Visibility, Culling, Deferred Shading

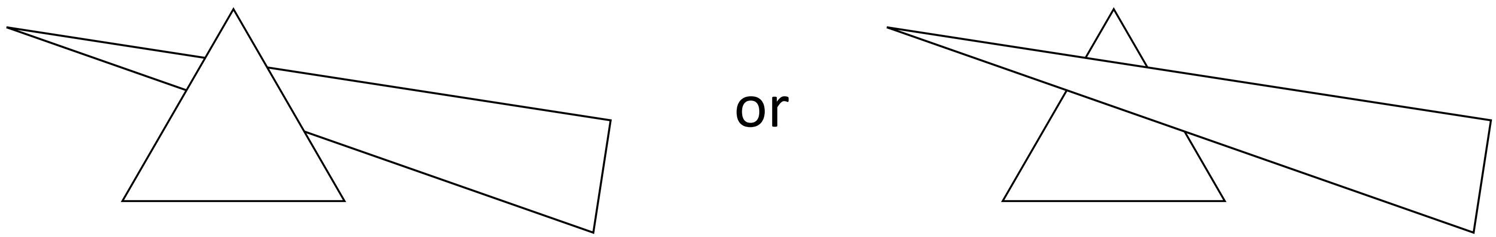
Computer Graphics

Winter Term 2016/17

Marc Stamminger / Roberto Grosso

Occlusion

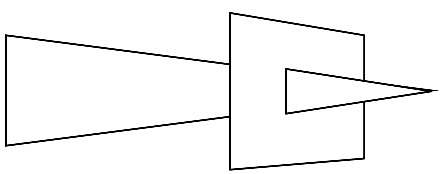
- Occlusion
  - Essential in 3D graphics



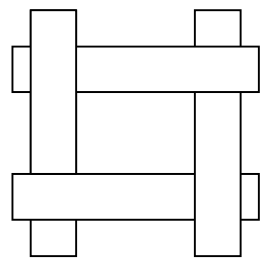
- How to create occlusion correctly?
  - Painter’s Algorithm
  - Z-Buffer
  - Ray tracing (after Christmas)

Occlusion

- Painter's Algorithm
  - Sort objects from back to front
  - Render them in this order
    - front objects draw over back objects
  - Very expensive, e.g. sorting of 1 million triangles!
  - Cannot handle
    - Penetration



Cyclic occlusion



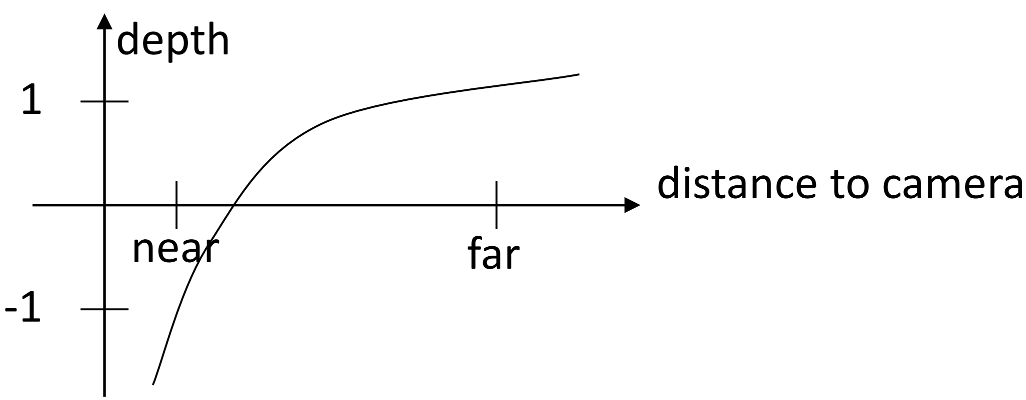
Z-Buffer

- Z-coordinates of 3D primitives equal depth values
  - after normalization → z-values are from unit interval $[-1,1]$
- Interpolate depth value during rasterization, i.e. per pixel depth
  - just as colors for Gouraud-shading
- Z-Buffer
  - buffer with same size as image.
  - Stores depth of currently closest object visible through this pixel
  - Occlusion by simple depth test (z at pixel (x, y)) → can be implemented in hardware

```
setpixel(x,y,depth,color)
       if(zBuffer(x,y) > depth)
              screen(x,y) := color
              zBuffer(x,y) := depth
       endif
```

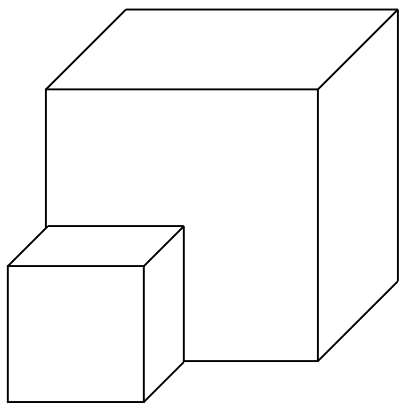
Z-Buffer

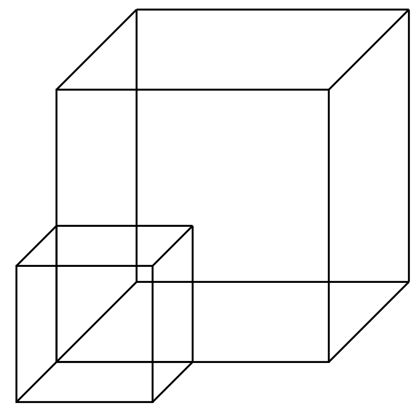
- "Depth" is z-value after normalization $\Rightarrow z \in [-1,1]$
- Projective mapping maps lines to lines
  → triangles mapped to triangles by normalization
  (unless triangle intersects $z = 0$ → problems with clipping after normalization)
- non-perspective interpolation of z-values
- Precision of z-values in z-buffer important
  - depth values mostly close to 1 (comes from perspective mapping)
  - differences in depth become small for distant objects
  - choose n reasonably large
  - at least 24 bit integer or 32 bit float needed



