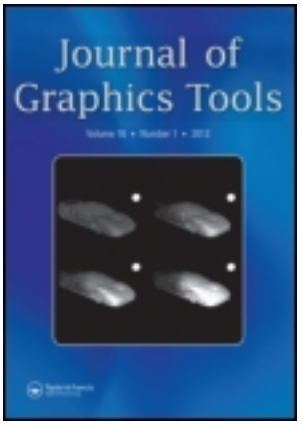


Journal of Graphics Tools
Publication details, including instructions for authors and subscription information:
http://www.tandfonline.com/loi/ujgt21

# Rendering Tubes from Discrete Curves Using Hardware Tessellation

Gustavo Nunes [a], Alexandre Valdetaro [a], Alberto Raposo [a], Bruno Feijó [a] & Rodrigo de Toledo [b]

[a] Pontifical Catholic University of Rio de Janeiro, Dept. of Informatics

[b] Federal University of Rio de Janeiro, DCC

Version of record first published: 02 Aug 2012

To cite this article: Gustavo Nunes, Alexandre Valdetaro, Alberto Raposo, Bruno Feijó & Rodrigo de Toledo (2012): Rendering Tubes from Discrete Curves Using Hardware Tessellation, Journal of Graphics Tools, 16:3, 123-143

To link to this article: http://dx.doi.org/10.1080/2165347X.2012.659610

Vol. 16, No. 3: 123–143



# Rendering Tubes from Discrete Curves Using Hardware Tessellation

Gustavo Nunes, Alexandre Valdetaro, Alberto Raposo, and Bruno Feijó
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 by 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, thereby reducing the bottleneck of the CPU to GPU bandwidth and enabling real-time frame rates for models with a massive number of tubes. Because the proposed solution creates the meshes dynamically, it enables viewing of the tubes even when the camera is far away from them, without the typical aliasing problem. This approach demonstrates a well-balanced solution for 3D-tube rendering in CAD applications. We made considerable gains on four major concerns in real-time rendering: performance, image quality, simplicity of implementation, and memory consumption.

## 1. Introduction

3D-tube rendering has many applications. We will illustrate several of these applications 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 [Stoll et al. 05]. In medicine, there are multiple applications that can make use of tubes, for example, visualization of white matter tracts [Merhof et al. 06] and heart fibrillation



ISSN: 2165-347X print/2165-3488 online

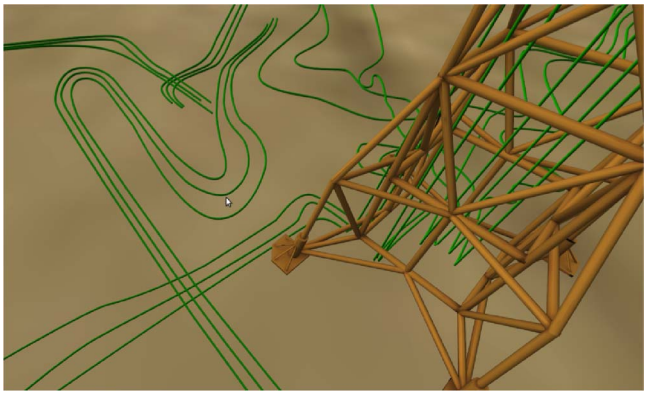

**Figure 1.** 3D tubes rendered with our technique in an oil field visualizer.

[Kondratieva et al. 05]. In physics, vector fields can also be viewed as a group of tubes. In the oil industry, tubes can be used in 3D simulations and for representing specific objects, such as wells and risers. This application is the main motivation for our technique (see Figure 1).

3D tubes are represented by 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-tube rendering, but these have been demonstrated to be too Arithmetic Logic Utility (ALU) intensive for thousands of tubes, and they may provide some unwanted artifacts [Toledo and Levy]. A basic approach such as rendering the tubes as simple lines is reasonable from a performance point of view, but it lacks in visual quality and suffers with aliasing.

