Analytic Rasterization on GPGPUs

Thomas Auzinger
Vienna University of Technology

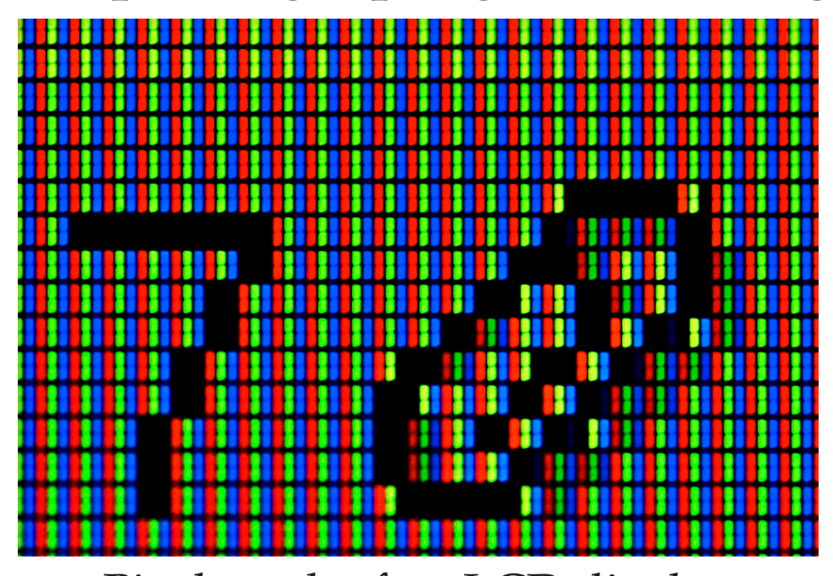
PUMPS 2013 Summer School
July 8-12, Barcelona, Spain

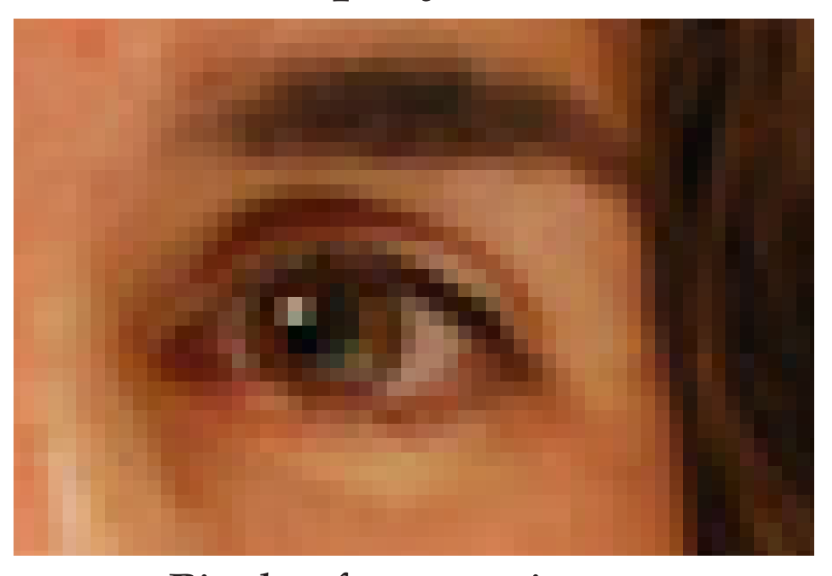
TECHNISCHE UNIVERSITÄT WIEN
Vienna University of Technology

Introduction

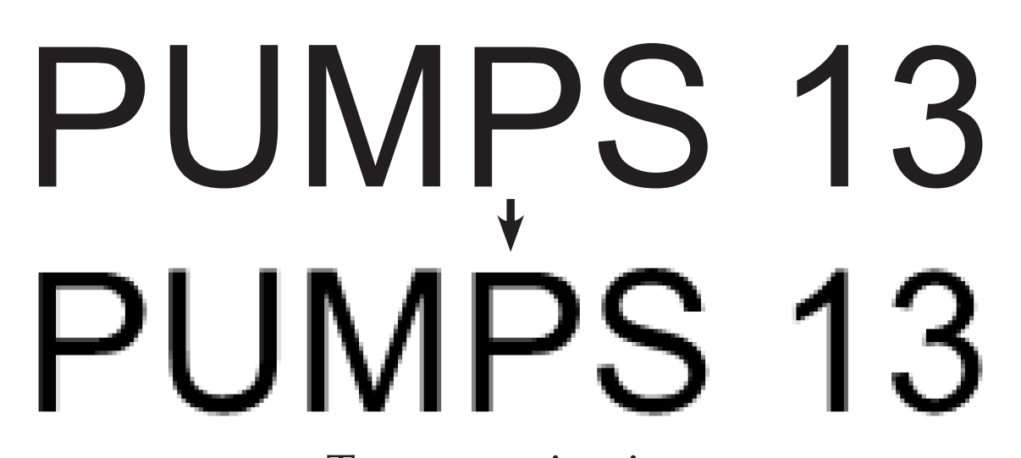
Rasterization

Common display and image formats can be classified in two main groups. **Vector formats** store the exact mathematical description of the content and are used, for example, to store line graphics, text or geometry. Raster formats store color values on regular grids and are used for photograps, general images and for displays.

