Fast and versatile texture-based wireframe rendering

Article *in* The Visual Computer · October 2011

DOI: 10.1007/s00371-011-0623-6 · Source: DBLP

2 authors:

Waldemar Celes
Tecgraf/PUC-Rio Institute
85 PUBLICATIONS **1,432** CITATIONS

SEE PROFILE

Frederico Abraham
TECGRAF / PUC-Rio
4 PUBLICATIONS **40** CITATIONS

SEE PROFILE

Vis Comput (2011) 27:939–948
DOI 10.1007/s00371-011-0623-6

ORIGINAL ARTICLE

Fast and versatile texture-based wireframe rendering

Waldemar Celes · Frederico Abraham

Published online: 27 August 2011

Abstract This paper revisits the problem of wireframe rendering, which, at first, appears to be an easily solved problem. However, the conventional solution is inefficient and does not result in high-quality images. Recently, graphics hardware programming has been employed to achieve high-quality solid line rendering. In this paper, we present a simpler and faster technique for wireframe rendering based on texture mapping. Our technique does not require (but can benefit from) graphics hardware programming and thus can be easily integrated into existing rendering engines, while resulting in fast, accurate, high-quality, anti-aliased, and still versatile, wireframe drawing. For topologically structured meshes, our approach allows the rendering of wireframe decoupled from the underlying mesh, making possible the rendering of original wireframes on top of decimated meshes.

Keywords Wireframe rendering · Anti-aliased lines · Mipmapping · Geometry shader

1 Introduction

Wireframe rendering is important for several applications, including CAD, solid modeling, non-photorealistic rendering, and technical illustration. Rendering the wireframe is important for revealing structural information of the underlying mesh used to model the object. This is especially true in scientific visualization of models submitted to numerical simulations, where the quality of the mesh is of great significance and thus has to be inspected. An application can choose to render the wireframe in isolation or combined with shaded surfaces.

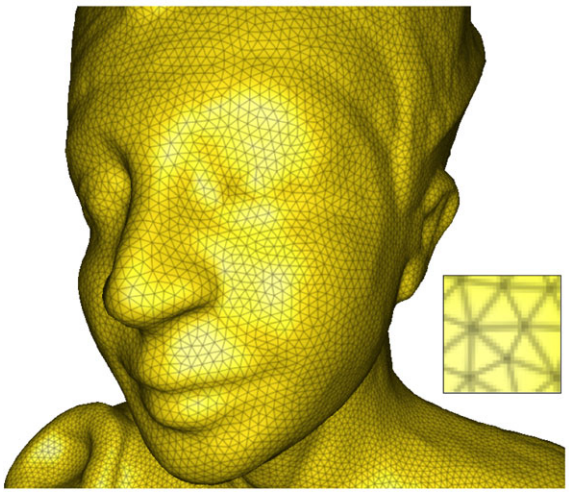
Wireframe rendering is conceptually an easily solved problem. The goal is to visualize the wireframe of a mesh, i.e., to render the edges of a mesh. In practice, this turns out to be a tricky problem because the pixels computed by the rasterization of lines do not always match the border pixels computed by the rasterization of polygons, due to different depth value interpolation. The conventional solution consists of a two-pass rendering algorithm. The first pass renders the polygons with a small depth offset; the second pass renders the lines. This solution is inefficient and inaccurate.

For years, this conventional algorithm was assumed as a standard for real-time applications. Previous proposals for line drawing based on textures do not preserve line thickness [7, 9]. A few alternatives based on the stencil buffer or the A-buffer were proposed but lack efficiency [8, 12]. Recently, Bærentzen et al. [2, 3] proposed a new efficient algorithm based on graphics hardware programming.

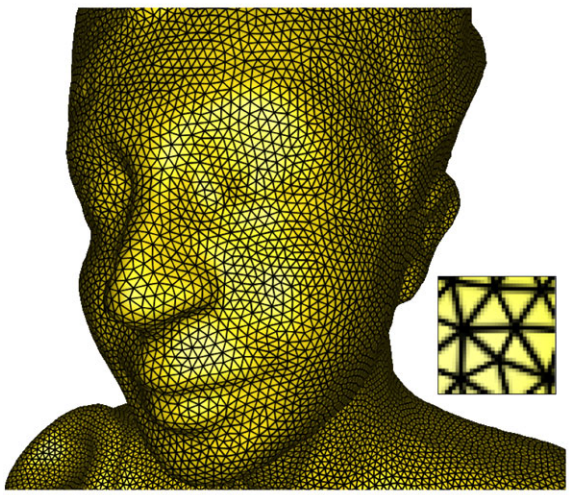
In this paper, we revisit the problem of wireframe rendering and propose a new single-pass algorithm based on texture mapping. Our technique is simpler and faster than previous ones, while resulting in accurate, high-quality, anti-aliased images. We have used it in a variety of industrial applications. It does not require (but can benefit from) graphics hardware programming and is still versatile enough to handle arbitrary line thickness values and attenuation (depth cueing). Figure 1 shows some images achieved with our technique for different thickness values. In this figure, the rendered image is locally enlarged to illustrate the achieved anti-aliased lines. Our texture-based proposal is particularly

W. Celes (✉) · F. Abraham
Tecgraf/PUC-Rio—Computer Science Department, Pontifical Catholic University of Rio de Janeiro, Rio de Janeiro, Brazil
e-mail: celes@inf.puc-rio.br

F. Abraham
e-mail: fabraham@tecgraf.puc-rio.br



(a) $\sigma = 0.5$



(b) $\sigma = 2.0$

Fig. 1 Texture-based wireframe rendering with different thickness values (σ) on the Ramsés model

suitable for rendering the wireframe of topologically structured meshes, which are widely used in numerical simulations. The proposed technique also handles the rendering of lines decoupled from the mesh, which has some important applications, such as the rendering of grids over curved surfaces or of isolines of scalar fields in scientific visualization.

