Shader-Based Antialiased Dashed Stroked Polylines

Nicolas Rougier

Shader-Based Antialiased Dashed Stroked Polylines

Nicolas P. Rougier
INRIA, Université de Bordeaux, IMN UMR CNRS 5293, Labri, UMR 5800 CNRS
Bordeaux, France

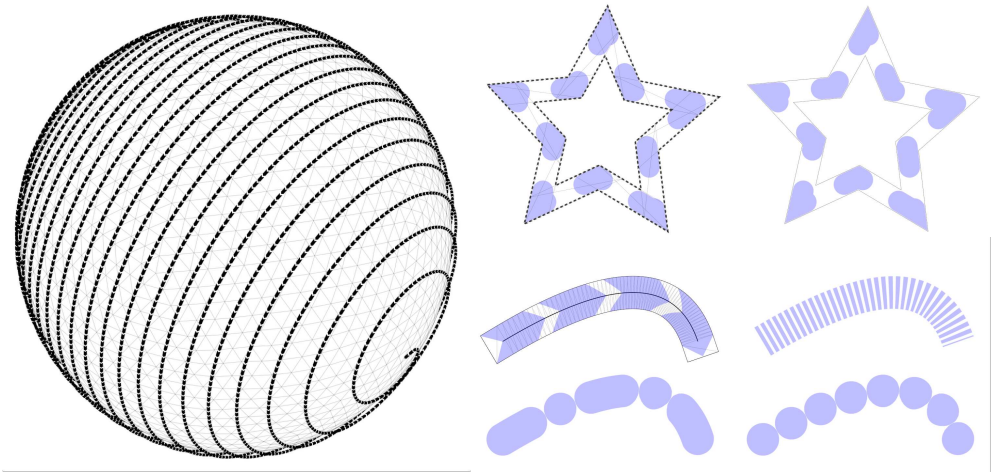

Figure 1. These dashed stroked paths are computed in the fragment shader and do not require extra tessellation. This allows changing line width, line joins, dash pattern and dash phase at no extra cost.

Abstract

Dashed stroked paths are a widely-used feature found in the vast majority of vector drawing software and libraries. They allow, for example, highlighting a given path such as the current selection in drawing software or distinguishing curves in the case of a scientific plotting package. This paper introduces a shader-based method for rendering arbitrary dash patterns along any continuous polyline (smooth or broken). The proposed method does not tessellate individual dash patterns and allows for fast and nearly accurate rendering of any user-defined dash pattern and caps. Benchmarks indicates a slowdown ratio between 1.1 and 2.1 with an increased memory consumption between 3 and 6. Furthermore, the method can be used for solid thick polylines with correct caps and joins with *only* a slowdown of factor 1.1.

1. Introduction

During the past decade, a lot of progress has been made in GPU-accelerated resolution-independent 2D graphics (a.k.a. vector graphics) and methods have been provided for antialiased lines ([Chan and Durand 2005]), Bézier curves, glyph rendering, vector and more recently, generic path rendering ([Kilgard and Bolz 2012]). However, to our best knowledge, few concerns have been given to dashed strokes even though it is a widely-used feature found in vector drawing software and libraries. Dashed strokes were also present in the first OpenGL API specifications through the line stipple functionality was been deprecated. Several alternatives have been provided such as sketchy strokes ([Markosian et al. 1997]), stylized lines ([Bénard et al. 2010; Cole and Finkelstein 2010]) or approximated dashes, but only the path rendering approach seems to offer correct dashing according to the definition of the standard vector graphic format (SVG 1.1). Implementation details have been given sparingly, but [Kilgard 2011] seems to indicate that dashed strokes are first converted into segment sequences (6 triangles / dash), which is usually done in most vector libraries. Consequently, this approach requires a pre-processing stage on the CPU each time a new dash pattern is used (if the dash period is different) or if the dash phase changes. The resulting number of triangles is linearly correlated with the number of visible dashes. Even if the number of triangles remains relatively low compared to the capability of modern graphic cards, the pre-processing stage is cumbersome if one wants to animate the dash pattern as is done in most drawing software where the active selection is animated in order to make it salient.

This paper introduces a new method for rendering dash stroked antialiased polylines using the GPU, a very light pre-processing stage and without extra tessellation. The specific contributions are:

- A dash atlas for efficient texture-based storage of dash patterns
- A shader-based rendering algorithm for dash patterns along a polyline

Since a solid line is a singular case of dashing, the method is valid for rendering thick solid antialiased polylines with correct caps and joins without the need to tessellate the joins.

2. Dashes

A dash pattern is defined as a cyclic sequence of successive *on* (dash) and *off* (gap) segments of variable lengths as illustrated in figure 2. The **period** ω of a dash pattern is defined as the sum of all the *on* and *off* segments. The **dash phase** ϕ corresponds to the length by which the dash is shifted at the start. *On* segments have **caps** at their start and end, overlapping the *off* segment if necessary. If the *off* segment is not large

enough, start and end caps may overlap (the user has to take care of this when defining a dash pattern).

