
GDC
09
learn
network
inspire
www.GDConf.com
Game Developers Conference®
March 23-27, 2009 | Moscone Center, San Francisco

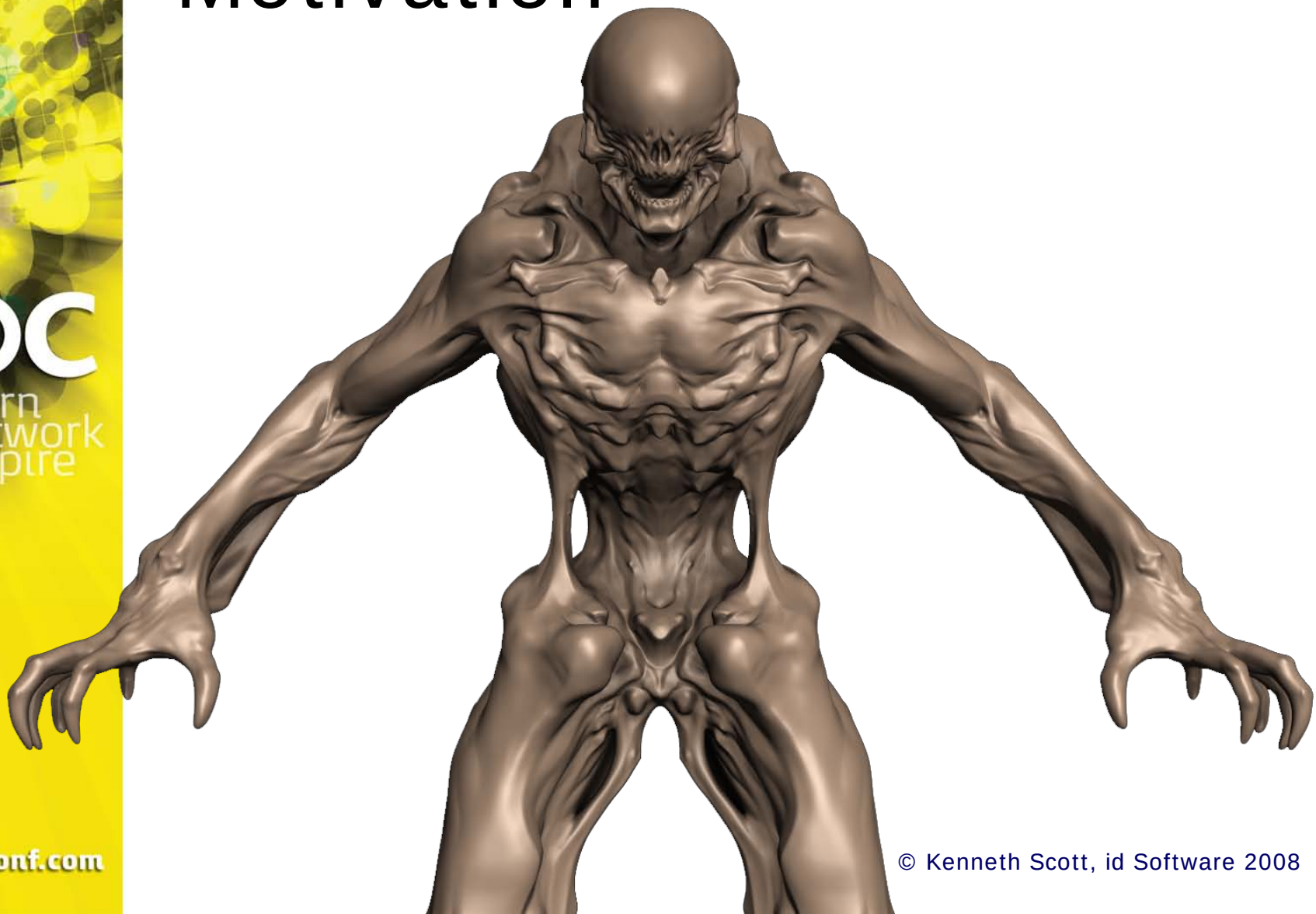
D3D11 Tessellation

Sarah Tariq
NVIDIA

Outline

- Motivation
- Tessellation Pipeline Introduction
- PN Triangles
- Code
- Tessellation Shaders Details
- Silhouette Refinement with PN triangles

Motivation



Motivation

- Enable unprecedented visuals
  - Highly detailed characters
  - Realistic animation

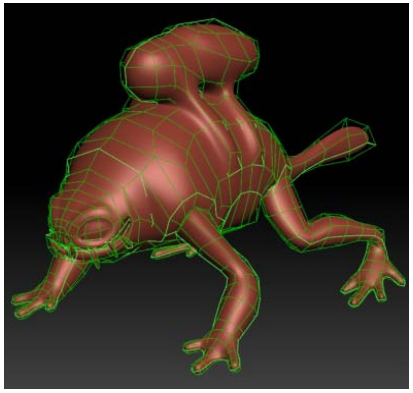
Motivation

- Examples
  - Subdivision Surfaces
    - Easy modeling and flexible animation
    - Widespread use in the movie industry
    - Readily available in modeling and sculpting tools
  - Displacement Mapping
  - Terrain Tessellation

