CRACK-FREE POINT-NORMAL TRIANGLES USING ADJACENT EDGE NORMALS

Crack-Free Procedural Tessellation

DOCUMENT CHANGE HISTORY

WP-05621-001_v01

Version	Date	Authors	Description of Change
01	December 13, 2010	John McDonald, Mark Kilgard	Initial release

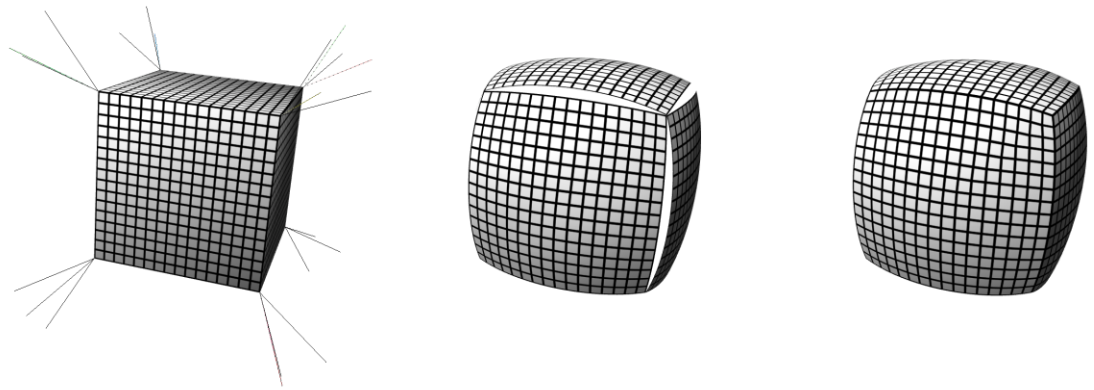
TABLE OF CONTENTS

LIST OF FIGURES

OVERVIEW

Crack-Free Point-Normal Triangles using Adjacent Edge Normals (PN-AEN) is an approach to real-time tessellation on existing hardware that produces crack-free meshes without additional artist involvement. The solution uses existing vertex buffer data along with index buffer data that can be easily computed from index buffers used by existing triangle-based meshes, such as those found in real-time rendering applications (e.g. games).

This approach extends PN triangles, originally proposed by Vlachos, et al, in the paper *Curved PN Triangles*.



A source cube with divergent normals on each face (left), the resulting PN-Triangles Tessellation (middle), and the resulting crack-free Tessellation (right).

Figure 1. Tessellation Comparison

HARDWARE TESSELLATION INTRODUCTION

As this may be the first exposure to hardware tessellation for many readers, a brief overview is provided. Readers familiar with hardware tessellation can safely skip this section.

TESSELLATION OVERVIEW

Programmable tessellation is a new feature of Direct3D 11 and OpenGL 4.0, and allows the hardware to create contiguous geometry on-GPU. This allows for techniques that were previously infeasible due to memory or complexity requirements.

Tessellation modifies the conceptual pipeline by adding three new stages, shown in Figure 2:

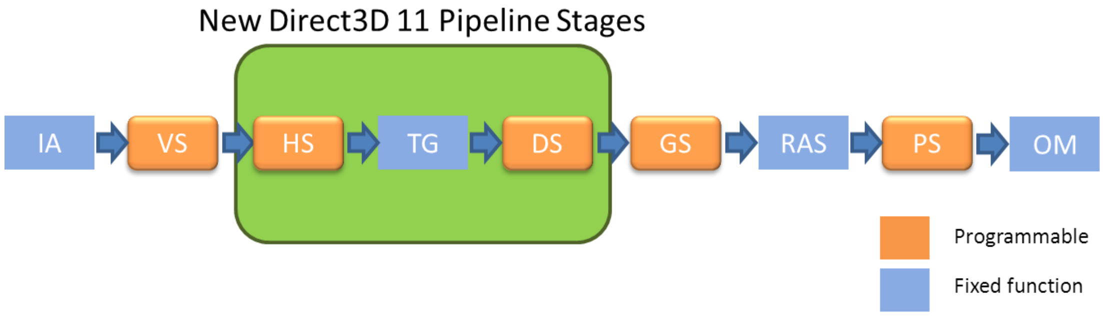


Figure 2. Direct3D 11 Pipeline Modification

Tessellation requires three things to happen:

- Control Point Basis Transformation
  Converting an input surface basis (parameterized by an ordered set of control points) into a simpler to evaluate surface basis.
- Patch Geometry Generation
  Generating geometry (lines, triangles, quads) from specified level-of-detail (LOD) parameters.
- Vertex Shading
  Transforming vertices from patch space to clip space.

In the implementation exposed by Direct3D 11, these tasks roughly correspond to the three new stages.

IN DEPTH

New Primitive Topologies

New primitive topology types for tessellated patches are included in Direct3D 11, which allow you to specify patches using between 1 and 32 control points per patch. The term "Control Point" is used to refer to a vertex that is input to tessellation, while when discussing tessellation a vertex is typically thought of as the result of the tessellation.

The new topology types follow the form `D3D11_PRIMITIVE_TOPOLOGY_X_CONTROL_POINT_PATCHLIST`, where X indicates the number of control points per patch.

> **Note:** When using new index buffer types with tessellation disabled, the first three indices of each patch will be interpreted directly as a triangle list. For example, if using the index buffer type PATCH_9 with the input data { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18 }: if the hull and domain shader are set to NULL, two triangles would be drawn: {0, 1, 2} and {9, 10, 11}.

Hull Shader

The hull shader is responsible for these tasks:

- Converts the control points of the input patch basis to the control points of the output patch basis. A basis is a mathematical function, parameterized by control points, which describes a surface.

  The goal is to produce outputs useful for post-tessellation Vertex Shading (post-tessellation). This process is completely application defined.

- Assigns LOD parameters for usage during Geometry Generation.

  In the case of triangle tessellation, one factor must be set for each edge as well as one factor for the interior of the triangle. In order to produce crack-free meshes, one must ensure that the LOD values for each edge match up exactly. The safest way to do this is to only use values available to both patches (ie, only shared data along the edge).

