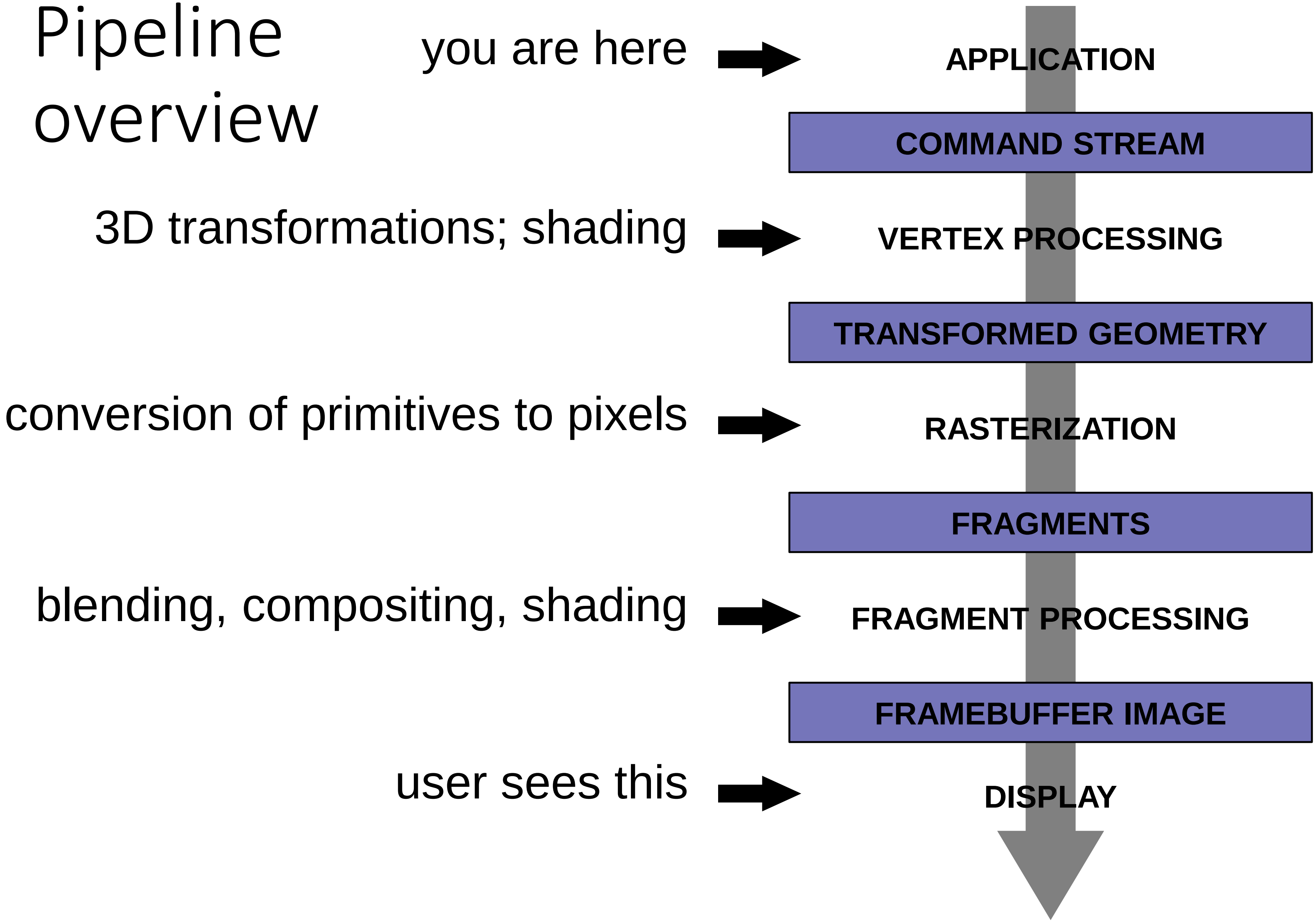
# Pipeline Operations



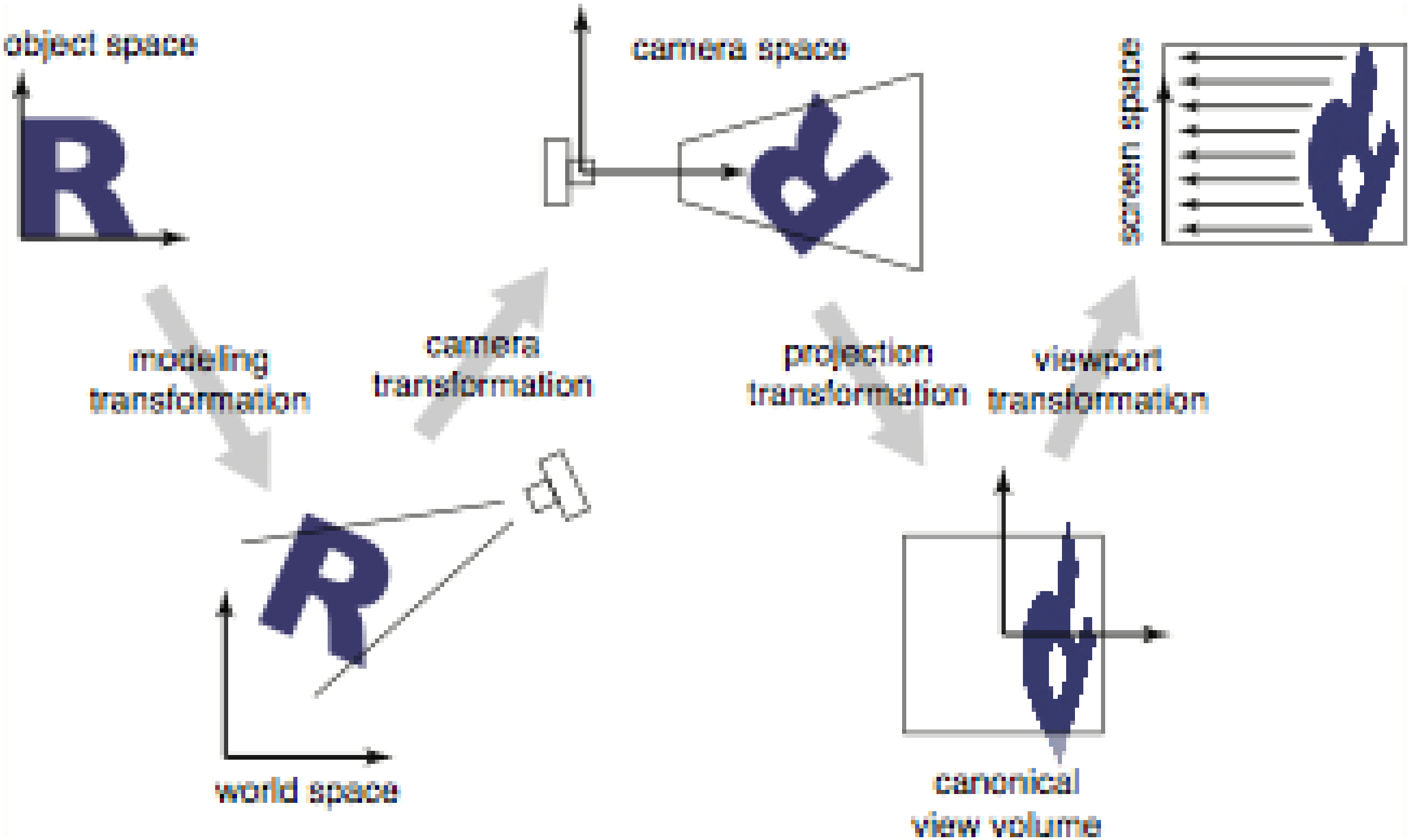
Based on slides by Steve Marschner

# Pipeline overview



# Pipeline of transformations

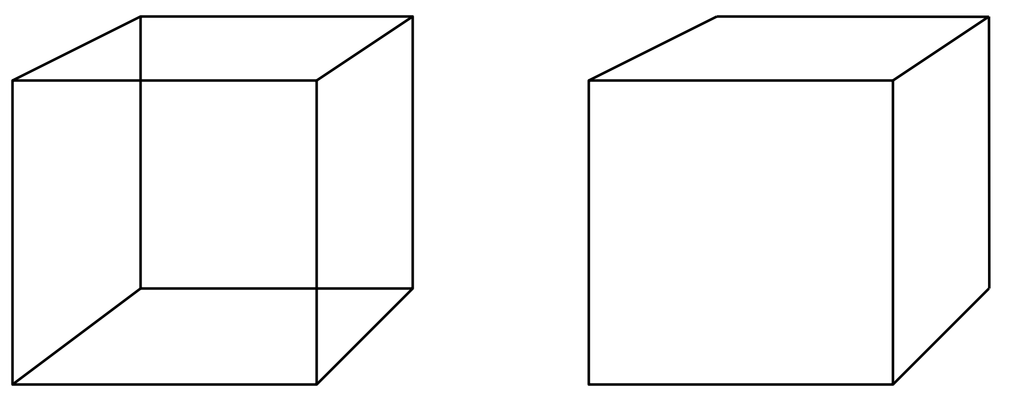
- Standard sequence of transforms




Based on slides by Steve Marschner

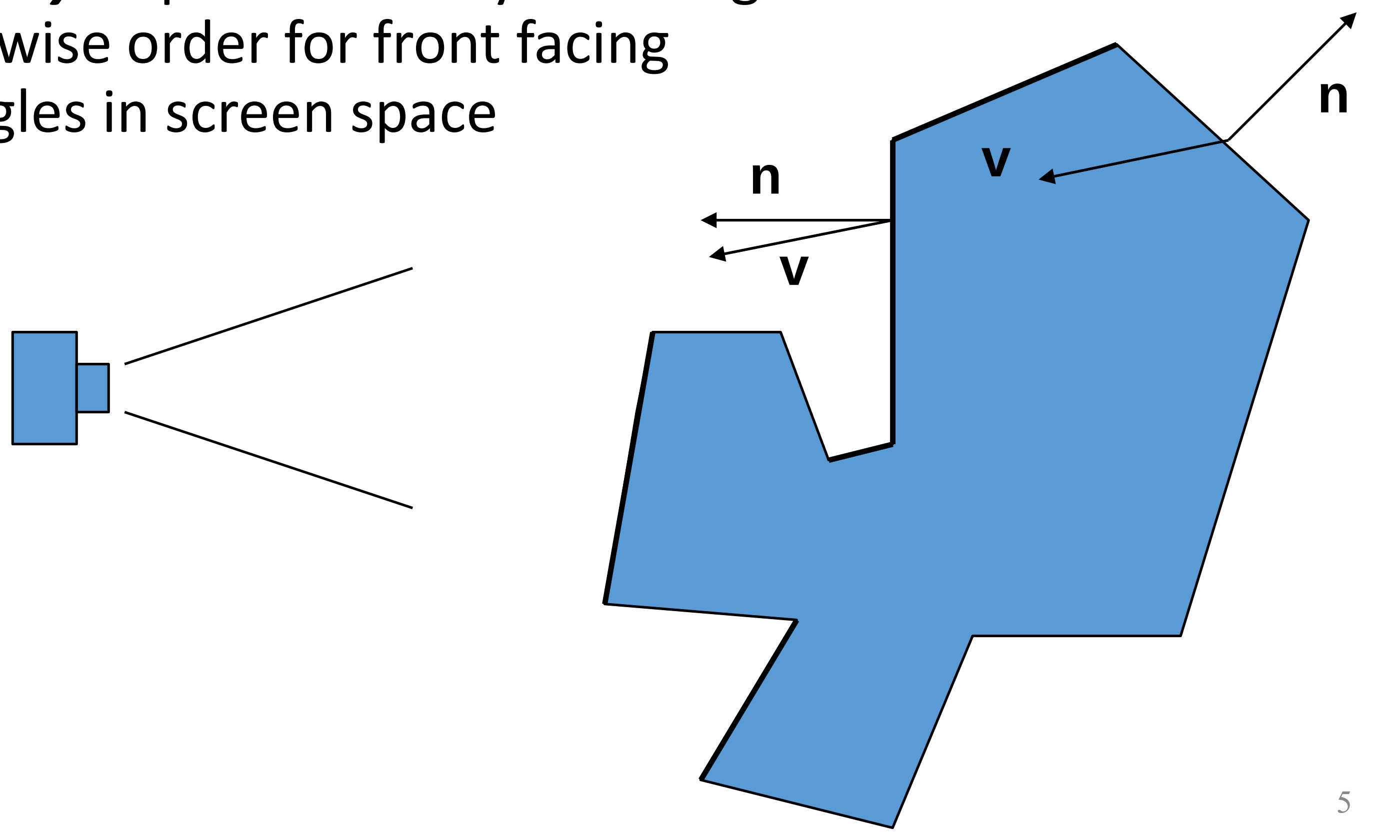
# Hidden surface elimination

- We have discussed how to map primitives to image space
  - projection and perspective are depth cues
  - occlusion is another very important cue