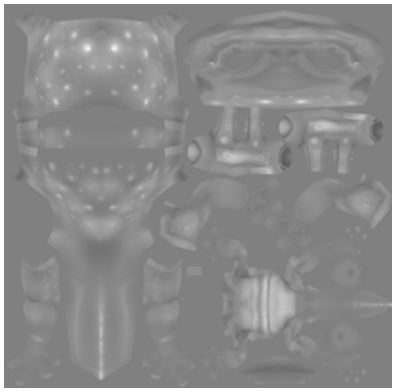
Compression

- Save memory and bandwidth
  - Memory is the main bottleneck to render highly detailed surfaces

=



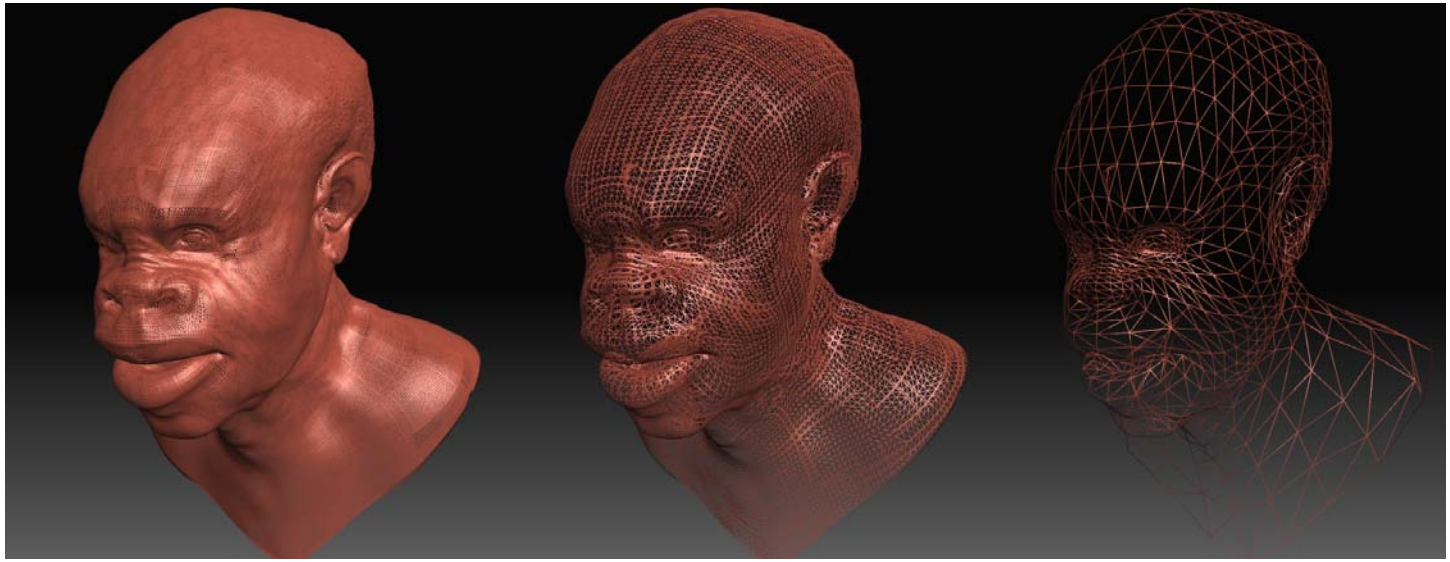
+



	Level 8	Level 16	Level 32	Level 64
Regular Triangle Mesh	16MB	59MB	236MB	943MB
Displaced Subdivision Surface	1.9MB	7.5MB	30MB	118MB

