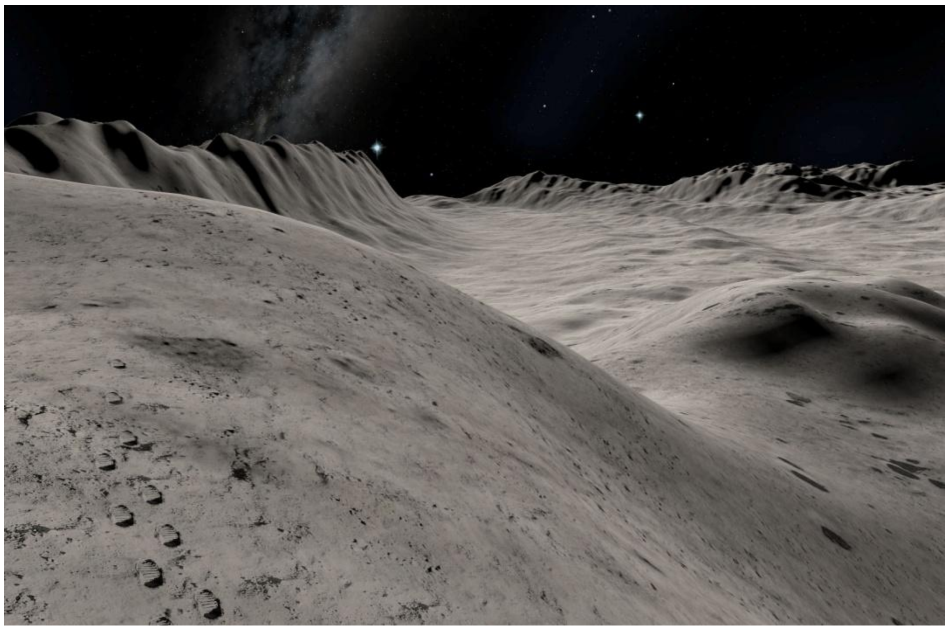


Figure 16: Our fBm detail noise displacement map.



