Comp 410/510

Computer Graphics

Spring 2023

Overview - Graphics Pipeline

Recall: Basic Graphics System

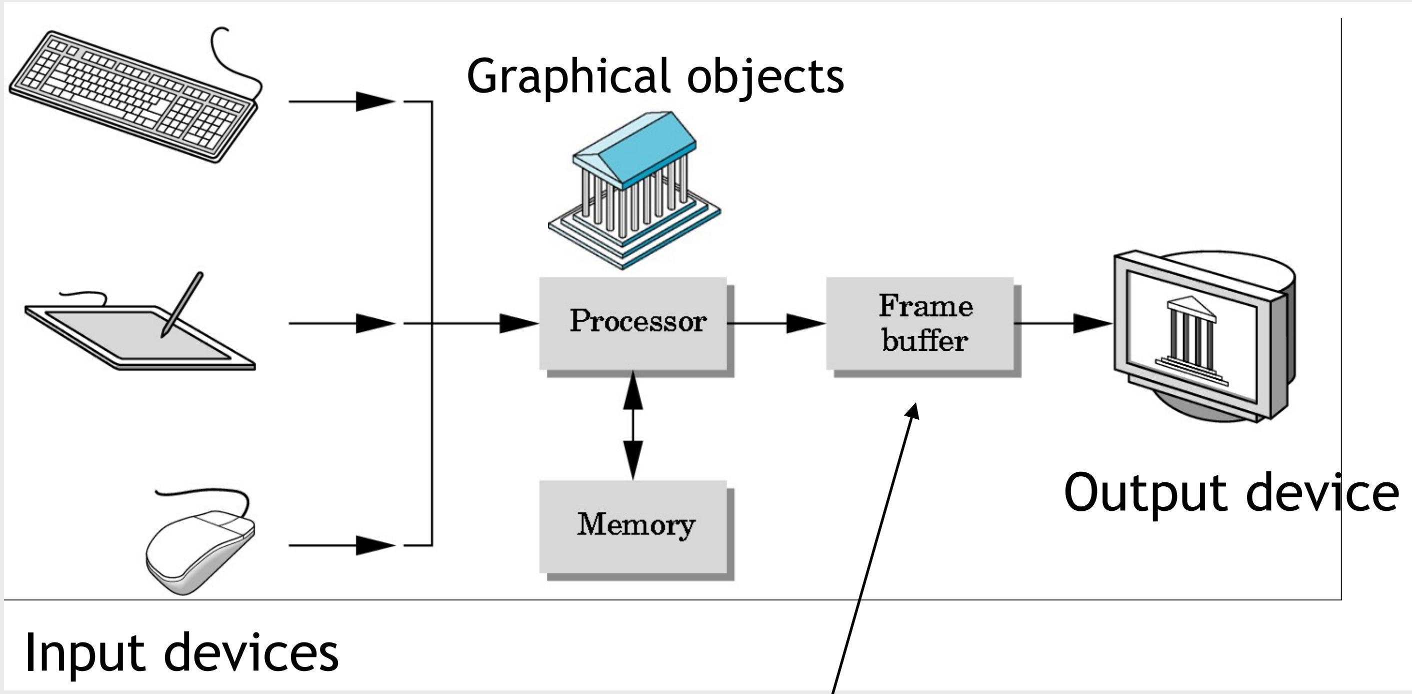


Image is formed in Frame Buffer
via the process Rendering

Image Formation

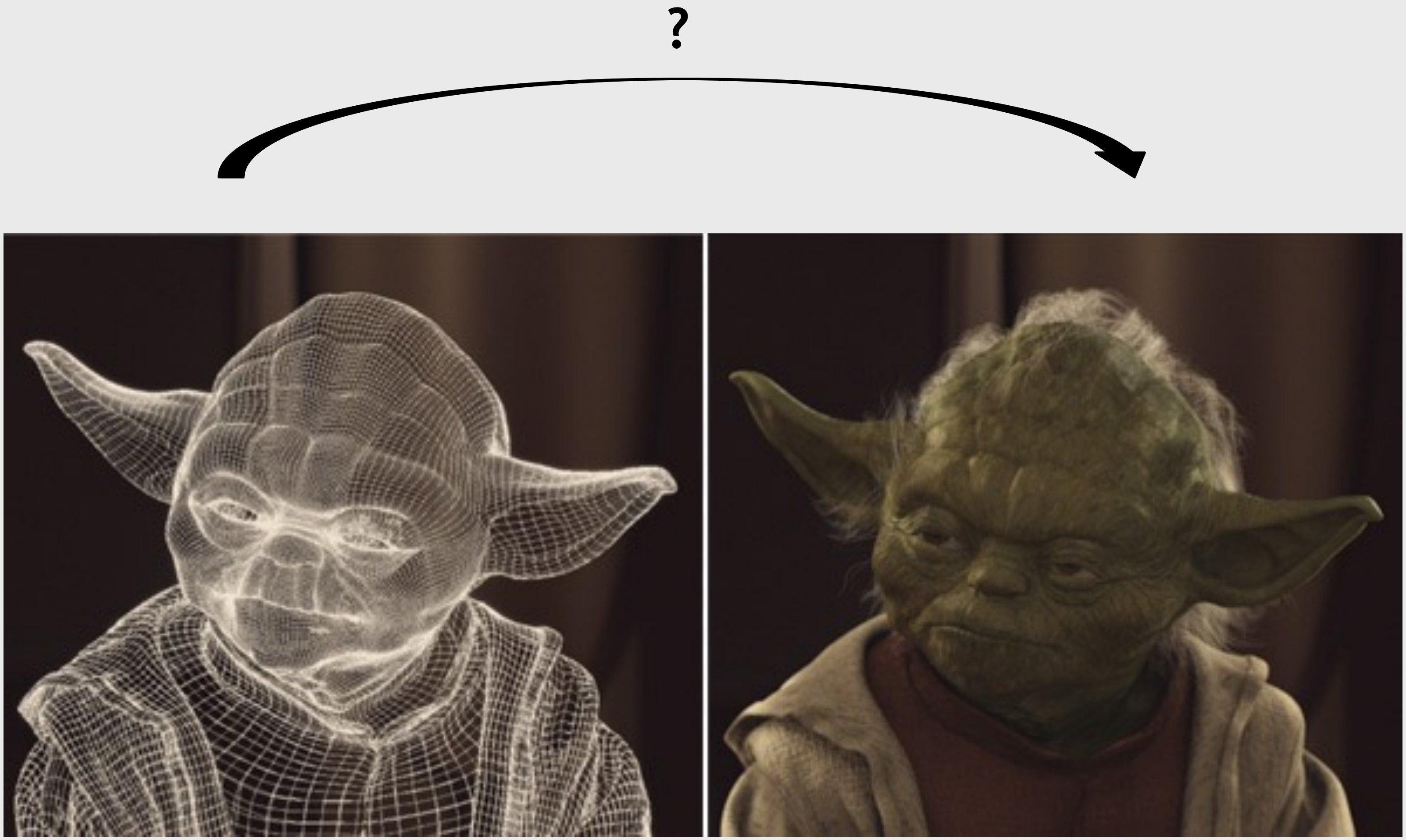
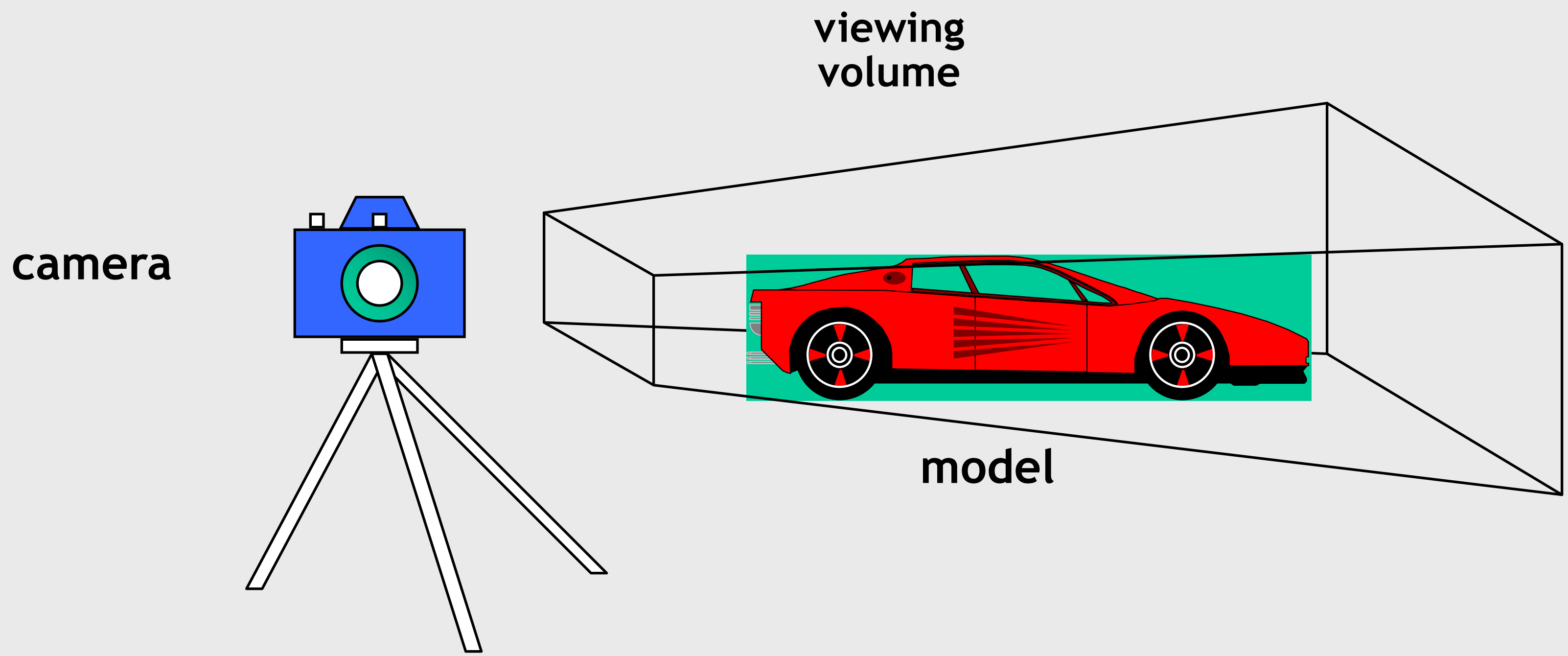


