1.1

Smoothed N-Patches

Holger Gruen, **Intel Corp.**

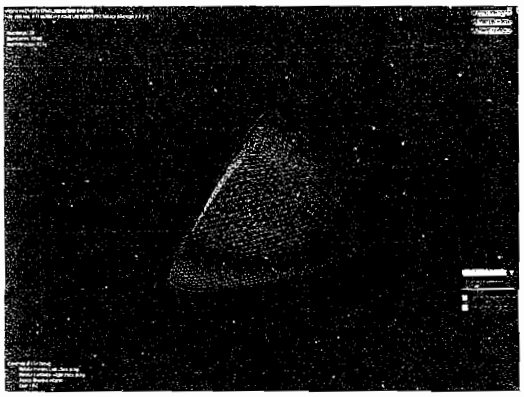

FIGURE 1.1.1A A tetrahedron refined using the original N-Patch.

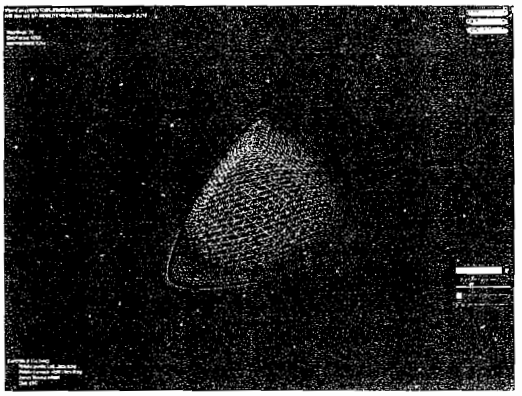

FIGURE 1.1.1B A tetrahedron refined using the new patch.

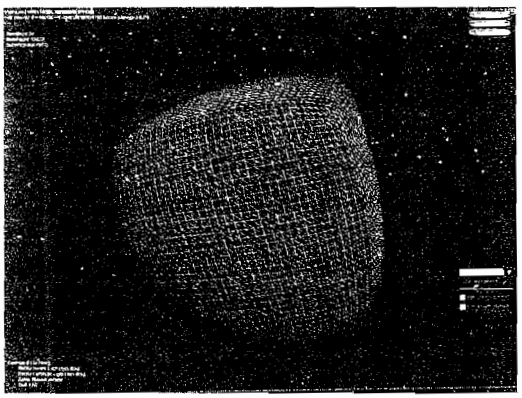

FIGURE 1.1.1C A cube refined using the original N-Patch.

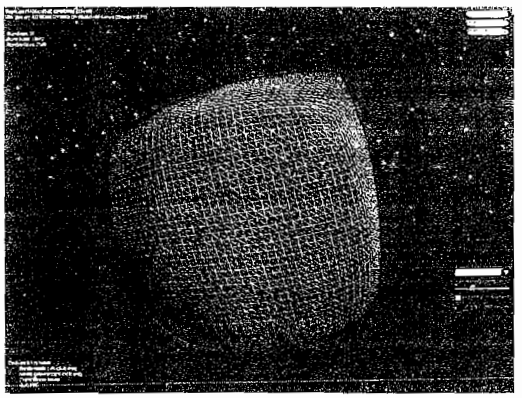

FIGURE 1.1.1D A cube refined using the new N-Patch.

Abstract

N-Patches [Vlachos01] provide a simple way to visually smoothen triangular objects. The smoothing suffers, though, because the composite surface generated by two N-Patches is visually only smooth at the vertices and along edges shared by two object triangles. If you consider directions across (e.g., not parallel to) a shared edge, the generated surface does not necessarily look smooth. This is especially obvious for objects with very low triangle counts.

This article introduces a way to improve N-Patches to make them visually smooth across edges also. The method described does not resort to using subdivision surfaces and ideally only needs the normals of neighboring faces. The surfaces generated by the method in general look smoother than normal N-Patches for objects with low polygon counts.

With the introduction of DirectX10® Geometry Shaders and techniques such as *tessellation through instancing* [Gruen05], it is possible to evaluate the described patches on-the-fly and efficiently for complete objects on the GPU and even on DirectX9 class hardware. Also, for many pass-rendering architectures, an efficient multithreaded CPU implementation that makes use of SSE and writes to dynamic vertex buffers is possible.

Introduction

The idea of having a low-polygon-count (low-poly) object representation that can be refined later on and smoothed on-the-fly is very desirable. A technology like that will result in smooth silhouettes and produce an organic look. Relatively low-poly objects can then be used to save precious video memory, something that is especially relevant for game consoles.

The downside of a low-poly mesh is obvious; it simply does not look as good as a high-polygon-count (high-poly) object. Some aspects of this low-poly look can be improved by using normal maps to emulate fine details through per-pixel lighting (see, e.g., [Cloward]). Still, the silhouettes of such an object look crude.

Several researchers have tackled the problem of how to improve silhouettes of low-poly object approximations. These solutions range from view-dependent level-of-detail (LOD) techniques for high-poly objects (see, e.g., [Hoppe97]) to approaches that only draw an improved silhouette (see, e.g., [Loviscach03], [Sander00], or [Sander01]). The downside of view-dependent LOD techniques is that they need at least as much memory as a high-poly object. Approaches to only improve the silhouette are more interesting in terms of memory footprint. Unfortunately they put a high burden on the CPU to identify silhouette edges and usually issue a large number of batches. All this does not contribute to optimal performance. Applying such approaches to a potentially large number of objects as featured by many games is problematic.

A completely different approach to generate nice silhouettes for low-poly objects is to replace triangles of the object with curved surface patches. Alternatively, one can

represent the object completely by the control points necessary to define these patches. The kind of curved surfaces to be used ranges from parametric patches to subdivision surfaces.