### *1.1. The Problem*

We were challenged to represent a massive oil field with wells and ducts as sets of points by using Geographic Information System (GIS) software. Rendering these objects as lines was not an option, as there was a requirement for the tubes to be viewed close up and their size needed to be comparable to other geometry nearby. We started by assigning a designer to create a model for each tube, and we added them as fixed meshes to our scene. Such an approach is not only impractical because we constantly needed the designer to create tubes, it also resulted in an unwanted aliasing effect when the tubes were viewed from a distance. Also, we always needed to see the tubes, even if they were distant and would not otherwise be rendered. So, we started to develop our current technique. At first, we implemented a technique that could render

the 3D tubes from a set of points. The tubes were created in the CPU. Thus, there was no use of the tessellators in the new pipeline. This approach resulted in an intense bottleneck in our visualizer at CPU to GPU bandwidth, as we were rendering thousands of ducts, wells, and risers. Our tubes were created in the CPU and passed to every frame the GPU. Also, in Computer aided design (CAD) applications it can be important to see the 3D tubes, even when the camera is far away from them, so the tubes could not be culled. This requirement resulted in an unwanted aliasing.

Considering these two main obstacles, bandwidth bottleneck and aliasing, we then developed a new technique to solve these problems. The new solution presented successful results for both visual quality and performance benchmarks.

### *1.2. The Technique*

In this paper, we present the technique behind our new solution. This technique explores the new GPU pipeline—hull, tessellator, and domain stages—so that the tube mesh can be created at the last possible step inside the pipeline. In order to achieve such a 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 pass only a set of points to the GPU, and some per-section auxiliary data (detailed later on), and with this data the mesh can be created. Also, because our solution creates the meshes dynamically, we are also able to solve the aliasing issue. Moreover, our approach exercises every new stage of the programmable pipeline with both clarity and simplicity. Our approach has proven to be a well-balanced solution for 3D-tube rendering in CAD applications. We make considerable gains on four major concerns in real-time rendering: performance, image quality, simplicity of implementation, and memory consumption.

## 2. Background

Since GPUs became programmable, there have been many attempts to render 3D tubes more efficiently. Restricted to the first programmable stages (vertex and fragment), some work has used GPU ray casting combined with rasterization [Merhof et al. 06, Stoll et al. 05]. This technique, also known as extended GPU primitives [Toledo and Levy], 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 a high-curvature location, when it is aligned with the viewing

angle, quadstrips flip directions, preventing correct ray casting. There are two approaches, already proposed in previous work, to address this limitation: the use of an extra sprite circle [Merhof et al. 06] and a locally increased quadstrip resolution [Stoll et al. 05]. In both cases, an extra pass is necessary to relieve the visual problem. In our work, we do not use any ray-casting algorithms, and the tubes are rendered solely 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 the geometry stage could also create vertices on the fly, it is not tailored for massive polygon creation because of low performance of the geometry shader).

There are already some solutions based on real-time vertex generation using the new programmable GPU stages for different applications: terrains [Valdetaro et al. 10] and subdivision surfaces [Loop et al. 09], but, to the best of our knowledge, those new stages have not 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 into three main parts. In the first, which occurs during a preprocessing phase, we categorize the different kinds of point sequences that can be used as input and approach them in different ways accordingly. We extract all the necessary data needed to feed the graphics pipeline from this point sequence and move on. In the second part, we create the geometry that will be rendered inside the GPU. In the third part, we 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 constant section areas. They are guided by given curves as their paths 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 each path. Because the points are sequential, we can assume that each point is connected to its neighbors by a straight line segment. Therefore, each point shall have a numerical derivative. However, not every point sequence is a curve representing the actual tube (blue line in Figure 2). There is still

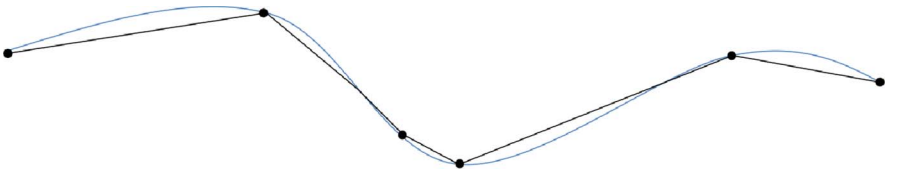

**Figure 2.** A set of points in black representing a tube's path. In blue, the central line of a tube.



a problem when the point sequence is a mere path that a smooth cylinder should touch. In such situations, we transform the sparse points defining a mere path. We enhance the point sequence precision by smoothly interpolating with Catmull-Rom splines [Catmull and Rom 74] so that the point sequence more precisely defines the central line of the cylinder, as shown in Figure 2. The interpolation increment factor required by the spline decides how many points are going to be generated.