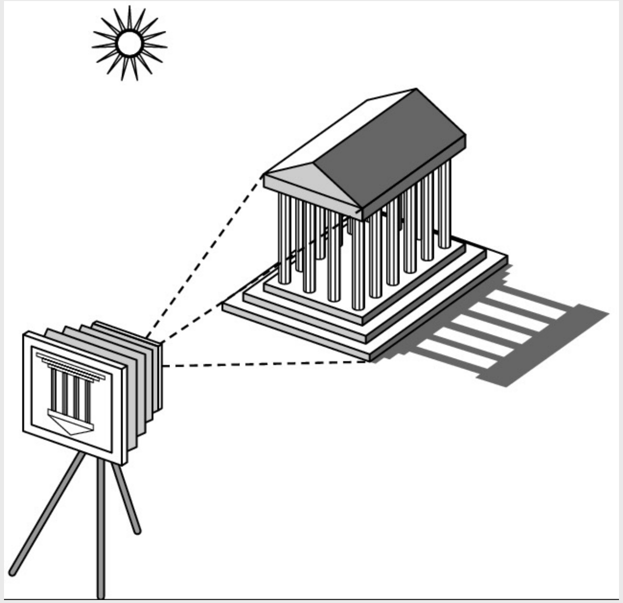
Image Formation

- In computer graphics, we form images which are generally two dimensional using a process analogous to how images are formed by physical imaging systems
  - Cameras, Microscopes,Telescopes, Human visual system



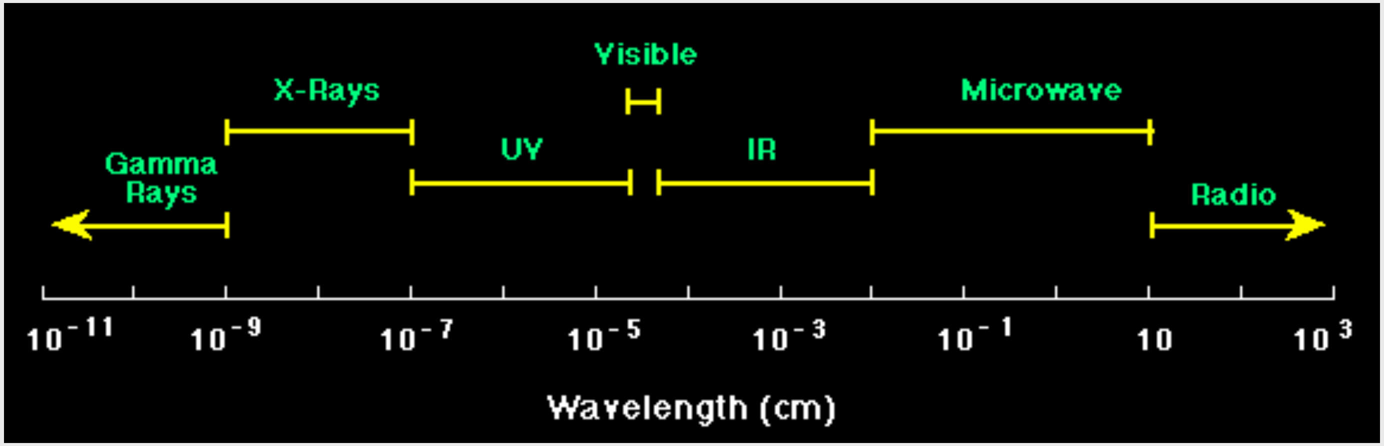
Elements of Image Formation

- Objects
- Viewer
- Light source(s)
- Attributes (that govern how light interacts with materials in the scene)

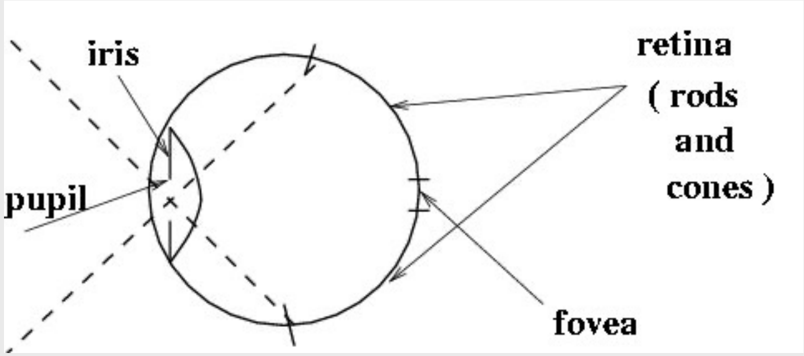


Light

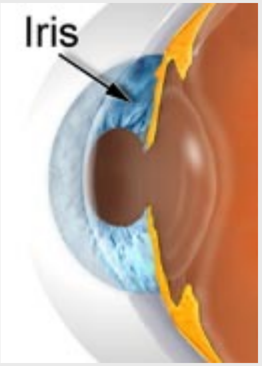
- *Light* is the part of the electromagnetic spectrum that causes a reaction in our visual system
- Generally, these are wavelengths in the range of about 350-750 nm (nanometers)
- Long wavelengths appear as reds and short wavelengths as blues

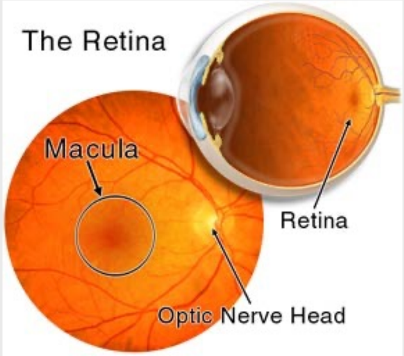


Human Eye as a Spherical Camera



- ~100M sensors in retina
- Rods sense only intensity
- 3 types of cones sense color
- Fovea has tightly packed sensors, more cones
- Periphery has more rods
- Focal length is about 20mm
- Pupil/iris controls light entry





Luminance and Color Images

- Luminance Image
  - Monochromatic
  - Values are gray levels

- Color Image
  - Has perceptional attributes of hue, saturation, and brightness
  - Can use three primaries (red, green and blue) to approximate any color we can perceive.

