Vol. [VOL], No. [ISS]: 1–20

Rendering Tubes from Discrete Curves with Anti-Aliasing and Continuous LOD with Hardware Tessellation

Gustavo Nunes, Alexandre Valdetaro, Alberto Raposo, Bruno Feijo

Pontifical Catholic University of Rio de Janeiro, Dept. of Informatics

Rodrigo de Toledo Federal University of Rio de Janeiro, DCC

Abstract. This paper introduces an approach to render 3D tubes using hardware tessellation. The proposed technique explores the new GPU pipeline, hull, tessellator and domain stages, so that the tube mesh can be created in the last possible step inside the pipeline, reducing the bottleneck of the CPU to GPU bandwidth, enabling real time rates for models with a massive number of tubes. As the proposed solution creates the meshes dynamically, it also enables to view the tubes even when the camera is far-away from them, without the typical aliasing problem. This approach demonstrated to be a well balanced solution for 3D tube rendering in CAD applications. We had considerable gains on four major concerns in real time rendering: performance, image quality, simplicity of implementation and memory consumption.

1086-7651/06 $0.50 per page

1. Introduction

3D Tubes can be a solution for a multitude of problems in a wide range of areas. To illustrate such cases we will walk through some examples. Visualization of the path taken by particles in time, in the form of streamlines is an appropriate case. Flow simulation is another example of this kind of application [7]. In medicine, there are multiple applications which can make use of tubes, for example visualization of white matter tracts [6] and heart fibrillation [4]. In physics, vector fields can also be viewed as a group of tubes. In the oil industry, tubes can also be used in 3D simulations and for rendering specific real objects. Such real objects as wells and risers can be rendered as 3D tubes and were the main motivation for the technique, as shown in Figure 1.

3D tubes are represented through a set of points in R3 and a scalar for its radius. In the conventional graphics pipeline, one needs to construct a polygonal mesh based on this representation before sending it to the GPU. There are also ray-casting approaches of 3D tubes rendering demonstrated to be too ALU intensive for thousands of tubes and they also may provide some unwanted artifacts [8]. A naive approach such as rendering the tubes as simple lines is reasonable from the performance point of view but lacks in visual quality and suffers with aliasing.

1.1. The Problem

We faced a massive oil field with wells and ducts represented as sets of points by GIS software. We started by developing a technique that could render the 3D tubes without the aid of tessellators new pipeline. However, there was an intense bottleneck in our visualizer at CPU to GPU bandwidth while rendering thousands of ducts, wells and risers. Our tubes were created in the CPU and passed to GPU every frame. Also, in CAD applications it is also important to see the 3D tubes even when the camera is far-away from them. This requirement resulted in an unwanted aliasing.

By approaching these two main requirements, bandwith bottleneck and aliasing, we then developed a new technique based on our previous one to solve these problems. The new solution presented successful results.

1.2. The Technique

In this paper, we present the technique of our new solution, The technique explores the new GPU pipeline, hull, tessellator and domain stages (detailed further on), so that the tube mesh can be created in the last possible step

inside the pipeline. In order to achieve such delay with mesh creation, we consider two ways to construct the 3D tubes from a set of points: they can be either directly connected or used as control points of a spline. In both cases we only pass to the GPU a set of points, and some per-section auxiliary data (detailed later on), and with this data the mesh can be created. Also, as our solution creates the meshes dynamically we were also able to solve the aliasing issue. Moreover, our approach exercises every new stage of the programmable pipeline and it is also simple and straightforward. Our approach demonstrated to be a well balanced solution for 3D tube rendering in CAD applications. We had considerable gains on four major concerns in real time rendering: performance, image quality, simplicity of implementation and memory consumption.

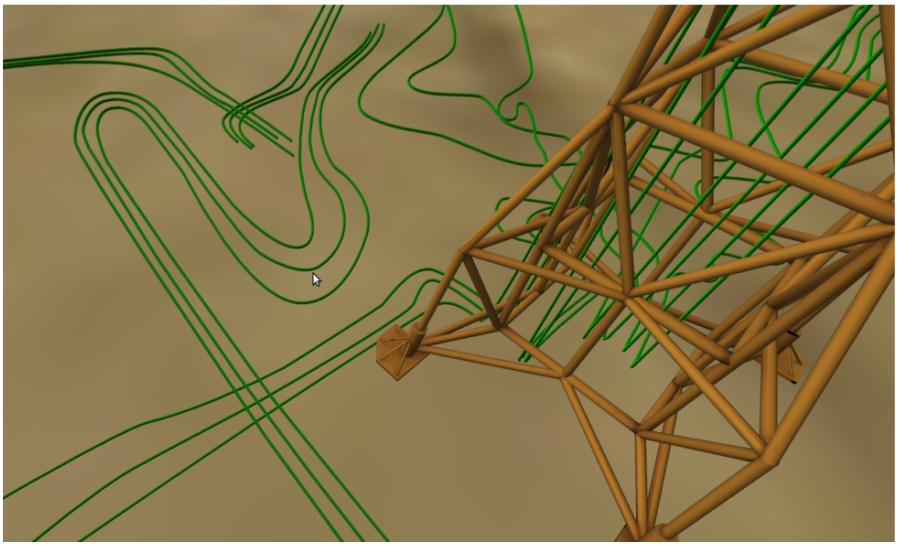

Figure 1. 3D tubes rendered with our technique in an oil field visualizer.

