

CSCI 420 Computer Graphics

Lecture 4

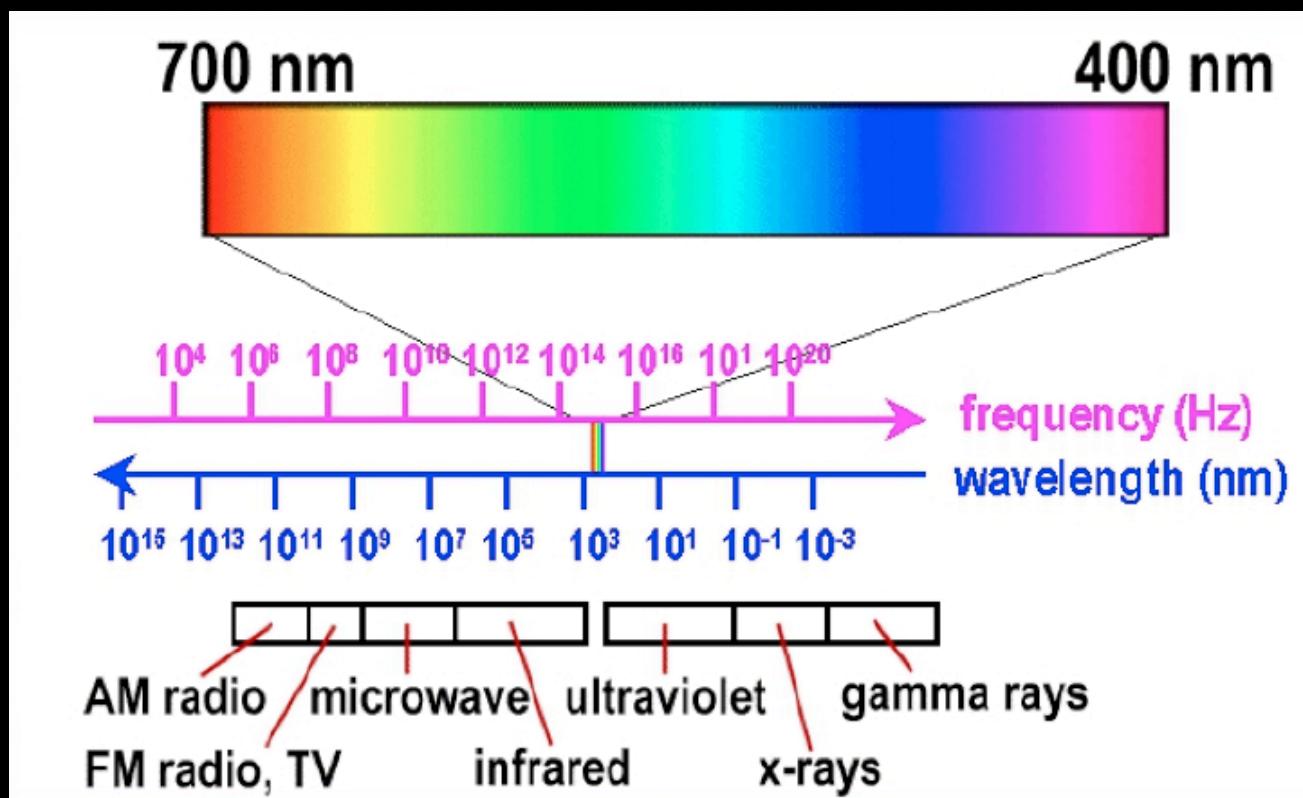
Color and Hidden Surface Removal

- Client/Server Model
- Callbacks
- Double Buffering
- Physics of Color
- Flat vs Smooth Shading
- Hidden Surface Removal
- [Angel Ch. 2]

Oded Stein
University of Southern California

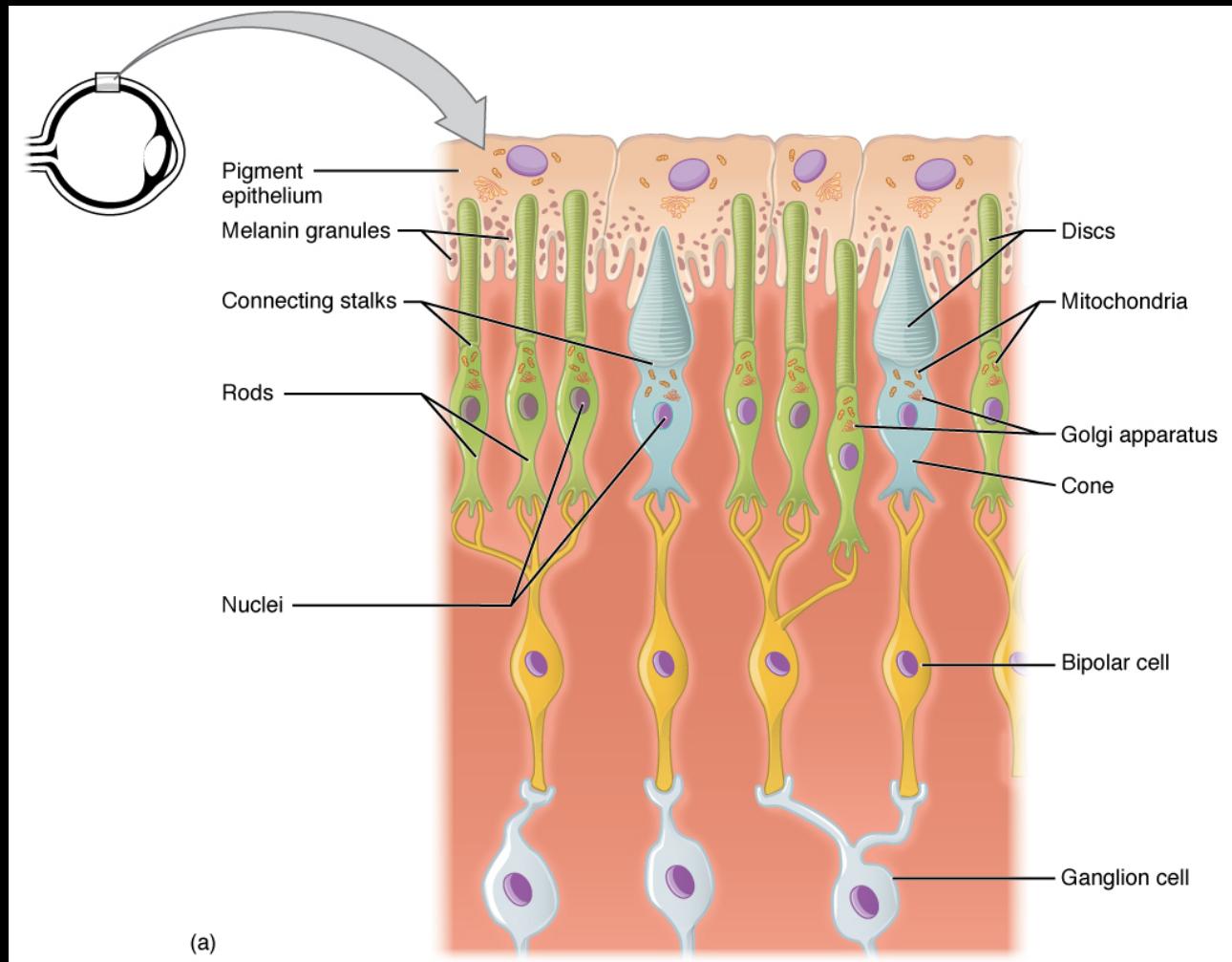
Physics of Color

- Electromagnetic radiation
- Can see only a tiny piece of the spectrum



Physics of Color

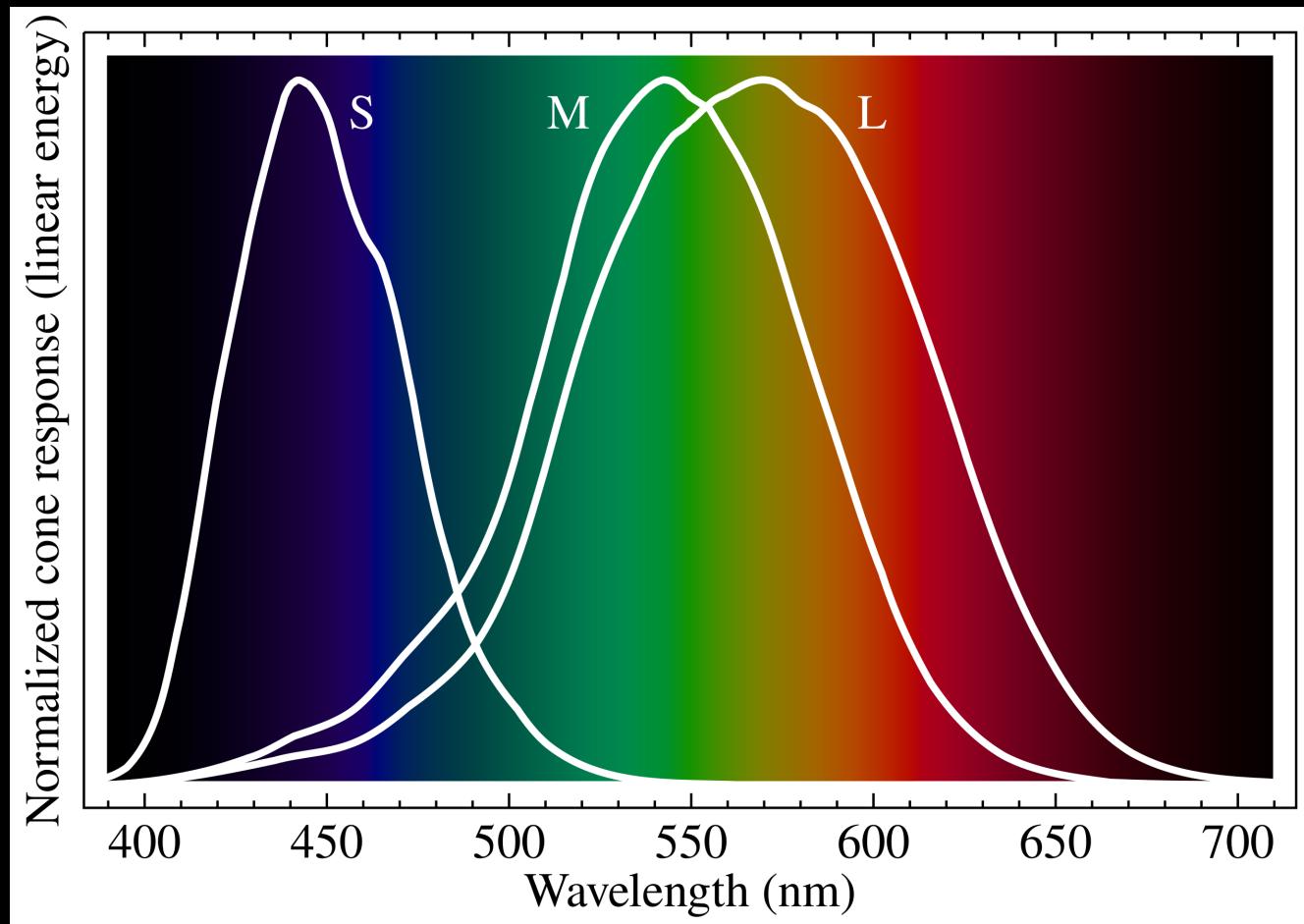
- "rods" see contrast
- "cones" see color