Z-Buffer

- Tricks: Hidden-Line-Rendering ↔ Wireframe Rendering





  - render polygons to depth buffer only in 1st pass
  - render outlines in 2nd pass and use contents of depth buffer from 1st pass

Z-Buffer

- Problem: “z-buffer fighting”
  - Pixels from 2nd pass exactly on surface from 1st pass.
  - Effects of rounding
  - Some pixels in 2nd pass occluded

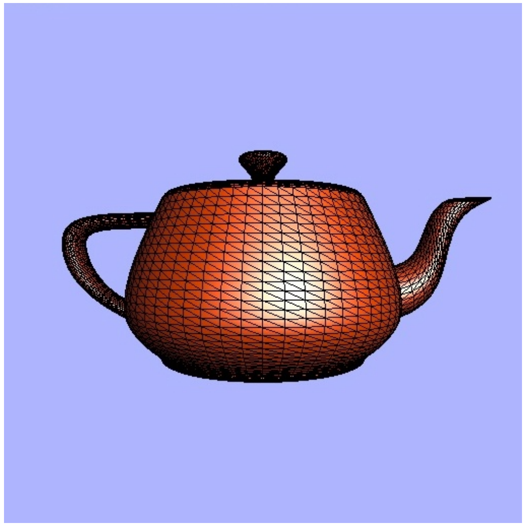
- Solution
  - Move outline towards camera by some delta (or move polygon away from camera)
  - Careful choice of delta required to avoid unwanted additional occlusion effects

Z-Buffer

wireframe

hidden line with polygon offset

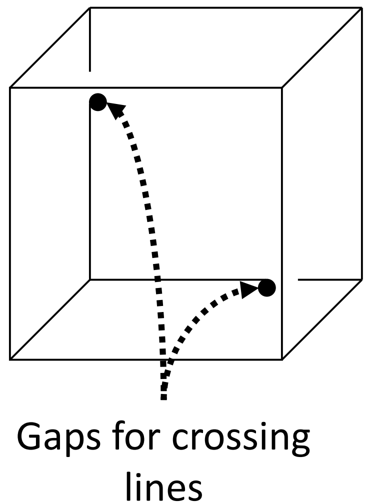
Z-fighting problem

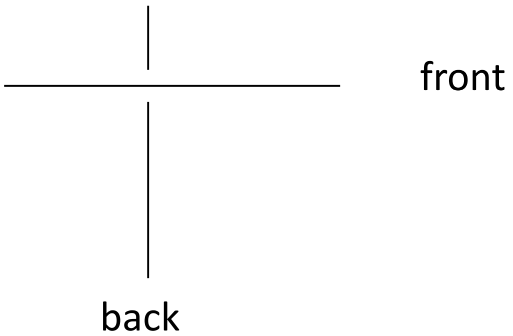


polygon and polygon outline with polygon offset

Z-Buffer

- Tricks: Haloing



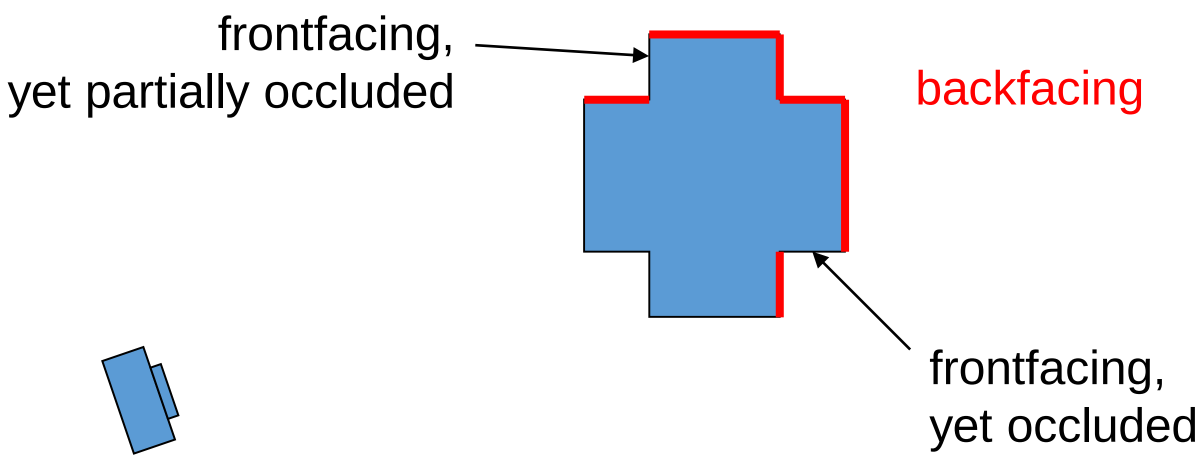


- Algorithm
  - Render thick outlines to depth buffer only in 1st pass
  - Render lines again in normal thickness with offset
  - Invisible thick lines hide back lines
  - Gaps occur

