9 Dynamic Tessellation

Earlier in the book, the new tessellation-specific stages of the pipeline were introduced (Chapter 3) and the key parameters and implementation details were discussed in depth (Chapter 4). However, neither of these sections provided real-world demonstrations of this key new Direct3D 11 technology.

In the introduction to Chapter 4, two common approaches to tessellation were introduced, namely *subdivision* and *higher-order surfaces.* This chapter offers demonstrations of both, to allow comparison between the two methods, and to demonstrate how the various parameters and new shader units work together to produce a final output.

Subdivision is a form of tessellation based on refinement, in which we progressively add more detail (in the form of additional triangles) until the final mesh better represents an ideal shape. The first section of this chapter, "Terrain Tessellation," follows this approach.

Thinking about higher order is closer to the understanding that tessellation is about representing curved or smooth surfaces. The ideal mesh will usually be described in terms of a parametric mathematical equation, typically quadratic or cubic; the second section of this chapter demonstrates this.

Broadly speaking, using subdivision is more algorithmic, while using higher-order surfaces is more mathematical. We will explore the differences between these two approaches throughout this chapter, and will also build on the earlier introduction to the tessellation stages to provide a well-rounded understanding of the technology.

9.1 Terrain Tessellation

Terrain rendering is probably one of the most common forms of procedural or run-time generated graphics in the real-time rendering space. Unlike the majority of other art assets that will be rendered to make the final image, the terrain is often created by software algorithms, rather than hand-sculpted by an artist (however, a combination of both is common). A typical input is a *height map*—a monochrome texture where each pixel represents the elevation of the terrain at that particular point on the grid.

Using terrain rendering can easily require large data sets, both in terms of textures and geometry. It is not uncommon to have huge draw distances, stretching from the immediate foreground where the camera is located, out to the distant horizon. Many algorithms have been invented and refined over the years to efficiently solve this problem and, for the most part, the problem is well understood. As hardware performance and features have improved (such as the move from CPU to GPU processing) these algorithms have been adapted to take advantage.

Direct3D 11 's tessellation capabilities may not revolutionize this problem space, but they do offer some interesting alternatives. Terrain rendering stands as a good example of the new functionality. Most current algorithms require at least some intervention and processing by the CPU before the GPU does the actual rendering; but with Direct3D 11 it is now possible to offload all processing to the GPU. Both existing "classic" algorithms can be adapted to suit this new hardware and the reverse is also true—new hardware opens up avenues for wholly new algorithms and rendering approaches.

9.1.1 GPU-Accelerated Interlocking Tiles Algorithm

Figure 9.1 shows a nai've rendering of a traditional heightmap-based terrain. The geometry is uniformly distributed across the entire terrain, with no regard to the characteristics of the terrain or viewer. Figure 9.2 shows the same inputs rendered using the interlocking terrain tiles algorithm algorithm introduced in this section—detail is now applied only where it is

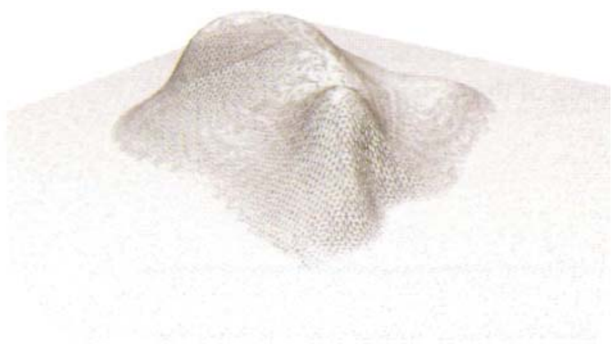


Figure 9.1. Naive rendering.

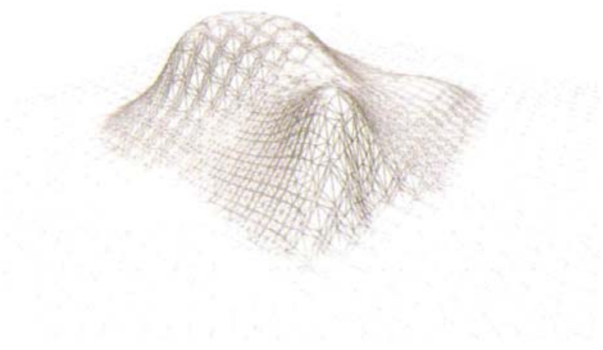


Figure 9.2. Interlocking Terrain Tiles Algorithm.

legitimately required, and performance is no longer wasted by extra geometry that has no impact on the final image.

In "Simplified terrain using interlocking tiles," (Snook, 2001), Greg Snook introduced an algorithm that mapped very nicely to early GPU architectures—around the time when hardware transform and lighting were possible, but slightly predating programmable shader units.

The algorithm was particularly useful because it didn't require modifying the vertex or index buffers; instead, is used multiple index buffers to provide a LOD mechanism. This removed any need for the CPU to modify resources used for rendering, an operation that both then and now can be prohibitively expensive, but which is important for leveraging hardware transform and lighting capabilities. By simply rendering with a different index buffer, Snook's algorithm could alter the LOD and balance performance and image quality.

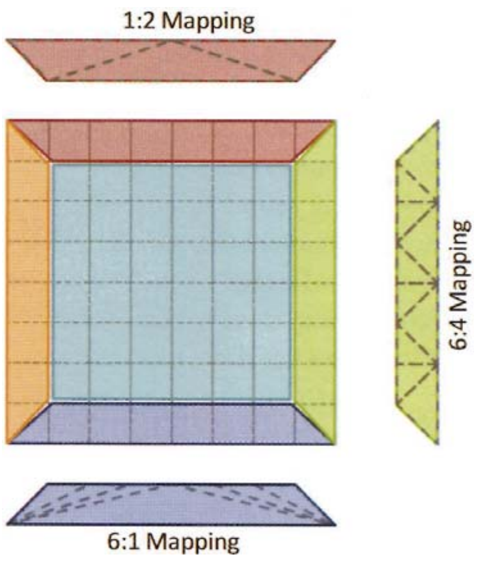


Figure 9.3. Index buffer layout for a single tile.

The algorithm breaks up the 2D height-map texture into a number of smaller tiles, with each tile representing a 2^n+l-dimension area of vertices (9x9 was used in the original implementation). This vertex data was fixed at load-time and simply represented the height at each corresponding point on the height map, which also corresponds to the highest possible detail that can be rendered.

For each level of detail, there are 5 index buffers. For n=3, there are 4 LODs (n+1) in a 9-dimensional area of vertices (2^3+l=9). The layout of the regions that these index buffers represent is depicted in Figure 9.3.

The five index buffers represent the central area of the tile, plus one index buffer for each of the four edges shown in Figure 9.3. These are each assigned a different color. The break-out edge diagrams show examples of mappings between different LODs, shown in innenouter ratios. An observant reader might notice that this is suspiciously similar to the quad tessellation introduced in Chapter 4!

The index buffer for the middle area represents all but the outer ring of vertices, so for the 9x9 grid used in our example, it will always represent 7x7 vertices, regardless of the chosen level of detail. The four "skirt" index buffers essentially provide a mapping between neighboring tiles. It is possible to match either the higher or lower LOD, but it's more common to blend down toward the lower one as shown in Figure 9.4. Technically, a jump between any two levels of detail is possible, but as mentioned in Chapter 4, this may

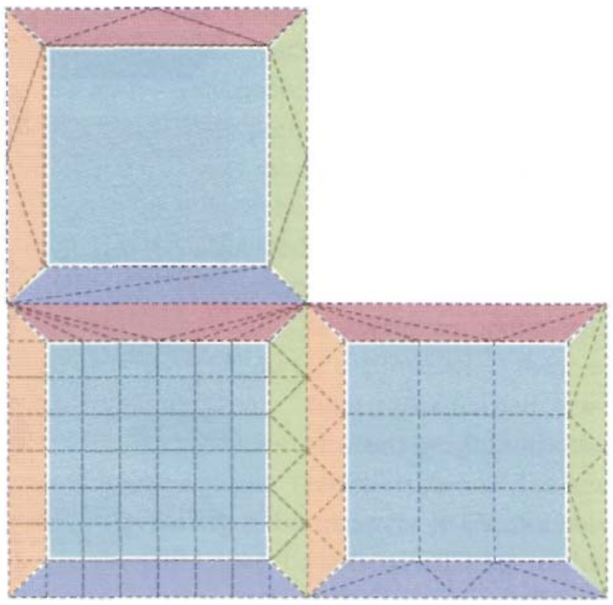


Figure 9.4. Neighboring tiles blending down.

cause visual *popping* to become noticeable, detracting from the final result. The simplest way to avoid this popping is to only transition between neighboring LOD levels, or in severe cases, to consider vertex blending or other morphing enhancements.

This algorithm was published around the time that Direct3D 8 was current, and could only use a single pair of index and vertex buffers. Consequently, it became a useful trick to use DrawIndexedPrimitive() parameters to offset the base vertex for rendering each patch, to avoid unnecessary state changes (an expensive operation with previous versions of Direct3D). In addition, due to the small storage size required for an index, it was possible to store all unique combinations and simply tweak the draw call parameters to change the LOD for each patch.

This means that there can be a great disadvantage, because of the potentially large number of draw calls. The GPUs of the time could not dynamically select which index buffer to use for each segment, so this had to be set up per-tile by the CPU, requiring a draw call for each patch. Clearly, this is not good, given that developers spend significant effort to reduce the number of these calls. The often-cited target of 500 or fewer draw calls per frame thus effectively limits rendering to 484 tiles per frame in a 22x22 grid (sqrt(500) = 22.36, which when rounded down to 22x22 is 484). The use of quad trees[1] and frustum culling[2]

[1] A two-dimensional terrain height-map would be subdivided in half along both the *X* and Y axis to produce smaller tiles; each subdivision would then result in 4 new tiles. Being hierarchical allows for fast rejection or acceptance of large areas of terrain.

[2] Culling is a mathematical test to determine if geometry is visible given the current view properties.

usually meant this wasn't a huge problem, but it definitely one that had to be handled. Additionally, this handling stole CPU cycles that could have been used for better things and further coupled the CPU and GPU.

One could also reduce draw calls by increasing the tile size—going from 9x9 to 17x17 or even 33x33 was perfectly legitimate. However, this made the algorithms other drawback, popping, even more noticeable. As discussed in Chapter 4 and earlier in this section, popping can be a very unsightly and irritating artifact. And in the context of terrain rendering, it can be even more irritating for the viewer, because of the viewing angles used. The primary axis of change (vertical) when moving between levels of detail also tends to be the primary vertical axis (7) on the final image; hence, problems are more noticeable than if the axis of change were lined up with the view direction (such as into and out from the final image).

Typically, the biggest and most noticeable pops are transitions at the lower end of the LOD scale, such as from 0 to 1, where the geometric difference between the two states is highest. Transitioning at the top-end, such as from 4 to 5, won't be as noticeable, since the mesh is already quite detailed, and the triangles are relatively small, making the geometric difference small. Large changes to the silhouette of the terrain (such as mountains on the horizon) tend to be particularly noticeable due to their prominence in the final image, and should ideally be avoided. Unfortunately, the simplest metrics for deciding the level of detail will typically set the geometry farthest from the camera to the lowest level of detail. However, this can be mitigated with a more intelligent metric.

In such cases, working around this problem becomes non-trivial, because we do not want to dynamically change the underlying buffers (as stated earlier, this is typically an expensive operation, negating many benefits of using this algorithm). Also, in the days of Direct3D 8 and Direct3D 9, vertex and geometry processing on the GPU was still relatively simplistic. The first good solution to the problem was using more elaborate schemes for LOD selection—weighting by screen contribution (projecting the bounding box to the screen and using the 2D area as the LOD scalar) could handle the case where distant objects on the horizon popped unnecessarily. Vertex blending[3] was trickier to implement, but was also an option that allowed for interpolation between detail levels.

Ultimately, the high number of draw calls and the popping artifacts were the major problems in an otherwise GPU-friendly algorithm, and this approach was not used extensively as it possibly merited. However, the Direct3D 11 implementation suffers from neither of these issues—a single draw call can render the entire landscape, and the tessellator can handle smooth transitions between levels of detail.

The Implementation

As a high-level overview, this demonstration of Direct3D 11 tessellation focuses on the input assembler, hull shader, and domain shader stages. The sample application will build

[3] A method commonly used in animation to interpolate (blend) between two vertex positions based on weights associated with dynamic transformation matrices.

vertex and index buffers and texture resources representing the inputs to the pipeline, and all remaining functionality will be implemented in HLSL shaders.

Tn this case, the vertex shader will be responsible for transforming model-space geometry into world-space and nothing else. The majority of the work will be done in the hull shader constant function, which determines the LOD for the current patch, using the patch being rendered, as well as its four immediate neighbors. The main hull shader stage will then act as a simple pass-through of the four corner vertices for the current patch. The domain shader will then take the tessellated points and will implement displacement mapping to generate the actual terrain geometry that will finally be rasterized.

Here are the steps for implementing the interlocking terrain tiles algorithm.

Step 1: Creating the Input Data

Three sources of input data need to be created for this algorithm to work: a vertex buffer, and index buffer, and heightmap texture.

Listing 9.1 creates the vertex buffer. It is very straightforward, as it is simply a grid of points that bounds new geometry generated in the space between these points.

```
// Set up the actual resource
SAFE_RELEASE( m_pTerrainGeometry );
m_pTerrainGeometry = new GeometryDXll( );

// Create the vertex data
VertexElementDXll *pPositions
    = new VertexElementDXll( 3, (TERRAIN_X_LEN + 1) * (TERRAIN_Z_LEN + 1) );
    pPositions->m_SemanticName = "CONTR0L_P0INT_P0SITI0N";
    pPositions->m_uiSemanticIndex = 0;
    pPositions->m_Format = DXGI_FORMAT_R32G32B32_FLOAT;
    pPositions->m_uiInputSlot = 0;
    pPositions->m_uiAlignedByteOffset = 0;
    pPositions->m_InputSlotClass = D3D11_INPUT_PER_VERTEX_DATA;
    pPositions->m_uiInstanceDataStepRate = 0;

VertexElementDXll *pTexCoords
    = new VertexElementDX11( 2, (TERRAIN_X_LEN + 1) * (TERRAIN_Z_LEN + 1) );
    pTexCoords->m_SemanticName = "C0NTR0L_P0INT_TEXC00RD";
    pTexCoords->m_uiSemanticIndex = 0;
    pTexCoords->m_Format = DXGI_FORMAT_R32G32_FLOAT;
    pTexCoords->m_uiInputSlot = 0;
    pTexCoords->m_uiAlignedByteOffset = D3D11_APPEND_ALIGNED_ELEMENT;
    pTexCoords->m_InputSlotClass = D3D11_INPUT_PER_VERTEX_DATA;
    pTexCoords->m_uiInstanceDataStepRate = 0;

Vector3f *pPosData = pPositions->Get3f( 0 );
Vector2f *pTCData = pTexCoords->Get2f( 0 );

float fWidth = static_cast< float >( TERRAIN_X_LEN );
float fHeight = static_cast< float >( TERRAIN_Z_LEN );

for( int x = 0; x < TERRAIN_X_LEN + 1j ++x )
{
    for( int Z = 0; z < TERRAIN_Z_LEN + 1; ++Z )
    {
        float fX = static_cast<float>(x) / fwidth - 0.5f;
        float fZ = static_cast<float>(z) / fHeight - 0.5f;
        pPosDataf x + z * (TERRAIN_X_LEN + 1) ] = Vector3f( fX, 0.0f, fZ );
        pTCData[ x + z * (TERRAIN_X_LEN + 1) ] = Vector2f( fX + 0.5f, fZ + 0.5f );
    }
}

m_pTerrainGeometry->AddElement( pPositions );
m_pTerrainGeometry->AddElement( pTexCoords );
```

Listing 9.1. Creating the vertex buffer.

The TERRAIN_X_LEN and TERRAIN_Z_LEN constants are defined in the context of how many tiles should be generated. Therefore, 1 is added to each of these constants in various places—for example, a 3x3 grid of tiles will require a 4x4 grid of vertices.

It is also worth noting that the raw geometry is defined as being a flat grid on the *XZ* plane in 3D space, and that the height of the terrain is defined entirely along the *Y* axis. Listing 9.1 initializes all height points to 0.0 at this stage.

