Fall 2018

CSCI 420: **Computer Graphics**

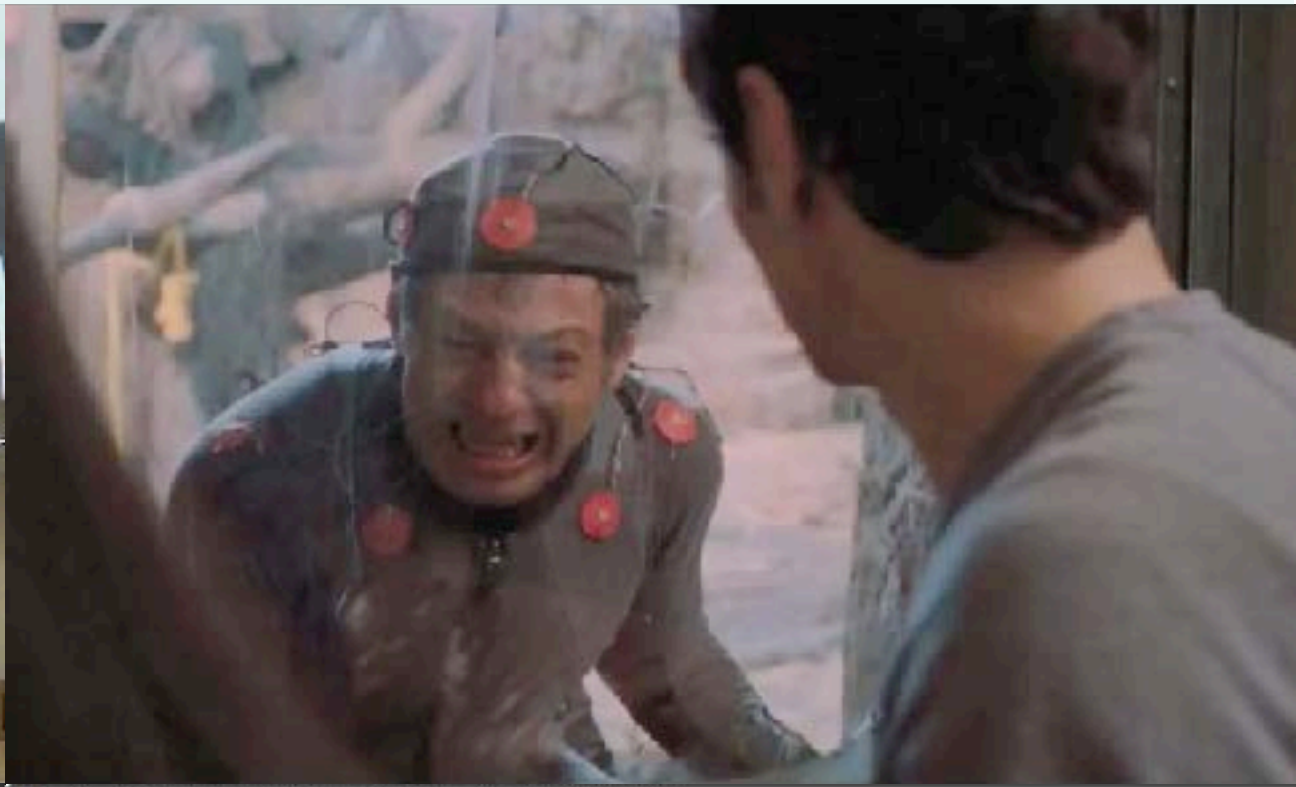
1.2 Basic Graphics Programming

Hao Li

http://cs420.hao-li.com

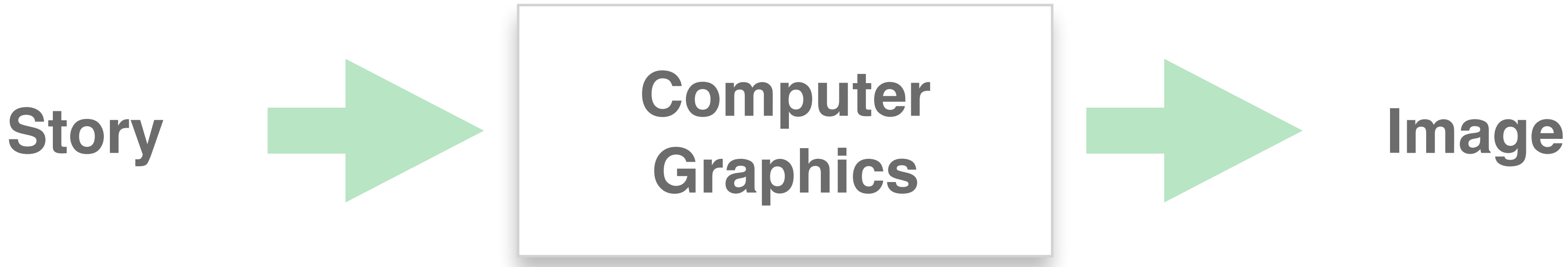
Last time

element



Ps
File Edit Image Layer Type Select Filter 3D View Window Help
Mode: Normal
Brush Presets
Brush Tip Shape
Shape Dynamics
Scattering
Texture
Dual Brush
Color Dynamics
Transfer
Brush Pose
Noise
Wet Edges
Build-up
Smoothing
Protect Texture
Bristle Qualities
Shape: Round Blunt
Length
Thickness
Softness
Angle
Spacing
Mini Bridge
Computer

Last Time

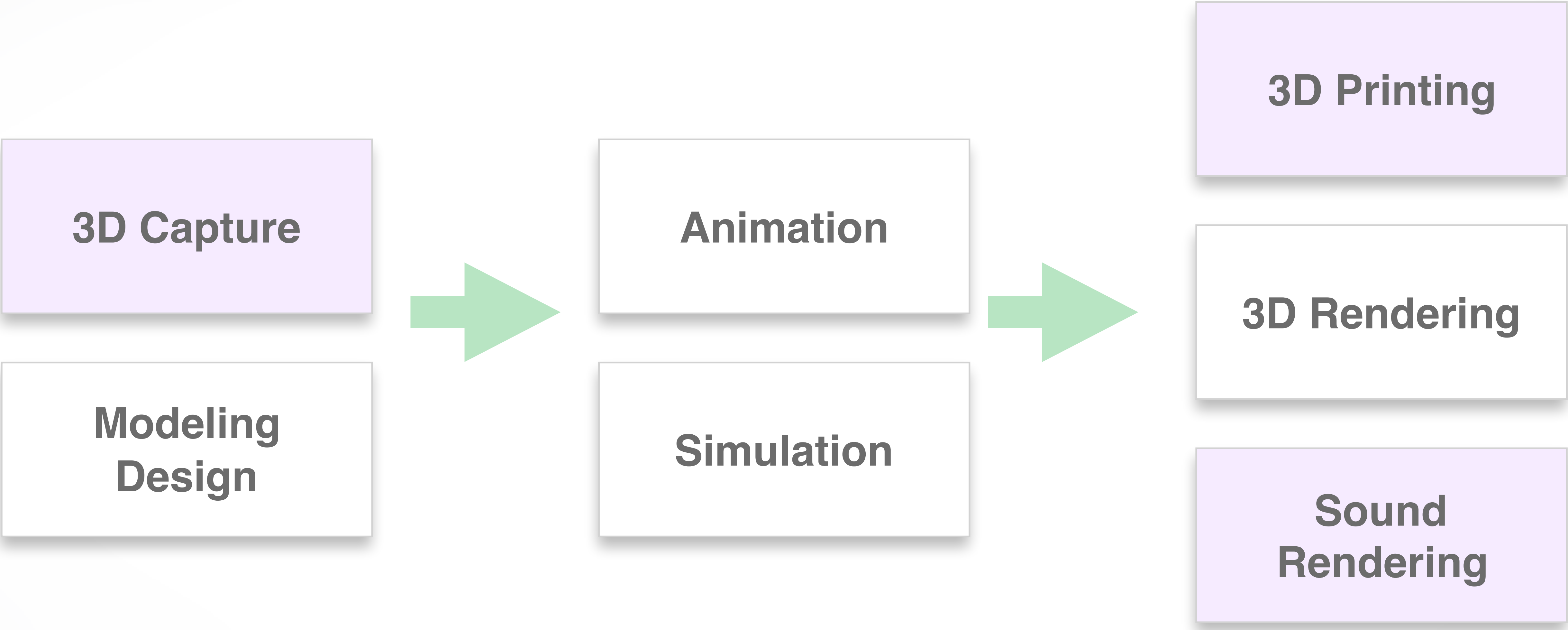


Story

Computer Graphics

Image

Last Time



3D Capture

Modeling Design

Animation

Simulation

3D Printing

3D Rendering

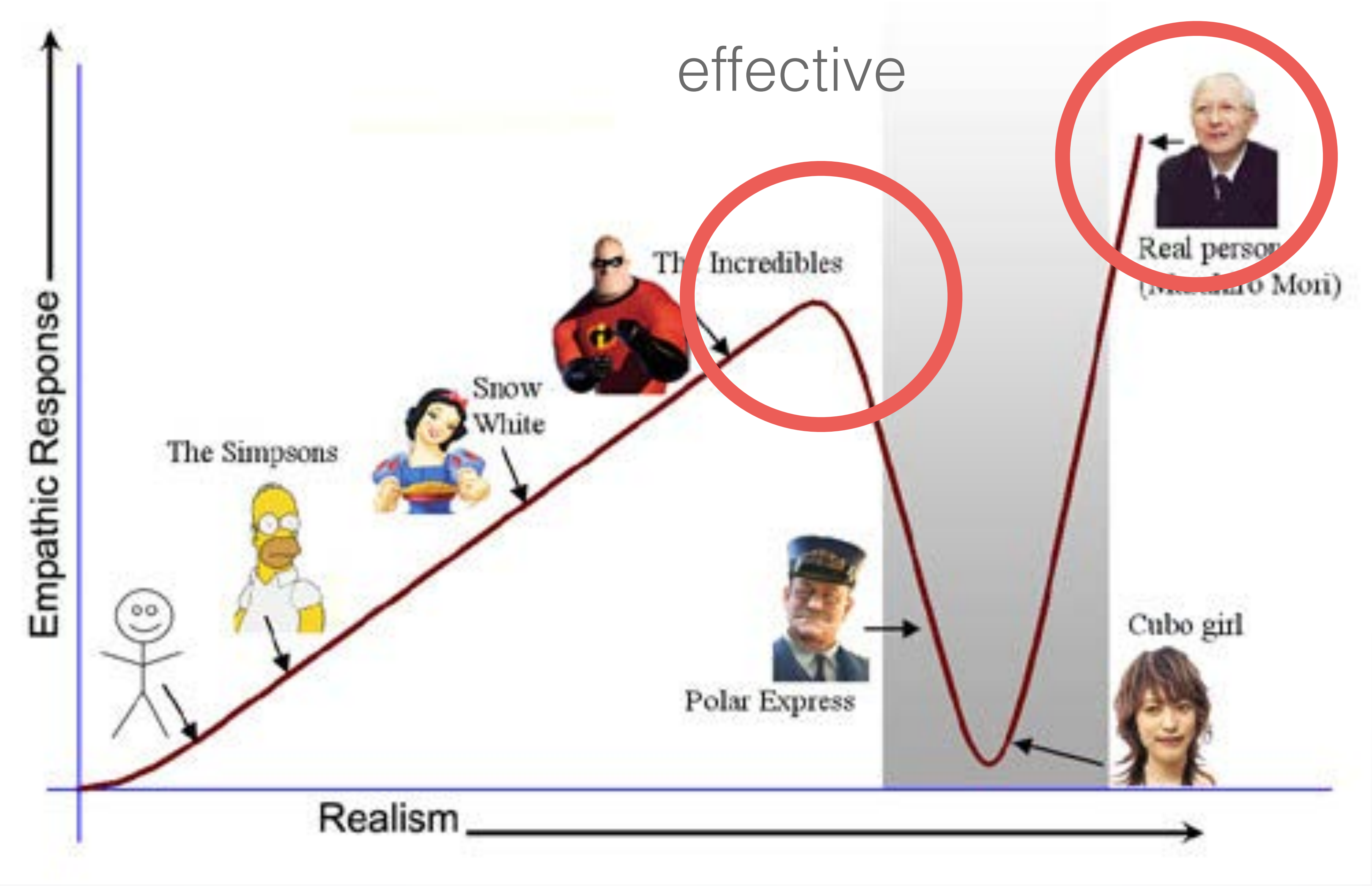
Sound Rendering

emerging fields

Last Time

realistic

effective



Last Time

From Offline to Realtime

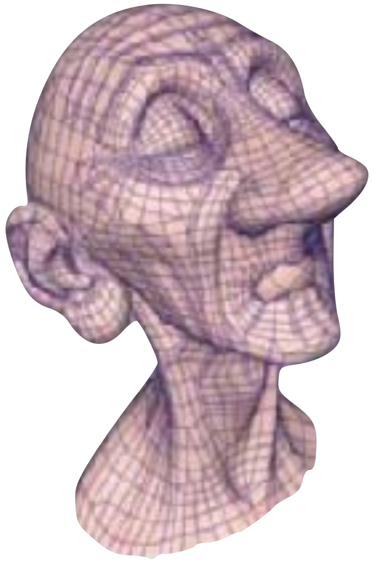
From Graphics to Vision

From Graphics to Fabrication

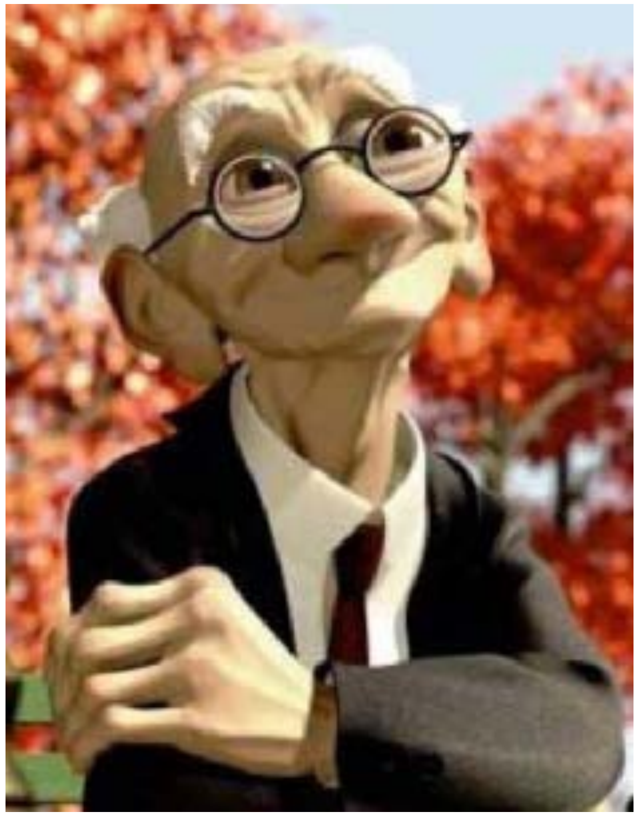
From Production to Consumers

Render [ren-der]

To generate an image or animation



input data



output rendering

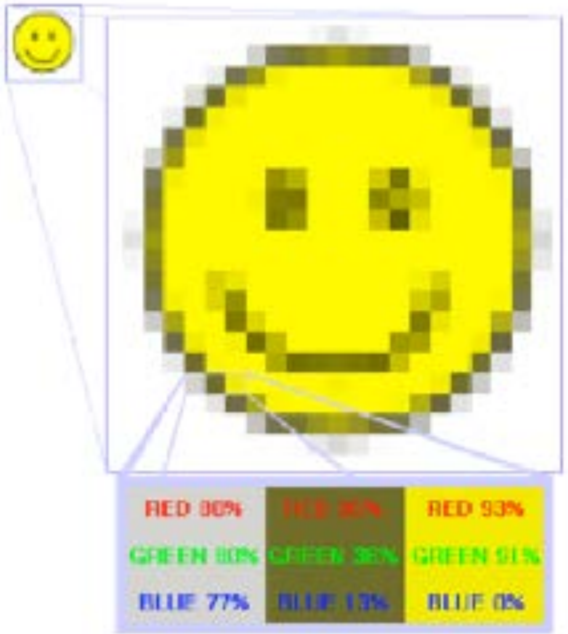
How to make an image?

