Technical Report

Fake Volumetric lines

Abstract

Fake Volumetric lines

This whitepaper and corresponding SDK sample demonstrate a technique to render volumetric lines. The technique is using a way to render quads which are able to fake a volume in whatever orientation it will be. This effect need to perform some computation in the Vertex shader more than in the fragment shader. The demo is coded by using Cg and OpenGL API.

Tristan Lorach (tlorach@nvidia.com)
sdkfeedback@nvidia.com

NVIDIA Corporation
2701 San Tomas Expressway
Santa Clara, CA 95050

February 14, 2005

Table of Contents

Introduction

Rendering simple primitives is part of many scenes in games or professional applications.

In a game, laser beam and specific wireframe objects could be rendered with this while in CAD/DCC applications, these lines could help in highlighting some objects.

Direct3D doesn't have any line primitives while on OpenGL API, line is really part of the API. OpenGL is even able to render more-than-one pixel thick line. The historical reason for that is coming from the fact that CAD systems are great line-primitive customers.

On our NVIDIA chips, lines are processed in different way, depending on the family you're using : on GeForce family, lines won't be anti-aliased and the two edges will be approximated depending upon the orientation of the line; on Quadro family, lines are anti-aliased and edges are conforming to a 90 degrees angle.

However, even though OpenGL is able to render lines, it doesn't perform what we want to do here: no perspective applied to the line and not texturing available.

This effect is not made to replace OpenGL line drawing, but to show how to render a perspective-deformed, thick and glowing line.

Although the glowing is representative of this effect, we could imagine using other rendering styles. But the glowing effect is interesting because it allows enforcing the line without any expensive post-processing (like Gaussian filtering).

Since we are dealing with a fake volume, there is also the question of how to render the line when it is facing you. This problem is addressed by offsetting the clip-space coordinates and by playing with a set of 16 textures, which will slightly change from a profile shape to a facing shape, depending on the line orientation.

The Technique

First steps : send the vertices

The basic idea is to feed the API with a list of quads representing lines. What you want to do is send the starting point A and the endpoint B. But as it is a quad, we will send two starting points (A and A'), and two ending points (B and B').

The interest of such an effect is to ask the GPU to do as much as possible so that the CPU (and you) wouldn't have to care too much of the details. So, instead if computing A, A', B and B' on the CPU, we will simply send two time A and two times B then ask the vertex shader to do the offset computation by itself.

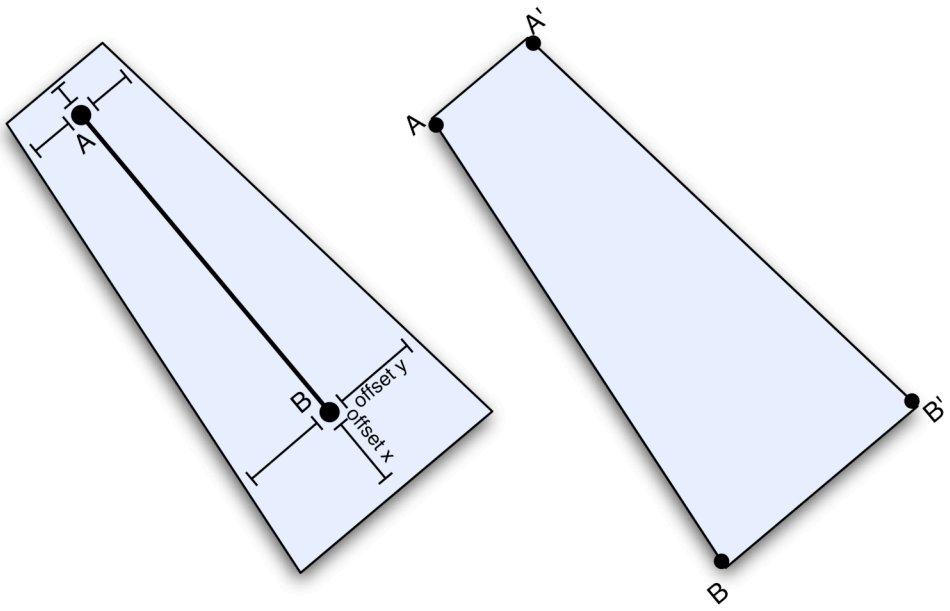


Fig.1

However, even though you want to send A and B, you'll still need to feed the vertex shader with offset and line direction to make the quad look like what we want. We will simply pass these data by using other attributes than vertex coordinates (point size attribute, texture coordinate…).

