# Understanding Shader Model 5.0 with DirectX 11

**Alexandre Valdetaro[1], Gustavo Nunes[1], Alberto Raposo[1], Bruno Feijó[2]**

[1]Tecgraf – PUC-RJ
Rio de Janeiro – Brazil

[2]Departmento de Informática – PUC-RJ
Rio de Janeiro – Brazil.

alexandre@xtunt.com, gustavo@xtunt.com, abraposo@tecgraf.puc-rio.br, bfeijo@inf.puc-rio.br

***Abstract.*** *Being DirectX the Graphics API for Games used by the majority of game developers, its newer release poses great relevance for those interested in real-time computer graphics and, especially, games. We have been working on projects exploring this new API since its first release, and we want to enlighten others, as there is still little available documentation. This tutorial can cover a broad audience. For those who are beginners in the area of computer graphics we advise to avoid low level details and focus on concepts (such as the Graphics Pipeline evolution to the present day and the new pipeline stages) and applications. Intermediate graphics programmers should be especially interested in learning all DirectX 11 features, acquiring knowledge of how to implement the new techniques, and making performance comparisons. Seasoned graphics programmers that are still unacquainted to Shader Model 5.0 should be the individuals that make the best use of this tutorial.*

*The tutorial is divided in 3 parts. The first part is an overview of highlights of DirectX 11, a brief description of applications and scenarios. The second part contains a solid insight into Shader Model 5.0 and the new graphics pipeline. The last part gives some implementation walkthroughs and some code samples.*

## 1. Shader Model Introduction

### 1.1 What is a Shader Model

*Shader Model* is an abstraction to a well defined set of *Shader* capabilities created by *DirectX*. So, what is a Shader? The direct answer is: A script that tells a programmable stage of the graphics hardware what calculations to do to achieve a material, transformation, or effect. On DirectX 11 the languade used to create shaders is called HLSL, and stands for "High Level Shader Language". The term Shader Model has been used ever since the *GPU*s became programmable, and since then, the available features at every new model is dramatically increased . The first shipped graphics card to be assigned a *Shader Model* was the *Geforce 3* with *SM1.1*. On that former version, the shader programmability was reduced to a few assembly instructions and did not

have even float data. However, sequential Shader Model versions naturally did much to increase the programmability: by allowing the use of a higher level programming language, increasing instructions set, flow control, among others. Fig. 1 shows the comparison chart between Shader Models. DirectX 11 has brought a Shader Model 5.0 and a new paradigm in shader programming which will be explained thoroughly through this tutorial.

| Feature | 1.1 2001 | 2.0 2002 | 3.0 2004† | 4.0 2006 |
|---|---|---|---|---|
| instruction slots | 128 | 256 | ≥512 | ≥64K |
| | 4+8‡ | 32+64‡ | ≥512 | |
| constant registers | ≥96 | ≥256 | ≥256 | 16x4096 |
| | 8 | 32 | 224 | |
| tmp registers | 12 | 12 | 32 | 4096 |
| | 2 | 12 | 32 | |
| input registers | 16 | 16 | 16 | 16 |
| | 4+2§ | 8+2§ | 10 | 32 |
| render targets | 1 | 4 | 4 | 8 |
| samplers | 8 | 16 | 16 | 16 |
| textures | | | 4 | 128 |
| | 8 | 16 | 16 | |
| 2D tex size | | | 2Kx2K | 8Kx8K |
| integer ops | | | | ✓ |
| load op | | | | ✓ |
| sample offsets | | | | ✓ |
| transcendental ops | ✓ | ✓ | ✓ | ✓ |
| | | ✓ | ✓ | |
| derivative op | | | ✓ | ✓ |
| flow control | | static | stat/dyn | dynamic |
| | | | stat/dyn | |

**Figure 1**

### 1.2 What has improved in Shader Model 5.0

Prior to *Shader Model 4.0*, there was no possibility of doing per-primitive manipulation, and not also inserting or removing any primitive inside the graphics pipeline. Both the vertex and the *Pixel Shader* could only apply their program to data already in memory. However, *DirectX 10* with the Graphics cards supporting *SM4* provided the addition of the new pipeline stage called the *Geometry Shader*. This new feature unlocked a myriad of new Shading algorithms, generating procedural meshes on-the-fly, silhouette detection, among others. However, the G*eometry Shader* has a small limit of primitives to add/subtract and these operations are fairly expensive. Figure 2 shows the advent of Geometry Shader.

In order to understand the purpose of the new *DirectX 11 Shader Model 5.0*, one of the main bottlenecks of the rendering algorithm must be analyzed, the transfer of highly refined meshes between the *CPU* and the *GPU*, which will be explained shortly.

Models intended to represent real bodies should have smooth and continuous surfaces. In order to create them, one has a multitude of algorithms. These, could be coarsely subdivided into two groups: The first group would contain the methods that have a coarse mesh as its domain, and a refined mesh, real body analogous as its image, some examples are: *Bézier Surfaces, Catmull-Clark Subdivision Surfaces* and surfaces with different Levels of Detail. The second group assign the algorithms that have an already refined mesh as its domain, and also a refined mesh as its image, these only

apply transforms in its domain, in contrast with the first group that refine and transform its domain, some examples of algorithms are: *Parametric Surfaces*, height map terrains, fluids, among others.

Without the possibility of addition and subtraction of primitives, it is obviously impossible to execute any algorithm of the first group fully in the *GPU*, as the only place a mesh could be refined would be the *CPU*, this issue could affect performance thoroughly. Aside from this impossibility, there is also the matter of transferring a dense model to the Graphics Card, which is needed prior the execution of the second group of algorithms in the Graphics Pipeline, as mentioned earlier, the bandwidth between the *CPU* and the *GPU* is constantly a limiting issue to the rendering algorithm.

