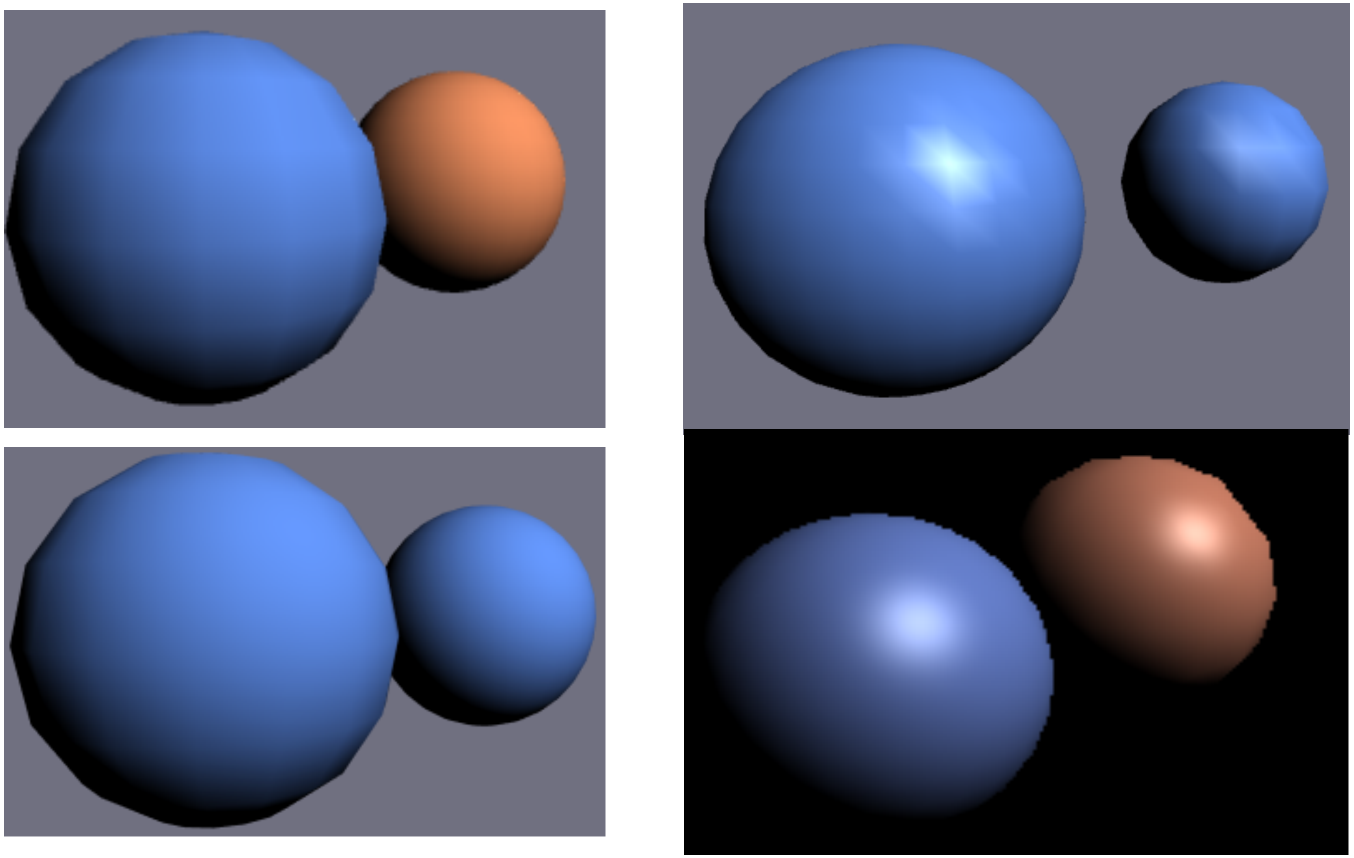


Computer Graphics

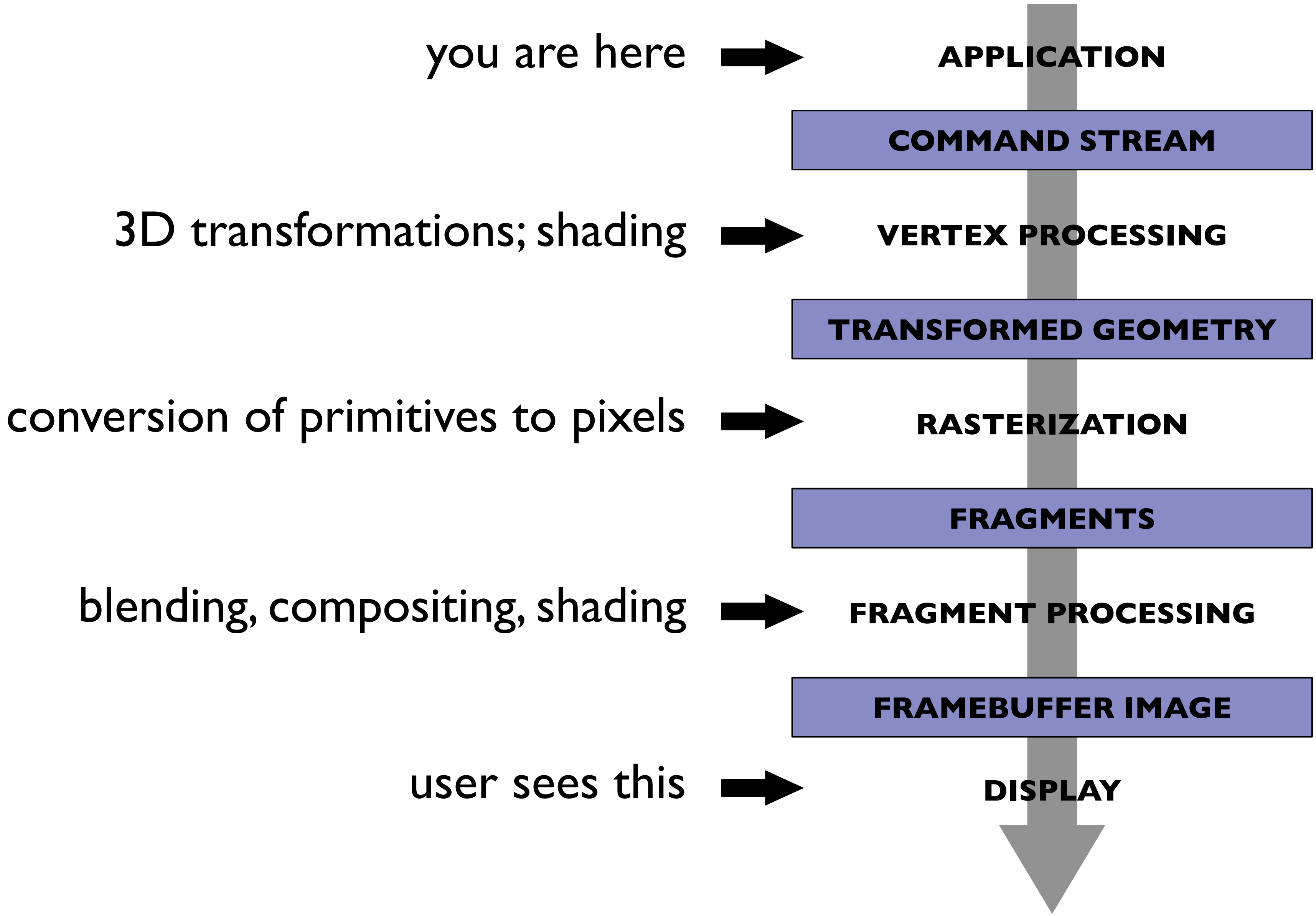
Lecture 23

Shading in the Graphics Pipeline

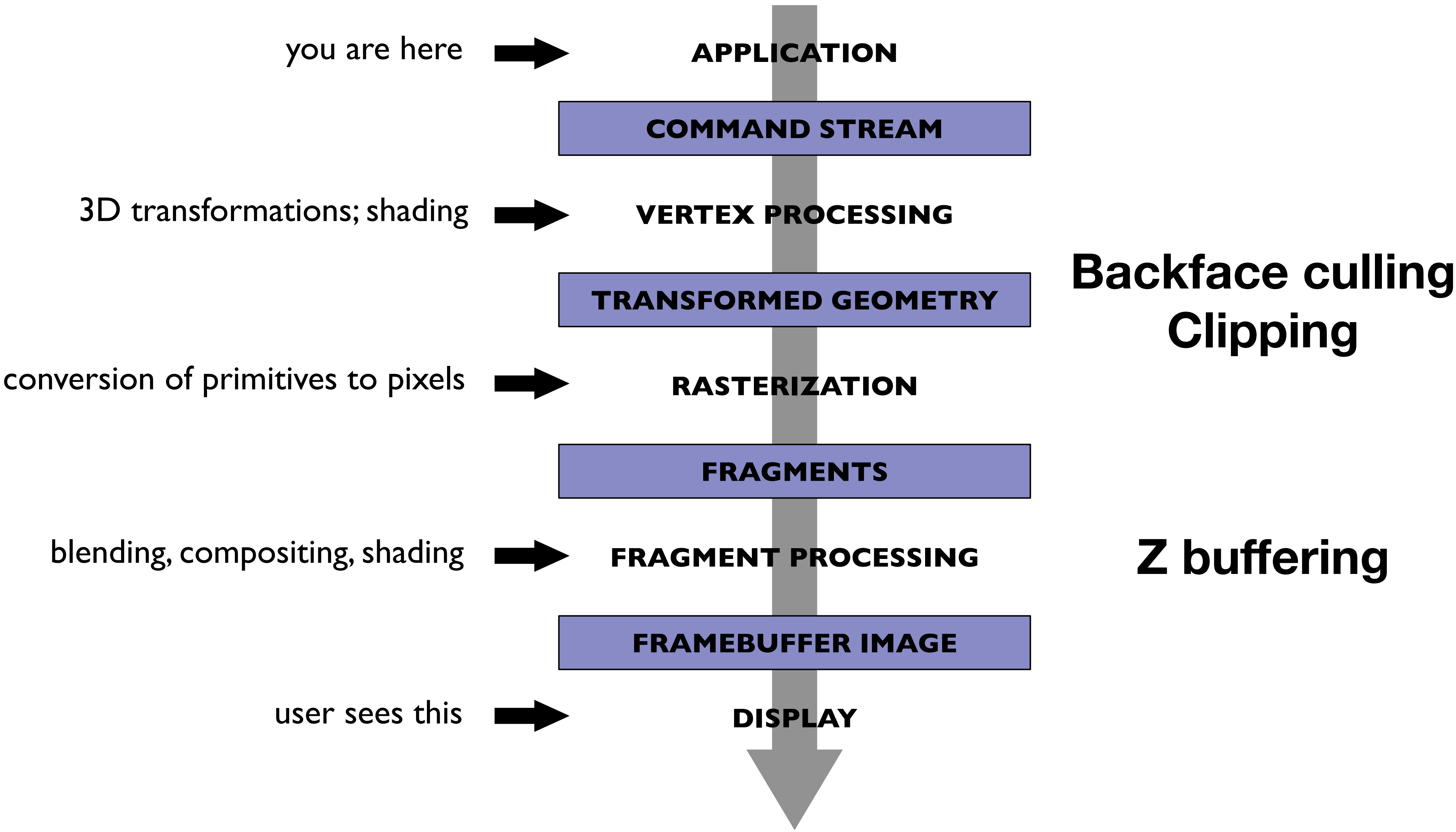
Announcements

- Mini-lab tomorrow on OpenGL: bring a laptop if you can
- Anyone still looking for a final project group?

Graphics Pipeline: Overview

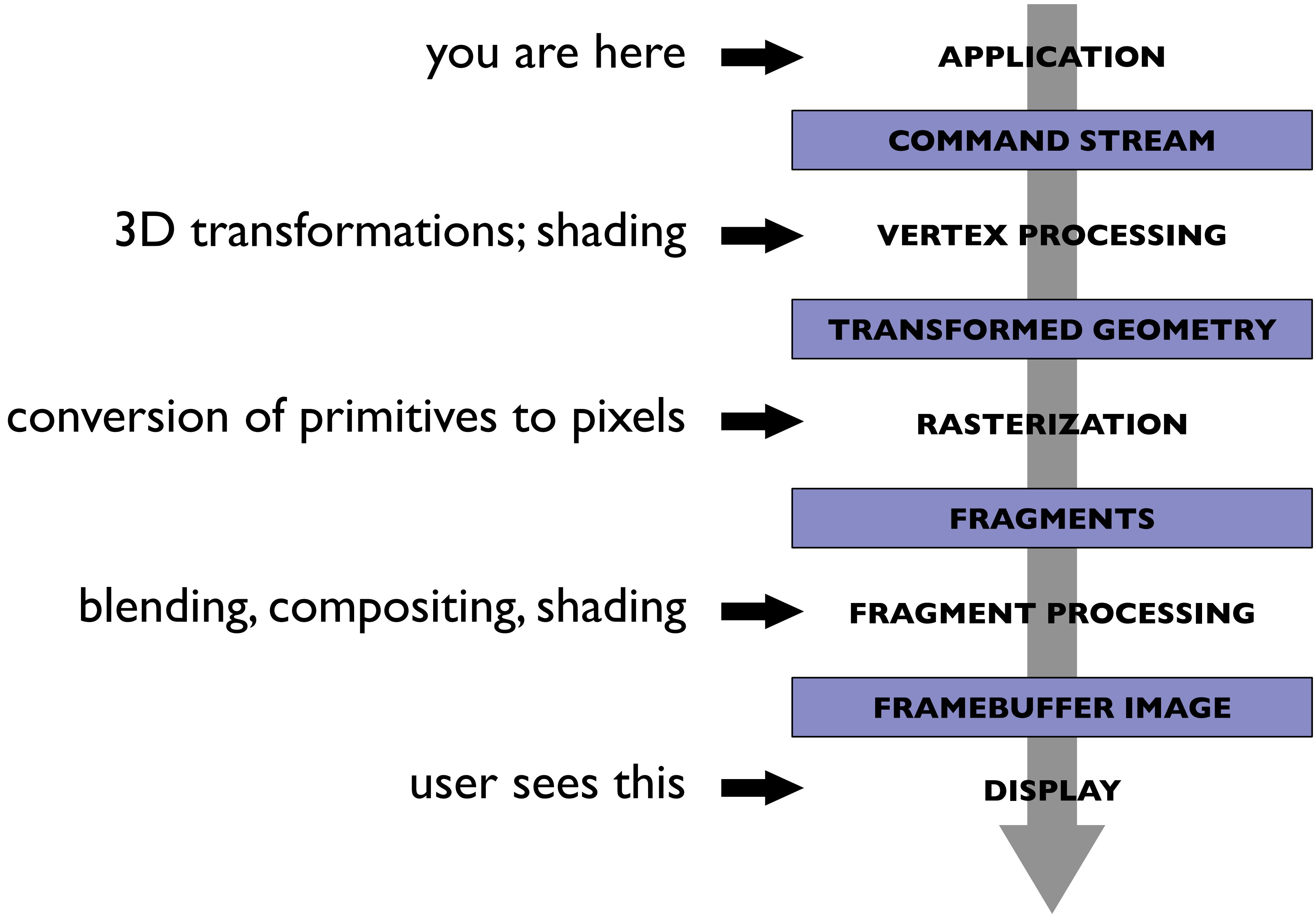


Last time



you are here
APPLICATION
COMMAND STREAM
3D transformations; shading
VERTEX PROCESSING
Backface culling
Clipping
TRANSFORMED GEOMETRY
conversion of primitives to pixels
RASTERIZATION
FRAGMENTS
blending, compositing, shading
FRAGMENT PROCESSING
Z buffering
FRAMEBUFFER IMAGE
user sees this
DISPLAY

Graphics Pipeline: Overview



OpenGL: One implementation of the graphics pipeline.

OpenGL: One implementation of the graphics pipeline.

And now: a highly abridged and only somewhat accurate history of OpenGL.

OpenGL: The Bad Old Days

- OpenGL was (still is) a state machine.
- Basic usage:

1. Set flags for shading mode
2. Set model, view, and projection matrices
3. Set GL to triangle mode
4. Send vertices to GPU one at a time.
5. Call draw function to draw to the screen.

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();

