White Paper

Solid Wireframe

February 2007
WP-03014-001_v01

Document Change History

Version	Date	Responsible	Reason for Change
_v01	February 20, 2007	SG, TS	Initial release

Go to sdkfeedback@nvidia.com to provide feedback on SolidWireframe.

SolidWireframe

Abstract

The **`SolidWireframe`** technique efficiently draws a high quality, anti-aliased wireframe representation of a mesh (see Figure 1). Instead of using conventional line primitives, the technique draws triangle primitives and displays only the fragments on the edges of the triangles. Interior fragments are transparent.

The new Geometry Shader available on the GeForce 8800 with DirectX10 can access all three vertex positions of a triangle, unlike previous Vertex Shaders, which process only one vertex at a time. Thus the Geometry Shader can compute results based on the edge information of a triangle. The fragment shader can then appropriately color the fragments to draw only the edges of the triangle. For hidden line removal, the depth test works with no error because the technique uses exactly the same filled faces as the depth pass.

The effect is achieved with one pass requiring only the position attribute, thus making it straight forward to insert in a graphics engine.

Samuel Gateau
NVIDIA Corporation

Figure 1. Mesh Rendered with the Solid Wireframe Technique

Motivation

The wireframe representation of a mesh is widely used when modeling in DCC, CAD and 3D tools as it is the best way to visualize the layout of faces.

Previously, under DirectX this could be achieved using the line features of the API, drawing the mesh with the fill mode set to **`D3DFILL_WIREFRAME`**. But the line rendering has several limitations:

- The thickness of the line is always 1 pixel and cannot be modified
- The line is not anti-aliased
- The lines z-fight with triangles rendered using the same vertex positions.

To remove hidden lines, the classic procedure uses two passes. In the first pass, the filled faces are rendered with color writes disabled, but z writes enabled. This fills the z-buffer with depth data for the object's triangles. In the second pass, the wireframe is drawn with depth test enabled. The primitives used to draw the two passes are different—filled triangles verses lines—thus the depth rasterized from the two passes may be different which causes z fighting and incorrect occlusion of some pixels of the wireframe. To compensate, using the depth bias offset feature, the depth of the fragment rendered by the first pass can be slightly offset back to ensure it is behind the depth of the fragment drawn by the wireframe. The displayed result is good but this still causes issues because the mesh depth is not strictly correct.

In this sample, we implement, under DirectX10, a new technique which solves all the issues of the classic approach. The method was presented in the SIGGRAPH 2006 sketch *"Single-Pass Wireframe Rendering"* [1].

The only drawback of this method is that the outline edges of the mesh look thinner and are not anti-aliased on the outer side (because only one side of the edge is rendered).

The performance of the technique is extremely good compared to the classic line rendering approach.

Introducing this effect in an existing graphics engine is easy and straight forward as it is a single pass and requires only the position attributes of the mesh.

The technique can be derived and enhanced to draw different patterns along the edges or inside the triangles such as stippled lines; or only on specific edges for different kinds of face outlining (quad, polygon).

How It Works

This new technique displays the wireframe of a mesh by drawing filled triangles, coloring opaque only the fragments close to the edges of the triangles; hence its name **`SolidWireframe`**. The main steps are:

1. Apply the standard model-to-projection transform to the input vertices of the mesh

2. Provide each fragment the geometric information from the triangle to permit computation of the fragment's distance to each edge in viewport space.
3. Compute the shortest distance to the edges of the triangle in viewport space for each fragment of the triangle.
4. Use that distance to draw the fragment only if it is close enough to the edge to represent the line.

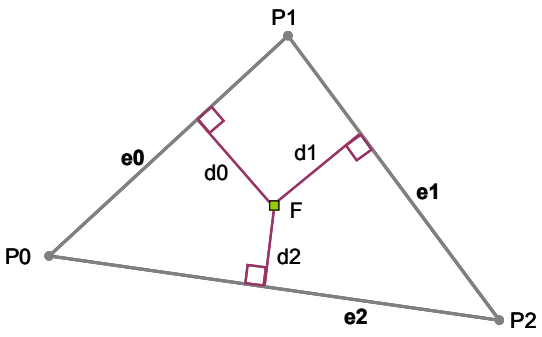
Step 1 is trivial as it uses a standard vertex shader to transform the vertex positions of the mesh from model space to projection space, ready for rasterization. Note that mesh animation could be done naturally here.

Step 2, 3 and 4 are detailed in the following paragraphs; they rely on the geometry and pixel shaders.

