

CS 428: Fall 2009

Introduction to Computer Graphics

Polygon rendering: additional topics

z-buffer algorithm

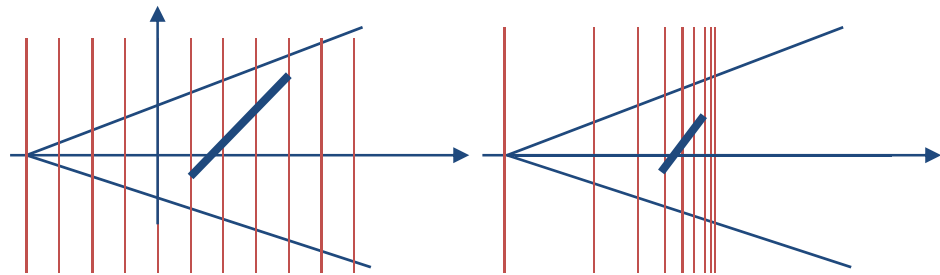
- The depth buffer was suggested in 1974 but not implemented (too expensive)
- For each pixel store a z-value (depth image)
- Initialization
 - Frame buffer = clear color
 - z-buffer » maximal z-value
- Raster all scene objects sequentially
 - The order is not essential (for opaque objects)

z-buffer algorithm

- For each point (x,y) of each polygon

- Compute $z(x,y)$

- Perspective transformation



- If $z(x,y)$ is smaller than the stored value at (x,y)
 - Write $z(x,y)$ into z-buffer, and write the associated color value at (x,y) into the frame buffer
- After this terminates, only visible parts of the surface(s) are visible in the frame buffer

z-buffer algorithm

Advantages

- Any scene with any object representation can be handled (entirely image-based)
- Complexity is independent of depth complexity
- Objects can be added into a rendered scene
 - Interesting when adding objects to camera shots
- Simple to implement in hardware

z-buffer algorithm

Drawbacks

- Only one object stored per image pixel
 - Resulting sampling errors can be reduced by supersampling [higher image resolution], but not entirely removed
- Transparency is not possible with an active depth test
- The precision of the z-buffer is limited
 - Separate objects have the same z-value
 - The pixel color is then entirely determined by the rendering order (and **glDepthFunc (...)**)

z-buffer algorithm

OpenGL details

- Active when **GL_DEPTH_TEST** is enabled
 - Initially, depth testing is disabled
- **glDepthFunc** (GLenum *func*) determines the nature of the depth test
 - The initial value of *func* is **GL_LESS**
 - Also available **GL_NEVER**, **GL_EQUAL**, **GL_LEQUAL**, **GL_GREATER**, **GL_NOTEQUAL**, **GL_GEQUAL**, and **GL_ALWAYS**

OpenGL polygon rendering modes

- Determined by `glPolygonMode` (*face*, *mode*)
 - *face*
`GL_FRONT`, `GL_BACK`,
`GL_FRONT_AND_BACK`
 - *mode*
`GL_POINT`, `GL_LINE`, `GL_FILL`

Preventing *z-fighting*

- Use `glPolygonOffset` (*factor*, *units*)
 - Adds offset = $(\Delta z \cdot \textit{factor} + r \cdot \textit{units})$ to the **depth buffer** value **before** the depth test
 - $\Delta z = \Delta \text{depth} / \text{area}$ (per primitive/polygon)
 - $r = \text{z-buffer precision}$ (hardware dependent)
 - Use *factor* in project 2
 - `glEnable(GL_POLYGON_OFFSET_FILL)`

