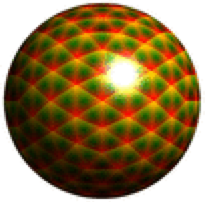
HPG
HIGH-PERFORMANCE GRAPHICS
Algorithms Hardware Systems

Morphological Antialiasing

Alexander Reshetov

Intel Labs

2 August 2009



Talk Outline

- **Prior Art**
  - **Problems, solutions,** ideas
- **Morphological Antialiasing $\in$ image-based AA**
  - Input: **image.** Output: **‘better looking’ image**
  - **Algorithm, features,** limitations
- **Demos during the talk (hopefully)**

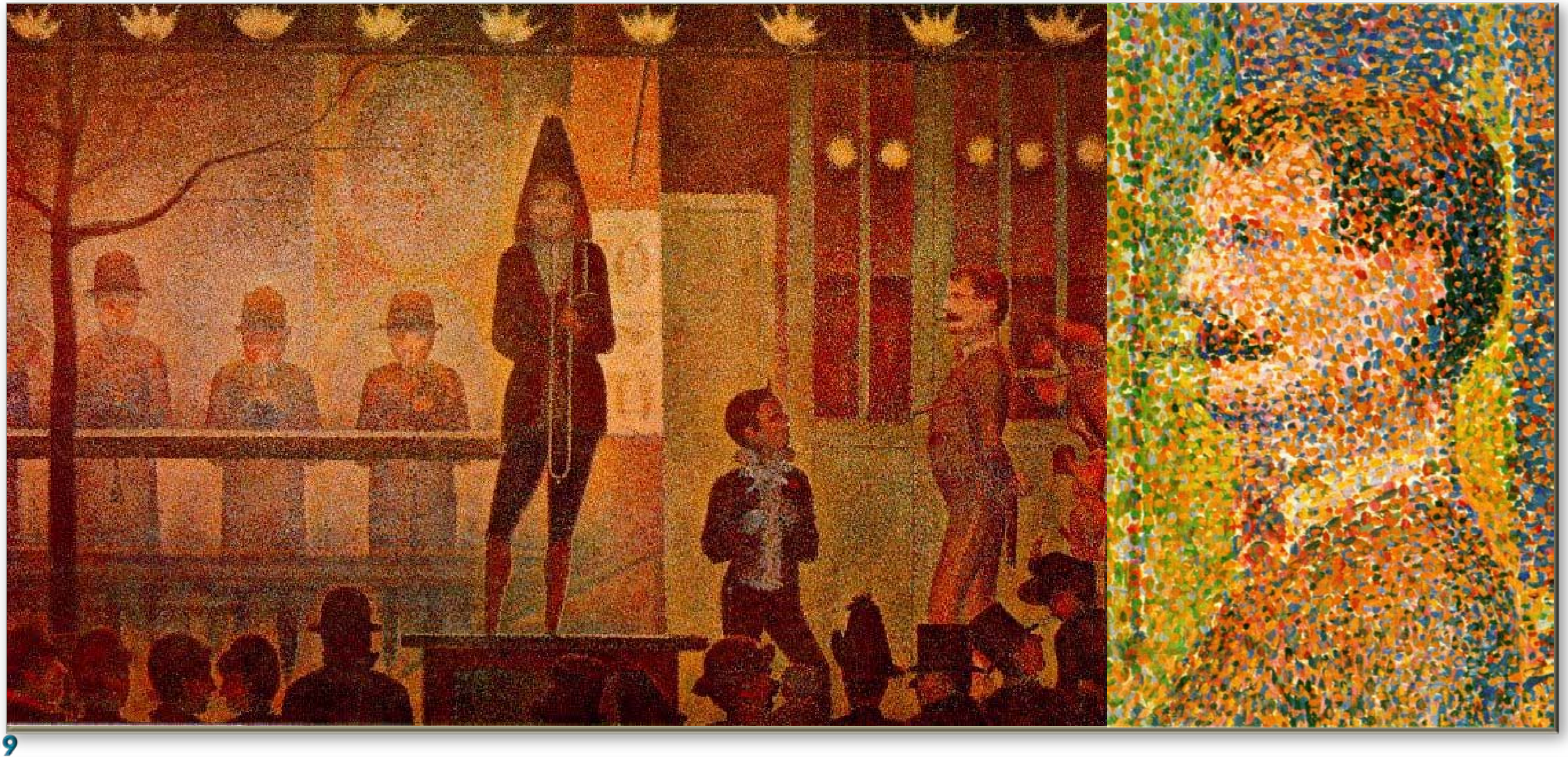
Prior Art: the Importance of Creating Nice Pictures

