STAG: Smart Tools & Apps for Graphics (2014)
Andrea Giachetti (Editor)

Practical line rasterization for multi-resolution textures

Javier Taibo[1], Alberto Jaspe[2], Antonio Seoane[1], Marco Agus[2] and Luis Hernández[1]

[1]University of A Coruña, Spain
[2]Visual Computing Group - CRS4, Italy

Abstract

Draping 2D vectorial information over a 3D terrain elevation model is usually performed by real-time rendering to texture. In the case of linear feature representation, there are several specific problems using the texturing approach, specially when using multi-resolution textures. These problems are related to visual quality, aliasing artifacts and rendering performance.

In this paper, we address the problems of 2D line rasterization on a multi-resolution texturing engine from a pragmatical point of view; some alternative solutions are presented, compared and evaluated. For each solution we have analyzed the visual quality, the impact on the rendering performance and the memory consumption. The study performed in this work is based on an OpenGL implementation of a clipmap-based multi-resolution texturing system, and is oriented towards the use of inexpensive consumer graphics hardware.

1. Introduction

It is very common in interactive 3D GIS environments to project 2D vectorial information (non-protruding features) over 3D geometric terrain models. There are important issues in dealing with this data, especially when they have to be integrated in visualization systems for massive terrain models [PG07]. In general terms, there are several alternatives for rendering this data, mainly in two opposite directions. The first one is to create the geometric primitives for the 2D information draped over the 3D surface, as described in [WKW*03], [ARJ06] and [SBZ07]. The other approach is to rasterize this 2D information and then apply it to the geometry using the texturing capabilities of the graphics hardware, as suggested by [KD02], [BW05] or more recently by [VTW11]. This work follows the second strategy because of its better ability to adapt to dynamic geometry, as there is no coupling with the terrain engine and because its lower cost with complex vectorial data. This data is rendered through a multi-resolution texturing engine, as described in [TSH09], that constitutes the context in which the proposed techniques are developed, tested and analyzed. In such a system, line primitives present some specific rendering problems that do not exist in other types of primitive, such as points or polygons.

In this paper, these problems are analyzed and described, and several possible solutions are implemented and evaluated. It is not a theoretical study, but a practical approach in the seek of a working implementation that uses the current available consumer graphics hardware in order to obtain the highest quality without an important performance penalty to high demanding interactive applications. The proposed solutions are practical and can be effectively integrated in multiresolution terrain rendering frameworks [GMC*06, YWDF11, LC10, TSH09]. The rest of the paper is organized as follows. Section 2 describes the texturing engine that will be the environment where the proposed techniques are developed, integrated and tested. Section 3 explain the problems found in a naive implementation of the line rendering in the texturing environment. Those problems are faced in section 4 with the development of different alternative solutions. Section 5 give the results for each solutions and some conclusions are presented in section 6.

2. System environment

The working environment is the multi-resolution virtual texturing system described in [TSH*07] and [TSH09], based on the idea of the MIPmap developed by [Cos94] in the global terrain texture and [TMJ98] with the clipmap to bypass the texture size limits imposed by the graphics hardware, using in this case a 3D texture [TSH09]. The levels of detail (LOD) of the virtual texture are divided in tiles that feed the cached subset of the whole texture pyramid, i.e. the clipmap. In this case, the work is focused on vectorial information, so the

DOI: 10.2312/stag.20141234

texture tiles are not loaded, but generated by rendering to texture the contents of a 2D scene graph.

The quality of the texture filtering is specially important in this case, as thin lines rendered over the terrain are a difficult test for avoiding aliasing. In close-to-horizontal views, those thin lines are very noticeably overblurred by trilinear filtering as they get further from the viewer, so the use of anisotropic filtering in the texturing engine is critical to the final image quality. The vectorial information is stored and managed in a 2D scene graph, that is rendered to texture using an orthographic camera adjusted to the size and position of each texture tile and LOD to render. The vectorial contents are organized as layers in the scene graph and rendered in the user defined order, with no depth test. This is important when overlapping semitransparent information is rendered, as the final view is determined by how the layers are sorted.

3. Description of the problem

When 2D lines are draped over the 3D terrain model in a perspective view, the vectorial line data rendered in the different texture levels of detail of the MIPmap pyramid (or clipmap) must be coherent in its perceived width, to avoid undesirable artifacts in the visualization. In a 2D visualization this is not a problem because the scale is uniform across all the view, and it is merely a problem of cartographic generalization [MS92]. But in a 3D visualization, whether a perspective or orthographic projection, as the camera is tilted from a zenithal view towards a horizontal one, different scales or texture resolutions are used along the screen. Using the same line width in texels for every texture level will produce discontinuities in the perceived line width where texture LODs change and an incorrect visualization of thinner nearer lines and thicker further ones (see Figure 1).

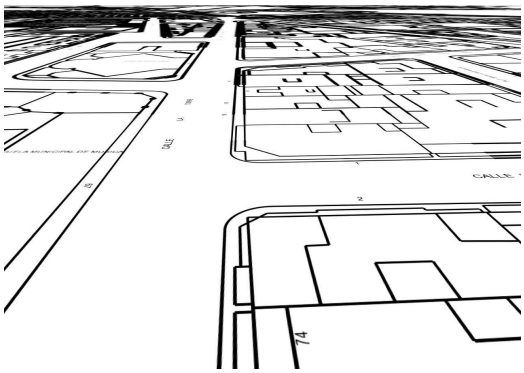

Figure 1: *Constant line width (5 texels) in texture LOD space.*

This is solved by assigning a fixed line width in world space (instead of texture level space) that will derive in different widths for each pyramid level. Moreover, the use of real units, such as meters, is of great help to the operator at the time of assigning width to the different GIS layers and objects. Nevertheless, this world space width does not need to be constant. It can vary as a function of the distance between camera and terrain, implementing a mechanism similar to cartographic generalization, but extended to a 3D view to keep the coherence of the perceived line width between texture LODs. For the rendering of the texture tiles, the straightforward approach is to render to texture the lines using the OpenGL line primitive types. At the beginning of the processing of each pyramid level, the established world space width is translated to the texture space in this level of detail, so the line can be correctly rendered.