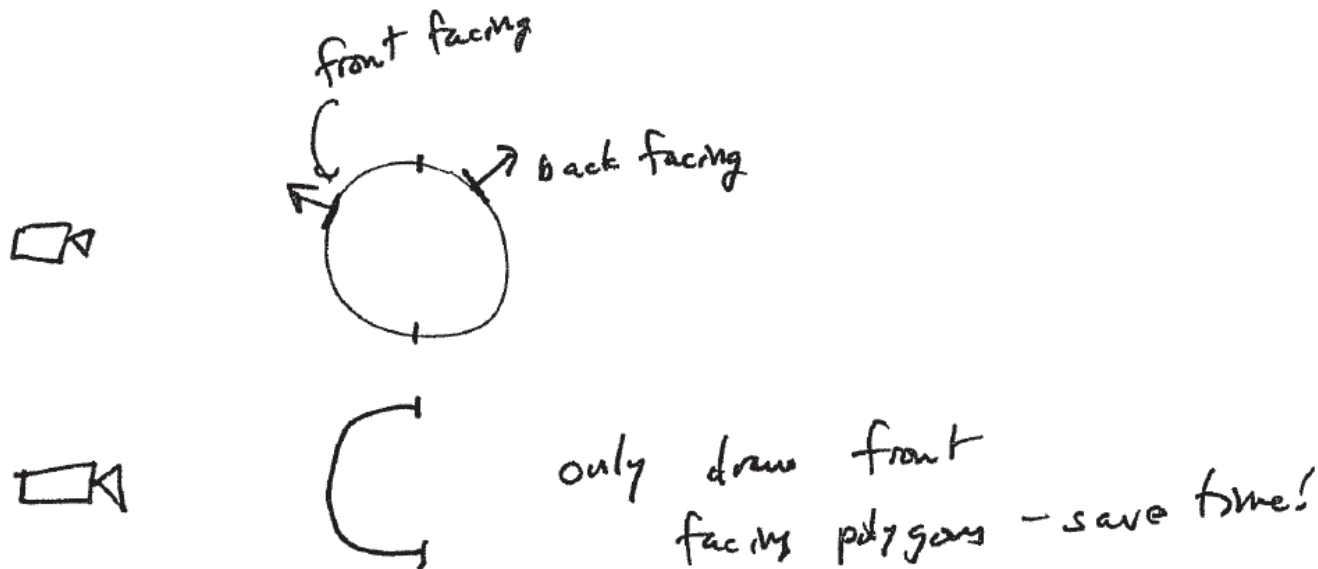
Color and depth masks

- **glColorMask** (*r, g, b, a*)
 - *r, g, b, a* are GLboolean values (**true** by default)
 - Selectively enable/disable writing to the frame buffer during rendering

- **glDepthMask** (*d*)
 - *d* is a GLboolean value (**true** by default)
 - enable/disable writing to the z-buffer during rendering

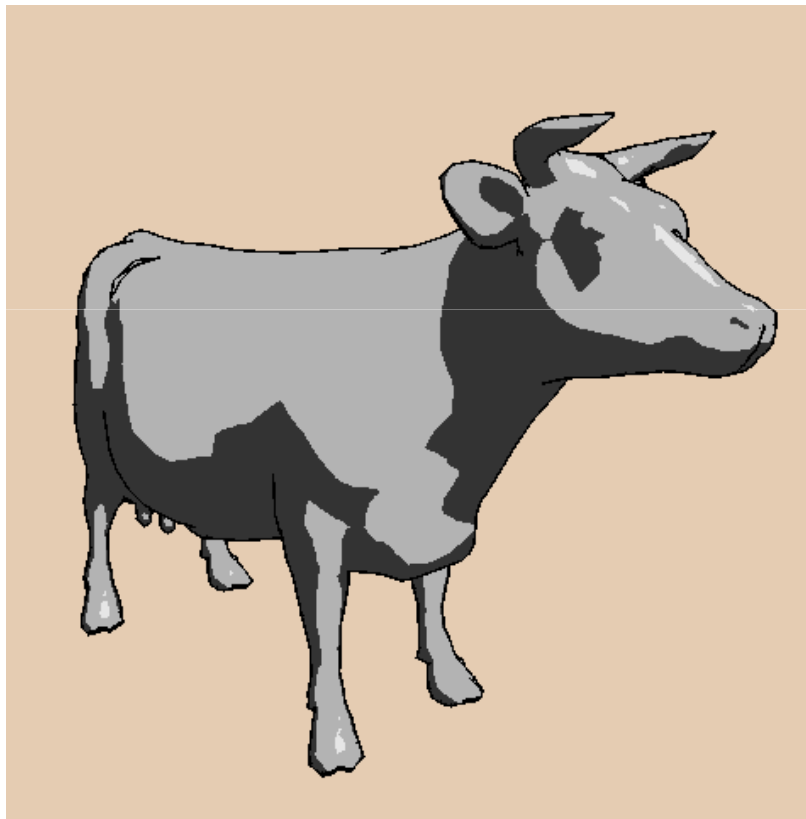
Face culling

- Given consistent polygon orientation (CCW)



