COSE436

Lecture 17: Blending & Buffers

Nov 12, 2024

Won-Ki Jeong
(wkjeong@korea.ac.kr)

KOREA UNIVERSITY

Outline

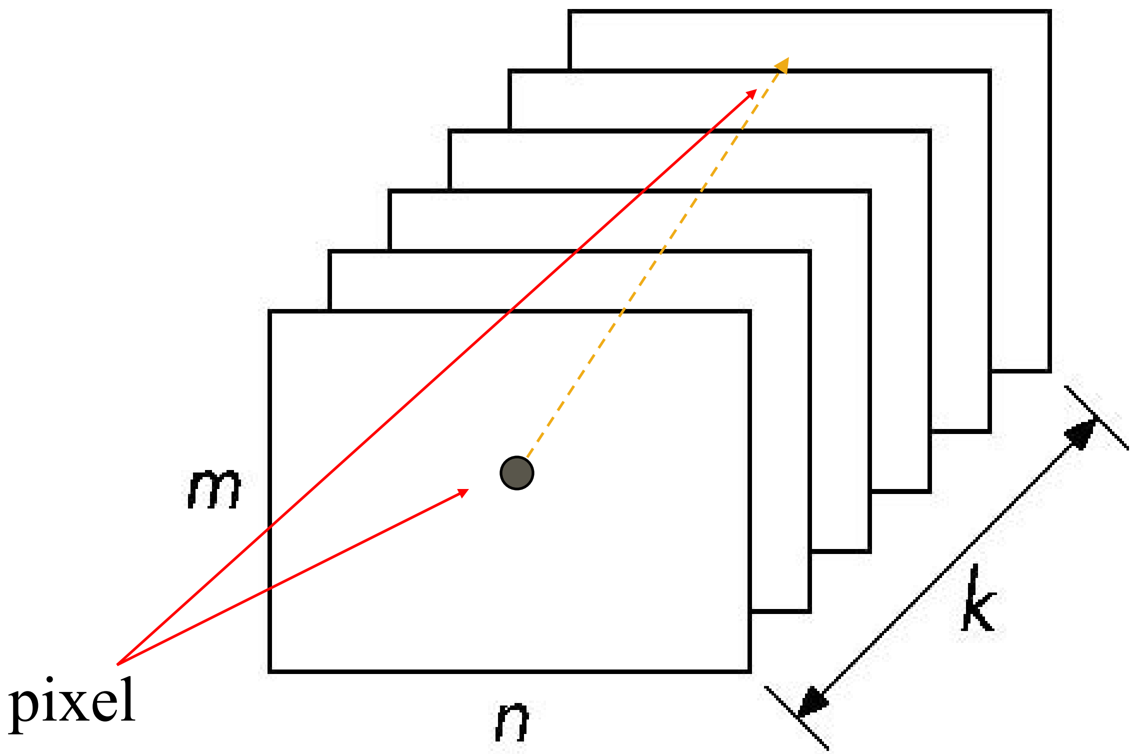
- OpenGL buffers
- Blending
- Off-screen rendering

Outline

- OpenGL buffers
- Blending
- Off-screen rendering

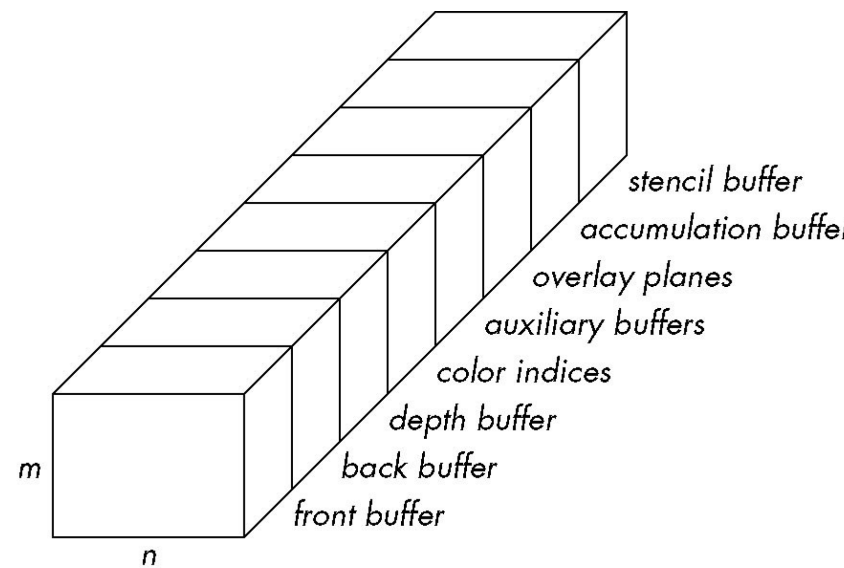
Buffer

- Spatial resolution (n x m) and precision (k bits)



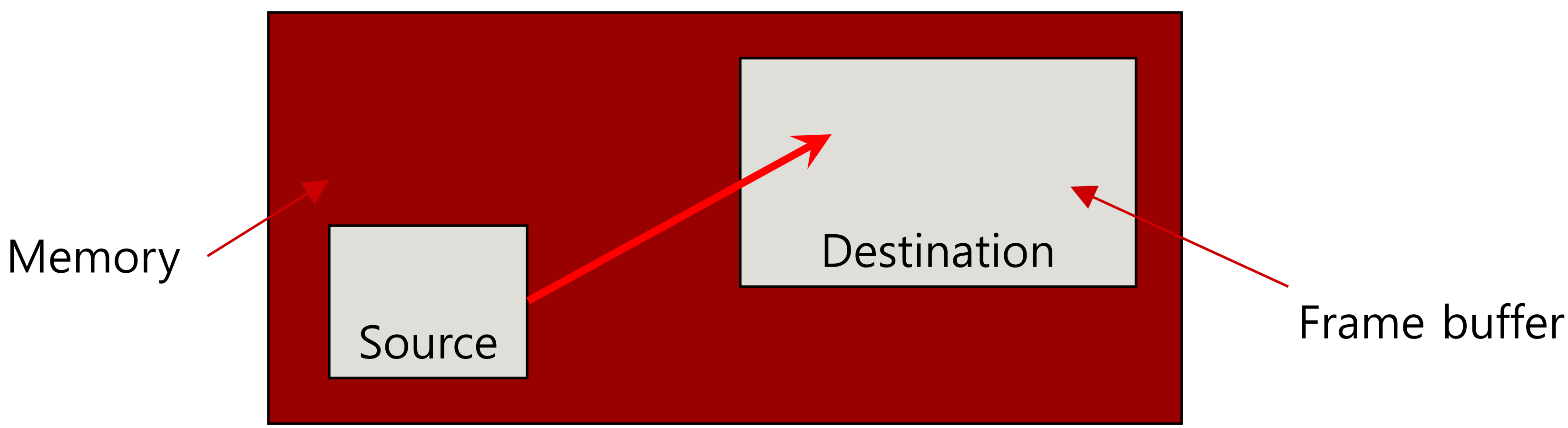
OpenGL Buffers

- Color buffers can be displayed
  - Front
  - Back
  - Auxiliary
  - Overlay
- Depth
- Accumulation
  - High resolution buffer
- Stencil
  - Hold masks



Writing Buffers

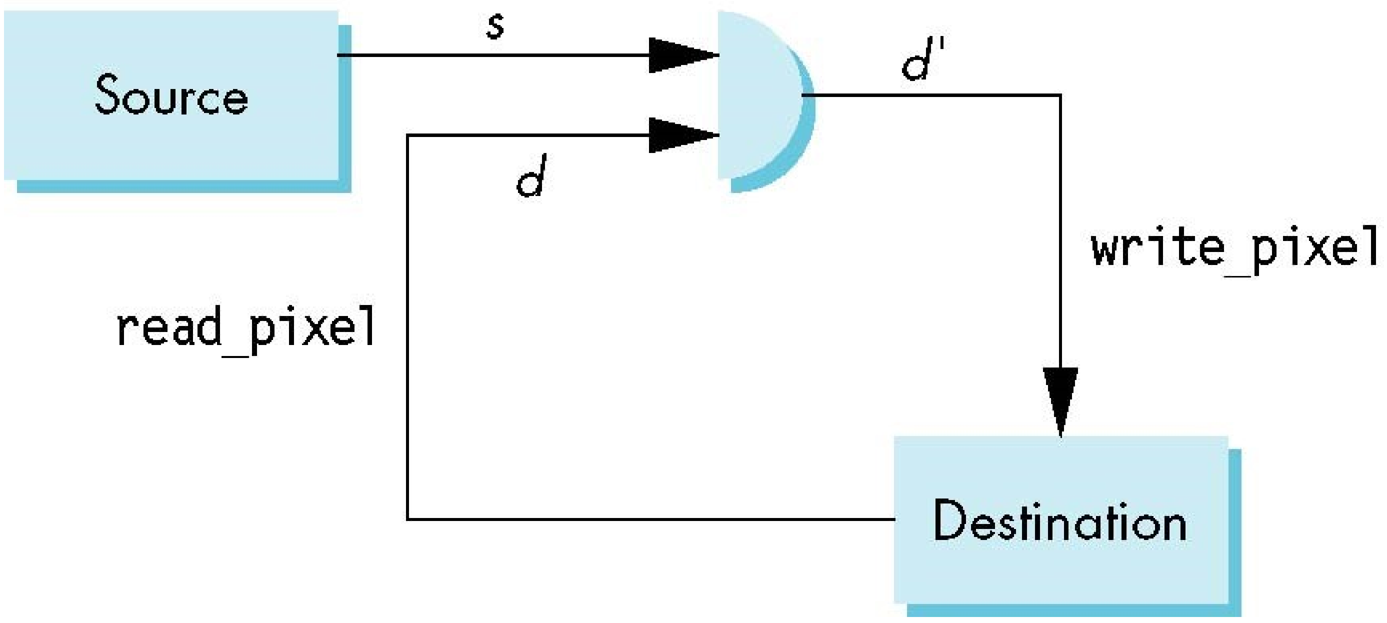
- Conceptually, we can consider all of memory as a large two-dimensional array of pixels
- We read and write rectangular block of pixels
  - ***Bit block transfer*** *(**bitblt**) operations*
- The frame buffer is part of this memory



Writing into frame buffer

Writing Model

- Read destination pixel before writing source



Bit Writing Modes

- Source and destination bits are combined bitwise
- 16 possible functions (one per column in table)

Replace XOR OR

s	*d*	0	1	2	3 (Replace)	4	5	6 (XOR)	7 (OR)	8	9	10	11	12	13	14	15
0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
0	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	0	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

Logic Operation

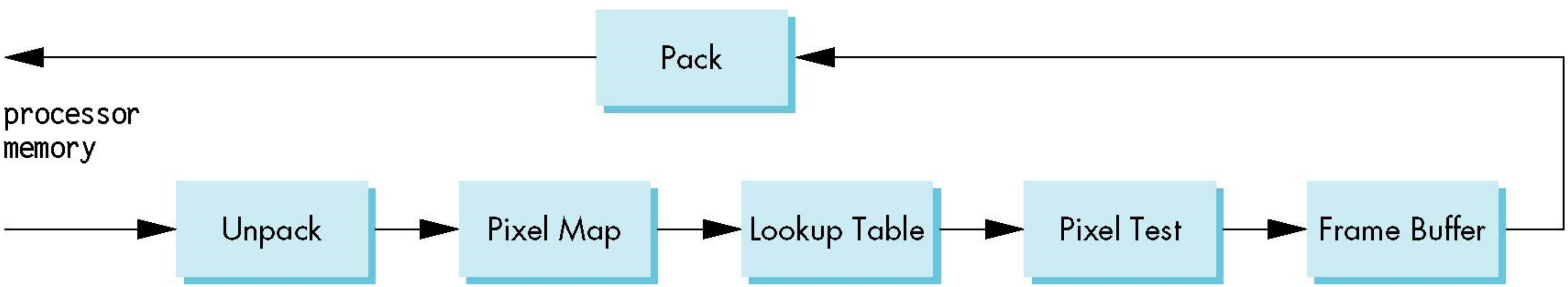
- glLogicOp()

Opcode	Resulting Operation
GL_CLEAR	0
GL_SET	1
GL_COPY	s
GL_COPY_INVERTED	~s
GL_NOOP	d
GL_INVERT	~d
GL_AND	s & d
GL_NAND	~(s & d)
GL_OR	s | d
GL_NOR	~(s | d)
GL_XOR	s ^ d
GL_EQUIV	~(s ^ d)
GL_AND_REVERSE	s & ~d
GL_AND_INVERTED	~s & d
GL_OR_REVERSE	s | ~d
GL_OR_INVERTED	~s | d

```
glEnable( GL_COLOR_LOGIC_OP );
glLogicOp( GL_XOR );// default: GL_COPY
```

The Pixel Pipeline

- OpenGL has a separate pipeline for pixels
  - Writing pixels involves
    - Moving pixels from processor memory to the frame buffer
    - Format conversions
    - Mapping, lookups, tests
  - Reading pixels