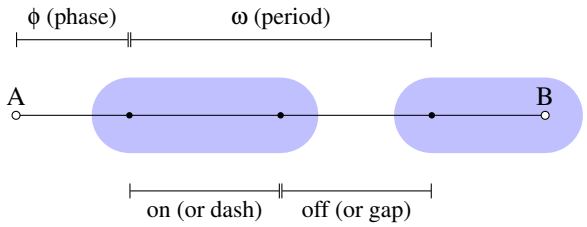

Figure 2. A thick dashed line between point A and B, with round caps (light blue areas) and a non-null dash phase.

3. Anti-aliased solid lines

Before diving in dash lines rendering, we need to expose the rendering of antialiased thick solid lines since the dash rendering techniques relies on it. Chan and Durand introduced an antialiasing technique for lines [Chan and Durand 2005]. Results are hardware-independent and ensures consistent line antialiasing across different GPUs while keeping implementations both fast and easy. Since then, several variants have been proposed but the main idea remains the same. We use a slightly different technique that differs mainly in the way line is parameterized. Let us consider a thick line between a point A and a point B with thickness w and a desired filter radius of size r as illustrated in figure 3.

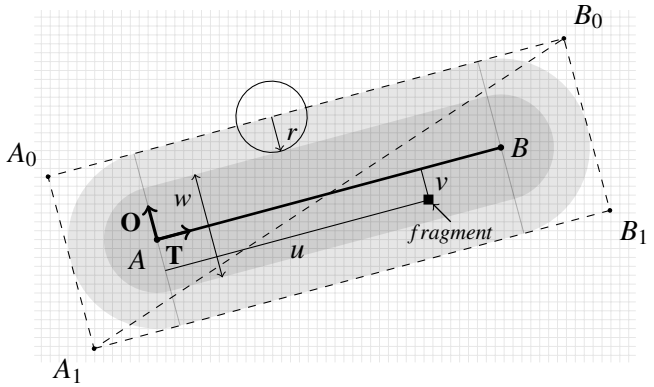

Figure 3. A thick line between A and B with round caps, thickness w and filter radius r. Using $d = ceil(w+2.5r)$, the domain of the (u,v) parameterization is given by $-d \leq u \leq \|\mathbf{AB}\|+d$ and $-d \leq v \leq +d$.

3.1. Body

The line is tessellated as two triangles (A_0, A_1, B_0) and (A_1, B_1, B_0) with texture coordinates (u, v) that represent the point coordinates in the Cartesian coordinate system $\{A, \mathbf{T}, \mathbf{O}\}$. The hardware rasterizer must generate all the fragments associated with a wide line. [Chan and Durand 2005] proposed to use an actual thickness equal to $ceil(w+2r)\sqrt{2}$ that may be excessively large; instead, we use $ceil(w+2.5r)$. When line thickness is below 1 pixel, the actual thickness is kept to 1 pixel while the alpha component of the color is decreased accordingly to simulate thickness below 1. The distance of any fragment to the actual line is given by the absolute value of the v texture coordinate.

3.2. Caps

Knowing the segment length $l = \|\mathbf{AB}\|$ and using the (u, v) parameterization, it is easy to find what part the fragment belongs to:

- if $u < 0$, the fragment is situated in the start cap area
- if $u > l$, the fragment is situated in the end cap area
- if $0 \leq u \leq l$, the fragment is situated in the body area

Using $dx = |min(u, u-l)|, dy = |v|$ when $u < 0$ or $u > l$, the shape of the cap can be controlled using the different formulas presented in table 1.

None		$d = +\infty$ (no cap)
Butt		$d = \max(dx + w/2 - 2r, dy)$
Square		$d = \max(dx, dy)$
Round		$d = \sqrt{dx^2 + dy^2}$
Triangle out		$d = dx + dy$
Triangle in		$d = \max(dy, w/2 - r + dx - dy)$

Table 1. Line caps and their corresponding formulas.

The computed distance d is then used to decide if a fragment belongs to the segment ($d \leq \frac{w}{2}$) or not ($d > \frac{w}{2}$) while for the actual segment body, we use the distance $d = |v|$ as explained earlier.

4. Dash atlas

To draw dashes inside the segment body, we need to know if a given fragment (u, v) belongs to a dash cap area (start or end), to the dash body or is in a gap area. Then, it

is easy to decide if a fragment needs to be rendered or discarded, using tests similar to the ones for the start/end line segment caps and body. The idea is thus to find the reference point u^* to which to compute the distance to.

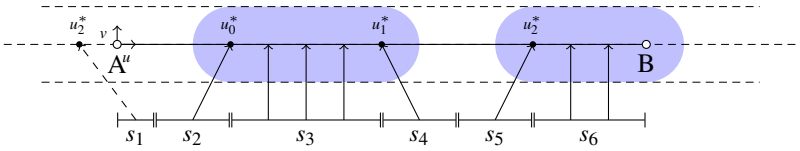

Figure 4. A thick dashed line between point A and B showing the reference points.