Back Face Culling

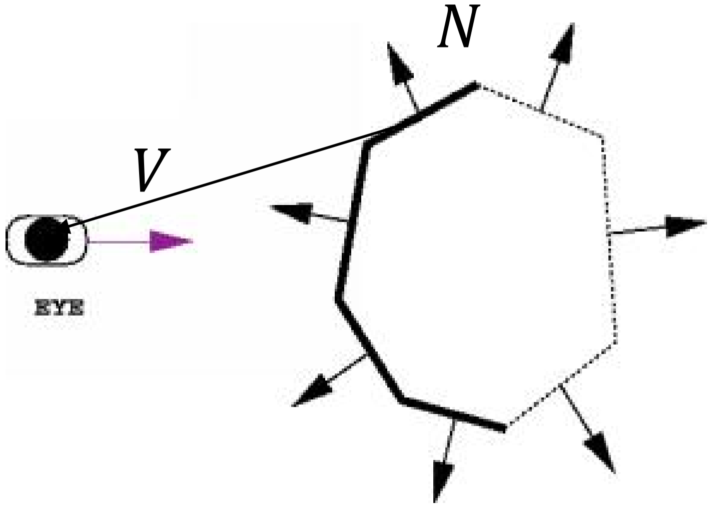
- For solid objects, every surface triangle has an outer side and an inner side
- Front facing triangle: triangle, of which we see the outer side
- Back facing triangle: non-front facing triangle
- We cannot see back facing triangles (unless we are inside the object)
- But: also front-facing triangles can be occluded (partially or completely)

- → **Back face culling**: remove such backfacing faces



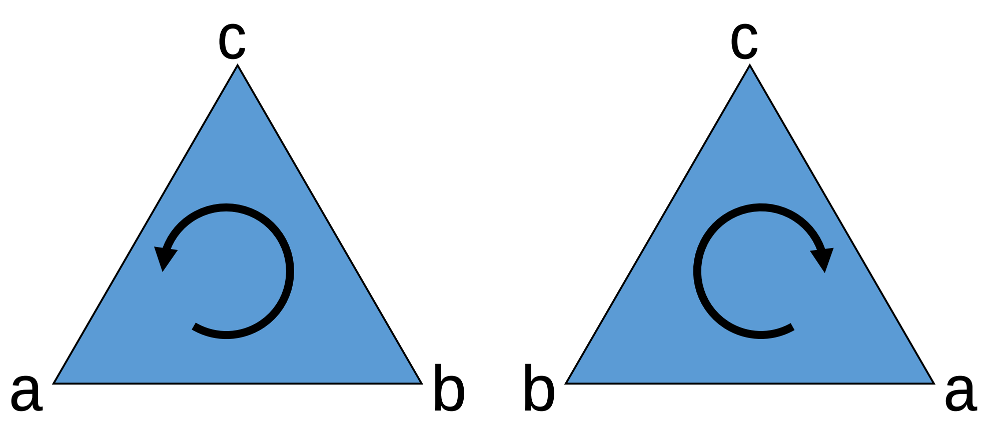
Back Face Culling

- How can we decide per triangle whether it is back facing ?
- Version 1 (world space)
  - assign a normal N to each face, pointing outwards
  - render triangle, only iff $V \circ N > 0$
  - problem: often N is not known,
    but only the lighting normal per vertex



Back Face Culling

- How can we decide per triangle whether it is back facing ?
- Version 2 (screen space)
  - orient vertices
    - when looking from the outside, order vertices counterclockwise
  - when projected to screen space
    - if also counterclockwise in screen space → front face → render
    - if orientation changes → back face → cull

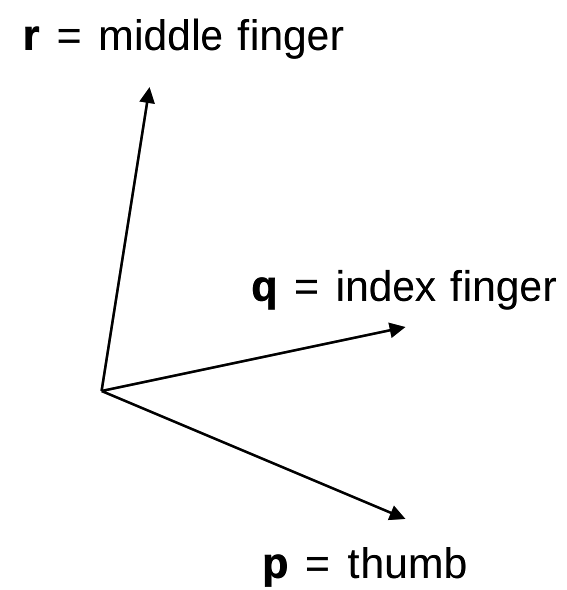


counterclockwise orientation

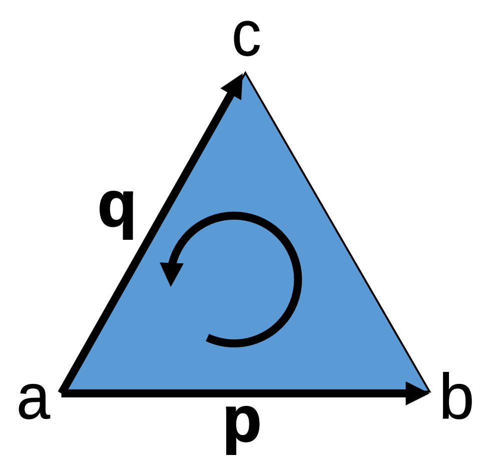
clockwise orientation

Back Face Culling

- how do we test orientation?
- Vector product defines orientation
  - given: 3D-vectors **p,q**
  - **r** = **p** x **q**:
    - **r** perpendicular to **p** and **q**
    - **p**,**q** and **r** are *"right handed"*