The proposed technique was first presented in [5]. In this paper, we extend that work, bringing the following new contributions:

- A new version of the algorithm that sets the texture coordinates for triangle meshes is described, significantly reducing the number of duplicated vertices.
- The technique is extended for drawing the wireframe of topologically structured meshes based on a global texture coordinate system. This allows the rendering of the original wireframe on the top of decimated versions of the mesh, which is especially relevant for rendering massive models using level-of-detail techniques.

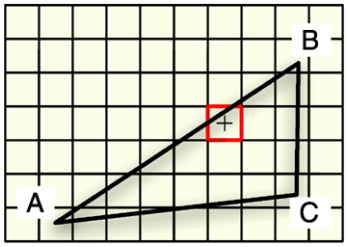


Fig. 2 The depth value of a line fragment differs from the corresponding depth value of a polygon fragment [4]

Additionally, we have run new computational experiments, applied to different models, using different generations of new graphics cards. The achieved results reinforce that our proposal performs better than previous ones.

The rest of this paper is organized as follows. Section 2 reviews the problem of wireframe rendering and describes previous proposals. Section 3 presents our texture-based proposal in detail, and a performance comparison is shown in Sect. 4. Section 5 shows the benefits of employing our technique for rendering the wireframe of topologically structured meshes. Section 6 discusses the use of our technique for drawing lines decoupled from the mesh geometry. Finally, concluding remarks are drawn in Sect. 7.

2 Related work

Rendering high-quality wireframe on top of a shaded surface is harder than it appears at first. The main problem is due to having different depth values assigned to fragments originated from line or polygon rasterization. Bærentzen et al. [4] illustrated this problem using the image in Fig. 2. The highlighted pixel belongs to the edge AB and also to the polygon border. However, the depth values at the center of this pixel computed by the two (line and polygon) rasterization algorithms differ: the depth value of the line fragment is computed by interpolating only vertices A and B, while the depth value computation of the corresponding polygon fragment also considers vertex C. Due to this difference, one cannot ensure which fragment will be rendered in front.

The conventional, two-rendering-pass algorithm alleviates the depth value conflict by adding a small z-bias to the polygon fragments. Although widely used in commercial applications, this algorithm is inefficient because it requires two rendering passes, doubling the geometry load; also, line rasterization itself is not efficiently implemented in some graphics cards. The algorithm is inaccurate because there is no standard z-bias value that completely eliminates the visibility conflict [1]. Moreover, in general, this algorithm produces aliased lines.

To disambiguate the visibility conflict between line and polygon fragments, Herrell et al. [8] proposed to use the stencil buffer. Their algorithm produces accurate images but requires rendering each triangle separately, thus being inefficient for real-time applications. Wang et al. [12] proposed to solve the visibility conflict by computing primitive intersection on a per-fragment basis. Although effective, their solution is not easily (if at all possible) mapped to modern graphics cards. Recently, Tang et al. [11] have proposed the use of graphics programming for encoding triangle's IDs in a texture. In the second rendering pass, the ID texture is used to identify if the edge being drawn is a border of the encoded visible triangle, thus being also visible. For large meshes, they reported competitive performance when compared to the conventional z-bias algorithm.

Bærentzen et al. [2, 3] proposed a single-pass wireframe-rendering algorithm based on graphics hardware programming. Their proposal does not use the line primitive at all; instead, the edges are drawn as part of the polygon rasterization. The general idea is to use a fragment shader that evaluates the distance of each fragment center to the closest edge. If this distance is smaller than a given threshold (the line thickness value), the fragment is colored with the wireframe color. As a result, their algorithm is able to render high-quality solid wireframes and presents better performance than the conventional two-pass approach. To compute the raster distance from each fragment to the edges, a geometry shader is employed: for each vertex, the distance to the opposite edge (assuming triangle primitives) is computed and assigned as a texture coordinate. The distance values are then interpolated by the rasterizer for each fragment. Although conceptually simple, this algorithm requires graphics hardware programming and faces the challenge of computing distance values after the projection transformation. When the vertex is close or behind the viewer, its distance in viewport space cannot be computed [6].

In this paper, we present a new algorithm for wireframe rendering. Like Bærentzen et al. [2, 3], we render the edges as part of the polygon rasterization but, instead of relying on graphics hardware programming, we simply use texture mapping. As a result, our algorithm presents better performance, is easier to implement, and preserves image quality. We also show that our approach is particularly suitable for rendering the wireframe of topologically structured meshes, even on the top of decimated versions of the original model.

The use of textures for line rendering is not new. Haeberli and Segal [7] suggested different ways for using textures for anti-aliased line drawing but did not identify a method for efficient wireframe rendering. Kuschfeldt and Holzner [9] focused on finite element mesh rendering and proposed the use of a two-dimensional texture for drawing the border of quadrilateral elements (triangular elements were drawn as collapsed quadrilaterals). Like ours, their proposal draws the mesh in a single rendering pass, but, unlike ours, it does not preserve line thickness. Also, their proposal does not render meshes in an efficient way. Later, Rose and Ertl [10] presented a specialized system for applying level-of-detail techniques over large finite element models. To draw an approximation of the original mesh wireframe over a simplified surface patch, they proposed to code distances to the edges in a 2D texture, using dependent texture lookup to render the lines. Their proposal is specifically designed for rendering simplified quasi-structured quadrilateral meshes.

3 Proposed texture-based technique

Our algorithm renders the wireframe in a single-rendering pass and basically consists in using texture mapping to draw edges together with polygon rasterization. The texture mapping is done using a 1D RGBA texture, the *wireframe texture*, that represents half of the line (across the thickness direction). Each polygon draws one side of the line, and the complete line is rendered after two adjacent polygons are rasterized. Figure 3a illustrates the texture used to render wireframes with a thickness value equal to 3.0. The alpha channel, which is illustrated in the figure, encodes opacity. In this case, half of the line thickness (a value of 1.5) is represented by setting the alpha value of the last texel to 1.0 and of its neighbor texel to 0.5. Mipmapping is used to ensure that line thickness is preserved despite the size of the primitive when mapped to the screen. At each level of the mipmapping pyramid, the texels representing the wireframe are preserved (Fig. 3b). Note that we can use different thickness values, as shown in Fig. 1.

