EUROGRAPHICS 2019 / P. Alliez and F. Pellacini
(Guest Editors)

Volume 38 (2019), Number 2

Hierarchical Rasterization of Curved Primitives for Vector Graphics Rendering on the GPU

Mark Dokter[1,2], Jozef Hladky[2], Mathias Parger[1], Dieter Schmalstieg[1], Hans-Peter Seidel[2], and Markus Steinberger[1,2]

[1]Graz University of Technology, Austria
[2]Max-Planck-Institut Informatik, Saarland Informatics Campus, Germany

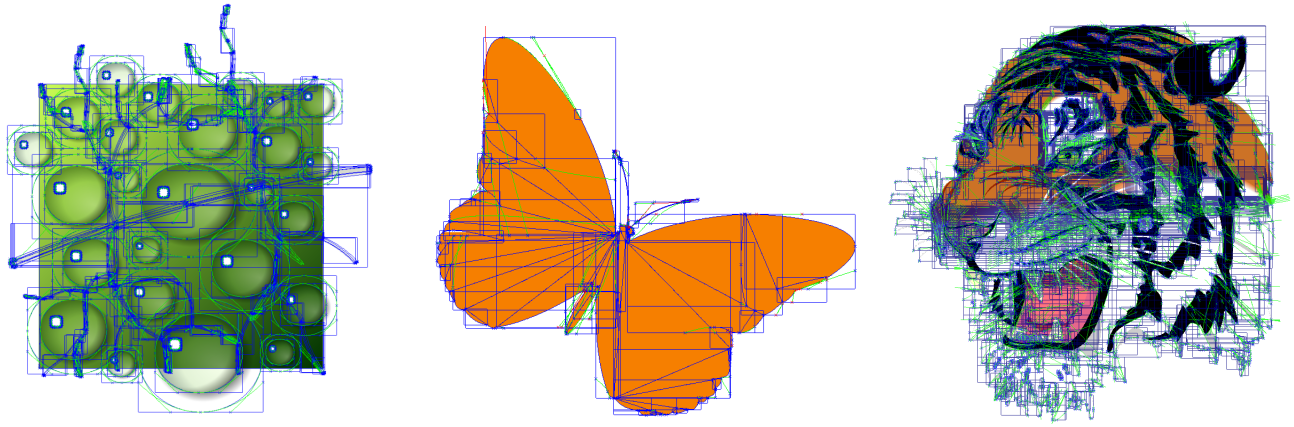


Figure 1: *Three simple vector graphics constructed from curved patches (CPatches). All CPatches are indicated with their bounding box in blue. For efficient rasterization, auxiliary curve are added during patch cutting. Patch outlines and auxiliary curves are shown in blue and green. CPatches are rendered by our hierarchical rasterizer completely in parallel on the GPU, leading to superior performance and flexibility compared to previous work.*

Abstract

In this paper, we introduce the CPatch, a curved primitive that can be used to construct arbitrary vector graphics. A CPatch is a generalization of a 2D polygon: Any number of curves up to a cubic degree bound a primitive. We show that a CPatch can be rasterized efficiently in a hierarchical manner on the GPU, locally discarding irrelevant portions of the curves. Our rasterizer is fast and scalable, works on all patches in parallel, and does not require any approximations. We show a parallel implementation of our rasterizer, which naturally supports all kinds of color spaces, blending and super-sampling. Additionally, we show how vector graphics input can efficiently be converted to a CPatch representation, solving challenges like patch self-intersections and false inside-outside classification. Results indicate that our approach is faster than the state-of-the-art, more flexible and could potentially be implemented in hardware.

CCS Concepts
• ***Computing methodologies*** → ***Rasterization;*** • ***Theory of computation*** → *Massively parallel algorithms;*

1. Introduction

Vector graphics precede raster graphics as a representation of digital content, yet, remain relevant today, since a resolution-independent representation allows artifact-free display on everything from a tiny smartwatch to a huge wall-size display. Consequently, vector graphics are ubiquitous in all kinds of data visualization, including font rendering, user interfaces, web pages, diagrams, charts, maps, games, and artistic illustrations.

However, vector graphics representations have not radically departed from the seminal work of Warnock and Wyatt [WW82]. Vector graphics are typically defined as a collection of paths, where each path is defined by a number of curves. Curves are commonly defined as straight lines, quadratic or cubic Bezier curves, or circular segments. A closed path separates an interior and exterior; the interior and the path itself can be filled using a variety of styles and patterns.

Unfortunately, efficient rendering of vector graphics at high resolutions still forms a challenging task for computer graphics. Rendering on the CPU does not scale well to high resolutions and super-sampling. Consequently, parallel vector graphics rendering has been actively researched over the last decades [LB05, KB12, GLdFN14, BKKL15, LHZ16]. However, there is still no parallel approach for vector graphics rendering which comes close to the elegance and efficiency of triangle rasterization.

Recent approaches typically use two steps: *stencil, then cover* [LB05, KB12, LHZ16]. A first step determines which (sub-)pixels are inside a patch. A second step evaluates the actual shading of the marked pixels. The stencil generation is the costly step of the two, either involving a large number of overlapping triangles to modify the stencil [LB05, KB12] or scanline-curve intersections to generate bit masks [LHZ16].

In this paper, we propose a novel approach for vector graphics rendering in a single parallel rendering pass. Inspired by polygon rasterization, we propose a new primitive, the curved patch (*CPatch*), which is limited by a number of curves, each dividing the space into a positive and a negative half-space. The union of all positive half-spaces defines the inside of a CPatch, similar to the use of edge equations in polygon rasterization. Consequently, a CPatch can be seen as a generalization of a polygon. Even though representing the interior of a path might require multiple CPatches (Figure 1), all CPatches can be processed in parallel, leading to a very efficient algorithm. Hence, we make the following contributions:

- We introduce CPatches and their mathematical description.
- We derive a parallel, hierarchical rasterization approach that is efficient to evaluate and very fast on current GPU hardware.
- We show how CPatches can efficiently be constructed and how arbitrary vector graphics can be translated into a collection of CPatches.

An evaluation of our approach on modern GPU hardware indicates that it outperforms previous GPU solutions by a factor of 1.17× to 1.80× on average.

2. Related work

Previous curve rasterization techniques can be roughly classified into three categories: (1) scanline filling methods, (2) 'stencil, then cover' approaches, and (3) alternative vector graphics representations, such as data structures supporting spatial queries.

2.1. Scanline methods

Early scanline algorithms focus on rendering triangles [WREE67] and construct spans limited by pairs of edge-scanline intersections. To increase the efficiency of scanline algorithms, the intersections of a scanline with all edges can be computed before sorting and filling [NS79, AW81].

CPU scanline algorithms for vector graphic rendering are found in contemporary curve rendering packages such as Skia [Goo18] or Cairo [PWE18], which are used if no appropriate GPU accelerated alternative is available. Manson and Schaefer [MS13] used pixel-sized scanlines to implement analytic shading and anti-aliasing filters. To increase performance, spans can be merged and clipped for hidden surface removal, as shown by Whitington [Whi15].

While scanline approaches are usually designed for the CPU, Li et al. [LHZ16] recently showed that a GPU scanline algorithm can also be efficient. Their approach first builds an acceleration data structure of potential scanline-curve intersections and then evaluates them in parallel with simplified geometry. A final step rendered the generated spans using traditional OpenGL. The efficiency of this algorithm comes from the fact that not every filled pixel must be tested against the path. CPatch has the same advantageous property, while requiring only a single pass.