Listing 9.2, shown below, builds up the index buffer data that defines each patch. This is by far the most complex and most important resource that this algorithm must initialize. The input assembler will use this data to index into the vertex buffer and map all the necessary control points that the hull shader and domain shader need to do their job.

```
// Code below makes reference to the following macro:
#define clamp(value,minimum,maximum) (max(min((value), (maximum)), (minimum)))
for( int x = 0; x < TERRAIN_X_LEN; ++x )
{
    for( int z = 0; z < TERRAIN_Z_LEN; ++z )
    {
        // Define 12 control points per terrain quad

        // 0-3 are the actual quad vertices
        m_pTerrainGeontetry->AddIndex( (z + 0) + (x + 0) * (TERRAIN_X_LEN + 1) );
        m_pTerrainGeometry->AddIndex( (z + 1) + (x + 0) * (TERRAIN_X_LEN + 1) );
        m_pTerrainGeometry->AddIndex( (z + 0) + (x + 1) * (TERRAIN_X_LEN + 1) );
        m_pTerrainGeometry->AddIndex( (z + 1) + (x + 1) * (TERRAIN_X_LEN + 1) );

        // 4-5 are +x
        m_pTerrainGeometry->AddIndex
        (
            clamp(z + 0, 0, TERRAIN_Z_LEN)
```

```
                + clamp(x + 2, 8, TERRAIN_X_LEN) * (TERRAIN_X_LEN + 1)

            m_pTerrainGeometry->AddIndex

                clamp(z + \, 8, TERRAIN_Z_LEN)
                + clamp(x + 2, 8, TERRAIN_XJ_EN) * (TERRAIN_X_LEN + 1)
            ) i
            // 6-7 are +z
            m_pTerrainGeometry->AddIndex

                Clamp(z + 2, 8, TERRAIN_Z_LEN)
                + clamp(x + 0, 0, TERRAIN_X_LEN) * (TERRAIN_XJ_EN + 1)

            m_pTerrainGeometry->AddIndex

                clamp(z + 2, 8, TERRAIN_Z_LEN)
                + clamp(x + 1, 8, TERRAIN_X_LEN) * (TERRAIN_X_LEN + 1)


            // 8-9 are -x
            m_pTerrainGeometry->AddIndex

                clamp(z + 8, 8, TERRAIN_Z_LEN)
                + Clamp(x - 1, 8, TERRAIN_X_LEN) * (TERRAIN_X_LEN + 1)

            m_pTerrainGeometry->AddIndex

                Clatnp(z + 1, 8, TERRAIN_Z_LEN)
                + clamp(x - 1, 8, TERRAIN_X_LEN) * (TERRAIN_X_LEN + 1)
            ) i
            // 10-11 are -z
            m_pTerrainGeometry->AddIndex

                clamp(z - 1, 8, TERRAIN_Z_LEN)
                + clamp(x + 8, 8, TERRAIN_X_LEN) * (TERRAIN_X_LEN + 1)
            ) 3
            m_pTerrainGeometry->AddIndex

                clamp(z - 1, 8, TERRAIN_Z_LEN)
                + clamp(x + 1, 8, TERRAIN_X_LEN) * (TERRAIN_X_LEN + 1)
            ) 3
        }
    }
    // Move the in-memory geometry to be
    // an actual renderable resource
    m_pTerrainGeometry->LoadToBuffers();
    m_pTerrainGeometry->SetPrimitiveType( D3D1l_PRIMITIVE_TOPOLOGY_12_CONTROL_
    POINT_PATCHLIST );
```

Listing 9.2. Creating the index buffer.

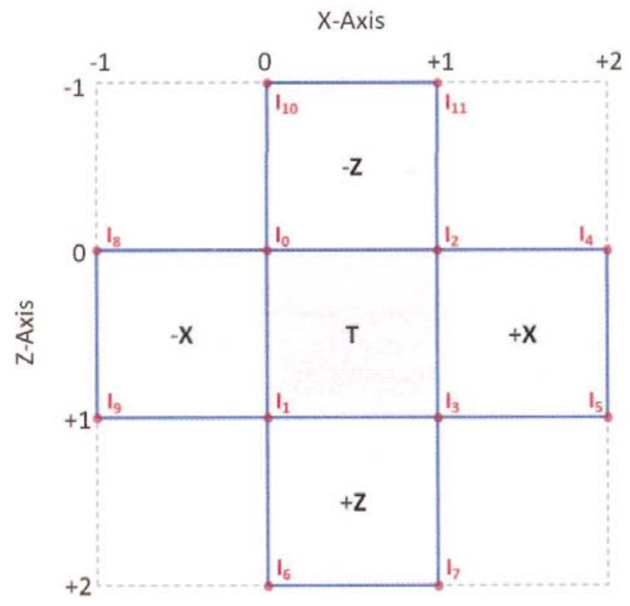


Figure 9.5. Input assembler layout for a single tile.

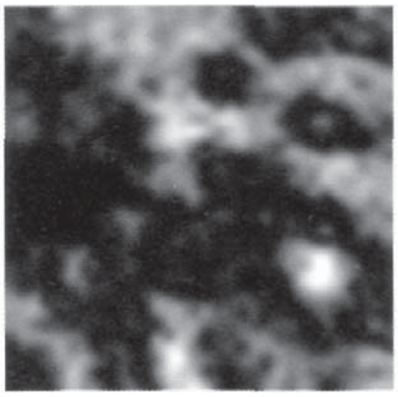


Figure 9.6. Example heightmap.

As shown in Figure 9.5, the first 4 control points (I_0-I_3 surrounding the red square) define the area that will have geometry generated and rendered, while the 8 control points defining the neighboring cells (I_4—I_{11}) are purely informational and allow the hull shader to correctly assign the tessellation factors. Note that the code makes use of the clamp() macro, which literally does what it suggests—clamps the first parameter to be in the range defined by the second and third parameters. The effects of this will be better demonstrated later, but for patches around the edge of the terrain, this sets the midpoint of the neighboring patch to be the midpoint of the edge, instead. This gives good results, is significantly better than any sort of modulus/wrapping/border operator, and is more analogous to texture gutters.

The final input required is the heightmap itself. The code in Listing 9.3 is a bit of an anticlimax compared with the previous two—it simply loads Figure 9.6 into memory as a texture resource.

```
// Load the texture
m_pHeightMapTexture = m_pRendererll->LoadTexture
                        ( std::wstring( L". ./Data/Textures/TerrainHeightMap.png" ) );
```

```
// Store the height/width to the param manager
D3D11_TEXTURE2D_DESC d = mjDHeightMapTexture->rn_pTexture2dConfig->GetTextureDesc();
Vector4f vTexDim = Vector4f(
                                        static_cast<float>(d. Width)
                                        , static_cast<float>(d.Height)
                                        , static_cast<float>(TERRAIN_X_LEN)
                                        , static_cast<float>(TERRAIN_Z_LEN)
                                    );
m_pRendererll->m_pParamMgr->SetVectorParameter( L"heightMapDimensions", &vTexDim );

// Create the SRV
ShaderResourceParameterDXll* pHeightMapTexParam = new ShaderResourceParameterDXllQj
pHeightMapTexParam->SetParameterData( &m_pHeightMapTexture->m_iResourceSRV );
pHeightMapTexParam->SetName( std::wstring( L"texHeightMap" ) );

// Map it to the param manager
m_pRendererll->m_pParamMgr->SetShaderResourceParameter
                                    ( L"texHeightMap"j m_pHeightMapTexture );

// Create a sampler
D3D11_SAMPLER_DESC sampDesc;
sampDesc.AddressU = D3D11_TEXTURE_ADDRESS_CLAMP;
sampDesc.AddressV = D3D11_TEXTURE_ADDRESS_CLAMP;
sampDesc.AddressW = D3D11_TEXTURE_ADDRESS_CLAMP;
sampDesc.BorderColor[0] =
    sampDesc.BorderColorfl] =
    sampDesc.BorderColor[2] =
    sampDesc.BorderColor[3] = 0;
sampDesc.ComparisonFunc = D3D11_C0MPARIS0N_ALWAYS;
sampDesc.Filter = D3D11_FILTER_MIN_MAG_MIP_LINEAR;
sampDesc.MaxAnisotropy = 16;
sampDesc.MaxLOD = D3D11_FL0AT32_MAX;
sampDesc.MinLOD = 0.0fj
sampDesc.MipLODBias = 0.0f;
int samplerState = m_pRendererll->CreateSamplerState( SsampDesc );

// Set it to the param manager
m_pRendererll->m_pParamMgr->SetSamplerParameter( L"smpHeightMap", SsamplerState );
```

Listing 9.3. Loading the heightmap as a texture.

Care needs to be taken when matching the size of the heightmap to the geometry being created. Keeping the heightmap 4-8 times larger than the underlying geometry typically works well. If it is less than this, the tessellation introduces very little new detail, and if it is much higher, this can lead to quite substantial differences between neighboring LODs. This consideration about sampling rate is a general one for tessellation—if extra generated geometry has no higher-frequency data to represent, there is little if any benefit in adding it. The example code for this uses a 512x512 monochrome texture for rendering a grid of 32x32 tiles.

Assuming successful execution of the previously described code, the application has now created all of the resources required for rendering.

Step 2: The Hull Shader

The previous step was all about generating the source data and binding it to the pipeline, in preparation for this step and the one after it. It is here that the first code is written to directly implement the algorithm described.

The introduction to Chapter 4 described the hull shader as being comprised of two pieces of HLSL—the patch constant function and the control point shader. Listing 9.4 implements our hull shader.

```
struct VSJXJTPUTww
{
    float3 position : WORLD_SPACE_C0NTROL_POINT_P0SITION;
    float2 texCoord : CONTROL_POINT_TEXCOORD;
};

struct HS_OUTPUT
{
    float3 position : C0NTROL_POINT_P0SITION;
    float2 texCoord : CONTROL_POINT_TEXCOORD;
};

[domain("quad")]
[partitioning("fractional_odd")]
[outputtopology("triangle_cw")]
[outputcontrolpoints(4)]
[patchconstantfunc("hsPerPatch")]
HS_OUTPUT hsSimple
            (
                InputPatch<VS_OUTPUT, 12> p,
                uint i : SV_OutputControlPointID
            )
{
    HS_OUTPUT o = (HS_OUTPUT)0;

    o.position = p[i].position;
    o.texCoord = p[i].texCoord;

    return o;
}
```

Listing 9.4. The hull shader control point function.

Listing 9.4 is the control point shader, which due to the attributes applied is also the main entry point as far as Direct3D is concerned. At first glance, the body of this shader appears to be a simple pass-through/identity operation. While this is partially, true a subtle detail exists in the definition of the InputPatch, which has 12 control points, and in the definition of outputcontrolpoints, which has 4.

As shown in the next section, the domain shader only needs to know about the 4 control points that define the boundary of the tile being currently rendered. Only the hull shader constant function needs the 8 neighboring points to generate tessellation factors. Consequently, there is no need to send the data further down the pipeline. This is good, because unless the GPU driver is extremely clever and automatically removes this data, it, would serve only to increase resource usage unnecessarily.

The patchconstantfunc attribute in Listing 9.4 points to the hsPerPatch entry point in Listing 9.5. This is where Greg Snook's algorithm is really implemented.

```
struct HS_PER_PATCH_OUTPUT
{
    float edgeTesselation[4]   : SV_TessFactor;
    float insideTesselation[2] : SV_InsideTessFactor;
};

HS_PER_PATCH_OUTPUT hsPerPatch
                    (
                       InputPatch<VS_OUTPUT, 12> ip
                       , uint PatchID : SV_PrimitiveID
                    )
{
    HS_PER_PATCH_OUTPUT o = (HS_PER_PATCH_OUTPUT)0;

    // Determine the midpoint of this patch
    float3 midpoints[] =
    {
        // Main quad
        ComputePatchMidPoint(ip[0].position,ip[1].position,ip[2].
              position,ip[3].position)

        // +x neighbor
        , ComputePatchMidPoint(ip[2].position,ip[3].position,ip[4].
              position,ip[5].position)

        // +z neighbor
        , ComputePatchMidPoint(ip[1].position,ip[3].position,ip[6].
              position,ip[7].position)

        // -x neighbor
        , ComputePatcnMidPoint(ip[0].position,ip[i].position,ip[8].
              position,ip[9].position)

        // -z neighbor
        , ComputePatchMidPoint(ip[0].position,ip[2].position,ip[10].
              position,ip[11].position)
    // Determine the appropriate LOD for this patch
    float dist[] =
    {
        // Main quad
        ComputePatchLOD( ntidPointsfaf )

        // +x neighbor
        , ComputePatchLOD( midPoints[l] )

        // +z neighbor
        , ComputePatchLOD( midPoints[2] )

        // -x neighbor
        , ComputePatchLOD( midPoints[3] )

        // -z neighbor
        , ComputePatchLOD( midpoints[4] )
    };

    // Set it up so that this patch always has an interior matching
    // the patch LOD.
    o.insideTesselation[0] =
        o.insideTesselation[l] = dist[0];

    // For the edges its more complex as we have to match
    // the neighboring patches. The rule in this case is:
    //
    // - If the neighbor patch is of a lower LOD we
    //   pick that LOD as the edge for this patch.
    //
    // - If the neighbor patch is a higher LOD then
    //   we stick with our LOD and expect them to blend down
    //   towards us
    o.edgeTesselation[0] = rnin( dist[0], dist[4] );
    o.edgeTesselation[l] = min( dist[0], dist[3] );
    o.edgeTesselation[2] = min( dist[0], dist[2] );
    o.edgeTesselation[3] = min( dist[0], dist[l] );

    return 0;
}
```

Listing 9.5. Hull shader constant function.

The code in Listing 9.5 can be divided into three stages of execution. First, the tile being rendered and its 4 neighbors have their midpoints computed. Second, the distance from each midpoint to the camera is used to generate a "raw" level of detail for each of the 5 patches. And finally, these LOD values are assigned to the 6 output values that Direct3D expects (2 inner factors, SV_InsideTessFactor and 4 edge factors, SV_TessFactor).

The control points input into this stage are all in world space, which is the only transformation that the vertex shader performs. Thus, ComputePatchMidPoint() (see

Listing 9.6) can be done as a simple mean average of the four corners. Two things to note about this section are that for the edge patches where the application will have clamped the inputs, the midpoint is actually the midpoint of the edge, and not the tile; second the indexing into the ip[ ] array matches Figure 9.5, which was part of step 1.

```
float3 ComputePatchMidPoint(float3 cp0, float3 cp1, float3 cp2, float3 cp3)
{
    return (cp0 + cp1 + cp2 + cp3) / 4.0f;
}
```

Listing 9.6. Definition of ComputePatchMidPoint().

With an array of five midpoints generated, these can be compared against the camera position to generate five individual scalars representing the raw LOD for each of the patches. Of particular interest is that the raw values are continuous in nature (as continuous as IEEE-754 floating point values can be!) and there is only clamping between the minLOD and maxLOD values provided by the application. This, when combined with the partitioning attribute set to fractional_odd is all that is necessary to ensure a smooth transition between levels of detail on a frame-by-frame basis.

The ComputePatchLOD() function, shown in Listing 9.7, is very naive and could be greatly improved, but for the purpose of an example it works well. The helper function simply scales linearly between application-provided minimum and maximum levels according to the distance from the camera—the further away, the lower the detail.

```
float ComputeScaledDistance(float3 from, float3 to)
{
    // Compute the raw distance from the camera to the midpoint of this patch
    float d = distance( from, to );

    // Scale this to be 0.0 (at the min dist) and 1.0 (at the max dist)
    return (d - minMaxDistance.x) / (minMaxDistance.y - minMaxDistance.x);
}

float ComputePatchLOD(float3 midPoint)
{
    // Compute the scaled distance
    float d = ComputeScaledDistance( cameraPosition.xyz, midPoint );

    // Transform this 0.0-1.0 distance scale into the desired LOD's
    // note: invert the distance so that close = high detail, far = low detail
    return lerp( minMaxLOD.x, minMaxLOD.y, 1.0f - d );
}
```

Listing 9.7. Definition of ComputePatchLOD().

More intelligent implementations could implement the idea of a near plane as well as a far plane, consider a nonlinear scale (similar to depth buffers) to apply more detail nearer the viewer, and consider using entirely different metrics. One more expensive possibility mentioned earlier is to use screen-space contribution as the LOD scalar; simply project the bounding coordinates to projection space and compute the area. A patch that contributes more pixels to the screen should have a higher LOD.

The final section of Listing 9.8 assigns the raw values to the outputs expected by Direct3D, which hides a couple of subtle details. First, as commented in the code, the choice of interpolating up or down the LOD scale—either is acceptable, but the choice will influence how much new geometry is generated—thus becomes a decision on quality versus performance. Second, and much less obvious, is the effect of the ordering of tessellation factors in the output array.

The Direct3D 11 specification defines, for a quad, that the 0th element is *U*= 0.0, the 1st is *V*= 0.0, the 2nd is *U*= 1.0, and the 3rd is *V*= 1.0. These *U* and *V* coordinates come into play downstream in the domain shader, but their life begins here. Incorrect ordering is a very easy bug to introduce and can generate unexpected results. Figure 9.7 shows this pattern in the context of this implementation.

In this case, the pattern is transposed from the more conventional texture coordinate system, but only for convenience and consistency throughout the implementation—there is no particular reason why it must be this way.

Note that the hull shader constant function has visibility of all upstream data provided by the vertex shader, and that this code, unlike the control point shader function, only produces additional intermediary values and does not remove any existing data.

