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 is the Vertex Processing stage in the DirectX rendering pipeline where patches of vertex data are subdivided into smaller Primitives.

This process is governed by two shader stages and a fixed-function stage.

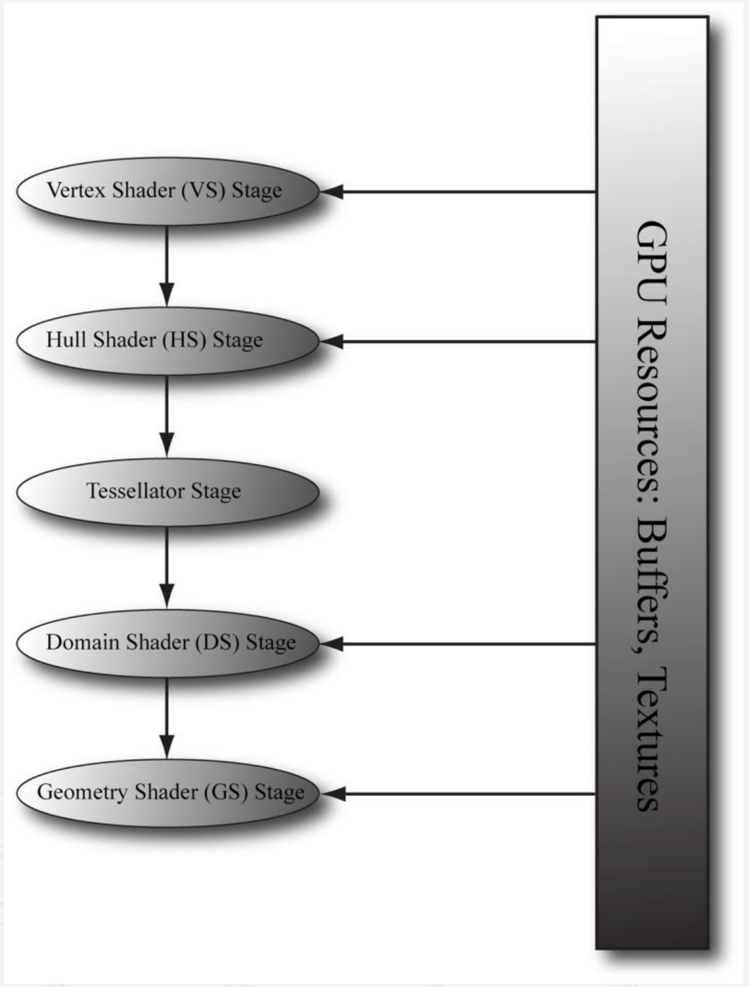
Increase the triangle count adds detail to the mesh

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.

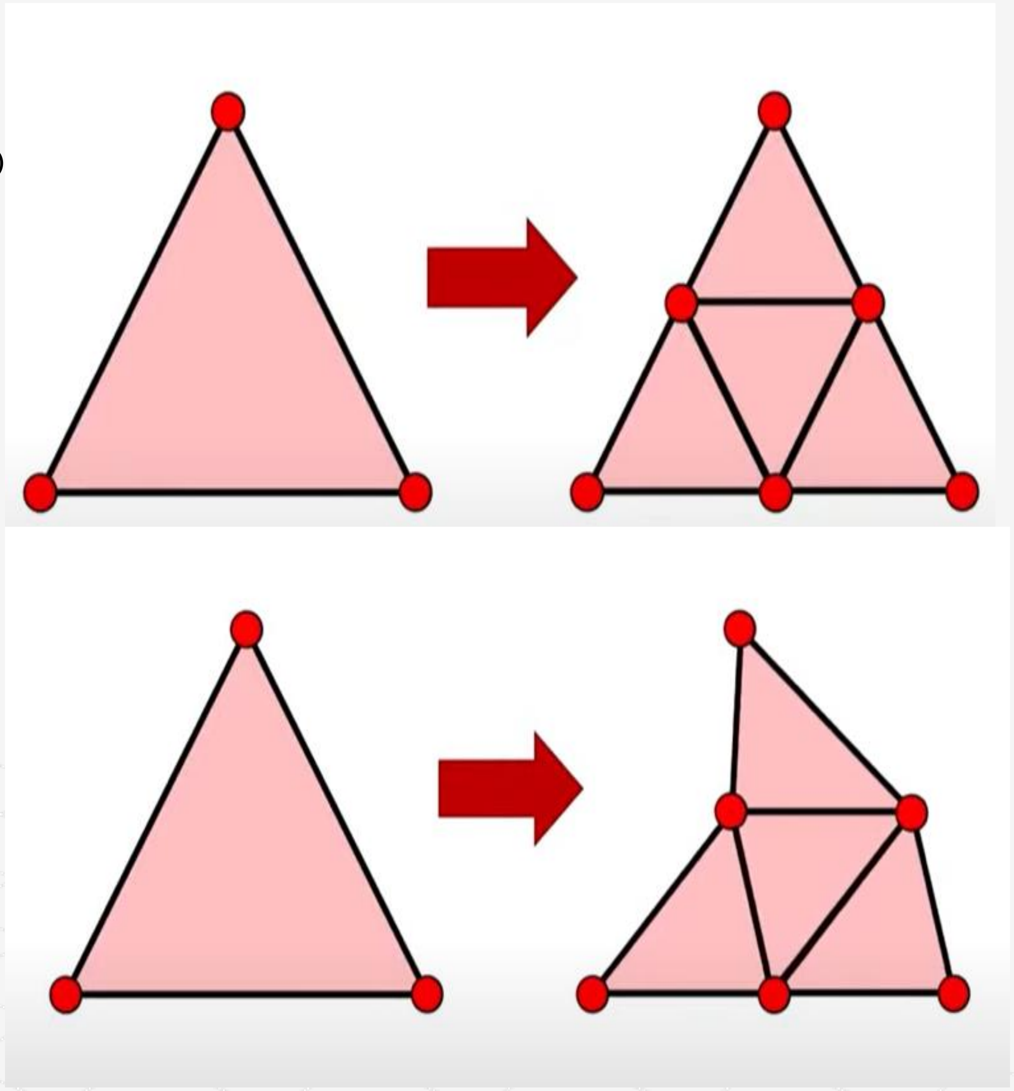
2. Physics and animation efficiency. We can perform physics and animation calculations on the low-poly mesh, and then tessellate to the higher polygon version.

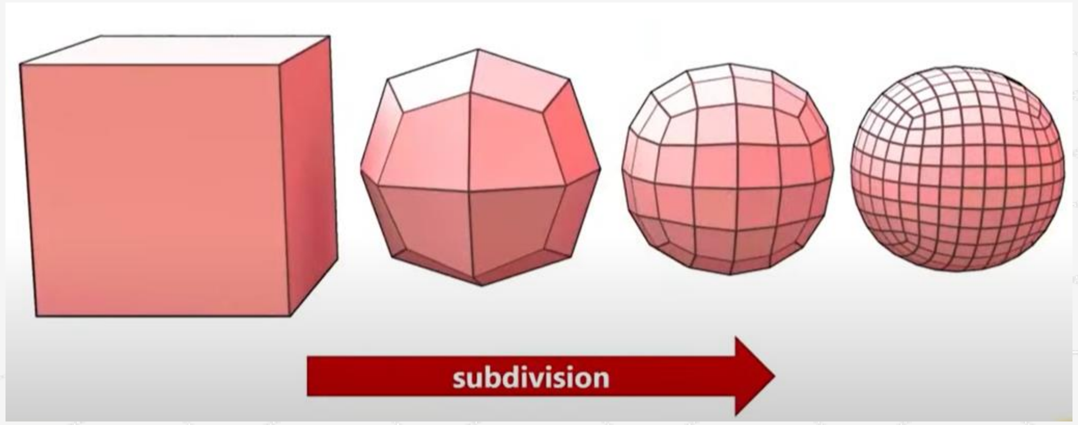
3. Memory savings. We can store lower polygon meshes in memory (on disk, RAM, and VRAM), and then have the GPU tessellate to the higher polygon version on the fly.



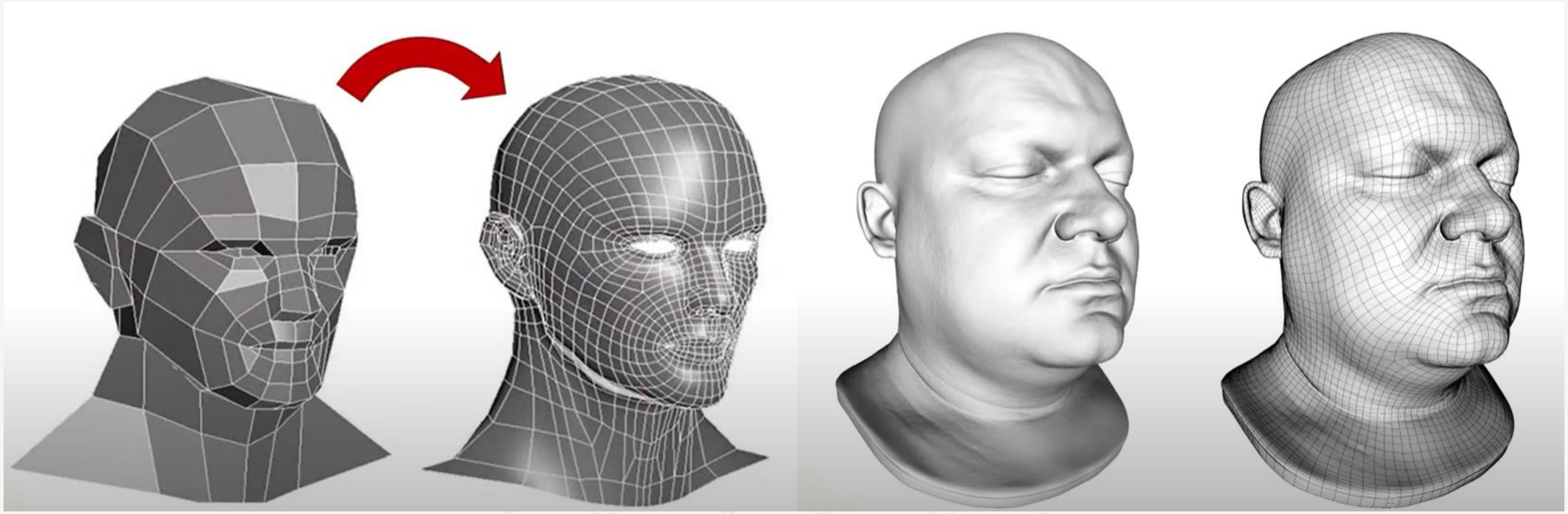
Why do we need a Tessellation step right in the pipeline?

1. You can perform adaptive subdivision based on a variety of criteria (size, curvature, etc.)
2. You can provide coarser models, but have finer ones displayed (≈ geometric compression)
3. You can apply detailed displacement maps without supplying equally detailed geometry
4. You can apply detailed normal maps without supplying equally detailed geometry
5. You can adapt visual quality to the required level of detail
6. You can create smoother silhouettes
7. You can do all of this, and someone else will supply the geometric patterns for you





Tessellation/Subdivision



New Pipeline Stages

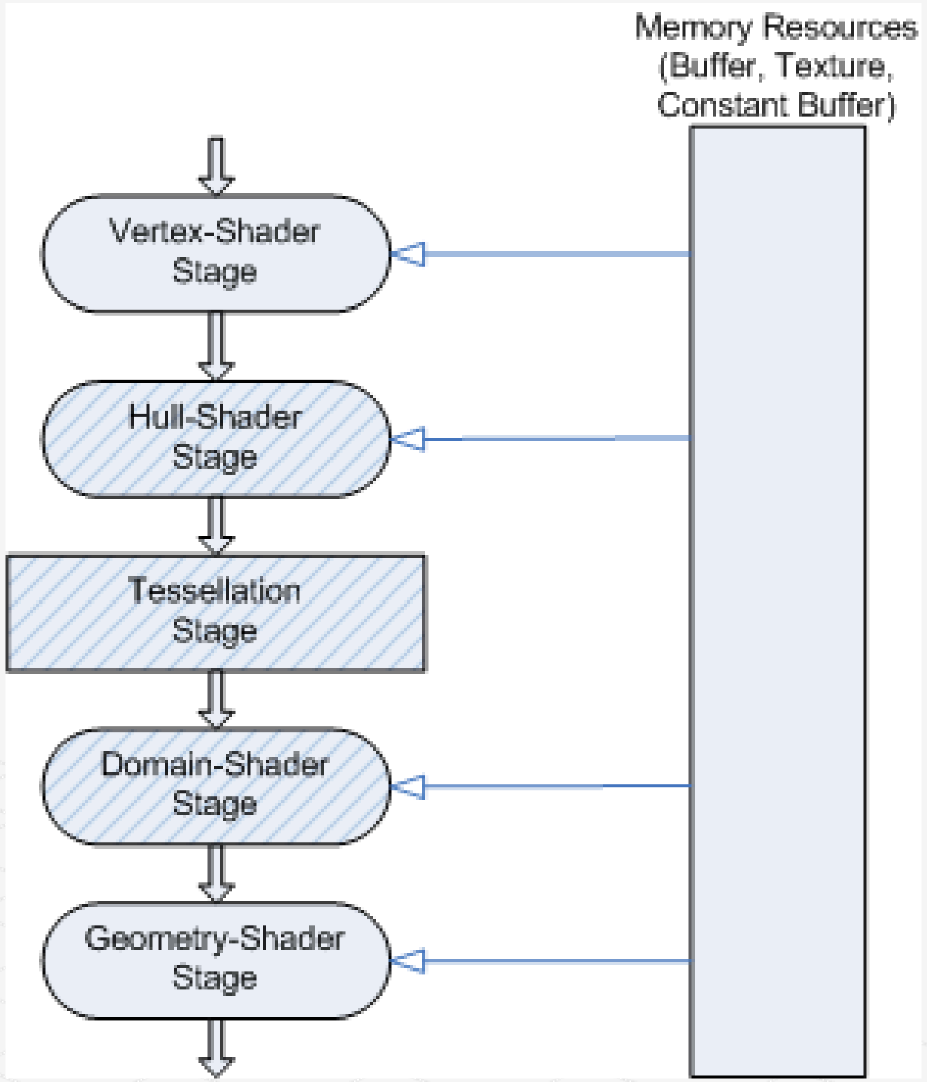
Generally, the process of tessellation involves subdividing a patch of some type, then computing new vertex values (position, color, texture coordinates, etc.) for each of the vertices generated by this process. Each stage of the tessellation pipeline performs part of this process.

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.

