Massively-Parallel Vector Graphics

Francisco Ganacim Rodolfo S. Lima
Luiz Henrique de Figueiredo Diego Nehab
IMPA

ACM Transactions on Graphics
(Proceedings of ACM SIGGRAPH Asia 2014)

Vector graphics are everywhere

clip-paths to the shortcut tree like any other path geometry, and maintain in each shortcut tree cell a stream that matches the scene grammar described in section 3. Clipping operations are performed per sample and with object precision.

When evaluating the color of each sample, the decision of whether or not to blend the paint of a filled path is based on a Boolean expression that involves the results of the inside-outside tests for the path and all currently active clip-paths. Since this expression can be arbitrarily nested, its evaluation seems to require one independent stack per sample (or recursion). This is undesirable in code that runs on GPUs. Fortunately, as discussed in section 4.3, certain conditions (see the pruning rules) allow us to skip the evaluation of large parts of the scene. These conditions are closely related to the short-circuit evaluation of Boolean expressions. Once we include these optimizations, it becomes apparent that the value at the top of the stack is never referenced. The successive simplifications that come from this key observation lead to the *flat clipping* algorithm, which does not require a stack (or recursion).

Flat clipping The intuition is that, during a union operation, the first inside-outside test that succeeds allows the algorithm to skip all remaining tests at that nesting level. The same happens during an intersection when the first failed inside-outside test is found. Values on the stack can therefore be replaced by knowledge of whether or not we are currently skipping the tests, and where to stop skipping. The required context can be maintained with a finite-state machine.

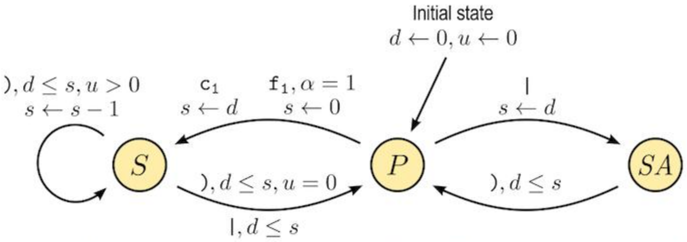
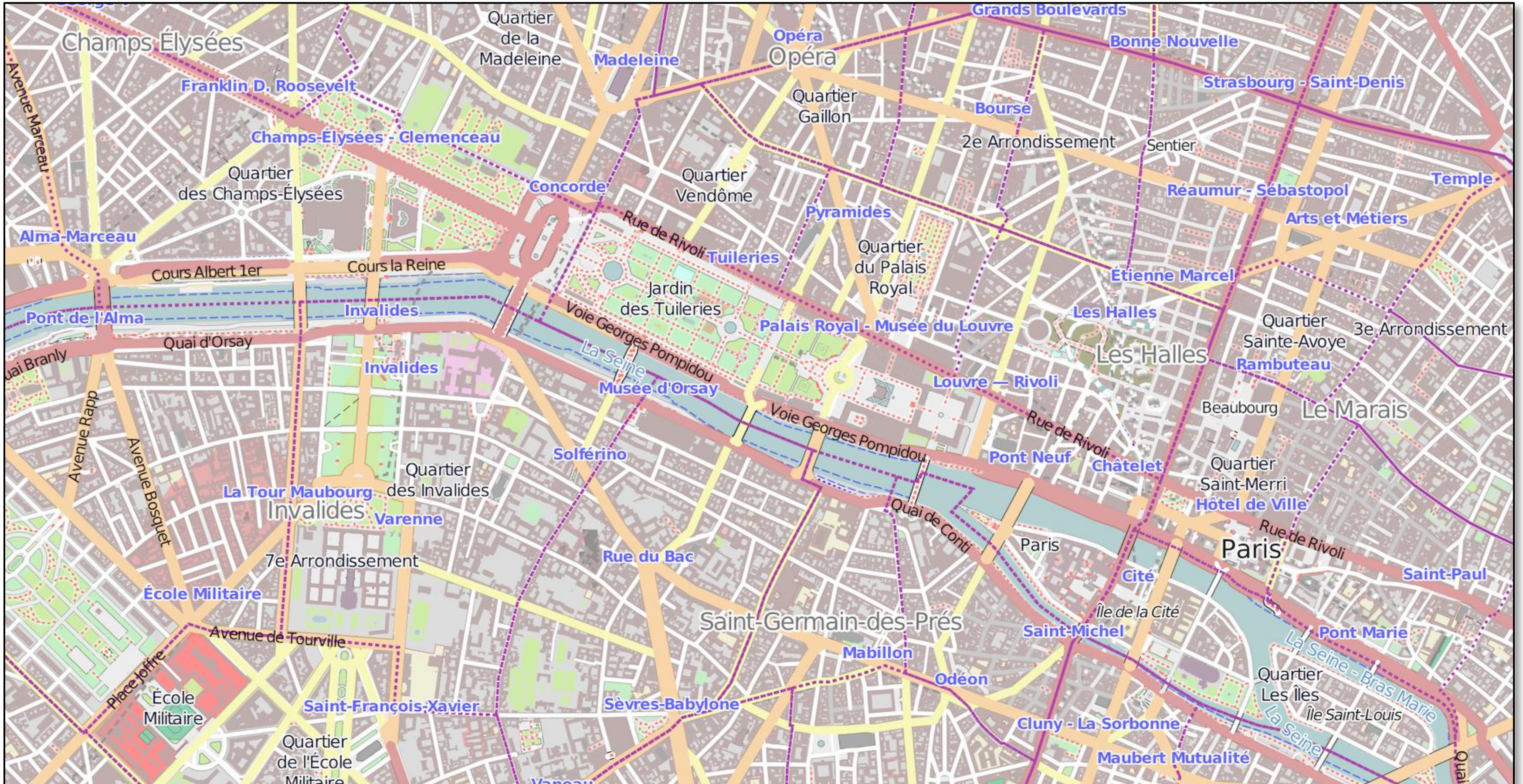


Figure 12: *State transition diagram for the finite-state machine of the flat-clipping algorithm.*

two transitions away from S. The first transition happens when an *activate* operation is found. Looking at the scene grammar, we see that this can only happen if the machine arrived at S due to a $\mathtt{c}_1$ transition from P. In other words, an entire clip-path test has succeeded, and therefore we transition unconditionally back to P. The second transition happens when a matching) is found. The condition $u = 0$ means the machine is not inside a nested clip-path test, so it simply transitions back to P. If the machine is skipping *inside* a nested clip-path test, one of the inner clip tests must have passed, and therefore the outer test can be short-circuited as well. The machine simply resets the stop depth to the outer level and continues in state S.

The remaining transitions are between P and SA. If the machine finds a | while in state P, it must have been performing a clip-path

Vector graphics are everywhere



Vector graphics are everywhere

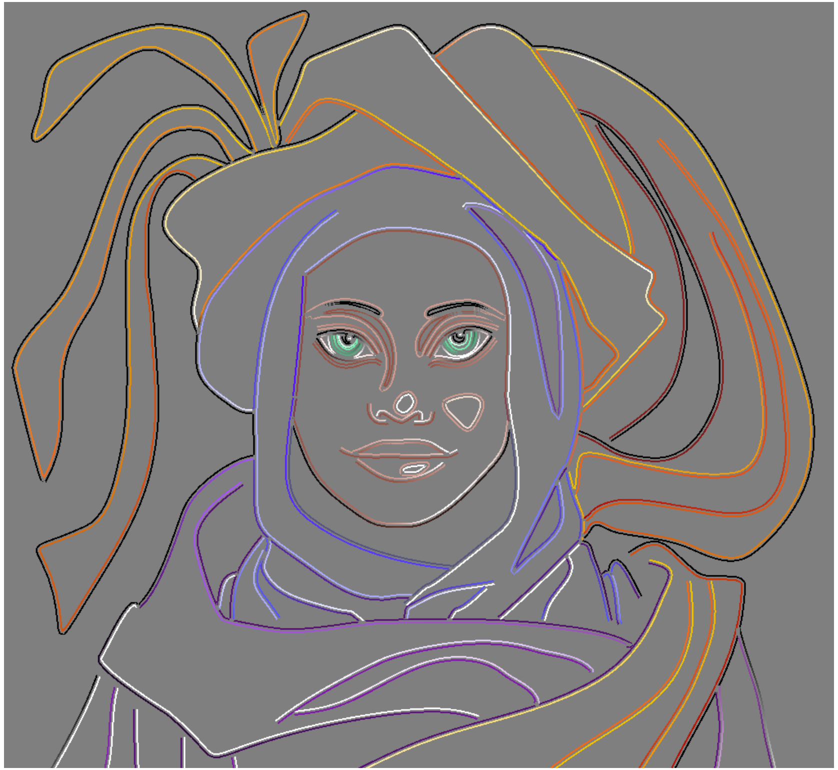
Points to be made

- 2D graphics incredibly prevalent
- 2D graphics is not a “solved problem”
- It deserves more attention

- Can benefit from parallelism
  - Increased computational power

- Needs new algorithms

Diffusion-based vector graphics



[Orzan et al. 2008] [Finch et al. 2011] [Sun et al. 2012 and 2014]

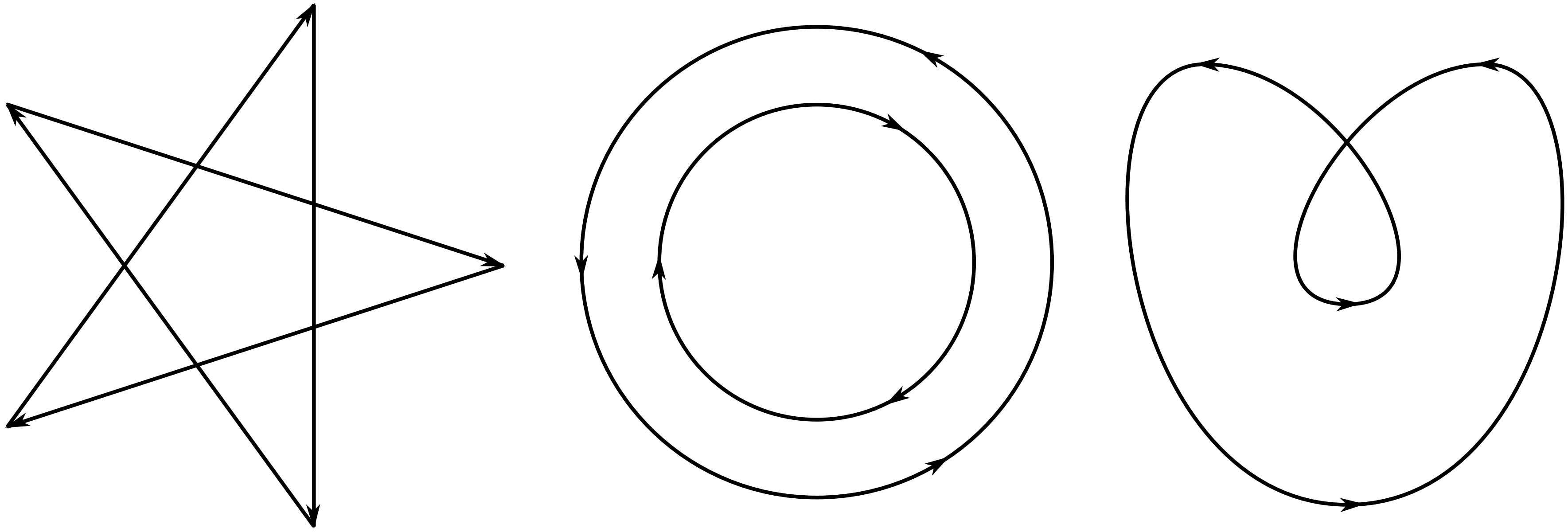
Related work

PATH-BASED VECTOR GRAPHICS

Basic concepts are *paths* and *paints*

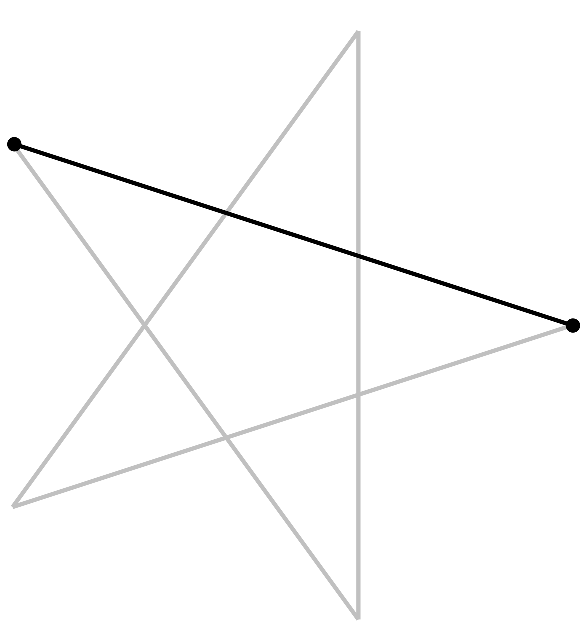
[Warnock & Wyatt 1982]