2. Background

Since GPU became programmable, there has been an effort dedicated to render 3D tubes more efficiently. Restricted to first programmable stages (vertex and fragment), some work has used GPU ray casting combined with rasterization [6]; [7]. This technique, also known as extended GPU primitives [8], may present some unwanted artifacts in the case of 3D tubes.

Extended GPU primitives need a supporting rasterized primitive to run the ray casting algorithm, and quadstrips are the natural choice for cylinders.

However, in high curvature location, when it is aligned with the viewing angle, quadstrips flip direction, preventing correct ray casting. There are two approaches already proposed in previous work to solve this limitation: the use of an extra sprite circle [6] and a locally increase quad-strip resolution [7]. In both cases, an extra pass is necessary to relieve the visual problem. In our work, we have not used any ray cast algorithm and the tubes are solely rendered as polygonal meshes.

In the new GPU pipeline, with hull and domain programmable stages, it is possible to create vertices inside the GPU to accelerate visualization (although geometry stage could also create vertices on the fly, it is not tailored for massive polygon creation purposes due to the low performance of the geometry shader).

There are already some solutions based on real-time vertices generation using the new programmable GPU stages for different applications: terrains (Valdetaro et al., 2010) and subdivision surfaces [5], but, to the best of our knowledge, those new stages havent been used to generate tubes in massive models.

3. The Algorithm

The following sections explain the algorithm used in our technique. The workflow is divided in three main parts. In the first part, which happens during a pre-processing phase, we categorize the different kinds of point sequences that can be used as input and approach them in different ways according to the category. We extract all the necessary data needed to feed the graphics pipeline from this point sequence and move on. In the second part, inside the GPU we create the geometry that will be rendered. In the third part, we then transform the created geometry to a generalized cylinder in world coordinates. We describe these three parts thoroughly in the following sections.

3.1. Preparing Data for Rendering

3.1.1. Defining the 3D tubes from Point Sequences

3D tubes are considered generalized cylinders with a constant section area. They are guided by a given curve as its path in 3D space. Our algorithm creates a geometry that reconstructs these generalized cylinders. In order to create the geometry, it is necessary to obtain the sequence of points that define the path. As the points are sequential, we can assume that each point is connected to its neighbours by a straight line segment. Therefore, each point shall have a numerical derivative. However, point sequences must be

interpreted in two different ways depending on the case. There is a good case, where the point sequence is a curve, representing the actual tube (Blue line in Figure 2). There is still a bad case when the point sequence is a mere path that a smooth cylinder should touch. In such situation, the tube would have jagged corners (Black line in Figure 2). This unwanted case can be simply converted to a good case with a small pre-processing. We need to transform the sparse points defining a mere path to a more refined set of points that defines the central line of the cylinder, as we show in Figure 2. We enhance the point sequence precision by smoothly interpolating with Catmull-Rom splines [2]. The interpolation increment factor required by the spline decides how many points are going to be generated.

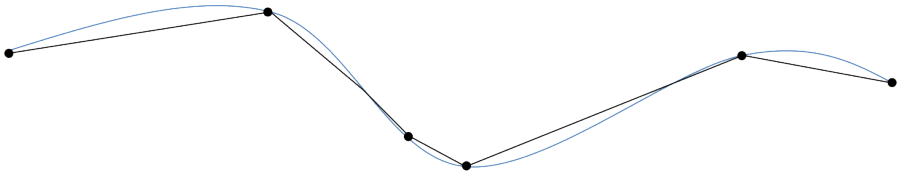

Figure 2. A set of points in black representing a tube's path. In blue the central line of a tube

3.1.2. Making the Best Approximation Possible

Our algorithm approximates a continuous curved generalized cylinder by a discrete set of smaller straight cylinders. Naturally, the point amount enhances the reconstruction precision. Moreover, spots containing high derivative values, such as turns and knees require more points (Figure 3). However each point will have a geometry associated with it during the rendering. Thus we maintain the point amount to a minimum necessary (as seen in Figure 3). The problem is rather simple, we need to cull points that have a low derivative and maintain points with high derivative. This filter is applied to the internal points by navigating through the point sequence and selecting the relevant points whenever the direction deviation of the curve hits a threshold.

The desired amount of geometry determines the number of selected points. We describe the geometry generation in further sections.

3.1.3. Tangents, Normals and Binormals

Our algorithm needs a normal and binormal in order to transform the generated geometry into a 3D tube, as is explained in next section. The normal $\vec{N_i}$ of a point $\mathbf{P_i}$ is any vector orthogonal to the tangent $\vec{T_i}$ of the curve at $\mathbf{P_i}$. The binormal $\vec{B_i}$ of $\mathbf{P_i}$ is any vector orthogonal to both $\vec{N_i}$ and $\vec{T_i}$. The tangent $\vec{T_i}$ is the numerical derivative at $\mathbf{P_i}$. For every point $\mathbf{P_i}$ that we selected