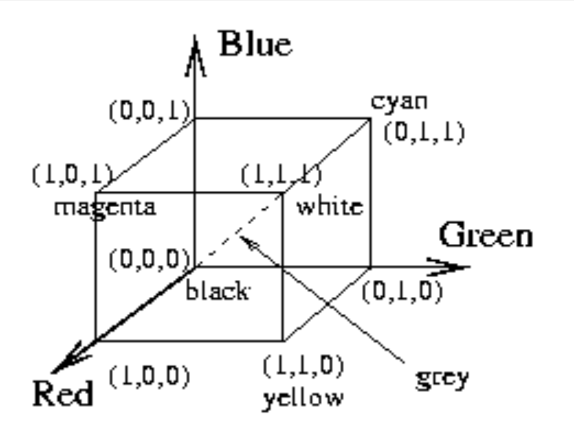
Color Formation (or Synthesis)

- Form a color by adding amounts of three primaries
  - monitors, projection systems, positive film
- Primaries are Red (R), Green (G), Blue (B)

Color Systems

	RGB	CMY	HSI
RED	(255, 0, 0)	(0,255,255)	(0.0 , 1.0, 255)
YELLOW	(255,255, 0)	(0, 0,255)	(1.05, 1.0, 255)
	(100,100, 50)	(155,155,205)	(1.05, 0.5, 100)
GREEN	(0,255, 0)	(255, 0,255)	(2.09, 1.0, 255)
BLUE	(0, 0,255)	(255,255, 0)	(4.19, 1.0, 255)
WHITE	(255,255,255)	(0, 0, 0)	(-1.0, 0.0, 255)
GREY	(192,192,192)	(63, 63, 63)	(-1.0, 0.0, 192)
	(127,127,127)	(128,128,128)	(-1.0, 0.0, 127)
	(63, 63, 63)	(192,192,192)	(-1.0, 0.0, 63)
	...		
BLACK	(0, 0, 0)	(255,255,255)	(-1.0, 0.0, 0)

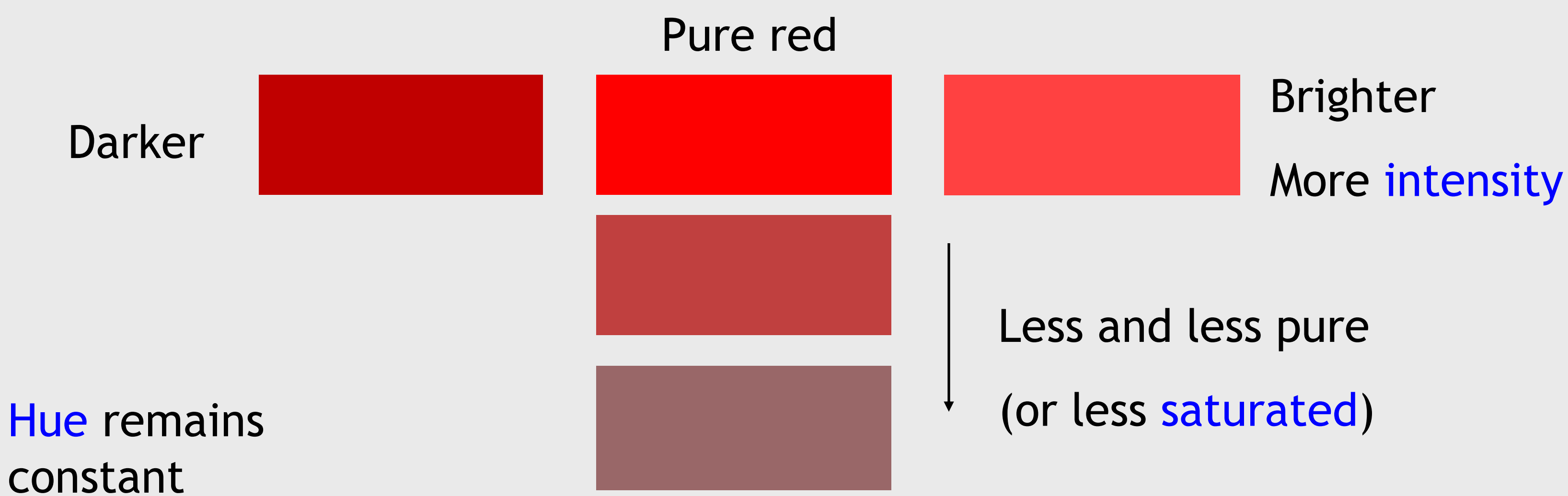
- convenient to scale values in the range 0 to 1 in algorithms
- HSI values are computed from RGB values using Alg.
- $H \in [0.0, 2\pi)$, $S \in [0.0, 1.0]$ and $I \in [0, 255]$.
- Equal proportions of RGB yield grey.
- Equal proportions of R and G yield yellow.



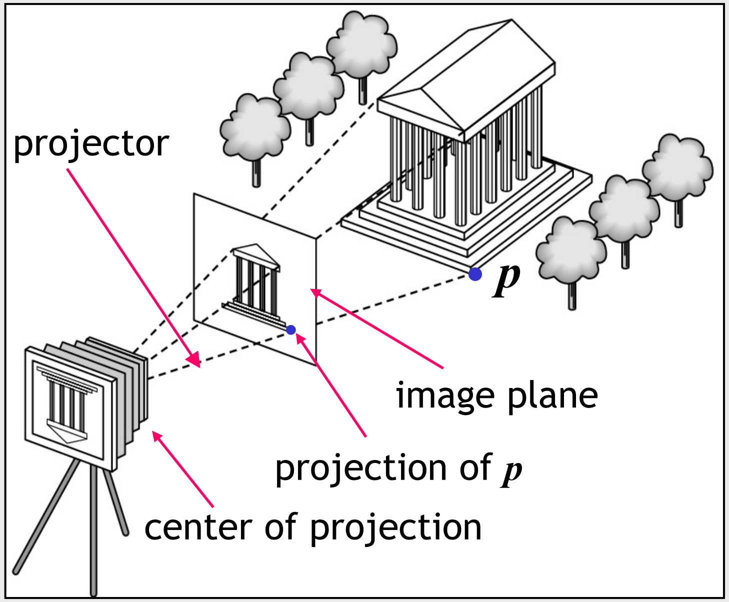
- R, G, B values normalized to (0, 1) interval
- Humans perceive gray for triples on the diagonal
- “Pure colors” on corners