These six output tessellation factors, stored in the HS_PER_PATCH_OUTPUT struct, are forwarded to the fixed-function tessellator stage, which is the next pipeline stage after the hull shader.

Step 3: The Domain Shader

By the time execution reaches the domain shader, we are only concerned with the patch being rendered. This is confirmed by the fact that it only has visibility of the four control points (CP_0-CP_3 in Figure 9.7) making up the patch itself, and has no knowledge of the four neighboring patches. The fixed-function tessellator has now generated a number of new vertices to match the primitives required to render this tessellated patch.

It is now the job of the domain shader to take individual *UV* coordinates and turn them into

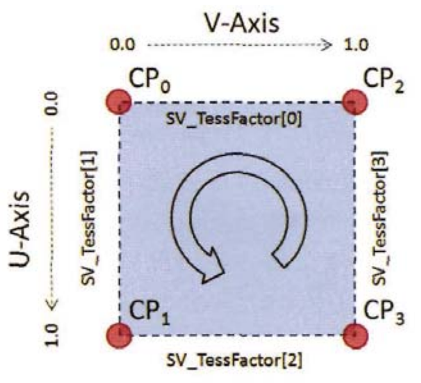


Figure 9.7. Output element ordering for a single tile.

projection space outputs that the rasterizer can work with. In particular, all geometry up until this point has been in world space, but upon exit from the domain shader, it must be in clip space—the only exception to this is if a geometry shader is bound to the pipeline, in which case you can defer this projection until then.

```
struct DS_OUTPUT
{
    float4 position : SV_Position;
    float3 colour : COLOUR;
};

float SampleHeightMap(float2 uv)
{
    // - Must use SampleLevel() so we can specify the mip-map. The DS has no
    //   gradient information it can use to derive this detail...

    // arbitrary bias to make the output more aesthetically pleasing...
    const float SCALE = 3.0f;
    return SCALE * texHeightMap.SampleLevel( smpHeightMap, uv, 0.0f ).r;
}

[domain("quad")]
DS_OUTPUT dsHain( HS_PER_PATCH_OUTPUT input,
                  float2 uv : SV_DomainLocation,
                  const OutputPatch<HS_OUTPUT, 4> patch )
{
    DS_OUTPUT o = (DS_OUTPUT)0;

    // We need to take the three world space
    // coordinates in patch[] and the interpolation
    // values in uvw (barycentric coords) and determine
    // the appropriate interpolated position.
    float3 finalVertexCoord = float3( 0.0f, 0.0f, 0.0f );

    // u,v
    // 0,0 is patch[0].position
    // 1,0 is patchfl].position
    // 0,1 is patch[2].position
    // 1,1 is patch[3].position

    /*
    0--1
      /
     /
    2--3
    */

    finalVertexCoord.xz = patch[0].position.xz * (1.0f-uv.x) * (1.0f-uv.y)
                        + patchfl].position.xz * uv.x * (1.0f-uv.y)
                        + patch[2].position.xz * (1.0f-uv.x) * uv.y
                        + patch[3].position.xz * uv.x * uv.y
                        ;
    float2 texcoord    = patch[0].texCoord * (1.0f-uv.x) * (1.0f-uv.y)
                        + patchfl].texCoord * uv.x * (1.0f-uv.y)
                        + patch[2].texCoord * (1.0f-uv.x) * uv.y
                        + patch[3].texCoord * uv.x * uv.y
                        ;

    // Determine the height from the texture
    finalVertexCoord.y = SampleHeightMap(texcoord);

    // We then need to transform the world-space
    // coord to be a proper projection space output
    // that the rasterizer can deal with. Could delegate
    // to the GSj but no need this time!
    o.position = mul( float4( finalVertexCoord, 1.0f ), mViewProj );

    // Perform a sobel filter on the heightmap to determine an appropriate
    // normal vector
    float3 normal = Sobel( texcoord );
    normal = normalize( mul( float4(normal, 1.0f)j mlnvTposeWorld ).xyz );
    o.colour = min(0.75f, max(0.0f, dot( normal, float3( 0.9f, 1.0f, 0.0f ) ) ) );

    return o;
}
```

Listing 9.8. The domain shader.

Listing 9.8 is all that is necessary to build the final terrain geometry that will be rendered. The domain shader shown above does not really implement a specific part of Greg Snook's original algorithm; rather, it implements a simple form of displacement mapping.

The output of the hull shader and fixed-function tessellator stages essentially just define a pattern on the *XZ* plane for where the heightmap texture should be sampled. Unlike more trivial terrain Tenderers, which generate a uniform grid of sample locations, this implementation generates an increasing density of sample locations, according to the LOD scheme in use. The domain shader simply takes these new points as being the correct ones to generate more detail for—the above code does not actively decide or influence the distribution of geometric detail.

The domain shader presented above can be broken down into three fundamental sections. First, the *UV* coordinate is decoded into a final world-space position on the *XZ* plane. This is a straightforward interpolation that follows the diagram in the preceding section. The same set of equations is used to determine the texture coordinate for this new vertex, which is then used to look up a value from the heightmap texture. These values are put together into the final vertex position and then transformed into projection space for the rasterizer.

The final section is a trivial lighting model to give the rendered terrain a more aesthetically pleasing appearance. Naturally, a more robust implementation would implement proper texturing and a more realistic lighting model. The above code makes references to the Sobel() function, which is shown in Listing 9.9.

```
cbuffer sampleparams
{
    // xy = pixel dimensions
    // zw = geometry dimensions
    float4 heightMapDimensions;
}

float3 Sobel( float2 tc )
{
    // Useful aliases
    float2 pxSz = float2( 1.0f / heightMapDimensions.x, 1.0f
                            / heightMapDimensions.y );

    // Compute the necessary offsets:
    float2 000 = tc + float2( -pxSz.x, -pxSz.y );
    float2 o10 = tc + fioat2(    0.0f, -pxSz.y );
    float2 o20 = tc +  float2(  pxSz.x, -pxSz.y );

    ftoat2 o01 = tc + fioat2( -pxSz.x, 0.0f    );
    float2 o21 = tc +  float2(  pxSz.x, 0.0f    );

    float2 002 = tc + float2( -pxSz.x,  pxSz.y );
    float2 ol2 = tc +   float2(   0.0f,  pxSz.y );
    float2 o22 = tc +  float2(  pxSz.x,  pxSz.y )j

    // Use of the sobel filter requires the eight samples
    // surrounding the current pixel:
    float h00 = SampleHeightMap(o00); // NB: Definition provided in listing 9.8
    float hl0 = SampleHeightMap(ol0)j
    float h20 = SampleHeightMap(o20);

    float h01 = SampleHeightMap(o01);
    float h21 = SampleHeightMap(o21);

    float h02 = SampleHeightMap(o02)j
    float hl2 = SampleHeightMap(o12)j
    float h22 = SampleHeightMap(o22);
    // Evaluate the Sobel filters
    float Gx = h00 - h20 + 2.0f * h01 - 2.0f * h21 + h02 - h22;
    float Gy = h00 + 2.0f * hl0 + h20 - h02 - 2.0f * hl2 - h22;

    // Generate the missing Z
    float Gz = 0.01f * sqrt( max(0.0f„ 1.0f - Gx * Gx - Gy * Gy ) );

    // Make sure the returned normal is of unit length
    return normalize( float3( 2.0f * Gx, Gz, 2.0f * Gy ) );
}
```

Listing 9.9. Definition of the Sobel () function.

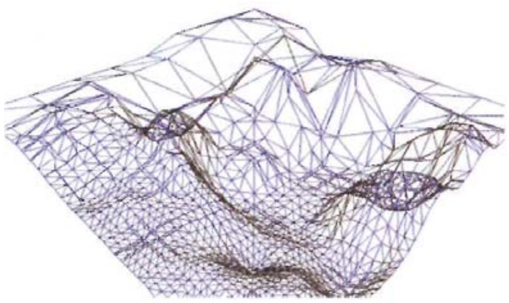


Figure 9.8. Wireframe.

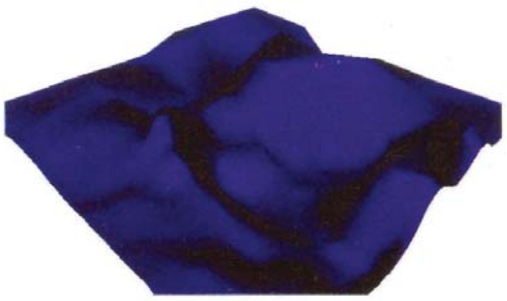


Figure 9.9. Solid.

This is just a simple Sobel operator on the heightmap, a useful trick that efficiently generates acceptable normal vectors from a heightmap by using gradient detection. The important detail to note is that this technique lets the domain shader generate normal vectors in isolation, so that it does not need to be aware of any of its neighboring triangles or the underlying surface. For a displacement-mapping algorithm, this is very useful and greatly simplifies the implementation. Without this trick, the normal vectors would need to be read from a matching texture (which could increase quality), or simple "face normals" would need to be generated by the geometry shader.

The Result

The three steps shown above are all that is necessary to entirely implement Greg Snook's interlocking tiles algorithm on a Direct3D 11 GPU. Figures 9.8 and 9.9 show examples of the output, firstly in wireframe and secondly as a solid-shaded. The following is an analysis of the results:

- Resources:
    - Vertex Buffer: 81 vertices, forming a 9x9 grid, requiring 1,620 bytes
    - Index Buffer: 768 indices, with 12 for each of the 64 patches, requiring 1,536 bytes
    - Texture: 64x64 32-bit texture, requiring 16,384 bytes
    - Total storage: 19.1 KB

  Rendering:
    - 1 DrawIndexed() call
    - 768 vertex shader invocations

- 64 hull shader constant function invocations
- 256 hull shader control point function invocations
- 3,596 domain shader invocations
- 2,872 triangles generated

Regarding the storage requirements, it is worth noting that a CPU implementation of the same geometry pattern could require as much as 70 KB just for the vertex data, and this would vary according to the LODs used for any given frame. For less than one third of this storage, the inputs remain totally fixed and constant, and the underlying runtime and drivers have to handle the varying output sizes internally.

9.1.2 Extending the Interlocking Tiles implementation

It was noted in the previous section that the LOD calculation in Listing 9.7 was very naive. This aspect proves very useful for exploring one of the best enhancements that can be made to this implementation: the LOD equation.

Better results can be achieved on both the quality and performance axes. With a given performance budget (for example, being able to render 1 million triangles per second), for the sake of quality it is preferable to focus tessellation on areas where it is most noticeable—why waste triangles on pieces of terrain that won't benefit from them? An alternative view on the performance axis is that for a given terrain surface, there is no point in over-tessellating and creating more work than necessary; if 650 triangles can represent a given area, why generate 1000?

Figure 9.11 takes the inside tessellation factor for a patch and converts it into a shade of red (high detail), green (mid detail) and blue (low detail). It is immediately obvious that there is little red in the image, just the corners of three patches that are visible along the bottom edge. It is also noticeable that a large part of the final image is comprised of tiles that are shaded blue; in particular, many of the tiles shaded in blue are the main geographical features of the terrain being represented!

For reference, Figure 9.10 shows the heightmap that was used to generate Figure 9.11; areas of significant change (from dark to light or vice-versa) do not correlate directly with the shading in Figure 9.11. This is a good indication that the algorithm is currently naive in its distribution of detail.

The above image clearly demonstrates that a naive LOD algorithm like the one originally presented will neither generate high quality, nor high-performance results. The areas of the image that need the most detail are being given the least, and the areas with the most detail are being clipped by the rasterizer and not even shown.

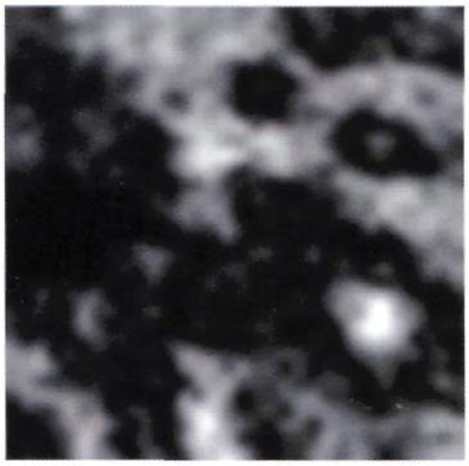


Figure 9.10. Original height map.

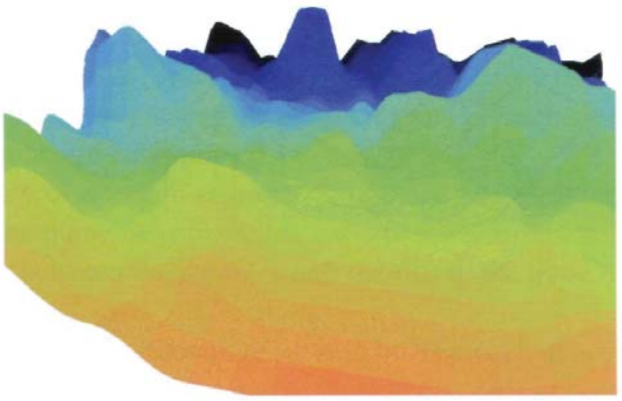


Figure 9.11. Debug rendering using original algorithm.

One simple change to the algorithm can help reduce this problem and also exercise another new feature in the Direct3D 11 pipeline—the compute shader.

The preceding diagrams show a case where a 256x256 heightmap is represented by a 16x16 grid. This means that for each patch rendered, the domain shader has a 16 x 16 grid of pixels to draw samples from. However, the hull shader, which is responsible for selecting the LOD, has no knowledge of this information. If it were able to analyze the appropriate grid of pixels later used by the domain shader, it could quite easily make a much more informed decision about the tessellation factors.

While a hull shader can read pixels from a texture bound to one of its samplers, it would be unnecessarily inefficient to have the constant function make 256 (16x16) texture samples on each invocation. However, if it had a lookup table that stored precomputed values, it could simply pull in the necessary values with a single sample operation.

Pre-Processing the Height Map

Although it is a pre-processing step in this context, the approach that will be taken is very similar to the concept of post-processing, which has been common in real-time graphics for several years.

The input heightmap will be divided up into kernels; each kernel area will have its pixels read in, and four values will be generated from the raw data, which can then be stored in a 2D output texture. This texture will then be indexed by the hull shader to enable it to have the previously described context when making LOD computations.

The key design decision is how to map the 16x16 height map samples down to a single per-patch output value.

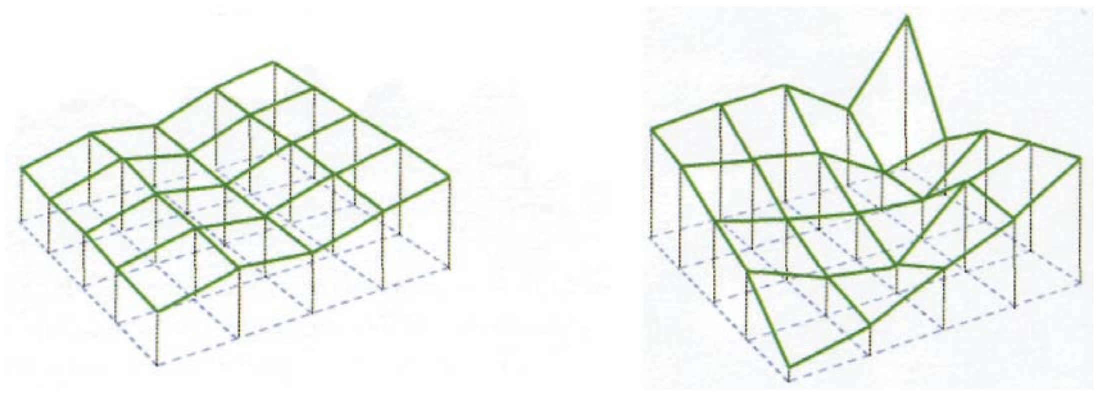


Figure 9.12. Examples of coplanarity.

It is relatively straightforward to compute a variety of statistics from the source data, but really, all that the hull shader cares about is having a measure of how much detail the patch requires. Is this piece of terrain flat? If yes, generate less detail. Alternatively, is this piece of terrain very bumpy and noisy? If yes, generate more detail.

A good objective for this pre-pass is to find a statistical measure *of coplanarity*—to what extent do the 256 samples lie on the same plane in 3D space?

This section covers a good solution to this question, one that maps cleanly to the GPU. However, there is scope for potentially higher-quality results using Fourier or Harmonic Analysis techniques familiar to mathematicians and physicists for deriving LOD metrics. This is left as an exercise to the more adventurous reader!