- Programs the TG stage parameters:
  - Output primitive type
  - Triangle winding specification
  - Geometry generation method ("fractional_even", "fractional_odd", "int", "pow2")

  However, these parameters are all set at shader creation time via attributes, not at shader execution time.

- Optionally performs application-defined frustum culling.

  Any patch with an EdgeLOD of zero will be culled. If the Hull Shader doesn't perform culling, geometry can be generated outside of the frustum, leading to performance degradation.

The Hull shader operates in two phases. The first phase, Hull Control Point shading, executes in *single-program-multiple-data* (SPMD), where a distinct thread executes for each output control point. One can specify the number of threads that run in parallel during this phase by modifying the "outputcontrolpoints" shader attribute.

Upon completion of all of the Hull Control Point threads belonging to a single patch, the hardware runs the Hull Patch Constant Function as a single thread. Your Patch Constant Function may read the results of the Hull Control Point Functions by specifying an input parameter of type const `OutputPatch<ControlPointOutput, OUTPUT_CONTROL_POINTS>`. The control points cannot be modified during this stage.

The Hull Patch Constant Function is responsible for setting the LOD parameters, which will be used by the Tessellation Generator (TG) to generate new geometry.

Note: The ordering of SV_TessFactor is described by language uncommonly seen in triangle rasterization. SV_TessFactor must be correctly specified or cracks between patches will result due to the mismatch of LOD between edges.

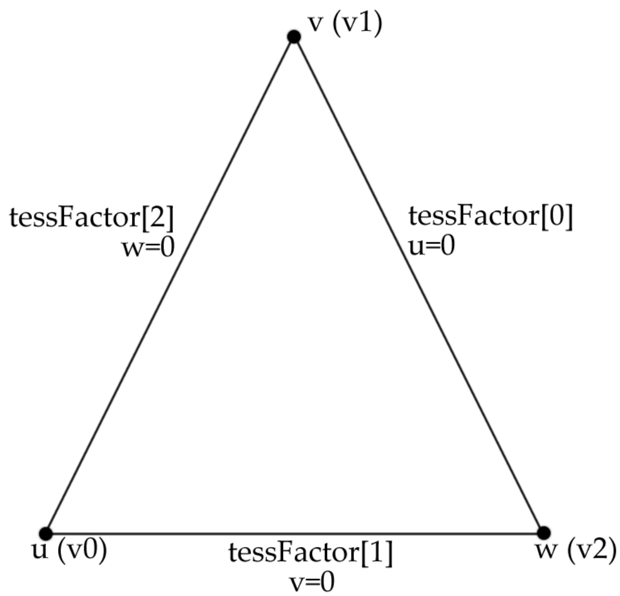


Figure 3. Tessellation Factor specification

Tessellation Factors are specified as the edge opposite the N[th] vertex for tessellation in the triangle domain.

The Patch Constant Shader can additionally output a set of constants for use during Domain Shading.

Tessellation Generator (TG)

TG is a fixed-function unit, run upon completion of the Hull Patch Constant Function. It uses the information specified above (Output Primitive Type, Forward Winding Specification, Geometry Generation Method) to produce geometry in *Patch Space*. *Patch Space* is a space that is specifically defined for each primitive type. The values output are guaranteed to have bitwise accuracy even across patch edges, as long as the input LOD values are identical for neighboring edges. Thus, proper Edge LOD specification is critically important in producing crack-free meshes.

For example, when outputting triangles, TG will output a set of vertices using Barycentric Coordinates. When outputting quads, TG will output a set of UV coordinates.

Domain Shader

After TG has generated geometry according to compile-time Hull Shader parameters—along with runtime-specified LOD parameters—the Domain Shader will begin to run on the newly created vertices. The domain shader typically takes three items as input, an `InputPatch<ControlPointOutput, OUTPUT_CONTROL_POINTS>`, an optional PatchConstant (output by the Patch Constant portion of the Hull Shader), and a parameter with the semantic SV_DomainLocation, which corresponds to the location in patch space that is currently being shaded.

The Domain Shader combines all three pieces of information to convert vertices from Patch Space to Clip (or Projection) space.

EFFICIENT WORK PARTITIONING

Now that the basics of tessellation have been covered, it's worthwhile to discuss how to efficiently partition tessellation workloads. As with the standard pipeline, work should be performed at the lowest frequency possible. This means that it is often worth performing slightly more complex work in an earlier stage to allow for simpler work at a later stage.

Here's a specific breakdown that seems to work well for numerous tessellation techniques:

- **Vertex Shader:** Transform control points, normals, etc to camera space
- **Hull Shader (Control Point):** Compute Control Point locations, copy per-vertex data to output control points. If possible, compute per-control point culling information.
- **Hull Shader (Patch Constant):** Use data from HS-Control Point to compute per-edge LOD as well as interior LOD. Determine whether patch is outside of view frustum (set edge-LOD values to 0 if so).
- **Domain Shader:** Perform tessellated multiply (patch->eye space) by multiplying tessellation equation with control points and patch space location. Transform from eye space to clip space.

This breakdown has several advantages:

- Transforming to eye space in the Vertex Shader, instead of the Hull Shader, will utilize the Vertex Shader cache, which leverages optimized mesh layouts to reduce the total amount of geometry being processed by the Vertex Shader.
- Culling geometry entirely outside of the view frustum saves significant geometry processing when a model is close to the camera.
- Transforming from eye space to projected space in the domain shader can utilize an optimized version of the projection transform which uses only 5 MADs instead of a full 4x4 matrix transform per-vertex (on heavily magnified geometry).

CURVED POINT-NORMAL TRIANGLES WITH EDGE-ADJACENT NORMALS

PREVIOUS WORK AND OVERVIEW

In the paper Curved PN Triangles, Vlachos, et al, prove that with purely local information and non-C1 source data, it is impossible to avoid cracking in the output mesh.