# Back face culling

- For closed shapes you will never see the inside
  - therefore only draw surfaces that face the camera
  - ***Could*** implement by checking **n · v**
    but v varies across the surface…
  - ***Actually*** implemented by checking counter clockwise order for front facing triangles in screen space

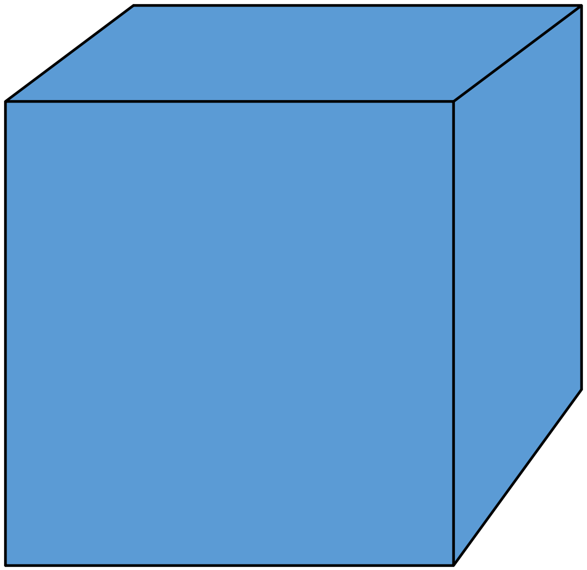





Based on slides by Steve Marschner

# Painter's algorithm

- Simplest way to do hidden surfaces
- Draw from back to front, use overwriting in framebuffer

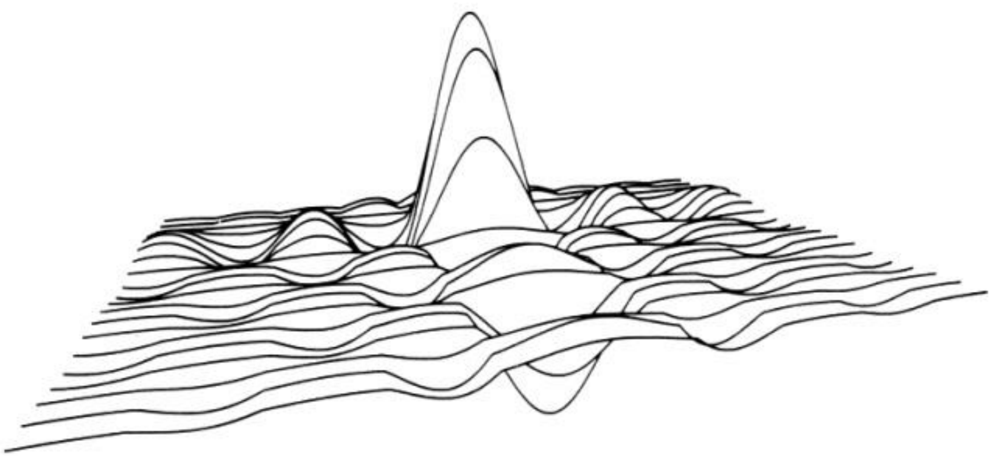




Based on slides by Steve Marschner

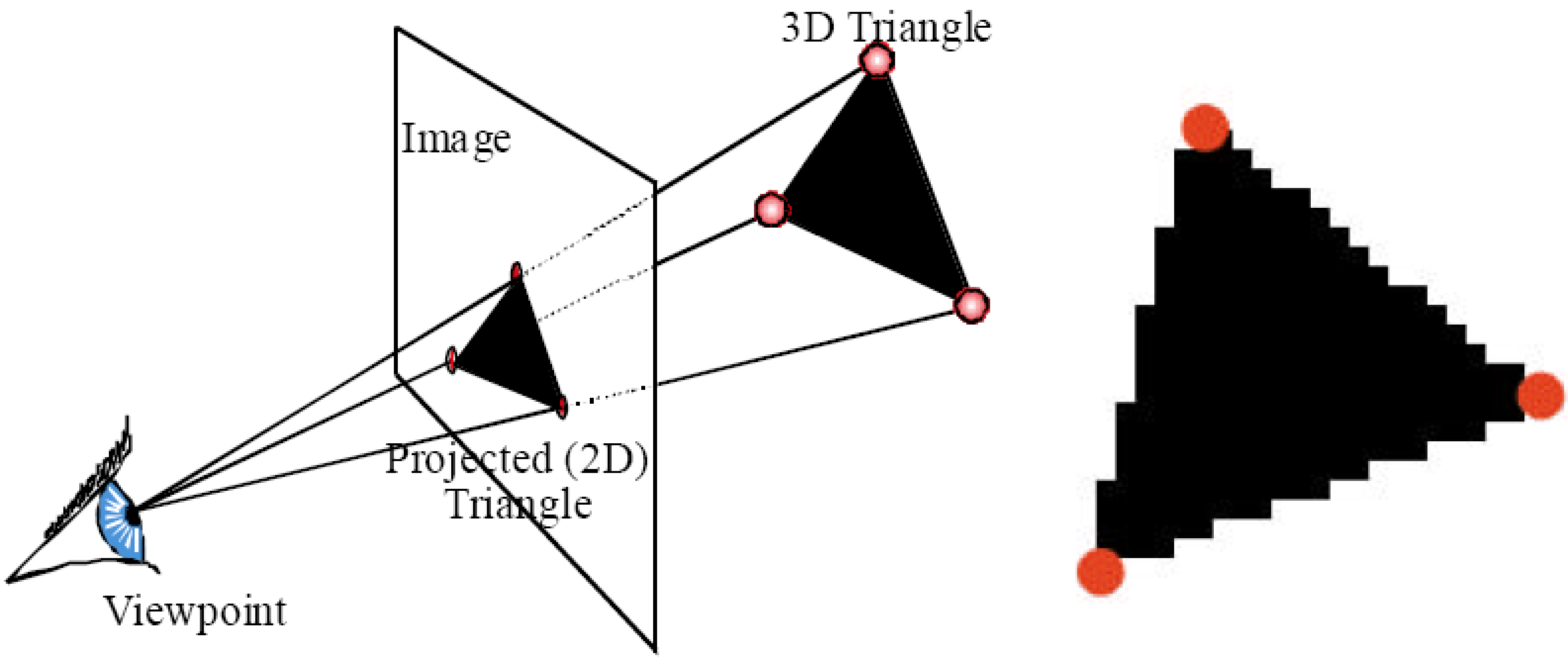
# Painter's algorithm

- Useful when a valid order is easy to come by
- Compatible with alpha blending



[Foley et al.]

# Drawing

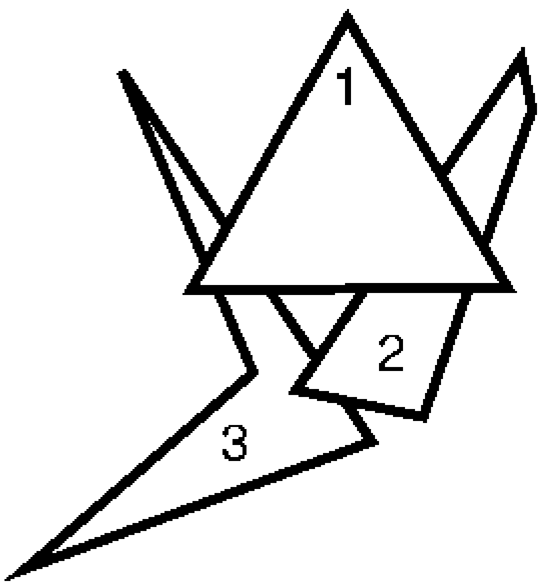


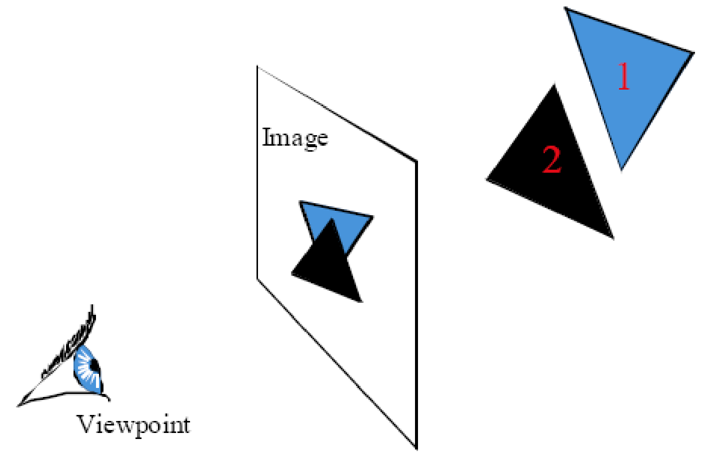

Projection (left) and rasterization (right) of a triangle.

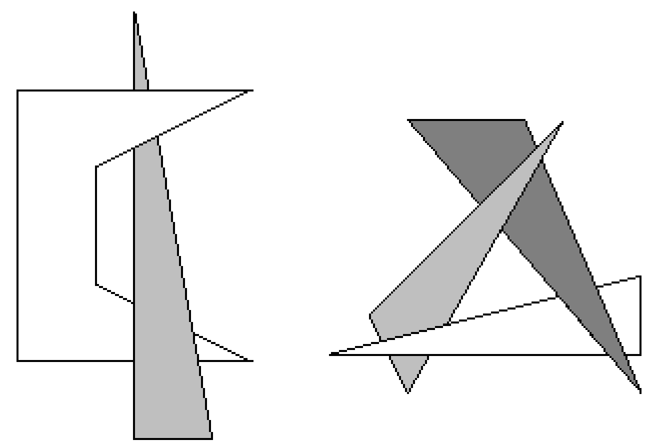
[Durand]

# Visibility

- Painter's algorithm
  - Sort back to front
  - Draw!