Consider figure 4 and the (u,v) coordinate of a fragment. Since the dash pattern is cyclic with a dash period ω, we can immediately compute the quantity $\overline{u} = u \bmod \omega$. If $\overline{u}$ lies between the start and end of the same dash, then we consider the reference point to be $\overline{u}$. Otherwise, we consider the reference point to be the nearest dash end or start. For example in figure 4, all fragments such that $\overline{u} \in s_2$ use u_0^* as reference point while all fragments such that $\overline{u} \in s_3$ uses themselves as reference points. Such a function can easily be stored in a single texture row that indicate where is the reference point according to the normalized $\overline{u}$ coordinate as illustrated on figure 5.

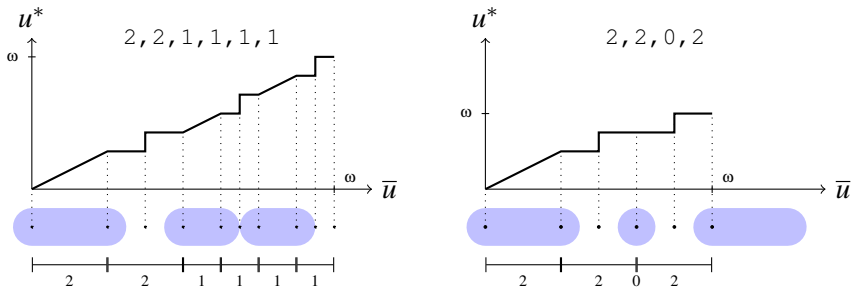

Figure 5. Examples of functions encoding the dash reference point u^* as a function of $\overline{u}$ for a given dash pattern.

However, this information is not sufficient to render a dash because we also need to know if a given reference point u^* refers to a cap start, a cap end or to the dash body. We could use the sign of the information to distinguish between cap start and cap end and use the null value to indicate a dash body (since in this case, we can use the $\overline{u}$ value) but we'll see later that the start and the end points of the current dash need to be stored as well (for broken polylines as opposed to smooth polylines, even though both are continuous). We will use a two-dimensional RGBA floating-point texture in order to store information relative to a dash pattern using the following structure:

- **R** - Reference point (u^*)

- **G** - Dash subtype (*start cap*, *body* or *end cap*)
- **B** - Dash start
- **A** - Dash end

A dash pattern can thus be stored as a row in this two-dimensional texture. This allows an arbitrary large number of dash patterns stored in a single texture that can be re-used for any dashed stroke.

5. Rendering

5.1. 2D Line segment

Rendering a line segment requires distinguishing between all the aforementioned cases as illustrated in figure 6:

1. The fragment belongs to a dash that ends before a line start
2. The fragment belongs to a dash that starts after a line stop
3. The fragment is before a line start and the dash extends across a line start
4. The fragment is after a line stop and the dash extends across a line stop
5. The fragment belongs to a dash cap start
6. The fragment belongs to a dash cap stop
7. The fragment belongs to a dash body

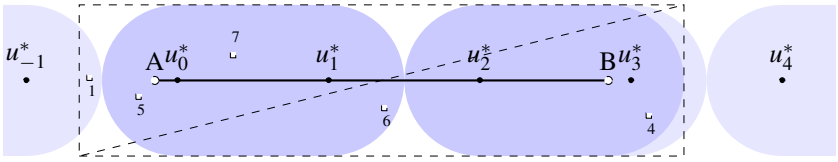

Figure 6. A dashed pattern between points A and B. The corresponding triangulation is shown as dashed lines and some fragment positions have been marked whose corresponding tests can be found in the listing 1. Not all tests are displayed because the displayed dash pattern does not need tests 2 and 3. The lighter blue thick dashed line corresponds to the uncorrected dash pattern, i.e. if the line was infinite in both directions.

The corresponding fragment shader is in listing 1.

```
uniform sampler2D dash_atlas;
uniform float     dash_index, dash_phase, dash_period;
uniform float     linelength, linewidth, antialias;
uniform vec2      caps, texcoord;
uniform vec4      color;

int main() {
  float w = linewidth;
  float freq = w*dash_period;
  float u = texcoord.x;
  float v = texcoord.y;
  float u_ = mod( u + w*dash_phase, freq );
  vec4 dash = texture2D(dash_atlas, vec2(u_/freq, dash_index));
  float dash_ref   = dash.x;
  float dash_type  = dash.y;
  float dash_start = (u - u_) + w * dash.z;
  float dash_stop  = (u - u_) + w * dash.w;
  float line_start = 0.0;
  float line_stop = linelength;
  bool cross_start = (dash_start <= line_start) &&
                     (dash_stop  >= line_start);
  bool cross_stop =  (dash_stop  >= line_stop) &&
                     (dash_start <= line_stop);
  float t = linewidth/2.0 - antialias;

  // Default distance to the line body (7)
  float d = abs(v);
  // Dash stop is before line start
  if( dash_stop <= line_start )
      discard;
  // Dash start is beyond line stop
  else if( dash_start >= line_stop )
      discard;
  // Dash is across line start and fragment before line start (1)
  else if( (u <= line_start) && (cross_start) )
      d = cap( caps.x, u, v, t);
  // Dash is across line stop and fragment after line stop (4)
  else if( (u >= line_stop) && (cross_stop) )
      d = cap( caps.y, u - line_stop, v, t );
  // Dash cap start (5)
  else if( dash_type < 0.0 )
      d = cap( caps.y, u-dash_ref, v, t );
  // Dash cap stop (6)
  else if( dash_type > 0.0 )
      d = cap( caps.x, dash_ref-u, v, t );

  // Antialias test
  d -= t;
  if( d < 0.0 ) {
```

```
        gl_FragColor = color;
    } else {
        d /= antialias;
        gl_FragColor = vec4(color.rgb, exp(-d*d)*color.a);
    }
}
```