Buffer Selection

- OpenGL can draw into or read from any of the color buffers (front, back, auxiliary)
  - Default to the back buffer
  - Change with **`glDrawBuffer`** and **`glReadBuffer`** functions
- Note that format of the pixels in the frame buffer is different from that of processor memory and these two types of memory reside in different places
  - Need packing and unpacking
  - Drawing and reading can be slow

Pixel Maps

- OpenGL works with rectangular array of pixels called pixel maps or images
- Pixels are in one byte (8 bit) chunks
  - Luminance (gray scale) images 1 byte/pixel
  - RGB 3 bytes/pixel
- Three functions
  - Draw pixels: processor memory to frame buffer
  - Read pixels: frame buffer to processor memory
  - Copy pixels: frame buffer to frame buffer

OpenGL Pixel Functions

```
glReadPixels(x, y, width, height, format, type, myimage);
```

Start pixel in frame buffer

Size

Type of image

Type of pixels

Pointer to processor memory

Ex)
```
Glubyte myimage[512][512][3];
glReadPixels( 0, 0, 512, 512, GL_RGB,
              GL_UNSIGNED_BYTE, myimage );
```

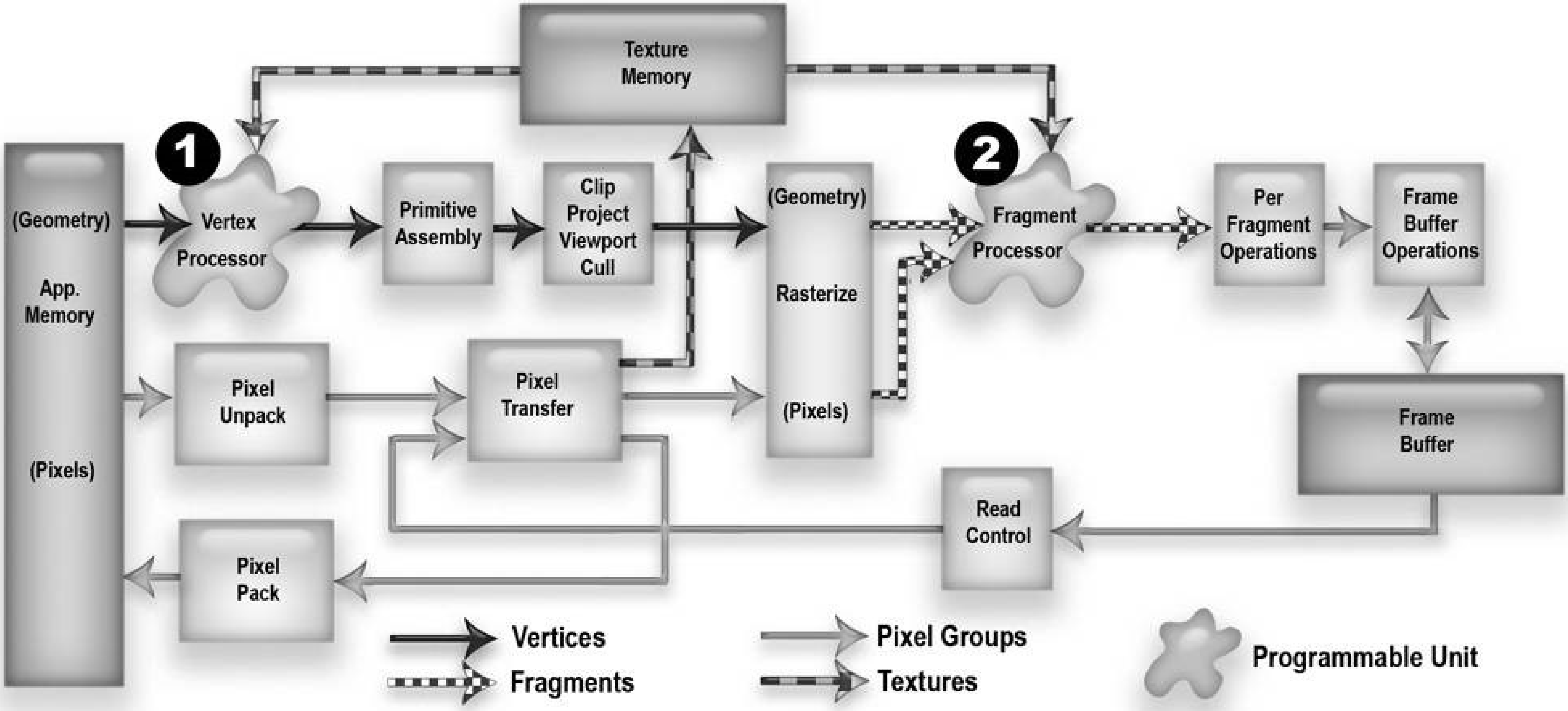
```
glDrawPixels(width, height, format, type, myimage);
```

Start at raster position (glRasterPos to set the current raster position (lower-left))

```
glCopyPixels(x, y, width, height, type);
```

(x,y): window coordinate, copy to the current raster position

OpenGL Pipeline



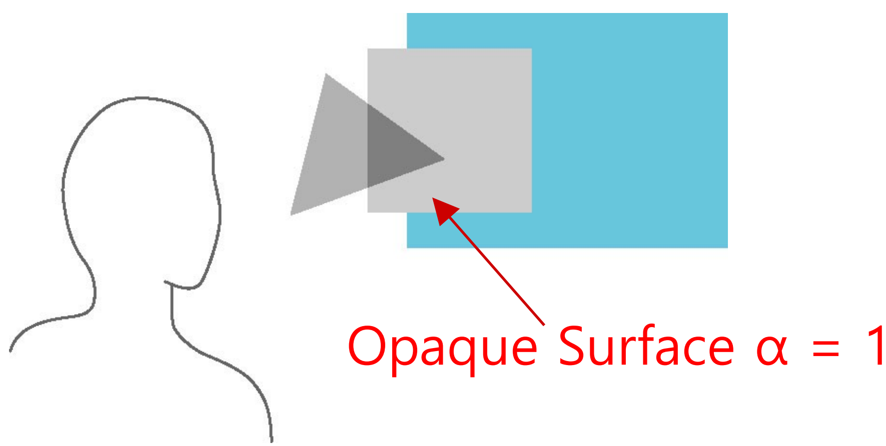
Outline

- OpenGL buffers
- Blending
  - Translucent object rendering
  - Depth peeling
- Off-screen rendering

Opacity and Transparency

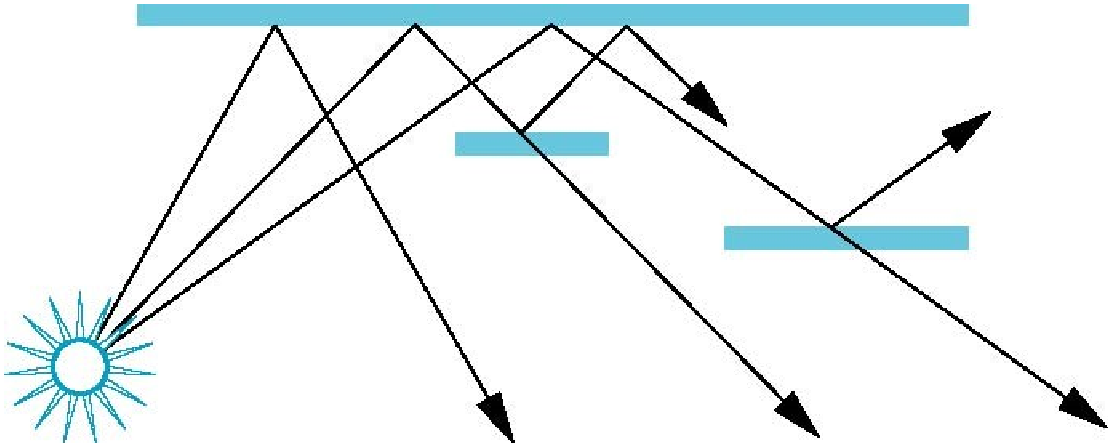
- Opaque surfaces permit no light to pass through
- Transparent surfaces permit all light to pass
- Translucent surfaces pass some light

Translucency = 1 – Opacity(α)



Physical Models

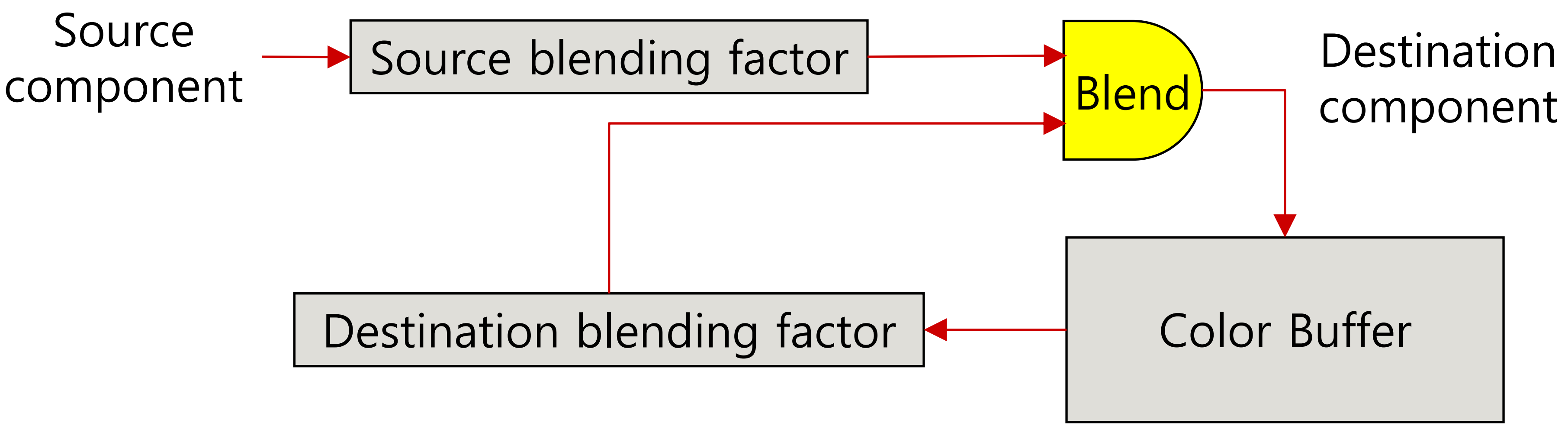
- Dealing with translucency in a physically correct manner is difficult due to
  - The complexity of the internal interactions of light and matter
  - Limitation of a raster pipeline renderer



Scene with translucent objects

Writing Model for Blending

- Use A (alpha) component of RGBA color to store opacity
- During rendering we can expand our writing model to use RGBA values



Blending Equation

- We can define source and destination blending factors for each RGBA component

$$\mathbf{s} = [s_r, s_g, s_b, s_\alpha]$$
$$\mathbf{d} = [d_r, d_g, d_b, d_\alpha]$$

- Suppose that the source and destination colors are

$$\mathbf{b} = [b_r, b_g, b_b, b_\alpha]$$
$$\mathbf{c} = [c_r, c_g, c_b, c_\alpha]$$

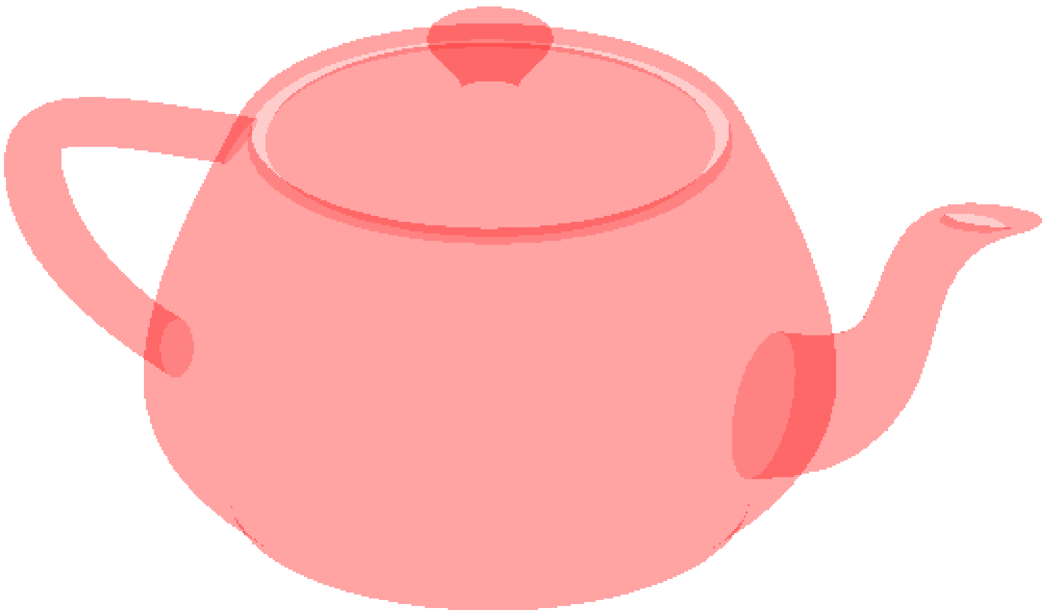
- Blend as

$$\mathbf{d} = \mathbf{b} \cdot \mathbf{s} + \mathbf{c} \cdot \mathbf{d}$$

$$\mathbf{d} = [b_r\, s_r + c_r\, d_r,\ b_g\, s_g + c_g\, d_g\,,\ b_b\, s_b + c_b\, d_b\,,\ b_\alpha\, s_\alpha + c_\alpha\, d_\alpha\,]$$