Paths

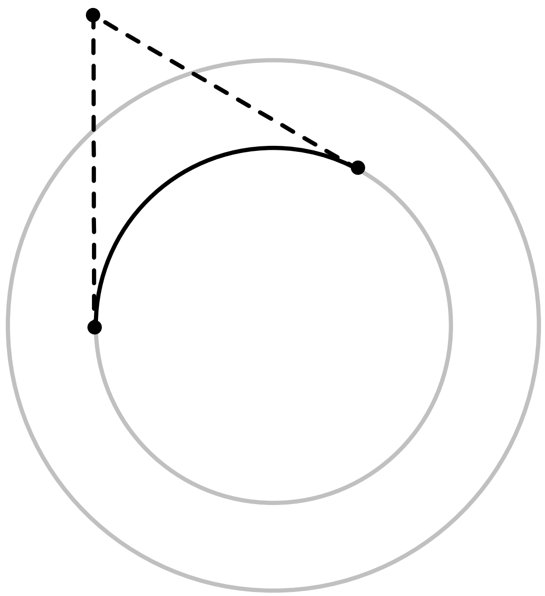


Closed contours

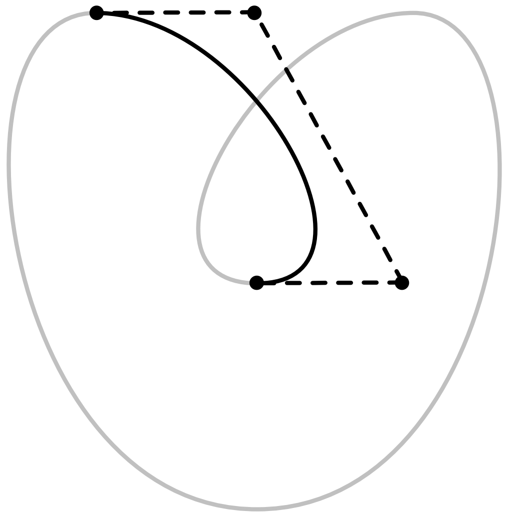
Segments



Linear



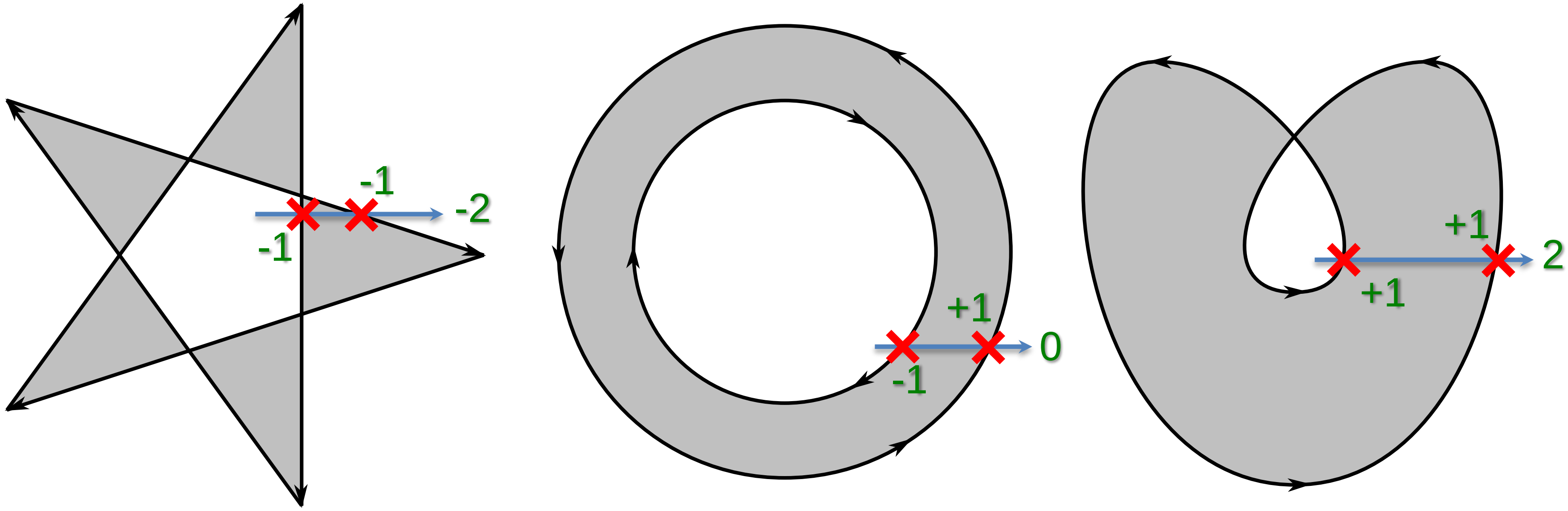
Quadratic



Cubic

Inside-outside test

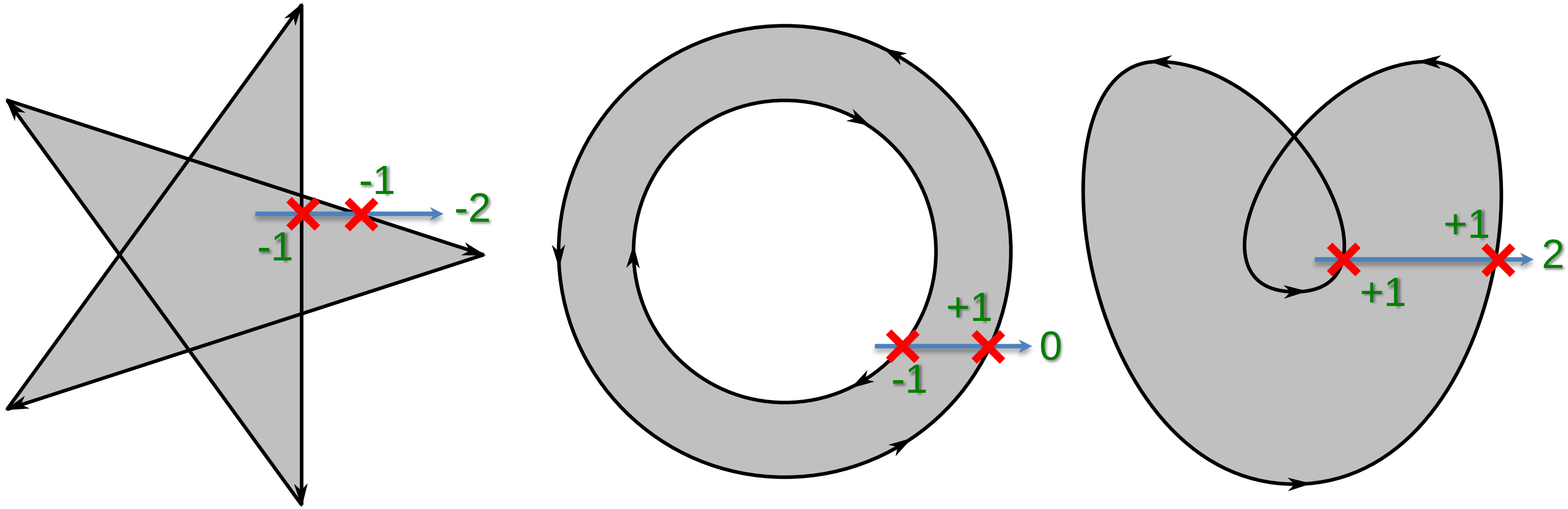
Winding numbers



Even-odd rule

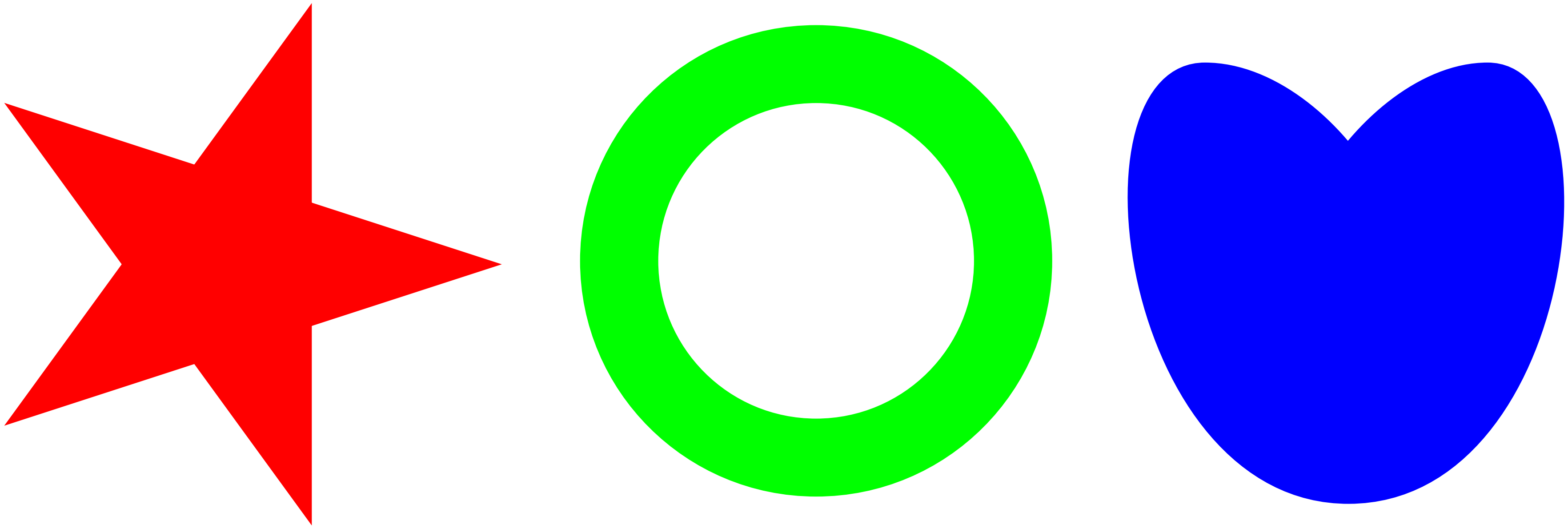
Inside-outside test

Winding numbers



Non-zero rule

Paints



Solid

Paints

Radial gradient

Linear gradient

Texture

Availability

- Formats & languages
  - PostScript, CDR, PDF, SVG, OpenXPS, AI
  - TTF fonts, Type 1 fonts
- Editors
  - Adobe Illustrator, CorelDraw, Inkscape, FontForge, …
- Rendering tools & APIs
  - NV_Path_Rendering, OpenVG, Cairo, Qt, MuPDF, GhostScript, Apple’s, Adobe’s, Microsoft’s, …

Rasterization or rendering

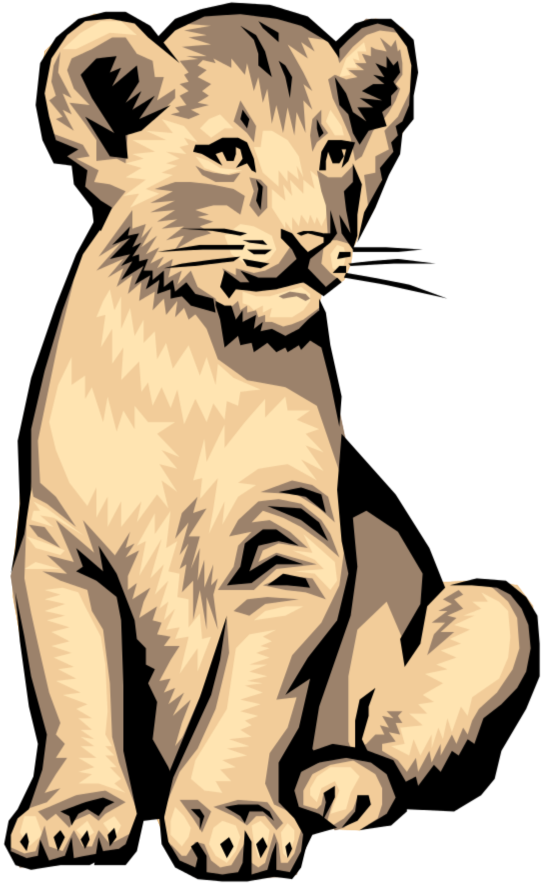
Generate image at chosen resolution for display or printing

Traditional rendering algorithm

- Render one shape after the other

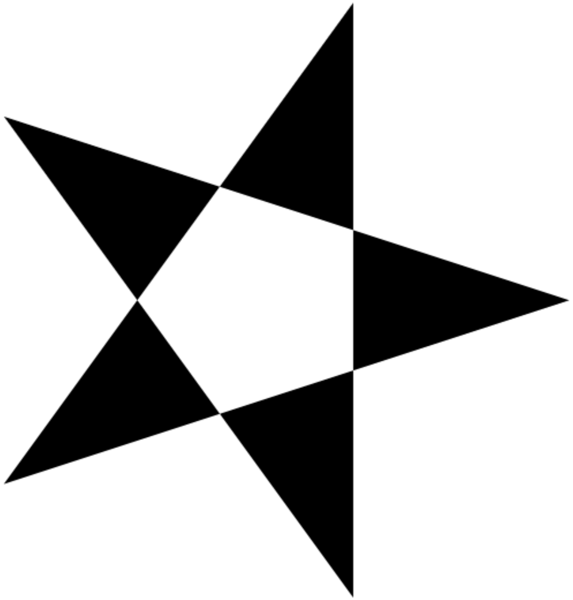
```
for all shapes
  prepare for acceleration
  for all samples in shape
    blend paint over output
```

- Most tools follow this approach



Active-edge-list polygon filling

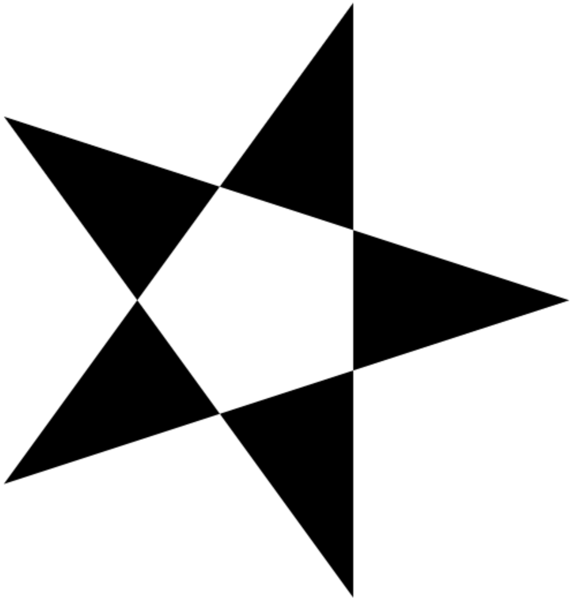
- Uses spatial coherence in horizontal spans



[Wylie et al. 1967]

Stencil-based polygon filling

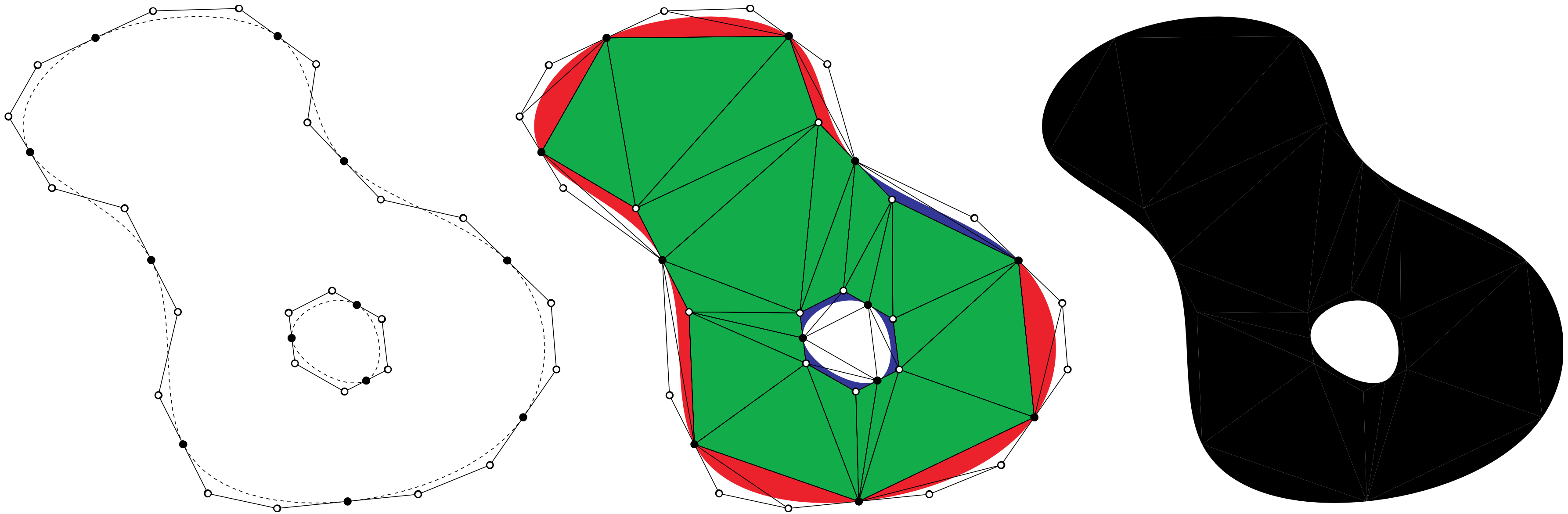
- Rasterize winding numbers into stencil



[Neider et al. 1993]

Curve rendering by graphics hardware

- Constrained triangulation + affine implicitization



[Loop & Blinn 2005]