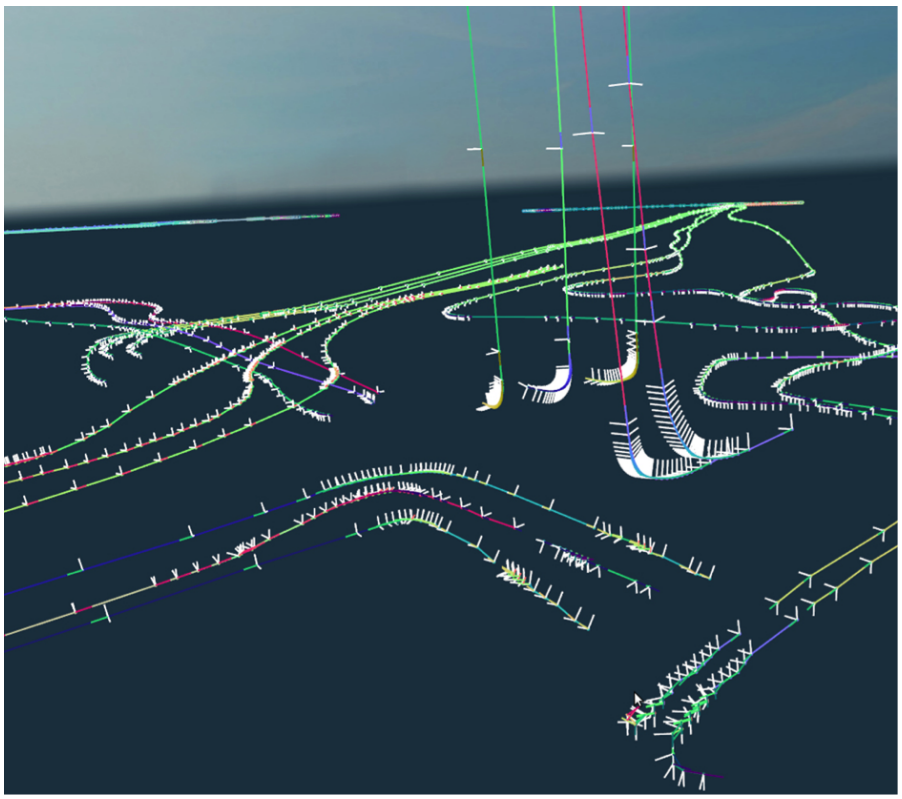

Figure 3. Point selection with associated tangent, normal and binormal along a tubes curve. High derivative areas require more points for a precise reconstruction.

in the previous section we must store $\vec{N}_i$ and $\vec{B}_i$. Figure 3 shows a sample of tangents normals and binormals along tubes curves.

3.2. Creating the geometry

The geometry of the tubes mesh is generated by the tessellator (Section 4), resulting in a very efficient rendering. The CPU-GPU bandwidth is used only to invoke the pipeline and pass some topological information. As is explained in Section 4, the tessellator can output a set of generated vertices in a chosen domain. We choose a quad topology, thus our vertices fall into a UV [0..1]2 domain (Figure 4E). The goal is to transform a regular 2D grid and a set of points into a 3D tube. Let us isolate the problem to one single point $\mathbf{P_i}$ of the curve. For now, consider that our domain is a straight line L_i (Figure 4F)

composed by a discrete set of points $\{\mathbf{p_0}, ..., \mathbf{p_n}\}$. Our objective is to transform L_i into a circular orthogonal cut of the tube at $\mathbf{P_i}$ (Figure 4H). In order to convert each point of L_i to the cross section, we start by transforming every point $\{\mathbf{p_0}, ..., \mathbf{p_n}\}$ from 2D line to a point $\{\mathbf{p'_0}, ..., \mathbf{p'_n}\}$ in a 3D circle (Figure 4G) on the $y = 0$ plane by directly transforming the position of $\mathbf{p_i}$ relative to the total length of L_i to an angle:

$$\Theta = 2\pi(\frac{\mathbf{p_i}}{\mathbf{p_n}}) \tag{1}$$

Consider $\mathbf{p_i}/\mathbf{p_n}$ as the length of $\mathbf{p_i}$ along L_i. Now that the points are mapped to a simple circle on $y = 0$ plane, the problem is to transform this circle to the tubes cross section circle at $\mathbf{P_i}$. This transformation is done as the calculation of reference frames along a space curve [1], where the points final position $\{\mathbf{p''_0}, ..., \mathbf{p''_n}\}$ (Figure 4H) in world space is:

$$F_p = (\mathbf{P}_{ix} + \mathbf{p'}_x N_x + \mathbf{p'}_z B_x, \mathbf{P}_{iy} + \mathbf{p'}_x N_y + \mathbf{p'}_z B_y, \mathbf{P}_{iz} + \mathbf{p'}_x N_z + \mathbf{p'}_z B_z) \tag{2}$$

Where N and B stand for the Normal and Binormal at $\mathbf{P}_i$. Further details are explained at the implementation section. With the algorithm to convert a line to a cross section in world coordinates, there is just a small step to be able to render a full tube. As the Tessellator gives us a 2D regular grid of vertices, we just need to transform every line of the grid to a cross section as described by Figure 4. The circles radius is scaled adaptively as explained on Section 3.3, but for now we just assume it has a fixed value. We need a regular grid with 64 lines in one axis. Therefore, there is the need to maintain the tessellation factor of one axiss edges and intern factor as 64. As for the other axiss edges and intern factor, they have no fixed value, their only requeriment is equality as we need a regular grid. Thus, we make these non-fixed factors directly dependent to the camera distance to the quad, resulting in an on-the-fly LOD control.