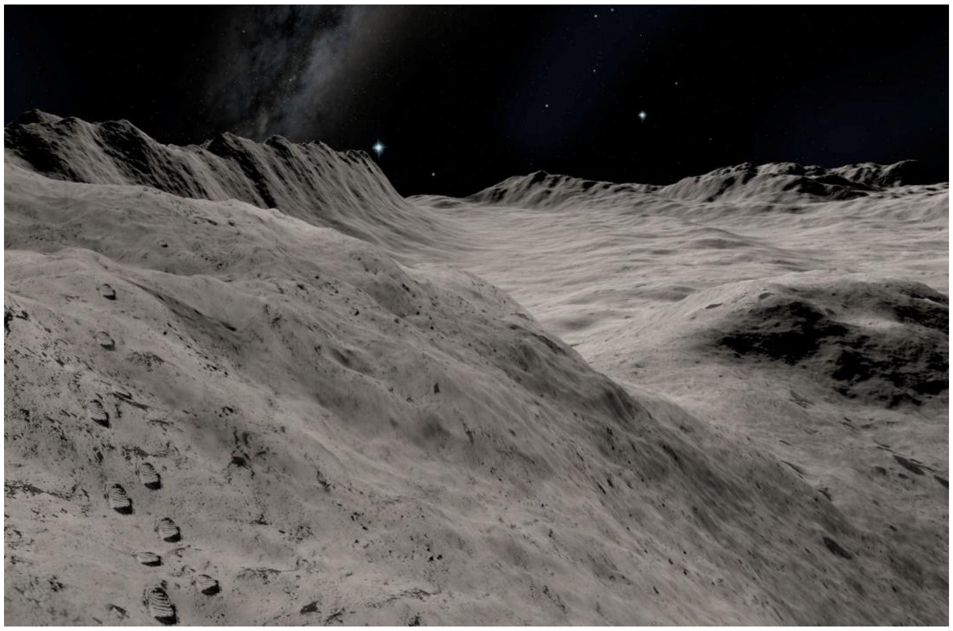
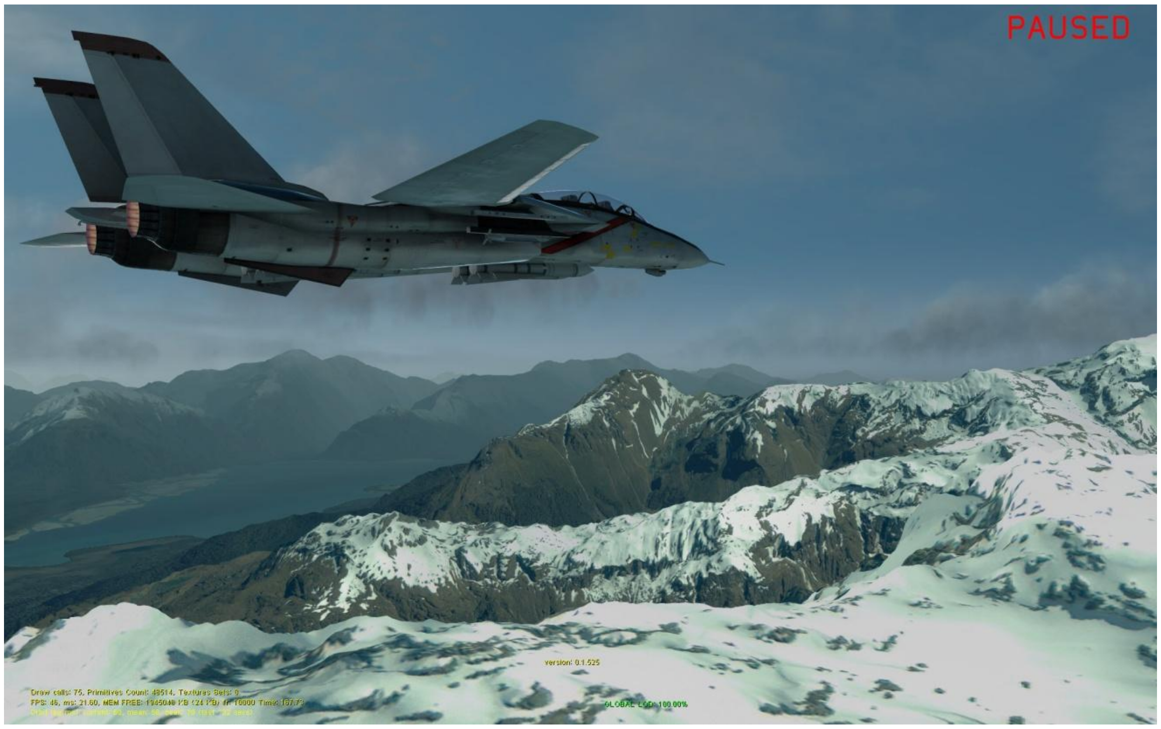


Figure 17a: The low resolution displacement map alone (above) and Figure 18b the result of adding detailed displacement (below).

Upgrade Path for Game Assets

It is often time-consuming to make changes to game assets and/or a content production process. This is often a barrier to adoption of new technology. However, our technique can provide an easy upgrade path to DirectX 11 tessellation for existing game assets. This was exploited in *Tom Clancy's H.A.W.X. 2* from Ubisoft.

Prior to adding DirectX 11, the *H.A.W.X. 2* engine used a similarly low-resolution height map to define the world; 1025x1025 samples covered each entire game level of 128x128km. The height maps in *H.A.W.X. 2* were authored by artists to represent specific real locations in the world. Our sample generates the equivalent displacements procedurally, using fractal ridge noise. But the two lower-resolution maps are similar in their scale, content and relationship to the detailed displacement map. In *H.A.W.X. 2,* the existing low-resolution height maps were used – unaltered – for the DirectX 11 version. The height maps were re-sampled with bi-cubic magnification to 4096x4096 and detailed displacement was added, exactly as in this sample. As Figure 19 shows, the result is highly detailed and realistic. No new assets were required, apart from the 5 octave fBm texture.



```
Draw calls: 75, Primitives Count: 48514, Textures Sets: 0
FPS: 46, ms: 21.60, MEM FREE: 1945048 KB (24 KB) fr: 10000 Time: 167.73
[illegible]
```

Figure 19: Different displacement detail scales used in *Tom Clancy's H.A.W.X 2*, courtesy of Ubisoft.