Implicitization

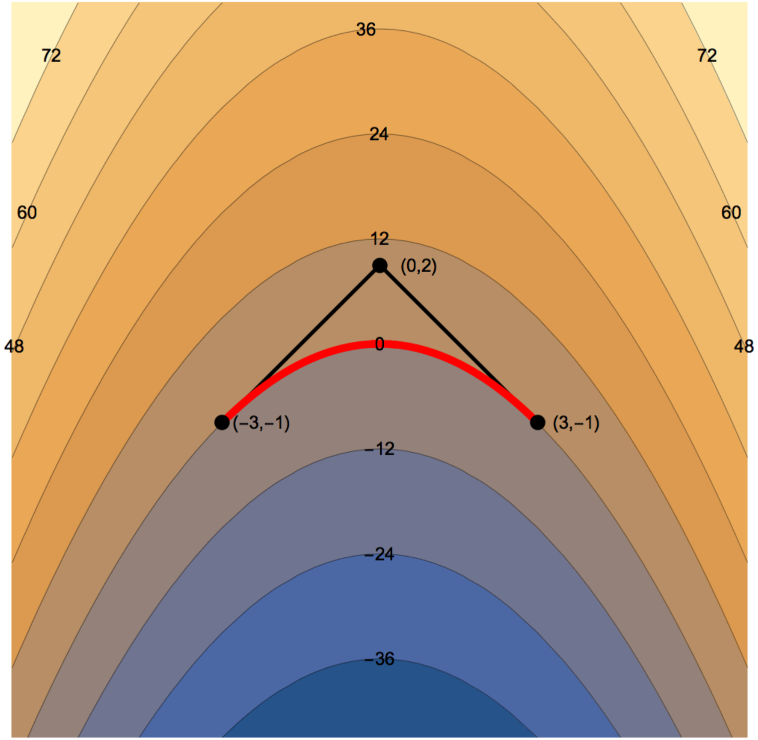
Theorem: *A polynomial parametric curve*

$$c(t) = \big(x(t), y(t)\big)$$

has a polynomial implicit form $C(x, y)$ *with*

$$C(x_p, y_p) = 0 \;\Leftrightarrow\; \exists t_p \mid c(t_p) = (x_p, y_p)$$

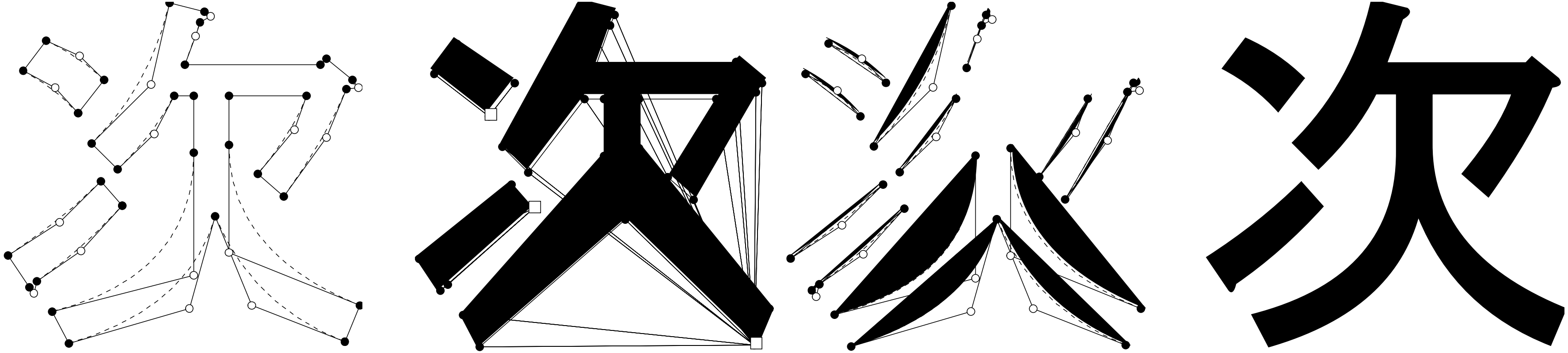
- Different methods
  - Sederberg [1984]
    - Based on Cayley-Bézout or Sylvester
  - Loop & Blinn [2005]
    - Based on Salmon (affine implicitization)



$$c(t) = \begin{bmatrix} -3 \\ -1 \end{bmatrix} (1-t)^2 + \begin{bmatrix} 0 \\ 2 \end{bmatrix} 2t(1-t) + \begin{bmatrix} 3 \\ -1 \end{bmatrix} t^2 \;\Leftrightarrow\; C(x, y) = x^2 + 6y - 3$$

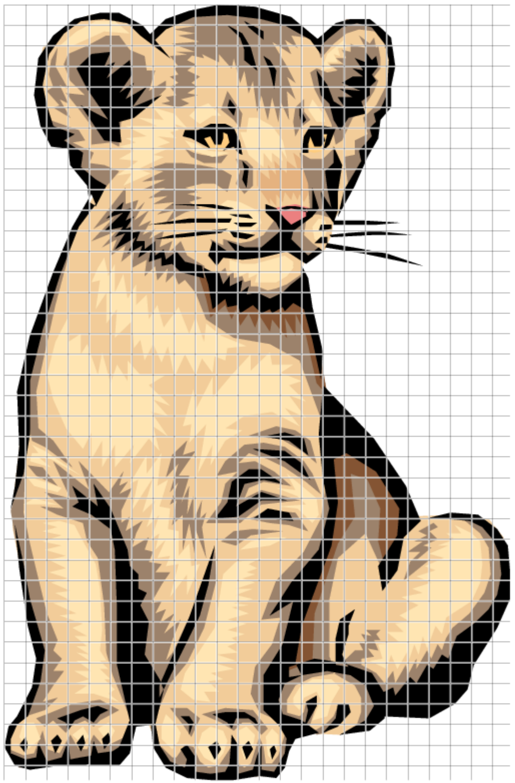
NV_Path_Rendering

- Stencil-based filling with affine implicitization
  - Complete, state-of-the-art pipeline

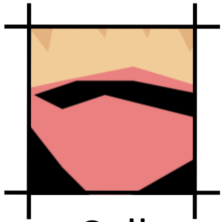


[Neider et al. 1993] + [Loop & Blinn 2005] ≈ [Kokojima et al. 2006] ≈ [Kilgard & Bolz 2012]

Alternative approach



Cell grid



Cell

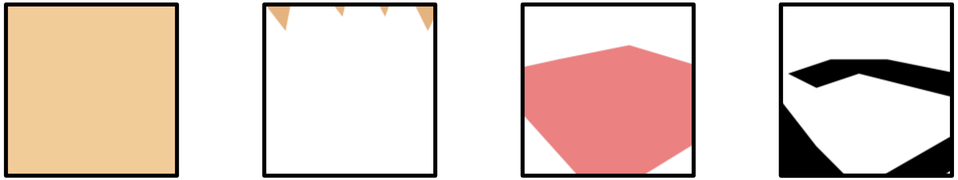


Illustration clipped against cell

Magnification with image textures

- Can become blurry at high magnification levels

[Nehab & Hoppe 2008]

Magnification with vector textures

- Maintains sharpness indefinitely

[Nehab & Hoppe 2008]

General warps in object space

[Nehab & Hoppe 2008]

Vector texture rendering algorithm

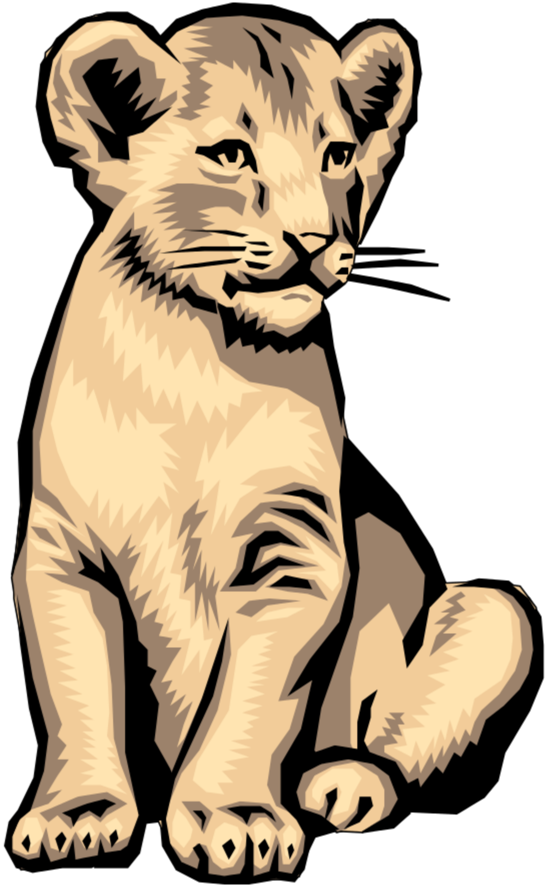
- For texture mapping and effects

```
for all shapes
  insert into acceleration structure

for all samples
  for subset of shapes containing sample
    blend paint into output
```

- Mostly limited to academia

[Sen 2004] [Ramanarayanan et al. 2004] [Qin et al. 2008]
[Parilov & Zorin 2008] [Nehab & Hoppe 2008]



Comparison of rendering algorithms

Vector textures

- Extensive pre-processing
- Retained mode
- Samples are independent
- General warps

- Analogous to Ray-tracing

Traditional

- Modest preprocessing
- Immediate mode
- Sample cost is amortized
- Limited warps

- Analogous to Z-buffering

State of the art in accelerated rendering

Vector textures

```
for all shapes
  insert in acceleration structure
```

```
for all output samples
  for subset of shapes covering sample
    blend paint into output
```

[Nehab & Hoppe 2008]

Traditional

```
for all shapes
  prepare for acceleration
```

```
for all shapes
  for all shape samples in parallel
    blend paint into output
```

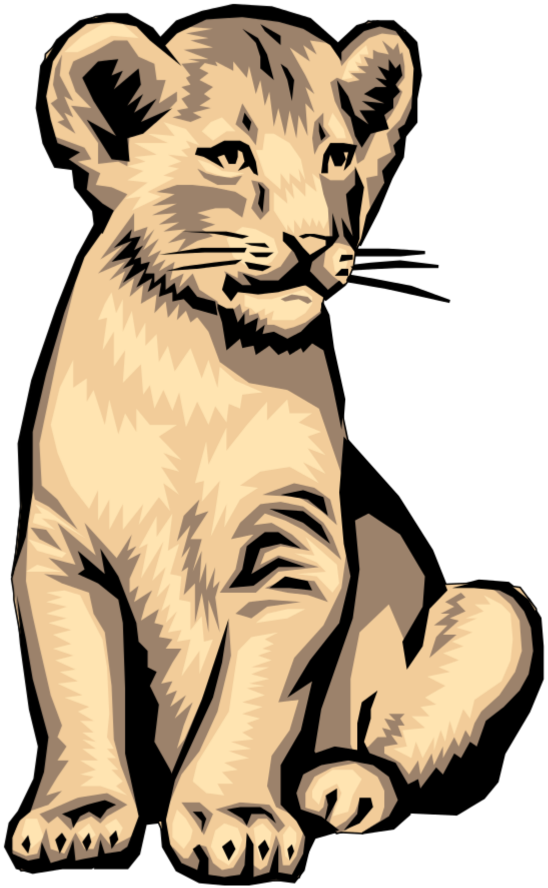
[Kilgard & Bolz 2012] (NV_Path_Rendering)

Massively-Parallel Vector Graphics

Goal

```
for all segments of all shapes
  insert in acceleration structure
```

```
for all output samples
  for subset of shapes covering sample
    blend paint into output
```



Ours [Ganacim et al. 2014]

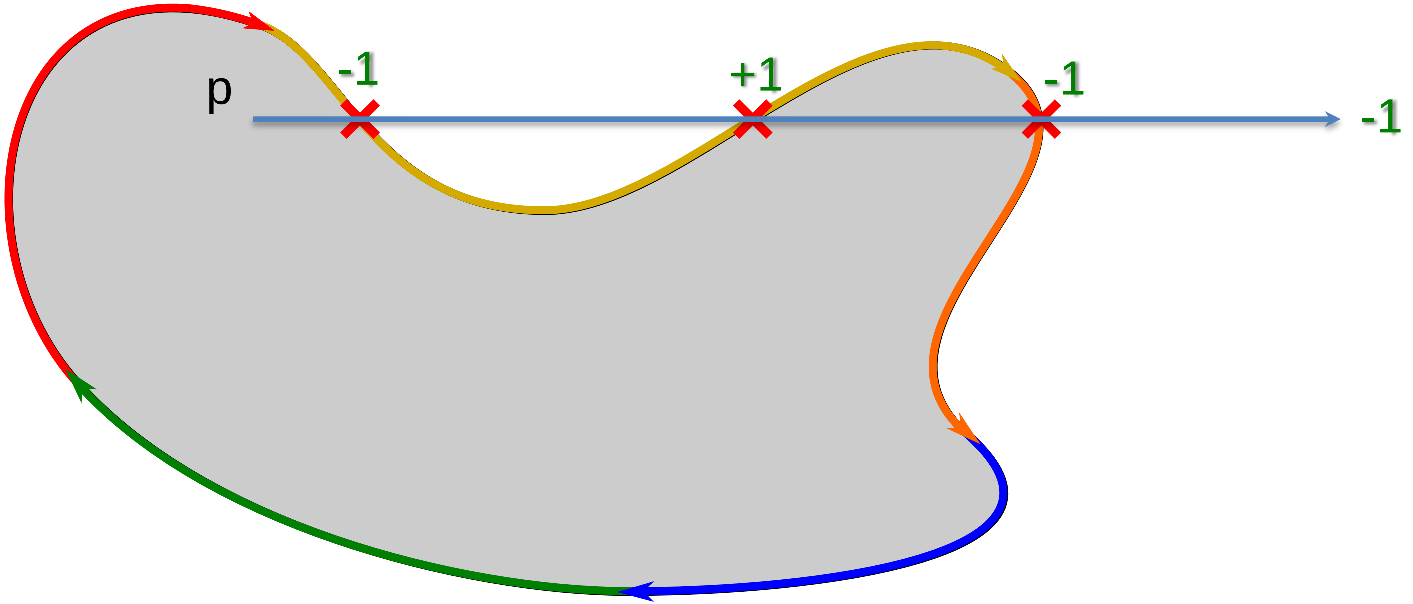
Contributions

- New primitive: *Abstract segment*
  - Based on implicitization, no intersection computations
- New acceleration data structure: *The Shortcut Tree*
  - Optimal, adaptive, segment-parallel construction
- State-of-the-art rendering quality
  - No compromises

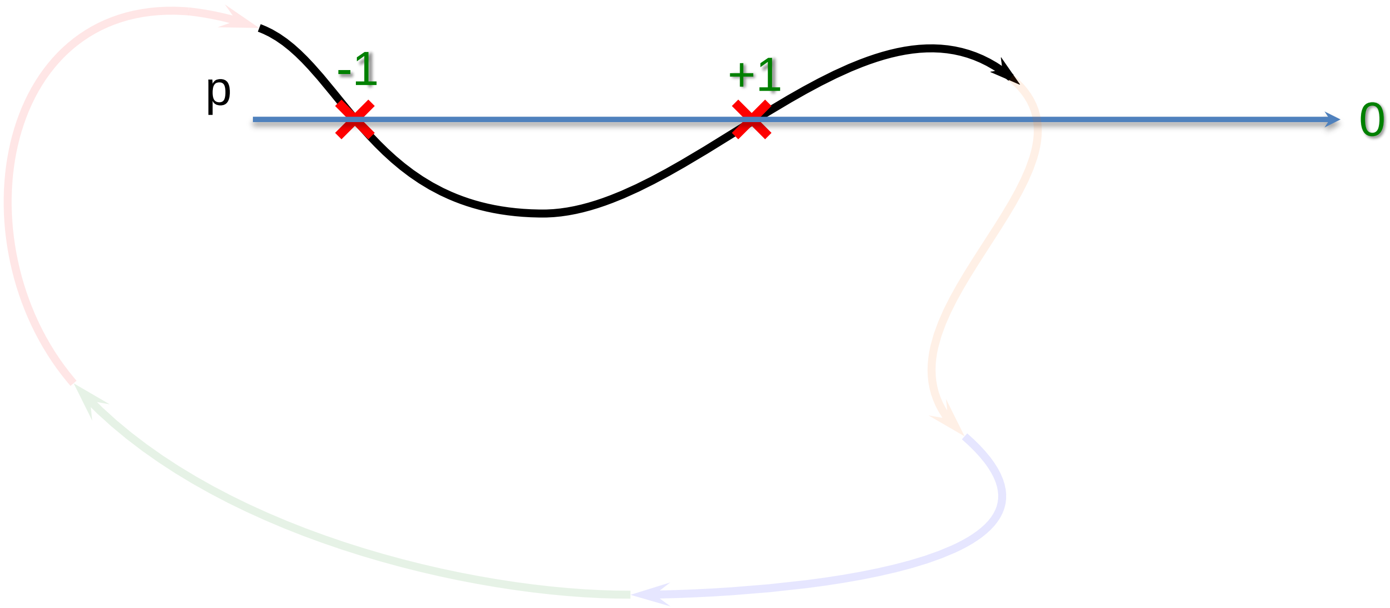
Finding the right primitive

ABSTRACT SEGMENTS

Does shape cover sample?