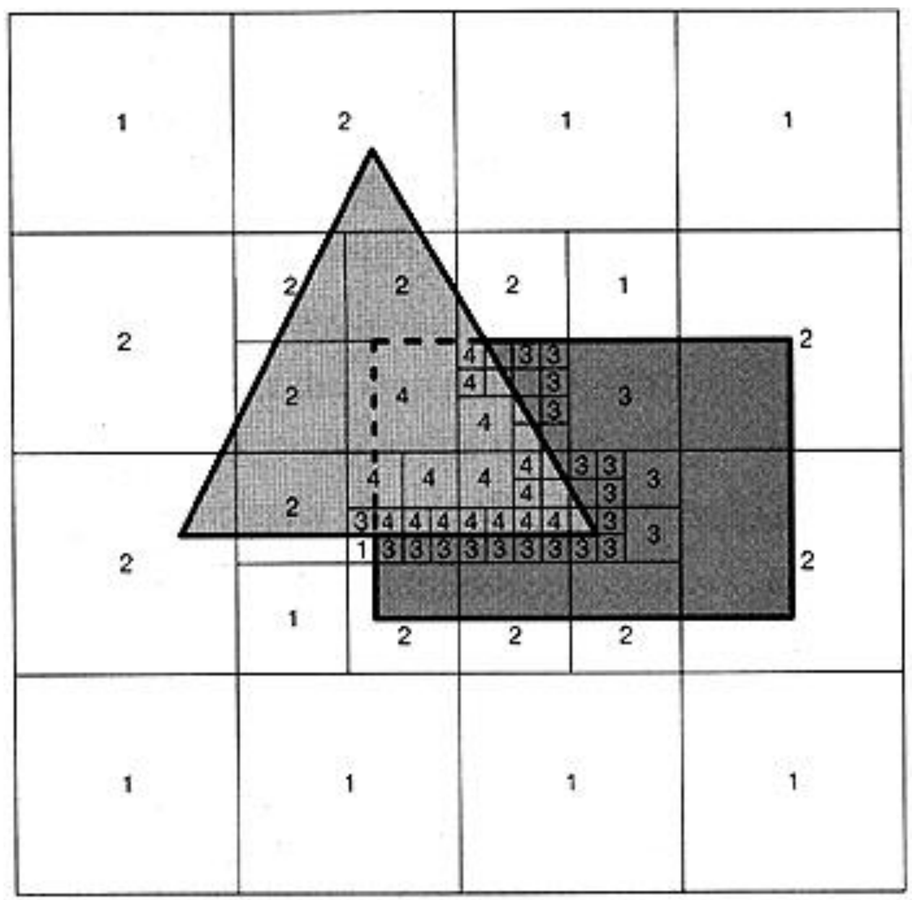



Based on slides by Steve Marschner

# Visibility

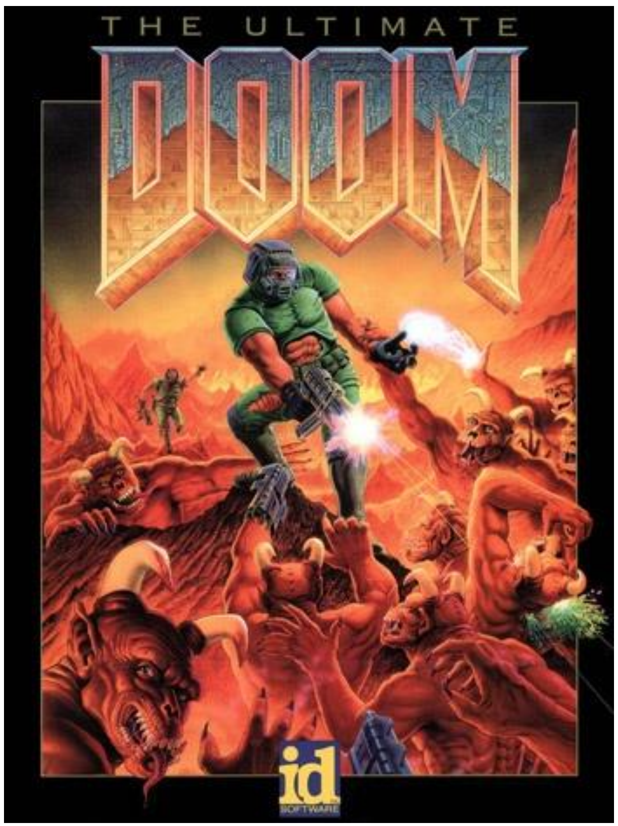
- Warnock's algorithm
  - Area subdivision
  - Apply Painter's when it will work (e.g., individual pixels)



Based on slides by Steve Marschner

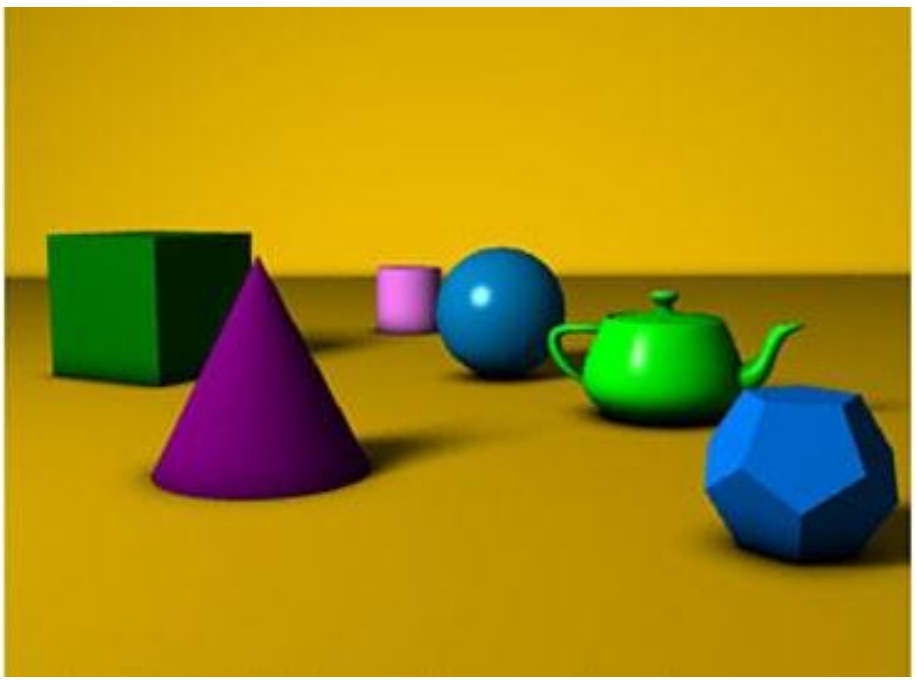
# Visibility

- Binary space partition
  - Linear time back to front sort
  - Key to 3D games before consumer level GPUs (Doom 1993)




# Visibility

- Z-buffer
  - Store depth at every pixel
  - Compare when rasterizing

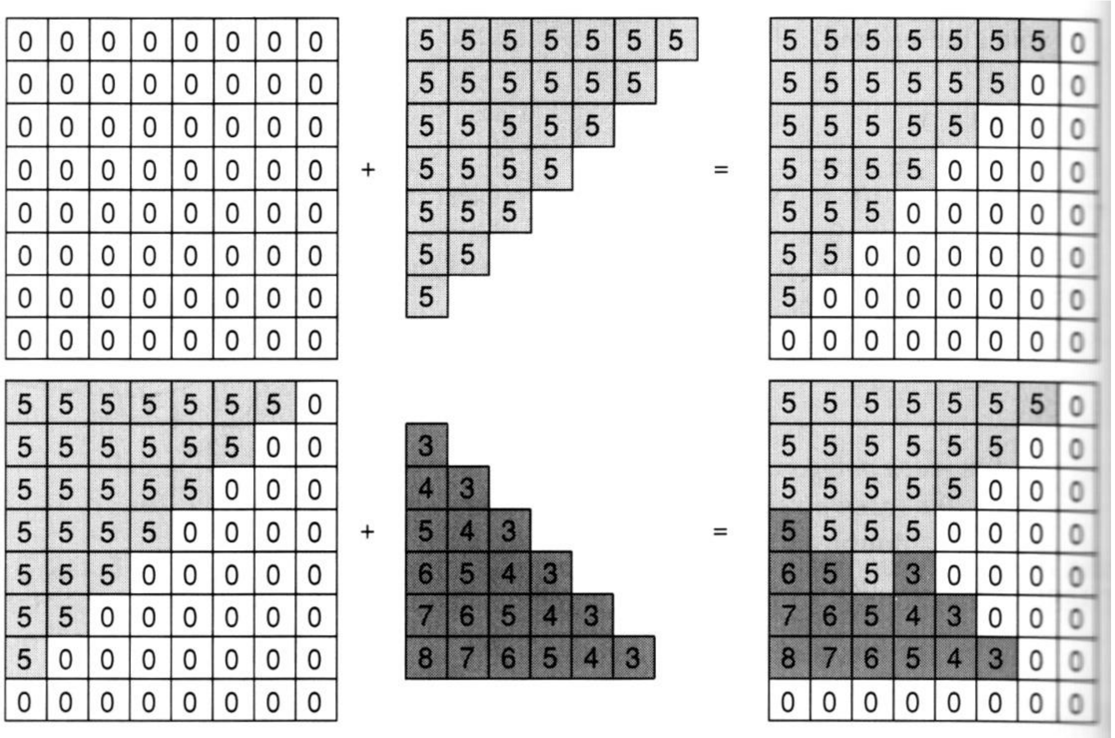


Based on slides by Steve Marschner

# The *z* buffer

- In many (most) applications maintaining a *z* sort is too expensive
  - changes all the time as the view changes
  - many data structures exist, but complex
- Solution: draw in any order, keep track of closest
  - allocate extra channel per pixel to keep track of closest depth so far
  - when drawing, compare object's depth to current closest depth and discard if greater
  - this works just like any other compositing operation

# The *z* buffer



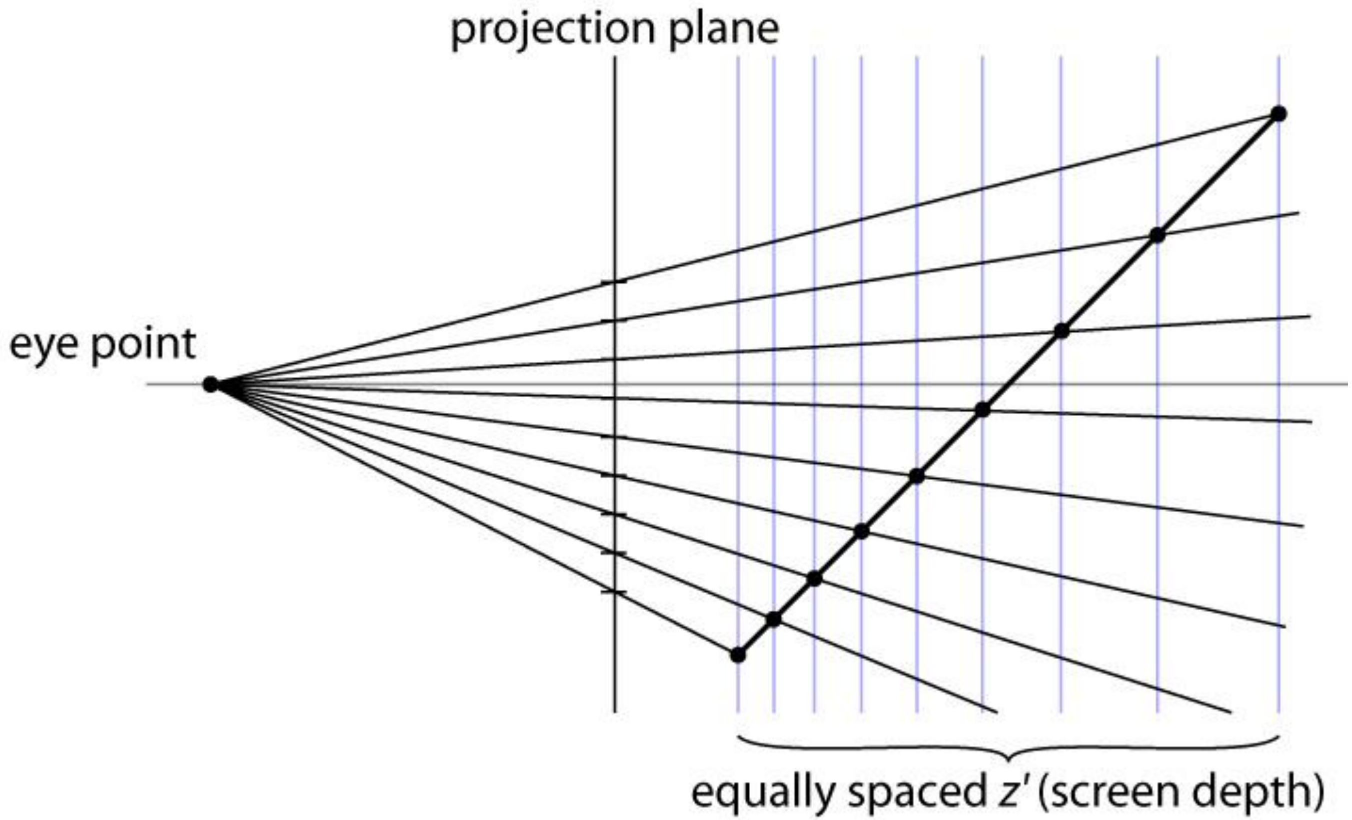
[Foley et al.]