All of those previous questions have been addressed by *DirectX 11*. With the *Shader Model 5.0*, three new stages have been introduced in the Graphics Pipeline, *Hull Shader*, *Tessellator* and *Domain Shader*, being the first and the latter programmable. There is a new type of primitive called *patch*, it consists in set of vertices or control points. Although a patch primitive is useful on its own, what really makes SM5 so special is the *Tessellator*. This new pipeline stage can create up to 8192 triangles for every primitive that it receives. The exact triangle amount is passed as a parameter. All these new triangles can be transformed programmatically with HLSL in a shader stage that the programmer can access not only the input primitive full vertices data, but also the GPU generated triangle's vertices coordinates. The possibility of creating such a massive primitive amount and manipulate it with such easy makes this new Pipeline a new paradigm in real-time rendering.

Shader programming with *DirectX 11* is extremely flexible, and tailored to create highly complex meshes on-the-fly and easy *Level of Detail* algorithms. Its efficiency is superb, as it trades *CPU-GPU* bandwidth for *GPU ALU* operations that are rarely limiting during rendering.

## 2. DirectX 11 Improvements

### 2.1 Compute Shader

*Compute Shaders* created a new possibility of creating programs on the graphics processor. Programmers are now able to use the GPU as a general processor. With the full parallel processing power of modern graphics cards at hand, programmers can create new techniques that can assist the existing rendering algorithms, or accelerate a variety of general purpose algorithms.

### 2.2 Improved Multithreading

When older Directc3D versions had been released, there was no real focus on supporting multithreading, as multi-core CPUs were not so popular back then. However, with the recent growth on CPU cores, there is an increasing need for a better way to control the GPU from a multithreaded scenario. DirectX 11 addressed this matter with great concern.

Asynchronous graphics device access is now possible in the Direct3D 11 device object. Now programmers are able make API calls from multiple threads. This feature is possible because of the improvements in synchronization between the device object and the graphics driver in DirectX 11.

Direct3D 11 device object has now the possibility of extra rendering contexts. The main immediate context that controls data flow to the GPU continues, but there is now additional deferred contexts, that can be created as needed. Deferred contexts can be created on separate threads and issues commands to the GPU that will be processed when the immediate context is ready to send a new task to the GPU.

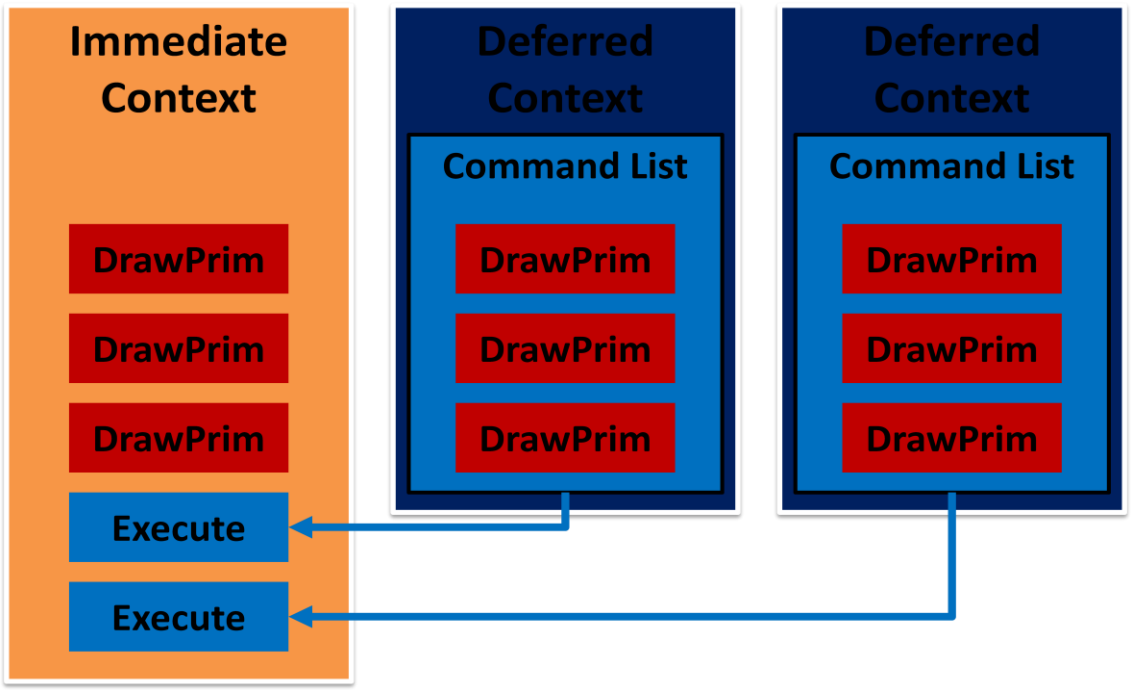


**Figure 1 - Direct3D 11 device object contexts**

## 3. The new Graphics Pipeline

### 3.1 What can now be achieved with the *Tessellator*?

The new graphics pipeline provides a way to adaptively tessellate a mesh on the GPU. This capability implies that we will be trading a lot of CPU-GPU bus bandwidth for GPU ALU operations, which is a fair trade as moderns GPUs have a massive processing power, and the bandwidth is constantly a bottleneck.

Aside from this straightforward advantage in performance, the *Tessellator* also enables a faster dynamic computations such as: skinning animation, collision detection, morphing and any per vertex transform on a model. These computations are now faster because they use the pre-tessellated mesh which is going to be

tessellated into a highly detailed mesh later on. Another advantage of the *Tessellator* usage, is the possibility of applying continuous Level-of-detail to a model, which has always been a crucial issue to be addressed in any rendering engine.