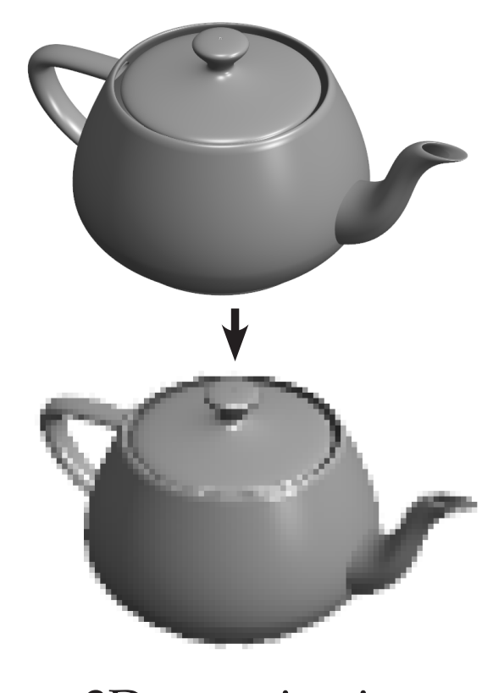

Pixel mask of an LCD display


Pixels of a raster image

The process of converting vector to raster format is called **rasterization** and is a fundamental step that has to be executed before displaying computer graphics.



Text rasterization

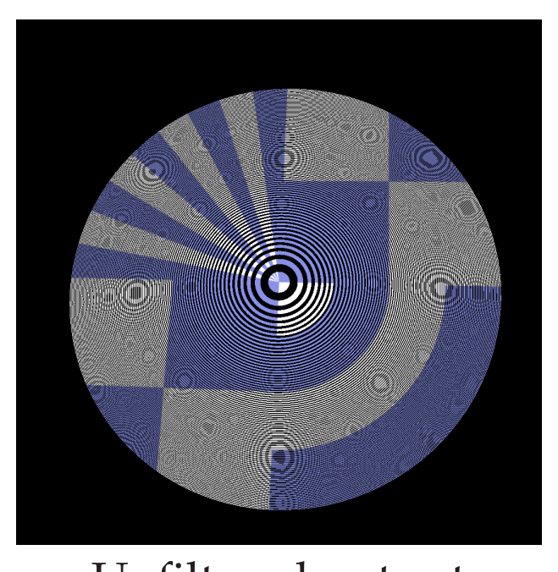

3D rasterization

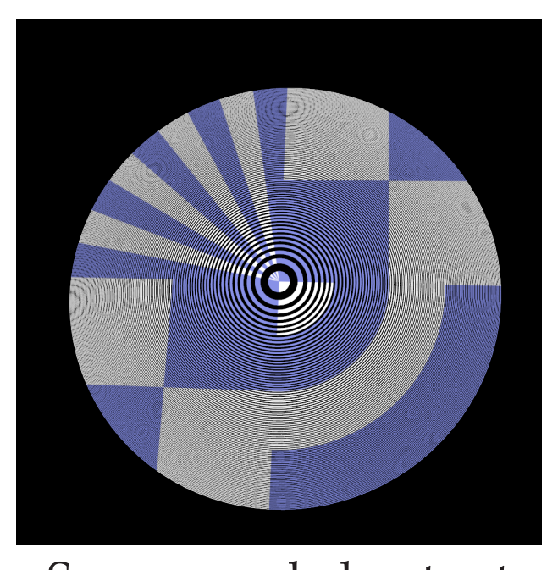
Analytic

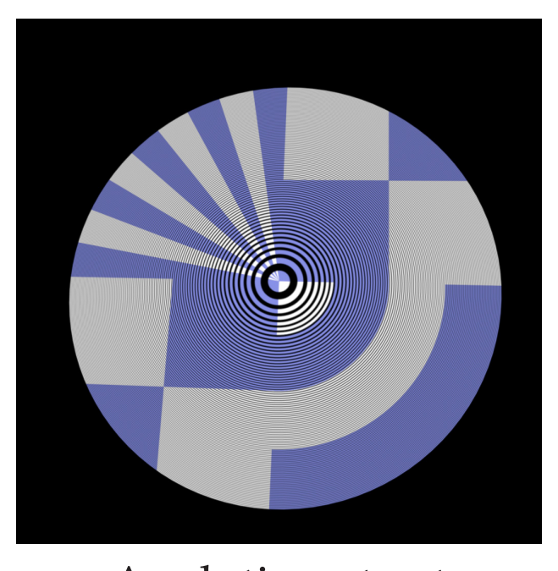
The rasterization process assigns each pixel (or **sample location**) a color. Generally, the raster output is just an approximation of the exact input and small details have to be removed from the data to avoid aliasing artifacts (cf. Section Anti-Aliasing). This removal is usually done via filtering.

In **traditional rasterization**, i.e. in the standard graphics pipeline, filtering achieved by placing more samples and averaging over them. While being very fast, this method has limitations that degrade the final image quality for highly detailed input.

