Compressed Coverage Masks for Path Rendering on Mobile GPUs

Pavel Krajcevski[*]
University of North Carolina at Chapel Hill

Dinesh Manocha[†]
University of North Carolina at Chapel Hill

Abstract

We present an algorithm to accelerate resolution independent curve rendering on mobile GPUs. The bottleneck for certain platform independent GPU implementations is in generating grayscale textures on the CPU containing the amount that each pixel is covered by the curve. In this paper, we demonstrate that generating a compressed grayscale texture prior to uploading it to the GPU creates faster rendering times in addition to the memory savings. We implement a real-time compression technique for coverage masks and compare our results against the GPU-based implementation of the highly optimized Skia rendering library. We observe up to a 2X speed improvement over the existing GPU-based methods in addition to up to a 9:1 improvement in GPU memory gains. We demonstrate the performance on multiple mobile platforms.

CR Categories: I.4.2 [Image Processing and Computer Vision]: Compression (Coding)—Approximate methods I.3.3 [Computer Graphics]: Picture/Image Generation—Bitmap and Framebuffer Operations I.3.4 [Computer Graphics]: Graphics Utilities—Software Support

Keywords: texture compression, coverage masks, 2D path rendering

1 Introduction

One of the main problems in computer graphics is the discretization of continuous functions used to display objects at a finite resolution. Improper discretization may lead to aliasing artifacts from insufficient sampling. In order to alleviate these artifacts, different techniques have emerged for computing proper discretizations [Barros and Fuchs 1979][Lane and M. Rarick 1983]. When rasterizing geometric objects, the main difficulty is determining what percentage of a pixel is covered by the screen-space projection of the object. This information, once calculated, can be stored in an image known as a *coverage mask*. Coverage masks are usually stored as eight-bit grayscale images and can be used in a variety of different ways in order to speed up the rendering of geometric primitives, including caching [Fiume et al. 1983] and GPU based rendering of 2D curves [Google 2014].

Pixel coverage remains an instrumental part of computer graphics. There are many applications where coverage masks are useful, from culling [Zhang et al. 1997] to visibility determination for more efficient lighting [Kautz et al. 2004]. In this paper, we mainly focus on coverage masks used in rendering non-convex piece-wise two-dimensional cubic and quadratic curves, or *paths*, with anti-aliasing (Figure 1). These curves are used in a majority of vector graphics

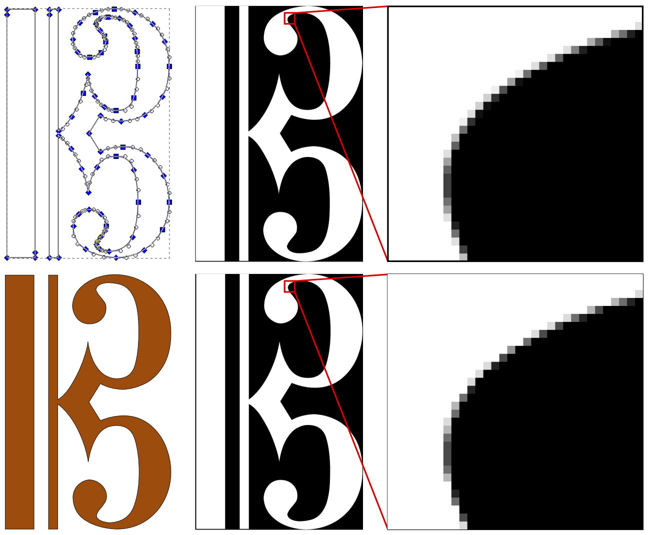


Figure 1: (Top left) *The piece-wise anti-aliased cubic curve used as input.* (Bottom Left) *The final rendered curve.* (Top right) *The uncompressed coverage mask passed to the GPU to determine the amount each pixel is covered by the curve.* (Bottom right) *The compressed coverage mask using our method. On the far right is a zoomed in comparison of the compressed and uncompressed masks. Although only a few pixels differ, using our method, these masks are compressed in real time and save time and memory during the rasterization of these curves.*

data, most importantly as the basis for resolution-independent text rendering using different fonts and sizes. These coverage masks, generated at run-time from network data such as web pages, are used billions of times on a daily basis [StatCounter 1999-2014]. From a sampling of over 750,000 web pages, we have observed that 51% draw arbitrary paths of which 19% are anti-aliased requiring dynamically generated textures. Of the paths that require coverage information, most of the web page rendering time is spent drawing the coverage mask of the path on the CPU prior to uploading it to the GPU.

In this paper, we show that coverage masks generated at run-time by the CPU can be compressed efficiently for GPU-based rendering with little loss in rendering fidelity. We present a way to augment the scan conversion process of non-convex path rendering to directly output compressed textures for use on the GPU. We demonstrate encoding into a variety of different compression formats in order to show applicability to a widespread range of commodity graphics hardware. In particular, we show that even with general 32-bit hardware, efficient coverage mask compression can be performed to target the DXTn, ETC, and ASTC texture compression formats [Iourcha et al. 1999][Ström and Pettersson 2007][Nystad et al. 2012]. Finally, we demonstrate a speedup of up to 2X in rendering speed using compressed coverage masks on current mobile platforms (e.g. tablets and smart phones). This savings in rendering speed is in addition to the GPU memory gains of 2X up to 9X depending on the compression format. Our method is integrated into

[*]e-mail:pavel@cs.unc.edu
[†]e-mail:dm@cs.unc.edu

the Skia[1] two-dimensional rendering library [Google 2014]. This library is the rendering backbone in the popular Google Chrome and Mozilla Firefox web browsers currently being used by billions of people [StatCounter 1999-2014]. Additionally, we test our results against a suite of benchmarks, correctness tests, and web-page data. Overall, our approach aligns well with the current hardware and software trends. The mobile GPU market is growing at a considerable rate with more than a billion sales per year [Shebanow 2014]. To address this trend and develop higher performance on mobile GPUs, hardware vendors are developing more aggressive compression formats that are designed specifically for GPUs [Nystad et al. 2012]. In particular, energy savings during rendering are becoming more important. Using a few extra CPU operations in order to decrease the texture bandwidth by 2-3X likely produces significant energy savings for texture-heavy mobile applications. Texture memory accesses are almost three orders of magnitude more expensive than standard ALU operations [Shebanow 2014]. Our method for compressing coverage masks leverages these trends and becomes increasingly useful as hardware advances.