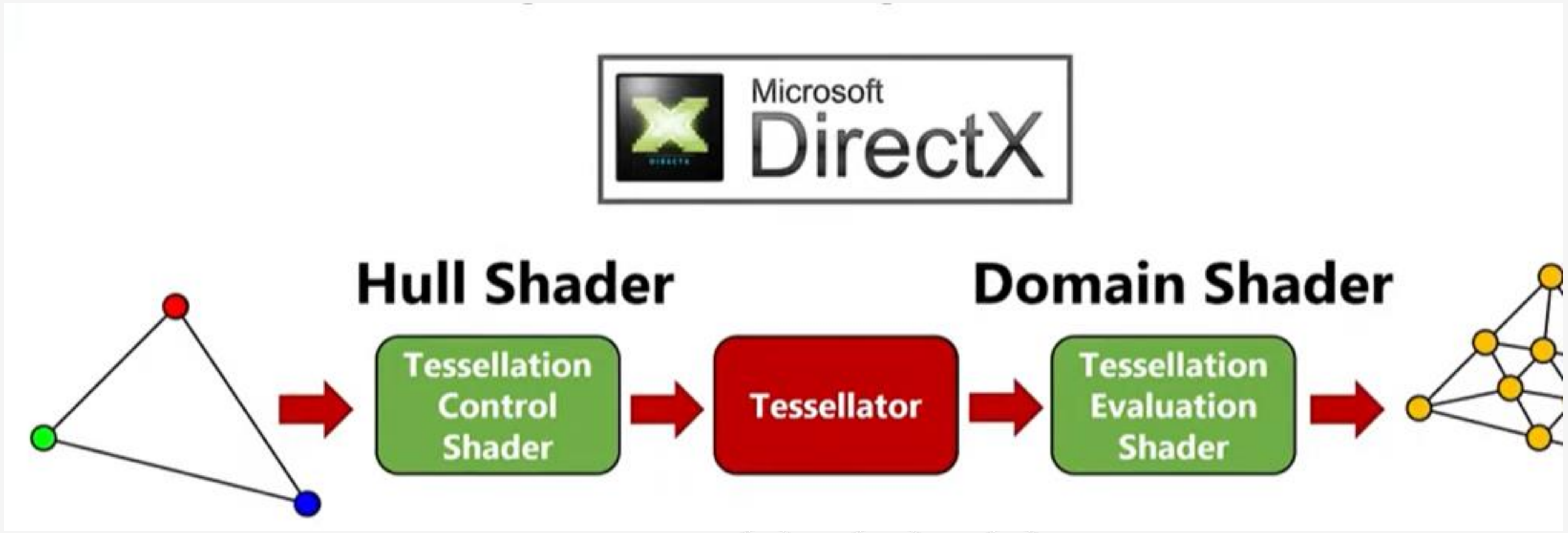
Hull-Shader Stage – A programmable shader stage that produces a geometry patch (and patch constants) that correspond to each input patch (quad, triangle, or line). HS determines how much tessellation to do along the edge or inside the patch.

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.



OpenGL (v4.0) vs. DirectX



Control Points

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

The Hull Shader works on a group of vertices called *Control Points* (CP).

The CPs don't have a well defined polygonal form such as a triangle, square, pentagon or whatever. Instead, they define a geometric surface.

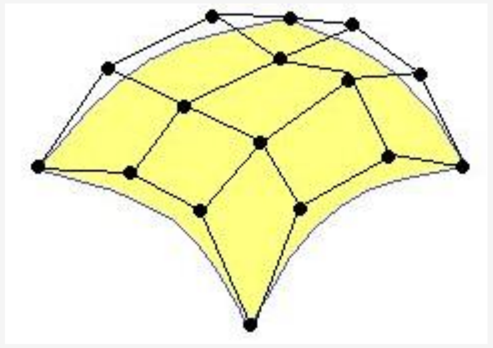
This surface is usually defined by some polynomial formula and the idea is that moving a CP has an effect on the entire surface.

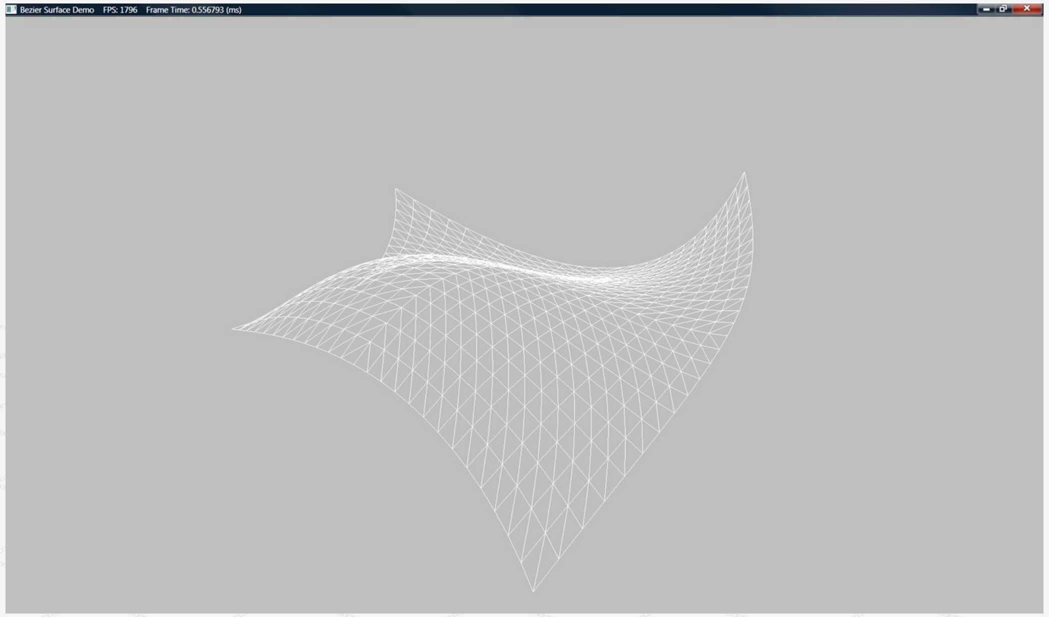
You are probably familiar with some graphic software that allows you to define surfaces or curves using a set of CPs and shape them by moving the CPs.

The group of CPs is usually called a *Patch*. The yellow surface in the following picture is defined by a patch with 16 CPs:

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.




Bezier Surface Demo   FPS: 1796   Frame Time: 0.556793 (ms)

The tessellation stages

The following diagram shows the progression through the tessellation stages.

The progression starts with the low-detail subdivision surface.

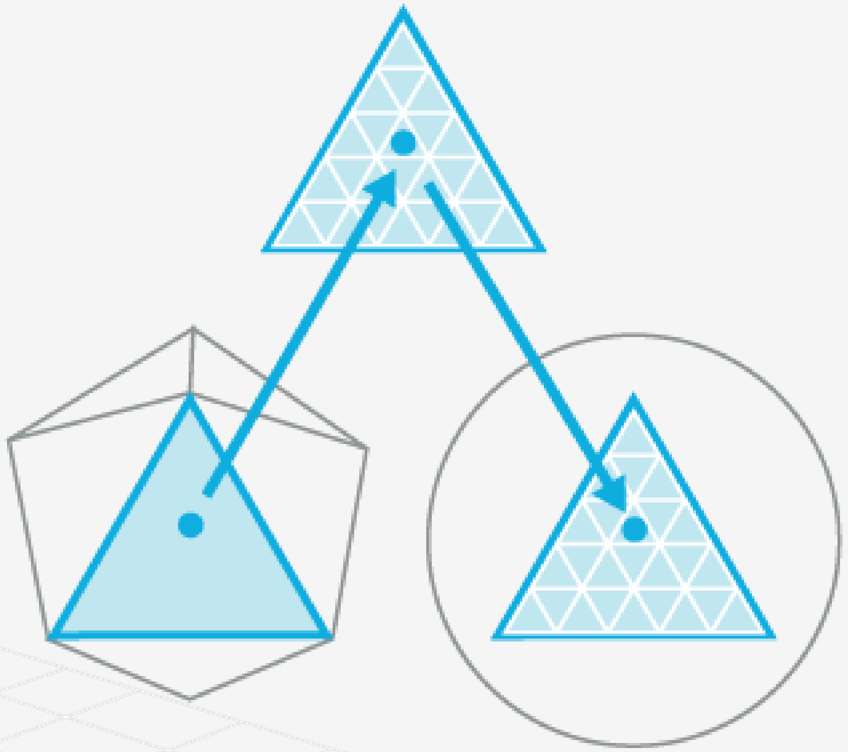
The Hull shader takes an input patch and emits an output patch. The developer has the option in the shader to do some transformation on the CPs or even add/delete CPs. In addition to the output patch the control shader calculates a set of numbers called *Tessellation Factors* (TF). The TFs determine the Tessellation level of detail - how many triangles to generate for the patch.

After the HS finishes, then it comes the fixed function Tessellation Stage whose job is to do the actual subdivision.

The thing is that the Tessellation Stage doesn't really subdivides the output patch of the HS. In fact, it doesn't even have access to it. Instead, it takes the Tessellation Factors and subdivides what is called a *Domain*.

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.



Triangle Domain

The “domain” mode controls the shape of the tessellated mesh that will be generated and fed through the domain shader.

The domain can either be a normalized (in the range of 0.0-1.0) square of 2D coordinates or an equilateral triangle defined by 3D barycentric coordinates.

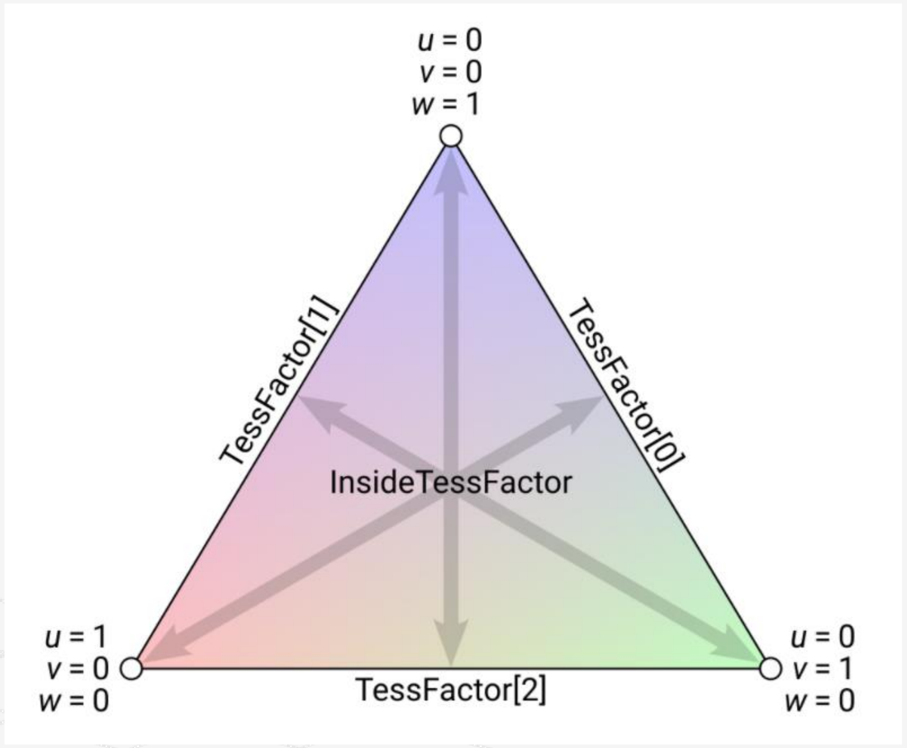
Barycentric coordinates of a triangle is a method of defining a location inside a triangle as a combination of the weight of the three vertices.

The triangle domain generates triangular output patches. It has three edge tess factors, which specify the number of segments that each edge of the triangle gets subdivided into. It has a single “inside” tess factor, which specifies the number of mesh segments from each edge to the opposite vertex. The domain shader receives three barycentric coordinates, which always sum to 1. The barycentrics are represented by colors (UVW = RGB) in the diagram below.

The vertices of the triangle as designated as U, V and W and as the location gets closer to one vertex its weight increases while the weight of the other vertices decreases.

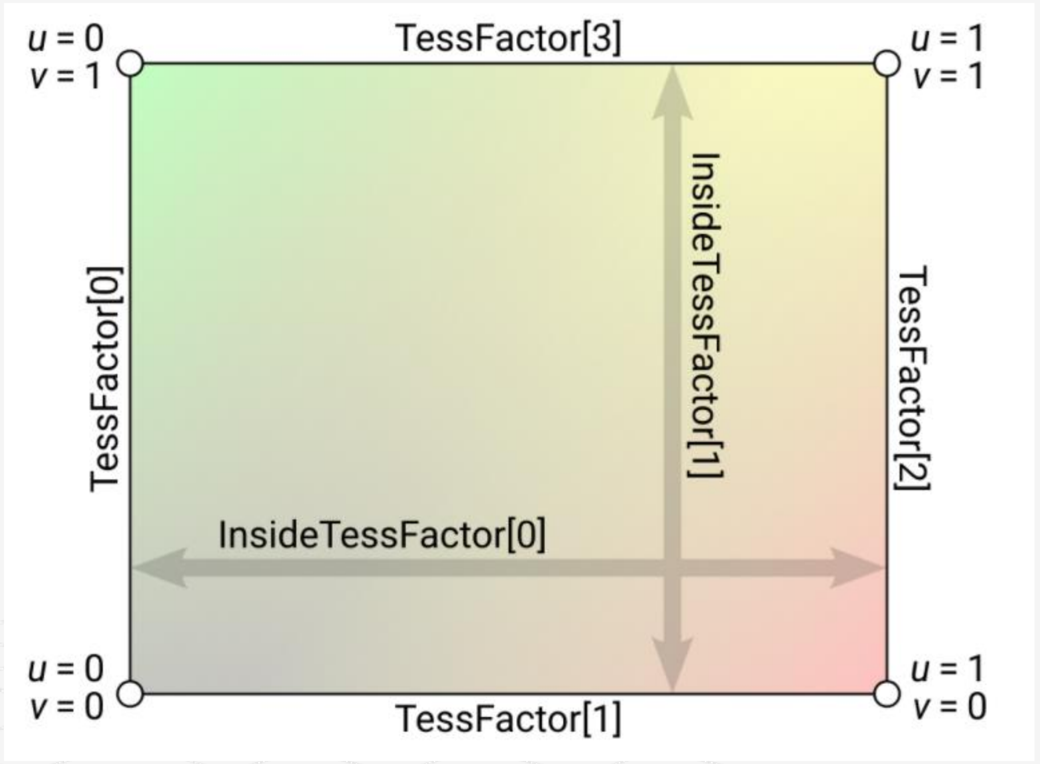
If the location is exactly on a vertex the weight of that vertex is 1 while the other two are zero. For example, the barycentric coordinate of U is (1,0,0), V is (0,1,0) and W is (0,0,1). The center of the triangle is on the barycentric coordinate of (1/3,1/3,1/3).

The interesting property of barycentric coordinates is that if we sum up the individual components of the barycentric coordinate of each and every point inside the triangle, we always get 1.



Quad Domain

The quad domain generates quadrilateral output patches; it has four edge tess factors, and two inside factors, which control the number of segments between pairs of opposite edges. The domain shader receives two-dimensional UV coordinates, represented as red and green in the diagram below.