2.2. Stencil, then cover

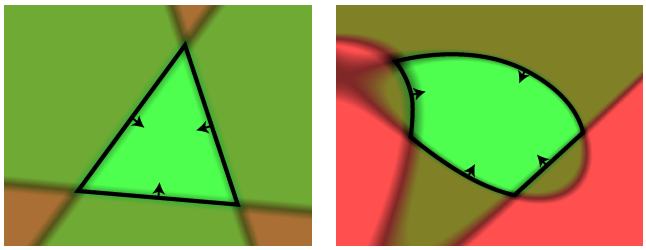
Many GPU curve rendering approaches follow a 'stencil, then cover' approach, where a mask is first generated for a path (stencil) before filling, while blending happens in a second step (cover). Stencil generation goes back to the work of Loop and Blinn [LB05], which allows to efficiently determine on which side of a curve a sample lies. In combination with Jordan's theorem [FSF97], fill rules can be computed in a discrete manner, which allows complex path stencils to be generated by rendering multiple overlapping triangles with implicit curve descriptions [KSST06, KB12]. While hardware support makes this method fast, it still requires a per-path multi-pass algorithm that potentially touches many samples which are not part of the final stencil. Note that the scanline approach by Li et al. [LHZ16] can be classified as "stencil, then cover" as well.

Several extensions exist: For example, the 'stencil, then cover' method used in Adobe Illustrator [BKKL15] extends color schemes and blending modes. Tile-based rendering of stencils [YLK*15] runs efficiently on mobile devices.

Like these approaches, our method classifies half-spaces, but it directly renders paths from CPatches, rather than using a separate cover pass. In that sense, our approach is closer to the original method of Loop and Blinn [LB05]. However, their method only supported a single curve per primitive, which makes it prohibitively complicated to construct complex shapes, like thin parallel curves. Our approach supports multiple curves and draws further efficiency from hierarchical rasterization.

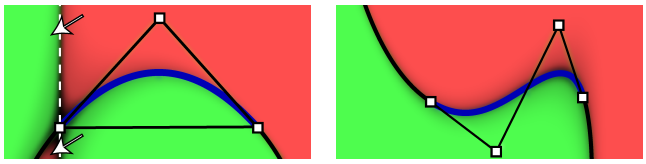
2.3. Alternative representations

Various alternative representations of vector graphics have been proposed. Motivated by rendering vector graphics on top of surfaces, vector texture methods try to encode sharp features in regularly sampled textures. Feature curves [PZ08] encode distances to quadratic Bezier curves and can thus render a limited number of sharp curves intersecting at one location. Precise vector textures [QMK08] encode the distances to monotonic curve segments in the texture. As long as the distance to the evaluated curves is not larger than the curve's curvature, the method delivers error-free results. Vector solid textures [WZYG10] use radial basis functions as primitive to construct sharp features. All the above representations allow highly flexible display transformation and are efficient to render using texture hardware. However, they cannot represent arbitrary vector graphics due to their limitation to the sample grid of the underlying texture or curves.



(a) *Triangle* **(b)** *CPatch*

Figure 2: *(a) Rasterization of a triangle classifies samples as inside a triangle, if all edge equations classify them as inside (green). (b) CPatches are constructed in the same spirit, with implicit curve equations classifying parts as inside.*



(a) *Quadratic* **(b)** *Cubic*

Figure 3: *(a) The implicit form for a quadratic Bézier curve shows a sharp edge (white dashed line) outside the control polygon, which inverts the function (arrows). (b) The implicit form of a cubic follows the curve extension (black).*

Nehab and Hoppe [NH08] use an adaptive lattice structure to describe vector graphics with appropriate detail where needed. They support distance evaluation in the lattice cells to linear, radial and quadratic curves. Cubic curves are not supported. While their lattice generation is carried out on the CPU, the rendering is performed on the GPU. Their lattice structure reduces the number of curves that need to be tested for each fragment, but still requires that all pixels within a potentially large cell be tested against all curves of that cell.

Shortcut trees [GLdFN14] allow efficient indexing into vector graphics. They can be built on the GPU and support cubic curves by monotonizing curve segments. While their tree representation is elegant and relatively fast to build, the resulting rendering performance can compete with hardware-supported 'stencil, then cover' strategies only at very high resolutions.

Diffusion curves [OBW*08, FSH11, STZ14] let a designer construct path outlines that implicitly control the color of the interior through a simulated diffusion process. This approach lends itself to parallel solving, but remains very computationally demanding overall. Our method does not build an auxiliary data structure, but converts the vector graphics data entirely into a set of new primitives supporting an object-order approach, rather than being constrained to image-order.

3. CPatch: A novel curved primitive

Our approach is based on CPatches—primitives limited by cubic curves (see Figure 1 for examples). In principle, a primitive with curved boundaries can be treated in the same way as a polygon (Figure 2b). For a polygon, inserting into all line equations lets one determine whether a sample is inside (as shown in Figure 2a). Salmon [Sal79] as well as Loop and Blinn [LB05] show how to translate quadratic and cubic Bézier curves into implicit form to determine on which side of a curve a sample lies: Depending on the type of the curve, three parameters k, l, m are computed for each control point. A linear interpolation of these parameters and evaluation of a simple cubic function

$$f_c(x,y) = k^3(x,y) - l(x,y) \cdot m(x,y)$$

yields positive values for one side of the curve and negatives for the other. As the factors only need to be interpolated linearly, the approach is well suited for GPU execution. For the classification of curves into 'serpentine', 'cusp', and 'loop' and the complete table of interpolation factors, see Loop and Blinn [LB05].

However, the implicit function can only be used for this half-space classification within the convex hull of the curve's control points. When extending a curve to $\pm\infty$, it may reach inside the CPatch and lead to an incorrect classification of sample points. One could avoid this problem by limiting patches to the convex hull of all curves, but at the cost of limiting the supported patch types to the single-curve approach of Loop and Blinn [LB05]. Representing thin curved objects, such as font characters, would lead to an excessive number of patches. Instead, we split a patch into two when a curve extension reaches into the patch.

Figure 3a shows how, outside the convex bounds for a quadratic curve, one side of the implicit function continues along the extension of the curve (black extension to the right), while the other one changes abruptly (inverting at the white dashed line). In contrast, the extension of an implicit function for a cubic curve essentially follows the curve's extension when running through the parameter from $-\infty$ to ∞, as shown in Figure 3b. This behavior is preferable, as it is more predictable, and the locations of the sign change in the implicit form can be reconstructed from the explicit formulation. Therefore, we elevate all quadratic curves to cubics [Far88] and limit our discussions to the cubic case in the remainder of this paper.

Note that, similar to triangles, CPatches only describe the interior of a primitive and not the shading of the boundaries. Thus, similar to previous work, we do not consider line shading as part of our approach. However, lines can be described by CPatches. For solid strokes, CPatches are easy to derive as two 'parallel' curves in combination with two end curves, which are easy and efficient to rasterize.

4. Hierarchical rasterization of CPatches

Constructing vector graphics from a collection of primitives has multiple advantages: First, all primitives can be treated completely in parallel without any constraints imposed by a multi-pass approach, such as *stencil, then cover*. Second, rendering can be implemented as a streaming pipeline, which keeps resource requirements low. Third, we can establish primitive order to address issues like a correct blending order.