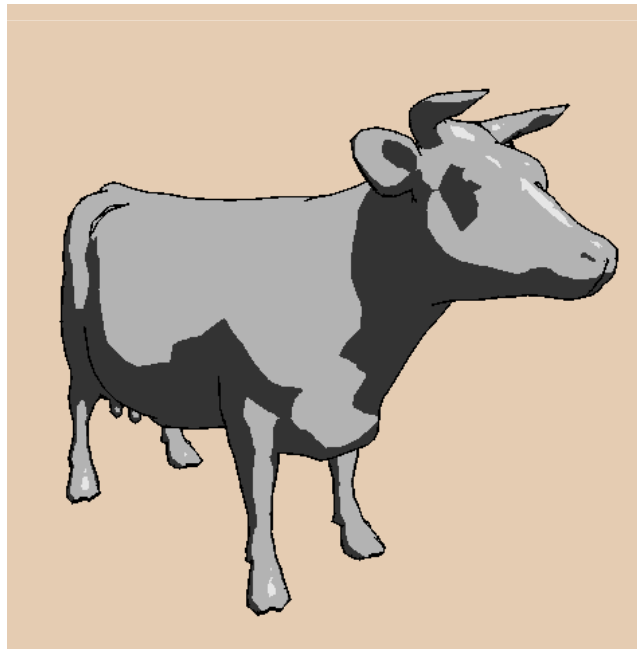
- `glCullFace ([GL_FRONT | GL_BACK])`
- Only when `GL_CULL_FACE` is enabled

Non-photorealistic rendering (NPR)

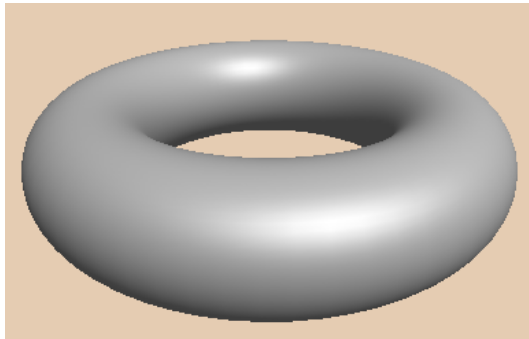


Non-photorealistic rendering (NPR)

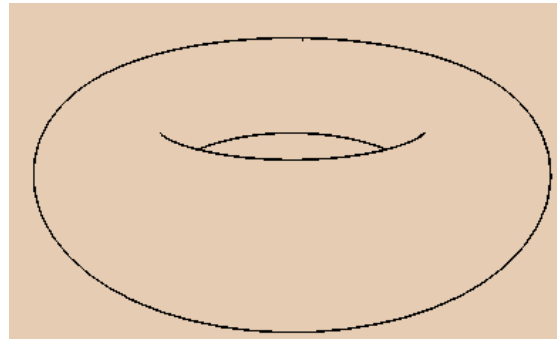
- Lines
 - Silhouettes, creases
- Shading
 - Toon shading
 - Hatching



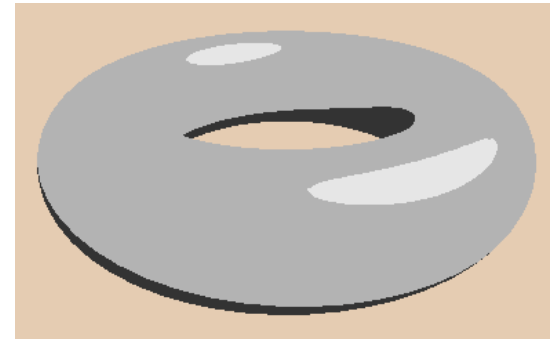
Silhouettes and Toon shading



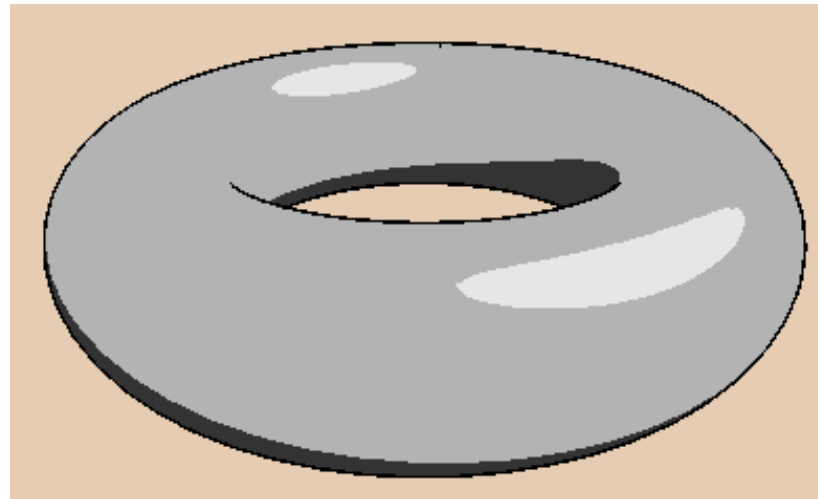
Normal shading



Silhouettes



Toon shading



Silhouettes + toon shading

Silhouettes (a.k.a. contours)