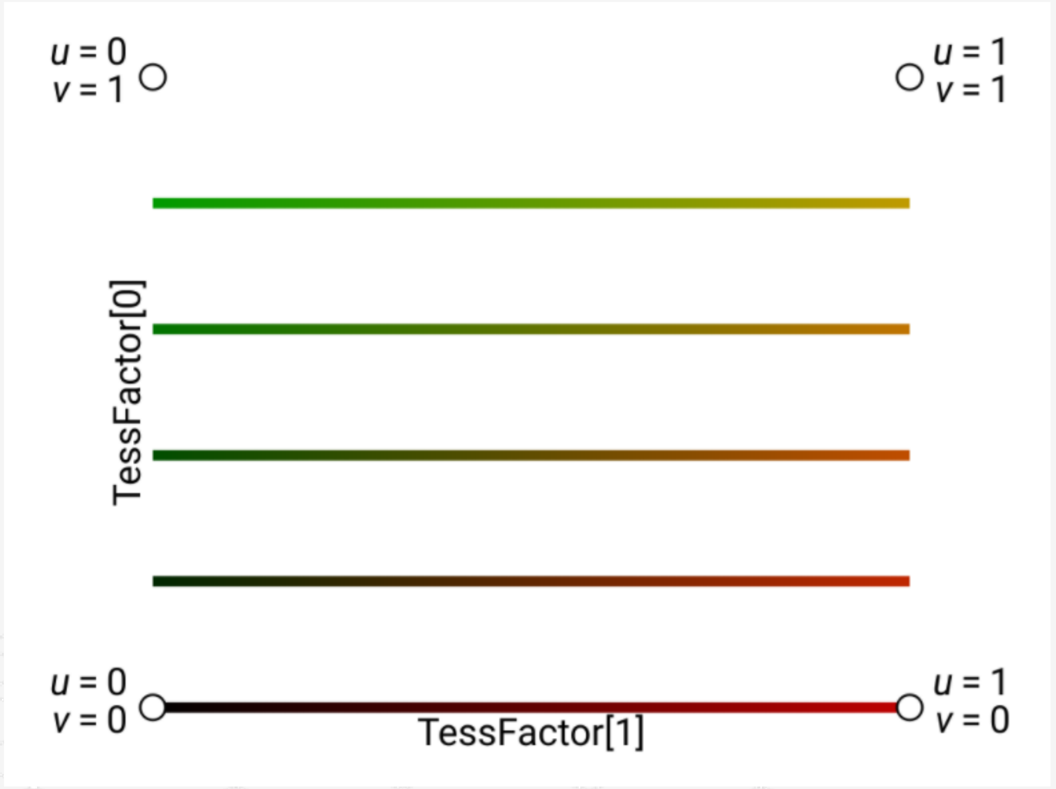
Isoline Domain

The isoline domain is an odder and less-used one.

Instead of producing triangles, it produces a set of line strips.

The line strips come in a quadrilateral shape, same as the quad domain, but they're subdivided along the U axis and discretely spaced along the V axis.

The isoline domain has only two edge tess factors (defined the same way as for the quad domain), and no inside factors.



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
```

THE TESSELLATION STAGE

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

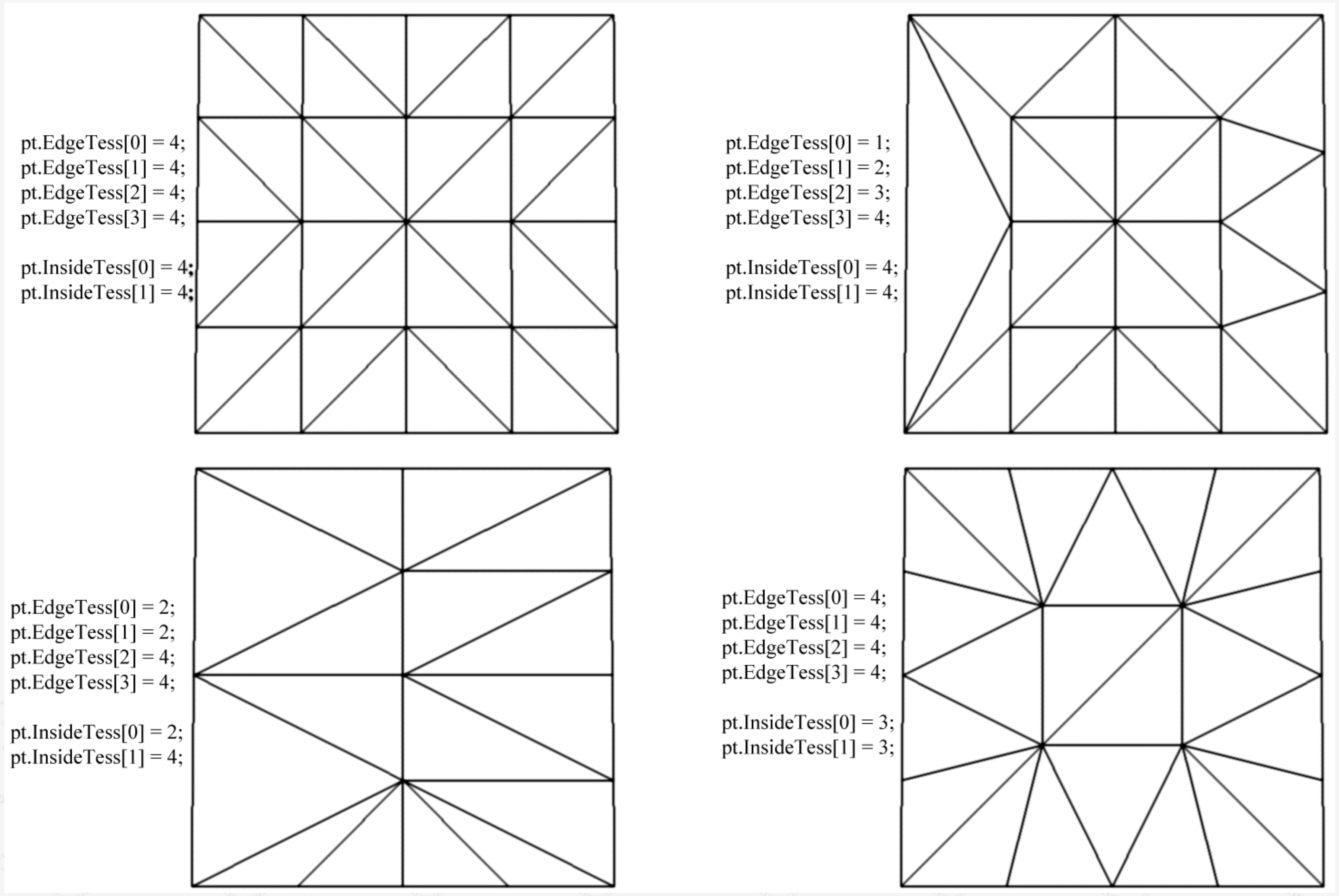
This stage is all done by the hardware, and tessellates the patches based on the tessellation factors output from the constant hull shader program.

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] = 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;
```

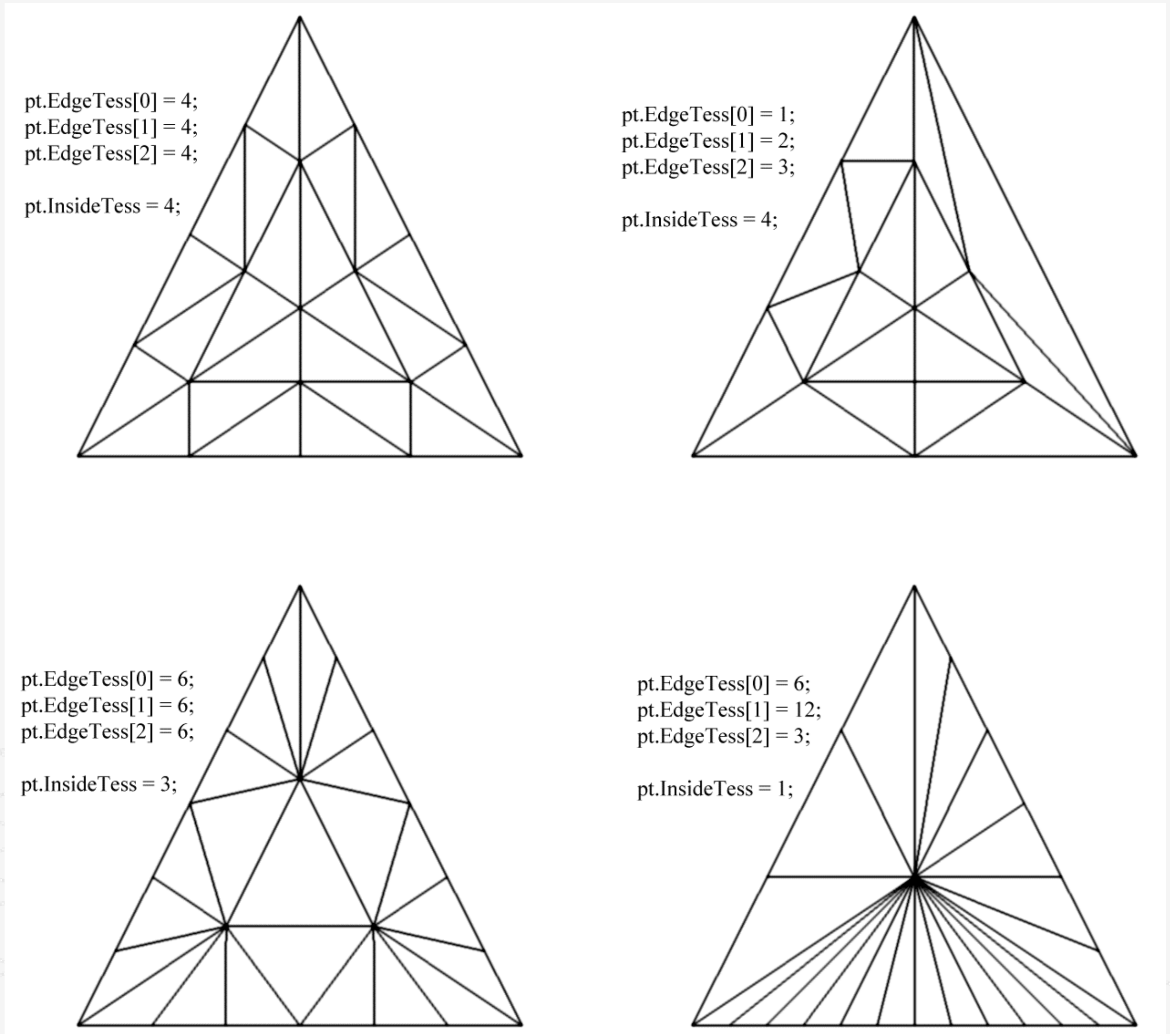
```
pt.EdgeTess[0] = 4;
pt.EdgeTess[1] = 4;
pt.EdgeTess[2] = 4;
pt.EdgeTess[3] = 4;

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

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.

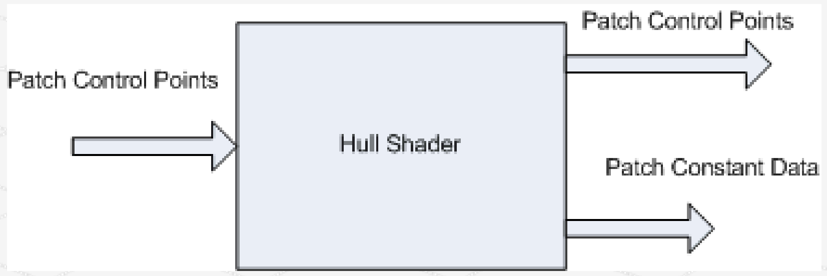
Tessellation factors determine how much to subdivide each patch.

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?

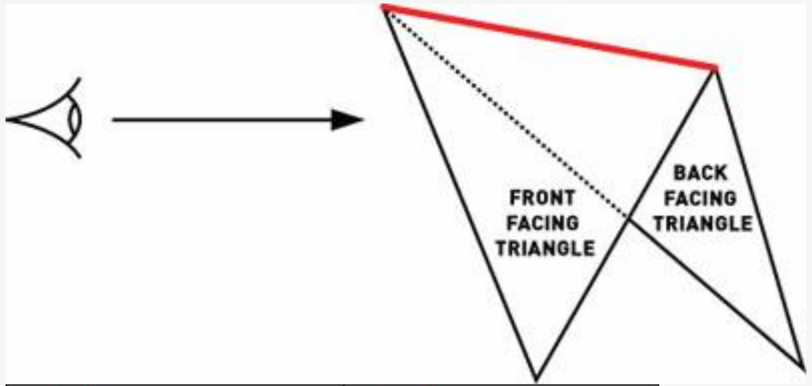
The following are some common metrics used to determine the amount to tessellate:

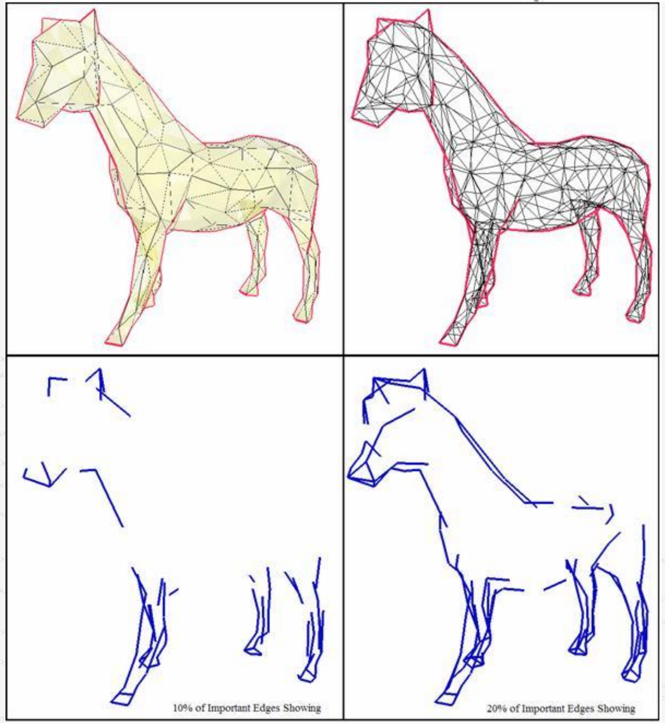
1. **Distance from the camera:** The further an object is from the eye, the less we will notice fine details; therefore, we can render a low-poly version of the object when it is far away, and tessellate more as it gets closer to the eye.

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.




10% of Important Edges Showing
20% of Important Edges Showing

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 is 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.

Partitioning Modes

Spacing modes (OpenGL) (actually called "partitioning" modes in D3D) affect the interpretation of the tessellation factors.

The tess factors are just the number of segments that a given edge or UV axis will be subdivided into, but there are three choices for the detailed behavior.

1. In integer spacing, also called equal spacing, fractional tess factors are rounded *up* to the nearest integer. The useful range is [1, 64]. There are no smooth transitions; subdivisions are always equally spaced in UV distance, and we just discretely pop from one to the next. https://www.youtube.com/watch?v=KXmV9o4VtOk

2. The fractional spacing modes provide smooth transitions between different subdivision levels, by morphing vertices around so that edges smoothly expand or collapse as the tess factors go up and down. In the case of fractional-odd mode, the number of segments is defined by rounding the tess factors up to the nearest odd integer, and the blend factor for vertex morphing is defined by how far you had to round. This mode matches integer spacing when you're exactly on an odd integer. The useful range is [1, 63].https://youtu.be/6sTI4yiAQEg

3. Fractional-even spacing is the same as fractional-odd, but using even integers instead. The useful range is [2, 64]. Note that the "identity" tess factor of 1 is not available in this mode! Everything always gets tessellated by at least a factor of 2. https://youtu.be/vtvvYEhRKcQ

THE DOMAIN SHADER

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

A domain shader is invoked once for each point generated by the fixed function tessellator.

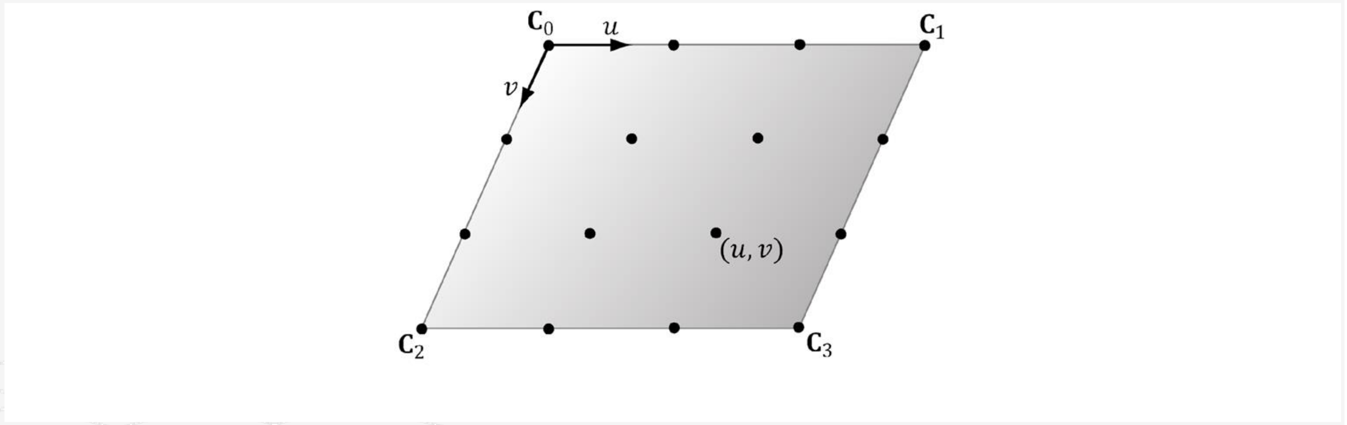
The inputs are the UV[W] parametric coordinates of the point on the patch, as well as all of the output data from the hull shader including control points and patch constants.

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.

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 for a Triangle Patch

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 triangle vertices are first processed by the Vertex Shader.

The Control Point Hull Shader receives the triangle as a patch with 3 control points and simply passes them through to the Domain Shader.

The constant hull shader sets the edges and inside tessellation factors.

The Tessellation stage subdivides an equilateral triangle into smaller triangles and executed the Domain Shader for every generated vertex.

In each Domain Shader invocation, we will access the barycentric coordinates (a.k.a Tessellation Coordinates) of the vertex in the 3D-float uvw.

Since the barycentric coordinates within a triangle represent a weight combination of the 3 vertices, we can use it to interpolate all the attributes of the new vertex.

The world space position for each new vertex is calculated for each new vertex.

The homogenous position of each new vertex is calculated by multiplying the world space position of new vertex by the view projection matrix.

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.

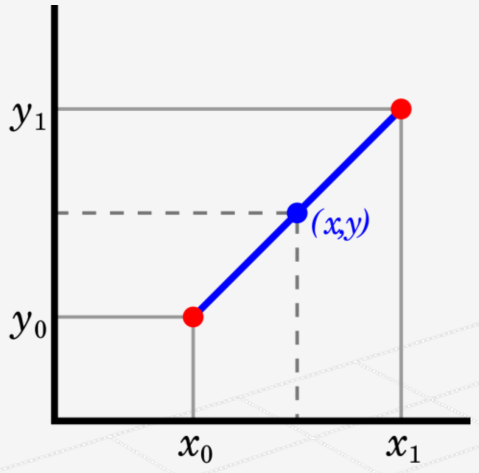
```
[domain("tri")]
DomainOut DS(PatchTess patchTess,
float3 uvw : SV_DomainLocation,
const OutputPatch<HullOut, 3> tri)
{
DomainOut dout;