OpenGL Blending

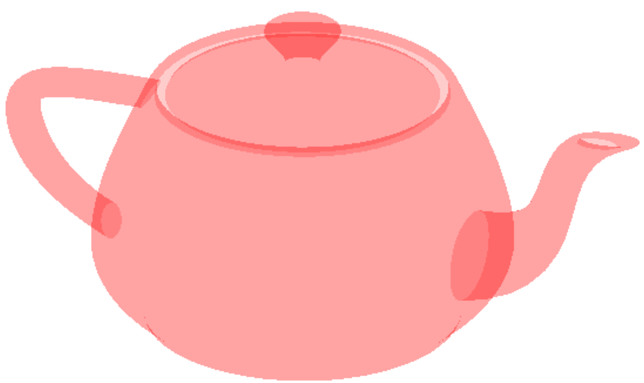
- Enable blending and pick source and destination blending factor



OpenGL Blending

- Must enable blending and pick source and destination factors
- Don't forget to assign alpha value

```
glEnable(GL_BLEND);
glBlendFunc(GL_SRC_ALPHA,GL_ONE_MINUS_SRC_ALPHA);
...
glColor4f(1,0,0,0.2);
glutSolidTeapot(60);
```



glBlendFunc(src, dest)

Constant	Relevant Factor	Computed Blend Factor
GL_ZERO	source or destination	(0, 0, 0, 0)
GL_ONE	s. or d.	(1, 1, 1, 1)
GL_DST_COLOR	source	(R_d, G_d, B_d, A_d)
GL_SRC_COLOR	destination	(R_s, G_s, B_s, A_s)
GL_ONE_MINUS_DST_COLOR	source	$(1,1,1,1)-(R_d, G_d, B_d, A_d)$
GL_ONE_MINUS_SRC_COLOR	destination	$(1,1,1,1)-(R_s, G_s, B_s, A_s)$
GL_SRC_ALPHA	s. or d.	(A_s, A_s, A_s, A_s)
GL_ONE_MINUS_SRC_ALPHA	s. or d.	$(1,1,1,1)-(A_s, A_s, A_s, A_s)$
GL_DST_ALPHA	s. or d.	(A_d, A_d, A_d, A_d)
GL_ONE_MINUS_DST_ALPHA	s. or d.	$(1,1,1,1)-(A_d, A_d, A_d, A_d)$
GL_SRC_ALPHA_SATURATE	source	$(f, f, f, 1): f=\min(A_s, 1-A_d)$

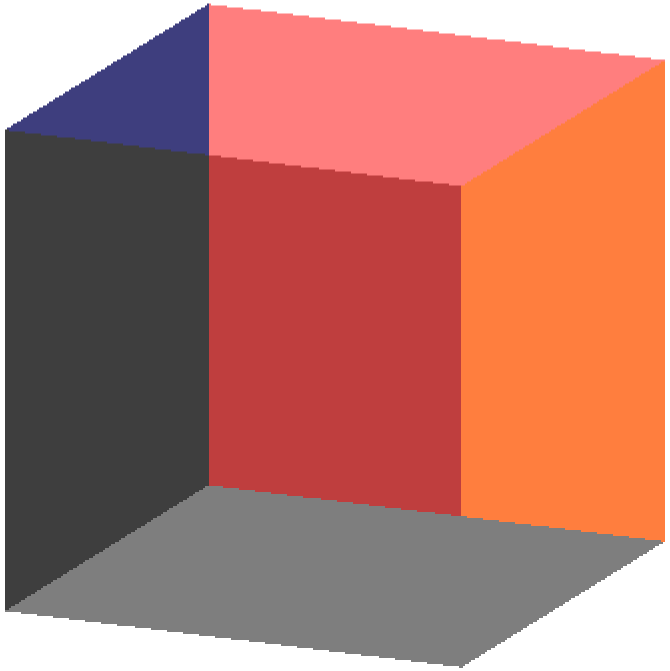
Example: Blending

- Suppose that we start with the opaque background color $(R_0,G_0,B_0,1)$
  - This color becomes the initial destination color
- We now want to blend in a translucent polygon with color (R_1,G_1,B_1,α_1)
- Select **GL_SRC_ALPHA** and **GL_ONE_MINUS_SRC_ALPHA** as the source and destination blending factors

$R'_1 = \alpha_1 R_1 + (1- \alpha_1) R_0$ $G'_1 = \alpha_1 G_1 + (1- \alpha_1) G_0$ $B'_1 = \alpha_1 B_1 + (1- \alpha_1) B_0$

Order Dependency

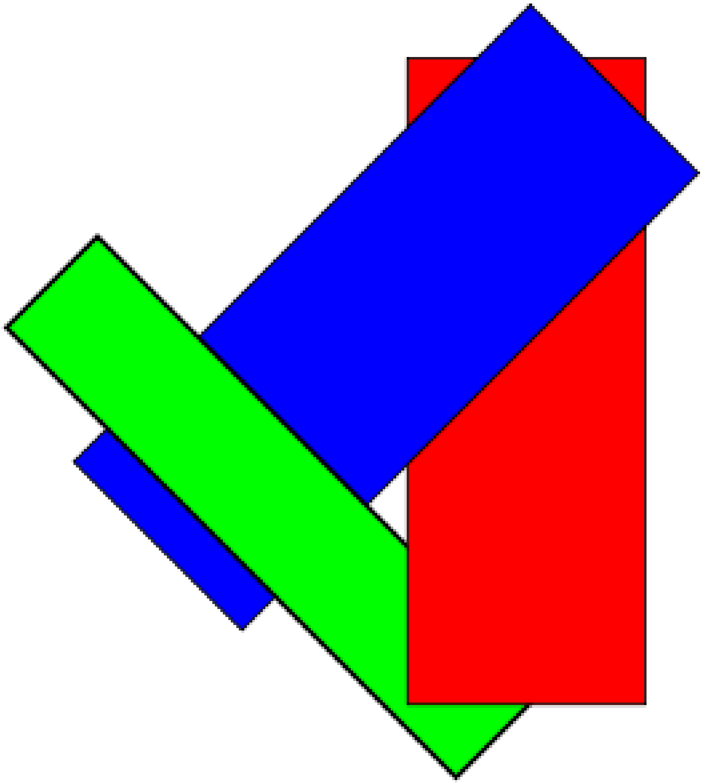
- Is this image correct?
  - Probably not
  - Polygons are rendered in the order they pass down the pipeline
  - Blending functions are <u>order dependent</u>



Opaque and Translucent Polygons

- Suppose that we have a group of polygons some of which are opaque and some translucent
  - How do we use hidden-surface removal?
  - Opaque polygons block all polygons behind them and affect the depth buffer
  - Translucent polygons should not affect depth buffer
    - Render with `glDepthMask(GL_FALSE)` which makes depth buffer read-only
- Sort polygons first to remove order dependency
  - Does not work all the time!

Partial Occlusion



Opaque & Translucent Objects

- Rendering order: Teapot -> Plane

Depth test is on. Plane does not change depth buffer but use it (updated by teapot), so the region on the plane hidden by the teapot is not drawn, and the region above teapot is rendered with blending

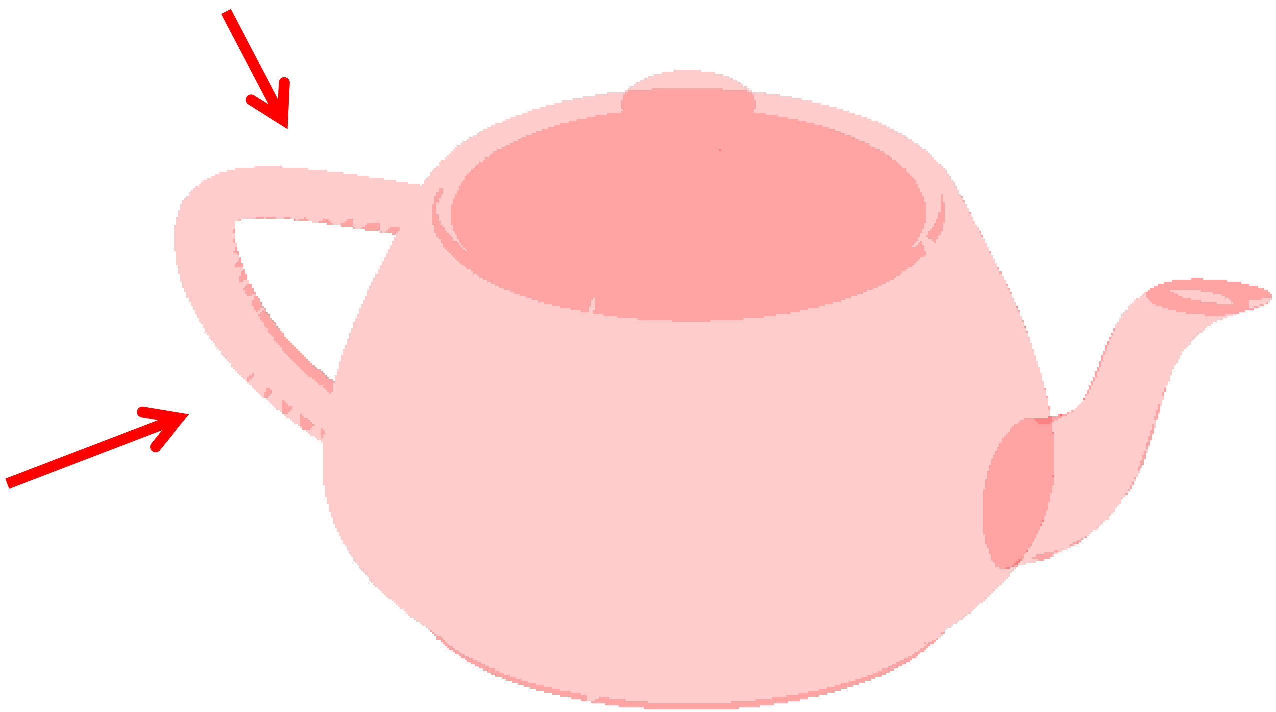
Opaque & Translucent Objects

- Plane -> Teapot (wrong: partial occlusion)

Plane does not change depth buffer due to translucency.