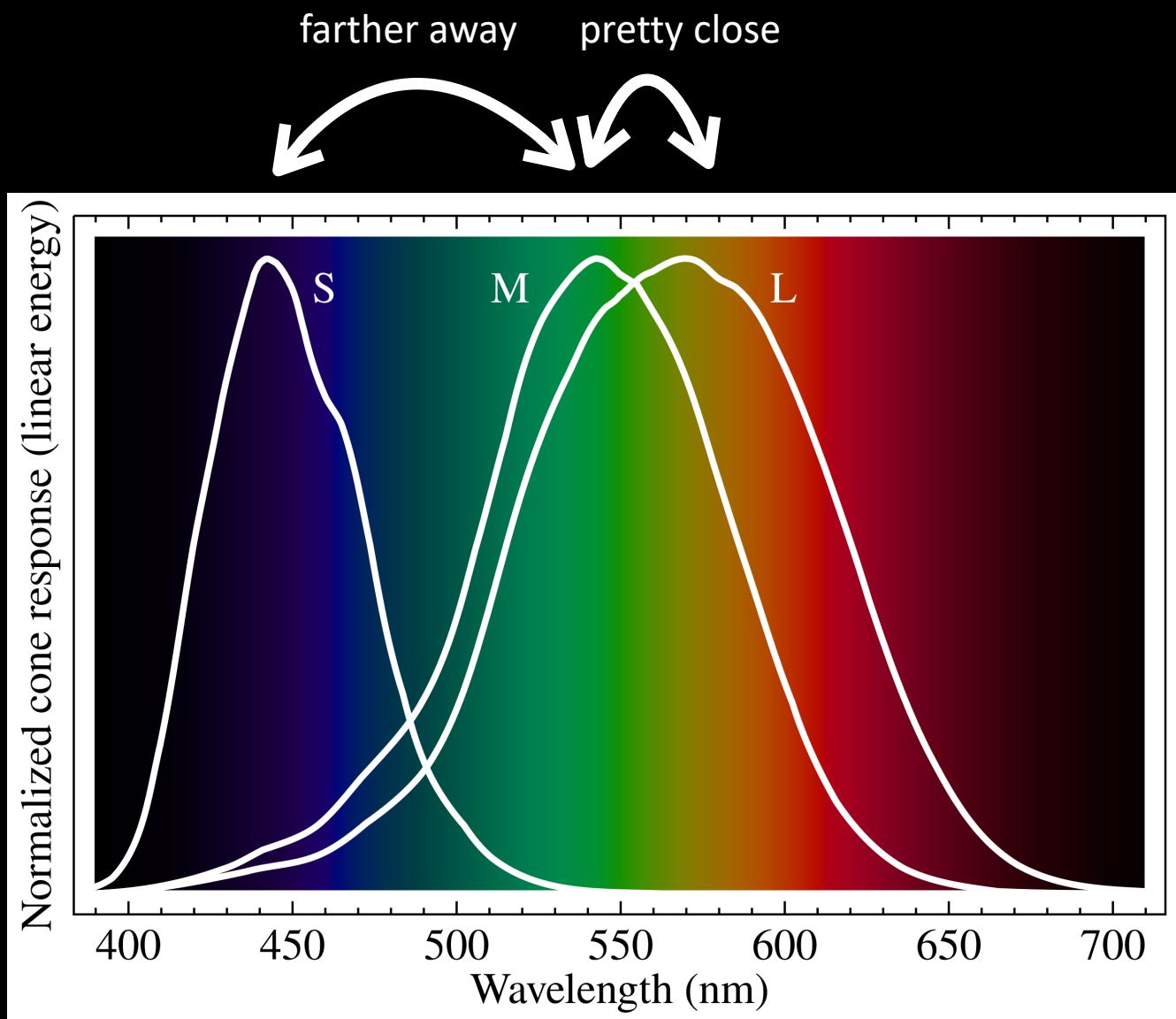
[https://
commons.wikimedia
.org/wiki/
File:1414_Rods_and
_Cones.jpg](https://commons.wikimedia.org/wiki/File:1414_Rods_and_Cones.jpg)

Color Filters

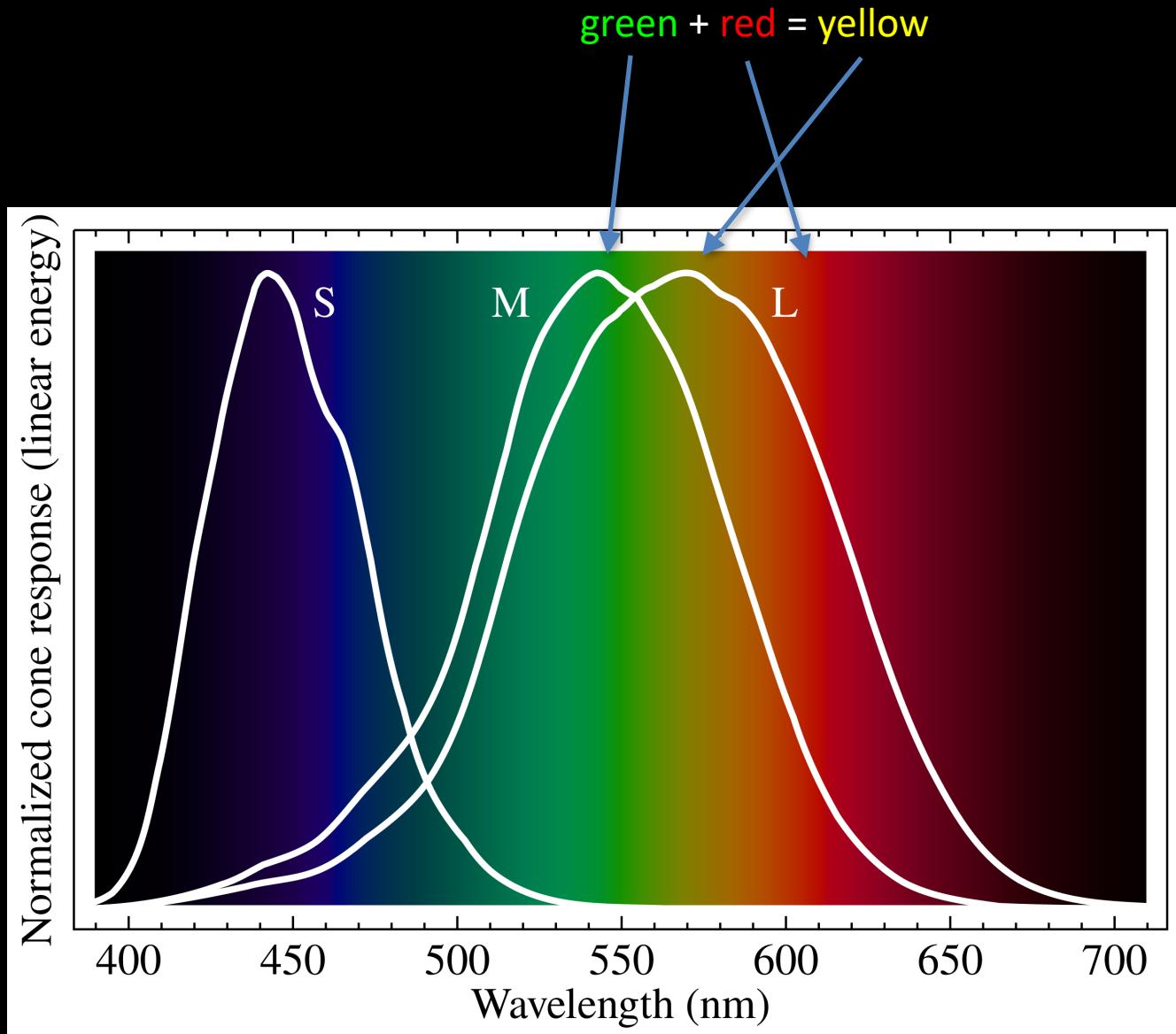
- Eye can perceive only 3 basic colors
- Computer screens designed accordingly