To achieve the desired results, we need to set appropriate texture coordinates when drawing the graphics primitives. For a triangle, we use three texture units, binding the same texture object to all of them. Each texture unit is used for drawing one triangle edge. For a given unit, we set the

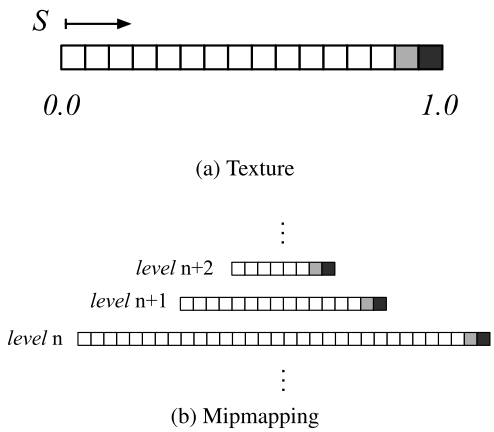
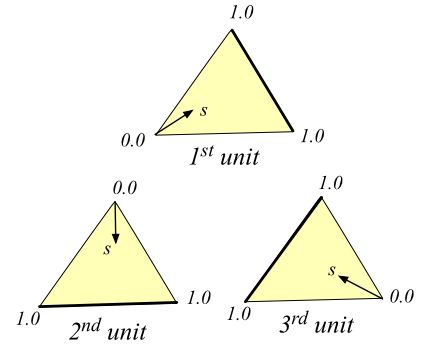
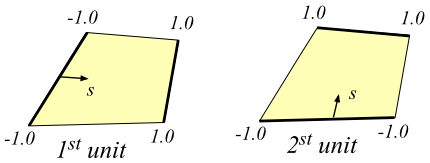


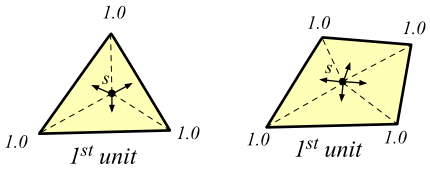
Fig. 3 Texture used to render the wireframe. The image illustrates opacity values set to texels



(a) Triangle



(b) Quad



(c) Alternative

Fig. 4 Vertex texture coordinates to render edges of triangles and quadrilaterals

texture coordinate equal to 0.0 for one vertex and equal to 1.0 for the other two vertices, as illustrated in Fig. 4a. For a quadrilateral, it suffices to use two texture units, setting texture coordinates -1.0 and 1.0 to vertices at opposite sides and mapping the texture with mirrored-repeat wrapping mode (Fig. 4b). An alternative approach would be to create an additional vertex in the middle of the primitive and use only one texture unit, rendering a different triangle for each edge of the original primitive, as illustrated in Fig. 4c. This is valid for any convex polygon. Naturally, this strategy imposes an additional load on the geometry stage of the graphics pipeline, but it can be useful for applying the method when the number of available texture units is limited.

Wireframe can be rendered in isolation or combined with shaded surfaces. When combined with a shaded surface, the texel RGB values are used to encode the wireframe color, and the texture function is set to *decal*. For representing the wireframe in isolation, we set the texel color to white, using *modulate* for the first texture unit and texture combiner for adding the contribution of each subsequent unit. The wireframe then receives the color of the primitive, as illustrated in Fig. 5.

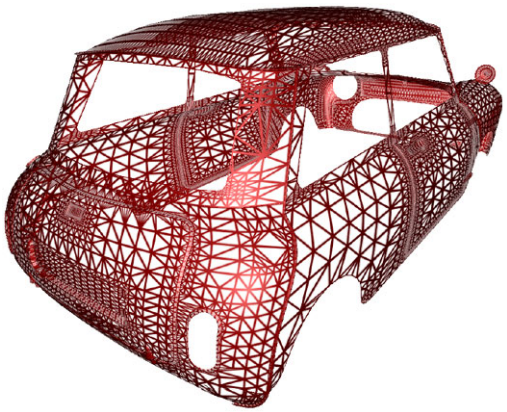


Fig. 5 Wireframe of a Mini Cooper model drawn in isolation, with thickness value set to 4

3.1 Avoiding saturation

One advantage of using texture mipmapping for wireframe rendering is that it naturally ensures line thickness in screen space, while achieving high-quality anti-aliased images. We can also adjust the highest levels of the mipmapping pyramid to avoid saturating the image with the wireframe. This would happen whenever the primitive in viewport space becomes smaller than the line thickness: the whole primitive would be filled with the wireframe color. We avoid this saturation by limiting the thickness in relation to the texture dimension at each level of the mipmapping pyramid. The images shown in this paper were generated with a limiting factor of $1/3$. This means that, at the highest levels of the pyramid, the thickness value is not honored, but reduced to $1/3$ of the level dimension. Avoiding image saturation is especially important for rendering dense meshes. Figure 6 compares the images obtained without and with this strategy for avoiding saturation on the Magalis' Hand model. In this case, as the wireframe is not illuminated, the saturated image does not reveal the shape of the surface. Note also that the thickness value at regions close to the viewer (e.g., tip of middle finger) is naturally honored in both cases.

3.2 Mesh rendering

For rendering individual primitives, it suffices to activate the texture and set the texture coordinates explicitly. However, for efficient rendering of complex models, we need to pack vertex attributes into arrays and share vertices among primitives to take full advantage of graphics card's cache. The assignment of texture coordinates as indicated in Fig. 4a and 4b perfectly matches the way vertices are arranged in triangle and quad strips, respectively. However, for mesh rendering, we need to duplicate vertices to ensure that, for each primitive, the texture coordinates assigned to the corresponding vertices follow the scheme illustrated in Fig. 4.