Listing 1. Line fragment code

The cap function, show in listing 2, is a direct translation of table 1.

```
// t = linewidth/2.0 - antialias;
float cap( float type, float u, float v, float t )
{
    // None
    if                  ( type < 0.5 ) discard;
    // Round
    else if ( abs(type - 1.0) < 0.5 ) return sqrt(u*u+v*v);
    // Triangle out
    else if ( abs(type - 2.0) < 0.5 ) return max(abs(v),(t+u-abs(v)));
    // Triangle in
    else if ( abs(type - 3.0) < 0.5 ) return (u+abs(v));
    // Square
    else if ( abs(type - 4.0) < 0.5 ) return max(u,v);
    // Butt
    else if ( abs(type - 5.0) < 0.5 ) return max(u+t,v);

    discard;
}
```

Listing 2. Cap function

5.2. 2D Polylines without folding

Rendering a polyline without folding, i.e. no self intersection between consecutive segments, requires considering the angle between two consecutive line segments in order to decide whether they are considered *continuous* or *broken*. Let us consider a set of n points $\{P_i\}_{i\in[1,n]}$. An open path $\mathbf{O}$ is described by the set of line segments $\{P_iP_{i+1}\}_{i\in[1,n-1]}$. A closed path $\mathbf{C}$ is described by the set of line segments $\mathbf{O}\cup P_nP_1$. To tessellate the path into a thick stroke, for each segment $P_{i-1}P_i$, we have to consider respective angles $\alpha_{i-1} = \angle P_{i-2}P_{i-1}P_i$ and $\alpha_i = \angle P_{i-1}P_iP_{i+1}$. If the path is closed, we set $\alpha_0 = 0$ and $\alpha_n = 0$; if the path is open, we have $\alpha_0 = \angle P_nP_0P_1$ and $\alpha_n = \angle P_{n-1}P_nP_0$. Depending on whether angle α_i is greater or lesser than an arbitrary chosen angle $\alpha_{lim} > 0$ (that has been set to 10° in the supplemental material), P_i^0 and P_i^1 must be set accordingly as illustrated in figure 7. α_{lim} represents *de facto* the limit under which we consider the curve to be C^1 continuous even though, mathematically, it is not.

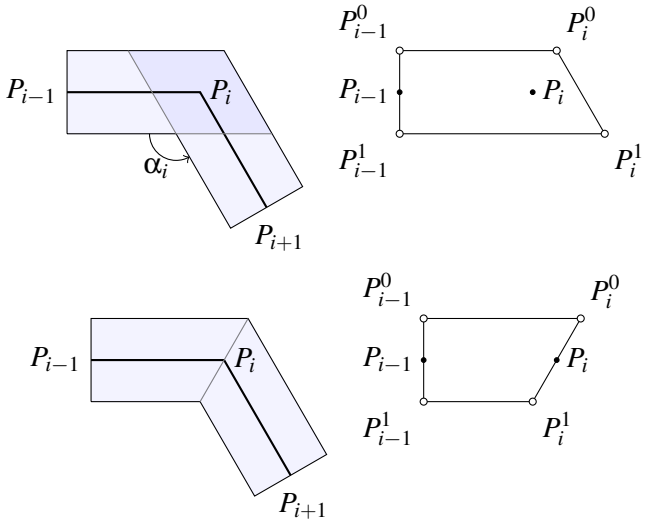

Figure 7. Connecting two thick line segments with overlap when $\alpha_i \geq \alpha_{lim}$ (top) or no overlap (bottom).

If two consecutive lines are continuous, so is the dash pattern. If two consecutive line segments are broken, we stop and restart the dash pattern accordingly. The difficulty using this approach is to keep the overall dash pattern consistent across all the segments as illustrated in figure 8.

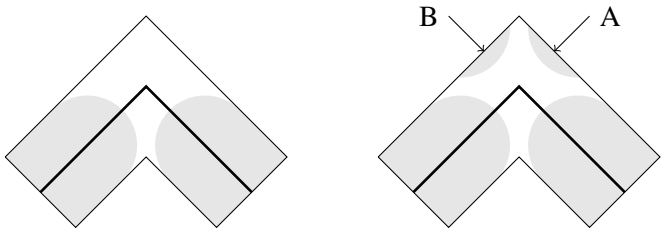

Figure 8. For broken angles, one has to take care to not start a dash pattern before the start of the current line segment (A) or beyond the end of the current line segment (B) or unaesthetic artifacts will result.