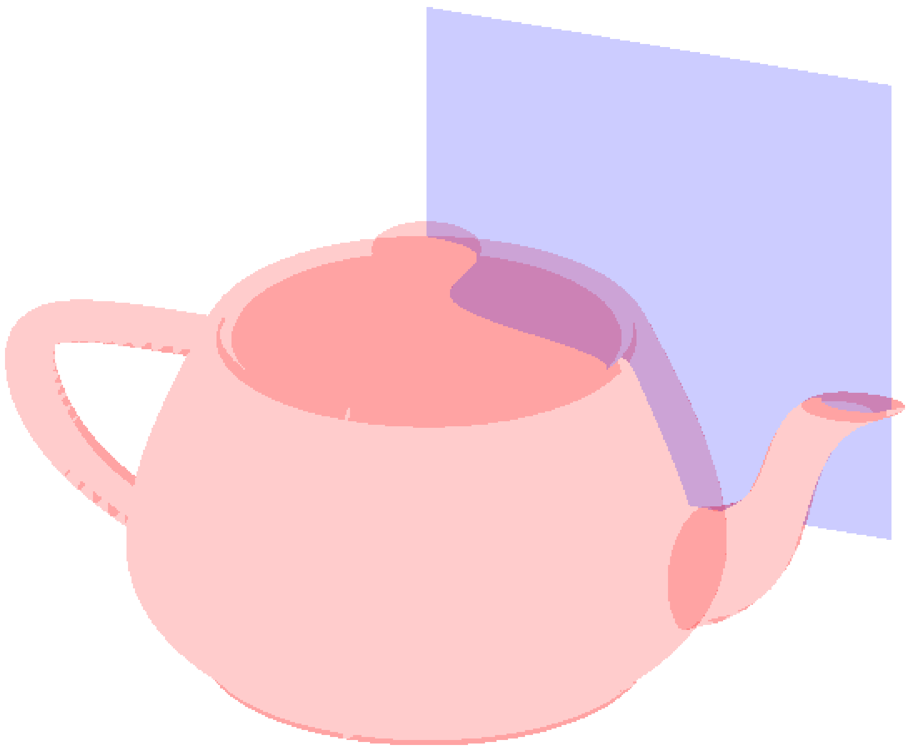
Translucent Object with Depth Test

- Depth test is not required for translucent objects



Translucent Objects with Depth Test

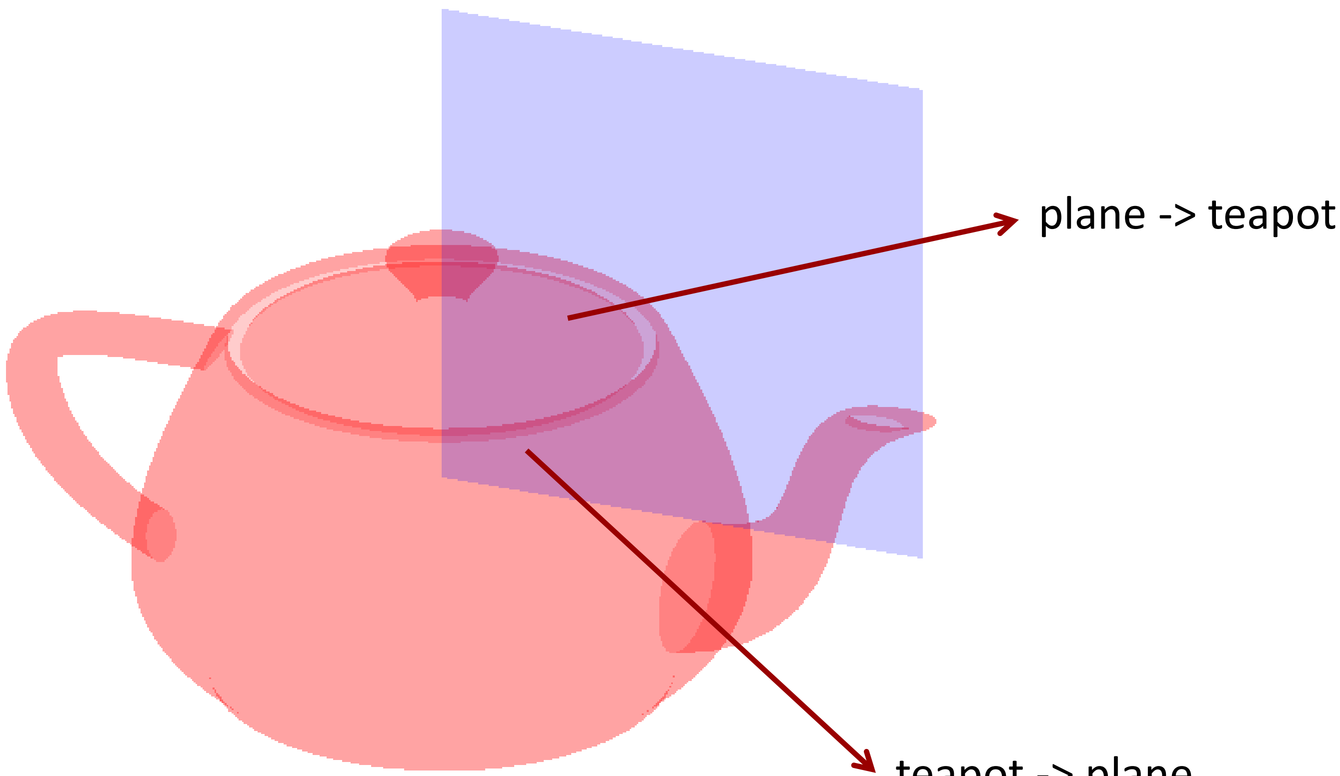
- Depth test is not required for translucent objects



Teapot is rendered before plane with depth test on. Therefore, plane intersected with teapot is not rendered -> incorrect.

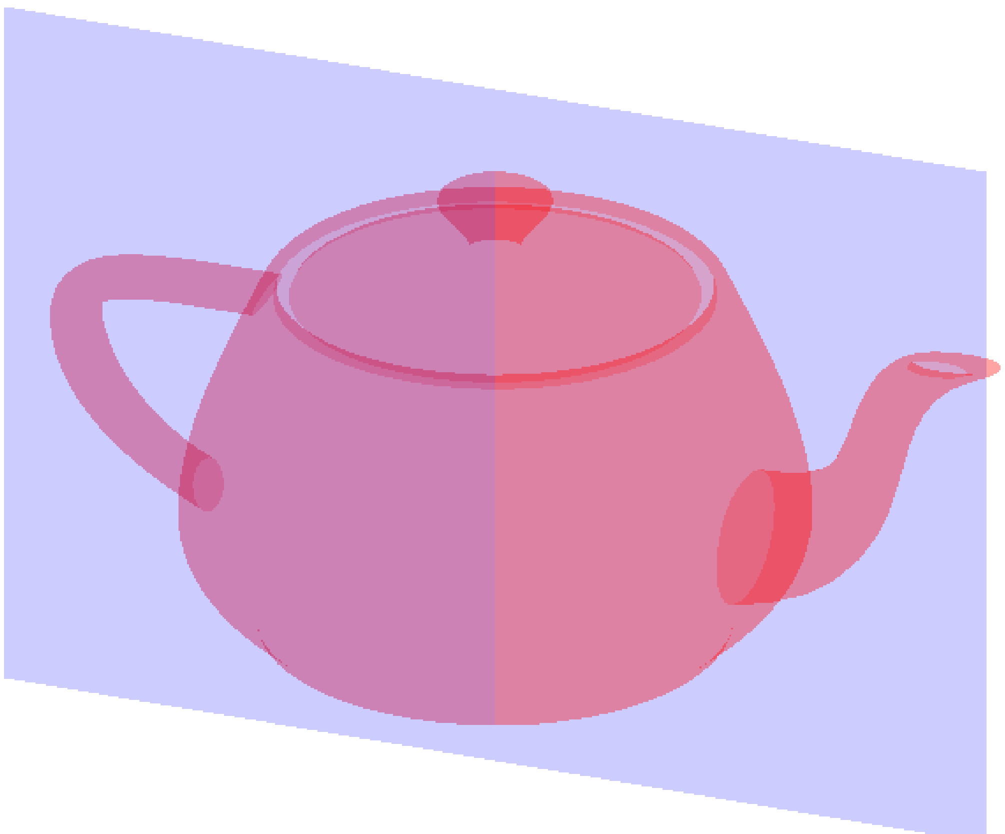
Blending w/o Depth test

- However, we need correct pixel ordering



Ordering is important

- Right plane -> Teapot -> Left plane



Example

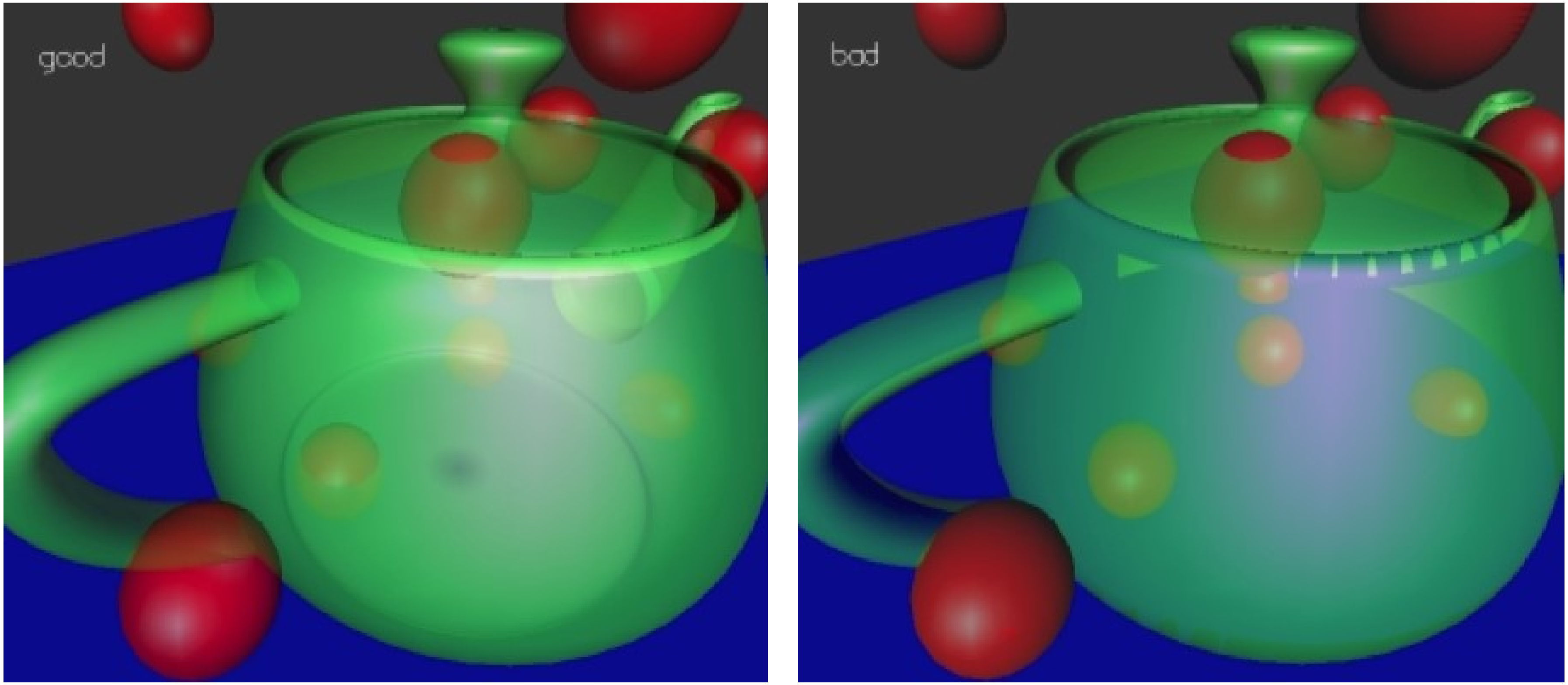


Image Courtesy of NVIDIA

Lesson learned:

For correct rendering of translucent objects, we need per-pixel depth sorting!

How?

Depth Peeling

- The algorithm uses an "implicit sort" to extract multiple depth layers
  - First pass render finds front-most fragment color/depth
  - Each successive pass render finds (extracts) the fragment color/depth for the next-nearest fragment on a per pixel basis
  - Use dual depth buffers to compare previous nearest fragment with the current
  - Second "depth buffer" used for comparison (read only) from texture

Cross-section View

- Depth peeling strips away depth layers with each successive pass. The image below show the frontmost (leftmost) surfaces as bold black lines, hidden surfaces as thin black lines, and “peeled away” surfaces as light grey lines.