Our method applies **filtering** in a **mathematically exact** way by evaluating the filter convolution integral as a closed-form expression, i.e. as a formula. This aleviates all supersampling issues and provides a near-perfect output image.


Unfiltered output


Supersampled output

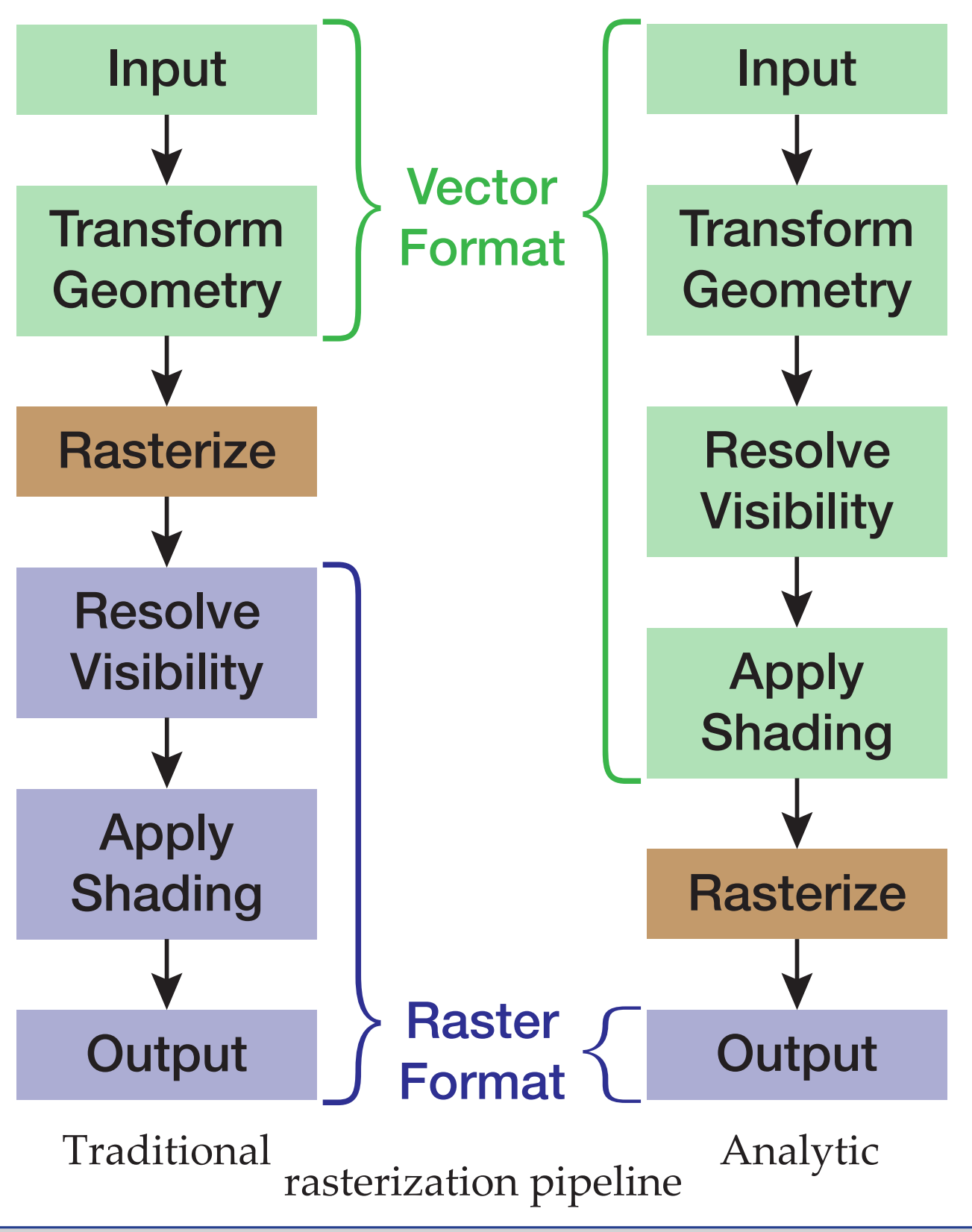

Analytic output

GPGPU

Similar to traditional rasterization, our analytic method is highly parallel and can be implemented efficiently on massive **SIMD architectures**, e.g. on GPUs. However, our analytic pipeline differs significantly from the traditional graphics pipeline and does not use the hardware rasterizer.

Exact filtering is enabled by keeping the input data in vector format as long as possible. For example, visibility and shading computations are executed in this format. Only the very last step is the actual rasterization.

The standard graphics pipeline design, in contrast, performs most computations in raster format. Thus, we cannot use a shader based programming model (OpenGL, DirectX,...) but rely on the GPGPU capabilities of modern GPUs. For our implementation we use **NVidia CUDA C**.