For a single line, we must draw a quad made of four sets of the following attributes :

```
struct app2vert
 {
      float4 startpos   : POSITION;
      float4 endpos     : PSIZE;
      float4 color      : DIFFUSE;
      float4 param8     : TEXCOORD0;
      float3 param9     : TEXCOORD1;
 };
```

- **Endpos.xyz** is needed to allow the vertex shader to compute the line direction while processing a single vertex of the quad : For the 2 vertex quad *A*, *startpos* will be *A* coordinate and *endpos* will be *B* coordinate.
- **Param8** contains some offsets that we'll talk later.
- **Param9** contains the thickness sizes

A line-thickness must be computed on a plane perpendicular to the viewing direction. Starting from the 2 points A and B of the line, we must find four points by shifting A and B onto a perpendicular plane to the viewing direction. On the other hand we want the line thickness to be changing depending on the perspective.

We will project the line onto the *screen-space* (x,y and z divided by w), then guess what is its direction, compute offsets but we will add these offsets into the *clip-space* (before we divide by w).

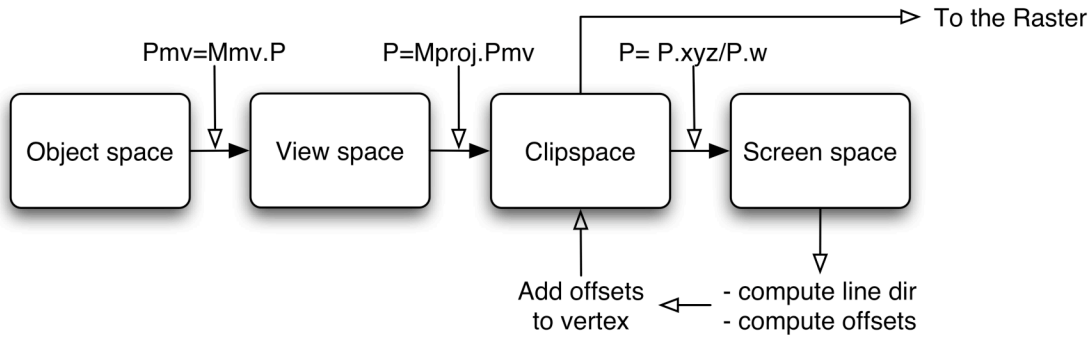


Fig. 2

These first considerations would lead to this first draft of Vertex shader Cg code :

1. Project in *clip-space* the start and end points :

```
posstart = mul(ModelViewProj, IN.startpos);
posend = mul(ModelViewProj, IN.endpos);
```

2. Project these points in the *Screen-space* (we only need x and y):

```
float2 startpos2d = posstart.xy / posstart.w;
float2 endpos2d = posend.xy / posend.w;
```

3. Compute the 2d line direction :

```
float2 linedir2d = normalize(startpos2d - endpos2d);
```

4. Shift vertex for thickness perpendicular to the line direction **in perspective space** :

```
linedir2d = IN.param9.x * linedir2d;
posstart.x = posstart.x + linedir2d.y; // vertical x
posstart.y = posstart.y - linedir2d.x; // vertical y
OUT.hpos = posstart;
```

IN.param9.x (*offset y* in figure 1) contains the offset value that we passed to the vertice as a varying attribute. Each of the four vertices of the quad will get this offset with different signs depending on how we want to shift the vertices. In the figure below, the sign of *thickness* value is relative to vector *u*, for *A* and vetor *u'* for *B*.

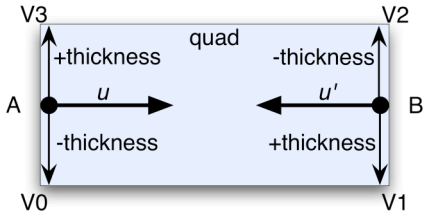


Fig. 3

In the code, you would send the attributes as follow (simplified code, here):

```
cgGLSetParameter3fv(hVarParam1, inToPos.v );
cgGLSetParameter3f(hVarParam9, -inFromSize,…);
cgGLSetParameter3fv(hVarParam0, inFromPos.v);

cgGLSetParameter3fv(hVarParam1, inFromPos.v);
cgGLSetParameter3f(hVarParam9, inToSize,…);
cgGLSetParameter3fv(hVarParam0, inToPos.v);

cgGLSetParameter3fv(hVarParam1, inFromPos.v);
cgGLSetParameter3f(hVarParam9, -inToSize,…);
cgGLSetParameter3fv(hVarParam0, inToPos.v);

cgGLSetParameter3fv(hVarParam1, inToPos.v);
cgGLSetParameter3f(hVarParam9, inFromSize,…);
cgGLSetParameter3fv(hVarParam0, inFromPos.v);
```

Fake volumetric rendering of the line

The previous code is fine for drawing a line in 2d. But what happens after we projected the line on screen space, when the line is almost facing us, for example when it is almost aligned along the z axis ? As the schematic is showing it, we would get a kind of near-zero length line.