Consider Figure 9.12. The left-hand side shows a relatively uniform slope, possibly the side of a hill or valley. However, the right-hand side originates from a more complex section of terrain and is consequently much more erratic and noisy. Ideally, the hull shader would give the right-hand side a much higher level of detail, because, quite simply, that

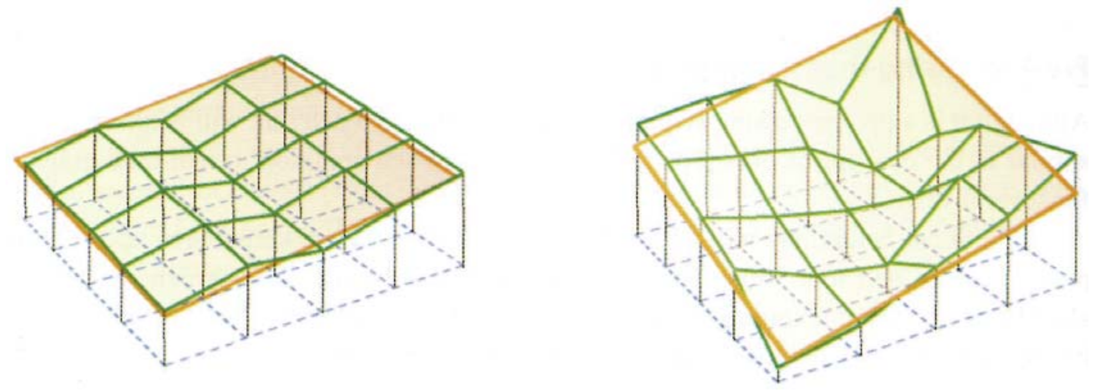


Figure 9.13. Planes fit over examples of different coplanarity.

side requires more triangles to accurately represent it. Consider Figure 9.13, which shows the same two examples as in Figure 9.12, but with a plane inserted into the dataset.

Although Direct3D cannot render quads natively,[4] the plane would be the best possible surface if there were no tessellation involved and only a single primitive were used to represent this piece of landscape. Notice that the plane in the left-hand side is a much closer match to the surface than in the right side of the figure.

Figure 9.14 shows a side-on view of a terrain segment, with plane and lines indicating how far each sample is from the plane. It is from this basis that we can measure coplanarity—the shorter the lines are between the samples and the plane, the more coplanar the data is.

Picking the plane to base these calculations off requires a "best fit" approach. It needs to be representative of the overall shape of the patch, yet it is unlikely that any plane generated will be a perfect match to the real data.

Figure 9.15 demonstrates one computationally efficient method of getting an acceptable "best fit" plane. On the left is the original patch geometry introduced earlier, and on the right is the same geometry, but with only the four corners joined together. Although this simplified primitive appears coplanar in this case, there is no guarantee that this will always be true.

For each of the 4 corners, that corner's 2 adjacent neighbors are also known, and from here it is trivial to generate the pairs of vectors denoted in red. The cross-product of each pair of vectors results in a normal vector for that corner, which is denoted in blue. Adding and

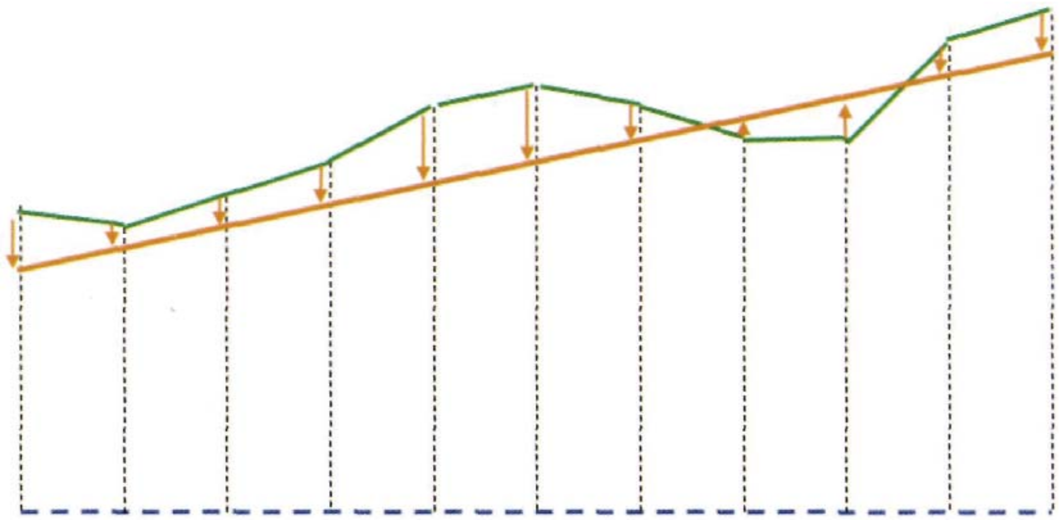


Figure 9.14. Side-on view of a terrain segment.

[4] It is true that geometry with four or more vertices can be sent into the pipeline, but these must be converted (by the tessellation stages) into triangles that are ultimately rasterized into the final image.

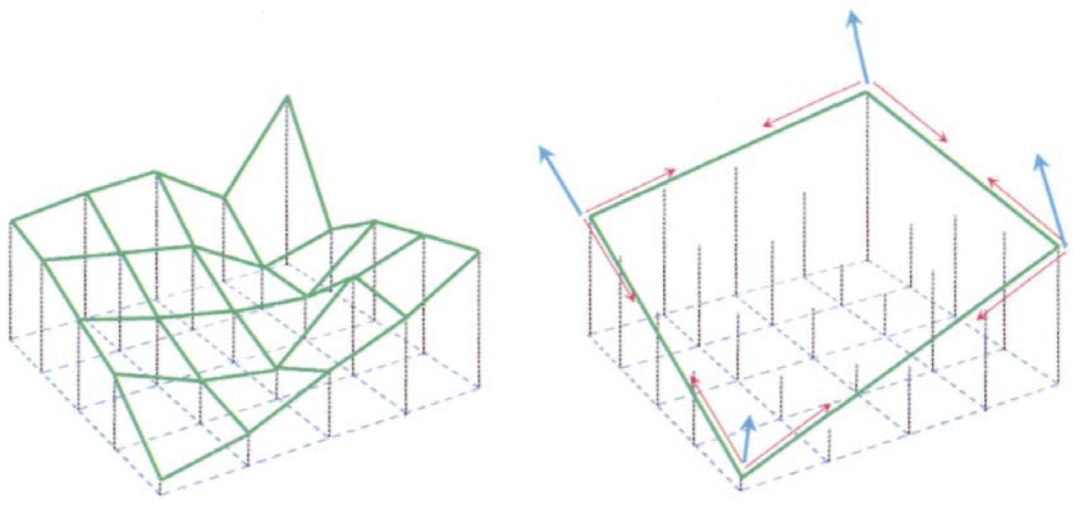


Figure 9.15. A "best fit" plane.

normalizing these 4 raw normal vectors results in a single unit-length normal vector for the patch, one that is generally representative of the underlying surface.[5] By taking any of the four corner positions, it is possible to derive a standard plane equation, as shown in Equation (9.1):

$$
\begin{aligned}
&Ax + By + Cz + D = 0,\\
&A = N ,\\
&B = N_y, \qquad (9.1)\\
&C = N_z ,\\
&D = -(N*P).
\end{aligned}
$$

With this plane equation known, the compute shader can evaluate each height map sample for the distance between it and the plane.

Implementing with a Compute Shader

Notation and indexes in the compute shader are not immediately obvious; Figure 9.16 introduces two of the key variables in the context of a terrain rendering pre-pass.[6] The core HLSL shader has an entry point, shown in Listing 9.10, with the [numthreads(x,*y,z)]* attribute attached to it.

[5] The process of calculating a normal vector is demonstrated in the vertex shader and rasterizer sections of Chapter 3.

[6] Compute shaders are discussed in detail in Chapter 5.

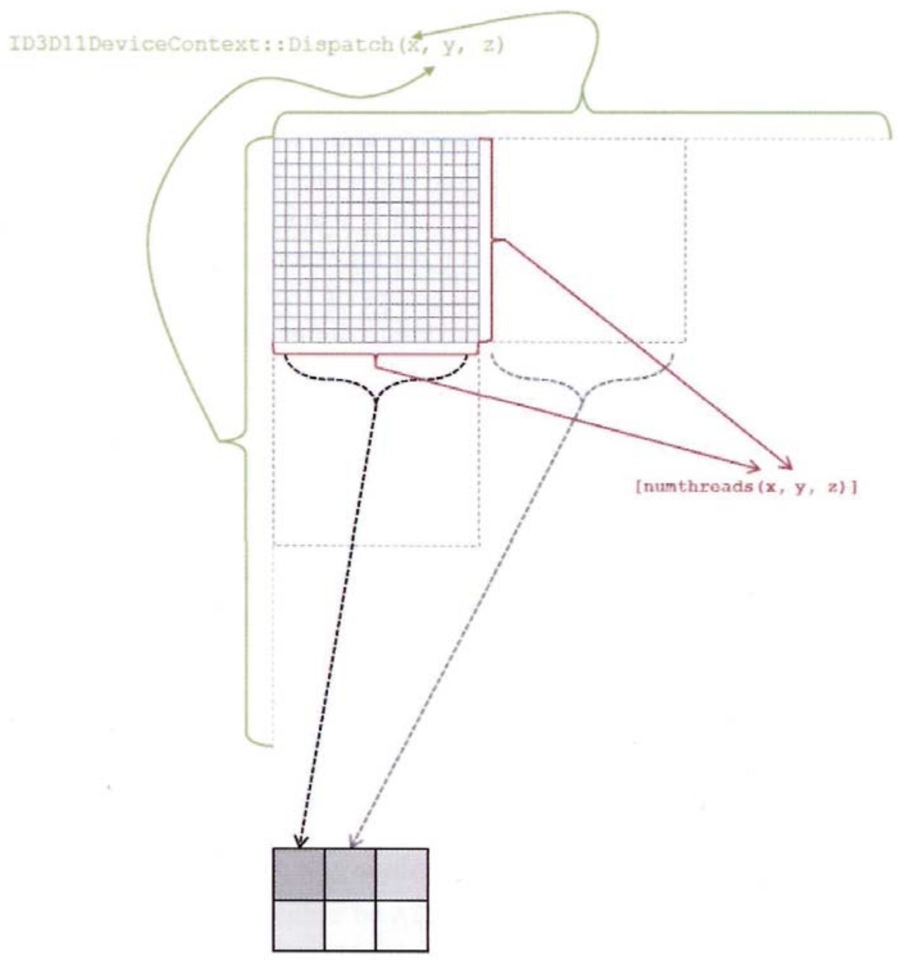


Figure 9.16. Compute shader parameters for a tile.

```
[numthreads(16, 16., 1)]
void csMain
      (
         uint3 Gid  : SV_GroupID
         , uint3 DTid  : SV_DispatchThreadID
         , uint3 GTid  : SV_GroupThreadID
         , uint GI  : SV_GroupIndex
      )
{
    /* Shader Code Here */
}
```

Listing 9.10. Compute shader entry point.

This attribute defines a thread group, also known as a *kernel.* In Listing 9.10 it is defining a 16x16x1[7] array of threads per group. The body of the csMain method is executed for a single thread, but through system generated values, it is able to identify which of these 256 (16x16x1) threads this actually is. Because it is know which thread this is, the code can be written to ensure that each thread reads from and writes to the correct location.

In Figure 9.16 the `Dispatch(x, y, z)` call is also introduced. This is made by the application and is analogous to a draw call as it begins execution of the compute shader. At this level, the parameters indicate how many groups of 16x16x1 thread groups to create. For this particular algorithm, the application simply divides the input height map texture dimensions by 16 and uses this as the number of kernels.[8]

For example, for a 1024x1024 height map, there will be 64x64 kernels, each kernel being 16x16x1 threads. Conceptually, this would imply a very large number of threads, one per pixel in this case, but it is up to the implementation how these tasks will be scheduled on the GPU and how many actually execute concurrently.

A key detail that has been omitted until now is how an invocation can identify itself relative to its group, as well as the entire dispatch call. Direct3D defines four system-generated values for this purpose:

1. SV_GroupID
   This `uint3` returns indexes into the parameters provided by ID3D11Device `Context::Dispatch()`. It allows this invocation to know which group this is, relative to all others being executed. In this algorithm, it is the index into the output texture where the results for the whole group are written.

2. SV_GroupThreadID
   This `uint3` returns indexes local to the current thread group—the parameters provided at compile-time as part of the `[numthreads()]` attribute. In this algorithm, it is used to know which threads represent corner pixels for the current 16x16 area.

3. SV_DispatchThreadID
   This `uint3` is a combination of the previous two. Whereas they index relative to only one set of input parameters (either `::Dispatch()` or `[numthreads()]`), this is a global index, essentially the two axes multiplied together. For a 64x64x1 dispatch of 16x16x1 threads, this system value will vary between 0 and 1023 in both axes (64*16=1024). Thus, for this algorithm, it provides the thread with the address of the source pixel to read from.

[7] The API expects *(X,Y,Z)* thread group notation, which is maintained in this text (and introduced in Chapter 5]—even if *Z*=1 is not actually significant to this implementation.

[8] Multiples of 16 were chosen for convenience in the sample code; this can be changed, as appropriate, for different-sized terrains.

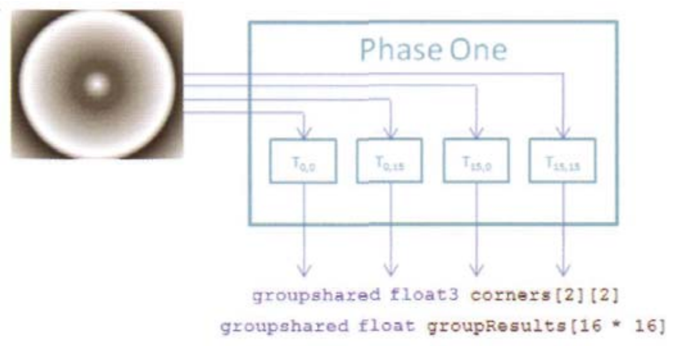
4. `SV_GroupIndex`
   This `uint` gives the flattened index into the current group. For a 16x16 area, this value will be between 0 and 255. For the purpose of this algorithm, it is essentially the thread ID, used only to coordinate work across the group.

The final piece in the puzzle is the ability for threads to communicate with each other. This is done through a 4-KB chunk of group shared memory, and synchronization intrinsics. Variables defined at the global scope, such as those shown in Listing 9.11 with the groupshared prefix, can be both read from and written to by all threads in the current group.

```
groupshared float             groupResults[16 * 16];
groupshared          float4   plane;
groupshared          float3   rawNormals[2][2];
groupshared float3            corners[2][2];
```

Listing 9.11. Compute shader state declarations.

Synchronization is done through a choice of six barrier functions. The code can be authored with either a *MemoryBarrier() or *MemoryBarrierWithGroupSync() call. The former blocks until memory operations have finished, but progress can continue before remaining ALU instructions complete. The latter blocks until all threads in the group have reached the specified point—both memory and arithmetic instructions must be complete. The barrier can either be All, Device, or Group—, with decreasing scope at each level. Thus, an AllMemoryBarrierWithGroupSync() is the heaviest intrinsic to employ, whereas GroupMemoryBarrier() is more lightweight. In this algorithm, only GroupMemoryBarrierWithGroupSync() is used. Figure 9.17 shows the first phase of the algorithm.



```
groupshared float3 corners[2][2]
groupshared float groupResults[16 * 16]
```

Figure 9.17. Compute shader phase one.

Notice that the first phase uses four threads, one for each corner of the 16x16 pixel group. Each of the four threads reads in a single sample. It then stores the height in groupResults[] and stores a 3D position in corners[][]. All other threads are idle at this point. Listing 9.12 shows the code for this.

```
if(
    ((GTid.x ==  0) && (GTid.y ==  0))
     ||
    ((GTid.x == 15) && (GTid.y ==  0))
     ||
    ((GTid.x ==  0) && (GTid.y == 15))
     ||
    ((GTid.x == 15) && (GTid.y == 15))
  )
{
    // This is a corner thread, so we want it to load
    // its value first
    groupResults[GI] = texHeightMap.Load( uint3( DTid.xy, 0 ) ).r;
    corners[GTid.x / 15][GTid.y / 15]
        = float3(GTid.x / 15, groupResults[GI], GTid.y / 15);
    // The above will unfairly bias based on the height ranges
    corners[GTid.x / 15][GTid.y / 15].x /= 64.0f;
    corners[GTid.x / 15][GTId.y / 15].z /= 64.0f;
}

// Block until all threads have finished reading
GroupMemoryBarrierWithGroupSync();
```

Listing 9.12. Phase one of the compute shader.