- Use this to test orientation of 2D points **a**,**b**,**c**
  - lift to 3D:
  - **a** → ($\mathbf{a}_1$,$\mathbf{a}_2$,0), **b,c** analog
  - **p** = **b** – **a**, **q** = **c** - **a**
  - compute **p** x **q**
  - **a,b,c** counterclockwise
    ⇔ $(\mathbf{p} \times \mathbf{q})_z > 0$



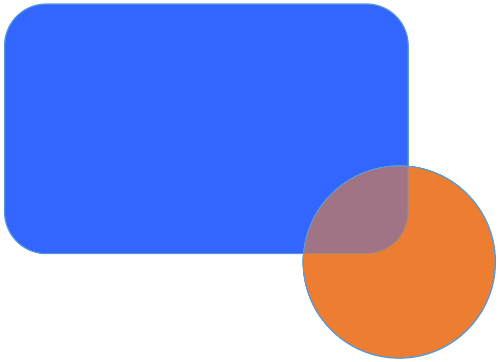
Back Face Culling

- supported by OpenGL:
  - // front faces: **c**ounter **c**lock **w**ise
    `glFrontFace(GL_CCW);`
    // cull back faces
    `glCullFace(GL_BACK);`
    // back face culling on
    `glEnable(GL_CULL_FACE);`

- Does not replace visibility test!
- It just quickly sorts out 50% of the triangles before rasterization!

Transparency

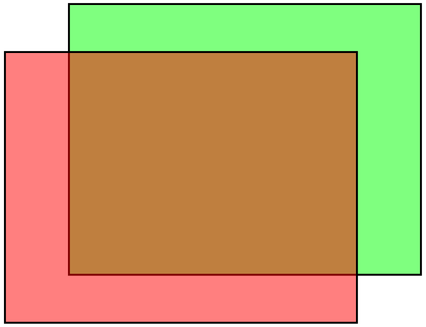
- Transparency
  - Technique: Blending
  - During rendering, new pixels do not overwrite previous ones but the values are "blended"



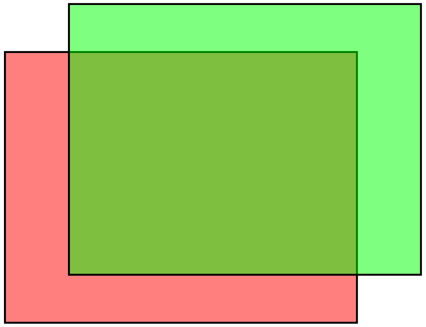
- α-Blending
  - pixel: $= (1 - \alpha) \cdot old \; + \; \alpha \cdot new$
- Allows drawing of semitransparent objects
  - $\alpha = 0.5 \Rightarrow$ pixel := ½ old + ½ new $\Rightarrow$ half transparent objects
  - α is not transparency but "opacity" = 1 – transparency

Transparency

- α is often 4th color component $\Rightarrow$ RGBA instead of RGB
- (1, 0, 0, 0.1) $\Rightarrow$ very transparent red
- α=1 corresponds to opaque rendering
  - (0, 1, 0, 1) $\Rightarrow$ opaque green (transparency == 0)
  - pixel := 0·old + 1·new = new $\Rightarrow$ overwrite
- α-blending is not commutative
  - Results change depending on order of blending
  - Rendering without sorting leads to wrong results



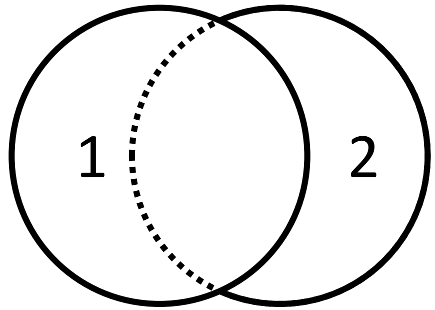
50% red over
50% green over
100% white



50% green over
50% red over
100% white

Transparency

- Problem: z-Buffer + Transparent objects
  - Example: render 2 semitransparent spheres (1 in front of 2) using α-blending and z-buffer



  - If 1 is drawn before 2, 1 will be opaque because z-buffer hides sphere 2
  - If 2 is drawn first, the result is correct

  - Z-Buffer assumes objects are opaque !
- So:
  - Opaque objects should be rendered first with z-buffer
  - Then, transparent objects should be rendered back-to-front

Transparency

- Simple methods to handle transparency
  - Approach a) (correct)
    - Do not use z-buffer at all but sort objects back to front
  - Approach b) (correct)
    - First render opaque objects with z-buffer
    - Then, “freeze” z-buffer (set to read-only)
    - Finally, sort transparent objects and render back to front
  - Approach c) (faster, but not always correct)
    - First render opaque objects
    - Then, “freeze” z-buffer
    - Finally, render transparent objects without sorting