glBegin(GL_TRIANGLES);
glVertex2f( -0.5f, -0.5f );
glVertex2f( 0.5f, -0.5f );
glVertex2f( 0.5f, 0.5f );
glEnd();
```

OpenGL: Nowadays

OpenGL: Nowadays

- Send buffers full of data to GPU up front.

OpenGL: Nowadays

- Send buffers full of data to GPU up front.
- Tell GL how to interpret them (triangles, line segments, ...)

OpenGL: Nowadays

- Send buffers full of data to GPU up front.
- Tell GL how to interpret them (triangles, line segments, ...)
- GL executes custom-written **vertex shader** program on each vertex (to determine its location in **clip space**)

OpenGL: Nowadays

- Send buffers full of data to GPU up front.
- Tell GL how to interpret them (triangles, line segments, ...)
- GL executes custom-written **vertex shader** program on each vertex (to determine its location in **clip space**) *= normalized device coordinates*

OpenGL: Nowadays

- Send buffers full of data to GPU up front.
- Tell GL how to interpret them (triangles, line segments, ...)
- GL executes custom-written **vertex shader** program on each vertex (to determine its location in **clip space**) *= normalized device coordinates*
- GL **rasterizes** primitives into pixel-shaped **fragments**

OpenGL: Nowadays

- Send buffers full of data to GPU up front.
- Tell GL how to interpret them (triangles, line segments, ...)
- GL executes custom-written **vertex shader** program on each vertex (to determine its location in **clip space**) *= normalized device coordinates*
- GL **rasterizes** primitives into pixel-shaped **fragments**
- GL executes custom-written **fragment shader** program on each fragment to determine its color.

OpenGL: Nowadays

- Send buffers full of data to GPU up front.
- Tell GL how to interpret them (triangles, line segments, ...)
- GL executes custom-written **vertex shader** program on each vertex (to determine its location in **clip space**) *= normalized device coordinates*
- GL **rasterizes** primitives into pixel-shaped **fragments**
- GL executes custom-written **fragment shader** program on each fragment to determine its color.
- GL writes fragment colors to framebuffer pixels; neat things appear on your screen.

OpenGL: Your job, conceptually

- Send buffers full of data to GPU up front.
- Tell GL how to interpret them (triangles, ...)
- GL executes custom-written **vertex shader** program on each vertex (to determine its location in **clip space**) *= normalized device coordinates*