To reduce the line aliasing in these renders, OpenGL offers two possibilities: the use of `GL_LINE_SMOOTH` mode with alpha blending or the use of multisampling. The problem with the latter is that the render to texture is done using a frame buffer object (FBO) with the texture attached as render target. Even though OpenGL has defined an extension to use multisampling with FBOs (`EXT_framebuffer_multisample`), they are meant to be used with renderbuffer objects, but no method is provided for creating multisample texture images. So, the only practical option provided by OpenGL for antialiasing the lines rendered to texture is alpha blending.

Using the programming capabilities of graphics hardware, there are other choices for line antialiasing, such as the one explored by [CD05], that implements in GPU the algorithm of [MMJ00]. It is based on a fragment shader, where lines are filtered using a precomputed array of intensity values. This allows us to use sophisticated filters without the corresponding time penalty, as runtime performance is independent of the complexity of the filter. The next problem imposed by OpenGL implementation in current consumer graphics hardware is that the line width is limited, clamping it when the finer levels of detail (that correspond to the thicker line widths) are used, as shown in Figure 2. As will be shown later, more than a problem, this clamping is actually an advantage, specially with line smoothing, because the thinner lines fade out in a nice way. This kind of restrictions, specially in low-end hardware, cannot be overcome, but there are some possible workarounds that will be analyzed in this paper. Those solutions are presented next, beginning with the simpler one and progressively solving the problems that arise. Some solutions are built upon previous ones in the quest for a final solution that solves the discovered problems with an affordable cost.

4. Proposed solutions

4.1. Solution 1. Discard nearest line segments

The first proposed solution to the problem of line width clamping because of hardware limits is to discard oversized line portions (the nearest ones) making them progressively fade away in the finest texture LODs. It can be thought of

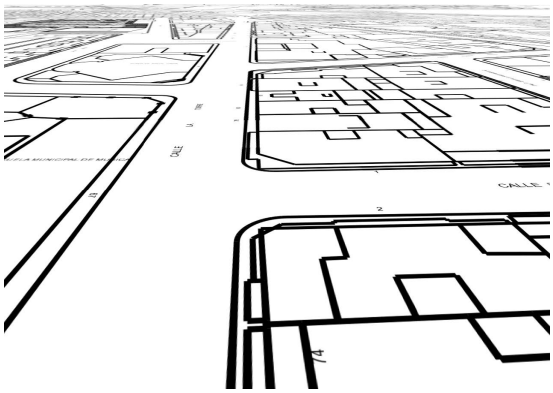

Figure 2: *World space line width (0.5 meters) clamping because of hardware limits.*

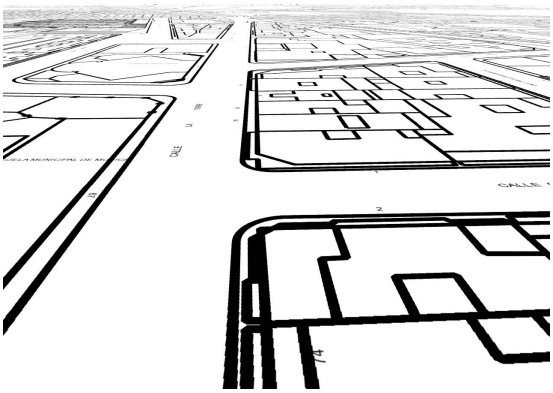

Figure 3: *Tessellation of the lines into polygon meshes. The width coherence is kept specially in the nearest segments.*

as a kind of cartographic generalization, where objects that are too big for the visualization scale are removed. This trivial solution is efficient because of its simplicity, and works pretty well for thin lines or those thicker ones that are not too close to the viewer. Activating the line smoothing with alpha blending gives a nice fade in the distance for the furthest lines. However, the disappearing of the nearest (or thicker) lines is unacceptable for most applications, because the hardware limit for the maximum line width is quite low (specially in low-end hardware) and that provokes the lines to disappear too soon. It is this upper limit which makes this technique unacceptable for a generic method implementable in all kind of graphic hardware.

4.2. Solution 2. Tessellate the lines into polygons

The second solution, an alternative to the use of OpenGL line primitives that suffer the width limit, is to tessellate the lines to convert them to polygon meshes, that can be scaled to any world space line width, keeping the coherency in the perceived line width for all the texture LODs. Figure 3 shows how this coherence is perfectly kept no matter how close the camera is or how wide the lines are.

This width coherence, that can be clearly seen in the pictures, is much more important in the interactive visualization when the camera is in motion. When moving at speeds that do not allow the texture cache to keep fully updated, the visual quality of the line primitive based solution is even more unacceptable because, apart from the loss of width coherence, there are width changes in one same view. As the rasterized texture drops quality or refines it, there are very noticeable width changes in the image, because each change in width can be up to twice (or half) the previous one.

The following sections describe different alternatives to perform this tessellation. Depending on the case we use different methods, combining them all to construct the final geometry.

4.2.1. Method 1. Isolated line segments

We begin the description of the tessellation process with the simplest case, where there is an isolated line segment. In this case, each vertex of the line is replaced by two new vertices, equidistant to the original vertex at both sides of the line segment along its perpendicular and with a distance between them equivalent to the desired line width (see Figure 4). For one line segment, these vertices can be assembled in different OpenGL primitives, like triangles, quadrilaterals, polygons, triangle strips or quadrilateral strips.