HSI (or HSV) Color System

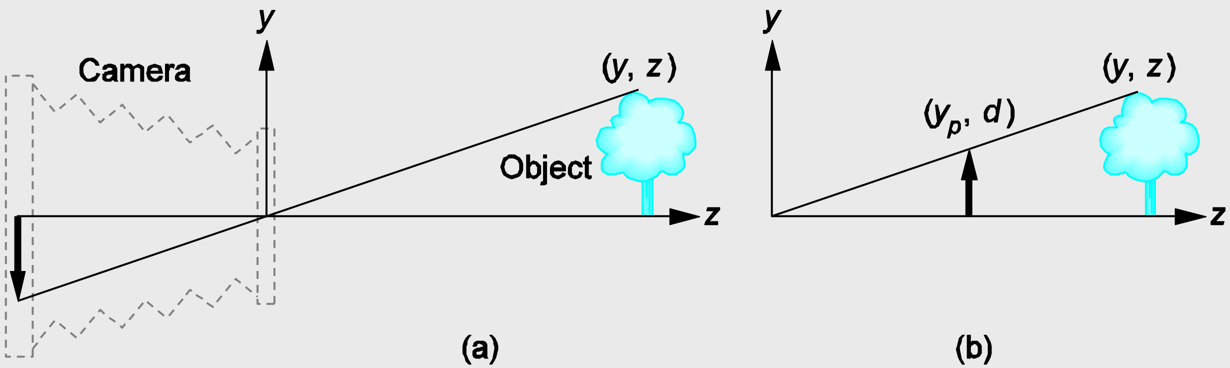
- Separates out intensity I from the coding
- Two values (H & S) encode *chromaticity*
- Convenient for designing colors; used in computer graphics and vision algorithms
- Hue H refers to the perceived color (like "purple")
- Saturation S models the *purity* of the color, that is, its dilution by white light ("light purple")
- I=(R+G+B)/3:   Conversion to gray-level
- Computation of H and S is a bit more complicated (see your textbook).



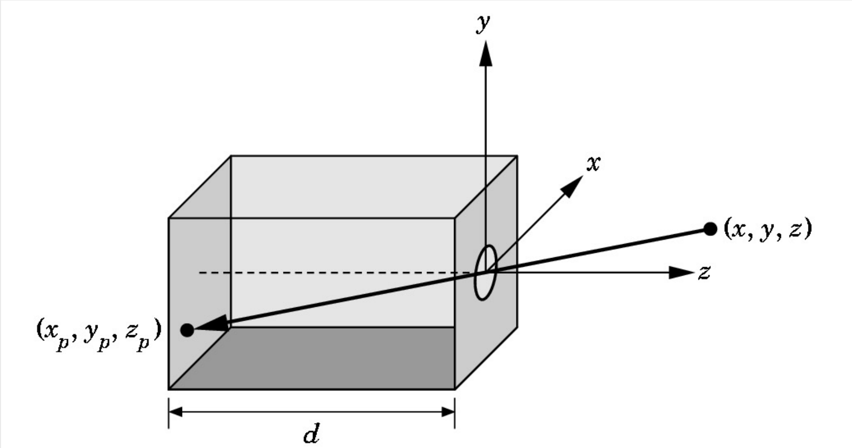
Synthetic Camera Model



Equivalent alternatives:



Pinhole Camera



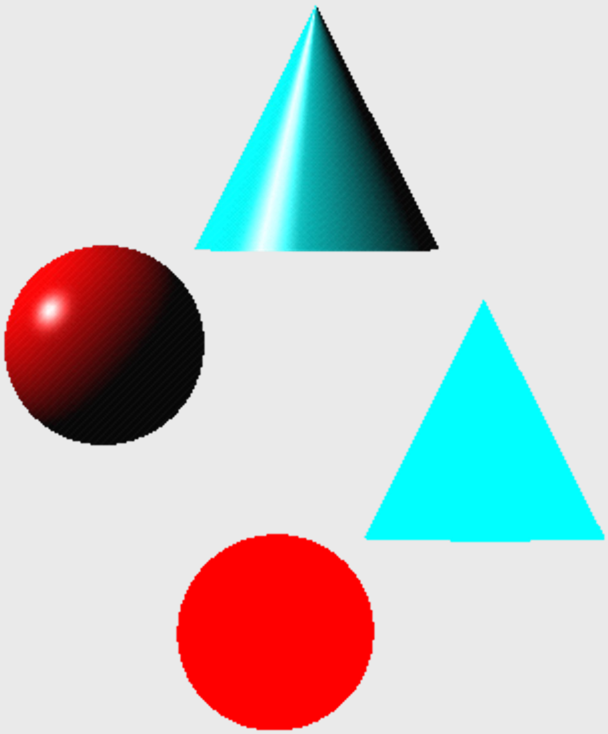
Use trigonometry to find projection of point at (x, y, z)

$$x_p = -\frac{x}{z/d} \qquad y_p = -\frac{y}{z/d} \qquad z_p = d$$

These are equations of simple perspective projection

Lights and Materials

- Types of lights
  - Point sources *vs* distributed sources
  - Spotlights
  - Near and far sources
  - Color properties
- Material properties
  - Absorption: color properties
  - Scattering
    - Diffuse
    - Specular

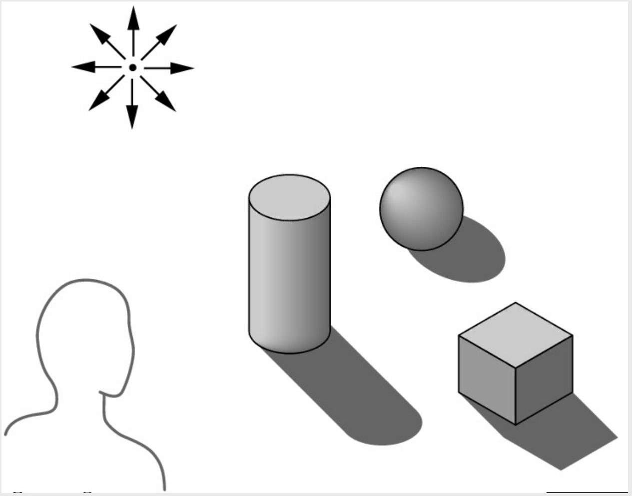


Application Programming Interface (API)

- Separation of objects, viewer, light sources, attributes
- Leads to simple software API
  - Specify objects, lights, camera, attributes
  - Let implementation determine the image
- Leads to fast hardware implementation
- But how is the API implemented?

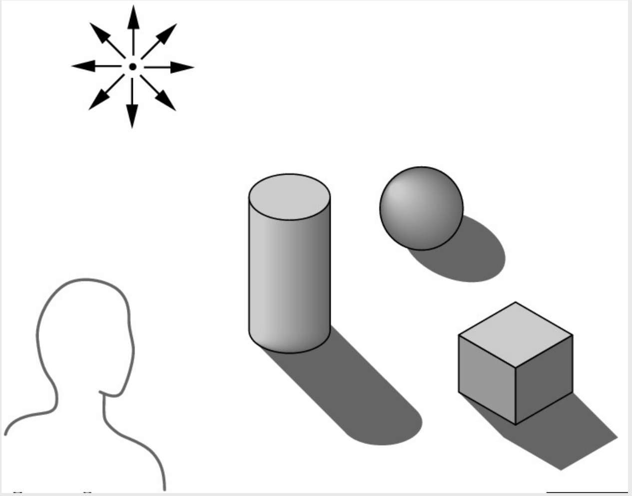
How to Model Illumination?