The rest of the paper is organized as follows. Section 2 gives an overview of recent work in coverage masks and compression formats. Section 3 presents our scan conversion algorithm used during rasterization, and the various compression formats used to store grayscale coverage information. We highlight the performance of our algorithm on various mobile devices in Section 4. Finally, we present conclusions, limitations, and future work in Section 5.

2 Background

In this section, we give a brief overview of prior work on coverage masks, GPU-based vector graphics, and texture compression.

2.1 Coverage Masks

One of the major problems in computer graphics has been to determine the amount that geometry covers a given pixel during rasterization [Barros and Fuchs 1979][Fiume et al. 1983]. This problem, also known as *pixel coverage*, is used to reduce aliasing artifacts caused by the discrete nature of our display devices and memory layouts. More recently, coverage masks have been used for more than simply anti-aliased rasterization. Zhang et. al. [1997] use occlusion maps, a variation of coverage masks, to quickly cull geometry during the rendering of large scenes. Kautz et. al. [2004] use coverage masks to cache hemispherical visibility information in order to perform efficient self-shadowing of objects. Coverage information has also been used to accelerate shading operations in the GPU pipeline, although these methods are more suited to hardware implementations than software [Aila et al. 2003][Fatahalian et al. 2010].

Coverage masks are used extensively to render 2D images from geometric primitives. In particular, coverage information is necessary when rasterizing anti-aliased polygons independent of the color and shading information. In order to render these polygons, first the pixel coverage mask is generated, and then the color of the polygon is modulated by the intensity of the pixel in the coverage mask. This technique is used in the 2D rendering library Skia [Google 2014] for GPU rasterization of non-convex anti-aliased paths.

2.2 GPU-based Vector Graphics

Resolution-independent rendering is important for many objects in graphics such as the arbitrary cubic and quadratic curves used to represent glyphs in most modern fonts. Until recently, these curves have been rendered using software rasterization algorithms. Given the recent advances in GPU development, there has been considerable groundbreaking work to use GPUs to perform resolution-independent rasterization [Loop and Blinn 2005][Kilgard and Bolz 2012][Qin 2009]. As pioneers in this work, Loop and Blinn [2005] devised a method to rasterize Bézier curves by assigning values to the texture coordinates of triangles derived from the control points of the curve. These values were used to calculate the distance from the curve in the given triangle, which was used for proper anti-aliasing. Kokojima [2006] improved the efficiency of this method by exploiting the stencil buffer. Qin [2009] presented a method to exploit the texture storage of a graphics processor to store curve information using approximate circular arcs. Finally, Kilgard and Bolz [2012] describe an approach that transmits control points directly to the GPU to render the curve. Although this method renders vector graphics very quickly, it requires additional proprietary hardware features. Further approaches using signed distance fields have been used by Green [2007] for artist generated vector graphics.

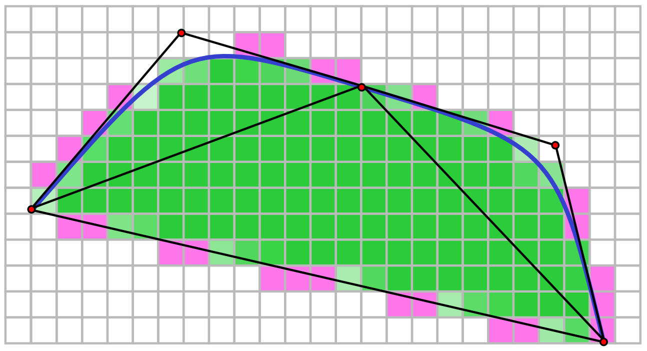


Figure 2: *A piece-wise quadratic curve is filled with green using the Loop-Blinn method. The pixels (pink) whose centers are not covered by the triangles circumscribing the curve will not be drawn if the GPU is not using a hardware anti-aliasing method. For power constrained GPUs, such as those on mobile devices, MSAA is prohibitively expensive due to the large number of fragment shader invocations. When the curve is non-convex, it is often more correct to default to software rendering of the pixel coverage in these situations.*

2.3 Anti-Aliasing Non-Convex Curves

Despite recent advances in using GPUs to accelerate vector graphics rasterization, certain classes of vector graphics still remain slow on mobile hardware. Of the techniques mentioned in Section 2.2, the Loop-Blinn method is among the fastest techniques for rendering resolution-independent vector graphics from arbitrary path data. The GPU-based method introduced by Kilgard and Bolz [2012] builds upon the Loop-Blinn method by implementing a conservative approach to determining coverage information in hardware. Most notably, as shown in Figure 2, for paths that generate smooth curves but are comprised of multiple control points, the triangles that conjoin quadratic and cubic pieces of a curve may not cover all necessary pixels. When these triangles are rasterized by the GPU, the centers of some pixels covered by the path may not be covered by the triangles. For GPUs that do not support hardware-based anti-aliasing, or where such anti-aliasing is too expensive due to power constraints, pixels that should have partial coverage from the path will not be drawn. This can cause aliasing artifacts when rendering curves whose details are on the order of a single pixel.

