Random-Access Rendering of Animated Vector Graphics

[Research paper submission for RMIT CS Research Methods course]*

Student:
Ivan Leben
ivan.leben@student.rmit.edu.au

Supervisor:
Geoff Leach
gl@rmit.edu.au

ABSTRACT

The modern graphics hardware requires random access to image data to perform texture mapping onto curved surfaces. Efficient random access allows texture mapping of raster images to be accelerated by the GPU. On the other hand, traditional vector image representation cannot be directly interpreted by the established GPU architecture. In order to be accelerated, the vector image data needs to be first translated into a suitable representation. Recent research allowed texture mapping of vector images by taking advantage of the programmable stages of the GPU. However, the new rendering approach introduces an expensive preprocessing step running on the CPU. In this thesis we investigate the techniques to fully accelerate vector image rendering and texture mapping on the GPU, including the preprocessing steps. We will consider both traditional graphics-based approaches as well as using the GPU as a general-purpose computing device to construct an alternative representation of a vector image, suitable for final rendering and texture mapping. The anticipated improvement in performance could broaden the applications and allow texture mapping of animated vector images.

Keywords

Vector graphics, rendering, animation, random access, GPU

1. INTRODUCTION

Random access rendering is a method whereby the colour of an image might be queried at an arbitrary image space coordinate in an arbitrary order. Efficient random access to image data is a requirement for texture mapping and has traditionally been associated with raster images. The definition of a raster image as a grid of coloured pixels lends itself perfectly to the random access scheme. However, the main drawback is the artifacts such as jagged or blurred edges arising from missing detail in information when the image is scaled.

*This paper is a research proposal for an Honours Thesis.

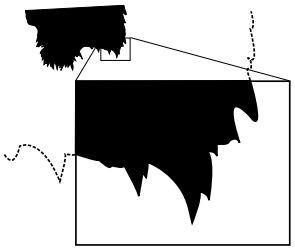


Figure 1: Sharp colour discontinuities are preserved at a large zoom. However, to find the colour of pixels on the screen, the parts of the shape outside the visible area need to be processed as well.

In contrast, vector graphics use mathematical equations and shapes to describe coloured regions. Such a definition allows scaling of a vector image while retaining sharp edges at colour discontinuities. As reviewed in Section 2, in order to classify a point as inside or outside a vector shape, we need to consider every one of its primitives. This makes random access inefficient and has until recently prevented the use of vector images as textures. Additionally, as shown in figure 1, when rendering a small portion of a shape under large zoom, the parts outside of the visible area still need to be processed, adding to unnecessary overhead as the zoom factor increases.

Applications of vector graphics are many, ranging from high detail maps to general scalable user interfaces. An efficient random access algorithm would further broaden the usability of vector images. Regions of a map outside of the view could be efficiently discarded as the user zooms in on a particular detail. Graphical user interfaces could be mapped onto surfaces in 3D space allowing new ways of navigation and screen estate management.

Historically, Graphics Processing Units (GPUs) have specialised into rendering of triangle primitives and have no built-in support for rasterization of curved shapes present in vector images. Nonetheless, dependency on the triangle-based rasterization has been alleviated by the increase in GPU programmability. Recently, [NH08] have developed a rendering technique that evaluates vector shapes directly in the programmable stages of the GPU, after localising the vector data in an expensive preprocessing step on the CPU. This allows efficient random access, but limits the use to static non-animated graphics.

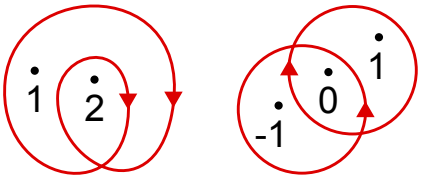
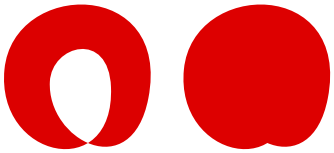


Figure 2: Points in curves and their respective winding numbers.



(a) Even-odd (b) Non-zero

Figure 3: Same shape filled using two different fill rules.

An important thing to consider is the paradigm shift taken by an approach such as [NH08]. The traditional graphics pipeline starts with triangle and line primitives and rasterizes them into the framebuffer. Where primitives overlap, this results in multiple executions of the same processing instructions for a single pixel with regard to a different primitive. In contrast, the approach being discussed preprocesses the entire set of primitives in the scene to build a look-up data structure. In the final rendering stage, all the primitives affecting a given pixel can be looked up efficiently in a single execution of processing instructions to determine the colour of the pixel. It is important to note that such an approach increases the per-pixel processing complexity, but is facilitated by the recent advance in the graphics hardware architecture.