- another example of a memory-intensive brute force approach that works and has become the standard

# Precision in *z* buffer

- The precision is distributed between the near and far clipping planes
  - this is why these planes have to exist
  - also why you can't always just set them to very small and very large distances
- Generally use $z'$ (not world $z$) in $z$ buffer

# Interpolating in projection




linear interp. in screen space ≠ linear interp. in world (eye) space

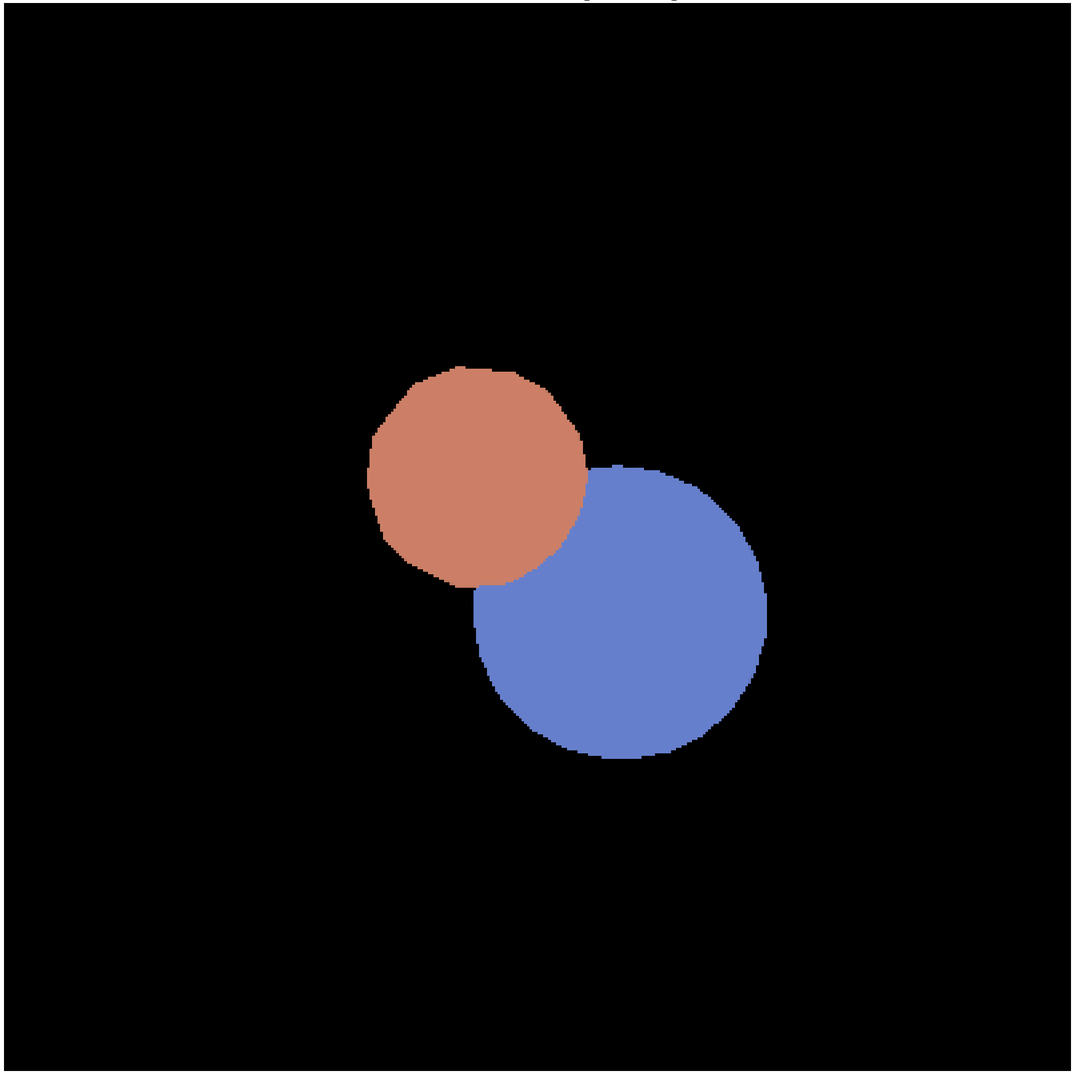


Based on slides by Steve Marschner

# Pipeline for minimal operation

- Vertex stage (input: position / vtx; color / tri)
  - transform position (object to screen space)
  - pass through color
- Rasterizer
  - pass through color
- Fragment stage (output: color)
  - write to color planes

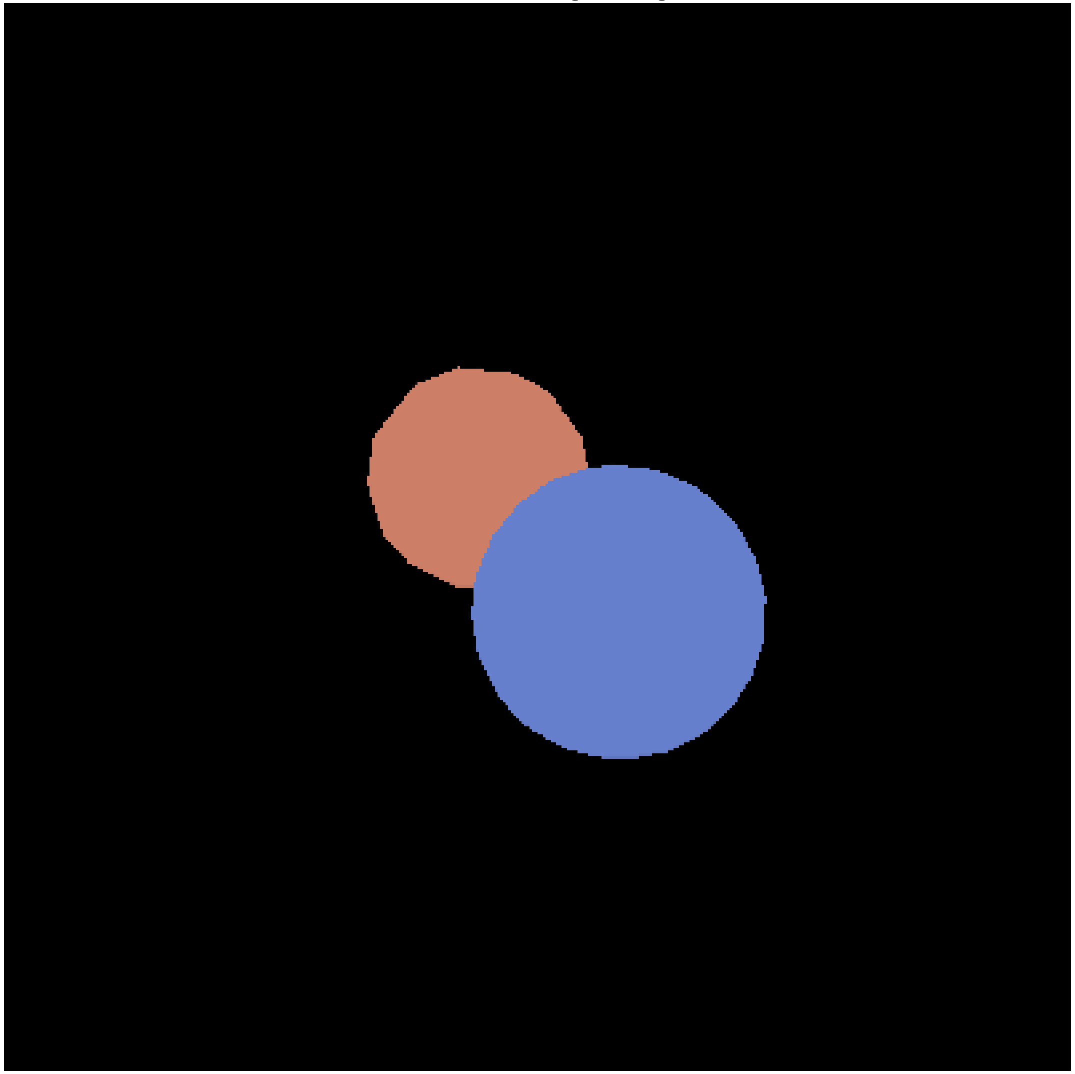
# Result of minimal pipeline



# Pipeline for basic $z$ buffer

- Vertex stage (input: position / vtx; color / tri)
  - transform position (object to screen space)
  - pass through color
- Rasterizer
  - interpolated parameter: $z'$ (screen $z$)
  - pass through color
- Fragment stage (output: color, $z'$)
  - write to color planes only if interpolated $z' <$ current $z'$

# Result of *z*-buffer pipeline



# Flat shading

- Shade using the real normal of the triangle
  - same result as ray tracing a bunch of triangles
- Leads to constant shading and faceted appearance
  - truest view of the mesh geometry