A large body of work describes how to realize efficient refinement and tessellation of objects and patches on the CPU (see, e.g., [Lien87] or [Bolz02]). Using these techniques, high-poly representations for every LOD needed can be generated. It is also possible to use even finer tessellations for patches that border silhouette edges of the mesh. The resulting meshes then automatically feature the desired smooth silhouettes. If cached properly, the different LODs of such an object can be computed without putting too much strain on the CPU. If the rendering pipeline used does use many render passes, for example, for lighting it can be faster to use objects that are tessellated on the CPU as opposed to tessellation on the GPU. Recent CPUs feature several cores. These can be successfully utilized to speed up object tessellation (see [Gruen06]).

Using subdivision surfaces to smoothen triangular objects results in smoothed objects with a good visual quality. The thing that makes subdivision surfaces undesirable is that one needs to consider a potentially unlimited number of vertices in the neighborhood of an irregular vertex of the subdivision to evaluate the surface.

NP-Triangles (also called N-Patches [Vlachos01]) were introduced to realize object refinement without any knowledge of neighboring triangles or vertices. N-Patches and other curved surfaces are part of DirectX but are supported in GPU hardware only by a very limited installed hardware base. This limited hardware base resulted in almost zero adoption of this tessellation technology. Current graphics hardware does not support on-the-fly N-Patches any more. Fortunately DirectX offers CPU-based routines to generate refined meshes.

Since N-Patches don't consider any neighboring vertices or faces, refinements of low-poly objects don't feature the visual smoothness of objects refined with subdivision schemes. The way N-Patches are defined implies that the overall surface is not always visually smooth in directions across edges shared by two triangles of an object. The goal of this article is to find an improved patch that behaves like the N-Patch in the interior of a triangle but also produces a reasonably smoothed surface along the directions across shared edges.

The upcoming Shader Model 4.0 supports *geometry shaders* that will enable GPU programs that introduce new vertices and triangles [Tatarchuk06]. Still, game developers need to support older platforms as well; fortunately, tessellation through instancing (see [Gruen05]) enables tessellation on the GPU on DirectX9-class hardware if tessellation on the CPU is not an option.

Before a new way to generate a smoother N-Patch is described, this article first describes how the original N-Patch is constructed. The curved surface patches that will be discussed are defined on a triangular parameter domain. Therefore, barycentric coordinates and Gregory coordinates [Farin99] will be covered as well since these coordinates are commonly used for defining triangular parameter domains.

The Original N-Patch

N-Patches [Vlachos01] are triangular Bezier patches of degree three. These patches are defined by 10 control points as shown in Figure 1.1.2.

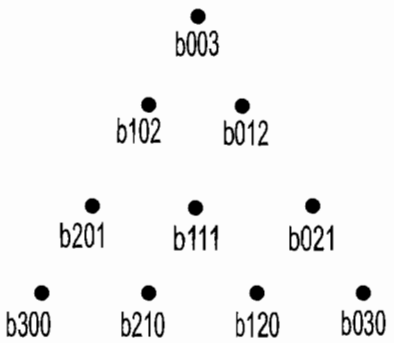

FIGURE 1.1.2 The control points of a bicubic triangular Bezier patch.

Each control point is derived from the three corners of an original triangle and the three normals at the corners. If *v0*, *v1*, and *v2* are the vertices and *n0*, *n1*, and *n2* are the normals of a triangle, then the control points can be computed using Equation 1.1.1.

$$
\begin{aligned}
b_{300} &= v0 \\
b_{030} &= v1 \\
b_{003} &= v2 \\
b_{210} &= (2v0 + v1 - dot(v1 - v0, n0) * n0) / 3 \\
b_{120} &= (2v1 + v0 - dot(v0 - v1, n1) * n1) / 3 \\
b_{021} &= (2v1 + v2 - dot(v2 - v1, n1) * n1) / 3 \\
b_{012} &= (2v2 + v1 - dot(v1 - v2, n2) * n2) / 3 \\
b_{201} &= (2v0 + v2 - dot(v2 - v0, n0) * n0) / 3 \\
b_{102} &= (2v2 + v0 - dot(v0 - v2, n2) * n2) / 3 \\
b_{111} &= 3 (b_{210} + b_{120} + b_{021} + b_{012} + b_{102} + b_{201}) / 12 - (v0 + v1 + v2) / 6
\end{aligned}
\qquad (1.1.1)
$$

To compute points on the surface of an N-Patch Equation 1.1.2 needs to be evaluated for a triplet of barycentric coordinates *b0, b1, b2.* How barycentric coordinates are defined on a triangle will be described later.

$$
\begin{aligned}
P(b0,b1,b2) = {} & v0\, b0^3 + v1\, b1^3 + v2\, b2^3 + \\
& 3b_{102}\, b0\, b2^2 + 3\, b_{012}\, b1\, b2^2 + \\
& 3b_{201}\, b0^2\, b2 + 3\, b_{021}\, b1^2\, b2 + \\
& 3b_{210}\, b0^2\, b1 + 3\, b_{120}\, b0\, b1^2 + \\
& 6b_{111}\, b0\, b1 b2
\end{aligned}
\qquad (1.1.2)
$$

If shared triangle edges agree on the normals at the shared vertices, N-Patches will produce a closed surface along shared edges.

Barycentric Coordinates and Gregory Coordinates

N-Patches are triangular Bezier patches. These patches are defined over a barycentric parameter domain. To understand how barycentric coordinates are defined, consider Figure 1.1.3.