float3 p = tri[0].PosL * uvw.x + tri[1].PosL
* uvw.y + tri[2].PosL * uvw.z;

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

dout.Color = tri[0].Color;
return dout;
}
```

THE DOMAIN SHADER for a Quad patch

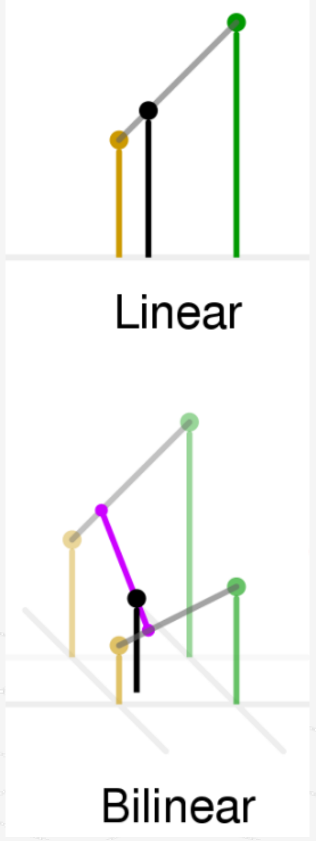
Lerp: Linear Interpolation

```
float3 lerp(float3 v0, float3 v1, float t)
 {
return (1 - t) * v0 + t * v1;
}
```

Given the two red points, the blue line is the linear interpolation between the points, and the value *y* at *x* can be found by linear interpolation.



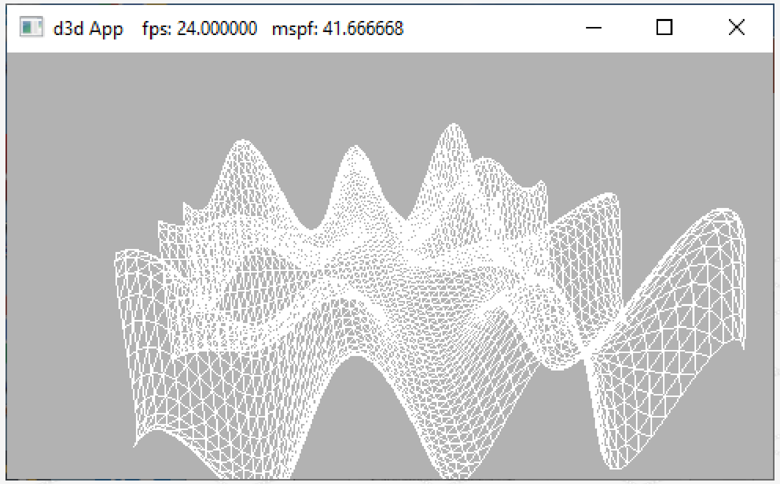
```
struct DomainOut
{
float4 PosH : SV_POSITION;
};

// 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 the 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 mathematical 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()

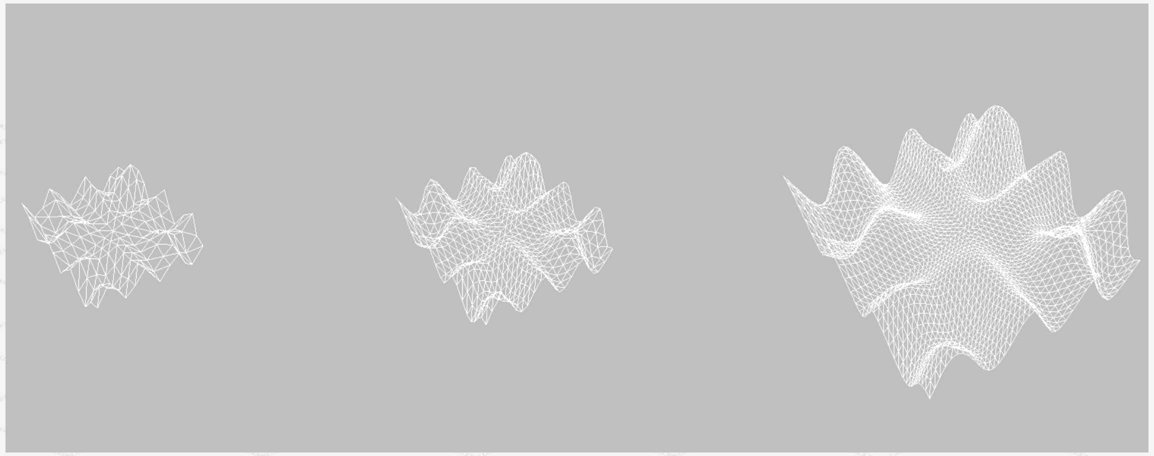
```
void BasicTessellationApp::BuildRenderItems()
{
auto quadPatchRitem = std::make_unique<RenderItem>();
quadPatchRitem->World = MathHelper::Identity4x4();
quadPatchRitem->TexTransform = MathHelper::Identity4x4();
quadPatchRitem->ObjCBIndex = 0;
quadPatchRitem->Mat = mMaterials["whiteMat"].get();
quadPatchRitem->Geo = 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

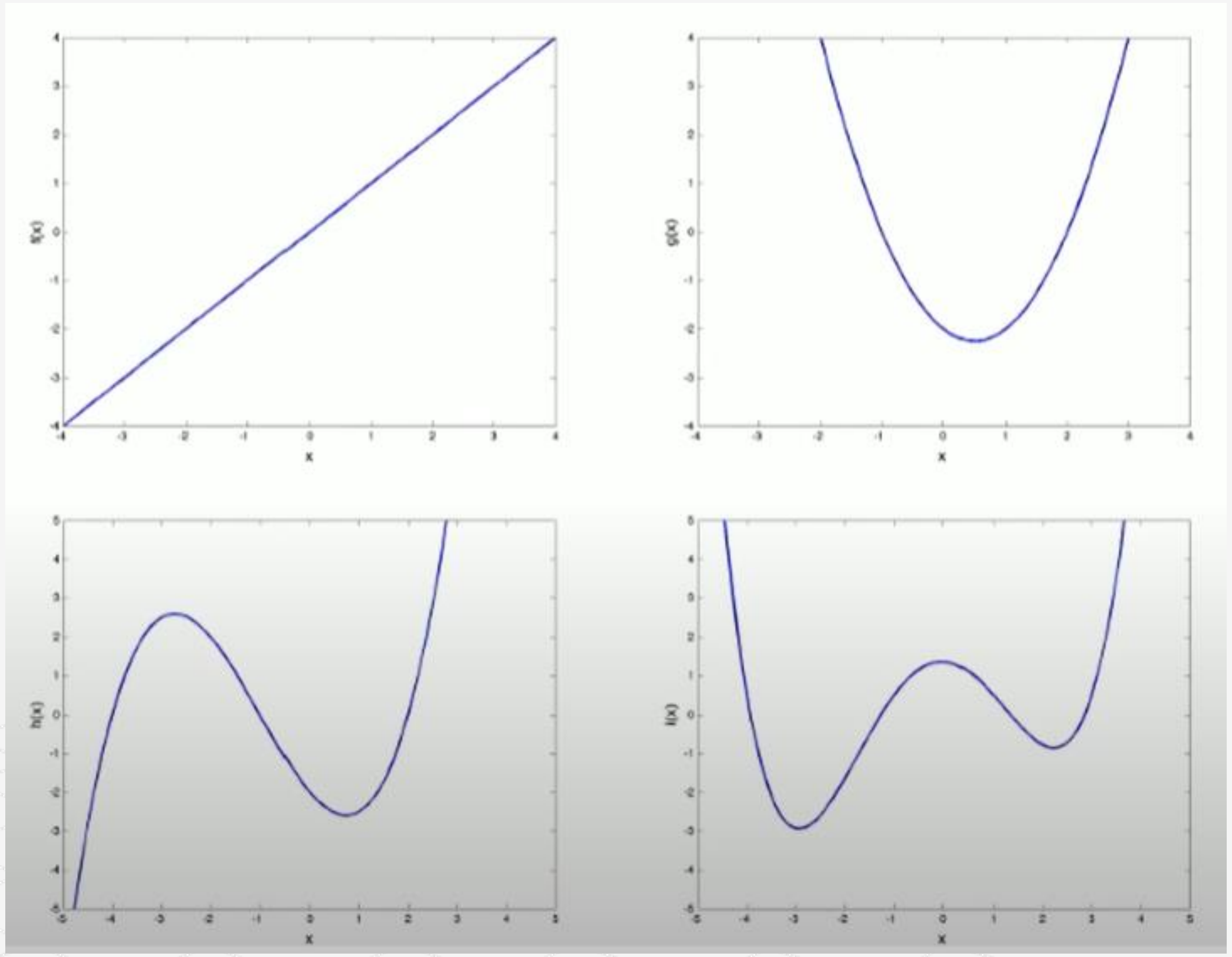
```
struct VertexIn
{
float3 PosL     : POSITION;
};
struct VertexOut
{
float3 PosL     : POSITION;
};
VertexOut VS(VertexIn vin)
{
VertexOut vout;
vout.PosL = vin.PosL;
return vout;
}
struct PatchTess
{
float EdgeTess[4]   : SV_TessFactor;
float InsideTess[2] : SV_InsideTessFactor;
};
```