#### 3.1.2. Making a Good Approximation

Our algorithm approximates a continuous curved generalized cylinder as a discrete set of smaller straight cylinders. Naturally, the point number 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 number at a necessary minimum, as seen in Figure 3. The problem is rather simple: we need to cull points that have a low derivative and maintain points with a 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 the following section.

### *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 to pass some topological information. As is explained in Section 4, the tessellator can output a set of generated vertices in a chosen domain. We chose 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

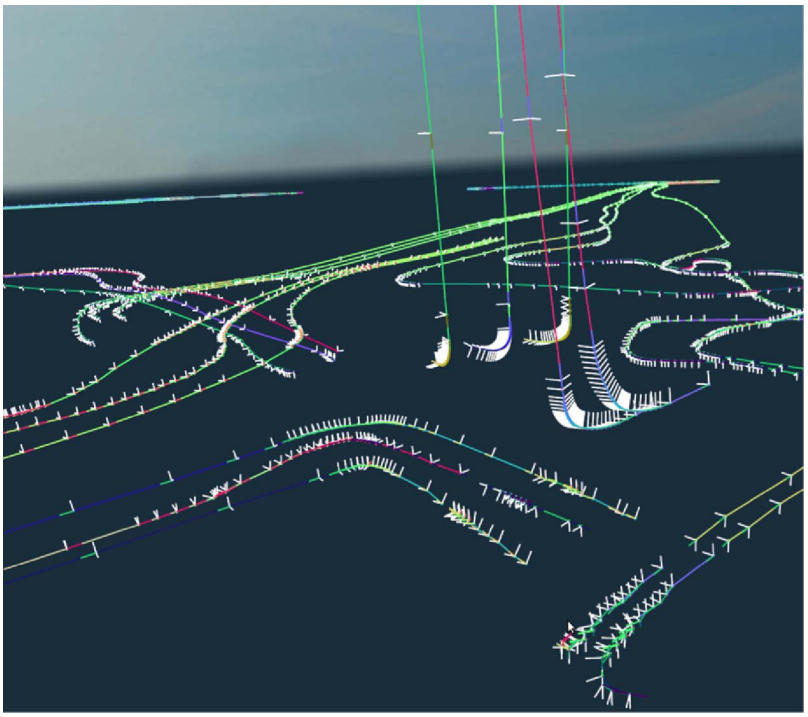

**Figure 3.** Point selection with associated tangent, normal and binormal, along the curve of a tube. High derivative areas require more points for a precise reconstruction.

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 $\{\mathbf{p}_0, \cdots \mathbf{p}_\mathrm{n}\}$. 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 $\{\mathbf{p}_0, \cdots \mathbf{p}_\mathrm{n}\}$ from 2D line to a point $\{\mathbf{p}'_0, \cdots \mathbf{p}'_\mathrm{n}\}$ 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:

$$\Theta = 2\pi \left( \frac{\mathbf{p}_\mathrm{i}}{\mathbf{p}_\mathrm{n}} \right) \tag{1}$$

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 cross section circle of the tube at $\mathbf{P_i}$. This transformation is made through a

