CEIG - Spanish Computer Graphics Conference (2015) - Short paper
Jorge Lopez-Moreno and Mateu Sbert (Editors)

Easy going vector graphics as textures on the GPU

Gustavo Patow[†]

ViRVIG-UdG

Abstract

One common problem of raster images when used as textures is its resolution dependence, which could produce artifacts such as blurring. On the contrary, vector graphics are resolution independent, and their direct use for real-time texture mapping would be desirable to avoid sampling artifacts. Usually, they composite images from layers of paths and strokes defined with different kinds of lines. Here I present a simple yet powerful technique for representing vector graphics as textures that organizes the graphic into a coarse grid of cells, structuring each cell into simple cell-sized BSP trees, evaluated at runtime within a pixel shader. Advantages include coherent low-bandwidth memory access and, although my implementation is limited to polygonal shapes, the ability to map general vector graphics onto arbitrary surfaces. A fast construction algorithm is presented, and the space and time efficiency of the representation are demonstrated on many practical examples.

1. Introduction

Vector graphics always have had a great appealing because of their seamless scaling capabilities. Unfortunately, methods that use them as textures are, in general, hard to implement, require a heavy preprocessing like segmenting the contour and embedding each segment in a triangle [LB05], or allocate unnecessary data and use somewhat intricate encodings [RNCL05, QMK06]. Recently, in [LH06] they introduced an efficient and elegant hashing approach.

However, previous approaches like the one by Lefebvre and co-authors require an implementation that is not immediate in terms of simplicity. So, the implementer is faced with a hard-to take decision: either rely on constant-access raster textures, with all their sampling issues; or use vector-based encodings, with their infinite quality but complex management and implementation issues.

This paper proposes an efficient but simple technique to evaluate vector graphics as textures for real time rendering. This technique is based on the subdivision of a vector image into a grid of cells, each encoding of a BSP tree of simple primitives to check. This brings the advantages of a hash-based system, plus the logarithmic evaluation cost of a tree, which is probably as far as we can get in terms of efficiency if a set of features must be exhaustively evaluated.

[†] dagush@imae.udg.edu

2. Previous work

Textures have been part of the real-time graphics pipeline since its very beginnings. However, the usage of Vector Graphics has not been widely adopted because these algorithms are, in general, hard to implement, and require a heavy preprocessing. For instance, the work by Loop and Blinn [LB05] required segmenting the contour and embedding each segment in a triangle. Other example is the work by Ray et al. [RNCL05] for off-line evaluation vector textures in a Photoshop-like application, or the work by Qin et al. [QMK06] to encode vector glyphs for real-time evaluation. Later, the same authors [QKM07] proposed the use of circular arcs as an approximation primitive for vector texture representations, slightly improving efficiency. Sroila et al. [SEH07] presented a rendering technique for smooth isosurfaces in clip art using implicit surfaces. Their implementation uses an extension of Appel's hidden line algorithm [App67] to solve the visibility problem, without resorting to further acceleration structures.

Lefebvre and Hoppe [LH06] introduced an efficient and elegant hashing approach which stored in each cell the minimal information possible, called perfect spatial hash. Our implementation can be also considered a perfect spatial hash that stores efficiently the vector texture information, and it can be proven that it is really not possible to store information more efficiently. Tumblin and Choudhury [TC04], Tarini and Cigoni [TC05] and Sen [Sen04] also used the concept of a spatial hash, but in their implementations each

DOI: 10.2312/ceig.20151204

sample kept extra information to set embedded geometric boundaries. However, that information only offers a set of fixed structures, in the geometrical sense, thus limiting the quality of the final embedded detail.

Ramanarayanan et al. [RBW04] proposed the use of Bezier splines to represent sub-texel features, but required a linear evaluation of the list stored in each texel. More recently, Nehab and Hoppe [NH08] presented an encoding similar to ours, also creating a hash texture, but the main difference is that it stored, for each cell of the hash the *commands* to construct such a polygon, resulting in an implementation that required the implementation of a small interpreter inside the fragment shaders. Our BSP-based implementation is much simpler in terms of coding, and, for non-convex polygons, a BSP tree can be shown to be more efficient, specially if it encodes only the information concerning that specific cell.