Does ray intersect with segment?



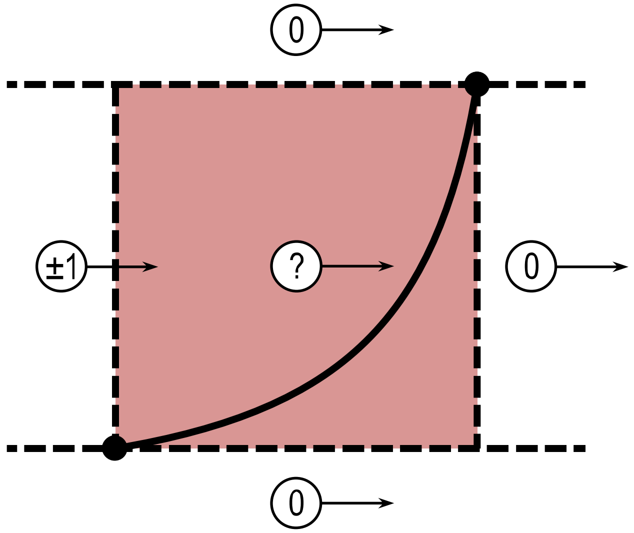
Computing intersections

- Segment is $c(t) = \big(x(t), y(t)\big), \quad t \in [0,1]$
- Sample at (x_s, y_s)

- Intersection test
  - Solve $y(t) = y_s$ for t
  - For each $t_i \in [0,1]$ such that $x(t_i) > x_s$
  - Test sign of $y'(t_i)$ to inc/dec winding number

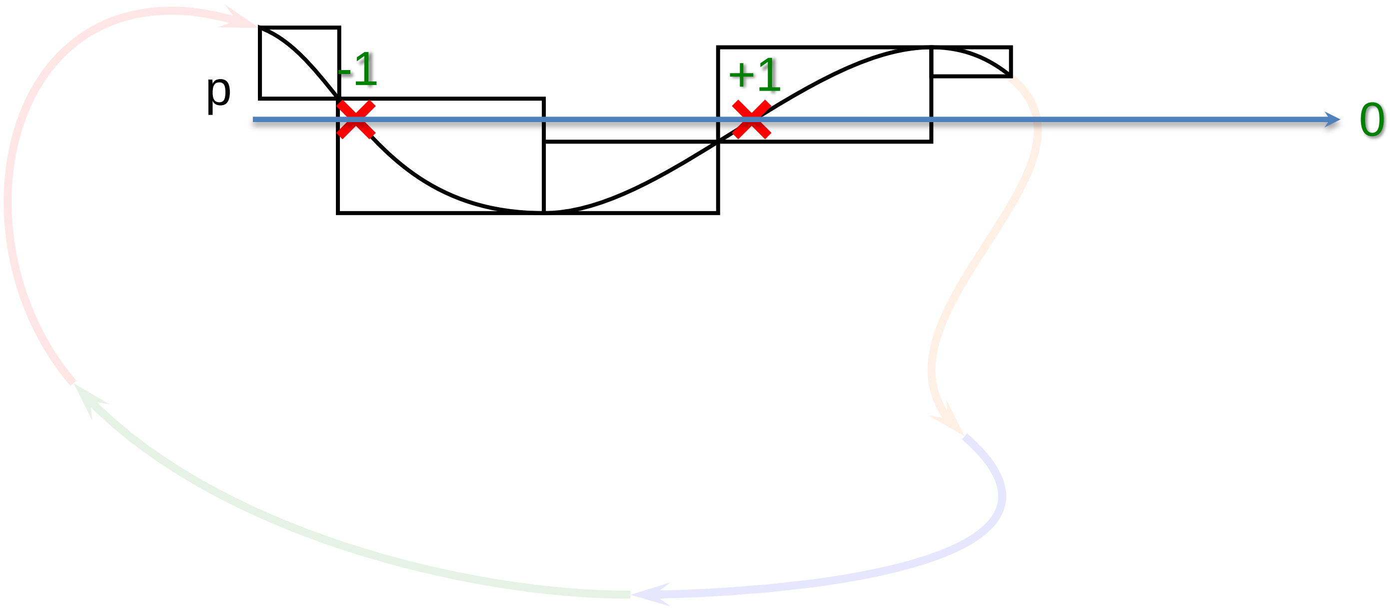
- Requires solving quadratics and cubic equations
  - Complicated, slow, not robust

Monotonic segments



Monotonization makes bounding-boxes very useful

Example of monotonized segment



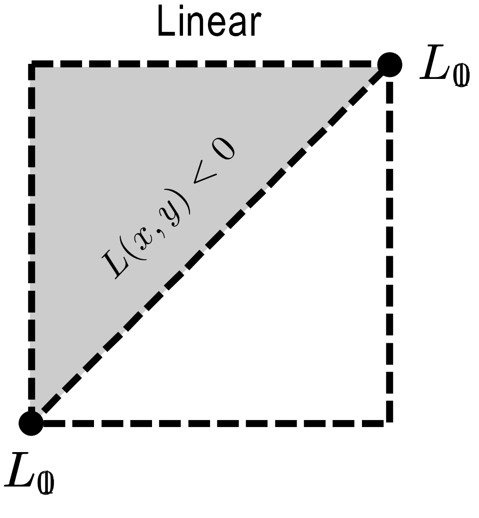
Computing intersections

- Split into monotonic segments during preprocess
  - Parts with $c_m(t_m) = \big(x_m(t_m), y_m(t_m)\big), \quad t_m \in [0,1]$
  - $x'(t_m)$ and $y'(t_m)$ have no roots for $t_m \in [0,1]$
  - Requires solving linear or quadratic equations

- Simpler intersection test during rendering
  - *One* intersection at $t_{mi} \in [0,1]$ if and only if
$$\min\big(y_m(0), y_m(1)\big) < y_s \leq \max\big(y_m(0), y_m(1)\big)$$
  - Find t_{mi} robustly (e.g., safe Newton–Raphson)
  - Check that $x(t_{mi}) > x_s$
  - Test sign of $y_m(1) - y_m(0)$ to inc/dec winding number

Implicit linear test

- Outside bounding box, trivial
- Inside bounding box, use implicitization



$$L(x, y) = a_{10}(x - x_0) + a_{01}(y - y_0)$$

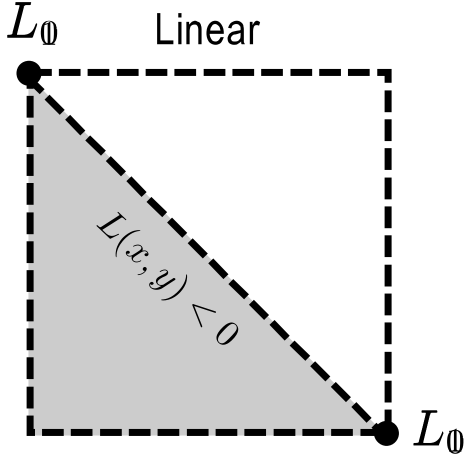
$$s = \text{sign}(y_1 - y_0)$$

$$a_{10} = s(y_1 - y_0)$$

$$a_{01} = s(x_0 - x_1)$$

Implicit linear test

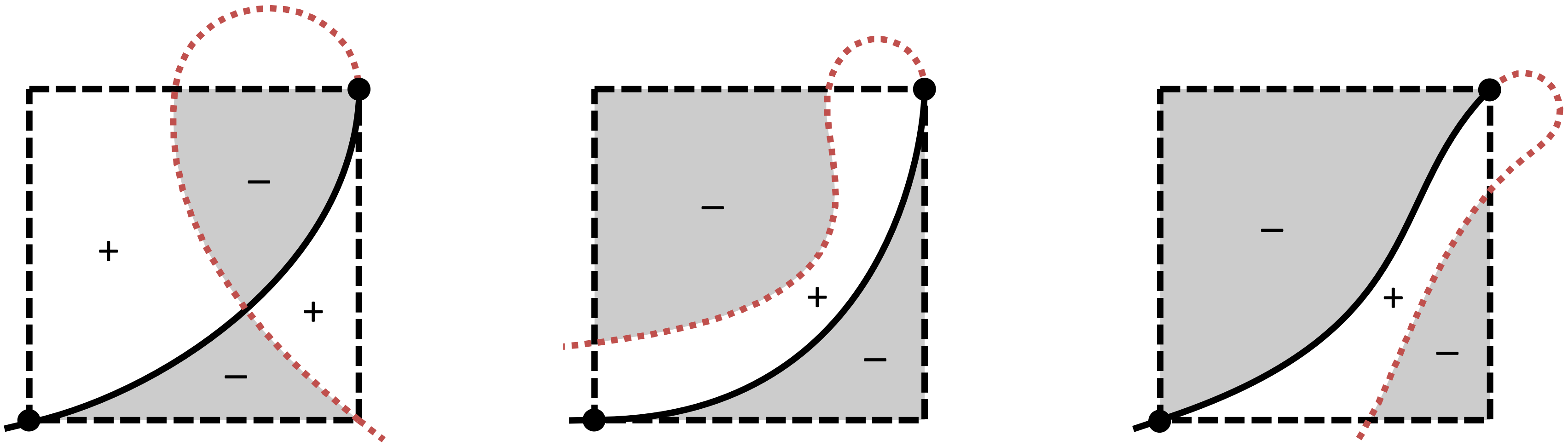
- Outside bounding box, trivial
- Inside bounding box, use implicitization



$$L(x,y) = a_{10}(x - x_0) + a_{01}(y - y_0)$$
$$s = \text{sign}(y_1 - y_0)$$
$$a_{10} = s(y_1 - y_0)$$
$$a_{01} = s(x_0 - x_1)$$

What about curves?

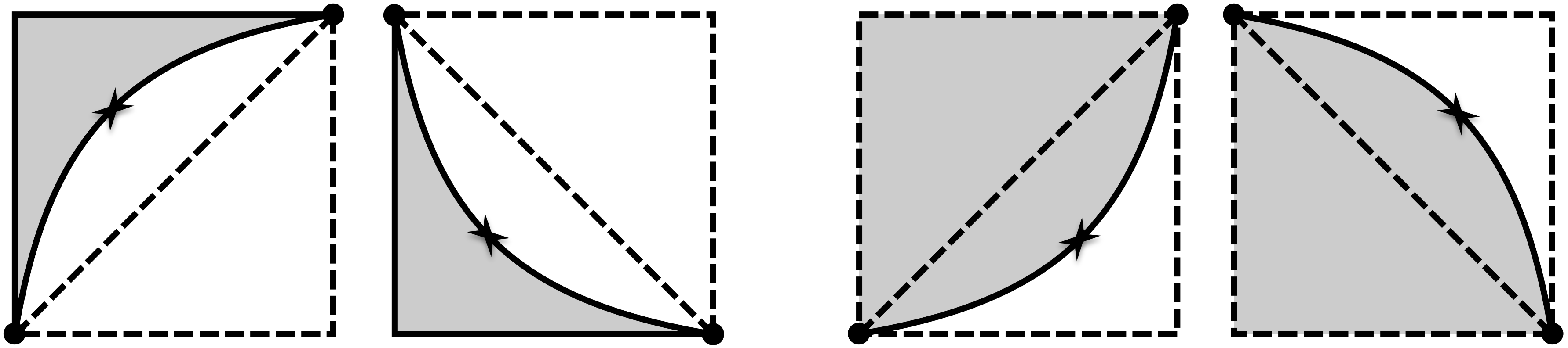
- Must be careful
  - Parametrization is *local* to [0,1]
  - Implicitization is *global*



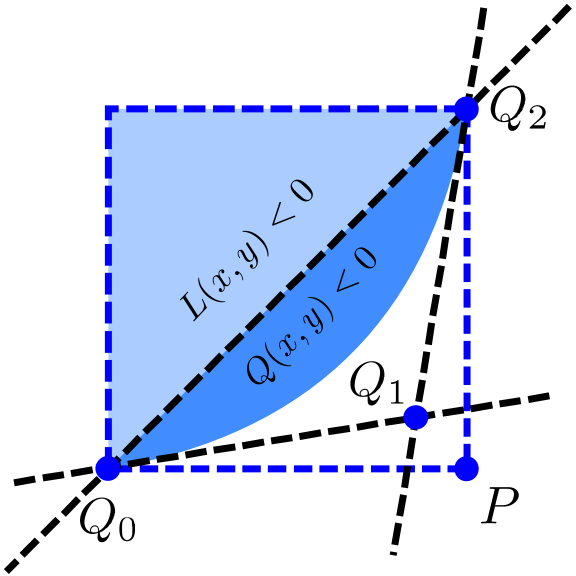
Monotonic segment with no inflections

Theorem: *Monotonic segments with no inflections cannot cross line connecting endpoints for* $t \in (0, 1)$

- After split, 8 configurations
  - Goes up/down
  - Connects diagonal/anti-diagonal
  - Entirely to left/right of diagonal



Monotonic quadratics



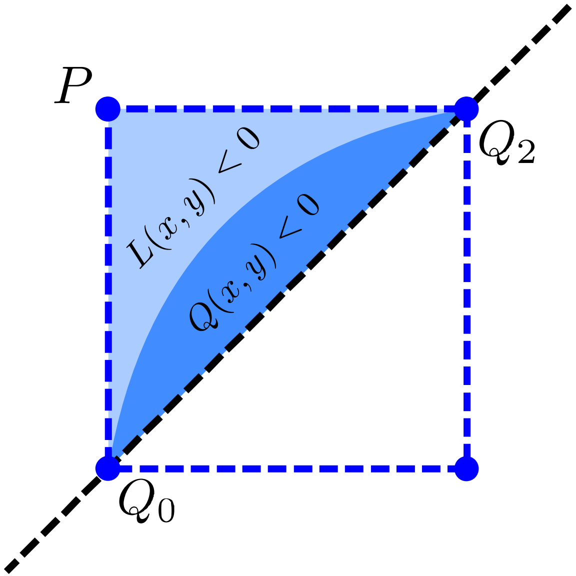
Theorem: *Quadratic* $q(t)$ *cannot reenter triangle* $Q_0Q_1Q_2$ *for* $t \notin [0,1]$