- Some objects are blocked from light
- Light can reflect from object to object
- Some objects may be translucent



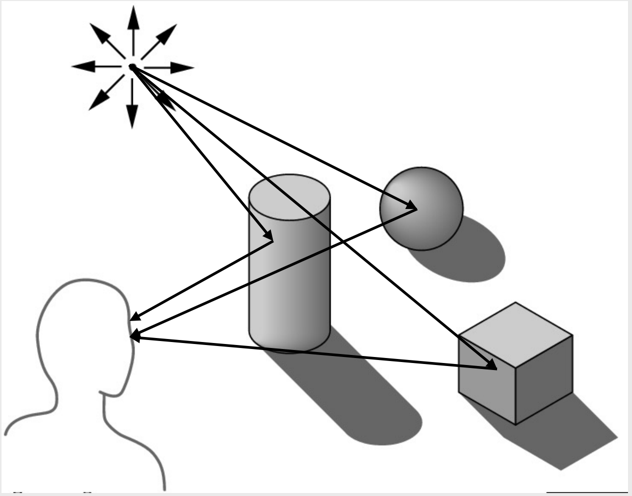
How to Model Illumination?

- Two main aproaches:
  - Local illumination
  - Global illumination



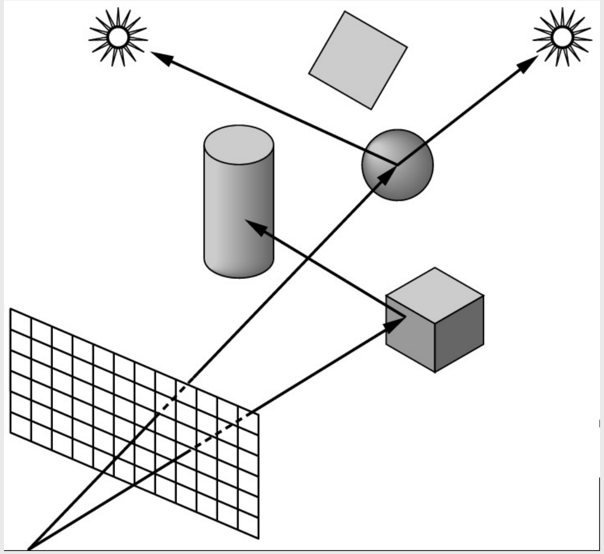
Local Illumination Approach

- Computes color or shade independently for each object and for each light source
- Does not take into account interactions between objects
- Not very realistic, but fast



Global Illumination - Ray tracing

- For each image pixel, follow rays of light from center of projection until they are absorbed by objects, or go off to infinity, or reach a light source
  - Can handle global effects
    - Multiple reflections
    - Translucent and reflecting objects
    - Shadows
  - Especially good in handling specular surfaces like mirrors
  - Slower
  - Need for whole scene data at once

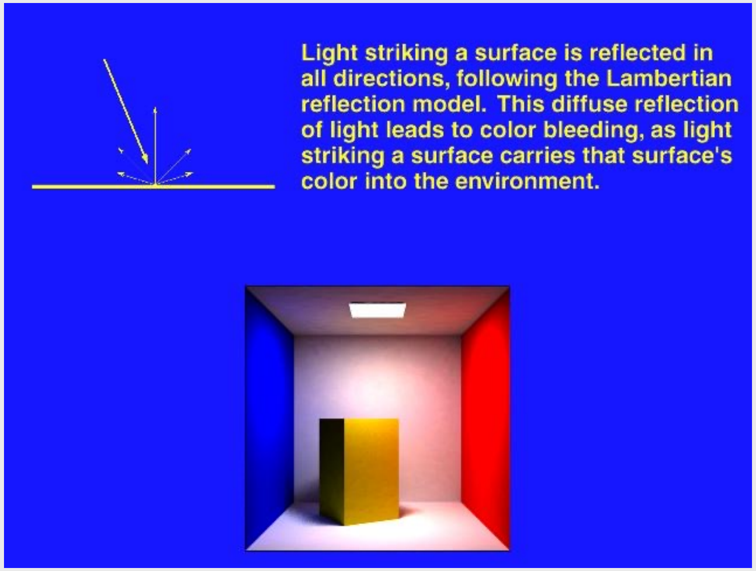


Global (Ray Tracing) vs Local Illumination

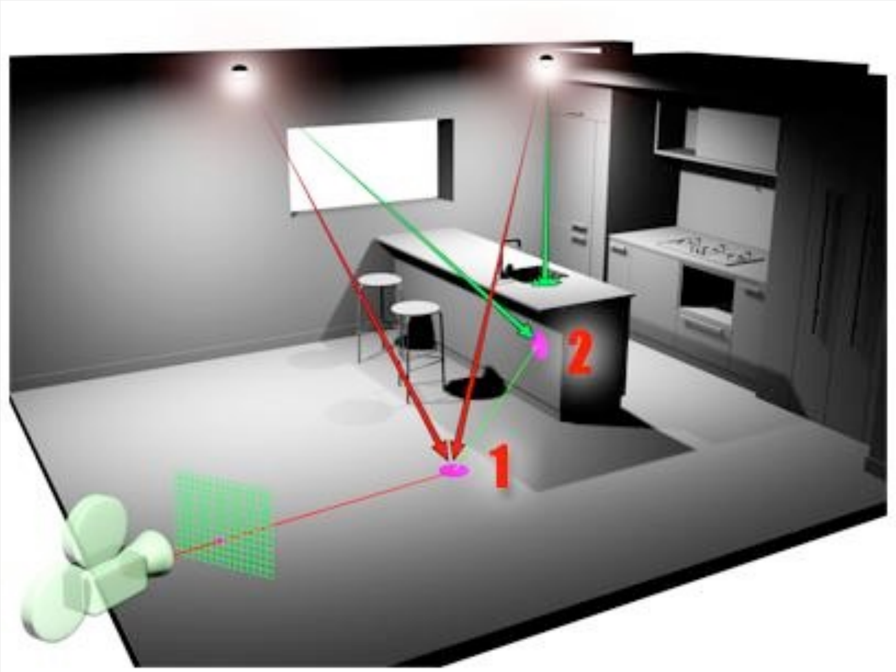
Ray Tracing Example

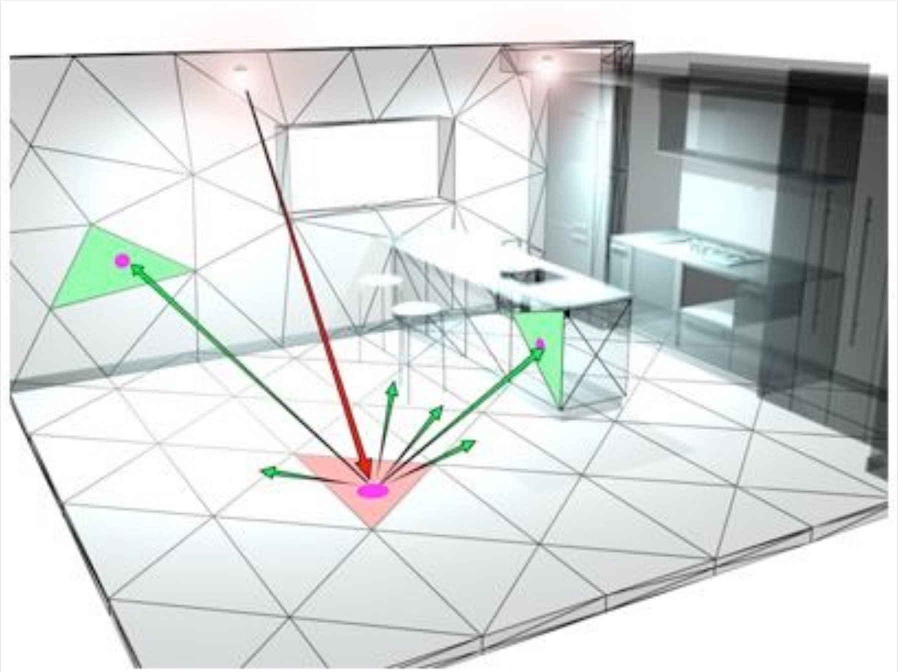
Another Approach for Global Illumination: Radiosity