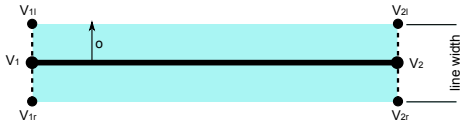

Figure 4: *Trivial conversion of a line segment to a polygonal area.*

The offset $\mathbf{o}$ from the original vertices $\mathbf{v_1}$ and $\mathbf{v_2}$ is computed as stated in the following equations, using the "perp" operator given by [Hil94] and considering w as the line width.

$$\mathbf{v} = \mathbf{v_2} - \mathbf{v_1} = (v_x v_y)$$
$$\mathbf{v}^{\perp} = (-v_y v_x)$$
$$\mathbf{o} = \frac{w\mathbf{v}^{\perp}}{2|\mathbf{v}^{\perp}|}$$

Then the new vertices are computed applying the offset to the original vertex in both directions.

$$\mathbf{v_{1l}} = \mathbf{v_1} + \mathbf{o} \;\; ; \;\; \mathbf{v_{1r}} = \mathbf{v_1} - \mathbf{o}$$
$$\mathbf{v_{2l}} = \mathbf{v_2} + \mathbf{o} \;\; ; \;\; \mathbf{v_{2r}} = \mathbf{v_2} - \mathbf{o}$$

4.2.2. Method 2. Connected line segments (corner)

The case of a single isolated line is the simplest case. But lines are more frequently found in the form of sets of connected line segments, usually called polylines or more precisely, in OpenGL, line strips or line loops, depending on whether they are open or closed, respectively. These line sets are more efficient representations, as the vertices are shared between every two consecutive line segments. The only vertices that are not shared are the first and last ones, in the case of open polylines. Treating every line segment of the polyline independently, as described above, will result in a rather unpleasant, unconnected final appearance, as shown in Figure 5.

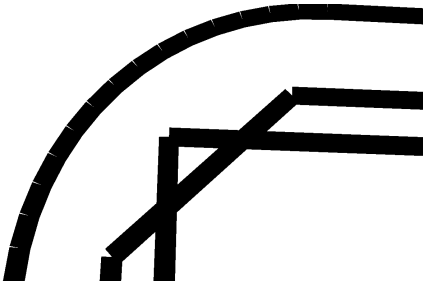

Figure 5: *Unconnected polyline.*

For the sake of both efficiency and continuity, only one vertex should be considered for the connection between two line segments. The unfolding of this vertex with the previous algorithm has two choices, one for each line segment, though neither will offer a satisfactory result. The vertices of the connection between two line segments must be computed taking into account both segments, and not just one, as previously described. For a situation like the one depicted in Figure 6, a fairly good solution is shown. We consider the lines parallel to each segment at both sides with a distance of half the desired line width. Those parallel lines enclose the polygonal area that represent the original line, as in the case for one isolated line segment. To differentiate and recognize both sides of the line segment and more concretely those two parallel lines (and the associated vertices), we have named one the *left* line (suffix *l*) and the other the *right* line (suffix *r*), considering the forward direction to be from $\mathbf{v_1}$ to $\mathbf{v_2}$.

The intersection of both the left and right line segment pairs are computed as described in [Bou89], and the resulting intersecting points are used as the vertices for the tessellation of the lines ($\mathbf{v_{2l}}$ and $\mathbf{v_{2r}}$). As the line segments are connected and their vertices shared, the most efficient OpenGL polygonal primitives to draw them are triangle strips (`GL_TRIANGLE_STRIP`) or quadrilateral strips (`GL_QUAD_STRIP`). The tests run in our implementation showed that there is no noticeable rendering performance

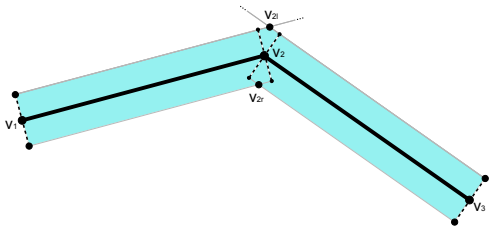

Figure 6: *Connection between two line segments.*

difference between the use of both primitive types. Actually, the rendering performance of the triangle or quadrilateral strips is quite similar to the lines', even though the number of vertices is doubled. The sequence of vertices is the same for both of them, being the straightforward way of alternating left and right vertices:

$$\mathbf{v_{1l}}, \mathbf{v_{1r}}, \mathbf{v_{2l}}, \mathbf{v_{2r}}, \mathbf{v_{3l}}, \mathbf{v_{3r}}, \cdots$$

4.2.3. Method 3. Connected line segments (bevel)

The joint of line segments through the intersection of its boundary lines, as just described in Method 2, works well for wide angles, such as in the example of Figure 6. However, in situations where there is an acute angle between line segments (Figure 7), the outer lines intersection can be located at a far distance from the original vertex. As the angle between segments becomes smaller, the distance to the intersection point grows towards infinite (representing the extreme case of parallel lines where there is actually no intersecting point).

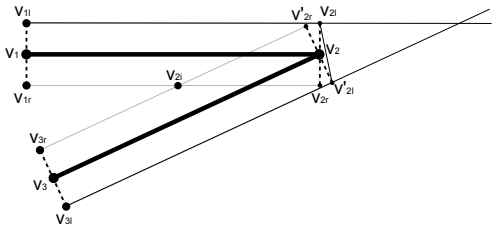

Figure 7: *Connection between two line segments joining in an acute angle.*

Because the outer vertices of acute angles are far away from the original vertices and so severely distort the real shape of the polyline, we define an angle threshold below which no outer intersections are computed. Instead, the corner is beveled, connecting the two outer vertices of the segments when considered isolated ($\mathbf{v_{2l}}$ and $\mathbf{v'_{2l}}$ in Figure 9). The angle threshold chosen is usually close to 90°, but can be any other value; its selection follows mainly aesthetic reasons, so there is not a unique valid angle. This tessellation method proposed for acute angles also works quite well for obtuse ones, that in this case are equally beveled. However,

the contrary does not work out, as Method 2 is not adequate for narrow angles (it produces exaggerated long pikes). For this reason, although the threshold can be an acute angle, it should not be much smaller than 90°.