Theorem: *Quadratic* $q(t)$ *cannot reenter triangle* Q_0PQ_2 *for* $t \notin [0,1]$

$L(x,y) < 0$ or $Q(x,y) < 0$

$$Q(x,y) = (a_{10} + a_{20}x)x + (a_{01} + a_{11}x + a_{02}y)y$$

Monotonic quadratics



Theorem: *Quadratic* $q(t)$ *cannot reenter triangle* $Q_0Q_1Q_2$ *for* $t \notin [0,1]$

Theorem: *Quadratic* $q(t)$ *cannot reenter triangle* Q_0PQ_2 *for* $t \notin [0,1]$

$L(x,y) < 0$ and $Q(x,y) < 0$

$$Q(x,y) = (a_{10} + a_{20}x)x + (a_{01} + a_{11}x + a_{02}y)y$$

Abstract segments

- Similar setup for cubics and rational quadratics
- Primitive of choice for vector graphics pipeline
- Encapsulates monotonic segment `s`
  - Bounding-box, up-down, precomputed implicitization
  - Method `s.winding(x,y)`
    - Returns +1 or -1 if ray from (x,y) to (∞,y) hits, 0 otherwise

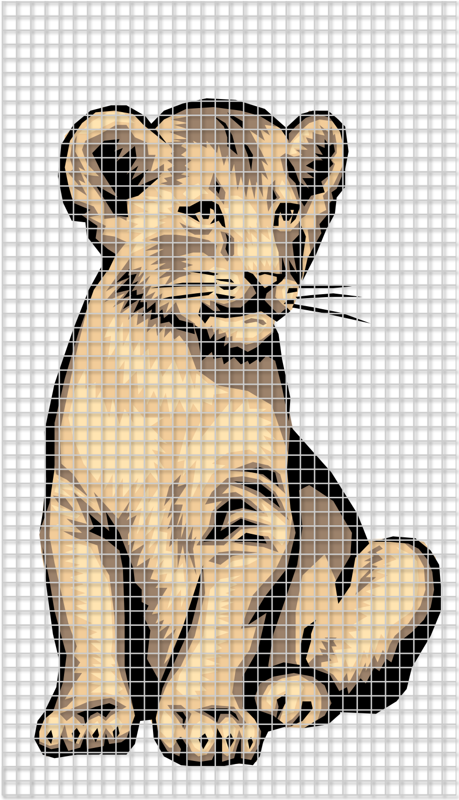
Sampling algorithm

```
for all samples (x,y)
  for all shapes
    winding number = 0
    for all segments s
      winding number += s.widing(x,y)
    if winding number implies inside
      blend paint into output
```

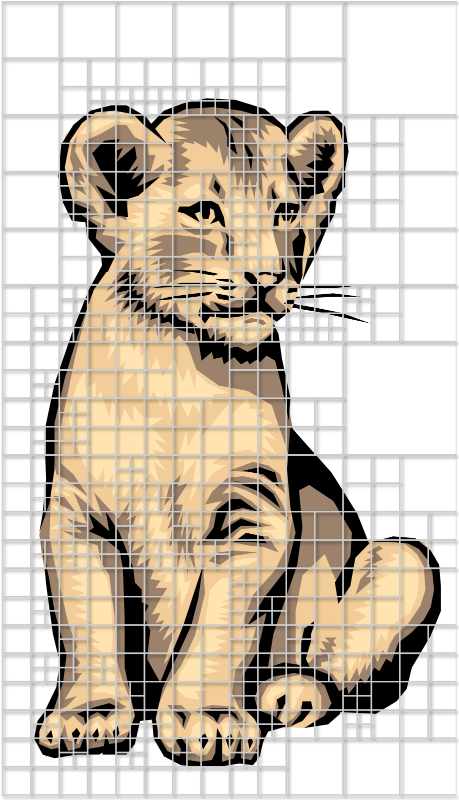
The right acceleration data structure

THE SHORTCUT TREE

Acceleration data structure



[Nehab & Hoppe 2008]: Regular grid



Ours [Ganacim et al. 2014]: Quadtree

Sampling algorithm

```
for all samples
  find cell containing sample
  for subset of shapes in cell
    winding number = 0
    for subset of segments s in cell
      winding number += s.winding(x,y)
    if winding number implies inside
      blend paint into output
```

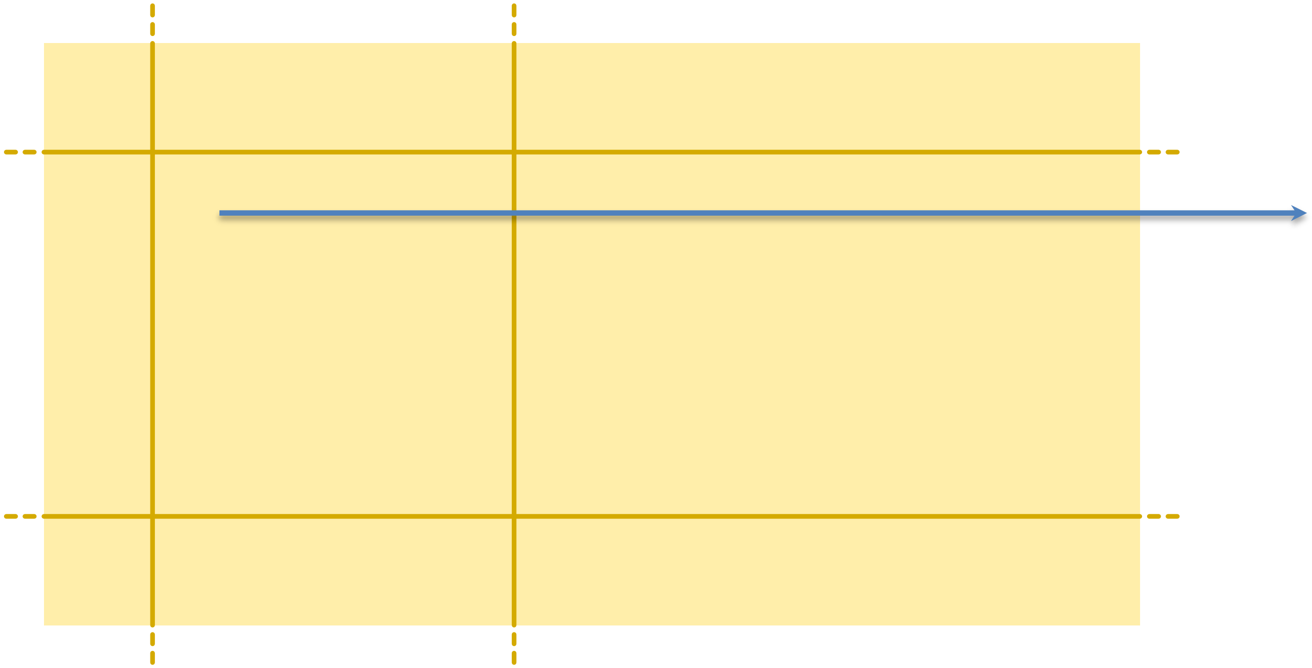
What goes on each cell?

- Specialized subset of illustration
- Everything that is needed to render cell region

Invariant: *The winding number of all paths about all samples in the cell region, computed from the cell contents, is exactly the same as in the complete illustration*

[Warnock 1969]

Clippnig is overkill



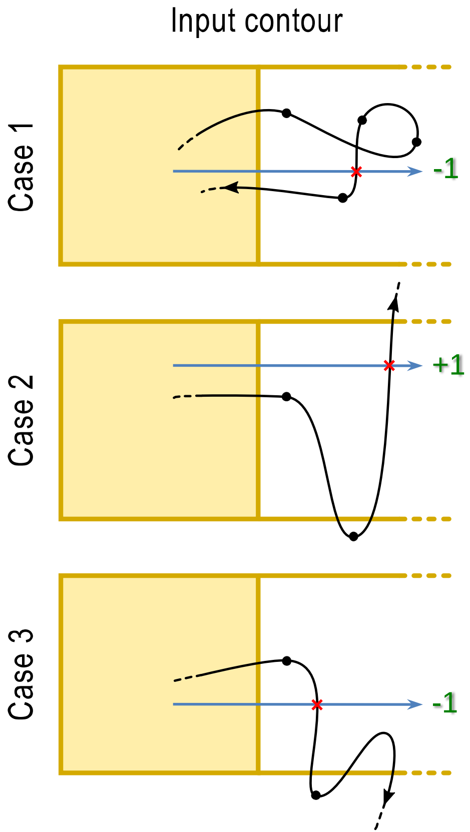
We only cast rays to the right

What goes on each cell?

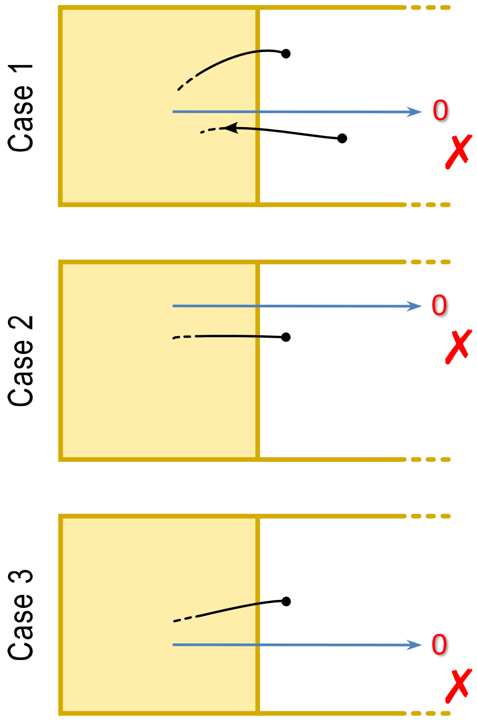
- Specialized subset of illustration
- Everything that is needed to render cell region
- *Only* what is needed to render cell region

Invariant: *The winding number of all paths about all samples in the cell region, computed from the cell contents, is exactly the same as in the complete illustration*

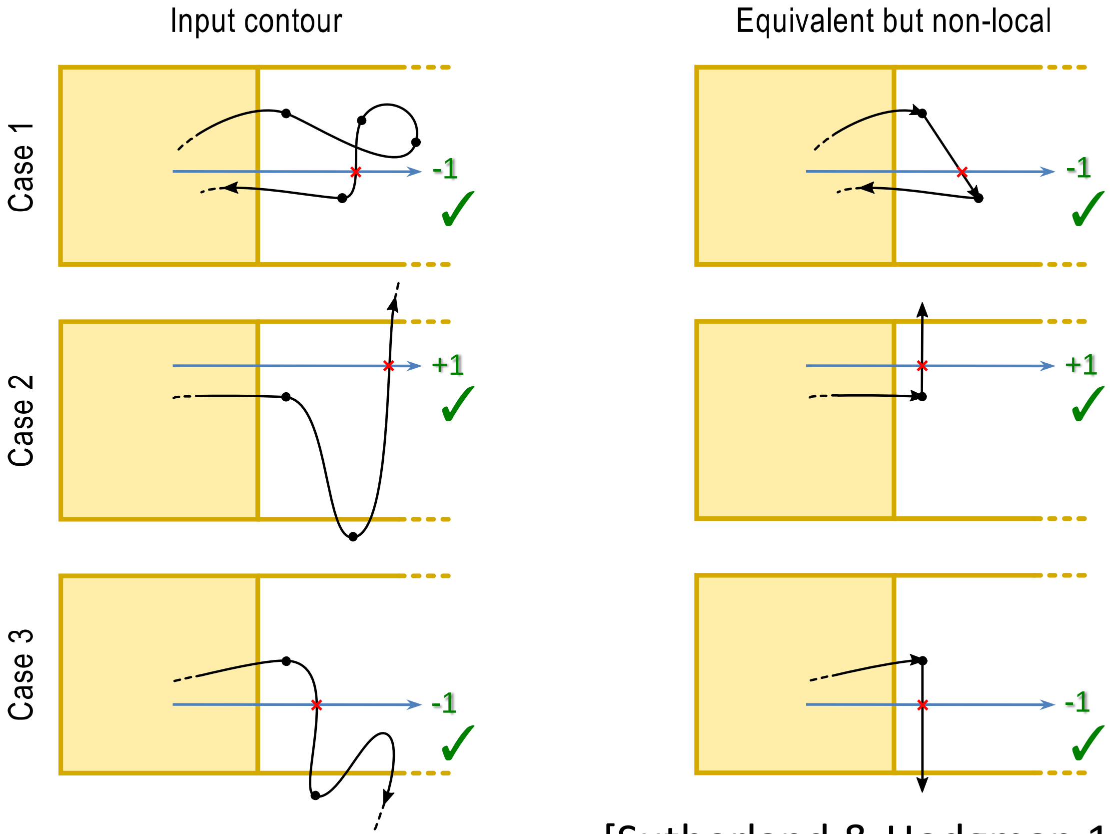
What about content to right of cell?