- GL **rasterizes** primitives into pixel-shaped **fragments**
- GL executes custom-written **fragment shader** program on each fragment to determine its color.
- GL writes fragment colors to framebuffer pixels; neat things appear on your screen.

OpenGL: Your job, conceptually

(send geometry)

- Send buffers full of data to GPU up front.
- Tell GL how to interpret them (triangles, ...)
- GL executes custom-written **vertex shader** program on each vertex (to determine its location in **clip space**) *= normalized device coordinates*
- GL **rasterizes** primitives into pixel-shaped **fragments**
- GL executes custom-written **fragment shader** program on each fragment to determine its color.
- GL writes fragment colors to framebuffer pixels; neat things appear on your screen.

OpenGL: Your job, conceptually

(send geometry)

- Send buffers full of data to GPU up front.
- Tell GL how to interpret them (triangles, ...)

(write vertex shader)

- GL executes custom-written **vertex shader** program on each vertex (to determine its location in **clip space**) *= normalized device coordinates*
- GL **rasterizes** primitives into pixel-shaped **fragments**
- GL executes custom-written **fragment shader** program on each fragment to determine its color.
- GL writes fragment colors to framebuffer pixels; neat things appear on your screen.

OpenGL: Your job, conceptually

(send geometry)

- Send buffers full of data to GPU up front.
- Tell GL how to interpret them (triangles, ...)

(write vertex shader)

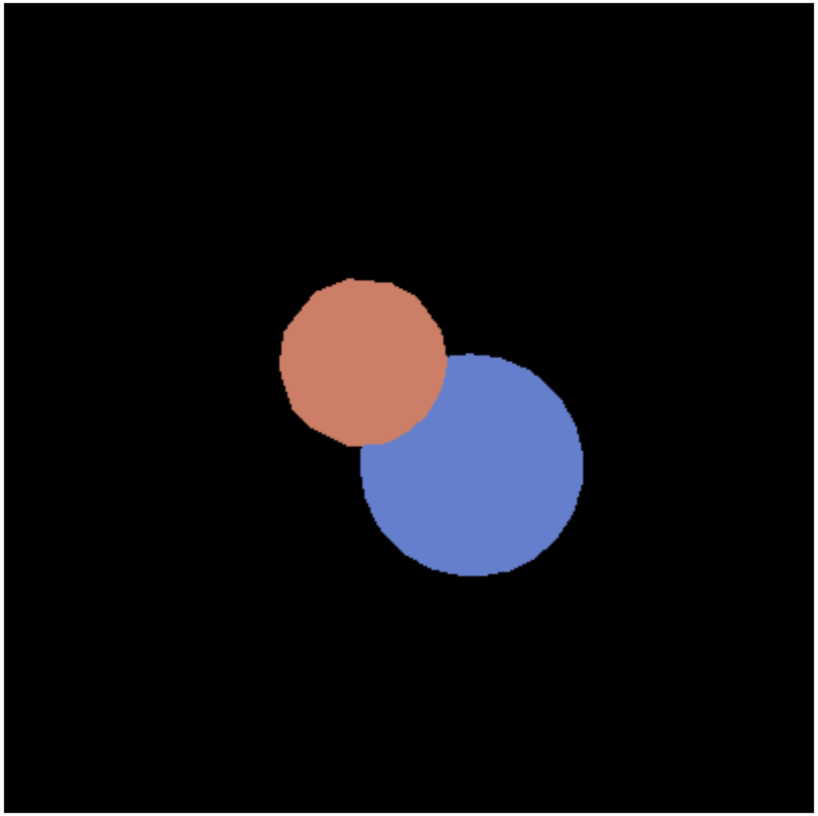
- GL executes custom-written **vertex shader** program on each vertex (to determine its location in **clip space**) *= normalized device coordinates*
- GL **rasterizes** primitives into pixel-shaped **fragments**

(write fragment shader)

- GL executes custom-written **fragment shader** program on each fragment to determine its color.
- GL writes fragment colors to framebuffer pixels; neat things appear on your screen.

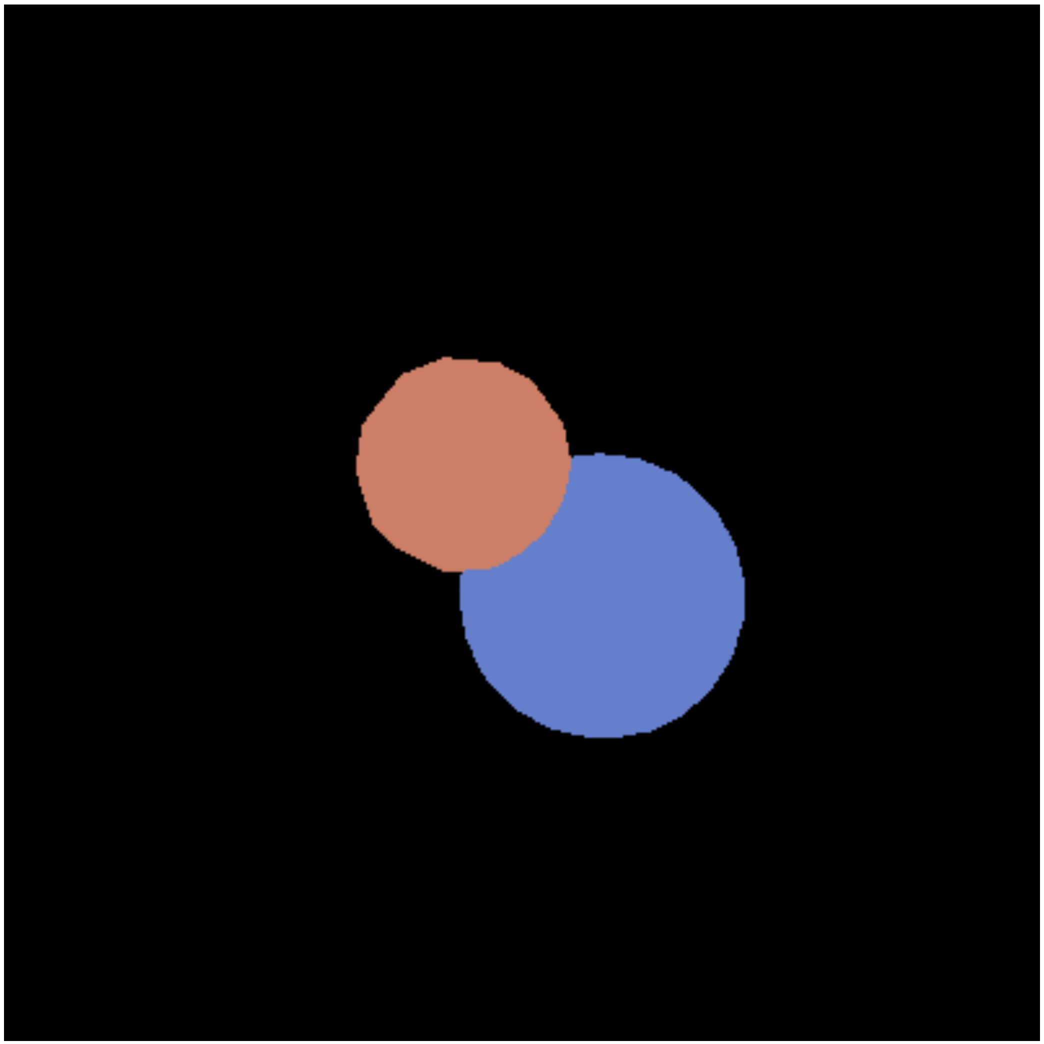
Pipeline for minimal operation

- 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

https://facultyweb.cs.wwu.edu/~wehrwes/courses/csci480_24f/pipeline_demo/



Rendering Realistic Images

Rendering Realistic Images

- We have a pipeline that gives us access to the compute power of shaders and does a bunch of nice things for us.