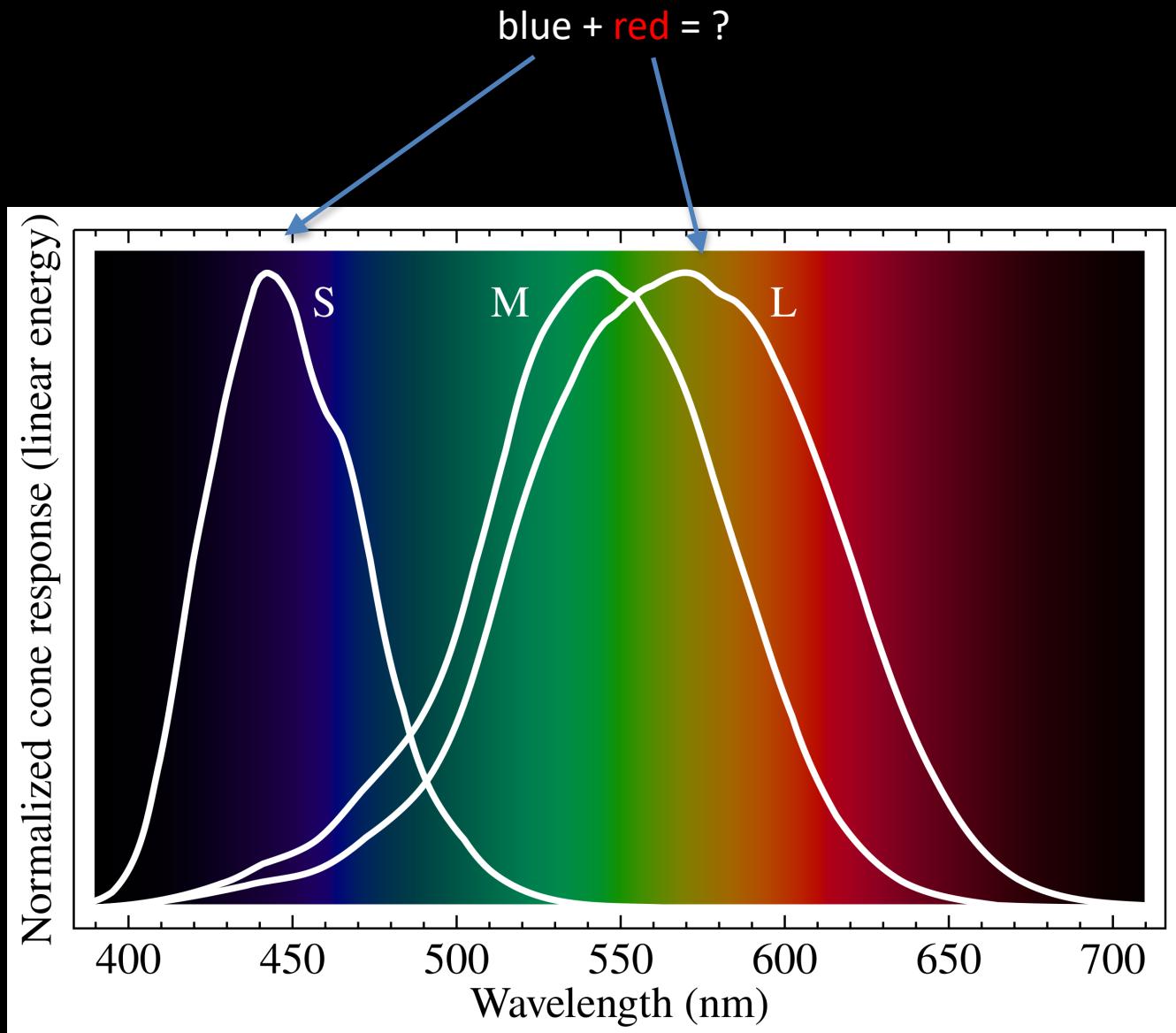
Color Filters



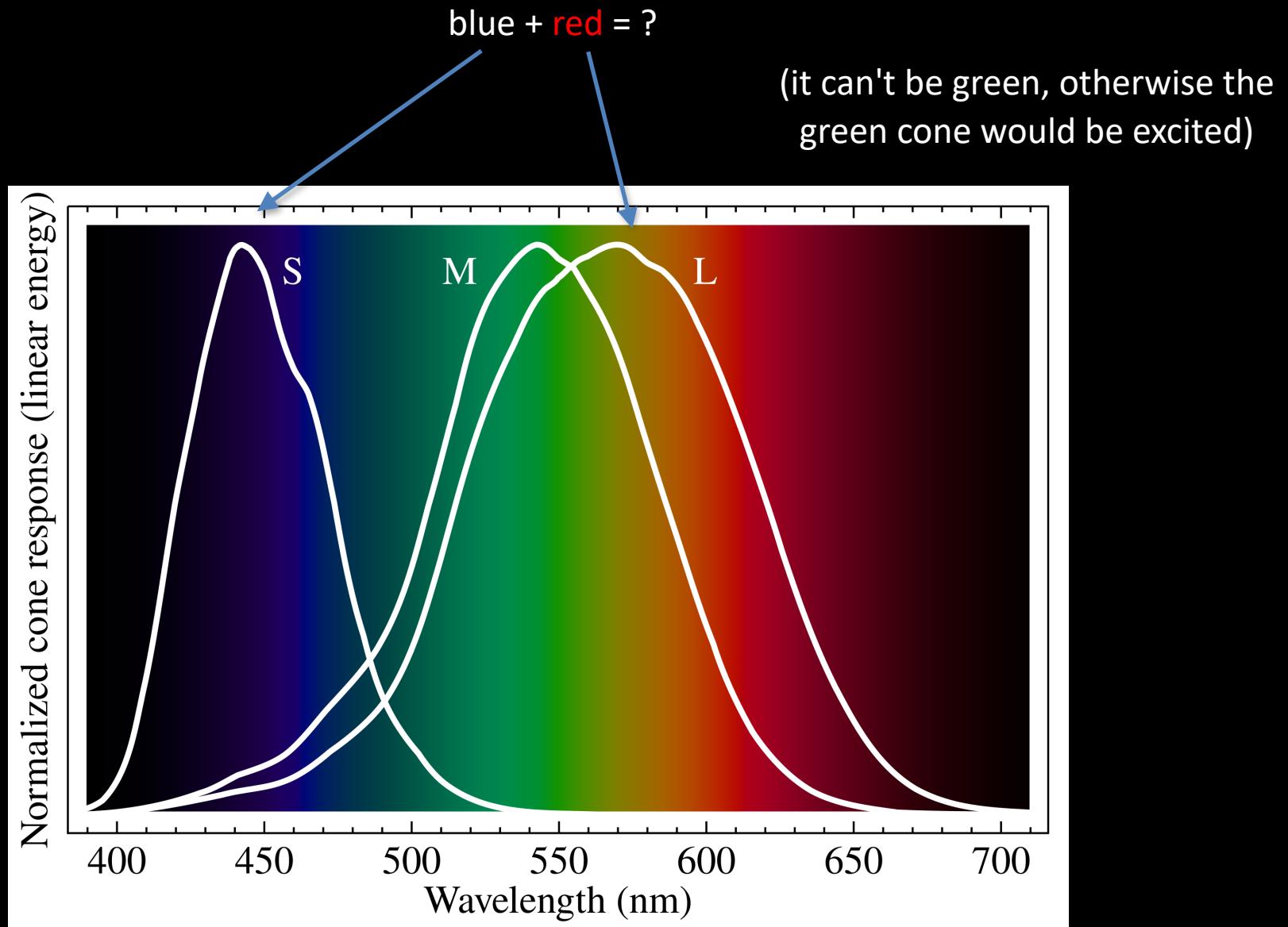
Fake colors exist



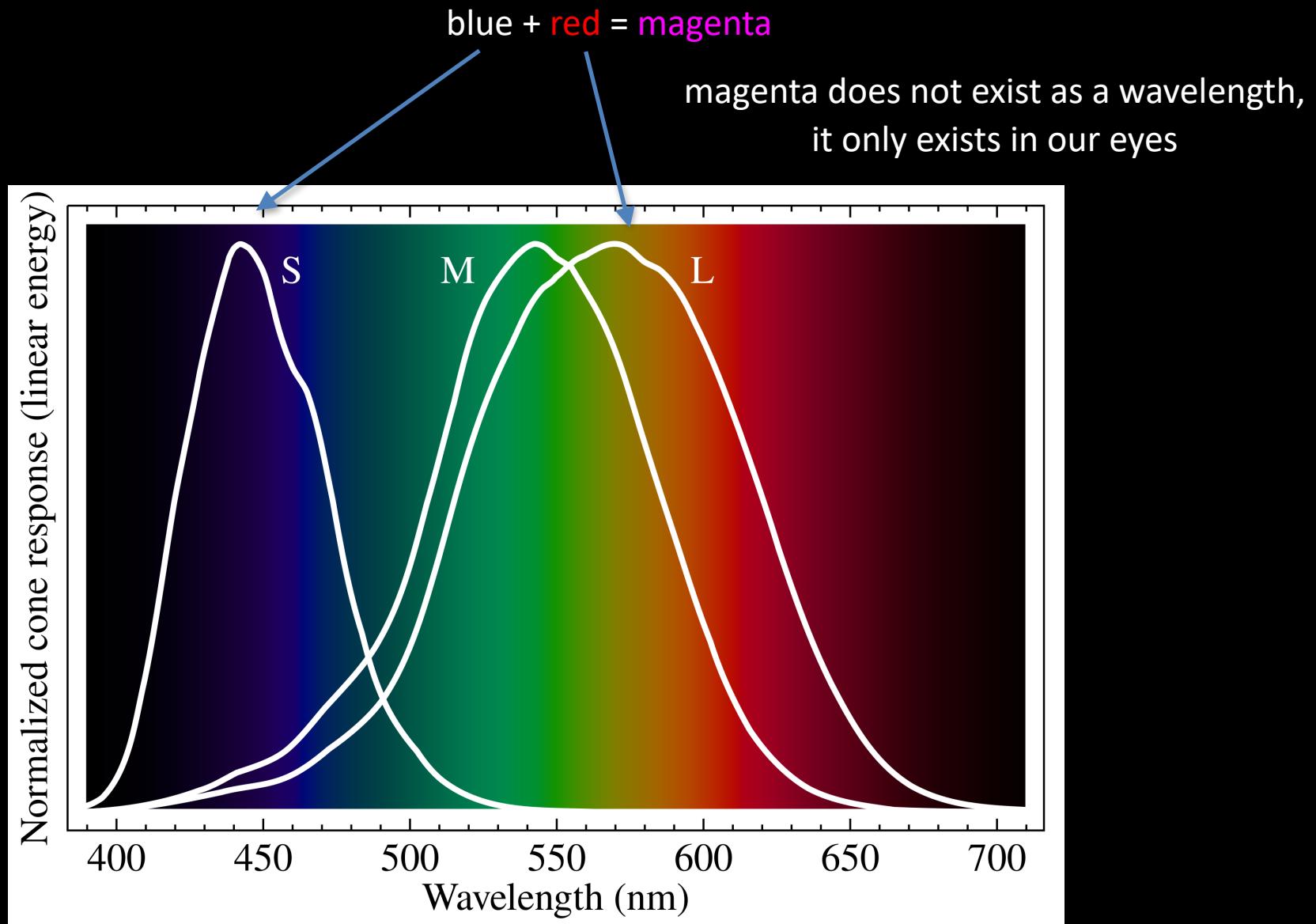
Fake colors exist



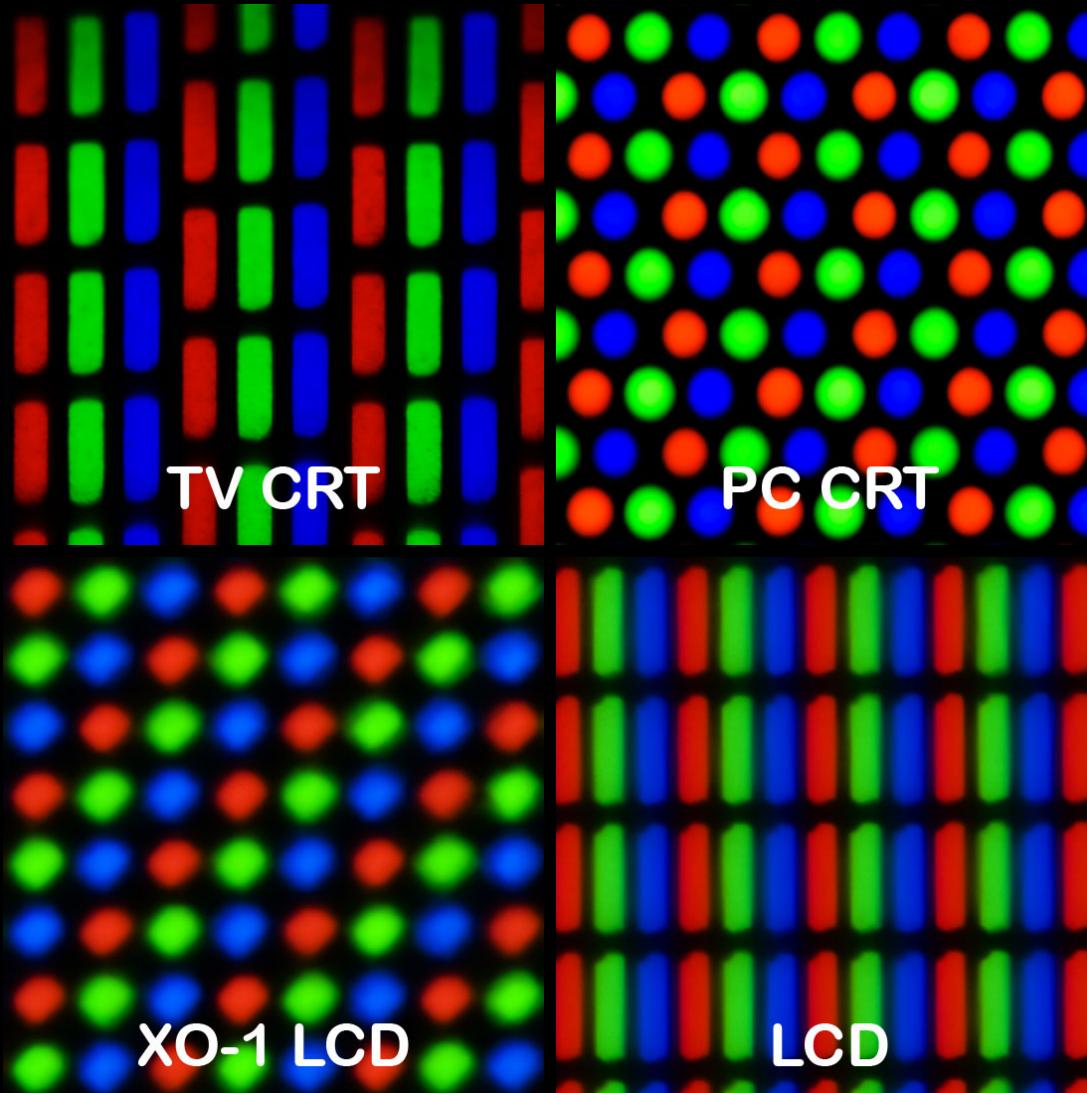
Fake colors exist



Fake colors exist



Pixel Geometry

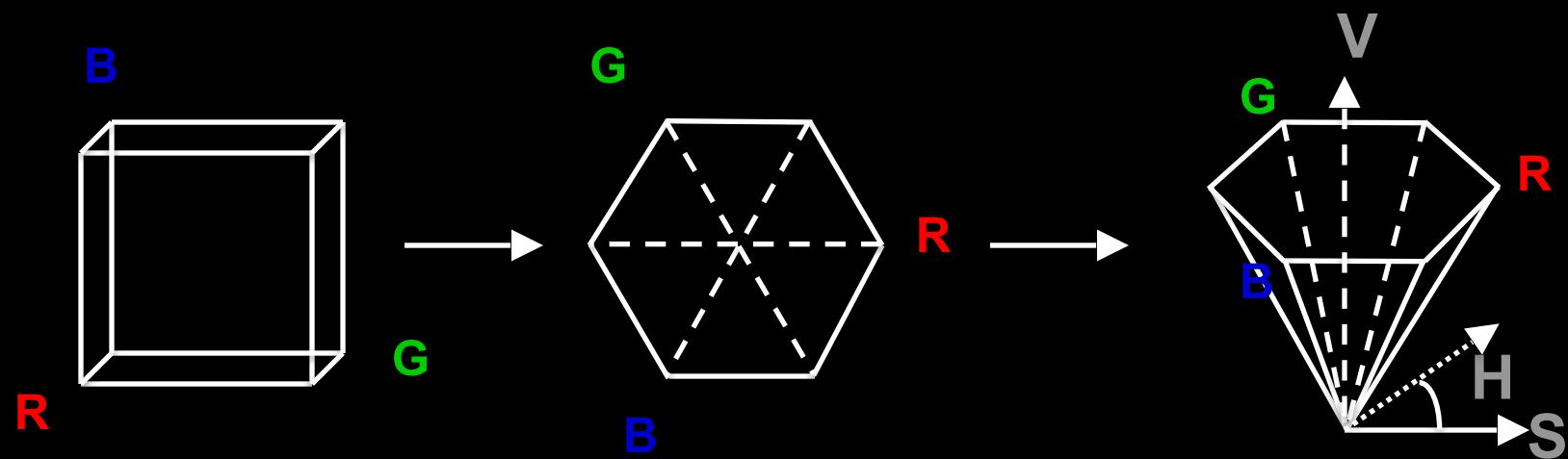
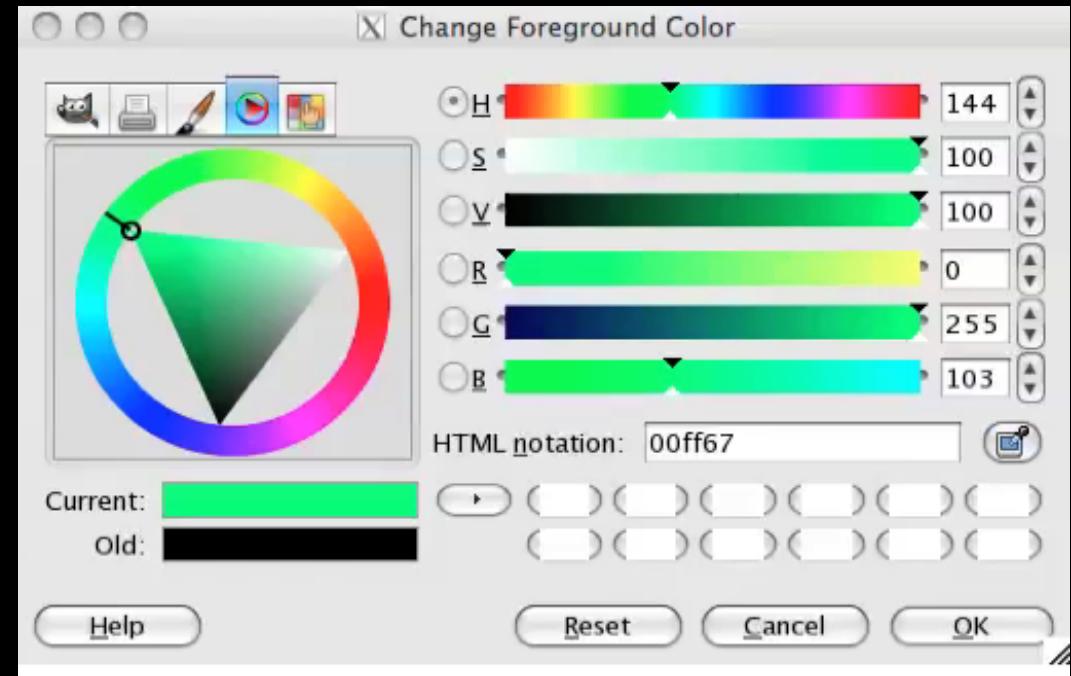