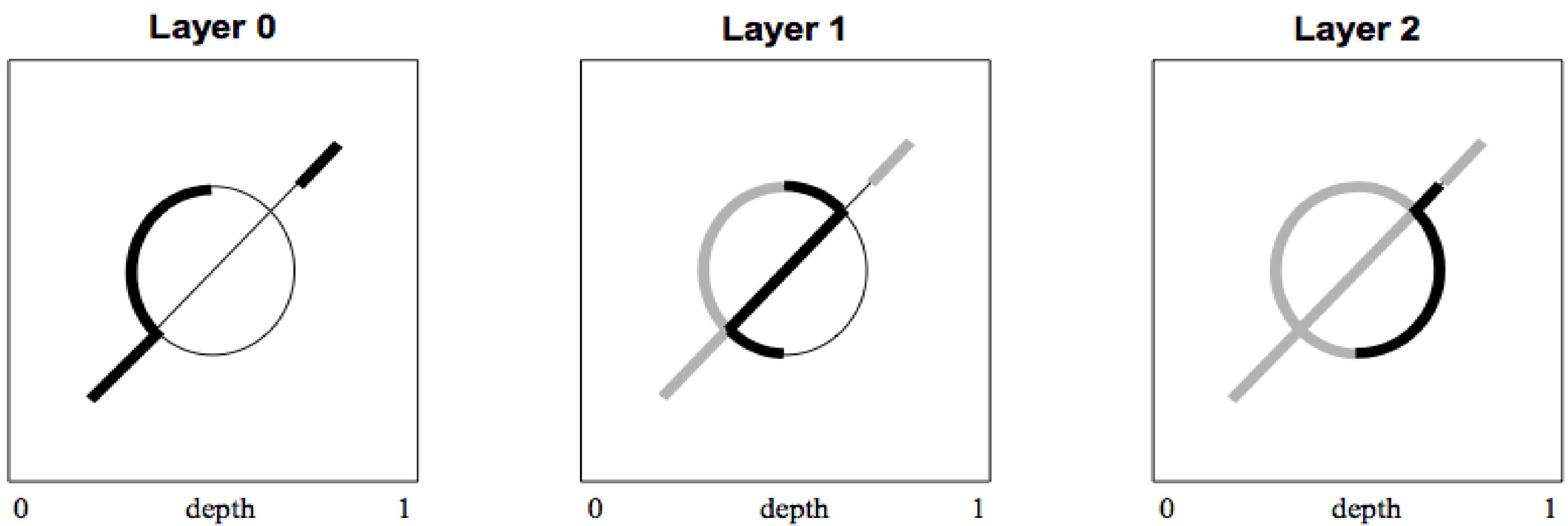


Image Courtesy of NVIDIA

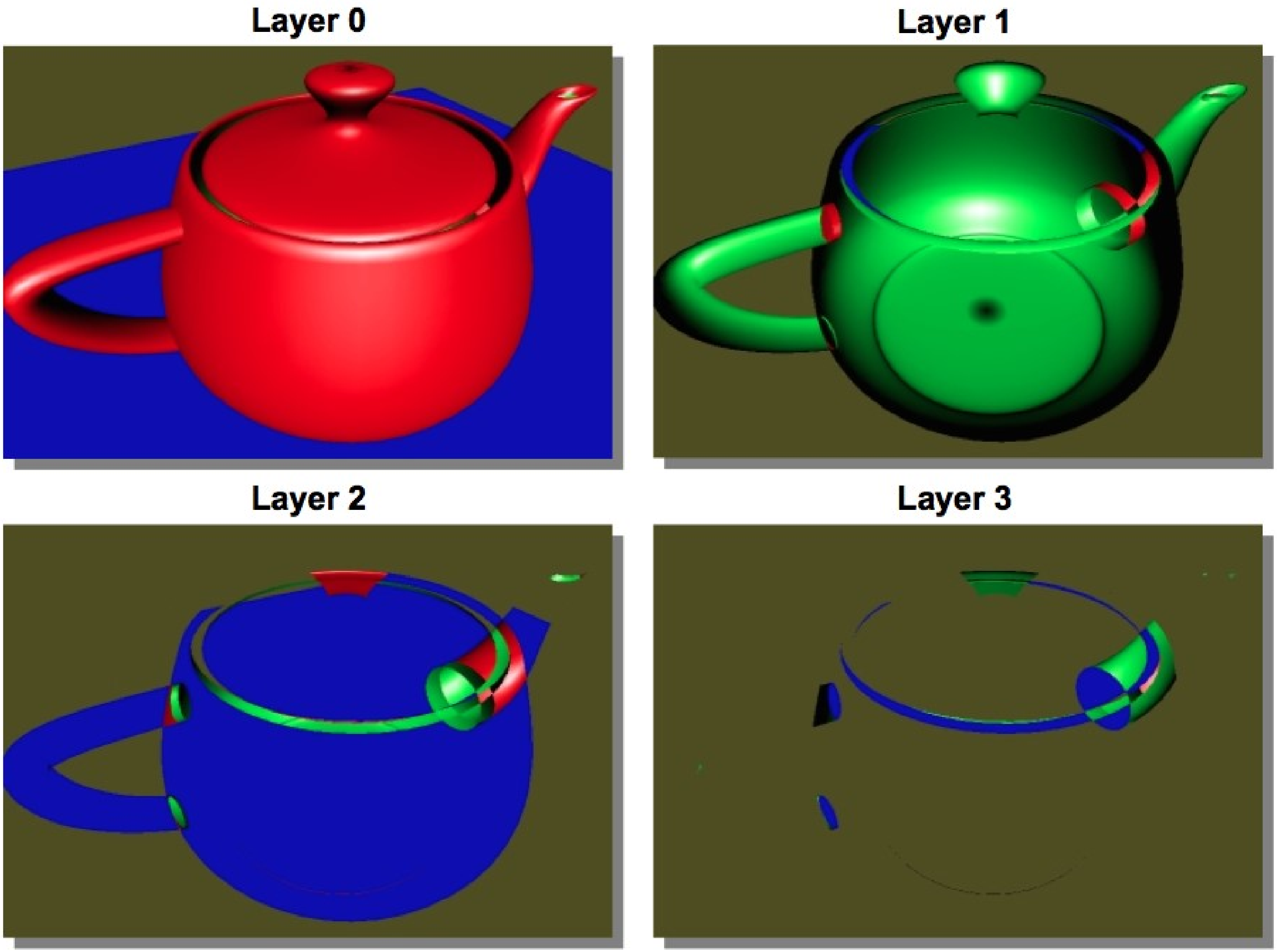


Image Courtesy of NVIDIA

Dual Depth Buffer Pseudo Code

```
for ( i = 0; i < num_passes; i++ )
{
  clear color buffer
  depth unit 0:
      if(i == 0) { disable depth test }
      else       { enable depth test  }
      bind depth buffer (i % 2)
      disable depth writes  /* read-only depth test */
      set depth func to GREATER
  depth unit 1:
      bind depth buffer ((i+1) % 2)
      clear depth buffer
      enable depth writes;
      enable depth test;
      set depth func to LESS
  render scene
  save color buffer RGBA as layer i
}
```

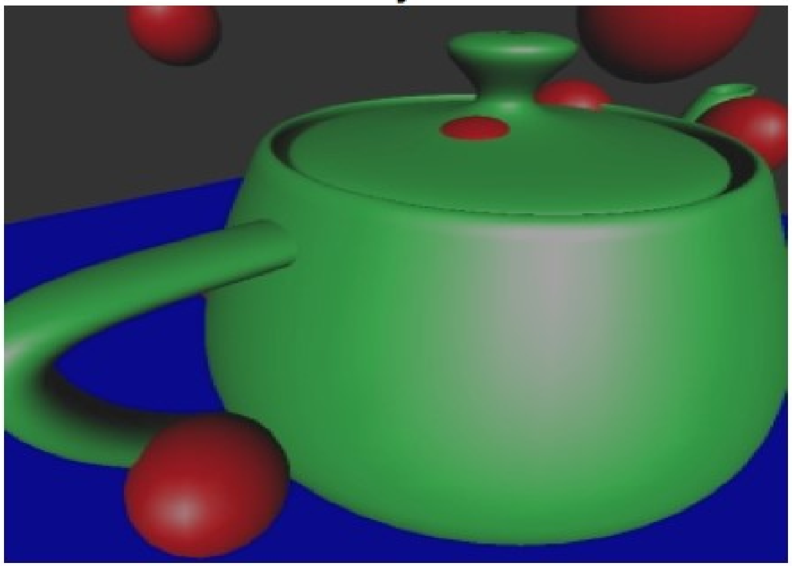
pick fragments that are far from the last peeled layer

keep track of the closest depth value

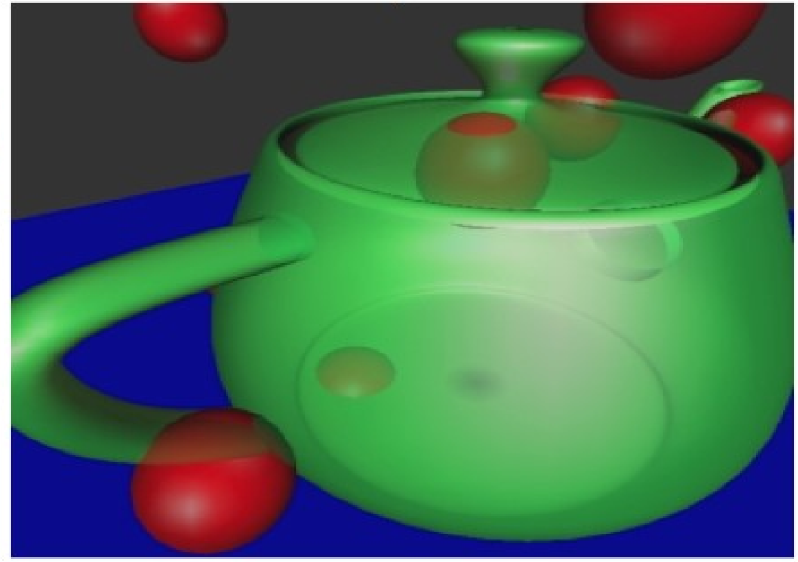
Compositing

- RGBA per each peel
- Back-to-front composition at the end
- Acceleration
  - Use fewer # of layers (approximation)
  - ~4 would be acceptable

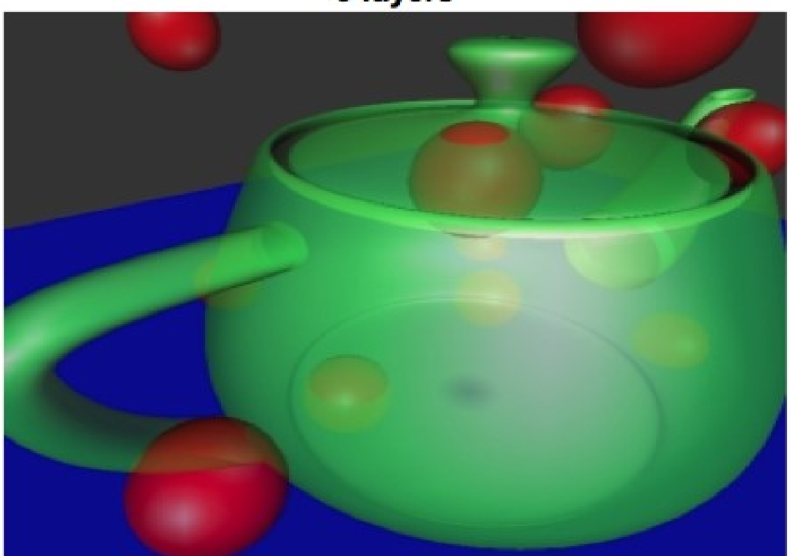
1 layer



2 layers



3 layers



4 layers

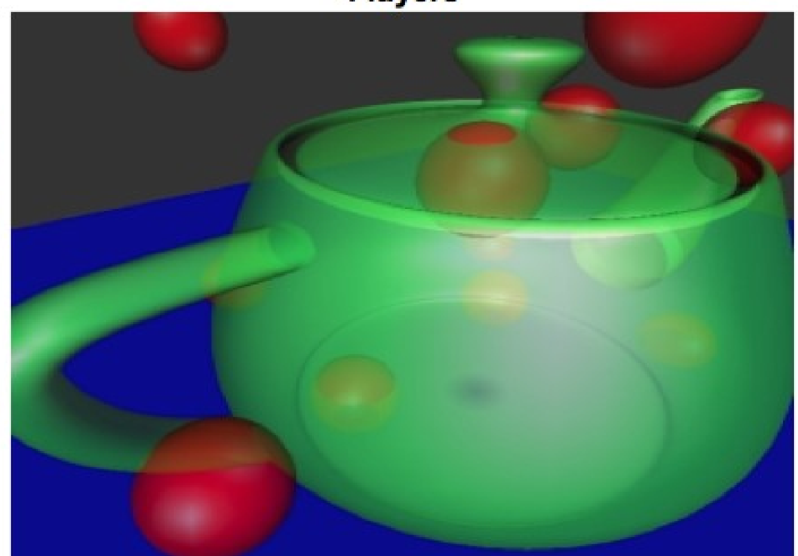


Image Courtesy of NVIDIA

Outline

- OpenGL buffers
- Blending
- Off-screen rendering
  - Framebuffer object (FBO)
  - Examples

Going between CPU and GPU

- We have already seen that we can write pixels as texels to texture memory
- Texture objects reduce transfers between CPU and GPU
- Transfer of pixel data back to CPU slow
- Want to manipulate pixels without going back to CPU : Multiple Render Pass
  - Image processing
  - GPGPU

Framebuffer Objects

- Framebuffer Objects (FBOs) are buffers that are created by the application
  - Not under control of window system
  - Cannot be displayed
  - Can attach a renderbuffer to a FBO and can render off screen into the attached buffer
  - Attached buffer can then be detached and <u>used as a texture map</u> for an on-screen render to the default frame buffer
  - Multiple render targets are available

Render to Texture

- Textures are shared by all instances of the fragment shader
- If we render to a texture attachment we can create a new texture image that can be used in subsequent renderings
- Use a double buffering strategy for operations such as convolution

Steps

- Create an Empty Texture Object
- Create a FBO
- Attach renderbuffer for texture image
- Bind FBO
- Render scene
- Detach renderbuffer
- Bind texture
- Render with new texture

Empty Texture Object

```
Gluint tex;
glGenTextures(1, &tex);
glBindTexture(GL_TEXTURE_2D, tex);

glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, 512, 512, 0,
             GL_RGBA, GL_UNSIGNED_BYTE, null);

glGenerateMipmap(GL_TEXTURE_2D);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
                GL_NEAREST_MIPMAP_LINEAR );
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
                GL_NEAREST )
```

Creating a FBO

- We create a framebuffer object in a similar manner to other objects
- Creating an FBO creates an empty FBO
- Must add needed resources
  - Can add a renderbuffer to render into
  - Can add a texture which can also be rendered into
  - For hidden surface removal we must add a depth buffer attachment to the renderbuffer