Performance

Rendering Efficiency

Creating geometry on the GPU with hardware tessellation is highly efficient. The GPU takes on most LOD calculations. On a mid-range GeForce GTX460, at

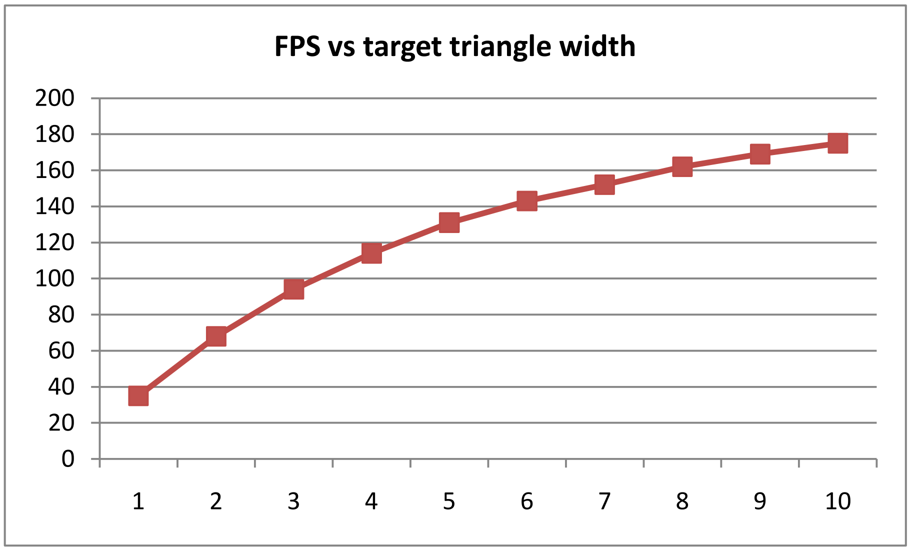
1920x1200 resolution, our sample runs at 92 FPS for the default view. The target triangle size defaults to 3 pixels wide and at this setting the default view contains 2.5 million triangles.

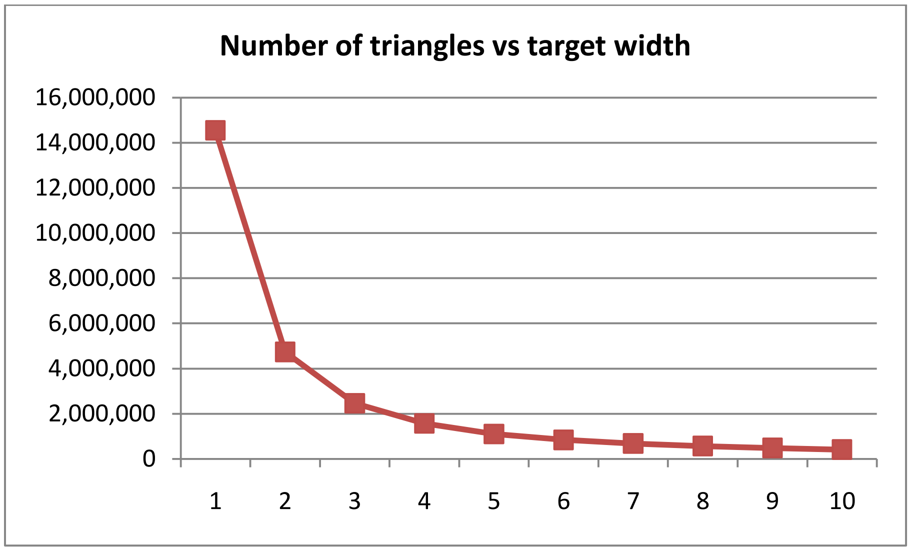
Some of this cost is due to the non-terrain objects and the complex pixel shader. Without the sky box and stars and with a trivial pixel shader, the terrain on its own runs at 102 FPS. The pixel shader is deliberately quite complex because this is representative of many terrain engines in games.