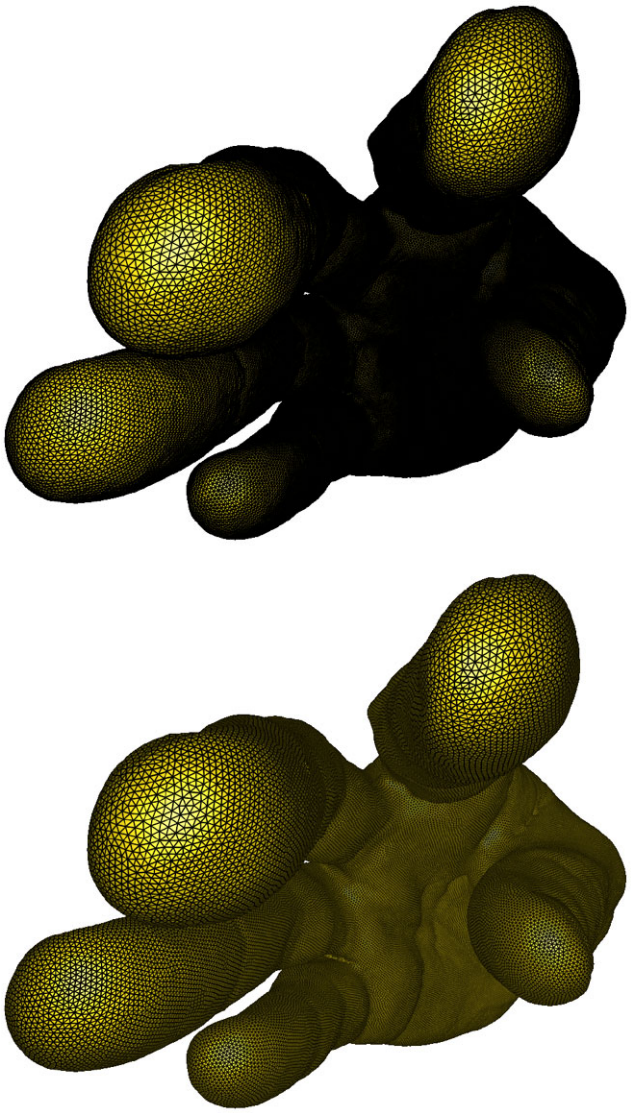


Fig. 6 Importance of avoiding image saturation: *on the top*, without limiting line thickness for the highest level of the mipmapping pyramid; *on the bottom*, with the proposed limiting strategy

For a triangle mesh, we have devised a simple algorithm to identify and duplicate the vertices that could not be shared by all its incident triangles. The algorithm sets a label α, β, or γ for each vertex of the mesh. Vertices labeled as α are assigned texture coordinates $s_0 = 0.0$, $s_1 = 1.0$, and $s_2 = 1.0$ for the first, second, and third texture units in use, respectively. Vertices labeled as β are assigned texture coordinates $s_0 = 1.0$, $s_1 = 0.0$, and $s_2 = 1.0$; and vertices labeled as γ are assigned $s_0 = 1.0$, $s_1 = 1.0$, and $s_2 = 0.0$. In the final mesh, the three vertices incident to each triangle have to be labeled with different letters, ensuring that all its three edges will be correctly rendered.

The algorithm starts by first setting labels α, β, and γ for the incident vertices of a first triangle. Then, it performs a depth-first search visiting each neighboring triangle across the edges. For each new visited triangle, it sets a label to the unlabeled vertex, choosing the label that does not conflict with the already labeled triangle vertices. If the vertex is already labeled with a conflicting label, the vertex is duplicated and the triangle incidence is updated [5]. In that case, instead of proceeding with the depth search across the triangle edges, we interrupt the search whenever a vertex duplication is required—the search proceeds at the next triangle on the top of the stack. An outside loop ensures that all triangles are traversed. For each vertex, the algorithm keeps a list of the corresponding duplicated vertices, so each duplicated vertex can be reused for another, not yet visited, incident triangle.

Table 1 Results of the algorithm on different models: number of triangles (#T), number of vertices in the original model (#V), number of non-conflicting vertices (#V′), and percentage of duplicated vertices

Model	#T	#V	#V′	Extra
Magalis' Hand	396,730	198,367	258,923	31%
Neptune	411,678	205,835	276,818	34%
Dragon	871,306	435,545	615,185	41%
Manuscript	4,305,818	2,155,617	2,412,601	12%

The algorithm was applied to different models, as illustrated in Table 1. As can be noted, for the tested models, the number of duplicated vertices is at most 40% of the original number of vertices in the mesh. These extra vertices impose an additional load on the graphics pipeline, increasing the amount of data transferred to the graphics card and reducing the effective use of vertex caches. However, even with this extra load, our texture-based algorithm is faster than previous proposals when rendering models in retained mode.

Nevertheless, we can completely avoid vertex duplication if we consider graphics hardware programming. More specifically, we can code a simple *geometry shader* that automatically generates appropriate texture coordinates, without processing any extra vertices. For each input triangle, the geometry shader outputs the same triangle with its already processed vertices, adding texture coordinates accordingly. The complete GLSL geometry shader code is presented in Fig. 7. The gain in performance is significant if vertex data have to be transferred to the graphics card at each frame.

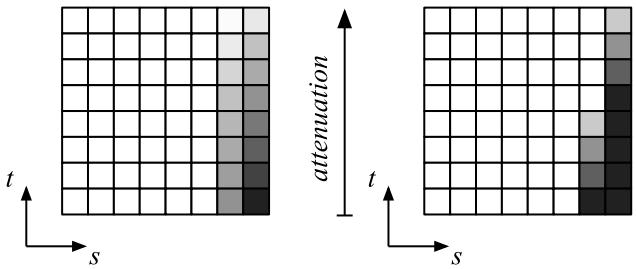
Unfortunately, on current graphics cards, the use of a geometry shader still requires the coding of a vertex shader. This may impose difficulties to integrate the use of a geometry shader into existing applications. Needless to say, the algorithm to duplicate the vertices is also useful where the geometry shader is not supported.