drawing

photography

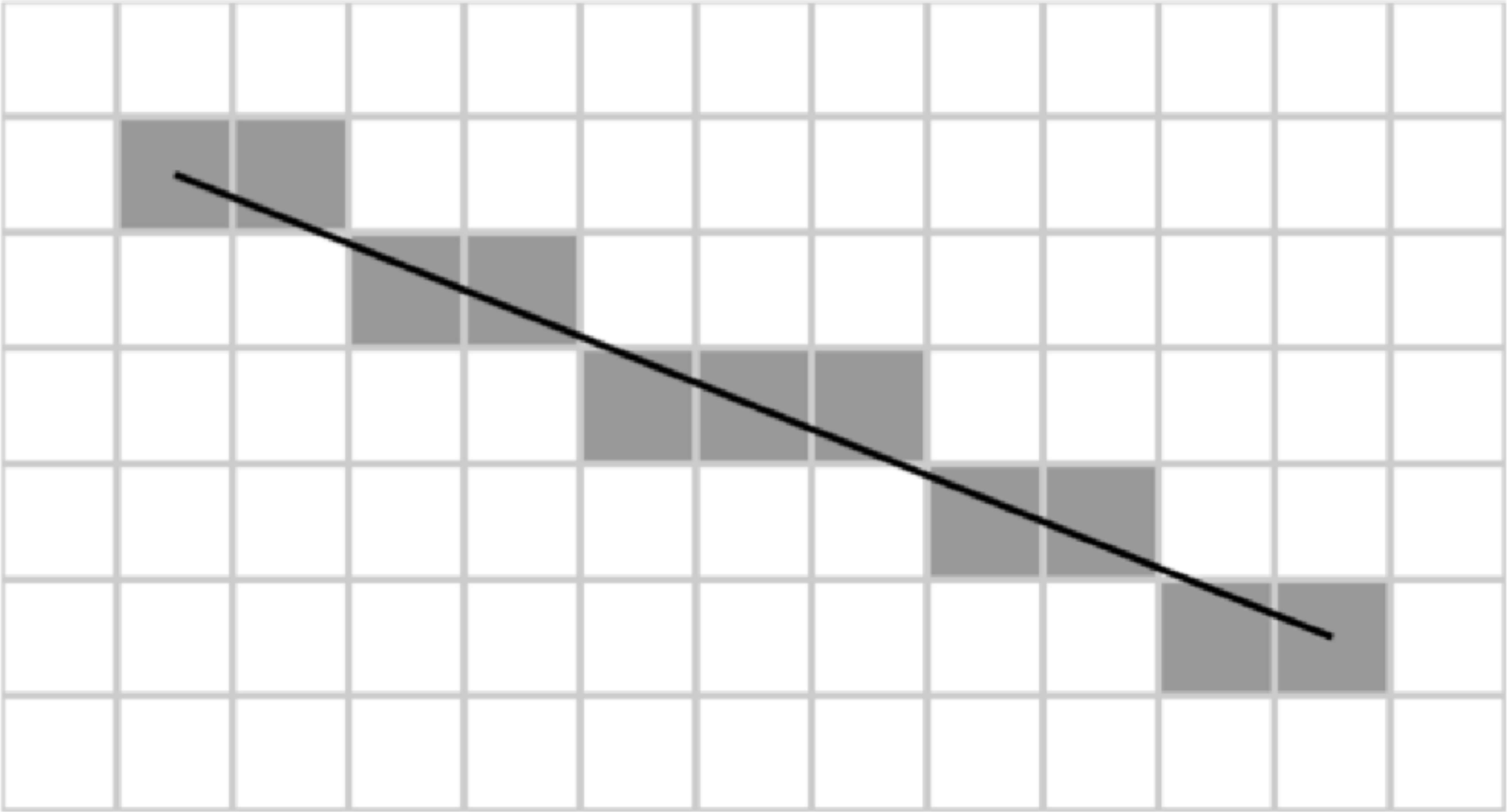
Output: Raster Image

- 2D array of pixels (**pic**ture **el**ements)
  - regular grid sampling of arbitrary 2D function
  - different formats, e.g., bitmaps, grayscale, color
  - different data types, e.g., boolean, int, float
  - color/bit depth: #bits/pixel
  - transparency handled by alpha channel, e.g., RGBA

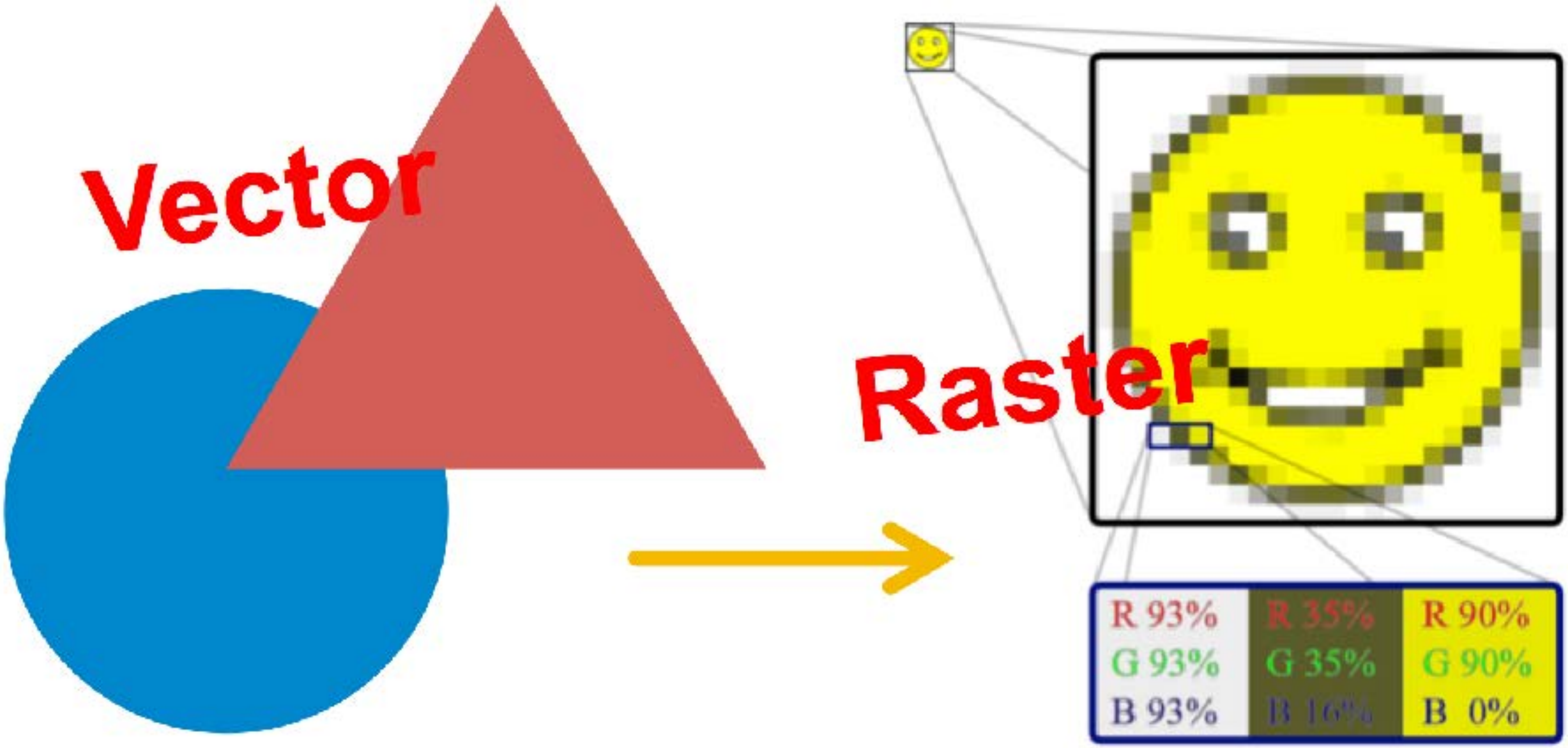


[wikipedia]

Rasterization



Rasterization



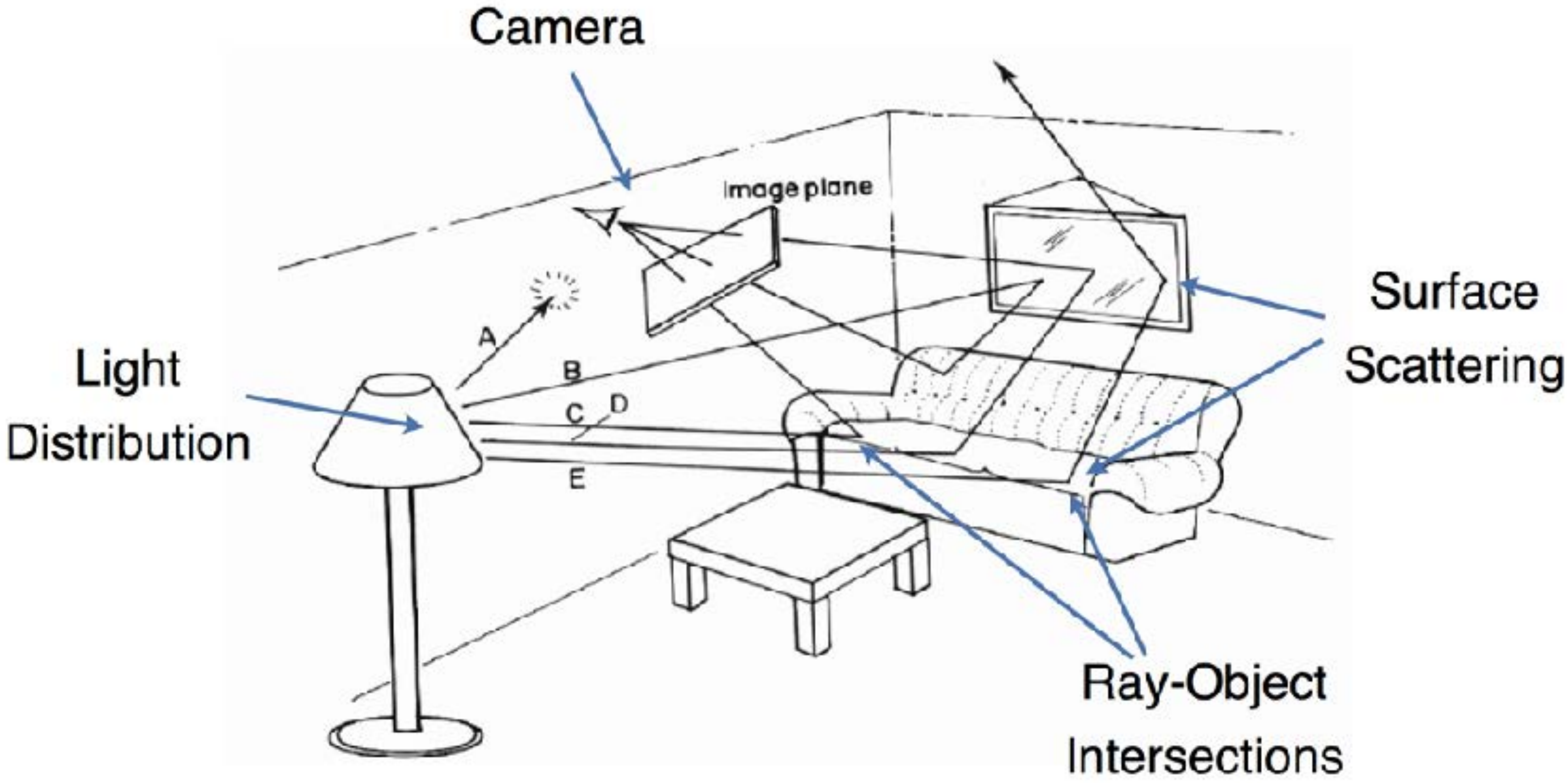
R 93%	R 35%	R 90%
G 93%	G 35%	G 90%
B 93%	B 16%	B 0%

"A triangle is here, a circle is there, …"

"This pixel is yellow…"

Okay… let’s take a step back

In the physical world



Camera

Light Distribution

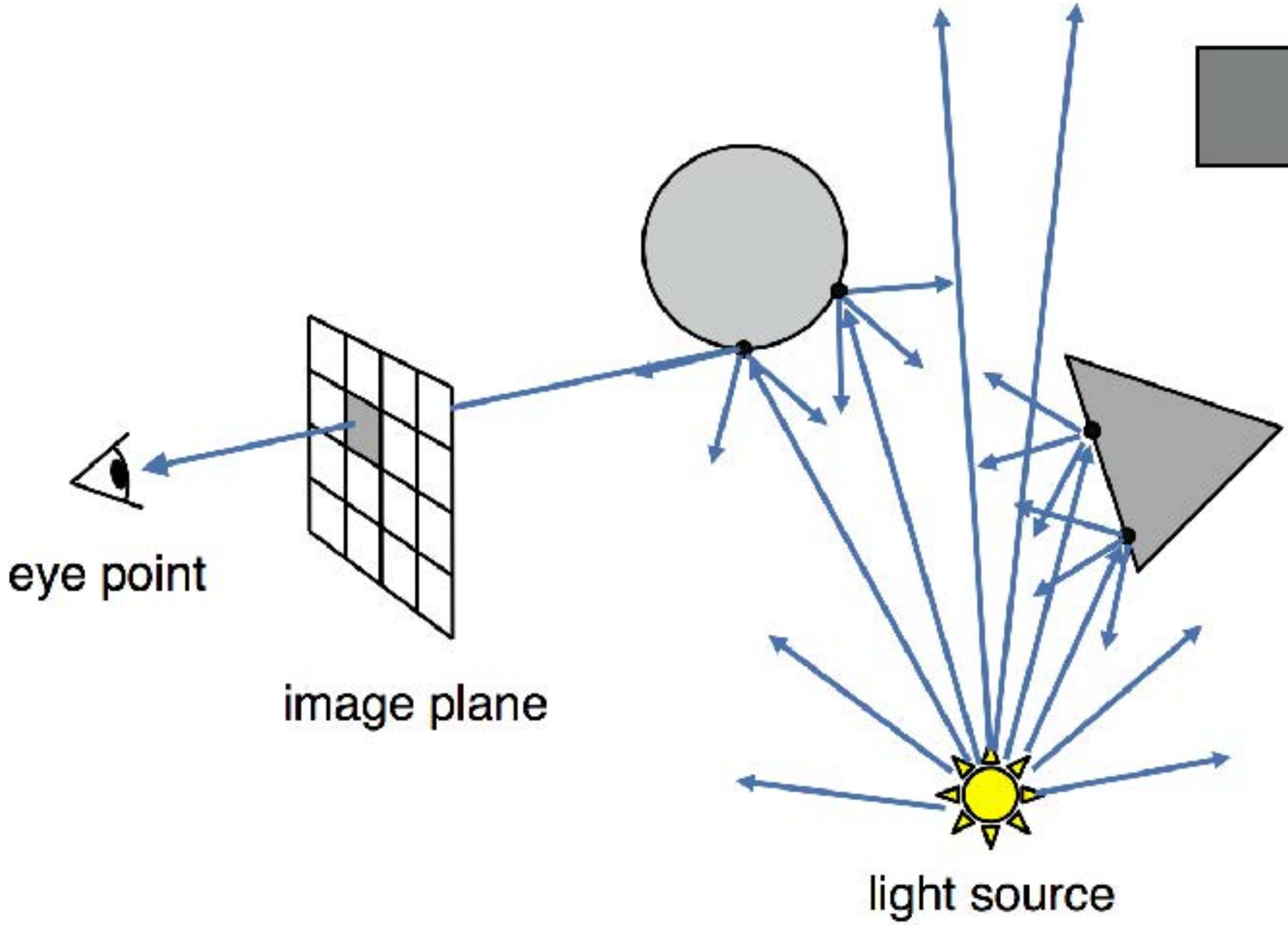
Surface Scattering

Ray-Object Intersections

Light Transport

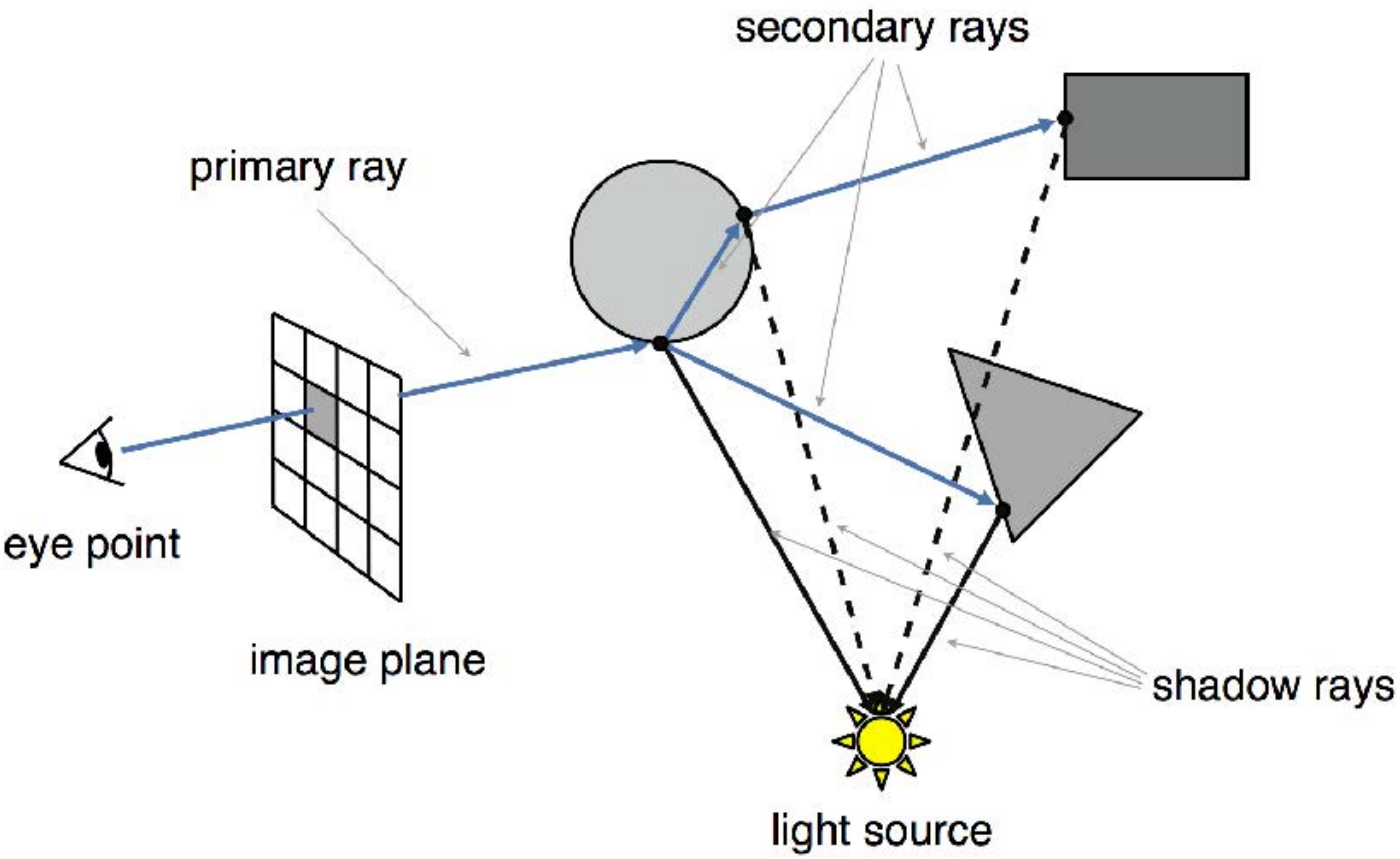
- Light travels in **straight lines**
- Light rays **do not interfere** with each other if they cross
- Light travels from the **light sources to the eye** (physics is invariant under path reversal reciprocity)