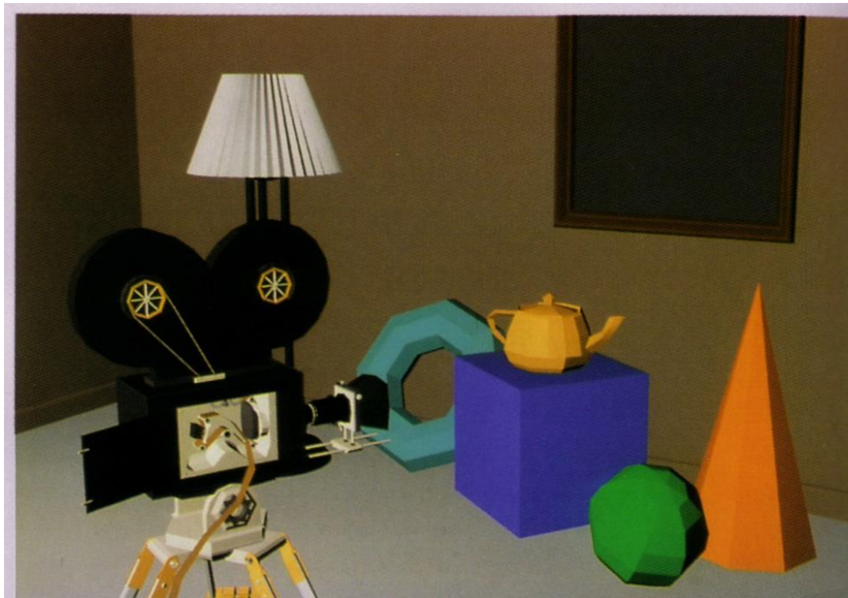


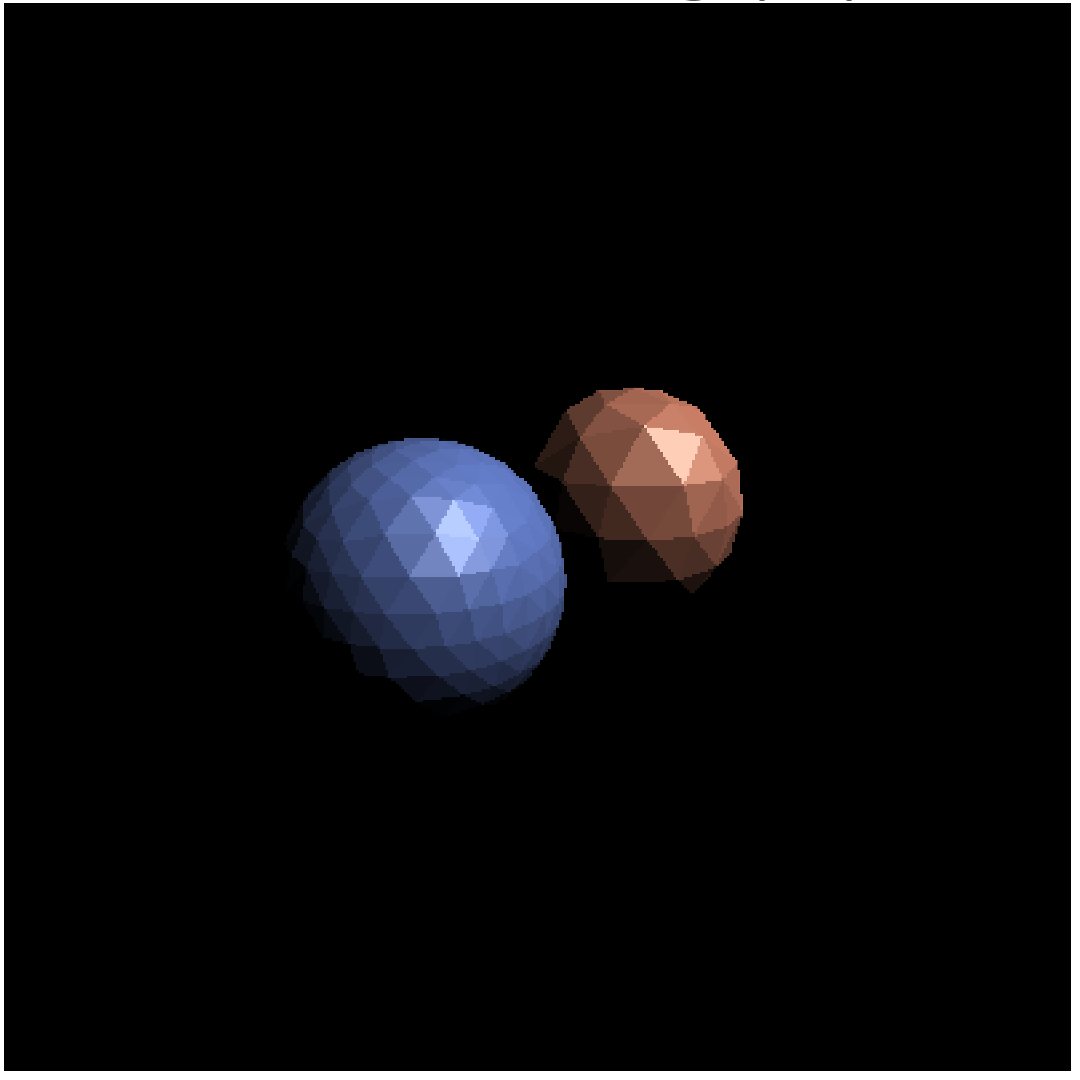
**Plate II.29** *Shutterbug.* Individually shaded polygons with diffuse reflection (Sections 14.4.2 and 16.2.3). (Copyright © 1990, Pixar. Rendered by Thomas Williams and H.B. Siegel using Pixar's PhotoRealistic RenderMan™ software.)

[Foley et al.]

# Pipeline for flat shading

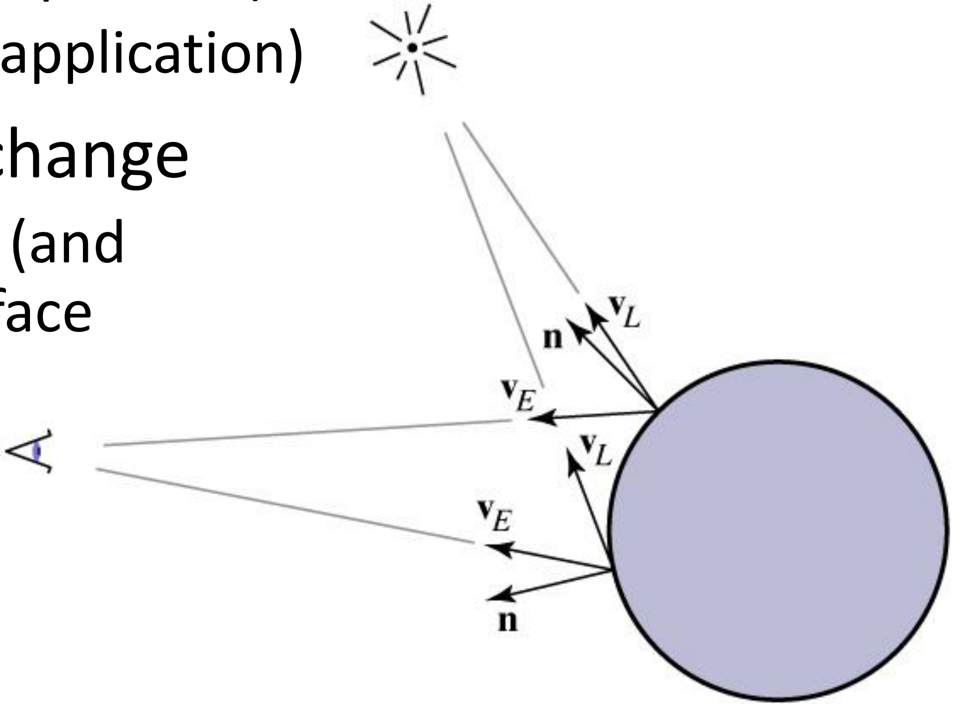
- Vertex stage (input: position / vtx; color and normal / tri)
  - transform position and normal (object to eye space)
  - compute shaded color per triangle using normal
  - transform position (eye to screen space)
- Rasterizer
  - interpolated parameters: $z'$ (screen $z$)
  - pass through color
- Fragment stage (output: color, $z'$)
  - write to color planes only if interpolated $z' <$ current $z'$

# Result of flat-shading pipeline



# Local vs. infinite viewer, light

- Phong illumination requires geometric information:
  - light vector (function of position)
  - eye vector (function of position)
  - surface normal (from application)
- Light and eye vectors change
  - need to be computed (and normalized) for each face





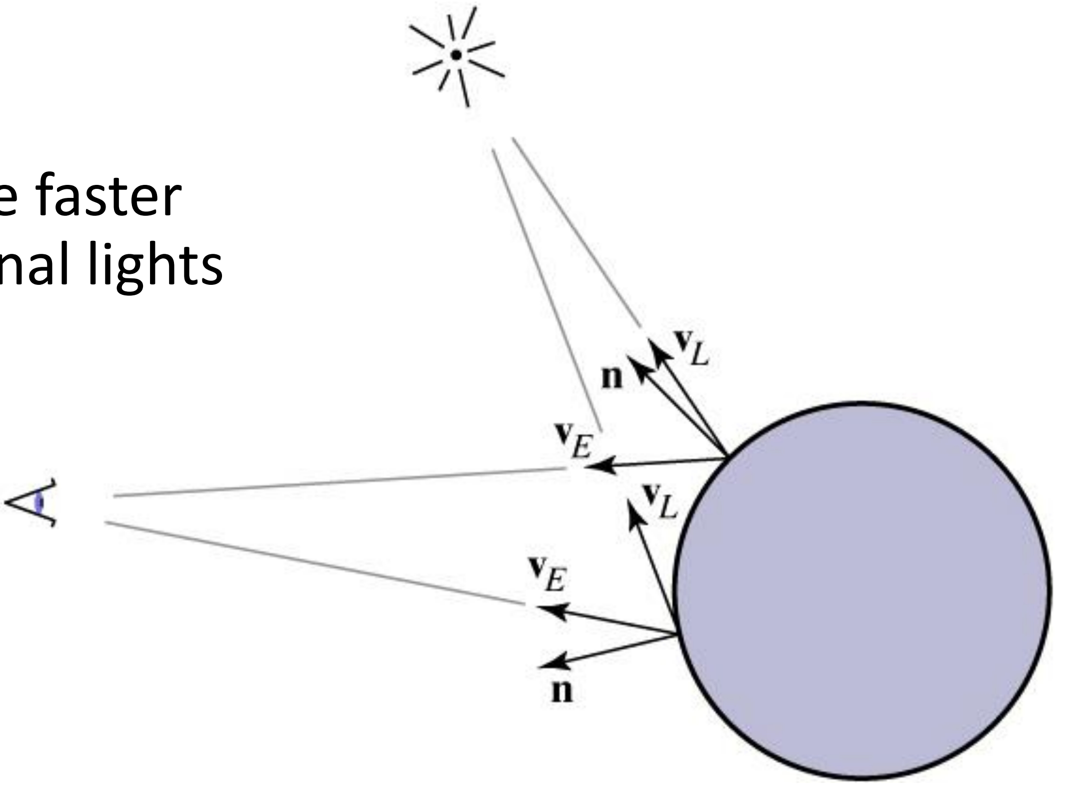

Based on slides by Steve Marschner

# Local vs. infinite viewer, light

- Look at case when eye or light is far away:
  - distant light source: nearly parallel illumination
  - distant eye point: nearly orthographic projection
  - in both cases, eye or light vector changes very little
- Optimization: approximate eye and/or light as infinitely far away

# Directional light

- Directional (infinitely distant) light source
  - light vector always points in the same direction
  - often specified by position [$x$  $y$  $z$  0]
  - many pipelines are faster if you use directional lights

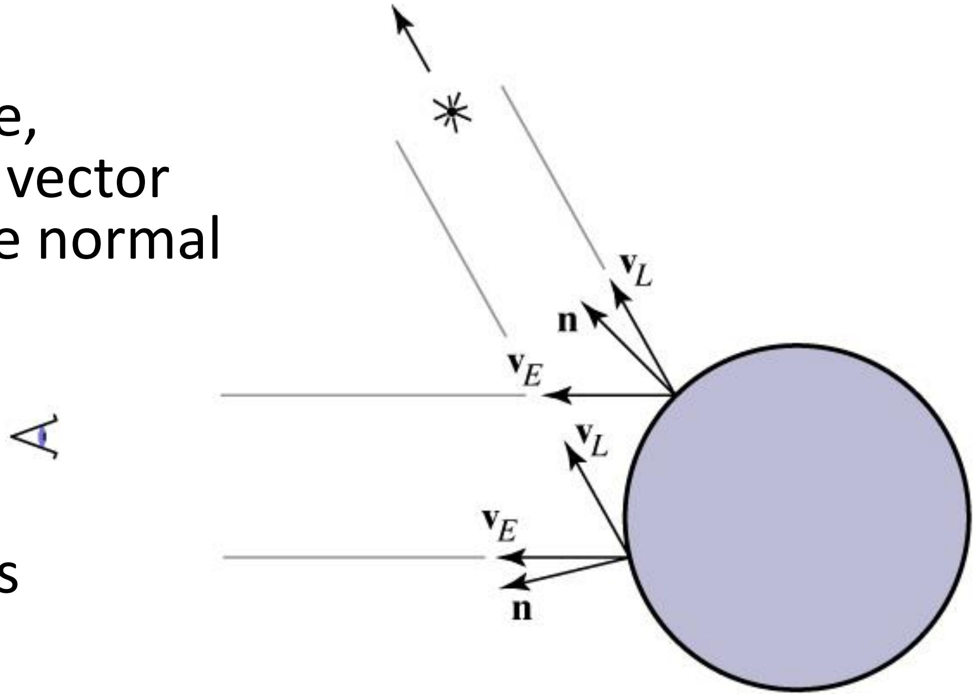





Based on slides by Steve Marschner

# Infinite viewer

- Orthographic camera
  - projection direction is constant
- "Infinite viewer"
  - even with perspective, can approximate eye vector using the image plane normal
  - can produce weirdness for wide-angle views
  - Blinn-Phong: light, eye, half vectors all constant!