Our arrangement of tiles, patches and instancing is also highly efficient in its use of the Direct3D API. Only one draw call is executed per tile ring. So the entire 2.5 million polygon terrain – including sophisticated adaptive LOD – requires a mere three draw calls.

The table below shows how the performance and number of triangles varies with target triangle size.

Tris size	FPS	# Tris
1	35	14,546,300
2	68	4,736,714
3	94	2,465,020
4	114	1,569,450
5	131	1,106,962
6	143	841,350
7	152	683860
8	162	570924
9	169	484,998
10	175	414,516





Figures 20a-c: The effect of target triangle size on FPS and total triangle count.

Tessellation LOD Targets

Careful choice of triangle size is important for maximum GPU rendering efficiency. Rendering too many triangles that are very small can negatively impact rasterizer efficiency (Demers, 2010). On the other hand, we wish to render many small triangles to produce a highly-detailed, natural terrain with no visible straight edges.

The tessellation LOD algorithm presented here allows easy control over these factors. The hull shader attempts to render triangles at a constant target size in screen-space which is exactly what is required for the best trade-off between rasterizer efficiency and geometric detail. A game can easily adjust the target triangle size to match the capabilities of the user's hardware.

Figure 21 shows the visual result of the algorithm in wireframe. The target triangle size is artificially large to make the visualization of the polygons clearer. The triangle size is approximately constant, irrespective of distance from the view point or quad patch size (the checkerboard pattern). However, some areas of under- and over-tessellation are shown. The algorithm may fail to achieve the target triangle size for several reasons:

1. The hardware imposes limits and the tessellation factor can only vary between 1 and 64. The upper limit often affects quads close to the eye-point. The non-uniform patches attempts to address this. But the size distribution is not guaranteed to avoid the limits.
2. The screen-space LOD algorithm contains multiple approximations. To avoid aliasing, the quad patch edges are enclosed by bounding spheres to deliberately avoid any dependence on the orientation of the edge. Also, the displacement is only approximately taken into account – by displacing the quad patch corners. There is often significant displacement within a patch that is not accounted for. The LOD calculation can be improved by adding

more knowledge of the patch shape after displacement. More sophisticated LOD calculations are discussed in (Cantlay, 2008).

3. Using fractional partitioning results in smaller sliver triangles within a patch that correspond to the fractional part of the tessellation pattern. This can only be avoided by using integer partitioning which results in small pops between tessellation factors.