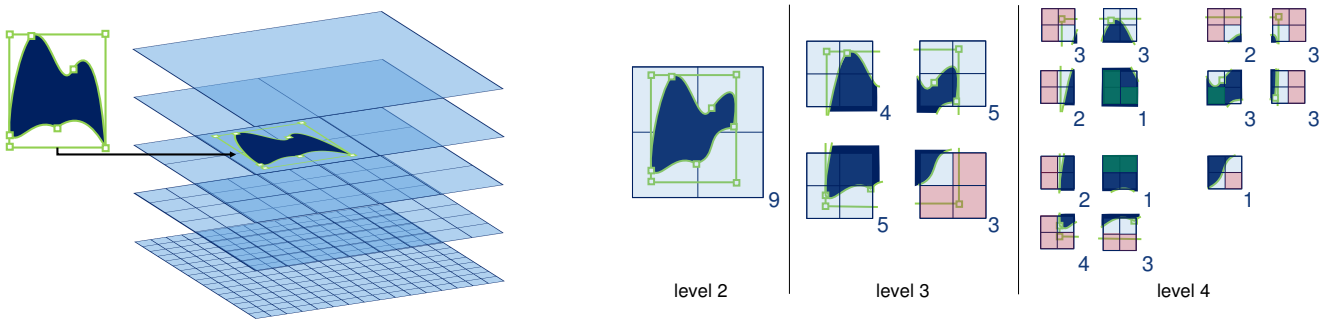


Figure 4: *(left) Our hierarchical tiling starts by choosing the most fitting hierarchy level for the CPatch. (right) We process the patch down to the lowest hierarchy level. Sub-tiles are classified as completely outside (red overlay) or completely inside (green overlay), if the patch does not require any more testing. For tiles that are classified completely inside, the enclosing curve can be removed from further sub-tile testing (blue numbers indicate the number of actives curves).*

4.1. CPatch representation

Before detailing our hierarchical rasterization approach, we need to give an exact definition of a CPatch. We limit CPatches to consist of a predefined maximum number of curves (four to eight curves have proven to work well in our experiments) inside a given bounding box (represented as four lines). We allow curves to be either straight lines or cubics; quadratic curves are elevated to cubics. For straight lines, we encode the line equations in k, l, m form, such that l is the signed normal distance to the line, while $k = 0$ and $m = 1$ everywhere. While this approach slightly increases the evaluations for straight lines, it offers the advantage of a uniform treatment with only slightly increased computations. Note that we treat circular segments separate, as discussed at the end of the section.

To represent curves, we interpolate k, l, and m over the entire space of the patch using homogeneous rasterization [OG97]. We can define k, l, and m for three arbitrary points in space—any three control points are good choices—and store them in vector form:

$$\mathbf{k} = [k_0, k_1, k_2]^T, \quad \mathbf{l} = [l_0, l_1, l_2]^T, \quad \mathbf{m} = [m_0, m_1, m_2]^T.$$

Furthermore, we store the transformation matrix $\mathbf{M}$ that captures the location of the interpolation points in space, at which $[x_0, y_0]^T$ is the location where $k = k_0$, $l = l_0$, and $m = m_0$:

$$\mathbf{M} = \begin{bmatrix} x_0 & x_1 & x_2 \\ y_0 & y_1 & y_2 \\ 1 & 1 & 1 \end{bmatrix}^{-1}.$$

For any sample point $\mathbf{s} = [x, y, 1]^T$, we can interpolate k, l, m:

$$\mathbf{u} = \mathbf{M} \cdot \mathbf{s}, \quad k_s = \mathbf{k}^T \cdot \mathbf{u}, \quad l_s = \mathbf{l}^T \cdot \mathbf{u}, \quad m_s = \mathbf{m}^T \cdot \mathbf{u}.$$

Transformations can be applied by multiplying $\mathbf{M}$ with any 3×3 transformation matrix. While we only consider 2D operations here, it is straight forward to extend our homogeneous rasterization to 3D, as long as patches remain planar. Similarly to the interpolation of k, l, and m, other parameters, like texture coordinates or color gradients, can be stored along a patch.

Commonly, a curve will be shared by multiple CPatches, *e.g.*, to construct a larger complex shape. Therefore, we propose an indirect storage format, similar to indexed triangle meshes. We store each curve separately (**k**, **l**, **m**, **M**), and represent a CPatch as a constant-size array of references to curves, padded with null pointers if necessary. Moreover, the CPatch stores a primitive id to look up additional shading parameters.

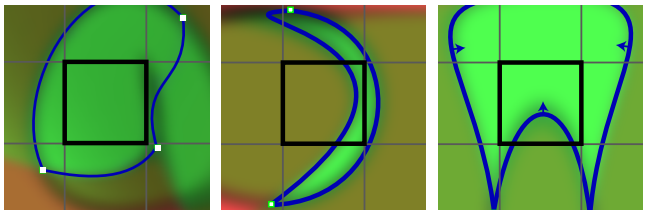
4.2. Tiled rasterization

A naive rasterization of CPatches would evaluate all curve equations for all pixels and fill those that lie in the intersection of all half-spaces. The main cost of such an approach is in the curve equation evaluation, which we would like to reduce as much as possible. Large homogeneous regions, which have the same classification, should be determined without visiting individual pixels. This consideration suggests a divide-and-conquer approach. We would like to concentrate on the regions close to curve boundaries, while the interior area can be filled in a single step.

Hierarchy Our hierarchical tiling approach is illustrated in Figure 4): Starting from the bounding rectangle of the patch, we determine the first level in the hierarchy where a patch should be tested. From there, the hierarchical rasterization removes irrelevant curves, while proceeding through the levels. All tiles of a level are processed in parallel. When the lowest level is reached, a fine rasterization determines the pixel fill state.

In the inner loop of this algorithm, we must determine whether a curve equation is uniformly positive or negative with respect to a given tile. Unfortunately, this test is complicated by the fact that boundaries are not lines, but implicit curves. Hence, testing the corners of a tile is not sufficient, as there is no guarantee that the curve does not change orientation between sample locations, as shown in Figure 5c. Furthermore, there is no efficient closed form solution to determine whether an entire tile is on one side of the curve, as this would require inserting two bounded linear functions into a cubic equation, leading to a higher order polynomial.

Tile evaluation For an efficient alternative solution to the problem, we rely on two facts. Since straight lines and cubic curves extend to infinity, it suffices to ensure that the implicit curves do not change sign along any tile boundary (Figure 5).



(a) *Inside tile* **(b)** *Intersections* **(c)** *Double crossing*

Figure 5: *Our tile rasterizer relies on the fact that curves reach to infinity and thus determining sign changes along the tile boundary is sufficient to identify sign changes within a tile.*

For this purpose, we rely on the intermediate value theorem. By determining the extremal values of a curve equation on the tile boundary, we determine whether there is a sign change.