Using filled triangles to draw the wireframe produces exactly the same result for the depth value as the standard depth pass. The depth test runs accurately without any z fighting issues.

Computing the Distance

Steps 2 and 3 of the technique compute for a fragment its distances to the three edges of the triangle projected in viewport space.



F	the fragment position in viewport space
P0, P1 P2	the projected point in viewport space of the vertices of the triangle
e0, **e1**, **e2**	the projected point in viewport space of the vertices of the triangle
d0, d1, d2	the distance expressed in pixels between F and each edges

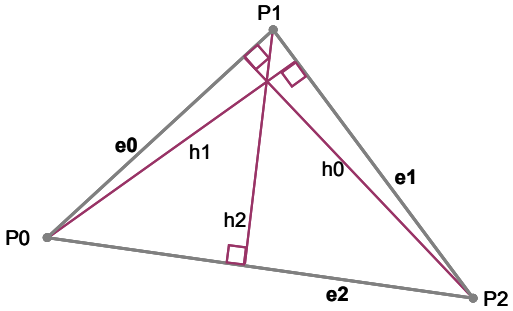
Figure 2. Distance From a Fragment to the Edges

The General Case

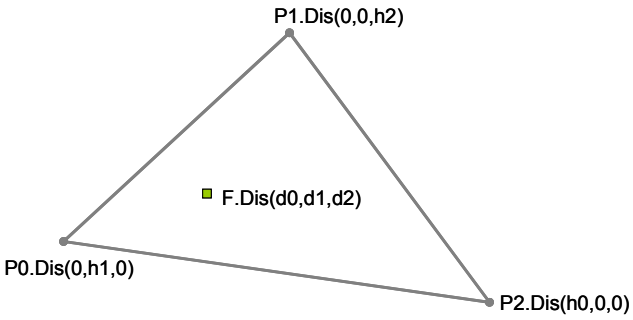
The algorithm is straightforward:

In the geometry shader, input the triangle:

- Project the three input vertex positions of the triangle to **`viewport space.`**
- Compute the three heights of the triangle in **`viewport space`**. These are the minimum distances between each vertex and its opposite edge.



- Output the same triangle as input but adding one more **`float3`** output attribute (**`Dis`**), the **`3`** distances between the vertex and the **`3`** edges:
  - **`P0`** would output its distances to **`e0, e1`** and **`e2`** which is (**`0, h1,0`**).



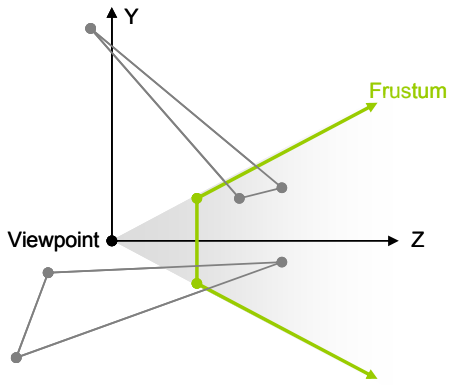
- **`Dis`** is specified to be interpolated with **no perspective correction**.
- At the fragment level, after interpolation, we get exactly the expected distances of the fragment to each edge in **viewport space.**

In the fragment shader

- Input the interpolated three distances to each edge. Get the minimum as the shortest distance between the fragment and the edges of the triangle.
- Alter the output transparency of the fragment as a function of this minimum distance to edge.

Issues

It is not always possible to project a vertex position into viewport space in the case where the vertex is very close to, or behind, the viewpoint. In this case, the resulting projected point makes it irrelevant to compute a corresponding height in viewport space and the interpolation leads to incorrect values. This occurs only when a triangle has at least one vertex visible, and at least one close to or behind the depth of the viewpoint. For a vertex **`P`** expressed in the view frame that means, **`P.z ≤ 0`**.



This situation can be avoided by never having any visible triangles with vertices too far behind the viewpoint. If the average size of the triangles is smaller than the near plane, then you would never see the issue.

However, in a professional application supporting arbitrary meshes, you cannot assume that this will never happen. So we need to find a cleaner approach for these cases.

The Tricky Case

To solve this issue, the approach is to not compute the three heights of the projected point because that is impossible, but rather to pass to the fragments the geometric elements of the triangle required to compute the distances. Let's first describe the computation done in the fragment shader, to then deduce what the geometry shader needs to do.

Fragment Shader

In the fragment shader, to compute its distances to the edges, we need to have the definitions of the edges as lines in viewport space. Although the points cannot be defined in the viewport space, the visible edges can. A line is defined by a point and a direction (Figure 3).