Diffusion curve images were introduced by Orzan et al. [OBW*08] using the concept of diffusion curves, which consist of curves with colors defined on either side. By diffusing these colors over the image, the final result includes sharp boundaries along the curves with smoothly shaded regions between them. Later, Jeschke et al. [JCW09] extended the application of diffusion curves to render high quality surface details on 3D objects. More recently, Sun et al. [SXD*12] improved this work by introducing *diffusion textures* by formulating the diffusion curves in terms of Green's functions. In these textures, diffusion curves have a strict containment relationship, creating a hierarchy that can be evaluated efficiently in CUDA. On the contrary, our implementation allows layers to be freely stacked on top of each other, without containment restrictions.

3. Overview

The technique to store and evaluate the vector texture in the GPU consists of a preprocessing stage and a runtime one. The preprocessing stage, in turn, encodes the vector texture in a grid of cells, and then converts each polygon of the input vector graphic into a BSP tree encoded in each cell covered by this polygon. Then, all trees for each cell are linked in a single data structure for that cell. The whole technique requires only *two* (regular) textures to encode each vector image: the *Hash Texture* that stores the grid, and the *Nodes Texture* that contains the nodes for all the trees for all the texels, indexed by the entries in the Hash Texture. Runtime evaluation simply requires, for each texture coordinate to sample, selecting the appropriate cell and traversing the tree until an intersection is found, which can be done quite efficiently in modern shader-based GPUs.

4. Preprocessing

A plain vector graphic can be extremely complex to evaluate in the GPU, as it can be formed by thousands of oriented

Figure 1: *An example of a vector graphic that can be used as textures with the algorithm presented here.*

polygonal figures organized into layers, each with hundreds of edges. See Figure 1. To improve access efficiency, we partition the vector graphic with a grid that is used as a spatial hash. This hash will be encoded in a texture we call the *Hash Texture*. The hash texture is thus formed by a grid of cells we call *hash entries*. Each one of these hash entries can either be a solid color, or store a pointer to a second texture that stores the actual vector information, called the *Nodes Texture*.

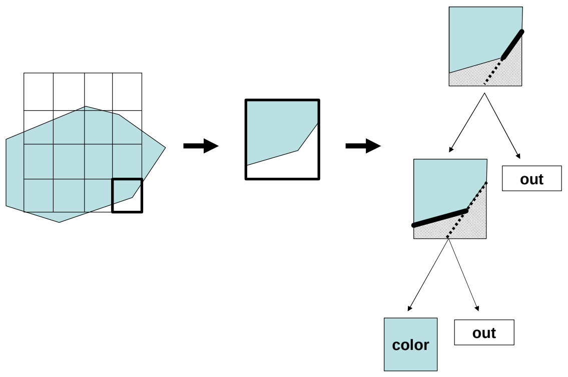

Figure 2: *The edges in a cell are structured using a BSP tree. The white rectangle represents a reference to the polygon exterior.*

To generate the nodes texture, firstly each polygonal figure that overlaps with the cell is converted into a BSP tree. See Figure 2. Each tree node contains a segment from the original polygon, which we use to split the cell in two parts. The node also contains two pointers to the two children (called "left" and right" in our implementation, although these names are arbitrary). However, if any of those is actually a reference to the "interior" solid color of the polygonal figure, the figure color is stored instead. Similarly, if any of the two entries has to refer to the "exterior" of the polygon, a null code is temporarily stored (see below). In a second step, if a tree node contains a null code, it is replaced by a pointer to the root node of the tree representing the next layer, unless it is the last one in which case we store the background color. See Figure 3. If, at a given layer, a polygonal figure completely covers the cell, the tree is pruned and child nodes are replaced by the corresponding figure color.

As a final step, the data structure for each cell, its tree, is balanced to minimize, in runtime, the average number of accesses needed to traverse the data structure, thus gaining efficiency and getting close to the theoretical cost of $O(ln(n))$ expected for binary trees.

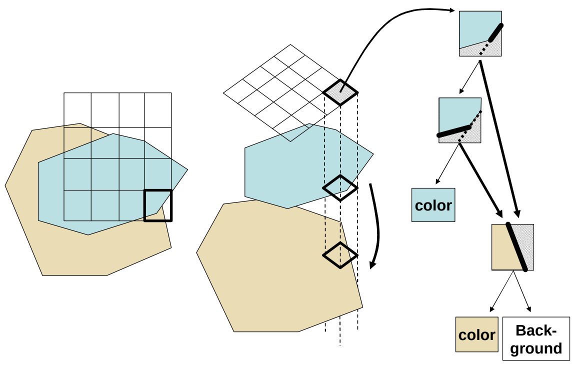

Figure 3: *References to the exterior are linked to the next layer. Here the white rectangle represents a reference to the exterior of all polygons, i.e., the background color.*

5. Run-Time Visualization

Evaluation is completely done in the fragment shaders. There, the sampling texture coordinate is received as input, which are then used to query the hash texture. If the value found there is a color code, it is returned and evaluation finished immediately. If it does not, a simple iterative traversal starts at the root node of the first tree reference, and descends until the code for a solid color is encountered. To decide how to proceed at each node, the position of the texture coordinate is compared with respect to the stored line segment, and corresponding one of the two children is chosen.