Figure 9.18 depicts the next phase, where the same four threads continue to process the corner points. In this instance, they need to know about their neighboring corners, so that they can generate the cross-product and hence a normal vector for each corner—entirely ALU work. Concurrently, the other 252 threads can be reading in the remaining height map samples. Listing 9.13 shows this in action.

```
if((GTid.x ==  0) && (GTid.y ==  0))
{
    rawNormals[0][0] = normalize(cross
                            (
                                corners[0][1] - corners[0][0],
                                corners[1][0] - corners[0][0]
                            ));
}
else if((GTid.x ==  15) && (GTid.y ==  0))
```

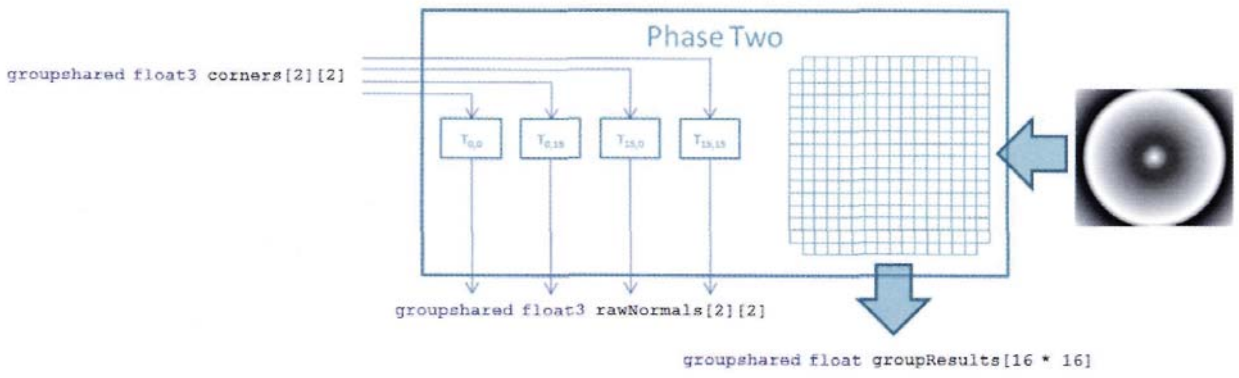

Phase Two
groupshared float3 corners[2][2]
T0,0
T0,15
T15,0
T15,15
groupshared float3 rawNormals[2][2]
groupshared float groupResults[16 * 16]

Figure 9.18. Compute shader phase two.

```
  {
      rawNormals[1][0] = normalize(cross
                                   (
                                       corners[0][0] - corners[1][0],
                                       corners[1][1] - corners[1][0]
                                   ));
  }
  else if((GTid.x == 0) && (GTid.y == 15))
  {
      rawNormals[0][1] = normalize(cross
                                   (
                                       corners[1][1] - corners[0][1],
                                       corners[0][0] - corners[0][1]
                                   ));
  }
  else if((GTid.x == 15) && (GTid.y == 15))
  {
      rawNormals[1][1] = normalize(cross
                                   (
                                       corners[1][0] - corners[1][1],
                                       corners[0][1] - corners[1][1]
                                   ));
  }
else
  {
      // This is just one of the other threads, so let it
      // load in its sample into shared memory
      groupResults[GI] = texHeightMap.Load( uint3( DTid.xy, 0 ) ).r;
  }

  // Block until all the data is ready
  GroupMemoryBarrierWithGroupSync();
```

Listing 9.13. Phase two of the compute shader.

Phase four, shown in Figure 9.20, is where the next big chunk of work takes place, but before this, the group must have a plane from which to measure offsets, as shown in Figure 9.19. This only requires a single thread and simply implements the plane-from-point-and-normal equations, as shown in Listing 9.14.

```
// The following fragment of the compute shader uses
// this utility function:
float4 CreatePlaneFromPointAndNormal(float3 n, float3 p)
{
    return float4(n, (-n.x*p.x - n.y*p.y - n.z*p.z));
}

// Phase 3 of the CS starts here:
if(GI == 0)
{
    // Let the first thread only determine the plane coefficients

    // First, decide on the average normal vector
    float3 n = normalize
                    (
                      rawNormals[0][8]
                      + rawNormals[0][1]
                      + rawNormals[1][0]
                      + rawNormals[1][1]
                    );

    // Second, decide the lowest point on which to base it
    float3 p = float3(0.0f,le9f,0.0f);
    for(int i = 8; i < 2; ++i)
        for(int j = 0; j < 2; ++j)
            if(corners[i][j].y < p.y)
                p = corners[i][j];

    // Third, derive the plane from point+normal
    plane = CreatePlaneFromPointAndNormal(n,p);
}

GroupMemoryBarrierWithGroupSync();
```

Listing 9.14. Phase three of the compute shader.

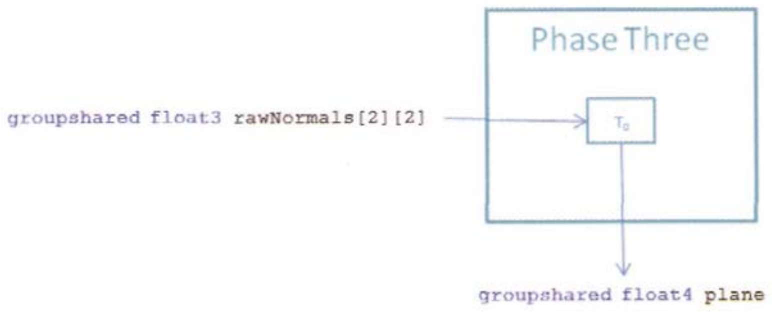


Figure 9.19. Compute shader phase three.

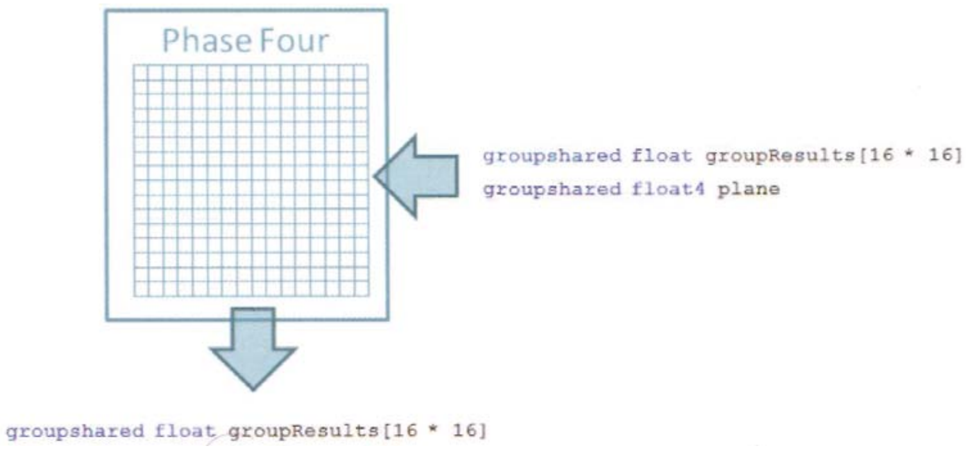

Phase Four
groupshared float groupResults[16 * 16]
groupshared float4 plane
groupshared float groupResults[16 * 16]

Figure 9.20. Compute shader phase four.

As mentioned earlier, Figure 9.20 shows phase four. With a plane available, it is necessary to process each of the raw heights as originally loaded from the height map. Each thread takes a single height. It then computes the distance between this sample and the previously-computed plane, replaces the original raw height value.

```
// Following fragment of CS relies on this
// utility function:
float ComputeDistanceFromPlane(float4 plane, float3 position)
{
    return dot(plane.xyz,position) - plane.w;
}

// Begin phase 5 of the compute shader:
groupResults[GI] = ComputeDistanceFromPlane
                    (
                      plane
                      , float3
                        (
                            (float)GTid.x / 15.0f
                           , groupResults[GI]
                           , (float)GTid.y / 15.0f
                        )
                    );

GroupMemoryBarrierWithGroupSync();
```

Listing 9.15. Phase four of the compute shader.

Figure 9.21 shows the final phase of the algorithm. This phase takes all of the height values and computes the standard deviation from the surface of the plane. This single value

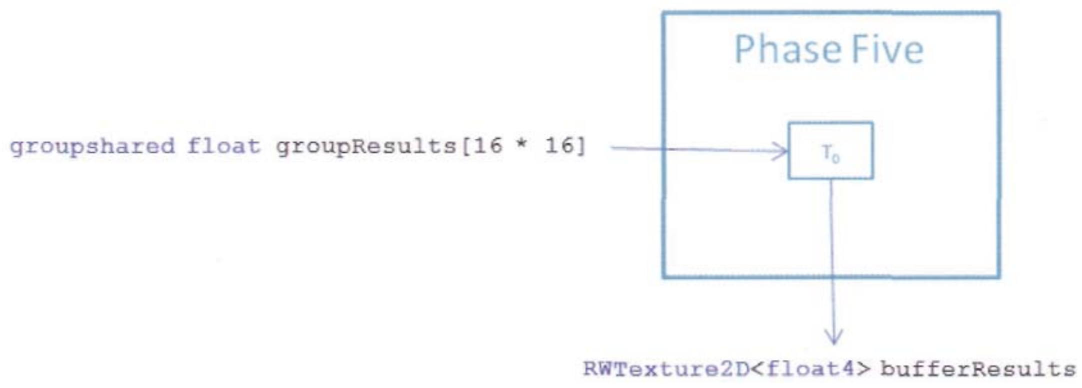

Phase Five
groupshared float groupResults[16 * 16]
T_0
RWTexture2D<float4> bufferResults

Figure 9.21. Computer shader phase five.

is a good metric of how coplanar the 256 individual height samples are—lower values imply a flatter surface, and higher values imply a noisier and more varying patch. This single value and the plane's normal vector are written out as a float4 in the output texture, so 256 height map samples have been reduced to four numbers. Listing 9.16 shows the last phase of the computer shader.

```
if(GI == 0)
{
    // Let the first thread compute the standard deviation for
    // this patch. The 'average' is really just going to be 0.0
    // as we want the deviation from the plane and any point on the
    // plane now has a 'height' of zero.
    float stddev = 0.0f;
        for(int i = 0; i < 16*16; ++i)
        stddev += pow(groupResults[i],2);
    stddev /= ((16.0f * 16.0f) - 1.0f);

    stddev = sqrt(stddev);

    // Then write the normal vector and standard deviation
    // to the output buffer for use by the Domain and Hull Shaders
    bufferResults[uint2(Gid.x, Gid.y)] = float4(plane.xyz, stddev);
}
```

Listing 9.16. Phase five of the compute shader.

Integrating the Compute Shader

The previous section details the actual compute shader that implements our algorithm, but the application must still coordinate this work.

First, the output texture needs to be created. This will be bound as an output to the compute shader, but used later as an input to the hull shader. The underlying resource type is a regular 2D texture, with the important detail of having a D3D11_BIND_UN0RDERED_ACCESS as one of its bind flags, as shown in Listing 9.17.

```
// Assert on the input texture dimensions
D3D11_TEXTURE2D_DESC d = m_pHeightHapTexture->m_pTexture2dConfig->GetTextureDesc();
_ASSERT( 0 == (d.Width % 16) );
_ASSERT( 0 == (d.Height % 16) );

// Create the output texture
Texture2dConfigDXIl LookupTextureConfig;
LookupTextureConfig.SetFormat( DXGI_F0RMAT_R32G32B32A32_FL0AT );
LookupTextureConfig.SetColorBuffer( TERRAIN_X_LEN, TERRAIN_Z_LEN );
LookupTextureConfig.SetBindFlags( D3D11_BIND_UN0RDERED_ACCESS
                                  | D3D11_BIND_SHADER_RES0URCE );

m_pLodLookupTexture = m_pRendererll->CreateTexture2D( SLookupTextureConfig, 0 );

// Create the effect
SAFE_DELETE( m_pComputeShaderEffect );
m_pComputeShaderEffect = new RenderEffectDXIl( );

// Compile the compute shader
m_pComputeShaderEffect->m_iComputeShader =
        m_pRendererll->LoadShader( COMPUTE_SHADER,
        std::wstring( L"../Data/Shaders/InterlockingTerrainTilesComputeShader.
                      hlsl" ),
        std::wstring( L"csMain" ),
        std::wstring( L"cs_5_0" ) )j
_ASSERT( -1 != m_pComputeShaderEffect->m_iComputeShader );
```

Listing 9.17. Creating the compute shader.

At this point, the necessary resources have been created, so they simply need to be bound to the pipeline. Also, the compute shader must be initiated, as shown in Listing 9.18.

```
// Bind the resources
m_pRendererll->m_pParamMgr->SetUnorderedAccessParameter
                                   ( L"bufferResults"j m_pLodLookupTexture );
m_pRendererll->m_pParamMgr->SetShaderResourceParameter
                                   ( L"texHeightMap", m_pHeightMapTexture );

// Determine number of threads
D3D11_TEXTURE2D_DESC d = m_pHeightMapTexture->m_pTexture2dConfig->GetTextureDesc();

// Run the compute shader
m_pRendererll->pImmPipeline->Dispatch
```

```
                                    (
                                       *m_pComputeShaderEffect
                                       , d.Width / 16
                                       , d.Height / 16
                                       , 1
                                       , m_pRenderer11->m_pParamMgr
                                  );
// Bind the output to the hull shader
m_pRenderer11->m_pParamMgr->SetShaderResourceParameter
                                ( L"texLODLookup", m_pLodLookupTexture );
```

Listing 9.18. Executing the compute shader.

The code after the Dispatch() call in Listing 9.18 is particularly important. Without this being executed, the *UAV* will still be bound to the pipeline referencing the 2D output texture; Direct3D will then keep it from also being bound as an input to the hull shader, since it is illegal to have a resource set to be both an input and output at the same time!

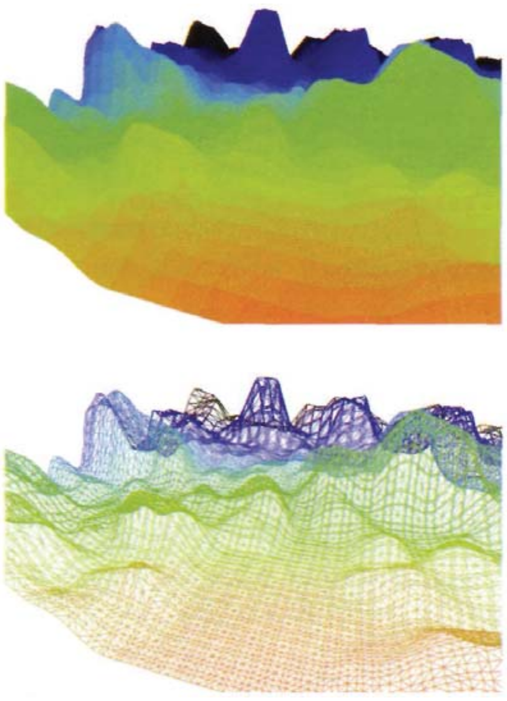


Figure 9.22. Naive Distance Based LOD 32,500 Triangles generated, 76% rasterized.

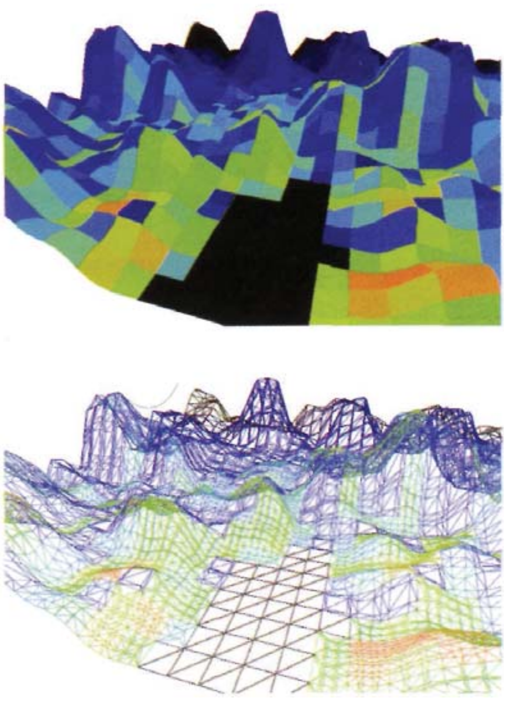


Figure 9.23. Compute Shader Based LOD 22,232 triangles generated, 77% rasterized.

Results

Figure 9.22 and Figure 9.23 demonstrate the difference made by the new algorithm.

Although Figure 9.22 may appear more aesthetically pleasing due to the smooth gradients, the more chaotically shaded Figure 9.23 is by far the better image, from a geometric perspective. In both images, the patch detail is translated into a color—red for high detail, green for mid detail, and blue for low detail. Black is the lowest detail.

The lower half of each image is the corresponding wireframe representation, which clearly and obviously shows the approximately 40% reduction in triangle count.

Consider the bottom center area of both image. In Figure 9.22 it is mostly shaded in orange, whereas in Figure 9.22, it is predominantly black. It is also important to notice that the surface being represented is flat for this section of terrain. The naive distance-based calculation assigns this geometrically simple piece of terrain a high number of triangles, although it simply doesn't need it. Why waste processing power and memory bandwidth generating and rasterizing extra triangles that add nothing to the final image?

