

Week9

The Tessellation Stages

Hooman Salamat

Objectives

1. To discover the patch primitive types used for tessellation.
2. To obtain an understanding of what each tessellation stage does, and what the expected inputs and outputs are for each stage.
3. To be able to tessellate geometry by writing hull and domain shader programs.
4. To become familiar with different strategies for determining when to tessellate and to become familiar with performance considerations regarding hardware tessellation.
5. To learn the mathematics of Bézier curves and surfaces and how to implement them in the tessellation stages.

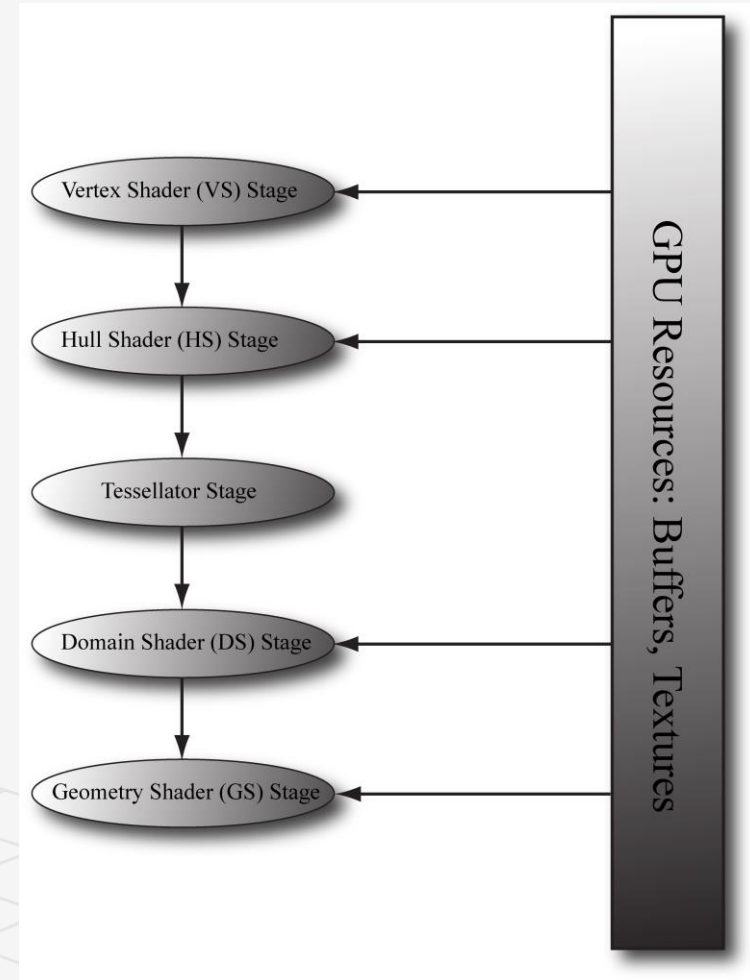


Tessellation

Tessellation refers to subdividing geometry into smaller triangles

Increase the triangle count adds detail to the mesh

1. Dynamic LOD on the GPU. We can dynamically adjust the detail of a mesh based on its distance from the camera and other factors.
 - As the object gets closer to the camera, we can continuously increase tessellation to increase the detail of the object.
2. Physics and animation efficiency. We can perform physics and animation calculations on the low-poly mesh, and then tessellate to the higher polygon version.
3. Memory savings. We can store lower polygon meshes in memory (on disk, RAM, and VRAM), and then have the GPU tessellate to the higher polygon version on the fly.



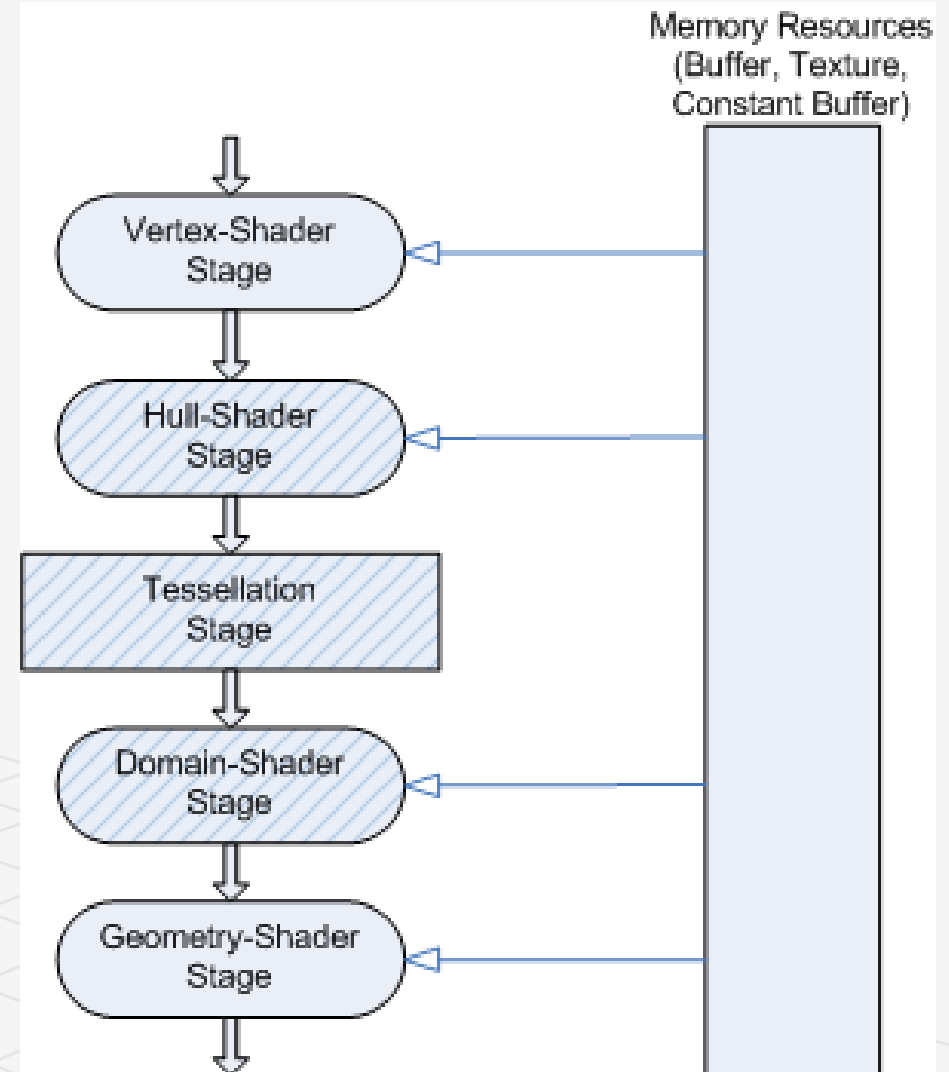
New Pipeline Stages

Tessellation uses the GPU to calculate a more detailed surface from a surface constructed from quad patches, triangle patches or isolines. To approximate the high-ordered surface, each patch is subdivided into triangles, points, or lines using tessellation factors.