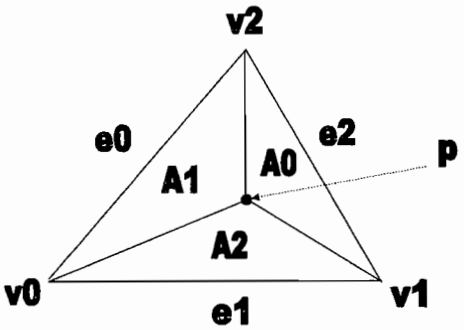

FIGURE 1.1.3 If A is the area of a triangle, barycentric coordinates of a point p are defined by the ratios of $A0/A$, $A1/A$, and $A2/A$.

A is the area of the full triangle defined by *v0, v1,* and *v2. A0, A1,* and *A2* are the areas of the corresponding subtriangles defined by a point p as shown in Figure 1.1.3. Given these triangles, the barycentric coordinates $b0$, $b1$, and $b2$ of the point p are defined by the following equation:

$$\begin{aligned} b0 &= A0/A \\ b1 &= A1/A \\ b2 &= A2/A \\ b0+b1+b2 &= 1. \end{aligned} \tag{1.1.3}$$

Equation 1.1.3 also implies that $b0 = 1.0$ at *v0*, $b1 = 1.0$ at *v1*, and $b2 = 1.0$ at *v2*.

Gregory coordinates (see [Farin99]) are the equivalent of barycentric coordinates. Each of the three Gregory coordinates equals 1.0 on exactly one of the edges of a triangle as opposed to one vertex for the barycentric coordinates. Later on this article describes the need for three blending factors that sum up to 1.0 everywhere in a triangle but that only one of them is supposed to be one on each of the edges. Gregory coordinates are defined in terms of barycentric coordinates as described by the following equation:

$$g0 = b0 \cdot b2 \Big/ (b0 \cdot b1 + b0 \cdot b2 + b1 \cdot b2)$$
$$g1 = b0 \cdot b1 \Big/ (b0 \cdot b1 + b0 \cdot b2 + b1 \cdot b2)$$
$$g2 = b1 \cdot b2 \Big/ (b0 \cdot b1 + b0 \cdot b2 + b1 \cdot b2)$$
$$g0 + g1 + g2 = 1. \tag{1.1.4}$$

Equation 1.1.4 implies that *g0 = 1.0* on *e0* (see Figure 1.1.3), *g1 = 1.0* on *e,1* and *g2 = 1.0* on *e0.* It has to be noted that Gregory coordinates are not defined at the vertices of the triangle, since $b0 \cdot b1 + b0 \cdot b2 + b1 \cdot b2$ is 0 at the vertices. As will be shown later, this does not really limit their usability for the purpose of this article.

First Step Toward a Smoother N-Patch

The way N-Patches are defined already generates a visually smooth surface along shared triangle edges and at the triangle's vertices. In order to make it smooth across the edges, one wants to ideally change the surface in a way that guarantees the same surface normal for two patches that meet along a curve above a shared triangle edge.

The exterior (e.g., not b_{111}) control points of the N-Patch as defined by Equation 1.1.1 define three cubic Bezier border curves. The plane normal formed by the cross product of the derivatives of two of these curves at every vertex of the triangle will be the same normal as the one provided as input for Equation 1.1.1. Considering this, to achieve overall visual smoothness, it could be a viable way to make sure that the surface on the edges has a normal that is a linear blend of the normals of the two vertices forming the edge.

The border curves of the original N-Patch are defined by four Bezier control points (see Figure 1.1.4).

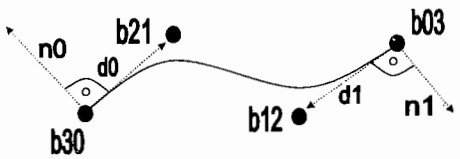

FIGURE 1.1.4 The start and end derivatives of a cubic Bezier curve point toward the inner control points.

The two control points *b30* and *b03* are the starting point and the endpoint of the curve and are interpolated by the curve. The interior control points are placed along the intended directions of the start and end derivatives *d0* and *d1* and are not interpolated. These derivatives are derived from the two points defining an edge and

the corresponding normals *n0* and *n1*. Note that *d0* is perpendicular to *n0* and *d1* is perpendicular to *n1*. Equation 1.1.5 describes how to compute these interior control points if two vertices and two normals are given.

$$b_{21} = (2\, b_{30} + b_{03} - dot(b_{03} - b_{30},\, n0) * n0) / 3$$
$$b_{12} = (2\, b_{03} + b_{30} - dot(b_{30} - b_{03},\, n1) * n1) / 3 \qquad (1.1.5)$$

For every point *p* on a curve (e.g., the curve for edge *e1*), as defined by the original N-Patch, a new curve can be set up. For this curve, *p* is chosen as the starting point and *v2* as the end point. The interior control points are derived from Equation 1.1.5. The starting normal is chosen to be a linear blend of *n0* and *n1*. The end-normal is *n2* (see top right of Figure 1.1.5).

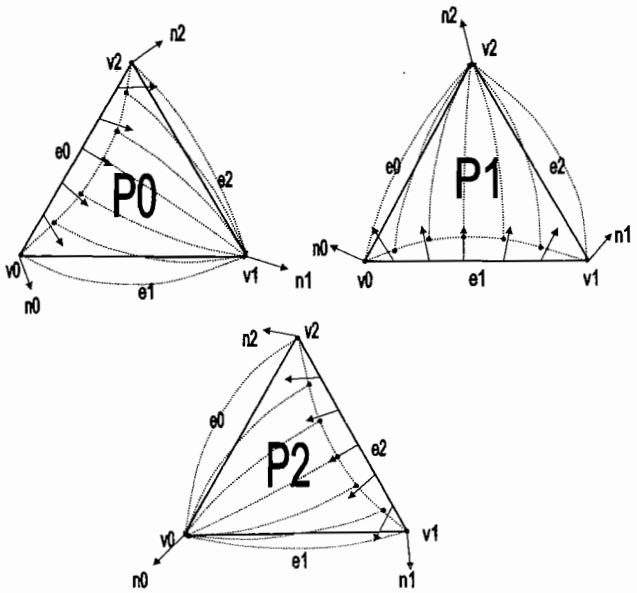

FIGURE 1.1.5 One creates one patch for every edge *e0*, *e1*, and *e2*.