### 3.2 The new Stages of The Graphics Pipeline

We will start describing the purpose of the three new Pipeline stages: the *Hull Shader*, the *Tesselator* and the *Domain Shader*. Prior graphics Pipeline knowledge is advised for this section. The *Hull Shader* is a fully programmable stage, it receives the output from the *Vertex Shader*. This stage is divided into two parts, the *Constant HS* and the *Main HS*. Both parts have access to full primitive information and their purpose is to do any per-primitive calculations. The *Constant HS* is invoked once per primitive, and its mandatory function is to output the *Tessellation Factor*, which is the parameter that will define how much the *Tessellator* will subdivide the primitive. The *Main HS* is invoked per vertex or per control point, and its purpose is to calculate any basis change on the primitive. The *Hull Shader* outputs the *Tessellation Factor* to the *Tessellator* and the control points to the *Domain Shader*.

The *Tessellator* is the stage responsible for subdividing the primitives. Based on the *Tessellation Factor* received from the *Constant HS*, this stage creates new triangles that compose a regular grid with texture coordinates UV varying from 0 to 1. These new triangles are outputted to the *Domain Shader* still with texture coordinates. The tessellation can be achieved in 3 different ways: *integer, even-fractional* and *odd-fractional*. The *integer* way is just a straightforward symmetric subdivision, whereas the fractional ones subdivide on the edges and morph the vertices based on the decimal part of the *Tesselation Factor* which gives a non-popping effect when used in a LOD scheme. A comparative view of the tessellations is on Figure 3.

As the *Hull Shader*, The *Domain Shader* is also a fully programmable stage. The stage has access to full primitive information, and is invoked once per new vertex outputted by the *Tessellator*. It receives the mesh control points from the *HS* and the regular grid with the newly created triangles from the *Tessellator*. The mandatory purpose of the *Domain Shader* is to interpolate the newly created vertices along the control points, in order to transform the parametric UV coordinates into world space XYZ coordinates. The *Domain Shader* outputs the vertices of the fully tessellated mesh to the *Pixel Shader* or optionally the *Geometry Shader*. Practical examples are going to be found in the next Part of this tutorial showing into more detail the functionality overviewed in this section. Figure 4 shows a schematic overview of the Pipeline.

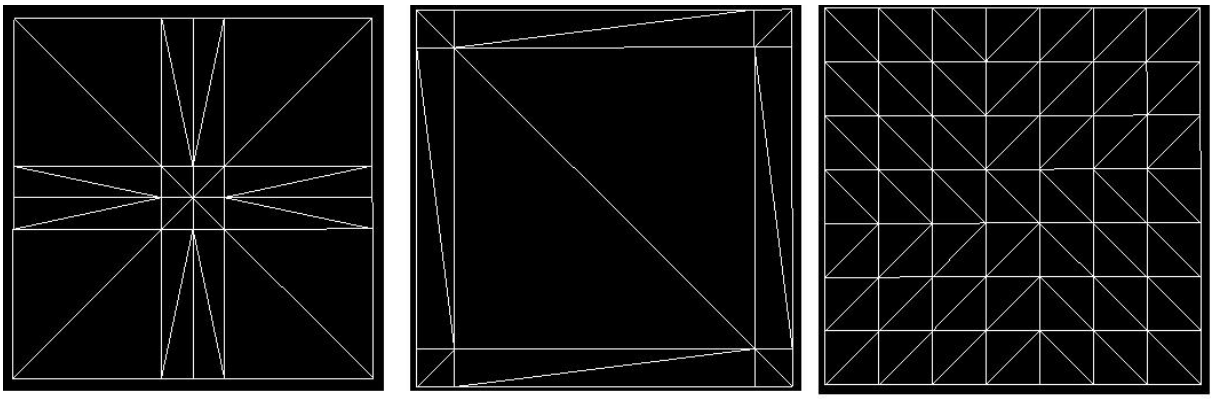


**Figure 2 - Tessellation types.**

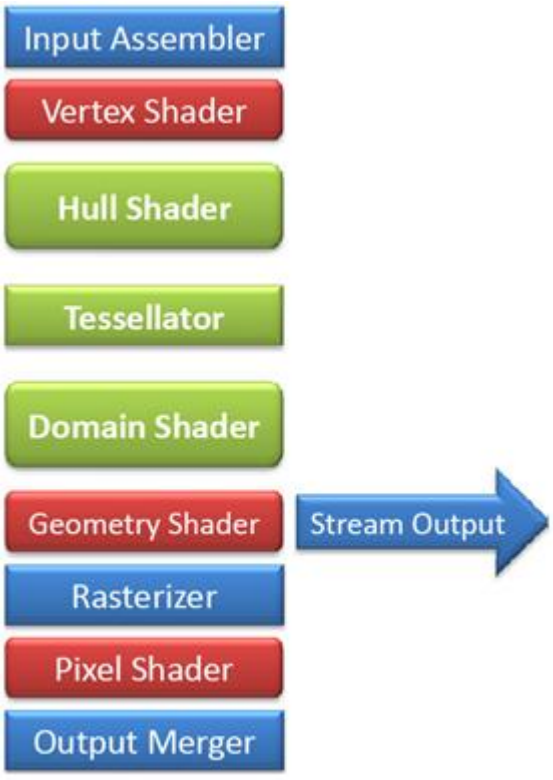


**Figure 4**

## 4. Implementation Examples

### 4.1 Tessellating a quad

Let's start with a simple example: tessellating a quad in the world. Our quad vertices will lie in the plane XZ with world coordinates: $v1 = (0,0,0), v2 = (0,0,100), v3 = (100,0,100), v4 = (100,0,0)$. The first thing we need to do to enable the *Tessellator* pipeline is setting the Input Assembler primitive topology to accept patches through the following line of code ( for simplicity we are not going to describe the full source code of a basic DirectX application, you can access the commented source code of all examples of this tutorial (http://www.xtunt.com/samples)):

```
pd3dImmediateContext->IASetPrimitiveTopology(
D3D11_PRIMITIVE_TOPOLOGY::D3D11_PRIMITIVE_TOPOLOGY_4_CONTROL_POINT_PATCHLIST);
```

Now the *Input Assembler* is set and the pipeline assumes that each primitive is a patch with 4 vertices (control points). A patch primitive may have from 1 through up to 32 control points. Now let's code the shader itself. First we will specify our per frame variables through a constant buffer:

```
cbuffer cbPerFrame : register( b0 )
{
    matrix g_mViewProjection;
    float  g_fTessellationFactor;
};
```

As you can see, for this example we are going to need only two per-frame variables, the View-Projection matrix and a *tessellation factor* which represents the amount of tessellation for our quad. This value goes from 1 up to 64.