- Simulates the propagation of light starting at light sources
- Accumulates illumination values on the surfaces of objects, as rays of light propagate from object to object
- Assumes that light striking a surface is reflected in all directions
- Computes interactions between lights and objects more accurately
- Radiosity calculation can be cast to solve a large set of equations involving all the surfaces
- Models well diffuse surfaces but not specular surfaces
- Very slow



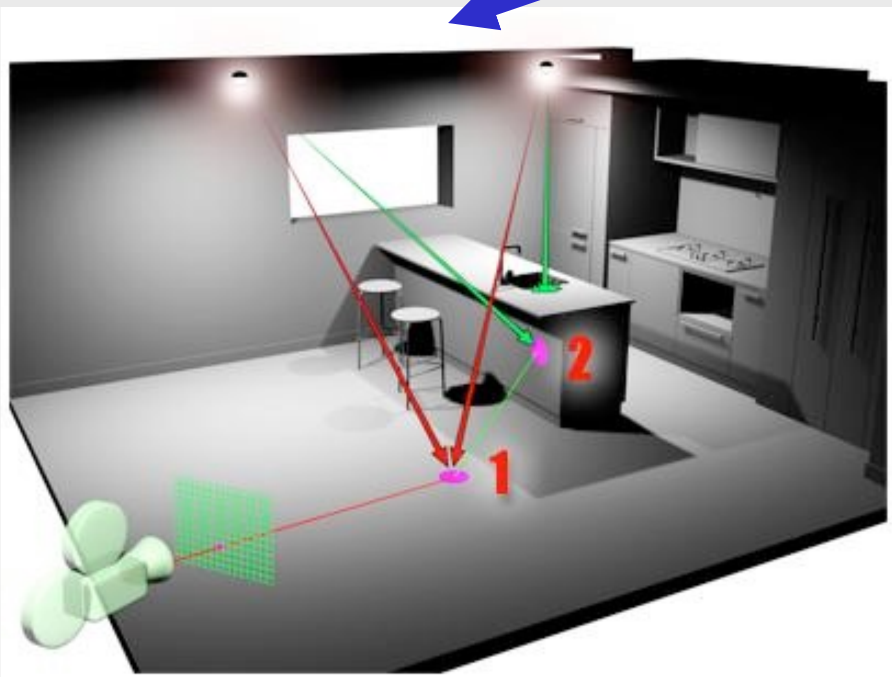
Radiosity vs Ray tracing



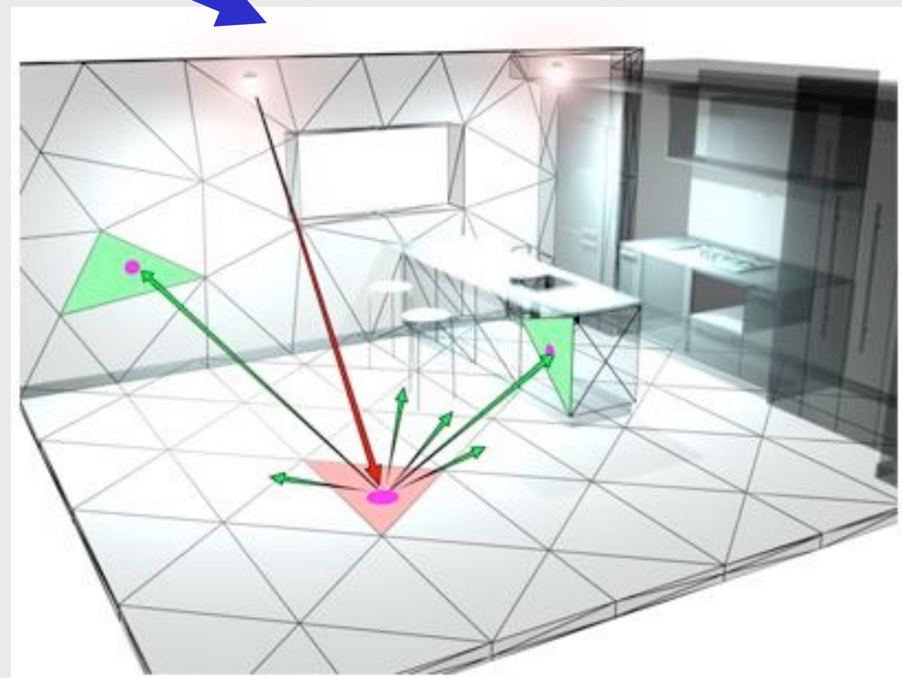


Which one is which?

Radiosity vs Ray tracing



view-dependent



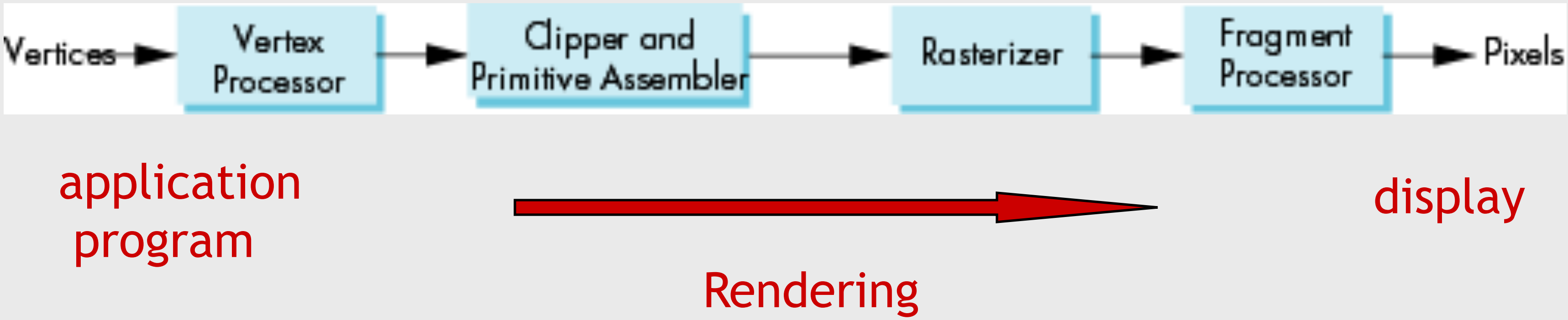
view-independent

Why not always use global illumination?

- Seems more physically-based compared to local illumination
- Relatively easy for simple objects such as polygons and quadrics with simple point light sources
- But slow and hence not well-suited for interactive applications
- Good for creating realistic movies
- Ray tracing with some of the latest GPUs is now almost real time!

Pipeline architecture

- Practical approach implemented by most API's such as OpenGL, Java3D, DirectX, Vulkan and various others.
- Uses only **local illumination**
- Processes objects one at a time in the order they are generated by the application



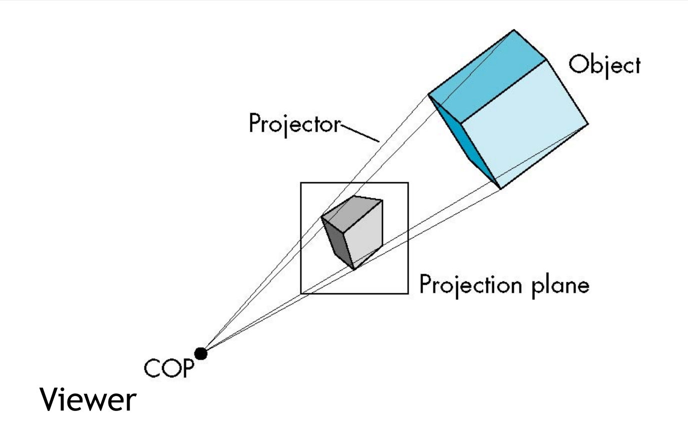
- All steps can be implemented in hardware on the graphics card