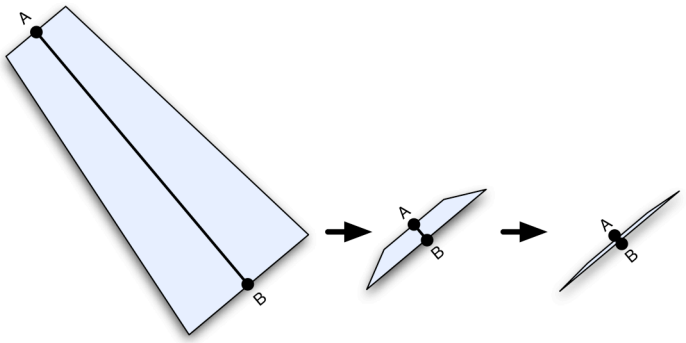


Fig. 2

When A and B are projected in almost the same 2d location, we want to have a kind of square sprite representing this the volumetric point.

For this, we need to shift the vertices along the 2d projected line according to a specific offset. In the Cg code of the previous section, just before step #4, we would add this :

```
posstart.xy = ((texcoef * IN.param8.x) * linedir2d.xy) + posstart.xy;
```

- texcoef → 1 : when the line turns to be parallel to our view direction, (projected points A and B would be close to each other)
- texcoef → 0 : when the line turns to a be perpendicular to our view direction

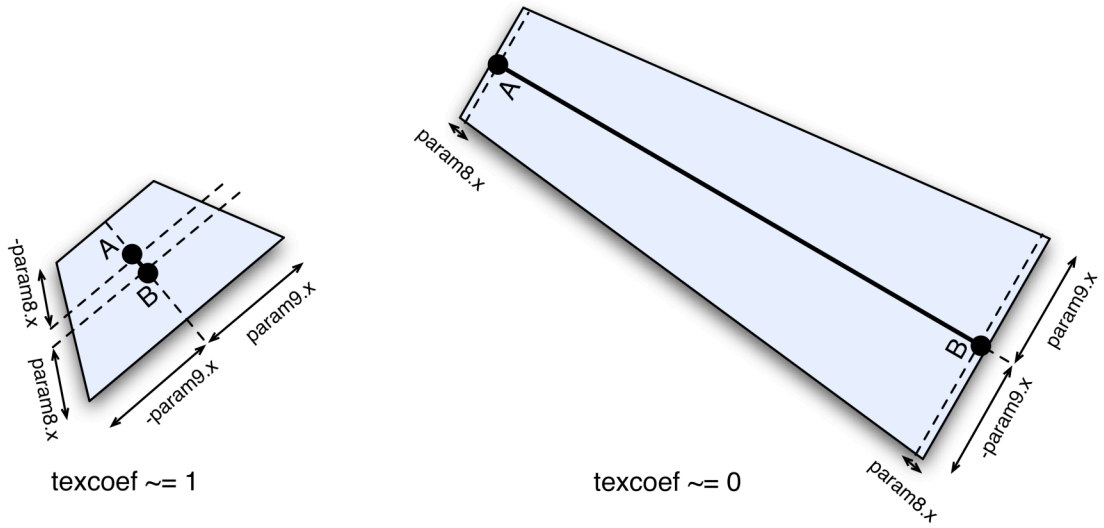


Fig. 3

To compute *texcoef*, we must guess how the 3D line is oriented with respect to the eye point. For this purpose we must first transform the 2 points of the line in *ModelView* space :

```
posstart = mul(ModelViewProj, IN.startpos);
posend = mul(ModelViewProj, IN.endpos);
```

Then we will take the center of the line to compute the vector from the line to the eye

```
float3      middlepoint = normalize((posstart.xyz + posend.xyz)/2.0);
```

Now we need to compute the dot product between the *line unit vector* and the *unit vector from the eye to the middle of the line* :

```
float3      lineoffset = posend.xyz - posstart.xyz;
float3      linedir = normalize(lineoffset);
float       texcoef = abs(dot(linedir, middlepoint));
```

texcoef will reach 0 when the line is perpendicular to the eye direction and will be 1 when it is aligned with it.