Now we need to code the *Vertex Shader*. In this new tessellation pipeline the *Vertex Shader* is mostly used for animation. The programmer may animate a coarse mesh and tessellate it later in the pipeline. That way GPU animation effort is saved. For this example we are not going to do any animation in the *Vertex Shader*, so it will be a simple pass-through shader. Let's specify the input and output structs of the *Vertex Shader*.

```
struct VS_CONTROL_POINT_INPUT
{
    float3 vPosition        : POSITION;
};

struct VS_CONTROL_POINT_OUTPUT
{
    float3 vPosition        : POSITION;
};
```

Our shader will only receive a position as input and will output the same position. Simple enough:

```
 VS_CONTROL_POINT_OUTPUT VS( VS_CONTROL_POINT_INPUT Input )
{
    VS_CONTROL_POINT_OUTPUT Output;
    Output.vPosition = Input.vPosition;
```

```
    return Output;
}
```

No big news yet. Now we will code the *Hull Shader*. This shader is composed of two parts: the Constant part and the *Hull Shader* itself. First let's see the input and output structs of the *Hull Shader*:

```
struct HS_CONSTANT_DATA_OUTPUT
{
    float Edges[4]             : SV_TessFactor;
    float Inside[2]            : SV_InsideTessFactor;
};

struct HS_OUTPUT
{
    float3 vPosition           : POSITION;
};
```

The input is what is outputted from the *Vertex Shader*, so in this example the input of the *Hull Shader* is the `VS_CONTROL_POINT_OUTPUT` structure. Now, the output of the constant part is the `HS_CONSTANT_DATA_OUTPUT` structure, in this struct the programmer should output any per-patch parameter. In this example we are outputting only the mandatory per-patch parameters, which are the *tessellation factors* of our quad. There are four edge *tessellation factors* (one per-edge) of the quad. Being able to specify per edge factors is important to keep watertight meshes. With these factors the programmer is able to guarantee same amount of vertices at the edges of neighbor patches. There are also two inner *tessellation factors* which represent horizontal and vertical subdivision for the center of the quad.

Next is the *Hull Shader* constant part:

```
HS_CONSTANT_DATA_OUTPUT ConstantHS( InputPatch<VS_CONTROL_POINT_OUTPUT, 4> ip,
                                    uint PatchID : SV_PrimitiveID )
{
    HS_CONSTANT_DATA_OUTPUT Output;

    Output.Edges[0] = Output.Edges[1] = Output.Edges[2] = Output.Edges[3] =
g_fTessellationFactor;
    Output.Inside[0] = Output.Inside[1] = g_fTessellationFactor;

    return Output;
}
```

As you can see this part outputs a `HS_CONSTANT_DATA_OUTPUT` structure and receives as input a patch with 4 control points and each control point is represented by the structure `VS_CONTROL_POINT_OUTPUT` which in our case it only contains a position. It also receives a `PatchID` that is an identifier number of the patch generated by the *Input Assembler*. This part is invoked once per-patch. Basically in this part we are passing to all edges and inner parts of the quad the same *tessellation factor*, so we expect as output an uniformly tessellated quad. It is the constant part of the *Hull Shader* that the programmer may do calculations for tessellating meshes adaptively based for example on: distance to the camera, gradient map, screen space edge size, if the patch is on the

silhouette, etc… Moreover, there is another important property of this shader part: if a tessellation factor for a patch is set to zero, the patch is culled from the rest of the pipeline. This is important to avoid tessellating patches that are out of the view frustum range or back-faced patches. Now let's see the *Hull Shader* main part:

```
[domain("quad")]
[partitioning("integer")]
[outputtopology("triangle_cw")]
[outputcontrolpoints(4)]
[patchconstantfunc("ConstantHS")]
HS_OUTPUT HS( InputPatch<VS_CONTROL_POINT_OUTPUT, 4> p,
                    uint i : SV_OutputControlPointID,
                    uint PatchID : SV_PrimitiveID )
{
    HS_OUTPUT Output;
    Output.vPosition = p[i].vPosition;
    return Output;
}
```

There is a lot of new information in this part. First of all, this part is invoked once per-output control point, but the programmer is able to consult the whole patch information. So, it receives as parameters the patch with four control points, a `SV_OutputControlPointID` that represents the current control point invocation ( for our quad this integer will vary from 0 to 3 ) and a `PatchID` that is an identifier number of the patch generated by the *Input Assembler*. This *Hull Shader* part is commonly used for a coordinate basis change, for example changing from a quad to a Bezier bi-cubic. The other parameters shown are:

- `domain(quad, tri, isoline)`: Specifies what are you tessellating: quad, triangle or isoline. In our example we must chose quad.
- `partitioning(integer, fractional_even, fractional_odd, pow2)`: This attribute tells the *Tessellator* how to interpret our tessellation factors. Integer partitioning means the tessellation factors are interpreted as integral values. The vertices are not created in a smooth manner. Parameters `fractional_even` and `fractional_odd` creates "in-between" tessellated vertices, so the vertices are created on a smooth manner. In the June 2010 SDK `pow2` is behaving just like integer partitioning.
- `outputtopology(triangle_cw, triangle_ccw, line)`: This attribute tells the *Tessellator* where the output primitives should face. The `triangle_cw` generates triangles in a clockwise manner, `triangle_ccw` generates triangles in a counter-clockwise manner and line generates lines.
- `outputcontrolpoints(1..32)`: The amount of times this *Hull Shader* part will be called. For example if you are changing basis from a quad to a bi-cubic Bezier patch, this parameter should be 16. Because you would be receiving four

control-points for the quad and outputting sixteen for the bi-cubic Bezier patch. For our example we will output the same amount of control points as the input.