We thus need to make sure that:

- a new dash pattern does not start beyond the current segment end (case A in figure 8)
- a dash pattern does not end before the actual segment start (case B on figure 8)

For any dash, we have to know anytime where it starts and where it ends. Fortunately, this information is in the dash atlas (in the blue and alpha channel respectively). The

corresponding shaders can be found in the supplemental materials. The full dash shader code takes into account the type of join (round, miter, bevel) and renders it without any extra tessellation.

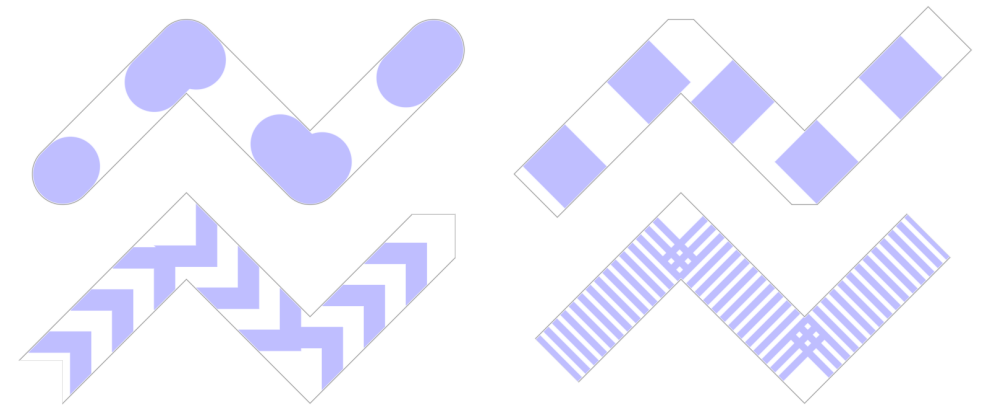

Figure 9. Broken polylines with different joins, dash patterns and periods.

5.3. 2D polylines with folding

The case of folded thick polylines can be solved by first unfolding the polyline using the algorithm introduced in [Asente 2010] and then applying the previous cases.

5.4. 3D polylines without folding

The 3D case is very similar to the 2D case if we consider a line as always facing the camera. That is, we consider a line segment in the 3D space to be defined as two points and a line thickness. This is different from, for example, a thick and flat ribbon that would have an orientation in space. Using line impostors, we can easily compute the 4 vertices that constitute the line. Figure 11 shows dashed 3D line segments while figure show the case of a continuous polyline around a sphere. Note the different apparent thickness of the lines depending on perspective.

6. Benchmarks

Benchmarks were performed on a Macbook Pro Retina 15-inch (early 2013), with 2.7Ghz Intel Core i7, 16GB 1600 MHz DDR3, NVIDIA GeForce GT 650M (1024 Mo) using OSX 10.9 (Mavericks) and a modified GLUT[1] version to take HiDPI into account.

[1] available from `http://iihm.imag.fr/blanch/software/glut-macosx/`

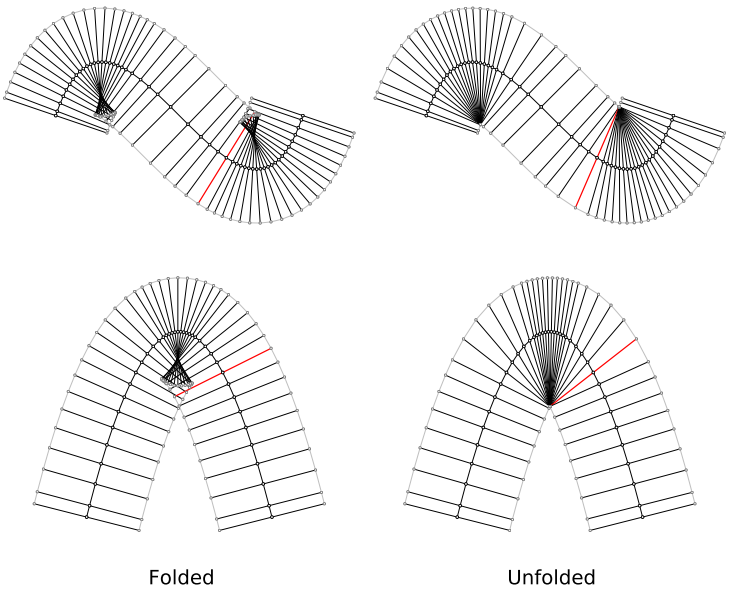

Figure 10. In some cases, folded thick polylines can be unfolded. The thick polyline can then be considered continuous and the previous case applies.

