II. Direct3D Foundations
14. The Tessellation Stages

Game Graphic Programming

Kyung Hee University

Software Convergence

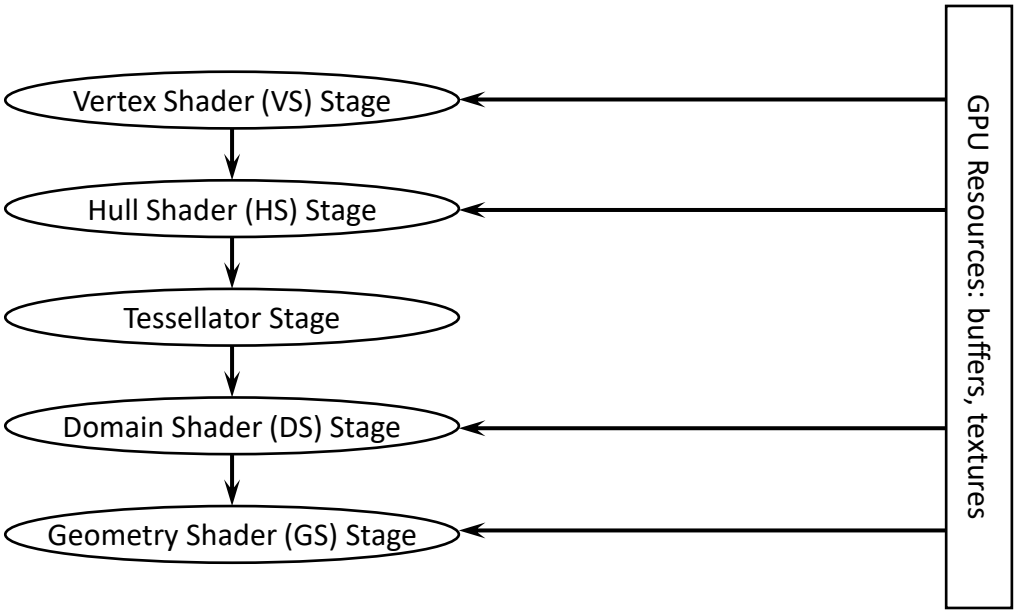
Prof. Daeho Lee

Tessellation (1)

- Three reasons for tessellation.
  - Dynamic LOD on the GPU.
    - We can dynamically adjust the detail of a mesh based on its distance from the camera and other factors.
    - For example, if a mesh is very far away, it would be wasteful to render a high-poly version of it, as we would not be able to see all that detail anyway.
    - As the object gets closer to the camera, we can continuously increase tessellation to increase the detail of the object.
  - Physics and animation efficiency.
    - We can perform physics and animation calculations on the low-poly mesh, and then tessellate to the higher-polygon version. This saves computation power by performing the physics and animation calculations at a lower frequency.
  - 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.

Tessellation (2)

- A subset of the rendering pipeline showing the tessellation stages.



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.
- Direct3D support patches with 1-32 control points, and these are described by the following primitive types:
  - **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,**
- A triangle can be thought of a triangle patch with three control points (**D3D_PRIMITIVE_3_CONTROL_POINT_PATCH**), so you can still submit your usual triangle meshes to be tessellated. A simple quad patch can be submitted with four control points (**D3D_PRIMITIVE_4_CONTROL_POINT_PATCH**).
- These patches are eventually tessellated into triangles by the tessellation stages

Hull Shader

- Hull shader actually consists of two shaders:
  - Constant hull shader: path constant function of hull shader
  - Control point hull shader: hull shader

```
PatchTess ConstantHS(InputPatch<VertexOut, 4> patch, uint
patchID : SV_PrimitiveID) {
    // …
}
[domain("quad")]
[partitioning("integer")]
[outputtopology("triangle_cw")]
[outputcontrolpoints(4)]
[patchconstantfunc("ConstantHS")]
[maxtessfactor(64.0f)]
HullOut HS(InputPatch<VertexOut, 4> p,
           uint i : SV_OutputControlPointID,
           uint patchId : SV_PrimitiveID) {
    // …
}
```

Constant Hull Shader (1)