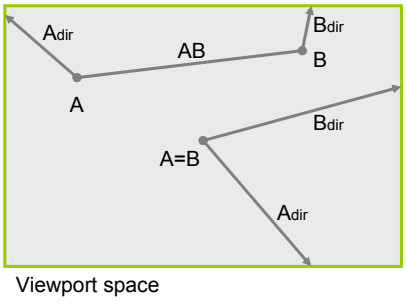


Figure 3. Two Triangles Projected in Viewpoint Space

Depending on how the triangle is projected:

- There are one or two vertices visible and thus well-defined in viewport space, these are named **A** and **B**. When only one vertex is visible, **A=B**.
- There are two or three edges visible, whose direction can be defined in viewport space:
  - Two vectors named **Adir** and **Bdir** are issued from **A** and **B** to the invisible vertices (or vertex) outside the viewport.
  - **AB from A to B**, in the case where **A≠B**.

The fragment, knowing **(A,Adir)** and **(B,Bdir)** can then compute its distances to each of the visible edges by doing a simple computation (assuming the vectors have been normalized).

For example:

```
distance from F to line (A,Adir) = sqrt( dot(AF,AF) - dot(Adir, AF))
```

Depending on the case, two or three distances need to be computed. The minimum distance is taken, as in the standard case.

The choice of case and the line elements needed at the fragment level are defined per triangle. So, the branching depending on the case is coherent for all the fragments rasterized in a triangle thus very efficient.

Geometry Shader

The geometry shader detects the case of triangle projection, and appropriately computes the two line definitions **(A,Adir)** and **(B,Bdir)** in viewport space.

Determining which case we are dealing with is based on the depth of each vertex. A binary combination of the three independent tests, one for each vertex, produce an index value in the range **[0,7].** The different cases are Table 1:

Table 1. Triangle Projection Cases

P0.z > 0	True				False			
P1.z > 0	True		False		True		False	
P2.z > 0	True	False	True	False	True	False	True	False
Case value	0	1	2	3	4	5	6	7
$A_{/p}$	Easy standard case	P0	P0	P0	P1	P1	P2	Triangle not visible
$B_{/p}$		P1	P2	P0	P2	P1	P2	
$A'_{/p}$		P2	P1	P1	P0	P0	P0	
$B'_{/p}$		P2	P1	P2	P0	P2	P1	

The table is implemented in the shader as a constant array of structures. For each case, the structure contains indices to address from the three input vertices.

Four points expressed in **projection space (/p)** are defined as:

- Points **$A_{/p}$** & **$B_{/p}$**
- Points **$A'_{/p}$** & **$B'_{/p}$**, represent the non-projectable vertices (or vertex) used to compute the direction vectors **Adir** & **Bdir**.

The computation of the line direction in viewport space **(/v),** for **Adir** is:

```
Adir = normalize( A/v - (A/p + A'A/p) /v )
```

Bdir is computed similarly using **B** and **B'**.

The definitions of the two lines are passed from the geometry shader to the fragment shader through four **float2** attributes. These values are constant over the triangle.

Two in One

As just shown, there are two algorithms used to compute the minimum distance of a fragment to the edges of the triangle.

Most of the triangles are entirely visible and require only the simple case. This case is the most efficient because most computations are done in the geometry shader, per triangle. In the rare case, the triangle will use the second more complex path which has more work done in the pixel shader.

All the fragments rasterized will have coherent branching because the case is per triangle, thus remaining extremely efficient in all cases.

Both paths use the same attributes to communicate different values from the geometry shader to the fragment shader

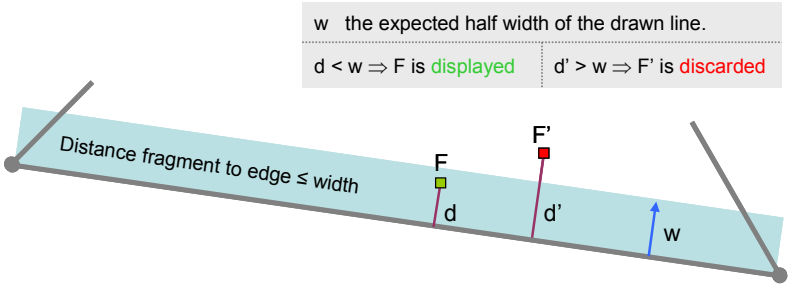
- In first path, three float distances are interpolated with no perspective correction.
- In second path, we pass four **`float2`**s which are constant over the triangle.