[1]`https://sites.google.com/site/skiadocs/home`

To support many different use-cases, the 2D rendering library Skia chooses different rendering paths dependent on the path being rendered. For non-convex paths without anti-aliasing, Skia approximates a path using line segments and then uses their endpoints as input to a triangle fan drawing both front and back facing triangles. Using the stencil buffer, pixels can be turned on or off based on whether they are inside or outside the path. However, line segments create significant aliasing artifacts during rendering, and this technique cannot be used for anti-aliased paths.

To perform anti-aliasing, in certain cases Skia uses the Blinn-Phong method followed by extruding the triangles along the normal to the path by the amount required to cover all of the pixels covered by the path. However, for general non-convex paths, this presents artifacts in areas where the extruded polygons of two different curves overlap leading to double-blending and incorrect pixel coverage. As a result, the GPU-based renderer in Skia draws the coverage information in software prior to uploading the resulting grayscale texture to the GPU for shading. This rendering algorithm used to support the use of GPUs can become a significant bottleneck during the rendering of anti-aliased concave paths [Google 2014]. In this paper, we show that the grayscale coverage information can be efficiently compressed to a texture format (Section 2.4) thereby significantly increasing the speed at which it is uploaded to the GPU.

2.4 Texture Compression Formats

Over the past few decades, there has been significant research into texture representations in GPU memory. The main restrictions to texture representation formats were outlined by Beers et al. [1996] as random access and hardware-based decompression. Real-time decoding is supported in modern GPUs, though the performance of the encoding step remains problematic [Krajcevski et al. 2013]. Over the years, many new compression formats have emerged offering quality versus performance trade-offs [Iourcha et al. 1999][Ström and Pettersson 2007][Nystad et al. 2012].

One of the earliest texture compression formats introduced in commodity graphics hardware was the DXTn family of compression formats [Iourcha et al. 1999]. Variations of this format have been implemented in hardware to support grayscale textures and textures with alpha. Subsequently, Ström and Akenine-Moller introduced ETC1, a texture compression format that uses scale and offset factors from look-up tables to reconstruct pixel values [Ström and Akenine-Möller 2005]. A few years later, Ström and Petterson introduced ETC2, which improved upon ETC1 by allowing invalid bit combinations to encode a wider range of pixel values [Ström and Pettersson 2007]. Single channel variations have also been introduced, but their adoption has not reached commodity graphics hardware [Wennersten and Ström 2009]. Nystad et. al. [2012] recently unveiled ASTC, which allows encoders to choose between a variety of compression methods and a variable bitrate from eight bits per pixel down to 0.89 bits per pixel. Although this flexibility in the compression format allows a large quality versus compression size trade-off, developing real-time encoders for ASTC remains difficult.

3 Compressed Scan Conversion

In this section we describe our technique for encoding the coverage information into a GPU-based compressed texture format. Given a piece-wise two-dimensional curve, or *path*, we augment the scan conversion algorithm on the CPU for generating coverage information. Our formulation is based on the assumption that the time spent writing the encoded coverage information into a GPU-specific format can be recovered during the time it takes to upload the texture

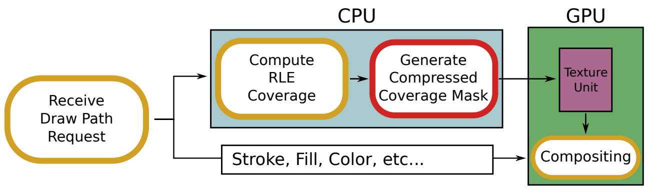

CPU
GPU
Receive Draw Path Request
Compute RLE Coverage
Generate Compressed Coverage Mask
Texture Unit
Stroke, Fill, Color, etc...
Compositing

Figure 3: *The different stages in GPU-based rendering of filled 2D regions using coverage masks. The only part that takes place on the GPU is the compositing. Our contribution is the stage outlined in red, where compressed textures are generated directly from the RLE coverage information. In doing so, we avoid both writing a full resolution texture into CPU memory and uploading a full resolution texture to GPU memory, providing savings on both ends.*

to the GPU. Even if the time saved by uploading a compressed representation is lost during the encoding step, we still gain memory savings from using compressed textures.

The input to our algorithm is a list of 2D curves defined using Bézier control points. From this list, our goal is to generate an accurate two-dimensional grid of pixels that best approximate the curve along with a specified *paint*. The paint determines the color and opacity of the pixels that are covered by the curve along with any other special operations such as anti-aliasing and gradient dithering. For pixels that are partially covered, they will be painted proportional to the amount that they are covered by the path. In a GPU-based rasterization pipeline, the coverage information is first generated and then used as a texture along with the paint to write to the framebuffer.

There are two operations commonly used for rasterizing paths. First, the path may be *filled* such that a single color is painted within the bounds defined by the path. In this case, the coverage information in conjunction with the paint opacity is used to determine how much of that color should be blended with the background color. If the path is being rendered using the GPU, the coverage information must be uploaded as a texture prior to determining the final color and blending. The other operation, known as *stroking*, draws an outline of a given thickness along the path. In this case, the Skia library computes a new path along the outline of the stroke. Rendering this new path filled with the stroke color is identical to rendering the original stroked path. We restrict our formulation to non-convex paths. Convex paths can be efficiently drawn on GPUs by using a triangle fan in conjunction with the stencil buffer in a modified Loop-Blinn method described in Section 2.2 [Google 2014].