[Hull-Shader Stage](#) - A programmable shader stage that produces a geometry patch (and patch constants) that correspond to each input patch (quad, triangle, or line).

[Tessellator Stage](#) - A fixed function pipeline stage that creates a sampling pattern of the domain that represents the geometry patch and generates a set of smaller objects (triangles, points, or lines) that connect these samples.

[Domain-Shader Stage](#) - A programmable shader stage that calculates the vertex position that corresponds to each domain sample.



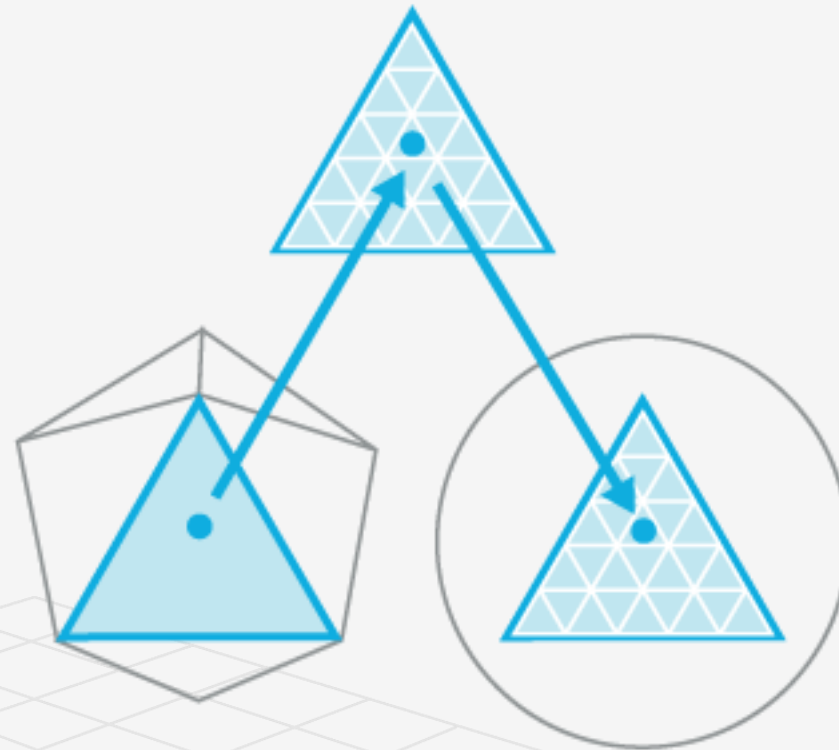
The tessellation stages

The following diagram shows the progression through the tessellation stages.

The progression starts with the low-detail subdivision surface.

The progression next highlights the input patch with the corresponding geometry patch, domain samples, and triangles that connect these samples.

The progression finally highlights the vertices that correspond to these samples.



TESSELLATION PRIMITIVE TYPES

When we render for tessellation, we do not submit triangles to the IA stage. Instead, we submit *patches* with a number of *control points*.

A triangle can be thought of a triangle patch with three control points

(D3D_PRIMITIVE_3_CONTROL_POINT_PATCH),

A simple quad patch can be submitted with four control points

(D3D_PRIMITIVE_4_CONTROL_POINT_PATCH)

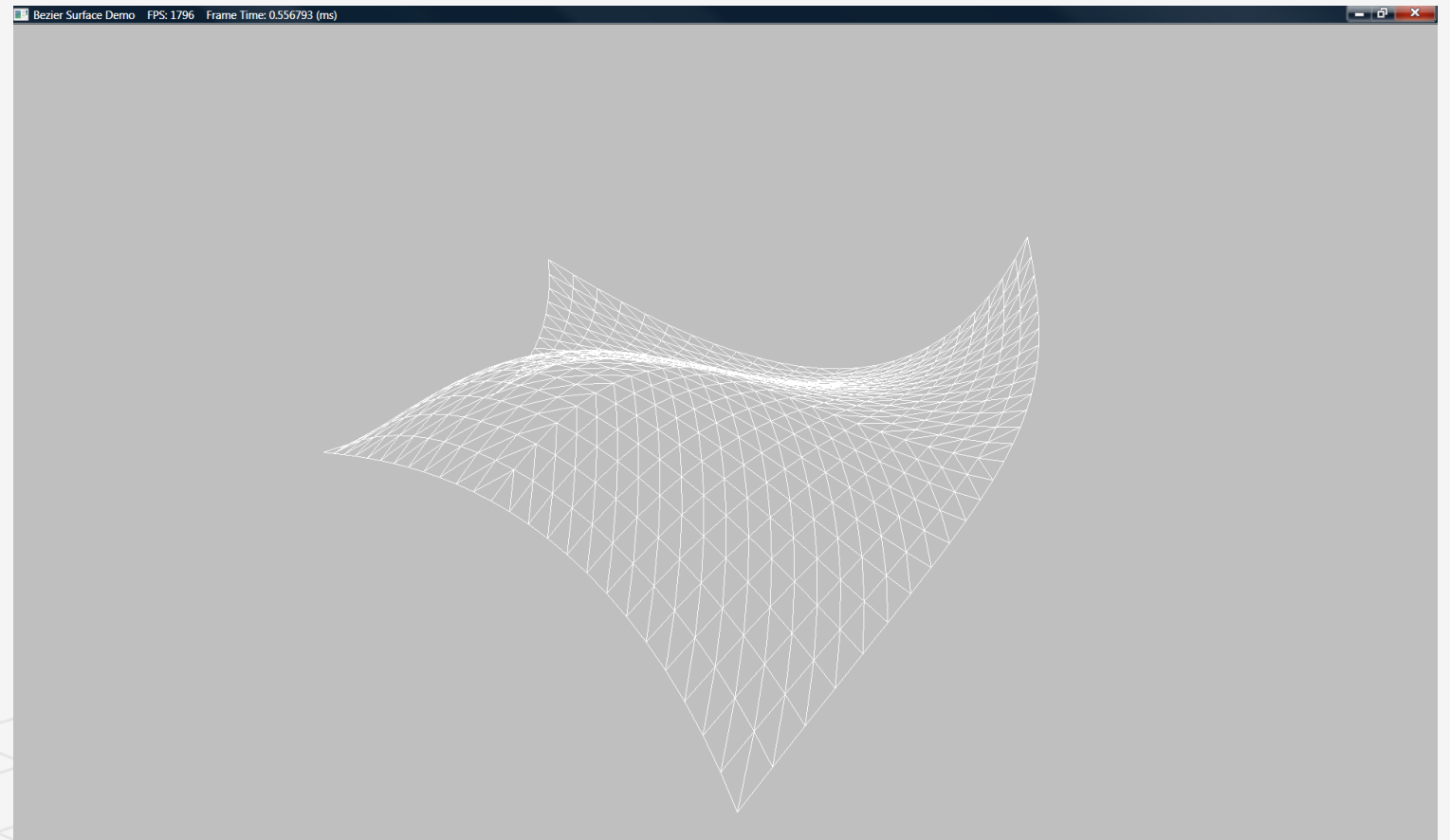
```
typedef enum D3D_PRIMITIVE_TOPOLOGY
{
    D3D_PRIMITIVE_TOPOLOGY_UNDEFINED= 0,
    D3D_PRIMITIVE_TOPOLOGY_POINTLIST= 1,
    D3D_PRIMITIVE_TOPOLOGY_LINELIST= 2,
    D3D_PRIMITIVE_TOPOLOGY_LINESTRIP= 3,
    D3D_PRIMITIVE_TOPOLOGY_TRIANGLELIST= 4,
    D3D_PRIMITIVE_TOPOLOGY_TRIANGLESTRIP= 5,
    D3D_PRIMITIVE_TOPOLOGY_LINELIST_ADJ= 10,
    D3D_PRIMITIVE_TOPOLOGY_LINESTRIP_ADJ= 11,
    D3D_PRIMITIVE_TOPOLOGY_TRIANGLELIST_ADJ= 12,
    D3D_PRIMITIVE_TOPOLOGY_TRIANGLESTRIP_ADJ= 13,
    D3D_PRIMITIVE_TOPOLOGY_1_CONTROL_POINT_PATCHLIST= 33,
    D3D_PRIMITIVE_TOPOLOGY_2_CONTROL_POINT_PATCHLIST= 34,
    D3D_PRIMITIVE_TOPOLOGY_3_CONTROL_POINT_PATCHLIST= 35,
    D3D_PRIMITIVE_TOPOLOGY_4_CONTROL_POINT_PATCHLIST= 36,
    ...
    D3D_PRIMITIVE_TOPOLOGY_31_CONTROL_POINT_PATCHLIST= 63,
    D3D_PRIMITIVE_TOPOLOGY_32_CONTROL_POINT_PATCHLIST= 64
}
```