Rendering Realistic Images

- We have a pipeline that gives us access to the compute power of shaders and does a bunch of nice things for us.
- Tomorrow we'll learn how to get data in and out

Rendering Realistic Images

- We have a pipeline that gives us access to the compute power of shaders and does a bunch of nice things for us.
- Tomorrow we'll learn how to get data in and out
- How do we make realistic-looking images using reflection models like Lambertian and Blinn-Phong?

Rendering Realistic Images

- We have a pipeline that gives us access to the compute power of shaders and does a bunch of nice things for us.
- Tomorrow we'll learn how to get data in and out
- How do we make realistic-looking images using reflection models like Lambertian and Blinn-Phong?

but first, a rant about terminology

Phong shading Blinn-Phong shading in the fragment shader

Phong shading Blinn-Phong shading in the fragment shader

- Shade (v.): determine color of a pixel

Phong shading Blinn-Phong shading in the fragment shader

- Shade (v.): determine color of a pixel

basically all of computer graphics...

Phong shading Blinn-Phong shading in the fragment shader

- Shade (v.): determine color of a pixel

basically all of computer graphics...

- Shader (n.): a program that runs on GPU

Phong shading Blinn-Phong shading in the fragment shader

- Shade (v.): determine color of a pixel
  basically all of computer graphics...
- Shader (n.): a program that runs on GPU
  vertex shader, fragment shader

Phong shading Blinn-Phong shading in the fragment shader

- Shade (v.): determine color of a pixel

  basically all of computer graphics...

- Shader (n.): a program that runs on GPU

  vertex shader, fragment shader

- Shading model (**reflection** or **illumination model**): light interaction model that determines a pixel's color

Phong shading Blinn-Phong shading in the fragment shader

- Shade (v.): determine color of a pixel
  basically all of computer graphics...
- Shader (n.): a program that runs on GPU
  vertex shader, fragment shader
- Shading model (**reflection** or **illumination model**): light interaction model that determines a pixel's color
  Lambertian reflection, Blinn-Phong reflection

Phong shading Blinn-Phong shading in the fragment shader

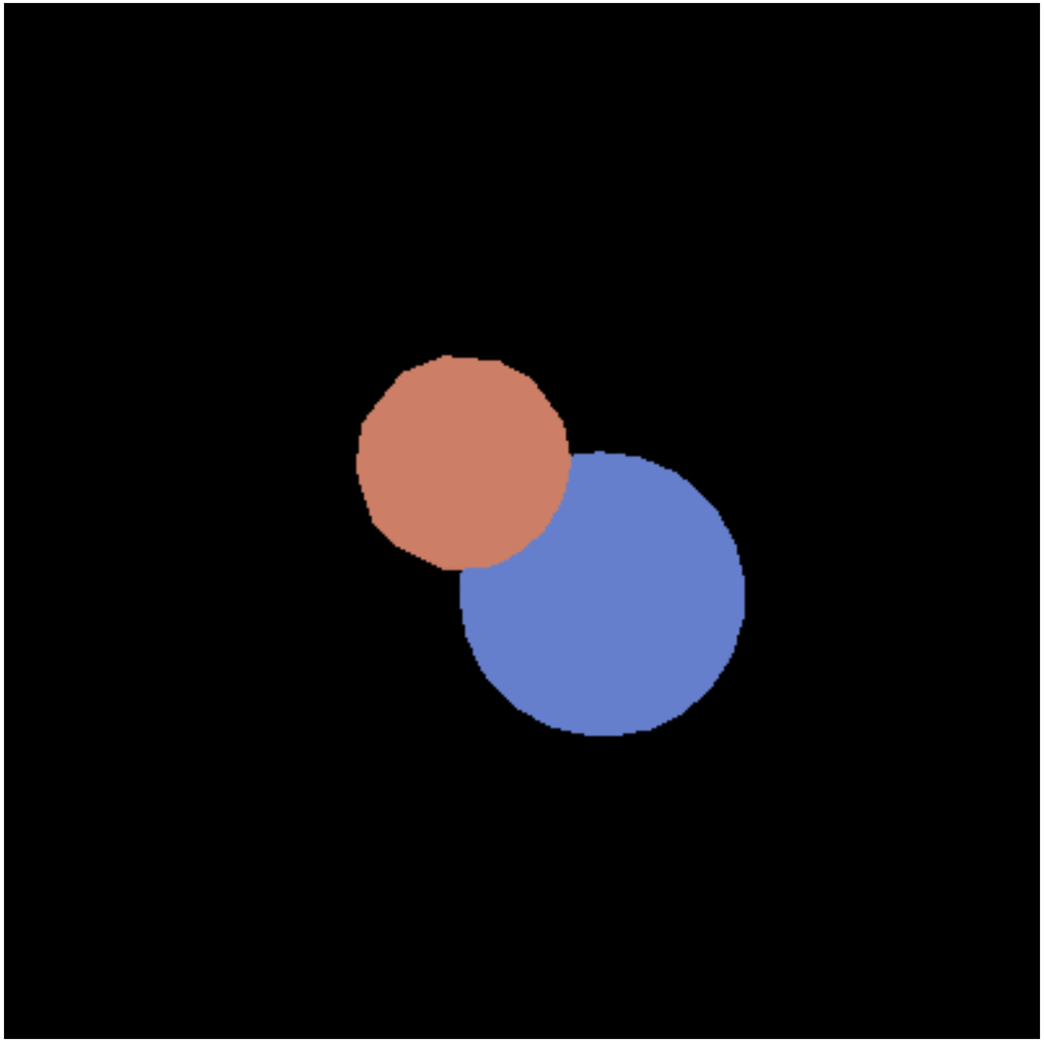