Light-Oriented (Forward Raytracing)



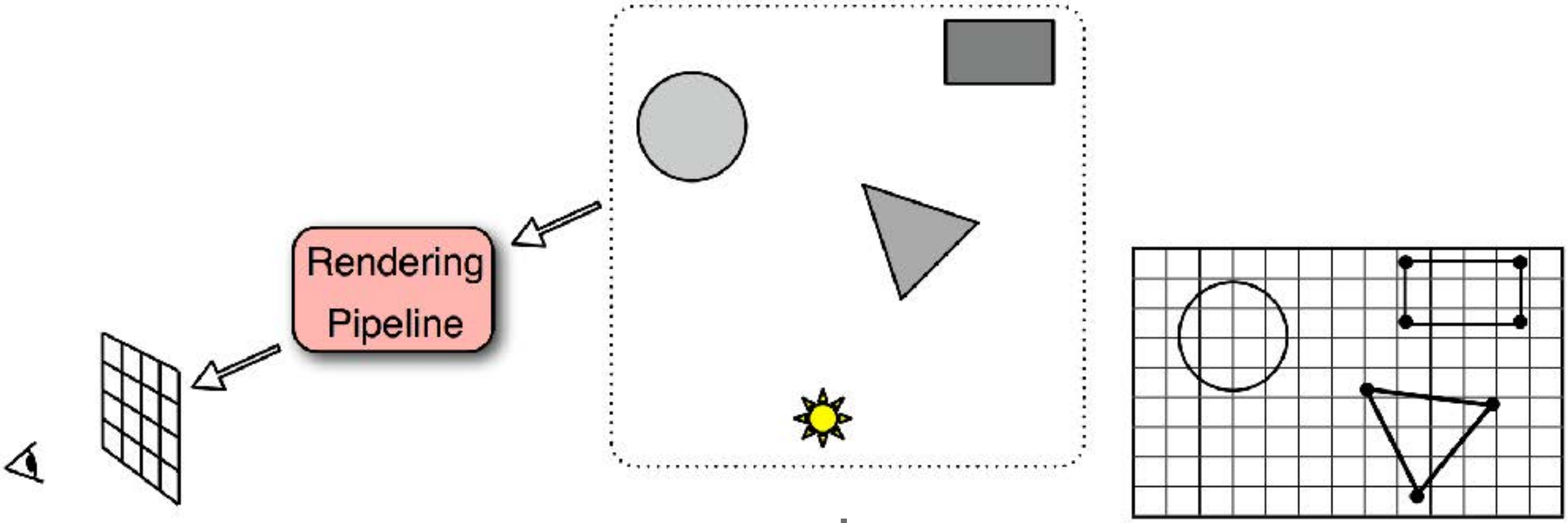
Only a **fraction** of light rays reach the image

Eye-Oriented (Backward Raytracing)



or simply **“Raytracing”**

Object-Oriented (Forward Rendering)



Rendering Pipeline

vector

raster

rasterization

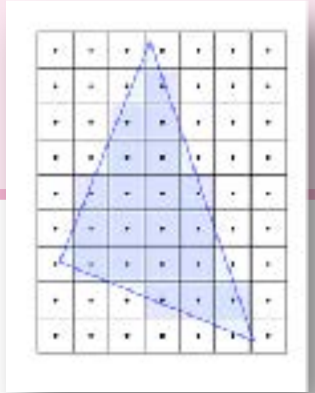
Scene is composed of **geometric structures** with the buiding block of **a triangle**. Each triangle is projected, colored, and painted on the screen

Light vs. Eye vs. Object-Oriented Rendering

- **Light-oriented (Forward Raytracing)**
  - light sources send off photons in all directions and hits camera

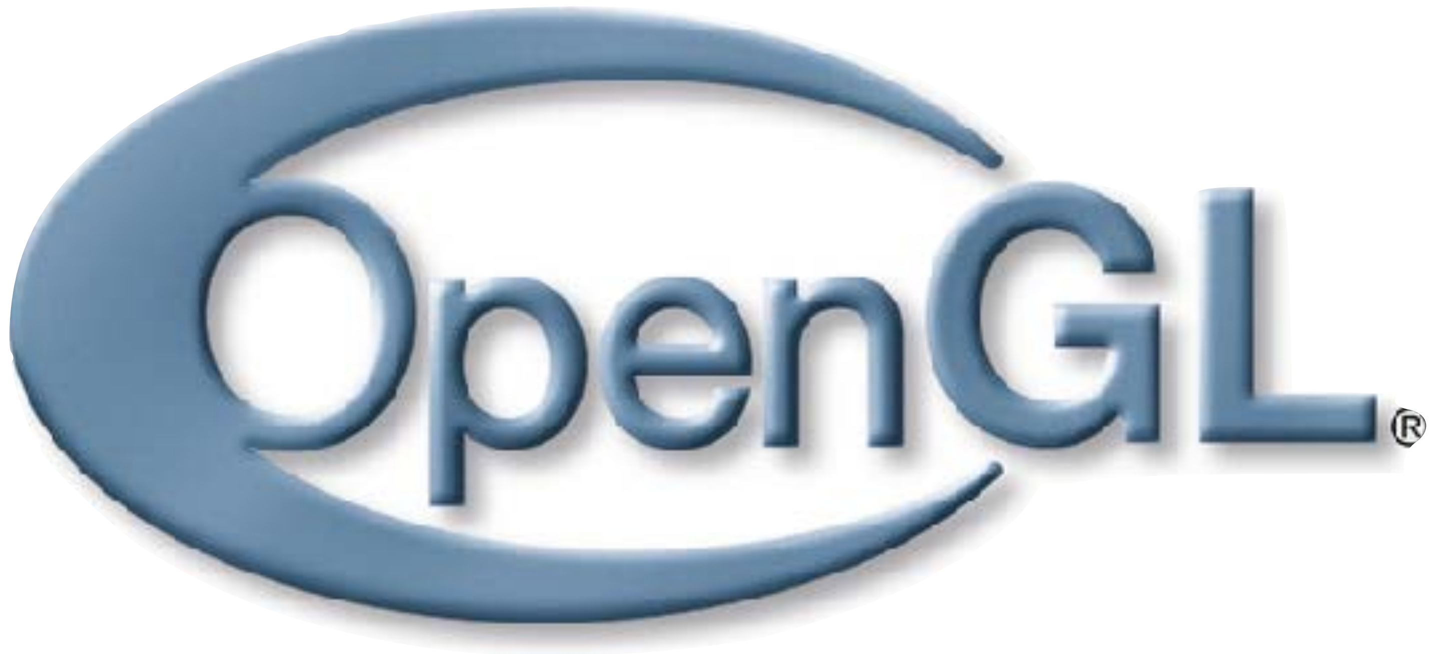
- **Eye-oriented (Backward Raytracing or simply Raytracing)**
  - walk through each pixel looking for what object (if any) should be shown there

- **Object-oriented (OpenGL):**
  - walk through objects, transforming and then drawing each one unless the z-buffer says that it's not in front



Let’s leave rasterization to the GPU

OpenGL



Industry Standard API for Computer Graphics

Alternatives

interactive, but not cross-platform

OpenGL Family

What is OpenGL?

- **Low-level graphics library (API)** for 2D and 3D interactive Graphics.
- Descendent of GL (from SGI)
- First version in 1992; now: 4.2 (2012)
- Managed by Khronos Group (non-profit consortium)
- API is governed by Architecture Review Board (part of Khronos)

Where is OpenGL used?

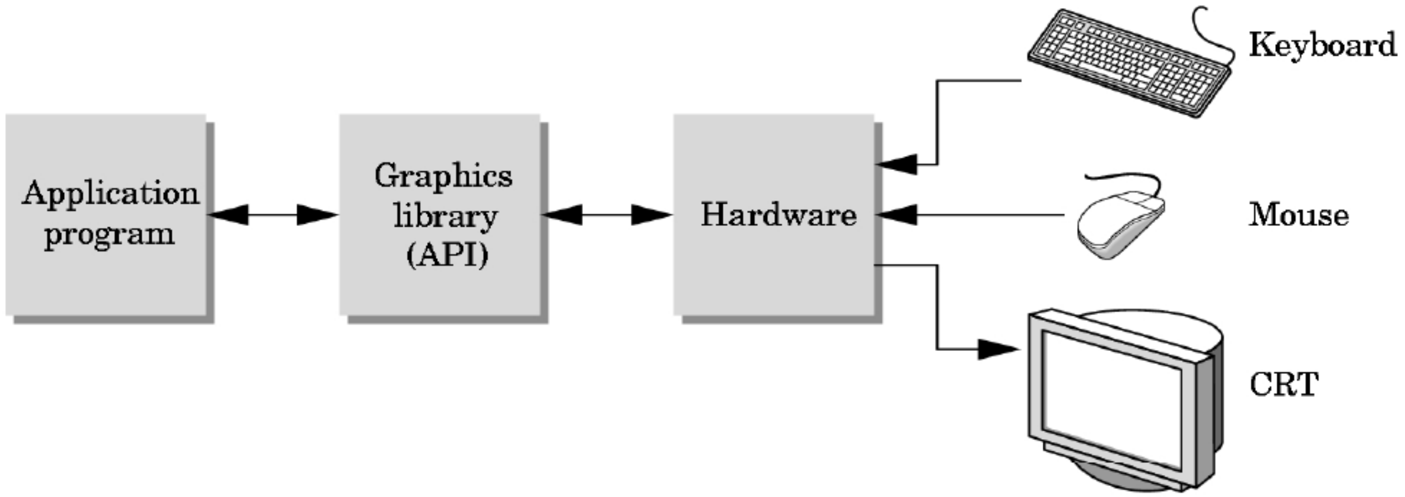
- CAD
- VR/AR
- Scientific Visualization
- Simulators
- Video games

Realtime Graphics Demo

Unreal Kite Demo (GTX TitanX)

Graphics Library (API)

- **Interface** between Application and Graphics Hardware



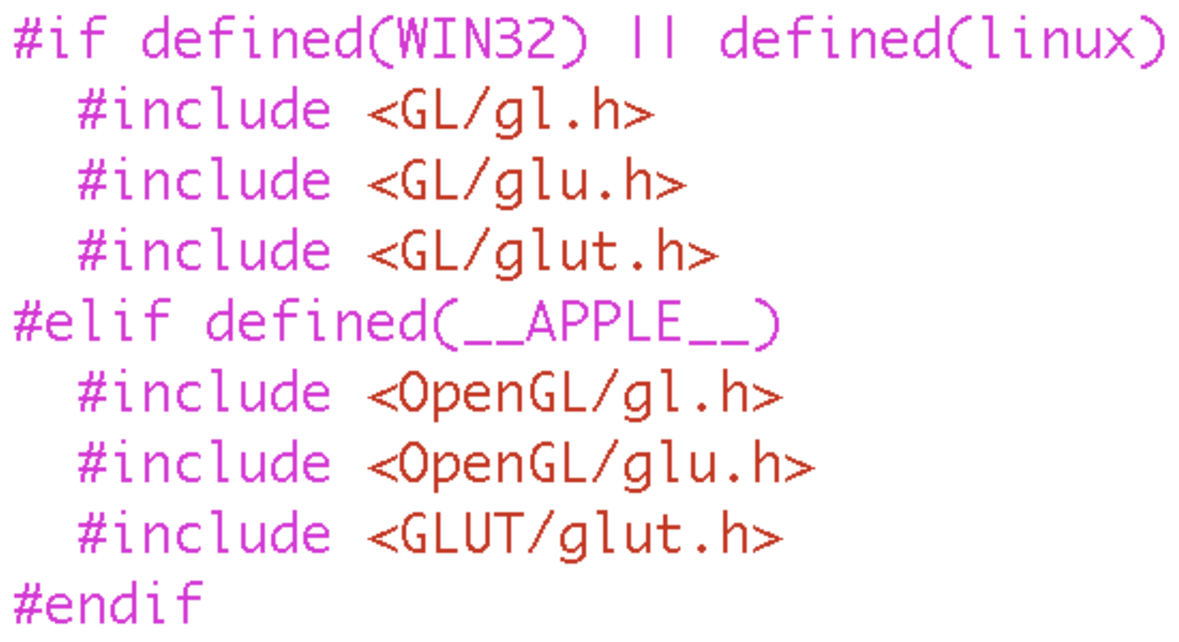
- Other popular APIs:
  - Direct3D (Microsoft) → XBox
  - OpenGL ES (embedded Devices)
  - X3D (successor of VRML)

OpenGL is cross-platform

- **Same code** works with little/no modifications

- **Implementations:**

Mac, Linux, Windows: ships with the OS
Linux: Mesa, freeware implementation

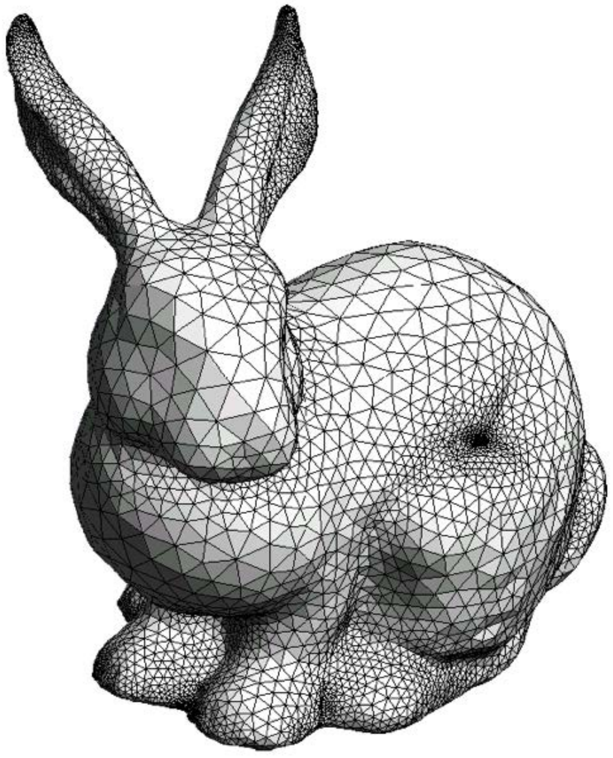


```
#if defined(WIN32) || defined(linux)
  #include <GL/gl.h>
  #include <GL/glu.h>
  #include <GL/glut.h>
#elif defined(__APPLE__)
  #include <OpenGL/gl.h>
  #include <OpenGL/glu.h>
  #include <GLUT/glut.h>
#endif
```

How does OpenGL work

From the programmer's point of view:

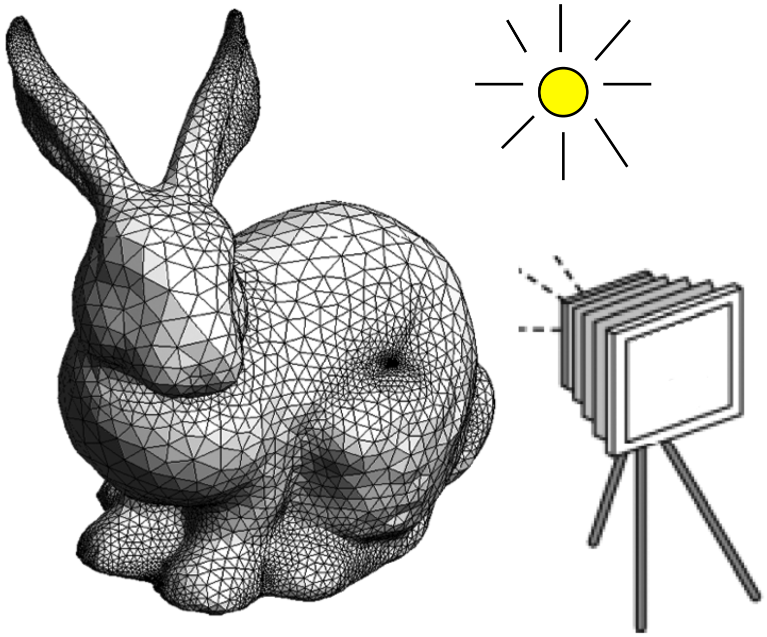
- Specify **geometric objects**
- Describe **object properties**
  - Color
  - How objects reflect light



How does OpenGL work (continued)

Define how objects should be viewed

- where is the camera?
- what type of camera?