In the sample, we use two **`float4`**s with no perspective correction interpolation for both cases.

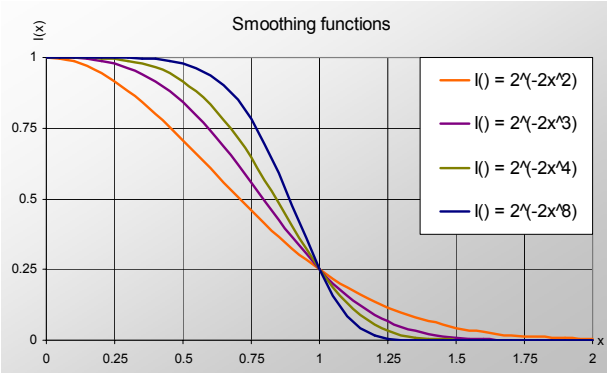
Drawing the edge

At the fragment level, the shortest distance in screen space between the fragment and the edges of the projected triangle is available, computed as explained above.

The shortest distance **`d`** of the three is compared to the expected thickness of the line. If the distance is smaller than the line width then the fragment is rendered, otherwise it is discarded.

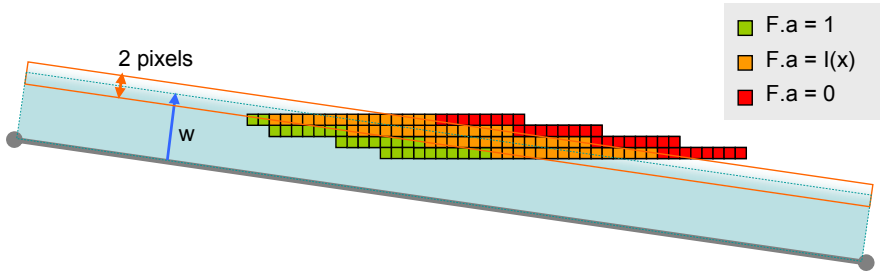


To anti-alias the border of the line, we use the distance **`d`** as input of a smoothing function **`I(x)`** which produces an intensity value used to fade out smoothly the alpha of the output fragment.



A standard smoothing function used is **`I(x) = 2^(-2 x^2).`** As shown in the graph, the intensity function is used for **`x`** in the range **`[0,2].`** The distance **`d`** is mapped according to the expected thickness **`w`** of the line so **`I(x)`** is applied on one pixel around the line border.

Outside this range but inside the line, the fragment is opaque; outside the line, the fragment is discarded.



Running the Sample

To run the sample, launch the **`SolidWireframe.exe`** application from the **`bin`** folder of the SDK (Figure 4).

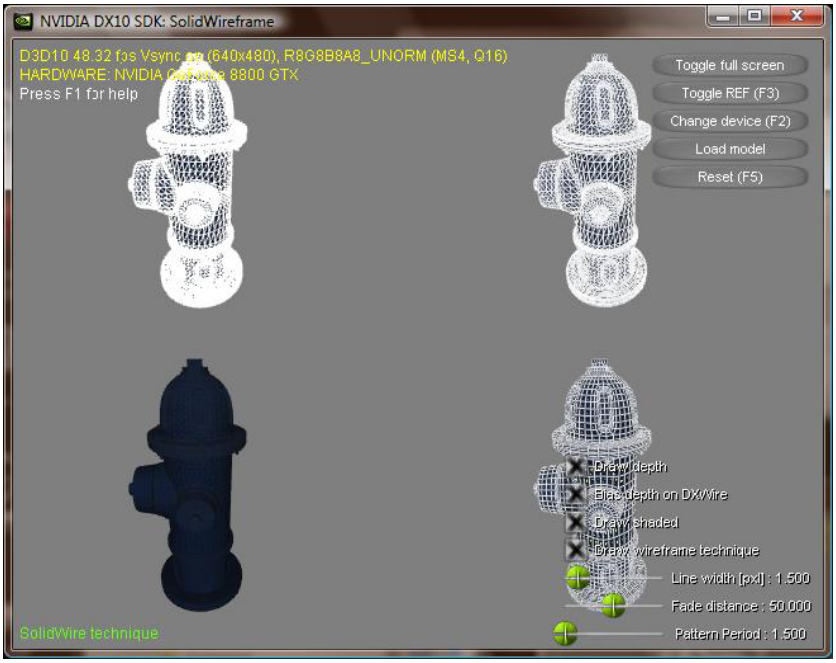


Figure 4. `SolidWireframe.exe` at Launch

Figure 4 displays a mosaic of four views of four different techniques. The first view displays, for comparison, the classic wireframe rendering supported natively by the DirectX API. The others views display the **`SolidWireframe`** technique. There are three variations in the final fragment shader.

