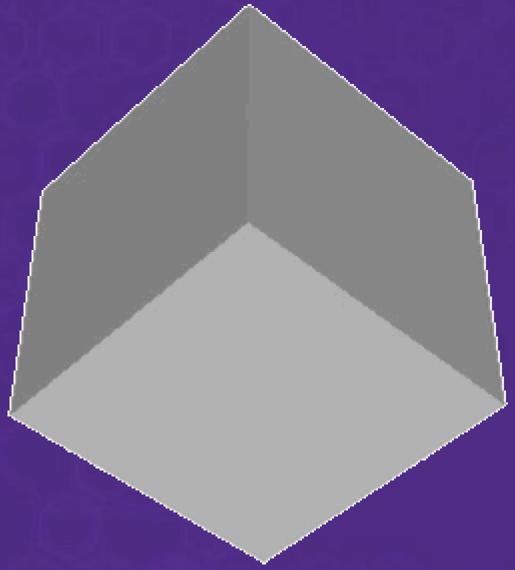
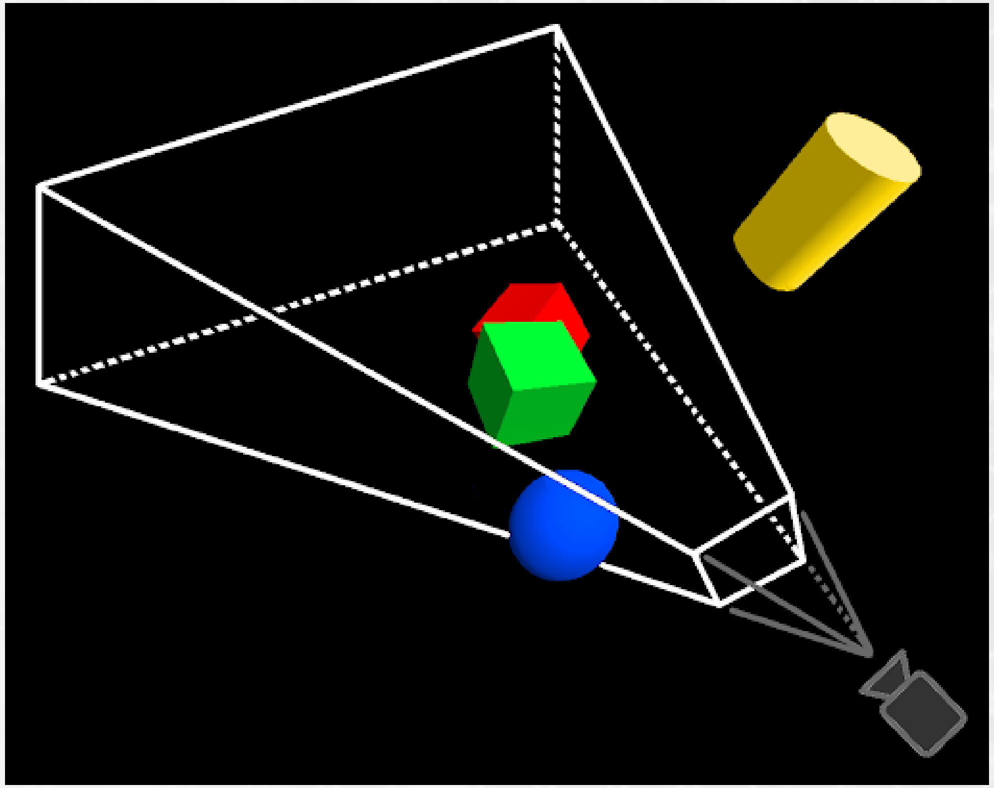
Developing Graphics Frameworks with Java and OpenGL