The overall surface generated by all these curves forms a new patch definition. The derivative of each curve at its starting point will be perpendicular to the interpolated (blended) normal, and the curves of an adjacent patch (sharing *e1*) will also be perpendicular to this normal. Since the shared Bezier curves of two adjacent triangles also agree on derivatives along the curve, the surface normals of both patches will be identical along a shared edge. Also note that a patch defined like this reproduces exactly the border curves of the original N-Patch.

Using this train of thought, a similar patch can be defined for every of the three edges (see patches *P0, P1,* and *P2* in Figure 1.1.5) of the triangle. If Gregory coordinates

(see Equation 1.1.4) are used to trilinearly blend these three patches, one ends up with surface patches that would agree on their normals on shared curves or edges. The following equation defines this blending operation:

$$P = g0^n \cdot P0 + g1^n \cdot P1 + g2^n \cdot P2 \tag{1.1.6}$$

The only thing left to define is how to evaluate the new patch, given a triplet of barycentric coordinates. The first thing to do for each of *P0, P1,* and *P2* is to find the appropriate starting points on the corresponding border curves. To compute these points, one needs to know where or when to evaluate the curves to get the starting points. In other words, three curve parameters (*st0, st1,* and *st2*) have to be derived from the barycentric coordinates *b0, b1,* and *b2.* As it turns out, it is easy to choose these parameters. Consult Equation 1.1.7 to understand how to choose them.

$$\begin{aligned}
st0 &= b1/(1-b2) \quad for \quad b2 \neq 1 \\
st0 &= 0.5 \quad for \quad b2 = 1 \\
st1 &= b2/(1-b0) \quad for \quad b0 \neq 1 \\
st1 &= 0.5 \quad for \quad b0 = 1 \\
st2 &= b0/(1-b1) \quad for \quad b1 \neq 1 \\
st2 &= 0.5 \quad for \quad b1 = 1.
\end{aligned} \tag{1.1.7}$$

Given *st0, st1,* and *st2,* one can evaluate the border curves to find the starting points of the current curves (e.g., *c0, c1,* and *c2*) for every patch *P0, P1,* and *P2.* After setting up the three curves, for every curve, one finally only needs to find the actual curve parameter. If these curve parameters are called *ct0, ct1,* and *ct2,* then the following equation shows how to choose them:

$$\begin{aligned}
ct0 &= b1 \\
ct1 &= b2 \\
ct2 &= b0
\end{aligned} \tag{1.1.8}$$

Now everything is in place to test the patch. The initial implementation shown below (see Listing 1.1.1) is realized with an HLSL vertex shader that is intended to work within the context of tessellation through instancing (see [Gruen05]). The basic idea is to instance a vertex buffer that contains a flat barycentric tessellation of a triangle (see Figure 1.1.6) with vertex data that only contain barycentric coordinates for every triangle of a mesh that needs to be refined. As per instance data, a second vertex buffer contains the vertices, normals, and texture coordinates of one mesh triangle. Note that the shader simply uses trilinearly interpolated normals and does not try to compute real normals. This is also the way normals are computed for N-Patches. Alternatively, a quadratic Bezier patch can be used for normal approximation (see [Vlachos01]).

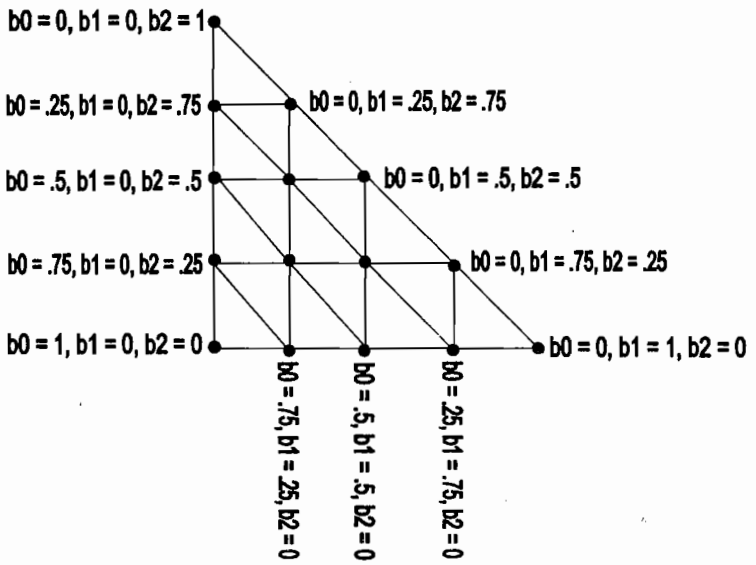

FIGURE 1.1.6 A regular barycentric tessellation of a triangle.