```
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;
};
```



Polynomials

$f(x) = x + 1,$	linear
$g(x) = x^2 + x + 1,$	quadratic
$h(x) = x^3 + x^2 + x + 1,$	cubic
$i(x) = x^4 + x^3 + x^2 + x + 1.$	quartic



Bezier Curve

Bezier Curves are very common in programming.

Bezier curves are expressed as parametric equations!

They are used in graphics to create scalable vector graphics of curves, allowing the curve to remain smooth when scaled.

Scalable Vector Graphics (**SVG**) is an XML-based vector image format for two-dimensional graphics with support for interactivity and animation

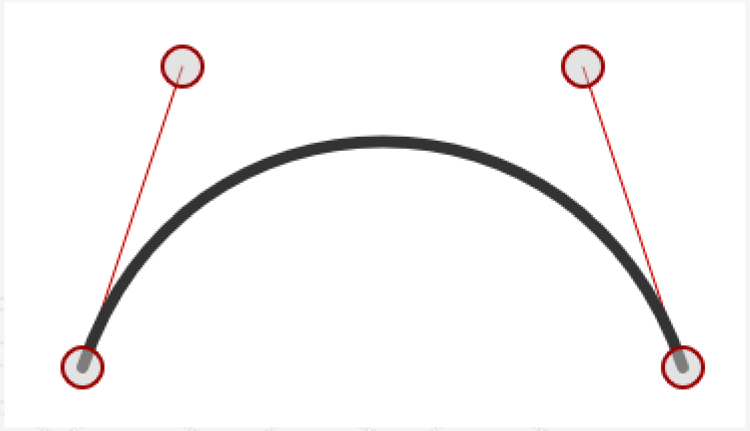
When using the HTML5 Canvas, they are commonly used to draw ellipses.

Try Week9/BezierPatch/Bezier.html

It shows how bézier curves can be drawn on a canvas element. Drag the line ends or the control points to change the curve.

To draw a Bezier Curve, you must know four points: the start point, the end point, and two other points called the control points.

The control points are used to control the curve. By adjusting the control points, you can change how much the curve, well, curves.

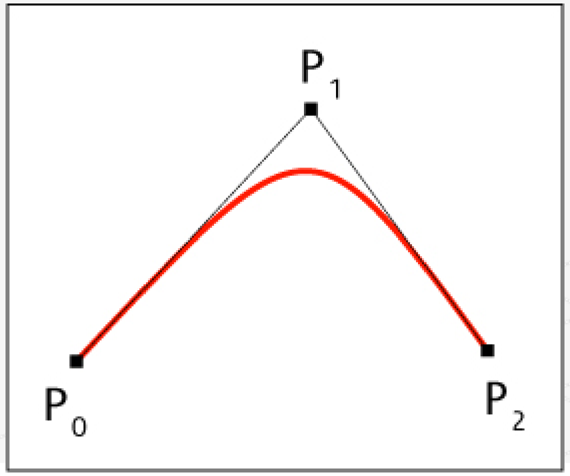


Quadratic Bezier Curve:

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

As a refresher, the formula for finding the midpoint of two points is a follows: $P_0^1 = \frac{(P_0 + P_1)}{2}$.

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.

The calculation first determines the midpoint of the start point P_0 and the first control point P_1, which gives us P_0^1. It then finds the midpoint of both control points P_1 and P_2, which gives us P_1^1.

$$\mathbf{p}_0^1 = (1-t)\mathbf{p}_0 + t\mathbf{p}_1$$

$$\mathbf{p}_1^1 = (1-t)\mathbf{p}_1 + t\mathbf{p}_2$$

Then $\mathbf{p}(t)$ is found by linearly interpolating between and by t:

$$\begin{aligned} \mathbf{p}(t) &= (1-t)\mathbf{p}_0^1 + t\mathbf{p}_1^1 \\ &= (1-t)\big((1-t)\mathbf{p}_0 + t\mathbf{p}_1\big) + t\big((1-t)\mathbf{p}_1 + t\mathbf{p}_2\big) \\ &= (1-t)^2\mathbf{p}_0 + 2(1-t)t\mathbf{p}_1 + t^2\mathbf{p}_2 \end{aligned}$$

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

$$\mathbf{p}(t) = (1-t)^2\mathbf{p}_0 + 2(1-t)t\mathbf{p}_1 + t^2\mathbf{p}_2$$

CUBIC BÉZIER Curve

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

We have 4 control point for cubic curve!

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:

$$P_0{}^1 = (1-t)P_0 + tP_1$$

$$P_1{}^1 = (1-t)P_1 + tP_2$$

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.

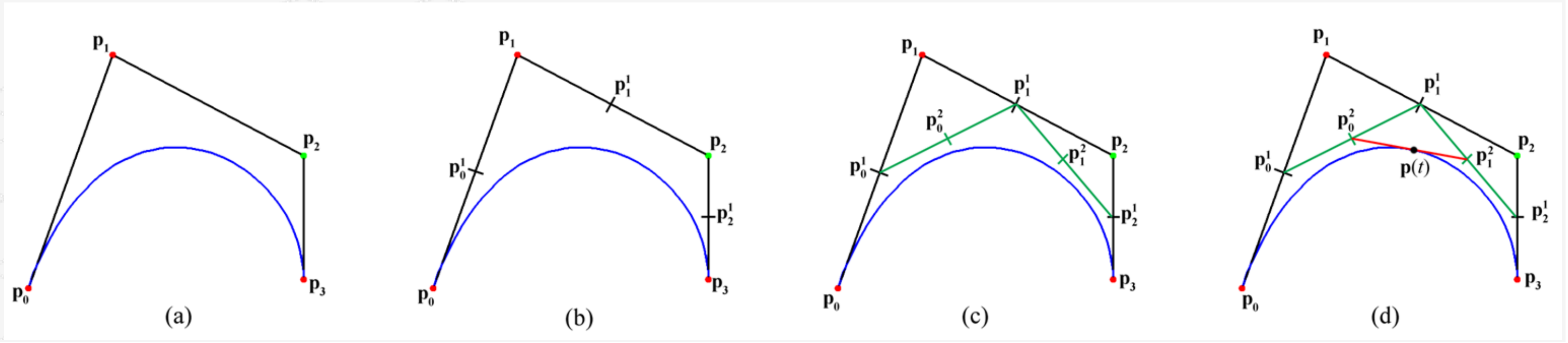
(a) The four control points and the curve they define.

(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 Curve

In a similar manner, four control points $\mathbf{p}_0$, $\mathbf{p}_1$, $\mathbf{p}_2$,and $\mathbf{p}_3$ define a cubic (degree 3) Bézier curve, and a point $\mathbf{p}(t)$ on the curve is found again by repeated interpolation.

$$\mathbf{p}_0^1 = (1-t)\mathbf{p}_0 + t\mathbf{p}_1$$
$$\mathbf{p}_1^1 = (1-t)\mathbf{p}_1 + t\mathbf{p}_2$$
$$\mathbf{p}_2^1 = (1-t)\mathbf{p}_2 + t\mathbf{p}_3$$

Next, linearly interpolate a long each line segment these first generation intermediate points define to get two second generation intermediate points:

$$B_0^3(t) = \frac{3!}{0!(3-0)!} t^0 (1-t)^{3-0} = (1-t)^3$$

$$B_1^3(t) = \frac{3!}{1!(3-1)!} t^1 (1-t)^{3-1} = 3t(1-t)^2$$

$$B_2^3(t) = \frac{3!}{2!(3-2)!} t^2 (1-t)^{3-2} = 3t^2(1-t)$$

Finally, $\mathbf{p}(t)$ is found by linearly interpolating between these last generation intermediate pints:

$$B_3^3(t) = \frac{3!}{3!(3-3)!} t^3 (1-t)^{3-3} = t^3$$

which simplifies to the parametric formula for a cubic (degree 3) Bézier curve:

$$\begin{aligned}\mathbf{p}(t) &= (1-t)\mathbf{p}_0^2 + t\mathbf{p}_1^2 \\ &= (1-t)\left((1-t)^2 \mathbf{p}_0 + 2(1-t)t\mathbf{p}_1 + t^2\mathbf{p}_2\right) + t\left((1-t)^2 \mathbf{p}_1 + 2(1-t)t\mathbf{p}_2 + t^2\mathbf{p}_3\right)\end{aligned}$$