Examining Figures 9.22 and 9.23 initially suggests that the computer shader approach has greatly reduced the LOD across the whole image. This is based on the observation that most of the terrain tiles are shades of blue, while the majority are green and yellow in the na'ive implementation. Figure 9.24 makes it easy to compare the naive approach (on the left) and the computer shader approach (on the right).

When rendering is performed using a simple N*L directional lighting shader with solid shading, we get an image much closer to what would be rendered in a real application. The only major omission is the lack of textures (such as grass and rock). In addition to naive and computer shader approaches shown, the middle of Figure 9.24 shows a simple image-based difference of the two results on either side. Here, black means no difference, and white means that they are completely different. Comparing the rasterized image is preferable to comparing the underlying geometry because ultimately, it is the

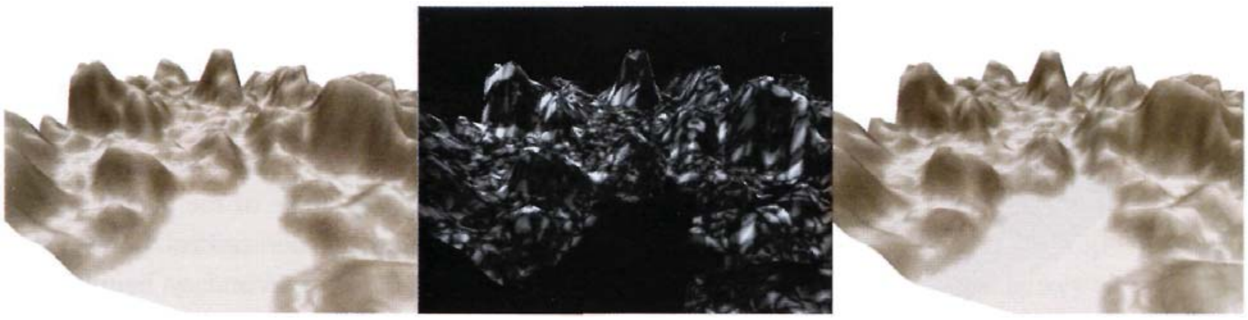


Figure 9.24. Comparison of naive and compute shader approaches.

image displayed on the screen that is most important, and geometry is just a means by which to construct it.

On close inspection it isn't too hard to see the differences between the left and right images in Figure 9.24. However, the differences are actually quite minimal. Apart from the silhouette (discussed next) the difference is very rarely above 15%, which, for a reduction in geometry of 40%, is a good tradeoff. The exact weightings and biases can be easily tweaked in the appropriate HLSL code to achieve an aesthetically pleasing result.

Further Extensions

By weighting the level of detail by the actual complexity of the landscape, the available processing time can be more accurately shifted to areas that genuinely benefit from it. Despite being a worthwhile improvement over the original, there are still further extensions that could be implemented.

The discussion so far has calculated LODs as a pre-pass before any rendering. However, there is no reason that dynamic terrain (where the shape changes on a regular basis) can't be used, given that a simple rerun of the compute shader will update all the necessary inputs for the actual rendering.

One particular limitation of this implementation is that it still retains an element of distance in the LOD calculation. Geometry further from the camera is always assumed to be less important, and to require less detail.

Consider the horizon shown in Figure 9.25. By definition, a horizon is a long way from the viewer; in the current implementation, that works to reduce the final level of detail, even though the horizon is a significant part of the image as a whole.

It is worth extending the terrain algorithm to weight features on the horizon higher than distant objects that are less pronounced. The silhouette of a model is one of the biggest visual clues of its true level of detail, and is one of the more obvious places for the human eye to pick up on approximations or other visual artifacts.

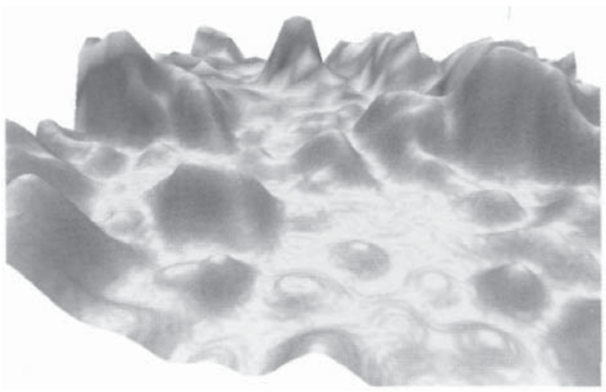


Figure 9.25. Example rendering showing a low-detail horizon.

Another possibility is to use a secondary height-map. We could use the approach already discussed to get the basic shape of the landscape for the hull shader control points, and could use a secondary height map to add further detail to the vertices generated by the domain shader. This has the neat potential of allowing overhangs or caves in the terrain, a feature that is typically missing from most height-map-based terrain Tenderers.

One significant problem that hasn't been mentioned thus far, and which is not really in the scope of this chapter, is that of interacting with the terrain.

In particular, the core application running on the CPU has no knowledge of the final geometry, which is generated entirely on the GPU. This is problematic for physics-oriented or user-oriented algorithms, such as picking (clicking to select geometric objects and features) or having objects interact (as in collision detection) or having objects track the landscape (such as having a car follow the terrain of a racing track).

Two ways to mitigate this problem are for the CPU to always use the highest level of detail, or to move picking or ray-casting to the GPU. In the former case, and for an adaptive GPU tessellation like the one described here, the rendered terrain should be very close to the highest LOD in all cases where it would be evident to a user. In the latter case, it is possible to use the geometry shader, combined stream output,[9] to output a candidate set of geometry that intersect points of interest. This is quite complicated and requires careful work to ensure that CPU read-back doesn't stall the GPU; but does present a robust solution.

9.2 Higher-Order Surfaces

This category of tessellation is the more commonly known approach. It covers the algorithms and equations used by the majority of the art and design software used over the last few decades, and is being closer to the assumption that tessellation is about *curved* or *smooth* surfaces.

Consider Equation (9.2), shown below. In particular, note that it has two basic constituents—constants (a, *b, c,* and *d)* and variables (only *x* in this case). In the context of tessellation, the constants would be control points output by the hull shader stage, and the variables would be the sample locations input into the domain shader stage from the fixed-function tessellator. The domain shader is responsible for taking both of these and evaluating the final result of the equation:

$$f(x) = a + bx + cx^2 + dx^3 \tag{9.2}$$

The exact form of an equation used in this style of tessellation will vary, and the properties (such as complexity or classes of shape or the surface it can replicate) will be key to

[9] Geometry shaders and stream output capabilities are discussed in detail in Chapter 3.

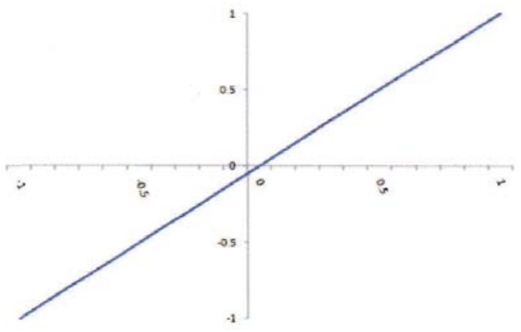


Figure 9.26. Linear.

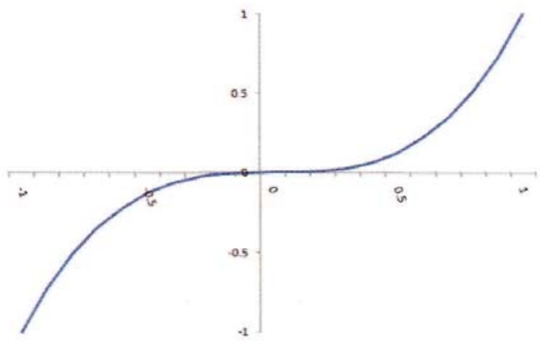


Figure 9.27. Quadratic.

evaluating the usefulness of it in any tessellation context. This can be seen in Figures 9.26 through Figure 9.28.

In most cases, the mathematical functions employed for higher-order surfaces will be quadratic (Figure 9.27) or cubic (Figure 9.28); Figure 9.26 is included for reference with regards to planar triangle/line segments used in conventional rendering.

Individual quadratic or cubic sections are limited in the number of shapes they can represent, such that any real-world application of higher-order surfaces will want to compose many of these smaller curves into a larger model representing far more complex surfaces. How these segments are joined together has strongly impacts the overall appearance of the surface and is described as *geometric continuity.* The following figures introduce four levels of continuity.

Initially, G_2 continuity appears to be the most desirable. While aesthetically it does result in "perfect" curves, some characteristics of the other continuity levels can be beneficial. The primary motivation for non-G_2 continuity is that of sharp edges, which are typically required for non-organic or otherwise artificial objects. For example, aspects of a

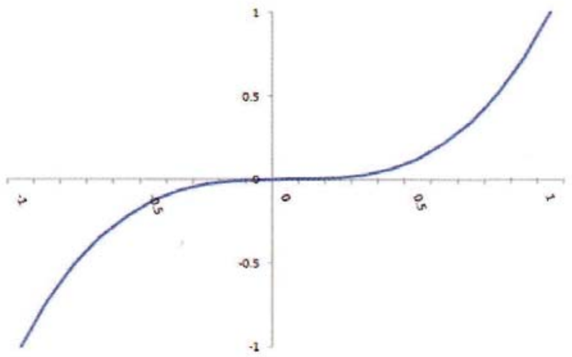


Figure 9.28. Cubic.

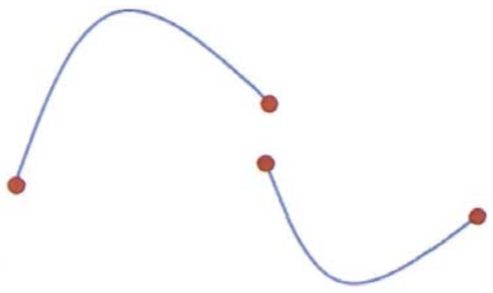


Figure 9.29. No continuity.

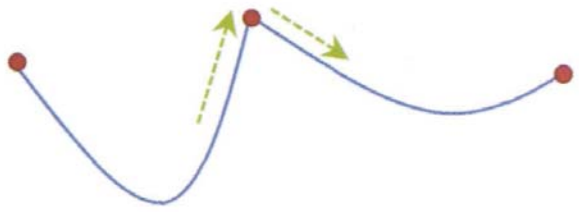


Figure 9.30. G_0 - start and end points match.

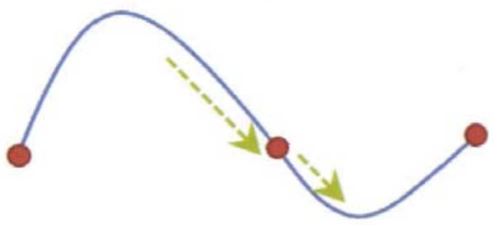


Figure 9.31. G_1 - incoming and outgoing tangents are parallel.

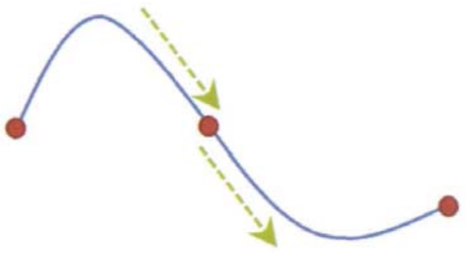


Figure 9.32. G_2 - Incoming and outgoing tangents are parallel and rate of change is same.

boat might be smooth and curved, but a boat still has a lot of edge detail that is not smooth or curved.

Utilizing G_0 and G_1 continuity can allow for sharper and more precise geometric shapes, but it is worth noting that this may require adding additional geometry around key details to ensure an accurate representation. Figures 9.34 and 9.35 demonstrate this particular case.

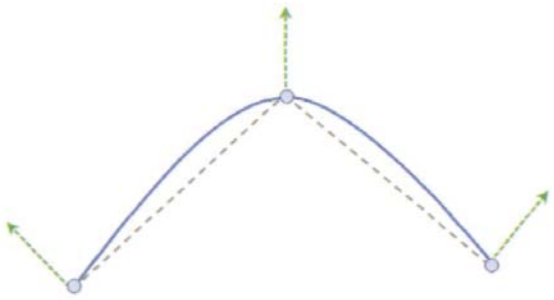


Figure 9.33. Without extra geometry.

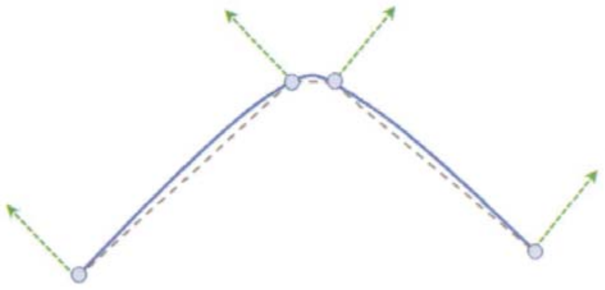


Figure 9.34. With extra geometry.

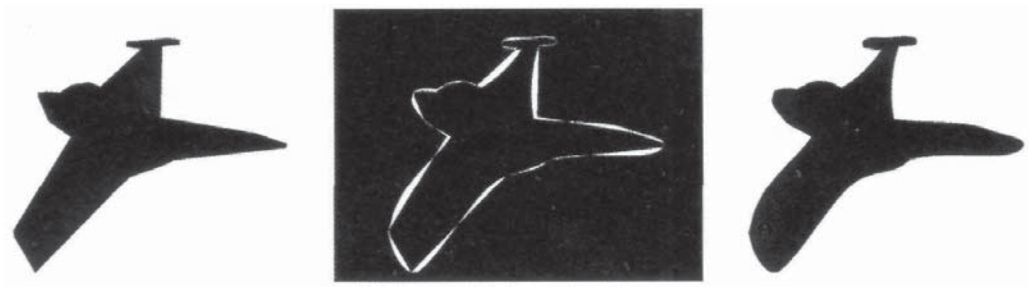


Figure 9.35. Example of silhouette enhancement.

9.2.1 Curved Point Normal Triangles

Chapter 4 introduced the history of tessellation in modern computer graphics. In particular, it mentioned the TruForm feature in ATI Technology's Radeon 8500 GPU and the fact that this was an early attempt at consumer hardware tessellation. Despite its various merits, the technology never attained critical mass at the time, due in part to fierce competition, as well as to software advances.

The obvious benefit of this algorithm both, originally and now, is the improved visual aesthetics. However, it isn't just the continuity and smoothness of the surface itself that is significant. It is also worth noting the silhouette. Take Figure 9.35 as a demonstration of the silhouette enhancements; the key is that the right side does not present an obviously artificial outline comprised of triangle edges. Advances in recent years with per-pixel lighting and bump mapping can greatly improve the perceived quality of a model's interior, but they still leave a reminder that we are merely seeing a handful of triangles, something the human brain is prone to pick up on and smash the illusion.

Although the Direct3D 8 API didn't explicitly expose details about the algorithm it used, it did resolve down to what was discussed in the "Curved PN Triangles" paper (Vlachos, Peters, Boyd, & Mitchell, 2001) which ATI Technologies released with its own hardware implementation. Direct3D 11 removes the main barriers to the acceptance of Vlachos et. al's Curved PN Triangles algorithm from first time around, and the remainder of this chapter will discuss an implementation of the original approach, along with several potential improvements.

Algorithm Overview

A key advantage of the algorithm is that it requires no additional data to be provided by the application. This was crucial for commercial success and ease of use when applied to consumer hardware of the era (even though that success didn't happen then!). The intention was that developers could flip a switch in the form of an API render state, and the hardware would transparently apply the algorithm, producing smooth surfaces without any substantial code changes. It also didn't require different geometry to be created by artists. This would have been prohibitively difficult, due to targeting multiple types of hardware, and to not suiting the commonly used tools of the period.

The mathematics of the algorithm therefore only require a conventional triangle comprised of three positions and three normal vectors. Even with Direct3D 11, where adjacency (up to 32 control points per primitive) is relatively easy to provide, this simple property can be quite convenient. Most importantly, allows for an "upgrade" path of sorts—existing art assets can be rendered via Direct3D 11 using this algorithm without any changes, while developers still have the option to use more complex modeling formats and algorithms when the resources support it.

Curved Point Normal Triangles also neatly map on to the Direct3D 11 tessellation pipeline. The first stage is to take the incoming triangle and generate a more detailed *control mesh*—a hull shader program that amplifies. After sample locations are generated (by the fixed-function tessellation unit) the mathematical surface is evaluated using the detailed control mesh—a classic Direct3D 11 Domain Shader program.

Geometric Components

Tessellation of the geometry that actually makes up the final rasterized surface is done on a cubic basis, requiring an additional seven control points.

Looking back at Figure 9.27 and Figure 9.28, it becomes clear that a quadratic function cannot represent a full range of surface variations. However, a cubic function can capture surface inflections, resulting in a more accurate and aesthetically pleasing representation.

Figure 9.36 shows the locations of the additional seven control points (green) as well as the original three vertex positions (blue), as provided by the application. These seven control points, two per side and one in the middle, are evenly distributed across the surface of the triangle.