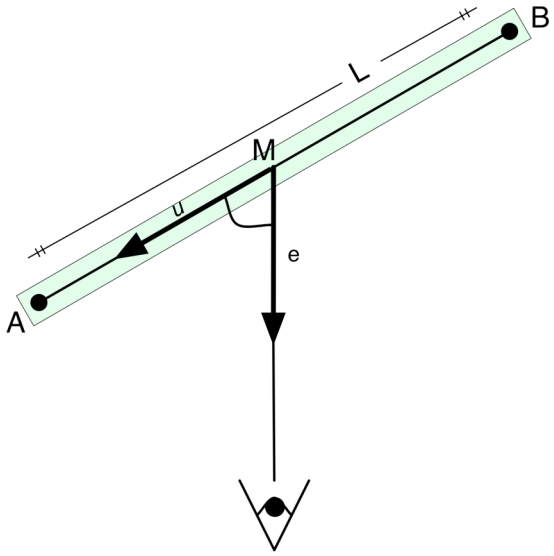


Fig. 4

If you test the sample with this *texcoef*, you'll se that the result isn't so interesting: something is wrong with the appearance of the line, essentially because of the textures. But before getting deeper into the trick to solve this, let's see how to map textures on the quad.

Fake Orientation of the line

The texture which is used to render the line is a 2d texture made of 4x4 tiles. Each tile will be accessed depending on the orientation factor that we computed into *texcoef*.

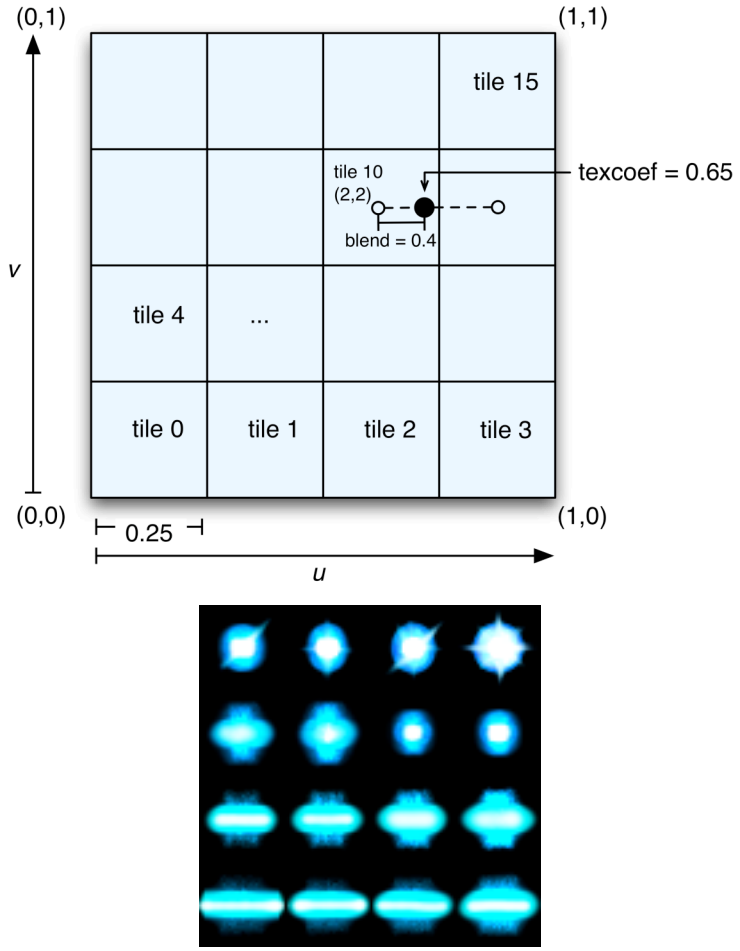


Fig. 4

- tile #0: texture used when the line is perpendicular to the eye direction.
- Tile #15: texture used when the line is 'facing' the eye direction
- Tiles #1 to #14 : slight transitions.

To access properly the textures, we can use *modf()* function, returning the fractional and the integer part of a value : *fp = modf(val, ip)*.