4. The Tesselator Pipeline

The first versions of the programmable pipeline were not able to create primitives in the graphics card. In 2006, DirectX 10 capable graphics cards were launched with the Geometry Shader stage. This stage is able to create new primitives in the GPU, but it is not very effective. If one codes a Geometry Shader that adds many primitives in a single call, the performance decreases drastically. In order to solve this issue, hardware vendors added a new tessellator unit to the graphics pipeline [3]. This new pipeline part has two programmable and one fixed but configurable part. We show a diagram of

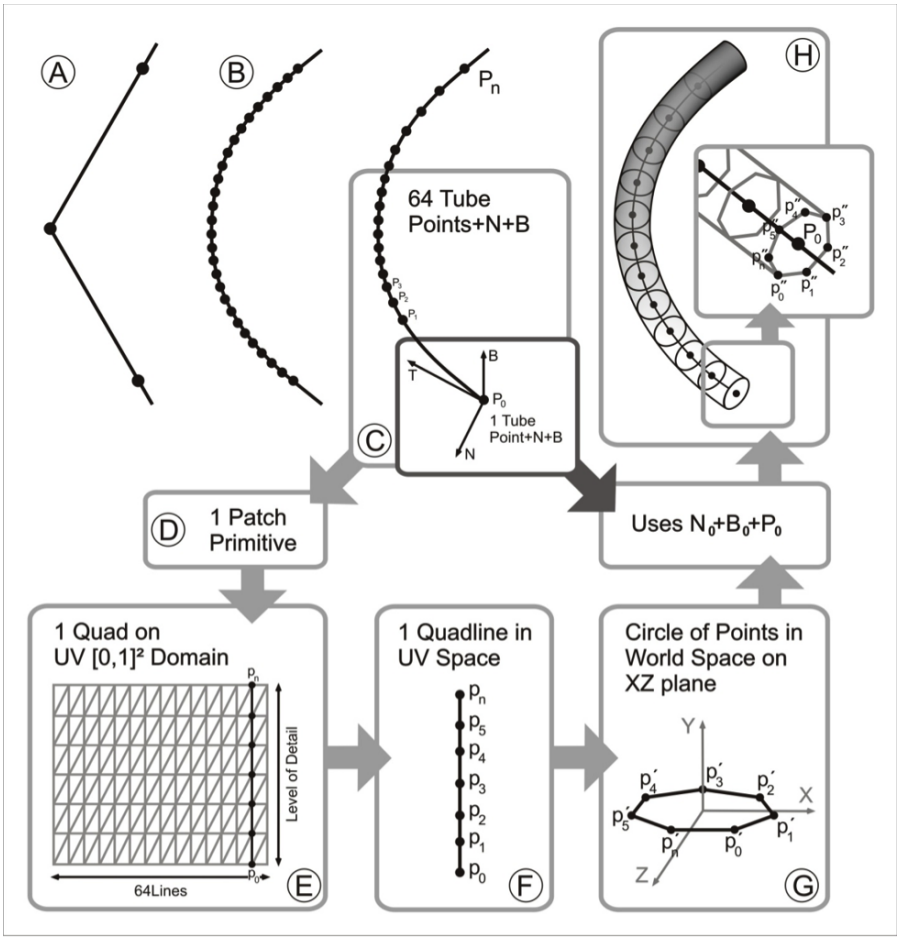

Figure 4. A) A bad case point sequence. B) the bad point sequence after interpolating with Catmull-Rom spline. C) The selection of relevant points from the sequence and calculation of Tangent Normal and Binormal. D) Every 64 points imply in one patch primitive. E) The patch primitive is a quad domain with 64xLOD lines. F) 1 line from the quad with a number of points determined by the LOD. G) Every line point is first transformed into a circle in 3D space on y=0 plane. H) Using the Normals and Binormals every point is transformed from the circle to a tube cross section.

this pipeline in Figure 5. The programmable stages are the Hull Shader and the Domain Shader, while the Tessellator is the only-configurable one. Another new concept is the lack of pre-defined topology. It is not specified to the Input Assembler if the incoming vertices are triangles, quads or lines. All

primitives are patches. Patches are primitives with no explicit topology, they can be interpreted as triangles, quads, control points for Bezier surfaces or just points that represents some sort of information for the shader programmer. A patch has from 1 to 32 control points. The Hull Shader is responsible for transforming the base of the input patch (e.g. quad Bezier bi-cubic) and also for specifying to the Tessellator the amount of subdivision for the tessellated domain. The domain may be quad, triangle or line. The tessellation levels are specified per edge and for the internal part of the domain. This allows to subdivide the surface without cracks at the junctions with other patches. The Tessellator is responsible for creating the vertices within the chosen domain and to output for the Domain Shader the coordinates (within the range $[0..1]^2$) where the vertices were created. It is Domain Shader's responsibility to move the created vertices to the correct place in the world.

5. Implementation

5.1. Pre-processing

In order to pass the correct normal, bi-normal and central point information for each tube cross section we need to execute an $O(n)$ pre-processing step as explained in Section 3. This information are passed to the GPU as control points information to be used by the domain shader. Each invocation of the tessellator builds a quad that is transformed into a 3D tube part by the domain shader. The tessellator is able to subdivide a quad up to 64 times in each axis. As we demonstrate in Figure 4, one axis of the quad domain has maximum tessellation factor. Each line of this axis is transformed to follow the cross-section defined by the point sequence that represents the tubes path. The other axis tessellation values are flexible for level of detail values as we show in Figure 7.