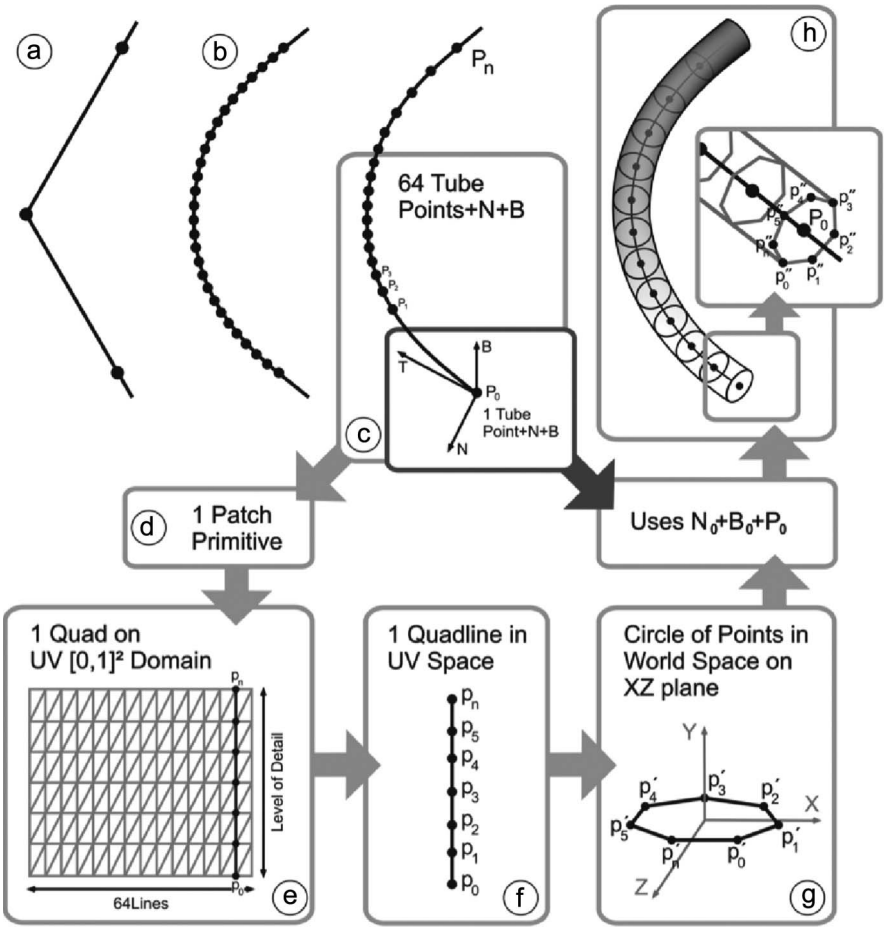


**Figure 4.** (a) A raw point sequence. (b) The refined point sequence after interpolating with Catmull-Rom spline. (c) The selection of relevant points from the sequence and calculation of tangent, normal, and binormal. (d) Every 64 points imply in one patch primitive. (e) The patch primitive is a quad domain with 64 × LOD lines. (f) One 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.

calculation of reference frames along a space curve [Bloomenthal 90], where the final position $\{\mathbf{p}''_0, \ldots, \mathbf{p}''_n\}$ of the point (Figure 4h) in world space is:

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

where N and B stand for the normal and binormal at $\mathbf{P}_i$. Further details are explained in the implementation section. With the algorithm to convert a line to a cross section in world coordinates, there is just a small additional step needed to be able to render a full tube. Because 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 radius of the circle is scaled adaptively, as explained in Section 3.3, but for now we just assume it has a fixed value. In order to use the tessellator as much as possible, our primitive to be transformed in a tube section is a regular grid with 64 lines in one axis, so that every line will be a cross section. As for the axis of the other grid, there is no fixed number of lines, as this number of lines will be regulated by the camera distance, resulting in an on-the-fly, *level-of-detail* (LOD) control. Refer to the next sections for further explanation of these requirements and of the tessellator. The connection between two tube sections (every primitive that represents 64 points) is at a cross section. Moreover, if the number of points along a cross section differs between two adjacent tube sections, there would be cracks in the rendering. However, as the number of points in the cross section depends only on the camera distance, and both adjacent sections connect at a point in the same position, there will be no cracks.

### *3.3. Rendering*

#### 3.3.1. Level of Detail

The tube geometry is rendered with a simple level-of-detail technique. As the number of cross sections along the curve is not supposed to vary, because it could result in a deformation of the tubes path, our LOD technique changes the mesh refinement only on the points along a cross section. The amount of geometry is regulated based on the camera distance; we used a simple quadratic function in our implementation. The number of points is restricted to a minimum of only three, so the cross section will be a triangle instead of a circle.

#### 3.3.2. Reducing the Aliasing

The aliasing effect is expected when rendering a line, because one of its dimensions is many times smaller than the other. Even with a GPU antialiasing 16x turned on, the quality can be undesirable (Figure 7). Our approach to reduce aliasing is rather simple and very effective. For each point $\mathbf{P_i}$ of the point set representing the 3D tube, we calculate the size of the diagonal of a pixel in world coordinates at the location of $\mathbf{P_i}$. If the radius $r$ of the cross section at the location of $\mathbf{P_i}$ is smaller than the diagonal of a pixel, we set $r$ with the

diagonal size of the pixel. This approach combined with the LOD described in the previous section solves the visual problem of aliasing. However, as the radius of the tubes is being changed, the scene will be disproportionately represented; objects near the tubes will seem smaller when viewed from a distance. In our renderer, however, the goal is to always view the tubes, because they have a different semantic than the rest of the scene, therefore this disproportion is natural. Nevertheless, if the tubes are supposed to disappear at a given distance, then an alpha correction should be applied alongside the radius inflation, so there will be no disproportion.