- Shade (v.): determine color of a pixel
  basically all of computer graphics...
- Shader (n.): a program that runs on GPU
  vertex shader, fragment shader
- Shading model (**reflection** or **illumination model**): light interaction model that determines a pixel's color
  Lambertian reflection, Blinn-Phong reflection
- Shading algorithm (**interpolation technique)**: when, and in which shader, is the reflection model computed, and using what normals?

Phong shading Blinn-Phong shading in the fragment shader

- Shade (v.): determine color of a pixel
  basically all of computer graphics...
- Shader (n.): a program that runs on GPU
  vertex shader, fragment shader
- Shading model (**reflection** or **illumination model**): light interaction model that determines a pixel's color
  Lambertian reflection, Blinn-Phong reflection
- Shading algorithm (**interpolation technique)**: when, and in which shader, is the reflection model computed, and using what normals?
  flat shading, Gouraud shading, Phong shading

Let's call this "not shading"

https://facultyweb.cs.wwu.edu/~wehrwes/courses/csci480_24f/pipeline_demo/



 •

Flat shading (interpolation)

- Shade using the real normal of the triangle
  - same result as ray tracing a bunch of triangles without normal interpolation
- Leads to constant shading and faceted appearance
  - truest view of the mesh geometry

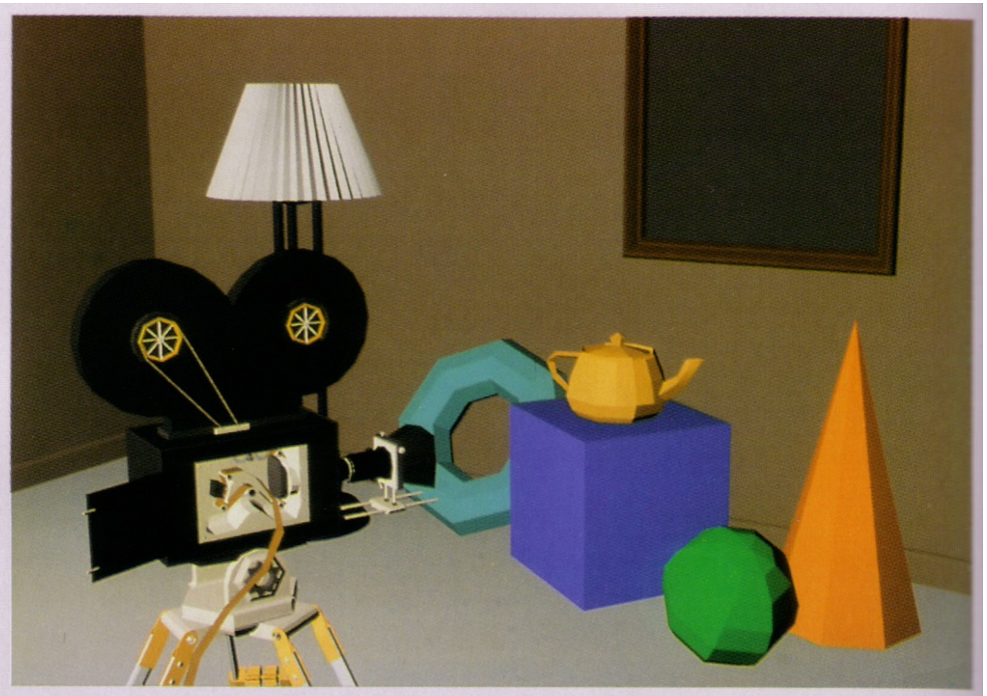


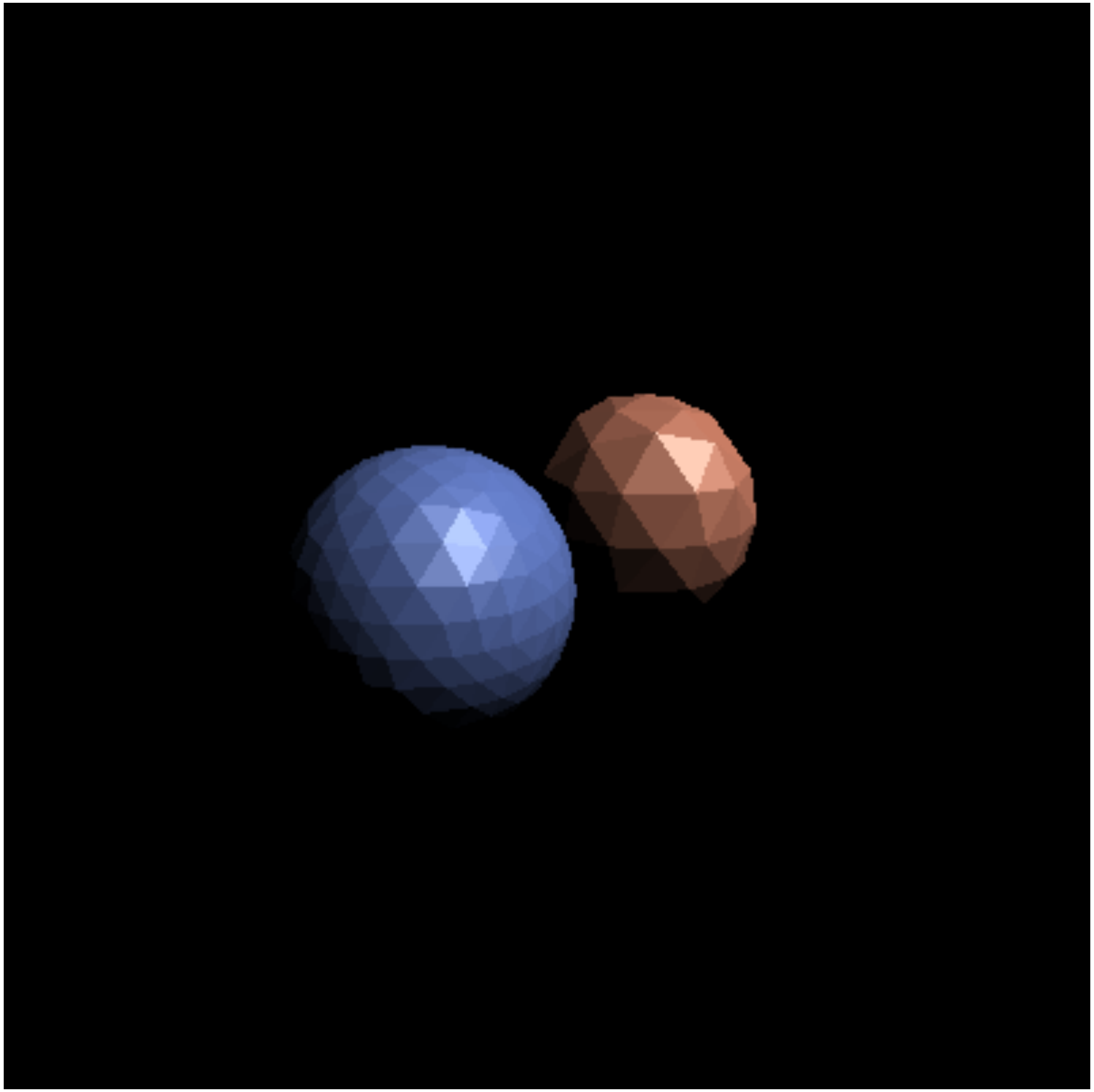
Plate II.29 *Shutterbug.* Individually shaded polygons with diffuse reflection (Sections 14.4.2 and 16.2.3). (Copyright © 1990, Pixar. Rendered by Thomas Williams and H.B. Siegel using Pixar's PhotoRealistic RenderMan™ software.)

[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



Summary: Shading and Interpolation Techniques

Pipeline Stage

	Vertex Shader	Rasterizer	Fragment Shader
Flat	p, n, l --> cam space color = $n \cdot \ell \cdot$ vtx_color p --> screen space	interp z	Write color if z buffer says so

Interpolation

Question: Why do we stop at camera space?

Summary: Shading and Interpolation Techniques

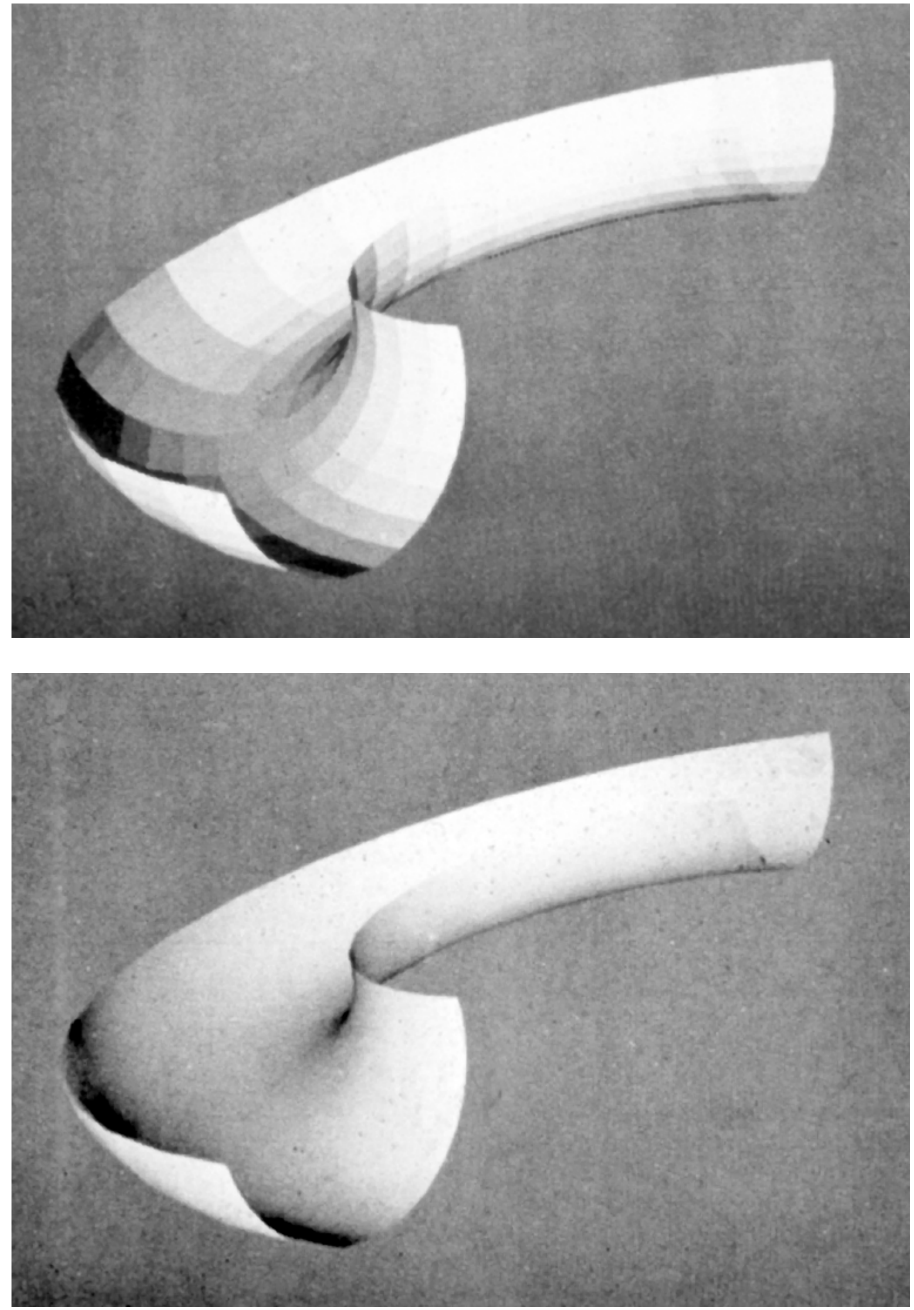
Pipeline Stage

Interpolation

	Vertex Shader	Rasterizer	Fragment Shader
Flat	p, n, l --> cam space color = $n \cdot \ell \cdot$ vtx_color p --> screen space		Write color if z buffer says so