We evaluate the curve equations at the tile corners and then look for the location of extrema in-between by constructing the interpolation factors along an edge: Let $\mathbf{c}_0$ and $\mathbf{c}_1$ be two corners of the tile edge. We compute the interpolation factors of k, l, m in a 3×2 matrix $\mathbf{I}$:

$$\mathbf{I} = \begin{bmatrix} \mathbf{k}^T \\ \mathbf{l}^T \\ \mathbf{m}^T \end{bmatrix} \cdot \mathbf{M} \cdot \begin{bmatrix} c_{0,x} & c_{1,x} - c_{0,x} \\ c_{0,y} & c_{1,y} - c_{0,y} \\ 1 & 0 \end{bmatrix}.$$

Using $\mathbf{I}$, we can evaluate the curve equation anywhere on the edge by multiplying with $\begin{bmatrix} 1 & i \end{bmatrix}^T$, where i is the relative location between $\mathbf{c}_0$ and $\mathbf{c}_1$. The general equation for evaluating the curve,

$$\begin{aligned} \mathbf{f}_{klm}(i) &= \mathbf{I} \cdot \begin{bmatrix} 1 & i \end{bmatrix}^T \\ f_c(i) &= f_k(i)^3 - f_l(i) \cdot f_m(i) \\ &= (I_{k0} + i \cdot I_{k1})^3 - (I_{l0} + i \cdot I_{l1}) \cdot (I_{m0} + i \cdot I_{m1}), \end{aligned}$$

has the derivative

$$f_c'(i) = 3 \cdot (I_{k0} + i \cdot I_{k1})^2 \cdot I_{k1} - I_{l0} f_{m1} - I_{m0} I_{l1} - 2 i f_{l1} I_{m1}.$$

We set $f_c'(i) = 0$ and directly solve the quadratic equation in i. If the found extrema lie within the tile border bounds ($0 < i < 1$), we evaluate the curve equation at these locations, again using $\mathbf{I}$, and determine the minimum and maximum along each tile border.

Parallel evaluation Performing the above steps individually for all tiles would be inefficient, as the same computations would be repeated many times. Thus, we perform the evaluation on a subgrid of tiles at once. Multiple threads can be employed for this evaluation, as shown in Figure 6 and Algorithm 1: We determine $\mathbf{I}$ for all rows of the grid using one thread per row. In step (1) (line 3–4), each thread evaluates the curve equation for all corners in its row. In step (2) (line 5), it applies the result to the surrounding tiles, updating their min/max. In step (3) (line 6–8), we determine the extrema for each row and update the min/max only for the touched tiles. Finally, we switch to columns and perform the min/max updates as well (line 9–13). This scheme reuses $\mathbf{I}$ for both the corner evaluation and the extrema computation, performing all computations only once for multiple tiles.

```
1  for all curves ∈ CPatch do
2      for all grid rows r in parallel do
3          compute c_{r,0}, c_{r,n} and I to get f_{c_r}(i) for the curve
4          for i ∈ [0,1] with increase 1/(n−1) do
5              evalute f_{c_r}(i) and MinMax to surrounding tiles
6          compute e_i from f'_{c_r}(i) = 0
7          for all 0 < e_i < 1 do
8              evalute f_{c_r}(e_i) and MinMax to surrounding tiles
9      for all grid columns c in parallel do
10         compute c_{c,0}, c_{c,n} and I to get f_{c_c}(i)
11         compute e_i from f'_{c_c}(i) = 0
12         for all 0 < e_i < 1 do
13             evalute f_{c_c}(e_i) and MinMax to surrounding tiles
14     for all tiles in parallel do
15         if Max < 0 then
16             discard tile
17         else if Min ≤ 0 and Max ≥ 0 then
18             add curve to tile
19 for all non-discarded tiles in parallel do
20     if tile has no curves or final level is reached then
21         forward to fine raster
22     else
23         forward with curves to next level rasterization
```

Algorithm 1: Parallel Tile Rasterization

While iterating over all curves that define a patch, we only add those to the tiles that can still influence it (line 18). In particular, if a curve completely marks a tile as outside, we discard the tile (line 16). After completing the step for one patch, we have created a per-tile curve list, *i.e.*, a new CPatch structure for each tile to pass down the hierarchy (line 23). A tile with an empty curve list that is not marked as outside can be passed on to the fine rasterization stage immediately (line 21).

Fine rasterizer The final rasterization stage (the fine rasterizer) is called for a final tile and only needs to evaluate the remaining curve equations for all (sub-)pixels. The fine rasterizer operates in parallel over all pixels and can make use of efficient on-chip memory on the GPU. If multi-sampling is desired, a coverage bitmask is forwarded to the final shading stage.

Circular curves Half-space classification for circular curves only requires the center and the radius; tile-circle intersection is simply derived from line-circle tests. The only difference to the infinitely extending curves discussed above is that a circle can be completely inside a tile, a situation that is trivial to detect. Taking all this into account, we employ the same parallel tile test as for curves.

4.3. GPU software rasterizer

To show the benefits of our proposed scheme, we discuss an implementation operating on the GPU in compute mode. To take advantage of the manycore architecture of the GPU, we want to perform

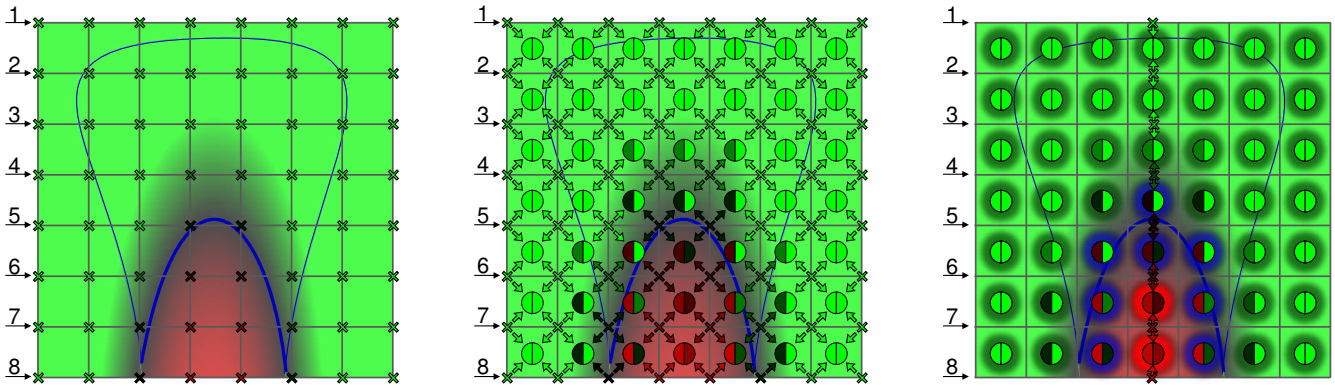


Figure 6: *Operating on an entire tile sub-grid with eight threads, we reduce the overall number of operations. Example for the strong blue curve: (1) parallel corner evaluation, (2) min/max update, shown as circles, (3) row extrema computations and update. After column extrema computation (not shown), most tiles can be classified as either inside the curve (green glow) or outside (red glow), while only eight tiles still need to test for the curve (blue glow). Note that the center tile is only classified correctly due to the extrema.*

as many operations in parallel as possible. This problem is complicated by different entry points into the tile hierarchy, the varying number of hierarchy levels to traverse, and the varying amount of parallelism per patch, mainly owed to its size. Moreover, blending needs to respect the depth order of patches.

To take best advantage of the parallelism of the problem, we rely on Whippletree [SKB*14], a task-scheduling framework based on CUDA. We use two task types: the tile rasterizer and the fine rasterizer. For the tile rasterizer, we generate multiple instances supporting a range of sub-grids, from which we choose the one most fitting the CPatch. We use grids sizes of 1×7, 7×1 and 7×7, each using eight threads, which allows Whippletree to fill up a warp (32 threads executing on a SIMD core) with four tasks.

The input data to the tile rasterizer includes the level, and the id of the tile to be rasterized. As the execution of tile rasterizer on different levels is identical, Whippletree can combine tasks for different levels for efficient computation. For example, four tiles of size 1×7 taken from different levels can be combined for one warp. The fine rasterizer uses a grid of 8×4 threads, each responsible for a single pixel. For sub-pixel coverage, we use a bitmask for multi-sampling, while super-sampling treats all sub pixels individually. Using one thread for all sub-pixel samples achieves better performance than using one thread per sub-pixel sample.