Handling Transparency with Depth Peeling

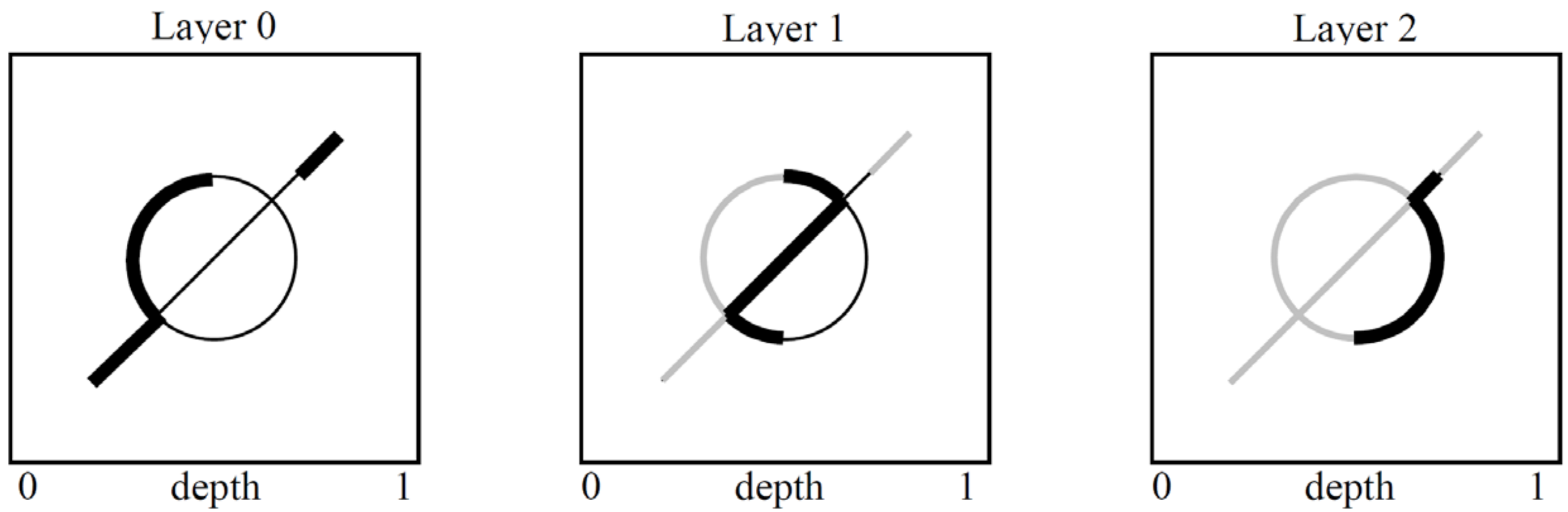
- Idea:
  - Render scene multiple times
  - At each render pass, let only fragments survive that are further away from camera than in previous pass
    - Easy to do in fragment shader
  - Single depth layers of the scene are “peeled”

- Result: Layered Depth Image (LDI) [Shade et al. 98]
  - n depth images, where n is maximum depth complexity

LDI:

! not relevant for exam !

Handling Transparency with Depth Peeling



- Depth peeling from front to back (left to right)
  - Surface: bold black lines
  - Hidden surface: thin black lines
  - “Peeled away” surface: light grey lines

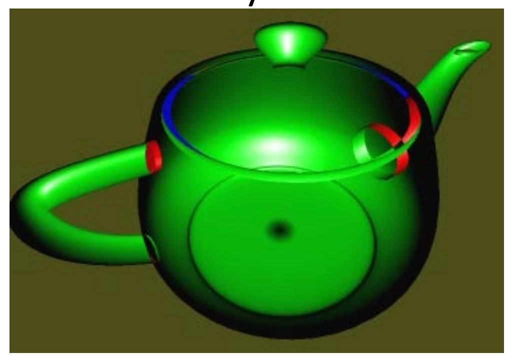
! not relevant for exam !

Handling Transparency with Depth Peeling

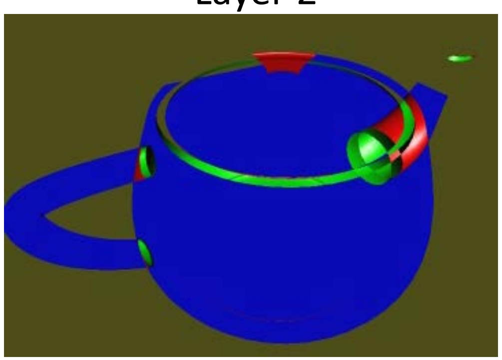
- Transparency can be achieved by blending these layers

Layer 0

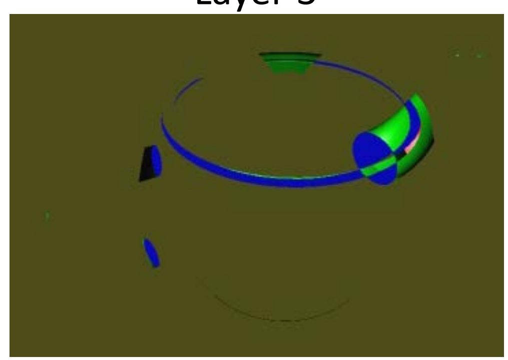
Layer 1



Layer 2



Layer 3



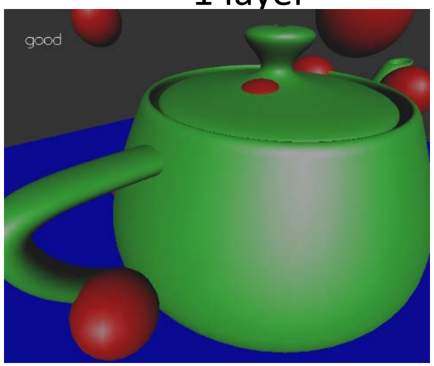
Interactive Order-Independent Transparency [Everitt 01]

! not relevant for exam !