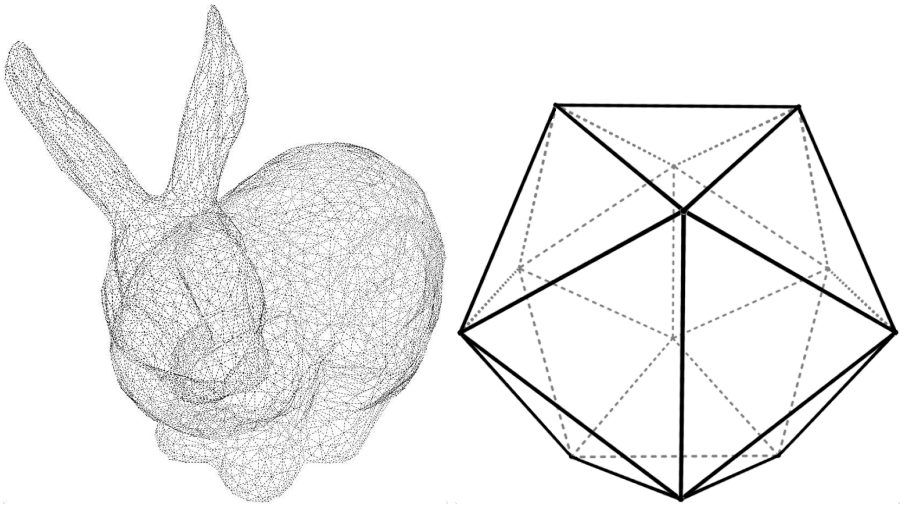

Figure 11. Left. The Stanford bunny rendered with dashed line segments. Right. Icosahedron rendered with dashed hidden line segments.

6.1. Speed

Rendering speed was measured using the total time to render 10,000 line segments of apparent width of 1 pixel for a total of 1,000 frames. The cost of rendering these frames with no lines was subtracted and divided by the number of frames. The reference time corresponds to the *raw* method that builds thick line segments using 2

triangles with no caps nor antialiasing. The *solid* method corresponds to a dedicated shader that handles only solid lines with caps, joins and antialiasing. The *dash solid* method corresponds to the shader program that handles dashed or solid lines with caps, joins and antialias. However, in this case, the rendering benefits from some internal optimizations due to the solid nature of the line. The *dash dotted* method corresponds to the full shader program but the line is rendered using a dot pattern that slows down rendering.

Method	Total time	Average time	Slowdown factor
No rendering	0.671 (s)		
Raw	18.427 (s)	17.756 (ms)	$\times$1.00
Solid	20.338 (s)	19.667 (ms)	$\times$1.10
Dash solid	20.387 (s)	19.716 (ms)	$\times$1.10
Dash dotted	37.872 (s)	37.201 (ms)	$\times$2.10

Table 2. Speed benchmarks for the three main rendering methods (code is available from the supplemental materials) for an apparent line thickness of 1.

Finally, I measured how the technique scales with the actual line width. Rendering speed was measured using the total time to render 1,000 line segments with a varying line width between 1 and 51 for a total of 1000 frames. The cost of rendering these frames with no lines was subtracted and figure 12 shows a linear correlation between the line width and the time to render.

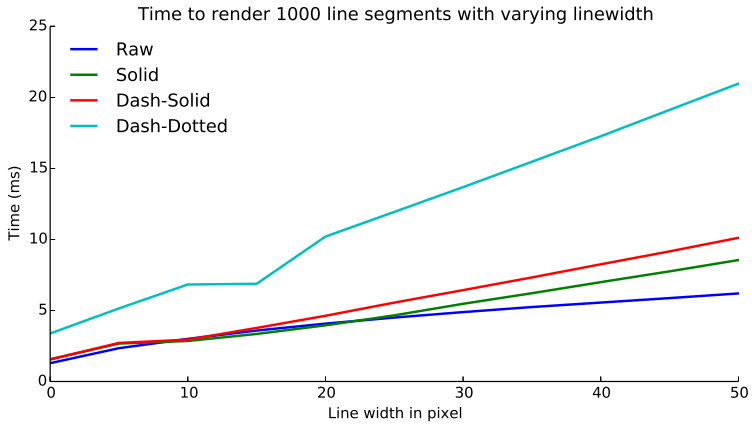

Figure 12. The time to render line segments with increasing line widths is linearly correlated with the line width.

6.2. Memory

In a polyline of n points, the minimum number of vertices to render a thick polyline is $2n$ using a triangle strip structure. In my case, I cannot use such structure since I need to parameterize vertices depending on their relative position in the current segment (start or end position). For example, if we consider three points A, B and C, I need to handle case where B is the end of segment AB and case where B is the start of segment BC. A polyline is thus rendered using triangles resulting in $4n-4$ vertices. Furthermore, a vertex needs to be augmented with several types of data in order to render dashes, joins and caps as shown in listing 3.

```
typedef struct {
  vec2  position;   // vertex position
  vec2  segment;    // vertex curvilinear coordinates
  vec2  angles;     // angles with previous and next segment
  vec4  tangents;   // tangents with previous and next vertex
  vec2  texcoord;   // Texture coordinates
} vtype;
```

Listing 3. Vertex type

Compared to a raw thick polyline rendering where only the vertex position and texture coordinate are needed (4 floats), this represents a $\times 3$ increase in memory consumption. The overall increase in memory consumption, taking into account the number of vertices and the memory size of a vertex is then of a factor $\times$ 6 compared to the raw line rendering.