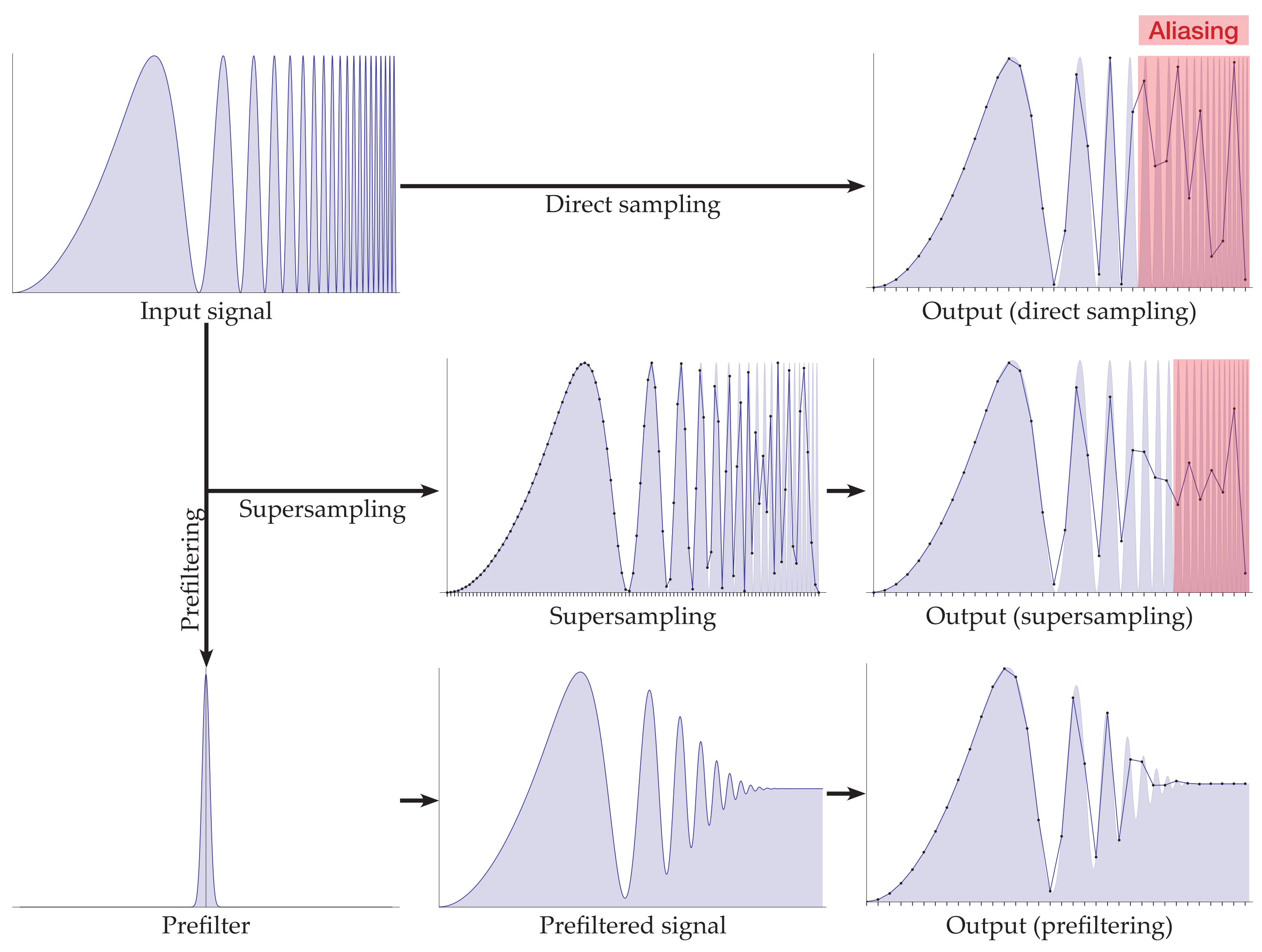


Anti-Aliasing

Rasterization of vector input to a rasterized output is equivalent to the **sampling** of a continuous input signal at discrete sample locations.

According to the **Nyquist-Shannon sampling theorem**, the resolution of the sample locations limits the finest details that are preserved by the sampling process.
Finer details lead to **aliasing artifacts** in the sampled output (see direct sampling figure). Two main methods exist to combat aliasing, i.e. to perform anti-aliasing:

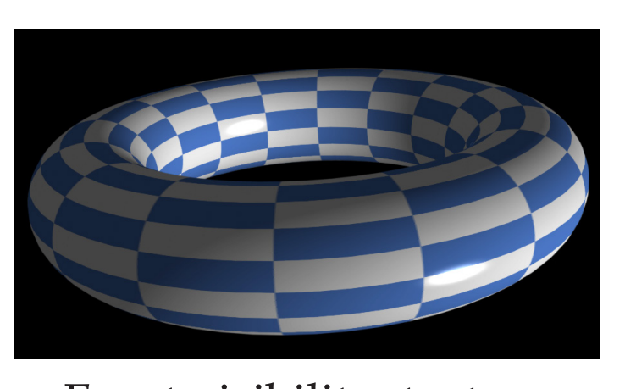
- **Supersampling**
  Sample the input data more densly and then reduce the number of samples by averaging the supersamples with a suitable filter. Since the filtering happens after sampling, this is also called postfiltering.
  Note that this method still suffers from aliasing.
- **Prefiltering**
  Filter the data before the sampling with a suitable prefilter. This is the mathematically correct way to address aliasing issues but computationally more expensive.
  This method is used in our analytic rasterization.