Gouraud	p, n, l --> cam space color = $n \cdot \ell \cdot$ vtx_color p --> screen space		Write color if z buffer says so
Phong	p, n, l --> cam space p --> screen space pass through vtx_color		color = n * l * frag_color Write color if z buffer says so

Gouraud shading

- Often we're trying to draw smooth surfaces, so facets are an artifact
  - compute colors at vertices using vertex normals
  - interpolate colors across triangles
  - "Gouraud shading"
  - "Smooth shading"

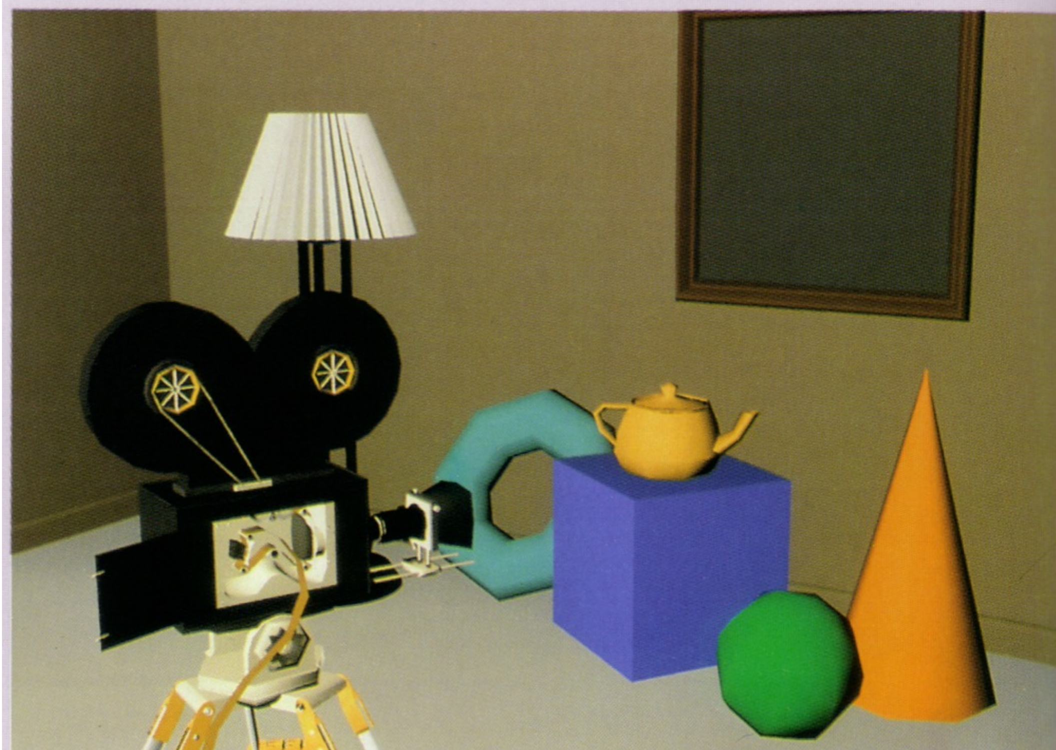


[Gouraud thesis]

Gouraud shading

- Often we're trying to draw smooth surfaces, so facets are an artifact
  - compute colors at vertices using vertex normals
  - interpolate colors across triangles
  - "Gouraud shading"
  - "Smooth shading"

Plate II.30 *Shutterbug.* Gouraud shaded polygons with diffuse reflection (Sections 14.4.3 and 16.2.4). (Copyright © 1990, Pixar. Rendered by Thomas Williams and H.B. Siegel using Pixar's PhotoRealistic RenderMan™ software.)

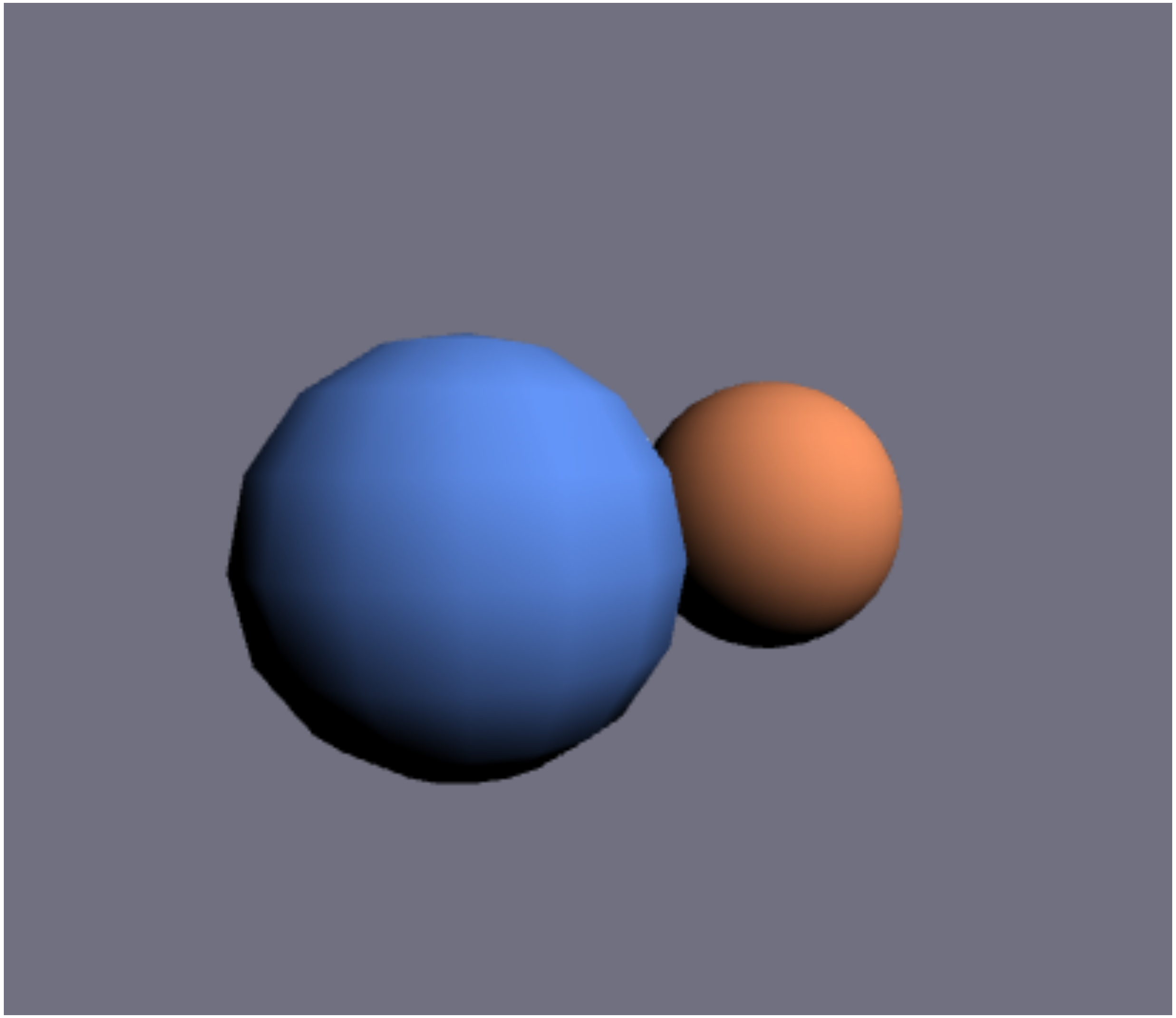


[Foley et al.]

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



Demo

Summary: Shading and Interpolation Techniques

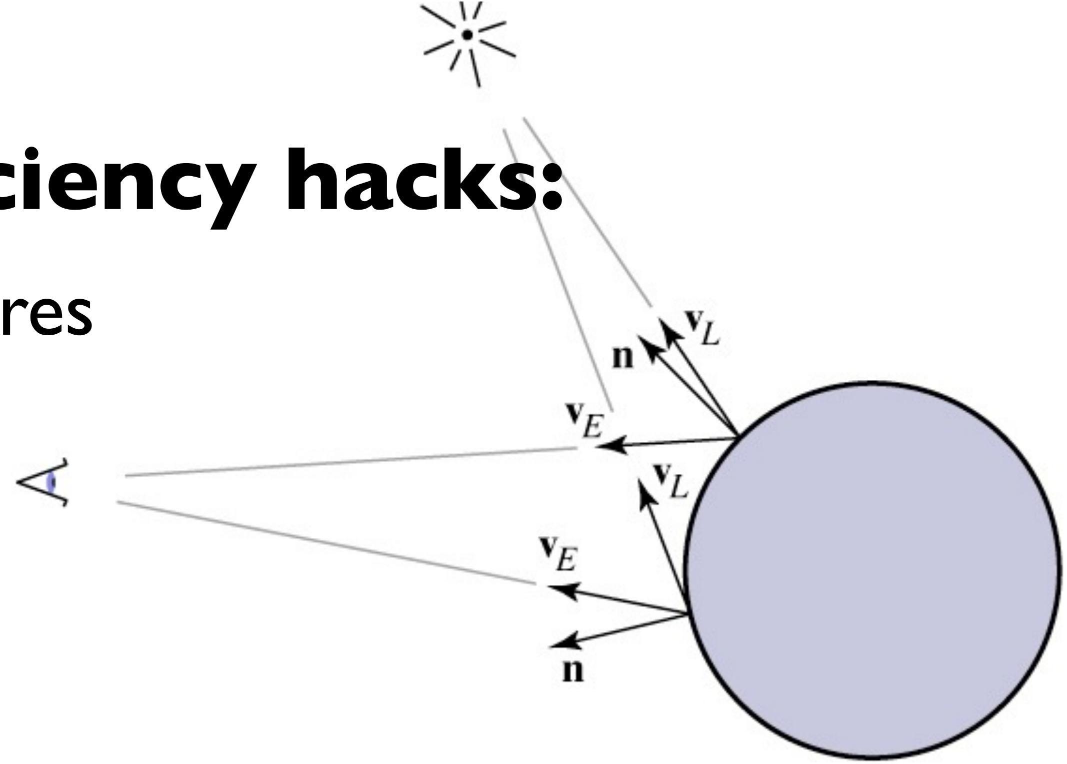
Pipeline Stage

Interpolation

	Vertex Shader	Rasterizer	Fragment Shader
Flat	p, n, l --> cam space color = $n \cdot \ell \cdot$ vtx_color p --> screen space	Interpolate z'	Write color if z buffer says so
Gouraud	p, n, l --> cam space color = $n \cdot \ell \cdot$ vtx_color p --> screen space		Write color if z buffer says so
Phong	p, n, l --> cam space p --> screen space pass through vtx_color		color = n * l * frag_color Write color if z buffer says so

Some possible efficiency hacks:

- Blinn-Phong model requires knowing
  - normal
  - light direction
  - view direction

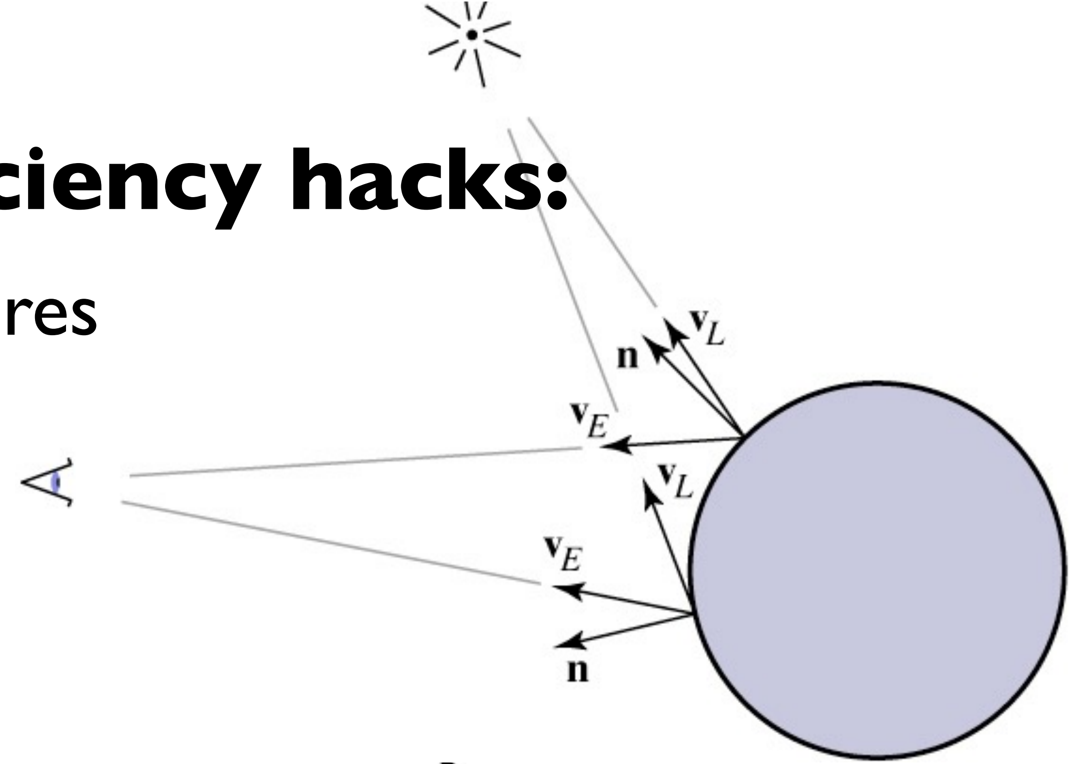


- Hack: use directional lights so ℓ doesn't change

- Hack: pretend viewer is infinitely distant so view direction doesn't change either

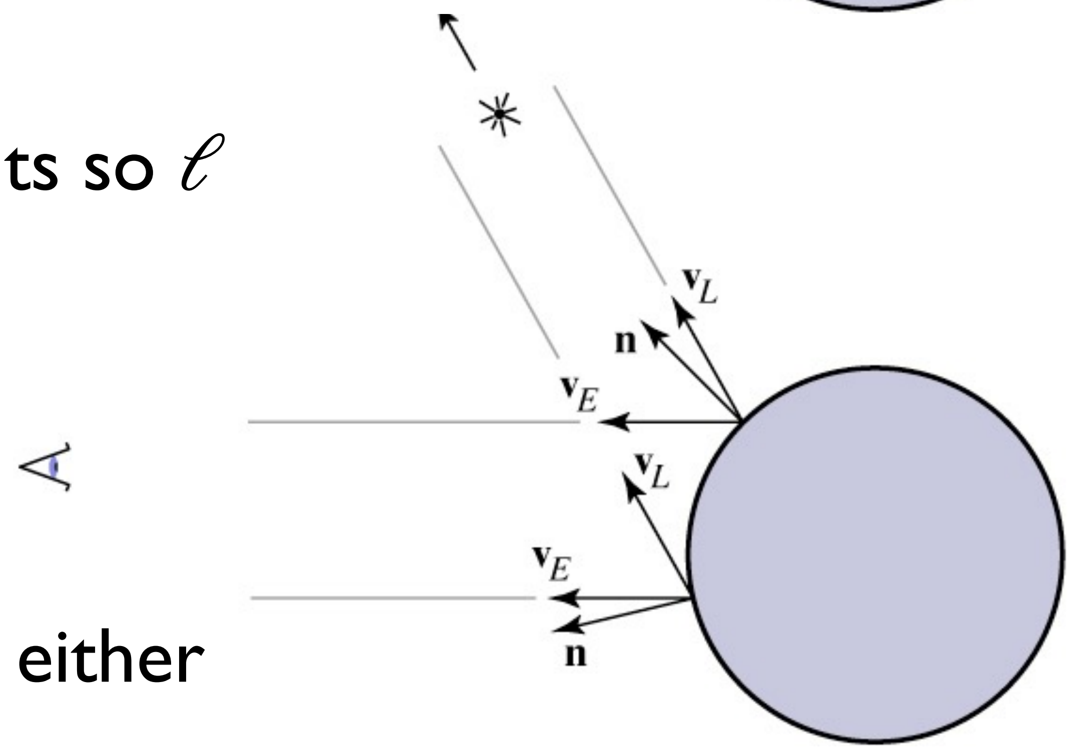
Some possible efficiency hacks:

- Blinn-Phong model requires knowing
  - normal
  - light direction
  - view direction



- Hack: use directional lights so ℓ doesn't change

- Hack: pretend viewer is infinitely distant so view direction doesn't change either



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
  - (demo)

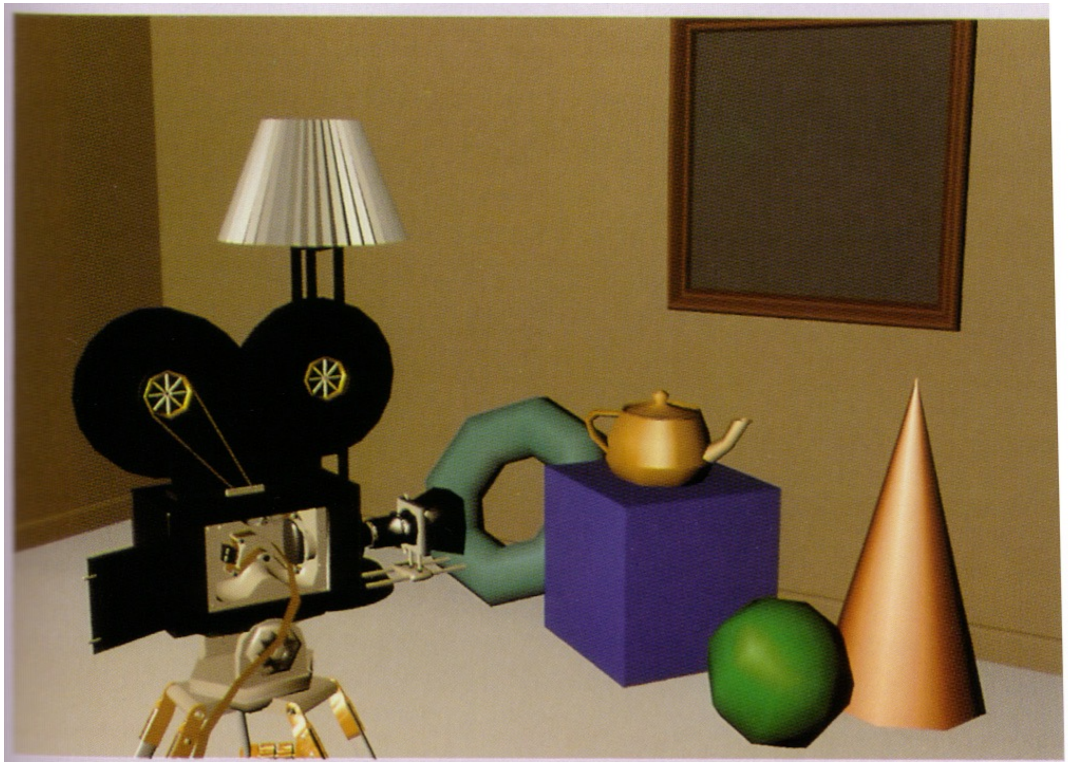


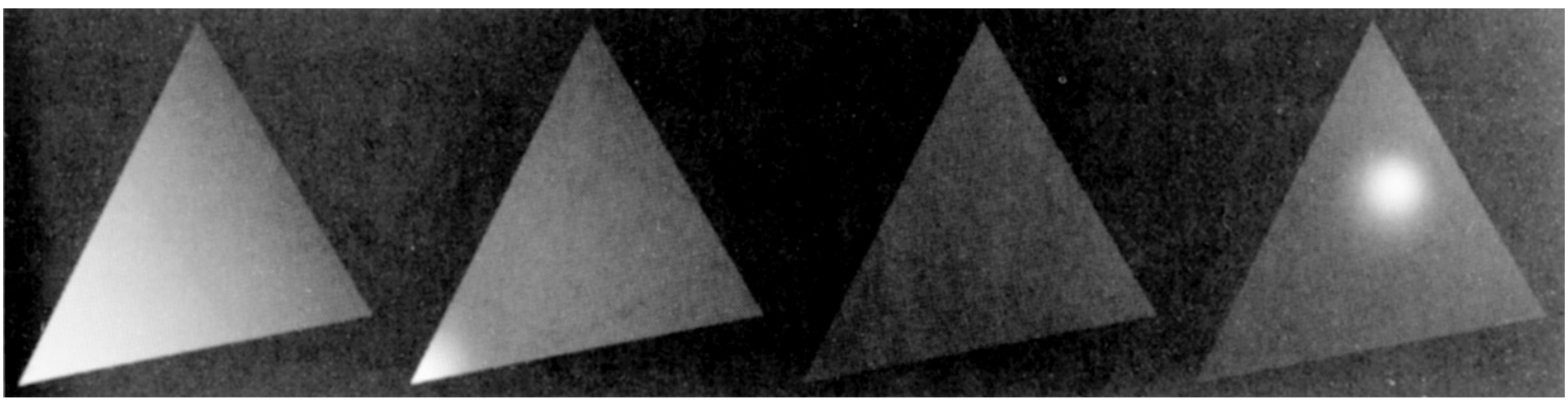