6.3. Dynamic lines

As explained previously, a vertex needs to be instrumented with several information, then a polyline can be modified in a number of ways that do not require any extra computation or tessellation as illustrated on table 3. The partial update is due to if a single point of a polyline is changed, this only requires updating:

1. Vertices' information related to these points (4 vertices)
2. Immediate neighbors' `tangents` and `angles` information (4 vertices)
3. `segment` information for all subsequent vertices

The only case where all information for all vertices need to be computed is when all the points are changed at once.

7. Conclusion

Our method suffers from several errors and limitations. First, dash caps are curved to follow the contour instead of being rendered independently of the curvature. The

Property	CPU dash	GPU dash
Rotation	✓	✓
Zoom	✓	✓
Line caps	✓	✓
Line thickness	×	✓
Join type	×	✓
Dash pattern	× (period-dependent)	✓
Dash phase	×	✓
Dash caps	✓	✓
Point modification	×	× (partial update)

Table 3. Table summarizing which property can be changed without the need for a new tessellation depending on the technique.

Open XML Paper Specification defines 4 types of caps: **flat, square, round**, and **triangle**. In all 4 cases, the boundaries of the caps

> *... are not curved to follow the contour, but are transformed using the effective render transform.*

As illustrated in figure 13, this means that there exist paths where the caps are not fully contained within the envelope of the stroked path.

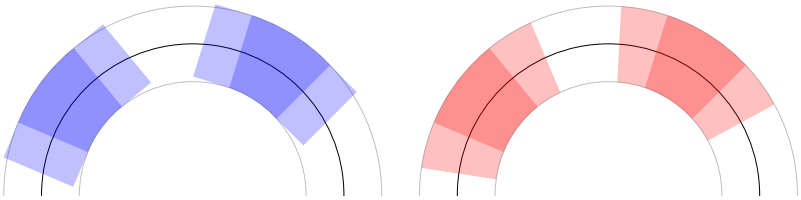

Figure 13. **Left** Square dash caps rendered following the Open XML specification. Note that caps are not fully contained within the stroke envelope. **Right** Approximated square dash caps that are fully contained within the stroke envelope and follow the contour curvature.

Our method does not follow this specification; instead dash caps are rendered as illustrated on the right part of the figure 13. This is not noticeable for straight lines, hardly noticeable for thin lines but becomes visible for thick lines, especially in the vicinity of high curvature. It is thus wrong according to the Open XML specification.

A second and more serious problem is that broken polylines create self-intersection areas, which result in artifacts when the stroke is painted using a transparent color. This can be partially fixed using the stencil buffer, but this also creates new artifacts

when antialiasing the line. This could also be fixed directly from within the fragment shader (by considering which segment is responsible for actually painting the join section) but I suspect this would add considerable complexity to the shader and I did not test it. Third, the dash pattern period cannot exceed the width of the dash atlas texture or this would bring severe imprecision during the rendering stage. Most common dash pattern periods are generally below 10 or 20 times the line width, and a texture width of 1024 pixels ensures more than enough precision. If one needs a very long dash period, it would become necessary to interpolate the texture over two or more consecutive lines.

Last, a fully antialiased, correctly joined, dashed line is roughly 2 times slower than a raw thick line. However, given the quality of the output and considering the fact that dash patterns are generally used scarcely in a given scene, the versatility and the ease of use may be worth considered as an alternative solution: line width, line caps, joins types, dash pattern, dash caps and dash phase can be all changed at no extra cost. Finally, the proposed implementation (in supplemental material) does not use geometry shaders, which are not available in OpenGL ES 3.0 or WebGL 1.0, even if the baking process could be made entirely on a geometry shader, hence offering a modern way of replacing the deprecated stipple GL feature without the need for any extract code on the CPU side.

References

ASENTE, P. J. 2010. Folding avoidance in skeletal strokes. In *Proceedings of the Seventh Sketch-Based Interfaces and Modeling Symposium*, Eurographics Association, SBIM '10, 33–40. 9

BÉNARD, P., COLE, F., GOLOVINSKIY, A., AND FINKELSTEIN, A. 2010. Self-similar texture for coherent line stylization. In *Proceedings of the 8th International Symposium on Non-Photorealistic Animation and Rendering*, ACM, 91–97. 2

CHAN, E., AND DURAND, F. 2005. *GPU Gems II: Programming Techniques for High-Performance Graphics and General-Purpose Computation* . Addison-Wesley Professional, ch. 22. Fast Prefiltered Lines, 345–369. 2, 3

COLE, F., AND FINKELSTEIN, A. 2010. Two Fast Methods for High-Quality Line Visibility. *IEEE Transactions on Visualization and Computer Graphics 16*, 5, 707–717. 2

KILGARD, M., AND BOLZ, J. 2012. GPU-accelerated Path Rendering. *ACM Transactions on Graphics (Proceedings of SIGGRAPH Asia 2012) 31*, 6. 2

KILGARD, M., 2011. Conversion of Dashed Strokes into Quadratic Bézier Segment Sequences. Patent application number: 20110285723. 2

MARKOSIAN, L., KOWALSKI, M. A., TRYCHIN, S. J., BOURDEV, L. D., GOLDSTEIN, D., AND HUGHES, J. F. 1997. Real-Time Nonphotorealistic Rendering. In *SIGGRAPH 97 Conference Proceedings*, 415–420. 2

Index of Supplemental Materials

Supplied code

The supplied python code demonstrates various dash lines in 2D and 3D. It requires the numpy library available from: `http://numpy.scipy.org` and the OpenGL python bindings available from: `http://pyopengl.sourceforge.net` All the screenshots from the article are available from the demo executable scripts. The code is BSD licensed and the **README** file explains each file.