3.3 Attenuation

Because the technique proposed by Bærentzen et al. [2, 3] is based on fragment shaders, several variations of the method

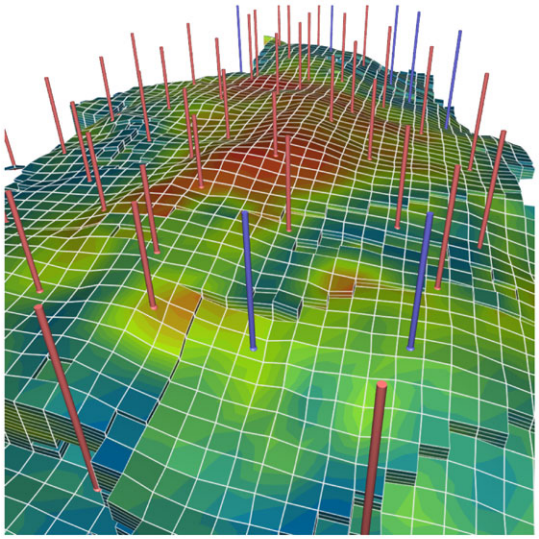
```
void main(void)
{
     gl_Position   = gl_PositionIn[0];
     gl_FrontColor = gl_FrontColorIn[0];
     gl_TexCoord[1] = vec4(1.0,0.0,0.0,1.0);
     gl_TexCoord[2] = vec4(1.0,0.0,0.0,1.0);
     gl_TexCoord[3] = vec4(0.0,0.0,0.0,1.0);
     EmitVertex();
     gl_Position   = gl_PositionIn[1];
     gl_FrontColor = gl_FrontColorIn[1];
     gl_TexCoord[1] = vec4(0.0,0.0,0.0,1.0);
     gl_TexCoord[2] = vec4(1.0,0.0,0.0,1.0);
     gl_TexCoord[3] = vec4(1.0,0.0,0.0,1.0);
     EmitVertex();
     gl_Position   = gl_PositionIn[2];
     gl_FrontColor = gl_FrontColorIn[2];
     gl_TexCoord[1] = vec4(1.0,0.0,0.0,1.0);
     gl_TexCoord[2] = vec4(0.0,0.0,0.0,1.0);
     gl_TexCoord[3] = vec4(1.0,0.0,0.0,1.0);
     EmitVertex();
     EndPrimitive();
}
```

Fig. 7 GLSL geometry shader

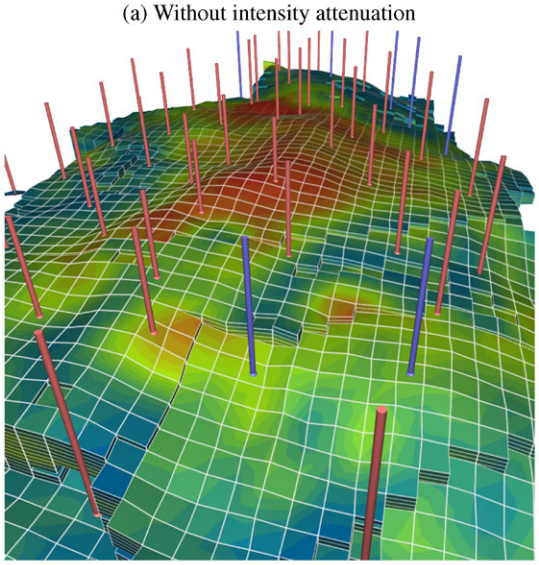


(a) Intensity (b) Thickness

Fig. 8 2D texture for line attenuation



(a) Without intensity attenuation



(b) With intensity attenuation

Fig. 9 A black oil reservoir model with wireframe rendering

are possible. They have illustrated such a versatility by using attenuation of line intensity and thickness. For line intensity attenuation, the lines fade away according to their distance to the viewer. Accordingly, the thickness attenuation ensures that distant lines appear thinner.

Attenuation is also possible and easy to achieve with our technique. For that purpose, we replace the 1D texture by a 2D texture, and use the second dimension to represent the attenuation effect. For intensity attenuation, we vary the texels' opacity along the t direction and, for thickness attenuation, we vary the represented line thickness, as illustrated in Fig. 8.

The s texture coordinates are set as already described, while the t coordinates are set by enabling automatic texture coordinate generation in eye space. Figure 9 illustrates the intensity attenuation effect on a black oil reservoir model.

3.4 Limitations

The proposed texture-based technique for wireframe rendering presents the same limitation as Bærentzen et al.'s proposal [3, 6]: it draws only half of the silhouette edges. This limitation is intrinsic to the strategy of drawing the lines as part of polygon rasterization. As silhouette edges have only one visible adjacent polygon, the edges appear thinner and aliased.

The proposed texture-based approach is only applicable for preserving line thickness in screen space; Bærentzen et al.'s proposal, on the other hand, also allows constant thickness in world space.

Our approach also limits the level of zoom-in without increasing line thickness; one may eventually reach the largest texture dimension in the mipmapping pyramid, thus forc-

ing texture magnification. However, in practice, this does not tend to be an actual limitation. In our code, we have set texture width to 4096, and we rarely have the case where a single primitive is projected with this size on the screen.

4 Performance comparison

For performance comparison, we have run a set of computational experiments. We compared the performance achieved by our algorithm with previous proposals for rendering different triangle meshes. For reference, the meshes were drawn with back face culling enabled at a resolution of 1000×800, using OpenGL 2.1 with two different NVIDIA GeForce cards: a GTX 295 (with multi-GPU disabled) and a GTX 480. In the first scenario, the meshes were drawn in immediate mode using client arrays, and thus transferring vertex data to the GPU at each frame. In the second scenario, the meshes were drawn in retained mode using vertex buffer objects, thus eliminating the cost of data transferring. Table 2 (immediate mode) and Table 3 (retained mode) summarize the obtained results, expressed in *frames per second* (fps), achieved by the following algorithms:

- *None*: The rendering of the shaded model without wireframe representation, included in the tables for reference.
- *Bias*: The traditional two-pass algorithm using polygon offset; naturally, lighting was disabled for line drawing.
- *GPU*: Our implementation of Bærentzen et al.'s proposal [3] as described in [6], translated to GLSL.
- *Tex*: Our proposal on fixed-function graphics pipeline, duplicating the vertices in a pre-processing phase as described.
- *TexGS*: Our proposal using the geometry shader, without duplicating vertices.