$$\mathbf{p}(t) = (1-t)^3 \mathbf{p}_0 + 3t(1-t)^2 \mathbf{p}_1 + 3t^2(1-t)\mathbf{p}_2 + t^3\mathbf{p}_3$$

Finding the control points

To find any point P along a line, use the formula: $P = (1-t)P0 + (t)P1$, where t is the percentage along the line the point lies and P0 is the start point and P1 is the end point.

Knowing this, we can now solve for the unknown control point. We know P0 (the start point), P (the point along the line), and t (the percentage along the line the point lies). If we rewrite the formula to find P1 (the unknown end point), we get: $P1 = (P - (1-t)P0) / t$.

If you look at the picture of how to calculate the curve again, you may have noticed that point M_5 (the midpoint of the curve) lies on the same horizontal line as M_3.

You may have also noticed that M_3 appears to be on the same horizontal line as the midpoint of M_0 and C_0.

A quick calculation confirms this as C_0 and M_1 lie on the same horizontal line, their Y values are the same. Thus the midpoint of M_0 and C_0 and M_0 and M_1 will have the same Y value.

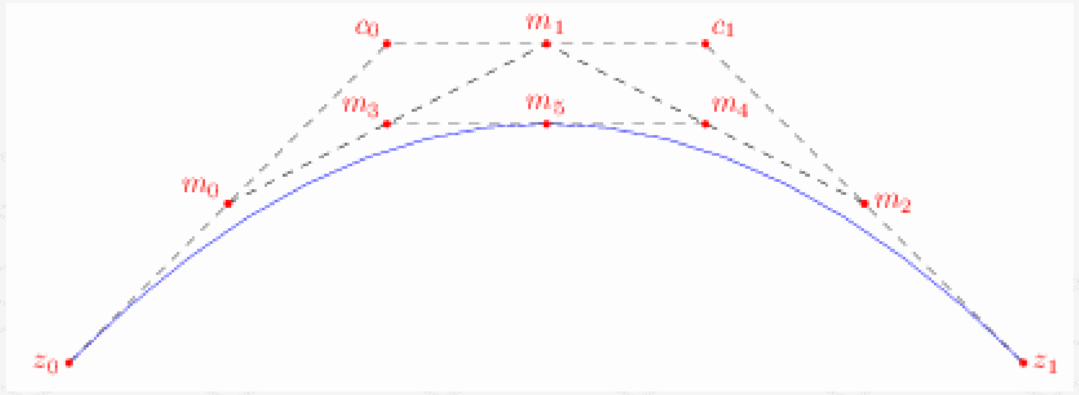
We can now plug in the Y value of our start point for P0, the Y value of the midpoint for P, and 3/4 for t, and this will give us the Y value of where the control point should be placed!

As a quick demonstration, let's say your starting point of the curve lies on at the point (1,1) and the midpoint lies at the point (1,7). Since we want the Y value of the control point, we will use the Y value of the points for the calculation. This gives us the equation

$$P1 = (7 - (1 - .75)) / .75 = 9.$$

Therefore, our control point should line at the Y value of 9.

This equation only works for finding how far above the point the control point should be located. This is because if you were to change the X value of the control point it would only change how much the curve bows, but it would still pass through the midpoint



Bernstein basis functions

It turns out, that the formula for Bézier curves of degree *n* can be written in terms of the *Bernstein basis functions.*

$$\mathbf{p}(t) = (1-t)^3\,\mathbf{p}_0 + 3t(1-t)^2\,\mathbf{p}_1 + 3t^2(1-t)\,\mathbf{p}_2 + t^3\,\mathbf{p}_3$$

For degree 3 curves, the Bernstein basis functions are:

$$B_0^3(t) = \frac{3!}{0!(3-0)!}t^0(1-t)^{3-0} = (1-t)^3$$

$$B_1^3(t) = \frac{3!}{1!(3-1)!}t^1(1-t)^{3-1} = 3t(1-t)^2$$

$$B_2^3(t) = \frac{3!}{2!(3-2)!}t^2(1-t)^{3-2} = 3t^2(1-t)$$

$$B_3^3(t) = \frac{3!}{3!(3-3)!}t^3(1-t)^{3-3} = t^3$$

$$\mathbf{p}(t) = \sum_{j=0}^{3} B_j^3(t)\,\mathbf{p}_j = B_0^3(t)\,\mathbf{p}_0 + B_1^3(t)\,\mathbf{p}_1 + B_2^3(t)\,\mathbf{p}_2 + B_3^3(t)\,\mathbf{p}_3$$

Bernstein Basis

The polynomials is defined by $$B_{i,n}(t) = \binom{n}{i} t^i (1-t)^{n-i},$$

where $\binom{n}{k}$ is a binomial coefficient. The Bernstein polynomials of degree n form a basis for the power polynomials of degree n. The first few polynomials are:

$B_{0,0}(t) = 1$

$B_{0,1}(t) = (1-t)$

$B_{1,1}(t) = t$

$B_{0,2}(t) = (1-t)2$

$B_{1,2}(t) = 2\,(1-t)\,t$

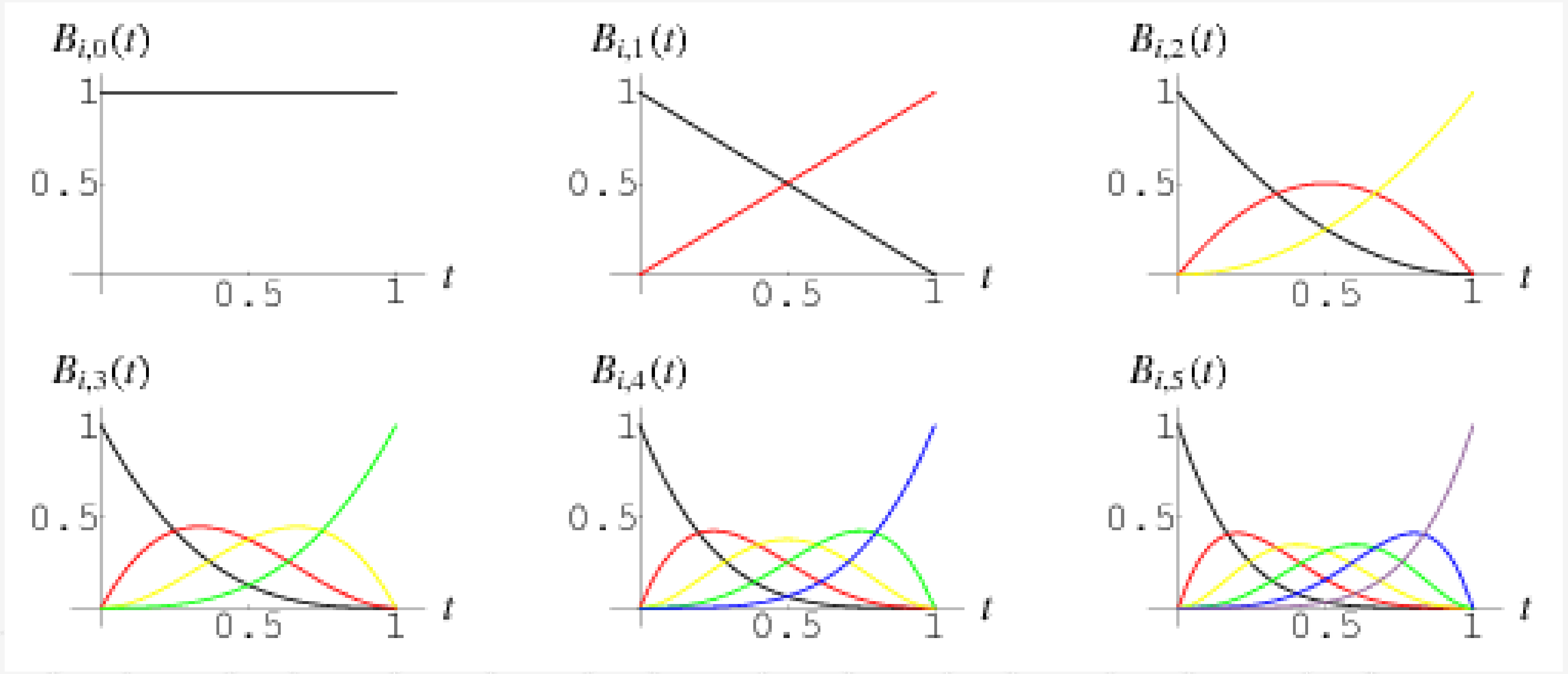
$B_{2,2}(t) = t^2$

$B_{0,3}(t) = (1-t)3$

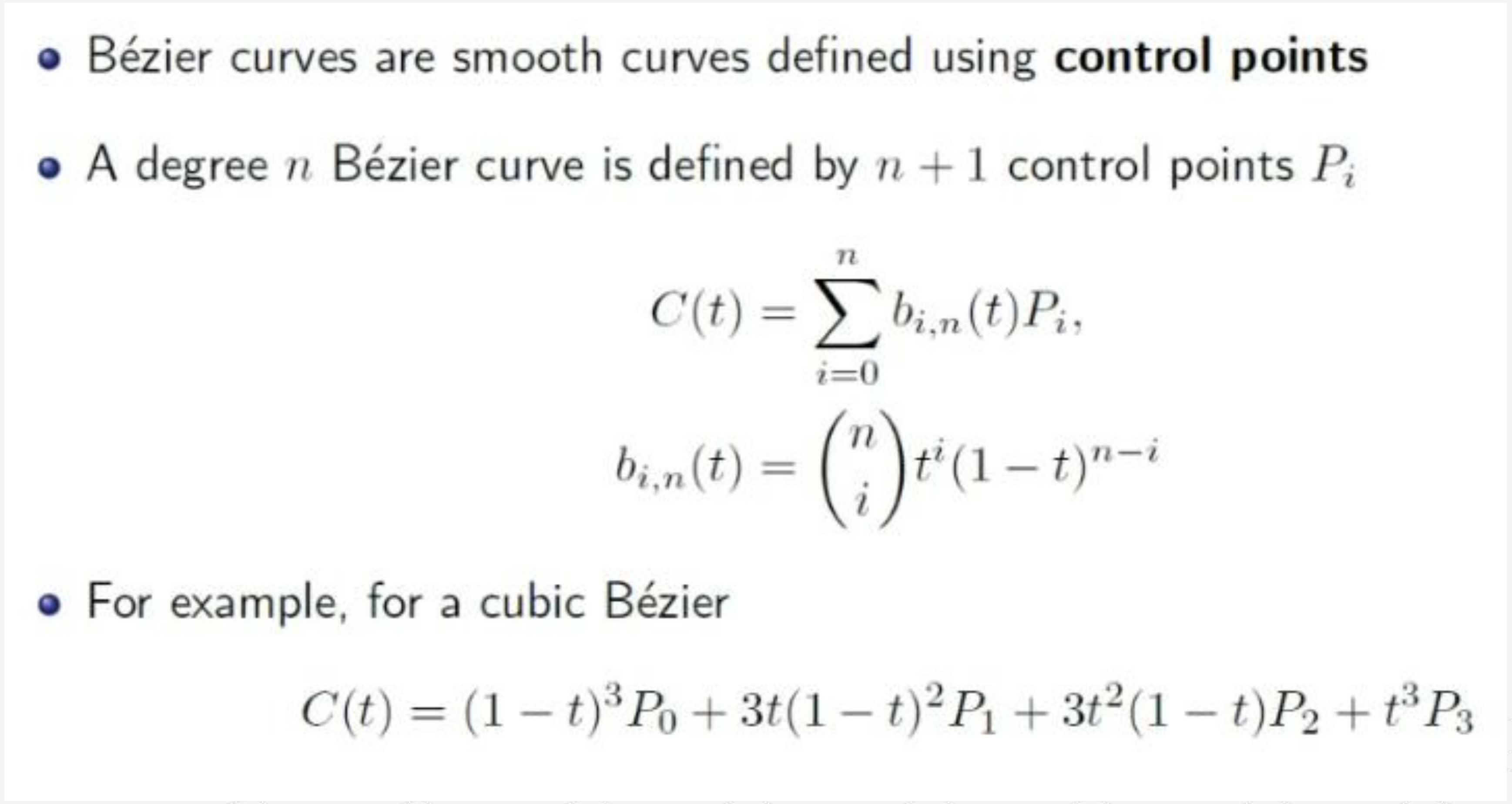
$B_{1,3}(t) = 3\,(1-t)^2\,t$

$B_{1,3}(t) = 3\,(1-t)\,t^2$

$B_{3,3}(t) = t^3$



Recap Bezier Curve



- Bézier curves are smooth curves defined using **control points**
- A degree n Bézier curve is defined by $n+1$ control points P_i

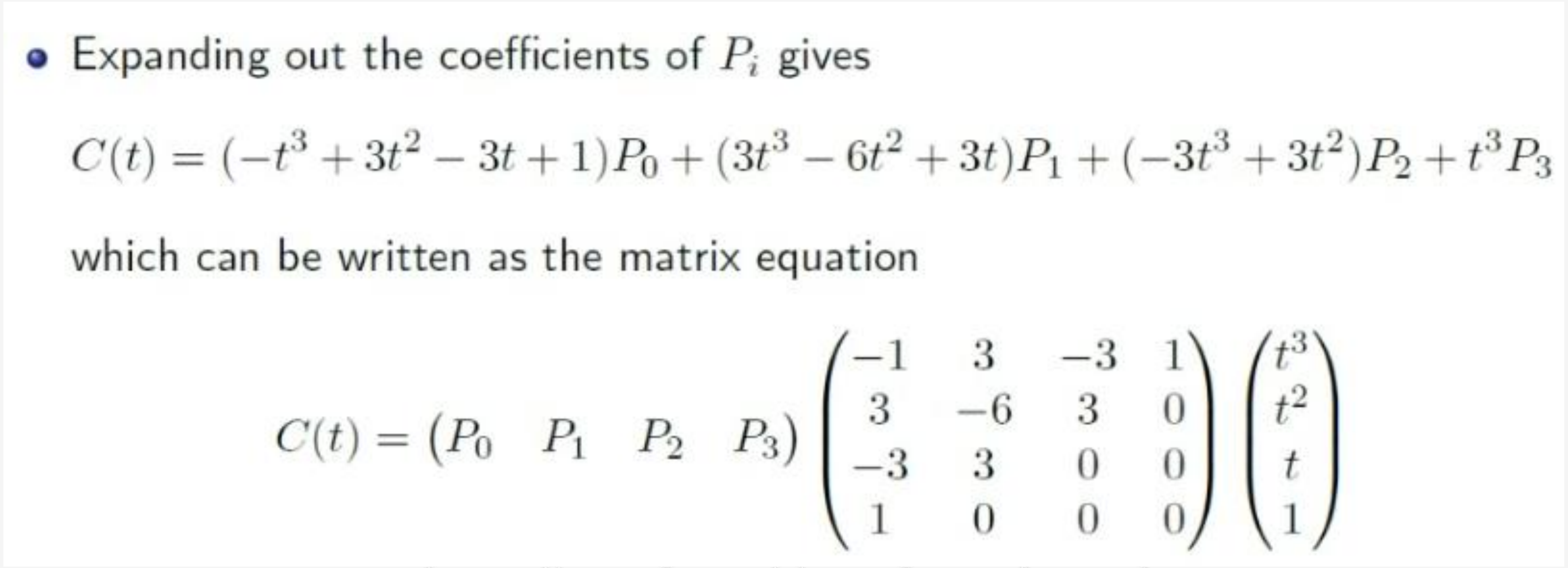
$$C(t) = \sum_{i=0}^{n} b_{i,n}(t) P_i,$$

$$b_{i,n}(t) = \binom{n}{i} t^i (1-t)^{n-i}$$

- For example, for a cubic Bézier

$$C(t) = (1-t)^3 P_0 + 3t(1-t)^2 P_1 + 3t^2(1-t) P_2 + t^3 P_3$$

Cubic Bezier Curve



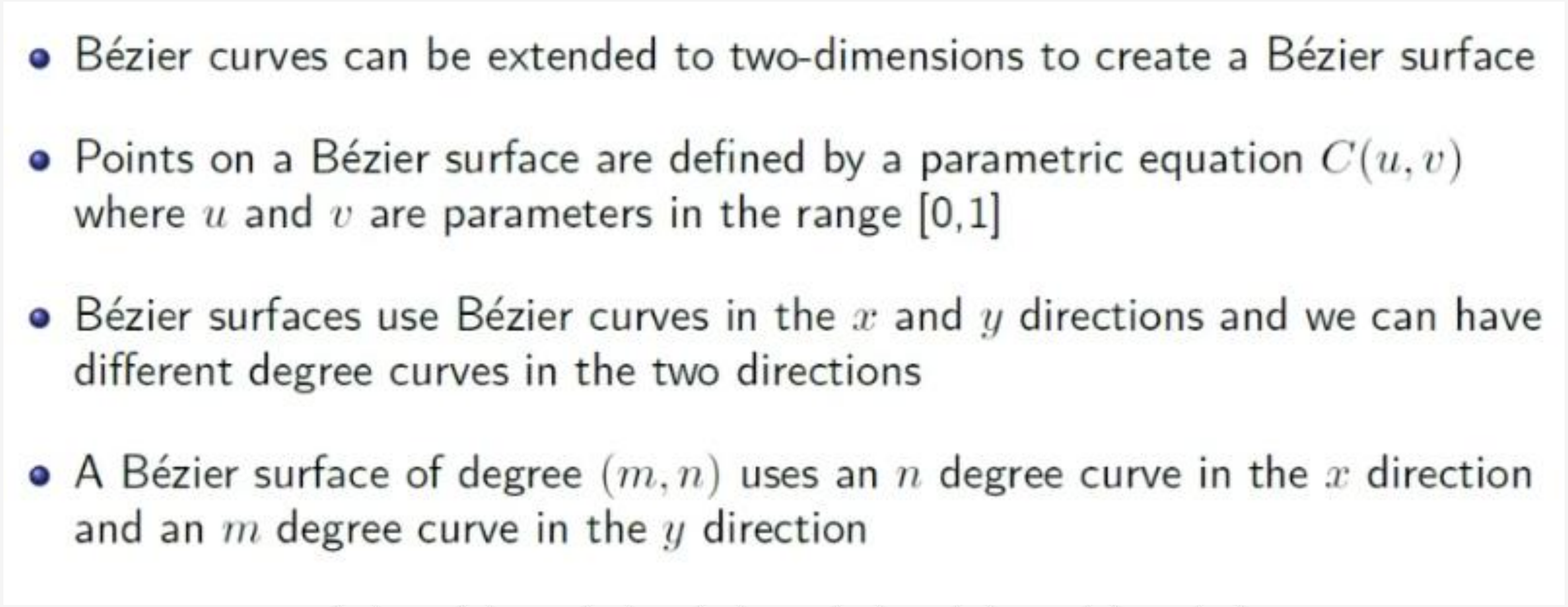