- Mark changes in visibility

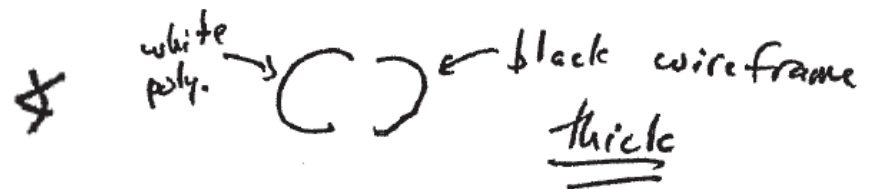


- Separate front and back facing polygons



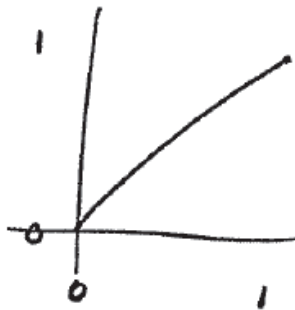
- Direct: compute sil edges + render visible ones

- Indirect: render scene so that sils are visible

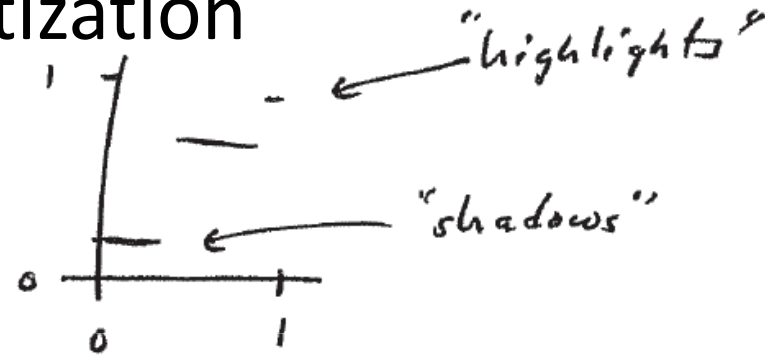
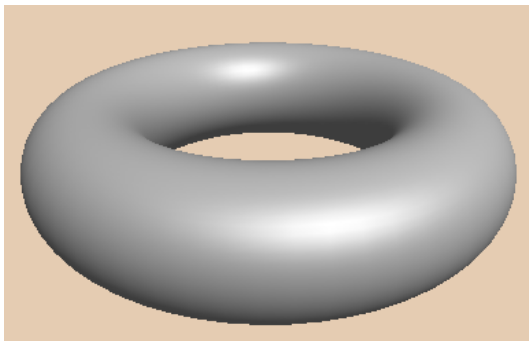


Toon shading

- A form of color quantization

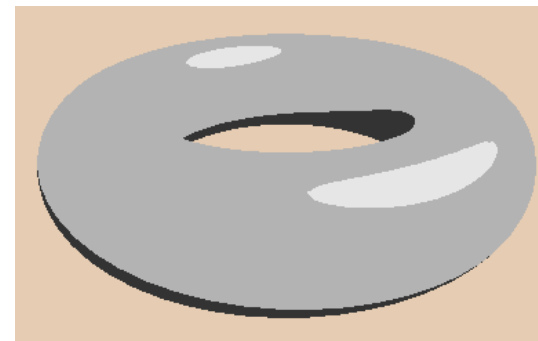


normal shading



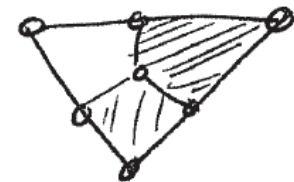
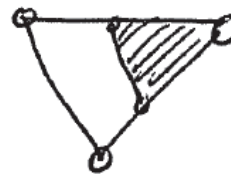
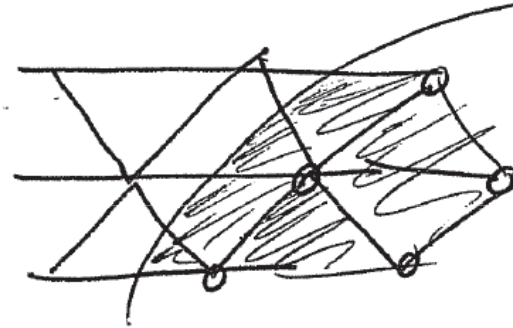
toon shading

maps $[0,1) \rightarrow [0,1]$



Toon shading in OpenGL

- Turn off OpenGL lighting and use `glColor` directly
- Not necessary when using GLSL
 - Instead, compute local lighting and mapping per pixel