Plate II.31 *Shutterbug.* Gouraud shaded polygons with specular reflection (Sections 14.4.4 and 16.2.5). (Copyright © 1990, Pixar. Rendered by Thomas Williams and H.B. Siegel using Pixar's PhotoRealistic RenderMan™ software.)

[Foley et al.]

(*not the same thing as Blinn-Phong reflection)

Per-pixel (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



Per-pixel (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.)

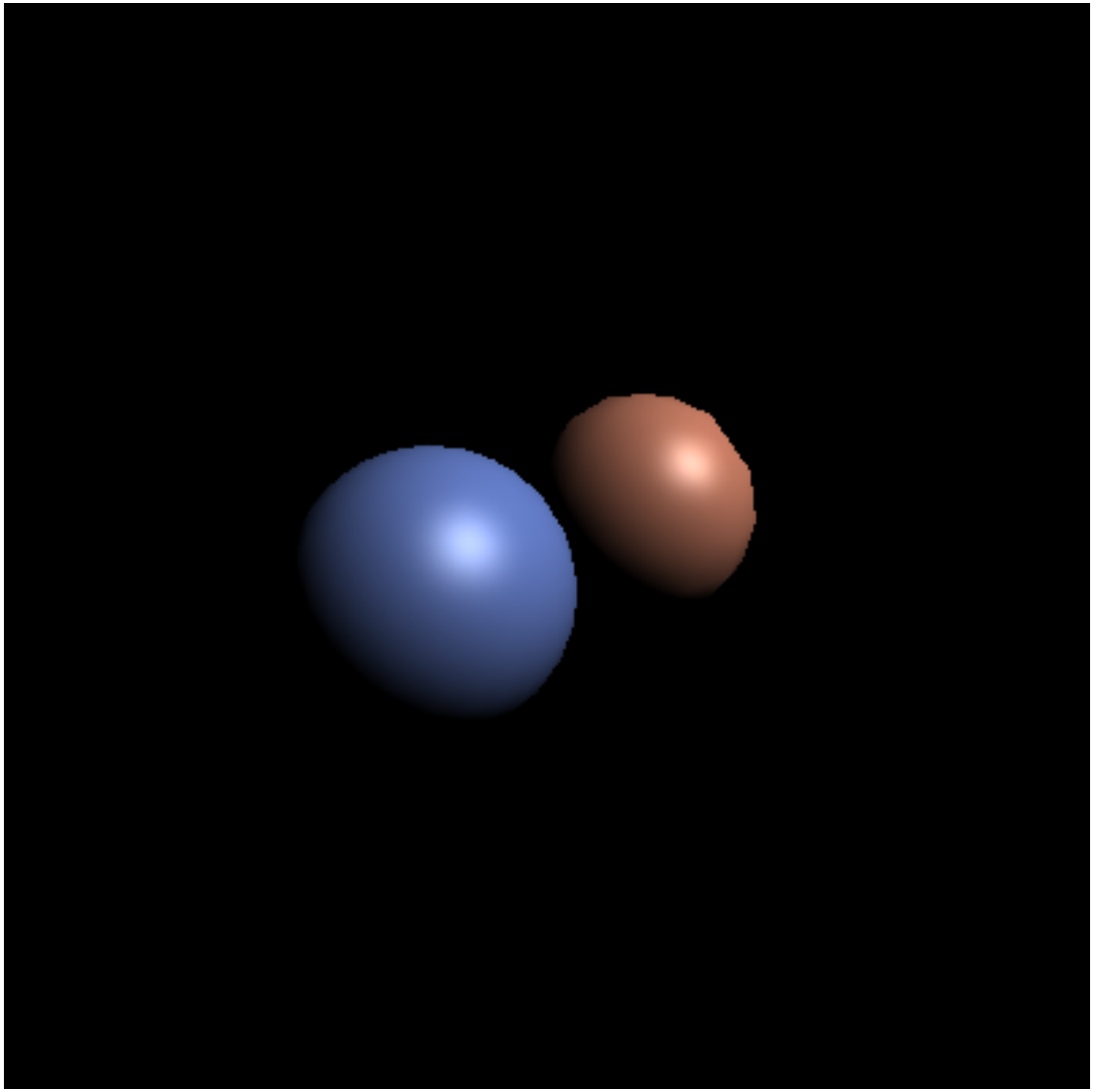
Plate II.32 *Shutterbug.* Phong shaded polygons with specular reflection (Sections 14.4.4 and 16.2.5). (Copyright © 1990, Pixar. Rendered by Thomas Williams and H.B. Siegel using Pixar's PhotoRealistic RenderMan™ software.)

[Foley et al.]

Pipeline for per-pixel 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'

Result of per-pixel shading pipeline



(demo)

Summary: Shading and Interpolation Techniques

Pipeline Stage

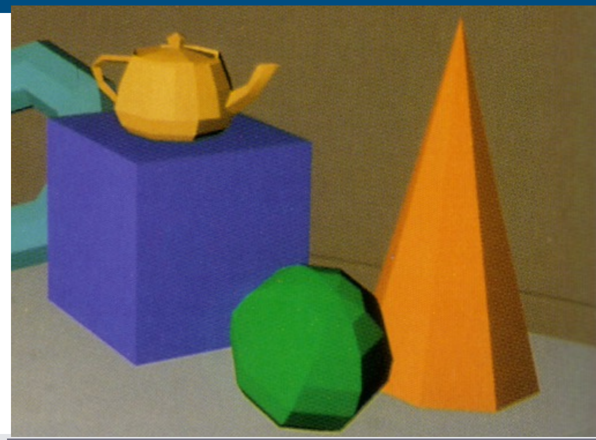
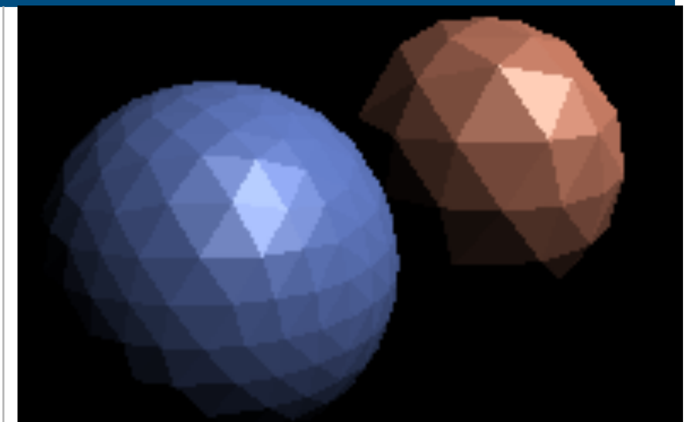
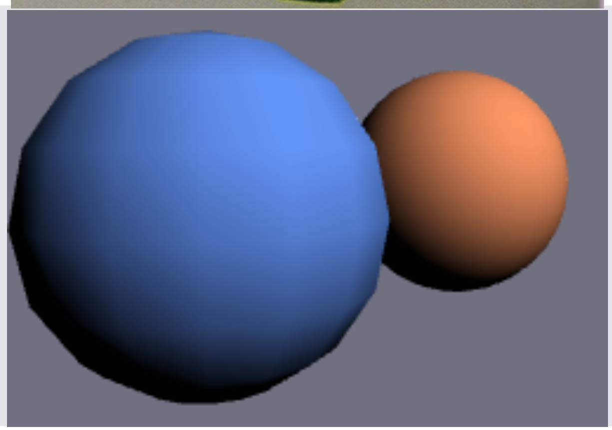
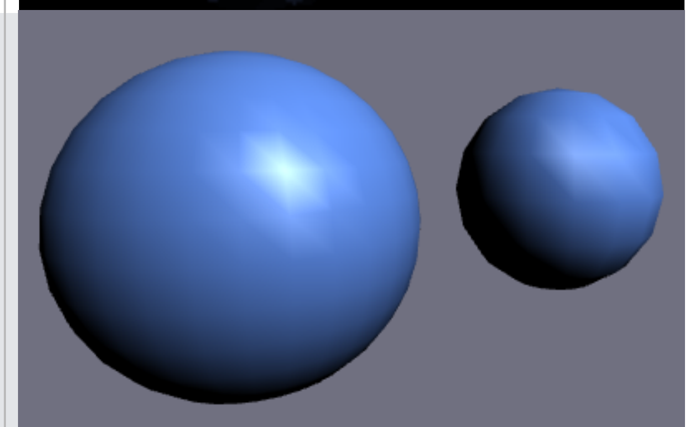
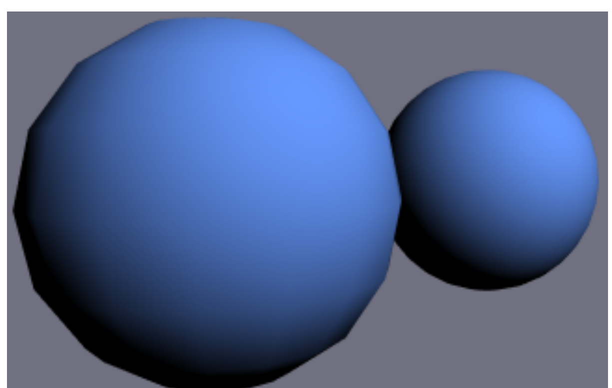
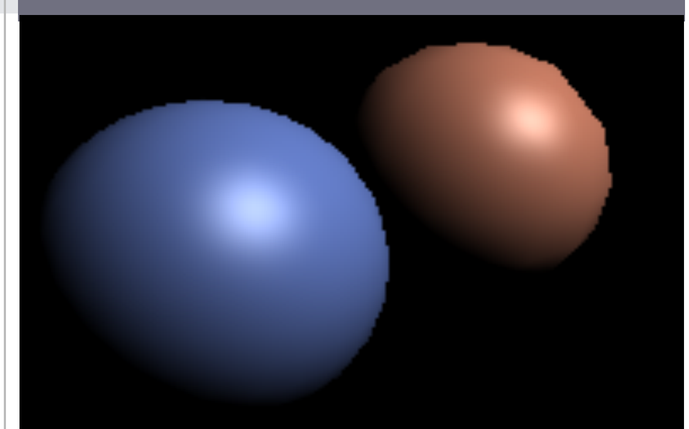
Interpolation

	Vertex Shader	Rasterizer	Fragment Shader
Flat	p, n, l --> cam space color = $n \cdot \ell \cdot$ vtx_color p --> screen space		Write color if z buffer says so
Gouraud	p, n, l --> cam space color = $n \cdot \ell \cdot$ vtx_color p --> screen space		Write color if z buffer says so
Phong	p, n, l --> cam space p --> screen space pass through vtx_color		color = n * l * frag_color Write color if z buffer says so

Summary: Shading and Interpolation Techniques

reflection

interpolation

	Lambertian	Blinn-phong
Flat		
Gouraud		
Phong		

Summary: Shading and Interpolation Techniques

Pipeline Stage

Interpolation

	Vertex Shader	Rasterizer	Fragment Shader
Flat	p, n, l --> cam space color = $n \cdot \ell \cdot$ vtx_color p --> screen space	Interpolate z' Pass through color	Write color if z buffer says so
Gouraud	p, n, l --> cam space color = $n \cdot \ell \cdot$ vtx_color p --> screen space	Interpolate z' Interpolate color	Write color if z buffer says so
Phong	p, n, l --> cam space p --> screen space pass through vtx_color	Interpolate z' Interpolate vtx_color Interpolate normal	color = n * l * frag_color Write color if z buffer says so

Terminology, so far

- Clipping
- Rasterization
- Interpolation
- Fragment
- Shader