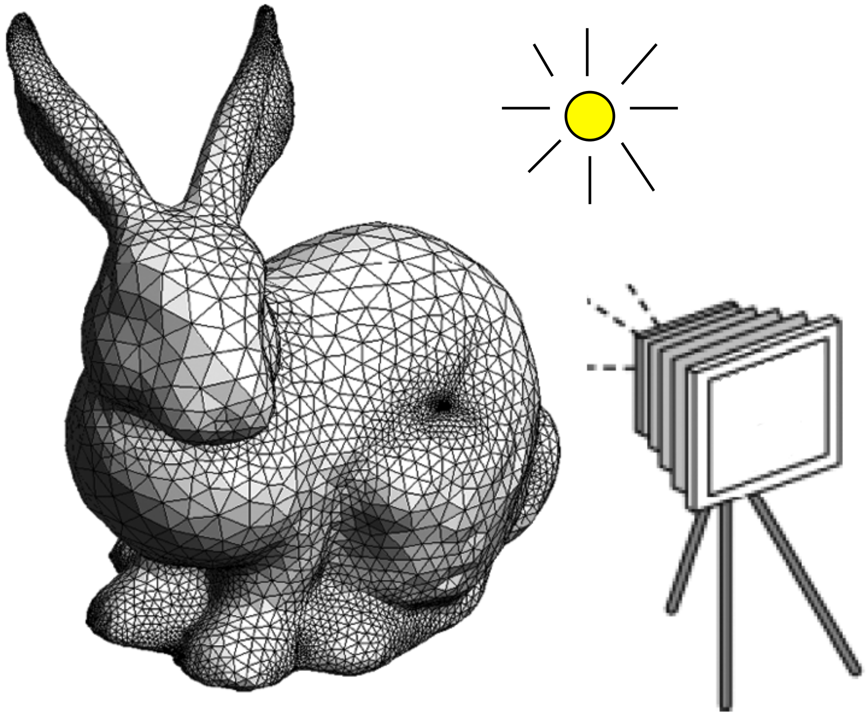


Specify light sources

- where, what kind?

Move camera or objects around for animation

The result



the scene

the result

OpenGL is a state machine

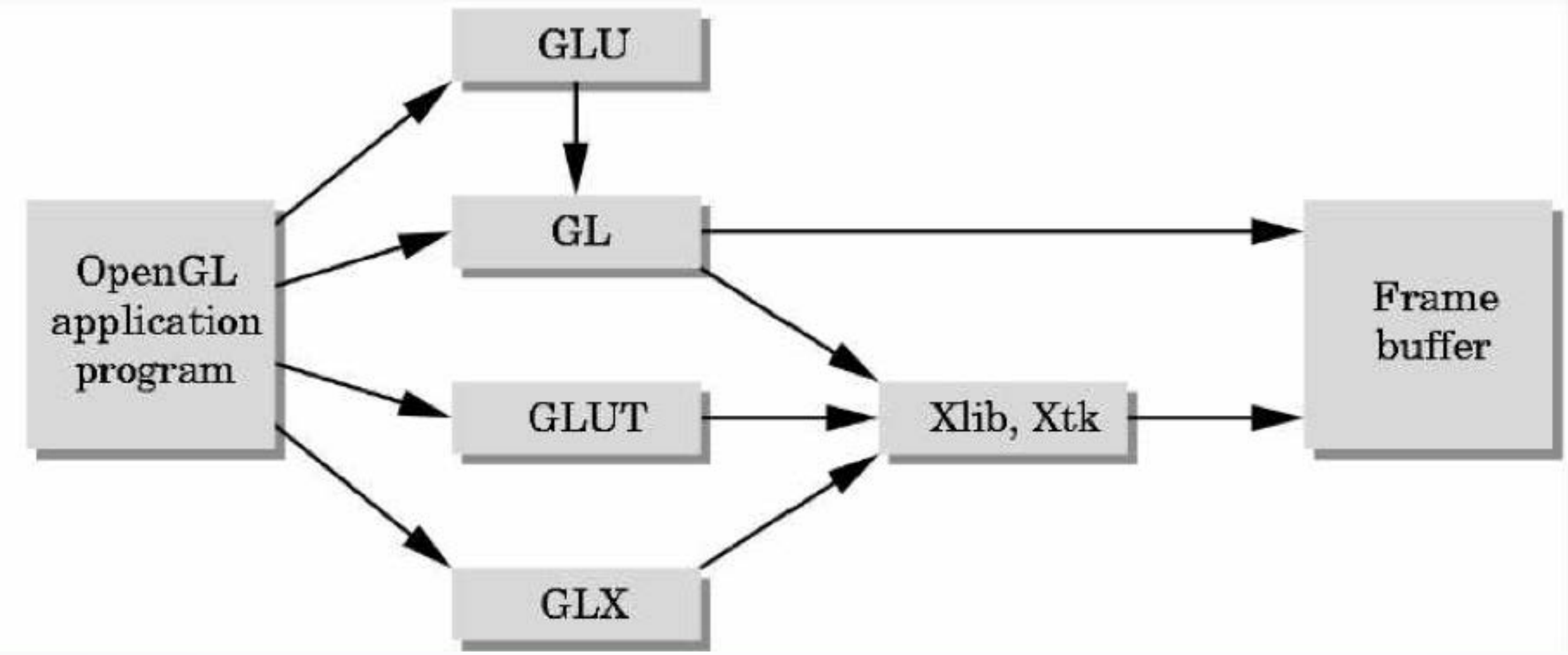
State variables: color, camera position, light position, material properties…

These variables (**the state**) then apply to every subsequent drawing command.

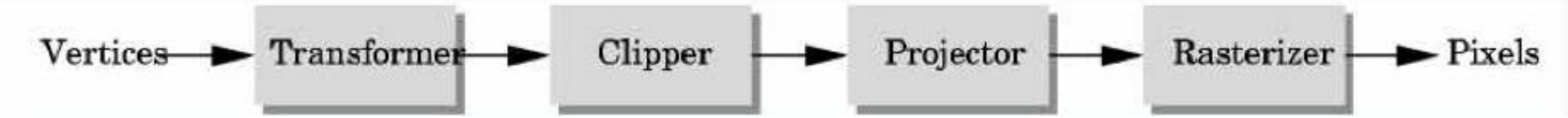
They **persist** until set to new values by the programmer.

OpenGL Library Organization

- **GL (Graphics Library):** core graphics capabilities
- **GLU (OpenGL Utility Library):** utilities on top of GL
- **GLUT (OpenGL Utility Toolkit):** input and windowing wrapper



OpenGL **Graphics Pipeline**



primitives+
material
properties

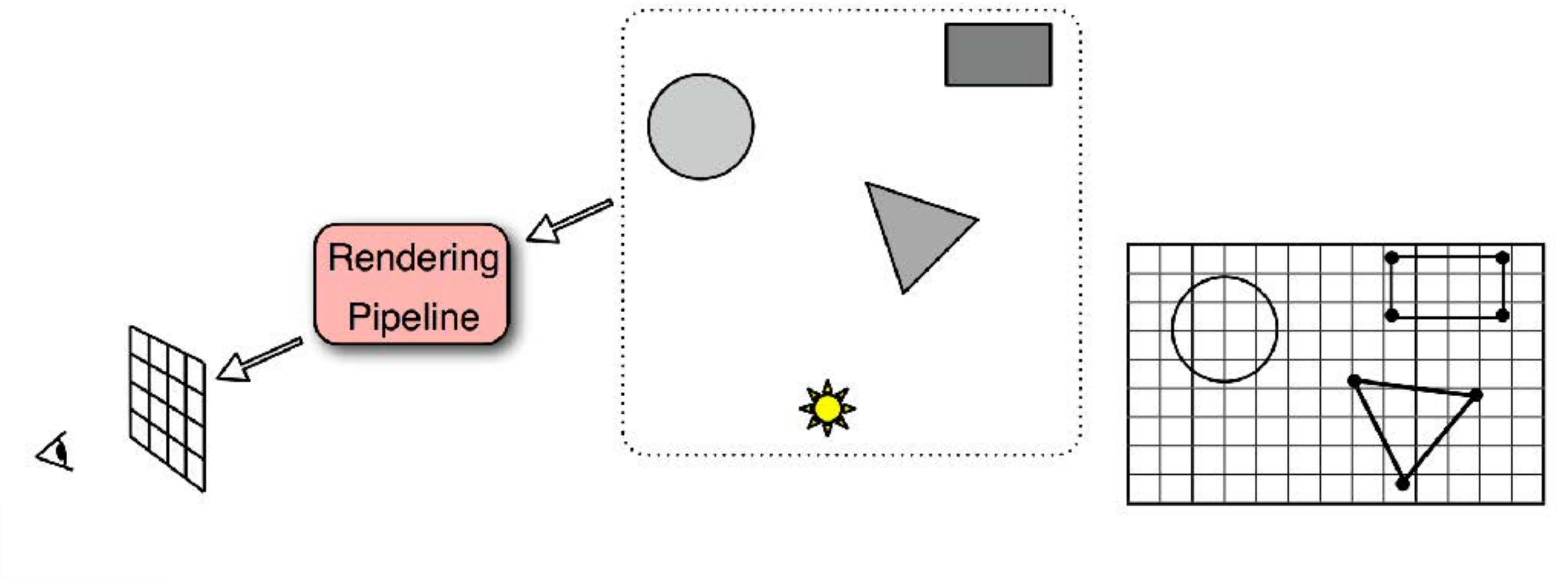
translate
rotate
scale

is it visible
on screen?

3D to 2D

convert to
pixels

shown
on the screen
(framebuffer)



OpenGL uses immediate-mode rendering

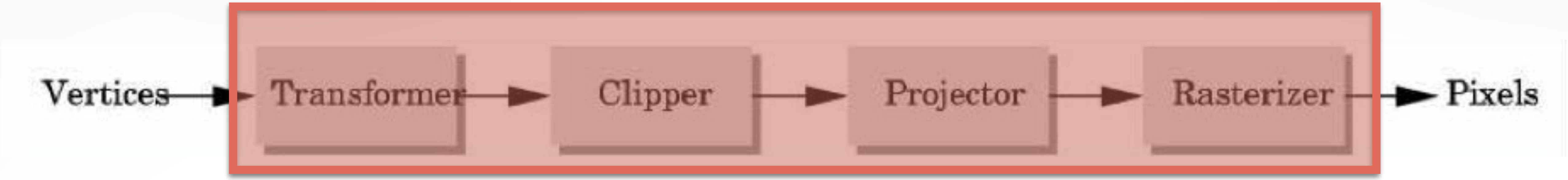
Application generates **stream of geometric primitives** (polygons, lines)

System **draws** each one into the **frame buffer**

Entire scene is **redrawn** for every frame

Compare to: offline rendering (e.g., Pixar Renderman, ray tracers…)

OpenGL **Graphics Pipeline**



primitives+ material properties

translate rotate scale

is it visible on screen?

3D to 2D

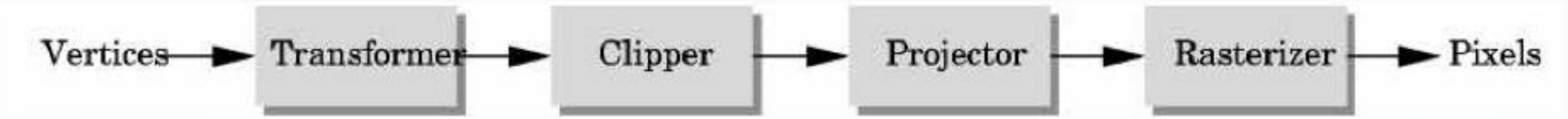
convert to pixels

shown on the screen (framebuffer)

implemented by **OpenGL, graphics driver, graphics hardware**

OpenGL programmer does not need to implement the pipeline, but can **reconfigure it through shaders**

OpenGL **Graphics Pipeline**



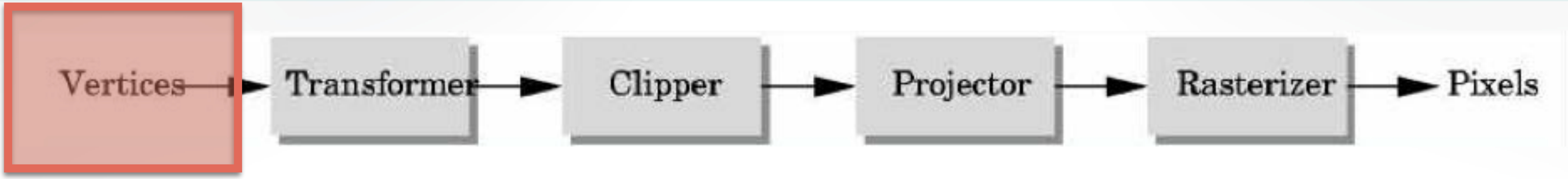
- **Efficiently implementable in hardware** (but not in software)

- Each stage can employ **multiple** specialized processors, working in **parallel**, busses between stages

- **#processors per stage**, bus bandwidths are fully tuned for typical graphics use

- **Latency vs throughput**

Vertices



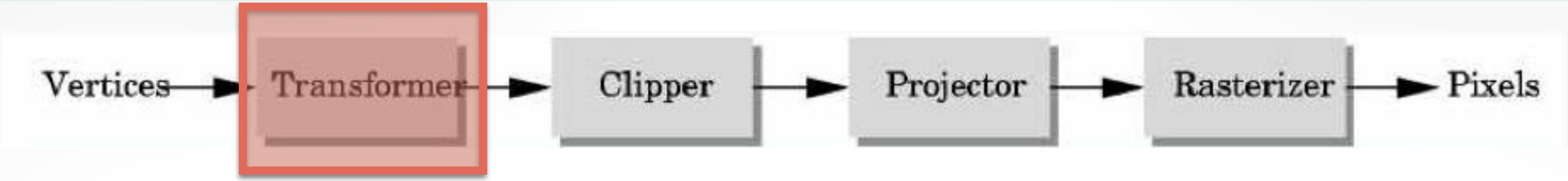
- Vertices in **world coordinates**

- **void glVertex3f(GLfloat x, GLfloat y, GLfloat z)**
  - Vertex(x,y,z) is sent down the pipeline.
  - Function call then returns

- Use **GLtype (e.g., GLfloat)** for portability and consistency

- glVertex{234}{sfid}(TYPE coords)

Transformer

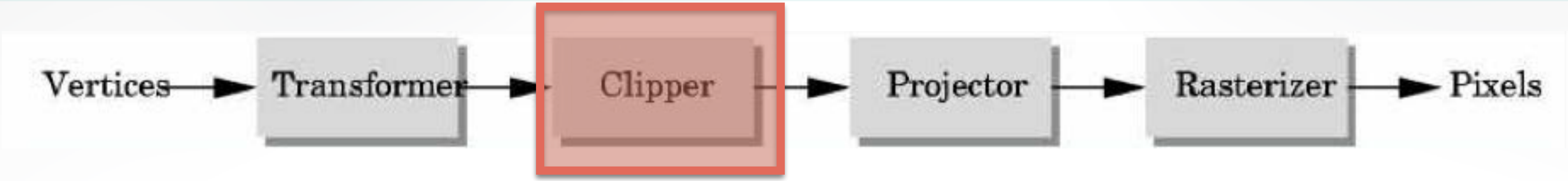


- Transformer in **world coordinates**

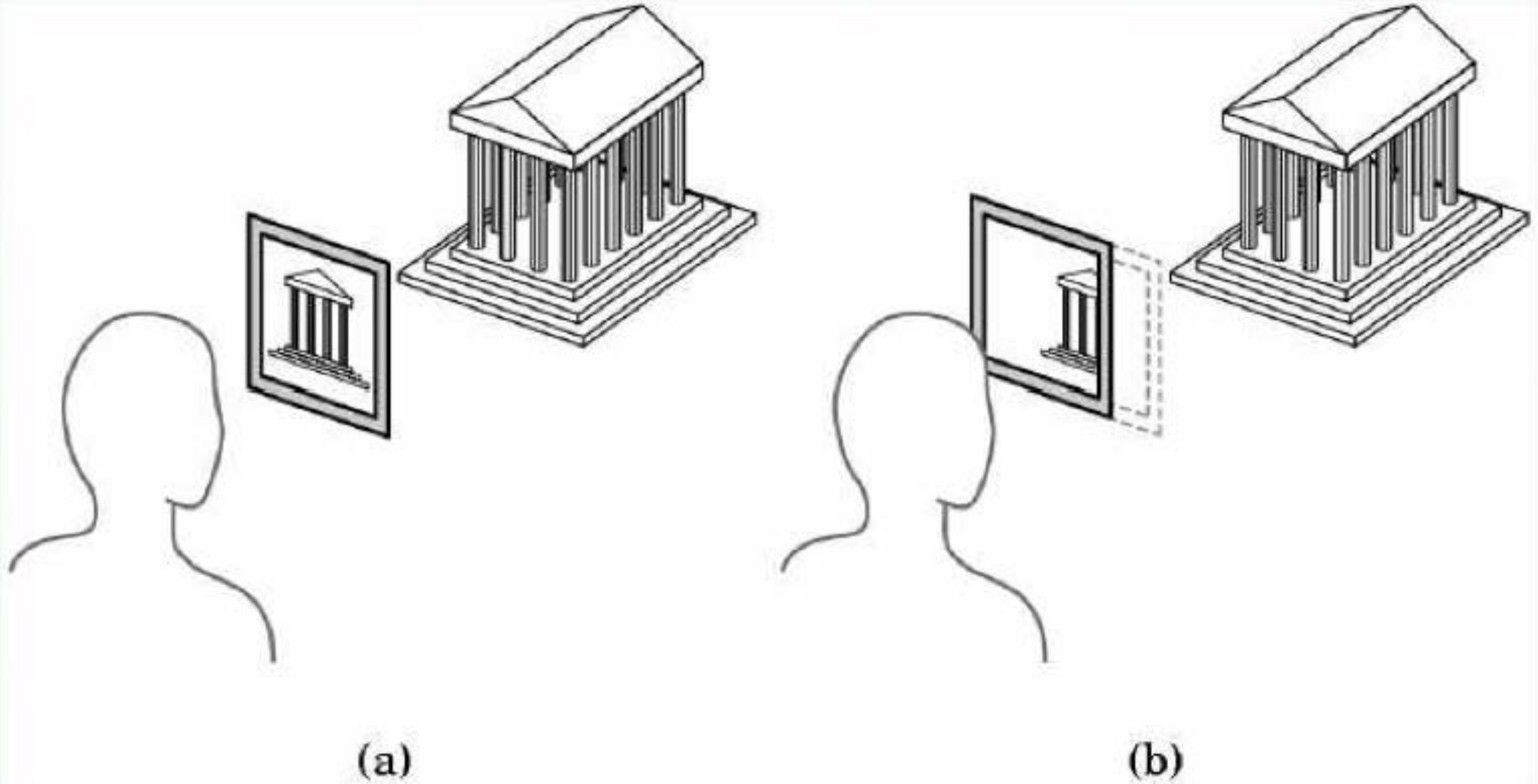
- **Must be set before object is drawn!**
  - glRotate (45.0, 0.0, 0.0, -1.0);
  - glVertex2f(1.0, 0.0);