Results

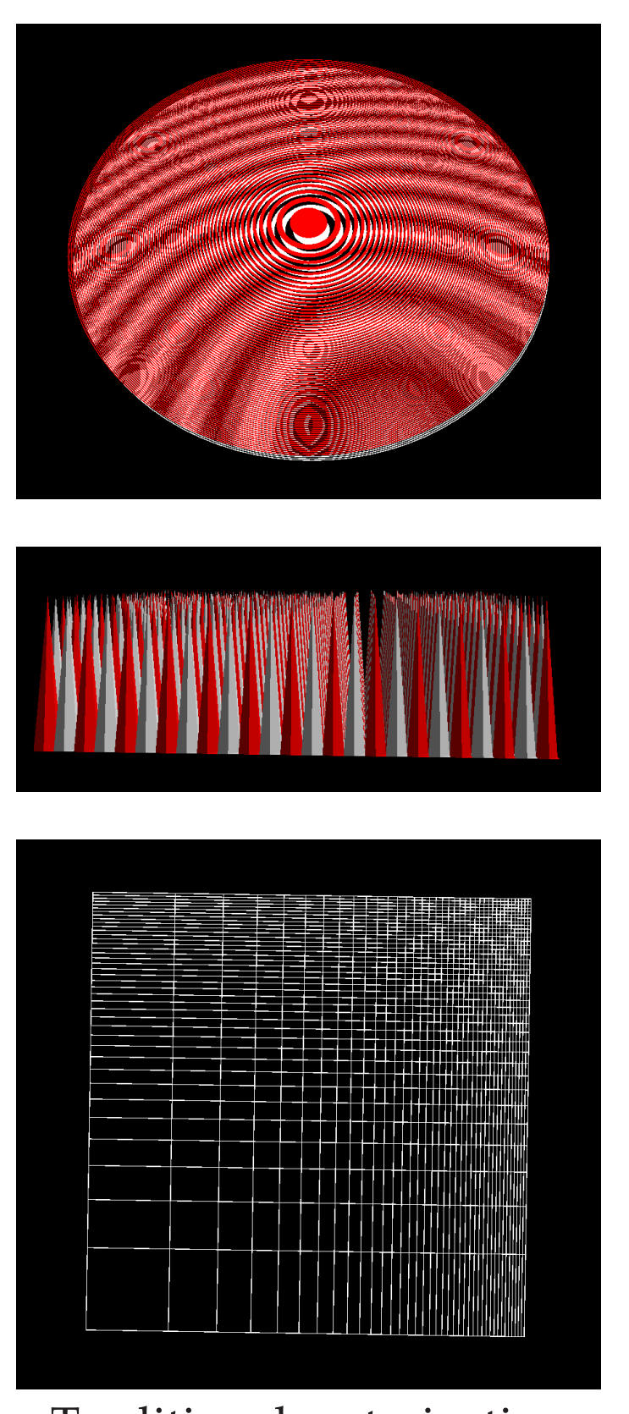
With our method we present the following contributions:

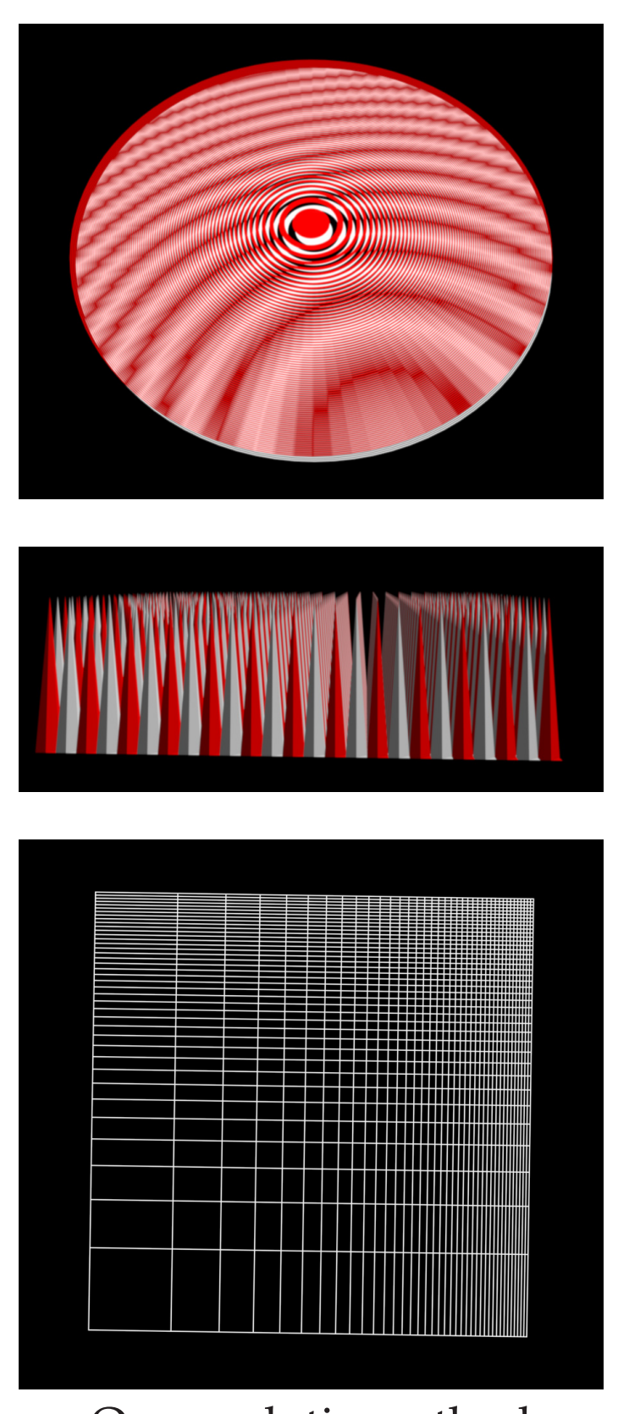
Exact visibility computation
- Full hidden surface elimination
- Handles geometry of sub-pixel size