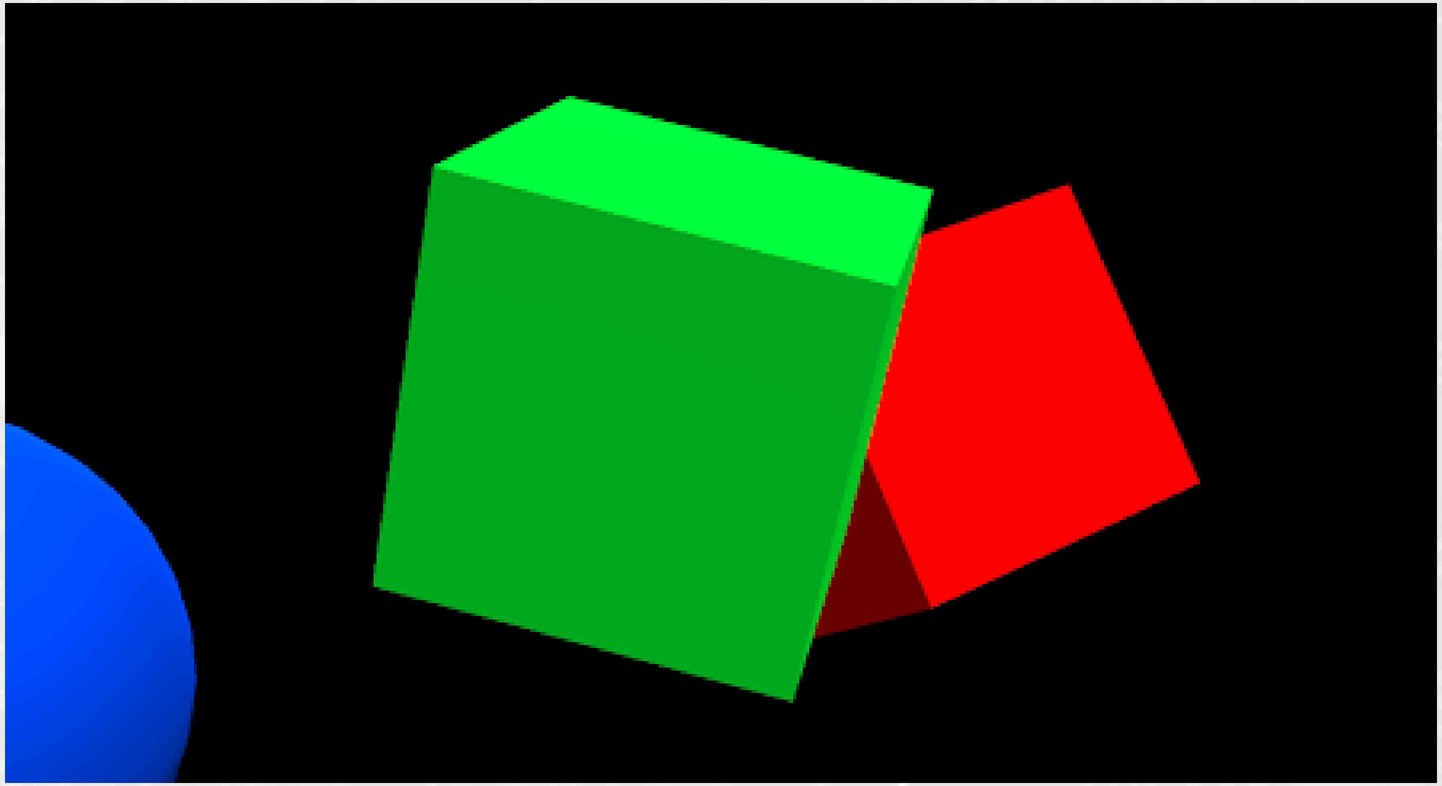


Part 01: Core Concepts

Rendering

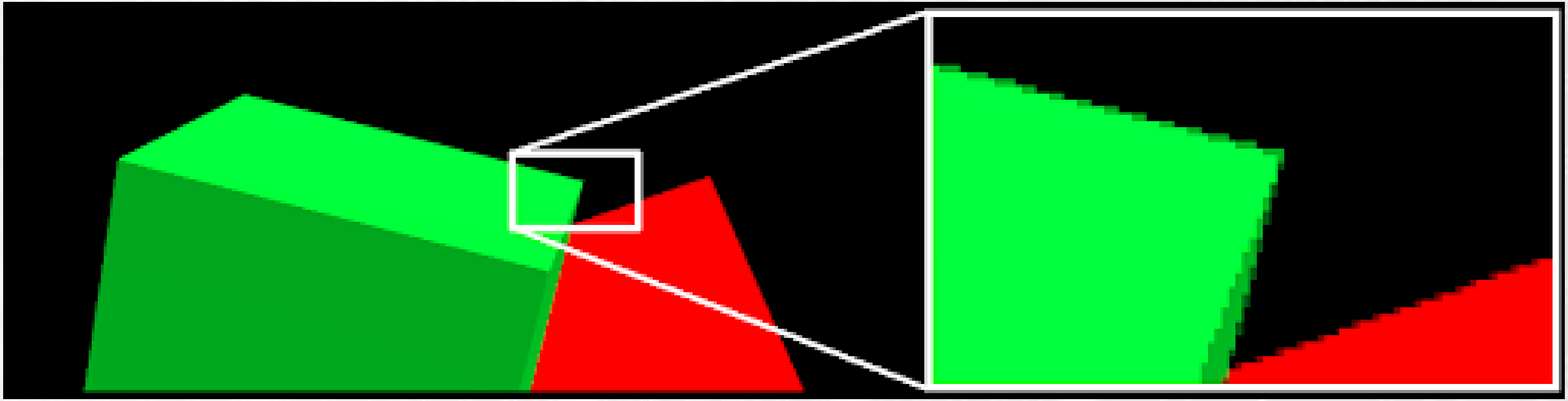
- *Rendering*: generating a two-dimensional image of a three-dimensional scene





Pixels

- *Pixels*: picture elements
- *Raster*: an array of pixels, displayed on a screen