- Expanding out the coefficients of P_i gives

$$C(t) = (-t^3 + 3t^2 - 3t + 1)P_0 + (3t^3 - 6t^2 + 3t)P_1 + (-3t^3 + 3t^2)P_2 + t^3 P_3$$

which can be written as the matrix equation

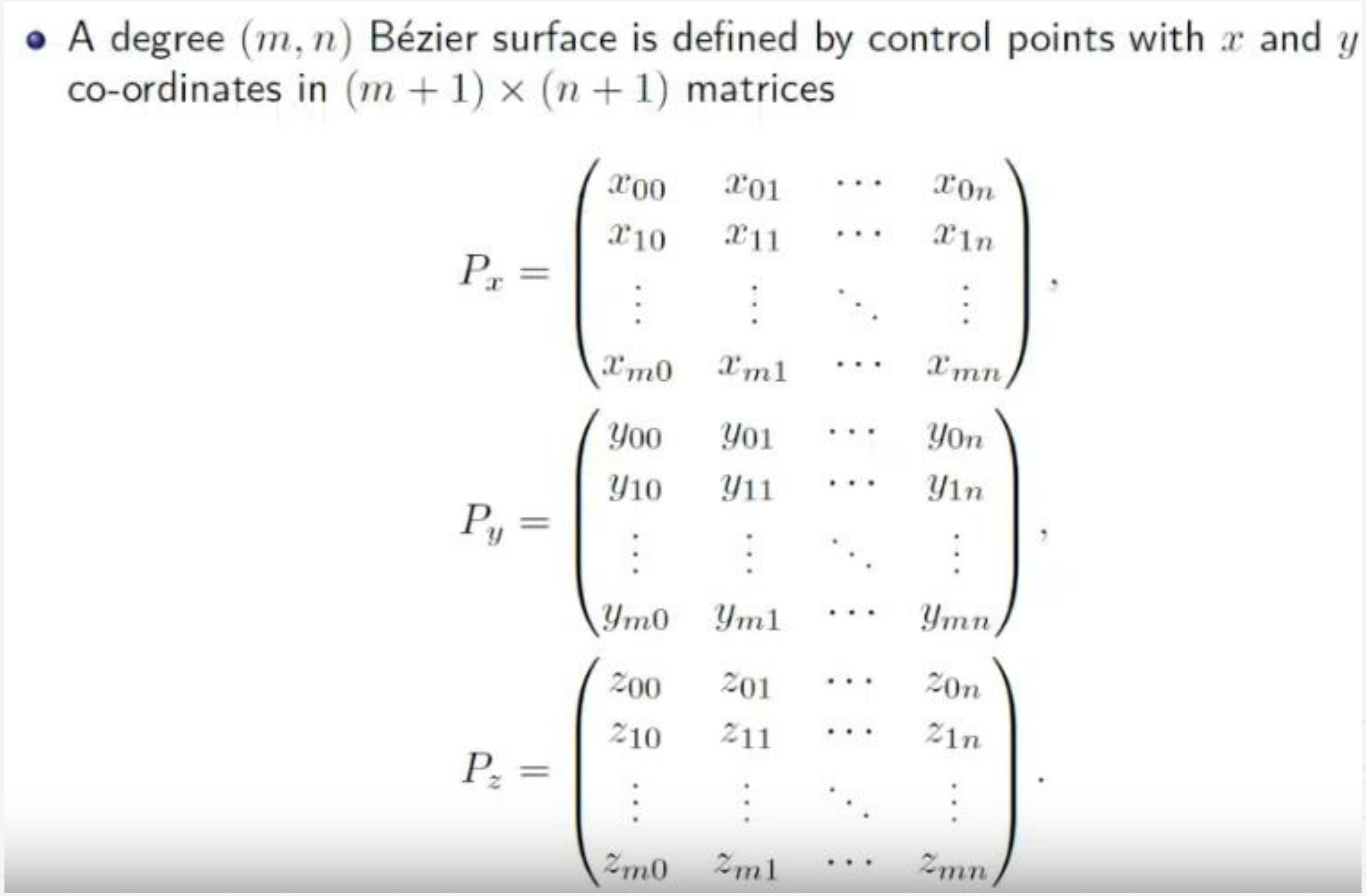
$$C(t) = \begin{pmatrix} P_0 & P_1 & P_2 & P_3 \end{pmatrix} \begin{pmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} t^3 \\ t^2 \\ t \\ 1 \end{pmatrix}$$

Bezier Surfaces



- Bézier curves can be extended to two-dimensions to create a Bézier surface
- Points on a Bézier surface are defined by a parametric equation $C(u, v)$ where u and v are parameters in the range [0,1]
- Bézier surfaces use Bézier curves in the x and y directions and we can have different degree curves in the two directions
- A Bézier surface of degree (m, n) uses an n degree curve in the x direction and an m degree curve in the y direction

Bezier Surface Control Points



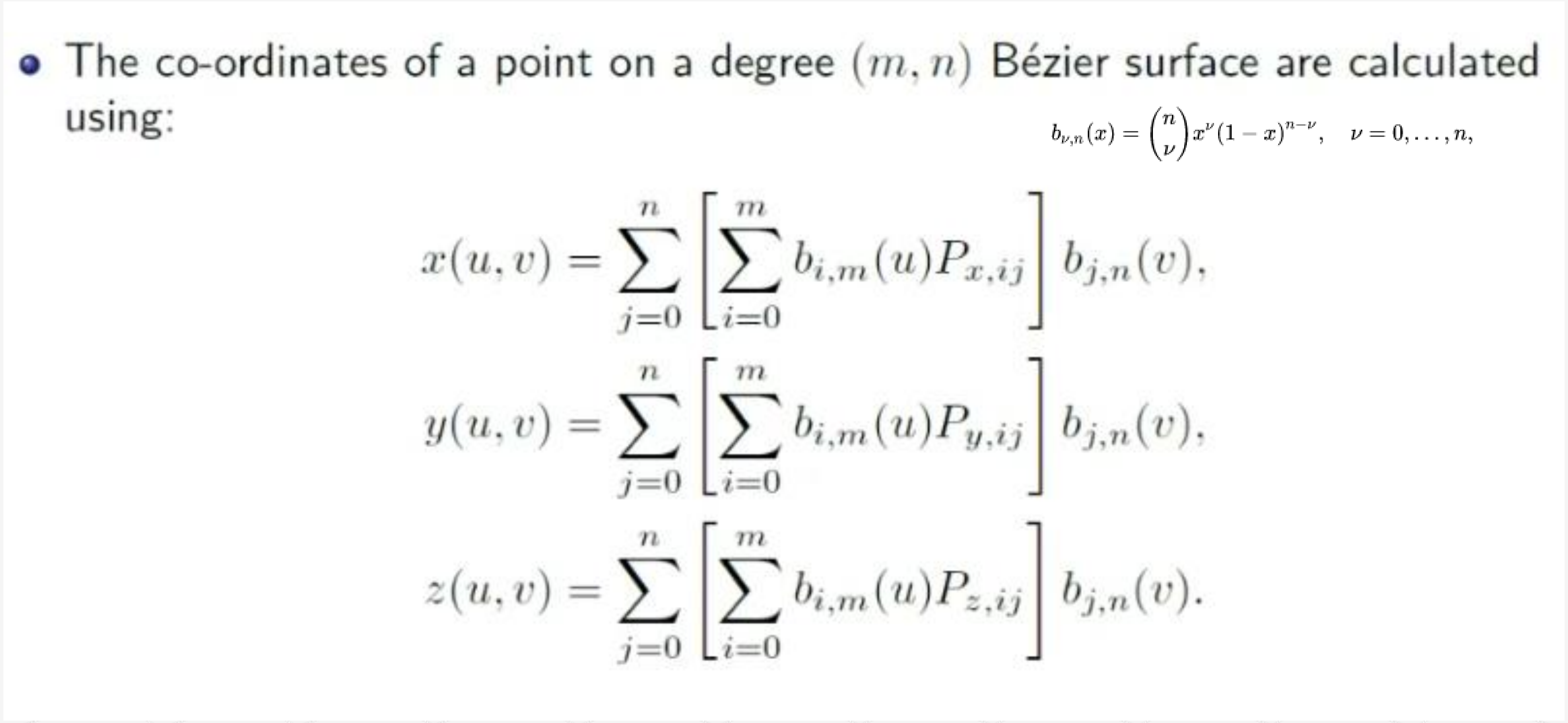
- A degree (m, n) Bézier surface is defined by control points with x and y co-ordinates in $(m+1) \times (n+1)$ matrices

$$P_x = \begin{pmatrix} x_{00} & x_{01} & \cdots & x_{0n} \\ x_{10} & x_{11} & \cdots & x_{1n} \\ \vdots & \vdots & \ddots & \vdots \\ x_{m0} & x_{m1} & \cdots & x_{mn} \end{pmatrix},$$

$$P_y = \begin{pmatrix} y_{00} & y_{01} & \cdots & y_{0n} \\ y_{10} & y_{11} & \cdots & y_{1n} \\ \vdots & \vdots & \ddots & \vdots \\ y_{m0} & y_{m1} & \cdots & y_{mn} \end{pmatrix},$$

$$P_z = \begin{pmatrix} z_{00} & z_{01} & \cdots & z_{0n} \\ z_{10} & z_{11} & \cdots & z_{1n} \\ \vdots & \vdots & \ddots & \vdots \\ z_{m0} & z_{m1} & \cdots & z_{mn} \end{pmatrix}.$$

The coordinates of a point on a Bezier surface



- The co-ordinates of a point on a degree (m, n) Bézier surface are calculated using:

$$b_{\nu,n}(x) = \binom{n}{\nu} x^{\nu}(1-x)^{n-\nu}, \quad \nu = 0, \ldots, n,$$

$$x(u,v) = \sum_{j=0}^{n}\left[\sum_{i=0}^{m} b_{i,m}(u)P_{x,ij}\right] b_{j,n}(v),$$

$$y(u,v) = \sum_{j=0}^{n}\left[\sum_{i=0}^{m} b_{i,m}(u)P_{y,ij}\right] b_{j,n}(v),$$

$$z(u,v) = \sum_{j=0}^{n}\left[\sum_{i=0}^{m} b_{i,m}(u)P_{z,ij}\right] b_{j,n}(v).$$

CUBIC BÉZIER QUAD PATCHES

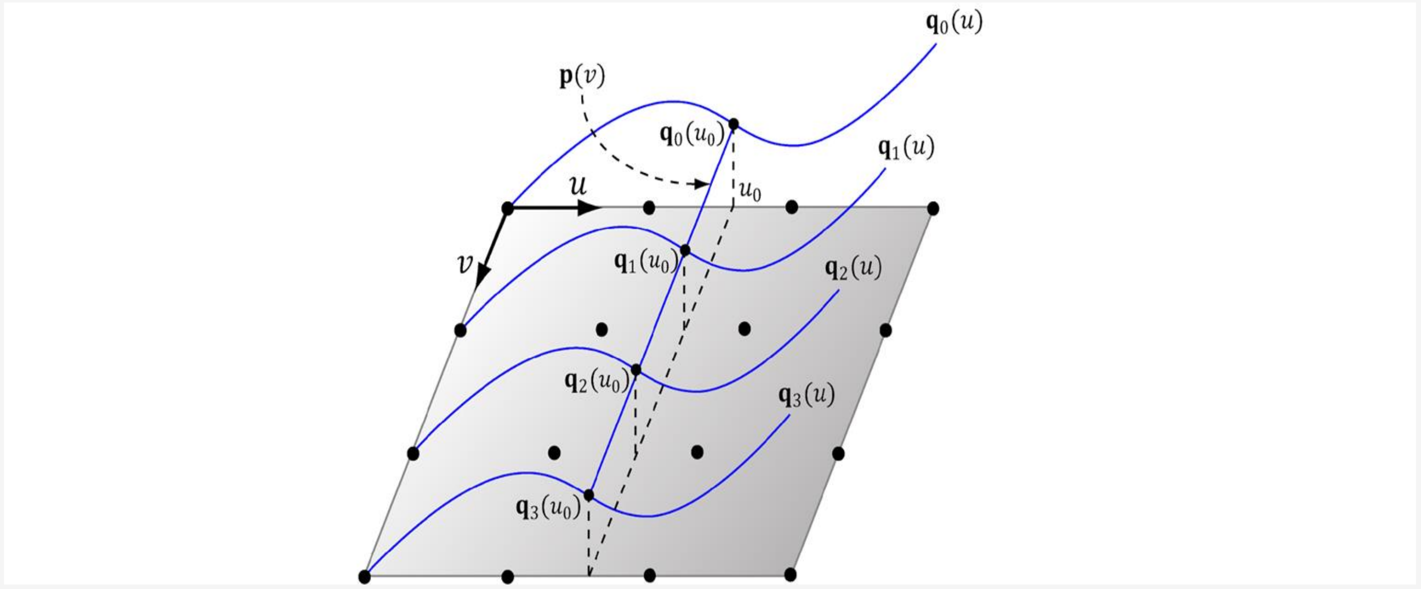
Consider a patch of 4 × 4 control points.

Each row, therefore, contains 4 control points that can be used to define cubic Bézier curve; the Bézier curve of the *ith row is given by:*

$$\mathbf{q}_i(u) = \sum_{j=0}^{3} B_j^3(u)\,\mathbf{p}_{i,j}$$

If we evaluate each of these Bézier curves at say u_0, then we get a “column” of 4 points, one along each curve. We can use these 4 points to define another Bézier curve that lies on the *Bézier surface* at u_0:

$$\mathbf{p}(v) = \sum_{i=0}^{3} B_i^3(v)\,\mathbf{q}_i(u_0)$$



Now, if we let *u* vary as well, we sweep out a family of cubic Bézier curves that form the *cubic Bézier surface:*

$$\begin{aligned}\mathbf{p}(u,v) &= \sum_{i=0}^{3} B_i^3(v)\,\mathbf{q}_i(u)\\ &= \sum_{i=0}^{3} B_i^3(v) \sum_{j=0}^{3} B_j^3(u)\,\mathbf{p}_{i,j}\end{aligned}$$

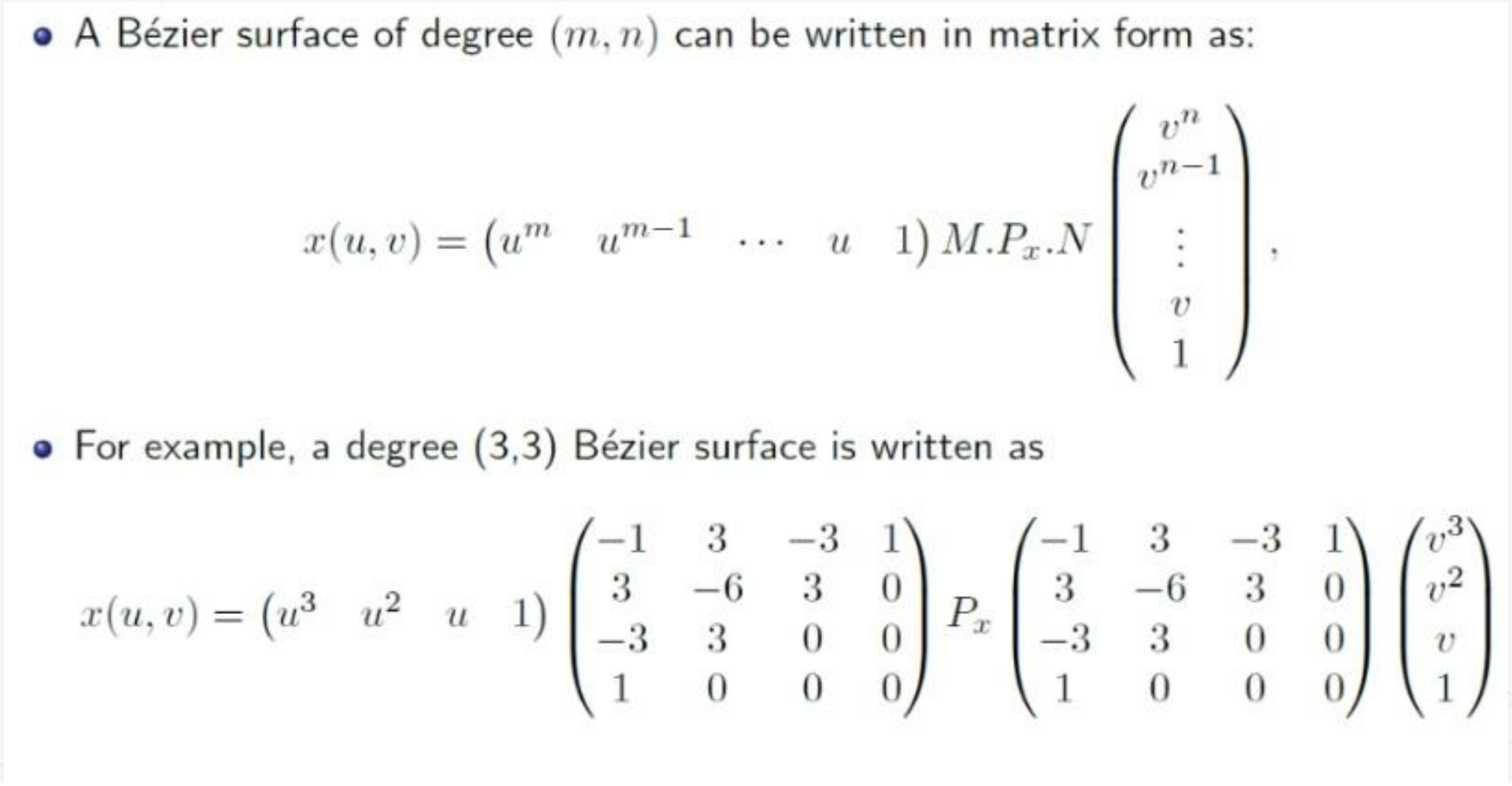
Cubic Bézier Surface Evaluation Code

To help understand the cubic Bézier surface code, we expand out the summation notation:

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

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

Matrix form of Bezier surface



- A Bézier surface of degree (m, n) can be written in matrix form as:

$$x(u,v) = \begin{pmatrix} u^m & u^{m-1} & \cdots & u & 1 \end{pmatrix} M.P_x.N \begin{pmatrix} v^n \\ v^{n-1} \\ \vdots \\ v \\ 1 \end{pmatrix},$$

- For example, a degree (3,3) Bézier surface is written as

$$x(u,v) = \begin{pmatrix} u^3 & u^2 & u & 1 \end{pmatrix} \begin{pmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix} P_x \begin{pmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} v^3 \\ v^2 \\ v \\ 1 \end{pmatrix}$$

MATLAB code