Exact visibility, texture mapping and non-linear shading

Exact prefiltering
- Closed-form solution for prefiltering
- Artifact-free near-perfect anti-alising
- Exact shading for simple shading models
- Exact visibility weighting for complex models

Parallel hardware adaptation
- Formulated as highly parallel algorithm
- Efficient implementation on SIMD hardware
- Interactive performance


Traditional rasterization

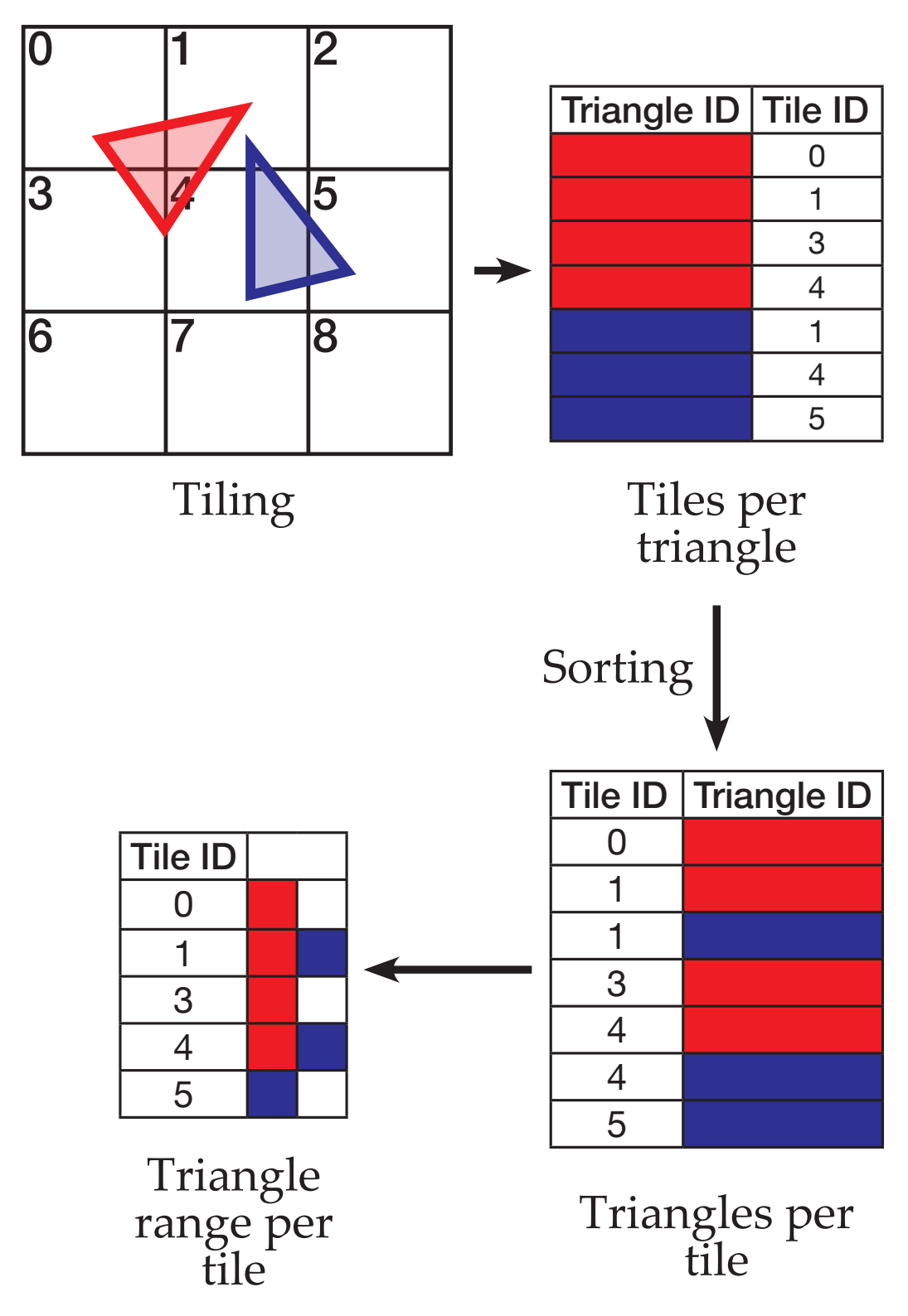

Our analytic method

Our *CUDA* Pipeline

Load Balancing

Our analytic rasterization pipeline computes the contribution of each input triangle to each sample location of the output raster image. Most contributions are zero as triangles only influence sample locations close to them.
To avoid computing all triangle-sample pairs we tile the output image and assign triangles to their tiles. This enables localized per-tile computations.

Load balancing steps:
- Subdivide output region into regular tiles.
- Compute overlapped tiles IDs for each triangle ID in parallel.
- Sort triangle IDs according to tile IDs (radix sort using thrust).
- Compute triangle ID ranges for each tile ID.



All following computations are executed per-tile. The corresponding input data is accessed via the triangle ID ranges.