Tile store Finally, we need to resolve blend order. For order independent transparency in conventional polygon rendering, a common approach is constructing per-fragment linked lists [YHGT10]. We could employ a similar approach for CPatch blending, storing samples that lie inside patches in dynamically constructed linked lists. This approach would require many lists and all samples would need to be generated and stored, before consuming any of the data.

Thus, instead of storing lists for each fragment, our tile-based rasterizer can be extended to create lists for the final tiles. This strategy allows to delay the execution of the fine rasterizer to a second pass operating on sorted tile lists. Each list entry needs to store the CPatch data, *i.e.*, the remaining curve references and the primitive id. This design implies a trade-off: While the number of lists is reduced significantly in comparison to per-pixel lists, the amount of data stored per entry is larger. Nonetheless, the resulting memory requirement is usually lower. Moreover, sorting becomes less expensive, as its cost is proportional to the number of lists.

Upon closer inspection, this approach closely resembles triangle rasterization on NVIDIA hardware: The hardware pipeline assigns primitives to tiles for final rasterization [KKSS17]; processing is carried out with a parallel sorting step before final rasterization [Pur10]. However, our approach can still not be classified as a full streaming solution, since it temporarily stores all output data and performs a complete sort. A full streaming approach could reduce sorting cost further, but would require a more complex implementation.

Note that we evaluate shading only in the final pass, which inherits properties of deferred shading. We operate on the sorted lists from front to back and stop list traversal as soon as full opacity is reached. This not only reduces the shading and blending cost, but also the rasterization cost. In case advanced blend modes are needed that do not support front-to-back processing, the process can be reversed.

5. Converting vector graphics to CPatches

In the last section, we have described a hierarchical rasterizer for CPatches. For a complete pipeline, it is left to show that general vector graphics can be represented as CPatches. To this end, we present a simple conversion pipeline. Our current implementation takes an SVG image as input, and converts all its path elements to a CPatch representation. Strokes must be converted to filled paths in a preprocessing step. The converter supports lines, quadratic Bézier curves, cubic Bézier curves and circular arcs, with non-zero and even-odd fill rules. The six stages of our pipeline are outlined in Figure 7, including examples of each stage.

Graph flattening SVG paths can have arbitrary cycles and overlaps; intersections of curves are not explicitly captured in the SVG. To simplify later processing, we build a flattened graph for each

(a) *Graph Flatten* **(b)** *Fill Scoring* **(c)** *Cycle Extraction* **(d)** *Patch Cutting* **(e)** *Self-intersections* **(f)** *Extension Correction*

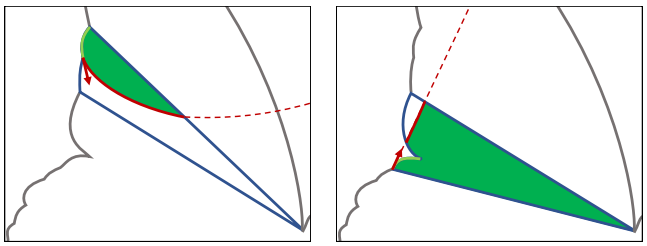
Figure 7: *Our six-stage conversion pipeline for arbitrary vector graphics to a CPatch representation: (a) A flat graph is constructed by computing all curve intersections. (b) By shooting two rays for each edge, the fill score is determined. (c) After removing edges with identical fill score on both sides, complete cycles are extracted. (d) Cycles with many curves are cut down to smaller patches. (e) Self-intersecting curves are handled by splitting patches along the self-intersections. (f) Additional straight lines are added to cut away wrongly filled outside areas.*

path: We iteratively add curves to the existing path representation, until the complete path is captured by the graph. When adding a new curve, we determine matching nodes in the graph (end points of curves) and perform curve-curve intersection testing with all existing curves via Bézier clipping [SN90]. For each intersection, we add a new node to the graph and break open the curves at the intersection. This process results in a flattened graph for each path, where nodes capture all intersections of curves, and edges represent segments of the original curves connecting to the nodes.

Fill scoring For each flattened graph, we determine where the path should be filled. While fill scores are typically defined for each sample in the drawing, we only determine the fill score for each graph edge to either side of the curve. To this end, we shoot a ray normal to the edge at the half-way point of the curve. For each ray, we determine the fill score by computing the intersections with all curves and applying the fill score rule accordingly (non-zero or even-odd). The result of the fill score test is stored with each edge. If both sides of the edge yield the same fill score, we simply remove the edge, as it is not relevant for the drawing. Note that this computation is very light-weight, as we only determine the fill score twice per edge and not per sample in the drawing.

Cycle extraction As CPatches should represent primitives, we extract cycles in the graph at an early point. In this way, we can later ignore interaction between loosely connected sub-patches. To perform the extraction, we start with a random edge and walk along the graph. At each node, we choose the curve with the smallest outgoing angle to the incoming edge, considering which side of the edge should be filled. For this angle computation, we compute the derivative of the involved curves at the node. When we encounter the starting edge again, we have extracted a full cycle.

In this way, we separate each path into multiple independent cycles. The outlined approach works well even if cycles are touching. However, nested cycles need additional treatment, as both the outer and the inner cycle are required to construct a CPatch representation. From each inner cycle we shoot a ray to find the first outer cycle and split ring-like paths into two separate touching cycles, as shown in Figure 7c. Note that there could be multiple inner cycles. In this case, we first connect the inner cycles and then make the connection to the outermost cycle to avoid 'cutting' inner cycles.



(a) **(b)**

Figure 8: *Self-intersections of patches arise, when a curve reaches back into the patch and wrongly classifies parts as outside the patch, which can happen when a curve (light green) extension points inward the patch (a), or comes back (b). Cutting the patch in two resolves the problem.*

Patch cutting While cycles, per definition, already form a patch, they might consist of a large number of curves. As we limit the number of curves for efficient rendering, we cut cycles that exceed the limit, using an algorithm inspired by ear clipping [Mei75]:

```
1  while Patch has more than MaxCurves curves do
2      for N ← MaxCurves − 1 to 1 do
3          for all edges in patch do
4              mark edge and next N-1 edges
5              connect end-points of marked edges with line
6              if line does not intersect any edge then
7                  split off marked edges and make new patch
8                  add line to original patch
9                  break
10     if Patch still has more than MaxCurves curves then
11         split longest edge in the middle and add new node
```

Although this heuristic is rather simple, it worked for all drawings we tested. In some cases, a large number of additional nodes are inserted when straight lines cannot be placed in the interior. Curved cuts could be an option to avoid these additional nodes.

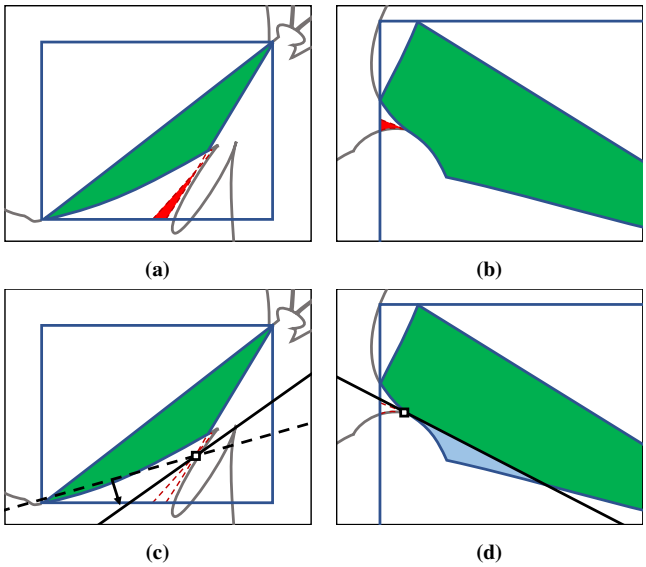


Figure 9: *(a) Extension correction is necessary when curve extensions cross outside the patch and thus wrongly classify regions as inside. (b) Such errors are also common for cusps, where they happen directly next to the patch. (c) We fit an additional line to cut these regions. (d) Sometimes a straight curve cannot successfully perform the cut, in which case we split the patch in two.*

Self-intersection cutting Self-intersecting curves are one of the major challenges for CPatch generation, since they lead to incorrect half-space classifications, as shown in Figure 8. We distinguish two cases, a curve which extends into a patch from its starting node (8a) and a curves that returns into the patch after leaving it (8b).