Color Spaces

- RGB (Red, Green, Blue)
 - Convenient for display
 - Can be unintuitive (3 floats in OpenGL)
- HSV (Hue, Saturation, Value)
 - Hue: what color
 - Saturation: how far away from gray
 - Value: how bright
- Other formats for movies and printing

RGB vs HSV

Gimp Color Picker

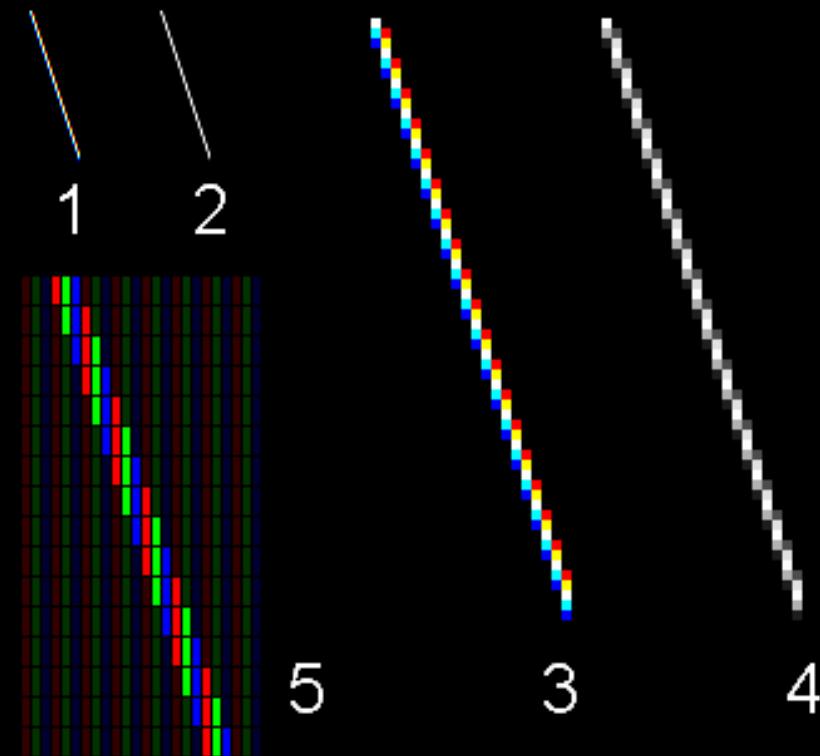


Screens output only color

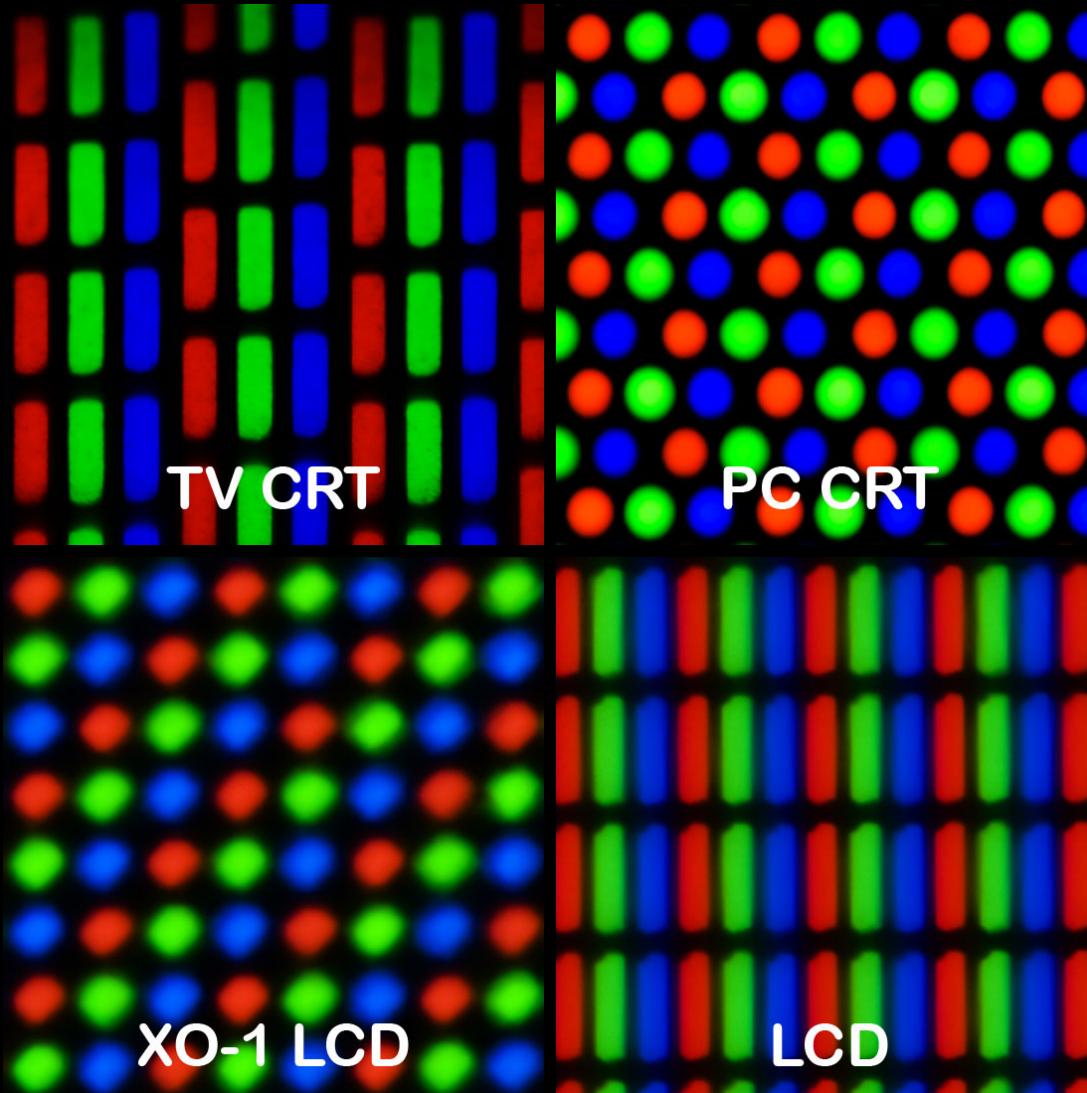
- Your job is to decide what color each pixel of the framebuffer has.
- You set a background color
- Each primitive you draw will generate fragments that overlay color onto the background
 - Fragments are pixels, but not really.
- All the geometry that went in is only colors in the end.

Colors can be used for so many things

- You can implement lots of effects with color that go beyond just coloring an object.
 - subpixel effects
 - ClearType Antialiasing



Pixel Geometry

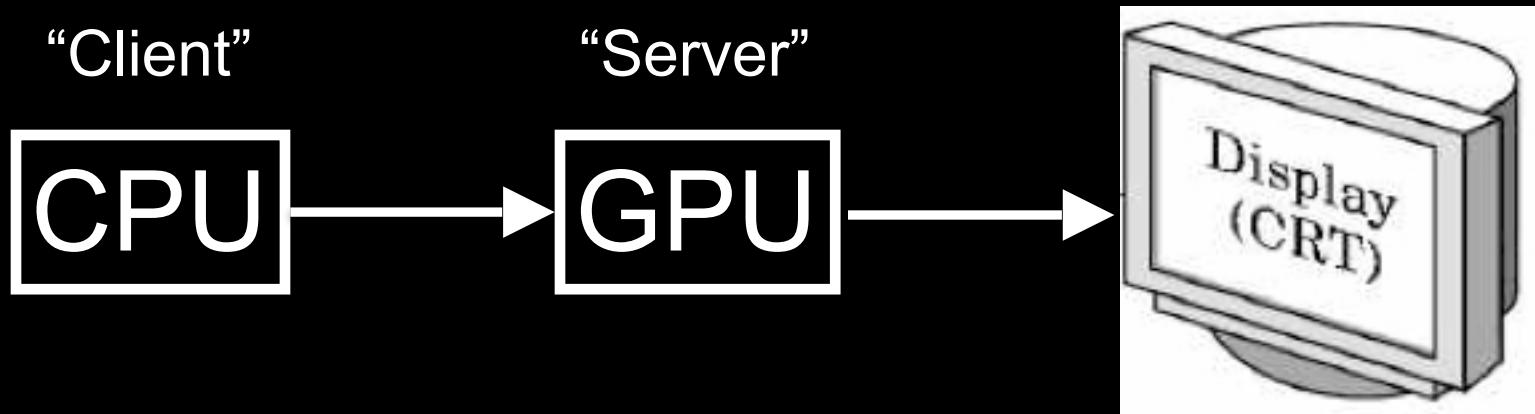


How does color work in our pipeline?

- Color is the final output data of the pipeline.
 - Geometry goes in, color goes out.
- If we want to associate colors to specific vertices (or primitives), we need to pass color as an attribute through the pipeline.
- Many effects have to be implemented by just manipulating colors.
 - Lighting

Remember: Client/Server Model

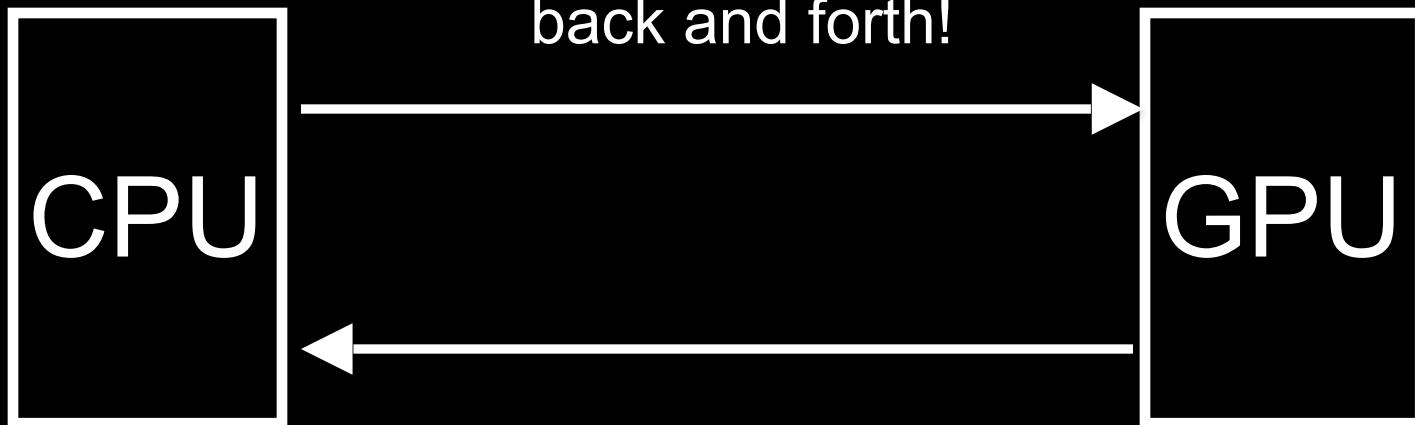
- Graphics hardware and caching



- Important for efficiency
- Need to be aware where data are stored
- Graphics driver code is on the CPU
- Rendering resources (buffers, shaders, textures, etc.) are on the GPU

The CPU-GPU bus

try to limit how often
you shuttle data
back and forth!