The texture uploaded to the GPU is the image that stores the pixel coverage information. We proceed by first describing a variety of compression methods that we use to encode grayscale information on commodity graphics hardware. We then describe how we augment the scan conversion process to rows of compressed texture data.

3.1 Compression Formats

Due to the large schism of hardware support for various texture compression formats, our goal is to develop an approach that is portable between different GPUs. Decoding algorithms tend to be relatively simple because of the necessity of hardware-based implementations of GPU-encoded textures. Our encoding algorithm exploits this simplicity inherent in all compression formats. As described in Section 3.2, neighborhoods of pixels in coverage masks usually contain either fully transparent or fully opaque pixels. This allows us to precompute many of the parameters for our compression formats prior to the actual encoding. However, the reconstruction of the coverage information from these formats is necessarily lossy, due to the nature of the random access constraints. The following is a detailed overview of the algorithm applied to the DXTn, ETC2, and ASTC families of compression formats.

3.1.1 DXTn

In the DXT family of texture compression formats, introduced by Iourcha et. al. [1999], 4×4 pixel blocks are encoded by storing two pixel values per block and a two-bit index per pixel. The two separate pixel values stored in the block generate a palette of colors from which the per-pixel index selects the final color. The palette is based on intermediate values chosen by linearly interpolating the two stored pixels. For coverage information, we use the DXTn format designed specifically for grayscale known as LATC, or Luminance-Alpha Texture Compression (also known as RGTC, 3DC, and BC4). This format supports two eight-bit grayscale values and sixteen three-bit index values per pixel for a total of 64 bits per block, giving a compression ratio of two-to-one for grayscale images. In order to reach the full range of grayscale values, we store 0 and 255 as endpoints for our coverage mask. Due to the indexing scheme of DXTn, the mapping of coverage values to interpolation indices can not be directly taken from the high three bits of each coverage value. We first quantize each grayscale value to three bits such that their reconstruction into eight bits by bit replication minimizes the error from the original grayscale value. Once these three bits are computed, we must use a mapping from the quantized bits to the proper DXTn indices

$$0, 1, 2, 3, 4, 5, 6, 7 \rightarrow 1, 7, 6, 5, 4, 3, 2, 0.$$

This mapping can be performed without branches on commodity hardware using eight bits per index. If we treat each block row as four 8-bit grayscale values, we can store an entire block row in a single 32-bit register. Furthermore, 32-bit integer operations can be used to perform byte-wise SIMD computations without requiring special SIMD hardware.

3.1.2 ETC2

One variant of the ETC2 compression format is a table-based compression algorithm that takes 4×4 blocks of grayscale pixels, and reconstructs 11-bit grayscale values from 64-bit encoded data, giving a two-to-one compression ratio similar to DXTn. The procedure by which the coverage value for pixel c_i is reconstructed is

$$c_i = b \times 8 + 4 + (\mathbf{T}_v)_{t_i} \times 8,$$

where the encoded data stores an 8-bit base codeword b, a 4-bit multiplier m, a 4-bit modulation index v, and sixteen 3-bit indices t_i. $\mathbf{T}$ is a table containing sets of modulation values constant across all the encodings. This table has sixteen entries, indexed by v. Each t_i selects a final modulation value from the set $\mathbf{T}_v$. The result c_i is then clamped to the range [0, 2047].

To compress the grayscale coverage information, we first fix values for v, b, and m such that they generate the tightest bounds to the entire range of grayscale values. We compute these values by performing an exhaustive search through all possible combinations of v, b, and m offline. In order to compress the coverage information, we perform a quantization to three bits as described in Section 3.1.1. However, due to the indexing method of ETC2, we must use a different mapping

$$0, 1, 2, 3, 4, 5, 6, 7 \rightarrow 3, 2, 1, 0, 4, 5, 6, 7.$$

This mapping is also admits the same implementation advantages as DXTn.

3.1.3 ASTC

Finally, we demonstrate fast compression of our coverage information using the ASTC format introduced by Nystad et. al [2012]. This format has a variable block size that must be chosen prior to compression, and we have noticed that even at the highest compression rate, 12×12, rendering artifacts were negligible. This is possible due to the high compressibility resulting from the low entropy of the coverage mask described in Section 3.2.

ASTC encoded blocks may choose from many different compression options. One such option is whether or not to partition the block into separate subsets of pixels with different compression parameters. Similar to DXTn and ETC2, ASTC uses per-pixel indices to reconstruct the block of pixels. However, there may be fewer indices than pixels, in which case the indices are stored in a grid and interpolated across the block. Finally, similar to DXTn, ASTC reconstructs pixels by using generated indices to lookup palette entries. However, ASTC allows the block encoding to choose how many bits are allocated towards endpoint representation versus index representation.

In order to maximize the fidelity of the ASTC compressed coverage mask, we outline a list of the choices that we made for each 12×12 block of pixels. The main insight is to maximize the number of pixel index values and their bit depth. We are able to maximize the index size because the endpoints must cover the full range of grayscale values and hence require very few bits. For this reason, we are able to generate a valid ASTC encoding using the following choices:

- 6×5 texel index grid to maximize the number of samples in a 12×12 pixel block
- Three bits per texel index
- Single plane encoding (redundant due to single-channel input). This is chosen because we do not use multi-channel pixels
- Only one color endpoint mode: direct luminance
- Single partition encoding with two 8-bit endpoints: 0, 255

Using these constants for all coverage information, there is no special need for the base-three and base-five integer sequences supported by ASTC [Nystad et al. 2012]. Since we know the dimensions of the grid versus the dimensions of the block size, we can precompute the amount that each pixel contributes to each index, and store this in a look-up table. During compression, for each texel grid index we store the top three bits of a weighted average of the pixels that are affected by the index. The final result is 144 grayscale pixels compressed into 128 bits, providing a compression ratio of nine to one. Although compression of ASTC is slower than DXTn and ETC2, the generated compressed textures are significantly faster to load into GPU memory.