Control Points

The idea of control points comes from the construction of certain kinds of mathematical curves and surfaces.

If you have ever worked with Bézier curves in a drawing program like Adobe Illustrator, then you know that you mold the shape of the curve via control points.

Increasing the number of control points gives you more degrees of freedom in shaping the patch.



THE TESSELLATION STAGE

As programmers, we do not have control of the tessellation stage.

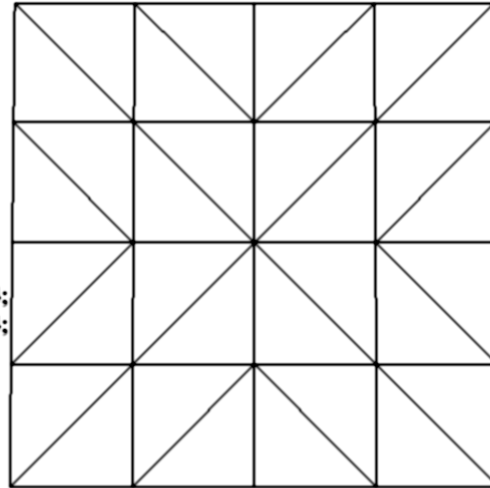
This stage is all done by the hardware, and tessellates the patches based on the tessellation factors output from the constant hull shader program.

The following figures illustrate different subdivisions based on the tessellation factors.

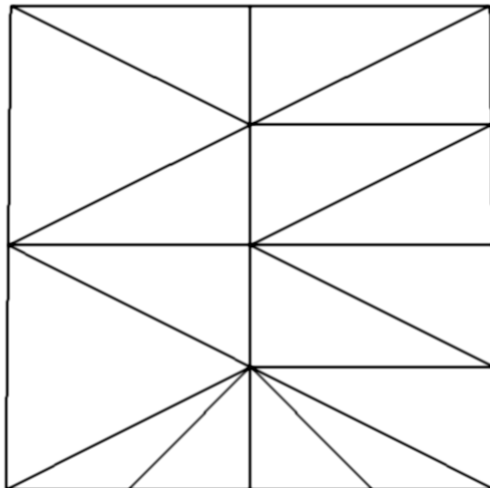
Tessellating a quad patch consists of two parts:

1. Four edge tessellation factors control how much to tessellate along each edge.
2. Two interior tessellation factors indicate how to tessellate the quad patch (one tessellation factor for the horizontal dimension of the quad, and one tessellation factor for the vertical dimension of the quad).

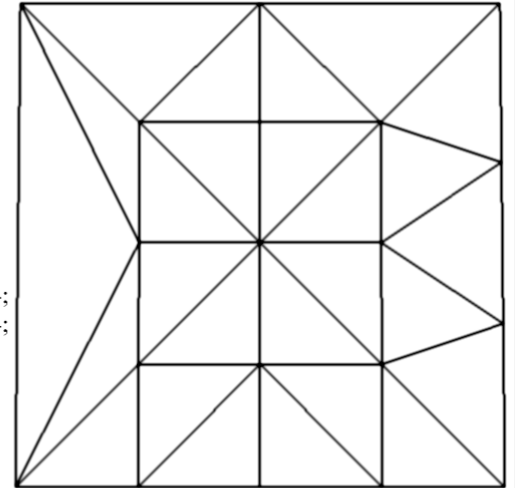
```
pt.EdgeTess[0] = 4;  
pt.EdgeTess[1] = 4;  
pt.EdgeTess[2] = 4;  
pt.EdgeTess[3] = 4;  
  
pt.InsideTess[0] = 4;  
pt.InsideTess[1] = 4;
```



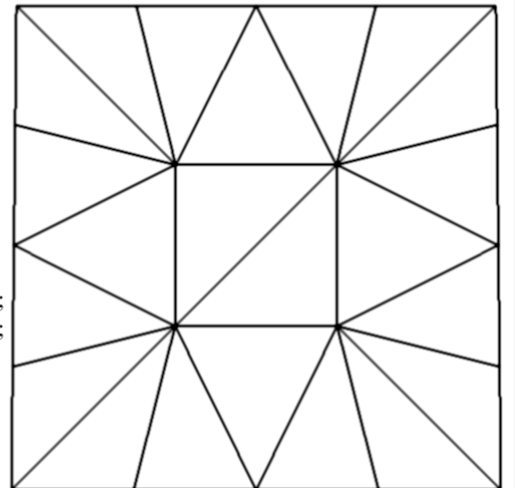
```
pt.EdgeTess[0] = 2;  
pt.EdgeTess[1] = 2;  
pt.EdgeTess[2] = 4;  
pt.EdgeTess[3] = 4;  
  
pt.InsideTess[0] = 2;  
pt.InsideTess[1] = 4;
```



```
pt.EdgeTess[0] = 1;  
pt.EdgeTess[1] = 2;  
pt.EdgeTess[2] = 3;  
pt.EdgeTess[3] = 4;  
  
pt.InsideTess[0] = 4;  
pt.InsideTess[1] = 4;
```



```
pt.EdgeTess[0] = 4;  
pt.EdgeTess[1] = 4;  
pt.EdgeTess[2] = 4;  
pt.EdgeTess[3] = 4;  
  
pt.InsideTess[0] = 3;  
pt.InsideTess[1] = 3;
```