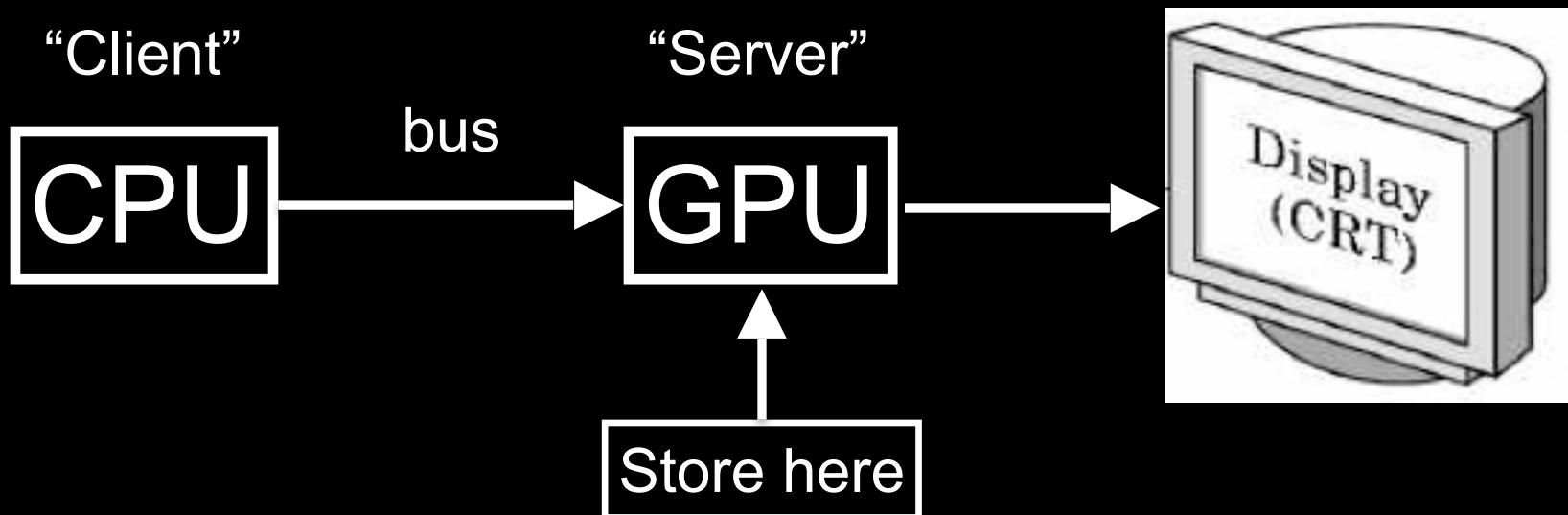


try to limit how
much data you
shuttle back and
forth!

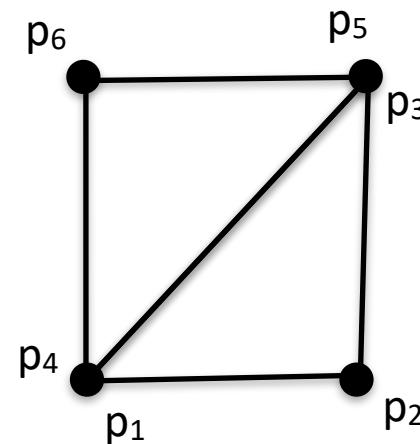


Buffer Objects

- Store rendering data: vertex positions, normals, texture coordinates, colors, vertex indices, etc.
- Optimize and store on server (GPU)



Let's write a quad into the buffer



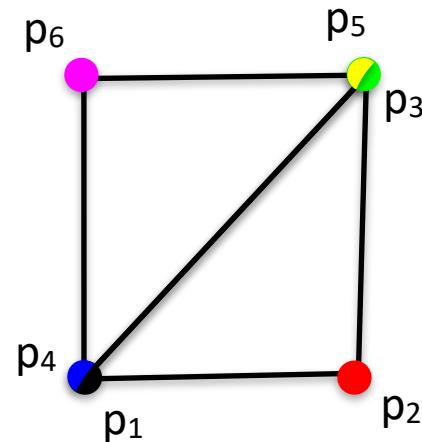
$$p_1 = p_4 = (-1, -1)$$

$$p_2 = (1, -1)$$

$$p_3 = p_5 = (1, 1)$$

$$p_6 = (-1, 1)$$

Our quad now has colors



$$p_1 = p_4 = (-1, -1)$$

$$p_2 = (1, -1)$$

$$p_3 = p_5 = (1, 1)$$

$$p_6 = (-1, 1)$$

Vertex Buffer Objects

- Caches vertex geometric data: positions, normals, texture coordinates, colors
- Optimize and store on server (GPU)
- Required for core OpenGL profile

```
/* vertices of the quad (will form two triangles;  
   rendered via GL_TRIANGLES) */  
  
float positions[6][3] =  
{{{-1.0, -1.0, -1.0}, {1.0, -1.0, -1.0}, {1.0, 1.0, -1.0},  
  {-1.0, -1.0, -1.0}, {1.0, 1.0, -1.0}, {-1.0, 1.0, -1.0}}};  
  
/* colors to be assigned to vertices (but why is it 4 values?) */  
  
float colors[6][4] =  
{{{0.0, 0.0, 0.0, 1.0}, {1.0, 0.0, 0.0, 1.0}, {0.0, 1.0, 0.0, 1.0},  
  {0.0, 0.0, 1.0, 1.0}, {1.0, 1.0, 0.0, 1.0}, {1.0, 0.0, 1.0, 1.0}}}; 22
```

Vertex Buffer Objects

- Caches vertex geometric data: positions, normals, texture coordinates, colors
- Optimize and store on server (GPU)
- Required for core OpenGL profile

```
/* vertices of the quad (will form two triangles;  
   rendered via GL_TRIANGLES) */  
  
float positions[6][3] =  
{{{-1.0, -1.0, -1.0}, {1.0, -1.0, -1.0}, {1.0, 1.0, -1.0},  
  {-1.0, -1.0, -1.0}, {1.0, 1.0, -1.0}, {-1.0, 1.0, -1.0}}};  
  
/* colors to be assigned to vertices (4th value is the alpha channel) */  
  
float colors[6][4] =  
{{{0.0, 0.0, 0.0, 1.0}, {1.0, 0.0, 0.0, 1.0}, {0.0, 1.0, 0.0, 1.0},  
  {0.0, 0.0, 1.0, 1.0}, {1.0, 1.0, 0.0, 1.0}, {1.0, 0.0, 1.0, 1.0}}}; 23
```

Vertex Buffer Objects

- You know the drill by now...
 - glGenBuffers
 - glBindBuffer
 - glBufferData
- But we're using C++, so you can write wrapper code to do all of this for you!
- Homework contains VBO and VAO classes to help you with this

Vertex Buffer Object: Initialization

(here's what's not to do)

(and what you will find in the HW code)

```
int numVertices = 6;  
VBO * vboVertices;  
VBO * vboColors;  
  
void initVBOs()  
{  
    // 3 values per vertex, namely x,y,z coordinates  
    vboVertices = new VBO(numVertices, 3, positions, GL_STATIC_DRAW);  
  
    // 4 values per vertex, namely r,g,b,a channels  
    vboColors = new VBO(numVertices, 4, colors, GL_STATIC_DRAW);  
}
```

Vertex Buffer Object: Initialization

- It's 2024 - use a smart pointer

```
int numVertices = 6;
std::unique_ptr<VBO> vboVertices;
std::unique_ptr<VBO> vboColors;

void initVBOs()
{
    // 3 values per vertex, namely x,y,z coordinates
    vboVertices = std::make_unique<VBO>(numVertices, 3, positions,
                                         GL_STATIC_DRAW);

    // 4 values per vertex, namely r,g,b,a channels
    vboColors = std::make_unique<VBO>(numVertices, 4, colors,
                                       GL_STATIC_DRAW);
}
```

Vertex Buffer Object: Initialization

- (If it was up to me, I'd just put it all on the stack...)
- (But that is a bit difficult with the way OpenGL programs traditionally like global state.)

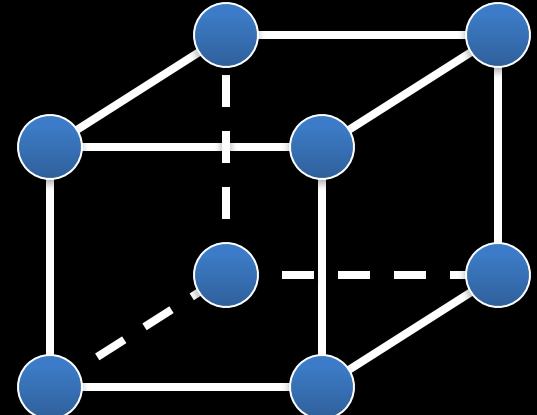
```
int numVertices = 6;  
// 3 values per vertex, namely x,y,z coordinates  
VBO vboVertices(numVertices, 3, positions, GL_STATIC_DRAW);  
// 4 values per vertex, namely r,g,b,a channels  
VBO vboColors(numVertices, 4, colors, GL_STATIC_DRAW);
```

Element Arrays (meshes)

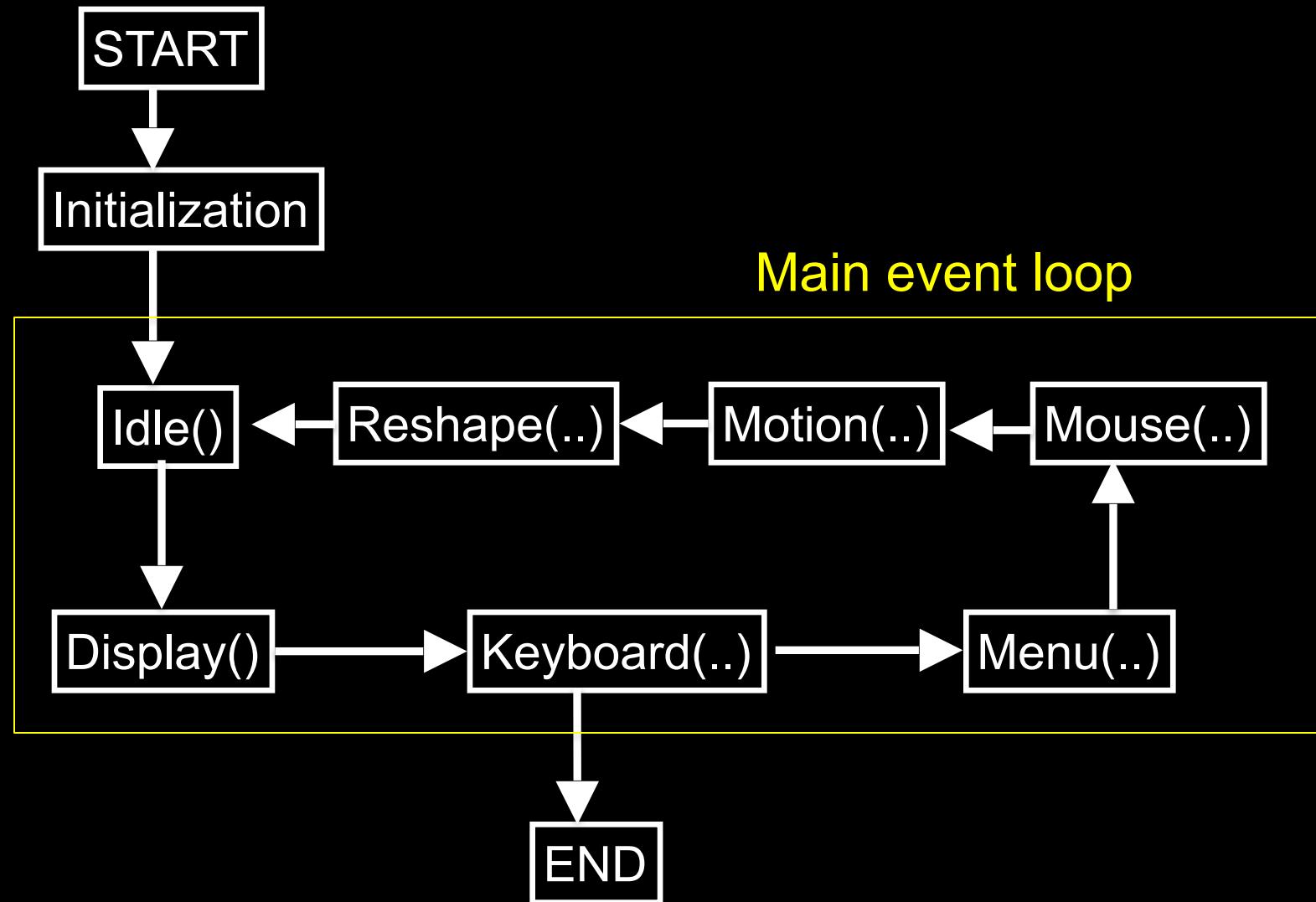
- Draw cube with $6*2*3=36$ or with 8 vertices?
- Expense in drawing and transformation
- Triangle strips help to some extent
- Element arrays provide general solution
- Define (transmit) array of vertices, colors, normals
- Draw using index into array(s) :

```
// (must first set up the GL_ELEMENT_ARRAY_BUFFER) ...
glDrawElements(GL_TRIANGLES, 36, GL_UNSIGNED_INT, 0);
```