Cannot be simply discarded

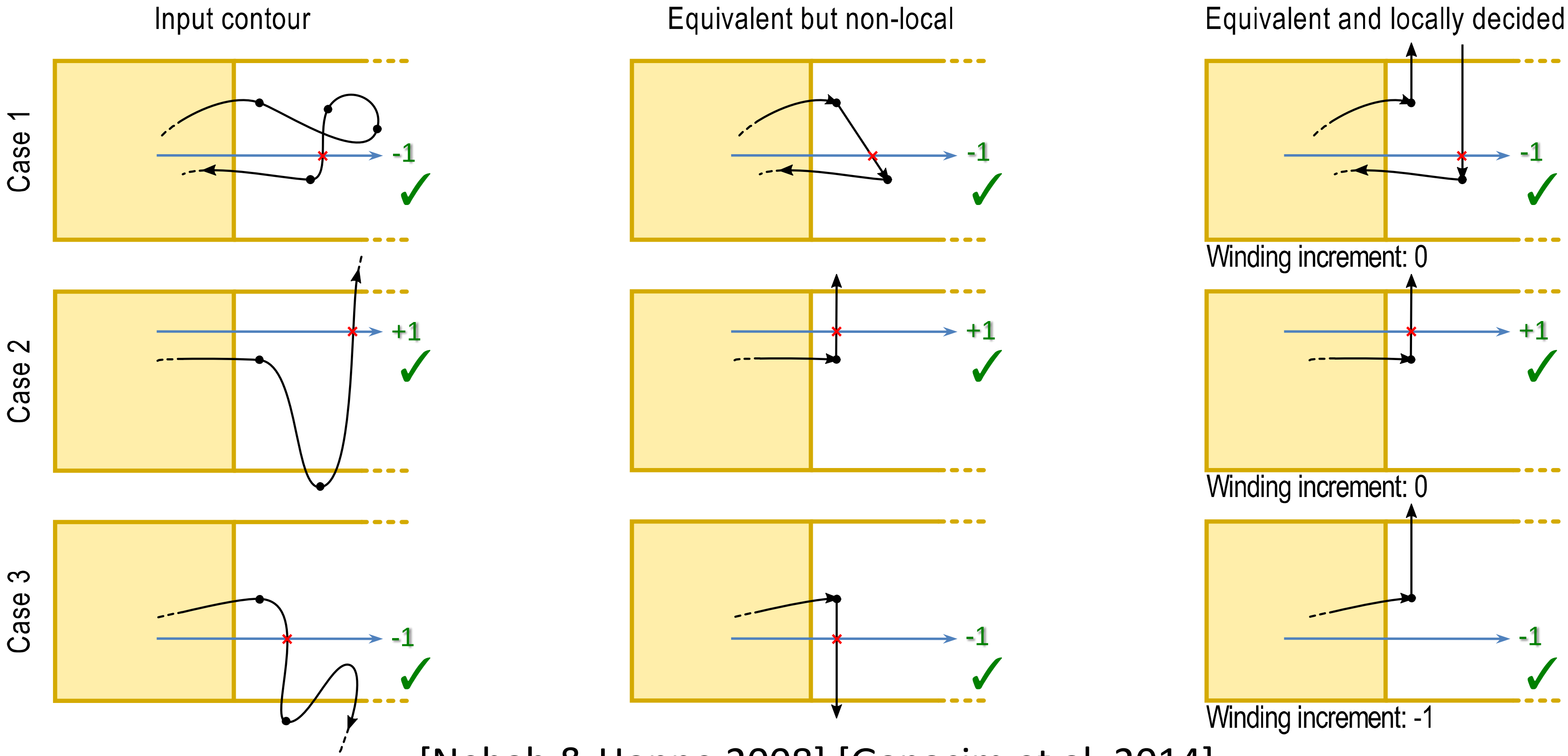


Could use clipping



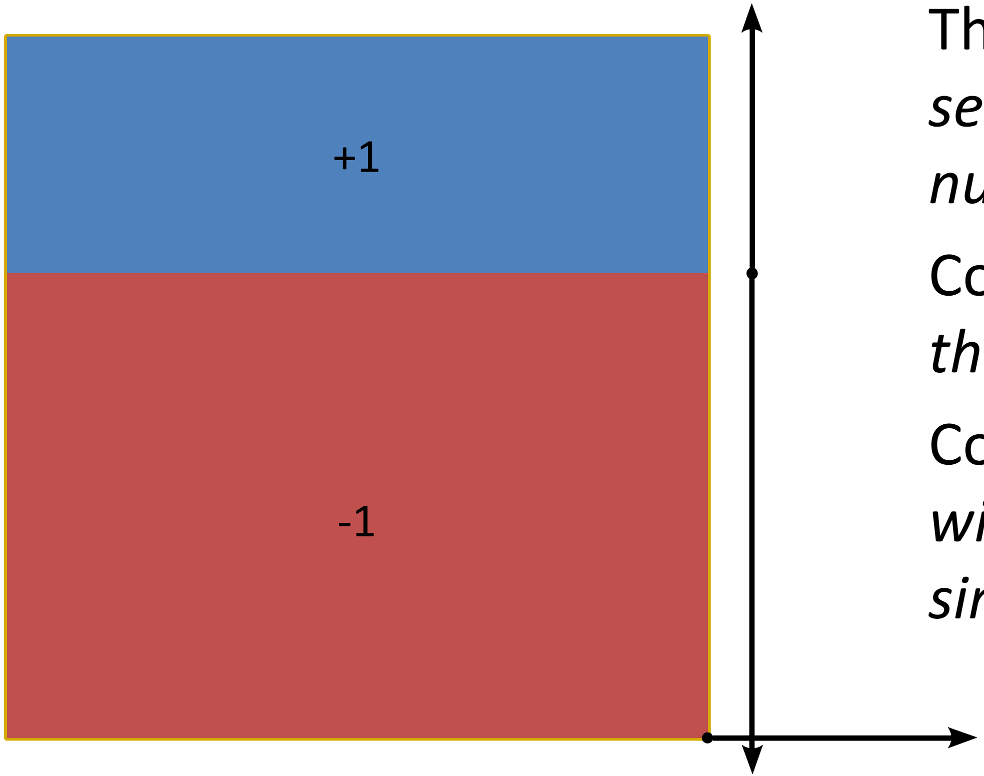
[Sutherland & Hodgman 1974]

Shortcut simplification



[Nehab & Hoppe 2008] [Ganacim et al. 2014]

Correctness of shortcut simplification

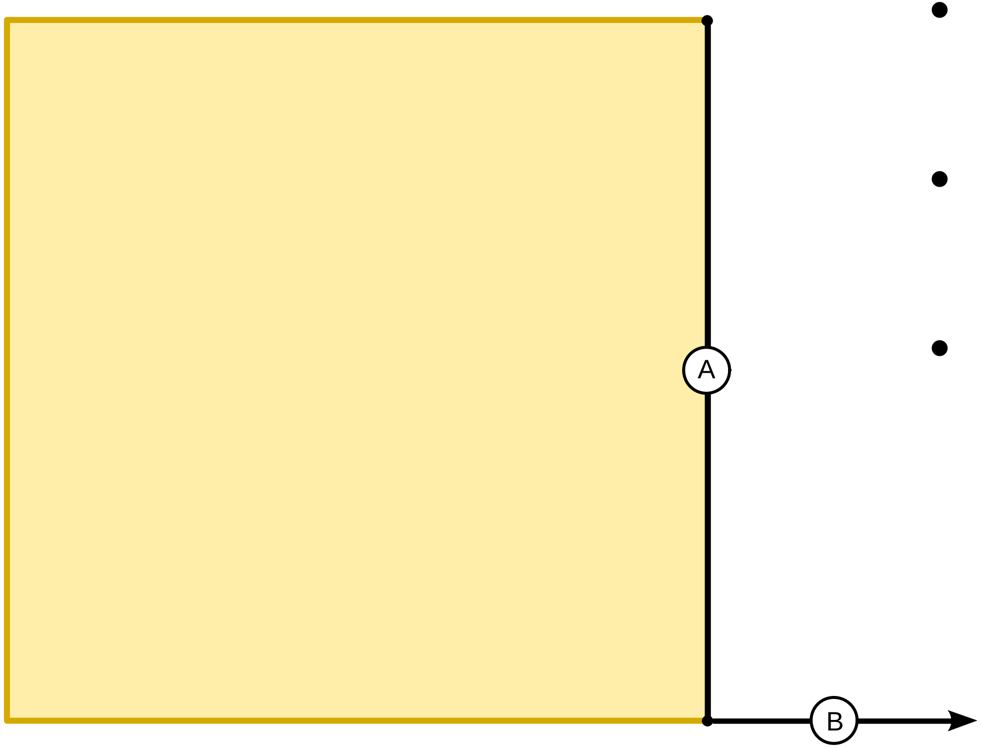


Theorem: *Flipping a shortcut segment adds ±1 to all winding numbers in the cell*

Corollary: *There is an integer k that restores the invariant*

Corollary: *The residual between winding number of input and simplification at* any *point gives k*

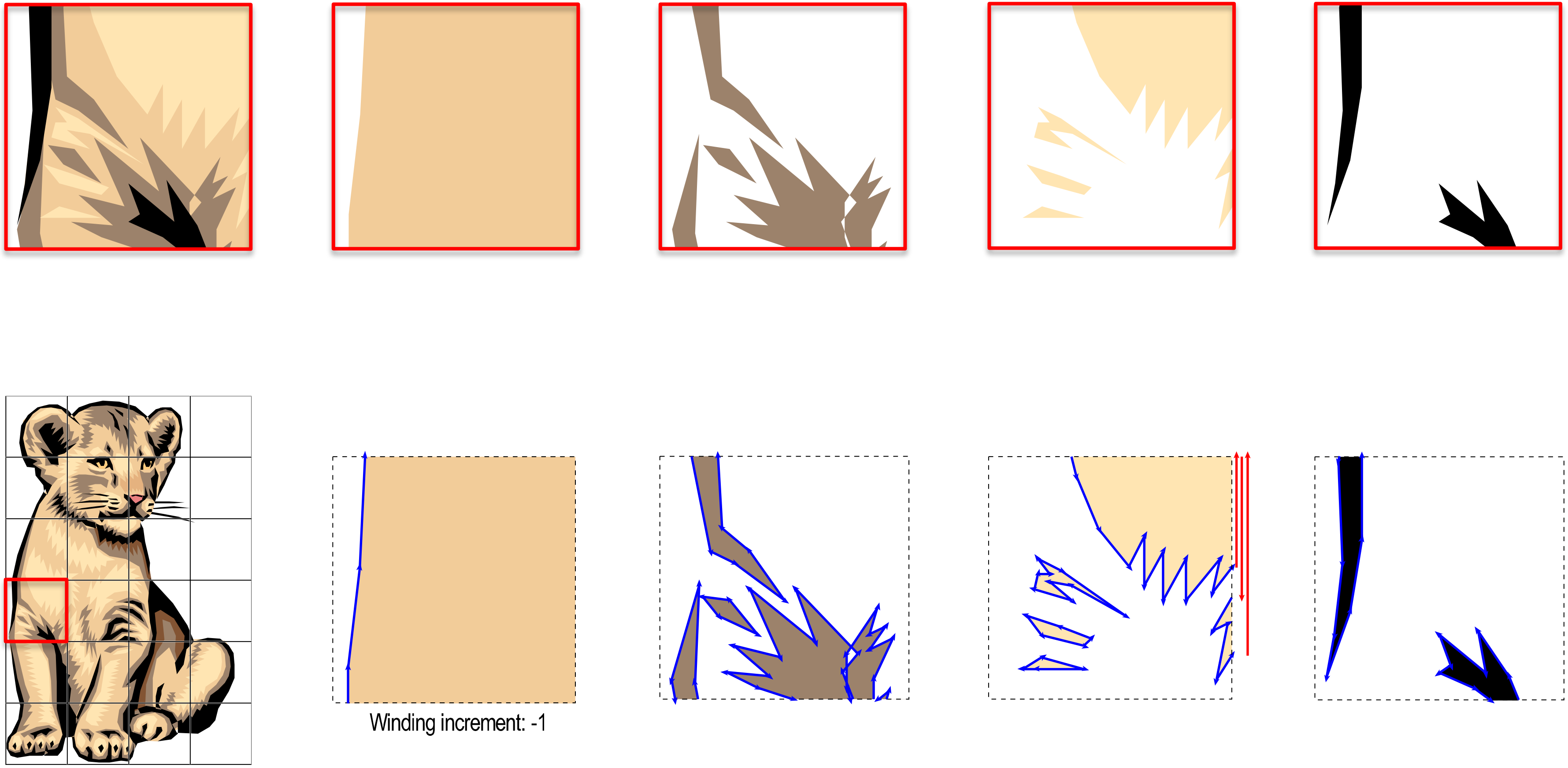
Shortcut simplification summary



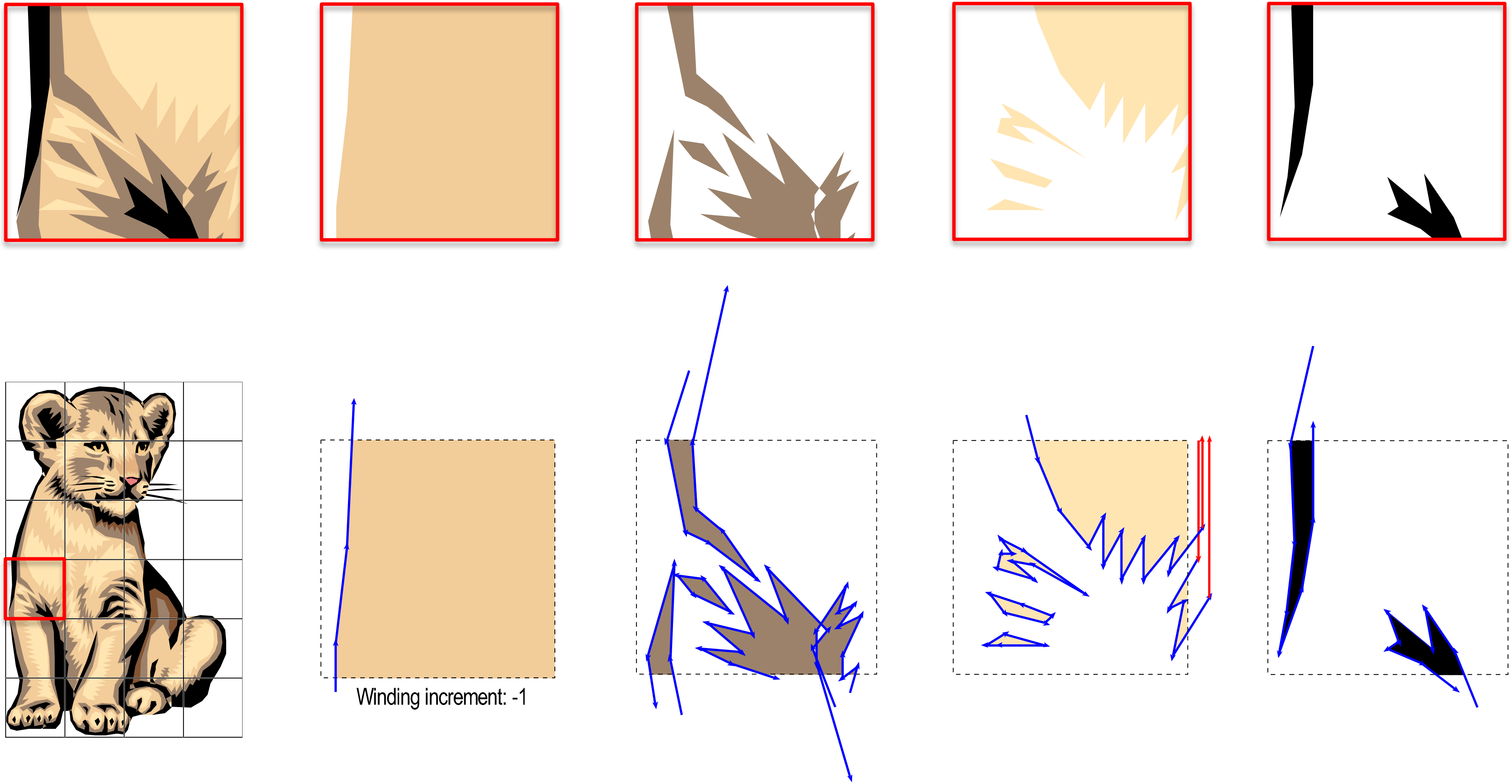
- Include segment if and only if it overlaps with cell
- Add shortcut *up* for segments that cross border Ⓐ
- Add winding increments for segments that cross border Ⓑ

Shortcut simplification preserves the invariant

[Warnock 1969]