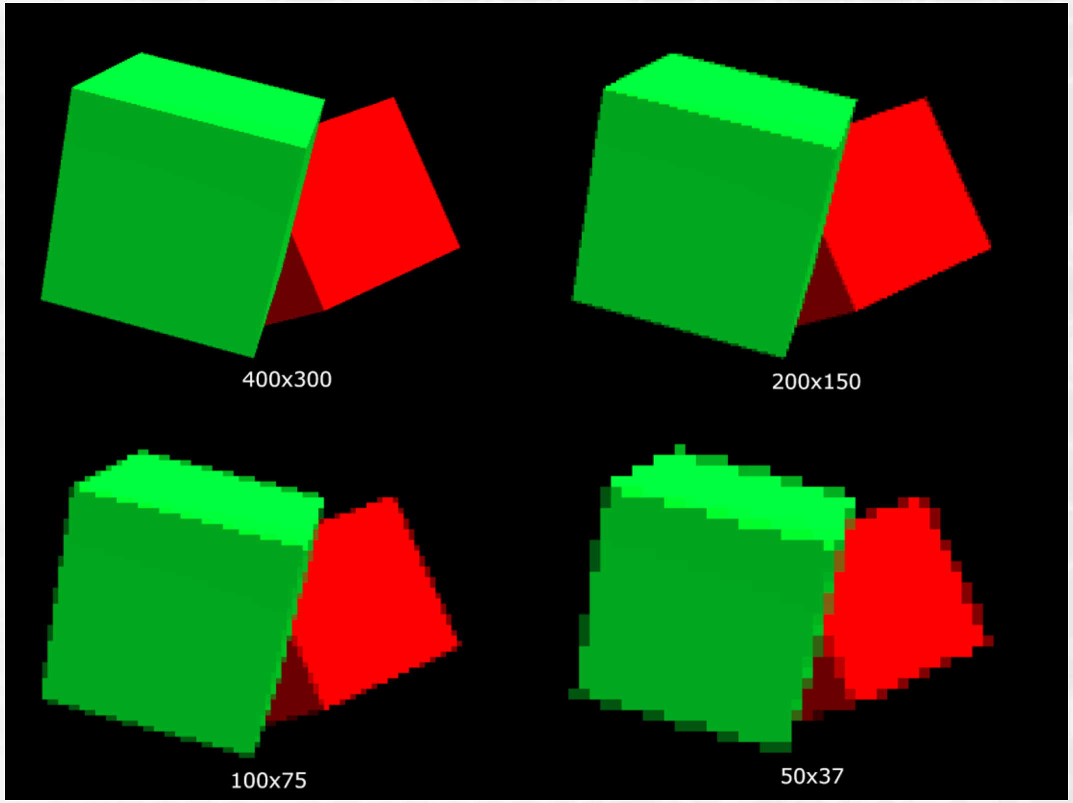
RGB Values

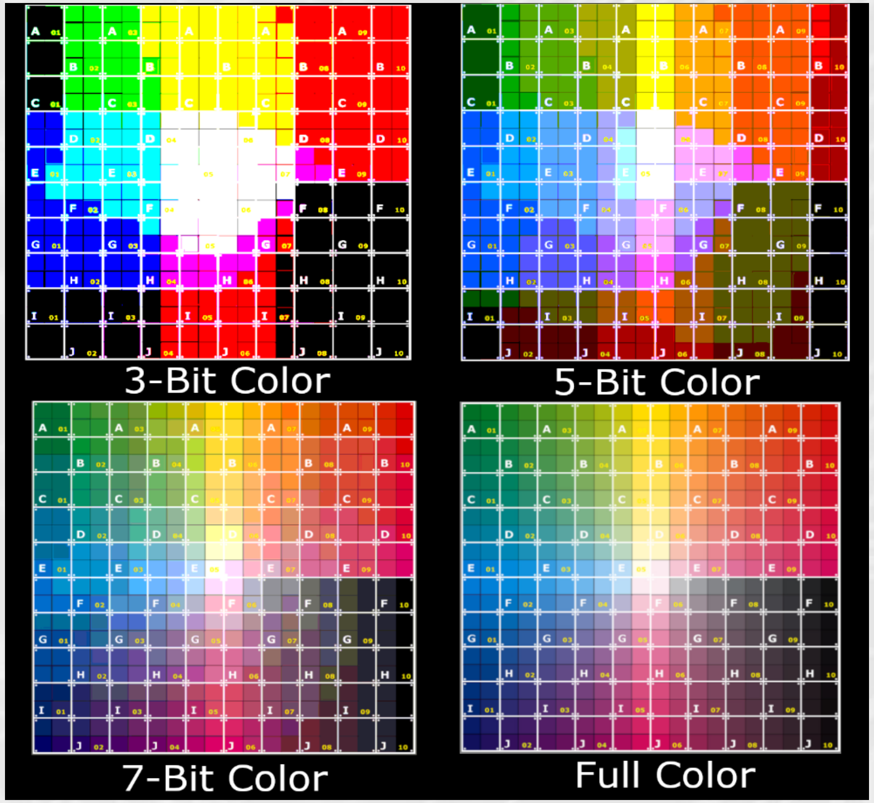
	R	G	B
red	1	0	0
orange	1	0.5	0
yellow	1	1	0
green	0	1	0
blue	0	0	1
violet	0.5	0	1

	R	G	B
black	0	0	0
white	1	1	1
gray	0.5	0.5	0.5
brown	0.5	0.2	0
pink	1	0.5	0.5
cyan	0	1	1

Graphics Quality

- *Resolution*: the number of pixels in the raster
- *Precision*: the number of bits used for each pixel





Buffers

- *Buffer* (or *data buffer*, or *buffer memory*):
  computer memory that serves as temporary storage
  for data while being moved
- *Frame buffer*: stores pixel-related data
  - *color buffer*: stores RGB values
  - *depth buffer*: stores distances
    from points on scene objects to the virtual camera
  - *stencil buffer*: stores values for advanced effects
    such as shadows, reflections, portals

Animation

- *Animation*: sequence of images displayed in quick enough succession that the viewer interprets the objects in the images to be continuously moving or changing in appearance

- *Frame*: each image displayed in an animation
- *Frame rate*: speed at which these images appear; measured in *frames per second* (FPS)