- Complex [Angel Ch. 4]

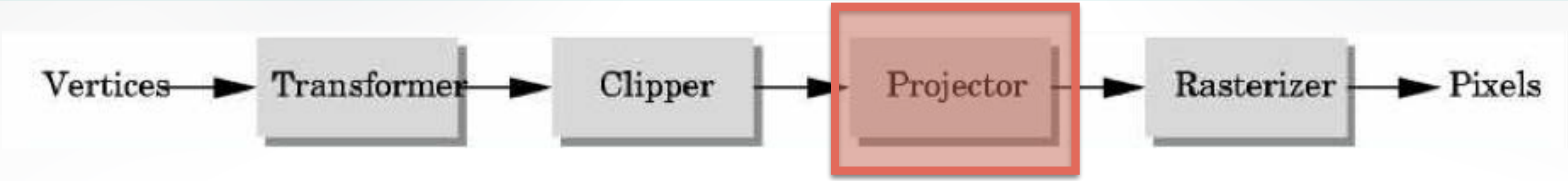
Clipper



- Mostly automatic (**must set viewport**)



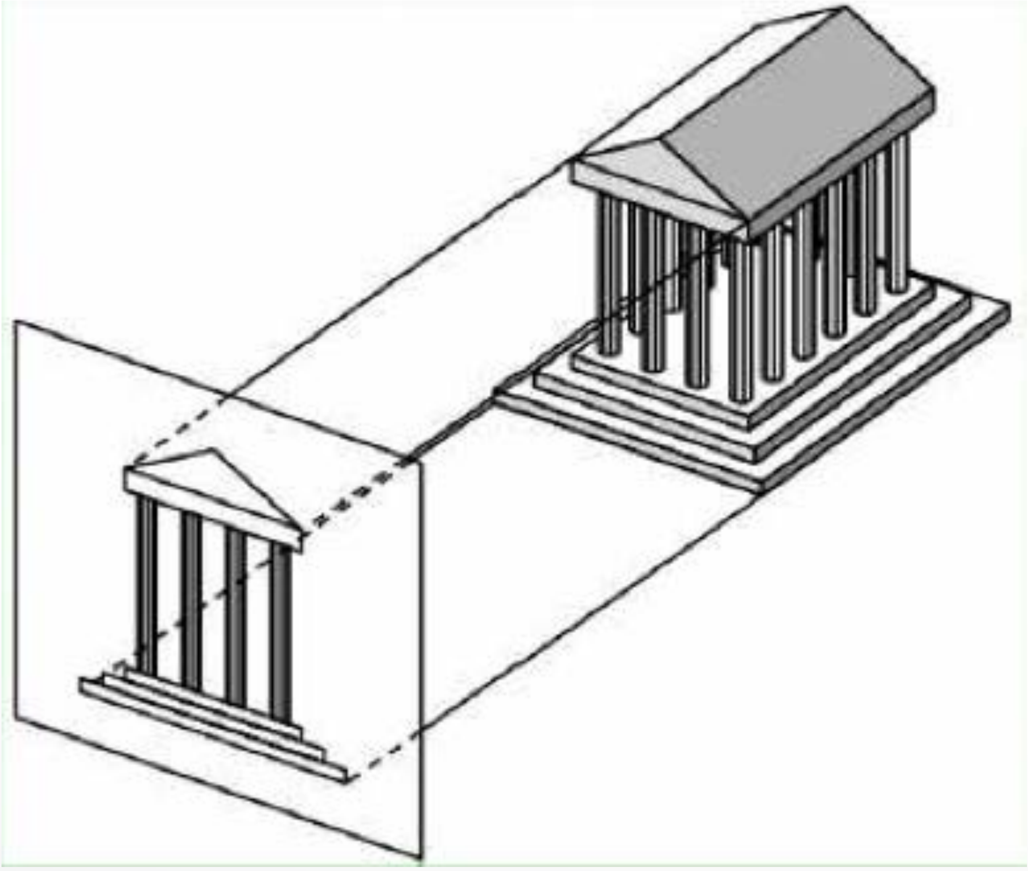
Projector



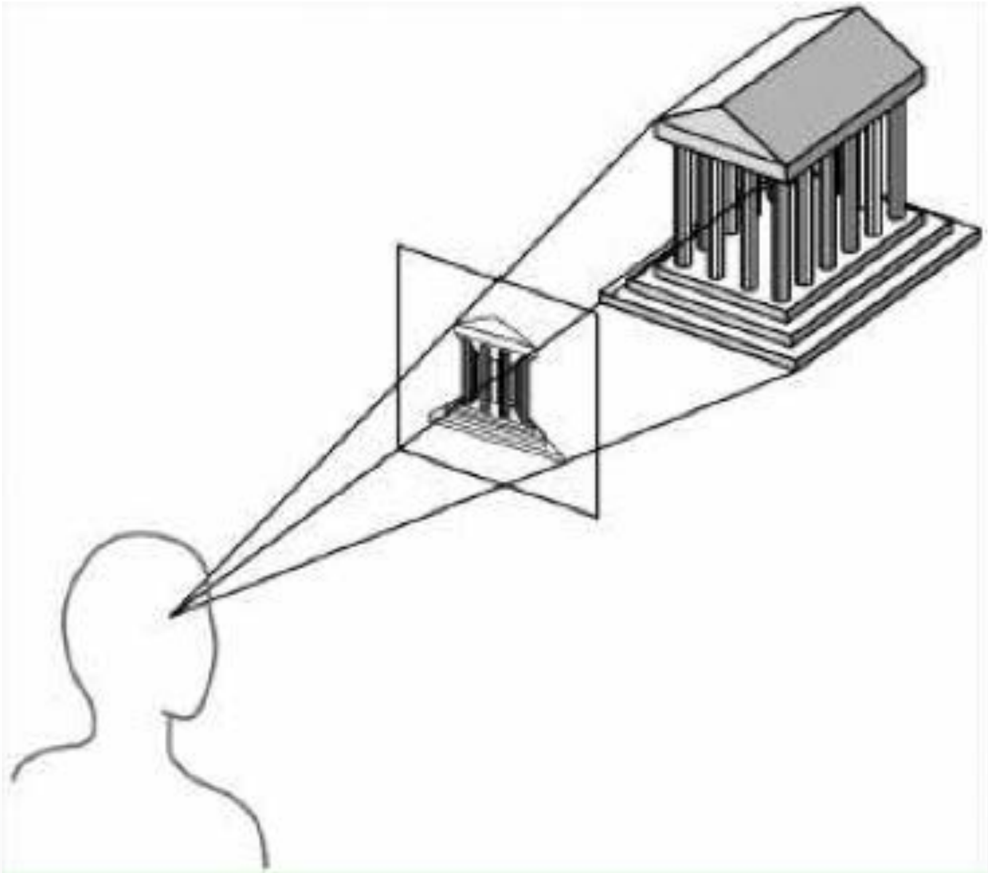
Vertices → Transformer → Clipper → Projector → Rasterizer → Pixels

- Complex transformation [Angel Ch. 5]

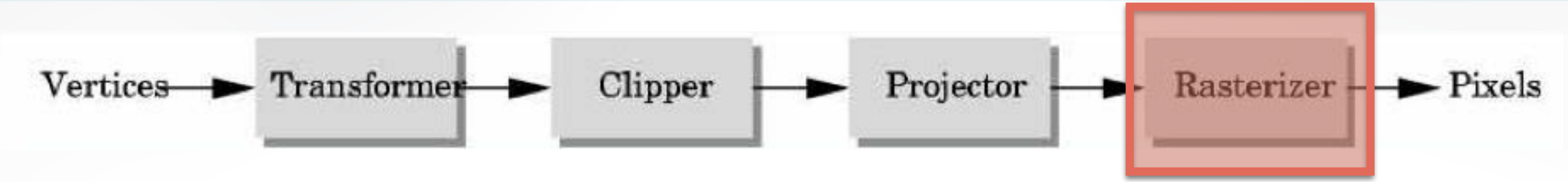
orthographic



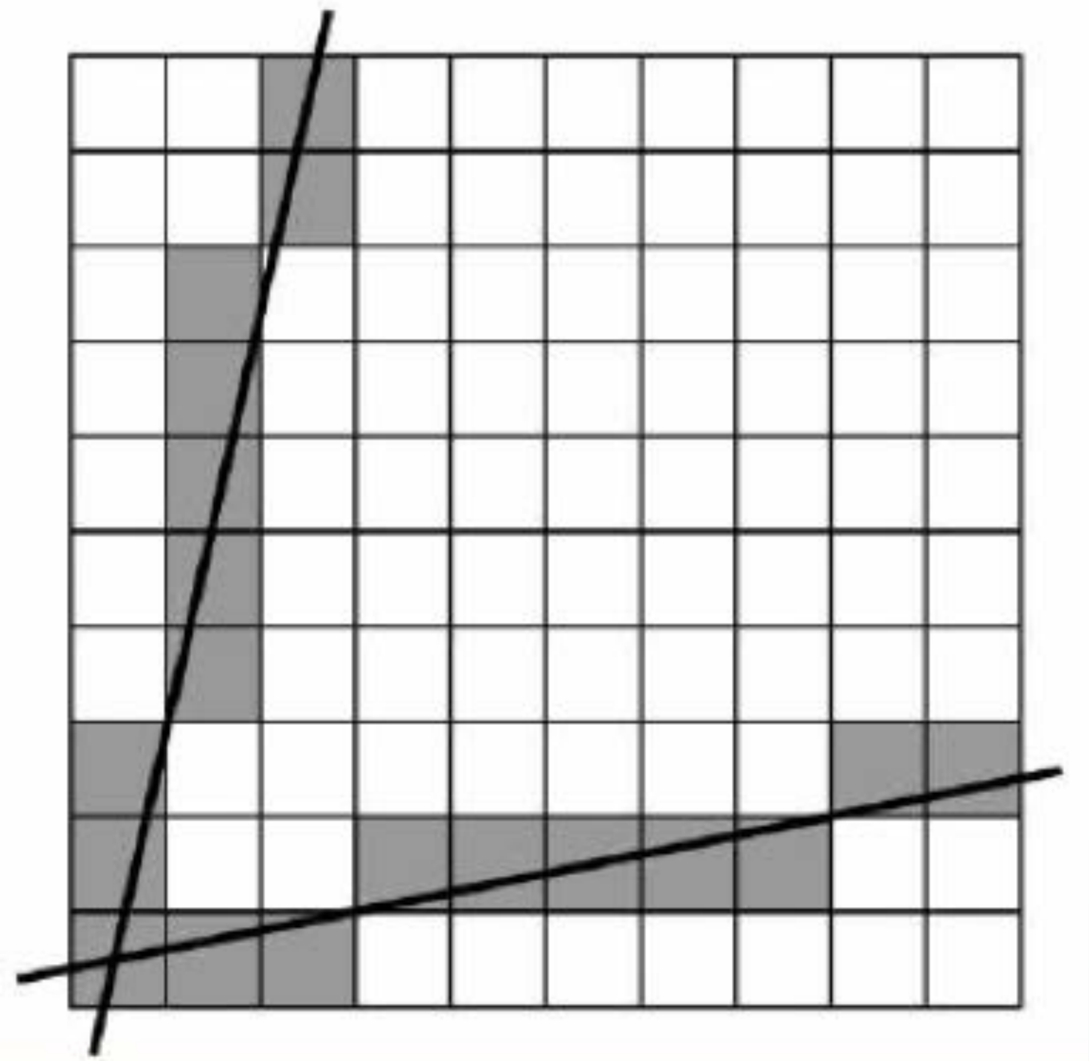
perspective



Rasterizer



- Interesting algorithms [Angel Ch. 7]
- **To window coordinates**
- Antialiasing

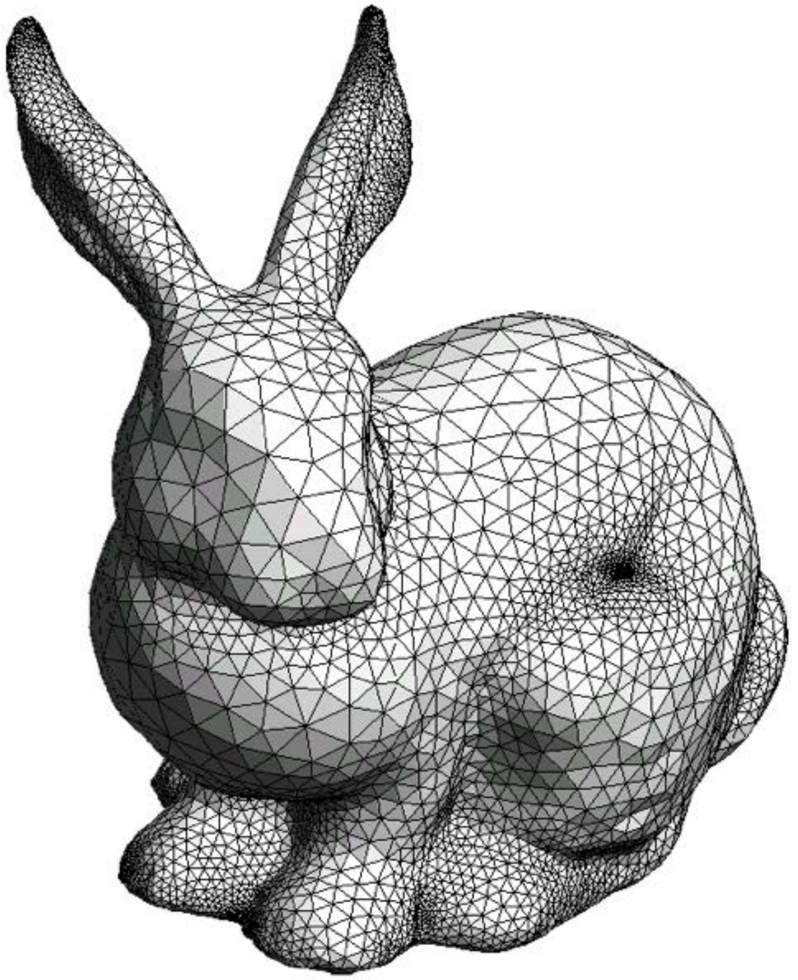


Primitives

- Specified via vertices

- General scheme

```
glBegin(type);
    glVertex3f(x1,y1,z1);
    …
    glVertex3f(xN,yN,zN);
glEnd();
```



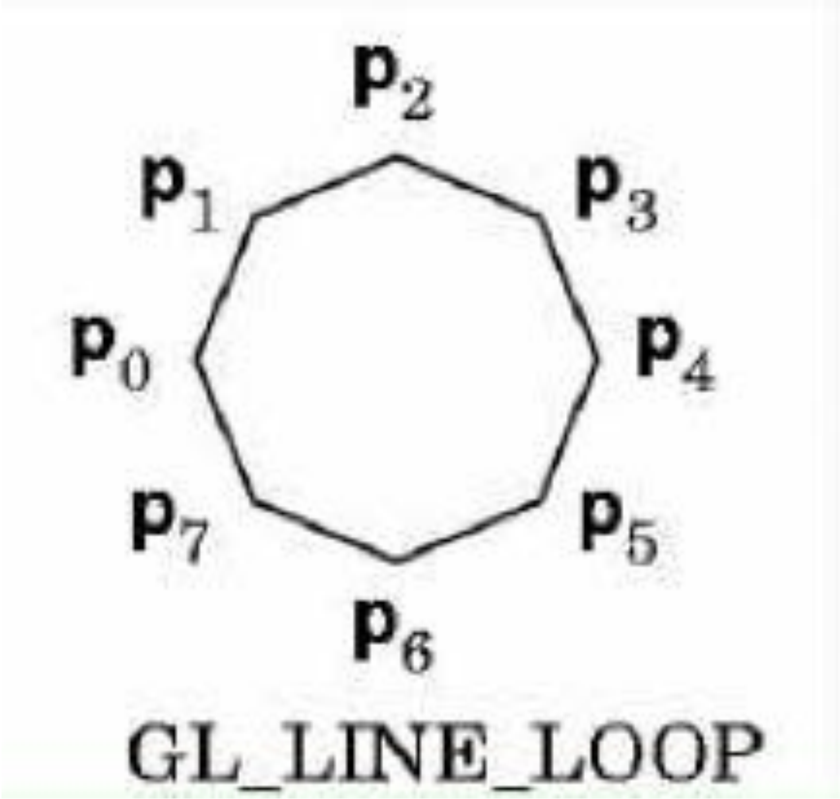
- **type** determines interpretation of vertices