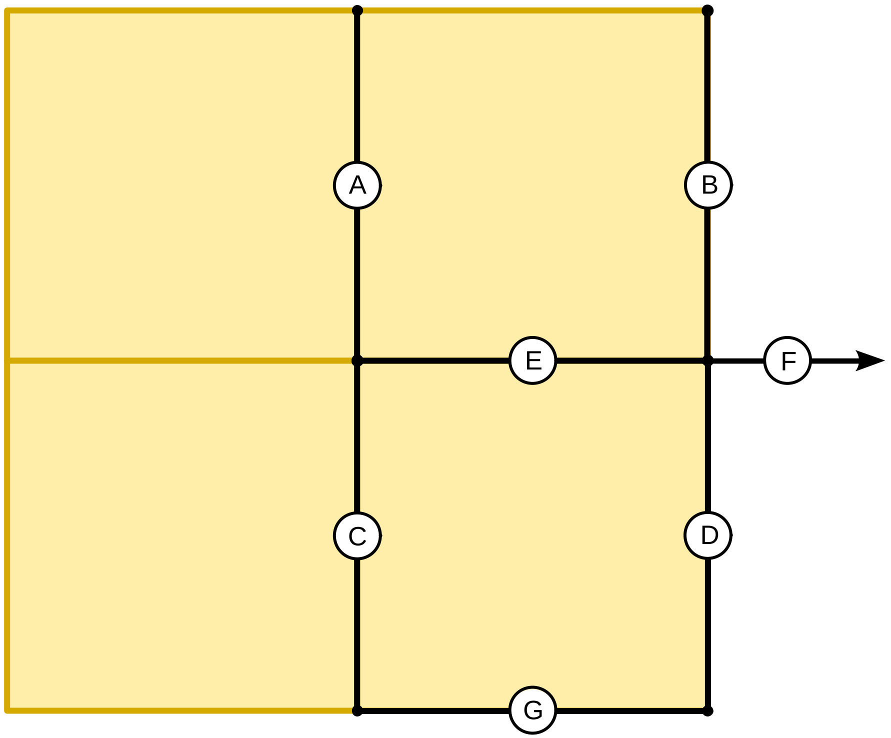


[Nehab & Hoppe 2008] (cut segments to cell boundaries)



Ours [Ganacim et al. 2014] (preserve original segments)

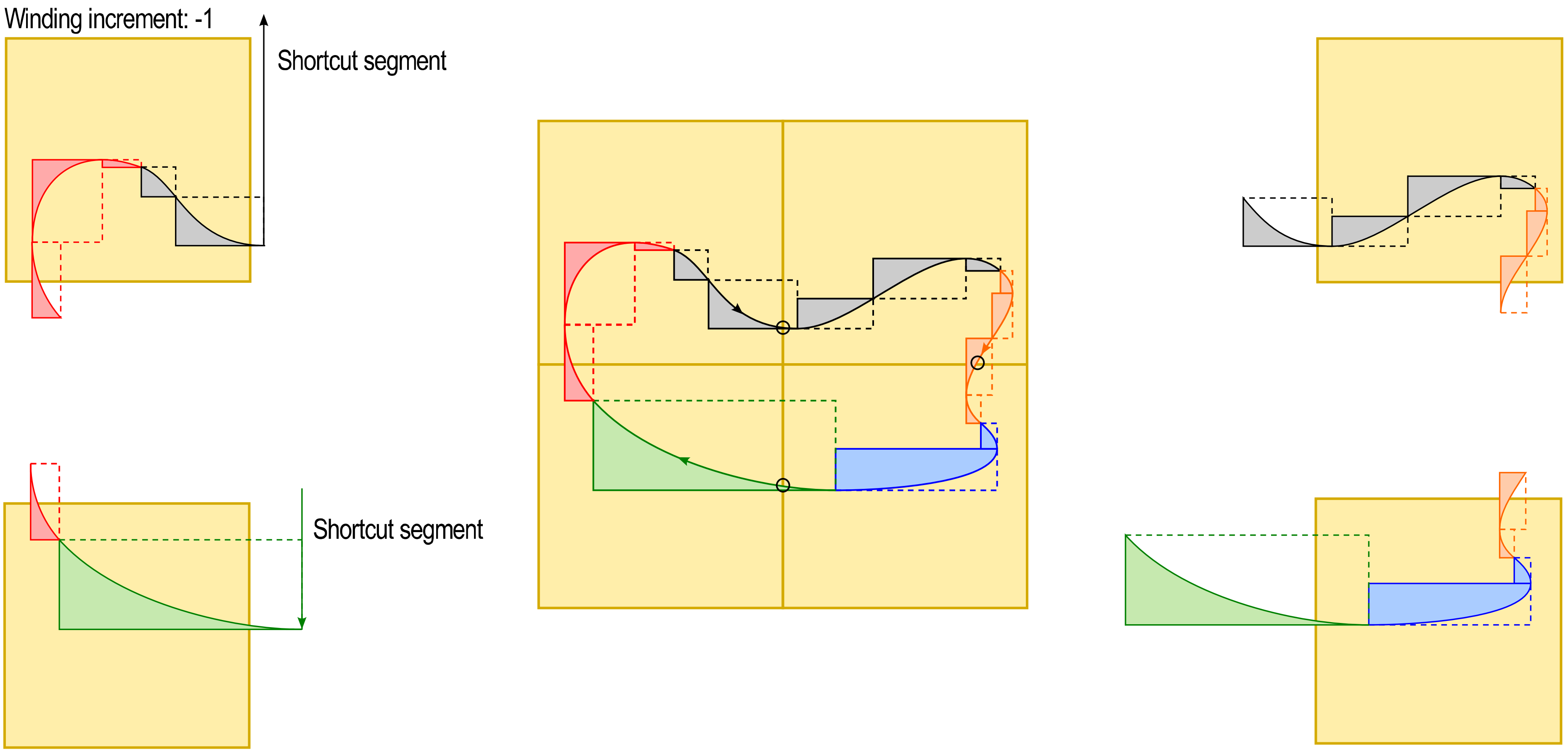
Subdivision



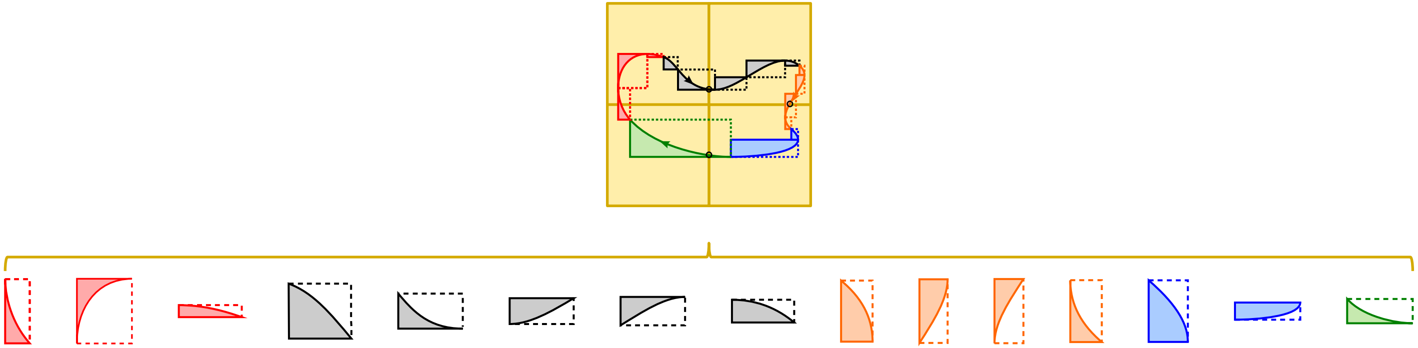
- Child cells inherit winding increments from parent
- Include those segments that overlap with each child cell
- Check border crossings with Ⓐ Ⓑ Ⓒ Ⓓ for shortcuts
- Check border crossings with Ⓔ Ⓕ Ⓖ for increments

Subdivision preserves the invariant

Example subdivision

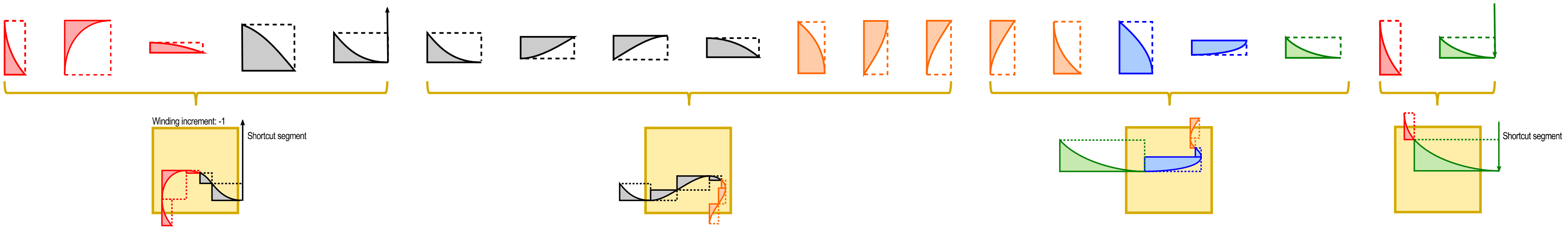


Parallel subdivision



Segment-parallel classification

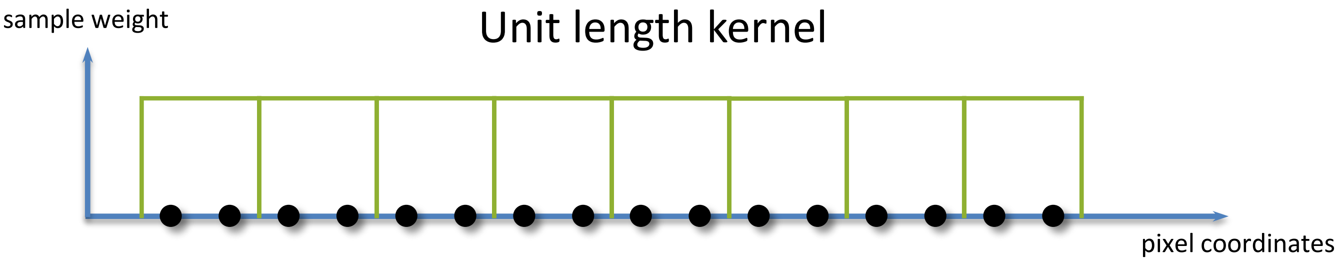
Parallel-scan followed by segment-parallel copy