- Vertex sharing for efficient operations
- Extra credit for first assignment



GLUT Program with Callbacks



Main Event Loop

- Standard technique for interaction
(GLUT, Qt, wxWidgets, ...)
- Main loop processes events
- Dispatch to functions specified by client
- Callbacks also common in operating systems
- “Poor man’s functional programming”

Types of Callbacks

- Display () : when window must be drawn
- Idle () : when no other events to be handled
- Keyboard (unsigned char key, int x, int y) : key pressed
- Menu (...) : after selection from menu
- Mouse (int button, int state, int x, int y) : mouse button
- Motion (...) : mouse movement
- Reshape (int w, int h) : window resize
- Any callback can be NULL

Screen Refresh

- Common: 60-100 Hz
- Flicker if drawing overlaps screen refresh
- Problem during animation
- Solution: use two separate **frame buffers**
(double buffering):
 - Draw into one buffer
 - Swap and display, while drawing into other buffer
- Desirable frame rate ≥ 30 fps (frames/second)

Enabling Single/Double Buffering

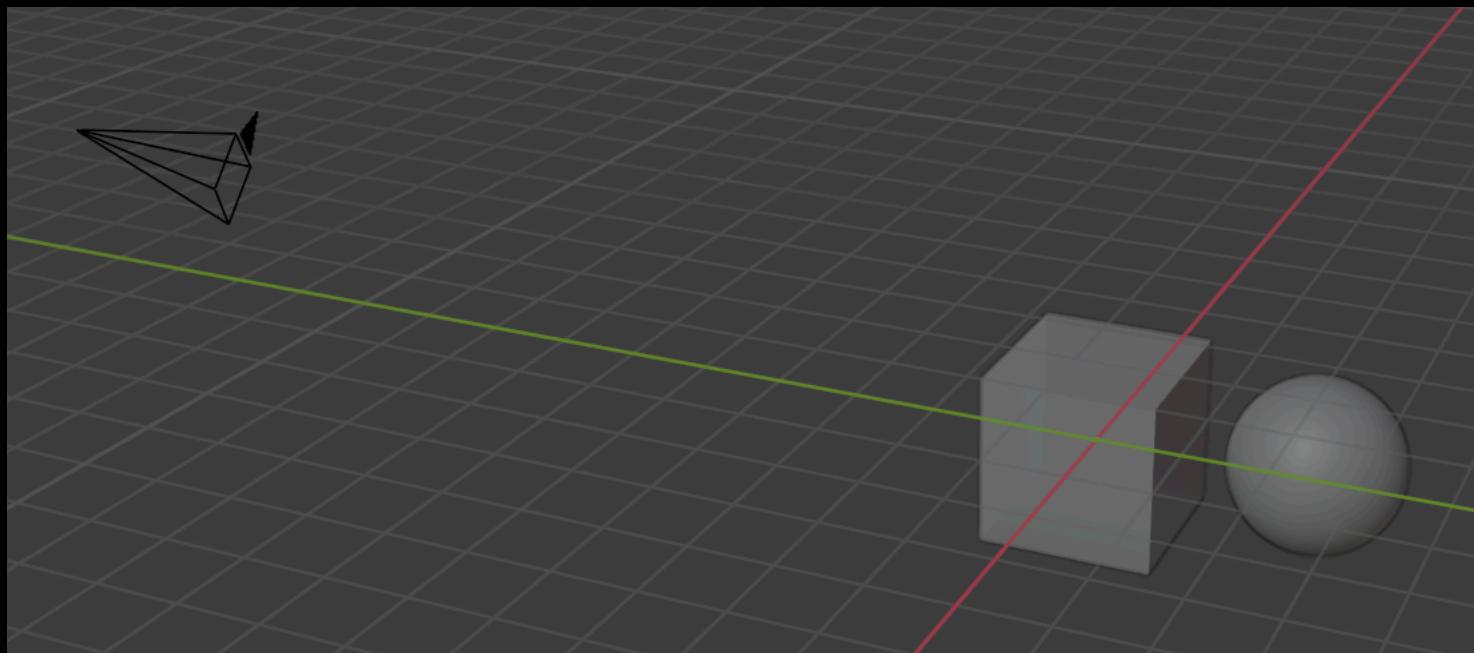
- `glutInitDisplayMode(GLUT_SINGLE);`
- `glutInitDisplayMode(GLUT_DOUBLE);`
- **Single buffering:**
Must call `glFinish()` at the end of `Display()`
- **Double buffering:**
Must call `glutSwapBuffers()` at the end of `Display()`
- Must call `glutPostRedisplay()` at the end of `Idle()`
- If something in OpenGL has no effect or does not work, check the modes in `glutInitDisplayMode`

Hidden Surface Removal

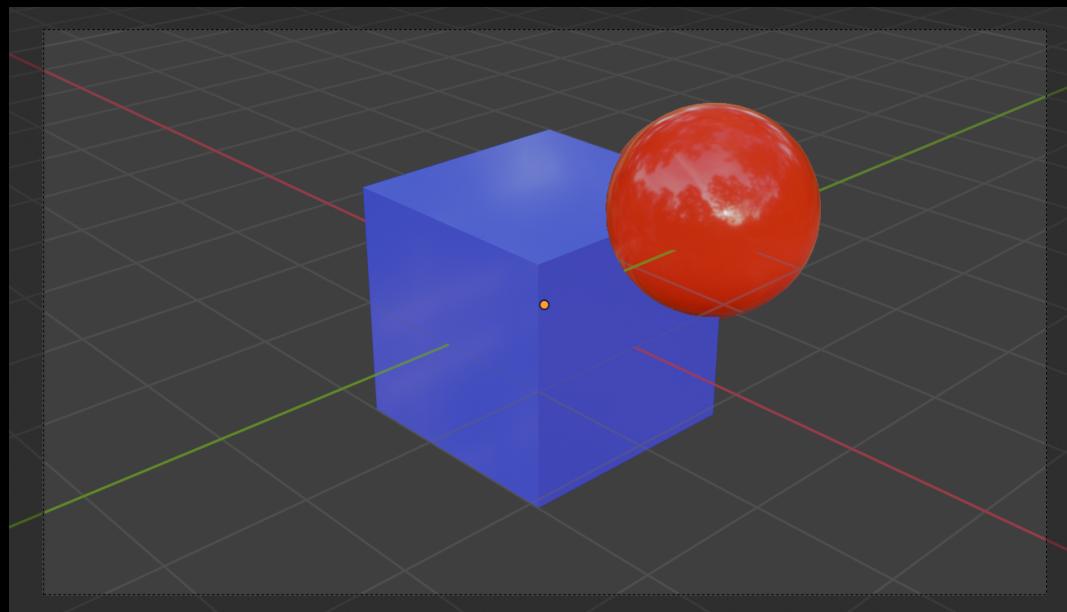
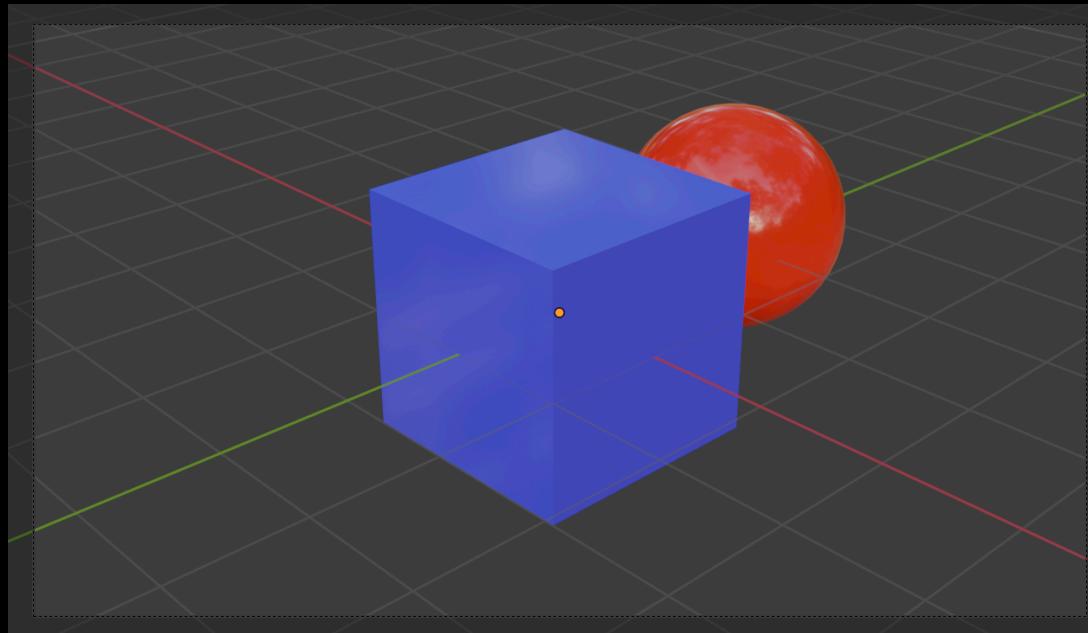
- Classic problem of computer graphics
- What is visible after clipping and projection?
- Object-space vs image-space approaches
- Object space: depth sort (Painter's algorithm)
- Image space: *z-buffer* algorithm
- Related: back-face culling

Hidden Surface Removal

- You might think that the rasterizer's job is as easy as just "turn all triangles into pixels"
- What should happen to these 2 objects in front of the camera?