In Tables 2 and 3, for each configuration, the best achieved performance for wireframe drawing is highlighted in bold. As can be noted, our texture-based techniques (the last two columns in the tables) present better performance than previous proposals. In immediate mode (Table 2), the cost of transferring data to the GPU limits the performance. As a consequence, proposals that minimize the amount of vertex data present better performance. In retained mode (Table 3), data transferring no longer limits the performance; as a result, our texture-based approaches perform much better than previous ones.

5 Topologically structured meshes

A nice feature of the algorithm described in Sect. 3.2 is that it results in no vertex duplication when the input is topologically structured triangle mesh, like the ones illustrated in Fig. 10. This kind of mesh is widely used in solid modeling and numerical simulations.

Table 2 Performance in *fps* for rendering different triangle meshes using immediate mode

Model	Card	None	Bias	GPU	Tex	TexGS
Hand	GTX 295	42	21	25	22	**42**
	GTX 480	92	47	**90**	48	**90**
Neptune	GTX 295	36	18	23	17	**36**
	GTX 480	78	40	**77**	40	**77**
Dragon	GTX 295	39	13	13	23	**38**
	GTX 480	52	26	**52**	27	**52**
Manuscript	GTX 295	8	1	3	6	**8**
	GTX 480	12	6	**12**	7	**12**

Table 3 Performance in *fps* for rendering different triangle meshes using retained mode

Model	Card	None	Bias	GPU	Tex	TexGS
Hand	GTX 295	246	36	23	92	**168**
	GTX 480	1640	586	266	**901**	595
Neptune	GTX 295	132	33	23	51	**119**
	GTX 480	1218	472	195	**675**	512
Dragon	GTX 295	250	17	12	60	**71**
	GTX 480	980	276	135	**478**	309
Manuscript	GTX 295	63	1	4	**35**	19
	GTX 480	172	32	39	**139**	39

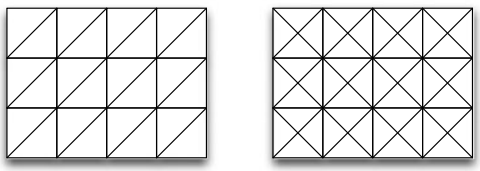
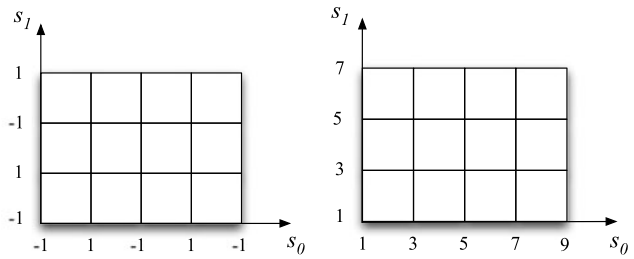


Fig. 10 Topologically structured triangular meshes

Similarly, topologically structured quadrilateral meshes are also widely employed. In this case, it is straightforward to choose non-conflicting texture coordinates for drawing the wireframe. It suffices to alternate the coordinates -1.0 and 1.0 in both directions, each one corresponding to a texture unit, as illustrated in Fig. 11a. However, for such meshes, it is advantageous to set the texture coordinates based on a global reference system. Looking at the topological grid, the texture coordinates of each row or column of edges are set to consecutive odd values, as illustrated in Fig. 11b. The use of a global reference system allows us to interpolate texture coordinates, making it possible to render the wireframe decoupled from the underlying mesh used to represent the model, which has important applications.