Introducing GPUs

- *central processing unit* (CPU); *graphics processing unit* (GPU)

- Sony GPU (Playstation, 1994) performed graphics-related computational tasks including managing a framebuffer, drawing polygons with textures, shading, transparency

- popularized by the Nvidia corporation (GeForce 256, 1999), a single-chip processor that also performed geometric transformations and lighting calculations

Introducing GPUs

- Nvidia: first company to produce a programmable GPU (GeForce 3, 2001; used in XBox)
- every geometric vertex and rendered pixel could be processed by a short program before the image was displayed
- features a highly parallel structure, more efficient than CPUs for rendering computer graphics

Shader Programs

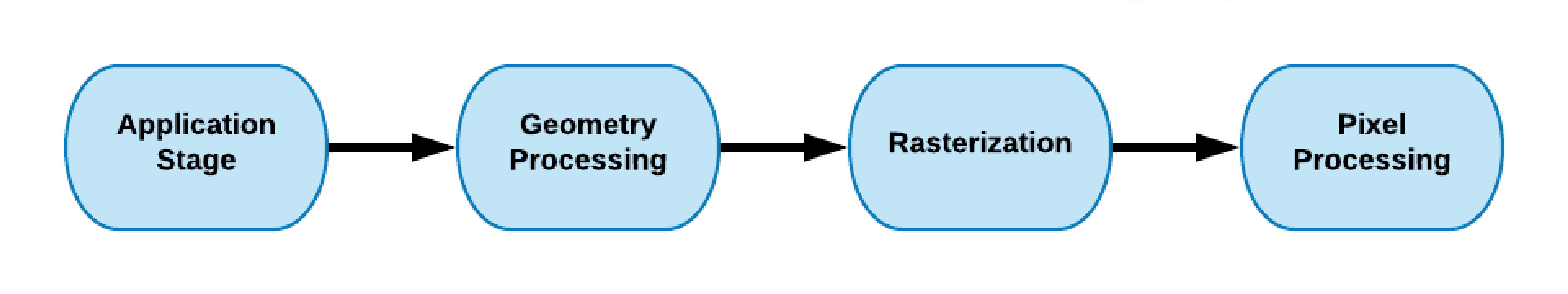
- *Shaders*: programs that are run by GPUs
- Many shader programming languages exist; each implements an *application programming interface* (API) that defines interaction with the GPU
  - The DirectX API and High-Level Shading Language (HLSL), used on Microsoft platforms, including the XBox game console
  - The Metal API and Metal Shading Language, used on modern Mac computers, iPhones, and iPads
  - The OpenGL (Open Graphics Library) API and OpenGL Shading Language, a cross-platform library

OpenGL

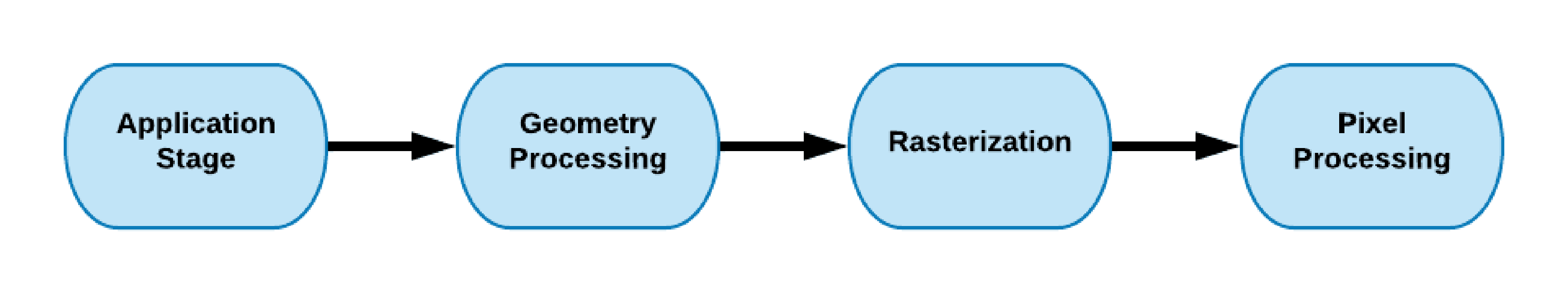
- Released by Silicon Graphics, Inc. (SGI) in 1992
- Managed by the Khronos Group since 2006
- Most widely adopted graphics API
- Visual results will be consistent on any supported OS
- Can be used with a variety of high-level languages using bindings: software libraries that bridge two programming languages
  - JOGL for Java, WebGL for JavaScript, PyOpenGL for Python