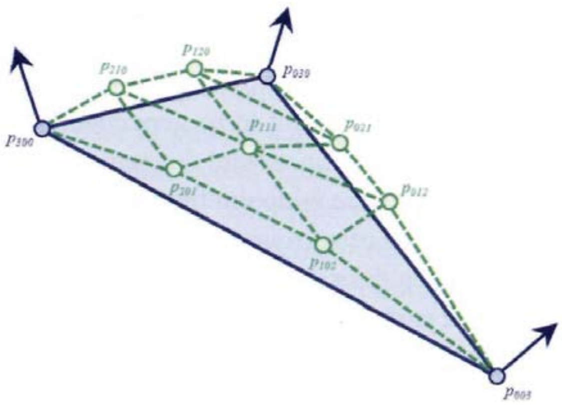


Figure 9.36. Additional control points.

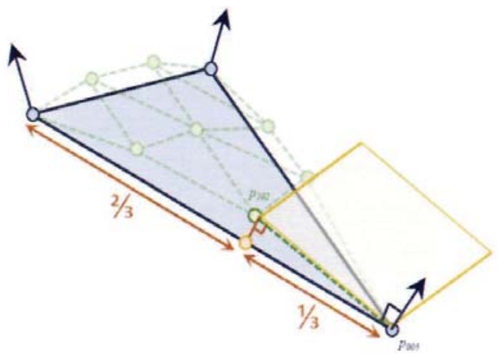


Figure 9.37. Control point equation shown visually.

The six control points along the edge are computed using the tangent plane defined by each control point's nearest vertex. Identifying the location of the edge mid□point is a trivial linear interpolation along the edge based on the corresponding vertices, as shown in Equation (9.3). Defining the tangent plane using the vertex and its normal, Equation (9.4), is a stock formula used throughout computer graphics. Combining these two elements into Equation (9.5) will generate the appropriate control point. Figure 9.37 shows how these three equations work together to generate the control points introduced in Figure 9.36:

$$Q=p_i + \tau\{p_j - p_i); \quad (93)$$

$$\begin{aligned} P &= (P_x, P_y, P_z), \\ N &= (N, N, N), \\ 0 &= xN_x + yN_y + zN_z + (-N*P); \end{aligned} \quad (9.4)$$

$$Q' = Q - N\times((Q-P)*N). \quad (9.5)$$

This still leaves the center control point to be generated. Computing the six control points along the edges actually forms a ring around this remaining location, a property that can be used to our advantage. Raising the midpoint from its position on the original triangle (Equation (9.6)) relative to this ring of surrounding control points (Equation (9.7)) using simple linear interpolation yields a desirable result. Choosing the halfway point between these keeps the final tessellated surface close to the original triangle geometry, leading to more predictable outputs with respect to their original coarse triangulations. Equation (9.8) shows the final position for this control point:

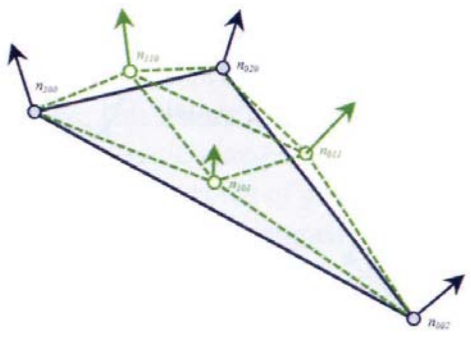


Figure 9.38. Additional normal vector control points.

$$V = \frac{P_{300} + P_{030} + P_{003}}{3}, \tag{9.6}$$

$$E = \frac{P_{120} + P_{210} + P_{021} + P_{012} + P_{201} + P_{102}}{6}, \tag{9.7}$$

$$P_{111} = E + \frac{E - V}{2}. \tag{9.8}$$

Normal Vectors

Tessellation of normal vectors is done on a quadratic basis, requiring an additional four control points. Initially, this seems unexpected, because it differs from the positional tessellation, which is cubic.

The original research paper (Vlachos, Peters, Boyd, & Mitchell, 2001) makes a particular case for this decision, based on simplicity. The added complexity of calculations to generate matching cubic normals for a generalized case is prohibitively difficult without producing an equivalent improvement in final image quality. While modern GPUs are orders of magnitude faster than those that introduced TruForm, this argument is still valid—why waste valuable processing time for no good reason?

Figure 9.38 shows the additional control points (green) required for quadratic normal interpolation: it is simply a case of putting a midpoint on each edge of the triangle, along with the original per-vertex normal vectors (blue) provided by the application.

To generate the control vector for the midpoint of each edge, we use reflection about a plane perpendicular to the edge currently being processed. This is necessary to allow the quadratic basis to approximate the geometric inflections that are possible with the cubic position tessellation.

Consider Figure 9.39, which shows the midpoint normal vector (orange) as a simple average of the vector at either end of the edge (dark green). Figure 9.40 shows the same

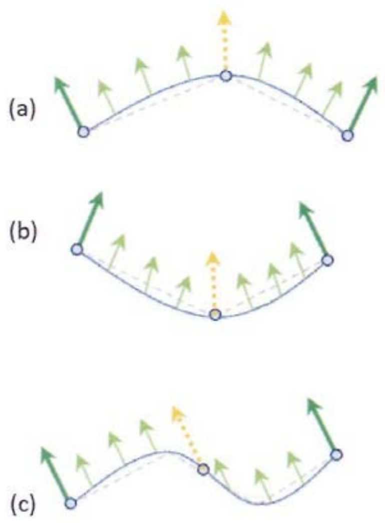


Figure 9.39. Simple averaging.

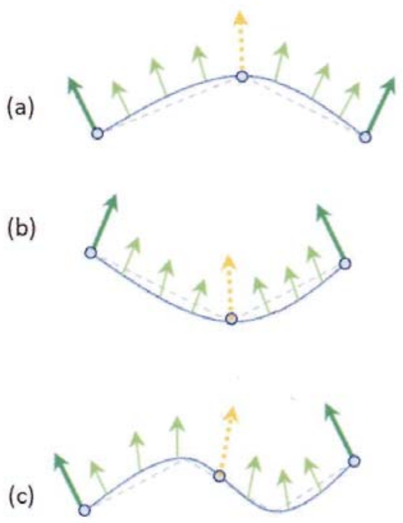


Figure 9.40. Averaging reflected.

calculation, but reflected about a plane perpendicular to the edge. Both figures show the three primary cases for normal vectors—(a) both pointing outwards, (b) both pointing inwards and (c) both pointing in the same direction. In all cases, it is assumed that the geometric surface is subject to a cubic tessellation function.

Equation (9.9) demonstrates how to generate a reflected normal vector in the general case (where *n* is the vector to reflect, and *b* is the unnormalized vector for the plane), while Equation (9.10) applies this to the context of quadratic normal vector interpolation:

$$n' = n - 2\times\left(\frac{b\cdot n}{b\cdot b}\right)\times b; \tag{9.9}$$

$$\begin{aligned}
N_1 &= n_{200},\\
N_2 &= n_{020},\\
N_3 &= n_{002},\\
P_1 &= p_{300},\\
P_2 &= p_{030},\\
P_3 &= p_{003},\\
v_{ij} &= 2\times\frac{(P_j - P_i)\cdot(N_i - N_j)}{(P_j - P_i)\cdot(P_j - P_i)},\\
n_{110} &= N_1 + N_2 - v_{12}\times(P_2 - P_1),\\
n_{011} &= N_2 + N_3 - v_{23}\times(P_3 - P_2),\\
n_{101} &= N_3 + N_1 - v_{31}\times(P_1 - P_3).
\end{aligned} \tag{9.10}$$

Generating Normals and Positions in the Hull Shader

With the equations for calculating the control mesh understood, it is now necessary to fit these into the Direct3D pipeline. This naturally aligns with the hull shader's responsibilities. It also demonstrates a characteristic introduced as part of Chapter 4, in that the hull shader stage can amplify or reduce the amount of geometry passed down the pipeline—in this case it is necessary to amplify. From the original three vertices, the hull shader must output ten positions and six normals.

There is also an interesting design decision to be made when implementing this part of the algorithm with regard to whether the control mesh is generated by the constant or by the per-element function. Logically, it fits in the per-element definition, but this results in code with more branches and conditionals (to determine which equation to apply for a given input value of SV_OutputControlPointID), which may not be as efficient to execute on a GPU. And not only may it be less efficient to execute a shader program with branches, but this branched code will be executed many times, once for each output control point. To avoid branching, it becomes necessary to place the code in the constant function, which ensures only a single evaluation.

However, it is important to note that this choice only exists for tessellation algorithms with small amounts of output data from the hull shader. The constant function is limited to 128 scalars of output (32 float4s), which must include the tessellation factors. Using Curved Point Normal Triangles requires at least 48 scalars of output (10 positions and 6 normals, each float 3 in size) before considering additional properties such as texture coordinates, color, or tangent vectors.

Listing 9.19 shows how to implement this using the logical approach of evaluating each output control point with branching. Readers can refer to the PNTriangles11 (AMD & Microsoft) sample in the DirectX SDK for an implementation that uses the constant function approach.

```
cbuffer TessellationParameters
{
    float4 EdgeFactors;
};

struct VS_OUTPUT
{
    float3 position        : WORLD_SPACE_CONTROL_POINT_POSITION;
    float3 normal          : WORLD_SPACE_CONTROL_POINT_NORMAL;
};

struct HS_OUTPUT
{
    float3 position        : WORLD_SPACE_CONTROL_POINT_POSITION;
    float3 normal          : WORLD_SPACE_CONTROL_POINT_NORMAL;
};
Struct HS_CONSTANT_DATA_OUTPUT
```

```
{
    float  Edges[3]        : SV_TessFactor;
    float Inside           : SV_InsideTessFactor;
};
HS__CONSTANT_DATA_OUTPUT hsConstantFunc( InputPatch<VS_OUTPUT, 3> ip, uint
PatchID : SV_PrimitiveID )
{
    HS_CONSTANT_DATA_OUTPUT output;

    output.Edges[0] = EdgeFactors.x;
    output.Edges[1] = EdgeFactors.y;
    output.Edges[2] = EdgeFactors.z;

    output.Inside = EdgeFactors.w;

    return output;
}

[domain("tri")]
[partitioning("fractional_even")]
[outputtopology("triangle_cw")]
[outputcontrolpoints(13)]
[patchconstantfunc("hsConstantFunc")]
HS_OUTPUT hsDefault( InputPatch<VS_OUTPUT., 3> ip, uint i : SV_
OutputControlPointID, uint PatchID : SV_PrimitiveID )
{
    HS_OUTPUT output;

    // Must provide a default definition just in
    // case we don't match any branch below
    output.position = float3(0.0f, 0.0f, 0.0f);
    output.normal  = float3(0.0f, 0.0f, 0.0f);
        switch(i)
    {
        // Three actual vertices:

        // b(306)
        case 0:
        // b(030)
        case 1:
        // b(003)
        case 2:

            output.position = ip[i].position;
            output.normal = ip[i].normal;
            break;

        // Edge between v0 and v1

        // b(210)
        case 3:
            output.position = ComputeEdgePosition(ip, 0, 1);
            break;
```

```
// b(120)
case 4:
    output.position = ComputeEdgePosition(ip, 1, 0);
    break;

// Edge between v1 and v2

// b(021)
case 5:
    output.position = ComputeEdgePosition(ip, 1, 2);
    break;
// b(012)
case 6:
    output.position = ComputeEdgePosition(ip, 2, 1);
    break;

// Edge between v2 and v0

// b(102)
case 7:
    output.position = ComputeEdgePosition(ip, 2, 0);
    break;
// b(201)
case 8:
    output.position = ComputeEdgePosition(ip, 0, 2);
    break;

// Middle of triangle

// b(111)
case 9:
    float3 E =
                (
                    ComputeEdgePosition(ip, 0, 1) +
                    ComputeEdgePosition(ip, 1, 0)
                    +
                    ComputeEdgePosition(ipj 1, 2) +
                    ComputeEdgePosition(ipj 2, 1)
                    +
                    ComputeEdgePosition(ip, 2, 0) +
                    ComputeEdgePosition(ip, 0, 2)
                ) / 6.0f;
    float3 V = (ip[0].position + ip[1].position + ip[2].position)
               / 3.0f;

    output.position = E + ( (E - V) / 2.0f );

    break;

// Normals

// n(110) - between v0 and v1
case 10:
    output.normal = ComputeEdgeNormal(ip, 0, 1);
    break;
```

```
            // n(011) - between vl and v2
            case 11:
                output.normal = ComputeEdgeNormal(ip, 1, 2);
                break;

            // n(101) - between v2 and v0
            case 12:
                output.normal = ComputeEdgeNormal(ip, 2, 0);
                break;
        }

        return output;
    }
```

Listing 9.19. Hull shader for curved point normal triangles.

Equations (9.61) and (9.70) allow for aspects of each evaluation to be broken out into separate functions. Listing 9.20 covers these definitions.

```
float ComputeWeight(InputPatch<VS_OUTPUT, 3> inPatch, int i, int j)
{
    return dot(inPatch[j].position - inPatch[i].position, inPatch[i].normal);
}

float3 ComputeEdgePosition(InputPatch<VS_OUTPUT, 3> inPatch, int i, int j)
{
    return (
            (2.0f * inPatch[i].position) + inPatch[j].position
            - (ComputeWeight(inPatch, i, j) * inPatch[i].normal)
            ) / 3.0f;
}
float3 ComputeEdgeNormal(InputPatch<VS_OUTPUT, 3> inPatch, int i, int j)
{
    float t = dot
              (
               inPatch[j].position - inPatch[i].position
               , inPatch[i].normal + inPatch[j].normal
              );

    float b = dot
              (
               inPatch[j].position - inPatch[i].position
               , inPatch[j] .position - inPatch[i].position
              );

    float v = 2.0f * (t / b);
        return normalize
            (
                inPatchfi].normal + inPatch[j].normal
                - v * (inPatch[j].position - inPatch[i].position)
            );
}
```

Listing 9.20. Hull shader utility functions.

The Domain Shader

So far in this section, the discussion has centered on construction of the control mesh from the input positions and normal vectors. As discussed earlier, these are analogous to the coefficients of the equation that represents the ideal curved surface.

It is the domain shader's responsibility to take these coefficients, along with the sampling points generated by the fixed-function tessellator, and evaluate the new geometry that makes up the final curved surface:

$$\begin{aligned} P_{uvw} = {} & w^3 p_{300} + u^3 p_{030} + v^3 p_{003} \\ & + 3w^2 u p_{210} + 3u^2 w p_{120} \\ & + 3w^2 u p_{201} + 3v^2 w p_{102} \\ & + 3u^2 v p_{021} + 3v^2 u p_{012} \\ & + 6uvw p_{111}; \end{aligned} \tag{9.11}$$

$$n_{uvw} = w^2 n_{200} + u^2 n_{020} + v^2 n_{002} + uwn_{110} + uvn_{011} + vwn_{101}. \tag{9.12}$$

Equation (9.11) shows the cubic function that needs to be evaluated for each output position from the domain shader stage. Similarly, Equation (9.12) shows the quadratic function for computing each normal vector. Listing 9.21 shows an HLSL implementation of these two functions.

```
cbuffer Transforms
{
    matrix mWorld;
    matrix mViewProj;
    matrix mInvTposeWorld;
};

cbuffer TessellationParameters
{
    float4 EdgeFactors;
};
cbuffer RenderingParameters
{
    float3 cameraPosition;
    float3 cameraLookAt;
};
struct DS_OUTPUT
{
    float4 Position     : SV_Position;
    float3 Colour       : COLOUR;
};

[domain("tri")]
DS_OUTPUT dsMain
```

```
  (
    const OutputPatch<HS_OUTPUT, 13> TrianglePatch
    , float3 BarycentricCoordinates : SV_DomainLocation
    , HS_CONSTANT_DATA_OUTPUT input
  )
{
  DSJXJTPUT output;

  float u = BarycentricCoordinates.x;
  float v = BarycentricCoordinates.y;
  float w = BarycentricCoordinates.z;

  // Original Vertices
  float3 p308 = TrianglePatch[0].position;
  float3 p030 = TrianglePatch[1].position;
  float3 p003 = TrianglePatch[2].position;

  // Edge between v0 and v1
  float3 p210 = TrianglePatch[3].position;
  float3 pl20 = TrianglePatch[4].position;
    // Edge between vl and v2
  float3 p021 = TrianglePatch[5].position;
  float3 p012 = TrianglePatch[6].position;

  // Edge between v2 and v0
  float3 pl02 = TrianglePatch[7].position;
  float3 p201 = TrianglePatch[8].position;

  // Middle of triangle
  float3 p111 = TrianglePatch[9].position;

  // Calculate this sample point
  float3 p = (p300 * pow(w,3)) + (p030 * pow(u,3)) + (p003 * pow(v,3))
          + (p210 * 3.0f * pow(w,2) * u)
          + (pl20 * 3.0f * w * pow(u,2))
          + (p201 * 3.0f * pow(w,2) * v)
          + (p021 * 3.0f * pow(u,2) * v)
          + (pl02 * 3.0f * w * pow(v,2))
          + (p012 * 3.0f * u * pow(v,2))
          + (p111 * 6.0f * w * u * v);
  // Transform world position with view-projection matrix
  output.Position = mul( float4(p, 1.0), mViewProj );

  // Compute the normal - QUADRATIC
  float3 n200 = TrianglePatch[0].normal;
  float3 n020 = TrianglePatch[l].normal;
  float3 n002 = TrianglePatch[2].normal;

  float3 n110 = TrianglePatch[10].normal;
  float3 n011 = TrianglePatch[11].normal;
  float3 nl01 = TrianglePatch[12].normal;

  float3 vWorldNorm = (pow(w,2) * n200) + (pow(u,2) * n020) + (pow(v,2) * n002)
                    + (w * u * nll0) + (u * v * n011) + (w * v * nl01);

  vWorldNorm = normalize( vWorldNorm );
```

```
        // Perform a simple shading calc
    float3 toCamera = nonmalize( cameraPosition.xyz );
    output.Colour = saturate( dot(vWorldNorm, toCamera) ) * float3( 0.4, 0.4, 1.0 );

    return output;
}
```

Listing 9.21. Domain shader for curved point normal triangles.

