Rendering 2D vector graphics on mobile GPU Devices

H. Kumar[†][1] and A. Sud[1]

[1]Adobe Inc., Noida, Uttar Pradesh, India

Abstract

Designers and artists world-wide rely on vector graphics to design and edit 2D artwork, illustrations and typographic content. There is a recent trend of vector graphic applications moving to mobile platforms such as iPads, iPhone and mobile phones and with that there is new interest in optimised techniques of rendering vector graphics on these devices. These vector applications are not read only but also requires real time vector editing experience. Our solution builds upon standard 'stencil then cover' paradigm and develops an algorithm targeted for GPUs based on tile based deferred rendering architecture. Our technique provides an efficient way to use signed distance based anti-aliasing techniques with 'stencil then cover' paradigm by employing a state machine during the fragment shader stage of graphics pipeline.

CCS Concepts

• ***Computing methodologies*** $\rightarrow$ ***Antialiasing; Rasterization;***

1. Introduction

We present a method to render anti-aliased 2D vector objects natively on mobile GPUs. Our technique builds upon [KSST] and develops a solution for tile based deferred GPU rendering architecture. Mobile GPUs (tile-based GPUs) allows reading current pixel memory without any performance penalty (without any need of texture barriers or different draw calls). Our technique exploits this fundamental property of mobile GPUs and builds an algorithm for rendering 2D vector graphics (including all kind of shapes – convex, concave, compound paths, overlapping paths bounded by cubic Bezier splines).

Our method uses spread based anti-aliasing techniques instead of multisampling. Our method does not require multi-sampled textures and per sample shading. Standard 'stencil then cover' techniques rely on two GPU render passes – one render pass to write coverage values to stencil buffer and second render pass to write color values to render texture. Further, it requires additional draw calls to clear the stencil buffer and since draw calls leaves stencil buffer in dirty state, multiple vector objects cannot be drawn with a single render pass, significantly limiting the overall rendering performance. Our technique uses a single render pass to write coverage and color values and also clears the coverage buffer in a single render pass. This allows to batch and render multiple vector objects in a single render call leading to significant performance gains. Our technique implements batching of vector objects by introducing concept of interleaved coverage and color triangles.

[†] Chairman Eurographics Publications Board

2. Our Solution

Following section details out our solution in two phases, Triangulation and GPU Render Pass.

2.1. Triangulation

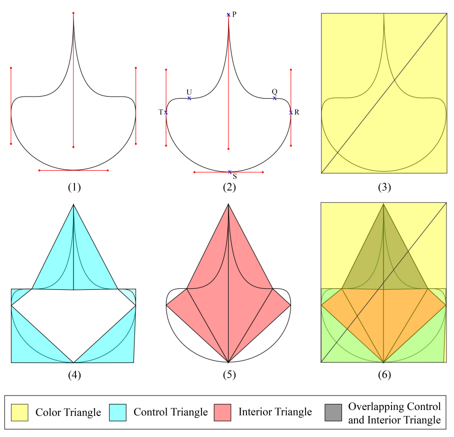

Figure 1: *1. Input vector shape 2. Cubic Beziers approximated by Quadratic Beziers 3. Color Triangles 4. Quadratic Bezier Control Triangles 5. Interior Polygon Triangulation 6. Total Coverage triangles and color triangles*

For any input vector shape, we first approximate cubic Bezier

DOI: 10.2312/egp.20211025

curves of shape by multiple quadratic Bezier curves . By joining contiguous end points of quadratic Bezier curves, we generate an interior polygon. To triangulate this interior polygon, We extend standard algorithm of triangulating the interior of 2D polygons [Ope]. Then, we add the control triangles of quadratic Bezier curves to output of interior polygon triangulation to complete the set of 'coverage triangles'.

Additionally, Coverage triangles are expanded [NH] to cover pixels around the edges of the curve to avoid aliasing due to under coverage. These triangles may overlap with each other or extend to pixel outside the shape of input curve. GPU render pass resolves these overlaps and ensures that a pixel is marked only once and the pixels lying outside the shape are discarded.

In addition to coverage triangles, we generate 'Color Triangles' for each input vector shape. Color Triangles are generated by splitting the bounding box of input geometry along the diagonal. Irrespective of the input geometry, we always have two color triangles. These are appended post coverage triangles in the output buffer.

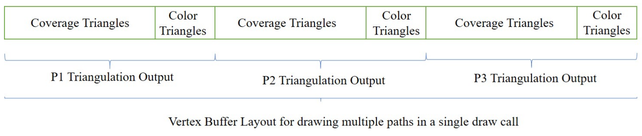

Figure 2: *Triangulated buffer representation.*

Coverage and color triangles are packaged and uploaded to the GPU. A uniform marker is set to distinguish between two types of triangles in GPU render pass when multiple paths are batched together. This uniform marker is maintained in a separate buffer which de-marks boundary of each path.