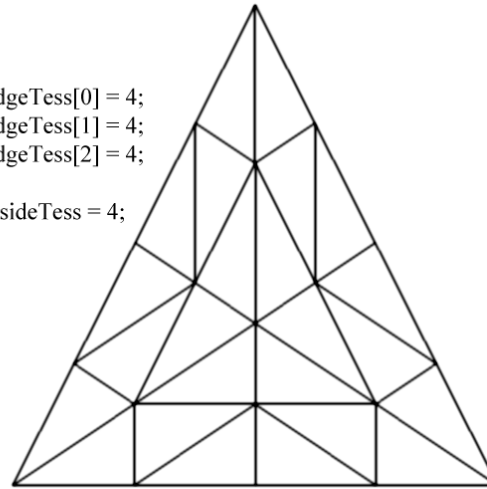


Tessellating a *triangle patch*

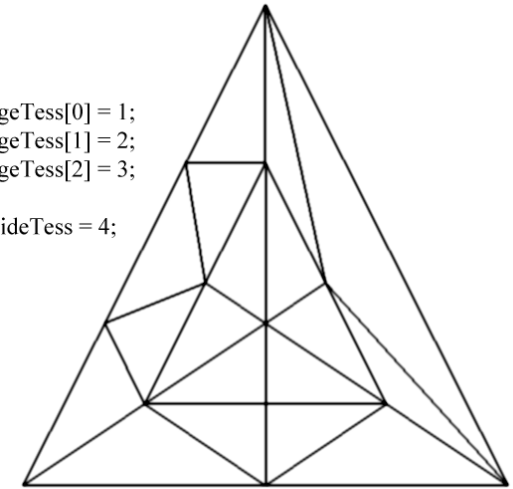
Tessellating a *triangle patch* also consists of two parts:

1. Three edge tessellation factors control how much to tessellate along each edge.
2. One interior tessellation factor indicates how much to tessellate the triangle patch.

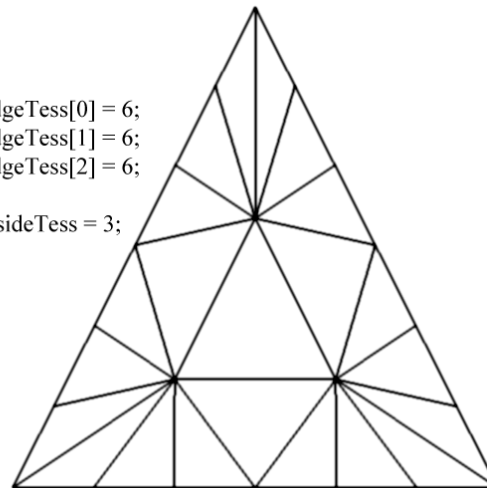
```
pt.EdgeTess[0] = 4;  
pt.EdgeTess[1] = 4;  
pt.EdgeTess[2] = 4;  
  
pt.InsideTess = 4;
```



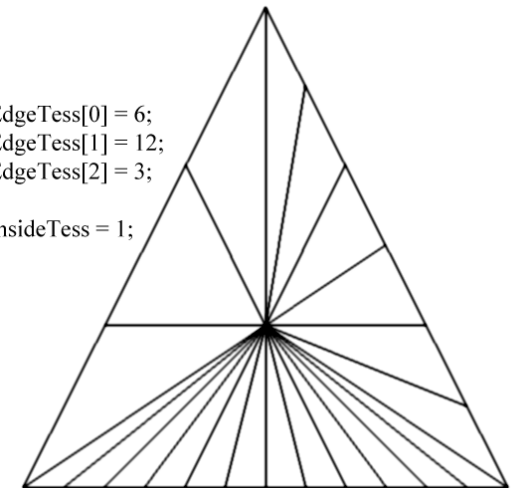
```
pt.EdgeTess[0] = 1;  
pt.EdgeTess[1] = 2;  
pt.EdgeTess[2] = 3;  
  
pt.InsideTess = 4;
```



```
pt.EdgeTess[0] = 6;  
pt.EdgeTess[1] = 6;  
pt.EdgeTess[2] = 6;  
  
pt.InsideTess = 3;
```



```
pt.EdgeTess[0] = 6;  
pt.EdgeTess[1] = 12;  
pt.EdgeTess[2] = 3;  
  
pt.InsideTess = 1;
```



IASetPrimitiveTopology

This is how we passed primitive types when we drew the box:

```
mCommandList->IASetPrimitiveTopology(D3D_PRIMITIVE_TOPOLOGY_TRIANGLELIST);
```

With

```
psoDesc.PrimitiveTopologyType = D3D12_PRIMITIVE_TOPOLOGY_TYPE_TRIANGLE;
```

When passing control point primitive types to [ID3D12GraphicsCommandList::IASetPrimitiveTopology](#),

```
quadPatchRitem->PrimitiveType = D3D_PRIMITIVE_TOPOLOGY_4_CONTROL_POINT_PATCHLIST;
```

```
cmdList->IASetPrimitiveTopology(ri->PrimitiveType);
```

you need to set the

`D3D12_GRAPHICS_PIPELINE_STATE_DESC::PrimitiveTopologyType` field to `D3D12_PRIMITIVE_TOPOLOGY_TYPE_PATCH`:

```
opaquePsoDesc.PrimitiveTopologyType = D3D12_PRIMITIVE_TOPOLOGY_TYPE_PATCH;
```

Because we submit patch control points to the rendering pipeline, the control points are what get pumped through the vertex shader.

Therefore, when tessellation is enabled, the vertex shader is really a “vertex shader for control points,” and we can do any control point work we need before tessellation starts.

Hull-Shader Stage

A hull shader -- which is invoked once per patch -- transforms input control points that define a low-order surface into control points that make up a patch.

It also does some per patch calculations to provide data for the tessellation stage and the domain stage.

A hull shader is implemented with an HLSL function, and has the following properties:

The shader input is between 1 and 32 control points.

The shader output is between 1 and 32 control points, regardless of the number of tessellation factors. The control-points output from a hull shader can be consumed by the domain-shader stage. Patch constant data can be consumed by a domain shader; tessellation factors can be consumed by the domain shader and the tessellation stage.

Tessellation factors determine how much to subdivide each patch.

The shader declares the state required by the tessellator stage. This includes information such as the number of control points, the type of patch face and the type of partitioning to use when tessellating. This information appears as declarations typically at the front of the shader code.

If the hull-shader stage sets any edge tessellation factor to $= 0$ or NaN, the patch will be culled. As a result, the tessellator stage may or may not run, the domain shader will not run, and no visible output will be produced for that patch.

The control-point phase operates once for each control-point, reading the control points for a patch, and generating one output control point (identified by a `ControlPointID`).