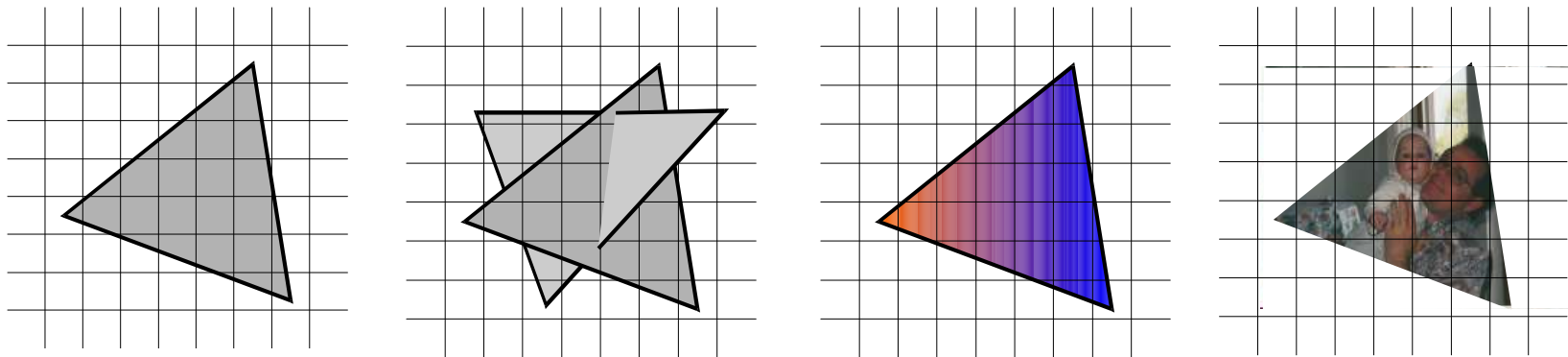
subdivide

triangles w/ 2 colors + 3 colors

Excursion: rasterization

- Rasterization

- Primitives (lines, polygons) are mapped to pixels



- Additional operations per pixel

- Visibility (including transparency)
 - Shading and
 - Texturing

Rasterization of lines

Differential Digital Analyzer

- Lines, where start and endpoints lie on the grid

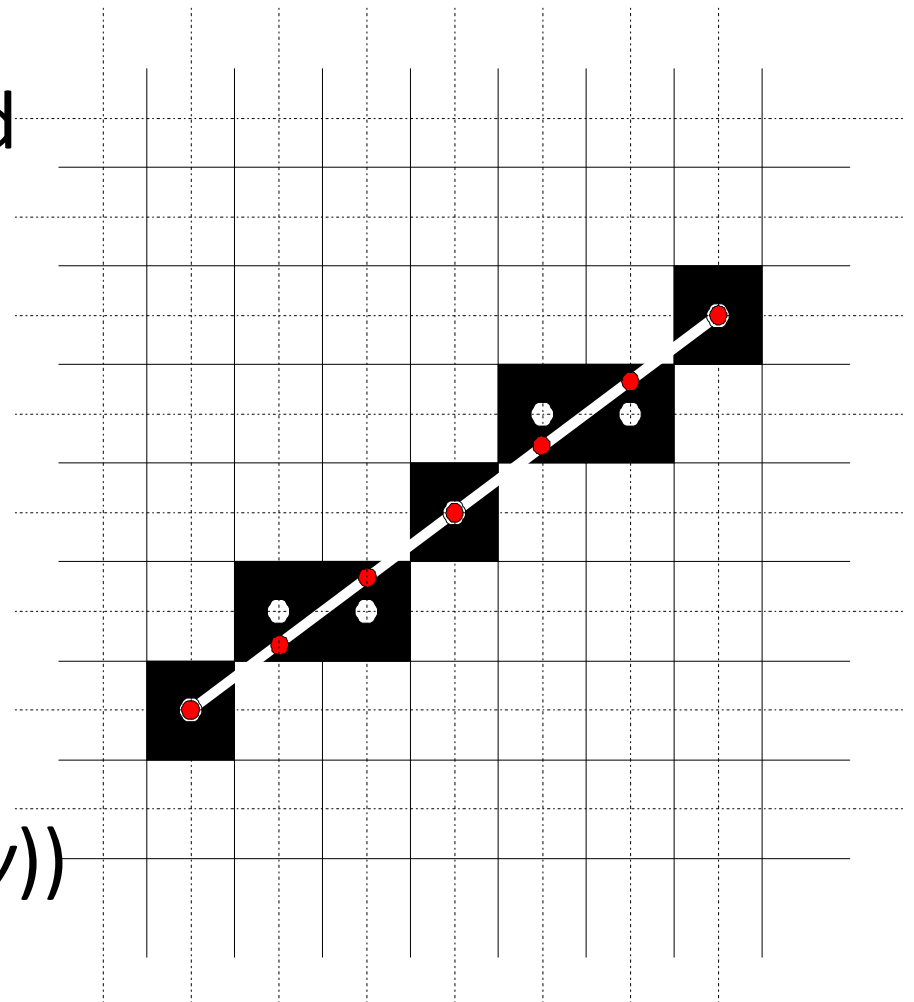
$$\Delta x = x_2 - x_1, \quad -1 \leq \frac{\Delta x}{\Delta y} \leq 1$$
$$\Delta y = y_2 - y_1,$$