3.2 Scan conversion

From a given path, coverage information for each pixel is computed by sampling the path N times per pixel, commonly $N = 16$ with the samples arranged in a regular grid (Figure 4). Each sample is applied a boolean value $b_i \in \{0, 1\}$ such that the final coverage for a given pixel in image $\mathbf{I}$ is

$$\mathbf{I}(x, y) = \frac{1}{N} \sum_{i=1}^{N} b_i.$$

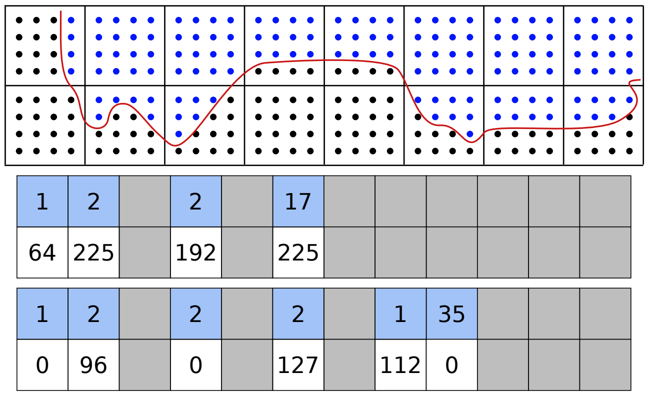


Figure 4: *Sparse run length encoded (RLE) buffers. These buffers are used to store the coverage information for a row of pixels prior to writing them into the coverage mask. For each pixel row, the RLE buffer is allocated to contain as many RLE entries as there are pixels. The scan converter operates on rows of super-sampled pixels, shown here as a 4×4 grid within each pixel, and updates the corresponding RLE buffer. In this figure, the blue entries contain the number of runs of the corresponding pixel value. Grey entries are uninitialized and never written to nor read. Samples which contribute to the coverage of the red curve are drawn in blue and samples that are uncovered are drawn in black.*

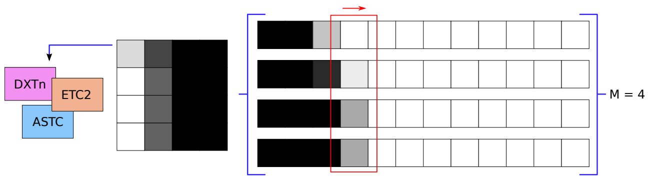


Figure 5: *Our scan conversion pipeline augmented to output GPU-compressed blocks. For $M \times M$ compressed block sizes, our pipeline operates on M sparse RLE buffers in parallel (Figure 4). Once M columns are processed, they are compressed into the target compressed format. For a given column, we read from the entries in the associated sparse RLE buffers. If any of the row values have changed, we update the corresponding pixel for the current column (outlined in red). Otherwise, we simply copy the previous column. For 8-bit coverage values and 4x4 compressed block sizes, each column fits in a single 32-bit register.*

For a value corresponding to $N = 16$, this implies that $\mathbf{I}$ can take up to 17 possible values for any $(x, y) \in \mathbf{N} \times \mathbf{N}$.

In a scanline of samples, the edges of the curve can be computed analytically in order to properly set the corresponding b_i. As shown in Figure 4, the per-pixel coverage information, i.e. the number of samples covered by the path, is stored in a sparse run-length encoded (RLE) buffer. This buffer is updated for each new scanline of samples within a row of pixels. The sparsity of the buffer prevents unnecessary allocation when an initial scanline of samples is altered by a subsequent scanline. In this situation, the samples within a pixel may be identical in the first scanline of samples but different in the second.

The pixels containing intermediate values, i.e. those that are neither fully opaque (covered) or transparent (uncovered), are only found along the boundaries of the 2D path. For this reason, a majority of the pixels in a coverage mask take extremal values (0 or 255) and very few, along the edges of the path, tend to have intermediate values. This means that most of the image can be stored as a binary

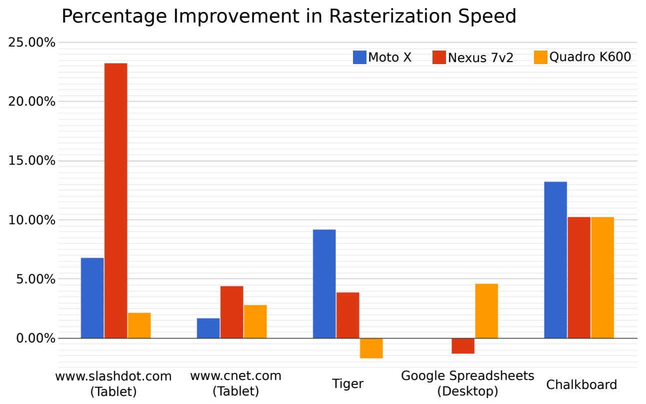


Figure 6: *Performance improvements using compressed textures on a variety of different benchmarks. Two of the tests performed were on tablet versions of popular websites. The Google Spreadsheets benchmark data was gathered from the desktop version of the site using many stroked paths. The other two were the vector images in Figure 7.*

image, producing an entropy close to one [Shannon 1948]. This extremely low entropy property of coverage masks makes them highly compressible.