- Can use glVertex2f(x,y) in 2D

Example: Draw Square Outline

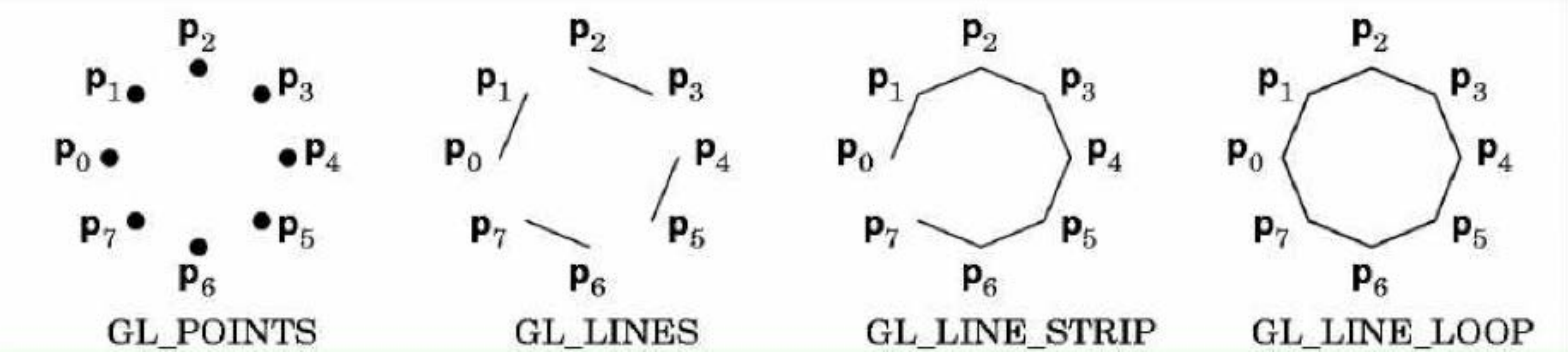
- **Type = GL_LINE_LOOP**

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



- Calls to other functions are allowed between glBegin(Type) and glEnd()

Points and Line Segments

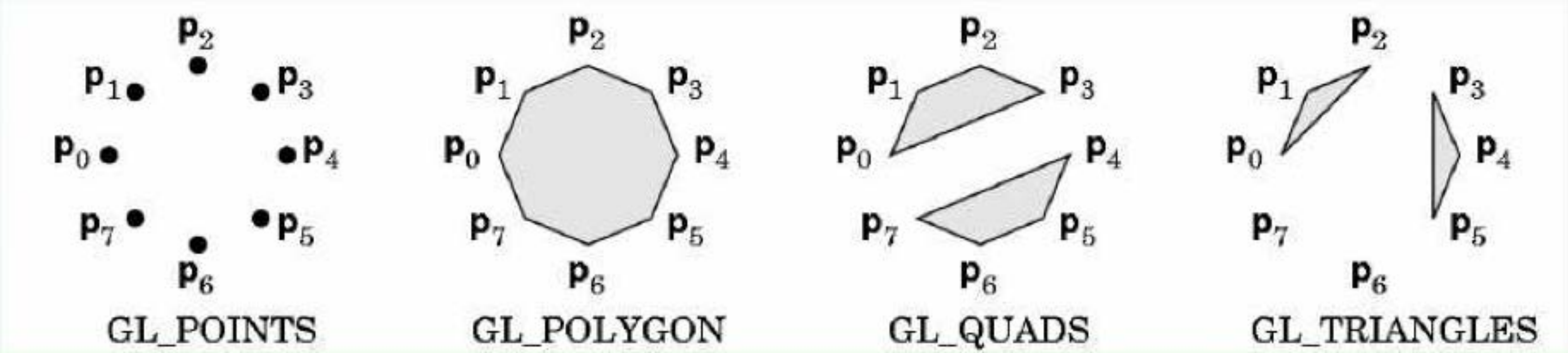


```
glBegin(GL_POINTS);
    glVertex3f(…);
    …
    glVertex3f(…);
glEnd()
```

draw points

Polygons

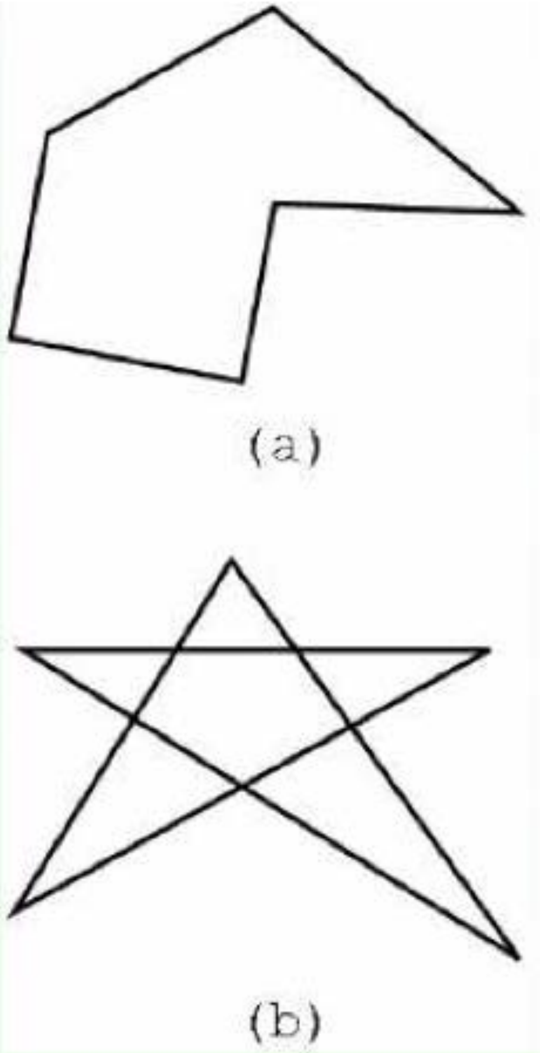
- **Polygons enclose an area**



- Rendering of area (fill) depends on attributes
- **All vertices must be in one plane in 3D**

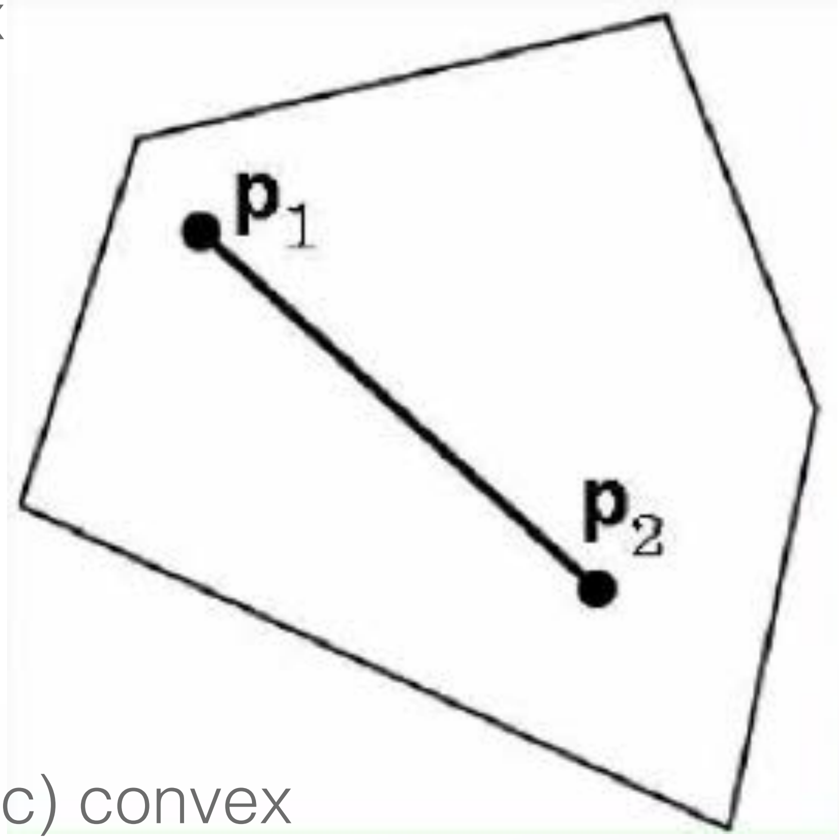
Polygons Restrictions

- OpenGL Polygons must be **simple**
- OpenGL Polygons must be **convex**



(a) simple, but not convex

(b) non-simple



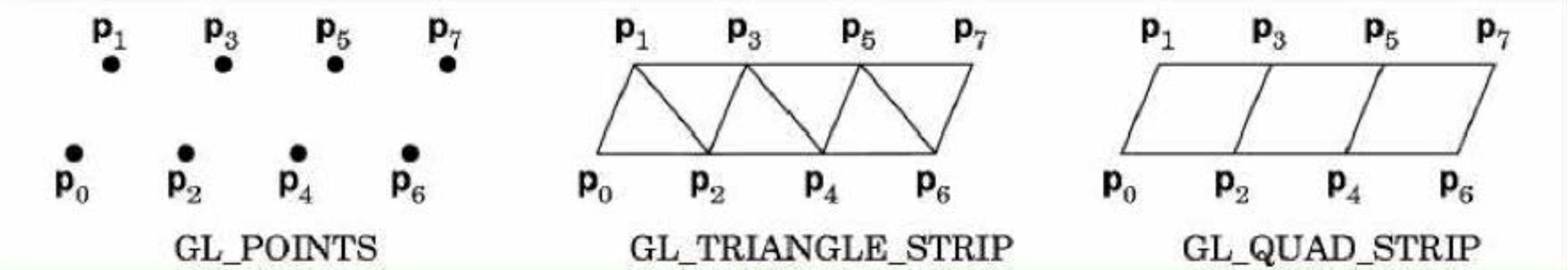
(c) convex

Why **Polygons Restrictions?**

- Non-convex and non-simple polygons are **expensive to process and render**
- Convexity and simplicity is **expensive to test**
- Behavior of **OpenGL** implementation on disallowed polygons is **“undefined”**
- Some tools in GLU for decomposing complex polygons (**tessellation**)
- **Triangles are most efficient**

Polygons Strips

- **Efficiency in space and time**
- Reduces visual **artifacts**



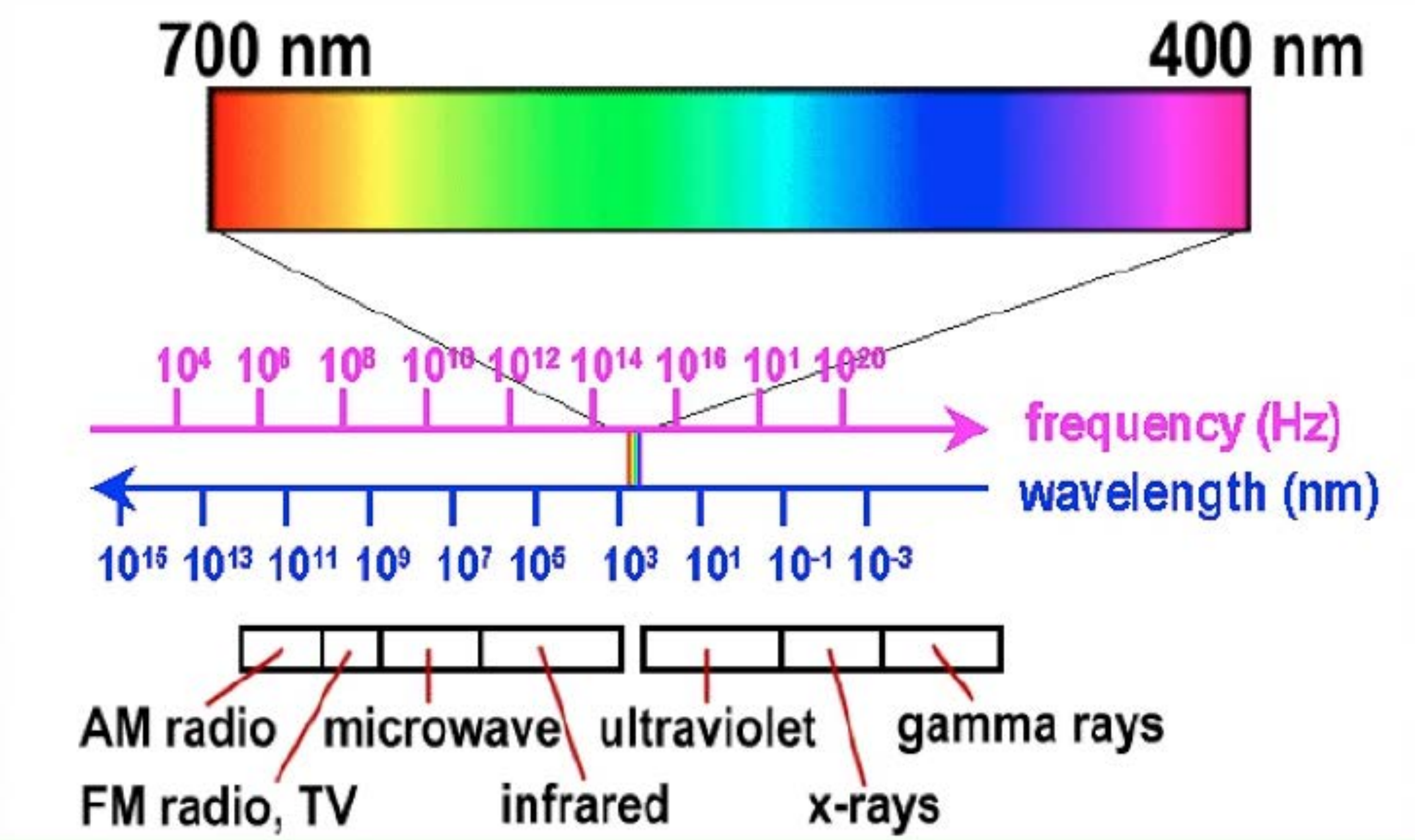
- Polygons have a **front and a back**, possibly with different attributes!