The patch-constant phase operates once per patch to generate edge tessellation factors and other per-patch constants. Internally, many patch-constant phases may run at the same time. The patch-constant phase has read-only access to all input and output control points.



THE Constant HULL SHADER

Hull Shader consists of two shaders:

1. Constant Hull Shader
2. Control Point Hull Shader

The *constant hull shader* is evaluated per patch, and is tasked with outputting the so called *tessellation factors* of the mesh.

The tessellation factors (*SV_TessFactor* and *SV_InsideTessFactor*) instruct the tessellation stage how much to tessellate the patch.

Here is an example of a *quad patch* with four control points, where we tessellate it uniformly three times.

The constant hull shader inputs all the control points of the patch, which is defined by the type *InputPatch<VertexOut, 4>*.

Tessellating a quad patch consists of two parts:

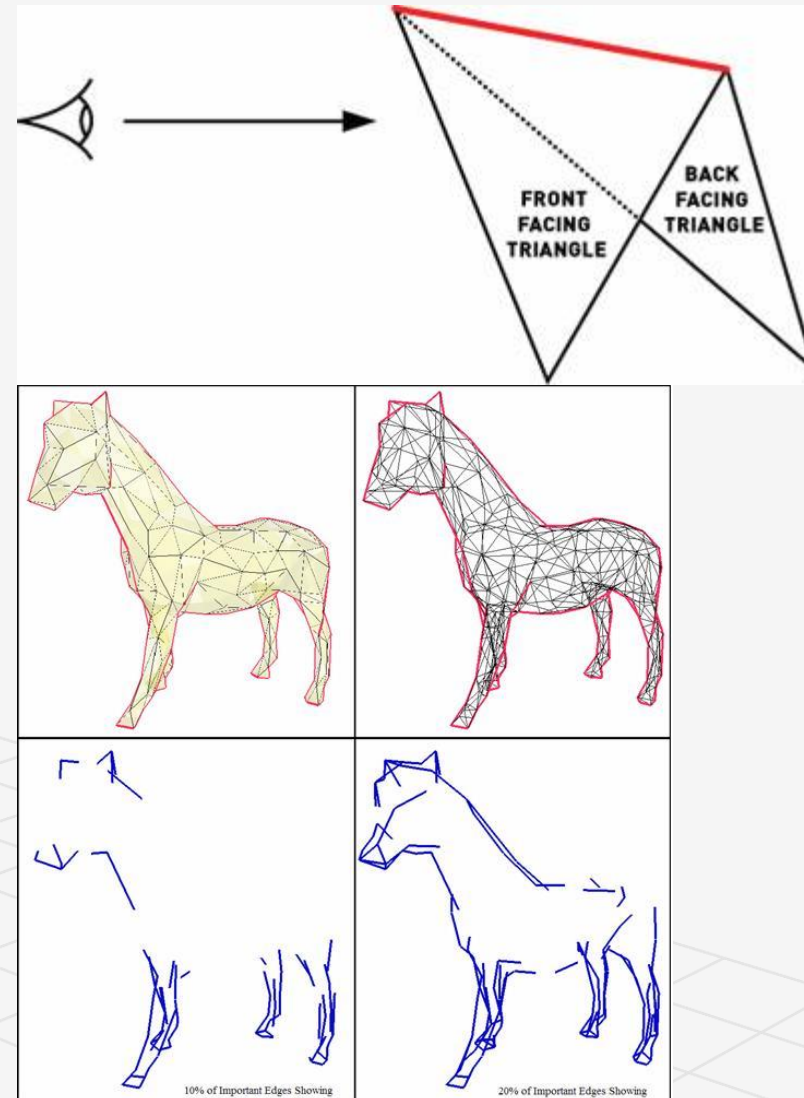
1. Four edge tessellation factors control how much to tessellate along each edge.
2. Two interior tessellation factors indicate how to tessellate the quad patch (one tessellation factor for the horizontal dimension of the quad, and one tessellation factor for the vertical dimension of the quad).

```
struct PatchTess
{
    float EdgeTess[4] : SV_TessFactor;
    float InsideTess[2] : SV_InsideTessFactor;
    // Additional info you want associated per patch.
};
PatchTess ConstantHS(InputPatch<VertexOut, 4> patch,
    uint patchID : SV_PrimitiveID)
{
    PatchTess pt;
    // Uniformly tessellate the patch 3 times.
    pt.EdgeTess[0] = 3; // Left edge
    pt.EdgeTess[1] = 3; // Top edge
    pt.EdgeTess[2] = 3; // Right edge
    pt.EdgeTess[3] = 3; // Bottom edge
    pt.InsideTess[0] = 3; // u-axis (columns)
    pt.InsideTess[1] = 3; // v-axis (rows)
    return pt;
}
```

How much should you tessellate?

The following are some common metrics used to determine the amount to tessellate:

1. **Distance from the camera:** The further an object is from the eye, the less we will notice fine details; therefore, we can render a low-poly version of the object when it is far away, and tessellate more as it gets closer to the eye.
2. **Screen area coverage:** We can estimate the number of pixels an object covers on the screen. If this number is small, then we can render a low-poly version of the object. As its screen area coverage increases, we can tessellate more.
3. **Orientation:** The orientation of the triangle with respect to the eye is taken into consideration with the idea that triangles along silhouette edges will be more refined than other triangles.
4. **Roughness:** Rough surfaces with lots of details will need more tessellation than smooth surfaces. A roughness value can be precomputed by examining the surface textures, which can be used to decide how much to tessellate.



Some performance advice

1. If the tessellation factors are 1 (which basically means we are not really tessellating), consider rendering the patch without tessellation, as we will be wasting GPU overhead going through the tessellation stages when they are not doing anything.
2. For performance reasons related to GPU implementations, do not tessellate such that the triangles are so small they cover less than eight pixels.
3. Batch draw calls that use tessellation (i.e., turning tessellation on and off between draw calls is expensive).



Control Point Hull Shader

The control point hull shader inputs a number of control points and outputs a number of control points.

The control point hull shader is invoked once per control point output.

Example: change surface representations from an ordinary triangle (submitted to the pipeline with three control points) to a cubic Bézier triangle patch (a patch with ten control points). You can use the hull shader to augment the triangle to a higher order cubic Bézier triangle patch with 10 control points.

This strategy is the so-called *N-patches scheme* or *PN triangles scheme*

For our first demo, it will be a simple *pass-through* shader, where we just pass the control point through unmodified.

```
struct HullOut
{
    float3 PosL : POSITION;
};

[domain("quad")] patch type
[partitioning("integer")] vertices are added/removed only at integer tessellation factor values
[outputtopology("triangle_cw")] clockwise winding order
[outputcontrolpoints(4)] outputting 4 control points
[patchconstantfunc("ConstantHS")] the constant hull shader function name
[maxtessfactor(64.0f)] HullOut HS(InputPatch<VertexOut, 4> p,
    uint i : SV_OutputControlPointID,
    uint patchId : SV_PrimitiveID)
{
    HullOut hout;

    hout.PosL = p[i].PosL;

    return hout;
}
```

Control Point Hull Shader

```
HullOut HS(InputPatch<VertexOut, 4> p, uint i : SV_OutputControlPointID, uint patchId : SV_PrimitiveID)
```

The hull shader inputs all of the control points of the patch via the `InputPatch` parameter.