The use of Method 3 somehow breaks the sequence of primitives based on vertex pairs, because the outer vertex ($\mathbf{v_{2o}}$) in this case is actually split into two vertices ($\mathbf{v_{2l}}$ and $\mathbf{v'_{2l}}$ in Figure 9). In order to tessellate these vertices there are several ways of overcoming this problem. The only immutable restriction is to keep the first and last pairs of vertices in the correct order (left-right, subindices *l* and *r*) so they connect perfectly with the adjacent segments.

The first solution (Method 3a) implicates the same computations as in the case of an isolated line segment (described previously), where $\mathbf{v_{2l}}$ and $\mathbf{v_{2r}}$ are the vertices corresponding to the middle vertex $\mathbf{v_2}$ for the first segment ($\mathbf{v_1}$ to $\mathbf{v_2}$), and $\mathbf{v'_{2l}}$ and $\mathbf{v'_{2r}}$ the vertices corresponding to that same middle vertex for the second segment ($\mathbf{v_2}$ to $\mathbf{v_3}$) (see Figure 7). The vertices are then tessellated in the following order:

$$\mathbf{v_{1l}}, \mathbf{v_{1r}}, \mathbf{v_{2l}}, \mathbf{v_{2r}}, \mathbf{v'_{2l}}, \mathbf{v'_{2r}}, \mathbf{v_{3l}}, \mathbf{v_{3r}}, \dots$$

This solution appears different whether triangle strips (Figure 8a) or quadrilateral strips (Figure 8b). With the shown difference in the boundary of the polygonal area, both cases work perfectly with opaque geometry, but if the lines are drawn with some transparency level, some parts of the polygonal area are covered several times, and so appear denser than others. The overlapping areas are shown in a darker color in the figure.

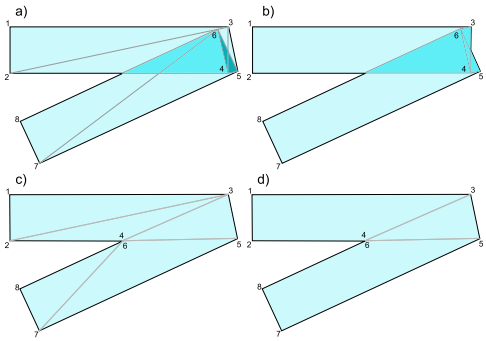

Figure 8: *Alternatives for the tessellation of two line segments joining in an acute angle.*

Method 3b avoids overlapping areas using a new vertex located in the intersection point between the two inner boundaries of the line segments, represented as $\mathbf{v_{2i}}$ in Figure 7. This inner intersection point can be located on the left or right side of the line, depending on the turning direction of the polyline in this joint. If the polyline turns left, the inner point will be on the left side and if it turns right (as in the example of Figure 7), the inner point will be on the right side. The sequence of vertices for this tessellation is the following for a right turn:

$$\mathbf{v_{1l}}, \mathbf{v_{1r}}, \mathbf{v_{2l}}, \mathbf{v_{2i}}, \mathbf{v'_{2l}}, \mathbf{v_{2i}}, \mathbf{v_{3l}}, \mathbf{v_{3r}}, \dots$$

In case of a left turn, the vertex sequence will be as follows:

$$\mathbf{v_{1l}}, \mathbf{v_{1r}}, \mathbf{v_{2i}}, \mathbf{v_{2r}}, \mathbf{v'_{2i}}, \mathbf{v_{2r}}, \mathbf{v_{3l}}, \mathbf{v_{3r}}, \dots$$

These vertex sequences are adequate for both triangle strips (Figure 8c) and quadrilateral strips (Figure 8d). Vertex $\mathbf{v_{2i}}$ is repeated twice to adapt to the tessellation of those primitives, so the triangle 4-5-6 is degenerated to a line and the quadrilateral 3-4-5-6 is degenerated to a triangle, because vertices 4 and 6 are actually the same.

With this tessellation method that uses the inner intersection $\mathbf{v_{2i}}$, a possible problem may arise when the line width is high in comparison to its length: the line segment formed by $\mathbf{v_{1s}}$ and $\mathbf{v_{2s}}$ and the one formed by $\mathbf{v'_{2s}}$ and $\mathbf{v_{3s}}$ (being *s* the inner side, *l* or *r*) may not intersect. In these cases, a possible solution is to fall back to Method 3a. However, such a situation of extreme proportions between length and width of the line segments should not happen, and if it does, it is more a problem of line simplification, a process previous to the rendering, that lies beyond the present work.

4.2.4. Combination of the tessellation methods

The different methods described above are combined to construct the final geometry to render the lines. The objective is to choose in each case the method that achieves the best visual quality and rendering performance. Tessellation is an iterative process executed once per vertex of the original polyline. For an open polyline, the first and last vertices are managed by the technique described first for isolated line segments (Method 1). The remaining vertices, and the first and last if they are connected, are managed as described in Algorithm 1, combining tessellation methods 2 and 3b. This algorithm assumes that previous vertices are already given in the correct order ($\mathbf{v_{1l}}$ and $\mathbf{v_{1r}}$), and leaves the vertex sequence in a state that connects correctly with the next iteration. This next iteration will supply vertices $\mathbf{v_{3l}}$ and $\mathbf{v_{3r}}$ or a compatible vertex sequence that tessellates correctly with the primitive type used (triangle strip or quadrilateral strip). The function *isect* implements the algorithm described in [Bou89] that computes the intersection point between the two line segments, passed as their two end points.