- Compute

$$y_i = \frac{\Delta y}{\Delta x} * x_i + b,$$

$$x_i = x_1 + i, \quad i = 1, \dots, \Delta x$$

- Draw pixel at $(x, \text{round}(y))$



Rasterization of lines

Differential Digital Analyzer

- Not efficient: every pixel operation requires
 - fp multiplication + Addition + Rounding
- Idea: incremental Algorithm

$$\begin{aligned}y_{i+1} &= \frac{\Delta y}{\Delta x} * x_{i+1} + b & \Delta x &= x_2 - x_1; \Delta y = y_2 - y_1; \\ &= \frac{\Delta y}{\Delta x} (x_i + (x_{i+1} - x_i)) + b & m &= \frac{\Delta y}{\Delta x} \\ &= y_i + \frac{\Delta y}{\Delta x} (x_{i+1} - x_i) & x &= x_1; y = y_1; \\ & & \text{for } (x = x_1, x \leq x_2, x++) \{ & \\ & & \quad \text{DrawPixel } (x, y); & \\ & & \quad y = \text{round } (y+m); \} & \end{aligned}$$

- For $x_{i+1} - x_i = 1$ we have $y_{i+1} = y_i + \frac{\Delta y}{\Delta x}$

Rasterization of lines

Bresenham's algorithm

- First integer-algorithm for line drawing

- Bresenham (1965)

- Derivation

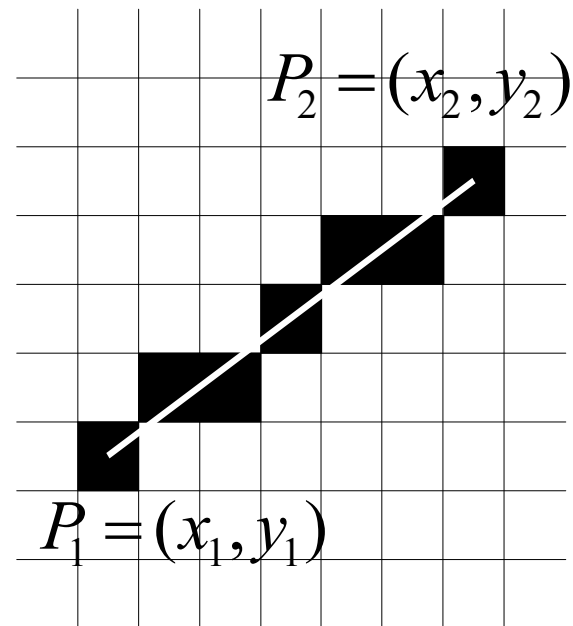
- Begin and endpoint on the grid

- Slope between 0 and 1

$$\Delta x = x_2 - x_1 \geq 0,$$

$$\Delta y = y_2 - y_1 \geq 0,$$

$$\Delta x \geq \Delta y.$$



Rasterization of lines

Bresenham's algorithm

- Which pixel center is closer to the line?

Is $d \leq 1/2$ or is $d > 1/2$?