In order to generate compressed textures, we must adhere to the random access requirements in texture representations. Random access ensures the renderer that it has access to all pixels regardless of when they are needed. This requirement implies a fixed block size for each compression format: 4×4 for DXTn and ETC, and 12×12 for ASTC. Once a scanline of pixels is computed, it can be stored in a row of an 8-bit grayscale texture. We generate compressed representations of the grayscale textures by consuming M rows of run-length encoded data at a time, where M is the dimension of the (square) block size of the texture compression format. As shown in Figure 5, we read the leftmost column of grayscale values and update the corresponding byte as we walk down our M RLE buffers. At each step, we advance to the column with the earliest ending run length. Once we advance past M columns, we efficiently compute a compressed representation of the $M \times M$ block that we have read from the RLE buffers, as described in Section 3.1. For the most common case, $M = 4$, the four grayscale values are represented as a 32-bit integer, and we can perform SIMD byte-wise operations using integer shifts and adds. As an optimization, if we advance the current column farther than M pixels at once due to the RLE encoding, we can copy the previous block encoding into its neighbor to the right.

4 Results

To test our results, we have integrated our real-time compression pipeline into the 2D graphics library Skia [2014]. This library is used as the backbone to many cross-platform 2D programs and operating systems including Android, Google Chrome, and Mozilla Firefox. In order to maintain performance and regression tests across all platforms, Skia includes two types of comprehensive tests. For any given change to the implementation, Skia tests the new rendered image against existing baseline images. If any pixels differ by a significant amount, these tests fail and the change is invalid. The second test measures performance against a suite of microbenchmarks and a suite of rendering commands that are invoked during the rendering of common web pages. In order for these tests to pass, their running time must be within a small thresh-

Mobile Platform	CPU	GPU	Uncompressed	Compressed	Texture Format	Memory Benefit
Moto X	1.7 GHz Qualcomm Krait	Qualcomm Adreno 320	163ms	137ms	ETC2	2:1
Galaxy Note 3	1.9 GHz ARM Cortex-A15	ARM Mali-T628	171ms	161ms	ETC2	2:1
HTC One M8	2.3 GHz Qualcomm Krait 400	Qualcomm Adreno 330	114ms	102ms	ETC2	2:1
Galaxy Note 10.1	1.9 GHz ARM Cortex-A15	ARM Mali-T628	171ms	136ms	ASTC	9:1
Galaxy S5	1.3 GHz ARM Cortex-A7	ARM Mali-T628	311ms	157ms	ASTC	9:1

Table 1: *The rendering times for the polygon benchmark (Figure 7) from Skia using both compressed and uncompressed texturing on a variety of CPU/GPU combinations. The polygon benchmark generates a large sequence of thin, concave polygons and stores them as piece-wise 2D paths on the GPU. These polygons are then both stroked and filled to generate a large amount of paths that must be rasterized. From these results, we notice an increase in rendering speed of the heavily optimized Skia library on all mobile devices. Most importantly, the increase in memory efficiency from ETC2 (2:1 ratio) to ASTC (9:1 ratio) provides significant improvements in rendering time. These results were generated from the mean runtime of 100 executions.*

old of the previously passed test. In each of our examples, we have maintained both correctness and performant code with respect to the existing implementations.

First, we must show that our implementation runs fast on modern hardware. In Figure 6, we show different classes of benchmarks that have been run on a variety of different mobile GPUs. In each case, we see a general increase in the rendering speed of certain web pages and common vector graphics benchmarks. As we can see, the desktop GPU does not receive as much of a benefit from the compression routine as the mobile GPUs. The authors speculate that mobile GPUs are more sensitive to transmitting large amounts of data from the CPU to the GPU due to power restrictions and hence receive more benefits. Mobile GPU performance increases are better demonstrated in Table 1 where various mobile GPUs render the polygon image (Figure 7) from the Skia performance tests. From this table, we observe that both CPU speed (Galaxy Note 10.1 vs Galaxy S5) and compression ratio (Galaxy Note 10.1 vs Galaxy Note 3) play a vital role in rendering performance on mobile devices.

In order to test correctness, we perform both a visual comparison against the reference images (without compression) and measure the difference using the *Peak Signal to Noise Ratio*, or PSNR:

$$PSNR = 10\log_{10}\left(\frac{3 \times 255^2 \times w \times h}{\sum_{x,y}\left(\Delta R_{xy}^2 + \Delta G_{xy}^2 + \Delta B_{xy}^2\right)}\right)$$

In Figure 8, we compare the various use cases of rendered paths and the difference in their rendering. We observe that only pixels along the borders of the paths are affected by the compression scheme. This homogeneity in the coverage masks is the primary reason why they are highly compressible. From the zoomed in comparisons, we notice that there is little to no quality loss in the final images. However, the pixels that differ do so by a non-trivial amount. This difference causes the relatively low PSNR values calculated for the images.

From the performance and quality results, we observe a benefit to compressing coverage masks prior to usage, with little visible loss in quality. The method described in Section 3 that yields these results relies heavily on 32-bit integer operations but is otherwise portable to a wide variety of platforms. These performance metrics also do not take into account the possible benefits from multi-threading approaches. Although these methods are highly parallelizable, the main benefit is reducing the latency of uploading the coverage masks to the GPU. Hence, any GPU compression method that would require the data uploaded prior to compression would lose this benefit. However, if the coverage information is generated on the GPU, then our method could be used to compress the mask very quickly using only a handful of low-latency integer operations.

5 Conclusion, Limitations, and Future Work

In this paper we have shown that coverage masks used for rendering 2D anti-aliased non-convex paths are perfect candidates for real-time compression. Their low-entropy properties make compression algorithms very efficient and the masks themselves highly compressible. We have also shown that these masks can be compressed in real-time often speeding up the rendering of 2D curves and saving valuable GPU memory.