Scalability

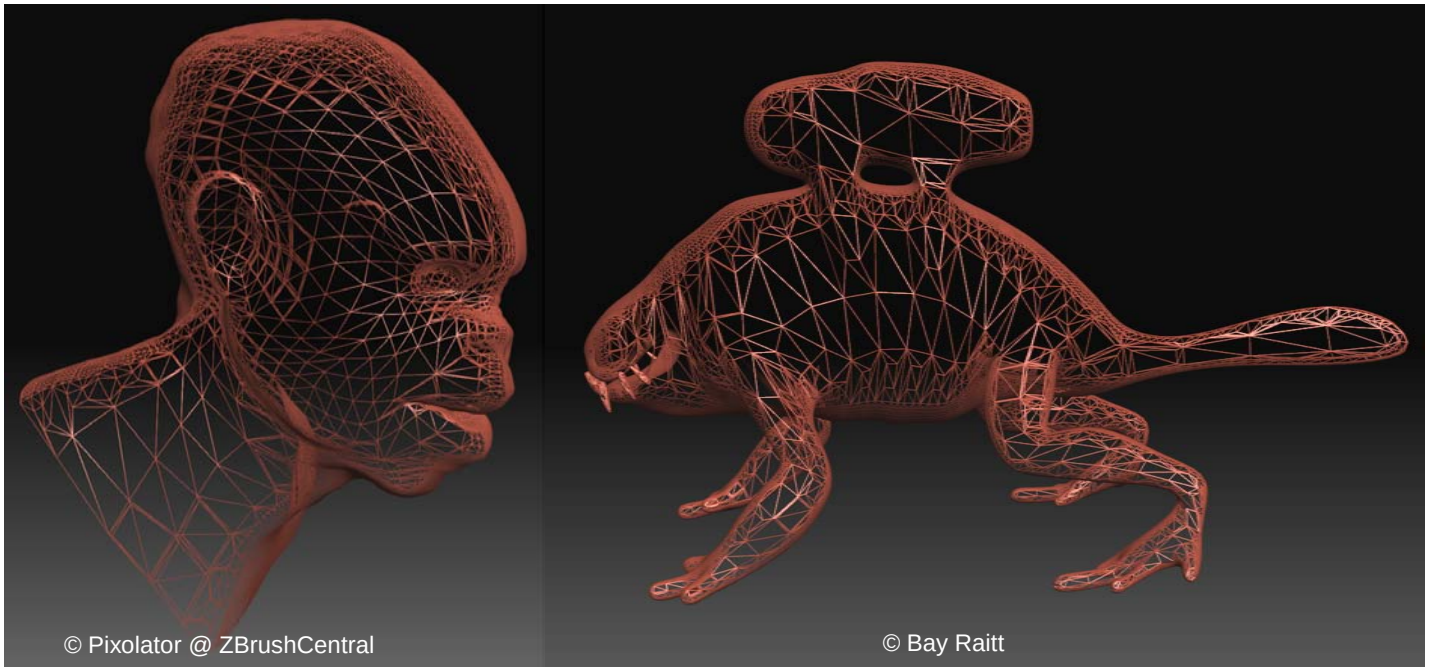
- Continuous Level of Detail



© Pixolator @ ZBrushCentral

Scalability

- View Dependent Level of Detail



Animation & Simulation

- Perform Expensive Computations at lower frequency:

  Realistic animation: blend shapes, morph targets, etc.

  Physics, collision detection, soft body dynamics, etc.

Tessellation Pipeline

- Direct3D11 has support for programmable tessellation
- Two new programable shader stages:
  - Hull Shader (HS)
  - Domain Shader (DS)
- One fixed function stage:
  - Tessellator (TS)

Tessellation Pipeline

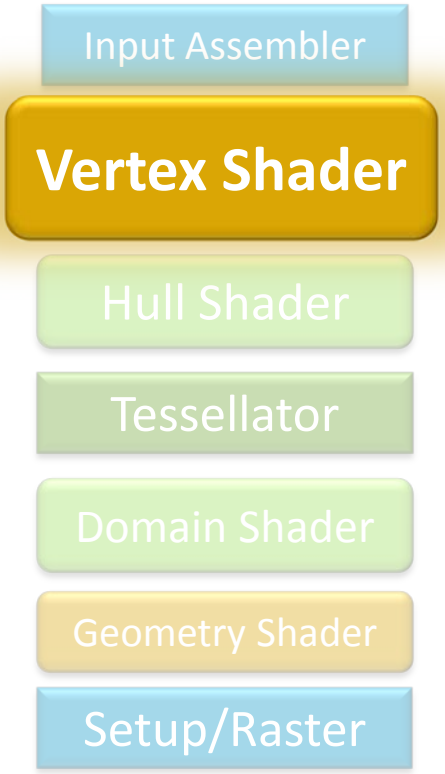
- Hull Shader transforms basis functions from base mesh to surface patches
- Tessellator produces a semi-regular tessellation pattern for each patch
- Domain Shader evaluates surface

Input Assembler

- New patch primitive type
  - Arbitrary vertex count (up to 32)
  - No implied topology
  - Only supported primitive when tessellation is enabled

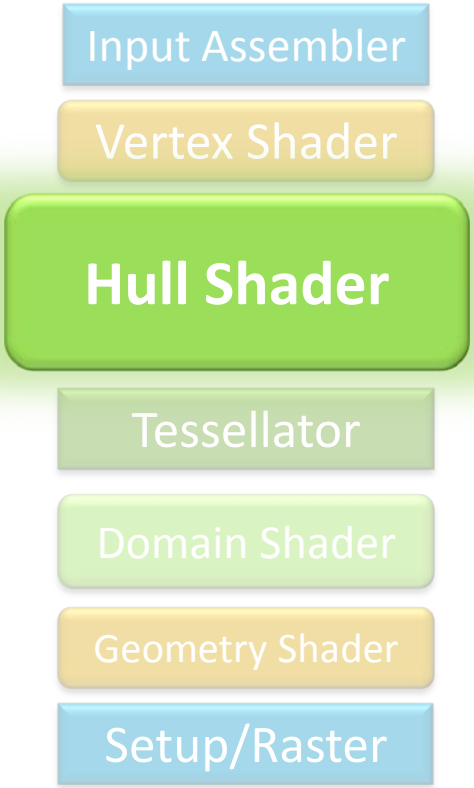
Vertex Shader

- Transforms patch control points
- Usually used for:
  - Animation (skinning, blend shapes)
  - Physics simulation
- Allows more expensive animation at a lower frequency



Hull Shader (HS)