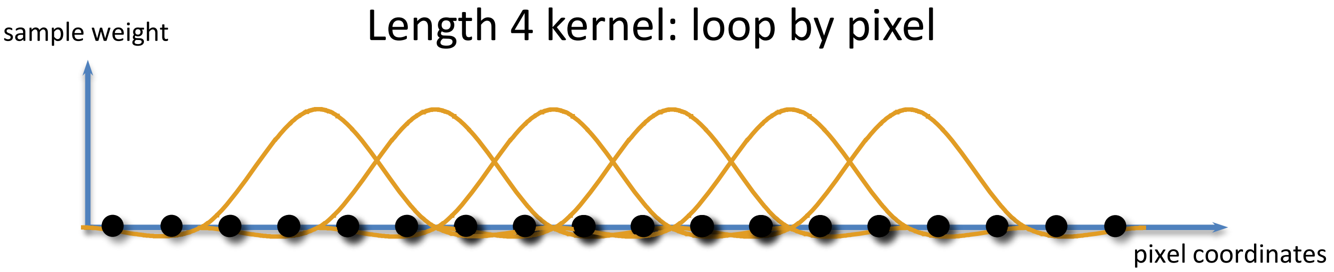


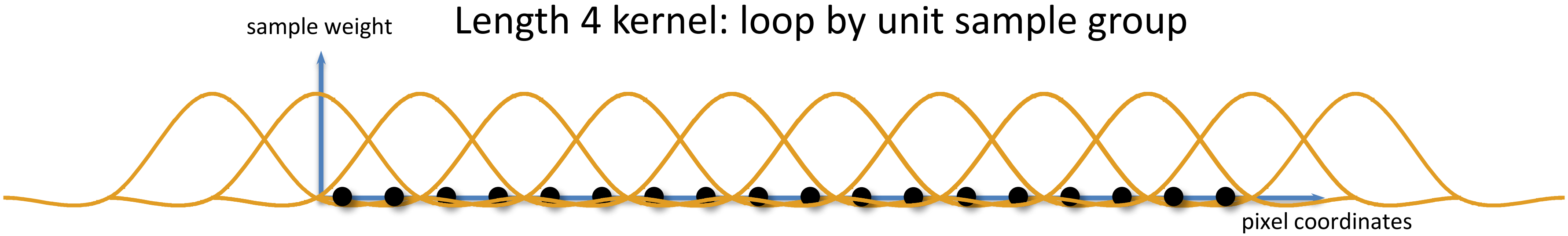
Sample sharing

SAMPLE SCHEDULER

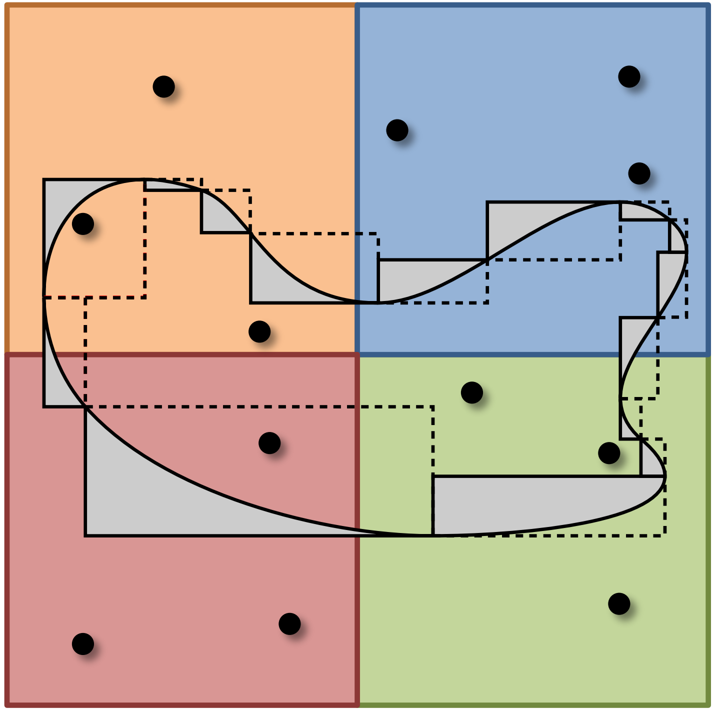
Sample sharing







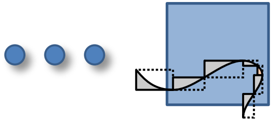
Parallel sample scheduling



Find tree cell for each sample

Group samples by cell

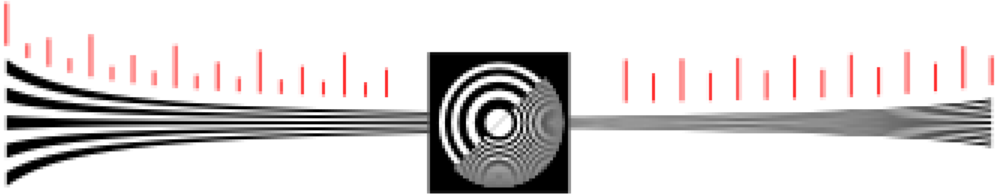
Compute sample colors…



… and integrate

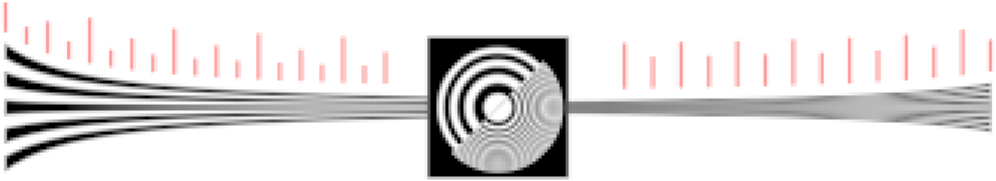
RESULTS

Alias, noise, and gamma

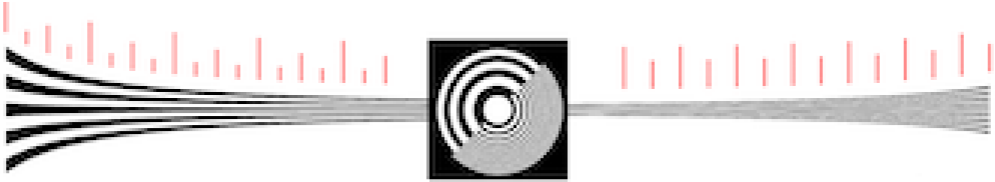


Most renderers
Gamma,
Box weights

NVPR
Linear, 8spp multisampling

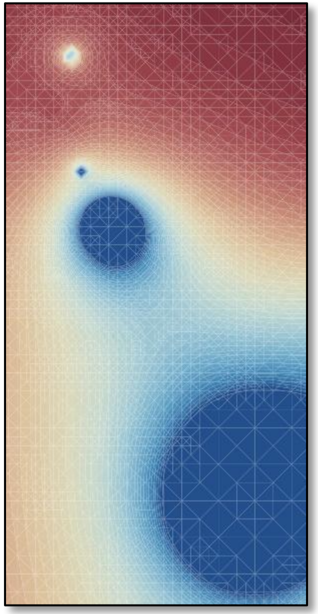


[Nehab & Hoppe 2008]
Linear, 1spp,
Prefilter approximation

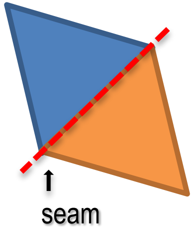


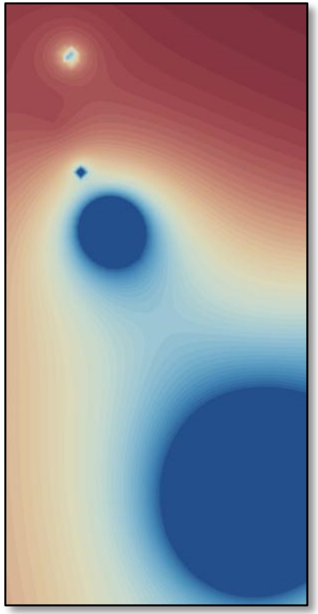
Ours [Ganacim et al. 2014]
Linear, 32x4x4 spp,
Cardinal Cubic B-spline weights

Conflation



Most renderers





Ours

- Resolving each path to pixels before blending causes artifacts
- Correct results require blending each sample independently
  - NVPR also correct

Examples of user-defined warps

foreshorten

swirl

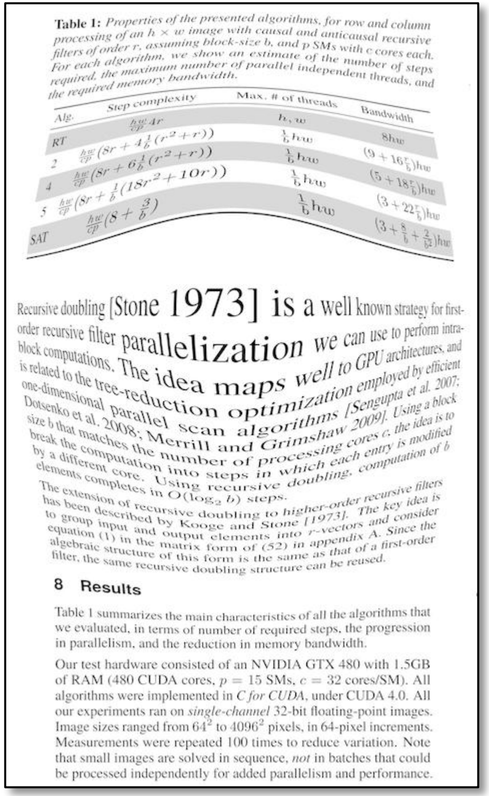


Table 1: *Properties of the presented algorithms, for row and column processing of an $h \times w$ image with causal and anticausal recursive filters of order r, assuming block-size b, and p SMs with c cores each. For each algorithm, we show an estimate of the number of steps required, the maximum number of parallel independent threads, and the required memory bandwidth.*

Alg.	Step complexity	Max. # of threads	Bandwidth
RT	$\frac{hw}{cp}4r$	h, w	$8hw$
2	$\frac{hw}{cp}(8r + 4\frac{1}{b}(r^2+r))$	$\frac{1}{b}hw$	$(9 + 16\frac{r}{b})hw$
4	$\frac{hw}{cp}(8r + 6\frac{1}{b}(r^2+r))$	$\frac{1}{b}hw$	$(5 + 18\frac{r}{b})hw$
5	$\frac{hw}{cp}(8r + \frac{1}{b}(18r^2+10r))$	$\frac{1}{b}hw$	$(3 + 22\frac{r}{b})hw$
SAT	$\frac{hw}{cp}(8 + \frac{3}{b})$	$\frac{1}{b}hw$	$(3 + \frac{8}{b} + \frac{2}{b^2})hw$

Recursive doubling [Stone 1973] is a well known strategy for first-order recursive filter parallelization we can use to perform intra-block computations. The idea maps well to GPU architectures, and is related to the tree-reduction optimization employed by efficient one-dimensional parallel scan algorithms [Sengupta et al. 2007; Dotsenko et al. 2008; Merrill and Grimshaw 2009]. Using a block size b that matches the number of processing cores c, the idea is to break the computation into steps in which each entry is modified by a different core. Using recursive doubling, computation of b elements completes in O($\log_2 b$) steps.

The extension of recursive doubling to higher-order recursive filters has been described by Kooge and Stone [1973]. The key idea is to group input and output elements into r-vectors and consider equation (1) in the matrix form of (52) in appendix A. Since the algebraic structure of this form is the same as that of a first-order filter, the same recursive doubling structure can be reused.

8 Results

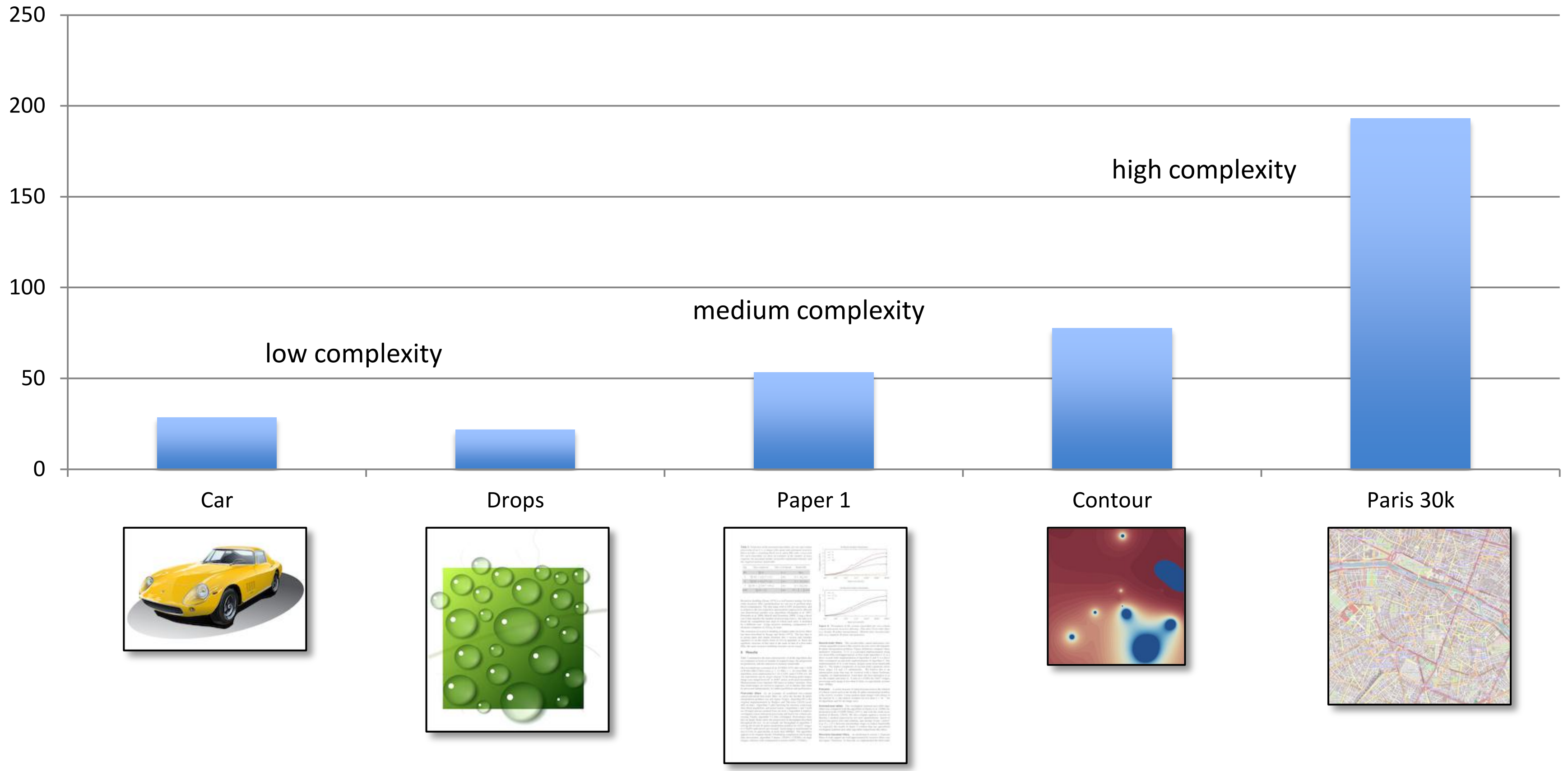
Table 1 summarizes the main characteristics of all the algorithms that we evaluated, in terms of number of required steps, the progression in parallelism, and the reduction in memory bandwidth.

Our test hardware consisted of an NVIDIA GTX 480 with 1.5GB of RAM (480 CUDA cores, $p = 15$ SMs, $c = 32$ cores/SM). All algorithms were implemented in *C for CUDA*, under CUDA 4.0. All our experiments ran on *single-channel* 32-bit floating-point images. Image sizes ranged from 64^2 to 4096^2 pixels, in 64-pixel increments. Measurements were repeated 100 times to reduce variation. Note that small images are solved in sequence, *not* in batches that could be processed independently for added parallelism and performance.

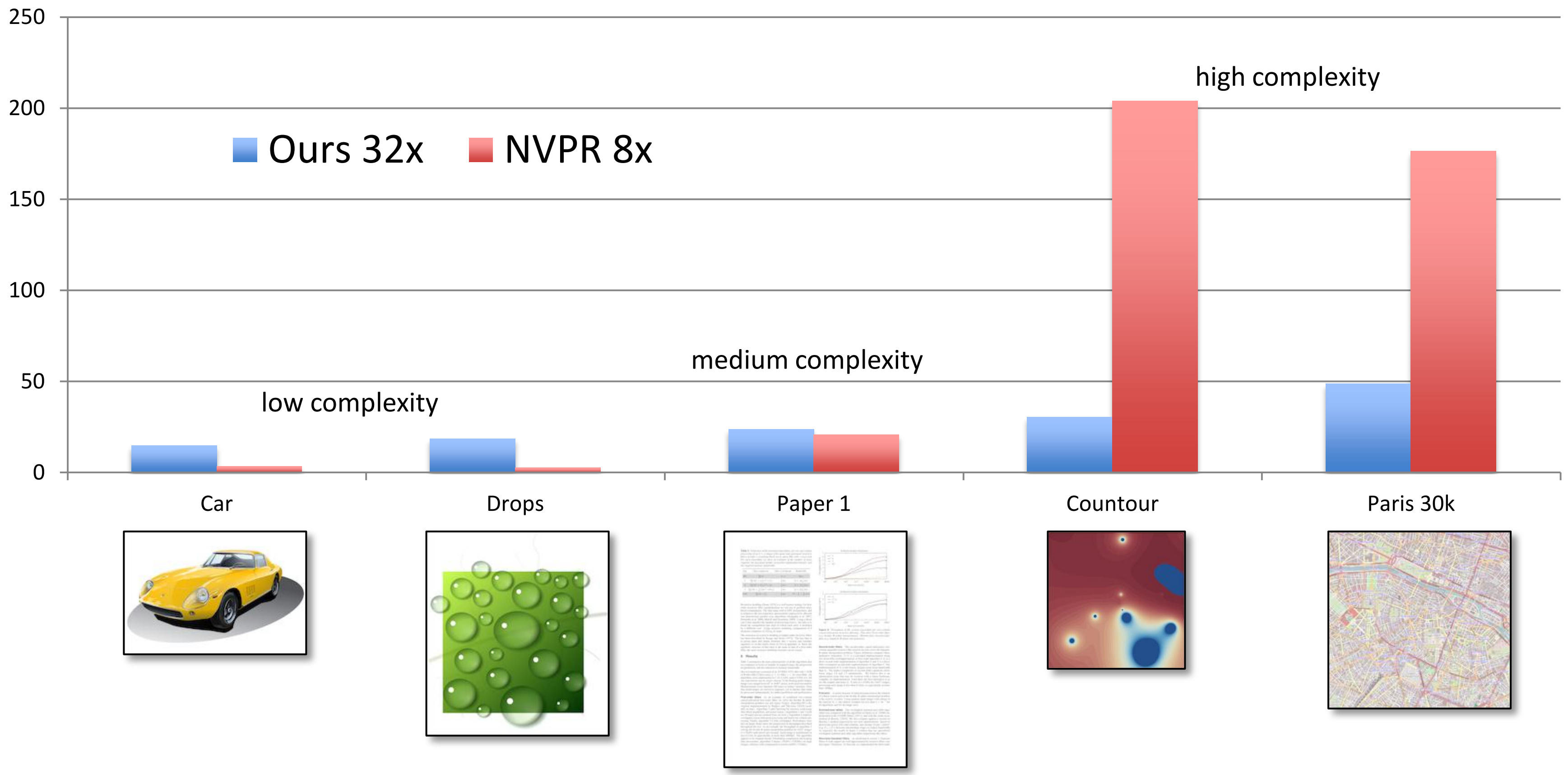
lens

- Warp each sample position
  - *Not* a post-processing step
- Engages sample scheduler

Preprocessing time (ms)



Rendering time (ms)



Left out of talk

- New algorithm for rendering with clip-paths
  - Full SVG semantics, no stack or recursion
- Parallel pruning algorithm in preprocessing
  - Eliminates occluded or clipped paths from cells
- Please see paper for other omitted details

Several ideas for future work

- Support for rational cubics (enable object-space warps)
- Support for mesh-based gradients
- Parallel stroke-to-fill conversion
- Transparency groups
- Subpixel rendering (e.g., ClearType)
- Raster effects over groups (e.g., Gaussian Blur)
- Different subdivision strategies (e.g, kd-tree)
- Port back to CPU with multi-threaded vector code
- Hardware implementation

Conclusions

- Fully parallel vector graphics rendering solution
- Interactive preprocessing times
- Unprecedented *output quality*
- Support for user-defined warps
- Best option *for complex illustrations*
- Source-code available

www.impa.br/~diego/projects/GanEtAl14/