We handle self-intersections by cutting the patch in two. We iterate over all curves of the patch and test the derivative at the end nodes to determine whether the curve points inward. Then, we compute all intersections of the curve extension with the patch. We simply extent the curve to a large multiple of the original length using the De Casteljau algorithm [DC86] and again perform Bézier clipping to check for intersection. According to our experiments, the alternative of solving for intersections using the implicit curve is more time consuming and less accurate.

After finding all intersections, we sort them and split the patch in two (Figure 8). After creating the two new patches, we continue the process for both newly generated patches. For efficiency reasons, we retain the information about which curves of the new patch have already been tested. Curves that have loops need special treatment, if a complete loop is formed by the extension. In this case, an additional patch only consisting of the loop might be needed.

Extension correction One final issue concerns curves that cross outside of the patch, but within the bounding box. This might create wrongly filled areas, as shown in Figure 9. Again, this issue might arise directly at the end of the curve, *e.g.*, with cusps (Figure 9a), or from an intersection of two curve extensions (Figure 9b). These unwanted regions can be handled by locating them and pruning the offending crossings by inserting an additional curve to the patch.

	Input		CPatch		Tile Raster		
Data	Pth.	C.	Ptc.	C./Ptc.	Ptc.	Ptc./Tile	C./Ptc.
drops	204	1k	1k	3.58	18k	2.24	1.09
embrace	225	5k	4k	3.20	43k	5.35	1.37
tiger	240	2k	3k	3.38	22k	2.74	1.81
car	420	12k	7k	3.33	38k	4.75	1.95
sample_v2	691	7k	7k	3.29	34k	4.21	2.28
hawaii	1137	53k	41k	2.83	102k	12.48	2.21
boston	1922	28k	14k	3.23	46k	5.71	1.85
paris-70k	45k	545k	303k	3.36	531k	64.91	2.88
contour	53k	188k	57k	3.42	115k	14.06	2.81

Table 1: *Statistics of the test data sets and processing results. The input SVG datasets range from 200 to 53k paths (Pth) with up to 545k curves (C). Our preprocessing generates up to 303k patches (Ptc) with an average of about 3.3 curves per patch. After tile rasterization (1k resolution), lists capture up to half a million patches.*

To locate such offending crossings, we consider all intersections of curve extension (which are guaranteed to be outside of the patch after the execution of the previous stage) as well as all intersections of curve extensions with the bounding box. For each of those points, we evaluate all implicit curve equations and keep only those that yield a wrong result. From the offending crossings, we construct connected cycles (there might be multiple).

Then, we find the point that is closest to the original patch—for cusps, this could even be a node of the patch. We use this point as anchor and place a line to cut the wrong region, which we add as a curve to the patch. There are infinitely many line directions to consider. We optimize by starting with a random direction and rotate it depending on where we hit the falsely positive region (or the patch), as shown in Figure 9c. We iterate with a reduced rotation angle, until we find a fitting direction or end up with no movement. In case no solution is found, we cut the patch in two (Figure 9d).

Remarks Even though the preprocessing sounds complex, our non-optimized, single-threaded CPU code runs efficiently. For example, it loads and processes the Tiger image (Figure 1, right), in less than a second. Given our simple implementation, there is a large optimization potential. Furthermore, a CPatch representation only needs to be constructed once; it could easily be stored as additional information alongside the vector drawing. Especially when using our technique in a graphics editor, such as Adobe Illustrator, only one path is edited at a time, and thus only a single CPatch representation needs to be computed, which can easily be done at interactive rates.

6. Results

To evaluate the performance of our approach, we tested a variety of common vector graphics benchmark drawings, as outlined in Table 1 and Figure 1 and 10. All tests were run on an NVIDIA GeForce GTX 1080Ti (3584 CUDA cores, 11GB of global memory) hosted by an Intel Core i7-6850K CPU 3.60GHz with 64GB of system memory. As comparison methods, we use NVIDIA path rendering [KB12] (NV) and Li et al.'s GPU scanline rasterizer [LHZ16] (SL). We use their original published implementation.

(a) *embrace* **(b)** *tiger* **(c)** *car* **(d)** *sample_v2*

(e) *hawaii* **(f)** *boston* **(g)** *paris-70k* **(h)** *contour*

Figure 10: *The test data set spans simple (≈ 200 paths a,b), medium sized (1000–2000 paths e,f), and large graphics (> 40 000 paths g,h).*

Our approach uses a tile size of 8×4, a maximum number of four curves per patch, and every element in the tile list can hold 32 elements. For sorting, we use per-block radix sort and choose the best fitting block size among 32, 64 and 128, depending on the average guessed number of patches per tile. While these choices put slightly more pressure on the preprocessing and increase memory requirements, they favor speed.

Preprocessing As can be seen in Table 1, our preprocessing usually cuts each input path into 5–40 CPatches on average, creating up to 300 000 CPatches for the largest input. Drawings with smaller and more complex curved structures, *e.g.*, *hawaii* or *boston*, are cut into more patches than rather simple drawings, like *drops*. As *contour* mostly consists of triangular and rectangular data, it is already very close to a usable CPatch representation and thus hardly needs any processing.

After tile rasterization, the overall number of patch references throughout all lists ranges from 18 000 to 530 000 (1k resolution). List lengths are relatively short on average for most simple drawings with 2–14 entries. *paris-70k* forms an exception with its large number of small patches. The number of referenced curves after tile rasterization is strongly reduced to 1.9–2.9 on average, indicating the success of the hierarchical approach.

Timing Performance numbers are shown in Table 2. When multisampling is disabled, our approach shows the best performance in eight out of nine cases for 1k resolution and four out of nine cases for 2k resolution. NV takes the lead in two and four cases, respectively. SL is always the slowest approach. For 16× multisampling, the situation slightly shifts, with our approach winning in four and three cases, NV in four and two cases, and SL in one and three cases, respectively. Overall, we achieve a mean (harmonic) speed up of 1.48× and 1.80× without multisampling and 1.43× and 1.17× for 16× multisampling over NV and SL, respectively. Our approach shows the most balanced performance, keeping up with NV for smaller drawings (*drops*, *tiger*, *car*) and showing very competitive performance for large drawings with complex structures (*paris-70k*, *contour*), which are typically vastly in favor of alternative approaches.