LISTING 1.1.1 Initial Implementation

```
// evaluate a curve defined by two vertices and two normals at b
   (see Equation 1.1.5)
float3 evaluateCurve( float3 v0,
                      float3 v1,
                      float3 n0,
                      float3 n1,
                      float  b
                     )
{
float  b_   = 1.0 - b;

// compute internal control points
  float3 b21 = ( 2 * v0 + v1 - dot( v1 - v0, n0 ) * n0 );
  float3 b12 = ( 2 * v1 + v0 - dot( v0 - v1, n1 ) * n1 );

  // evaluate curve
  return  b_ *b_* b_ * v0  +
          b * b * b  * v1  +
          b_ *b_* b  * b21 +
          b_ *b * b  * b12;
}

void VSInstancedPatch( float3 b    : POSITION,  // barycentric
                                                   coords
                       float3 v0          : TEXCOORD0,
                                            // v0 of triangle
                       float3 v1          : TEXCOORD1,
                                            // v1 of triangle
```

```
                        float3 v2          : TEXCOORD2,
                                             // v2 of triangle
                        float3 n0          : TEXCOORD3,
                                             // normal for v0
                        float3 n1          : TEXCOORD4,
                                             // normal for v1
                        float3 n2          : TEXCOORD5,
                                             // normal for v2
                        float2 t0          : TEXCOORD6,
                                             // texcoord for v0
                        float2 t1          : TEXCOORD7,
                                             // texcoord for v1
                        float2 t2          : TEXCOORD8,
                                             // texcoord for v2
                        out float4 oPos    : POSITION,
                        out float2 oTex    : TEXCOORD0,
                        out float4 Diffuse : COLOR0  )
{
float3 g; // gregory coords
float3 c; // parameters for border curves/edges

  float  b0b2 = b.x * b.z, b0b1 = b.x * b.y, b1b2 = b.y * b.z;
float  sum = b0b2 + b0b1 + b1b2;

  // compute gregory coords from barycentric coords
  if( b.x > 0.999f || b.y > 0.999f || b.z > 0.999f )
  {
    if( b.x > 0.999f )
      g.x = 1.0f, g.y = g.z = 0.0f;
    else if( b.y > 0.999f )
      g.y = 1.0f, g.x = g.z = 0.0f;
    else
      g.z = 1.0f, g.x = g.y = 0.0f;
  }
  else
  {
    g.x = b0b2/sum;
    g.y = b0b1/sum;
    g.z = b1b2/sum;
  }

// compute parameters for border curves
  if( 1.0f - b.z != 0.0f )
    c.x = b.y / ( 1.0f - b.z );
  else
    c.x = 0.5f;
  if( 1.0f - b.x != 0.0f )
    c.y = b.z / ( 1.0f - b.x );
  else
    c.y = 0.5f;
  if( 1.0f - b.y != 0.0f )
    c.z = b.x / ( 1.0f - b.y );
  else
    c.z = 0.5f;
```

```
        // compute points on border curves
        float3 pos0  = evaluateCurve( v0, v1, n0, n1, c.x );
        float3 pos1  = evaluateCurve( v1, v2, n1, n2, c.y );
        float3 pos2  = evaluateCurve( v2, v0, n2, n0, c.z );

        // compute interpolated normals
        float3 norm0 = normalize( lerp( n0, n1, c.x ) );
        float3 norm1 = normalize( lerp( n1, n2, c.y ) );
        float3 norm2 = normalize( lerp( n2, n0, c2.z ) );

        // evaluate points on each of the three curves
        pos0 = evaluateCurve( pos0, v2, norm0, n2, b.z );
        pos1 = evaluateCurve( pos1, v0, norm1, n0, b.x );
        pos2 = evaluateCurve( pos2, v1, norm2, n1, b.y );

        // gregory blend points on curves
        float3 pos = g.x * pos2 + g.y * pos0 + g.z * pos1;

        // transform
        oPos = mul( float4( pos, 1 ), g_mWorldViewProjection );

        float3 Normal = mul( normalize( b.x * n0 + b.y * n1 + b.z * n2
), (float3x3)g_mWorld );
        Diffuse = saturate( dot( Normal, float3( 0.0f, 0.0f, -1.0f ) )
) * g_vDiffuse;
        Diffuse.w = 1.0;

        oTex = b.x * t0 + b.y * t1 + b.z * t2;
    }
```

An initial test of the patch for a tetrahedron produces the results shown in Figures 1.1.7. It can be seen that the new patch produces a visually smoother result than the original N-Patch. Obviously more tests need to be run to check if the patch works properly for other objects as well.

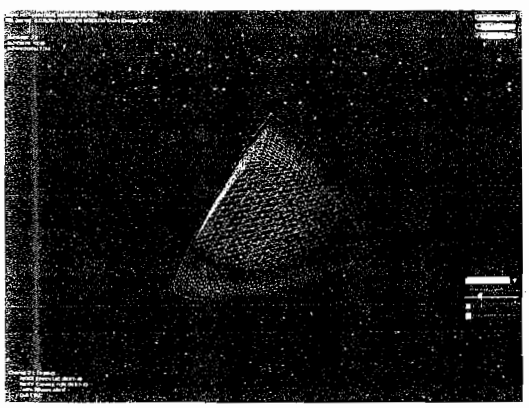

FIGURE 1.1.7A A tetrahedron refined using the original N-Patch.

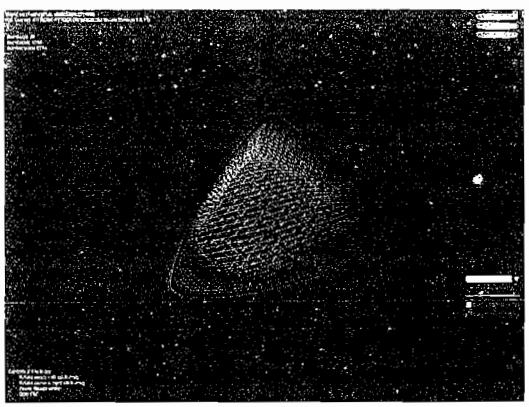

FIGURE 1.1.7B A tetrahedron refined using the new intermediate patch.

Unfortunately the next test object, which is a cube (part of the media that comes with the DirectX SDK), already reveals that the new patch does not always behave nicely. Figure 1.1.8 shows a refined cube. Although it certainly is visually smooth, it produces a smooth but undesirable valley along the edges of triangles that form one of the six faces of the cube.