Leonardo da Vinci – Inventor of Antialiasing

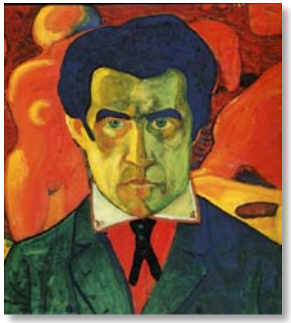
sfumato: **painting technique "without lines or borders, in the manner of smoke or beyond the focus plane"**

Pixels Debut

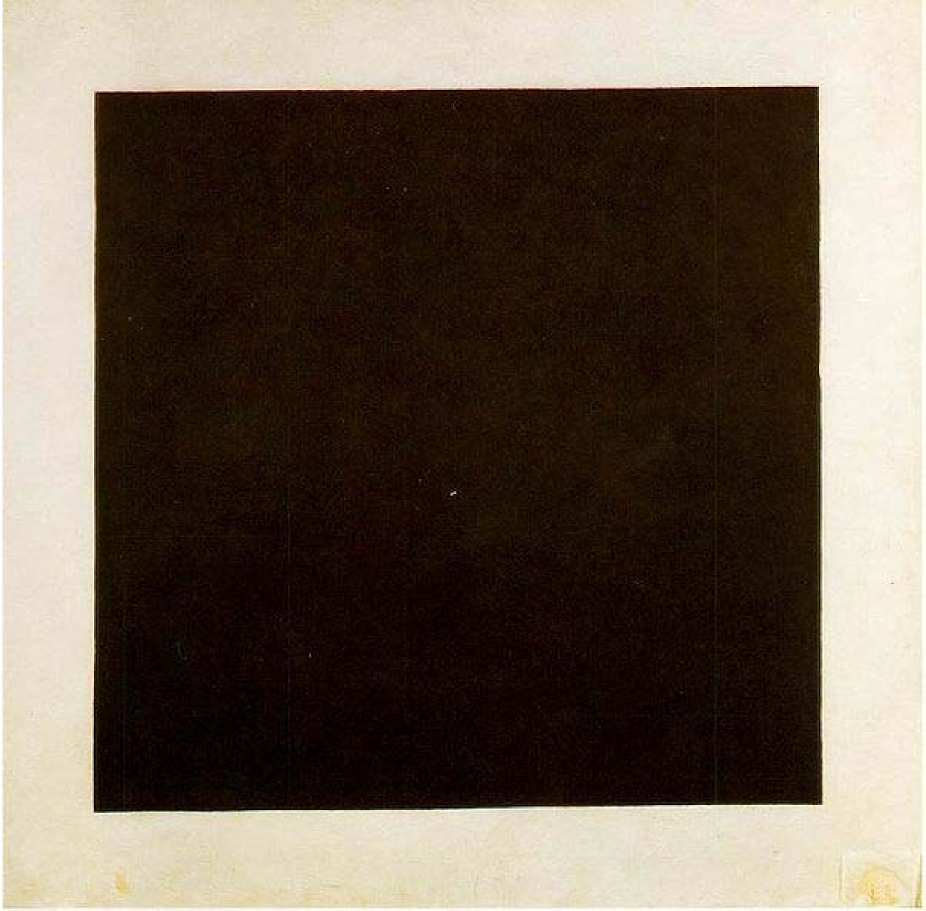
Georges-Pierre Seurat's *La Parade*, 1888 (from Wikipedia):
"The tiny juxtaposed dots of multi-colored paint allow the viewer's eye to blend colors optically, rather than having the colors blended on the canvas or pre-blended as a material pigment."