	res	1× Multisampling			16× Multisampling		
		Our	NV	SL	Our	NV	SL
drops	1k	**0.51**	0.61	1.62	0.72	**0.71**	1.75
	2k	1.31	**0.61**	1.72	2.12	**1.44**	2.10
embrace	1k	0.75	**0.63**	1.95	0.93	**0.84**	2.08
	2k	0.92	**0.62**	2.01	1.81	**1.75**	2.39
tiger	1k	**0.66**	**0.66**	1.63	0.87	**0.81**	1.73
	2k	0.79	**0.66**	1.69	**1.72**	1.75	2.04
car	1k	**0.82**	1.17	2.16	1.38	**1.12**	2.35
	2k	**1.07**	1.17	2.21	**1.91**	2.22	2.54
sample_v2	1k	**0.47**	2.57	1.37	**0.83**	2.57	1.52
	2k	**0.88**	2.53	1.39	**1.66**	2.56	1.72
hawaii	1k	**0.88**	2.07	2.49	**1.90**	2.10	2.97
	2k	2.31	**2.06**	5.53	**4.45**	5.44	9.96
boston	1k	**0.64**	3.42	1.30	**1.04**	3.41	1.44
	2k	**1.26**	3.43	1.33	2.14	3.41	**1.73**
paris-70k	1k	**1.96**	74.6	2.43	3.13	74.1	**2.58**
	2k	3.52	72.5	**2.52**	4.81	73.5	**2.99**
contour	1k	**0.63**	90.9	1.48	**1.53**	90.9	1.57
	2k	**0.85**	90.1	1.55	3.24	90.9	**1.89**

Table 2: *Runtime performance of our approach in milliseconds compared to NV path rendering and GPU scanline rasterization.*

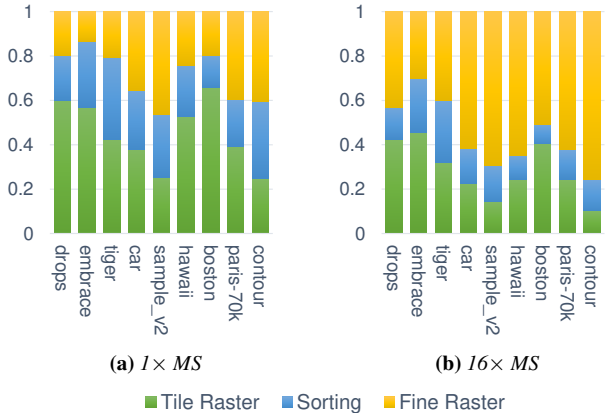


Figure 11: *Relative run time of the three major steps of our approach. Multisampling influences fine raster time only.*

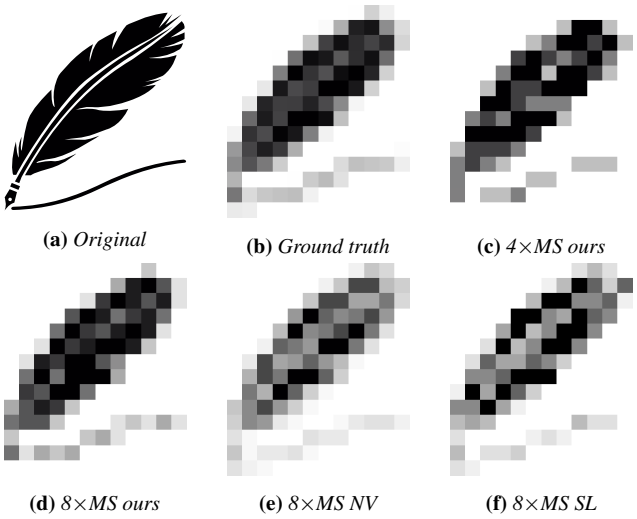


Figure 12: *Quality example for 12×14 pixels large renderings of the feather image compared to 256× super-sampled ground truth of the feather image. SL and NV show higher errors due to their treatment of subpixels. Even our 4×MS image achieves high accuracy in comparison to the 8×MS renderings of NV and SL.*

It should be noted that NV, in many cases, is not limited by the compute power of the GPU, but rather suffers from synchronization delays due to the 'stencil, then cover' approach, which reduces the amount of parallel workload. Thus, increasing the resolution or multisampling has hardly any influence for NV. While SL also follows a 'stencil, then cover' approach, they first generate strides that are then rendered in parallel by OpenGL. SL uses an approximation for stride boundaries and thus multisampling is less costly in their approach. Therefore, SL results may slightly differ visually. Our approach scales with the workload, reducing performance when the resolution is increased or multisampling is activated.

The relative performance of the three steps of our approach is shown in Figure 11. If multisampling is disabled, tile rasterization is usually the most time consuming step. Fine rasterization is slightly more costly than sorting. When multisampling is enabled, fine raster takes over the majority of the workload for most tests, which is not surprising, as the number of tested samples is increased 16×.

Quality Figure 12 shows quality examples for 8× multisampling of the tested approaches in comparison to a 256× supersampled ground truth (16 × 16 downsampled image). Our approach clearly achieves the best result for this challenging case (even 4× multisampling is superior in image quality). We can only speculate about the errors of the other approaches which both rely on hardware multisampling. Both NV and SL render geometry for the fine structures, which is subject to subpixel snapping for fixed point rasterization, which may influence the evaluated equations and generated stencils. Additionally, SL represents both scanline ends with simplified geometry, which leads to additional errors. As our approach does not perform any boundary simplifications and fully evaluates curve equations for all sub-pixels, we achieve a higher quality.

Discussion While CPatches are usable for all vector graphics, some drawings, like `contour`, are already close to a CPatch representation and thus more efficient. Paths with many curves, like the butterfly in Figure 1, will typically get cut into many patches, which explains the high expansion factor of some drawings. However, while the number of CPatches increases, memory requirements only increase marginally when using references to the original curves.

Our approach is most efficient when rendering patches that fill out their bounding box well, *e.g.*, rectangular CPatches result in most efficient rendering. However, also thin and slanted patches can be rasterized efficiently, as empty regions are discarded early in the hierarchical rasterizer, whereas traditional 'stencil, then cover' methods would test all curves for all pixels in the bounding polygon. Thus, our approach is also well suited for boundary rasterization which naturally consist of many thin segments.

Due to the nature of our approach, all types of fill types and blending can easily be integrated. As the fine rasterizer is executed in compute mode, not only color gradient or textures are naturally supported as fill types, but any type of computations can be performed, *e.g.*, noise evaluations, complex sampling, or involved lighting are possible. Similarly, as blending is performed in software, we can use any combination of color spaces and blend functions.

7. Conclusions

We have presented a novel approach for representing and rendering vector graphics using curved primitives, CPatches, which enable parallel rendering, similar to how triangle rasterization is performed in real-time rendering. A CPatch representation allows the construction of a complete parallel hierarchical rasterizer on the GPU. Our software prototype, running in GPU compute mode, shows competitive performance when compared to the hardware supported NVPR for small vector graphics. It performs on the same level as previous state-of-the-art methods for complex drawings, while completely avoiding all approximations. Thus, our approach not only achieves speedups of 17% to 80% over the previous state-of-the-art, but also achieves higher quality for multi sampling. Our approach shows the best performance, when the input vector graphics is already close to a CPatch representation.

To show the applicability of our approach, we have also presented a preprocessing pipeline to convert arbitrary vector graphics to a CPatch representation. We see high potential for better preprocessing in the future, which would not only increase preprocessing speed, but also generate CPatches that can be rendered more efficiently.

To increase rendering speed, we see two further potential directions. First, building a complete streaming rendering pipeline could reduce memory traffic and thus increase speed, similar to recent work on software real-time rendering [KKSS18]. Second, our approach could be hardware-accelerated, expanding the hardware rasterizer to not only support straight edge equations for triangle rasterization, but also implicit curve equations.