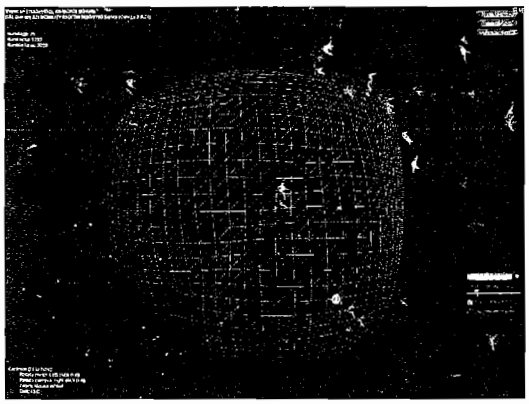

FIGURE 1.1.8A Refined cube showing smooth edges where cube faces meet.

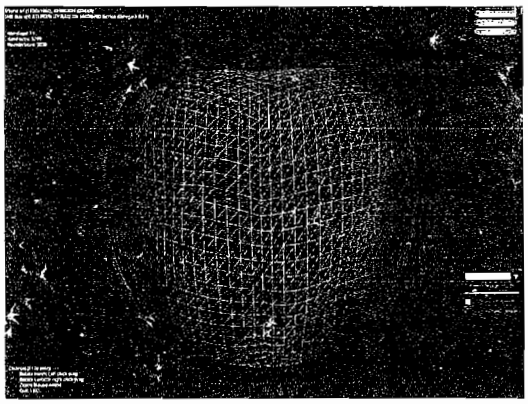

FIGURE 1.1.8B Refined cube showing ridges along the coplanar triangles that form the faces of the cube.

One way to explain these ridges is shown in Figure 1.1.9 and explained in the following text.

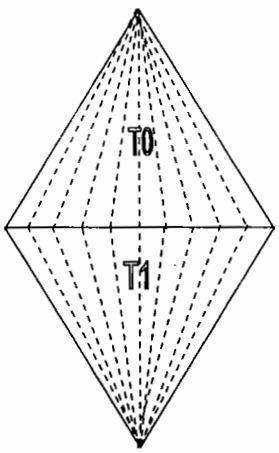

FIGURE 1.1.9 Parameter space lines take off into different directions from the shared position on a shared triangle edge.

Along the edges of coplanar triangles (e.g., triangles forming a face of a cube), patches share the same normal on the shared curve, but each of the curves (see dashed lines in Figure 1.1.9; two triangles *T0* and *T1* are shown) are not parallel when they meet. The normal of the surface is still the same, but a small local neighborhood of these nonparallel curves forms a little wedge. The visible valley is created when the curves sweep this wedge. In addition to this, the patch bulges out more than the original N-Patch does near the center of the triangle. This makes the valleys even more prominent.

Interestingly, the surface looks smooth without ridges where edges from adjacent cube faces meet. The angle between the surface normals of the triangles that meet at these edges is 90°. Therefore, the wedge sweeps a ridge that bulges out and does not hamper visual smoothness.

From observation, the original N-Patch does a good job producing visual smoothness if the normals of adjacent triangles are equal or if they are pointing in similar directions. This suggests that a method to blend the N-Patch and the new patch needs to be found. This method needs to create a patch that behaves like the N-Patch near the center of the triangle and near shared edges with triangles with almost parallel normals.

Finalizing the Smoother N-Patch

In accordance with what has just been discussed, the first thing to find is a term that is 1 at the center of the triangle and slowly goes down to 0 on the edges. Equation 1.1.2 luckily already contains a term that can be used. It is the polynomial $b_{111}b0 \cdot b1 \cdot b2$. Since its value does not go up to 1.0 at the center of the triangle, it needs to be scaled up by 3^3. The resulting blending term is one ingredient of the solution that is about to be described.

$$bl = 3^3 b_{111} b0 \cdot b1 \cdot b2 \tag{1.1.9}$$

The second ingredient is the absolute value of the dot product of the face normals of two of the triangles that are adjacent along an edge. Since there are usually three shared edges, there are also three of these values *fn0, fn1,* and *fn2.* In order to really control the transition from the edge to the center, for every triangle edge one needs three blending factors to blend between the new patch and the N-Patch. Equation 1.1.10 shows how to set up these blending factors.

$$\begin{aligned} bf0 &= \min\left(1, fn0 + bl\right) \\ bf1 &= \min\left(1, fn1 + bl\right) \\ bf2 &= \min\left(1, fn2 + bl\right) \end{aligned} \tag{1.1.10}$$

To use these factors, one first evaluates the original N-Patch () resulting in a point *np*. Then for every patch *P0, P1,* and *P2* (see Figure 1.1.5), blend between *np* and the position on the corresponding patch. This results in the following equation:

$$pos0 = lerp(P0, np, bf0)$$
$$pos1 = lerp(P1, np, bf1)$$
$$pos2 = lerp(P2, np, bf2) \quad (1.1.11)$$

Now the only thing left is to do a Gregory blend of theses three positions. Bringing it all together, an implementation—again in a vertex shader designed to run within a tessellation through instancing context—looks like this:

LISTING 1.1.2 Implementation

```
float3 evaluateNPatch( float3 v0,
                       float3 v1,
                       float3 v2,
                       float3 n0,
                       float3 n1,
                       float3 n2,
                       float3 b
                     )
{
  float3 b102 = ( 2 * v2 + v0 - dot( v0 - v2, n2 ) * n2 );
  float3 b012 = ( 2 * v2 + v1 - dot( v1 - v2, n2 ) * n2 );
  float3 b201 = ( 2 * v0 + v2 - dot( v2 - v0, n0 ) * n0 );
  float3 b021 = ( 2 * v1 + v2 - dot( v2 - v1, n1 ) * n1 );
  float3 b210 = ( 2 * v0 + v1 - dot( v1 - v0, n0 ) * n0 );
  float3 b120 = ( 2 * v1 + v0 - dot( v0 - v1, n1 ) * n1 );

  return  b.x*b.x*b.x * v0 + b.y*b.y*b.y * v1 + b.z*b.z*b.z * v
  2 + b.x*b.z*b.z * b102 +
        b.y*b.z*b.z * b012 + b.x*b.x*b.z * b201 + b.y*b.y*b.z*
        b021 +
        b.x*b.x*b.y * b210 + b.x*b.y*b.y * b120 +
        b.x*b.y*b.z * ( 0.5 * ( b210 + b120 + b021 + b012.xyz +
        b102 + b201 ) -
                                    v0 - v1 - v2 );
}

void VSInstancedPatch( float3 b          : POSITION,
                                           // barycentric coords
                       float3 v0         : TEXCOORD0,
                                           // v0 of triangle
                       float3 v1         : TEXCOORD1,
                                           // v1 of triangle
                       float3 v2         : TEXCOORD2,
                                           // v2 of triangle
                       float3 n0         : TEXCOORD3,
                                           // normal for v0
                       float3 n1         : TEXCOORD4,
                                           // normal for v1
                       float3 n2         : TEXCOORD5,
                                           // normal for v2
```

```
                                float3 v0_          : TEXCOORD6,
                                                      // vtx opposite v0
                                                      // across shared edge
                                float3 v1_          : TEXCOORD7,
                                                      // vtx opposite v1
                                                      // across shared edge
                                float3 v2_          : TEXCOORD8,
                                                      // vtx opposite v2
                                                      // across shared edge
                                float2 t0           : TEXCOORD9,
                                                      // texcoord for v0
                                float2 t1           : TEXCOORD10,
                                                      // texcoord for v1
                                float2 t2           : TEXCOORD11,
                                                      // texcoord for v2
                                out float4 oPos     : POSITION,
                                out float2 oTex     : TEXCOORD0,
                                out float4 Diffuse : COLOR0  )
{
float3 g; // gregory coords
float3 c; // parameters for border curves/edges

  float  b0b2 = b.x * b.z, b0b1 = b.x * b.y, b1b2 = b.y * b.z;
float  sum = b0b2 + b0b1 + b1b2;

  // compute gregory coords from barycentric coords
  if( b.x > 0.999f || b.y > 0.999f || b.z > 0.999f )
  {
    if( b.x > 0.999f )
      g.x = 1.0f, g.y = g.z = 0.0f;
    else if( b.y > 0.999f )
      g.y = 1.0f, g.x = g.z = 0.0f;
    else
      g.z = 1.0f, g.x = g.y = 0.0f;
  }
  else
  {
    g.x = b0b2/sum;
    g.y = b0b1/sum;
    g.z = b1b2/sum;
  }

// compute parameters for border curves
  if( 1.0f - b.z != 0.0f )
    c.x = b.y / ( 1.0f - b.z );
  else
    c.x = 0.5f;
  if( 1.0f - b.x != 0.0f )
    c.y = b.z / ( 1.0f - b.x );
  else
    c.y = 0.5f;
  if( 1.0f - b.y != 0.0f )
    c.z = b.x / ( 1.0f - b.y );
  else
    c.z = 0.5f;
```

```
  // compute points on border curves
  float3 pos0  = evaluateCurve( v0, v1, n0, n1, c.x );
  float3 pos1  = evaluateCurve( v1, v2, n1, n2, c.y );
  float3 pos2  = evaluateCurve( v2, v0, n2, n0, c.z );

  // compute interpolated normals
  float3 norm0 = normalize( lerp( n0, n1, c.x ) );
  float3 norm1 = normalize( lerp( n1, n2, c.y ) );
  float3 norm2 = normalize( lerp( n2, n0, c2.z ) );

  // evaluate points on each of the three curves
  pos0 = evaluateCurve( pos0, v2, norm0, n2, b.z );
  pos1 = evaluateCurve( pos1, v0, norm1, n0, b.x );
  pos2 = evaluateCurve( pos2, v1, norm2, n1, b.y );

  // face normal based blending factors
  float  fn0 = abs( dot( n, normalize( cross( v2_ - v0 ,
                                              v1  - v0 ) ) ) );
  float  fn1 = abs( dot( n, normalize( cross( v0_ - v1,
                                              v2  - v1 ) ) ) );
  float  fn2 = abs( dot( n, normalize( cross( v2  - v0 ,
                                              v1_ - v0 ) ) ) );

  // evaluate N-Patch
float3 posN  = evaluateNPatch( v0, v1, v2, n0, n1, n2, b );

// blend factor to blend towards N-Patch based on vicinity of the
// center of the tri
  float bl = 3 * 3 * 3 * b.x*b.y*b.z;

// blend each patch towards the N-Patch
  pos0 = lerp( pos0, posN, min( 1, fn0 + bl ) );
  pos1 = lerp( pos1, posN, min( 1, fn1 + bl ) );
  pos2 = lerp( pos2, posN, min( 1, fn2 + bl ) );

  // gregory blend points on curves and blend with N-Patch again
  float3 pos = lerp( g.x * pos2 + g.y * pos0 + g.z * pos1, posN,
  bl );

  // transform
  oPos = mul( float4( pos, 1 ), g_mWorldViewProjection );

  float3 Normal = mul( normalize( b.x * n0 + b.y * n1 + b.z *
  n2 ), (float3x3)g_mWorld );
  Diffuse = saturate( dot( Normal, float3( 0.0f, 0.0f, -1.0f ) ) )
  * g_vDiffuse;
  Diffuse.w = 1.0;

  oTex = b.x * t0 + b.y * t1 + b.z * t2;
}
```

Note that in order to compute face normals of adjacent faces inside the shader, the per-instance data has to be augmented with the vertices of the adjacent triangles. Obviously one only needs to add the vertices that oppose shared edges.