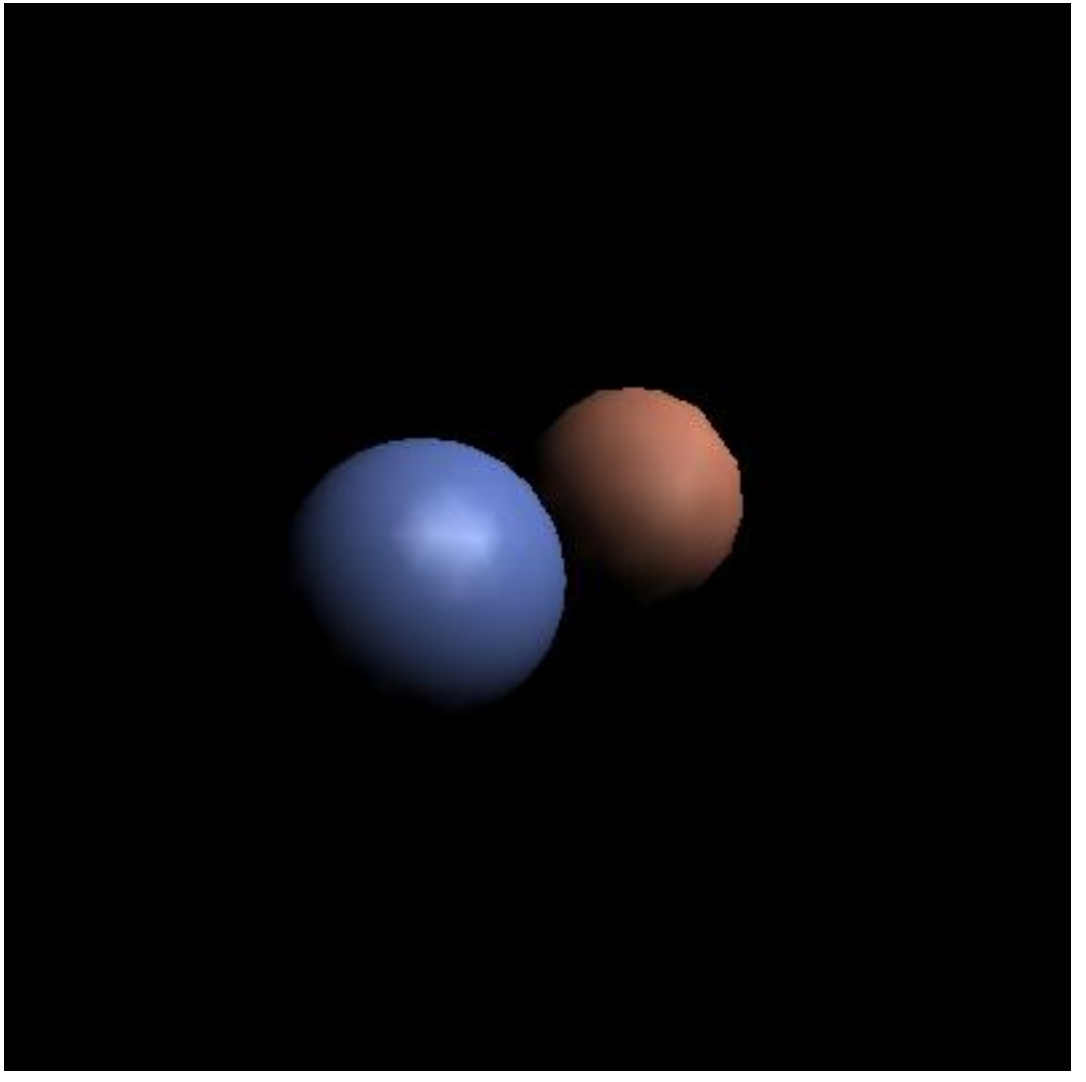



Based on slides by Steve Marschner

# Pipeline for Gouraud shading

- Vertex stage (input: position, color, and normal / vtx)
  - transform position and normal (object to eye space)
  - compute shaded color per vertex
  - transform position (eye to screen space)
- Rasterizer
  - interpolated parameters: $z'$ (screen $z$); $r, g, b$ color
- Fragment stage (output: color, $z'$)
  - write to color planes only if interpolated $z'$ < current $z'$

# Result of Gouraud shading pipeline



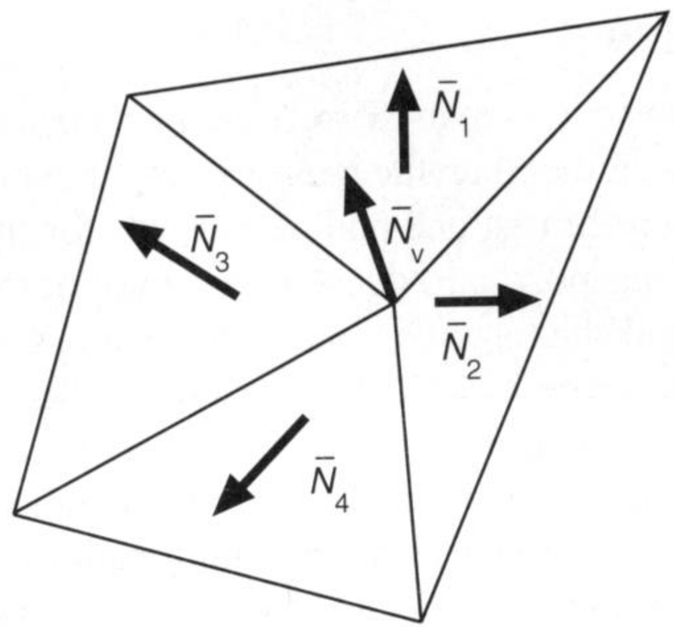


Based on slides by Steve Marschner

# Vertex normals

- Need normals at vertices to compute Gouraud shading
- Best to get vtx. normals from the underlying geometry
  - e. g. spheres example
- Otherwise have to infer vtx. normals from triangles
  - simple scheme: average surrounding face normals

$$N_v = \frac{\sum_i N_i}{\|\sum_i N_i\|}$$




[Foley et al.]



Based on slides by Steve Marschner

# Non-diffuse Gouraud shading

- Can apply Gouraud shading to any illumination model
  - it's just an interpolation method
- Results are not so good with fast-varying models like specular ones
  - problems with any highlights smaller than a triangle

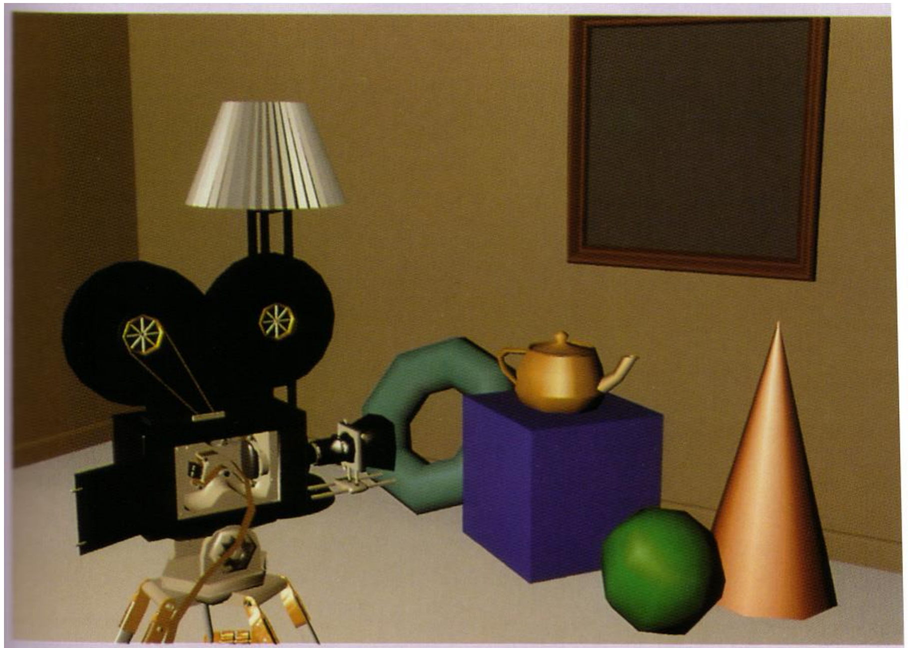


**Plate II.31** *Shutterbug.* Gouraud shaded polygons with specular reflection (Sections 14.4.4 and 16.2.5). (Copyright © 1990, Pixar. Rendered by Thomas Williams and H.B. Siegel using Pixar's PhotoRealistic RenderMan™ software.)

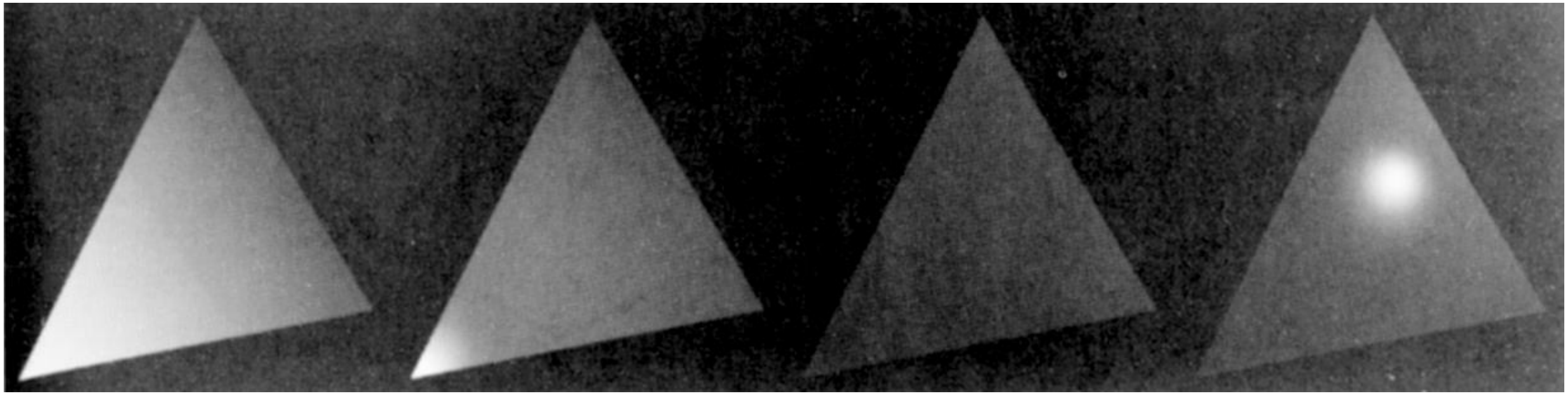
[Foley et al.]



Based on slides by Steve Marschner

# Phong shading

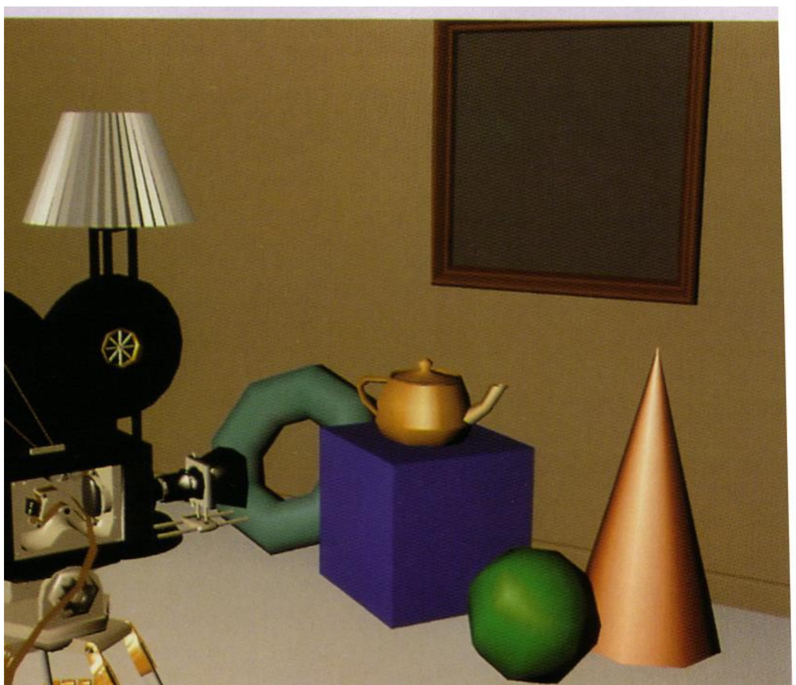
- Get higher quality by interpolating the normal
  - just as easy as interpolating the color
  - but now we are evaluating the illumination model per pixel rather than per vertex (and normalizing the normal first)
  - in pipeline, this means we are moving illumination from the vertex processing stage to the fragment processing stage



[Foley et al.]

# Phong shading

- Bottom line: produces much better highlights



tterbug. Gouraud shaded polygons with specular reflection (Sections 14.4.4
yright © 1990, Pixar. Rendered by Thomas Williams and H.B. Siegel using
listic RenderMan™ software.)

**Plate II.32** *Shutterbug.* Phong shaded polygons with specular reflection (Sections 14.4.4 and 16.2.5). (Copyright © 1990, Pixar. Rendered by Thomas Williams and H.B. Siegel using Pixar's PhotoRealistic RenderMan™ software.)

[Foley et al.]