Acknowledgments

This research was supported by the Max Planck Center for Visual Computing and Communication, the German Research Foundation (DFG) grant STE 2565/1-1, and the Austrian Science Fund (FWF) grant I 3007.

References

[AW81] ACKLAND B. D., WESTE N. H.: The Edge Flag Algorithm: A Fill Method for Raster Scan Displays. *IEEE Trans. Comput. 30*, 1 (Jan. 1981), 41–48. 2

[BKKL15] BATRA V., KILGARD M. J., KUMAR H., LORACH T.: Accelerating Vector Graphics Rendering Using the Graphics Hardware Pipeline. *ACM Trans. Graph. 34*, 4 (July 2015), 146:1–146:15. 2

[DC86] DE CASTELJAU P. D. F.: *Shape mathematics and CAD*, vol. 2. Kogan Page, 1986. 8

[Far88] FARIN G.: *Curves and surfaces for computer-aided geometric design: a practical guide*. Academic Press, 1988. 3

[FSF97] FABRIS A. E., SILVA L., FORREST A. R.: An efficient filling algorithm for non-simple closed curves using the point containment paradigm. In *Proceedings X Brazilian Symposium on Computer Graphics and Image Processing* (Oct 1997), pp. 2–9. 2

[FSH11] FINCH M., SNYDER J., HOPPE H.: Freeform Vector Graphics with Controlled Thin-plate Splines. *ACM Trans. Graph. 30*, 6 (Dec. 2011), 166:1–166:10. 3

[GLdFN14] GANACIM F., LIMA R. S., DE FIGUEIREDO L. H., NEHAB D.: Massively-parallel Vector Graphics. *ACM Trans. Graph. 33*, 6 (Nov. 2014), 229:1–229:14. 2, 3

[Goo18] GOOGLE: *Skia Graphics Library*, 2018. https://skia.org/. 2

[KB12] KILGARD M. J., BOLZ J.: GPU-accelerated Path Rendering. *ACM Trans. Graph. 31*, 6 (Nov. 2012), 172:1–172:10. 2, 8

[KKSS17] KERBL B., KENZEL M., SCHMALSTIEG D., STEINBERGER M.: Effective Static Bin Patterns for Sort-middle Rendering. In *Proceedings of High Performance Graphics* (New York, NY, USA, 2017), HPG '17, ACM, pp. 14:1–14:10. 6

[KKSS18] KENZEL M., KERBL B., SCHMALSTIEG D., STEINBERGER M.: A High-performance Software Graphics Pipeline Architecture for the GPU. *ACM Trans. Graph. 37*, 4 (July 2018), 140:1–140:15. 11

[KSST06] KOKOJIMA Y., SUGITA K., SAITO T., TAKEMOTO T.: Resolution Independent Rendering of Deformable Vector Objects Using Graphics Hardware. In *ACM SIGGRAPH 2006 Sketches* (New York, NY, USA, 2006), SIGGRAPH '06, ACM. 2

[LB05] LOOP C., BLINN J.: Resolution Independent Curve Rendering Using Programmable Graphics Hardware. *ACM Trans. Graph. 24*, 3 (July 2005), 1000–1009. 2, 3

[LHZ16] LI R., HOU Q., ZHOU K.: Efficient GPU Path Rendering Using Scanline Rasterization. *ACM Trans. Graph. 35*, 6 (Nov. 2016), 228:1–228:12. 2, 8

[Mei75] MEISTERS G. H.: Polygons have ears. *The American Mathematical Monthly 82*, 6 (1975), 648–651. 7

[MS13] MANSON J., SCHAEFER S.: Analytic Rasterization of Curves with Polynomial Filters. *Computer Graphics Forum 32*, 2pt4 (2013), 499–507. 2

[NH08] NEHAB D., HOPPE H.: Random-access Rendering of General Vector Graphics. *ACM Trans. Graph. 27*, 5 (Dec. 2008), 135:1–135:10. 3

[NS79] NEWMAN W. M., SPROULL R. F. (Eds.): *Principles of Interactive Computer Graphics (2Nd Ed.)*. McGraw-Hill, Inc., New York, NY, USA, 1979. 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. 3

[OG97] OLANO M., GREER T.: Triangle Scan Conversion Using 2D Homogeneous Coordinates. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware* (New York, NY, USA, 1997), HWWS '97, ACM, pp. 89–95. 4

[Pur10] PURCELL T.: Fast tessellated rendering on Fermi GF100. In *High Performance Graphics Conf., Hot 3D presentation* (2010). 6

[PWE18] PACKARD K., WORTH C., ESFAHBOD B.: *Cairo: A Vector Graphics Library*, 2018. https://www.cairographics.org/. 2

[PZ08] PARILOV E., ZORIN D.: Real-time Rendering of Textures with Feature Curves. *ACM Trans. Graph. 27*, 1 (Mar. 2008), 3:1–3:15. 2

[QMK08] QIN Z., MCCOOL M. D., KAPLAN C.: Precise Vector Textures for Real-time 3D Rendering. In *Proceedings of the 2008 Symposium on Interactive 3D Graphics and Games* (New York, NY, USA, 2008), I3D '08, ACM, pp. 199–206. 2

[Sal79] SALMON G.: *A treatise on the higher plane curves: intended as a sequel to A treatise on conic sections*. Hodges, Foster, and Figgis, 1879. 3

[SKB*14] STEINBERGER M., KENZEL M., BOECHAT P., KERBL B., DOKTER M., SCHMALSTIEG D.: Whippletree: Task-based Scheduling of Dynamic Workloads on the GPU. *ACM Trans. Graph. 33*, 6 (Nov. 2014), 228:1–228:11. 6

[SN90] SEDERBERG T. W., NISHITA T.: Curve intersection using Bézier clipping. *Computer-Aided Design 22*, 9 (1990), 538–549. 7

[STZ14] SUN T., THAMJAROENPORN P., ZHENG C.: Fast Multipole Representation of Diffusion Curves and Points. *ACM Trans. Graph. 33*, 4 (July 2014), 53:1–53:12. 3

[Whi15] WHITINGTON J. G.: Two Dimensional Hidden Surface Removal with Frame-to-frame Coherence. In *Proceedings of the 31st Spring Conference on Computer Graphics* (New York, NY, USA, 2015), SCCG '15, ACM, pp. 141–149. 2

[WREE67] WYLIE C., ROMNEY G., EVANS D., ERDAHL A.: Half-tone Perspective Drawings by Computer. In *Proceedings of the November 14-16, 1967, Fall Joint Computer Conference* (New York, NY, USA, 1967), AFIPS '67 (Fall), ACM, pp. 49–58. 2

[WW82] WARNOCK J., WYATT D. K.: A Device Independent Graphics Imaging Model for Use with Raster Devices. *SIGGRAPH Comput. Graph. 16*, 3 (July 1982), 313–319. 1

[WZYG10] WANG L., ZHOU K., YU Y., GUO B.: Vector Solid Textures. *ACM Trans. Graph. 29*, 4 (July 2010), 86:1–86:8. 2

[YHGT10] YANG J. C., HENSLEY J., GRÃIJN H., THIBIEROZ N.: Real-Time Concurrent Linked List Construction on the GPU. *Computer Graphics Forum 29*, 4 (2010), 1297–1304. 6

[YLK*15] YOO J.-J., LEE J., KRISHNADASAN S., LEE W., BROTHERS J., RYU S.: Tile-based Path Rendering for Mobile Device. In *SIGGRAPH Asia 2015 Mobile Graphics and Interactive Applications* (New York, NY, USA, 2015), SA '15, ACM, pp. 5:1–5:6. 2