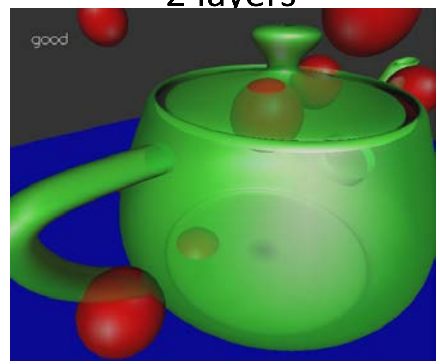
Handling Transparency with Depth Peeling

- Transparency can be achieved by blending these layers

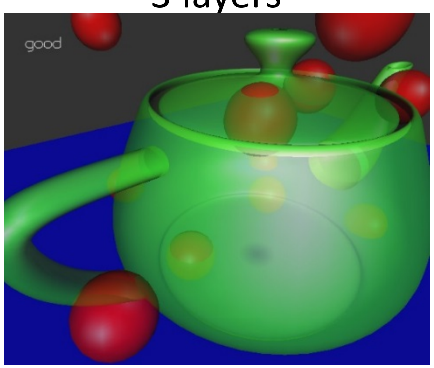
1 layer



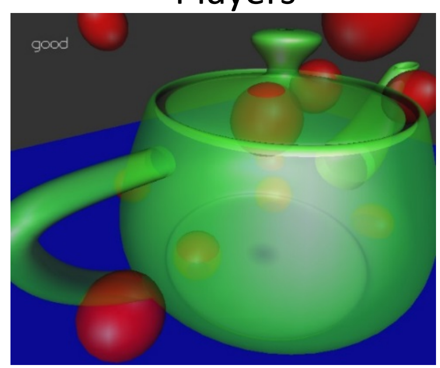
2 layers



3 layers



4 layers



Interactive Order-Independent Transparency [Everitt 01]

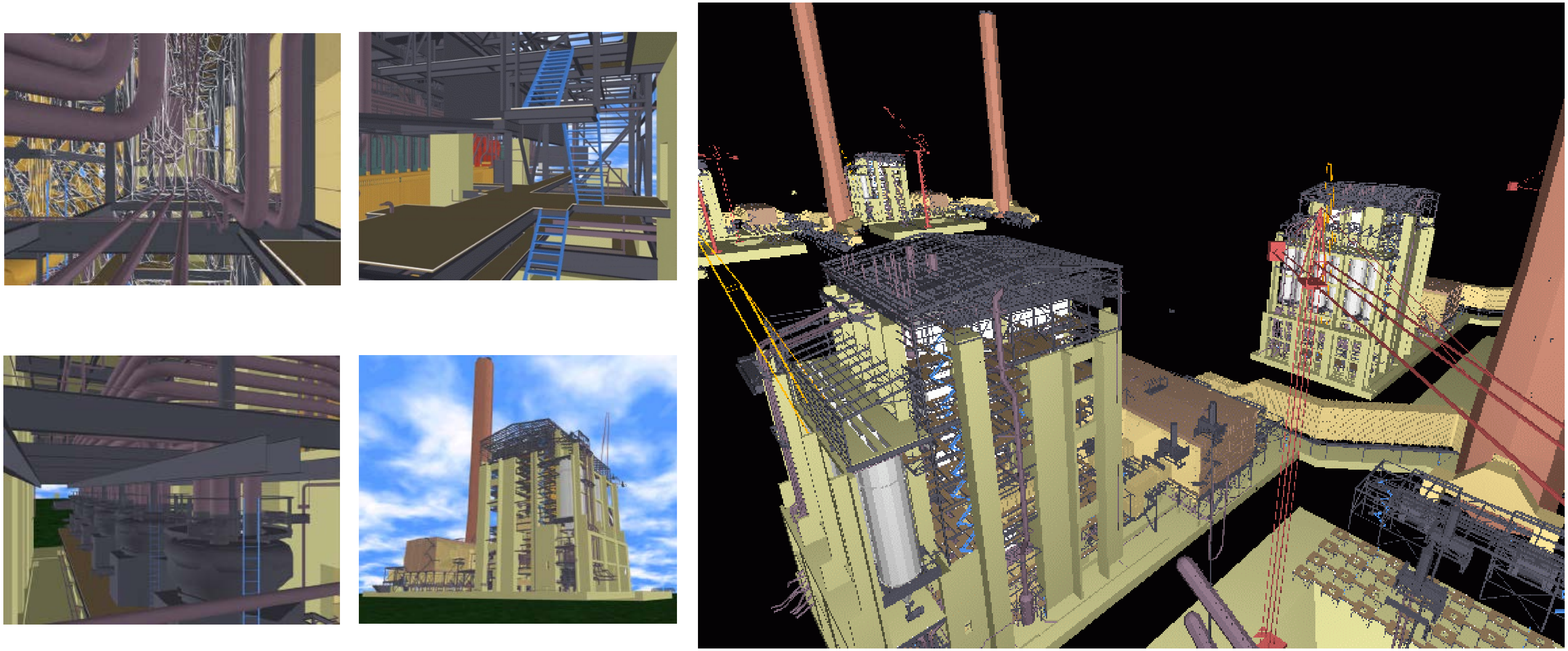
! not relevant for exam !