```
function [x,y,z] = bezier_surface(Px,Py,Pz,u,v)

% This function calculates the points on a Bezier surface defined
% by the control points Px, Py, Pz and the parameter values u and v

% define M, U and V
M = [ -1 3 -3 1 ; 3 -6 3 0 ; -3 3 0 0 ; 1 0 0 0 ];
U = [ u.^3 ; u.^2 ; u ; u.^0 ]’;
V = [ v.^3 ; v.^2 ; v ; v.^0 ];

% calculate S
x = U*M*Px*M*V;
y = U*M*Py*M*V;
z = U*M*Pz*M*V;
```

Examples

On the left you have 4x4 = 16 control points (4 in each direction) randomly created by a matrix.

If you apply these control points to cubic Bezier formula, you will construct a nice smooth surface only by 16 control points!

Car manufacturers use Bezier surfaces to design mirrors or different parts of car to create a smooth curve.

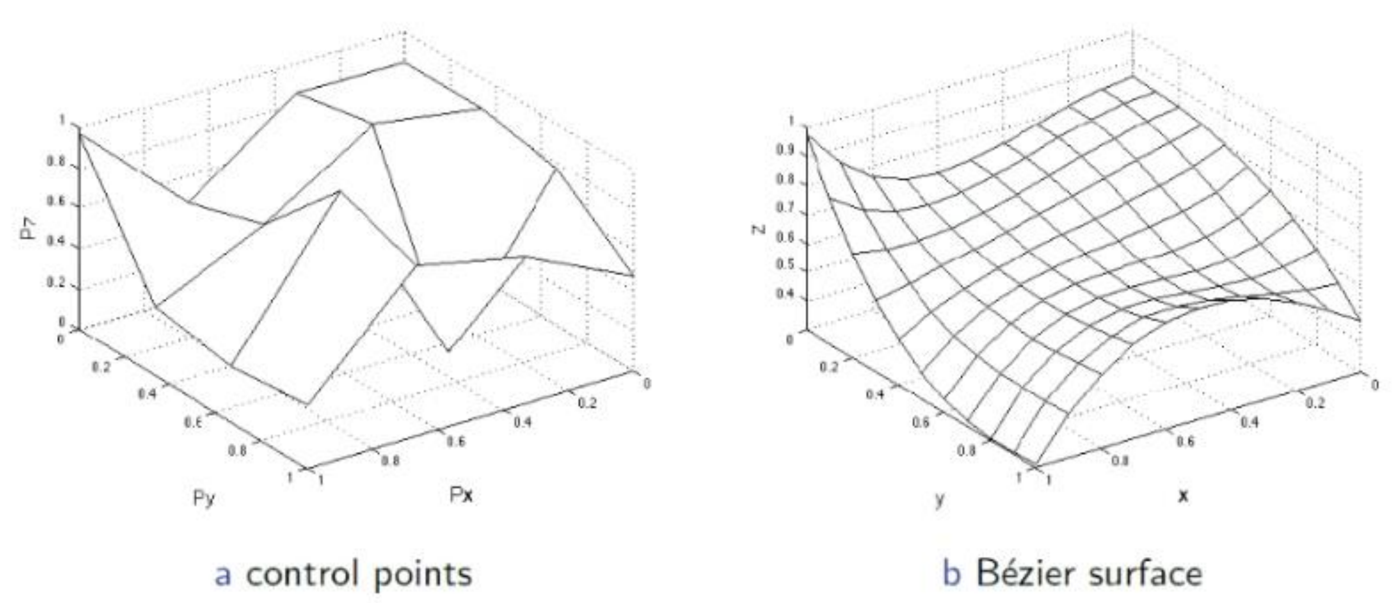


Figure : Bézier surface generated using random control points.

Tessellation.hlsl

we pass the evaluated basis function values to CubicBezierSum.

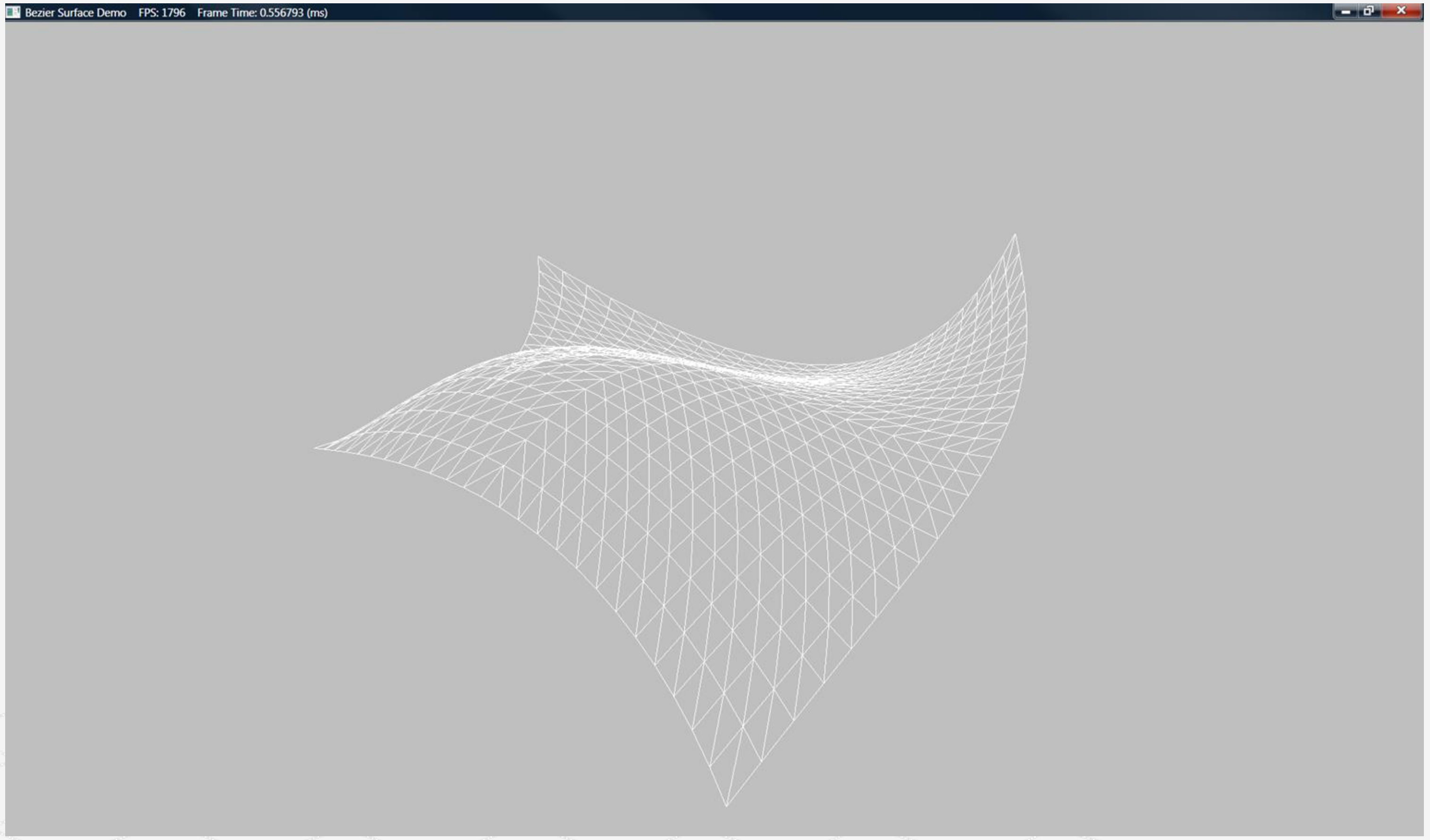
This enables us to use CubicBezierSum for evaluating both **p**(u, v) and the partial derivatives, as the summation form is the same, the only difference being the basis functions.

```
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 );
}
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 );
}
```

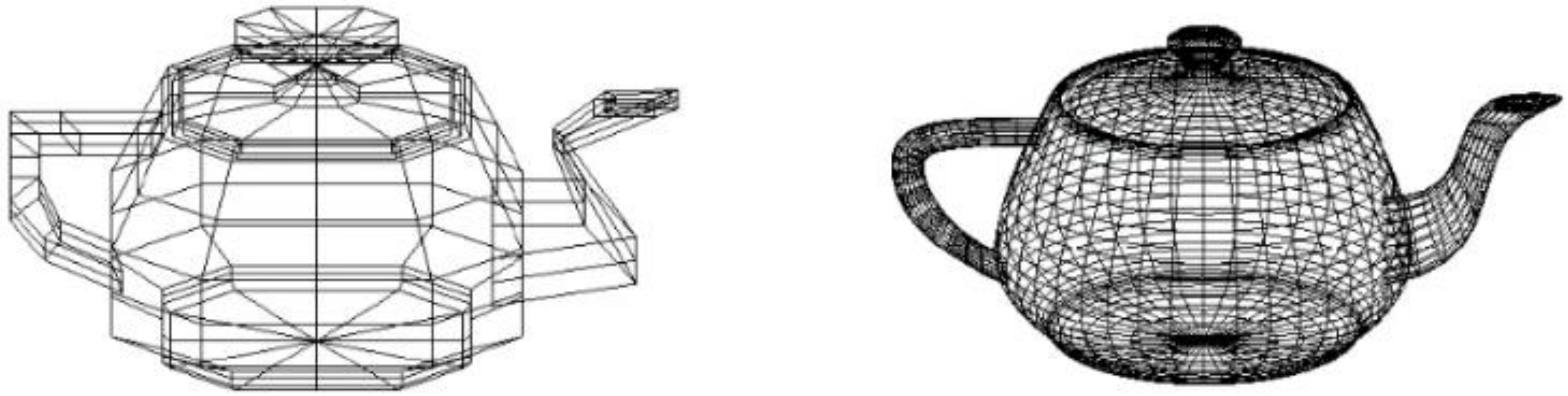
```
float3 CubicBezierSum(const OutputPatch<HullOut, 16> bezpatch, float4 basisU, float4 basisV)
{
   float3 sum = float3(0.0f, 0.0f, 0.0f);
   sum  = basisV.x * (basisU.x*bezpatch[0].PosL  + basisU.y*bezpatch[1].PosL  + basisU.z*bezpatch[2].PosL
+ basisU.w*bezpatch[3].PosL );
   sum += basisV.y * (basisU.x*bezpatch[4].PosL  + 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;
}

[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
};
```



Teapot



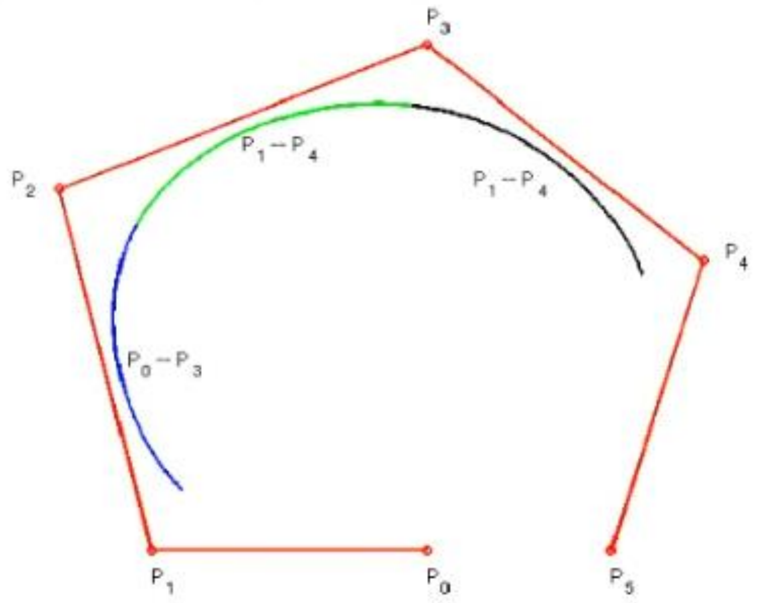
a control points

b Bézier surfaces

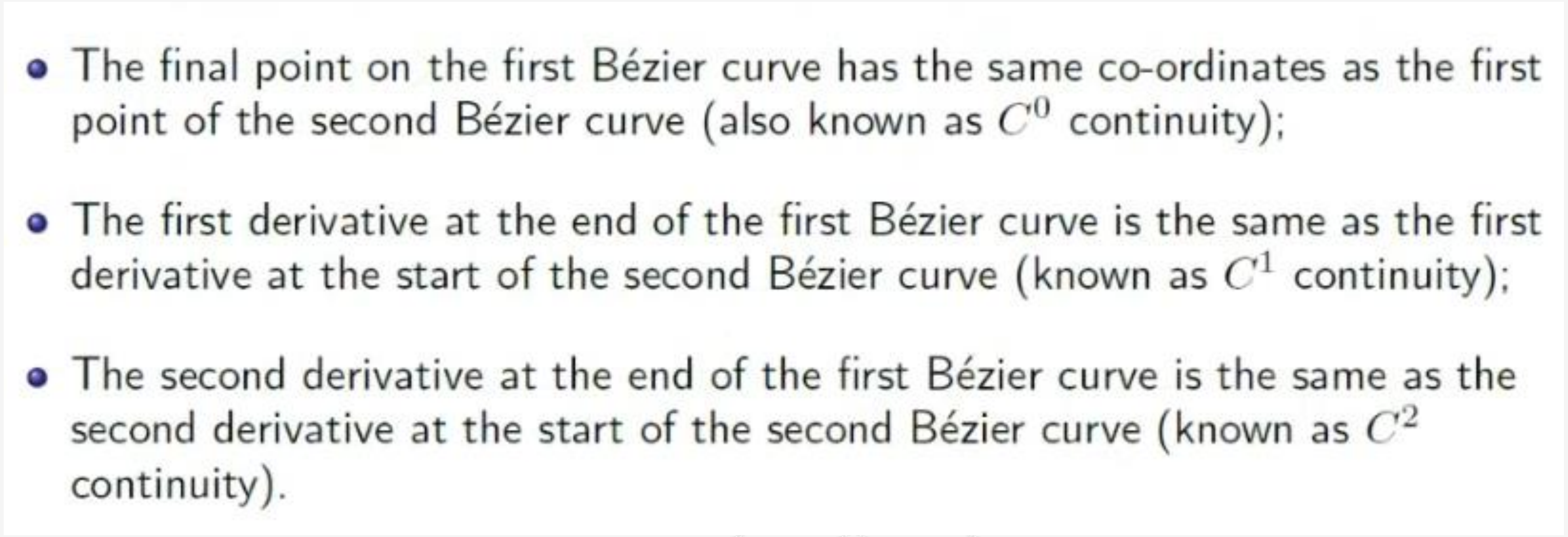
Figure : The Utah teapot control points and with Bézier surfaces applied.

B-splines

- **B-splines** (short for **Basis splines**) use several Bézier curves joined end on end
- A k degree B-spline curve defined by $n+1$ control points will consist of $n-k+1$ Bézier curves
- For example, a cubic B-spline defined by 6 control points $P_0, P_1, \ldots, P_5$ consists of $n-k+1 = 5-3+1 = 3$ Bézier curves



B-Spline Assumptions



- The final point on the first Bézier curve has the same co-ordinates as the first point of the second Bézier curve (also known as C^0 continuity);
- The first derivative at the end of the first Bézier curve is the same as the first derivative at the start of the second Bézier curve (known as C^1 continuity);
- The second derivative at the end of the first Bézier curve is the same as the second derivative at the start of the second Bézier curve (known as C^2 continuity).