- Constant hull shader
  - Constant hull shader is evaluated per patch, and is tasked with outputting the so-called tessellation factors of the mesh.
    - The tessellation factors instruct the tessellation stage how much to tessellate the patch.
    - Input
      - All the control points of the patch: control points are the output of the vertex shader (e.g., **`InputPatch<VertexOut, 4>`**)
      - The system also provides a patch ID value via the **`SV_PrimitiveID`** semantic that can be used if needed; the ID uniquely identifies the patches in a draw call.
    - Output
      - The tessellation factors (**`SV_TessFactor`** and **`SV_InsideTessFactor`**)
      - The tessellation factors depend on the topology of the patch.
      - Tessellating a quad patch
        - Four edge tessellation factors control how much to tessellate along each edge.
        - Two interior tessellation factors indicate how to tessellate the quad patch (horizontal dimension and vertical dimension of the quad).
      - Tessellating a triangle patch
        - Three edge tessellation factors control how much to tessellate along each edge.
        - One interior tessellation factor

Constant Hull Shader (2)

```
// a quad patch with four control point
struct PatchTess {
  float EdgeTess[4]   : SV_TessFactor;
  float InsideTess[2] : SV_InsideTessFactor;
};
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;
}
```

Control Point Hull Shader (1)

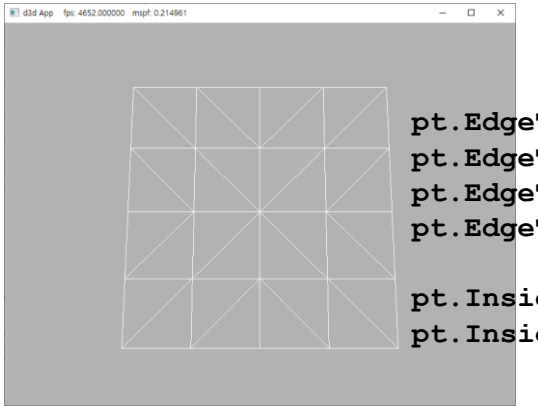
- The control point hull shader inputs a number of control points and outputs a number of control points.
- The control point hull shader is invoked once per control point output.

```
struct HullOut {
   float3 PosL : POSITION;
};
[domain("quad")]
[partitioning("integer")]
[outputtopology("triangle_cw")]
[outputcontrolpoints(4)]
[patchconstantfunc("ConstantHS")]
[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;
}
```

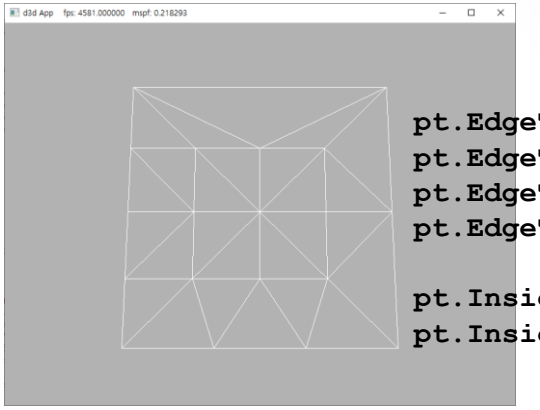
Control Point Hull Shader (2)

- Attributes
  - **`domain`**: The patch type. Valid arguments are **`tri`**, **`quad`**, or **`isoline`**.
  - **`partitioning`**: Specifies the subdivision mode of the tessellation.
    - integer: New vertices are added/removed only at integer tessellation factor values.
    - 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`**: The winding order of the triangles created via subdivision.
    - **`triangle_cw`**: clockwise winding order.
    - **`triangle_ccw`**: counterclockwise winding order.
    - **`line`**: For line tessellation.
  - **`outputcontrolpoints`**: The number of times the hull shader executes, outputting one control point each time.
  - **`patchconstantfunc`**: A string specifying the constant hull shader function name.
  - **`maxtessfactor`**: A hint to the driver specifying the maximum tessellation factor your shader uses.