5.2. Setting up the input assembler

On a quad domain the tessellator is able to create 64 lines on each axis. In order to make the most of tessellator effort we should be able to pass 64 differents positions, normals and bi-normals to the tessellator for each tessellated quad. But the input assembler only accepts 32 control points per quad. We are able to work this issue around by passing two positions, normals and bi-normals per control point. So the input assembler is configured to accept a patch with 32 control points and each of these control points should carry information to correctly position two cross-sections. The following structure snippet shows the information that we need for each control point:

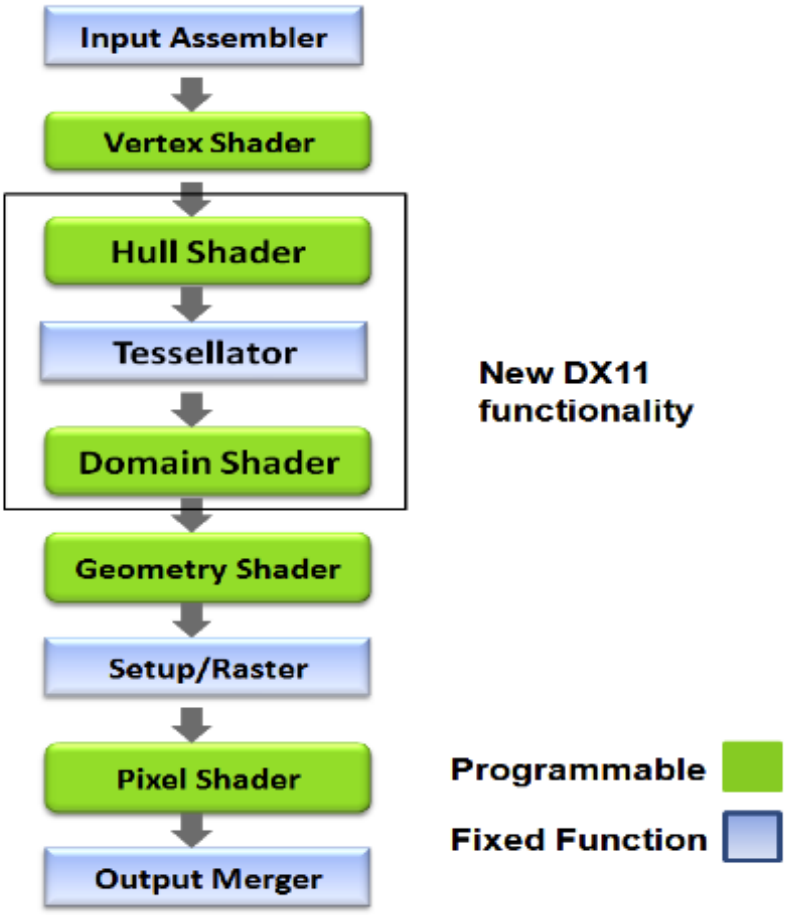

Figure 5. The new Tessellator pipeline

```
struct VS_CONTROL_POINT_INPUT
{
  float3 vPosition   : POSITION;
  float3 vBinormal   : TEXCOORD0;
  float3 vNormal     : NORMAL0;
  float3 vPosition2  : TEXCOORD1;
  float3 vBinormal   : TEXCOORD2;
  float3 vNormal2    : TEXCOORD3;
};
```

These control points are the only vertex buffer per-quad data that our algorithm passes to the GPU. Choosing a quad domain the tessellator is able to create up to 8192 triangles. Also a radius parameter may be optionally passed.

5.3. The Constant Hull Shader part

The Hull Shader constant part is the part of the new pipeline where the tessellation factor must be passed to the tessellator. Adaptive tessellation calculations are also made here. The tessellation factor is in the range [1..64]. There are six tessellation factors in the quad domain, four of them are edge factors and two are inside factors (one for each quad axis). Our algorithm transforms each line of the quad domain into a circle as explained in Section 3. In order to take maximum advantage of the tessellation power, we need to maximize the number of lines in one axis of the quad domain. Consequently, to achieve our objective, we must set the tessellation factor of two edges and one inside axis to 64. This configuration guarantees that we transform the maximum amount of lines into tube sections. The three remaining parameters are varied adaptively in a straightforward Level-of-Detail according to the distance from camera or edge screen-space size. Moreover, the camera position must be passed as a constant per-frame parameter to the shader. Figure 6 shows the same tube with two different LODs configured by the Hull Shader.

The following code snippet shows our Hull Shader constant part:

```
HS_CONSTANT_DATA_OUTPUT ConstantHS( InputPatch<
    VS_CONTROL_POINT_OUTPUT, INPUT_PATCH_SIZE> ip,  uint
     PatchID : SV_PrimitiveID )
{
   HS_CONSTANT_DATA_OUTPUT Output;
//these factors should be configured according to an
    adaptive tessellation                              //
    parameter(camera or edge screen-space size)
Output.Edges[0]= Output.Edges[2]= Output.Inside[1] =
    g_fTessellationFactor;
//maximum tessellator factor for one axis
Output.Edges[1] = Output.Edges[3] = Output.Inside[0] =
    64;

return Output;
}
```

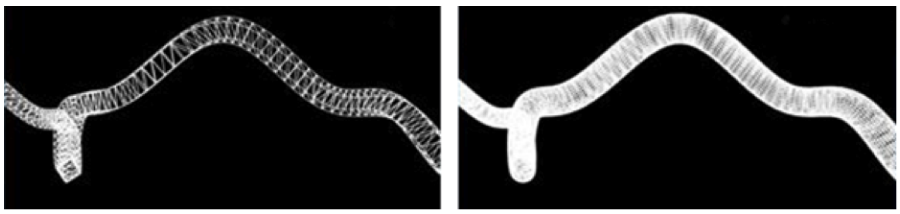

Figure 6. Left Tube with low LOD. Right Tube with high LOD.