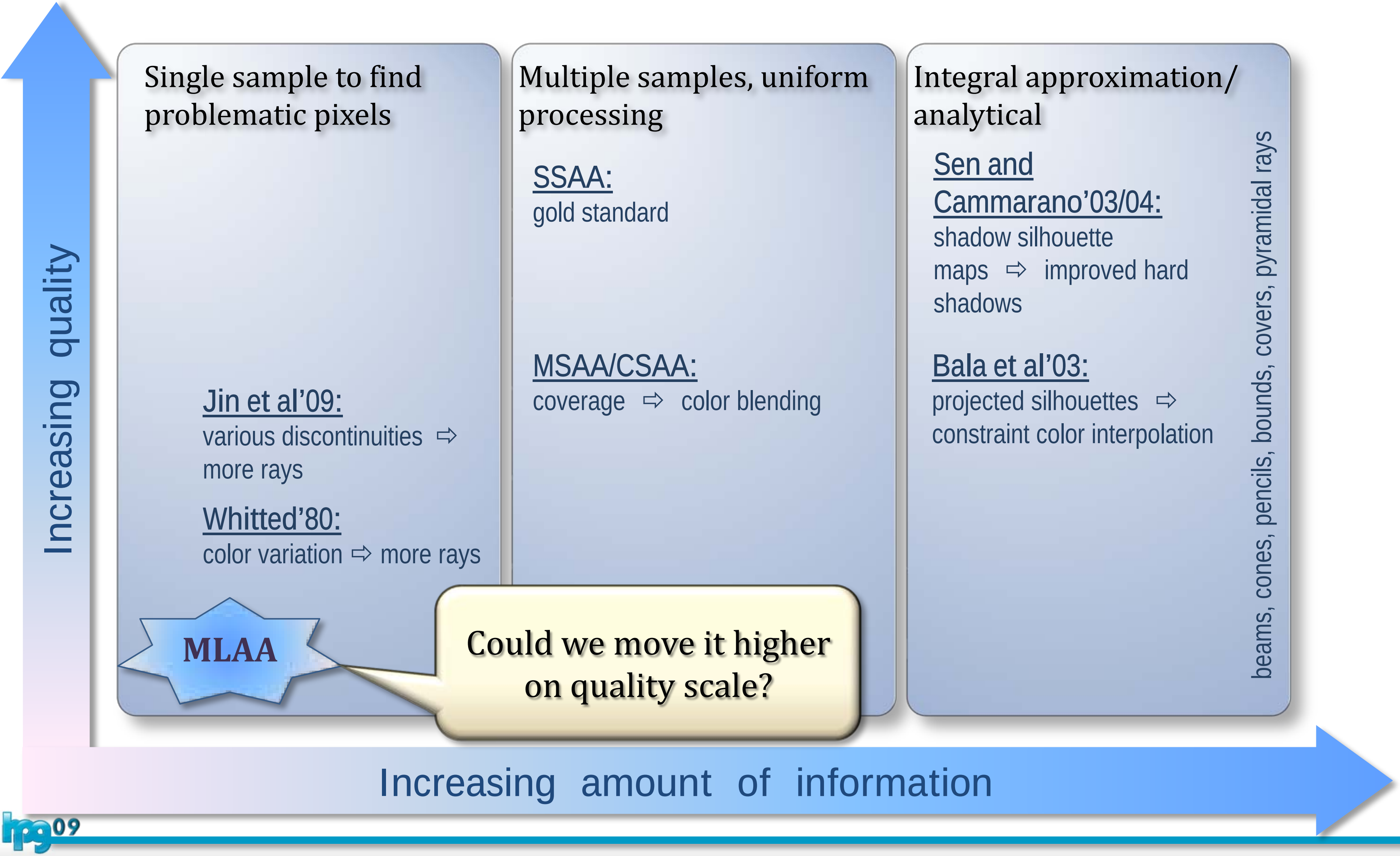
The One And Only…



Kazimir Malevich's *Black Square*, 1915, Oil on Canvas



The More We Know…



Increasing quality

Increasing amount of information

Single sample to find problematic pixels

Jin et al’09:
various discontinuities ⇨ more rays

Whitted’80:
color variation ⇨ more rays

MLAA

Could we move it higher on quality scale?