The focus of this research is to extend the above approach by investigating alternative vector image data structures and preprocessing techniques, suitable to parallelisation and acceleration via the GPU. Our goal is to implement the entire rendering process, including the preprocessing steps, on the GPU. By taking advantage of the parallel architecture we anticipate great performance improvement in the combined preprocessing and rendering time, allowing application to animated vector images where continuous preprocessing is required.

2. BACKGROUND

2.1 Vector Image Definition

A vector image is a set of vector shapes, each defined as an open or closed spline of straight line and curved segments. Closed shapes can be filled, meaning the spline defines a border between exterior and the coloured interior of the shape. Shapes can also be stroked, meaning the lines have their visible width defined. For practical purposes, stroked shapes can be adequately approximated by another more complex vector shape describing the resulting outline of the thick line, thus reducing the definition to closed, filled shapes.

The interior of the filled shapes is determined according to the value called *winding number*. For any point, the winding number represents the total number of times that the curve travels around the point. Clockwise movement counts as positive whereas anticlockwise movement counts as negative. Figure 2 shows examples of points in curves and their respective winding numbers. Note that one shape can consist of multiple contours in which case the winding number is the sum of the individual winding numbers of all the contours.

There are two common rules for determining the interior of a shape based on the winding number, namely *even-odd* and *non-zero*. The former classifies the areas with an odd winding number as interior while the latter considers any area with a winding number different than 0 to be inside as shown in figure 3.

The following subsection describes how to evaluate points on a vector spline and find the winding number of a point for the purpose of rendering.

2.2 Evaluation of spline segments

Individual segments of a vector spline are most commonly defined using Bézier curves of first to third degree, as described in [Far02]. In general a Bézier curve of degree n is defined as follows:

$$\mathbf{B}(t) = \sum_{i=0}^{n} \mathbf{b}_{i,n}(t)\mathbf{P}_i, \quad t \in [0,1]$$

where the points $\mathbf{P}_i$ are called *control points* of the Bézier curve and the polynomials

$$\mathbf{b}_{i,n}(t) = \binom{n}{i} t^i (1-t)^{n-i}, \quad i = 0, ...n$$

are known as Bernstein basis polynomials of degree n with the following property:

$$\sum_{i=0}^{n} \mathbf{b}_{i,n}(t) = 1, \quad t \in [0,1]$$

meaning that the sum of all the polynomials of a certain degree is always 1. The Bézier curve equation therefore interpolates between the given control points according to the parameter t, beginning exactly at $\mathbf{P}_0$ and ending exactly at $\mathbf{P}_n$. A curve of degree n has $n+1$ control points. Obviously, a Bézier curve of degree 1 is essentially a straight line. Consecutive spline segments share a control point where one ends and the next one begins. The ordered set of control

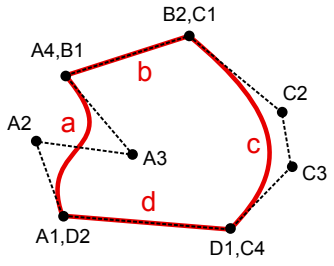


Figure 4: A closed spline (red) consisting of two line segments and two cubic Bézier curve segments with its control polygon (black).

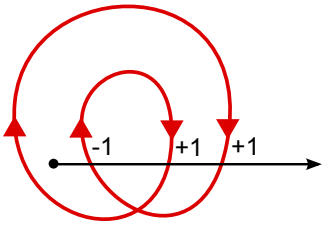


Figure 5: The direction of the intersections between a horizontal ray and the segments of a spline count towards the winding number of the point.

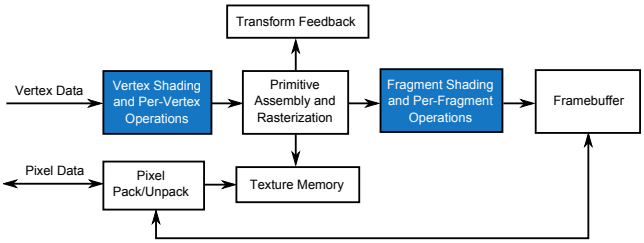


Figure 6: OpenGL rendering pipeline. Programmable parts are coloured in blue.

points of all the segments defines the *control polygon* of the spline as shown in figure 4.