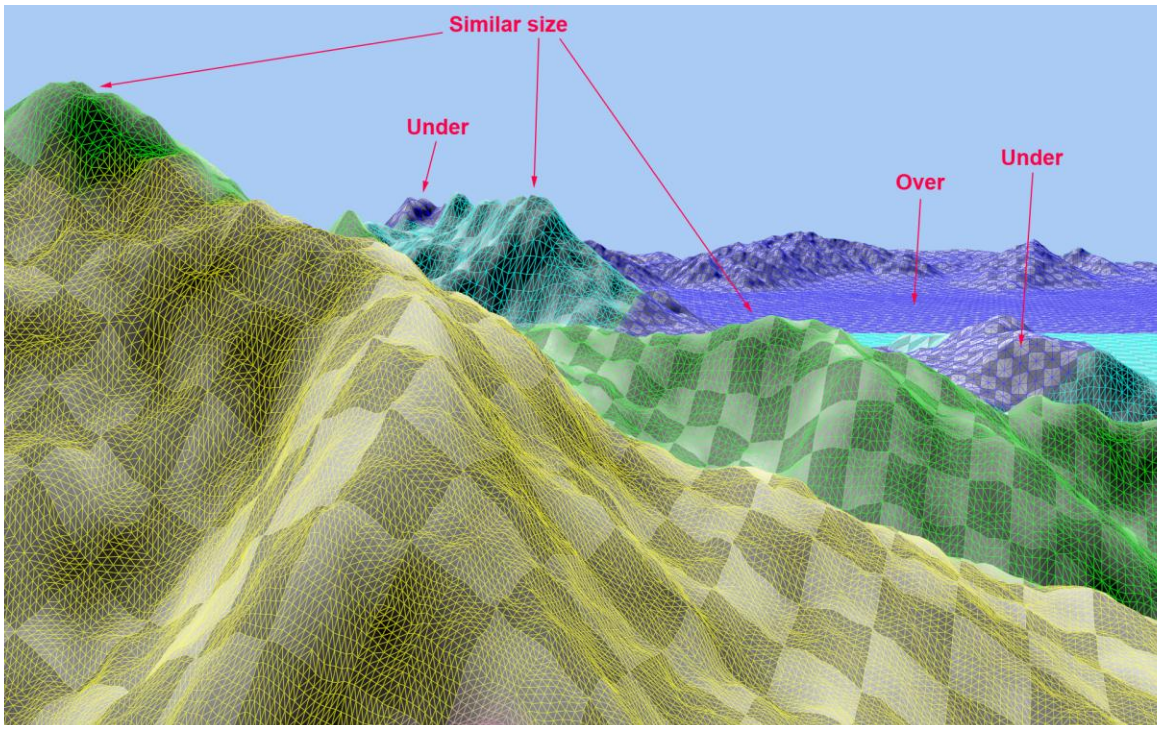


Figure 21: Performance of the tessellation LOD and its visual effect on triangle size.

Tessellation Level Histograms

Tom Clancy's H.A.W.X. 2 from Ubisoft uses a very similar terrain technique. The value of `g_tessellatedTriSize` used by the game is 6 which should result in triangles of 18 pixels in area. We have done some detailed analysis on how well the technique meets that target in *H.A.W.X. 2*.

Average triangle size was computed by adding Direct3D queries. For each draw call, we measured both the number of primitives rasterized (using D3D11_QUERY_DATA_PIPELINE_STATISTICS::CInvocations) and the result of an occlusion query with the depth test set to ALWAYS. Hence, the figures are averages per draw call or 8x8 patches.