Multiple samples, uniform processing

SSAA:
gold standard

MSAA/CSAA:
coverage ⇨ color blending

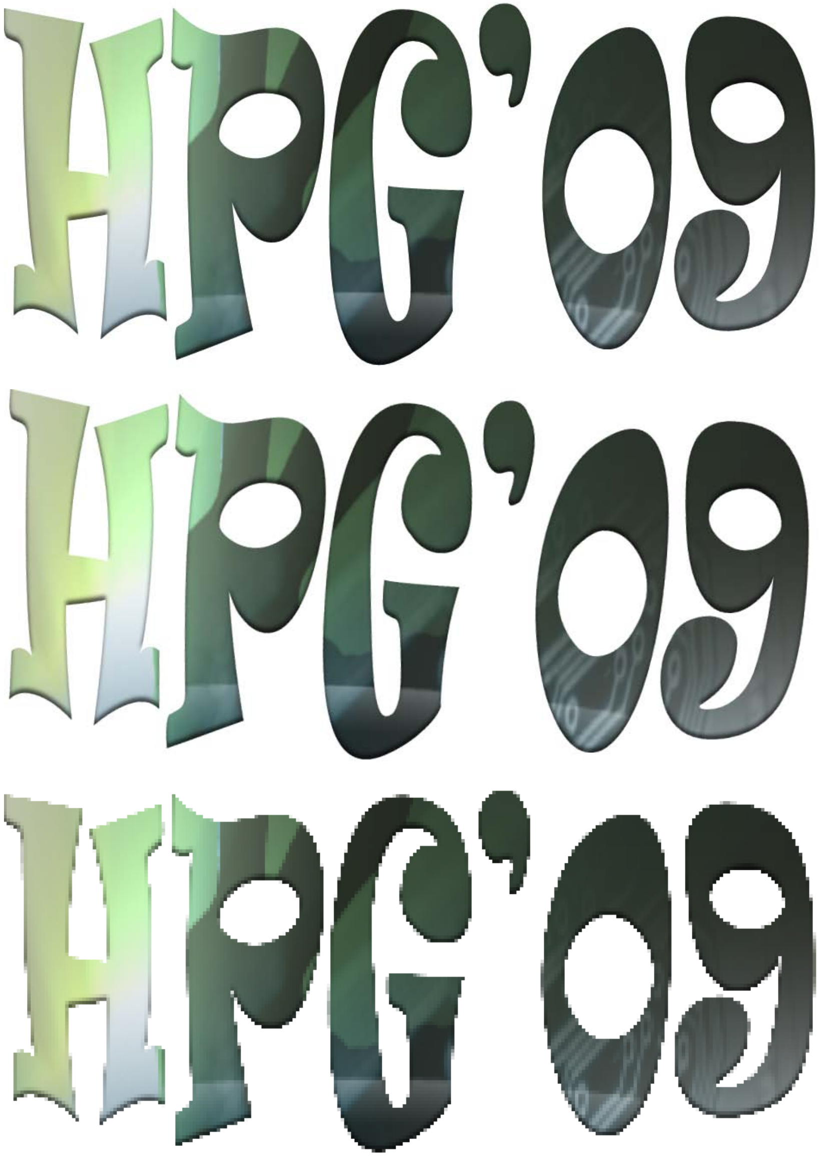
Integral approximation/ analytical

Sen and Cammarano’03/04:
shadow silhouette maps ⇨ improved hard shadows

Bala et al’03:
projected silhouettes ⇨ constraint color interpolation

beams, cones, pencils, bounds, covers, pyramidal rays

What is ‘Better Looking’ Image?



- **top:** ***reference image***
- **psnr(top, middle) = 14.8**
- **psnr(top, bottom) = 23.2**

(**peak-signal-to-noise-ratio**: bigger number means smaller average error)

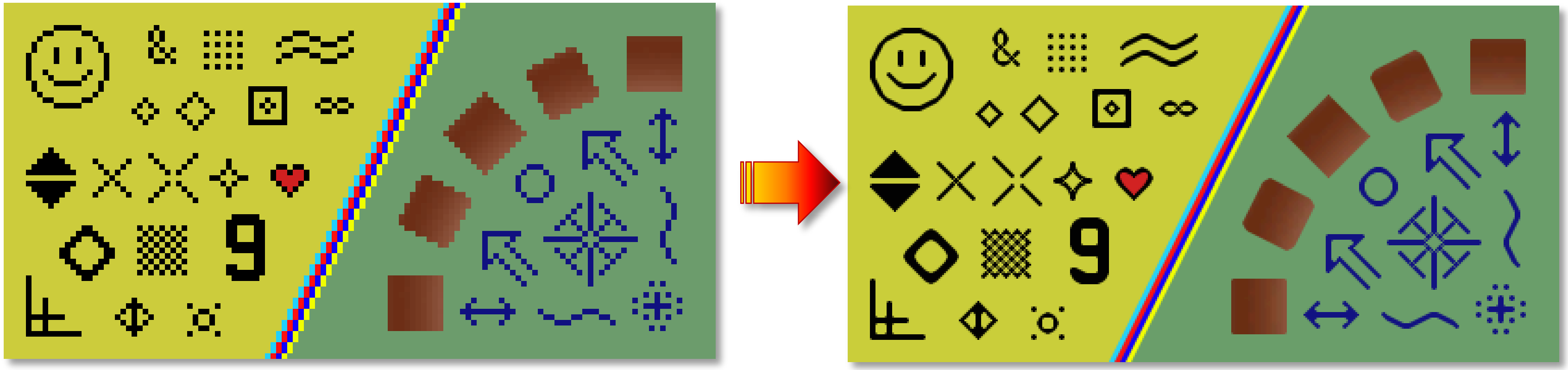
Bottom line:

We’re in business of creating nice pictures (mostly)

Early Pixel Art Scaling Algorithms

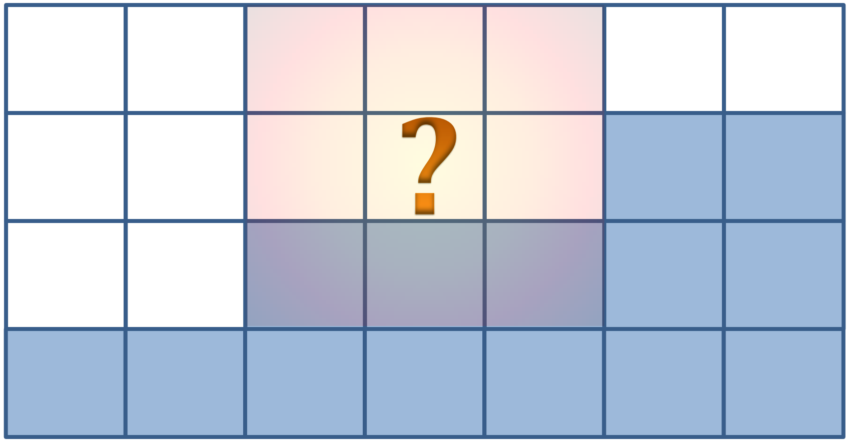
- Motivation: **to allow original low-res computer games run on better hardware**

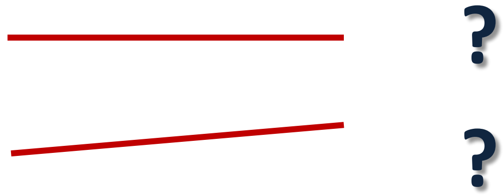
- Algorithms **(from Wikipedia): EPX/Scale2x/AdvMAME2x, Scale3x/AdvMAME3x, Eagle, 2xSaI, hq*n*x**



Pixel Art Scaling in a Nutshell

- General approach: **local rule-based filter**
- **It is fine approach for the task at hand**
- Problem: **non-local influence is ignored**





- **Is it** ?
- **Or** ?

The Plan

- **How can we do better? (since filtering doesn't work that well)**
- **We can borrow ideas from Bala et al'03 and Sen and Cammarano'03/04:**
  - **Reconstruct (linear) discontinuities in the image**
  - **Filter around these discontinuities**
- Goal: **create better looking image**
- Non-goal: **compete with SSAA (we simply don't have data for this)**

A picture is worth a thousand words (a Chinese proverb?)

"畫意能達萬言"

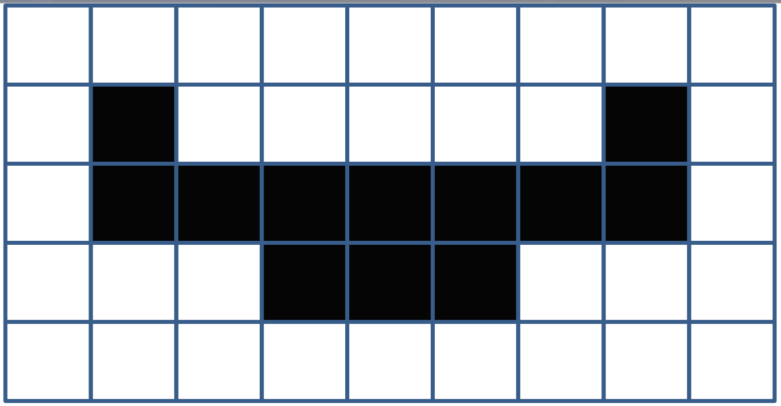
Before describing the algorithm, let's run this demo…