A simple way of finding a point on the Bézier curve at an arbitrary parameter value is to plug the value into every basis function, compute the products with the respective control points and add them together. While very straightforward, this approach is not numerically stable. Instead, as described in [BM99] we can evaluate the points on the curve by recursive subdivision of the sides of its control polygon. This technique, known as *de Casteljau's algorithm* reduces numerical errors during the course of evaluation.

The Bézier curve equations can also be used to evaluate the winding number of any point with regards to a vector spline. We start by casting a ray from the point into an arbitrary direction. Based on the *Jordan Curve Theorem*, the point is found to be inside the polygon, if the ray crosses an edge of the polygon an odd number of times. Counting the number of intersections gives a value known as the *crossing number*. Now, by checking the direction of the intersecting edge, we instead add $+1$ for every left-to-right crossing and -1 for every right-to-left crossing. The resulting value is the winding number of the point.

Imagine the ray is being cast from point $P(x, y)$ horizontally in the positive direction of the x axis. We can find an intersection with any segment of the spline by finding the roots t_i of the equation $y = y(t)$ where $y(t)$ is the linear or higher order polynomial describing the spline segment. When $t \in [0, 1]$ and $x < x(t)$, the ray intersects the segment.

Obviously, the winding number at any point depends on the geometry of the entire shape outline. It is this requirement that poses the greatest obstacle when designing an efficient random access rendering algorithm.

2.3 GPU Architecture

Traditional display devices require input in the form of a grid of coloured pixels, called a *raster*. For a graphical object to be displayed, the pixels covered by its shape need to be determined in a process called *rasterization* and the colour of the pixels in the display raster changed accordingly. The architecture of modern GPUs is optimised for rasterization of two types of primitives: lines and triangles. The roots of this design reach into the early history of computer graphics when parallel computing architecture was not as powerful and programmable as today. Triangle rasterization coupled with texture mapping of raster images was much simpler to implement in hardware compared to direct evaluation of higher order mathematical equations. Complex shapes have to be described using triangles, in order to accommodate and benefit from hardware-accelerated rendering.

The GPU has evolved enormously in terms of speed and programmability, however the fundamental approach has not changed much, with the triangle rasterization unit still being a core built-in component today. Nonetheless, the increase in programmability has enabled new ways of using the GPU.

Firstly, programmers can now write code that defines the behaviour of certain stages of the rendering pipeline as exposed by the graphical APIs such as OpenGL and DirectX. The pipeline is still based around the line and triangle being the basic rasterizable primitives, however, custom programs for processing of vertices and pixels can be uploaded onto the hardware. The stage processing the vertices is called a *vertex shader*, while the pixel processing stage is named a *fragment shader*.

More importantly, new programming interfaces have emerged that expose the GPU hardware as a general-purpose multicore computing device. APIs such as CUDA and OpenCL allow programmers to supply data and upload the code onto the GPU to process that data by hundreds of processors in parallel. This approach, known as *GPU Computing*, may be completely devoid of any graphics-related paradigms, although often the results of computation are used for graphical purposes or end up being visualised via traditional rendering pipeline.

3. PREVIOUS WORK

In a typical CPU implementation, every spline is first approximated by linear segments. Intermediate points on the curved segments can be obtained in a straightforward manner by curve evaluation at a fixed step. Alternatively, the process can be optimised via adaptive algorithms that generate less points at the straight regions and more points at the regions of higher curvature. Common approaches include forward difference evaluation and recursive subdivision based on de Casteljau algorithm. The interior of the resulting piecewise linear polygon is triangulated according to the fill-rule applied to the shape. Finally, the triangles are rasterized into the framebuffer.

Unfortunately, the known curve subdivision and polygon triangulation algorithms are expensive in terms of required processing time compared to the final rendering of the image.

Moreover, they are iterative, which means they cannot be efficiently parallelised and therefore cannot take advantage of the GPU.