- `patchconstantfunc(string)`: The name of the function that represents your *Hull Shader* constant function. In our example it is `ConstantHS`.

Now we will look at the *Domain Shader*. This shader part should be looked like a "*Vertex Shader* after the tessellation". In this shader you are able to manipulate all the vertices that were created by the *Tessellator*. First let's declare the struct that the *Domain Shader* will output:

```
struct DS_OUTPUT
{
    float4 vPosition        : SV_POSITION;
};
```

As you can see our output to the *Pixel Shader* will be just the position of the vertex after the conversion by the View and Projection matrices. The *Domain Shader* itself is as follows:

```
[domain("quad")]
DS_OUTPUT DS( HS_CONSTANT_DATA_OUTPUT input,
                    float2 UV : SV_DomainLocation,
                    const OutputPatch<HS_OUTPUT, 4> quad )
{
    DS_OUTPUT Output;
    float3 verticalPos1 = lerp(quad[0].vPosition,quad[1].vPosition,UV.y);
    float3 verticalPos2 = lerp(quad[3].vPosition,quad[2].vPosition,UV.y);
    float3 finalPos = lerp(verticalPos1,verticalPos2,UV.x);

    Output.vPosition = mul( float4(finalPos,1), g_mViewProjection );
    return Output;
}
```

The *Domain Shader* receives as input the `HS_CONSTANT_DATA_OUTPUT` structure that has any per-patch parameter passed by the *Hull Shader* constant part and we need to specify the same domain attribute that we chose at the *Hull Shader*, which in our case is "quad". It also receives the patch control points data and a float2 UV coordinate that is where the *Tessellator* created the new vertices through the domain in a parametric space from 0 to 1. Suppose a *Domain Shader* invocation with UV = (0.5, 0.5), that means that the *Domain Shader* is being invoked for a *vertex* that was created in the middle of the quad by the *Tessellator* as shows figure 5:

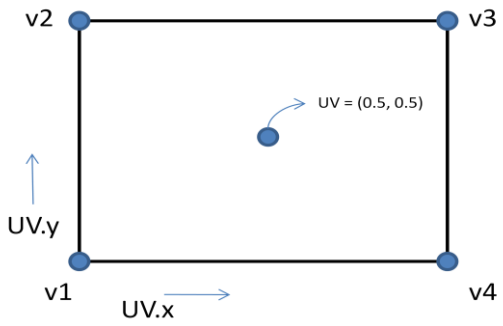


**Figure 3**

The UV parameter represents the vertices created by the *Tessellator*, but we need to put those vertices in world space according to our quad. To do that we have to make three linear interpolations as figure 6 shows for an UV = (0.5, 0.5) invocation:

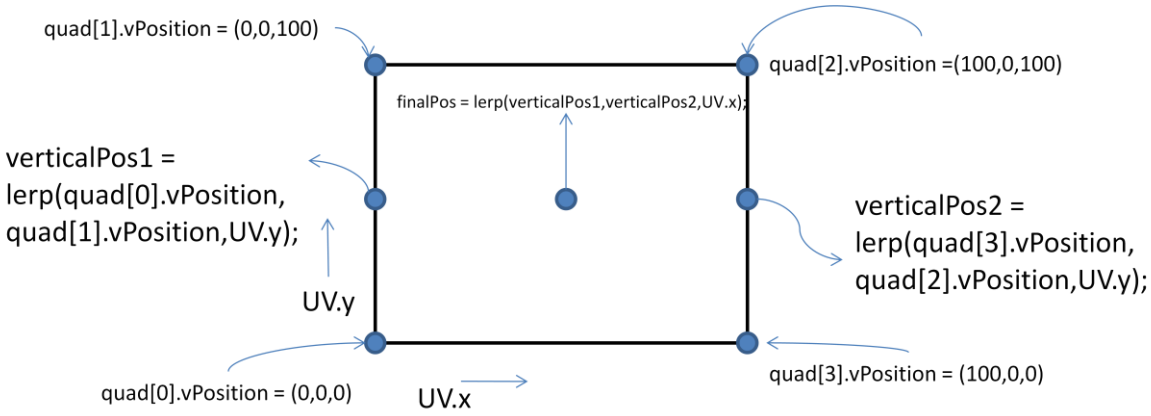


**Figure 4**

For each vertex generated by the *Tessellator* those linear interpolation are made and the vertex is placed at the right place in the world. Now, to complete the pipeline we need a simple *Pixel Shader*:

```
float4 SolidColorPS( DS_OUTPUT Input ) : SV_TARGET
{
    return float4( 1,1,1, 1 );
}
```

This *Pixel Shader* just outputs a solid white color. If we needed a Geometry shader for our technique it would be placed after the *Domain Shader* in the pipeline, but we are not going to use it for this example. This is the final result of our sample, our tessellated quad:

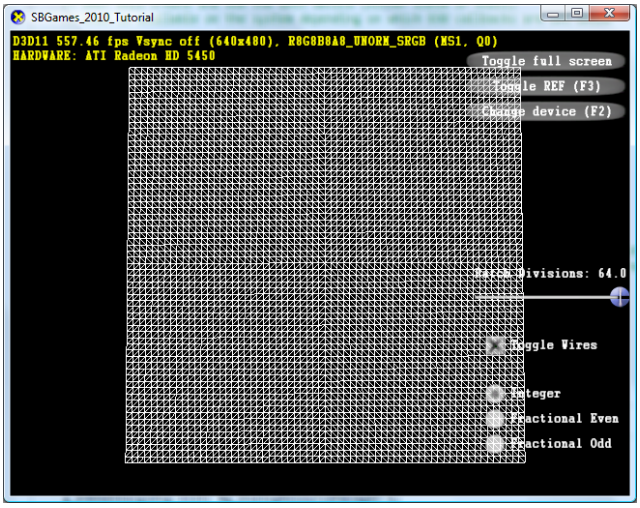


**Figure 5**

That is a simple screenshot but it means a lot. There are 8192 triangles generated by the hardware in this screenshot. We had to pass just 48 bytes of information from the CPU to the GPU. Doing this screenshot with a traditional CPU tessellation approach we would have spent 8192*3*12 = 294,91 Kbytes of bandwidth, which is the most expensive resource of today's rendering pipeline.