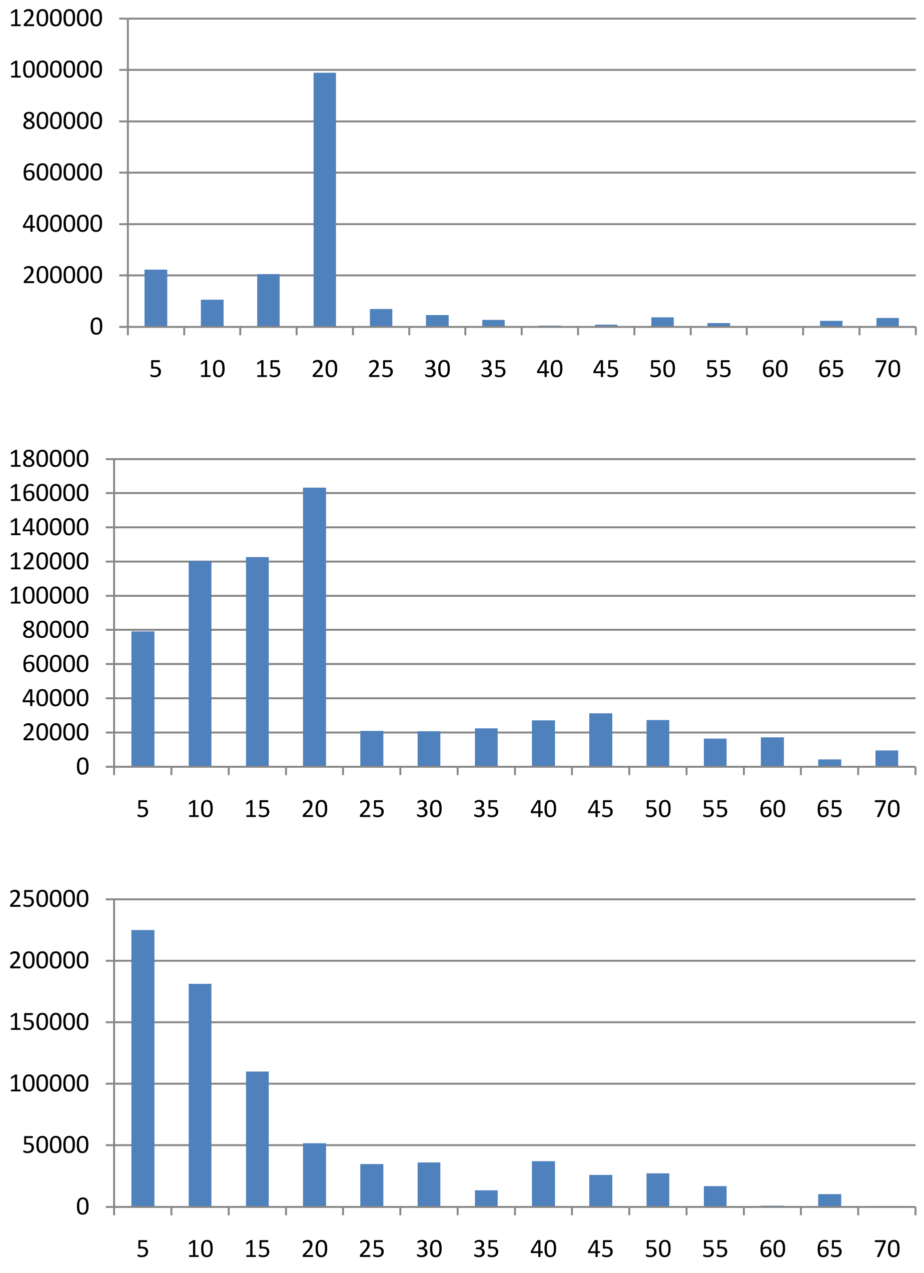


Figure 22: Terrain triangle size distribution histograms measured for *Tom Clancy's H.A.W.X. 2*. Triangle count (vertical) vs. triangle size buckets (horiz).

The histograms in Figure 22 show the distribution of triangle size at three representative points in the game. The first two graphs show that the algorithm is often perfectly on-target for 18-pixel triangles. The third graph shows that larger numbers of small polygons can be generated in some view points. The third graph corresponds to a low-altitude view-point, close to the ground. Although the third histogram is not ideal according to our own triangle-size goal, the game still runs at 127 FPS at this point on a mid-range GeForce GTX460.

Displacement Map Sampling and Aliasing

Displacement mapping from a texture is a sampling operation. It can alias like any texturing. Almost nothing in our sample explicitly addresses aliasing – there is no MIP-mapping of the displacement maps and our choice of tessellation levels does not attempt to observe the Nyquist rate.

Moreover, our choice of fractional partitioning causes the tessellated vertices to continuously slide through the patches. This is bound to exacerbate any aliasing as the sample positions are constantly in motion. See (Sylvan, 2010) for a discussion of displacement map aliasing with DirectX 11 tessellation.

Yet our sample does not suffer from obvious aliasing. *Tom Clancy's H.A.W.X. 2* takes the same approach and also works extremely well. We believe that aliasing is minimized for several reasons:

1. The high tessellation levels often result in sampling frequencies that are high relative to the coarse displacement map. So it is mostly over-sampled.
2. The fine tessellation and the screen-space adaptive LOD means that any aliasing is small in screen-space. Hence, it is visually insignificant.
3. The fractal construction of the displacement maps minimizes high frequencies: the higher frequencies are added with successively smaller amplitudes. A detail map that contains ridge noise does not work so well because it contains sharp edges and large-amplitude high frequencies.

Aliasing can be seen. It is most obvious in the detail displacement map, especially if the displacement amplitude is increased.

MIP-mapping and careful choice of vertex locations can eliminate aliasing.

Scrolling in Displacement Map Generation

To implement an infinite landscape, we keep the terrain tiles and quad patches fixed relative to the eye and move the displacement map as the view point moves in the horizontal plane. When the view point moves, the coarse displacement map is regenerated by a render-to-texture on the GPU. This implementation is only possible because the sample uses purely procedural terrain.

Aliasing is much more visible if the displacement map is allowed to move continuously, relative to the eye. To avoid this, we snap the translation of the displacement map co-ordinates. The displacement map only moves in multiples of the largest patch size. Thus, the positions of the patch corner vertices remain constant, relative to the contents of the coarse displacement map. The snapping fix can be disabled by setting `SNAP_GRID_SIZE` to zero in TerrainTessellation.cpp (comment out its initialization).