The fractional part will be used to access 4 tiles in the *u* axis, while the integer part will be used to access the 4 levels in the *v* axis. For this purpose we will scale the *texcoef* range [0,1] to [0,4] and isolate integer from fractional parts :

```
u_n = modf(texcoef * 4.0, v_m4);
```

u_n is ranging from [0,1] while *v_m4* is an integer from [0,3]. Now let's do the same operation of getting the integer value for the *u* axis :

```
bf = modf(u_n * 4.0, u_m4);
```

u_m4 and *v_m4* are now both integers values in [0,3]. *bf* is a fractional part that we are going to use in order to blend 2 textures parts for a smooth transition. The last thing to do is to normalize *u_m4* and *v_m4* and add an offset depending on which vertices of the quad we addressed (in pseudo-Cg-code):

```
{u,v} = ({u_m4,v_m4} / 4.0) + (IN.param8).zw;
```

To blend two stages of the texture, we must compute another set of texture coordinates by using *texcoef+(1/16)* instead of *texcoef*. Then passing *bf* value to the fragment will allow us to do the blending at the fragment level with a simple *lerp()*.

Because we're doing this second computation at *texcoef+1/16*, we must cap the to 15/16 to avoid going outside of the texture. Thus we must replace a line above like this:

```
u_n = modf(min(15.0/16.0, texcoef[+1/16])* 4.0, v_m4);
```

Note: It is possible to get the **same result by using a 3D texture**. 3D texture will do the bilinear filtering in z for you, instead of doing it by hand in the vertex shader.

Tweak the orientation

As I said before the way that *texcoef* is changing from 0 to 1 is used both to change the length of the line and to change the texture offset.

When trying the demo with the simple computation of *texcoef* mentioned above, the transitions of the texture doesn't seem to be correct.

To correct this problem I propose a simple correction of *texcoef* as follow:

- When the line length is equal to its thickness, don't change texcoef: the transition is consistent (L (length)=W (width), black line in the figure below)
- When the line length turns to be greater that its thickness, then let's try to postpone the texture transition until *texcoef* almost reaches the 1 value.

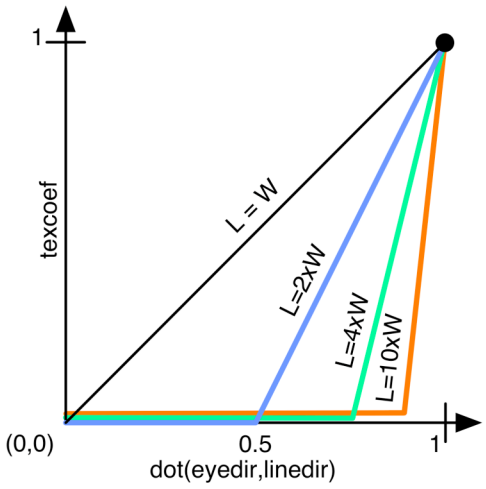


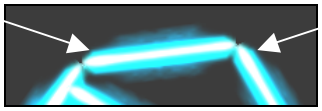
Fig. 5

```
texcoef = max(((texcoef - 1)*(line_length / line_thickness)) + 1, 0);
```

This trick will help the to get a more consistent texture transition, depending on the line shape.

Known issues & limitations

When used with a glowing texture, this effect seems to not be so well fitting for series of lines connected to each other (poly-line). The reason is because each line creates some discontinuity because of the texture shape.

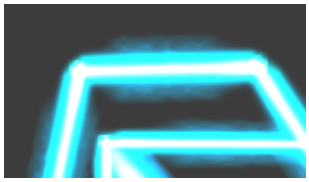


As we saw before, *texcoef* is the value that avoids the lines to overlap. Further work could be done to find a good tradeoff between the issues we worked around thanks to *texcoef* and the issue we can see when lines are badly connected to each other.

For example, shifting *texcoef* to 0.2 in the vertex shader code:

```
posstart.xy = (((texcoef + 0.2) * IN.param8.x) * linedir2d.xy) +
posstart.xy;
```

will make the lines to overlap.



However this value will depend on the texture we used to render the lines.

Conclusion

The fake volume-line can be an efficient way to render perspective-corrected lines in various applications with very few polygons. The glowing effect can also bring you a cheap approximation of a Gaussian filter post-processing.

Notice

Trademarks

Copyright

NVIDIA Corporation
2701 San Tomas Expressway
Santa Clara, CA 95050
www.nvidia.com