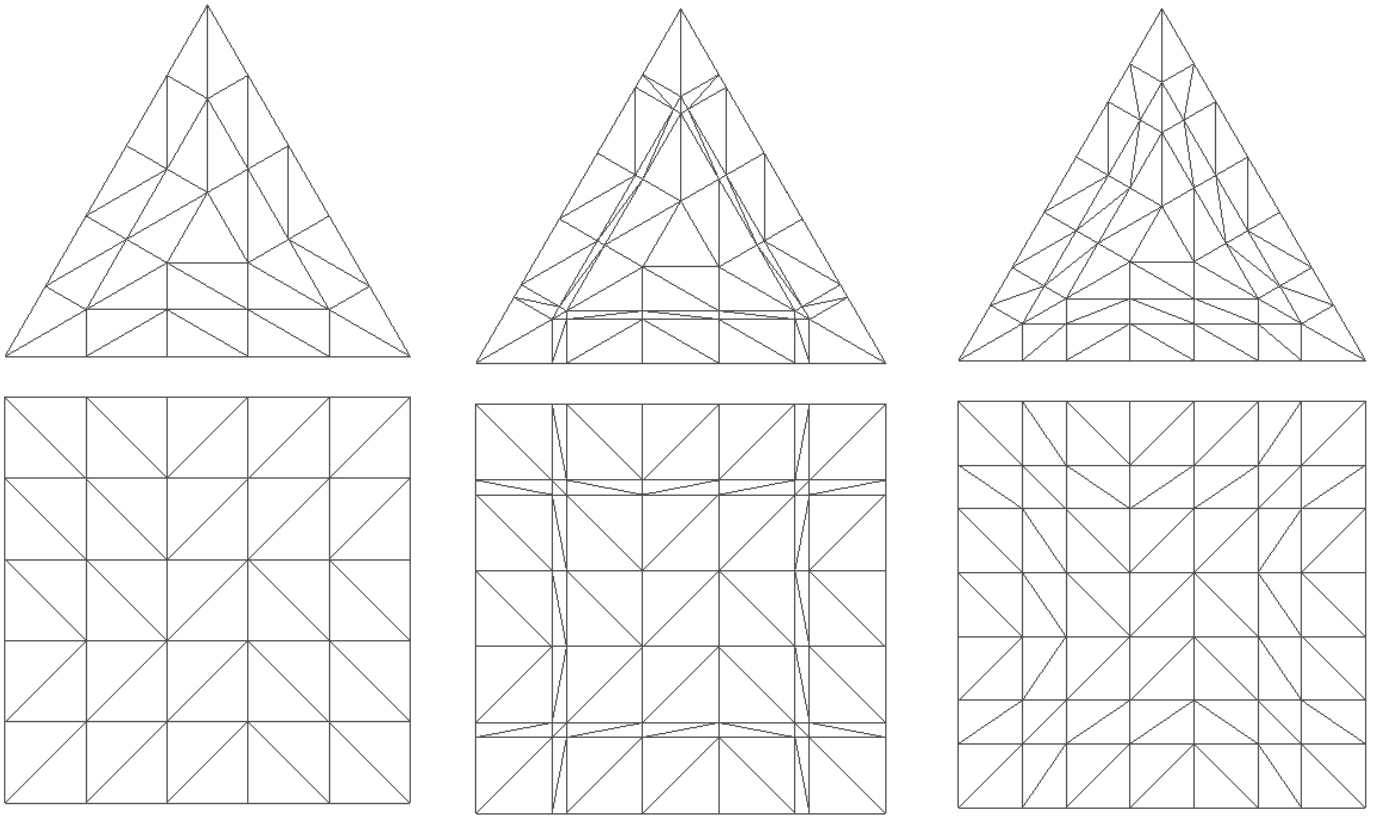
- Transforms control points to a different basis
- Computes edge tessellation levels



Tessellator (TS)

- Fixed function stage, but configurable
- Fully symmetric
- Domains:
  - Triangle, Quad, Isolines
- Spacing:
  - Discrete, Continuous, Pow2

Tessellator (TS)



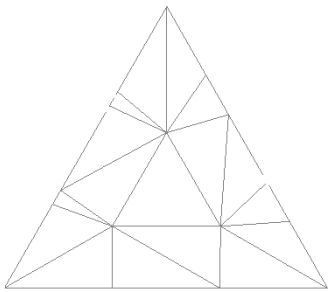
Level 5

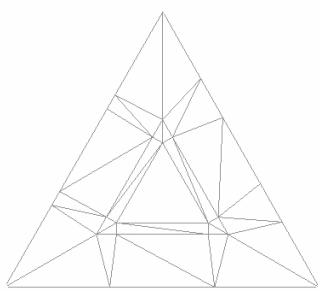
Level 5.4

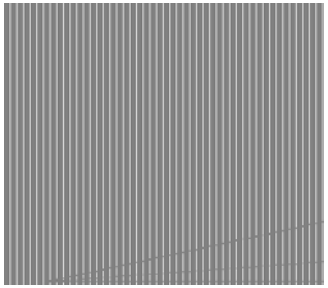
Level 6.6

Tessellator (TS)