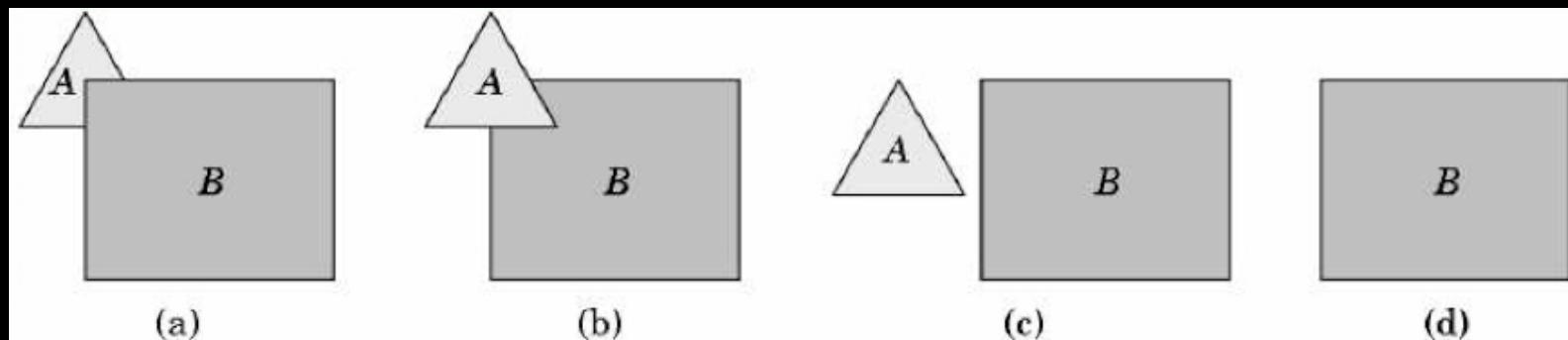


Hidden Surface Removal



Object-Space Approach

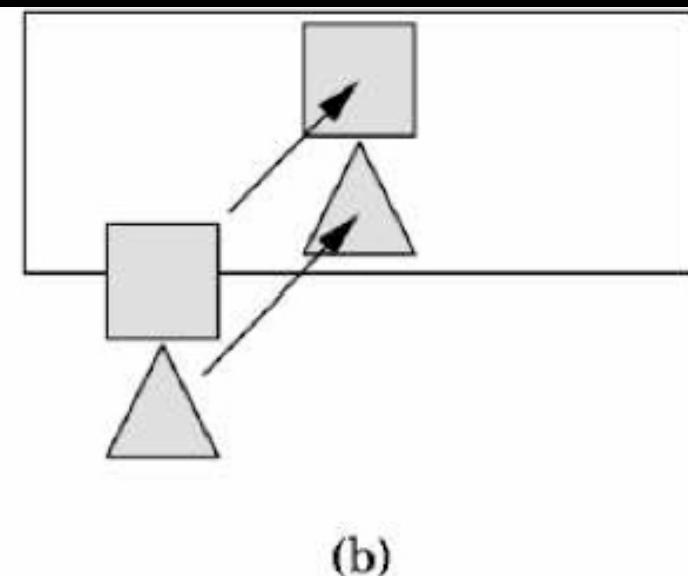
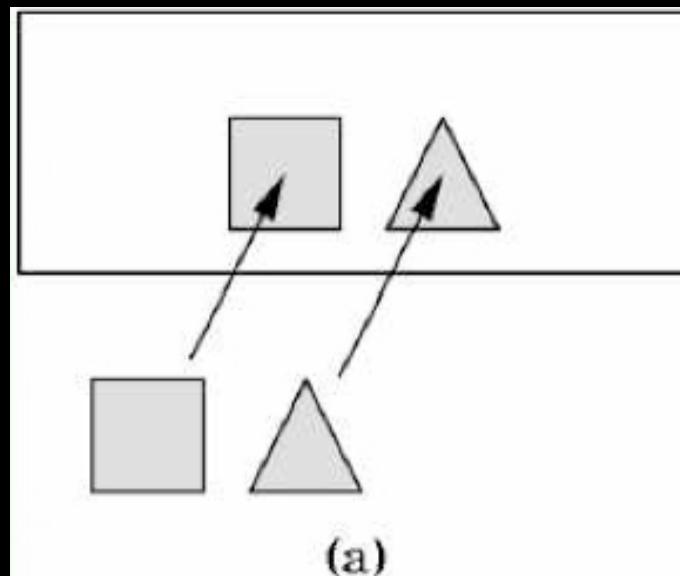
- Consider objects pairwise



- Painter's algorithm: render back-to-front
- “Paint” over invisible polygons
- How to sort and how to test overlap?

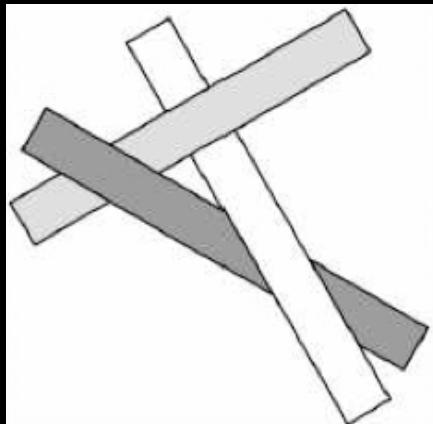
Depth Sorting

- First, sort by furthest distance z from viewer
- If minimum depth of A is greater than maximum depth of B, A can be drawn before B
- If either x or y extents do not overlap, A and B can be drawn independently

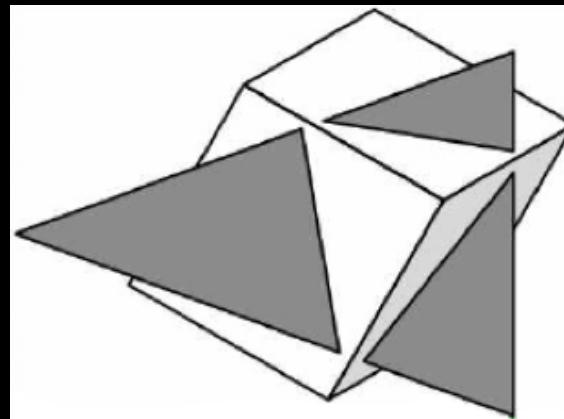


Some Difficult Cases

- Sometimes cannot sort polygons!



Cyclic overlap



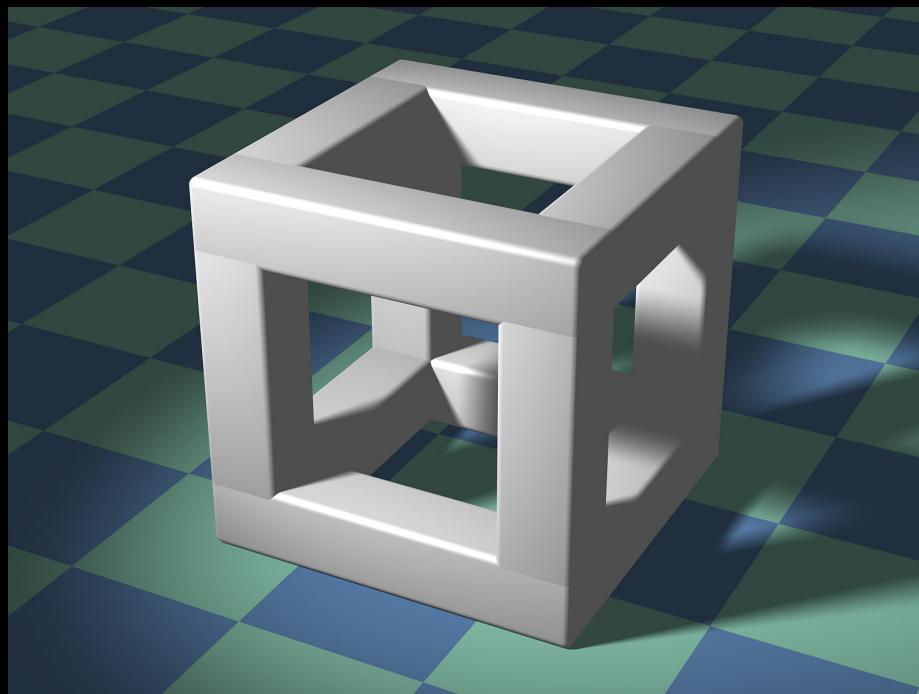
Piercing Polygons

- One solution: compute intersections & subdivide
- Do while rasterizing (difficult in object space)

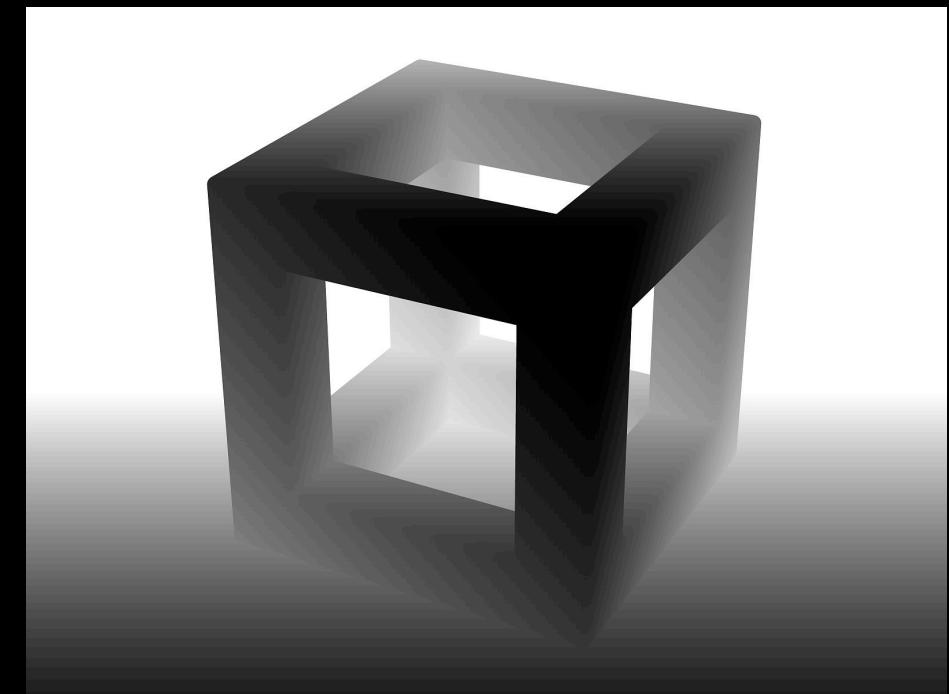
Painter's Algorithm Assessment

- Strengths
 - Simple (most of the time)
 - Handles transparency well
 - Sometimes, no need to sort (e.g., heightfield)
- Weaknesses
 - Clumsy when geometry is complex
 - Sorting can be expensive
- Usage
 - PostScript interpreters
 - OpenGL: not supported
(must implement Painter's Algorithm manually)

Image-space approach



3D geometry



Depth image
darker color is closer

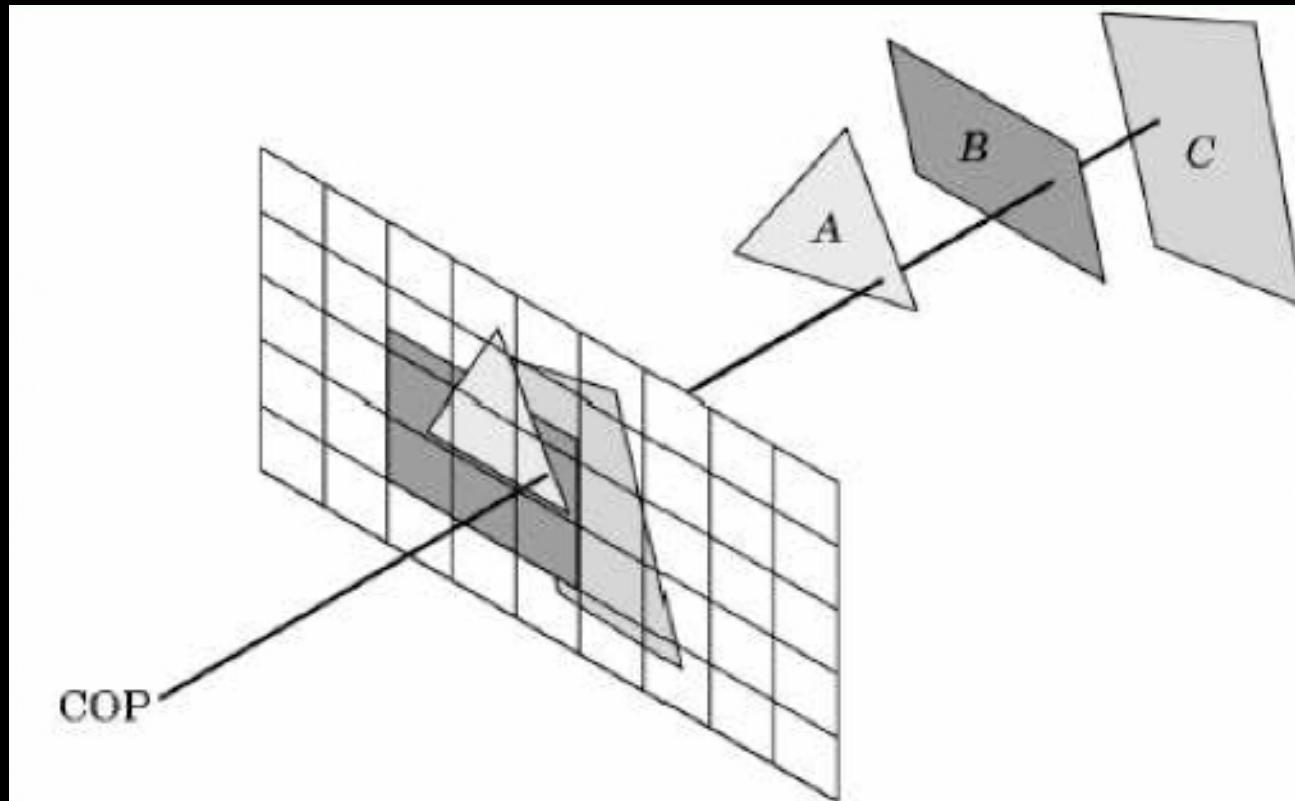
Depth sensor camera



KINECT™
for  XBOX 360.