The system value `SV_OutputControlPointID` gives an index identifying the output control point the hull shader is working on.

The input patch could have 4 control points and the output patch could have sixteen control points

The control point hull shader attributes:

`[domain("quad")]` The patch type. Valid arguments are tri, quad, or isoline.

`[partitioning("integer")]` Specifies the subdivision mode of the tessellation. Integer or fractional_even/fractional_odd

integer: New vertices are added/removed only at integer tessellation factor values. The fractional part of a tessellation factor is ignored. This creates a noticeable “popping” when a mesh changes its tessellation level.

Fractional tessellation (fractional_even/fractional_odd): New vertices are added/removed at integer tessellation factor values, but “slide” in gradually based on the fractional part of the tessellation factor.

`[outputtopology("triangle_cw")]` The winding order of the triangles created via subdivision: `triangle_cw` or `triangle_ccw`, `line` for line tessellation

`[outputcontrolpoints(4)]` The number of times the hull shader executes, outputting one control point each time

`[patchconstantfunc("ConstantHS")]` A string specifying the constant hull shader function name

`[maxtessfactor(64.0f)]` the maximum tessellation factor. This can potentially enable optimizations by the hardware if it knows this upper bound, as it will know how much resources are needed for the tessellation.

THE DOMAIN SHADER

The tessellation stage outputs all of our newly created vertices and triangles.

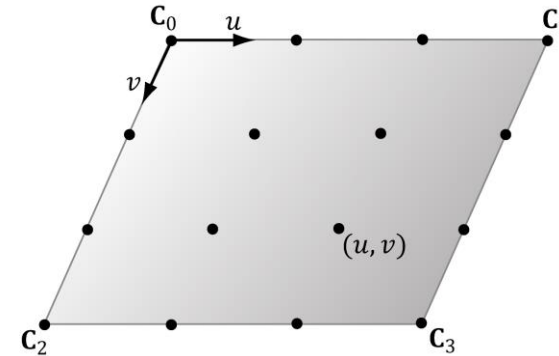
The domain shader is invoked for each new vertex.

With tessellation enabled, whereas the vertex shader acts as a vertex shader for each control point, the hull shader is essentially the vertex shader for the tessellated patch.

For a quad patch, the domain shader inputs the tessellation factors, the parametric (u, v) coordinates of the tessellated vertex positions, and all the patch control points output from the control point hull shader.

The domain shader does not give you the actual tessellated vertex positions; instead it gives you the parametric (u, v) coordinates. It is up to you to use these parametric coordinates and the control points to derive the actual 3D vertex positions.

The tessellation of a quad patch with 4 control points generating 16 vertices in the normalized uv -space, with coordinates in $[0, 1]^2$.



THE DOMAIN SHADER

The domain shader for a triangle patch is similar, except that instead of the parametric (u, v) values being input, the float3 barycentric (u, v, w) coordinates of the vertex are input.

The reason for outputting barycentric coordinates for triangle patches is probably due to the fact that Bézier triangle patches are defined in terms of barycentric coordinates.

```
struct DomainOut
{
    float4 PosH : SV_POSITION;
};

// The domain shader is called for every vertex created by the tessellator.
// It is like the vertex shader after tessellation.
[domain("quad")]
DomainOut DS(PatchTess patchTess,
             float2 uv : SV_DomainLocation,
             const OutputPatch<HullOut, 4> quad)
{
    DomainOut dout;

    // Bilinear interpolation.
    float3 v1 = lerp(quad[0].PosL, quad[1].PosL, uv.x);
    float3 v2 = lerp(quad[2].PosL, quad[3].PosL, uv.x);
    float3 p = lerp(v1, v2, uv.y);

    // Displacement mapping
    p.y = 0.3f*( p.z*sin(p.x) + p.x*cos(p.z) );

    float4 posW = mul(float4(p, 1.0f), gWorld);
    dout.PosH = mul(posW, gViewProj);

    return dout;
}
```

TESSELLATING A QUAD

In BasicTessellation demo, we submit a quad patch to the rendering pipeline, tessellate it based on the distance from the camera, and displace the generated vertices by a mathematic function that is similar to the one we have been using for this “hills” in our past demos.

Our vertex buffer storing the four control points is created like so:

```
void BasicTessellationApp::BuildQuadPatchGeometry()
{
    std::array<XMFLOAT3,4> vertices =
    {
        XMFLOAT3(-10.0f, 0.0f, +10.0f),
        XMFLOAT3(+10.0f, 0.0f, +10.0f),
        XMFLOAT3(-10.0f, 0.0f, -10.0f),
        XMFLOAT3(+10.0f, 0.0f, -10.0f)
    };

    std::array<std::int16_t, 4> indices = { 0, 1, 2, 3 };


    const UINT vbByteSize = (UINT)vertices.size() * sizeof(Vertex);
    const UINT ibByteSize = (UINT)indices.size() * sizeof(std::uint16_t);

    auto geo = std::make_unique<MeshGeometry>();
    geo->Name = "quadpatchGeo";
```

BasicTessellationApp::BuildRenderItems()

```
void BasicTessellationApp::BuildRenderItems()
{
    auto quadPatchRitem = std::make_unique<RenderItem>();
    quadPatchRitem->World = MathHelper::Identity4x4();
    quadPatchRitem->TexTransform = MathHelper::Identity4x4();
    quadPatchRitem->ObjCBIndex = 0;
    quadPatchRitem->Mat = mMaterials["whiteMat"].get();
    quadPatchRitem->Geo = mGeometries["quadpatchGeo"].get();
    quadPatchRitem->PrimitiveType = D3D_PRIMITIVE_TOPOLOGY_4_CONTROL_POINT_PATCHLIST;
    quadPatchRitem->IndexCount = quadPatchRitem->Geo->DrawArgs["quadpatch"].IndexCount;
    quadPatchRitem->StartIndexLocation = quadPatchRitem->Geo->DrawArgs["quadpatch"].StartIndexLocation;
    quadPatchRitem->BaseVertexLocation = quadPatchRitem->Geo->DrawArgs["quadpatch"].BaseVertexLocation;
    mRitemLayer[(int)RenderLayer::Opaque].push_back(quadPatchRitem.get());

    mAllRitems.push_back(std::move(quadPatchRitem));
}
```



The hull shader.