Left = 3.5
Right = 4.4
Bottom = 3.0

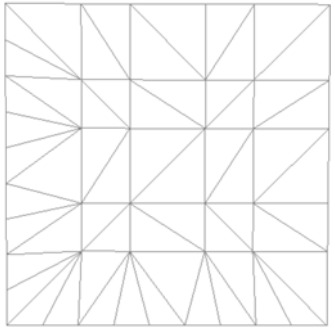


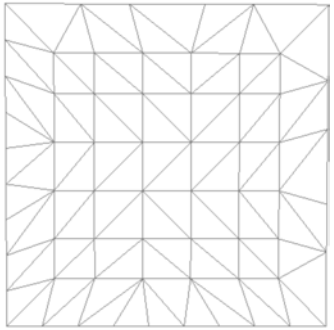


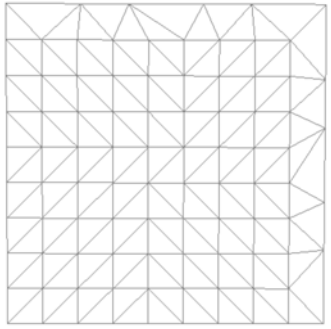


Top,Right = 4.5

Bottom,Left = 9.0







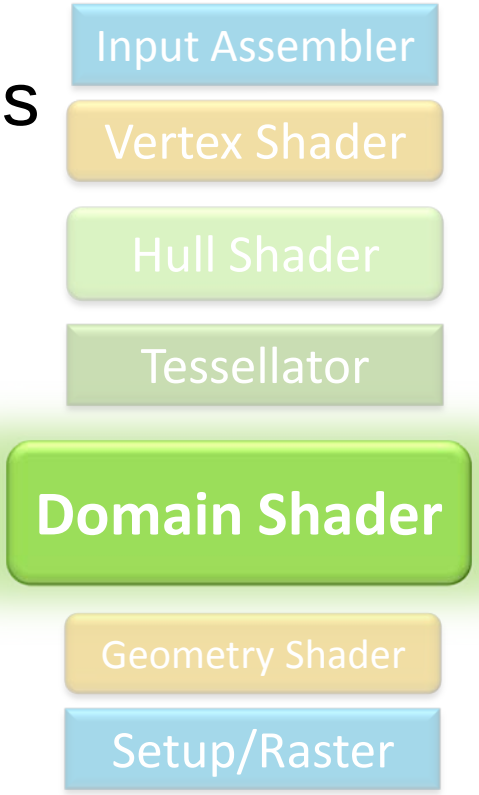
Inside Tess: minimum

Inside Tess: average

Inside Tess: maximum

Domain Shader (DS)

- Evaluate surface given parametric UV coordinates
- Interpolate attributes
- Apply displacements



Example - PN Triangles

- Simple tessellation scheme
  - Provides smoother silhouettes and better shading
- Operates directly on triangle meshes with per vertex Positions and Normals
  - Easily integrated into existing engines

Input Triangles

Output
Curved PN triangles

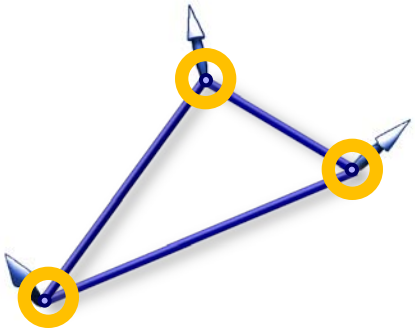
Vlachos et al, http://ati.amd.com/developer/curvedpntriangles.pdf

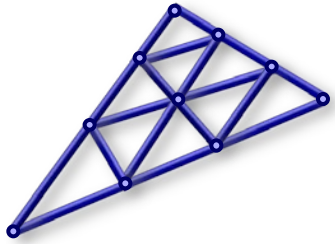
PN Triangles - positions

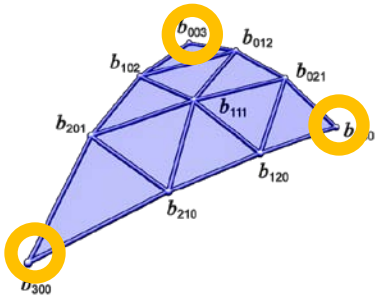
- 1- Replace input triangle with a bezier patch
  - Use Hull Shader
- 2- Triangulated bezier patch into a specified number of sub triangles
  - Use Tessellator and Domain Shader
  - Number of Sub triangles specified by Hull Shader

PN Triangles- positions

Computing Position Control Points







Exterior control point positions:

same as input vertex positions

$$b_{300} = P_1$$
$$b_{030} = P_2$$
$$b_{003} = P_3$$

Interior control point positions:

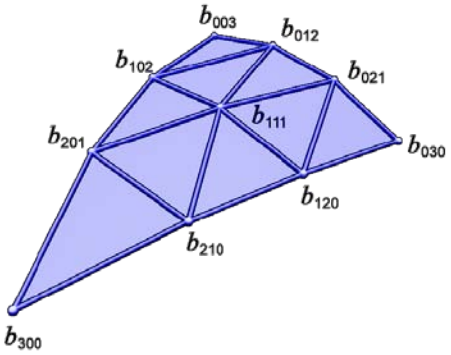
Weighted combinations of input positions and normals

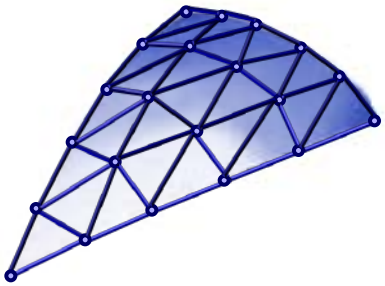
$$w_{ij} = (P_j - P_i) \bullet N_i$$
$$b_{210} = (2P_1 + P_2 - w_{12}N_1)/3$$
$$b_{120} = (2P_2 + P_1 - w_{21}N_2)/3$$

...