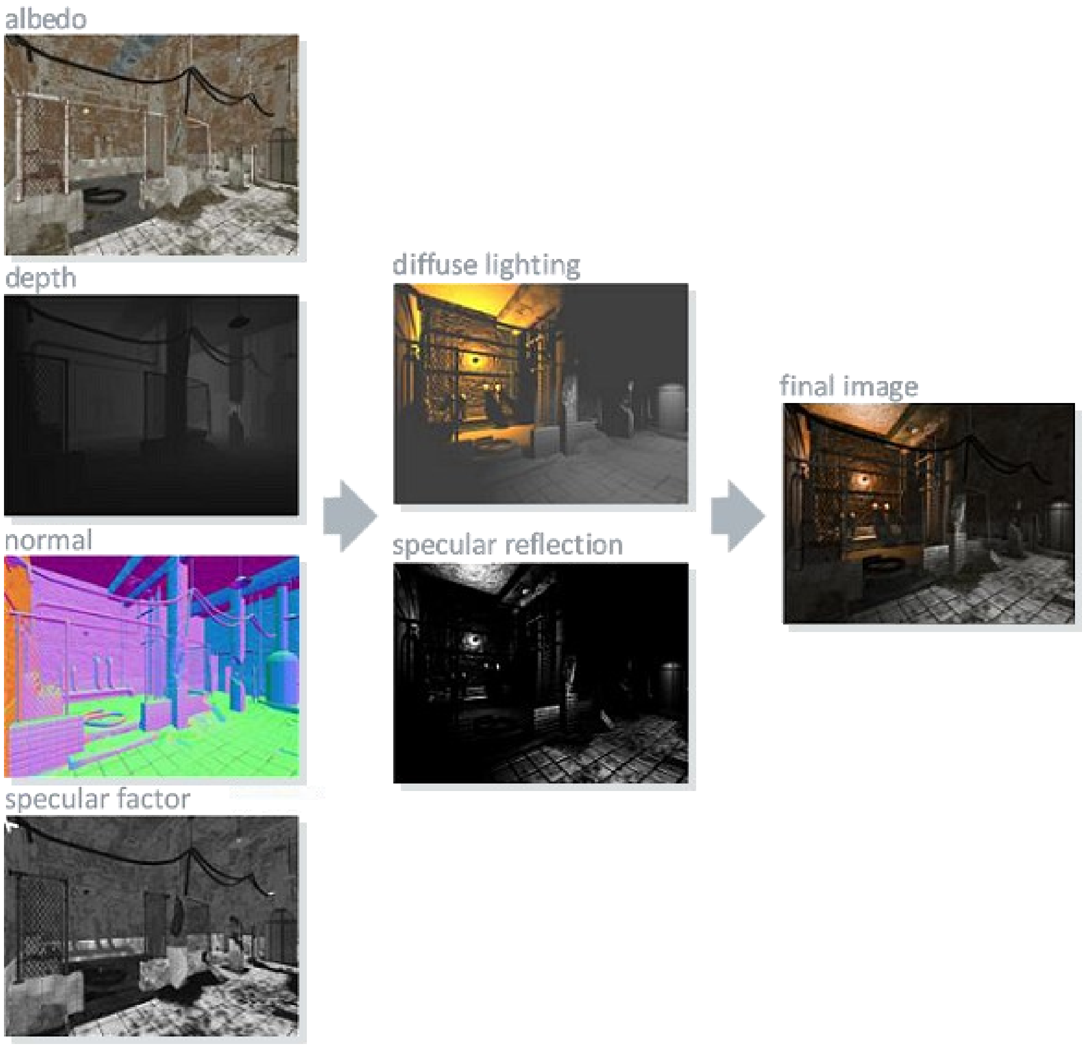
Frame Buffer Object

```
Gluint fb, rb;
glGenFramebuffers(1, &fb);
glBindFramebuffer(GL_FRAMEBUFFER, fb);

glGenRenderbuffers(1, &rb);
glBindRenderbuffer(GL_RENDERBUFFER, rb);
glRenderbufferStorage(GL_RENDERBUFFER,
                      GL_DEPTH_COMPONENT16, 512, 512);

// Attach texture (color buffer)
glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0,
                       GL_TEXTURE_2D, tex, 0);

// Attach render buffer (depth buffer)
glFramebufferRenderbuffer(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT,
                          GL_RENDERBUFFER, rb);

// check for completeness
var status = glCheckFramebufferStatus(GL_FRAMEBUFFER);
if(status != GL_FRAMEBUFFER_COMPLETE)
   std::out << "Frame Buffer Not Complete" << std::endl;
```

Example: Deferred Shading

- Application of multiple render targets
- Separate geometry rendering stage from lighting stage

- Algorithm
  - Render geometry to G-buffer
    - Normal, position, diffusion, specular, etc
  - Compute lighting/shading on G-buffer
    - Defer light calculation at the end