5.4. The Main Hull Shader

A good advantage of generating the tube mesh in the GPU is that the radius might be set dynamically. As explained in Section 3.3, with this feature we are able to solve aliasing problems. The Hull Shader main part is executed once per control point, it is here that we dynamically control the radius of each part of our quad (which will be transformed into a cross section later in the Domain Shader). Let $\mathbf{P}_i$ be the central point of a cross section as explained above (there are 2 such points per control-point as explained in section 5.2). First we need to find the current depth of $\mathbf{P}_i$ in screen space. Multiplying $\mathbf{P}_i$ by the ViewProjection matrix yields $\mathbf{P}_i$ in screen space ($\mathbf{P}_{screen}$). The depth is obtained dividing the z-coordinate of $\mathbf{P}_{screen}$ by the w-coordinate of it:

$$\mathbf{P}_{screen} = \mathbf{P} * ViewProjection \tag{3}$$

$$\mathbf{P}_{depth} = \frac{\mathbf{P}_{screenZ}}{\mathbf{P}_{screenW}} \tag{4}$$

For simplicity we assume a canvas with aspect ratio of 1. Let res be the resolution of the canvas. $\mathbf{S}_1 = (0, 0, \mathbf{P}_{depth}, 1)$ and $\mathbf{S}_2 = (2/res, 0, \mathbf{P}_{depth}, 1)$ represent a pixel width in screen space. Our goal is to find the size of a pixel in world space at the $\mathbf{P}_{depth}$s depth. Multiplying $\mathbf{S}_1$ and $\mathbf{S}_2$ by the inverse of ViewProjection matrix and dividing by the w-coordinate yields both points in world space. The pixel size in world space is the length of the vector formed by those points:

$$PixelSize_{world} = ||\frac{\mathbf{S}_2 * ViewProj{-}1}{S_{2w}} \frac{\mathbf{S}_1 * ViewProj{-}1}{S_{1w}}|| \tag{5}$$

The following code snippet shows our Hull Shader main part:

```
[domain("quad")]
[partitioning("integer")]
[outputtopology("triangle_cw")]
[outputcontrolpoints(32)]
```

```
[patchconstantfunc("ConstantHS")]
HS_OUTPUT HS( InputPatch<VS_CONTROL_POINT_OUTPUT, 32> p
    ,
                  uint i : SV_OutputControlPointID,
                  uint PatchID : SV_PrimitiveID )
{    HS_OUTPUT Output;
    Output.vPosition = p[i].vPosition;
    Output.vBinormal = p[i].vBinormal;
    Output.vNormal = p[i].vNormal;
    Output.vPosition2 = p[i].vPosition2;
    Output.vBinormal2 = p[i].vBinormal2;
    Output.vNormal2 = p[i].vNormal2;
    float4 pScreen = mul(float4(p[i].vPosition,1),
        g_mViewProjection);
    float pDepth = pScreen.z/pScreen.w;
    float4 S2 = float4(1.0f/res,0,pDepth,1);
    float4 S1 = float4(0,0,pDepth,1);
    float4 worldPixel2 = mul(S2,g_mInvViewProjection);
    float4 worldPixel1 = mul(S1,g_mInvViewProjection);
    worldPixel2.xyz = worldPixel2.xyz/worldPixel2.www;
    worldPixel1.xyz = worldPixel1.xyz/worldPixel1.www;
    float PixelSizeWorld = length(worldPixel2.xyz -
        worldPixel1.xyz);

    if(radius <= 2* PixelSizeWorld) radius = 2*
        PixelSizeWorld;
    Output.vRadius = radius;
    return Output;
}
```

5.5. The Domain Shader

The Domain Shader receives UV coordinates in the $[0..1]^2$ range. These coordinates specify where in the quad domain each new vertex generated by the tessellator is located. Each invocation of the domain shader corresponds to one vertex of the tessellated quad. First, we need to transform each quad line into a circle. Let $\mathbf{p}'_i$ be a point of one circle in the $y = 0$ plane and r is the radius of the tube (the radius comes from the Hull Shader output). The transformation is as following:

$$\Theta = 2\pi V \tag{6}$$

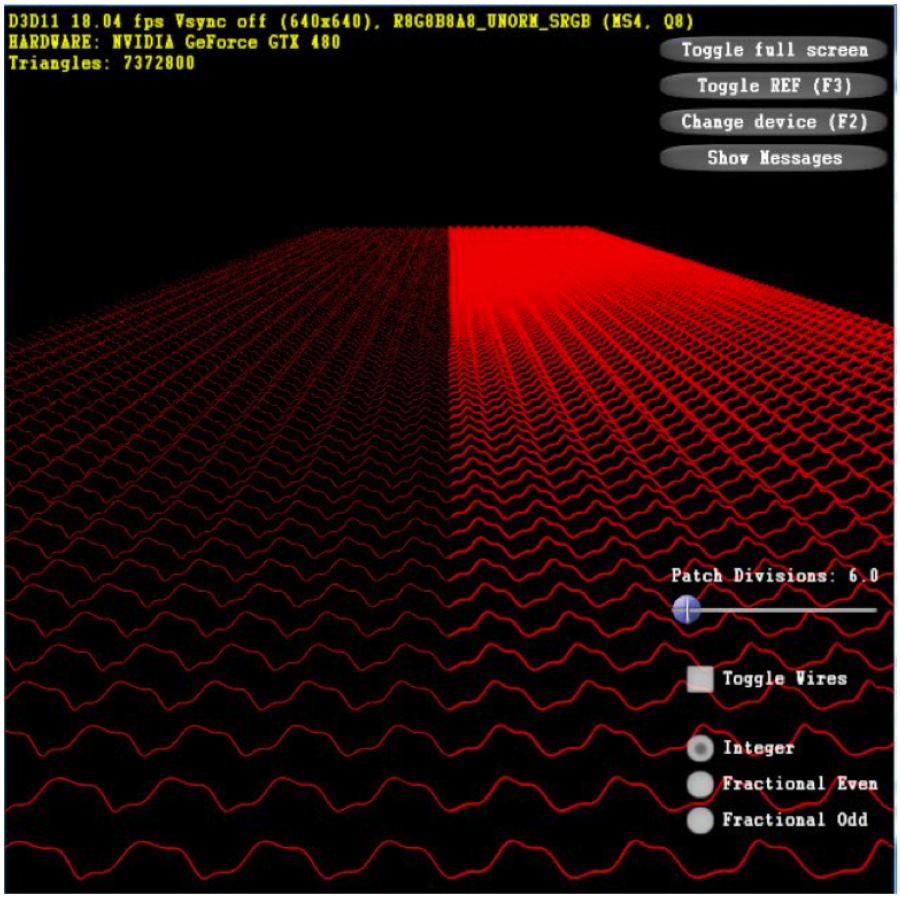

Figure 7. The Same scene: on the left with 16x GPU anti-aliasing (AA). On the right without GPU AA but with our radius correction AA

$$\mathbf{p'}_i = (rcos\Theta, 0, rsin\Theta) \tag{7}$$

Now the position ($\mathbf{P}_i$), normal ($\vec{N}$) and binormal ($\vec{B}$) must be fetched from control points data. As each control point has information for two cross sections we must get the right information for each Domain Shader invocation now. We simply multiply the U coordinate by 63 and do an even/odd choice with the modulus operator.

Now we need to correctly position and orient the tubes cross-section $\mathbf{C}$. We convert $\mathbf{p}'_i$ to $\mathbf{p}''_i$ according to equation 2. Then, $\mathbf{p}''_i$ is transformed by the view and projection matrices and is set as one of the domain shader outputs.

Another domain shader output is the vertex normal, $\vec{N}_v$. It can be easily found as following:

$$\vec{N}_v = (\mathbf{F}_p - \mathbf{P})/||\mathbf{F}_p - \mathbf{P}|| \tag{8}$$

The following code snippet shows our Domain Shader:

```
[domain("quad")]
DS_OUTPUT DS( HS_CONSTANT_DATA_OUTPUT input,
                      float2 UV : SV_DomainLocation,
                      const OutputPatch<HS_OUTPUT, 32> p)
{
        DS_OUTPUT Output;
        float v = UV.y;
        float u = UV.x;
        float pi2 = 6.28318530;
        float theta = v*pi2;
        float sinTheta,cosTheta;
        sincos(theta,sinTheta,cosTheta);
        int index = 63*u;
        float3 N,B,P;
        if( index % 2 != 0 )
        {
                index = (int)(index/2.0f);
                N = p[index].vNormal2;
                B = p[index].vBinormal2;
                P = p[index].vPosition2;
        }
        else
        {
                index = (int)(index/2.0f);
                N = p[index].vNormal;
                B = p[index].vBinormal;
                P = p[index].vPosition;
        }

        int radius = p[splineIndex].vRadius;
        float3 C = float3(raio*cosTheta,0,raio*sinTheta
            );
        float3 worldPos;

        worldPos.x = P.x + C.x*N.x + C.z*B.x;
        worldPos.y = P.y + C.x*N.y + C.z*B.y;
        worldPos.z = P.z + C.x*N.z + C.z*B.z;
```

```
        float3 normal = normalize(worldPos -
            cylinderPos);

        Output.vPosition = mul( float4(worldPos,1),
            g_mViewProjection );
        Output.vNormal = normal;
        Output.vWorldPos = float4(worldPos,1);

    return Output;
}
```

6. Results

Our tests were executed on an Intel Core i7 920 with a Nvidia GTX480 and 6GB of RAM. We did a comparison between approaches where all the triangles are passed from the CPU to the GPU with the tessellator disabled versus our GPU tessellation method. No acceleration algorithm such as frustum culling was used. The tests purpose was to measure the efficiency obtained with the trade of memory bandwidth for ALU GPU operations. It is important to note that optimizations are possible with the tessellator pipeline such as avoiding to subdivide back faced patches and tessellating according to screen-space edge patch size to avoid sub-pixel triangles. We show in Table 2 the results comparing the frame rate with and without our technique. The results show that our algorithm has a significant gain over the non-tessellator based approach. Furthermore, we are capable to accomplish up to 184 million triangles with interactive frame rates. Using the CPU approach with at least position and normals information per-vertex would result in a prohibitory 13.2GB vertex buffer size to be passed from the CPU to the GPU while our approach uses only 103.7MB for the same amount of triangles as we illustrate in Table 1. Moreover, our anti-aliasing solution demonstrated to be very efficient with a low frame rate drop against the one with no aliasing treatment. We expose in Figure 8 a graph with our algorithms gain percentage in FPS against a regular approach without anti-aliasing.

7. Improving per frame memory consumption even more

The technique explained above is a straightforward and performance balanced method of rendering 3D tubes. Although it is not our case, one might have an even more per frame memory consumer application. Instead of passing all the

Table 1. FPS comparison for several amounts of triangles.