we now determine the tessellation factors based on the distance from the eye. Use a low-poly mesh in the distance, and increase the tessellation (and hence triangle count) as the mesh approaches the eye

```
struct VertexIn
{
    float3 PosL      : POSITION;
};

struct VertexOut
{
    float3 PosL      : POSITION;
};

VertexOut VS(VertexIn vin)
{
    VertexOut vout;
    vout.PosL = vin.PosL;
    return vout;
}

struct PatchTess
{
    float EdgeTess[4]    : SV_TessFactor;
    float InsideTess[2] : SV_InsideTessFactor;
};
```

```
PatchTess ConstantHS(InputPatch<VertexOut, 4> patch, uint patchID : SV_PrimitiveID)
{
    PatchTess pt;

    float3 centerL = 0.25f*(patch[0].PosL + patch[1].PosL + patch[2].PosL + patch[3].PosL);
    float3 centerW = mul(float4(centerL, 1.0f), gWorld).xyz;

    float d = distance(centerW, gEyePosW);

    // Tessellate the patch based on distance from the eye such that
    // the tessellation is 0 if d >= d1 and 64 if d <= d0. The interval
    // [d0, d1] defines the range we tessellate in.

    const float d0 = 20.0f;
    const float d1 = 100.0f;
    float tess = 64.0f*saturate( (d1-d)/(d1-d0) );

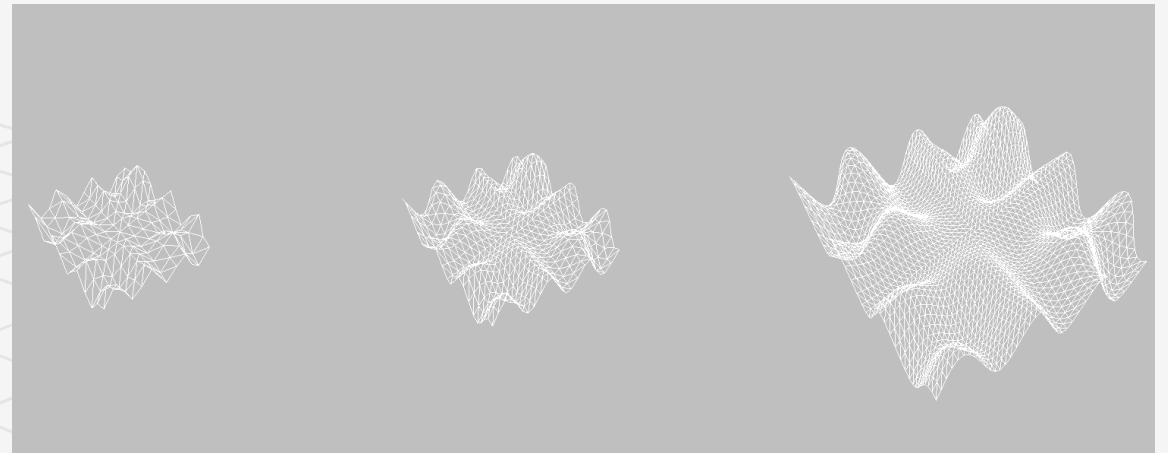
    // Uniformly tessellate the patch.

    pt.EdgeTess[0] = tess;
    pt.EdgeTess[1] = tess;
    pt.EdgeTess[2] = tess;
    pt.EdgeTess[3] = tess;

    pt.InsideTess[0] = tess;
    pt.InsideTess[1] = tess;

    return pt;
}

struct HullOut
{
    float3 PosL : POSITION;
};
```



CUBIC BÉZIER QUAD PATCHES

We describe cubic Bézier quad patches to show how surfaces are constructed via a higher number of control points.

Consider three noncollinear points \mathbf{p}_0 , \mathbf{p}_1 , and \mathbf{p}_2 which we will call the control points.

These three control points define a Bézier curve in the following way.

A point $\mathbf{p}(t)$ on the curve is first found by linearly interpolating between \mathbf{p}_0 and \mathbf{p}_1 by t , and \mathbf{p}_1 and \mathbf{p}_2 by t to get the intermediate points:

$$\mathbf{p}_0' = (1 - t)\mathbf{p}_0 + t\mathbf{p}_1$$

$$\mathbf{p}_1' = (1 - t)\mathbf{p}_1 + t\mathbf{p}_2$$

In other words, this construction by repeated interpolation leads to the parametric formula for a quadratic (degree 2) Bézier curve:

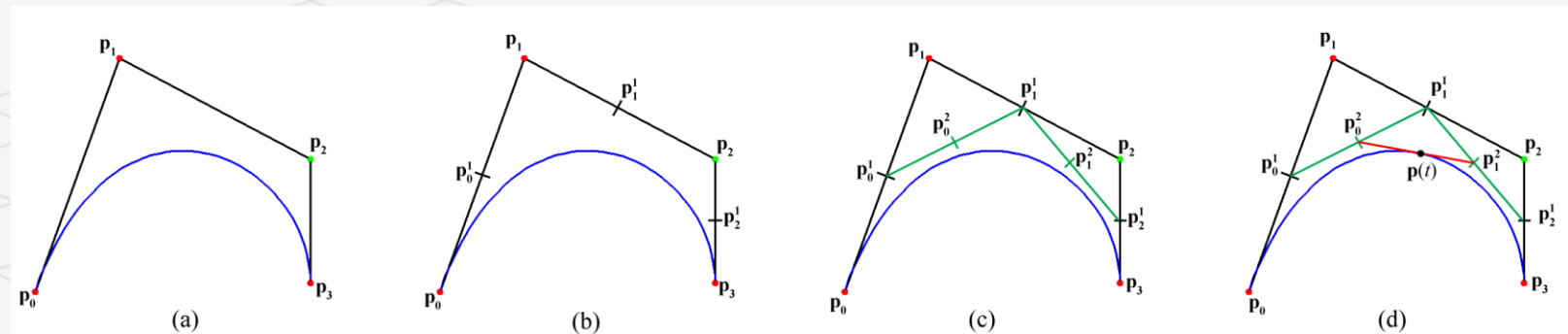
Repeated linear interpolation defined points on the cubic Bézier curve. The figure uses $t = 0.5$.

(a) The four control points and the curve they define.

(b) Linearly interpolate between the control points to calculate the first generation of intermediate points.

(c) Linearly interpolate between the first generation intermediate points to get the second generation intermediate points.

(d) Linearly interpolate between the second generation intermediate points to get the point on the curve.