Following the Pipeline: Vertex Processing

- Converts object representations from world coordinate system to camera and then to screen coordinates
- Every change of coordinates is equivalent to a matrix transformation
- Used to transform objects,e.g., rotate, translate and scale
- Vertex processor also computes vertex colors (or shades)

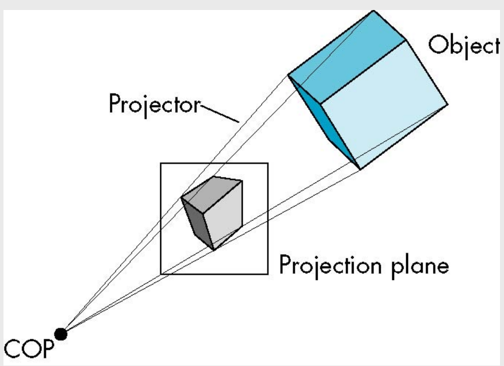
Projection

- Must carry out the process that combines the viewer with 3D objects to produce the 2D image

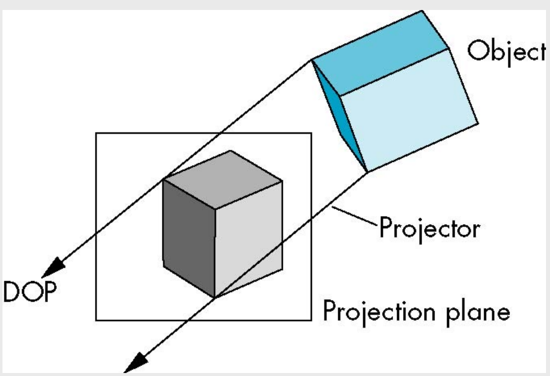


Projection

- Must carry out the process that combines the viewer with 3D objects to produce the 2D image
  - **Perspective projection:** all projectors meet at the center of projection
  - **Parallel projection:** projectors are parallel, center of projection is replaced by a direction of projection



Perspective

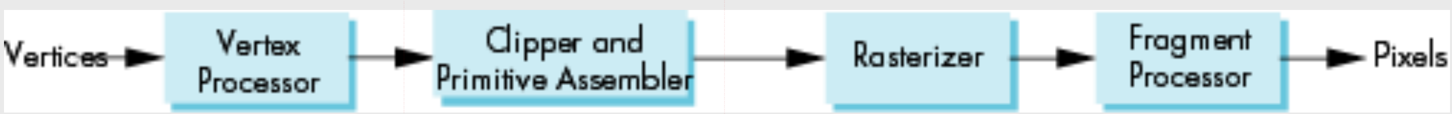


Parallel

Primitive Assembly

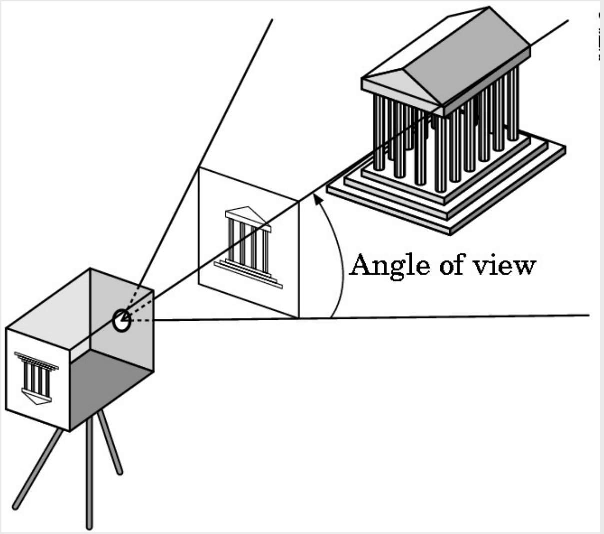
Vertices must be collected into geometric primitives so that rasterization can take place, such as

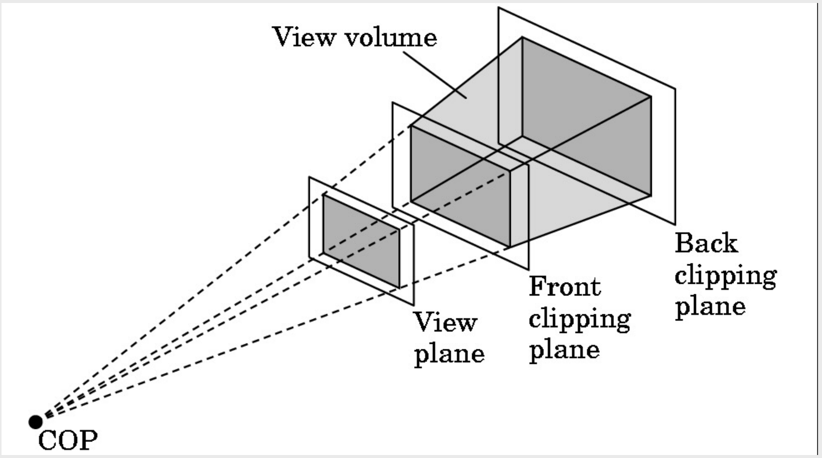
- Line segments
- Polygons
- Curves and surfaces



Clipping

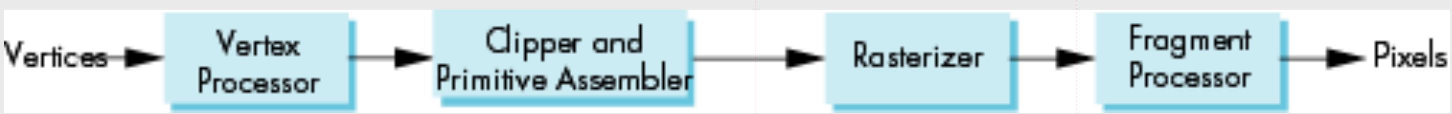
- Just as a real camera cannot “see” the whole world, the virtual camera can only see part of the world space
  - Objects (primitives) that are not within this volume are *clipped* out of the scene

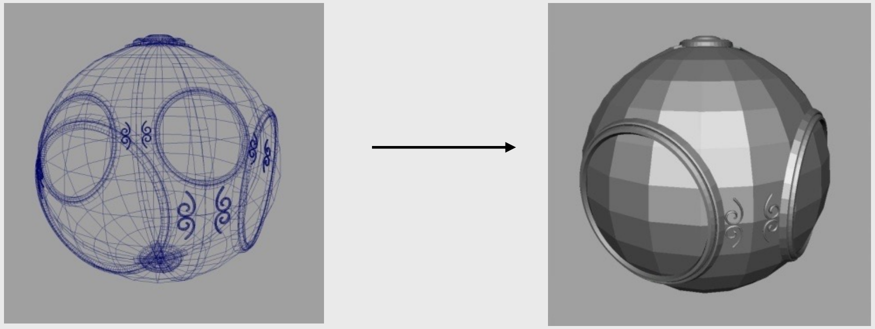




Rasterization

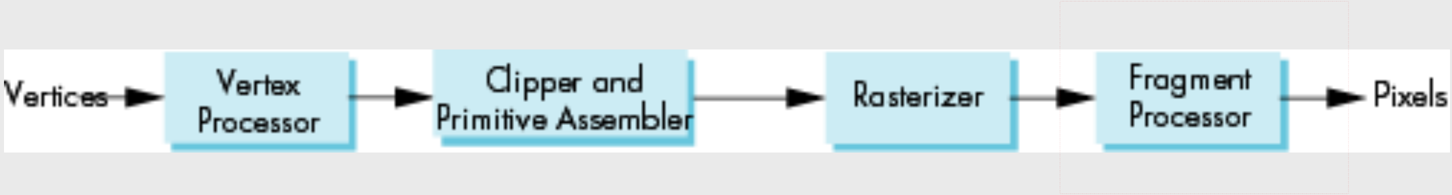
- If an object is visible in the image, the corresponding pixels in the frame buffer must be assigned colors
- Vertex attributes are interpolated over objects by the rasterizer
- Rasterizer produces a set of fragments for each object
- Fragments are “potential pixels”
  - Have a location in frame buffer
  - Have also color and depth attributes