The algorithm described has been simplified for the sake of clarity. There are some special situations that must be tested, especially when two consecutive segments are collinear or their angle of incidence is below a tolerance value ε. Otherwise, there will be mathematical precision issues when computing the intersections, that will produce incorrect results resulting in noticeable visual artifacts.

4.2.5. Final issues about line tessellation

When working with world space line width and lines converted to polygons, there is no need to worry about which

Algorithm 1 Tessellation of a connected line segments' joint

$s_1 \leftarrow v_2 - v_1$
$o \leftarrow normalize(-s_1.y, s_1.x) * w/2.0$
$s_2 \leftarrow v_3 - v_2$
$o' \leftarrow normalize(-s_2.y, s_2.x) * w/2.0$
$ang \leftarrow angle(-s_1, s_2)$
if $ang < threshold$ **then**
  if turn left **then**
    $v_{2i} \leftarrow isect(v_1 + o, v_2 + o, v_2 + o', v_3 + o')$
    $newvertices \leftarrow v_{2i}, v_2 - o, v_{2i}, v_2 - o'$
  else
    $v_{2i} \leftarrow isect(v_1 - o, v_2 - o, v_2 - o', v_3 - o')$
    $newvertices \leftarrow v_2 + o, v_{2i}, v_2 + o', v_{2i}$
  end if
else
  $nv_1 \leftarrow isect(v_1 + o, v_2 + o, v_2 + o', v_3 + o')$
  $nv_2 \leftarrow isect(v_1 - o, v_2 - o, v_2 - o', v_3 - o')$
  $newvertices \leftarrow nv_1, nv_2$
end if
return $newvertices$

texture LOD is being rendered to adjust the line width correspondingly. The only modification needed is when the world space line width is changed, usually to see the 3D view at a different scale. In this case, the whole scene graph must be traversed, relocating the vertices in the right place, and then texture tiles must be regenerated.

The cost of this process varies depending on the tessellation method used. Method 3a works correctly by simply scaling each pair of vertices considering their middle point as the scaling center. Method 3b cannot be resolved this way, so the information of the original lines must be kept to retessellate from it with the new line width in a slower process. The computing times of both alternatives are analyzed later in section 5. Implementation and thorough testing of the different approaches described above for the tessellation of lines into polygonal areas has proved that artifacts in near lines are solved and quality is quite good. The main advantage is that lines can be rendered without width restrictions. However, the look of these polygonal areas in far distances is very poor. Figure 9 shows a comparison of the same view with the same line width using line primitives (left) and polygon primitives (right). In the images we can see that aliasing is a huge problem in the latter, while the former shows nice smooth lines.

4.3. Solution 3. Combination of lines and polygons

As described above, neither solution 1 nor solution 2, are able to completely solve all the problems of line rendering for the multi-resolution texturing engine. Each solution solve the problems detected in the other, but none can solve the whole problem. Reaching this point, the evident approach is to combine both solutions, that were *a priori* mutually exclusive, in order to select one or the other depending on the apparent width of the lines for each texture LOD. The proposed strategy is to compute, at the beginning of the update phase for each texture LOD, the line width for every GIS layer containing line primitives.

Algorithm 2 Combination of solutions 1 and 2

for all texture LOD **do**
  for all line set **do**
    compute apparent line width (in texels)
    if apparent width $<$ threshold **then**
      select lines version
    else
      select polygons version
    end if
    render texture cache tiles
  end for
end for

This combination of solution 1 and 2, following Algorithm 2, can be done in two ways, that will be described below. In both cases, the selection of one method or the other (lines or polygons) is made as a function of the relation of the apparent line width with a preestablished threshold, so lines are used for thin line widths and polygons are used for thick line widths. Based on empirical observations, our choice for this threshold value is usually between 1 and 2 pixels.

4.3.1. Solution 3a. Combined scene graph

The first way to combine both solutions is to duplicate this part of the scene graph to keep in memory both versions (lines and polygons) for each line set. These versions can be chosen with a switch node, available in most (if not all) scene graph engines. This solution is expensive in memory usage, what can make it unfordable in case of huge data sets. The real memory cost is analyzed in section 5. The advantage of this solution is that the original line sets are kept in memory and can be used to regenerate the vertices of the polygonal version when tessellated with Method 3b, described in section 4.2.3 and illustrated in Figure 8 c) and d).

4.3.2. Solution 3b. Solid/wireframe mode

A more economic approach, in terms of memory usage is to keep in the scene graph only the polygonal version and activate the OpenGL line polygon mode (also known as wireframe mode), instead of the default fill (solid) polygon mode, in the situations when the lines are thinner than the preestablished threshold. Even though this solution is much more efficient in memory usage, and very quick and easy to implement, it has severe drawbacks.

First, the fact of not keeping the original line sets makes difficult the interactive change of apparent line width in tessellation method 3b, as described before. Another problem is about rendering quality. With this approach we are not really

Figure 9: *Comparison between line (left) and polygon (right) primitives for rendering of thin lines.*

drawing the original line, but a pair of parallel lines centered in the original with a very small separation between them. This can sometimes produce a noticeable change in the perceived line width. Finally, there is a huge overhead for the render, that will be measured in section 5. The rendering of polygons in wireframe mode is quite expensive compared to the solid mode and to the rendering of lines.

4.4. Solution 4. Geometry shader

None of the previous solutions completely satisfied our needs. Both ways of combining the partial solutions 1 and 2 have important drawbacks, as has been shown. This fourth and final solution proposed tries to join all the benefits of previous solutions, moving the process of tessellating the lines to a geometry shader, keeping in the scene graph stored in main memory only the original polylines.