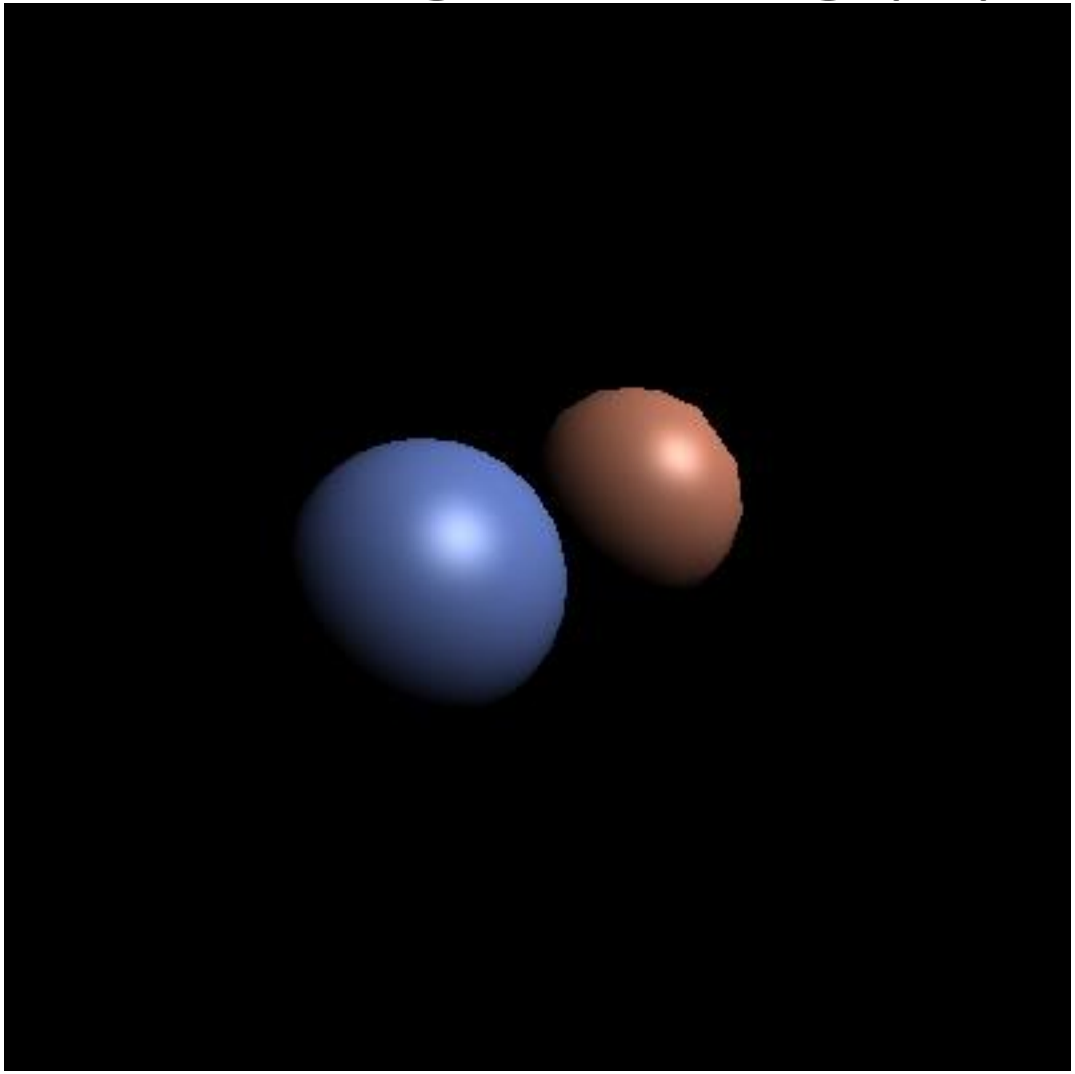
# Pipeline for Phong shading

- Vertex stage (input: position, color, and normal / vtx)
  - transform position and normal (object to eye space)
  - transform position (eye to screen space)
  - pass through color
- Rasterizer
  - interpolated parameters: $z'$ (screen $z$);
    $r, g, b$ color; $x, y, z$ **normal**
- Fragment stage (output: color, $z'$)
  - compute shading using interpolated color and normal
  - write to color planes only if interpolated $z' <$ current $z'$

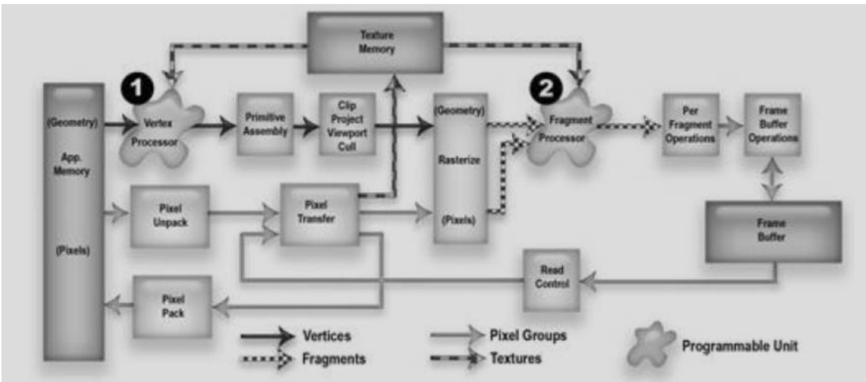


Based on slides by Steve Marschner

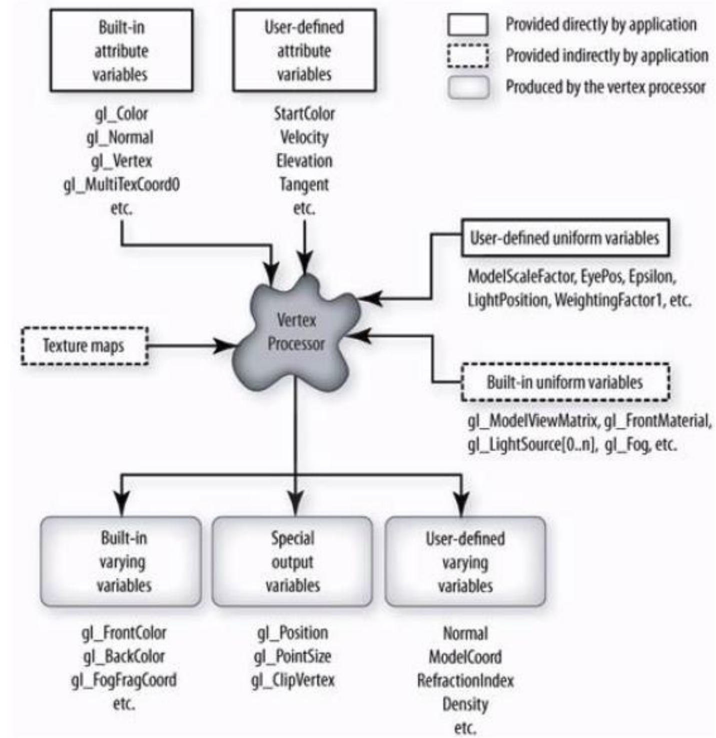
# Result of Phong shading pipeline

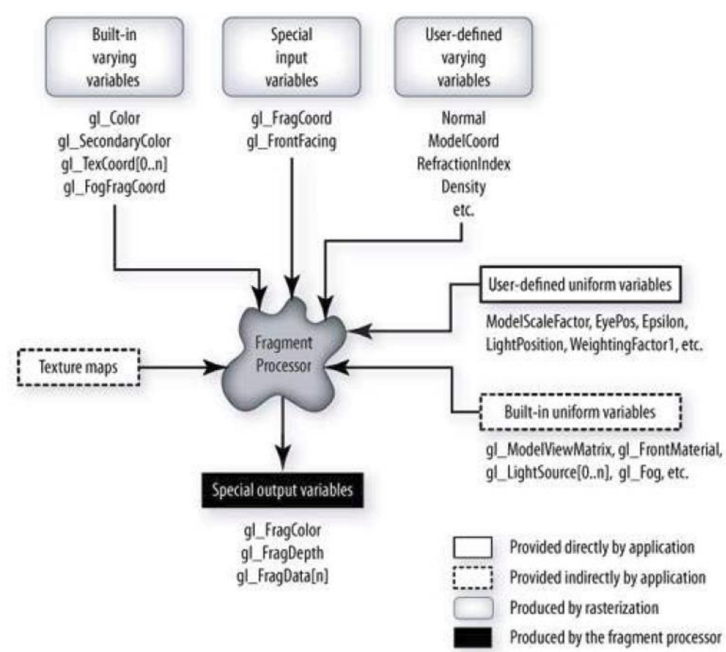


# OpenGL Programmable Processors




- Shader is a program that runs on the GPU
- Shading language is appropriate for implementing both common and complex graphics calculations
- ***Vertex shaders*** work at the vertex level
- ***Fragment shaders*** work at the pixel level
- Geometry and tessellation shaders also exist
- Intro in GLSL Orange Book 2.3 and Shirley book 18.3



# GLSL Data Types (Orange Book 3.2)

- Scalars: float, int (16 bits+), bool
  - Watch out as there is strict type matching, and no automatic type promotion

```
float x = 1; // this produces an error
float x = 1.0; // this is correct
```

- Vectors: vec2, vec3, vec4 (ivec and bvec too)
  - Components xyzw, rgba, or stpq
  - Easy access with swizzling: a contrived example…

```
vec4 pos = vec4( 1.0, 2.0, 3.0, 4.0);
vec4 dup = pos.xxyy; // dup = (1,1,2,2)
pos.sq = dup.bg; // pos = (2,1,2,1)
```

- Matrices: mat2, mat3, mat4 (floating point)

```
vec4 v1, v2; mat4 m;
v2 =  v1 * v2; // component-wise multiply
v2 = m*v1; // matrix vector multiply
```



Based on slides by Steve Marschner

# GLSL Data Types

- Samplers, for accessing textures
  - sampler1D, sampler2D, sampler3D, samplerCube, sampler1DShadow, sampler2DShadow
  - Assigned to a texture unit, with texture units set up on the application side

```
uniform sampler2D img;
vec2 coord = vec2( 0.5, 0.5 );
vec4 color = texture2D( img, coord );
```

- Arrays, of any type you like, but fixed size!
  - If a shader uses a non-constant variable to index the array, that shader must explicitly declare the array with the desired size.
- Structures can also be declared

```
struct light {
   vec3 position;
   vec3 color;
};
```

# GLSL Functions and syntax

- Similar to C++, but some important differences
- Functions have const, in, out, and inout parameters

```
vec3 f( const vec3 a, vec3 b, out vec3 c, inout vec3 d)
// a can only be read
// b can be read and written, but caller's variable not modified
// c can be read but is undefined, should be set before return
// d can be read, and should be written before return
// the function also returns a vec3
```

- Many useful built in functions (see Orange Book Ch. 5)

  radians, degrees, sin, cos, tan, asin, acos, atan…
  pow, exp, log, epx2, log2, sqrt, inversesqrt…
  abs, sign, floor, ceil, mod, min, max, clamp, mix…
  length distance, dot, cross, normalize…
  Fragment input argument derivatives: dFdx, dFdy
  noise



Based on slides by Steve Marschner

# GLSL, special variables

- Attributes
  - Per vertex data, such as position, normal, texture coordinates, skinning weights
- Uniforms
  - Data that doesn’t change during shader execution
  - But can change between drawing sets of primitives
    - Light positions
    - Material properties
    - Transformation matrices
- Varying
  - Data passed from vertex to fragment shaders
  - Fragment shader receives interpolated Per vertex data



Based on slides by Steve Marschner

# GLSL Built-in Uniform Variables

- OpenGL shading language provides access to many useful built-in uniform variables and constants (Orange Book Ch. 4.3)

  gl_ModelViewMatrix
  gl_ProjectionMatrix
  gl_ModelViewProjectionMatrix
  gl_NormalMatrix
  gl_FrontMaterial
  gl_LightSource[gl_MaxLights]

- OpenGL **ES**, common on phones, webGL, and ***embedded systems*** is a minimal implementation
  - Generaly discards much of the classic OpenGL pipeline
  - Few built-in uniform variables and constants