MLAA Steps

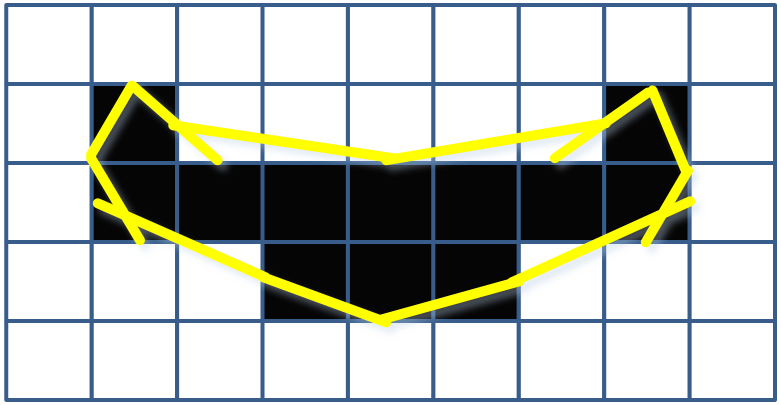
- **For any given image**
  1. **Find piecewise linear segments which, hopefully, will bound homogeneous areas in the image**
  2. **Interpolate colors near these segments**
- **And we want to do it as simple as possible, since we only have a color data anyway (**Occam's razor**)**

MLAA Steps Again (Illustration)