> **Note:** The C(number) notation refers to the continuity of derivatives of a surface. A surface is said to be C0 if it is crack-free (positions are continuous throughout the mesh, no T-junctions, etc). A surface is said to be C1 if every position on the surface has one and only one normal (no discontinuities in surface normals).

PN-AEN solves this problem by providing neighbor information during the Hull Shading phase (while determining the Bezier control points of the surface) so that the control points calculated along either side of a mesh edge can line up.

This requires:

- Performing a preprocess step that matches triangle edges to neighbor triangle edges, and builds a custom index buffer that communicates these relationships.
- Writing a hull shader that averages the values computed by this triangle as well as the adjacent triangle for each edge, resulting in control point selection that always agrees across triangle edges—irrespective of whether normals are continuous across the edge or not.

In this way, PN-AEN is able to produce crack free meshes for source material that is C0, compared to the PN Triangles requirement of C1.

Practical Usage

PN-AEN is suitable for usage in real-time graphics applications (such as games) where triangle meshes (including position and normal) already exist, and when additional artist time cannot be spared. PN-AEN can be used to improve silhouette and shadowing quality without a corresponding reduction in performance.

The following figures show two different models both with and without the usage of this technique. The left image is the original untessellated mesh, while the right image is tessellated using PN-AEN.

Figure 4. Wretch, used courtesy of Epic Games, Inc.

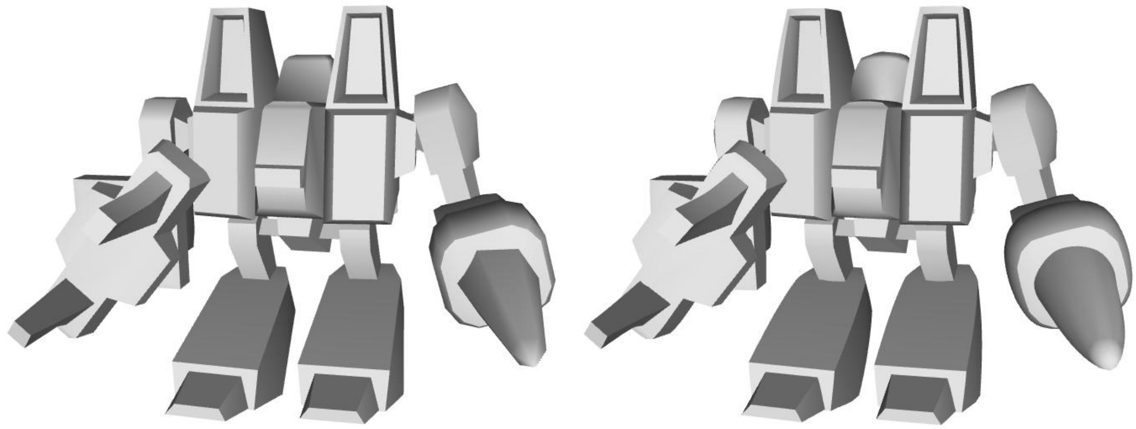


Figure 5. SCV, used courtesy of Blizzard Entertainment

IMPLEMENTATION

The PN-AEN Triangle Index Buffer

PN-AEN uses the 9-control point per-patch primitive type. By convention, the first 3 indices specify the triangle as though it were being specified in a TRIANGLELIST. This actually allows the same buffer to be used to draw non-tessellated geometry by specifying null Hull and Domain Shaders.

The remaining indices correspond to the neighboring vertex for each edge in the triangle. This is more clearly demonstrated in Figure 6. Figure 6.

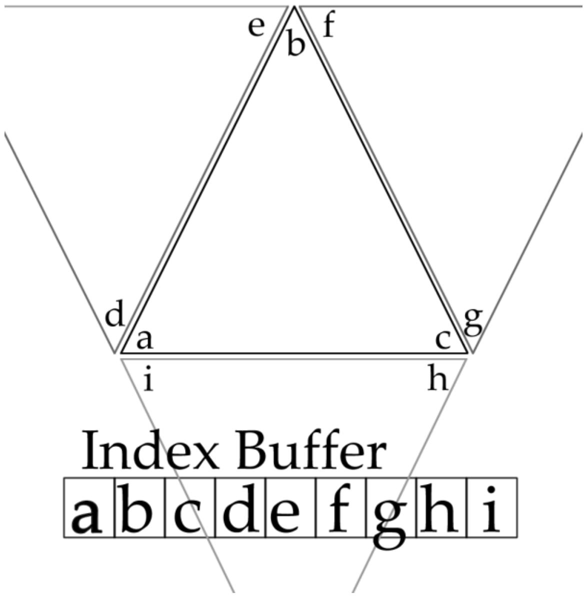


Figure 6. Index Buffer for a Single Patch in PN-AEN

Computing the PN-AEN Triangle Index Buffer

A pre-process step is used to compute the PN-AEN Triangle Index Buffer. A reference implementation of this algorithm can be obtained by contacting NVIDIA's Developer Support.

The algorithm is linear in the number of triangles, where the constant factor depends on the storage method used. It builds the adjacent edge information needed for PN-AEN as follows, given a source Index Buffer (IB) as a TRIANGLELIST and Vertex Buffer (VB):

1. Create an output IB that is 3 times the size of input IB.
2. For each input Triangle in IB, with indices i0, i1 and i2:
   a. Write out an initial output entry of: i0, i1, i2, i0, i1, i1, i2, i2, i0, which sets edges to initially be neighbors of themselves. This would produce identical results to PN Triangles.
   b. Lookup the positions p0, p1, and p2, using i0, i1 and i2 to perform a lookup for position of the associated vertex in VB.
   c. Define 3 Edges, which consist of the two indices and two positions that make up the corresponding Edge. An Edge should consist of the origin index, the destination index, the origin position and the destination position.
   d. For each edge, store the reverse of that edge in an easily searchable data structure for the next step. The reference implementation uses an stdext::hash_map<Edge, Edge> for this purpose. *Reverse* simply flips the sense of the edge (originating at the destination position and index and heading to the origin position and index).