The Graphics Pipeline

- *Graphics pipeline*: an abstract model, describes a sequence of steps to render a three-dimensional scene
- Allows a computational task to be split into subtasks, each of which can be worked on in parallel



The Graphics Pipeline



- *Application Stage*: initializing window where graphics will be displayed; sending data to the GPU
- *Geometry Processing*: determining the position of each vertex of the shapes being rendered, implemented by a program called a *vertex shader*
- *Rasterization*: determining which pixels correspond to the geometric shapes to be rendered
- *Pixel Processing*: determining the color of each pixel in the rendered image, involving a program called a *fragment shader*

Application Stage

- involves processes that run on the CPU
- create a window where the rendered graphics will be displayed;
  initialize so that graphics are read from the GPU frame buffer
- main application contains a loop that re-renders scene repeatedly
- monitor hardware for user input events
- reading data required for rendering and sending it to the GPU
  - *vertex attributes*: describe the appearance of geometric shapes rendered
  - images that will be applied to surfaces
  - source code for the vertex shader and fragment shader programs

Application Stage

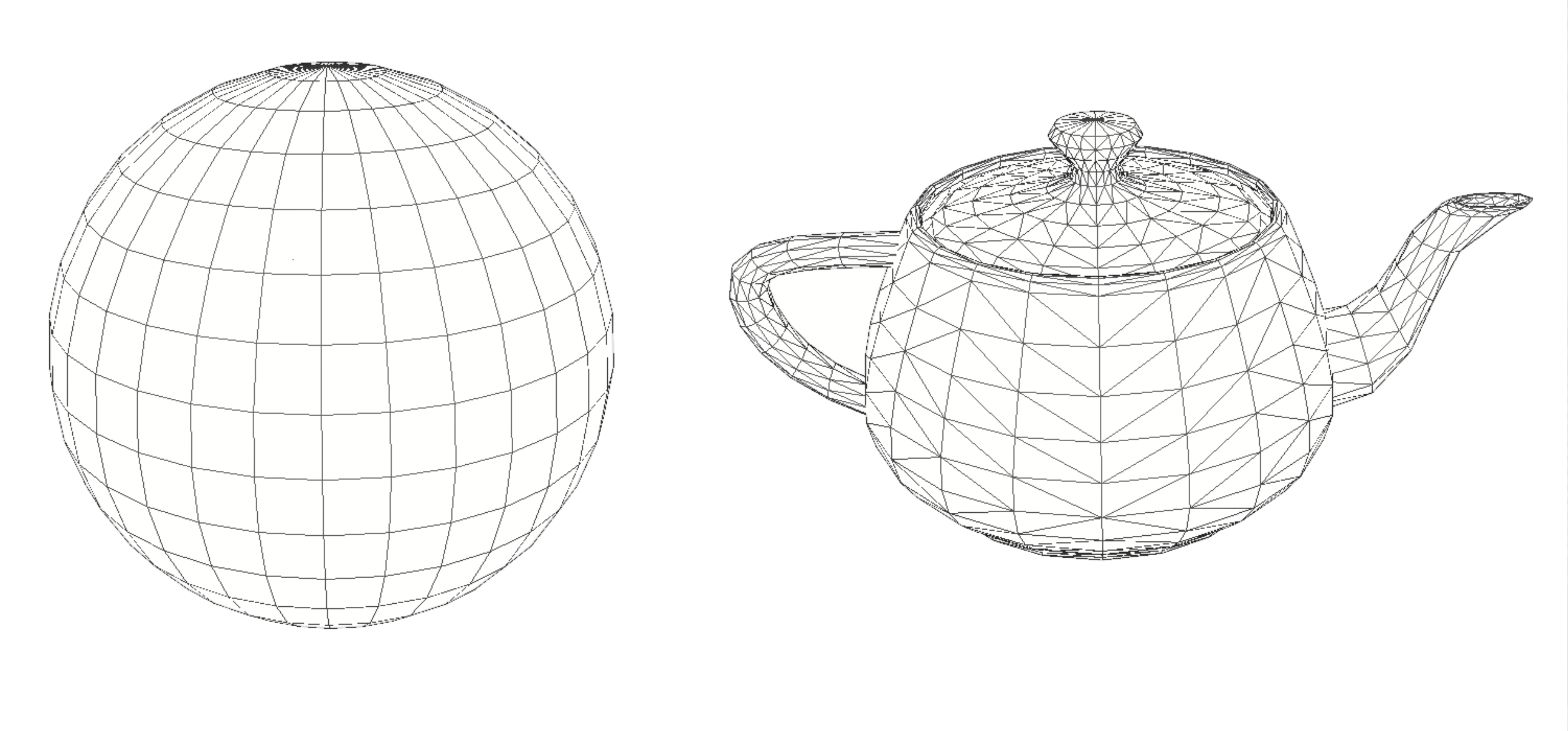
- vertex attribute data stored in *vertex buffer objects* (VBOs)
- images used as textures stored in *texture buffers*