Millions of Triangles	FPS		
	CPU	Our technique with AA	Our technique without AA
184,3	0	10	11
28,6	1	59	60
12,3	64	102	104
11	71	111	114
9,8	79	124	130
8,6	89	136	147
7,4	99	145	167
6,1	120	178	194
4,9	141	210	231
3,7	181	247	286
2,5	242	320	373
1,2	335	418	500
0,61	518	621	720
0,3	730	835	911
0,12	930	999	1050
0,061	970	1050	1112
0,0061	1100	1111	1135

data (Positions, Normals, Bi-Normals) through the control points, one might use only one control point per quad domain and map a texture with the information of positions, normals and bi-normals. The single control point per quad would bring the position where the texture should be fetched for the data. This approach would reduce the memory usage by a factor of 64. According to Table 1 and Table 2, this method wouldnt have much impact from the second row through the bottom but it might give a performance boost if one intends to draw more than a hundred million triangles.

8. Limitations and Drawbacks

The main limitation of our technique is that it is only compatible with DirectX 11/OpenGL 4 video cards. One might think of implementing it through Geometry Instancing for older cards support. It works, but theres a main issue with the rasterizer performance. Level-of-Detail control through Geometry Instancing is not as flexible as the new pipeline and if the application starts to create too many sub-pixels triangles the rasterizer starts to be the main

Table 2. CPU-GPU memory bandwidth comparison

Millions of Triangles	Memory bandwidth in MB	
	CPU	Our technique
184,3	13270	103.7
28,6	2059	16.1
12,3	886	6.9
11	792	6.2
9,8	706	5.5
8,6	619	4.8
7,4	533	4.2
6,1	439	3.4
4,9	353	2.8
3,7	266	2.1
2,5	180	1.4
1,2	86	0.7
0,61	44	0.3
0,3	22	0.2
0,12	9	0.07
0,061	4	0.03
0,0061	0	0.00

bottleneck of the program due to overshading. Another issue is that the anti-aliasing correction actually grows the radius of the tube in world space. If one has a scene like Figure 1, with other viewable meshes, an adjustment of maximum radius might be necessary to avoid creating an out of proportion scene. This last issue is not a major issue at all; we were able to easily find values that prevent the aliasing and maintain a proportional scene.

Acknowledgments. This work is funded by Petrobras, Brazilian Oil & Gas Company. Alberto Raposo also thanks FAPERJ for the individual support granted (#E-26/102.273/2009).

References

[1] Jules Bloomenthal. Graphics gems. In Andrew S. Glassner, editor, *Graphics gems*, chapter Calculation of reference frames along a space curve, pages 567–571. Academic Press Professional, Inc., San Diego, CA, USA, 1990.

[2] E. Catmull and R. Rom. A class of local interpolating splines. *Proc. of In-*

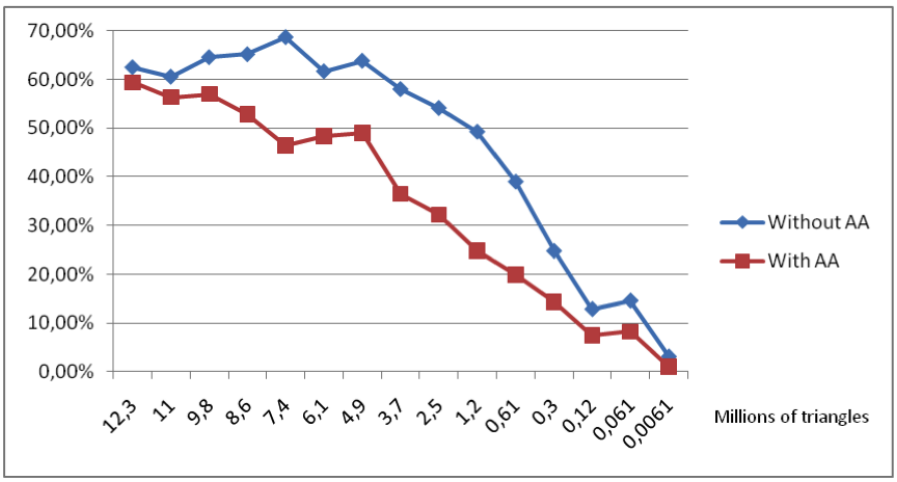

Figure 8. Graph showing the percentage gain in FPS of this algorithm with and without AA against a CPU approach without AA

ternational Conference on Computer Aided Geometric Design, pages 317–326, 1974.

[3] Drone and Oneppo. Direct3d 11 tessellation, 2008.

[4] Polina Kondratieva, Jens Krüger, and Rüdiger Westermann. The application of gpu particle tracing to diffusion tensor field visualization. In *Proceedings IEEE Visualization 2005*, 2005.

[5] Charles Loop, Scott Schaefer, Tianyun Ni, and Ignacio Castaño. Approximating subdivision surfaces with gregory patches for hardware tessellation. *ACM Trans. Graph.*, 28:151:1–151:9, December 2009.

[6] Dorit Merhof, Markus Sonntag, Frank Enders, Christopher Nimsky, Peter Hastreiter, and Guenther Greiner. Hybrid visualization for white matter tracts using triangle strips and point sprites. *IEEE Transactions on Visualization and Computer Graphics*, 12:1181–1188, September 2006.

[7] Carsten Stoll, Stefan Gumhold, and Hans-Peter Seidel. Visualization with stylized line primitives. In *IEEE Visualization*, page 88. IEEE Computer Society, 2005.

[8] Rodrigo Toledo and Bruno Levy. Extending the graphic pipeline with new gpu-accelerated primitives. In *International gOcad Meeting, Nancy, France*, 2004. Also presented in Visgraf Seminar 2004, IMPA, Rio de Janeiro, Brazil.