4.4.1. Geometry shader based solutions

This fourth proposed solution follows the same behavior described in Algorithm 2, drawing the original polylines when line width is below the given threshold and activating otherwise the geometry shader to tessellate nearest (or thicker) lines.

The geometry shader accepts the line primitives as its input and outputs the triangle strips, tessellated following the techniques described in section 4.2. The quadrilateral strips are no longer an option because this kind of primitive is not currently supported by the geometry shaders. If the tessellation is computed by the geometry shader, the line strip primitive cannot be supplied exactly as in the previous versions, but through the `LINE_STRIP_ADJACENCY` primitive type. This way, two additional vertices must be added at the beginning and the end of the line strip. These two additional vertices are not drawn when the geometry shader is not active, so in this case is equivalent to the original line strip. Each line segment in the polyline is processed by the geometry shader taking into account the previous and next vertices in order to correctly tessellate the lines into triangle strips. So a total of four vertices are the input to the geometry shader, and the output vertices that will compose the triangle strip will be between four and six (see Algorithm 1).

This way of processing the polylines is far from optimal when compared to the previous method of tessellating the lines in a preprocess in the CPU. Each line segment joint is processed twice by the geometry shader, once for each of the adjacent segments. Moreover, the kind of computations performed for each joint are not very suitable for the GPU, because there is a considerable number of conditionals to check whether the angle is obtuse or acute, the segments are parallel, the segment turns left or right, or exceptional cases such as divisions by zero in the computation of the lines intersection. These kind of conditional situations break the efficiency of the of the GPU architecture.

For all these reasons, we propose an alternative in between doing the computations and tessellate the polylines in the CPU and doing it in the geometry shader. The tessellation computations (including every conditional) are performed in a one-time preprocess step in the CPU, and as the result of this process the line strip vertices are generated and the offset vectors for each side of the line in every joint are stored as a vertex attribute for this line strip. The offsets for each vertex are 2D vectors so they can be stored in a unique per-vertex attribute (`vec4`). This line strip is not exactly the original one, because in the segment joints tessellated by Method 3b, two vertices are generated in the same position, to keep the

ratio of two triangle strip vertices for each line strip vertex. The position of the two triangle strip vertices are then trivially computed from the position of the line strip vertex by adding the correspondent offset multiplied by the scale factor, that depends on the world space line width and the texture LOD being rendered. The adjacency information is not needed, as each line strip vertex is autonomous to compute the triangle strip vertices. The number of vertices output by the geometry shader in this case is always four.

Following Algorithm 2, the line strip (the original one with potentially some duplicated vertices) is used in far distances, activating for the nearest ones the geometry shader that automatically generates the triangle strip in a very fast operation because of the extreme simplicity of the computations. Both strategies to the generation of the tessellated line were implemented and tested in this work, and the results are described in section 5. The first approach, receiving the line strip with adjacency information and computing the tessellation inside the geometry shader was called solution 4a. The second approach, with precomputed offsets and no adjacency information was called solution 4b. The main advantages of solution 4 are the following ones:

- Low memory consumption compared to the previous best visual quality solution (3a). Only the line strip and offset information is stored in the scene graph, the triangles are generated on the fly.
- Coherent line width between line and triangle primitives, unlike solution 3b, with very little memory overhead
- Free change of line width (the tessellation is computed every time using the line width passed to the shader as a uniform parameter). The use of polygonal meshes needed to recompute its scene graph every time the line width changed.

One of the drawbacks of this solution is that polyline geometry must be isolated in the scene graph in order to apply the geometry shader to it. This geometry separation complicate the most aggressive optimization techniques of the original system. These optimizations were strongly based on the hierarchical organization of the scene graph contents adapted to the tile structure of the texture cache. The geometry corresponding to each texture tile were combined in one unique geometry node (geode) of the scene graph. This increment in the number of nodes of the scene graph causes a higher memory consumption and increase the overhead of the scene graph traversals. It is more noticeable as the scene is more complex. However, this most aggressive optimization is not usually chosen because of other reasons. The 2D vectorial information from GIS is usually structured in layers of homogeneous nature, which are treated as a group for means of changing visual attributes like color, transparency or line width. So the line entities are usually already grouped together in GIS layers, and applying the geometry shader in these cases is straightforward.

Another choice that can affect precision is how to store the vertex offsets. The straightforward way is to store them as 2D vectors in rectangular coordinates, that will be added to the central vertex (after the pertinent scaling). But they can also be stored as polar coordinates, as the magnitude and the angle of the offset vector. This way the numerical errors can be reduced in some situations, at the cost of a little bit of computation in the shader. In both cases, the storage is equivalent (a `vec4` attribute is used).

5. Experimental results

The techniques described in this paper were executed with three different data sets containing 2D line information on NVidia GeForce commodity hardware. Test one contains polygonal data from a cadastral data base. These polygons were visualized only as its boundary lines for the purposes of this study, so they are all line loops. The layer contains about 150 000 polygonal features composed by more than 1 250 000 line segments distributed along the coast line of a geographical region of about 20 000 km^2. This data set is a relatively dense layer with an uneven geographical distribution. Test two is the hydrology layer of the region, so in this case there are open polylines corresponding to rivers. The layer contains about 1 220 000 line segments and 1 240 000 vertices distributed along a region of about 45 000 km^2. Unlike previous data set, this layer is quite homogeneous in the line distribution. Test three is the pathway information of a province, containing mostly open polylines. The layer contains about 11 120 000 line segments and 14 586 000 vertices distributed along a province of about 19 500 km^2. They are quite evenly distributed along the covered area. This data set is meant to be a test to the behavior of the system with a huge amount of vectorial information. All These vectorial layers were rendered to a virtual texture of $2^{20} \times 2^{20}$ texels (1 teratexel) with a tile size of 256×256 texels.