3. Walk the output index buffer (OB) constructed in step 2. For each patch of 9 indices:
   a. For each Edge in the current Patch, perform a lookup into Edge->Edge mapping created in step 2d.
   b. If found, replace the current indices with the indices found in the map. Note that two edges should be considered matching if their "from" and "to" indices match, OR if their "from" and "to" positions match.
   c. If not, continue to use the existing indices.

Upon completion of this algorithm, a buffer suitable for usage with PN-AEN will be available.

> **Note:** One-third of the data in a PN-AEN index buffers is identical to the information stored in a TRIANGLELIST index buffer. Therefore, it may be advantageous to store the "normal" index buffer data in one file and store only the data unique to PN-AEN in a different file—and to interleave them together at asset load time when tessellation is enabled.

PN-AEN Shaders

PN-AEN uses a Domain Shader that is identical to the Domain Shader used for PN Triangles—the difference is entirely in Hull Shading (the computation of Bezier Control Points).

In order to provide SIMD-efficient code, each hull shader control point program operates on a single edge at a time. The values output from each thread are described in Figure 7. In this figure, the ovals indicate the values worked out by a single thread. The square in the middle is computed by the Patch Constant portion of the Hull Shader.

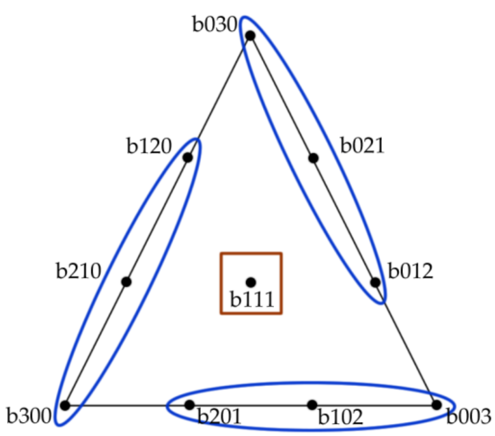


Figure 7. Hull Shader Thread Breakdown

A well-optimized reference implementation is provided beginning on the next page. For reference, a similarly optimized implementation of PN Triangles is provided in the Appendix, along with complete Vertex and Domain Shaders used with either PN or PN-AEN Triangles.

```
// The Patch Control Point portion of the Hull Shader.
[domain("tri")]
[partitioning(PARTITION_METHOD)]
[outputtopology("triangle_cw")]
[patchconstantfunc("HS_Constant")]
[outputcontrolpoints(3)]
HS_ControlPointOutput HS_PNAENTriangles(
    InputPatch<HS_RenderSceneInput, 9> I,
    uint uCPID : SV_OutputControlPointID
)
{
    HS_ControlPointOutput O = (HS_ControlPointOutput)0;

    // The PN-AEN Index buffer provides access to the neighbor across
    // the edge of the triangle. Compute where
    // they are here.
    const uint NextCPID = uCPID < 2 ? uCPID + 1 : 0; // (uCPID + 1) % 3
    const uint AdditionalData = 3 + 2 * uCPID;
    const uint NextAdditionalData = AdditionalData + 1;
    float3 myCP, otherCP;

    // Copies first.
    O.f3ViewPosition[0] = I[uCPID].f3ViewPosition;
    O.f3WorldNormal     = I[uCPID].f3WorldNormal;
    O.f2TexCoord        = I[uCPID].f2TexCoord;

    // Calculate control points next. To compute a crack-free control
    // point, we average the control point we'd like with the
    // control point our neighbor would like. The result is that we
    // both agree on where that control point should go--and that
    // results in crack-free tessellation!
    // This is the only difference between PN and PN-AEN tessellation,
    // made possible by modern programmable tessellation hardware.
    myCP = ComputeCP(I[uCPID].f3ViewPosition,
                     I[NextCPID].f3ViewPosition,
                     I[uCPID].f3ViewNormal);

    otherCP = ComputeCP(I[AdditionalData].f3ViewPosition,
                        I[NextAdditionalData].f3ViewPosition,
                        I[AdditionalData].f3ViewNormal);

    O.f3ViewPosition[1] = (myCP + otherCP) / 2;

    myCP = ComputeCP(I[NextCPID].f3ViewPosition,
                     I[uCPID].f3ViewPosition,
                     I[NextCPID].f3ViewNormal);

    otherCP = ComputeCP(I[NextAdditionalData].f3ViewPosition,
                        I[AdditionalData].f3ViewPosition,
```

```
                            I[NextAdditionalData].f3ViewNormal);

    O.f3ViewPosition[2] = (myCP + otherCP) / 2;


    // Note: We're relying on the optimizer to avoid projecting to
    // projection space twice. Probably better to be explicit,
    // but this is a bit clearer.
    if (g_clipping.x) {
        O.fClipped = ComputeClipping(g_f4x4Projection,
                                     O.f3ViewPosition[0],
                                     O.f3ViewPosition[1],
                                     O.f3ViewPosition[2]);
    } else {
        O.fClipped = 0.0f;
    }

    // Perform Adaptive Tessellation step here.
    if (g_adaptive.x) {
        O.fOppositeEdgeLOD = ComputeEdgeLOD(g_f4x4Projection,
                                            O.f3ViewPosition[0],
                                            O.f3ViewPosition[1],
                                            O.f3ViewPosition[2],
                                           I[NextCPID].f3ViewPosition);
    } else {
        O.fOppositeEdgeLOD = g_f4TessFactors.x;
    }

    return O;
}

// The Hull Shader Constant function, which is run after all threads
// of the Hull Shader Control Point function (above) complete.
HS_ConstantOutput HS_Constant(
    const OutputPatch<HS_ControlPointOutput, OUT_PATCH_SIZE> I
)
{
    HS_ConstantOutput O = (HS_ConstantOutput)0;

    // These were computed during the Control Point phase of either PN
    // or PN-AEN Triangles.
    // We're just aliasing them to better match our functionality with
    // the reference implementation.
    float3 f3B300 = I[0].f3ViewPosition[0],
           f3B210 = I[0].f3ViewPosition[1],
           f3B120 = I[0].f3ViewPosition[2],
           f3B030 = I[1].f3ViewPosition[0],
           f3B021 = I[1].f3ViewPosition[1],
           f3B012 = I[1].f3ViewPosition[2],
           f3B003 = I[2].f3ViewPosition[0],
           f3B102 = I[2].f3ViewPosition[1],
           f3B201 = I[2].f3ViewPosition[2];


    // The tessellation factors map up in a somewhat surprising way.
    // Specifically, if you think of a triangle with indices 0, 1 and
    // 2, the LOD values map to the edge that is opposite the index.
```

```
    // That is to say that TessFactor[0] is the LOD for edge (1,2).
    // Here's the complete table:
    // TessFactor[0] => Edge(1, 2)
    // TessFactor[1] => Edge(2, 0)
    // TessFactor[2] => Edge(0, 1)
    O.fTessFactor[0] = I[1].fOppositeEdgeLOD;
    O.fTessFactor[1] = I[2].fOppositeEdgeLOD;
    O.fTessFactor[2] = I[0].fOppositeEdgeLOD;

    // There's no right or wrong answer here. We've chosen to say that
    // the interior should be at least as tessellated as the most
    // tessellated exterior edge.
    O.fInsideTessFactor[0] = max(max(O.fTessFactor[0],
                                     O.fTessFactor[1]),
                                 O.fTessFactor[2]);

    // Center control point
    float3 f3E = (f3B210 + f3B120 + f3B021
                  + f3B012 + f3B102 + f3B201) / 6.0f;
    float3 f3V = (f3B003 + f3B030 + f3B300) / 3.0f;
    O.f3ViewB111 = f3E + ((f3E - f3V) / 2.0f);

    // Determine whether the center control point is visible or not.
    float fB111Clipped =
            IsClipped(ApplyProjection(g_f4x4Projection, O.f3ViewB111));

    if (I[0].fClipped && I[1].fClipped
        && I[2].fClipped && fB111Clipped) {
        // If all control points are clipped, the surface is
        // almost certainly not visible.
        O.fTessFactor[0] = O.fTessFactor[1] = O.fTessFactor[2] = 0;
    }

    return O;
}
```