[LB05] have developed an approach where the implicit equation of quadratic curves is evaluated directly in the fragment shader to perform inside/outside classification. They start by observing that "the implicit form of any rational parametric quadratic curve is a conic section; and that any conic section is the projected image of a single canonical parabola." They show that a quadratic curve with control points $(0,0)$, $(0.5,0)$, and $(1,1)$ plots the quadratic curve $v = u^2$ exactly. They assign these points as the (u,v) texture coordinates to the vertices of a triangle. Suppose the vertices are the control points of another arbitrary quadratic curve. As the hardware interpolates the texture coordinates across the triangle, it essentially projects the quadratic curve from texture space into the screen space. They determine if the pixel is inside or outside the curve by evaluating $f(u,v) = u^2 - v$ in a pixel shader program. If $f(u,v) < 0$ then the pixel is inside the curve, otherwise it is outside.

The work of [LB05] removes the need for curve subdivision, but the interior of the shape still needs to be triangulated. [KSST06] combine the projective rendering of the quadratic curves with a stencil polygon rendering technique. First, an arbitrary pivot point is selected. For each linear segment a triangle is drawn with one of its vertices at the pivot point and the other two at the vertices of the line segment. For each quadratic segment a triangle is drawn between the pivot point and the first and the last control point, skipping the middle control point. Additionally, a triangle is drawn for each quadratic curve, discarding the pixels based on the projective technique of [LB05]. For every pixel drawn, the binary value in the stencil buffer is inverted. After all the triangles have been drawn, the values in the stencil buffer essentially represent the crossing number for every point as if the intersection ray was cast towards the pivot point. This allows the pixels in the framebuffer to be coloured according to even-odd fill rule.

The algorithms described so far enable efficient rendering of animated vector graphics. However, they are based on forward oriented triangle rasterization. Instead, we are aiming for a random-access scheme, where the pixel colours might be queried simultaneously by multiple fragment shaders at an arbitrary point within the vector shape in an arbitrary order. To achieve this, we need to be able to evaluate the winding number of any point in the image space from within the fragment shader.

[NH08] presented an approach that localises the vector image representation to coarse lattice cells in a preprocessing step. Each cell contains a localised description of the graphics primitives it overlaps. This information is encoded into a texture and interpreted by the fragment shader. For every fragment, the vector data is read from the lattice cell that the texture coordinate falls in. The winding number is calculated using the ray-casting intersection technique, but thanks to the localisation, only the segments overlapping the cell need to be considered. They observe that increasing the lattice resolution decreases the number of spline segments to intersect per pixel, as long as the detail in the image is fairly uniform. This in turn reduces the rendering time and increases the achievable rendering framerate. However, they find that the preprocessing takes too long for real-time use which prevents their approach to be used with animated images.

4. RESEARCH QUESTIONS

In this research we extend the previous approaches by investigating the following questions:

1. How to efficiently localise vector image representation using programmable capabilities of the GPU?
2. What degree of vector data localisation provides an optimal balance between preprocessing and rendering time?
3. What is the benefit of vector data localisation with regard to animated vector graphics where continuous preprocessing is required?

5. OUR APPROACH

In order to take full advantage of the parallel architecture of the GPU, it is first necessary to find a suitable localised image respresentation satisfying two conditions. On the one hand, the representation must exhibit a high degree of localisation to allow efficient random-access. On the other hand, this representation must be obtainable via some process that can achieve a high degree of parallelism.

Like [NH08], we create a coarse lattice across the original vector image. However, our approach differs in the contents of the lattice cells. To preserve the winding number within a cell, their algorithm creates auxiliary segments on the right boundary of each cell. The direction of the auxiliary segments affects the resulting winding number and depends on the segments in other cells, to the right of the cell in question. Therefore, for every segment crossing a cell, the winding number needs to be integrated into the cells to its left. The dependency of each cell on the contents of other cells makes it challenging to devise a parallel localisation algorithm.

Instead, we start by observing that the winding number of any region in a vector shape can be determined by a single crossing of an edge, as long as the winding number of another adjacent region is known. As shown in figure 7 it is possible to move from one region to another, adjusting the winding

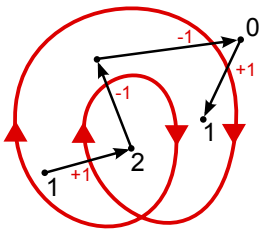


Figure 7: Starting from a known winding number, the winding number of adjacent regions can be determined by a single crossing of an edge.