5.1. Line width change

The world space line width change is an operation whose time is negligible when using line primitives (solution 1), because only the call `glLineWidth` is needed. It is also negligible in the fourth solution, the geometry shader based one, because the line tessellation is performed every frame. Only the uniform scale variable must be set. The polygon mesh based solutions (2 and 3) need to reconfigure the 2D scene graph in order to apply the changes in line width. The cost of this reconfiguration is dependent on which tessellation method is used. Lines tessellated by methods 1, 2 and 3a can be easily be reset to a new width just scaling each pair of vertices using its middle as pivot point, as described above. This operation takes 1.29 ms for the test 1 data set. In case of using Method 3b, needed to solve the overlapping artifacts when rendering semi-transparent lines, the reconfiguration is performed from the original vertices in a more expensive operation, that takes about 18 ms for the test 1 data set.

5.2. Memory usage

The memory consumption of the system is a critical issue, as the visualization of several complex layers simultaneously can overflow the available memory of the system. Figure 10 shows a comparison of the memory usage of each solution, in percentage relative to the best case, represented as 100%. Figure 11 shows the same comparison applied to the generation time of the optimized data sets.

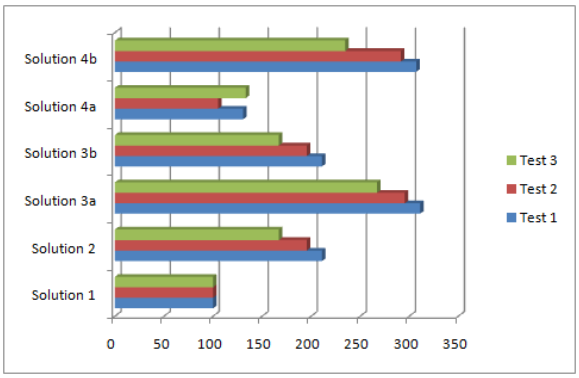

Figure 10: *Memory usage (KBytes) comparison of each solution with the different test data sets (percentage relative to the best case).*

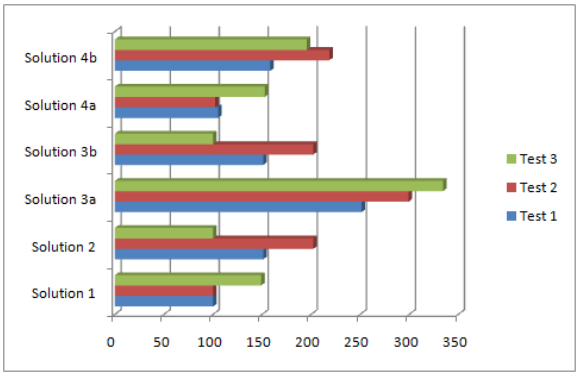

Figure 11: *Generation time (ms) comparison of each solution with the different test data sets (percentage relative to the best case).*

The optimal solution in terms of memory is the first simplest line based solution 1. The worst one is solution 3a, that keeps in memory both lines and polygon versions in a double scene graph choosing one or the other with a switch. In between, solutions 2 and 3 double the memory requirements of line based solution 1, as the number of vertices is doubled (at least) in the line tessellation process described in previous sections.

The geometry shader based solution is very dependent on the fact of using the precomputed offsets (solution 4b) or performing all the tessellation computations inside the geometry shader (solution 4a). The first case has an important memory overhead, because each vertex of the original polyline (`vec2`) is attached the two offsets for the tessellation (compacted in a `vec4`). Moreover, some vertices are duplicated because of the tessellation of acute angles as described in previous solutions. This makes solution 4b very memory consuming, quite close to the worst solution 3a. The heavy geometry shader case has a much lesser memory overhead, but still a little, due to the increase in the number of vertices for two reasons: the vertex duplication of converting line loops to line strips and the addition of the adjacency vertices. It is the least memory consuming solution with the exception of the original lines of solution 1.

5.3. Rendering time

The generation time of the different approaches was measured as the mean of a complete reconstruction of the texture cache (clipmap) repeated along 100 frames to avoid deviations due to the impact of external factors such as operating system activities, accesses to disk, etc. All these measurements showed a pretty stable results, with very little deviations, so the data analyzed in this section can be considered a fairly accurate estimation of the rendering times.

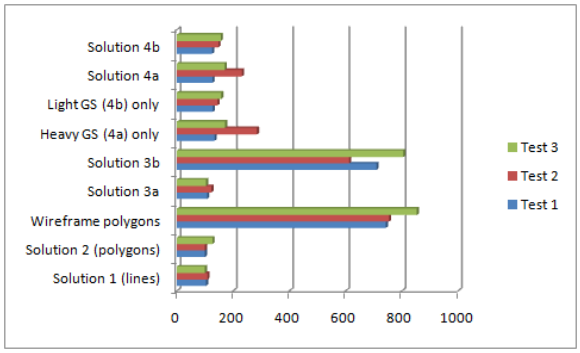

Figure 12: *Rendering times (ms) of the proposed solutions for the three test data sets (percentage relative to the best case).*

Solutions 3a, 3b, 4a and 4b are really a combination of two rendering strategies: the proper one of the solution and the lines for features that appear thin in texture level space. The performance in these cases will be somewhere between the two strategies, and the amount of contribution of each one to the total time will be dependent on many factors such as the scale of visualization, the level of detail of the texture and of course the threshold choice for selecting lines or polygons. For this reason we measured all the rendering strategies in

isolation as well as combined with lines as described in Algorithm 2. The results of the rendering time measurements for the three test data sets are illustrated in Figure 12.