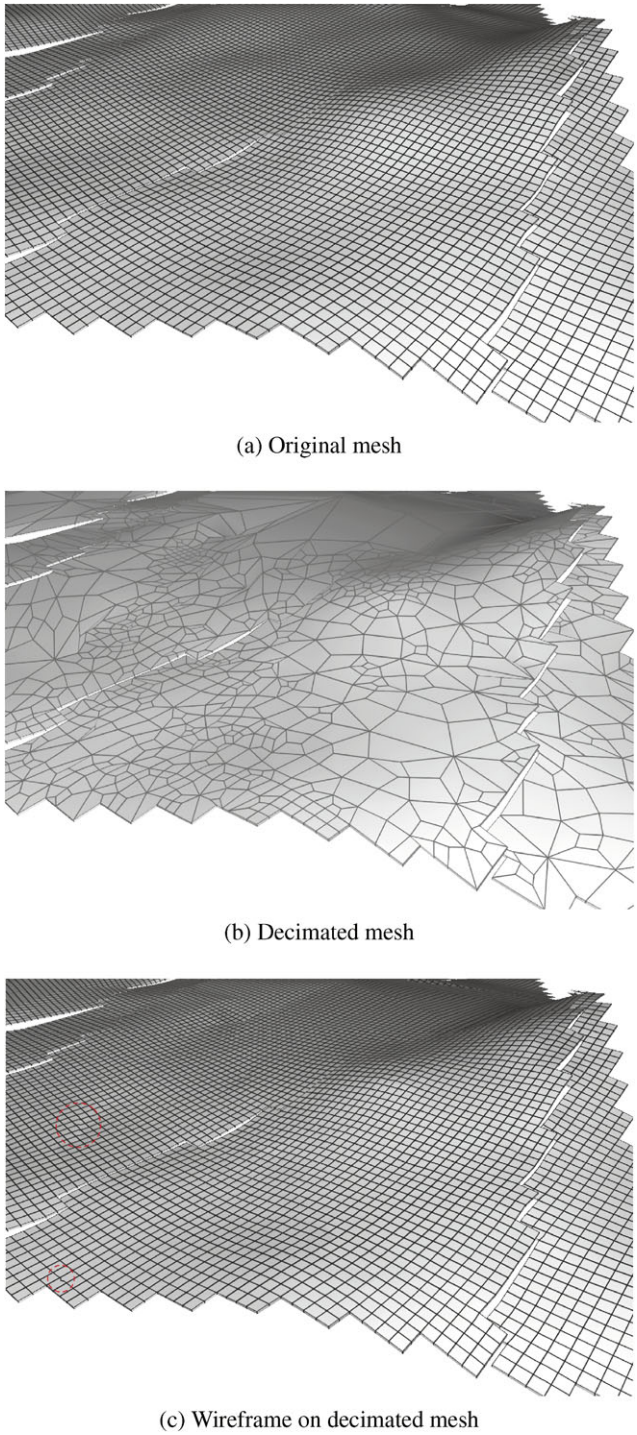


(a) Local system (b) Global system

Fig. 11 Topologically structured quadrilateral meshes

For example, in the oil industry, black oil reservoir numerical simulations are used to predict and plan field exploration. A black oil reservoir model is, in general, defined by a discrete mesh of hexahedral cells laid on a topological grid. Each cell is identified by a triple $[i, j, k]$. The geometry is usually irregular, with discontinuities characterizing geological faults. The surface of a *layer* is composed by the top faces of all cells of a given k, thus being represented by a topologically structured quadrilateral mesh, as illustrated in Fig. 9.

To achieve numerical accuracy, black oil reservoir simulations require highly discretized models. Visualization tools then face the challenge of rendering massive models efficiently. We have investigated a level-of-detail scheme for reservoir visualization. For wireframe rendering, the original model texture coordinates are given according to the global reference system just described. During mesh decimation, we apply simplification operators, creating new vertices. The texture coordinates assigned to these new vertices are simply computed by interpolation. We are then able to render the original wireframe on top of simplified versions of the model. Figure 12 illustrates the achieved results: Fig. 12a shows the original mesh with its corresponding wireframe; Fig. 12b shows a decimated mesh, and Fig. 12c shows the original wireframe rendered on the top of the decimated mesh. As can be noted, the achieved image is quite close to the original model, despite some, almost unnoticeable, distortions on the rendered wireframe.



(a) Original mesh

(b) Decimated mesh

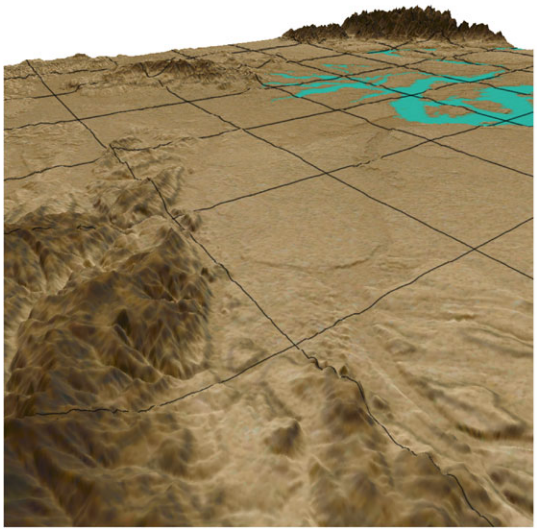
(c) Wireframe on decimated mesh

Fig. 12 Rendering of wireframe decoupled from the underlying mesh. In (**c**), small distortions on the wireframe can be noted in the regions marked by *red circles*

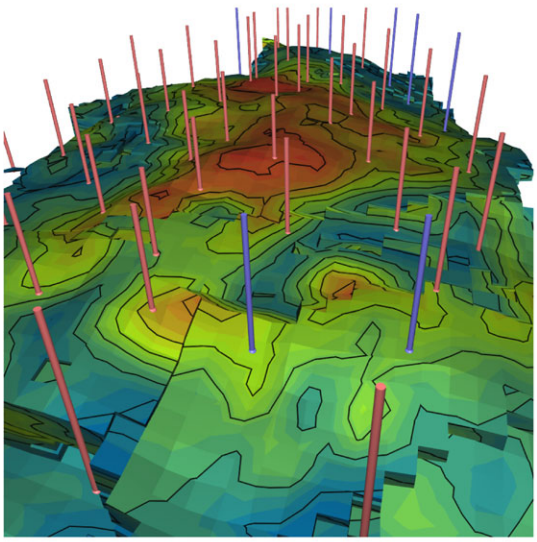
6 Lines decoupled from geometry

Our technique can also be used for rendering non-wireframe lines decoupled from the mesh geometry. All we need is to generate the appropriate texture coordinate for mapping the wireframe texture.

As a first example, consider the display of regularly spaced grid lines on the top of a terrain model, as illustrated in Fig. 13a. The grid lines are drawn by projecting the wireframe texture onto the terrain surface. Two texture units are used, one for drawing each set of parallel lines. The texture coordinates are simply determined by enabling appropriate automatic texture coordinate generation in object space. Another example of lines decoupled from the geometry is illustrated in Fig. 13b: a black oil reservoir model is rendered with the isolines of a given scalar field. In this case, it suf-



(a) Grid lines over curved surface



(b) Isolines of scalar field

Fig. 13 Rendering of lines decoupled from mesh geometry

fices to use one texture unit and to set the value of the scalar field at each vertex as its texture coordinate, with an appropriate texture matrix to control line spacing.

Haeberli and Segal [7] had already indicated the use of an unidimensional texture for drawing lines decoupled from the mesh. The challenge, however, is to preserve line thickness. As the lines are decoupled from the mesh geometry, texture magnification may be needed even for small primitives in screen space. To illustrate the problem, consider the example of drawing the isolines of a scalar field. If the gradient of the scalar field is large, the use of mipmapping, as proposed here, ensures that line thickness is preserved (see Fig. 14a and b). On the other hand, if the gradient is too small, texture magnification is called for and lines become thicker, as illustrated in Fig. 14c.