### 4.2 Tessellating a triangle

Now let's follow the same example above but this time we are going to tessellate a triangle instead of a quad. We are going to pass-through this example quicker than the previous one because they are very similar in most parts. Please refer to the tutorial sample source code if you have any doubts. Our triangle will have world coordinates: $v1 = (0,0,0), v2 = (0,0,10), v3 = (10,0,10)$. First we need to set the *Input Assembler* to accept a patch with three control points ( our triangle ):

```
pd3dImmediateContext->IASetPrimitiveTopology(
D3D11_PRIMITIVE_TOPOLOGY::D3D11_PRIMITIVE_TOPOLOGY_3_CONTROL_POINT_PATCHLIST);
```

Now we will code the shader. The constant buffer and *Vertex Shader* are the same of the previous example, we just need the tessellation factor, the View*Projection matrix and a pass-through *Vertex Shader*:

```
//--------------------------------------------------------------------------------
--------
// Constant Buffers
//--------------------------------------------------------------------------------
--------
cbuffer cbPerFrame : register( b0 )
{
    matrix g_mViewProjection;
    float  g_fTessellationFactor;
```

```
};
//--------------------------------------------------------------------------------
--------
// Vertex shader section
//--------------------------------------------------------------------------------
--------
struct VS_CONTROL_POINT_INPUT
{
    float3 vPosition        : POSITION;
};
struct VS_CONTROL_POINT_OUTPUT
{
    float3 vPosition        : POSITION;
};

VS_CONTROL_POINT_OUTPUT VS( VS_CONTROL_POINT_INPUT Input )
{
    VS_CONTROL_POINT_OUTPUT Output;
    Output.vPosition = Input.vPosition;
    return Output;
}
```

The *Hull Shader* changes slightly from the previous sample. Now the `HS_CONSTANT_DATA_OUTPUT` structure has three edge *tessellation factors* ( one per triangle edge ) and one inside *tessellation factor*. Also the patch now has only three control points instead of four. The `domain` attribute must be set to `"tri"` and `outputcontrolpoints` attribute must be set to `"3"`:

```
struct HS_CONSTANT_DATA_OUTPUT
{
    float Edges[3]             : SV_TessFactor;
    float Inside            : SV_InsideTessFactor;
};

struct HS_OUTPUT
{
    float3 vPosition           : POSITION;
};

HS_CONSTANT_DATA_OUTPUT ConstantHS( InputPatch<VS_CONTROL_POINT_OUTPUT, 3> ip,
                                    uint PatchID : SV_PrimitiveID )
{
    HS_CONSTANT_DATA_OUTPUT Output;
    Output.Edges[0]     =     Output.Edges[1]     =     Output.Edges[2]     =
g_fTessellationFactor;
    Output.Inside = g_fTessellationFactor;
    return Output;
}

[domain("tri")]
[partitioning("fractional_odd")]
[outputtopology("triangle_cw")]
[outputcontrolpoints(3)]
[patchconstantfunc("ConstantHS")]
HS_OUTPUT HS( InputPatch<VS_CONTROL_POINT_OUTPUT, 3> p,
                    uint i : SV_OutputControlPointID,
                    uint PatchID : SV_PrimitiveID )
{
    HS_OUTPUT Output;
    Output.vPosition = p[i].vPosition;
    return Output;
}
```

In the *Domain Shader* the vertices created by the *Tessellator* must be transformed to world space just like in the previous sample. However there is a difference. Now the *Tessellator* provides the vertices positions as normalized barycentric coordinates instead of the previous regular UV texture coordinates. The barycentric coordinates have three variables UVW, each represents the weight of a control point for the generated *vertex* coordinate where U+V+W = 1 and W = 1 - U - V. Figure 8 illustrates an example for U=V=W= 0.333, which means that each triangle control point has an even amount of contribution for this *vertex* coordinate. So, it will produce a *vertex* at the middle of the triangle.

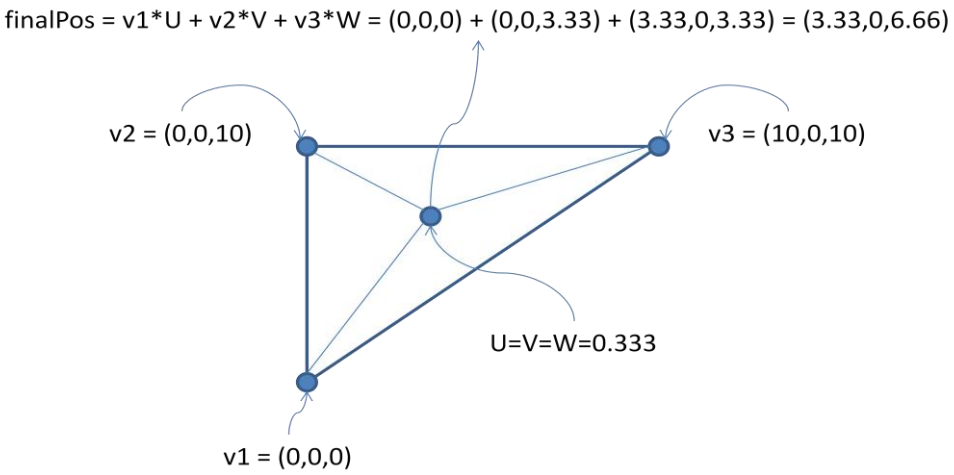


**Figure 6**

The same interpolation can be used to find texture coordinates or normals inside the triangle. Also, we have to change the `domain` attribute to `"tri"`. This is the *Domain Shader* snippet:

```
struct DS_OUTPUT
{
    float4 vPosition        : SV_POSITION;
};

[domain("tri")]
DS_OUTPUT DS( HS_CONSTANT_DATA_OUTPUT input,
                    float3 UVW : SV_DomainLocation,
                    const OutputPatch<HS_OUTPUT, 3> quad )
{
    DS_OUTPUT Output;

    float3 finalPos = UVW.x * quad[0].vPosition + UVW.y * quad[1].vPosition +
UVW.z * quad[2].vPosition;

    Output.vPosition = mul( float4(finalPos,1), g_mViewProjection );

    return Output;
}
```