PN Triangles- positions

Evaluating tessellated positions from control points

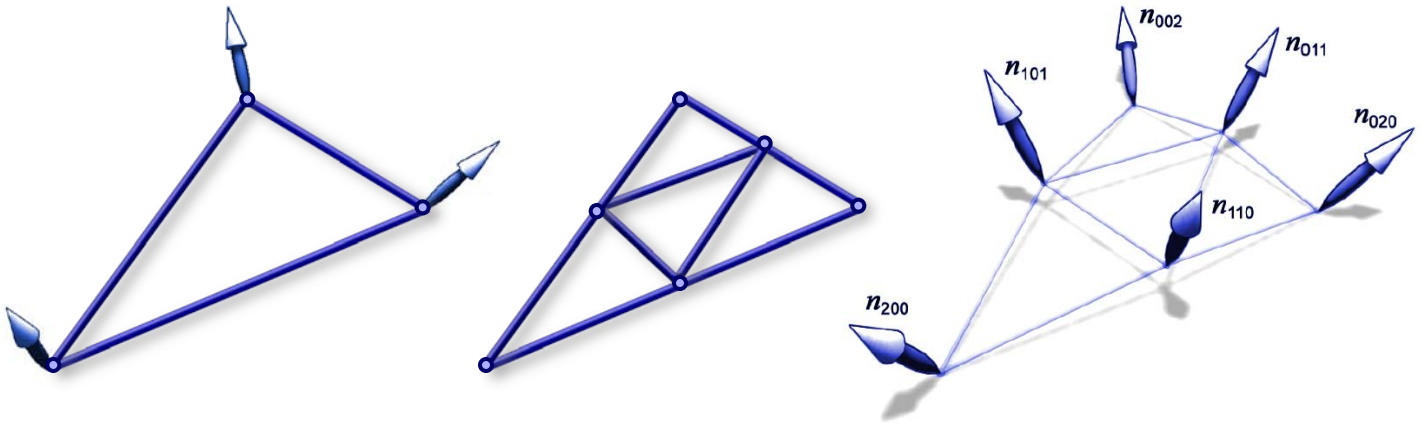




$$w = 1 - u - v \qquad u, v, w \geq 0$$

$$\begin{aligned} b(u,v) = \; & b_{300}w^3 + b_{030}u^3 + b_{003}v^3 \\ & + b_{210}3w^2u + b_{120}3wu^2 + b_{201}3w^2v \\ & + b_{021}3u^2v + b_{102}3wv^2 + b_{012}3uv^2 \\ & + b_{111}6wuv \end{aligned}$$

PN Triangles - Normals



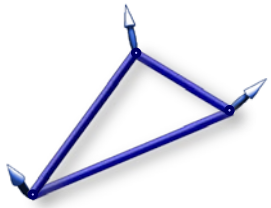
- Normal at a tessellated vertex is a quadratic function of position and normal data

$$w = 1 - u - v$$

$$n(u,v) = n_{200}w^2 + n_{020}u^2 + n_{002}v^2 + n_{110}wu + n_{011}uv + n_{101}wv$$

Tessellation Pipeline

HS input:
• input control points

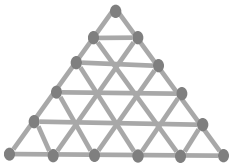


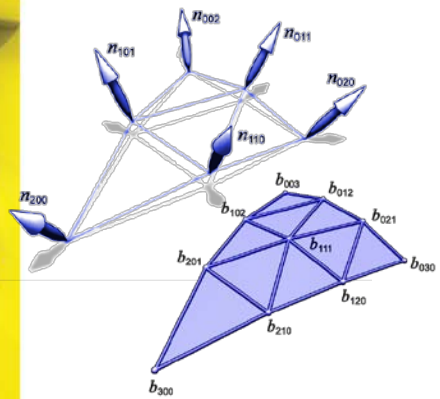
Hull Shader

HS output:
• Tessellation factors

Tessellator

Tessellator Output:
• uvw coordinates

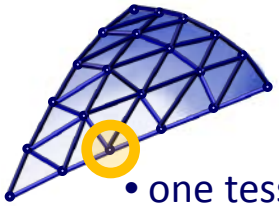




HS output:
• output control points
• Tessellation factors

Domain Shader

DS Input from Tessellator:
• uvw coordinates for one vertex



DS Output:
• one tessellated vertex

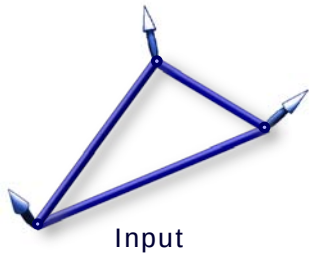
Hull Shader Stages

- Main Hull Shader
  - Calculate control point data
  - Invoked once per output control point
- Patch Constant Function
  - Must calculate tessellation factors
  - Has access to control point data calculated in the Main Hull Shader
  - Executes once per patch

PN Triangles Hull Shader

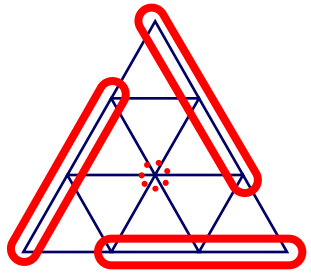
- Compute control positions and normals in main Hull Shader
- Compute tessellation factors and center location in patch constant function
  - The center location needs to average all the other control point locations so it belongs in the patch constant function

PN Triangles Hull Shader



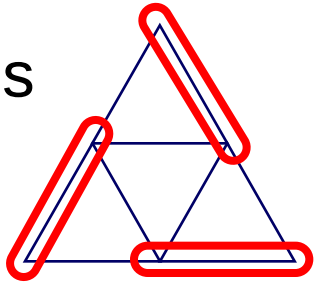
Input

- Partitioning the computation
- To balance the workload across threads we partition the control points into 3 uber control points