## 4. The GPU Pipeline with Tessellation

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 (see Microsoft's Presentation on Tessellation for further information)[1]. This new pipeline part has two programmable parts and one fixed but configurable part. We show a diagram of this pipeline in Figure 5. The programmable stages are the hull shader and the domain shader, whereas the tessellator is the only configurable one. Another new concept is the lack of predefined 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 represent 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 the amount of subdivision for the tessellated domain to the tessellator. 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 for subdividing the surface without cracks at the junctions with other patches. The tessellator is responsible for creating the vertices within the chosen domain and also for sending the domain shader the coordinates (within the range $[0. . .1]^2$) where the vertices were created. It is the domain shader's responsibility to move the created vertices to the correct place in the world.

[1] http://www.microsoft.com/downloads/en/details.aspx?familyid=2D5BC492-0E5C-4317-8170-E952DCA10D46.

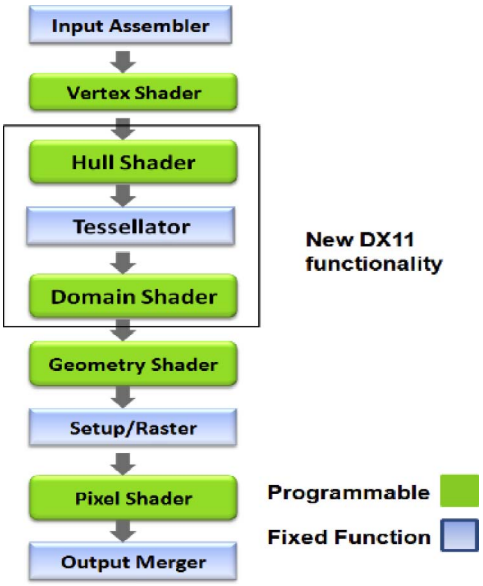


**Figure 5.** The new GPU pipeline with Tessellation.

## 5. Implementation

### *5.1. Preprocessing*

In order to pass the correct normal, bi-normal, and central point information for each tube cross section n, we need to execute an $O(n)$ preprocessing step as explained in Section 3. This information is 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 a maximum tessellation factor. Each line of this axis is transformed to follow the cross section defined by the point sequence that represents the path of the tube. As we show in Figure 7, the other axis tessellation values are flexible for LOD values.

### *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 different positions, normals, and bi-normals to the tessellator for each tessellated quad. But the input assembler accepts only 32 control points per quad. We are able to work around this issue 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 the information needed to correctly position two cross-sections. The following structure snippet shows the information that we need for each control point:

```
  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 number of lines into tube sections. The three remaining parameters are varied adaptively in a straightforward LOD according to either the distance from the camera or the 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.

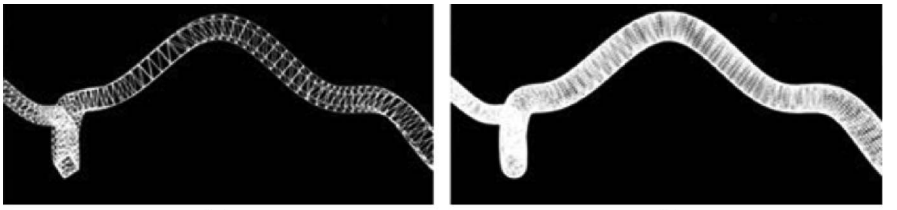


**Figure 6.** (a) Tube with low LOD. (b) Tube with high LOD.

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 ;
}
```

### *5.4. The Main Hull Shader*

A significant 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, and 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 two 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 ($\mathbf{P}_{\text{screen}}$). The depth is obtained dividing the $z$-coordinate of $\mathbf{P}_{\text{screen}}$ by the $w$-coordinate of it:

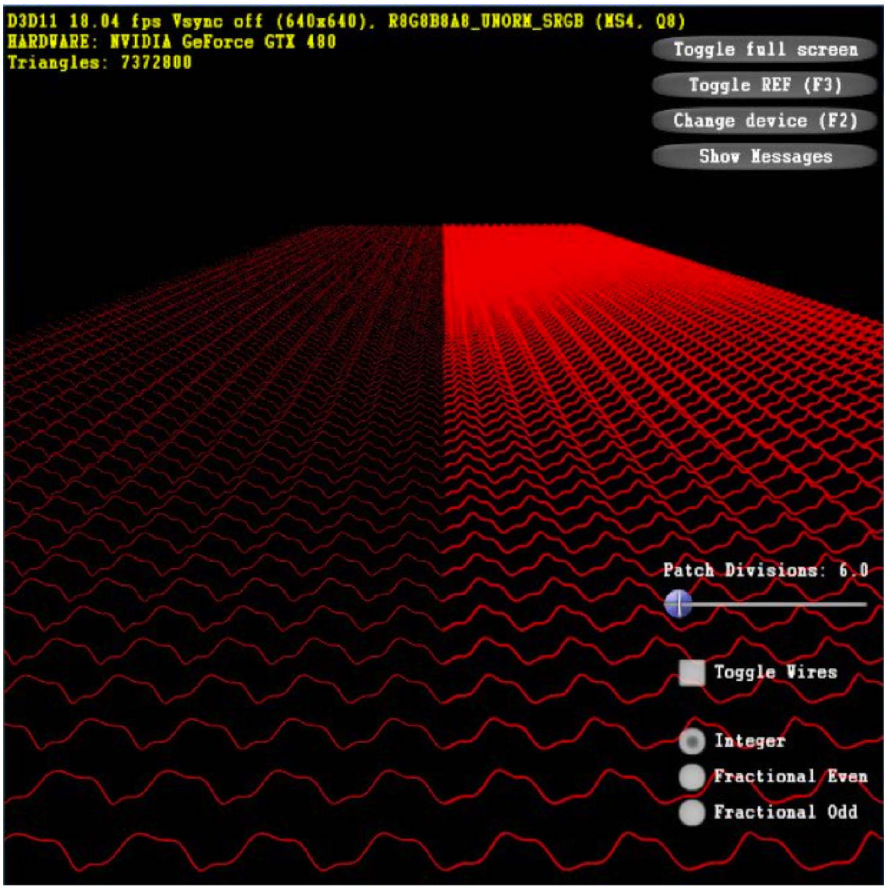


**Figure 7.** The same scene: (a) with 16x GPU antialiasing (AA), (b) without GPU AA but with our radius correction AA.

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

$$\mathbf{P}_{\text{depth}} = \frac{\mathbf{P}_{\text{screen}} z}{\mathbf{P}_{\text{screen}} w} \tag{4}$$

For simplicity, we assume a canvas with an aspect ratio of 1. Let res be the resolution of the canvas. $\mathbf{S}_1 = (0, 0, \mathbf{P}_{\text{depth}}, 1)$ and $\mathbf{S}_2 = (0.5/\text{res}, 0.5/\text{res}, \mathbf{P}_{\text{depth}}, 1)$ represent a pixel width in screen space. Our goal is to find the size of a pixel in world space at the depth $\mathbf{P}_{\text{depth}}$. Multiplying

$\mathbf{S}_1$ and $\mathbf{S}_2$ by the inverse of the 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:

$$\text{PixelSize}_{\text{world}} = \left|\left| \frac{\mathbf{S}_2 * \text{ViewProj}^{-1}}{S_{2w}} - \frac{\mathbf{S}_1 * \text{ViewProj}^{-1}}{S_{1w}} \right|\right| \tag{5}$$

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 ;
      float 4 pScreen = mul (float4 (p[i].vPosition, 1),
          g_mViewProjection) ;
      float pDepth = pScreen.z / pScreen.w;
      float4 S2 = float 4 (0.5 f / res, 0.5 f / res, pDepth, 1) ;
      float4 S1 = float 4 (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 PixelDiagonalRadiusWorld = length (worldPixel2.
            xyz − worldPixel1.xyz) ;
   if (radius <= PixelDiagonalRadiusWorld) radius =
          PixelDiagonalRadiusWorld ;
      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 follows:

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

$$\mathbf{p}'_i = (r\cos\Theta, 0, r\sin\Theta)\,. \tag{7}$$

Now the position ($\mathbf{P}_i$), normal ($\vec{N}$), and binormal ($\vec{B}$) must be fetched from control points data. Because 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 cross section $\mathbf{C}$ of the tube. 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 found easily by the 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 ,
              float 2 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 /2.0 f) ;
                N = p[index].vNormal2 ;
                B = p[index].vBinormal2 ;
                P = p[index].vPosition2 ;
        }
        else
        {
                index = ( int ) ( index /2.0 f ) ;
                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
          ) ;
        float 3 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 ;

        float 3 normal = normalize (worldPos - cylinderPos) ;

        Output.vPosition = mul(float4(worldPos, 1),
            gmViewProjection) ;
        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 6 GB 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 purpose of the test 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 the subdivision of back-faced patches, and tessellating,

according to screen-space edge patch size, helps to avoid subpixel triangles. In Table 2, we show the results comparing the frame rate with and without our technique. The results show that our algorithm makes significant gains over the non-tessellator-based approach. Furthermore, we are capable of accomplishing 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 prohibitive 13.2 GB vertex buffer size to be passed from the CPU to the GPU, whereas our approach uses only 103.7 MB for the same number of triangles, as illustrated in Table 1. Moreover, our antialiasing solution demonstrated to be very effective, with a low frame-rate drop against the test with no aliasing treatment. Figure 8 shows the gain percentage of our algorithm in FPS against a regular approach without antialiasing.

## 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 data (positions, normals, and bi-normals) through the control points, one might use only one control point per quad domain and map a texture with the information of the positions, normals, and bi-normals. The single

| | FPS | | |
|---|---|---|---|
| Triangle (Million) | CPU | GPU AA | GPU no 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 |

**Table 1.** FPS comparison for several numbers of triangles. CPU stands for the implementation of our technique without tessellation, whereas GPU stands for our technique with the aid of tessellation

| 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 |

**Table 2.** CPU-GPU memory bandwidth comparison

control point per quad would call 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 starts to show some improvement in memory consumption when the 1 million triangles mark is passed. Moreover, above the 20 million mark, this technique provides a strong performance boost.

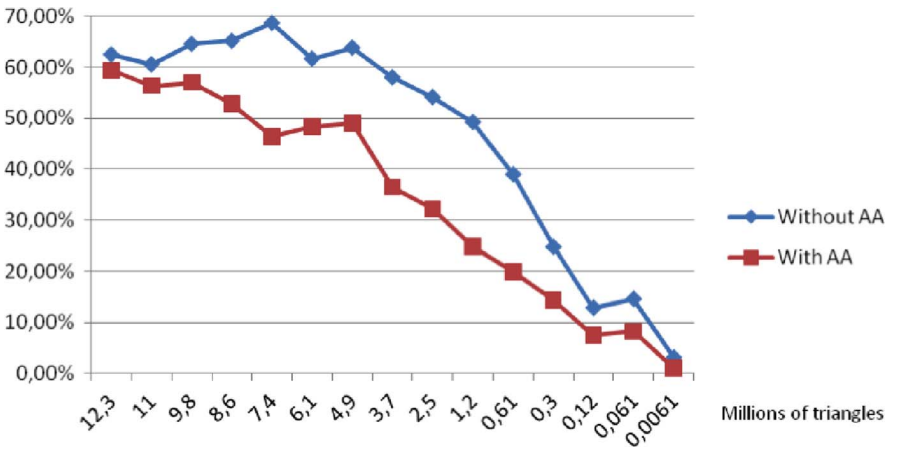


**Figure 8.** Graph showing the percentage gain in FPS of this algorithm with and without AA against a CPU approach without AA.

## 8. Limitations and Drawbacks

### *8.1. Our Use Scenario*

As described in Section 1.1, our application was an oil field with a great number of ducts, wells, and risers. All of them are static lines defined by point sets. We wanted visual quality, without aliasing, and with the tubes visible from any distance. Our technique fit this application well, because we provided good visual quality by having a refined geometry through regulating the LOD according to the camera distance. We also adjusted the radius of the tube so it never disappeared, and the there is no noticeable aliasing. The need for a small amount of pre-processing is a minor drawback that we could not avoid.