5.1. Minification and Magnification

The proposed algorithm can handle magnification quite gracefully, simply by supersampling the texel area. This can be simply done by taking several samples on a texture area defined by the projected are of the current pixel, as done in current hardware implementations. However, if samples can be grouped in the same texel, the tree evaluation algorithm could re-use evaluations to avoid wasted computations. However, this last optimization was not implemented in our prototype and remains as subject of future work.

Minification is currently implemented in a very simple but effective way: at a predefined distance from the observer, which can be defined absolutely or depending on the pixel projected area, we switch to a regular raster texture with mip-mapping enabled, so results are efficiently anti-aliased by the hardware itself. Thus, no special implementation is needed in this case.

6. Results

As we can see in Figure 4, our method works well for distances ranging from close to far views. Frame rates are sustained at more than 150 fps on an NVidia GeForce 7600, which is a rather old graphics card now, even with a large number of vector textured objects. Our algorithm is pixel-bound, which means it puts the most workload on the fragment shaders. Thus, the larger the size on screen of the area to be textured, the more evaluations that will be needed. In this respect, the computational cost grows linearly with the number of pixels, but this must be multiplied by the $O(ln(n))$ dependency on the number of primitives to evaluate.

One trivial change to improve speed would be to sort the root elements in the Nodes Texture following a Morton line, as done in Garcia's thesis [Gar12]. Currently, the information for the cells are simply sored in raster order, but regular texture access patterns show that encoding following the mentioned Morton line order would benefit coalesced memory access, thus improving the speed of our implementation.

We should discuss the influence of the grid size on the overall performance of the presented algorithm. Hash resolution only influences the amount of detail contained within each texel, with larger resolutions corresponding to less detail. However, this reduction is bounded from below: for cells where polygon boundaries overlap, for instance, there will always be the intersection itself to store, so at least two nodes are required, and more for more complex cases (e.g., a simple linear feature). On the other hand, increasing grid resolution implies performing fewer evaluations at each sample, which results in an increased performance.

7. Conclusions and Future Work

We have introduced a technique (data structures plus a runtime evaluation algorithm) for the efficient use of vector graphics as textures on the GPU. Our formulation brings the advantages of a hash-based system, plus the logarithmic evaluation cost of a tree. This computational cost probably is as far as we can get in terms of efficiency if a set of features must be exhaustively evaluated. We reduce to the minimum the features to evaluate for each sample by replacing whole subtrees when a solid color could be used for the entire area.

This technique relies intensively on modern GPU capabilities, as it depends on the compromise between the fill rate (number of rendered triangles) vs. the power of pixel shaders

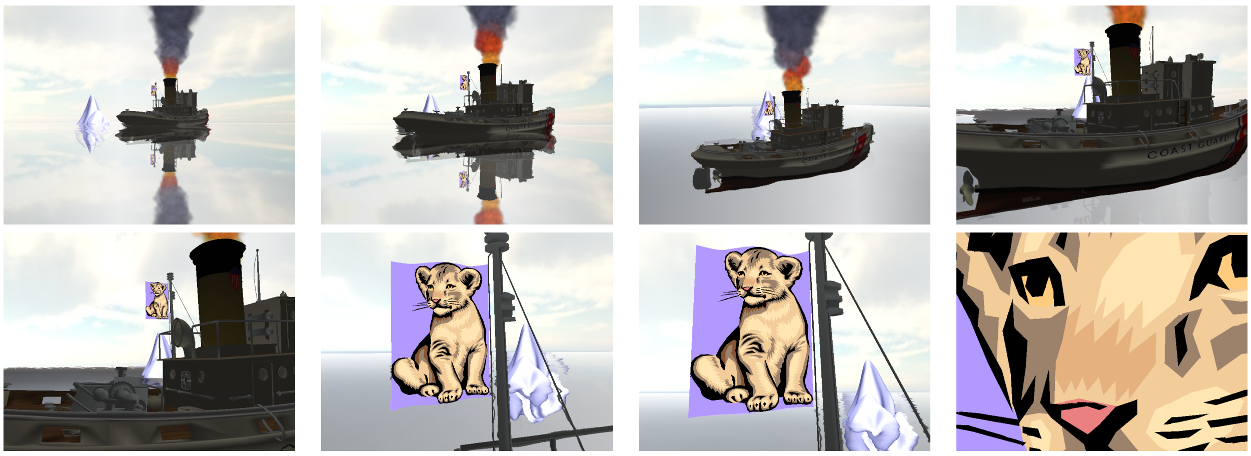

Figure 4: *some resulting images of our algorithm, showing perfect edges at any scale.*