Image-Space Approach

- Raycasting: intersect ray with polygons

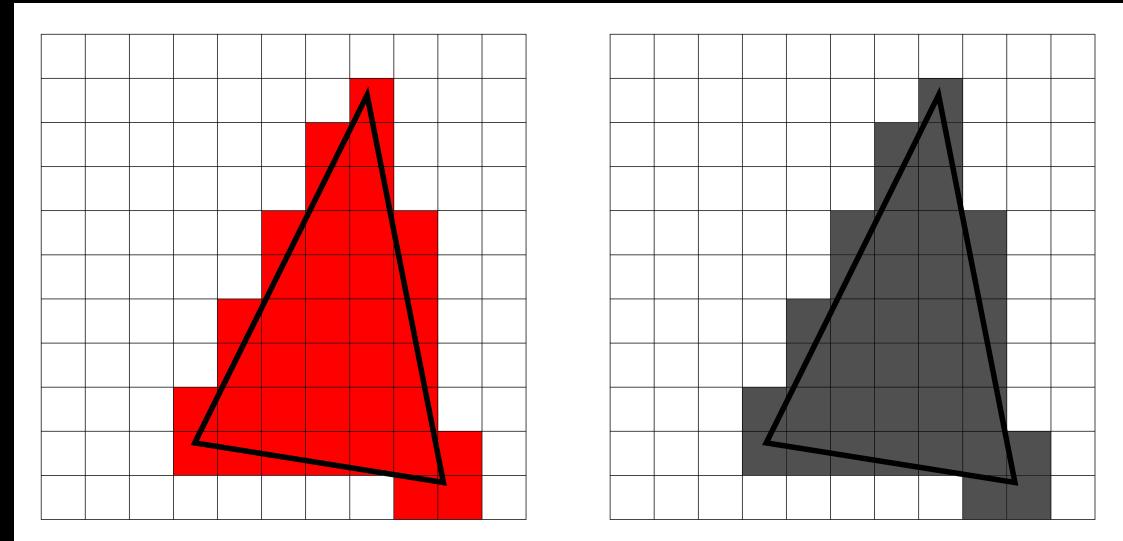
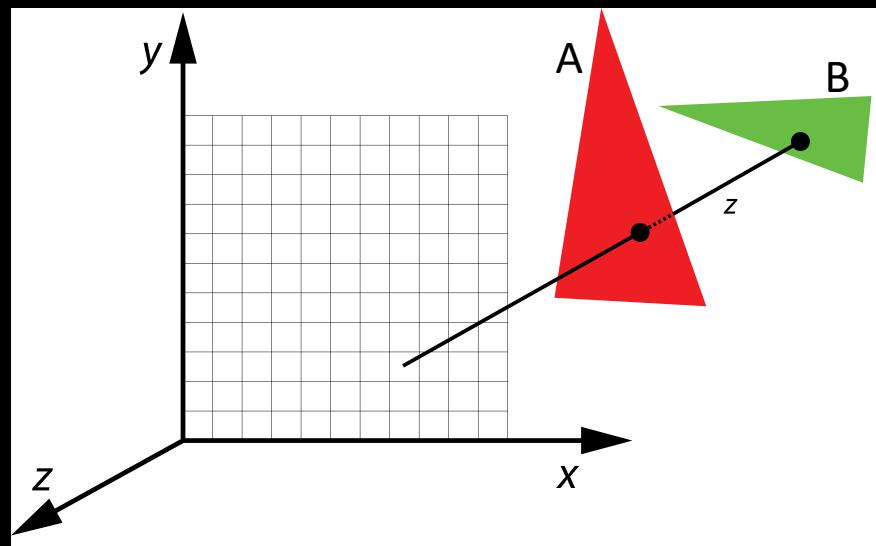


- $O(k)$ worst case (often better)
- Images can be more jagged (need anti-aliasing)

The z-Buffer Algorithm

- z-buffer stores depth values z for each pixel
- Before writing a pixel into framebuffer:
 - Compute distance z of pixel from viewer
 - If closer, write and update z-buffer, otherwise discard

After rendering A:



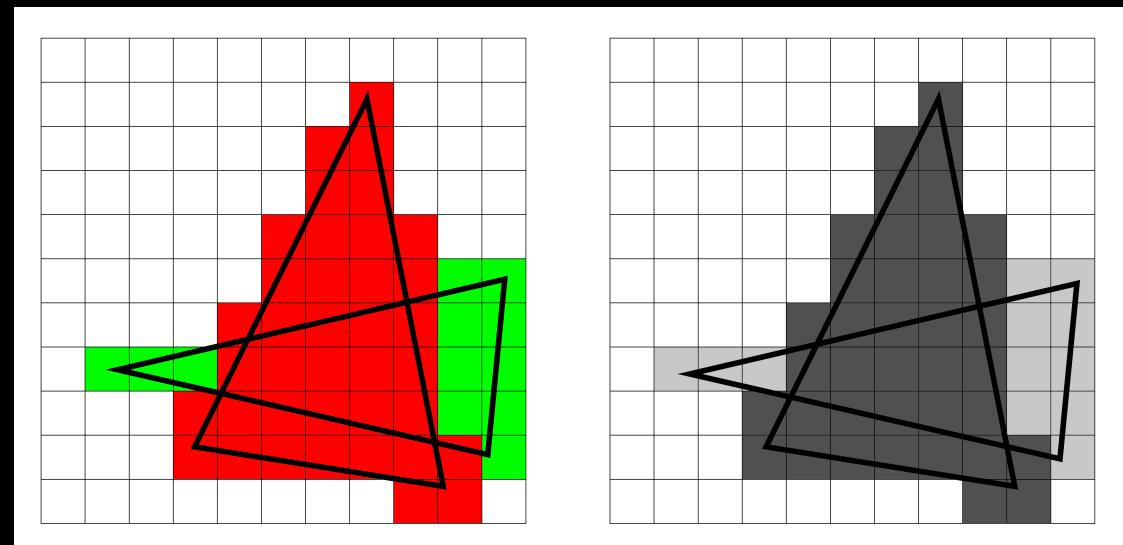
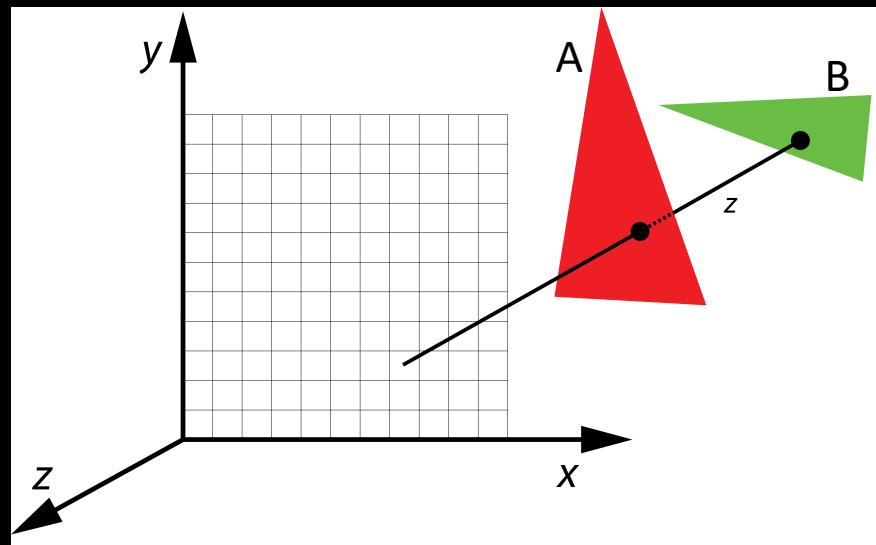
color

depth

The z-Buffer Algorithm

- z-buffer stores depth values z for each pixel
- Before writing a pixel into framebuffer:
 - Compute distance z of pixel from viewer
 - If closer, write and update z-buffer, otherwise discard

After rendering A and B:



color

depth

z-Buffer Algorithm Assessment

- Strengths
 - Simple (no sorting or splitting)
 - Independent of geometric primitives
- Weaknesses
 - Memory intensive (but memory is cheap now)
 - Tricky to handle transparency and blending
 - Depth-ordering artifacts
- Usage
 - z-Buffering comes standard with OpenGL;
disabled by default; must be enabled

Depth Buffer in OpenGL

- `glutInitDisplayMode (GLUT_DOUBLE | GLUT_RGBA | GLUT_DEPTH) ;`
- `glEnable (GL_DEPTH_TEST) ;`
- Inside `Display()`:
 `glClear (GL_DEPTH_BUFFER_BIT) ;`
- Remember all of these!
- Some “tricks” use z-buffer in read-only mode

Note for Mac computers

Must use the GLUT_3_2_CORE_PROFILE flag
to use the core profile:

```
glutInitDisplayMode(GLUT_3_2_CORE_PROFILE |  
GLUT_DOUBLE | GLUT_RGBA | GLUT_DEPTH);
```

z-Buffer

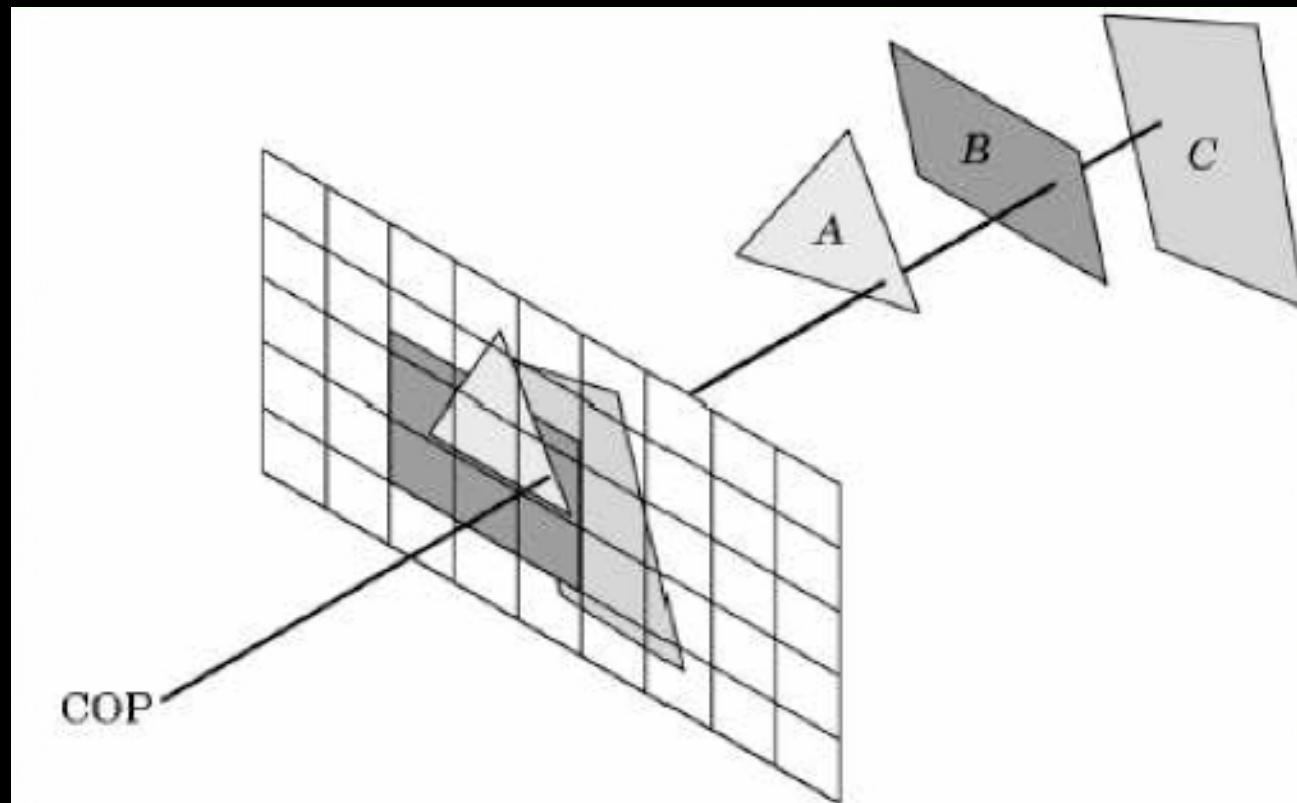
- per-fragment
- blending more difficult
(but possible)
- can lead to artifacts
when objects fight
each other
- less efficient
- USE THIS

painter

- per-primitive
- allows blending
- very difficult to handle
objects overlapping
- more efficient
- DON'T USE THIS

A note on raytracing

- When you raytrace, this is very easy



- Along each ray you cast, simple mathematical problem to solve of which surface you hit first.

A note on raytracing

- Why is this problem so much easier to solve when raytracing?

A note on raytracing

- Why is this problem so much easier to solve when raytracing?
 - OpenGL: draw each triangle into a frame buffer (efficient)
 - Raytracing: for each pixel, find out what to draw
- bucket sort vs. "real" sorting algorithms

Summary

- Client/Server Model
- Callbacks
- Double Buffering
- Physics of Color
- Hidden Surface Removal