Deferred Shading

- Pixel shaders often bottleneck
  - Costly shaders
  - Many light sources
  - High depth complexity
    - Draw front-to-back, but sorting within a draw call?
  - Too many fragments (tiny triangles)

Deferred Shading

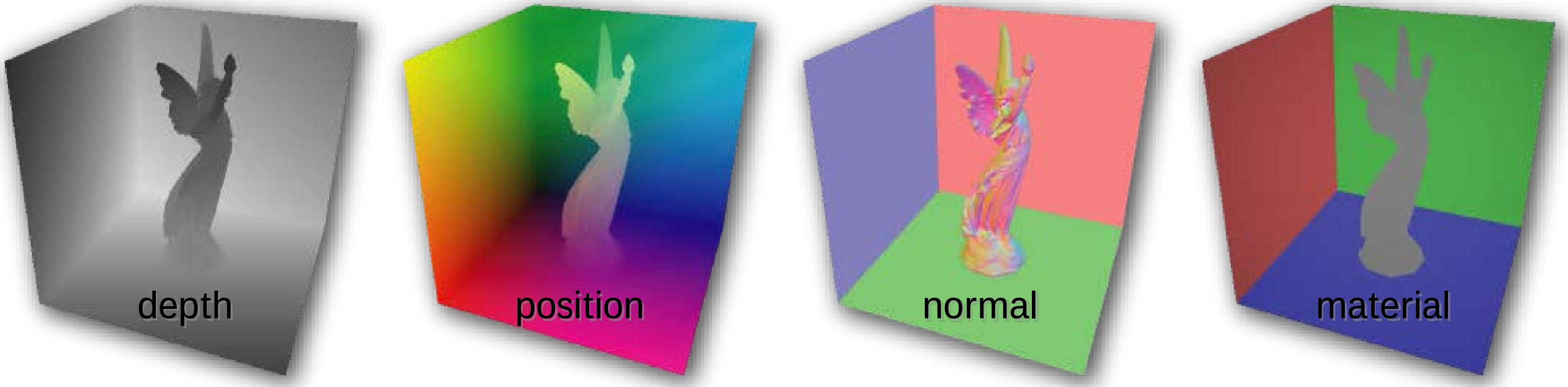
- High-depth complexity
  - Occluded fragments will be shaded (overdraw, fill-rate)



UNC Power-Plant ≈50 mio triangles [Aliaga et al. 99], [Wald et al. 01]

Deferred Shading

- Idea
  - first, fill a **geometry buffer (= “G-Buffer”)**
    → for each pixel, we have all information for the front-most pixel
  - Position
  - Normal
  - Material
  - …



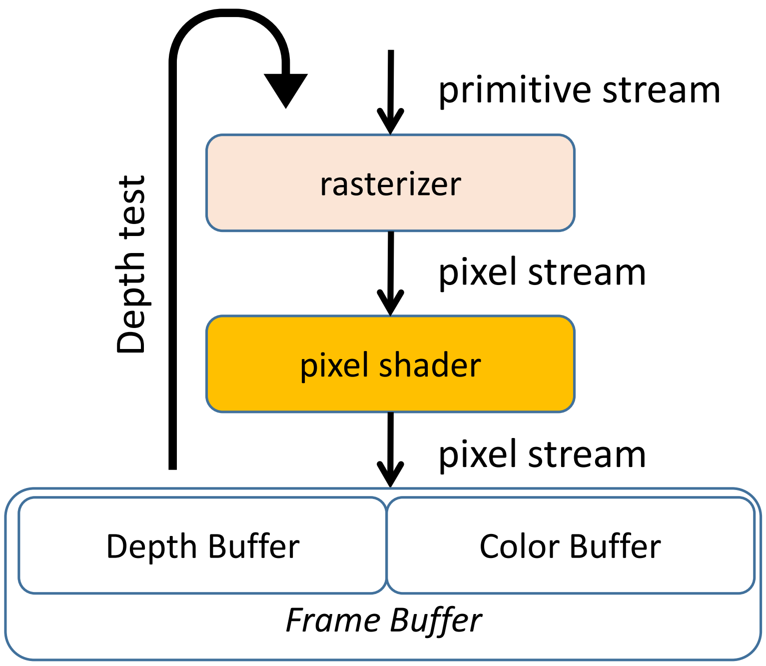
Deferred Shading

- Then, in a second pass, we compute the lighting for each pixel
  → only one lighting computation per pixel
- this computation is initialized by rendering a single quad over the entire screen
  → “Screen space aligned quad”
- or by a **compute shader**

Render Targets (RT)

- By default OpenGL has
  - Depth Buffer (e.g., 16 / 32 bit)
  - Color Buffer (e.g., RGBA8 / RGBA32F)