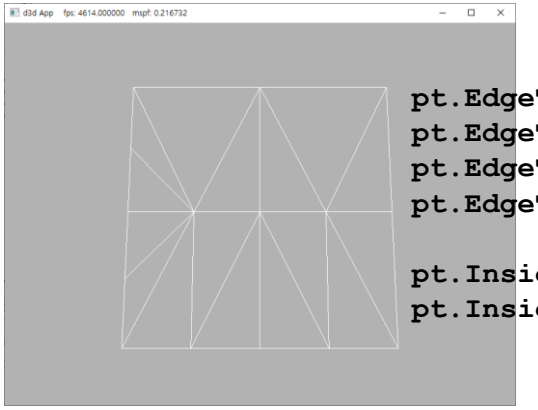
Quad Patch Tessellation Examples



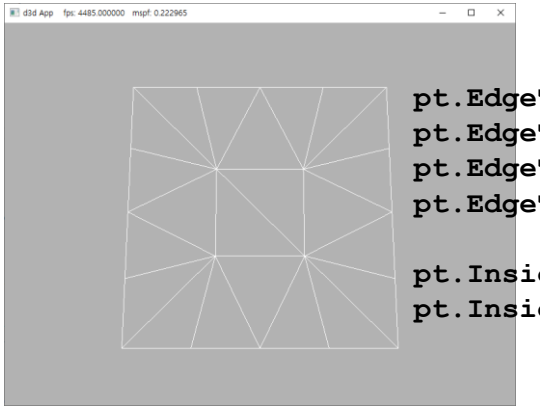
```
pt.EdgeTess[0] = 4;
pt.EdgeTess[1] = 4;
pt.EdgeTess[2] = 4;
pt.EdgeTess[3] = 4;

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



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

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

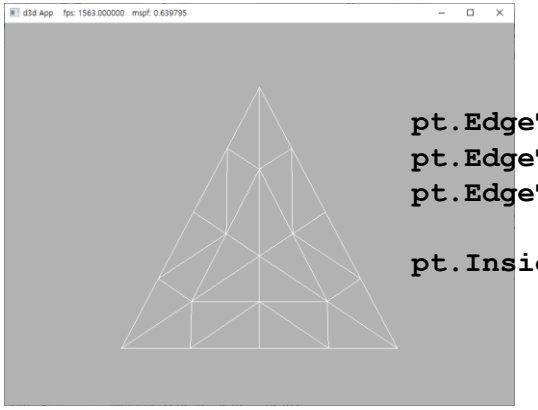


```
pt.EdgeTess[0] = 2;
pt.EdgeTess[1] = 2;
pt.EdgeTess[2] = 4;
pt.EdgeTess[3] = 4;

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

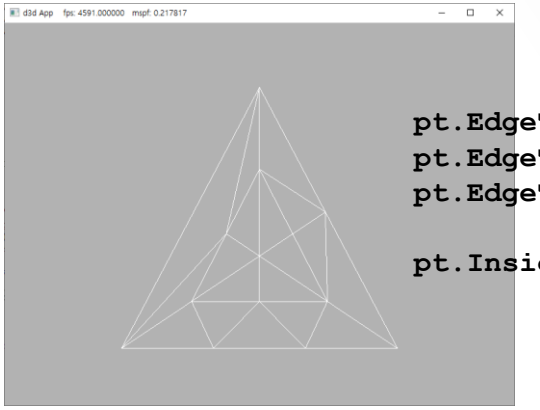


```
pt.EdgeTess[0] = 4;
pt.EdgeTess[1] = 4;
pt.EdgeTess[2] = 4;
pt.EdgeTess[3] = 4;

pt.InsideTess[0] = 3;
pt.InsideTess[1] = 3;
```

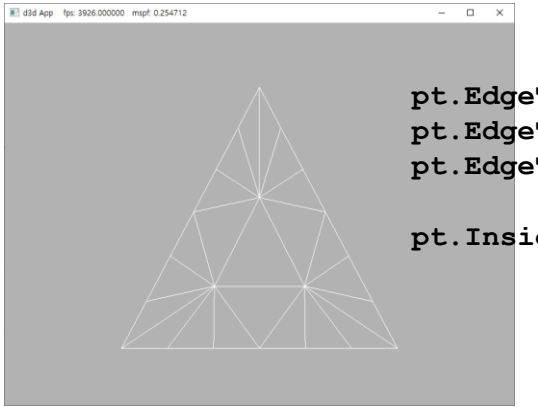
Triangle Patch Tessellation Examples



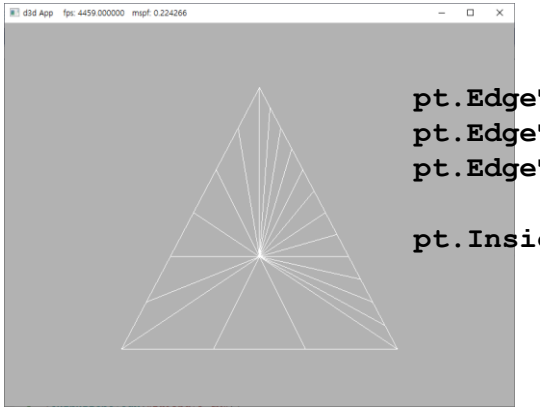
```
pt.EdgeTess[0] = 4;
pt.EdgeTess[1] = 4;
pt.EdgeTess[2] = 4;

pt.InsideTess = 4;
```