Limitations: Although the coverage masks can be compressed effectively, GPU-based methods for rendering arbitrary 2D-curves with anti-aliasing are still inferior to their CPU-based counterparts. In general, generating the coverage mask is by far the most expensive operation of the rasterization procedure. During CPU-based rendering, the rasterizer can perform the shading directly from the RLE buffer discussed in Section 3.2. This limitation can be observed from the time it takes to run the polygon benchmark from Table 1 on different platforms using the software renderer:

Rendering time for convex path benchmark *strokedrects*

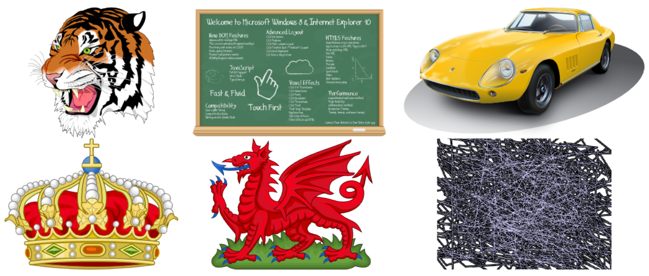
Platform	GPU	CPU
Moto X	6.9 µs	37.6 µs
Galaxy Note 10.1	3.76 µs	15.5 µs

Rendering time for non-convex path benchmark *polygon*

Platform	GPU		CPU
	Uncompressed	Compressed	
Moto X	163ms	137ms	83ms
Galaxy Note 10.1	171ms	136ms	46ms

However, many of the applications that require 2D rendering operate on many more primitives than non-convex 2D curves. In the table above, the GPU-based convex path rendering operation still outperforms its CPU counterpart. For this reason, it is advantageous to use a GPU-based framebuffer. As such, our method provides benefits to the least efficient aspect of GPU-based resolution independent graphics rendering.

Future Work: We have shown that coverage masks are very amenable to compression. Due to the very high fidelity of the rendered images even at the highest available compression ratios (12×12 ASTC) there is ample room for even more aggressive com-



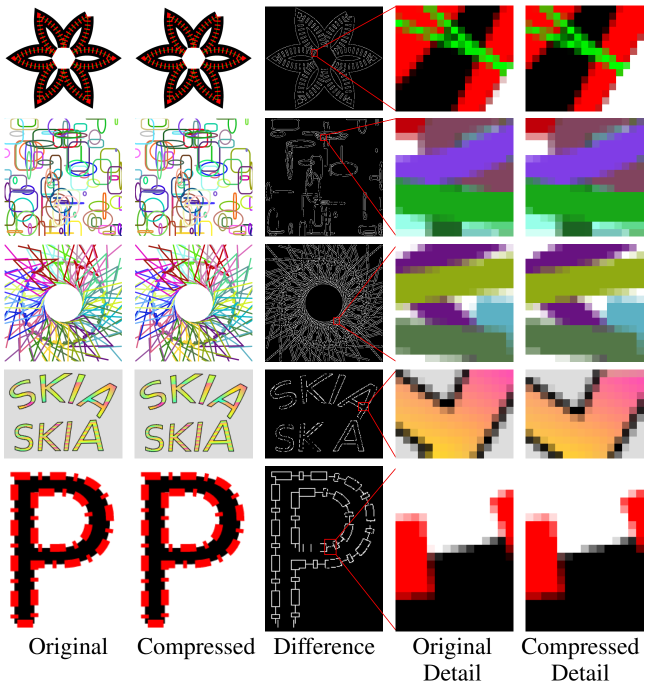
Uncompressed

Image	Min	Median	Mean	Max	σ
Tiger	95.3ms	96.6ms	97.8ms	109ms	3ms
Chalkboard	358ms	370ms	371ms	473ms	5ms
Car	368ms	385ms	385ms	403ms	2ms
Crown	121ms	127ms	137ms	200ms	15ms
Dragon	92ms	94.3ms	96ms	140ms	7ms
Polygon	149ms	152ms	154ms	208ms	5ms

Compressed

Image	Min	Median	Mean	Max	σ
Tiger	81ms	83ms	83ms	93ms	2ms
Chalkboard	339ms	349ms	350ms	495ms	5ms
Car	364ms	387ms	386ms	424ms	3ms
Crown	106ms	109ms	111ms	168ms	9ms
Dragon	87ms	92.2ms	101ms	156ms	19ms
Polygon	133ms	134ms	137ms	194ms	7ms

Figure 7: *Rendering times and resource use of the following images on a first generation Moto X (1.7 GHz Qualcomm Krait, Qualcomm Adreno 320) from 100 runs. From left to right the images are labeled Tiger, Chalkboard, Car, Crown, Dragon, Polygon.*



PSNR	dashed	rounded	poly	text	strokep
	35.457	41.028	35.457	37.736	53.876

Figure 8: *Detailed analysis of correctness tests within Skia most heavily affected by changes to anti-aliased non-convex path rendering. From top to bottom, the images are labeled as 'dashed', 'rounded', 'poly', 'text', and 'strokep'. We observe very few artifacts due to compression. Although the pixels along the anti-aliased edges in the rendered images do contain different pixel values contributing to the relatively low PSNR values, the detail in the edges remains. Pixels in the difference image are on if the shaded values in the corresponding original and compressed images differ. Most noticeable in 'strokep', the low entropy of the coverage masks causes pixel differences only in those along the edges of the filled paths.*

pression formats. Encodings that support block dimensions up to 32 or 64 may still produce nice results. The compression algorithms in Section 3.2 can be extended to support even better compression ratios, which will increase both the rendering speed and memory usage. Another direction for research is the ability to generate coverage information on the GPU itself. If such a technique existed, the compositing procedure using the coverage mask could be done at the same time as generating the coverage information itself. However, if the coverage mask were generated on the GPU and then used as input to a second compositing pass, compressing the GPU-generated coverage masks using this technique would incur trivial cost. Due to the random-access restrictions of compressed texture formats, they are perfect candidates for massively parallel encoding. Furthermore, to combat the original artifacts from the Blinn-Phong method, conservative rasterization may be used to cover every pixel touched by the bounding triangles [Akenine-Möller and Aila 2005]. Such a solution could eliminate the need for CPU-side rendering entirely.