Screenshots

Movies

Movies of animated demos are available from:

- http://www.loria.fr/ rougier/tmp/sphere.mov
- http://www.loria.fr/ rougier/tmp/stars.mov
- http://www.loria.fr/ rougier/tmp/tiger.mov
- http://www.loria.fr/ rougier/tmp/icosahedron.mov

Author Contact Information

Nicolas P. Rougier
Mnemosyne, INRIA Bordeaux - Sud Ouest
LaBRI, UMR 5800 CNRS, Bordeaux University
Institute of Neurodegenerative Diseases, UMR 5293
351, Cours de la Libération
33405 Talence Cedex, France
Nicolas.Rougier@inria.fr
http://www.loria.fr/~rougier

Nicolas P. Rougier, GPU-Accelerated Dashed Stroked Polylines, *Journal of Computer Graphics Techniques (JCGT)*, vol. ?, no. ?, 1–1, ????
????

Received:	November 9, 2013		
Recommended:	????-??-??	Corresponding Editor:	??? ???
Published:	????-??-??	Editor-in-Chief:	Name

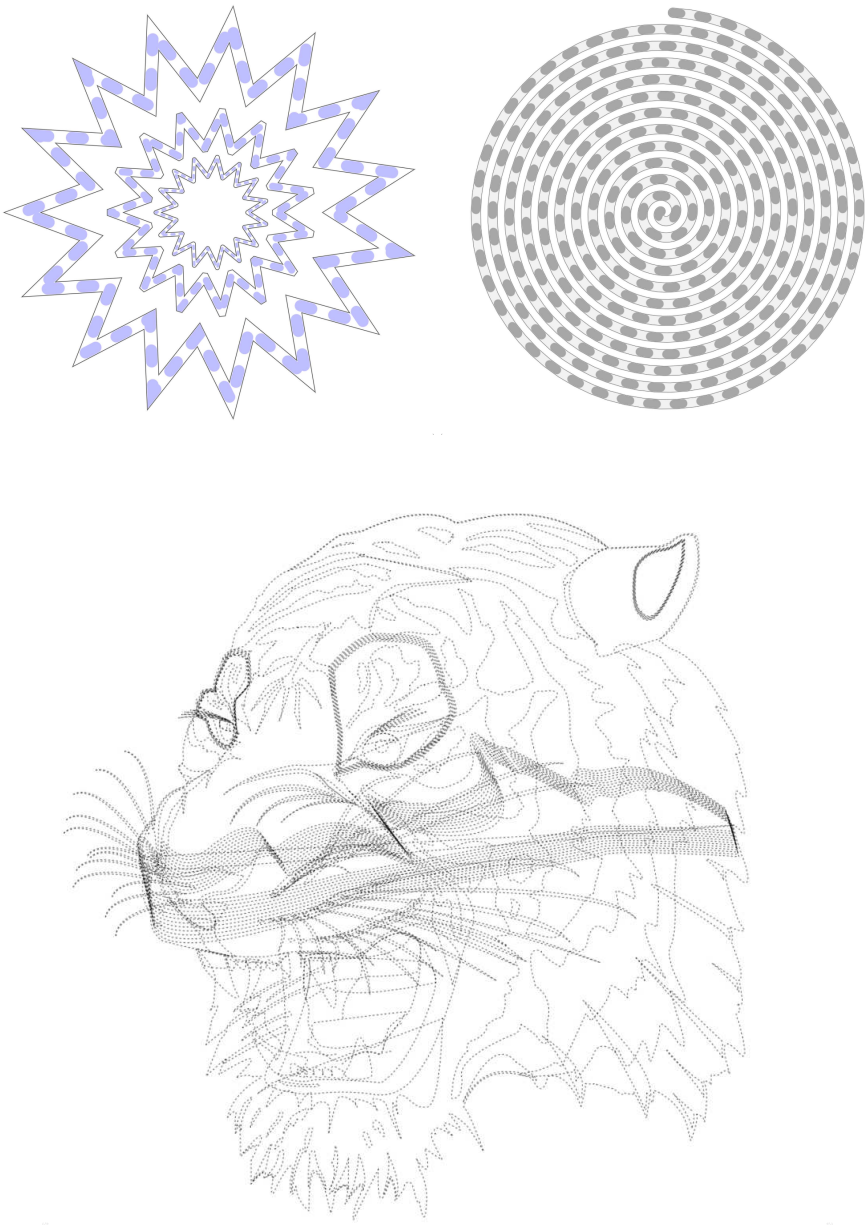

Figure 14. Top Left. Stars using broken polylines. Top Right. Spirals using a single continuous polyline. Bottom. Tiger using dashed Bézier paths.