The *Pixel Shader* is the same as the previous example:

```
float4 SolidColorPS( DS_OUTPUT Input ) : SV_TARGET
{
    return float4( 1, 1, 1, 1 );
}
```

This is final result of this example, the triangle tessellated by the hardware:

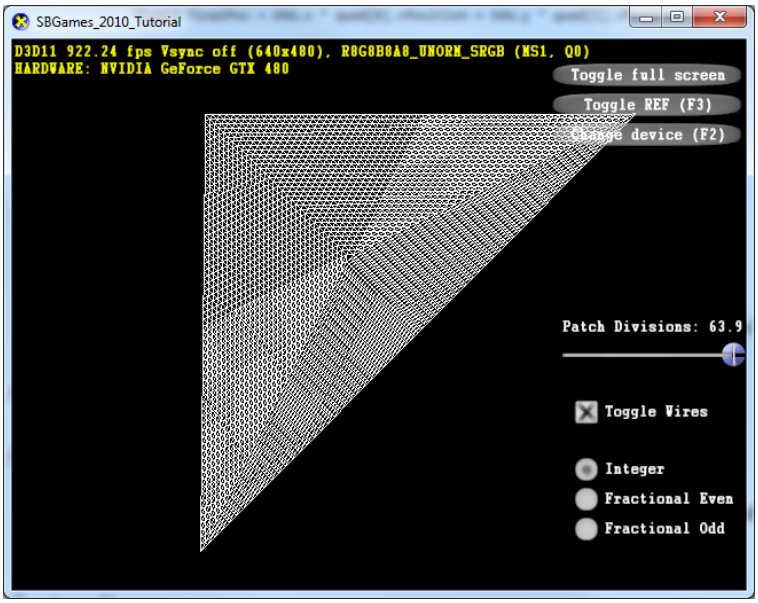


**Figure 7**

### 4.3 Creating parametric surfaces in the GPU

In this example we will use the new pipeline to create parametric surfaces in the GPU. We are going to pass just one vertex to the GPU. This vertex is just to enable the pipeline and do a draw call. Also, if the programmer wants to place the surface created somewhere that's not the origin, a properly world matrix should be passed to the shader. A quad domain will be tessellated by the *Tessellator* and in the *Domain Shader* we will displace each vertex created according to the parametric equation of the surface desired.

First of all we need to configure the *Input Assembler* to accept a patch with one control point. This control point is just to be able to do a draw call, since you are can't have a patch with no control points.

```
      pd3dImmediateContext->IASetPrimitiveTopology(
D3D11_PRIMITIVE_TOPOLOGY::D3D11_PRIMITIVE_TOPOLOGY_3_CONTROL_POINT_PATCHLIST);
```

Now we will code the shader. Once again the *Vertex Shader* will be just a pass-through shader and the constant buffer has the *tessellation factor* and the View*Projection matrix:

```
cbuffer cbPerFrame : register( b0 )
{
    matrix g_mViewProjection;
```

```
    float  g_fTessellationFactor;
};
struct VS_CONTROL_POINT_INPUT
{
    float3 vPosition        : POSITION;
};
struct VS_CONTROL_POINT_OUTPUT
{
    float3 vPosition        : POSITION;
};
VS_CONTROL_POINT_OUTPUT BezierVS( VS_CONTROL_POINT_INPUT Input )
{
    VS_CONTROL_POINT_OUTPUT Output;
    Output.vPosition = Input.vPosition;
    return Output;
}
```

The *Hull Shader* output structures will be the same as the first example. At the constant part we will set all edges and inner tessellation factors with the value that is in the constant buffer. It is nice to change this *tessellation factor* value with a slider in the GUI so you are able to see the parametric surface being "built" on the GPU. Here is the *Hull Shader* snippet:

```
struct HS_CONSTANT_DATA_OUTPUT
{
    float Edges[4]             : SV_TessFactor;
    float Inside[2]            : SV_InsideTessFactor;
};

struct HS_OUTPUT
{
    float3 vPosition           : BEZIERPOS;
};



HS_CONSTANT_DATA_OUTPUT ConstantHS( InputPatch<VS_CONTROL_POINT_OUTPUT, 4> ip,
                                    uint PatchID : SV_PrimitiveID )
{
    HS_CONSTANT_DATA_OUTPUT Output;
    float TessAmount = g_fTessellationFactor;
    Output.Edges[0]  =  Output.Edges[1]  =  Output.Edges[2]  =  Output.Edges[3]=
TessAmount;
    Output.Inside[0] = Output.Inside[1] = TessAmount;

    return Output;
}

[domain("quad")]
[partitioning("fractional_odd")]
[outputtopology("triangle_cw")]
[outputcontrolpoints(4)]
[patchconstantfunc("ConstantHS")]
HS_OUTPUT HS( InputPatch<VS_CONTROL_POINT_OUTPUT, 4> p,
                    uint i : SV_OutputControlPointID,
                    uint PatchID : SV_PrimitiveID )
{
    HS_OUTPUT Output;
    Output.vPosition = p[i].vPosition;
    return Output;
}
```