6 Acknowledgements

This research is supported in part by ARO Contract W911NF-14-1-0437, Intel and Samsung. The authors would also like to thank Robert Phillips, Mike Reed, and Brian Salomon for their help and guidance in understanding and interfacing with the Skia library.

References

AILA, T., MIETTINEN, V., AND NORDLUND, P. 2003. Delay streams for graphics hardware. *ACM Trans. Graph. 22*, 3 (July), 792–800.

AKENINE-MÖLLER, T., AND AILA, T. 2005. Conservative and tiled rasterization using a modified triangle set-up. *J. Graphics Tools 10*, 3, 1–8.

BARROS, J., AND FUCHS, H. 1979. Generating smooth 2-d monocolor line drawings on video displays. *SIGGRAPH Comput. Graph. 13*, 2 (Aug.), 260–269.

BEERS, A. C., AGRAWALA, M., AND CHADDHA, N. 1996. Rendering from compressed textures. In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, ACM, SIGGRAPH '96, 373–378.

FATAHALIAN, K., BOULOS, S., HEGARTY, J., AKELEY, K., MARK, W. R., MORETON, H., AND HANRAHAN, P. 2010. Reducing shading on gpus using quad-fragment merging. *ACM Trans. Graph. 29*, 4 (July), 67:1–67:8.

FIUME, E., FOURNIER, A., AND RUDOLPH, L. 1983. A parallel scan conversion algorithm with anti-aliasing for a general-purpose ultracomputer. *SIGGRAPH Comput. Graph. 17*, 3 (July), 141–150.

GOOGLE, I., 2014. Skia – 2d rendering library. `https://sites.google.com/site/skiadocs/`.

GREEN, C. 2007. Improved alpha-tested magnification for vector textures and special effects. In *ACM SIGGRAPH 2007 Courses*, ACM, New York, NY, USA, SIGGRAPH '07, 9–18.

IOURCHA, K. I., NAYAK, K. S., AND HONG, Z., 1999. System and method for fixed-rate block-based image compression with inferred pixel values. U. S. Patent 5956431.

KAUTZ, J., LEHTINEN, J., AND AILA, T. 2004. Hemispherical rasterization for self-shadowing of dynamic objects. In *Proceedings of the Fifteenth Eurographics Conference on Rendering Techniques*, Eurographics Association, Aire-la-Ville, Switzerland, Switzerland, EGSR'04, 179–184.

KILGARD, M. J., AND BOLZ, J. 2012. Gpu-accelerated path rendering. *ACM Trans. Graph. 31*, 6 (Nov.), 172:1–172:10.

KOKOJIMA, Y., SUGITA, K., SAITO, T., AND TAKEMOTO, T. 2006. Resolution independent rendering of deformable vector objects using graphics hardware. In *ACM SIGGRAPH 2006 Sketches*, ACM, New York, NY, USA, SIGGRAPH '06.

KRAJCEVSKI, P., LAKE, A., AND MANOCHA, D. 2013. FasTC: accelerated fixed-rate texture encoding. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, ACM, I3D '13, 137–144.

LANE, J. M., AND M. RARICK, R. A. 1983. An algorithm for filling regions on graphics display devices. *ACM Trans. Graph. 2*, 3 (July), 192–196.

LOOP, C., AND BLINN, J. F. 2005. Resolution independent curve rendering using programmable graphics hardware. In *July 2005 Transactions on Graphics (TOG) Volume 24 Issue 3 (Siggraph 2005)*, Association for Computing Machinery, Inc.

NYSTAD, J., LASSEN, A., POMIANOWSKI, A., ELLIS, S., AND OLSON, T. 2012. Adaptive scalable texture compression. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on High Performance Graphics*, Eurographics Association, HPG '12, 105–114.

QIN, Z. 2009. *Vector Graphics for Real-time 3D Rendering*. PhD thesis, University of Waterloo.

SHANNON, C. E. 1948. A mathematical theory of communication. *The Bell System Technical Journal 27* (July, October), 379–423, 623–656.

SHEBANOW, M. C. 2014. The evolution of mobile graphics and the potential impact on interactive applications. In *Keynote Address of the ACM SIGGRAPH Symposium on Mobile Graphics and Interactive Applications*, ACM, SIGGRAPH ASIA '14.

STATCOUNTER, I., 1999-2014. Global stats, top 5 desktop, tablet & console browsers from sept 2013 to sept 2014. `http://gs.statcounter.com`.

STRÖM, J., AND AKENINE-MÖLLER, T. 2005. iPACKMAN: high-quality, low-complexity texture compression for mobile phones. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, ACM, HWWS '05, 63–70.

STRÖM, J., AND PETTERSSON, M. 2007. ETC2: texture compression using invalid combinations. In *Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, Eurographics Association, GH '07, 49–54.

WENNERSTEN, P., AND STRÖM, J. 2009. Table-based alpha compression. *Computer Graphics Forum 28*, 2, 687–695.

ZHANG, H., MANOCHA, D., HUDSON, T., AND HOFF, III, K. E. 1997. Visibility culling using hierarchical occlusion maps. In *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques*, ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, SIGGRAPH '97, 77–88.