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

pt.InsideTess = 4;
```



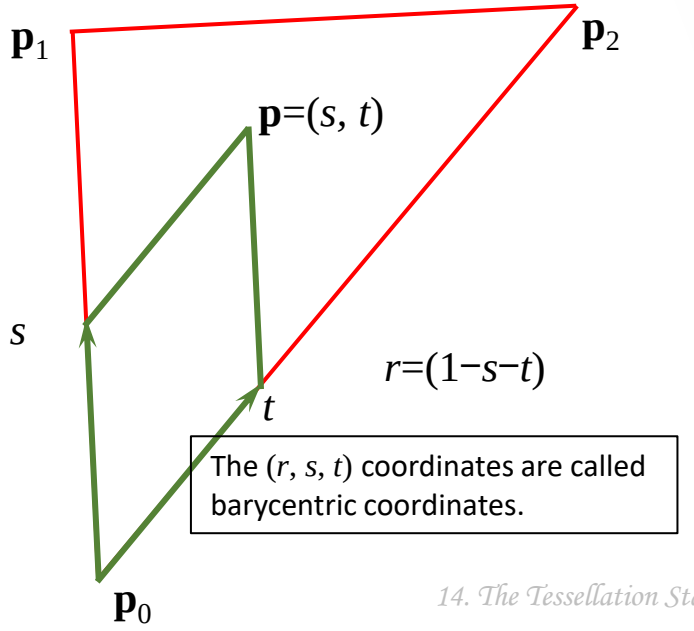
```
pt.EdgeTess[0] = 6;
pt.EdgeTess[1] = 6;
pt.EdgeTess[2] = 6;

pt.InsideTess = 3;
```



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

pt.InsideTess = 1;
```

The Domain Shader (1)

- The tessellation stage outputs all of our newly created vertices.
- The domain shader is invoked for each vertex created by the tessellation stage.
  - For a quad patch, the domain shader inputs the tessellation factors (and any other per patch information you output from the constant hull shader), the parametric (u, v) coordinates of the tessellated vertex positions.
  - For a triangle path, the **`float3`** barycentric (u, v, w) coordinates of the vertex are input.



The Domain Shader (2)

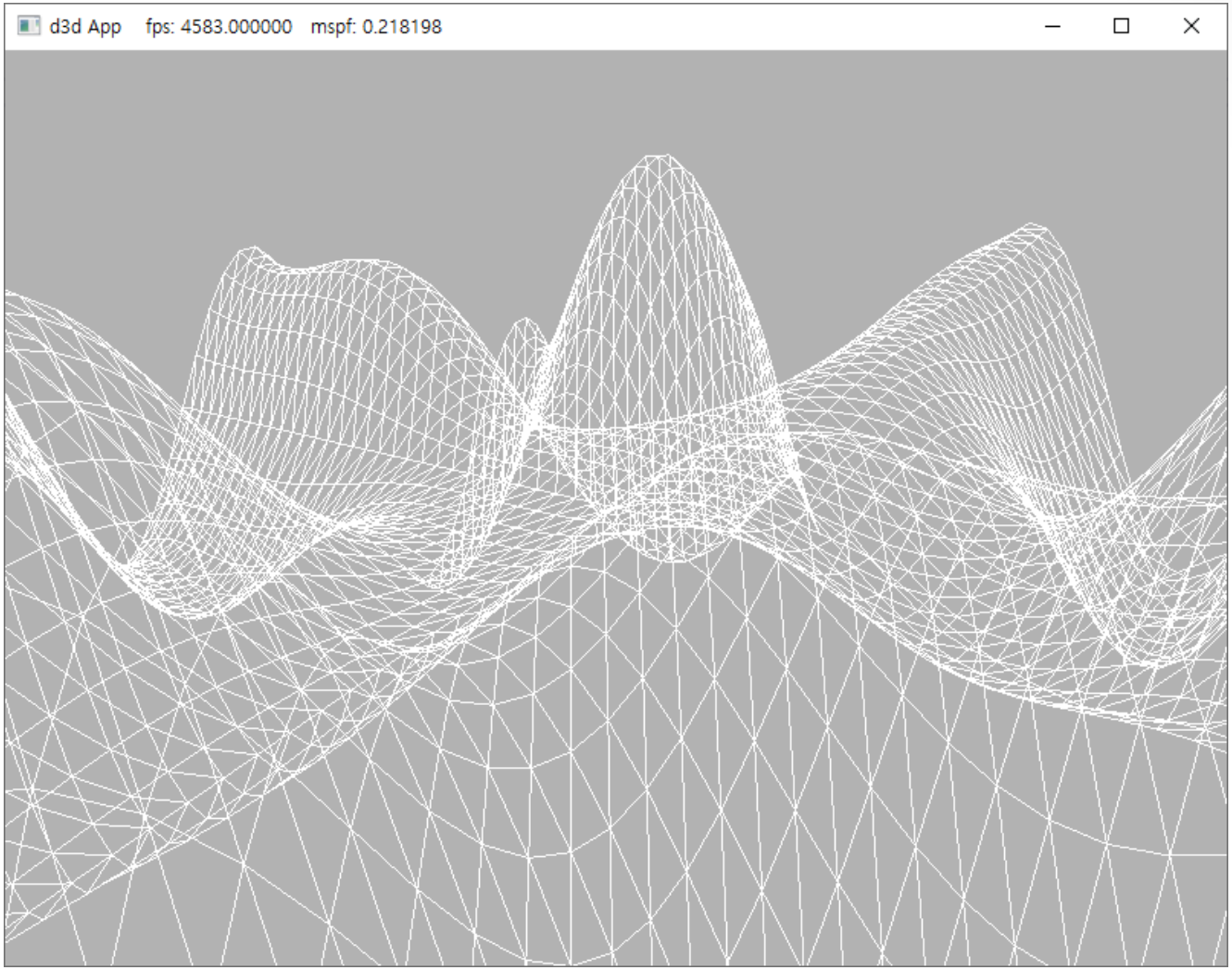
```
struct DomainOut {
    float4 PosH : SV_POSITION;
};

[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;
}
```