This source of aliasing is only present because of our procedural generation of the generic, coarse displacement map. More commonly in terrain engines and games, the coarse displacement would be generated offline to represent specific real-world or game locations, with specific features such as mountains and valleys. The necessary work-around is not likely to be necessary in such implementations.

Running the Sample

Most controls and GUI elements should be obvious. Press F1 for some on-screen description of the controls.

Hardware tessellation can be disabled with a check-box. Note that the non-tessellated rendering simply renders each quad patch as two triangles and displaces the vertices in the vertex shader. No attempt is made to make a crack-free non-tessellated surface.

In wireframe mode, the line colour approximately indicates the tessellation level. However, actual tessellation level is set per edge and patch center, whereas the line colours are set at patch corners and interpolated across the patch. So it the colour is not an exact indication of tessellation level.

Debug Options

For a crack-free surface, careful checking and debugging is essential. To assist testing, it is best to disable the sky dome and set the background to a bright, contrasting colour.

The sample also has useful debug camera controls. It maintains two view-points. One is used for the tessellation LOD calculations; the other is the centre of projection. Usually, they are both altered together by the movement controls ("Eye & LOD" radio button). But it is possible to freeze one and move only the other. If you select the "Eye only" radio button, the centre of projection moves and the LOD stays frozen. This is very useful for finding cracks and examining LOD behavior close-up. Many of the screenshots in this document use the "Eye only" view. The "LOD only" button does the obvious converse – fix the centre of projection and move the "eye" position used for LOD.

Appendices

Real Star Rendering

The files Stars.cpp and Stars.fx implement rendering of stars as individual polygons, based on real-world star data. Although we also draw a background cube-map, its resolution is not nearly sufficient to render pixel-sized stars.

Lunar Data

Our sample uses lunar data and shows an alien world, rather than something earth-like. We choose this for several reasons:

1. We use basic fractal terrain. The lack of an erosion model is accurate and realistic for the moon.
2. The result is not likely to suffer from the "uncanny valley" problem – being almost realistic, but not quite correct and disconcertingly odd. As an alien world, it is intended to be odd and eerie.
3. Plenty of NASA photographs are available free of copyright.

The use of the technique in *Tom Clancy's H.A.W.X. 2* as seen in Figure 2 and Figure 19 demonstrates that it is very applicable to real-world art assets.

Bibliography

Andreas Bærentzen, S. L. (2006). Single-Pass Wireframe Rendering. *Siggraph* . ACM.

Cantlay, I. (2008, August). *Siggraph 2008*. Retrieved from NVIDIA Developer Zone: http://developer.nvidia.com/object/siggraph-2008-terrain.html

Cebenoyan, C. (2010). *DirectX 11 Overview*. Retrieved from GPU Technlogy Conference: http://nvidia.fullviewmedia.com/gtc2010/0920-a5-2157.html

Demers, E. (2010, November 29). *Tessellation for All*. Retrieved from AMD Blogs: http://blogs.amd.com/play/2010/11/29/tessellation-for-all/

F. Kenton Musgrave, D. S. (2002). *Texturing and Modeling, Third Edition: A Procedural Approach*. Morgan Kaufmann.

Fournier, F. &. (1982). Computer Rendering of Stochastic Models. *Communications of the ACM* .

Samuel Gateau. (2007). *Solid Wireframe*. Retrieved from NVIDIA Developer Zone: http://developer.download.nvidia.com/SDK/10.5/direct3d/Source/SolidWireframe/Doc/SolidWireframe.pdf

Sylvan, S. (2010, April 18). *The problem with tessellation in DirectX 11*. Retrieved from A Random Walk Through Geek-Space: http://sebastiansylvan.wordpress.com/2010/04/18/the-problem-with-tessellation-in-directx-11/

Ulrich, T. (2002). Rendering Massive Terrains using Chunked Level of Detail Control. *Siggraph*.

Walbourn, C. (2010, October 27). *June 2010 HLSL Compiler Issue with Tessellation*. Retrieved from MSDN Blogs: http://blogs.msdn.com/b/chuckw/archive/2010/10/27/june-2010-hlsl-compiler-issue-with-tessellation.aspx