The best solutions in terms of rendering performance are the simplest ones: line based solution 1 and polygon based solution 2. They are pretty close in the first two data sets, being a little faster the polygons version with an overhead time of 4% and 9% respectively. In the third data set the difference is considerably higher (27% overhead) in favor of the lines version. The geometry shader solutions 4a and 4b are quite similar in rendering performance, only the second test data set showed a clear advantage of the light geometry shader (solution 4b). It is interesting to see that the huge computation difference between both geometry shaders is compensated by the transfer and storage of the per-vertex attribute containing the offsets to the lines vertices.

6. Conclusions

Line rendering for a multi-resolution virtual texturing engine as described in [TSH09] has some practical limitations that have been analyzed in this paper. Several solutions have been developed that solve the problems detected. The use of line based primitives has severe drawbacks due to the width limit of the hardware/driver implementation, the incorrect rendering of line continuity and overlapping of semitransparent line segments. The tessellation of these lines into polygons solve the problems found in the first approach. We have explained a way to tessellate this lines preserving the line continuity in the different possible cases of line segment joints. This tessellation methods also take care of the existence of semitransparent lines to avoid overlaps that will result in darker zones in line segment joints. The problem of the polygonal solution is that it causes strong aliasing in thin lines. Solutions 1 (lines) and 2 (polygons) are not usable per-se, as they do not offer a good rendering quality in every level of detail. They are the base for later solutions described in this work.

Analyzing the whole rendering times, it can be concluded that in general terms, the heavy geometry shader (solution 4a) is the best option. It offers the best compromise between rendering performance, memory consumption and visual quality. The rendering time overhead was between 27% and 131% for the test data sets, and the memory overhead was between 5% and 33% for the test data sets of our study. In the case of the geometry shaders not being supported in the available graphics hardware (OpenGL ES devices), a choice must be done between solutions 3a and 3b depending on where the bottleneck is. Solution 3a will offer a higher rendering performance while solution 3b uses considerable less memory.

Acknowledgments. This work is partially supported by the EU FP7 Program under the DIVA (290277) project.

References

[ARJ06] AGRAWAL A., RADHAKRISHNA M., JOSHI R. C.: Geometry-based Mapping and Rendering of Vector Data over LOD Phototex. 3D Terrain Models. In *WSCG'2006* (2006). 1

[Bou89] BOURKE P.: Intersection point of two lines (2 dimensions), 1989. 4, 5

[BW05] BROOKS S., WHALLEY J. L.: A 2D/3D hybrid geographical information system. In *GRAPHITE '05* (New York, NY, USA, 2005), ACM Press, pp. 323–330. 1

[CD05] CHAN E., DURAND F.: *GPU Gems 2. Programming Techniques for High-Performance Graphics and General-Purpose Computation*. Addison-Wesley, 2005, ch. Fast Prefiltered Lines, pp. 345–359. 2

[Cos94] COSMAN M. A.: Global Terrain Texture: Lowering the Cost. In *Proceedings of 1994 IMAGE VII Conference* (1994), The IMAGE Society, pp. 53–64. 1

[GMC*06] GOBBETTI E., MARTON F., CIGNONI P., BENEDETTO M. D., GANOVELLI F.: C-BDAM – compressed batched dynamic adaptive meshes for terrain rendering. *Computer Graphics Forum 25*, 3 (2006), 333–342. Proc. Eurographics 2006. 1

[Hil94] HILL JR F. S.: *Graphics gems IV*. Academic Press Professional, Inc., 1994, ch. The pleasures of "perp dot" products, pp. 138–148. 3

[KD02] KERSTING O., DÖLLNER J.: Interactive 3D visualization of vector data in GIS. In *Proc. of ACM GIS '02* (2002), ACM Press, pp. 107–112. 1

[LC10] LINDSTROM P., COHEN J. D.: On-the-fly decompression and rendering of multiresolution terrain. In *Proc. of 2010 ACM Sig. Symp. I3D* (2010), I3D '10, ACM, pp. 65–73. 1

[MMJ00] MCNAMARA R., MCCORMACK J., JOUPPI N. P.: Prefiltered antialiased lines using half-plane distance functions. In *HWWS '00* (2000), ACM, pp. 77–85. 2

[MS92] MCMASTER R. B., SHEA K. S.: *Generalization in Digital Cartography*. Assoc. of American Geographers, 1992. 2

[PG07] PAJAROLA R., GOBBETTI E.: Survey on semi-regular multiresolution models for interactive terrain rendering. *The Visual Computer 23*, 8 (2007), 583–605. 1

[SBZ07] SCHILLING A., BASANOW J., ZIPF A.: Vector Based Mapping of Polygons on Irregular Terrain Meshes for Web 3D Map Services. In *3rd Int. Conference WEBIST* (2007). 1

[TMJ98] TANNER C. C., MIGDAL C. J., JONES M. T.: The clipmap: a virtual mipmap. In *SIGGRAPH '98* (1998), ACM Press, pp. 151–158. 1

[TSH*07] TAIBO J., SEOANE A., HERNANDEZ L., LOPEZ R., JASPE A.: Hardware-independent clipmapping. *Journal of WSCG 15* (2007). 1

[TSH09] TAIBO J., SEOANE A., HERNANDEZ L.: Dynamic virtual textures. *Journal of WSCG 17*, 1 (2009), 25–32. 1, 10

[VTW11] VAARANIEMI M., TREIB M., WESTERMANN R.: High-quality cartographic roads on high-resolution dems. *Journal of WSCG* (2011). 1

[WKW*03] WARTELL Z., KANG E., WASILEWSKI T., RIBARSKY W., FAUST N.: Rendering vector data over global, multiresolution 3D terrain. In *VISSYM '03: Proc. SDV2003* (2003), Eurographics Association, pp. 213–222. 1

[YWDF11] YALÇIN M. A., WEISS K., DE FLORIANI L.: Gpu algorithms for diamond-based multiresolution terrain processing. In *Proc. of the 11th EGPGV* (2011), Eurographics Association, pp. 121–130. 1