Running this shader on the cube produces the results shown in Figure 1.1.10. It is obvious that the final patch improves the smoothness of the cube.

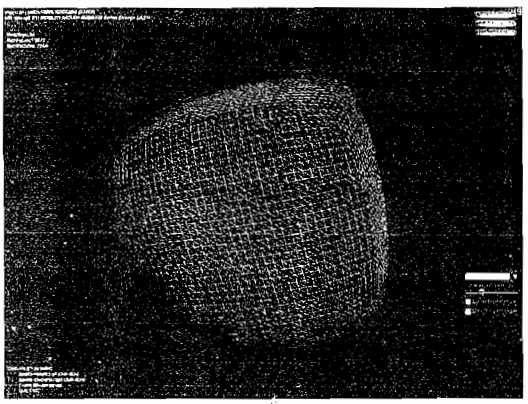

FIGURE 1.1.10A A cube refined using the original N-Patch.

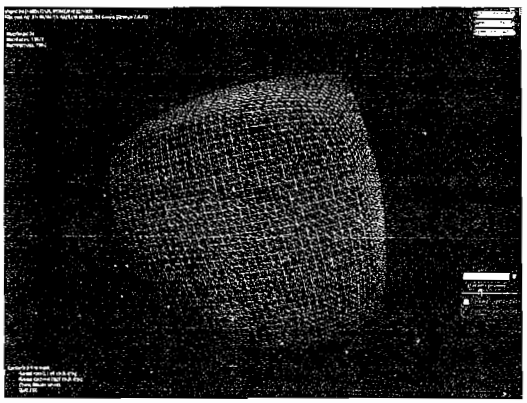

FIGURE 1.1.10B A cube refined using the final new patch.

Now that it has been shown that it is possible to construct a smoother N-Patch, performance considerations and conclusions will be discussed.

Performance Considerations and Conclusions

It is clear that it is more expensive to evaluate the new patches described in this article than it is to evaluate the N-Patch. Luckily enough, the original N-Patch can be mixed with the new patches across an object since the new patches create the same shared border curves that N-Patches create. Depending on the angle between triangles, it may not be necessary to evaluate *P0, P1,* or *P2* but to stick to the original N-Patch. Using per-edge information and appropriate *if()* statements inside the vertex shader can speed up patch evaluation enormously if there is proper hardware support for these statements.

The new patches show improved smoothness when compared to the original N-Patch for low-poly objects. As indicated for N-Patches in [Gruen05] they could also be used as a basis for surface compression.

Even if one concludes that the new patches are too slow to run inside a shader, it is still possible to evaluate them on the CPU and cache the results. Hardware that features several parallel processor cores, in particular, can be utilized easily. This is possible since tessellation scales in a data-parallel way (see [Gruen06]).

Conclusions

Geometry Shaders, which are part of DirectX10, allow geometry generation on the GPU. They can therefore be used to tessellate triangles and to evaluate patches. Interestingly DirectX10 also allows Geometry Shader to access adjacency information for triangles. These features can therefore be easily used to access the additional vertices that the patches proposed here need. Once there is DirectX10-capable hardware, it would be straightforward to implement the smoother N-Patch with Geometry Shaders. Currently the reference software device does not make such an experiment a very pleasurable experience because of speed considerations.

References

[Bolz02] Bolz, Jeffrey and Peter Schröder, "Rapid Evaluation of Catmull-Clark Subdivision Surfaces," February 2002, Proceeding of the Seventh International Conference on 3D Web Technology.

[Cloward] Cloward, Ben, "Creating and Using Normal Maps." Available online at *http://www.monitorstudios.com/bcloward/tutorials_normal_maps1.html.*

[Farin96] Farin, Gerald E., *Curves and Surfaces for Computer-Aided Geometric Design*, Academic Press Inc., 1996.

[Farin99] Farin, Gerald E."*Nurbs*," A K Peters, Ltd., 1999.

[Gruen05] Gruen, Holger, "Efficient Tessellation on the GPU through Instancing," *Journal of Game Development*, *1*(3), December 2005.

[Gruen06] Gruen, Holger, "Multi-threaded Terrain Smoothing." Available online at *www.gamasutra.com*, June 2006.

[Hoppe97] Hoppe, Hugues, "View-Dependent Refinement of Progressive Meshes," ACM SIGGRAPH, 1997, pp, 189–198.

[Lien87] Sheue-Ling, Lien, Michael Shantz, and Vaughan Pratt, "Adaptive Forward Differencing for Rendering Curves and Surfaces," ACM SIGGRAPH Computer Graphics, Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques, *21*(4), August 1987.

[Loviscach03] Loviscach, Jörn, "Silhouette Geometry Shaders," *ShaderX*^{3} *Advanced Rendering with DirectX and OpenGL*, Charles River Media, 2003.

[Sander00] Sander, P., X. Gu, S. Gortler, H. Hoppe, and J. Snyder, "Silhouette Clipping." ACM SIGGRAPH 2000: pp. 327–334.

[Sander01] Sander, P., H. Hoppe, J. Snyder, and S. Gortler, "Discontinuity Edge Overdraw," ACM Symposium on Interactive 3D Graphics 2001: pp. 167–174.

[Vlachos01] Vlachos, Alex, Jörg Peters, Chas Boyd, and Jason L. Mitchell, "Curved PN Triangles," *Proceedings of the 2001 Symposium on Interactive 3D Graphics*, March 2001.