Postfiltering anti-aliasing

- Alternative to supersampling anti-aliasing.
- Idea:
  Signal processing uses low-pass filtering for smoothing, which corresponds to local averaging.
- Approach:
  Scan convert many projected scenes, which are subpixel transformations of the original scene, and average the resulting images.
Postfiltering anti-aliasing

Scan conversion after subpixel transformation:

\[
(5 \times \text{original image}) + 1 \times \text{shifted vertically} + 1 \times \text{shifted horizontally} \div 7 = \text{post-filtered anti-aliased image}
\]
Supersampling vs. post-filtering

- Post-filtering is applied to the entire scene equally, while supersampling can be applied locally to certain fragments.
  For example, supersampling can be restricted to only the silhouettes of an object to not blur surface parts.
- Post-filtering can deal with any geometry, i.e., explicit or implicit object representation, while supersampling is typically targeted at polygonal mesh representation.
Anti-aliasing in OpenGL

- Removing the Jaggies
  - `glEnable( mode )`
    - `GL_POINT_SMOOTH`
    - `GL_LINE_SMOOTH`
    - `GL_POLYGON_SMOOTH`
  - coverage computed by computing supersampling.
  - coverage stored in alpha channel of RGBA representation.
  - blending needs to be enabled.
Blending

- Combine pixels with what’s already in the framebuffer
  - `glBlendFunc(src, dst)`

\[
\tilde{C}_r = src \tilde{C}_f + dst \tilde{C}_p
\]
Multi-pass rendering

- Blending allows results from multiple drawing passes to be combined together
  - enables more complex rendering algorithms

Example of bump-mapping done with a multi-pass OpenGL algorithm
Full scene antialiasing

- Implementation of post-filtering anti-aliasing
- Each time we move the viewer, the image shifts
  - Different aliasing artifacts in each image
  - Averaging images using accumulation buffer averages out these artifacts
Accumulation Buffer

- Problems of compositing into color buffers
  - limited color resolution
    - clamping
    - loss of accuracy
  - Accumulation buffer acts as a “floating point” color buffer
    - accumulate into accumulation buffer
    - transfer results to frame buffer
Accumulation Buffer

- `glAccum( op, value )`
  - operations
    - within the accumulation buffer: `GL_ADD`, `GL_MULT`
    - from read buffer: `GL_ACCUM`, `GL_LOAD`
    - transfer back to write buffer: `GL_RETURN`
  - `glAccum(GL_ACCUM, 0.5)` multiplies each value in read buffer (specified by `glReadBuffer()`) by 0.5 and adds to accumulation buffer
3.5 Visibility
Visibility

- When multiple objects are placed in the scene, one has to decide, which (parts of which) objects are visible to the viewer.
Hidden line removal

- Brute-force geometric approach for scenes with planar polygons.
- Targeted at wireframe renderings.
Hidden line removal

for all edges \( e \) in the scene
for all polygons \( P \) in the scene
   check which parts of \( e \) are occluded by \( P \)
   store all occluded parts
output the complement of the stored occluded parts
Spatial partitioning

- Performance of geometric algorithms can substantially benefit from the use of spatial partitioning strategies.
- Spatial partitioning divides the considered space into non-overlapping subspaces, whose union covers the entire space again.
- The processed geometry is assigned to the subspaces, which allows for local processing of the geometry within the subspaces.
- Different spatial partitioning strategies exist.
- They go along with a data structure to store the assigned geometry.
Cell raster

- Partition the projected scene in a fixed amount of equally-sized rectangular cells.
- Store in each cell all the polygons that intersect the cell.
- Execute visibility test for each cell individually.
Cell raster

- Cell raster with 4x4 cells:

  only 1 visibility check

- Easy to implement.
- But: Cell size independent of scene.
Quadtrees

- Start with a cell that uses available screen space.
- Iteratively partition the current cell into 4 equally-sized subcells.
- In each iteration, assign the polygons of the current cells to the respective subcells.
- Stop when the number of polygons per cell is below a threshold.
- The generated structure can be stored in a quadtree, i.e., a tree where each inner node has 4 children.
- Execute visibility test for each leaf-node cell individually.
Quadtree

- Example: Quadtree with 3 partitioning levels, where each leaf cell has no more than 2 assigned polygons:

- Still simple.
- But: Still many empty cells.
**kD-tree**

- Start with cell that uses available screen space.
- Iteratively partition the current cell into two subcells by an optimal cut orthogonal to a coordinate axes.
- The partitioning alternates in both dimensions.
- The optimal cut is found by splitting the amount of geometry in two halves.
- Stop when amount of geometry per cell is below threshold.
- The resulting spatial partitioning is stored in a binary tree, called kD-tree (kD: k-dimensional; here k=2).
- Execute visibility test for each leaf-node cell individually.
**kD-tree**

- Example: kD-tree with 3 partitioning levels, where each leaf cell has exactly 1 assigned polygon:

![Diagram of a kD-tree with 3 levels and leaf cells representing assigned polygons.]

- No empty cells.
- Balanced tree.
- (But: Storing of cuts is required.)
- But: axis-aligned cuts do not provide highest flexibility.
Binary partition

- Start with cell that uses available screen space.
- Iteratively partition the current cell into two subcells by an optimal cut.
- The optimal cut is found by splitting the amount of geometry into two halves using any line.
- Stop when amount of geometry per cell is below threshold.
- The resulting spatial partitioning is, again, stored in a binary tree.
- Execute visibility test for each leaf-node cell individually.
Binary partition

- Binary partition with 2 partitioning levels, where each leaf cell has exactly 1 assigned polygon:

- Triangles cannot be separated with axis-aligned cuts.
- Very flexible partitioning.
- But: cutting line needs to be stored.
- But: traversing the data structure gets complicated.
Scan line algorithm

- Goal: Compute visible parts of filled polygons.
- Idea: Apply scan line algorithm.
- Approach: Take scan line algorithm for polygon filling.

Where polygons overlap (between intersections 1 and 2), decide locally (i.e., for each pixel), which polygon is visible.
Painters algorithm

- Sort polygons back to front.
- Draw projected polygons in that order.
Painters algorithm

- Complexity: $\Theta(n \log n)$ with $n = \#$ polygons
- Problem: Intersecting polygons

$\Rightarrow$ detect intersections and split one of the polygons (expensive!)
Depth buffer algorithm

- Observation: Visibility test is only required at a finite set of discrete points, namely at the pixels of the framebuffer.
- Idea: Operate on discrete positions only and store intermediate results in discrete form in the framebuffer and in an additional buffer of same size (depth buffer or z-buffer).
- Framebuffer stores intermediate output in form of color values of projected visible polygons.
- Depth buffer stores intermediate depth values of the visible polygons, i.e. distance of polygons to the viewer.
Depth buffer algorithm

- Initialize depth buffer with range of sight.
- For all polygons in the scene
  - For all pixels p that are covered by the projected polygon
    - Compute distance d of the polygon from the viewer through pixel p
    - Look up depth z stored in depth buffer at position of pixel p
    - If (d < z)
      - Write color of polygon into framebuffer at pixel p
      - Replace depth z with depth d in the depth buffer
- Render framebuffer
Depth buffer

A simple three dimensional scene

Z-buffer representation