Positions

- Each uber control point computes
  - 3 positions
  - 2 normals



Normals

Thread distribution in Hull Shader

PN Triangles Hull Shader

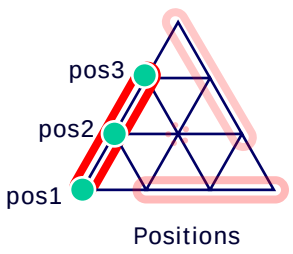
```
struct HS_PATCH_DATA
{
            float edges[3]   : SV_TessFactor;
            float inside     : SV_InsideTessFactor;
            float center[3]  : CENTER;
};
```

Data output by the patch constant function

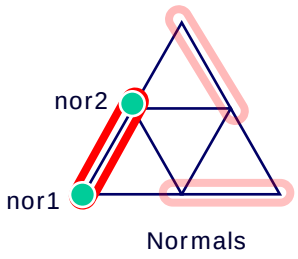
```
struct HS_CONTROL_POINT
{
            float pos1[3]  : POSITION1;
            float pos2[3]  : POSITION2;
            float pos3[3]  : POSITION3;
            float3 nor1  : NORMAL0;
            float3 nor2  : NORMAL1;
            float3 tex   : TEXCOORD0;
};
```

Data output by main tessellation function

Control point 1

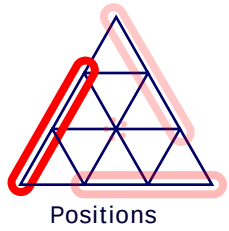


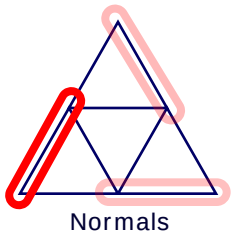
Positions



Normals

PN Triangles Hull Shader





Control point 1

```
[domain("tri")]
[outputtopology("triangle_cw")]
[outputcontrolpoints(3)]
[partitioning("fractional_odd")]
[patchconstantfunc("HullShaderPatchConstant")]
HS_CONTROL_POINT HullShaderControlPointPhase( InputPatch<HS_DATA_INPUT, 3> inputPatch,
                                uint tid : SV_OutputControlPointID, uint pid : SV_PrimitiveID)
{
            int next = (1 << tid) & 3; // (tid + 1) % 3

            float3 p1 = inputPatch[tid].position;
            float3 p2 = inputPatch[next].position;
            float3 n1 = inputPatch[tid].normal;
            float3 n2 = inputPatch[next].normal;

            HS_CONTROL_POINT output;
            //control points positions
            output.pos1 = (float[3])p1;
            output.pos2 = (float[3])(2 * p1 + p2 - dot(p2-p1, n1) * n1);
            output.pos3 = (float[3])(2 * p2 + p1 - dot(p1-p2, n2) * n2);
            //control points normals
            float3 v12 = 4 * dot(p2-p1, n1+n2) / dot(p2-p1, p2-p1);
            output.nor1 = n1;
            output.nor2 = n1 + n2 - v12 * (p2 - p1);

            output.tex = inputPatch[tid].texcoord;

            return output;
}
```

Read input data

Compute control points

PN Triangles Hull Shader

```
//patch constant data
HS_PATCH_DATA HullShaderPatchConstant( OutputPatch<HS_CONTROL_POINT, 3> controlPoints )
{
    HS_PATCH_DATA patch = (HS_PATCH_DATA)0;
    //calculate Tessellation factors
    HullShaderCalcTessFactor(patch, controlPoints, 0);
    HullShaderCalcTessFactor(patch, controlPoints, 1);
    HullShaderCalcTessFactor(patch, controlPoints, 2);
    patch.inside = max(max(patch.edges[0], patch.edges[1]), patch.edges[2]);
    //calculate center
   float3 center = ((float3)controlPoints[0].pos2 + (float3)controlPoints[0].pos3) * 0.5 -
                    (float3)controlPoints[0].pos1 +
                   ((float3)controlPoints[1].pos2 + (float3)controlPoints[1].pos3) * 0.5 –
                    (float3)controlPoints[1].pos1 +
                   ((float3)controlPoints[2].pos2 + (float3)controlPoints[2].pos3) * 0.5 –
                    (float3)controlPoints[2].pos1;

    patch.center = (float[3])center;
    return patch;
}

//helper functions
float edgeLod(float3 pos1, float3 pos2) { return dot(pos1, pos2); }
void HullShaderCalcTessFactor( inout HS_PATCH_DATA patch,
                  OutputPatch<HS_CONTROL_POINT, 3> controlPoints, uint tid : SV_InstanceID)
{
    int next = (1 << tid) & 3; // (tid + 1) % 3
    patch.edges[tid] = edgeLod((float3)controlPoints[tid].pos1,
                       (float3)controlPoints[next].pos1);
    return;
}
```

Patch Constant Phase

- Patch Constant phase is implicitly parallelized
  - Compiler looks for opportunities to create independent instances

```
HS_PATCH_DATA HullShaderPatchConstant( OutputPatch<HS_CONTROL_POINT, 3> controlPoints )
{    HS_PATCH_DATA patch = (HS_PATCH_DATA)0;
    //calculate Tessellation factors
    HullShaderCalcTessFactor(patch, controlPoints, 0);
    HullShaderCalcTessFactor(patch, controlPoints, 1);
    HullShaderCalcTessFactor(patch, controlPoints, 2);
```