Kyung Hee University
nize@khu.ac.kr

The Domain Shader (3)

```
float3 BarycentricInterpolate(float3 v0, float3 v1, float3 v2,
   float3 barycentric) {
  return barycentric.z * v0 + barycentric.x * v1 + barycentric.y * v2;
}

[domain("tri")]
DomainOut DS(PatchTess patchTess,
             float3 uvw : SV_DomainLocation,
             const OutputPatch<HullOut, 3> tri) {
   DomainOut dout;
   float3 p = BarycentricInterpolate(tri[0].PosL, tri[1].PosL,
                  tri[2].PosL, uvw);

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

   return dout;
}
```

Basic Tessellation Example (1)



Basic Tessellation Example (2)

```
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);
// …
}
void BasicTessellationApp::BuildRenderItems() {
    auto quadPatchRitem = std::make_unique<RenderItem>();
    quadPatchRitem->World = MathHelper::Identity4x4();
    quadPatchRitem->TexTransform = MathHelper::Identity4x4();
    quadPatchRitem->ObjCBIndex = 0;
    quadPatchRitem->PrimitiveType =
           D3D_PRIMITIVE_TOPOLOGY_4_CONTROL_POINT_PATCHLIST;
    // …

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

Basic Tessellation Example (3)

```
void BasicTessellationApp::BuildPSOs() {
    D3D12_GRAPHICS_PIPELINE_STATE_DESC opaquePsoDesc;
    ZeroMemory(&opaquePsoDesc,
       sizeof(D3D12_GRAPHICS_PIPELINE_STATE_DESC));
   opaquePsoDesc.InputLayout
       = {mInputLayout.data(), (UINT)mInputLayout.size() };
   opaquePsoDesc.pRootSignature = mRootSignature.Get();
   opaquePsoDesc.VS
       = {reinterpret_cast<BYTE*>(mShaders["tessVS"]->GetBufferPointer()),
       mShaders["tessVS"]->GetBufferSize()};
   opaquePsoDesc.HS
       = {reinterpret_cast<BYTE*>(mShaders["tessHS"]->GetBufferPointer()),
       mShaders["tessHS"]->GetBufferSize()};
   opaquePsoDesc.DS
       = {reinterpret_cast<BYTE*>(mShaders["tessDS"]->GetBufferPointer()),
       mShaders["tessDS"]->GetBufferSize()};
   opaquePsoDesc.PS
      =  {reinterpret_cast<BYTE*>(mShaders["tessPS"]->GetBufferPointer()),
       mShaders["tessPS"]->GetBufferSize()};
   opaquePsoDesc.PrimitiveTopologyType
       = D3D12_PRIMITIVE_TOPOLOGY_TYPE_PATCH;
// …
}
```