MESHES WITH NEGATIVE CURVATURE

While PN-AEN can produce crack free meshes from any C0 input mesh, there is still a minor caveat. Meshes have normals that point inwards--indicating that the surface is concave—can result in some mesh inter-penetration near the edges. While minor, this is an issue to be aware of. This is more obviously illustrated in Figure 8.

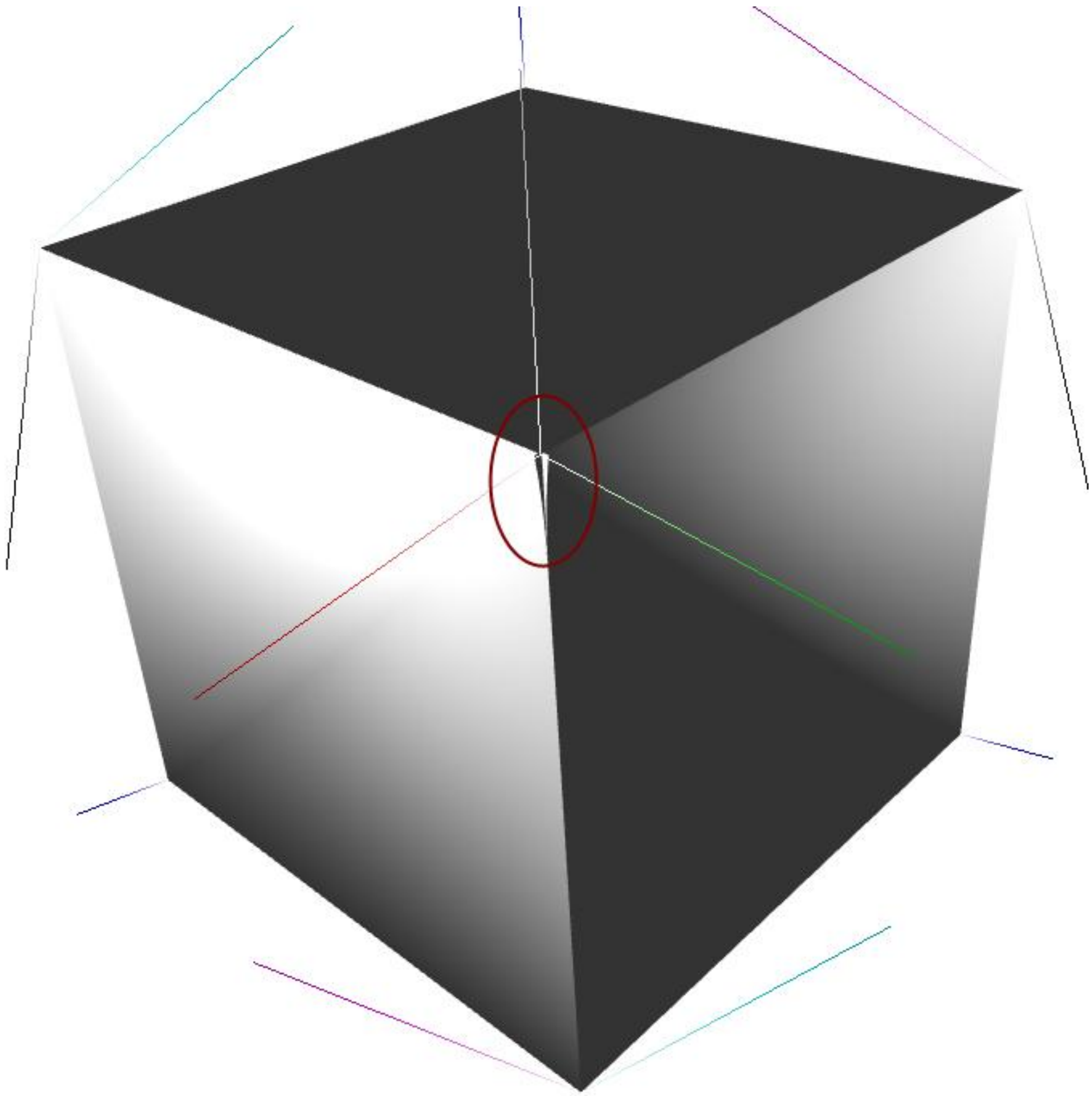


Figure 8 Concave Curvature

SUMMARY

PN-AEN is a novel improvement on PN Triangles, producing crack-free tessellation from a wider variety of input meshes and preserving hard edges when disjoint normals occur.