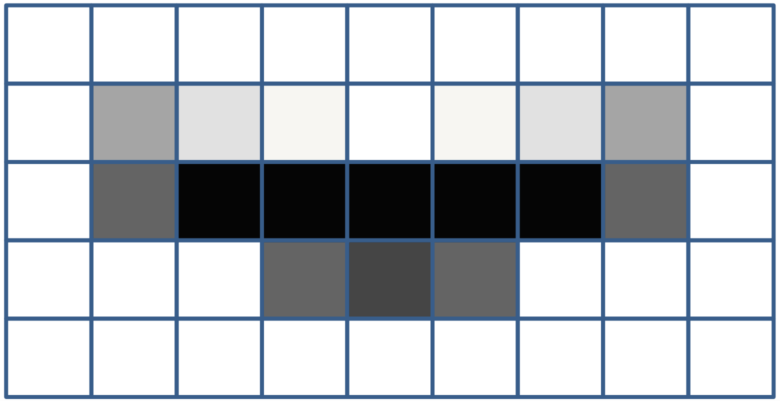
- For any given image



1. Find piecewise linear segments (don’t have to be connected)

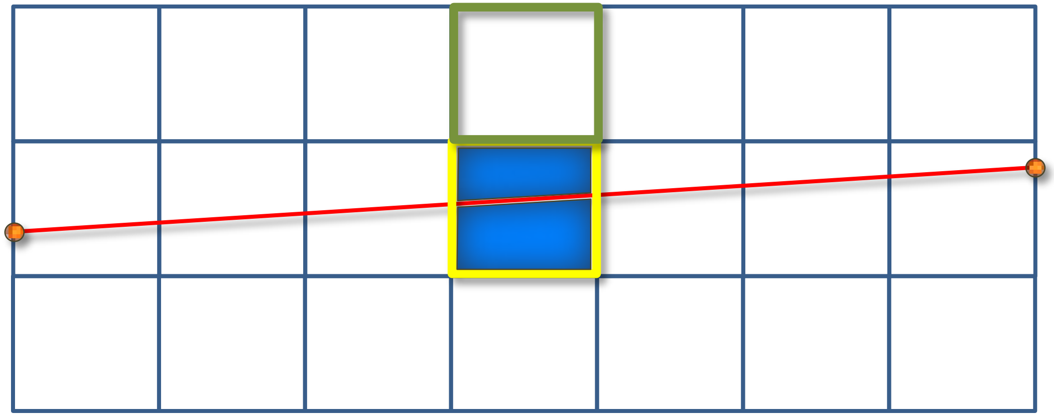


2. Interpolate colors near these segments (constrained filtering)



2. Color Blending

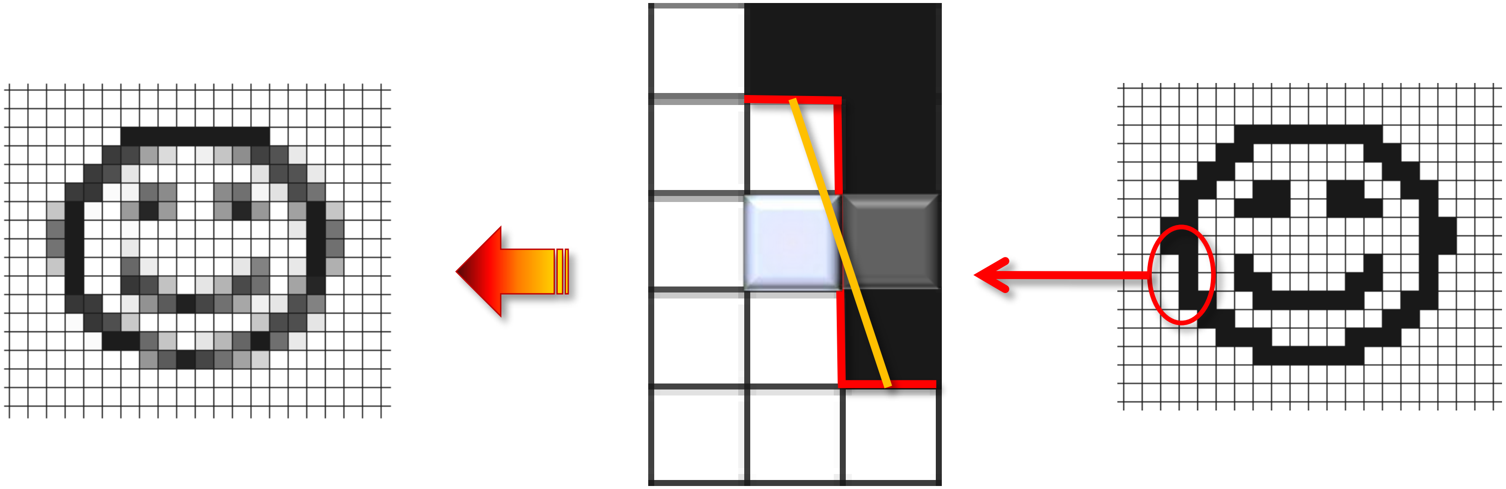
- For each segment
  - Segments start and end on pixel boundaries (by design)
  - For any pixel that is intersected by the segment
  - Compute areas of 2 trapezoids formed by segment ∩ pixel
  - Choose one neighboring pixel (defined by the segment)
  - Set the new color (of the yellow pixel) to the weighted sum of the old color and the color of the neighboring pixel with the weights ~ trapezoid areas



Note: each pixel could be blended multiple times

(we will have to take care of this if multiple threads are used)

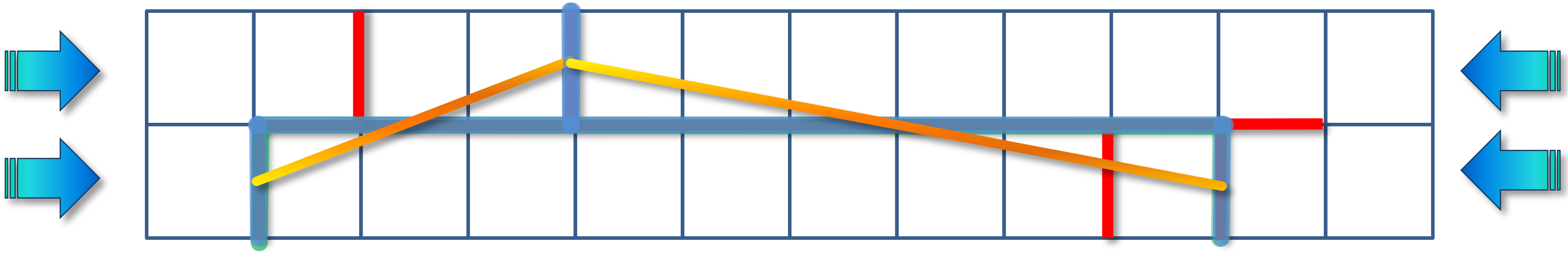
1. Finding Segments (B&W Example)



- What we want: yellow segment. We find it by looking for
- axis-aligned separation lines – ones which separate B and W pixels (or *different* ones in color images)
- For *__each__* separation line (going between pixels)
    - we find *all* orthogonal ones and use middle points of the corner pixels' edges
    - to recreate a line segment (considering *__all__* possible shapes, Z-shape is shown)
    - Pixels (□▩) on the opposite side of the separation line will be used for blending @ the second step

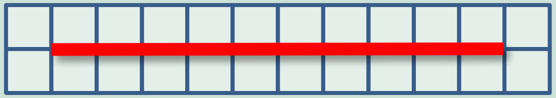
What is Different for Color Images

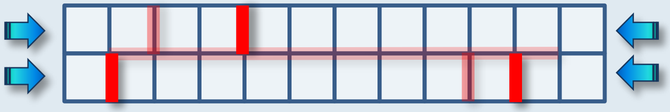
- All we need is *f*(pixel1,pixel2) which tells us if two pixels are “different”
- For each line separating different pixels there could be multiple orthogonal separation lines
- We execute 4 loops and
- use an additional criteria to identify orthogonal lines, which allow smooth color blending, and use only the first one found at each loop to create all possible shapes