Now the *Domain Shader*. Here you have to choose a parametric surface that you want to create in the GPU. Let's choose a sphere. The parametric equation of a sphere is :

$$(x, y, z) = (r * \sin\varphi * cos\theta, r * sin\varphi * sin\theta, r * cos\,\varphi)$$

Where $r$ is the radius of the sphere and $\varphi = [0..\pi], \theta = [0..2\pi]$. Remember that the UV coordinate that comes from the *Tessellator* to the *Domain Shader* is with the range $[0..1]$. What we are going to do is $\varphi = U * \pi$ and $\theta = V * 2\pi$ and apply the above equation. Also, let's put a color in the *Domain Shader* output so the surface will be colored even in the wireframe mode. The color will simply be the position of the vertex normalized plus a bias to avoid the black color. That's because our background is black and we do not want our mesh to be with the same color of the background.

```
[domain("quad")]
DS_OUTPUT DS( HS_CONSTANT_DATA_OUTPUT input,
                    float2 UV : SV_DomainLocation,
                    const OutputPatch<HS_OUTPUT, 4> quad )
{
    DS_OUTPUT Output;

    float pi2 = 6.28318530;
    float pi = pi2/2.0f;
    float R = 1.0;
    float fi = pi*UV.x;
    float theta = pi2*UV.y;
    float sinFi,cosFi,sinTheta,cosTheta;
    sincos( fi, sinFi, cosFi);
    sincos( theta, sinTheta,cosTheta);

    float3   spherePosition   =   float3(R*sinFi*cosTheta,   R*sinFi*sinTheta,
R*cosFi);
    Output.vColor = float3(normalize(spherePosition) + 0.4);

    Output.vPosition = mul( float4(spherePosition,1), g_mViewProjection );

    return Output;
}
```

The *Pixel Shader* just outputs the color of the input vertex.

```
float4 SolidColorPS( DS_OUTPUT Input ) : SV_TARGET
{
    return float4( Input.vColor, 1 );
}
```

This is the picture of our GPU generated sphere:

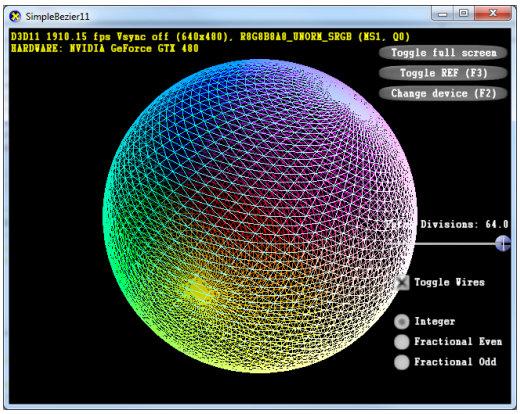


**Figure 8**

Others parametric surfaces generated with the same technique as shown on Figure 11:

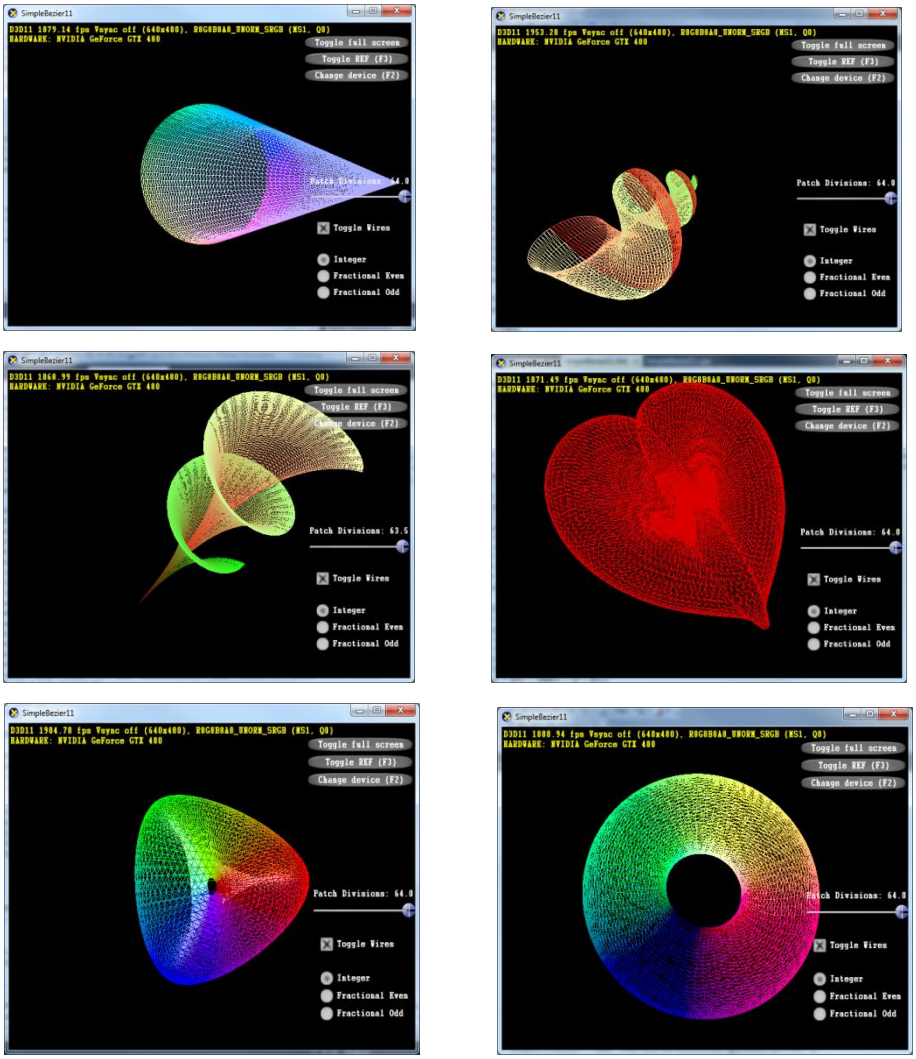


**Figure 11**