As with PN Triangles, PN-AEN is able to produce surfaces with significantly higher curvature than the memory footprint would imply without requiring additional artist intervention.

Additionally, we've provided the reader with an overview of hardware tessellation on modern GPUs, including an efficient, general work breakdown.

APPENDIX

COMPLETE SOURCE LISTING

```
#ifndef PARTITION_METHOD
#define PARTITION_METHOD "fractional_odd"
#endif

#define IN_PN_PATCH_SIZE 3
#define IN_KM_PATCH_SIZE 9
#define OUT_PATCH_SIZE 3

#define FAST_PROJECTION_XFORM 1

// Constant buffer
cbuffer cbPNTriangles : register( b0 )
{
    float4x4    g_f4x4World;                // World matrix for object
    float4x4    g_f4x4WorldView;            // World * View matrix
    float4x4    g_f4x4WorldViewProjection;  // World * View *
                                            // Projection matrix
    float4x4    g_f4x4ViewProjection;       // View * Projection matrix
    float4x4    g_f4x4Projection;           // Projection matrix
    float4      g_f4LightDir;               // Light direction vector
    float4      g_f4Eye;                    // Eye
    float4      g_f4TessFactors;            // Tessellation factors
                                            // x=Edge
    float4      g_f4ViewportScale;          // The X and Y half
                                            // resolution, 0, 0
    bool4       g_adaptive;                 // Should use adaptive
                                            // tessellation
    bool4       g_clipping;                 // Should run clipping
                                            // tests.
}

// Some global lighting constants
static float4 g_f4MaterialDiffuseColor  = float4( 1.0f, 1.0f, 1.0f,
1.0f );
```

```
static float4 g_f4LightDiffuse          = float4( 1.0f, 1.0f, 1.0f,
1.0f );
static float4 g_f4MaterialAmbientColor  = float4( 0.2f, 0.2f, 0.2f,
1.0f );


// Textures
Texture2D g_txDiffuse        : register( t0 );

// Samplers
SamplerState        g_SampleAniso  : register( s0 );

// Shader structures
struct VS_RenderSceneInput
{
    float3 f3Position   : POSITION;
    float3 f3Normal     : NORMAL;
    float2 f2TexCoord   : TEXCOORD;
};

struct HS_RenderSceneInput
{
    float3 f3ViewPosition   : POSITION;
    float3 f3WorldNormal    : NORMAL;
    float3 f3ViewNormal     : NORMAL1;
    float2 f2TexCoord       : TEXCOORD;
};

struct HS_ConstantOutput
{
    float fTessFactor[3]        : SV_TessFactor;
    float fInsideTessFactor[1]  : SV_InsideTessFactor;

    float3 f3ViewB111           : POSITION9;

};

struct HS_ControlPointOutput
{
    float3 f3ViewPosition[3]    : POSITION;
    float3 f3WorldNormal        : NORMAL;
    float2 f2TexCoord           : TEXCOORD;
    float fOppositeEdgeLOD      : LODDATA;
    float fClipped              : CLIPPED;  // 1.0 means clipped,
                                            // 0.0 means unclipped.
};

struct DS_Output
{
    float4 f4Position   : SV_Position;
    float2 f2TexCoord   : TEXCOORD0;
    float4 f4Diffuse    : COLOR;
};

struct PS_RenderSceneInput
{
    float4 f4Position   : SV_Position;
```

```
    float2 f2TexCoord    : TEXCOORD0;
    float4 f4Diffuse     : COLOR;
};

struct PS_RenderOutput
{
    float4 f4Color       : SV_Target0;
};

float3 ComputeCP(float3 posA, float3 posB, float3 normA) {
    return (2 * posA + posB - (dot((posB - posA), normA) * normA)) /
3.0f;
}

// Expects that projMatrix is the canonical projection matrix. Will be
// faster than performing a full 4x4 matrix multiply by an eye space
// position in that case.
float4 ApplyProjection(float4x4 projMatrix, float3 eyePosition)
{
    float4 clipPos;
#if FAST_PROJECTION_XFORM
    // In the canonical projection matrix, all other elements are zero
    // and eyePosition[3] == 1.
    clipPos[0] = projMatrix[0][0] * eyePosition[0];
    clipPos[1] = projMatrix[1][1] * eyePosition[1];
    clipPos[2] = projMatrix[2][2] * eyePosition[2] + projMatrix[3][2];
    clipPos[3] = eyePosition[2];
#else
    clipPos = mul(projMatrix, float4(eyePosition, 1));
#endif

    return clipPos;
}

// This will project the input eye-space position by the specified
// matrix, then compute an incorrect (but properly scaled) window
// position. Finally, we divide by the tessellation factor,
// which is approximately how many pixels we want per-triangle.
float2 ProjectAndScale(float4x4 projMatrix, float3 inPos)
{
    float4 posClip = ApplyProjection(projMatrix, inPos);
    float2 posNDC = posClip.xy / posClip.w;
    return posNDC * g_f4ViewportScale.xy / g_f4TessFactors.z;
}

float IsClipped(float4 clipPos)
{
    // Test whether the position is entirely inside the view frustum.
    return (-clipPos.w <= clipPos.x && clipPos.x <= clipPos.w
         && -clipPos.w <= clipPos.y && clipPos.y <= clipPos.w
         && -clipPos.w <= clipPos.z && clipPos.z <= clipPos.w)
       ? 0.0f
       : 1.0f;
}

// Compute whether all three control points along the edge are outside
// of the view frustum.
```

```
float ComputeClipping(float4x4 projMatrix, float3 cpA, float3 cpB,
float3 cpC)
{
    // Compute the projected position for each position, then check to
    // see whether they are clipped.
    float4 projPosA = ApplyProjection(projMatrix, cpA),
           projPosB = ApplyProjection(projMatrix, cpB),
           projPosC = ApplyProjection(projMatrix, cpC);

    return min(min(IsClipped(projPosA), IsClipped(projPosB)),
               IsClipped(projPosC));
}

// Compute the edge LOD for the four specifed control points, which
// should be the control points along one edge of the triangle. This is
// significantly more accurate than just using the end points of the
// triangle because it takes curvature into account. Note that will
// overestimate the number of triangles needed, but typically not by
// too much. It also ensures that we never cull a triangle by ensuring
// that the LOD is at least 1.
float ComputeEdgeLOD(float4x4 projMatrix,
                     float3 cpA, float3 cpB, float3 cpC, float3 cpD)
{
    float2 projCpA = ProjectAndScale(projMatrix, cpA).xy,
           projCpB = ProjectAndScale(projMatrix, cpB).xy,
           projCpC = ProjectAndScale(projMatrix, cpC).xy,
           projCpD = ProjectAndScale(projMatrix, cpD).xy;

    float edgeLOD = distance(projCpA, projCpB)
                  + distance(projCpB, projCpC)
                  + distance(projCpC, projCpD);

    return max(edgeLOD, 1);
}

// This vertex shader transforms the input data to view space for
// optimized versions of tessellation shaders to use.
HS_RenderSceneInput VS_RenderSceneWithTessellation(
                                     VS_RenderSceneInput I
)
{
    HS_RenderSceneInput O = (HS_RenderSceneInput)0;

    O.f3ViewPosition = float3(mul(float4(I.f3Position, 1),
                                  g_f4x4WorldView).xyz);
    O.f3WorldNormal = mul(I.f3Normal, (float3x3)g_f4x4World);
    O.f3ViewNormal = mul(I.f3Normal, (float3x3)g_f4x4WorldView);
    O.f2TexCoord = I.f2TexCoord;