instance 1

instance 2

instance 3

Tessellation Pipeline

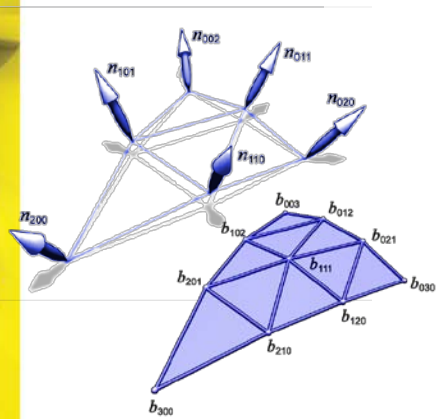
HS input:
• input control points

Hull Shader

HS output:
• Tessellation factors

Tessellator

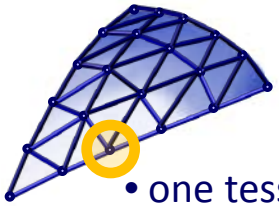
Tessellator Output:
• uvw coordinates



HS output:
• output control points
• Tessellation factors

Domain Shader

DS Input from Tessellator:
• uvw coordinates for one vertex



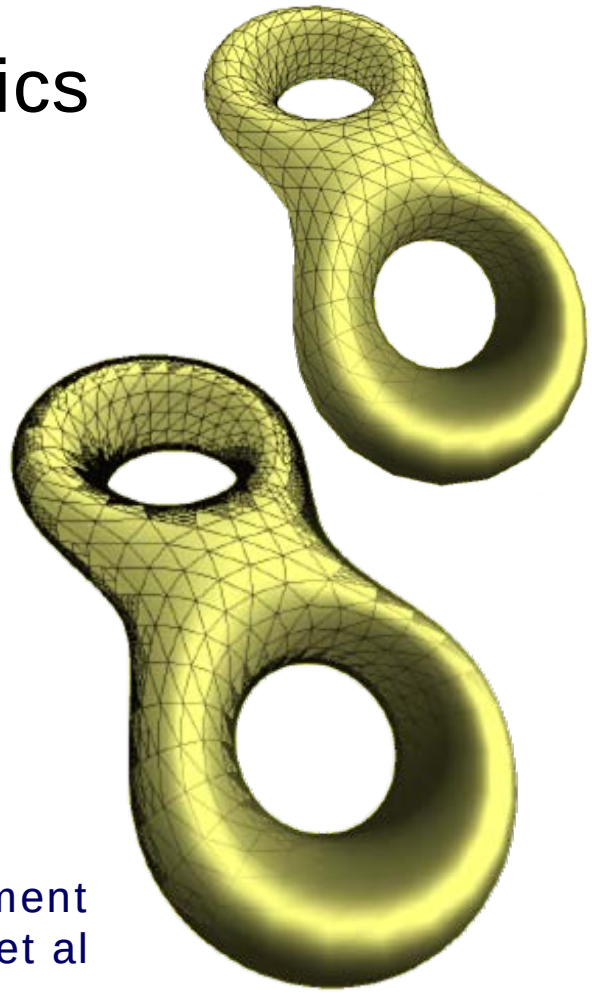
DS Output:
• one tessellated vertex

PN-Triangles - DS

```
[domain("triangle")]
DS_DATA_OUTPUT DomainShaderPN(HS_PATCH_DATA patchData,
            const OutputPatch<HS_CONTROL_POINT, 3> input, float3 uvw : SV_DomainLocation)
{
    DS_DATA_OUTPUT output;
    float u = uvw.x;
    float v = uvw.y;
    float w = uvw.z;

    //output position is weighted combination of all 10 position control points
    float3 pos =
(float3)input[0].pos1 * w*w*w +(float3)input[1].pos1 * u*u*u +(float3)input[2].pos1 * v*v*v +
(float3)input[0].pos2 * w*w*u +(float3)input[0].pos3 * w*u*u +(float3)input[1].pos2 * u*u*v +
(float3)input[1].pos3 * u*v*v +(float3)input[2].pos2 * v*v*w + float3)input[2].pos3 * v*w*w +
(float3)patchData.center * u*v*w;

    //output normal is weighted combination of all 10 position control points
    float3 nor =
                input[0].nor1 * w*w + input[1].nor1 * u*u + input[2].nor1 * v*v +
               input[0].nor2* w*u + input[1].nor2 * u*v + input[2].nor2 * v*w;

    //transform and output data
    output.position  = mul(float4(pos,1), g_mViewProjection);
    output.view = mul(float4(pos,1),g_mView).xyz;
    output.normal = mul(float4(normalize(nor),1),g_mNormal).xyz;
    output.vUV = input[0].tex * w + input[1].tex * u + input[2].tex * v;

    return output;
}
```

Adaptive Silhouette Enhancement

- Different useful metrics for deciding LOD per patch
- For example:
  - Only add detail at silhouettes



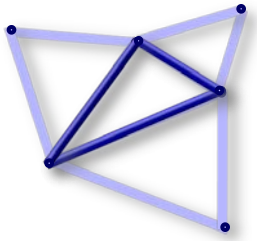
Real-time linear silhouette enhancement
Dyken et al

Adaptive Silhouette Enhancement

- A silhouette edge is an edge between a front facing and a back facing triangle
  - Determine triangle facing using dot product of triangle normal and vector to eye
- Can also use the multiplication of the dot products as a function to define smoothly varying LODs to avoid popping

Adaptive Silhouette Enhancement

- Need to know normals of adjacent triangles



- Use Patch primitive to represent triangle with neighborhood
- Hull Shader computes adaptive per edge tessellation factors based on silhouette function

Questions?

Thank you for your time!