- Decision variable $E := \frac{\Delta y}{\Delta x} - \frac{1}{2}$

$$E' := 2\Delta x E = 2\Delta y - \Delta x$$

$$E \leq 0:$$

$$E > 0:$$

$$x := x + 1$$

$$x := x + 1$$

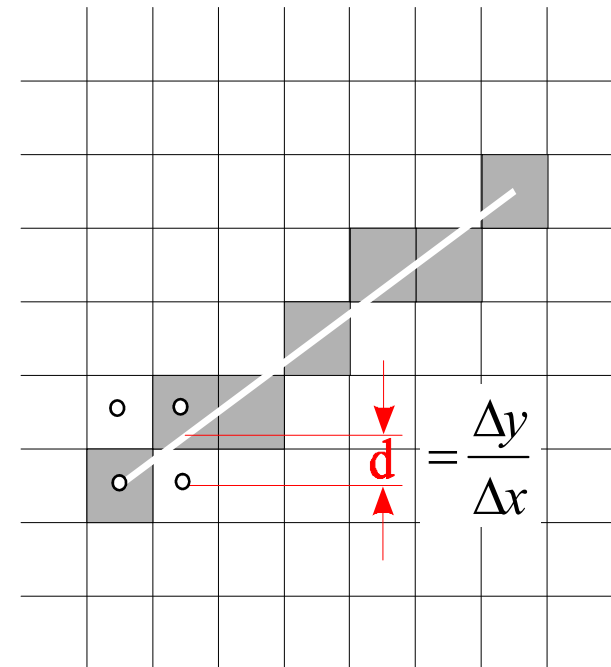
$$y := y + 1$$

$$E := E + \frac{\Delta y}{\Delta x}$$

$$E := E + \frac{\Delta y}{\Delta x} - 1$$

$$E' := E' + 2\Delta y$$

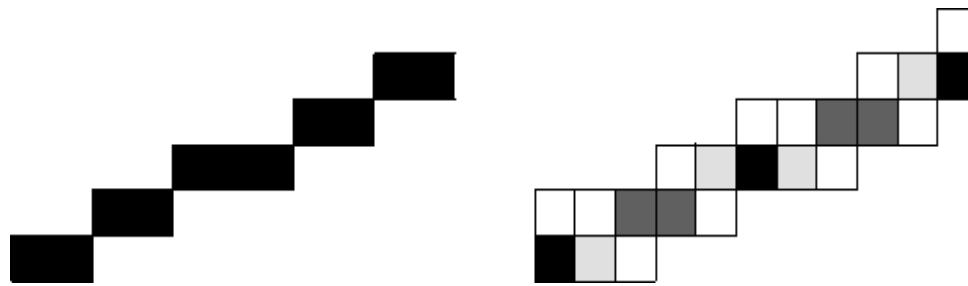
$$E' := E' + 2\Delta y - 2\Delta x$$



Rasterization of lines

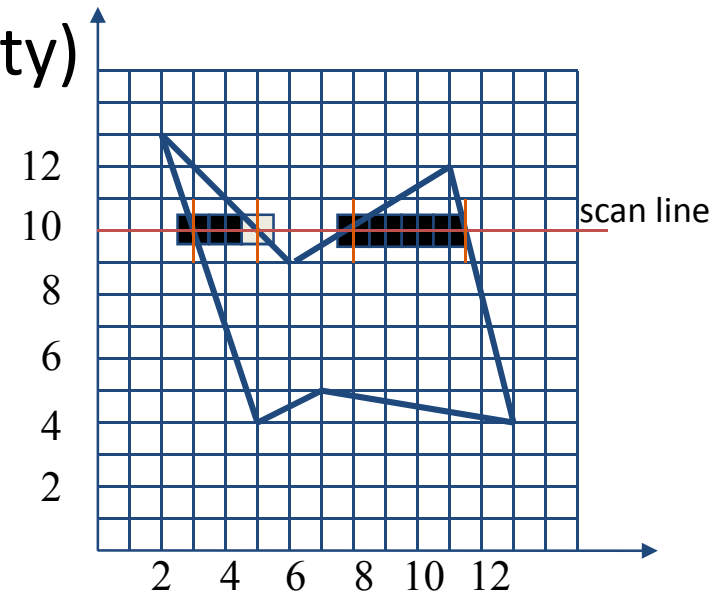
Smoothing

- For each x-value (columns) two pixels are colored
- Brightness in each column is equal
- Distribution proportional to the distance of each pixel to the ideal position
- Brightness decreases linearly with distance



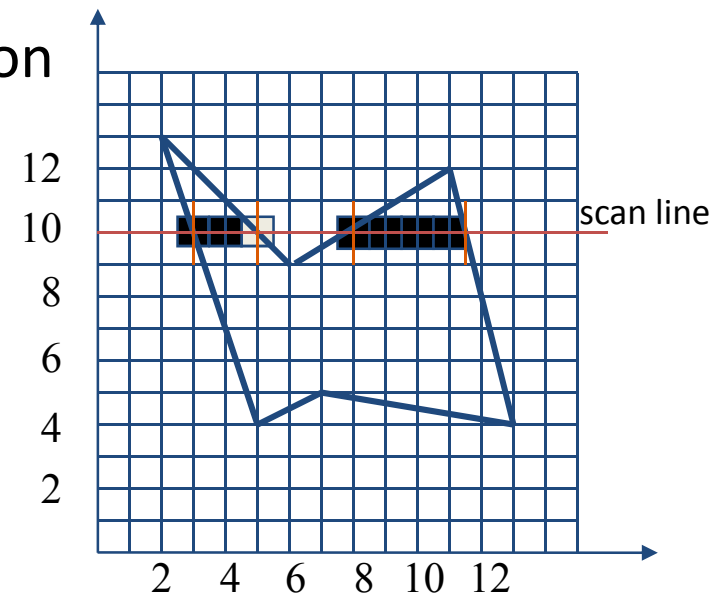
Rasterization of polygons

- Scanline algorithm
 - Intersect scan line with all edges of the polygon
 - Sort intersections by x-coordinate
 - Fill pixels between pairs of subsequent intersections (Rule of odd parity)
 - Parity is initially 0
 - Every intersection increases parity by 1
 - Draw pixel when parity is odd