### *8.2. Other Use Scenarios*

In a scenario where the point sets are animated through time, our technique may not be suitable. If the animation of the point is defined by a discrete set of key frames, and no blending is required, then our technique can be used with a pre processing of each key frame. Since our pre processing is very small, and the amount of data generated is in the same order of magnitude as the point set itself, our technique might provide a good solution. However, if there is a continuous animation by a time function or interpolation, then our technique cannot be applied.

In most of the scenarios, different from our case, the tubes should be culled with the distance. In such cases, the minimum radius of the tube should not be arbitrary and fixed so that the tube cross section occupies one pixel. Also, the tube's transparency should increase proportionally to the radius correction. The transparency should reach maximum when the tube is supposed to be culled by the distance. If this transparency regulation is not used, there will be a disproportion effect between the tubes and the rest of the geometry nearby, when viewed from far away.

### *8.3. Older Hardware*

The main limitation of our technique is that it is compatible only with DirectX 11/OpenGL 4 video cards. One might think of implementing it through geometry instancing for support of older cards. It works to a degree, but there are issues with the rasterizer performance. LOD control through geometry instancing is not as flexible as with the new pipeline, and if the application starts to create too many subpixel triangles, the rasterizer acts as the main bottleneck of the program due to overshading.

**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

[Bloomenthal 90] J. Bloomenthal. "Calculation of Reference Frames Along a Space Curve." In *Graphics Gems*, edited by Andrew S. Glassner, pp. 567–571. San Diego, CA, USA: Academic Press Professional, Inc., 1990.

[Catmull and Rom 74] E. Catmull and R. Rom. "A Class of Local Interpolating Splines." *Proc. of International Conference on Computer Aided Geometric Design*, (1974), 317–326.

[Kondratieva et al. 05] Polina Kondratieva, Jens Krüger, and Rüdiger Westermann. "The Application of GPU Particle Tracing to Diffusion Tensor Field Visualization." *Proc. IEEE Visualization 2005*, pp. 73–78, 2005.

[Loop et al. 09] Charles Loop, Scott Schaefer, Tianyun Ni, and Ignacio Castaño. "Approximating Subdivision Surfaces with Gregory Patches for Hardware Tessellation." *ACM Trans. Graph.*, 28:5 (2009), 1–9.

[Merhof et al. 06] 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*, (2006), 12:1181–1188.

[Stoll et al. 05] Carsten Stoll, Stefan Gumhold, and Hans-Peter Seidel. "Visualization with Stylized Line Primitives." *IEEE Visualization*, (2005), 695–702.

[Toledo and Levy 04] Rodrigo Toledo and Bruno Levy. "Extending the Graphic Pipeline with New GPU-Accelerated Primitives. *International gOcad Meeting*, 2004. Also presented in Visgraf Seminar 2004, IMPA, Rio de Janeiro, Brazil.

[Valdetaro et al. 10] Alexandre Valdetaro, Gustavo Nunes, Alberto Raposo, Bruno Feijo, and Rodrigo de Toledo. LOD Terrain Rendering by Local Parallel Processing on GPU. *Brazilian Symposium on Games and Digital Entertainment*, (2010), 182–188.

Gustavo Nunes, Dept. of Informatics/PUC-Rio, Rua Marques de Sao Vicente, 225 - Gavea 22451-900, Rio de Janeiro, RJ, Brazil (xnunes@msn.com)

Alexandre Valdetaro, Dept. of Informatics/PUC-Rio, Rua Marques de Sao Vicente, 225 - Gavea 22451-900, Rio de Janeiro, RJ, Brazil (avporto@tecgraf.puc-rio.br)

Alberto Raposo, Dept. of Informatics/PUC-Rio, Rua Marques de Sao Vicente, 225 - Gavea 22451-900, Rio de Janeiro, RJ, Brazil (abraposo@tecgraf.puc-rio.br)

Bruno Feijó, Dept. of Informatics/PUC-Rio, Rua Marques de Sao Vicente, 225 - Gavea 22451-900, Rio de Janeiro, RJ, Brazil (bfeijo@inf.puc-rio.br)

Rodrigo de Toledo, DCC/Inst. Matematica/UFRJ, Caixa Postal 68.530, 21941-590, Rio de Janeiro, RJ, Brazil (rodrigodetoledo@gmail.com)

Submitted June 13, 2011; Recommended October 12, 2011; Accepted January 10, 2012. Edited by David Hart.