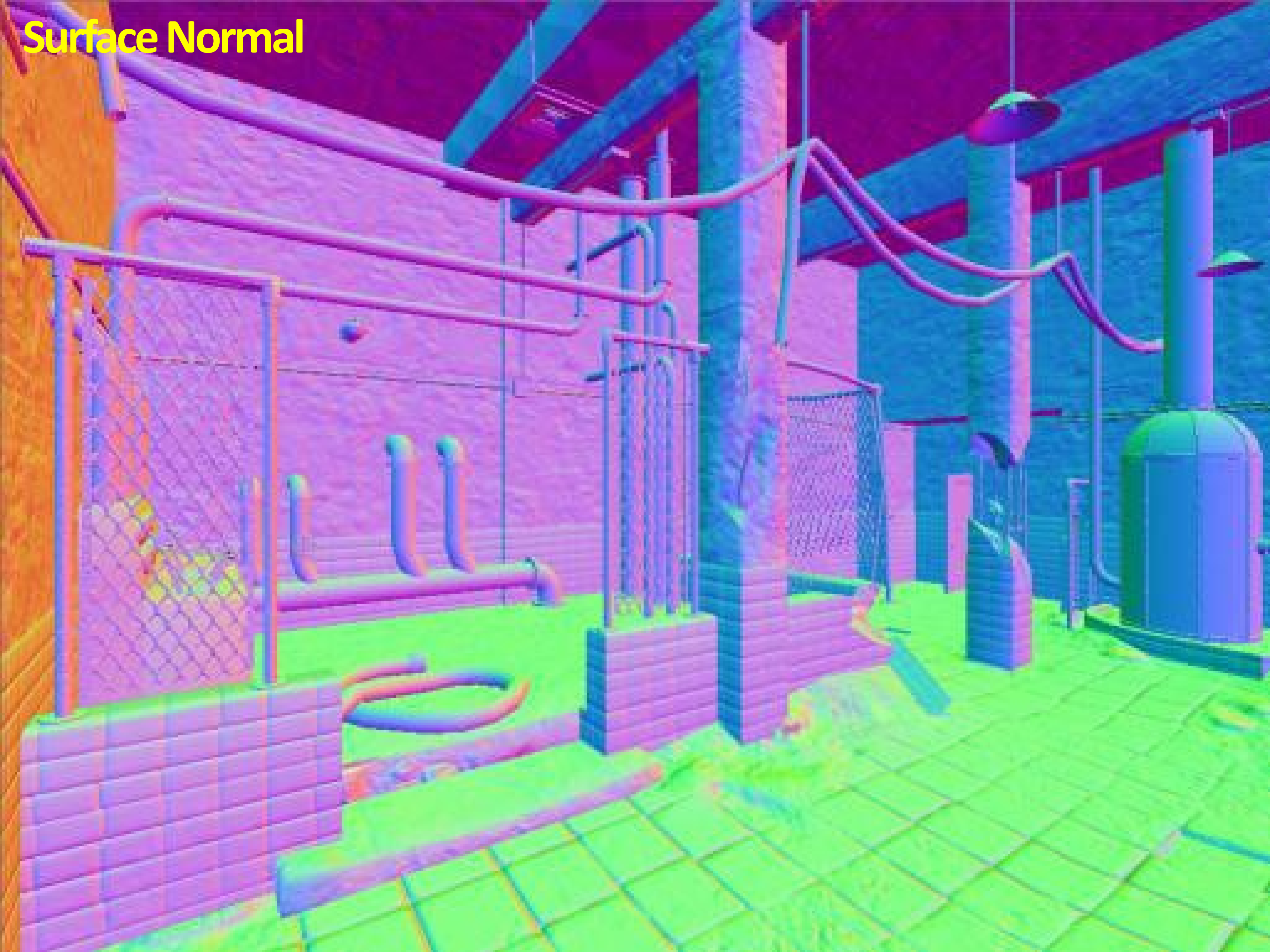


Examples

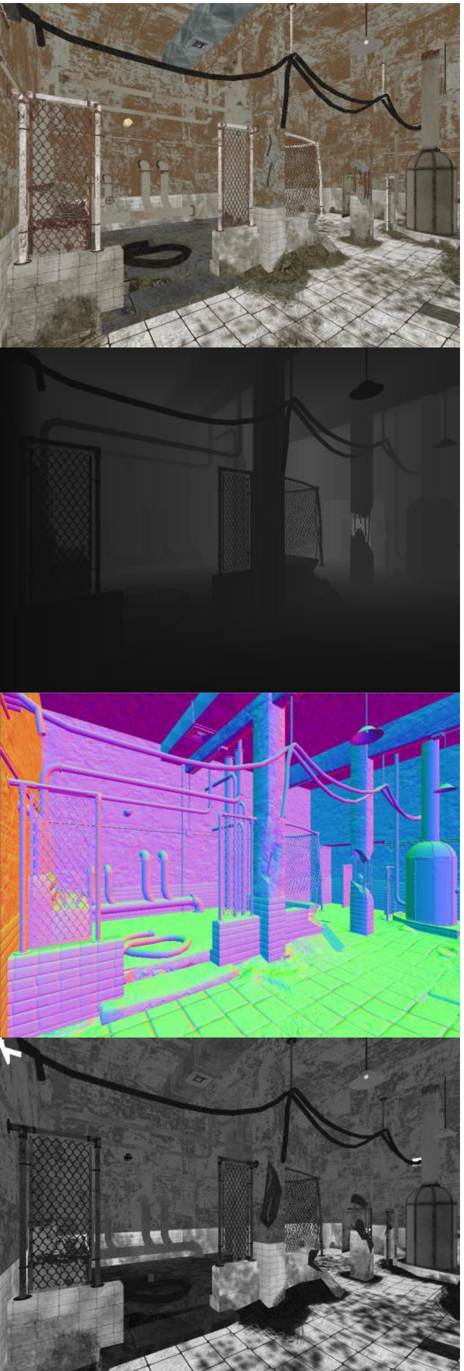
Deferred rendering examples


Diffusion Albedo

Depth


Surface Normal

Specular




Diffuse Lighting

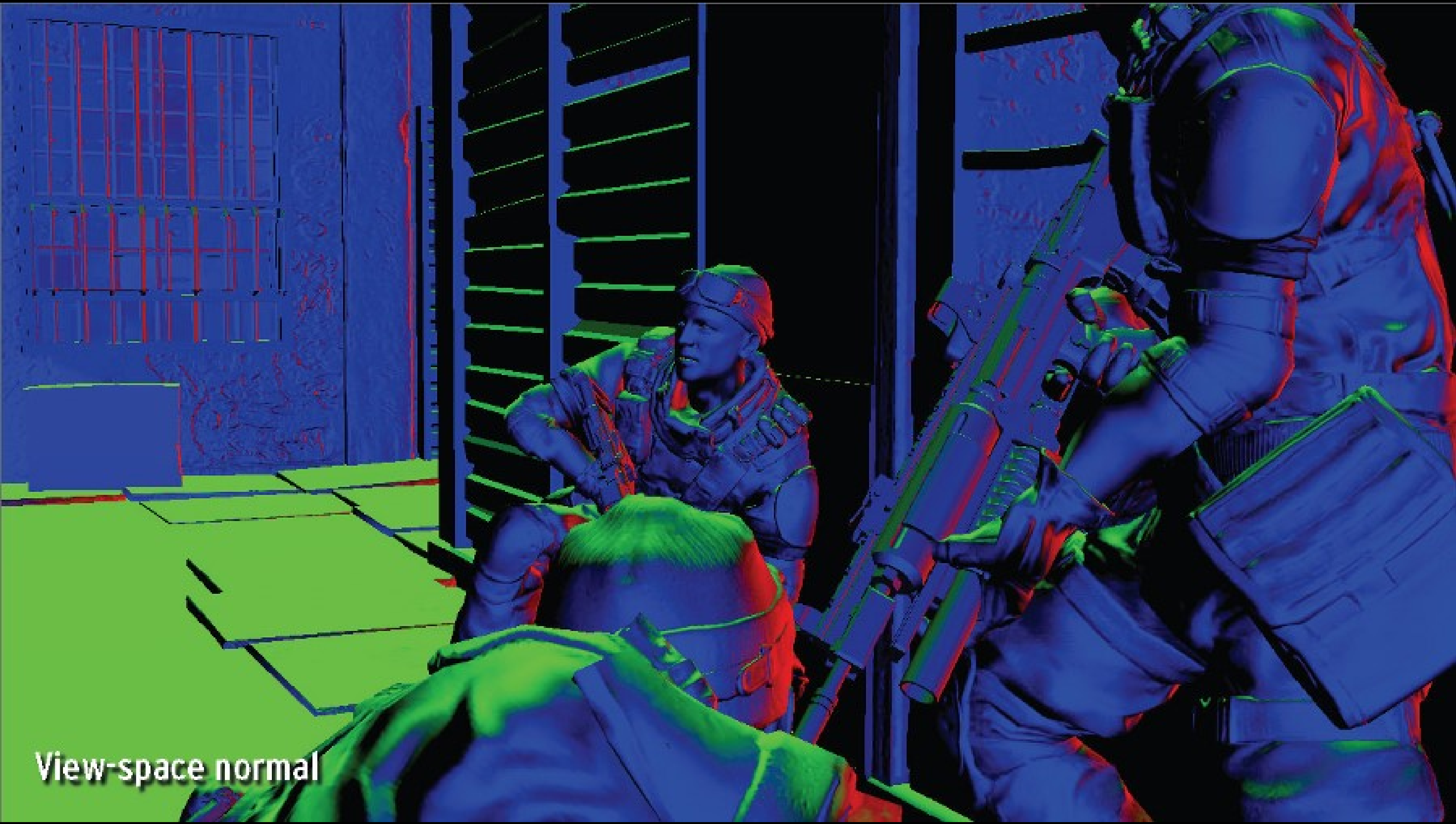

Specular Lighting


Final Rendering

Guerrilla's Killzone

Guerrilla's Killzone

Guerrilla's Killzone



Guerrilla's Killzone

Guerrilla's Killzone

Guerrilla's Killzone

Guerrilla's Killzone

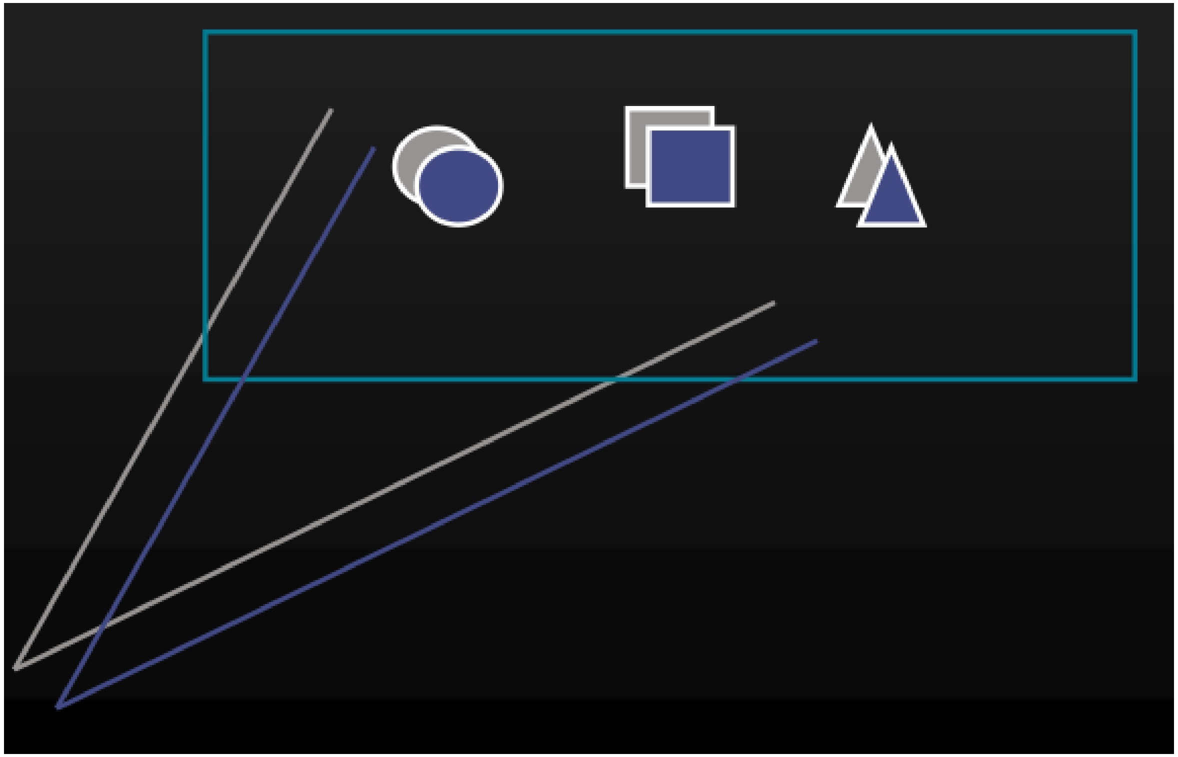
Guerrilla's Killzone

How deferred shading differs from early z-test?

More examples of FBO usage

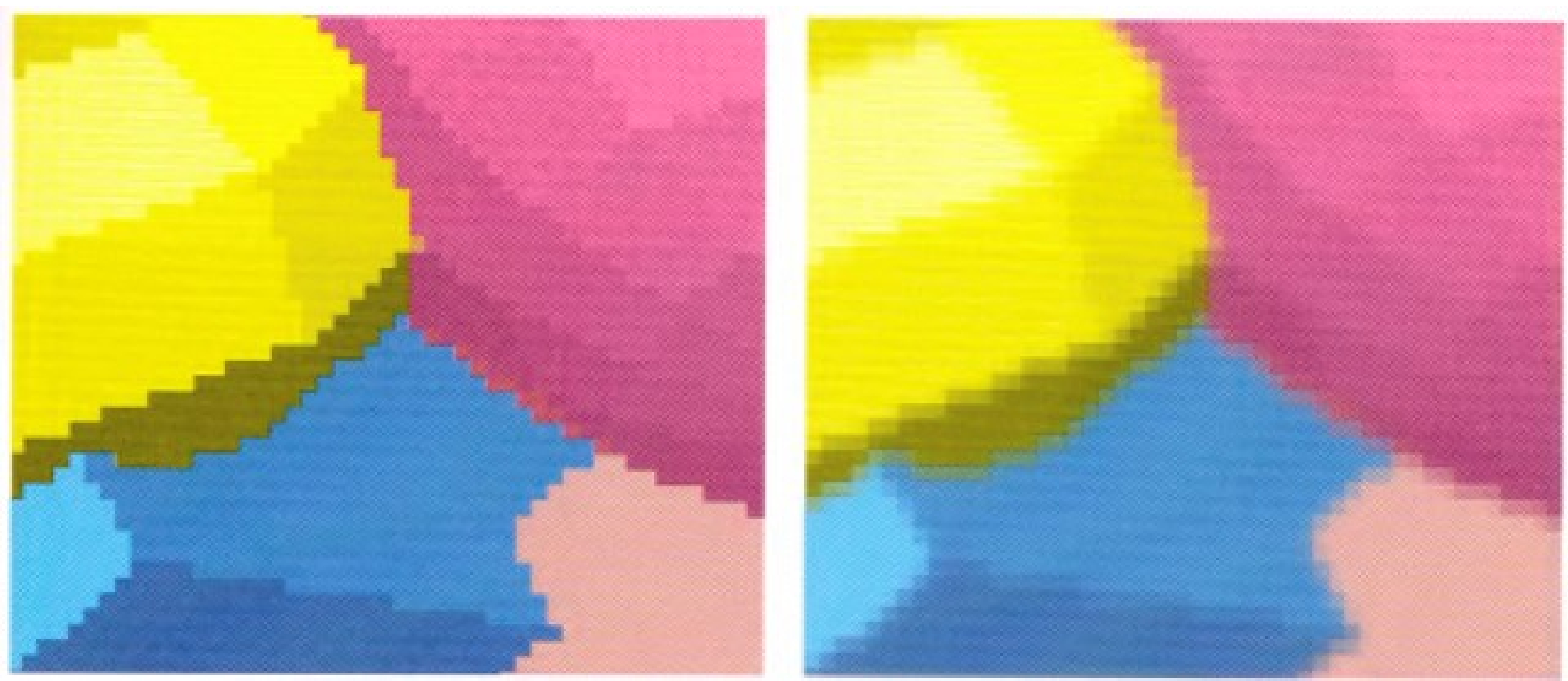
Full-scene Antialiasing

- Average multiple frames rendered with offsets (= multi-sampling per pixel)

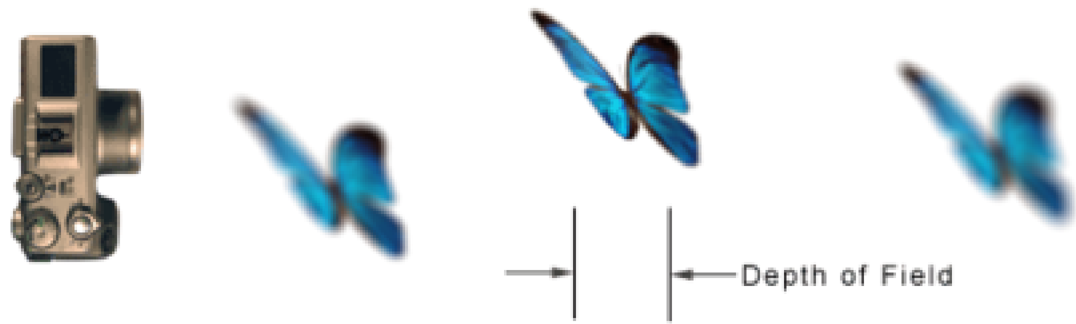


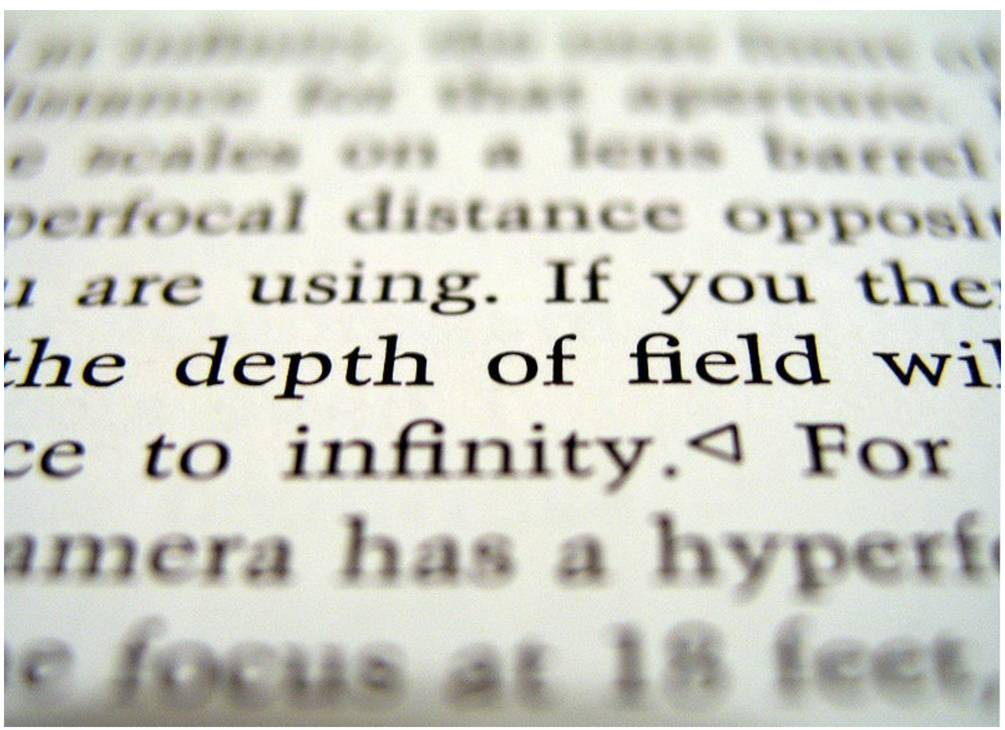
Full-scene Antialiasing

- Average multiple frames rendered with offsets (= multi-sampling per pixel)