- Front and Back Buffer
  - One is bound to pipeline
  - One is set to video out



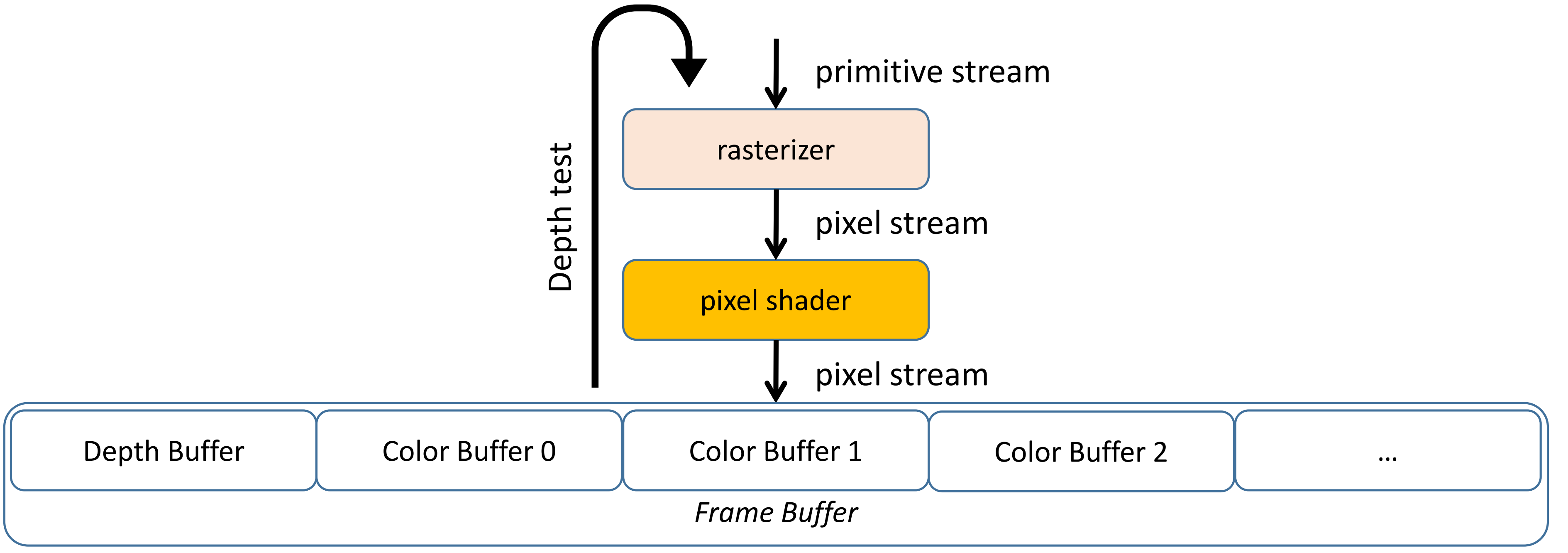

```
// typical render callback (each frame)
glClear(
     GL_COLOR_BUFFER_BIT |
     GL_DEPTH_BUFFER_BIT
     );

renderScene();


//swap front and back buffer
glutSwapBuffers();
```

```
// fragment shader output
void main(void) {
     ...
     gl_FragColor = color;
     gl_FragDepth = depth;
}
```

Multiple Render Targets (MRT)




```
// fragment shader output - MRT
void main(void) {
     ...
     gl_FragData[0] = color0;
     gl_FragData[1] = color1;
     gl_FragData[2] = color2;
     ...
     gl_FragDepth = depth;
}
```

Multiple Render Targets (MRT)

- Up to 8 (color) render targets
  - Can be any format R32F, RGB8, RGBA32F, etc.
  - Must have same dimensions (number of pixels)
  - All RTs must have some number of bits (you can mix channels)
    - OK:
      - RT0 = R32F (32 bit)
      - RT1 = R16G16F (32 bit)
      - RT2 = RGBA8 (32 bit)
    - Does NOT work:
      - RT0 = R16G16F (32 bit)
      - RT1 = RGBA16F (64 bit)

- Only 1 depth buffer
  - There is only one hardware Z-Buffer

Multiple Render Targets (MRT)

- Render to texture: Framebuffer Objects (FBOs)

```
// FBO: render to texture – create FBO
GLuint color_tex0, color_tex1, depth_tex;
glGenTextures(1, &color_tex0);
...

//allocate texture memory
glBindTexture(GL_TEXTURE_2D, color_tex0);
glTexParameter(...);          //set texture parameters
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA8, 256, 256, 0, GL_BGRA, GL_UNSIGNED_BYTE, NULL);
...

//generate framebuffer object
Gluint fbo;
glGenframebuffers(1, &fbo);
glBindFramebuffer(GL_FRAMEBUFFER, fbo);
glBindFramebufferTexture2D(
          GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT, GL_TEXTURE2D, depth_tex, 0);
glBindFramebufferTexture2D(
          GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0, GL_TEXTURE2D, color_tex0, 0);
glBindFramebufferTexture2D(
          GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT1, GL_TEXTURE2D, color_tex1, 0);
...
```

Multiple Render Targets (MRT)

- Render to texture: Framebuffer Objects (FBOs)

```
// FBO: render to texture – render / bind
glBindFrameBuffer(GL_FRAMEBUFFER, fbo);

//will write output into textures
Draw();

glBindFrameBuffer(GL_FRAMEBUFFER, 0);
```

- Corresponding fragment shader:

```
// fragment shader output - MRT
void main(void) {
     ...
     gl_FragData[0] = color0;
     gl_FragData[1] = color1;
     gl_FragDepth = depth;
}
```

G-Buffer

- Typical layout: 16-bit float MRT

RT1	Diffuse.r	Diffuse.g	Diffuse.b	Specular
RT0	Position.x	Position.y	Position.z	Emissive
RT2	Normal.x	Normal.y	Normal.z	Free

- 16-bit float is overkill for diffuse (8-bit would be sufficient)
  - But we need at least 3 x 16-bit for position
  - So don’t have a choice due to MRT rules but to make all 16-bit

Deferred Shading