Cubic Bézier Surface Evaluation Code

we give code to evaluate a cubic Bézier surface. To help understand the code that follows, we expand out the summation notation:

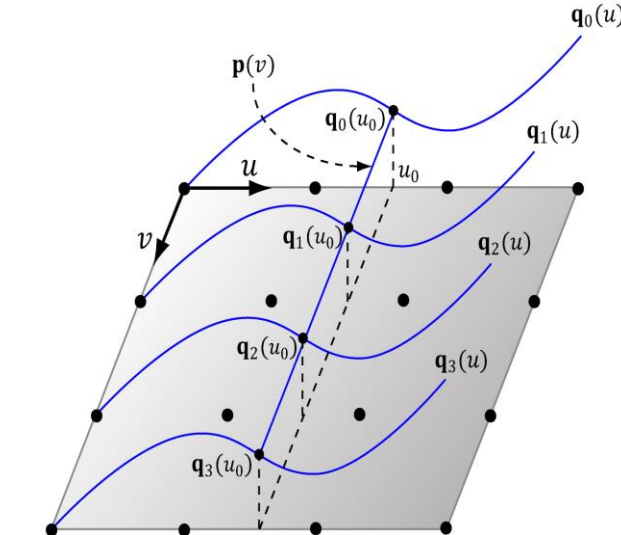
$$\mathbf{q}_0(u) = B_0^3(u)\mathbf{p}_{0,0} + B_1^3(u)\mathbf{p}_{0,1} + B_2^3(u)\mathbf{p}_{0,2} + B_3^3(u)\mathbf{p}_{0,3}$$

$$\mathbf{q}_1(u) = B_0^3(u)\mathbf{p}_{1,0} + B_1^3(u)\mathbf{p}_{1,1} + B_2^3(u)\mathbf{p}_{1,2} + B_3^3(u)\mathbf{p}_{1,3}$$

$$\mathbf{q}_2(u) = B_0^3(u)\mathbf{p}_{2,0} + B_1^3(u)\mathbf{p}_{2,1} + B_2^3(u)\mathbf{p}_{2,2} + B_3^3(u)\mathbf{p}_{2,3}$$

$$\mathbf{q}_3(u) = B_0^3(u)\mathbf{p}_{3,0} + B_1^3(u)\mathbf{p}_{3,1} + B_2^3(u)\mathbf{p}_{3,2} + B_3^3(u)\mathbf{p}_{3,3}$$

$$\begin{aligned}\mathbf{p}(u,v) &= B_0^3(v)\mathbf{q}_0(u) + B_1^3(v)\mathbf{q}_1(u) + B_2^3(v)\mathbf{q}_2(u) + B_3^3(v)\mathbf{q}_3(u) \\ &= B_0^3(v)\left[B_0^3(u)\mathbf{p}_{0,0} + B_1^3(u)\mathbf{p}_{0,1} + B_2^3(u)\mathbf{p}_{0,2} + B_3^3(u)\mathbf{p}_{0,3}\right] \\ &\quad + B_1^3(v)\left[B_0^3(u)\mathbf{p}_{1,0} + B_1^3(u)\mathbf{p}_{1,1} + B_2^3(u)\mathbf{p}_{1,2} + B_3^3(u)\mathbf{p}_{1,3}\right] \\ &\quad + B_2^3(v)\left[B_0^3(u)\mathbf{p}_{2,0} + B_1^3(u)\mathbf{p}_{2,1} + B_2^3(u)\mathbf{p}_{2,2} + B_3^3(u)\mathbf{p}_{2,3}\right] \\ &\quad + B_3^3(v)\left[B_0^3(u)\mathbf{p}_{3,0} + B_1^3(u)\mathbf{p}_{3,1} + B_2^3(u)\mathbf{p}_{3,2} + B_3^3(u)\mathbf{p}_{3,3}\right]\end{aligned}$$



Tessellation.hlsl

we pass the evaluated basis function values to CubicBezierSum.

This enables us to use CubicBezierSum for evaluating both $\mathbf{p}(u, v)$ and the partial derivatives, as the summation form is the same, the only differencing being the basis functions.

```
float4 BernsteinBasis(float t)
{
    float invT = 1.0f - t;

    return float4( invT * invT * invT,
                   3.0f * t * invT * invT,
                   3.0f * t * t * invT,
                   t * t * t );
}
```

```
float3 CubicBezierSum(const OutputPatch<HullOut, 16> bezpatch, float4 basisU, float4 basisV)
{
    float3 sum = float3(0.0f, 0.0f, 0.0f);
    sum = basisV.x * (basisU.x*bezpatch[0].PosL + basisU.y*bezpatch[1].PosL + basisU.z*bezpatch[2].PosL +
    basisU.w*bezpatch[3].PosL);
    sum += basisV.y * (basisU.x*bezpatch[4].PosL + basisU.y*bezpatch[5].PosL + basisU.z*bezpatch[6].PosL +
    basisU.w*bezpatch[7].PosL);
    sum += basisV.z * (basisU.x*bezpatch[8].PosL + basisU.y*bezpatch[9].PosL + basisU.z*bezpatch[10].PosL +
    basisU.w*bezpatch[11].PosL);
    sum += basisV.w * (basisU.x*bezpatch[12].PosL + basisU.y*bezpatch[13].PosL + basisU.z*bezpatch[14].PosL +
    basisU.w*bezpatch[15].PosL);

    return sum;
}

float4 dBernsteinBasis(float t)
{
    float invT = 1.0f - t;

    return float4( -3 * invT * invT,
                   3 * invT * invT - 6 * t * invT,
                   6 * t * invT - 3 * t * t,
                   3 * t * t );
}

[domain("quad")]
DomainOut DS(PatchTess patchTess,
              float2 uv : SV_DomainLocation,
              const OutputPatch<HullOut, 16> bezPatch)
{
    DomainOut dout;

    float4 basisU = BernsteinBasis(uv.x);
    float4 basisV = BernsteinBasis(uv.y);

    float3 p = CubicBezierSum(bezPatch, basisU, basisV);

    float4 posW = mul(float4(p, 1.0f), gWorld);
    dout.PosH = mul(posW, gViewProj);

    return dout;
}
```


Defining the Patch Geometry

```
void BezierPatchApp::BuildQuadPatchGeometry()
{
    std::array<XMFLOAT3,16> vertices =
    {
        // Row 0
        XMFLOAT3(-10.0f, -10.0f, +15.0f),
        XMFLOAT3(-5.0f,  0.0f, +15.0f),
        XMFLOAT3(+5.0f,  0.0f, +15.0f),
        XMFLOAT3(+10.0f, 0.0f, +15.0f),
        // Row 1
        XMFLOAT3(-15.0f, 0.0f, +5.0f),
        XMFLOAT3(-5.0f,  0.0f, +5.0f),
        XMFLOAT3(+5.0f, 20.0f, +5.0f),
        XMFLOAT3(+15.0f, 0.0f, +5.0f),
        // Row 2
        XMFLOAT3(-15.0f, 0.0f, -5.0f),
        XMFLOAT3(-5.0f,  0.0f, -5.0f),
        XMFLOAT3(+5.0f,  0.0f, -5.0f),
        XMFLOAT3(+15.0f, 0.0f, -5.0f),
        // Row 3
        XMFLOAT3(-10.0f, 10.0f, -15.0f),
        XMFLOAT3(-5.0f,  0.0f, -15.0f),
        XMFLOAT3(+5.0f,  0.0f, -15.0f),
        XMFLOAT3(+25.0f, 10.0f, -15.0f)
    };
    std::array<std::int16_t, 16> indices =
    {
        0, 1, 2, 3,
        4, 5, 6, 7,
        8, 9, 10, 11,
        12, 13, 14, 15
    };
};
```