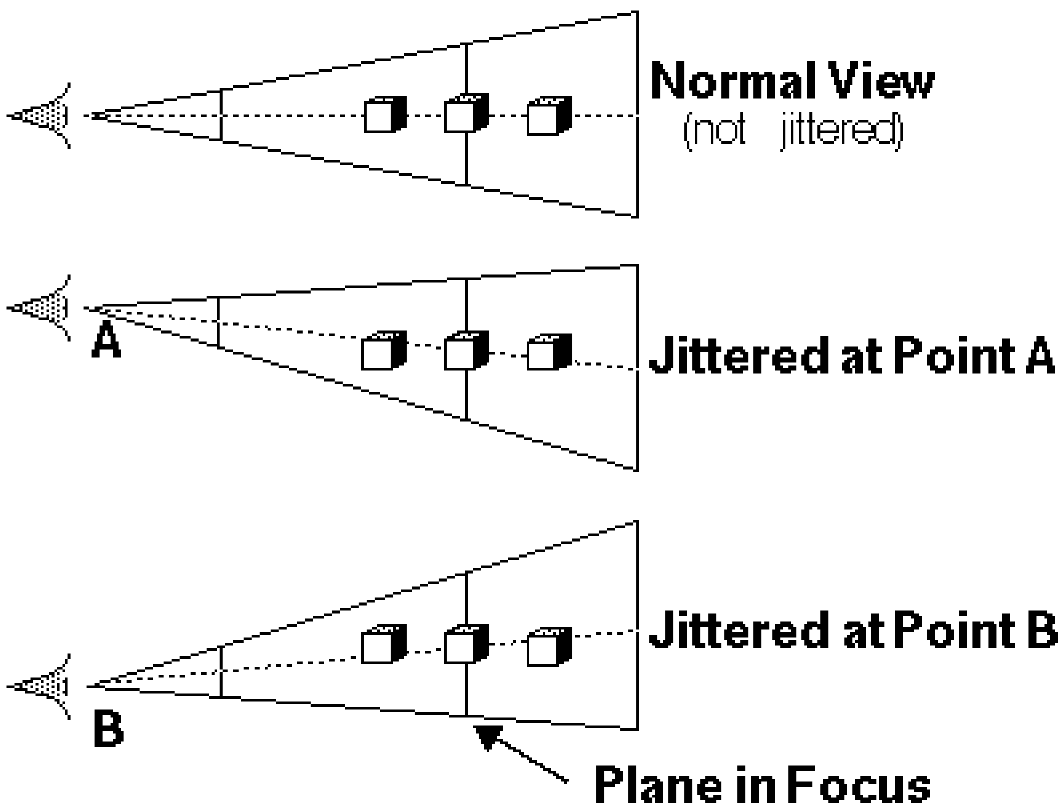
Depth of Field



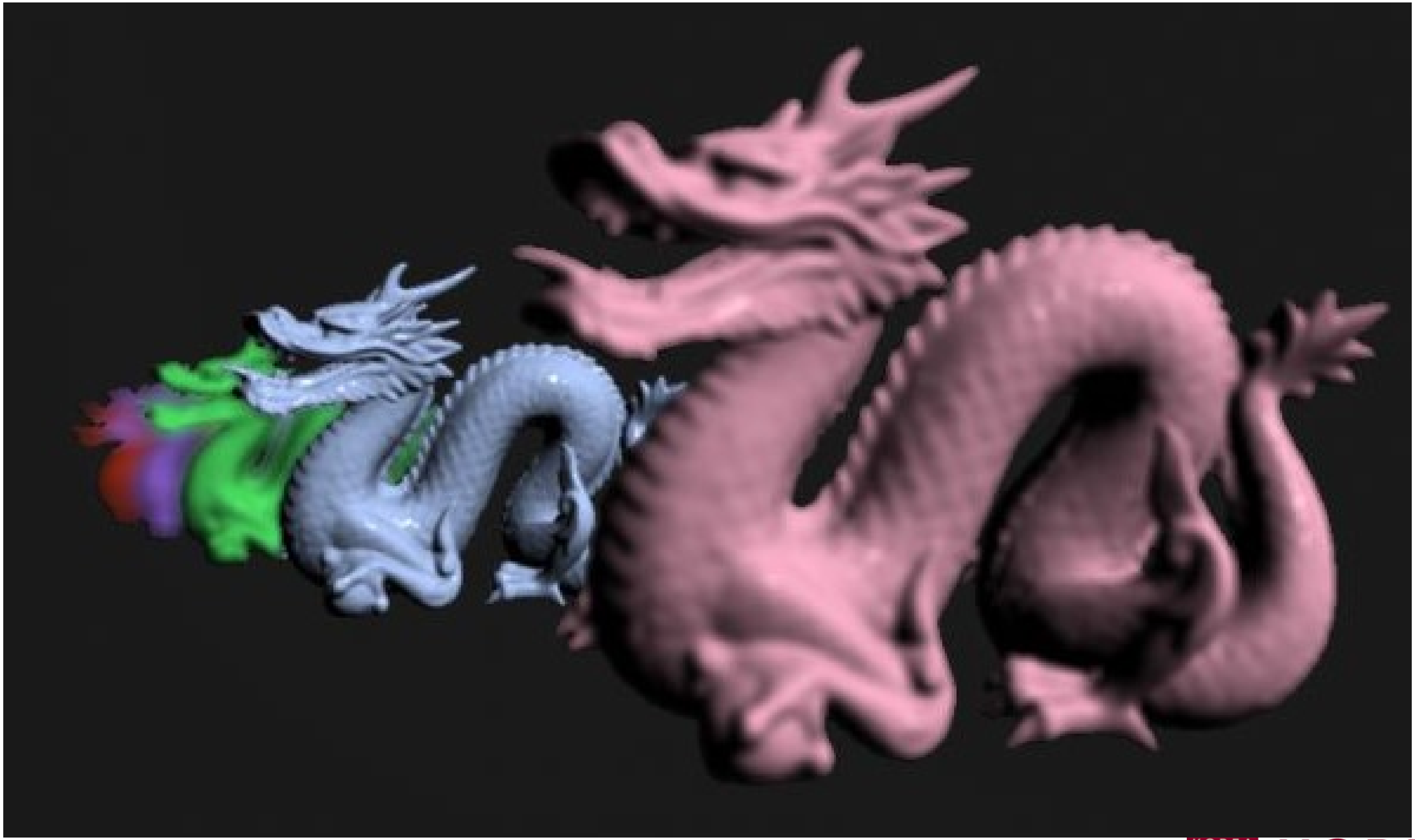


KOREA UNIVERSITY

Depth of Field

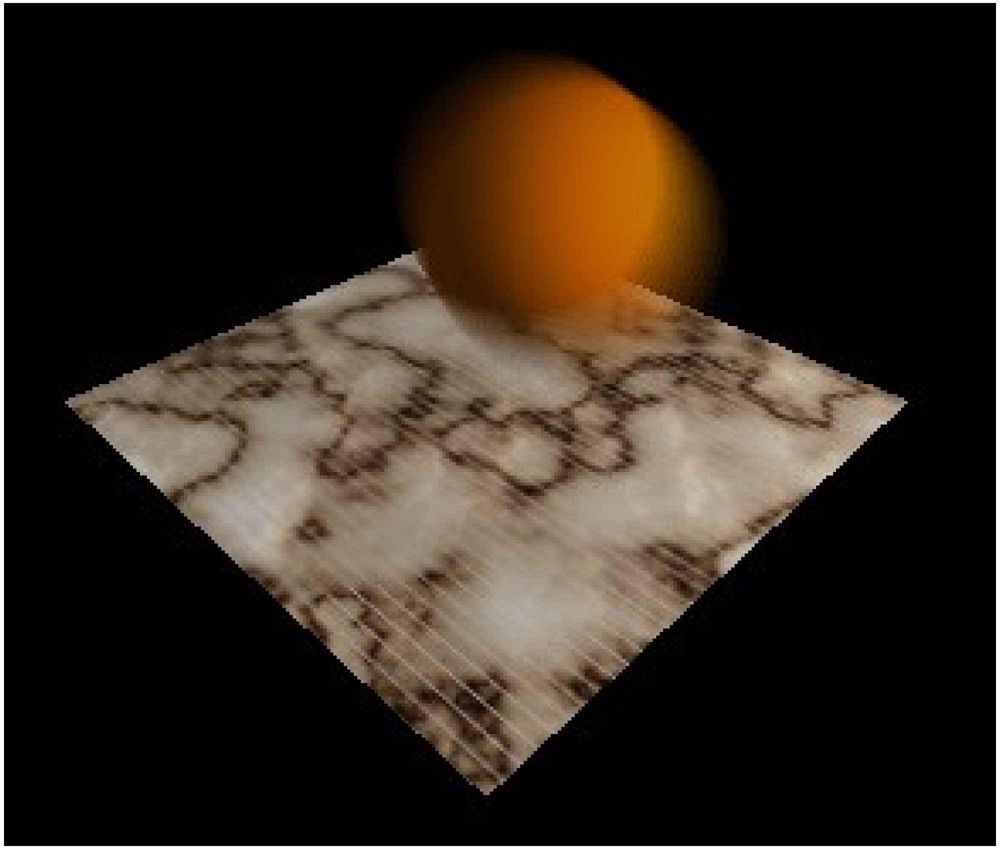


Depth of Field

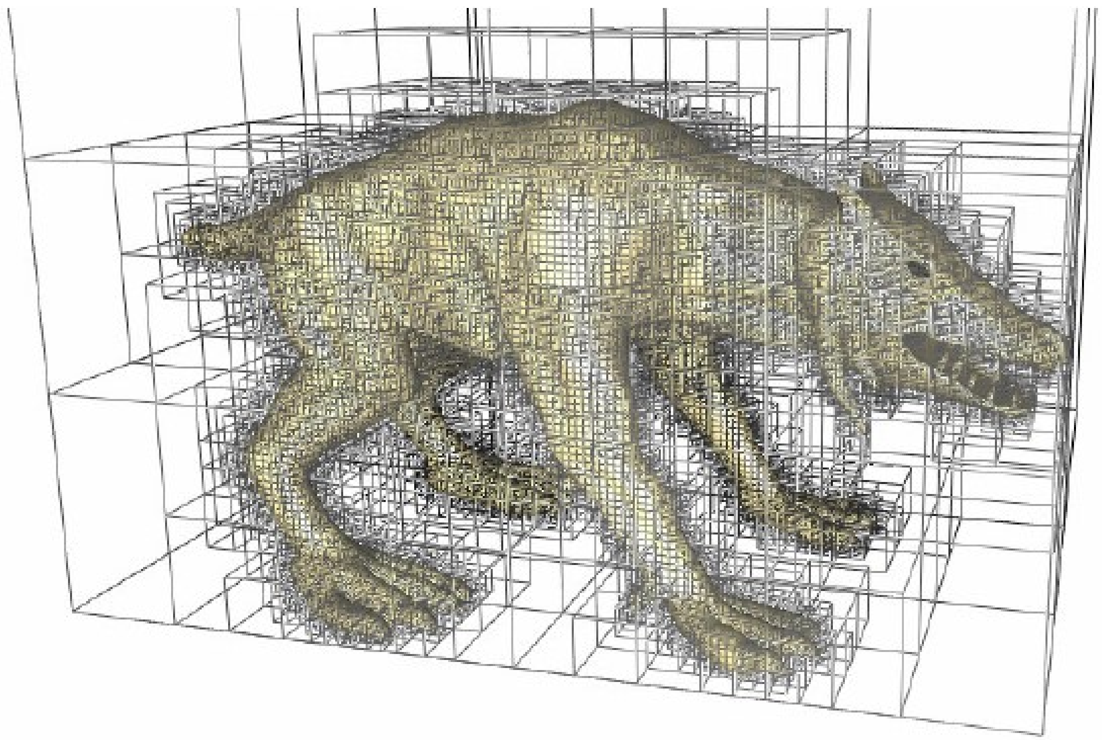


Motion Blur

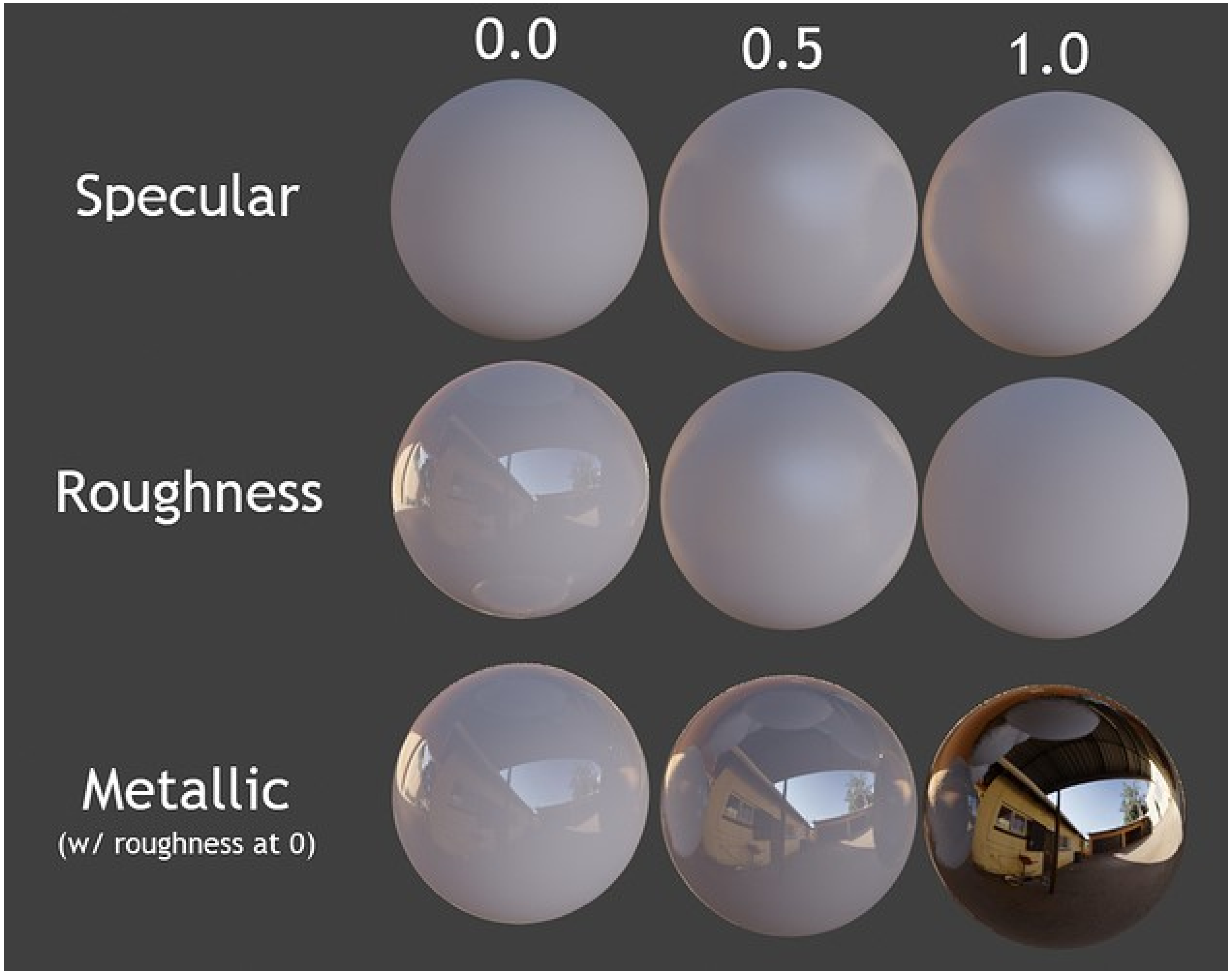
- Fast moving object



Questions?



Octree Textures


0.0
0.5
1.0
Specular
Roughness
Metallic
(w/ roughness at 0)