2.2. GPU Render Pass

GPU Render pass in our solution uses a framebuffer with two color attachments, color buffer (RGBA32) and coverage buffer (R8) respectively. Color buffer stores the final color value and coverage buffer stores the transient coverage of each pixel covered by input shape. GPU render pipeline comprises of vertex shader and fragment shader. Vertex shader simply transforms the vertices and outputs to rasterizer. Fragments output from rasterizer are processed by fragment shader stage where our technique introduces a state machine shown in figure 3.

Based on - type of fragment (if the fragment is generated from coverage triangle or color triangle), coverage value of the fragment, coverage value is written to coverage buffer. Fragment shader reads current values in frame buffer (pixel memory) to decide the state changes in pixel memory due to current fragment. Clearing coverage values to zero is necessary to render multiple overlapping vector objects in a single draw call to fully utilize parallel computing power of GPU.

Figure 4 shows the different states of frame buffer attachments for a single draw call for two vector graphic objects. Note that shown two objects are drawn in a single render pass and also leaves the coverage buffer to clear state for reuse in next draw call.

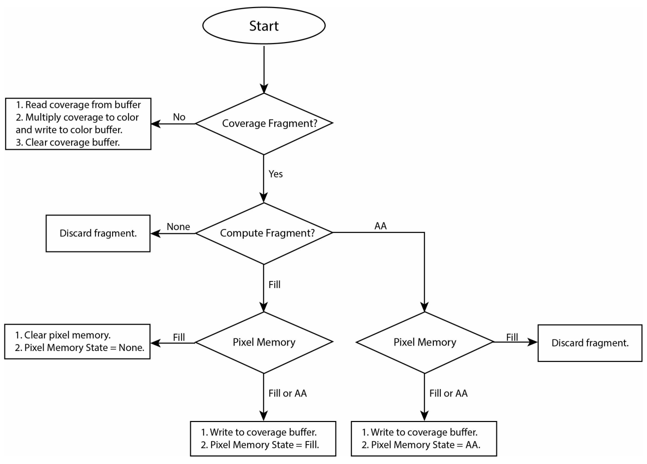
Start
Coverage Fragment?
No
1. Read coverage from buffer
2. Multiply coverage to color and write to color buffer.
3. Clear coverage buffer.
Yes
Compute Fragment?
None
Discard fragment.
AA
Fill
Pixel Memory
Fill
1. Clear pixel memory.
2. Pixel Memory State = None.
Fill or AA
1. Write to coverage buffer.
2. Pixel Memory State = Fill.
Pixel Memory
Fill
Discard fragment.
Fill or AA
1. Write to coverage buffer.
2. Pixel Memory State = AA.

Figure 3: *Fragment shader state machine.*

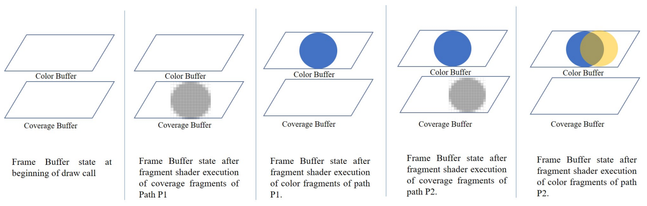
Color Buffer
Coverage Buffer
Frame Buffer state at beginning of draw call
Color Buffer
Coverage Buffer
Frame Buffer state after fragment shader execution of coverage fragments of Path P1
Color Buffer
Coverage Buffer
Frame Buffer state after fragment shader execution of color fragments of path P1.
Color Buffer
Coverage Buffer
Frame Buffer state after fragment shader execution of coverage fragments of path P2.
Color Buffer
Coverage Buffer
Frame Buffer state after fragment shader execution of color fragments of path P2.

Figure 4: *State machine transformations.*

Single render pass facilitates the use of memoryless coverage buffer, thereby further reducing the memory footprint.

3. Results

We achieve a performance speed up of 3-7x for first frame rendering (without cache) depending upon artwork complexity. This corresponds to editing workflow performances. We benchmark our technique against an existing path rendering technique where path object is tessellated into tightly bound non-overlapping triangles. In this tessellation scheme, control triangles are composed of quadratic Bézier curves (not lines), making tessellation output resolution independent. Also, our technique is at par or better performing in frame redraws (zoom in-out work-flows) when there are no changes in geometry and cached triangle buffer can be redrawn directly on GPU.

References

[KSST] Kokojima Y., Sugita K., Saito T., Takemoto T.: Resolution independent rendering of deformable vector objects using graphics hardware. *SIGGRAPH '06*. doi:10.1145/1179849.1179997. 1

[NH] Nehab D., Hoppe H.: Random-access rendering of general vector graphics. *SIGGRAPH '06*. doi:10.1145/1409060.1409088. 2

[Ope] URL: http://www.glprogramming.com/red/chapter14.html. 2