Attributes: Color, Shading, Reflections

- Part of the OpenGL **state**
- Set **before** primitives are drawn
- **Remain in effect until changed!**

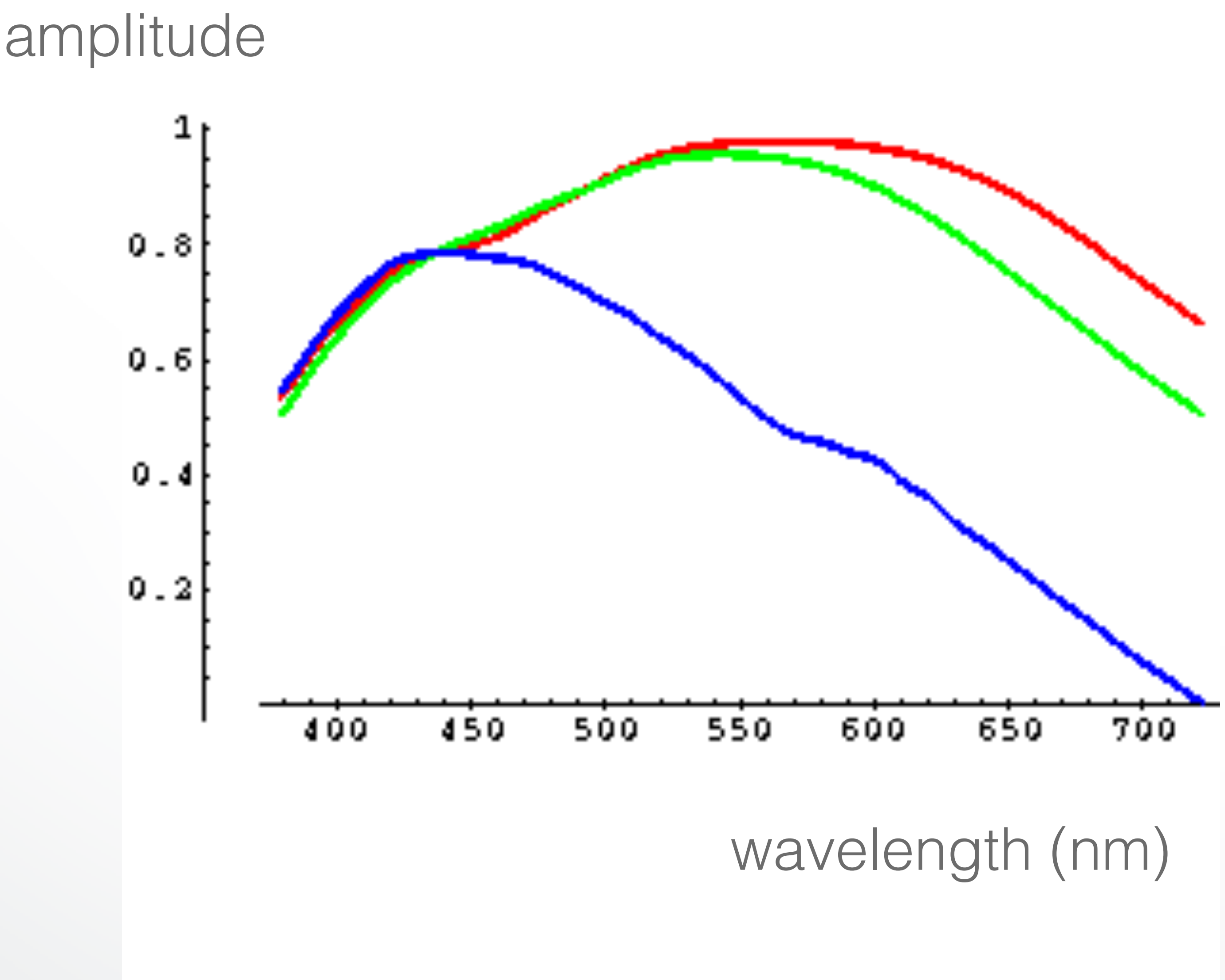
Physics of Color

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



Color Filters

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



Cone response

Source: VOS & Walraven

Color Spaces

- **RGB (Red, Green, Blue)**

  Convenient for display

  Can be unintuitive (3 floats in OpenGL)

- **HSV (Hue, Saturation, Value)**

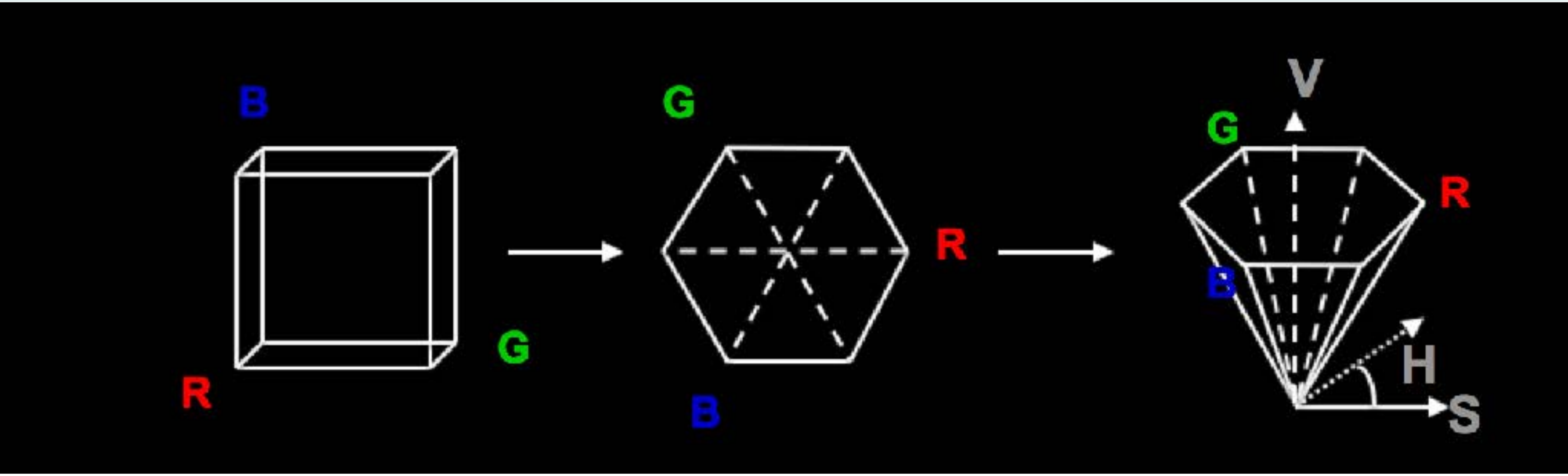
  Hue: what color?

  Saturation: how far away from gray?

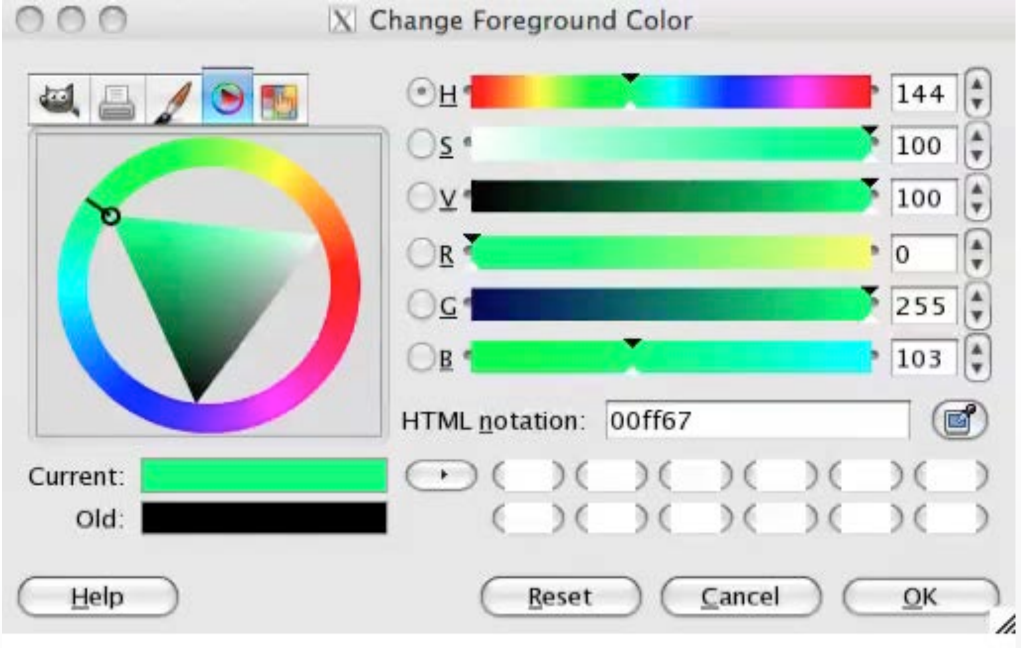
  Value: how bright?

- Other formats for **movies and printing**

RGB vs HSV



Gimp Color Picker



● H 144

○ S 100

○ V 100

○ R 0

○ G 255

○ B 103

HTML notation: 00ff67

Current:

Old:

Example: Drawing a shaded polygon

- Initialization: the “main” function

```
int main(int argc, char ** argv)
{
    glutInit(&argc,argv);
    glutInitDisplayMode(GLUT_DOUBLE|GLUT_RGB);
    glutInitWindowSize(500,500);
    glutInitWindowPosition(100,100);
    glutCreateWindow(argv[0]);
    init();
…
```

GLUT Callbacks

- Window system **independent** interaction
- glutMainLoop processes events

```
…
    glutDisplayFunc(display);
    glutReshapeFunc(reshape);
    glutKeyboardFunc(keyboard);
    glutMainLoop();
    return 0;
}
```

Initializing Attributes

- Separate in “init” function

```
void init()
{
   glClearColor (0.0,0.0,0.0,0.0);
   // glShadeModel (GL_FLAT);
   glShadeModel (GL_SMOOTH);
}
```

The Display Callback

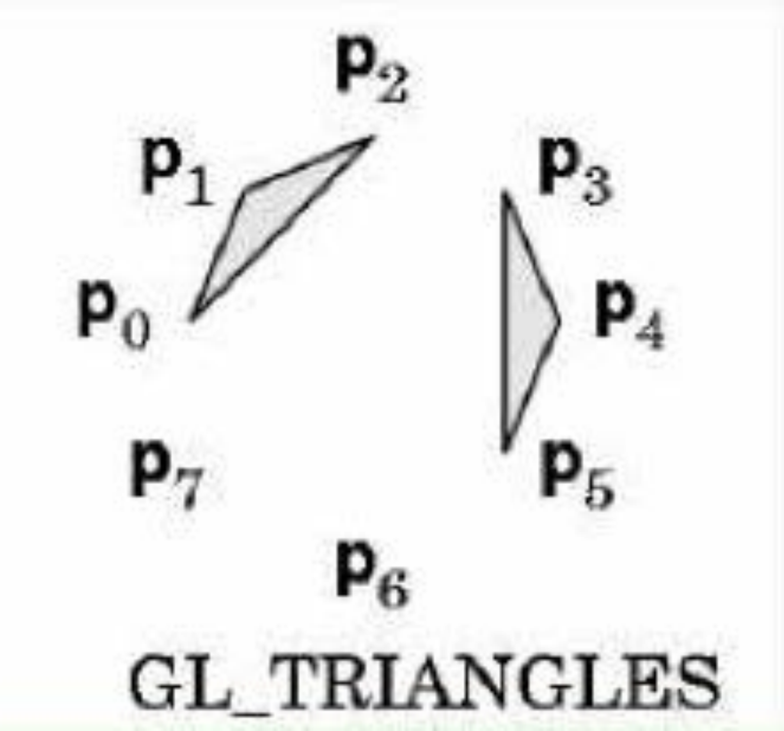
- The routine where you render the object
- Install with glutDisplayFunc(display)

```
void display()
{
   glClear(GL_COLOR_BUFFER_BIT); // clear buffer
   setupCamera();                // set up camera
   triangle();                   // draw triangle
   glutSwapBuffers();            // force display
}
```

Drawing

- In world coordinates; remember state!

```
void triangle()
{
   glBegin(GL_TRIANGLES);
      glColor3f(1.0,0.0,0.0); // red
      glVertex2f(5.0,5.0);
      glColor3f(0.0,1.0,0.0); // green
      glVertex2f(25.0,5.0);
      glColor3f(0.0,0.0,1.0); // blue
      glVertex2f(5.0,25.0);
   glEnd();
}
```



The Image

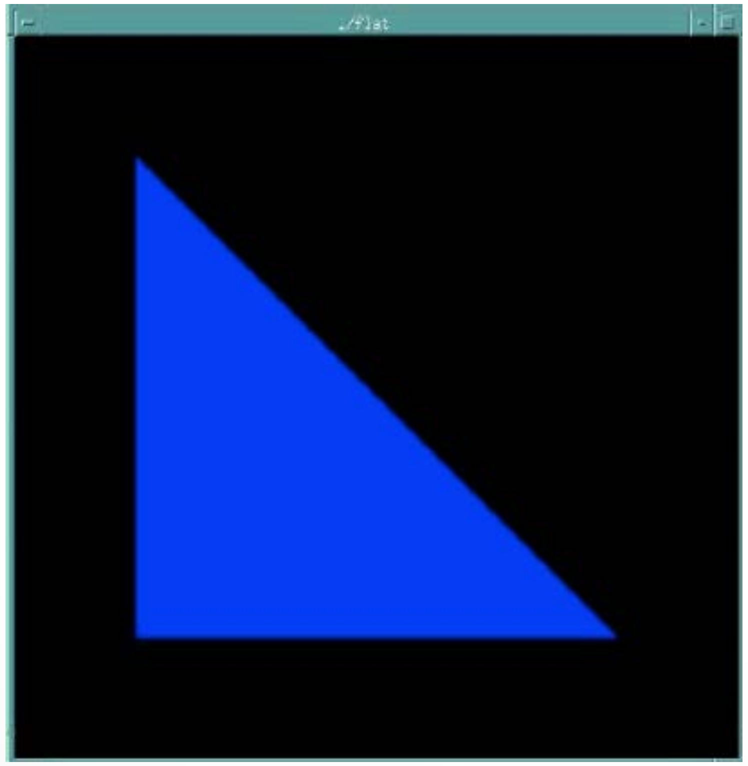
glShadeModel(GL_FLAT)

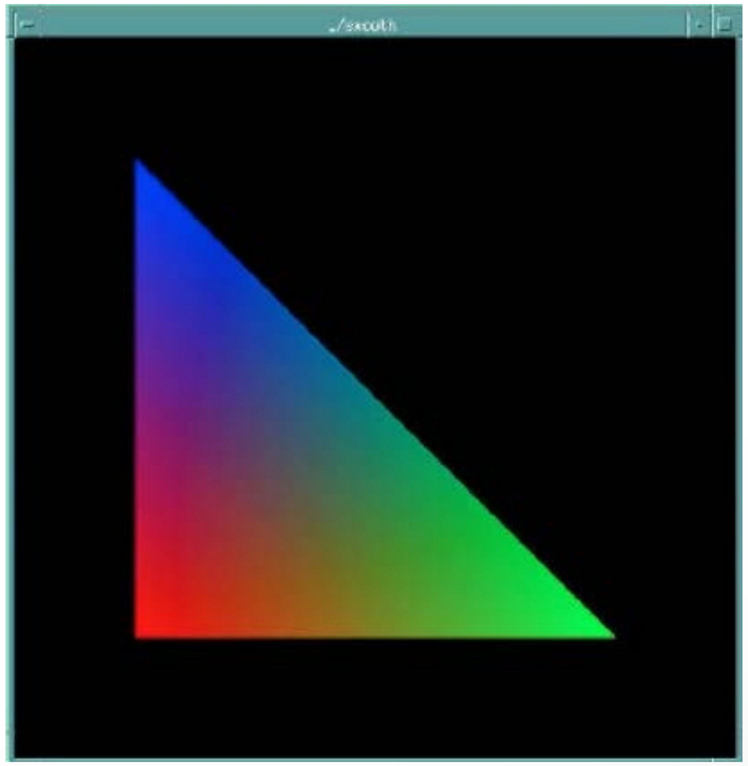
color of last vertex

glShadeModel(GL_SMOOTH)

each vertex separate color

smoothly interpolated





Flat vs Smooth Shading

Flat Shading

Smooth Shading

Projection

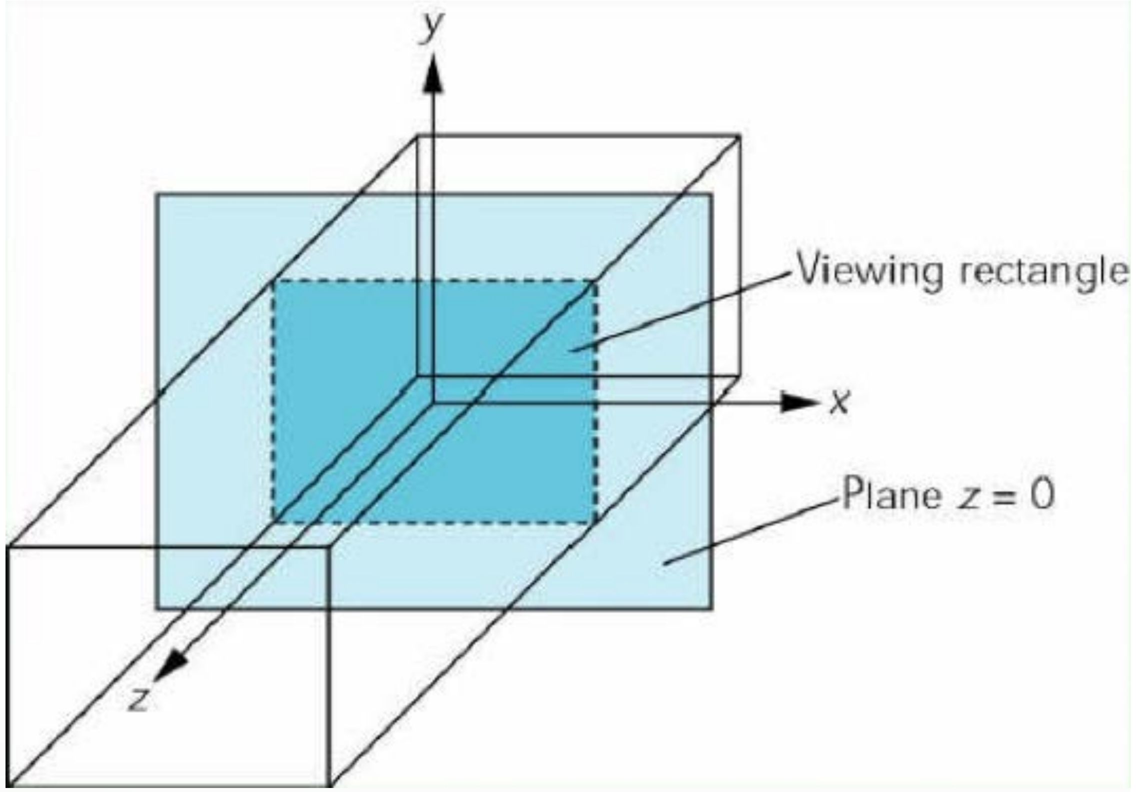
- Mapping world to screen coordinates

```
void reshape (int w, int h)
{
   glViewport(0, 0, (GLsizei) w, (GLsizei) h);
   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
   if(w<=h)
      gluOrtho2D(0.0,30.0,0.0,30.0 * (GLfloat) h/(GLfloat) w);
   else
      gluOrtho2D(0.0,30.0 * (GLfloat) w/(GLfloat) h, 0.0,30.0);
   glMatrixMode(GL_MODELVIEW);
}
```