to evaluate it. In the experiments I have found that the technique behaves very well in terms of speed when compared with rendering a model with the equivalent polygons as the vector texture, providing excellent quality when compared with other vector-based techniques.

Future lines of research include the possibility of adding gradient evaluations and diffusion curves, much in the line of Sun et al. [SXD*12]. However, this implies solving complex computations which might be difficult to implement without resorting to a sophisticated CUDA implementation.

Acknowledgements

I'd like to thank Ismael Garcia, Francisco Gonzalez and all the GGG team for their support during the development of this project. I also would like to thank the anonymous reviewers for their valuable comments. This work was partially funded by the TIN2014-52211-C2-2-R project from Ministerio de Economía y Competitividad, Spain.

References

[App67] APPEL A.: The notion of quantitative invisibility and the machine rendering of solids. In *In Proc. 22nd Natl. Conf.* (1967), pp. 387–393. 1

[Gar12] GARCIA I.: *Parallel spatial data structures for interactive rendering, Universitat de Girona, Spain*. 2012. 3

[JCW09] JESCHKE S., CLINE D., WONKA P.: Rendering surface details with diffusion curves. *ACM Trans. Graph. 28*, 5 (Dec. 2009), 117:1–117:8. 2

[LB05] LOOP C., BLINN J.: Resolution independent curve rendering using programmable graphics hardware. *ACM Trans. Graph. 24*, 3 (2005), 1000–1009. 1

[LH06] LEFEBVRE S., HOPPE H.: Perfect spatial hashing. In *SIGGRAPH '06: ACM SIGGRAPH 2006 Papers* (New York, NY, USA, 2006), ACM Press, pp. 579–588. 1

[NH08] NEHAB D., HOPPE H.: Random-access rendering of general vector graphics. *ACM Trans. Graph. 27*, 5 (Dec. 2008), 135:1–135:10. 2

[OBW*08] ORZAN A., BOUSSEAU A., WINNEMÖLLER H., BARLA P., THOLLOT J., SALESIN D.: Diffusion curves: A vector representation for smooth-shaded images. *ACM Trans. Graph. 27*, 3 (Aug. 2008), 92:1–92:8. 2

[QKM07] QIN Z., KAPLAN C. S., MCCOOL M. D.: Circular arcs as primitives for vector textures. In *Tech Rep CS-2007-41* (2007). 1

[QMK06] QIN Z., MCCOOL M. D., KAPLAN C. S.: Real-time texture-mapped vector glyphs. In *SI3D '06: Proceedings of the 2006 symposium on Interactive 3D graphics and games* (New York, NY, USA, 2006), ACM Press, pp. 125–132. 1

[RBW04] RAMANARAYANAN G., BALA K., WALTER B.: Feature-based textures. In *Eurographics Symposium on Rendering* (2004). 2

[RNCL05] RAY N., NEIGER T., CAVIN X., LEVY B.: Vector texture maps. In *Tech Report ALICE-TR-05-003* (2005). 1

[SEH07] STROILA M., EISEMANN E., HART J. C.: Clip art rendering of smooth isosurfaces, ieee transactions on visualization and computer graphics. 1

[Sen04] SEN P.: 2004. silhouette maps for improved texture magnification. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics Hardware, ACM, New York, NY, USA, HWWS'04* (2004), pp. 65–73. 1

[SXD*12] SUN X., XIE G., DONG Y., LIN S., XU W., WANG W., TONG X., GUO B.: Diffusion curve textures for resolution independent texture mapping. *ACM Trans. Graph. 31*, 4 (July 2012), 74:1–74:9. 2, 4

[TC04] TUMBLIN J., CHOUDHURY P.: Bixels: Picture samples with sharp embedded bounda. In *Eurographics Symposium on Rendering* (2004). 1

[TC05] TARINI M., CIGNONI P.: Pinchmaps: Textures with customizable discontinuities. *Computer Graphics Forum 24*, 3 (2005), 557–568. (Eurographics 2005 Conf. Proc.). 1