    return O;
}



[domain("tri")]
[partitioning(PARTITION_METHOD)]
```

```
[outputtopology("triangle_cw")]
[patchconstantfunc("HS_Constant")]
[outputcontrolpoints(OUT_PATCH_SIZE)]
HS_ControlPointOutput HS_PNTriangles( InputPatch<HS_RenderSceneInput,
IN_PN_PATCH_SIZE> I, uint uCPID : SV_OutputControlPointID )
{
    HS_ControlPointOutput O = (HS_ControlPointOutput)0;

    // Looking at HS_ControlPointOutput, we can see that we're really
    // outputing 3 control point positions
    // per real control point. Structuring the code this way allows us
    // to write efficient SIMD code,
    // because each thread can compute 3 control point values in
    // parallel.

    // Compute the next output control point ID so we know which edge
    // we're on.
    const uint NextCPID = uCPID < 2 ? uCPID + 1 : 0; // (uCPID + 1) % 3
    float3 myCP, otherCP;

    // Copies first.
    O.f3ViewPosition[0] = I[uCPID].f3ViewPosition;
    O.f3WorldNormal     = I[uCPID].f3WorldNormal;
    O.f2TexCoord        = I[uCPID].f2TexCoord;

    // Calculate control points next. The math in ComputeCP is the math
    // specified in Curved PN Triangles by Vlachos, et al.
    O.f3ViewPosition[1] = ComputeCP(I[uCPID].f3ViewPosition,
                                    I[NextCPID].f3ViewPosition,
                                    I[uCPID].f3ViewNormal);
    O.f3ViewPosition[2] = ComputeCP(I[NextCPID].f3ViewPosition,
                                    I[uCPID].f3ViewPosition,
                                    I[NextCPID].f3ViewNormal);

    // Note: We're relying on the optimizer to avoid projecting to
    // projection space twice. Probably better to be explicit,
    // but this is a bit clearer.
    if (g_clipping.x) {
        O.fClipped = ComputeClipping(g_f4x4Projection,
                                     O.f3ViewPosition[0],
                                     O.f3ViewPosition[1],
                                     O.f3ViewPosition[2]);
    } else {
        O.fClipped = 0.0f;
    }

    if (g_adaptive.x) {
        // Compute our adaptive tessellation factor based on each edge.
        // A few things worth noting:
        // In order to remain micro-crack free, neighboring patches
        // must compute the exact same LOD value for matching edges. To
        // ensure this, the safest method is to only use values that
        // are available to both
        // sides of the edge.
        // Additionally, we're computing LOD based on the view
        // projection, making this view-dependent.
        // We could also (for example), make this light-dependent, or
```

```
        // some combination of the two by using another matrix
        // and summing or maxing our results. The method we're using
        // will generate a bit more tessellation for an edge than is
        // really required, but will look correct when viewed edge-on,
        // and is why this method was chosen.
        O.fOppositeEdgeLOD = ComputeEdgeLOD(g_f4x4Projection,
                                            O.f3ViewPosition[0],
                                            O.f3ViewPosition[1],
                                            O.f3ViewPosition[2],
                                           I[NextCPID].f3ViewPosition);
    } else {
        O.fOppositeEdgeLOD = g_f4TessFactors.x;
    }

    return O;
}

// Our Domain Shader
[domain("tri")]
DS_Output DS_Shared( HS_ConstantOutput cdata,
                     const OutputPatch<HS_ControlPointOutput, 3> I,
                     float3 f3BarycentricCoords : SV_DomainLocation )
{
    DS_Output O = (DS_Output)0;
    // The barycentric coordinates
    float fU = f3BarycentricCoords.x;
    float fV = f3BarycentricCoords.y;
    float fW = f3BarycentricCoords.z;

    // Precompute squares and squares * 3
    float fUU = fU * fU;
    float fVV = fV * fV;
    float fWW = fW * fW;
    float fUU3 = fUU * 3.0f;
    float fVV3 = fVV * 3.0f;
    float fWW3 = fWW * 3.0f;

    // Although complicated, this is the canonical implementation of
    // PN, as per Vlachos, et al.
    float3 f3EyePosition = I[0].f3ViewPosition[0] * fUU * fU +
                           I[1].f3ViewPosition[0] * fVV * fV +
                           I[2].f3ViewPosition[0] * fWW * fW +

                           I[0].f3ViewPosition[1] * fUU3 * fV +
                           I[0].f3ViewPosition[2] * fVV3 * fU +

                           I[1].f3ViewPosition[1] * fVV3 * fW +
                           I[1].f3ViewPosition[2] * fWW3 * fV +

                           I[2].f3ViewPosition[1] * fWW3 * fU +
                           I[2].f3ViewPosition[2] * fUU3 * fW +

                           cdata.f3ViewB111 * 6.0f * fW * fU * fV;

    // After computing our eye position, apply projection to compute
    // the clip space position.  With tessellation enabled, you can
    // think of the bottom of the Domain Shader as being equivalent
```

```
    // to the bottom of the Vertex Shader when tessellation is
    // disabled.
    float4 f4ClipPosition = ApplyProjection(g_f4x4Projection,
                                            f3EyePosition);

    // In the canonical PN implementation, quadratic normals are
    // computed. However, this introduces high frequency lighting noise
    // in meshes with normals that point in the same direction but are
    // not perpendicular to the triangle surface. Moreover, quadtratic
    // normals in the face of normal maps would actually also require
    // per-pixel quadratic tangents and bitangents.
    float3 f3Normal = I[0].f3WorldNormal * fU
                    + I[1].f3WorldNormal * fV
                    + I[2].f3WorldNormal * fW;

    // Normalize the interpolated normal
    f3Normal = normalize( f3Normal );

    // Linearly interpolate the texture coords
    O.f2TexCoord = I[0].f2TexCoord * fU
                 + I[1].f2TexCoord * fV
                 + I[2].f2TexCoord * fW;

    // min(I[0].fOppositeEdgeLOD, 0) will evaluate to 0 always, but the
    // compiler cannot figure that out and fOppositeEdgeLOD (and
    // fClipped) need to be used in the domain shader to avoid being
    // compiled out of the output struct.
    // This code is only here to work around a compiler bug and will be
    // removed in a future version.
    float bogusCompilerWAR = min(I[0].fOppositeEdgeLOD, 0)
                           * I[0].fClipped;

    O.f2TexCoord.x += bogusCompilerWAR;

    // Calc diffuse color
    O.f4Diffuse = g_f4MaterialDiffuseColor * g_f4LightDiffuse
                * max( 0, dot( f3Normal, g_f4LightDir.xyz ) )
                + g_f4MaterialAmbientColor; //

    O.f4Diffuse.a = 1.0f;

    // Transform model position with view-projection matrix
    O.f4Position = f4ClipPosition;

    return O;
}
```

```
// Pixel Shader
PS_RenderOutput PS_RenderScene( PS_RenderSceneInput I )
{
    PS_RenderOutput O;

    O.f4Color = g_txDiffuse.Sample( g_SampleAniso, I.f2TexCoord )
              * I.f4Diffuse;

    return O;
}
```