This ensures that each triangle only computes its non-zero contributions to close pixels.

Analytic Visibility

When viewing a scene from a given camera location, objects in the front likely occlude other objects behing them. In the visibility stage the visibility of all scene triangles is resolved by removing their hidden parts.
This is computed in parallel for the edges of all triangles. The triangle count of a normal scene (10^5-10^6) is enough to sature the threads of a modern GPU.

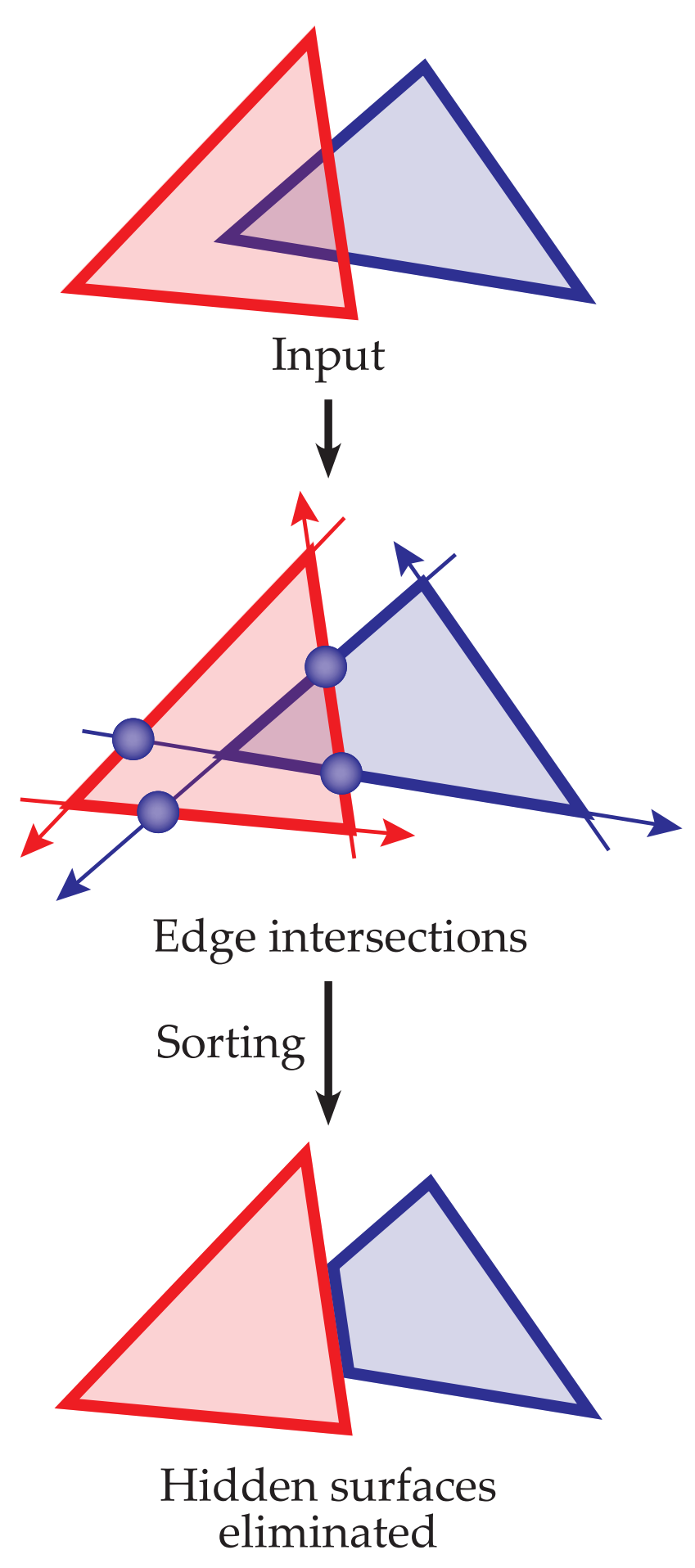
Intersection computation
The visibility of a triangle can only change at the edges of another triangle. For a given edge of a scene triangle we compute all intersections with the edges of other triangles along the line that contains this edge.

```
Input:  E (set of all edges in the scene)
Output: I (set of all intersections)

for each edge e of E in parallel
  T ← triangles of local tile
  for each triangle t of T in parallel
    o    ← get global memory offset
           // with atomics
    I[o] ← add intersections of e and t
  end for
end for
```
Pseudo-code of the intersection computation

Intersection sorting
The threads write the intersections unordered into global memory, thus we perform a sorting step to order the intersection along each edge.