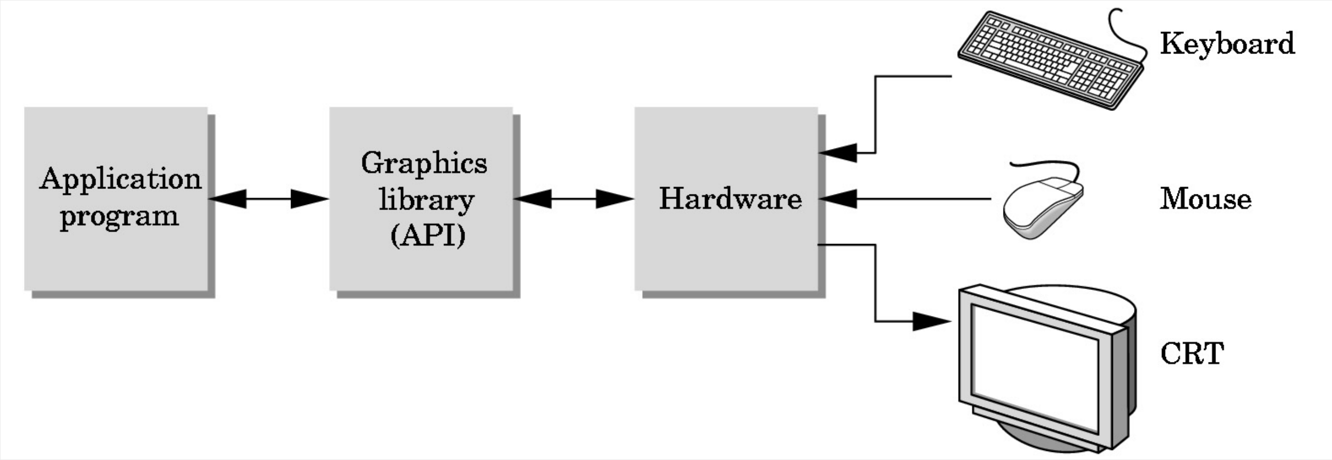
Fragment Processing

- Fragments are processed to determine the final color of the corresponding pixel in the frame buffer
- Fragments may be blocked by other fragments closer to the camera
  - So hidden-surface removal is neeed
- Colors can be determined by texture mapping as well as interpolation of vertex colors



The Programming Interface

Programmer sees the graphics system through a software interface: the Application Programming Interface (API)



API Contents

- Functions that specify what we need to form an image
  - Objects
  - Viewer
  - Light Source(s)
  - Materials
- Other information
  - Input from devices such as mouse and keyboard

Object Specification

- Most APIs support a limited set of primitives including
  - Points (0D object)
  - Line segments (1D objects)
  - Polygons (2D objects)
  - Some curves and surfaces
    - Quadrics
    - Parametric polynomials
- All are defined through locations in space or *vertices*

OpenGL Example (old style)

type of object

Alternatives: `GL_POINTS, GL_LINE_STRIP`

```
glBegin(GL_POLYGON)
 glVertex3f(0.0, 0.0, 0.0);
 glVertex3f(0.0, 1.0, 0.0);
 glVertex3f(0.0, 0.0, 1.0);
glEnd( );
```

location of vertex

end of object definition

Example (new style - shader based)

1. Put geometric data in a generic array:

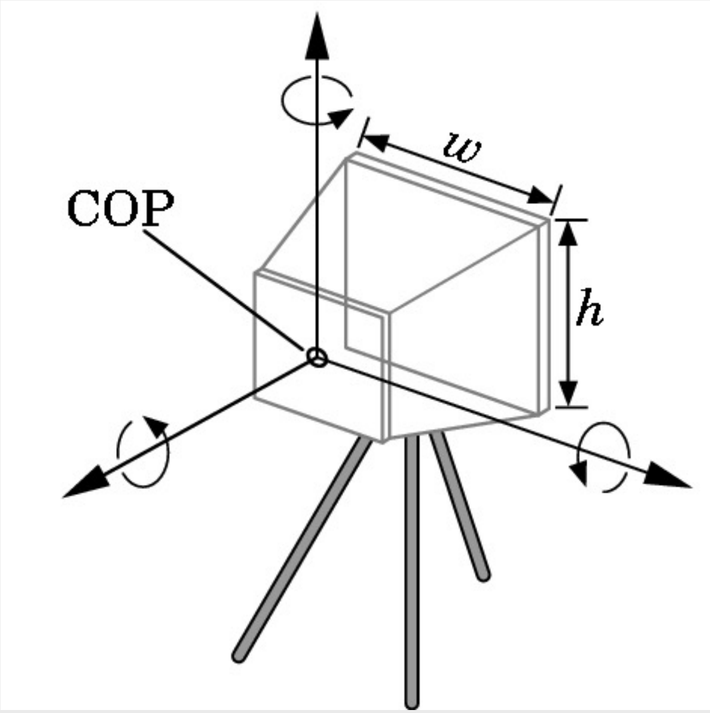
```
vec3 points[3];
points[0] = vec3(0.0, 0.0, 0.0);
points[1] = vec3(0.0, 1.0, 0.0);
points[2] = vec3(0.0, 0.0, 1.0);
```

2. Send array to GPU

3. Tell GPU to render it as triangles

Camera Specification

- Six degrees of freedom
  - Position of center of projection
  - Orientation
- Lens (focal length)
- Film plane
  - Size

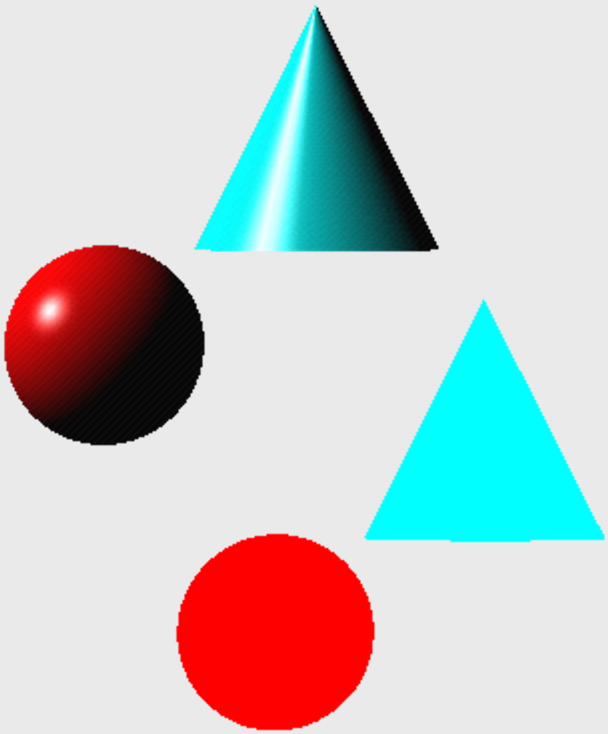


OpenGL API:

```
glOrtho(), glFrustum(), etc.
```

Lights and Materials

- Types of lights
  - Point sources
  - Spotlights
  - Near and far sources
  - Color properties
- Material properties
  - Absorption: color properties
  - Scattering
    - Diffuse
    - Specular



Old OpenGL: glColor(), glLight(),…

Shader-based OpenGL: Implement mostly in shaders using GLSL