SPECIAL THANKS

Special thanks to Blizzard Entertainment and Epic Games for allowing the usage of their assets, both in testing PN-AEN as well as in this whitepaper.

Notice

ALL NVIDIA DESIGN SPECIFICATIONS, REFERENCE BOARDS, FILES, DRAWINGS, DIAGNOSTICS, LISTS, AND OTHER DOCUMENTS (TOGETHER AND SEPARATELY, "MATERIALS") ARE BEING PROVIDED "AS IS." NVIDIA MAKES NO WARRANTIES, EXPRESSED, IMPLIED, STATUTORY, OR OTHERWISE WITH RESPECT TO THE MATERIALS, AND EXPRESSLY DISCLAIMS ALL IMPLIED WARRANTIES OF NONINFRINGEMENT, MERCHANTABILITY, AND FITNESS FOR A PARTICULAR PURPOSE.

Information furnished is believed to be accurate and reliable. However, NVIDIA Corporation assumes no responsibility for the consequences of use of such information or for any infringement of patents or other rights of third parties that may result from its use. No license is granted by implication of otherwise under any patent rights of NVIDIA Corporation. Specifications mentioned in this publication are subject to change without notice. This publication supersedes and replaces all other information previously supplied. NVIDIA Corporation products are not authorized as critical components in life support devices or systems without express written approval of NVIDIA Corporation.

HDMI

HDMI, the HDMI logo, and High-Definition Multimedia Interface are trademarks or registered trademarks of HDMI Licensing LLC.

ROVI Compliance Statement

NVIDIA Products that are ROVI-enabled can only be sold or distributed to buyers with a valid and existing authorization from ROVI to purchase and incorporate the device into buyer's products.

This device is protected by U.S. patent numbers 6,516,132; 5,583,936; 6,836,549; 7,050,698; and 7,492,896 and other intellectual property rights. The use of ROVI Corporation's copy protection technology in the device must be authorized by ROVI Corporation and is intended for home and other limited pay-per-view uses only, unless otherwise authorized in writing by ROVI Corporation. Reverse engineering or disassembly is prohibited.

OpenCL

OpenCL is a trademark of Apple Inc. used under license to the Khronos Group Inc.

Trademarks

NVIDIA and the NVIDIA logo are trademarks or registered trademarks of NVIDIA Corporation in the U.S. and other countries. Other company and product names may be trademarks of the respective companies with which they are associated.

Third Party Copyright

Wretch © 2010, Epic Games, Inc. All Rights Reserved. Epic and Epic Games are trademarks or registered trademarks of Epic Games, Inc. in the United States of America and elsewhere.
StarCraft® II SCV used with permission from Blizzard Entertainment, Inc.

Copyright