Hidden surface elimination
We walk along each edge and use the sorted intersections to count the number of triangles that occlude each intersection. A scan is used for this task. All visible edge segments, which are the output of this stage, are then stored in global memory.



Analytic Shading

A shading function is defined on each triangle depending on the assigned material and the local illumination. Our goal is to prefilter and then sample this function on the visible regions of all scene triangles. The output of the analytic visibility stage is used for this purpose.

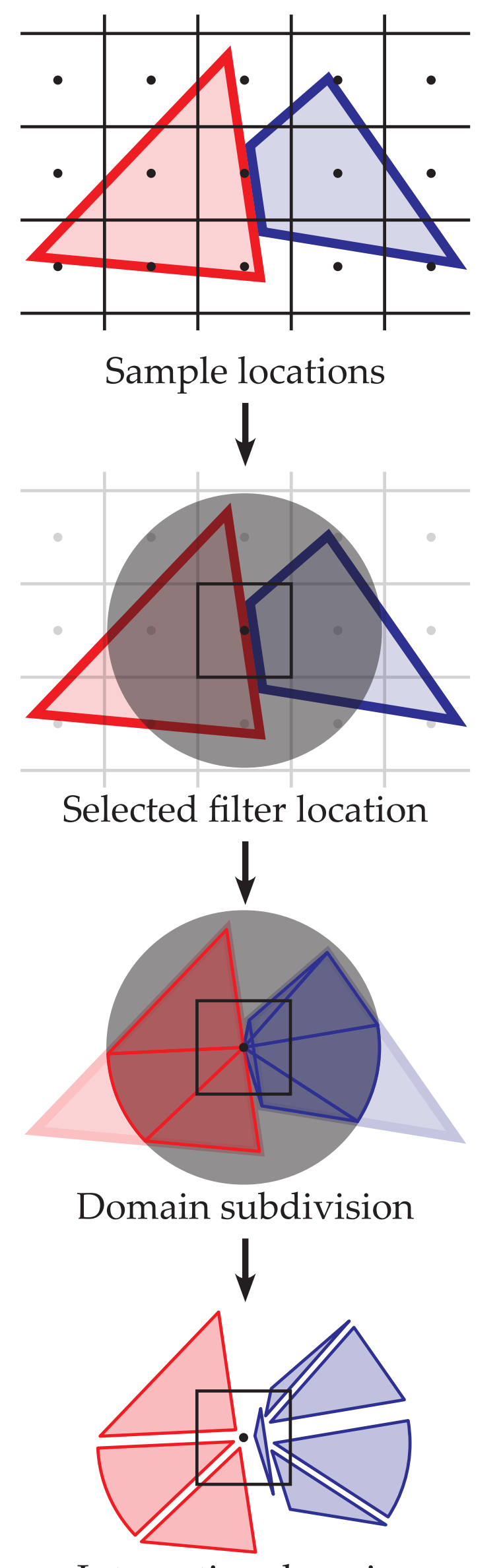
To perform exact filtering of the shading function, it has to be **symbolically integrable** in order to obtain a closed-form solution. If this is not the case, the visible area of each triangle is prefiltered to obtain the exact weights for sampled shading (this can be seen as a ground truth solution for multisampling).

Integral computation
For the given sample locations (usually the pixel centers of the output image) the contribution of all triangles to them is computed. At each sample location a radial prefilter is placed and convolved with the shading function of the visible region of all triangles.

The visible region of each triangle is given by its boundary, i.e. the line segment output of the analytic visibility stage.
Each line segment and the sample location spans an integration domain that is clipped against the support of the filter over which is integrated.

```
Input:  B (set of boundary segments of the
           visible regions of all triangles)
        S (set of sample locations)
Output: C (set of pixel colors)

for each sample location s of S in parallel
  L ← boundary segments of B of local tile
  for each segment l of L in parallel
    c    ← compute prefilter convolution of l
    C[s] ← add color c
  end for
end for
```
Pseudo-code of the integration computation



Contact

Name: Thomas Auzinger
Institution: Institute of Computer Graphics and Algorithms
Vienna University of Technology, Austria
Email: thomas.auzinger@cg.tuwien.ac.at

References: T. Auzinger, M. Wimmer, S. Jeschke, *Analytic Visibility on the GPU*, Computer Graphics Forum, 32(2):409-418, 2013
T. Auzinger, M. Guthe, S. Jeschke, *Analytic Anti-Aliasing of Linear Functions on Polytopes*, Computer Graphics Forum, 31(2):335-344, 2012