Based on slides by Steve Marschner

# Setting up GLSL in OpenGL / JOGL

- Load code, add to program, and compile shader
  - JOGL has ShaderCode and ShaderProgram helper classes
- Attach the program to a ShaderState helper class
- ShaderState.useProgram( gl, TRUE ) to enable
- Query uniforms with getUniformLocation
- Set uniforms with gl.glUniform* calls
- Query attributes with getAttributeLocation
- Per vertex data typically assembled into vertex attribute buffers and



Based on slides by Steve Marschner

# Shadertoy example

```
void mainImage( out vec4 fragColor, in vec2 fragCoord )
{
    vec4 backgroundColor = vec4(0.9, 0.99, 0.99, 1);
    float sphereSize = 0.7;
    vec4 sphereColor = vec4(0.5,0.4,0.3, 1);

    vec2 uv = gl_FragCoord.xy / iResolution.xy;
    vec2 p = uv * 2.0 - 1.0;
    p.x *= iResolution.x / iResolution.y;
    sphereColor[1] = 0.5*(sin(iGlobalTime*3.0)+1.0);

    vec4 color = backgroundColor;
    float r = sqrt(dot(p, p));
    if ( r < sphereSize ) {
        vec3 p3 = vec3( p.x, p.y, sqrt( sphereSize*sphereSize - r*r ) );
        //color = sphereColor;
        //color = sphereColor*texture2D( iChannel0, vec2(uv.x,1.0-uv.y)).rgba;
        // color = texture2D( iChannel0, vec2(uv.x,1.0-uv.y)).rgba;
    }
    fragColor = color;
}
```

# JOGL GLSL example

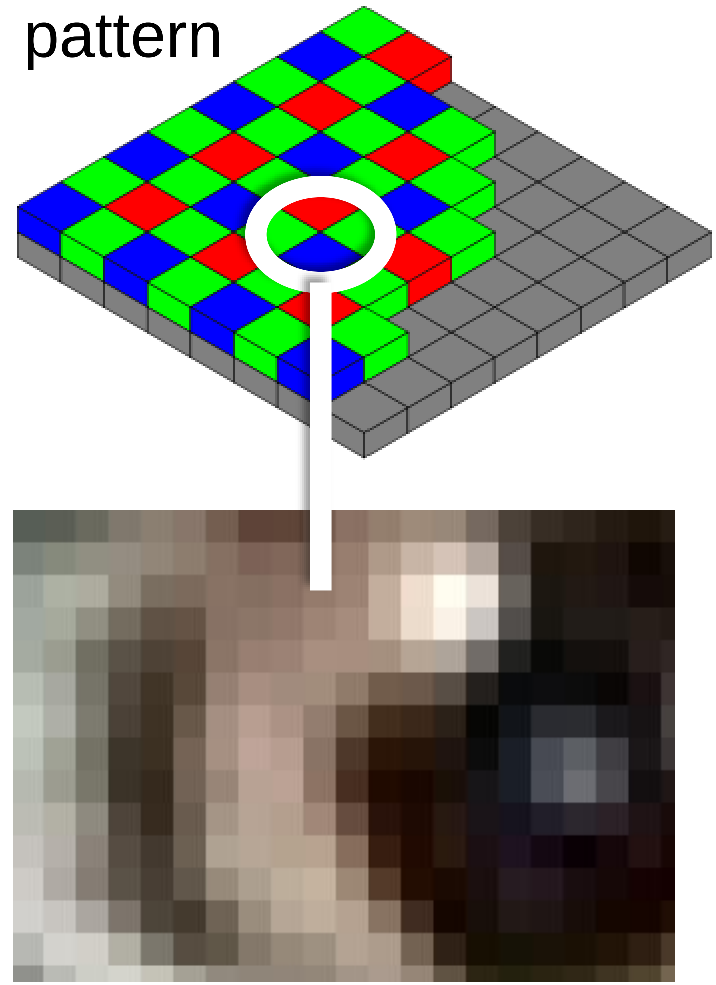
- Let’s run and modify the example code posted in the course content!

# Raster Images

- Lets briefly have a closer look at how we capture, store, and display images
- Material covered in Chapter 3 of text
  - We'll revisit gamma, colour, and transparency later in the term (Sections 3.2.2 to end of 3.4)

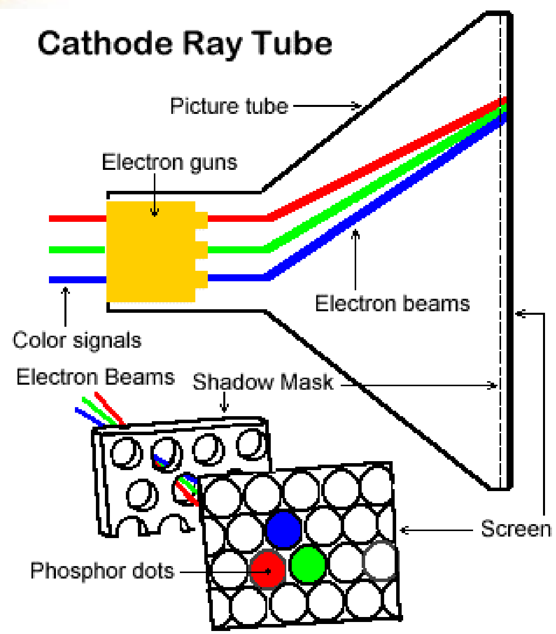
# Capturing Images

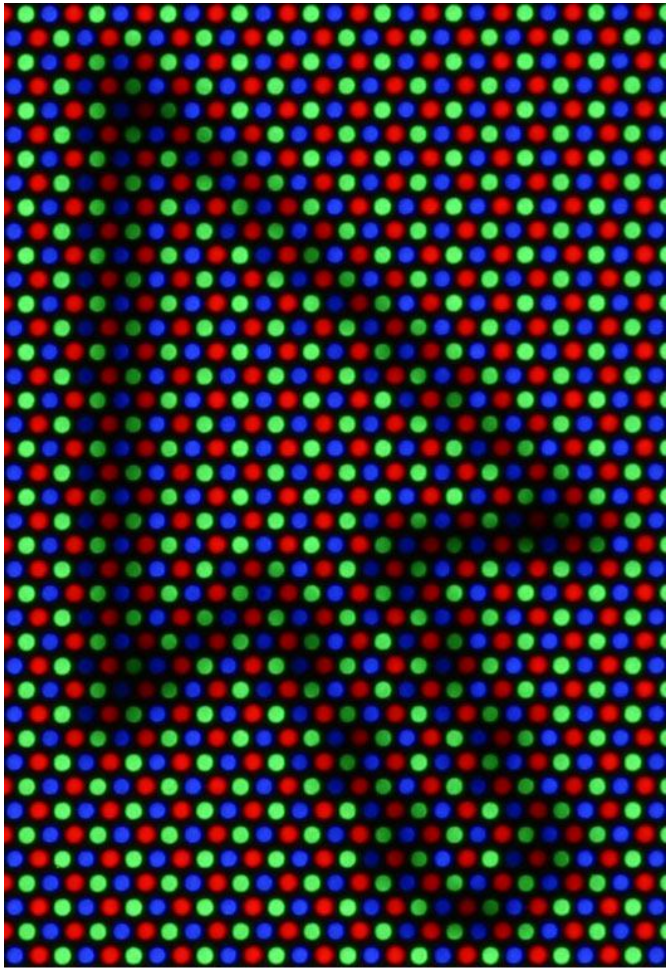

Bayer pattern



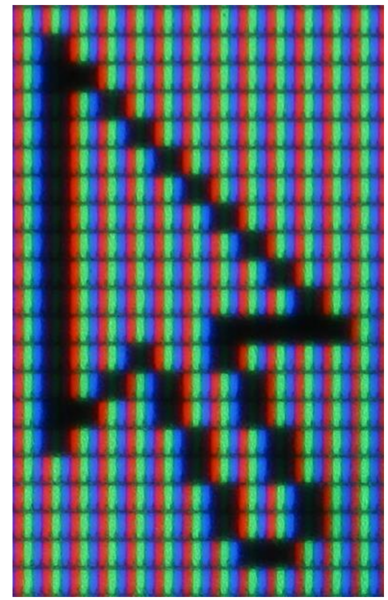
In a colour image, each picture element, or ***pixel***, can be represented with a red green and blue light intensity

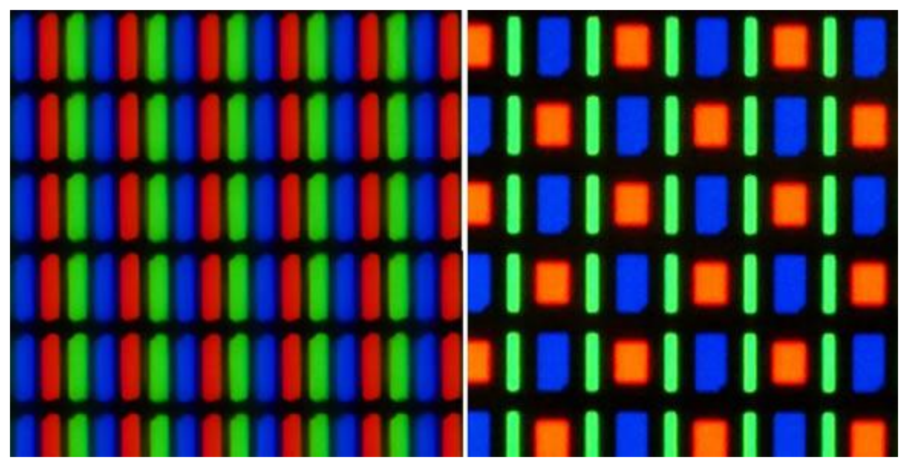
# How are Images Displayed?



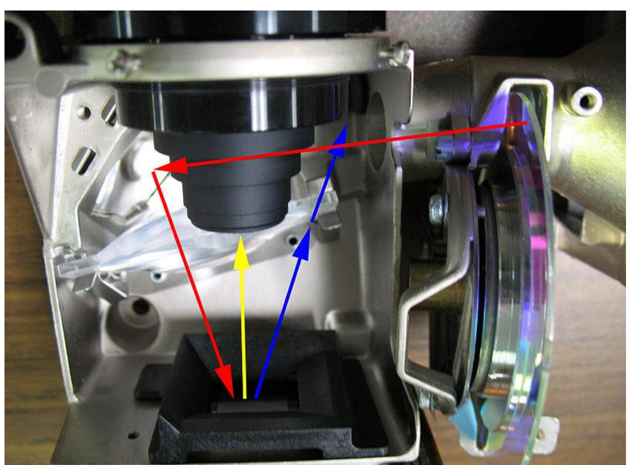



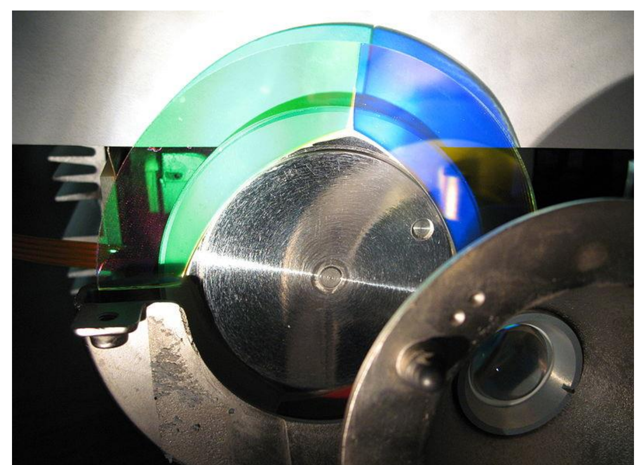
# How are Images Displayed?





# How are Images Displayed?




CATHODE-RAY OSCILLOGRAPH
TYPE 208-B
SERIAL NO
115/230 VOLTS
40-60 CYCLES
90 WATTS
F=FFB
207
1CF
Y POSITION
DOWN UP
X POSITION
LEFT RIGHT
INTENSITY
FOCUS
DUMONT
BEAM ON
POWER ON
BEAM CONTROLS

# Review and more information

- Chapter 8 the graphics pipeline
  - 8.2 pipeline variations
  - 8.4 culling
- Chapter 3 Raster images
  - 3.1 displays
  - 3.2 images, pixels and geometry
  - Could also review 3.3 on RBG colour, and 3.4 on alpha composoting, but we'll see more later