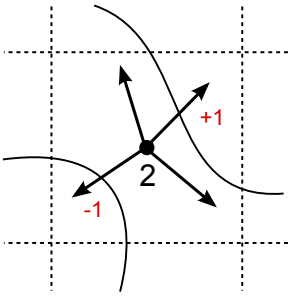


Figure 8: Casting a ray from the pivot point and intersecting the segments in the cell gives the winding number of any other point within the same cell.

number according to the direction of the crossing edge and thus finding the winding number of every region. We take advantage of this observation by assigning a *pivot point* to every cell and finding the winding number of that point in the preprocessing step. The location of the pivot point is implicitly defined at the centre of the cell. While [NH08] cast a horizontal ray in the positive x direction, we cast a ray from the pivot point of the cell in an arbitrary direction, towards any other point in the cell. The intersections of the ray with vector segments crossing the cell give the change in winding number from the original winding number of the pivot point.

Our localised image representation thus consists of a set of lattice cells containing a pivot winding number and the intersecting spline segments. No additional auxiliary segments are required. We will now describe the process of obtaining the winding numbers of the pivot points. Imagine each cell being a pixel of a low-resolution image. The winding number parity at the pivot point can be obtained by sampling the vector image at the centre of the pixel. Now, we have already described a technique of forward rendering of vector images presented by [KSST06]. Therefore, to obtain the grid of winding numbers we essentially render the image using the forward rendering technique at a low resolution matching the size of the pixels to the size of the lattice cells. When an even-odd fill rule is in use, the inversion operations on the stencil buffer used by [KSST06] give the required winding number parity. In the case of non-zero fill rule, the stencil operations are modified to produce the actual winding number.

Our final rendering algorithm combines the grid of winding numbers with the vector segment data encoded into variable-length cell streams described by [NH08]. Each cell stream encodes the control points of the segments crossing the respective cell, so the encoding process involves intersection of vector segments with the lattice cells. The work done so far implements the creation of winding grid on the GPU, while the cell stream encoding is done on the CPU. The future efforts of this research will investigate the implementation of cell stream encoding using the GPU.

6. CONCLUSIONS AND FUTURE WORK

Vector graphics offer superior image quality over raster images at various levels of zoom. Until recently, raster images allowed a greater variety of applications due to their unique property of efficient random access. We have described previous efforts to allow such applications of vector images and proposed our own approach which involves processes with a high degree of parallelism, suitable to implementation on the modern GPU.

We have introduced a new type of localised vector image representation that minimises the additional data required and simplifies the preprocessing step. Our approach utilises an existing forward rendering technique to generate a grid of winding numbers. This grid is then used in the final random-access rendering step, to determine the winding number of other points within the same coarse lattice cell.

The above preprocessing and rendering steps of our algorithm are performed entirely by the GPU and take full advantage of its parallel architecture. Besides the winding number grid, our final rendering step requires another input in the form of vector segment data encoded into a texture of variable-length cell streams. The avenues of future work include:

1. Investigation of general-purpose GPU computing techniques to accommodate non-graphical approaches.
2. Implementation of spline segment encoding into cell streams using the GPU.
3. Application to animated vector images being continuously preprocessed.
4. Analysis of the performance in an (occasionally) animated environment.

The effectiveness of our algorithm will be evaluated by measuring the frame rate achieved when rendering an animated image. The offset from the interactive rate of 30 frames per second will be taken as a measure of success.

7. REFERENCES

[BM99] Wolfgang Boehm and Andreas MÃijller. On de casteljau's algorithm. *Computer Aided Geometric Design*, 16(7):587 – 605, 1999.

[Far02] Gerald Farin. *Curves and surfaces for CAGD: a practical guide*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition, 2002.

[KSST06] Yoshiyuki Kokojima, Kaoru Sugita, Takahiro Saito, and Takashi Takemoto. Resolution independent rendering of deformable vector objects using graphics hardware. In *SIGGRAPH '06: ACM SIGGRAPH 2006 Sketches*, page 118, New York, NY, USA, 2006. ACM.

[LB05] Charles Loop and Jim Blinn. Resolution independent curve rendering using programmable graphics hardware. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Papers*, pages 1000–1009, New York, NY, USA, 2005. ACM.

[NH08] Diego Nehab and Hugues Hoppe. Random-access rendering of general vector graphics. In *SIGGRAPH Asia '08: ACM SIGGRAPH Asia 2008 papers*, pages 1–10, New York, NY, USA, 2008. ACM.