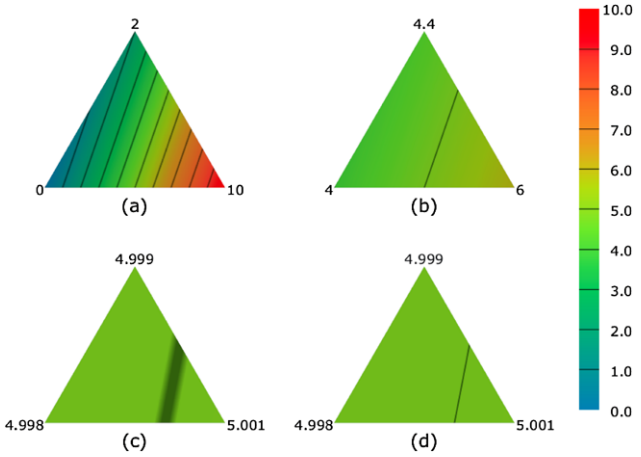


Fig. 14 Isoline thickness variation: (**a**), (**b**) for large gradients, the mipmapping preserves line thickness; (**c**) for small gradients, texture magnification may occur; (**d**) a geometry shader can be used to avoid texture magnification

```
void adjust (inout float s0, inout float s1, inout float s2)
{
  float M = max(s0, max(s1, s2));
  float m = min(s0, min(s1, s2));
  float fM = floor(M);
  float fm = floor(m);
  if ((fM - fm) == 1) {
      float range = M - m;
      s0 = (s0 - fM) / range + fM;
      s1 = (s1 - fM) / range + fM;
      s2 = (s2 - fM) / range + fM;
  }
}
```

Fig. 15 Geometry shader to avoid texture magnification for isoline rendering on triangle primitives

If geometry shaders are available, with our approach, we can easily (and efficiently) preserve line thickness for small gradients. The problem arises when we have a small range of texture coordinate values over a primitive. To fix it, we can use a geometry shader that, for each primitive, identifies whether the range of values encloses a single isoline. In such a case, it adjusts the texture coordinates by applying a scale around the isoline value, adjusting the range to 1.0 and thus mapping the entire texture on the primitive. Figure 15 shows the corresponding GLSL function that adjusts the three texture coordinates of triangle primitives, thus avoiding texture magnification for isoline rendering. The achieved result is illustrated in Fig. 14d.

7 Conclusion

In this paper, we present a texture-based approach for wireframe rendering. As demonstrated, our method is efficient, versatile, easy to implement and integrate into legacy graphics codes, and still produces high-quality, anti-aliased images.

For topologically structured meshes, we have identified that our approach allows the rendering of wireframe decoupled from the underlying mesh used to represent the model; for instance, we can draw the original wireframe on the top of decimated meshes. Our approach can also be used for rendering lines other than wireframes, such as the isolines of a scalar field, still preserving line thickness.

Acknowledgements We thank CNPq (Brazilian National Research and Development Council) for the financial support to conduct this research. We also thank the support provided by the Tecgraf/PUC-Rio laboratory, which is mainly funded by the Brazilian oil company, Petrobras.

Some models used on this research were made available by Oyonale, AIM@SHAPE, and the Stanford 3D Scanning repositories.

References

1. Akenine-Möller, T., Haines, E., Hoffman, N.: Real-Time Rendering, 3rd edn. AK Peters, Natick (2008)
2. Bærentzen, J.A., Munk-Lund, S., Gjøl, M., Larsen, B.D.: Two methods for antialiased wireframe drawing with hidden line removal. In: Proceedings of the Spring Conference in Computer Graphics (2008)
3. Bærentzen, J.A., Nielsen, S.L., Gjøl, M., Larsen, B.D., Christensen, N.J.: Single-pass wireframe rendering. In: SIGGRAPH'06: ACM SIGGRAPH 2006 Sketches, p. 149. ACM, New York (2006)
4. Bærentzen, J.A., Nielsen, S.L., Gjøl, M., Larsen, B.D., Christensen, N.J.: Single-pass wireframe rendering. Movie presentation (2006). http://portal.acm.org/citation.cfm?id=1180035
5. Celes, W., Abraham, F.: Texture-based wireframe rendering. In: Graphics, Patterns and Images, 23rd SIBGRAPI Conference (SIBGRAPI), 2010, pp. 149–155 (2010). doi:10.1109/SIBGRAPI.2010.28
6. Gateau, S.: Solid wireframe. NVIDIA White Paper (2007)
7. Haeberli, P., Segal, M.: Texture mapping as a fundamental drawing primitive. In: Cohen, M., Puech, C., Sillion, F. (eds.) Fourth EUROGRAPHICS Workshop on Rendering, pp. 259–266 (1993)
8. Herrell, R., Baldwin, J., Wilcox, C.: High-quality polygon edging. IEEE Comput. Graph. Appl. **15**(4), 68–74 (1995)
9. Kuschfeldt, S., Holzner, M., Sommer, O., Ertl, T.: Efficient visualization of crash-worthiness simulations. IEEE Comput. Graph. Appl. **18**(4), 60–65 (1998). doi:10.1109/38.689666
10. Rose, D., Ertl, T.: Interactive visualization of large finite element models. In: Workshop on Vision, Modelling, and Visualization VMV'03, pp. 585–592. Akad. Verlagsgesellschaft, Berlin (2003)
11. Tang, C., Li, S., Wang, G., Zang, Y.: Stable stylized wireframe rendering. Comput. Animat. Virtual Worlds **21**(3–4), 411–421 (2010). doi:10.1002/cav.370
12. Wang, W., Chen, Y., Wu, E.: A new method for polygon edging on shaded surfaces. J. Graph. Tools **4**(1), 1–10 (1999)

Waldemar Celes is a researcher professor in the Computer Science Department at PUC-Rio, Brazil, and a former postdoctoral associate at the Program of Computer Graphics, Cornell University. His current research interests in computer graphics include real-time rendering, scientific visualization, physical simulation, and distributed graphics applications. He is also one of the authors of the Lua programming language.

Frederico Abraham is a Ph.D. student in the Computer Graphics Program at the Computer Science Department at PUC-Rio, Brazil. He received his B.S. and M.S. degree in computer science from the same university. His current research interests include real-time out-ofcore rendering, scientific visualization, and distributed visualization. He is a student member of Eurographics.