Despite the importance of the domain shader stage, the code and mathematics involved are relatively simple and straightforward.

Example Output

When the previously described shader programs are plugged in with traditional triangle geometry, results such as those shown in Figure 9.41 can be generated. Considering that the

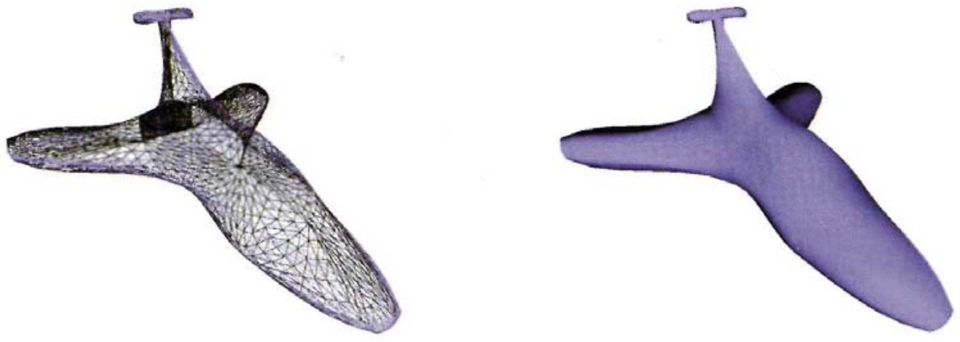


Figure 9.41. Example output with Curved Point Normal Triangles algorithm applied.

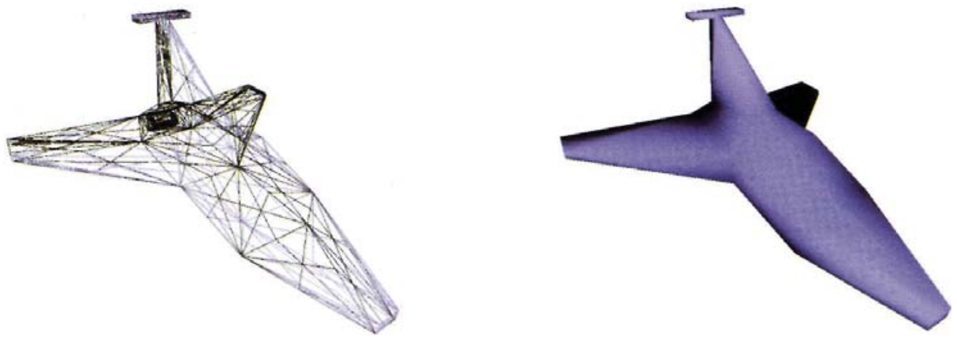


Figure 9.42. Original geometry without Curved Point Normal Triangles algorithm applied.

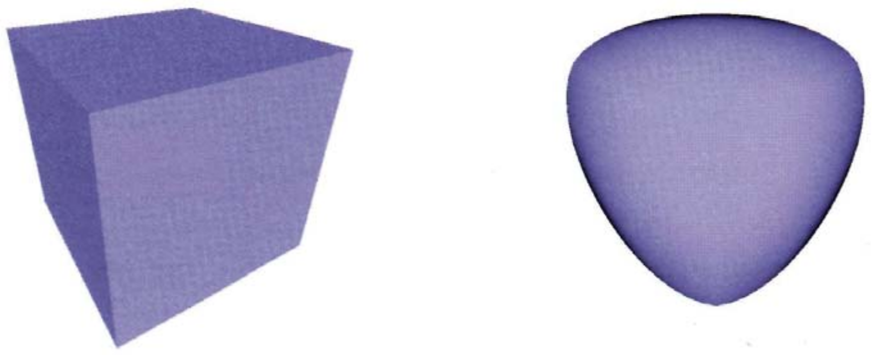


Figure 9.43. Excessive smoothed geometry loses expected sharp edges.

starting point of Figure 9.42 is an obviously low-polygon model, the improvement added by the Curved Point Normal Triangles algorithm is clear.

One characteristic to pay attention to in the wireframe representation in Figure 9.41 is that the tessellation is uniform across the entire mesh. This doesn't pose a problem with regard to the final visual results; but it is inefficient in some cases.

Despite generally good results, there are cases where this algorithm doesn't work very well. In Figure 9.43, notice that the result is probably too rounded (right hand side) and has actually lost the sharp edges that define the intended object (left side).

Encoding Sharp Edges

In this context, a *sharp edge* is an edge where the positional geometry between neighboring triangles matches, but the normal vectors do not. Figure 9.44 shows a simple example of this, in which both sides have a shared position in the centre, but the left side has differing normal vectors, and the right side has both a shared position and a shared normal.

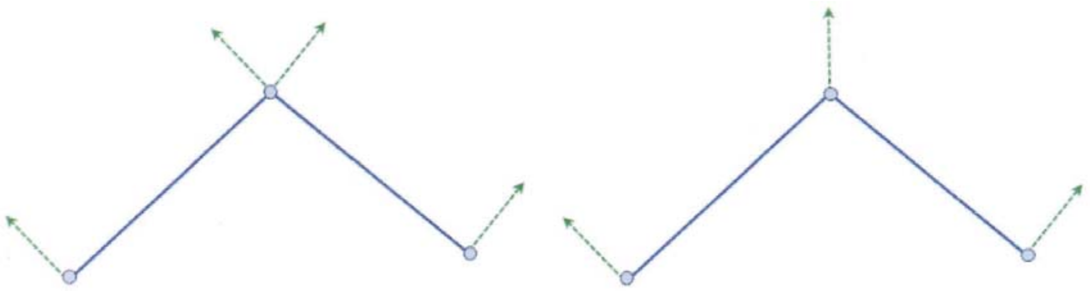


Figure 9.44. A simple example of a sharp edge.

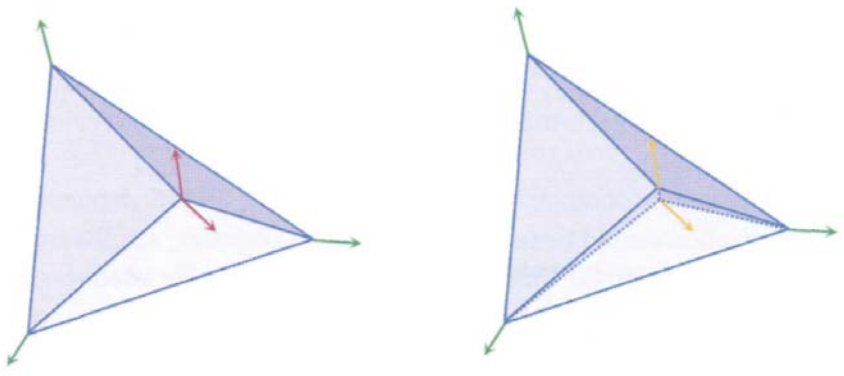


Figure 9.45. Splitting a shared vertex.

The original research paper for this algorithm makes a startling claim (with supporting proof) that simply splitting a vertex will often generate cracks in the surface of the object. Chapter 4 introduced the concept of tessellated surfaces being *watertight,* and this property of Curved Point Normal Triangles can shatter that!

Since the article proves that with only local information available, this problem cannot be avoided purely by using the GPU (remember, a key part of this algorithm was accepting unmodified triangle data without adjacency), a software pre-processing step becomes necessary.

Given that we're interested in edges, there are two possible variations—either one, or both, of the endpoints has differing normals. The software algorithm in use needs to examine all the vertex data in a given mesh and identify vertices where position information is the same, but normal vectors are not. Once these are identified, the simplest approach is to split the offending vertex in two by moving one endpoint very slightly away from the other. Figure 9.45 shows a simple case of this.

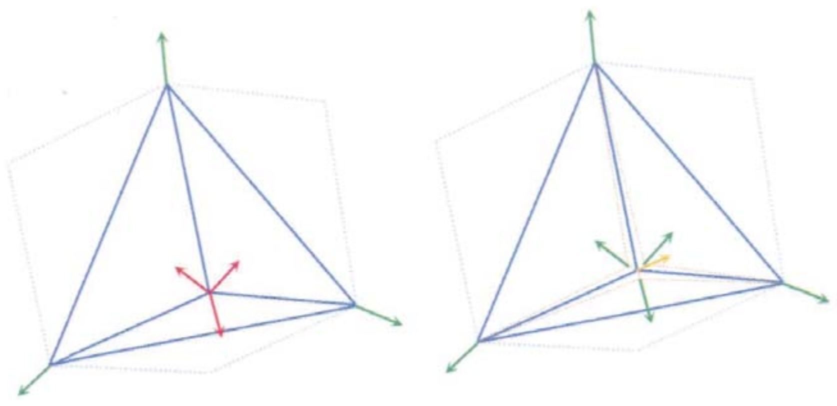


Figure 9.46. Splitting corner geometry.

Unfortunately, the general solution suffers because it can cause many triangles to share many different normal vectors. Figure 9.46 shows an example of corner geometry that first appeared in Figure 9.43. This has three triangles sharing the same position information at the corner, but each triangle requires a different normal vector to accurately represent the intended shape.

There are increasingly complex variations upon this algorithm, depending entirely on how strictly a sharp edge should be honored. At its most complex, a software implementation of the Curved Point Normal triangles implementation can be used to generate intermediary triangles that closely match the surface and control points that would otherwise be generated by the GPU.

While this does solve the problem, it is not an ideal solution, partly because it is nontrivial to implement, but mainly because it breaks the original goal of not having to modify the original geometry.

Back-Face Culling in the Hull Shader

Culling of primitives facing away from the viewer is a classic staple of computer graphics. Modern hardware pipelines will still perform this step (subject to pipeline configuration by the application), and it is largely transparent and oft ignored.

This is interesting with regard to tessellation, due to the ability to amplify geometry (geometry shaders are also interesting for the same reason). With the pipeline configuration used thus far in this chapter, back-face culling will occur before rasterization, exactly as expected, but it is crucial that this is after tessellation has occurred.

Simply put, the hardware may have generated hundreds, if not thousands, of new triangles, only to have them be rejected without making any contribution to the final rendered image. In a high-performance scenario, this wasteful processing is obviously undesirable. It would be far better if it could be avoided entirely.

Direct3D 11 hull shaders do allow this, and before any amplification or complex processing, it is possible to reject an entire patch and cease further processing. Implementing the check to determine if a patch is back facing is the responsibility of the hull shader author, and while this check for a triangle is trivial, it may not be so simple for all tessellation algorithms.

The developer needs to consider the final tessellated output and the possibility that the eventual surface may actually contain elements that aren't entirely back facing. In the context of Curved Point Normal Triangles, the final surface remains relatively close to the underlying control mesh, making this optimization possible.

```
HS_CONSTANT_DATA_OUTPUT hsConstantFunc( InputPatch<VS_OUTPUT, 3> ip, uint
PatchID : SV_PrimitiveID )
{
    HS_CONSTANT_DATA_OUTPUT output;
    float3  faceNormal
            = normalize
              (
                cross
                (
                    ip[2].position - ip[0].position
                    , ip[1].position - ip[0].position
                )
              );
    float3 viewDirection = normalize(cameraLookAt - cameraPosition);

    float backFace = sign(0.2 + dot(faceNormal,viewDirection));

    output.Edges[0] = EdgeFactors.x * backFace;
    output.Edges[1] = EdgeFactors.y * backFace;
    output.Edges[2] = EdgeFactors.z * backFace;

    output.Inside = EdgeFactors.w * backFace;

    return output;
}
```

Listing 9.22. Modified hull shader constant function.

Hull shader constant functions that output zero or negative edge tessellation factors will be culled by the pipeline. In Listing 9.22, this is determined by a simple check on the face normal for the incoming triangle. If the inner product of this normal and the current view direction is positive (both point in the same direction), the patch is considered back facing. There are two details to pay attention to with this code. First, the sign function which will return either -1.0 or +1.0 and can thus be used to multiply with the edge factors and avoid unnecessary conditionals/branching. Second, the test for being a back face has a "magic" 0.2 factor included, which acts as a simple threshold to stop premature culling of the patch. Due to the curvature of patches, having them culled on the precise turning point can lead to noticeable artifacts in the final image. This factor can be tweaked as necessary, closer to 0.0 will remove more patches, but with increased chance of visual artifacts.

It is important to note that the control point phase of hull shader execution occurs before evaluation of the per-patch constant function. Consequently, the hull shader may perform significant work before it is culled, and further downstream processing can be avoided. Figure 9.47 shows the results of this optimization (right side).

Note that by definition, the difference is hard to detect, but the wireframe rendering appears to be less dense, indicative of the fewer patches being rendered. In this example, the number of domain shader invocations drops from 19,488 to 9,570, and the number of triangles rasterized falls from 16,800 to 8,250—a 51% reduction of both of the unoptimized totals.

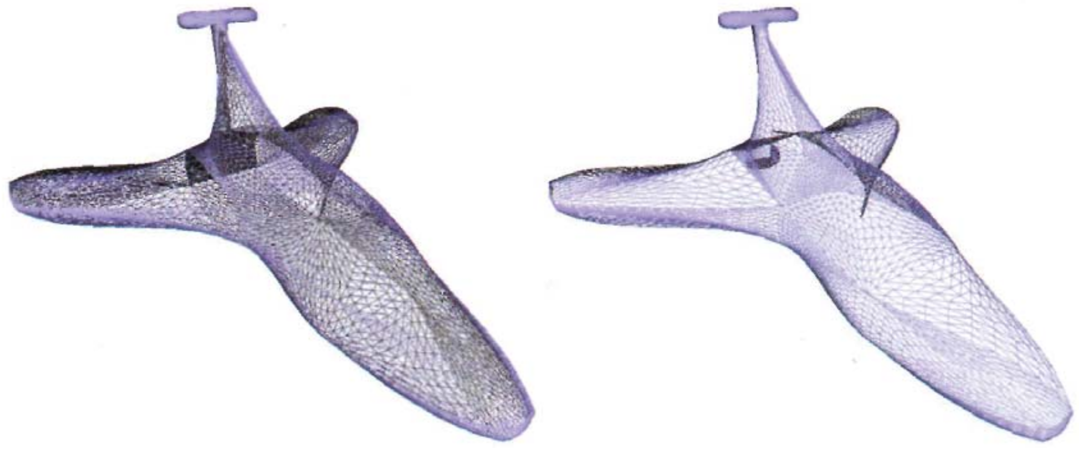


Figure 9.47. Results of using back-face culling.

Given that this optimization has no impact on the final image (unless transparency or similar complex blending operations are being used) it results in a non-trivial reduction in the work that the GPU has to perform.

9.2.2 Conclusion

As previously demonstrated, it is straightforward to implement the original Direct3D-8-era algorithm using Direct3D 11. Given that tessellation is a basic requirement of the specification, this also eliminates the original problem with ATI's TruForm implementation, which only worked on a limited set of hardware. Now, all Direct3D 11 generation hardware supports this algorithm.

The added expressiveness of the Direct3D 11 pipeline allows further extensions to the original algorithm. The main extension to be considered is to break the uniform tessellation across the entire surface of the mesh. As shown in previous images, triangles are generated and distributed evenly across the surface, even where there are only low-frequency variations.

By utilizing mathematics similar to that shown in the back-face culling section, one can dynamically alter the tessellation factors output by the hull shader and weight them to be higher near the edge of the mesh, and in areas of high-frequency variation. However, this requires modification of the input data, since adjacency becomes necessary to avoid cracks between neighboring tiles.

Without any of these changes, the Curved Point Normal Triangles algorithm is a good demonstration of Direct3D 11 's new functionality, as well as being a convenient technique to enhance existing triangle-based meshes.