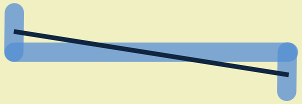


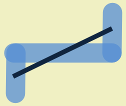
MLAA as-it-is on One Foil

```
// Get the next row/column for the current thread
while ((line = img.nextLine(threadid)) != NULL)  { // Lines are interleaved for different threads
   // Loop over all separation lines in the current row/column
  SeparationLine sep (0);
  while ((sep = img.nextSeparationLine(line, sep)) != NULL)  {
      enum {TOPLEFT, TOPRIGHT, BOTLEFT, BOTRIGHT, FOUR};
      int ort[FOUR];   // up to 4 suitable orthogonal separation lines
      float h[FOUR];   // height offsets from the separation line (0.5 for B&W)
      for (int path = TOPLEFT; path < FOUR; path++)   // Find all
         ort[path] = img.orthogonal(path, h, sep);       // suitable orthogonal lines
      int done = 0; // how many shapes are processed; each shape is defined by  2 ort indices  and 2 heights
      // z-shape     └┐  resulting in line-segment going as  \
      if (ort[TOPLEFT]!= -1 && ort[BOTRIGHT] != -1 && ort[TOPLEFT] < ort[BOTRIGHT])
         done += img.blendInterval(ort, h, TOPLEFT, BOTRIGHT);
      // z-shape     ┌┘  resulting in line-segment going as /
      if (ort[BOTLEFT]!= -1 && ort[TOPRIGHT] != -1 && ort[BOTLEFT] < ort[TOPRIGHT])
         done += img.blendInterval(ort, h, BOTLEFT, TOPRIGHT);
      // u-shape     ┌┐  resulting in 2 line-segments going as ∧ (only if there are no Z-shapes)
      if (!done && ort[TOPLEFT]!= -1 && ort[TOPRIGHT] != -1 && ort[TOPLEFT] < ort[TOPRIGHT])
         img.blendInterval(ort, h, TOPLEFT, TOPRIGHT);
      // u-shape     └┘  resulting in 2 line-segments going as ∨ (only if there are no Z-shapes)
      if (!done && ort[BOTLEFT]!= -1 && ort[BOTRIGHT] != -1 && ort[TOPLEFT] < ort[BOTRIGHT])
         img.blendInterval(ort, h, BOTLEFT, BOTRIGHT);
   }
}
```



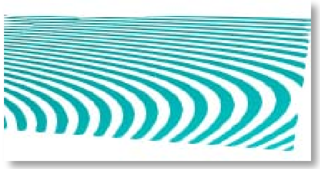


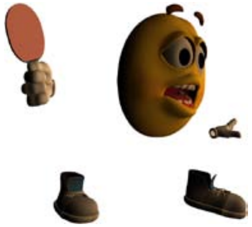




Usability of MLAA

- There are no limitations per se, but certain situations *will* cause artifacts.
- There are 2 groups of artifacts:
  - Typical for all one-sample-per-pixel algorithms @ Nyquist limit
  - Specific for MLAA (more or less)
    - Abrupt pixel color updates for slow moving objects
    - Varying lighting can trigger threshold-based color changes
    - Small fonts (esp. clear-typed) will look ugly





TOL game, courtesy of
Jacco Bikker
and his students

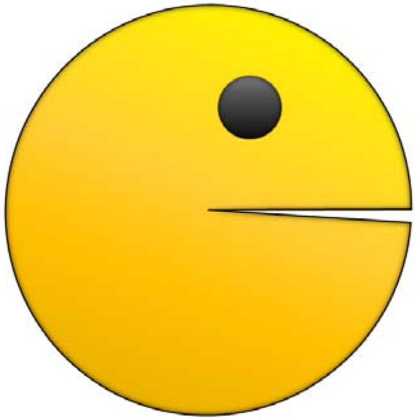
MLAA Characterization

- MLAA in a few words: **it is an image filtering constrained by the linear segments, reconstructed from the input image**
- Pros: **uses only color data**
  - **Applicable for all images**
  - **Can be executed asynchronously with the image creation algorithm (using double buffering)**
- Cons: **uses only color data**
  - **Artifacts are unavoidable**
  - **Performance/Quality tradeoffs are impossible**

Announcement: We *finally* managed to set a web site for our group at Intel up and running as **http://visual-computing.intel-research.net**

(it contains this paper, Carsten and Ingo paper, some older publications and also **the source code** for **MLAA**)

No need to write it down – get it from **Ke-Sen Huang** site (CG papers)



Thank You!