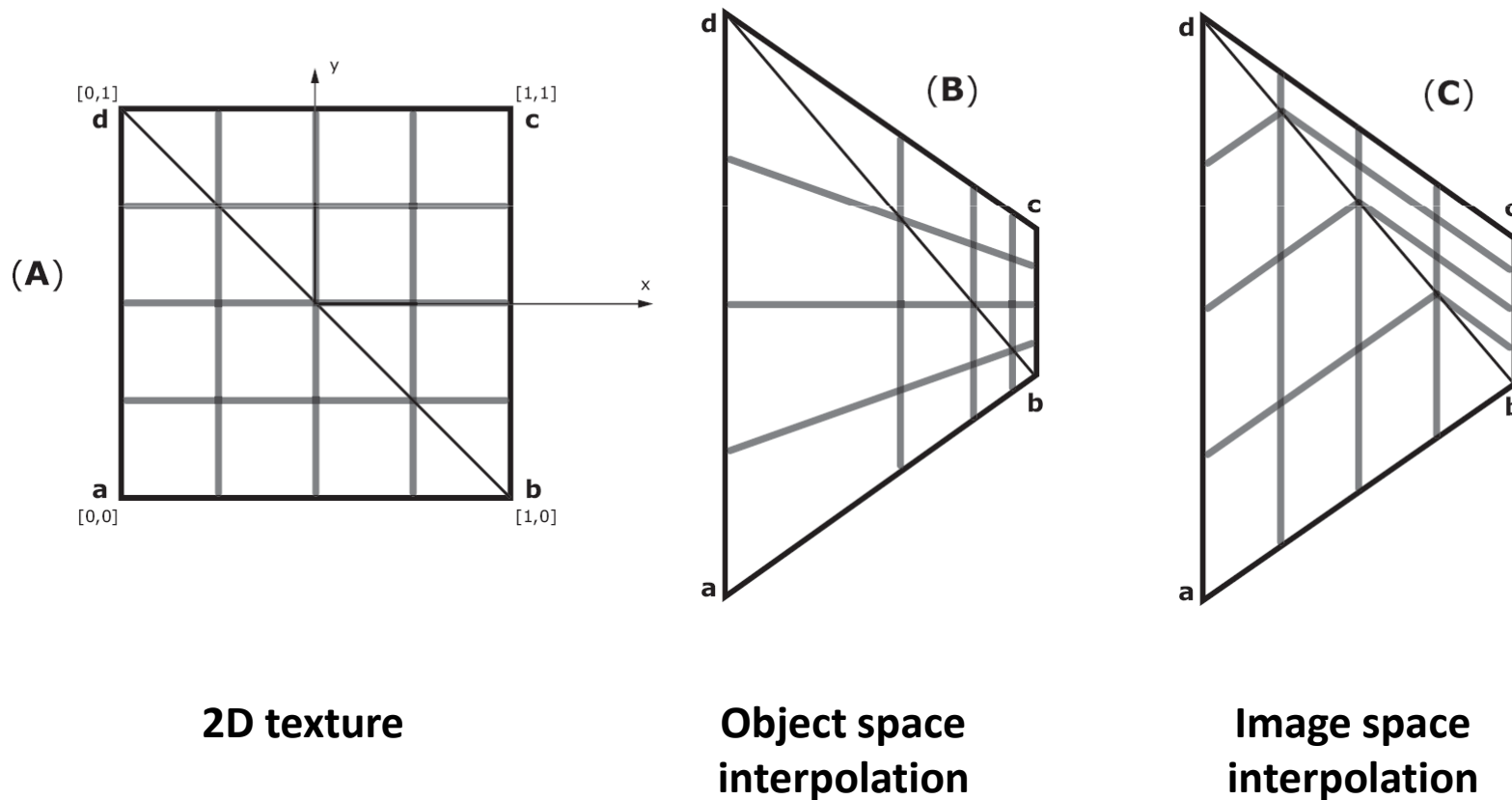
Rasterization of polygons

- Quantities from vertices are interpolated to the pixels
 - Colors (linear interpolation in screen space)
 - Texture coordinates (non-linear interpolation!)
 - Texture look-up after rasterization (e.g. in fragment shader)
- Next week
 - Texture mapping
 - Texture filtering (sampling)



Linear interpolation in screen coordinates (image space)

- Texture coordinates need special treatment



Linear interpolation in screen coordinates (image space)

- Linear interpolation in screen space works for (fake) lighting interpolation
 - But this breaks along T-joints (avoid them!)

A "T" joint