- **`SolidWire`** is the exact implementation of the exposed technique.
- **`SolidWireFade`** fades the intensity of the wireframe pixel depending on the depth of the fragment as suggested in [1].

- **`SolidWirePattern`** demonstrates a real application use case where only quads are displayed and the diagonal is displayed with a different dotted line pattern. The test to know on which edge to apply the pattern is simply based on the index of the edge in the primitive. For the model used it is always the first edge defined (Figure 5).

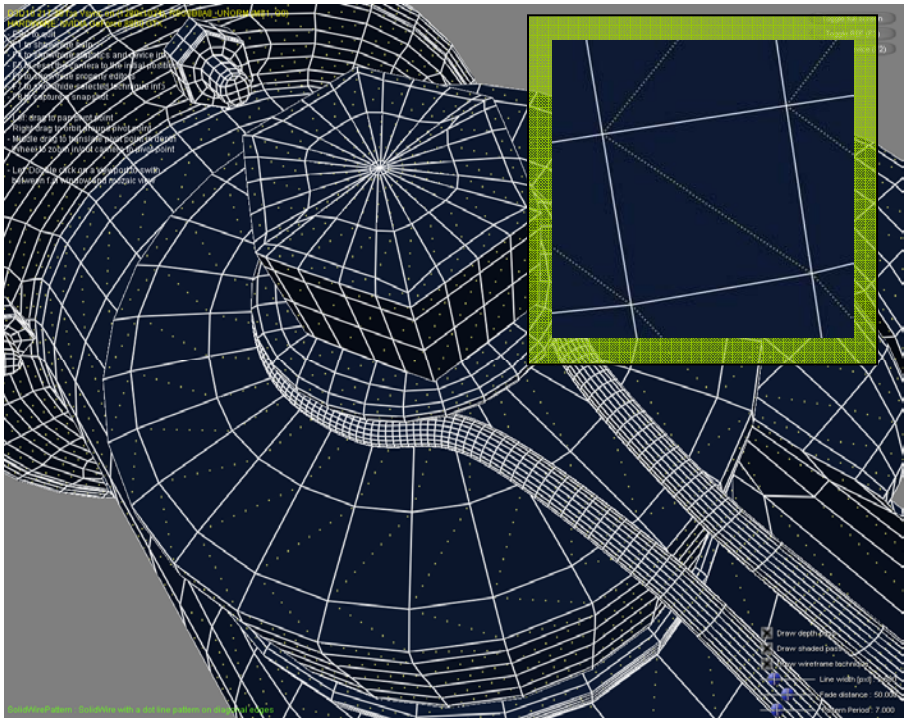


Figure 5. Third Technique Representing the Dotted Pattern on the First Edge of Each Primitive

All techniques include two optional passes; a depth pass and a flat-fill shading pass.

To interact with the **`SolidWireframe`** sample use the following key:

- Use the mouse; click-and-drag to manipulate the camera.
- Press **`F1`** for help.
- Double click on one viewport to select it and make it fill the window; double click again to go back to the four-viewport mosaic.

Performance

This technique is much faster (by a factor of three) than the classic technique on consumer cards.

Integration

The **`SolidWireframe`** technique is easy and straightforward to integrate into an existing graphics engine.

It can directly replace classic wireframe methods. No special pre-processing of the mesh data is required, so it can be applied to any mesh, without modification. It is only one pass with a specific geometry shader and pixel shader. The only needed attributes of the mesh are the vertex positions. The result is higher quality and permits more flexible rendering of the wireframe.

As shown in the sample, the technique can be enhanced to achieve more sophisticated patterns needed for real applications.

One limitation of the technique remains; it displays only half the wireframe line on the silhouette of the mesh. This could be solved by isolating the silhouette edges of the mesh in projection space and then using the geometry shader to draw extra triangle fins. For an example, see the Fur effect in this SDK. This will be probably a starting point for another sample…

References

1. *Single-Pass Wireframe Rendering* by Andreas Bærentzen, Steen Lund Nielsen, Mikkel Gjael, Bent D. Larsen & Niels Jaergen Christensen, Siggraph 2006

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