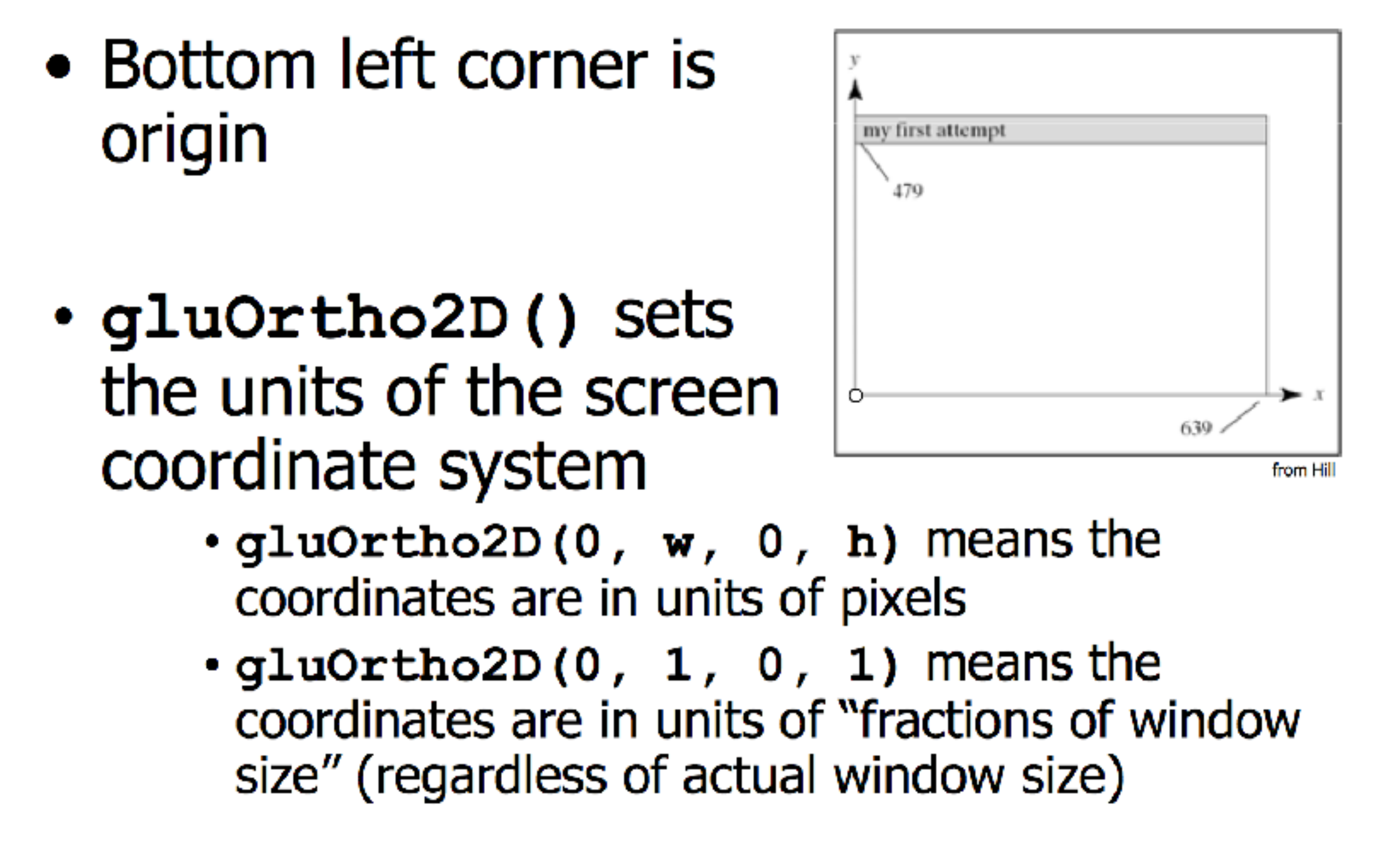
Orthographic Projection

- glOrtho2D(left, right, bottom, top)

- In world coordinates!



Screen coordinates



- Bottom left corner is origin

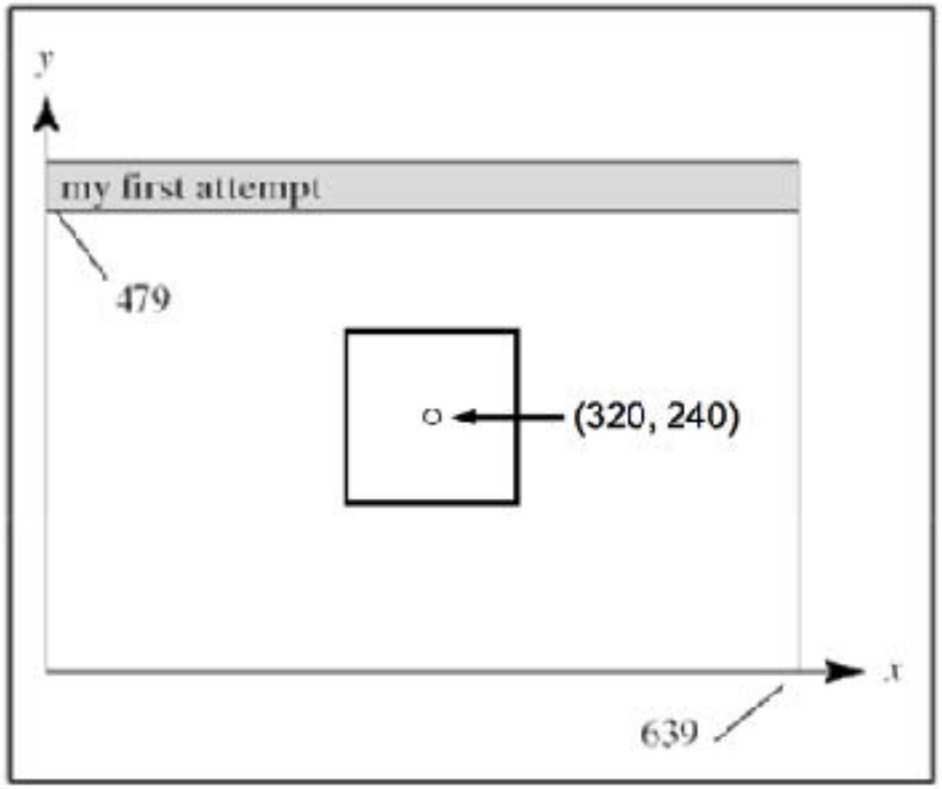
- **`gluOrtho2D()`** sets the units of the screen coordinate system
  - **`gluOrtho2D(0, w, 0, h)`** means the coordinates are in units of pixels
  - **`gluOrtho2D(0, 1, 0, 1)`** means the coordinates are in units of “fractions of window size” (regardless of actual window size)

y
my first attempt
479
639
x

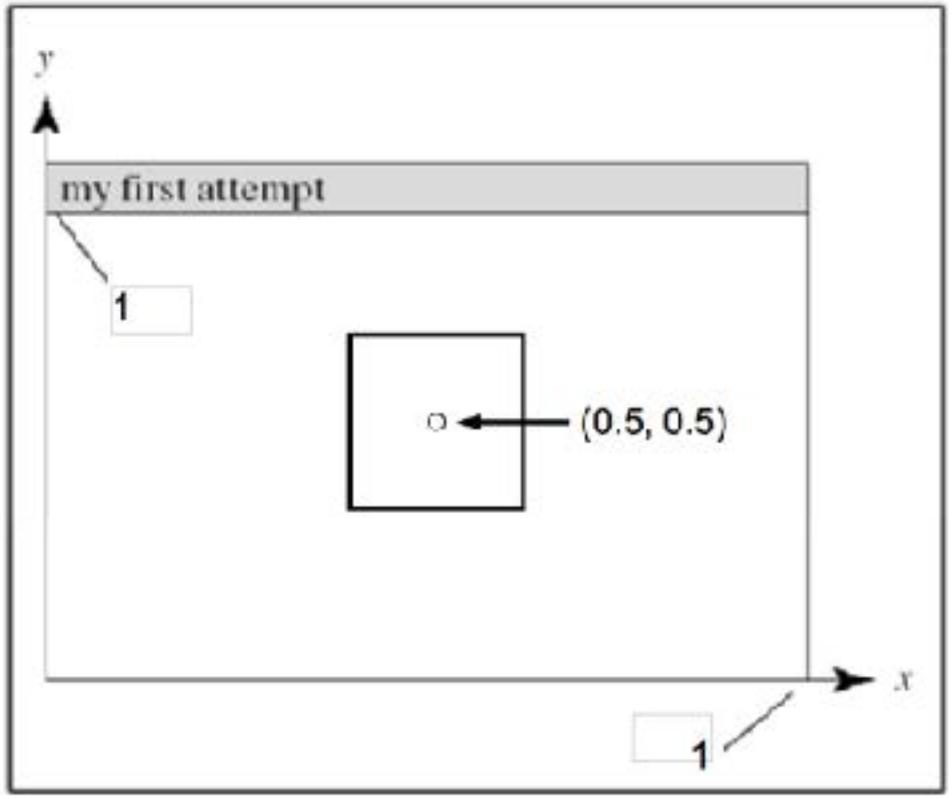
from Hill

Screen coordinates

```
gluOrtho2D(0, 640, 0, 480)
```

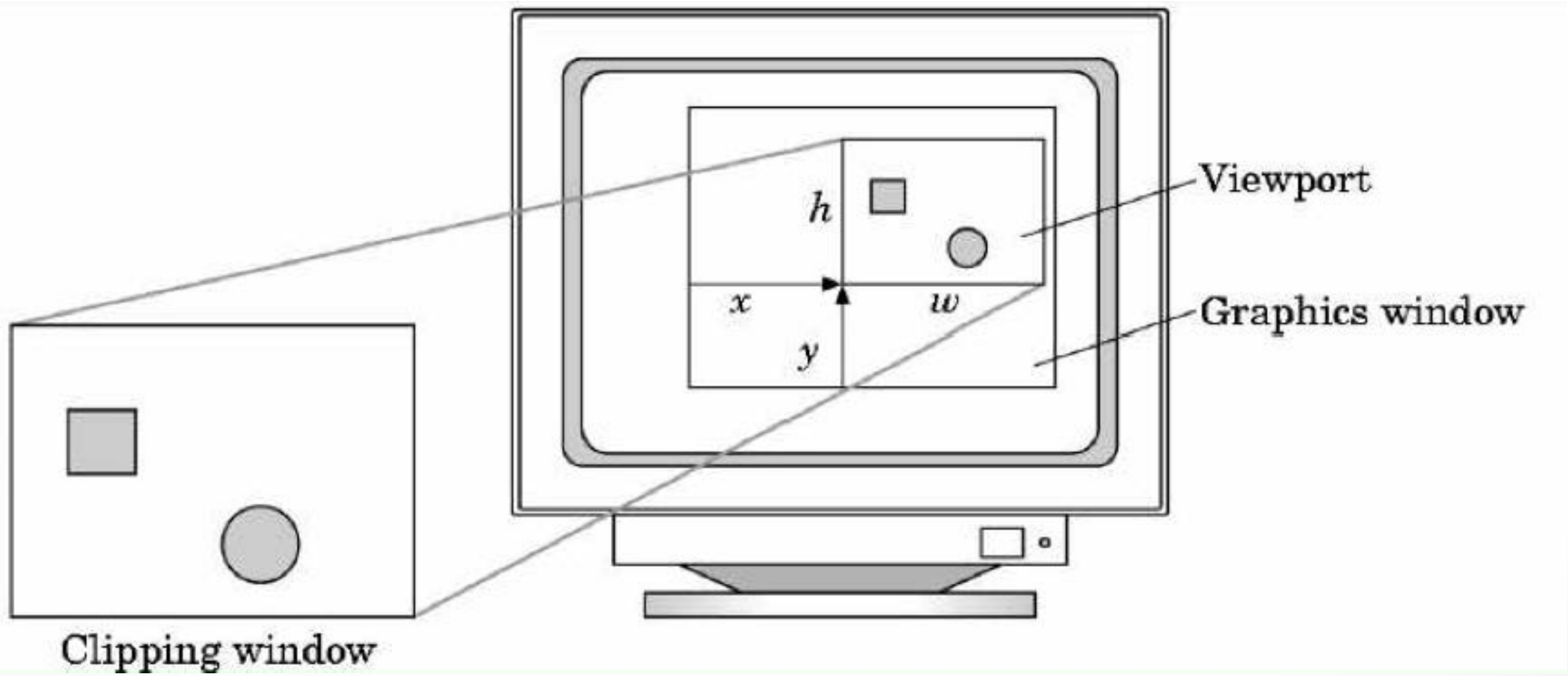


```
gluOrtho2D(0, 1, 0, 1)
```



Viewport

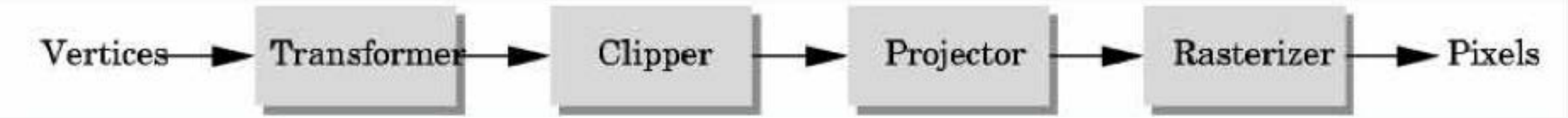
- Determines clipping in window coordinates
- glViewPort(x,y,w,h)



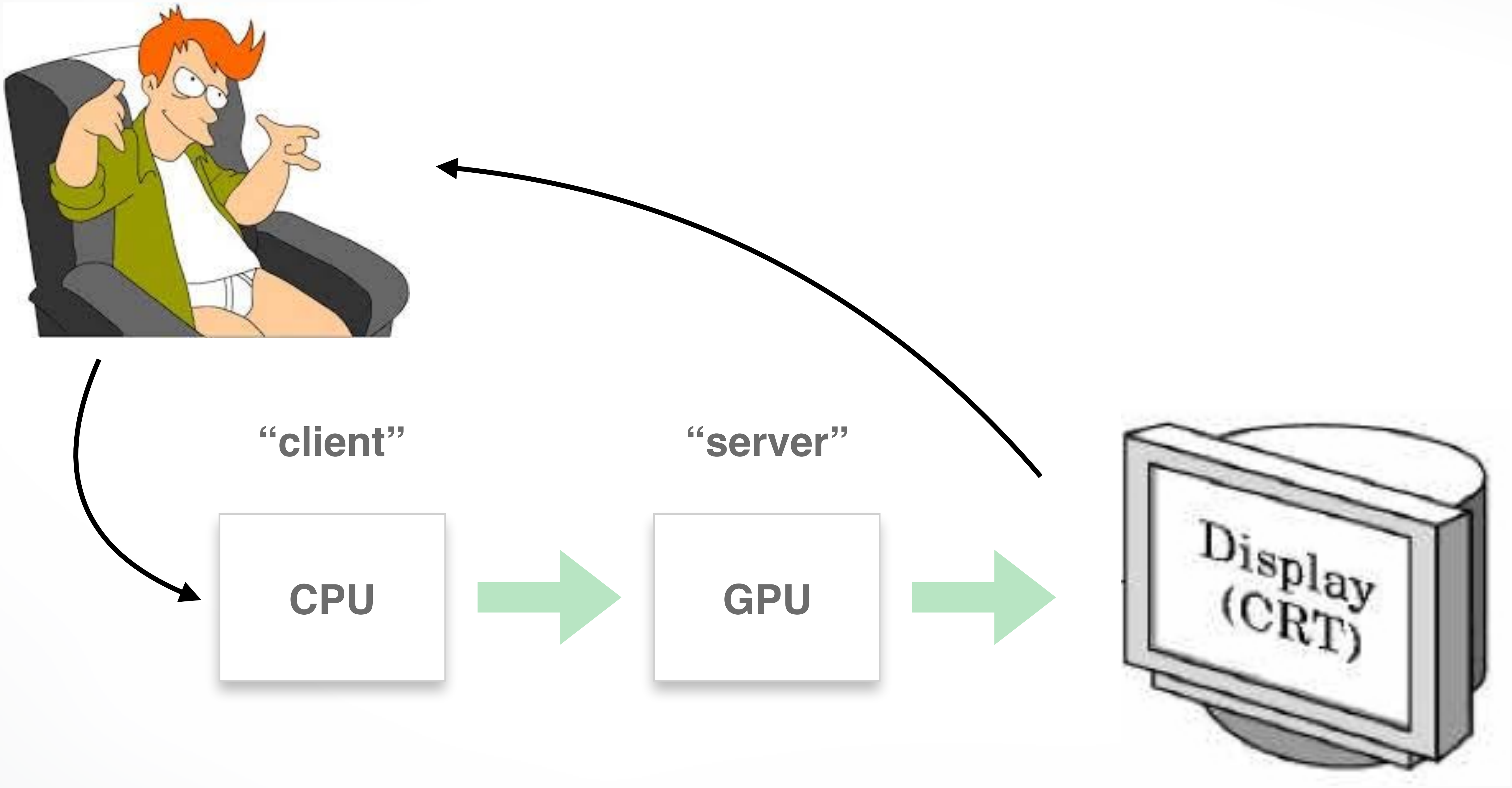
Let’s code a triangle!

Summary

- **A Graphics Pipeline**
- The OpenGL **API**
- **Primitives**: vertices, lines, polygons
- **Attributes**: color
- Example: drawing a **shaded triangle**



Next Time: Input & Interaction



“client”

CPU

“server”

GPU

http://cs420.hao-li.com

Thanks!