Note: stored data is not assigned to any particular program variables; these associations are specified later.

- source code for vertex shader and fragment shader programs needs to be sent to the GPU, compiled, and loaded
- need to specify associations between attribute data stored in vertex buffer objects and variables in the vertex shader program
- sets of associations managed using *vertex array objects* (VAOs); can be activated and deactivated as needed

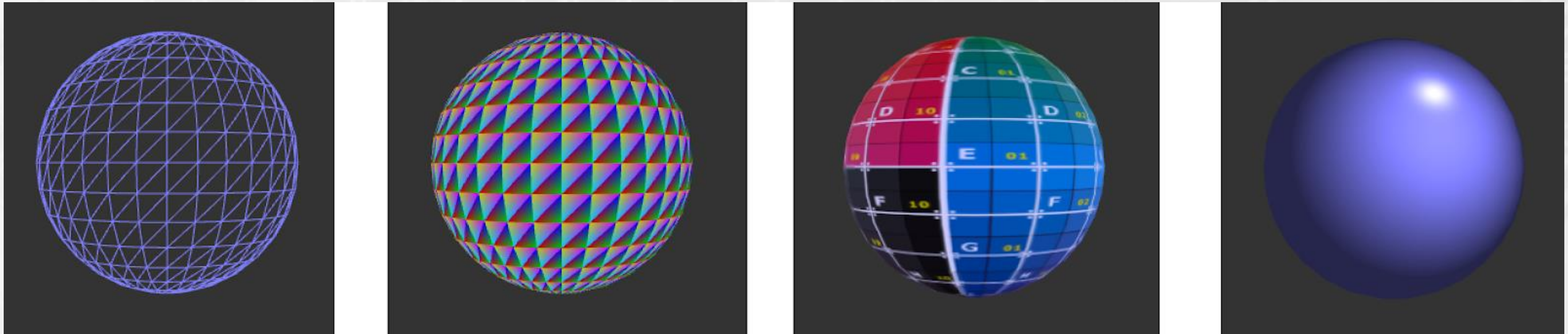
Geometry Processing

- *mesh*: a collections of points, grouped into lines or triangles



Geometry Processing

- attributes specific to rendering each point are grouped into a data structure called a *vertex*; must contain three-dimensional position; may also contain:
  - a *color* to be used when rendering the point
  - *texture coordinates* (or *UV coordinates*), that indicate which point in an image is mapped to a vertex
  - a *normal vector*, which indicates the direction perpendicular to a surface, typically used in lighting calculations

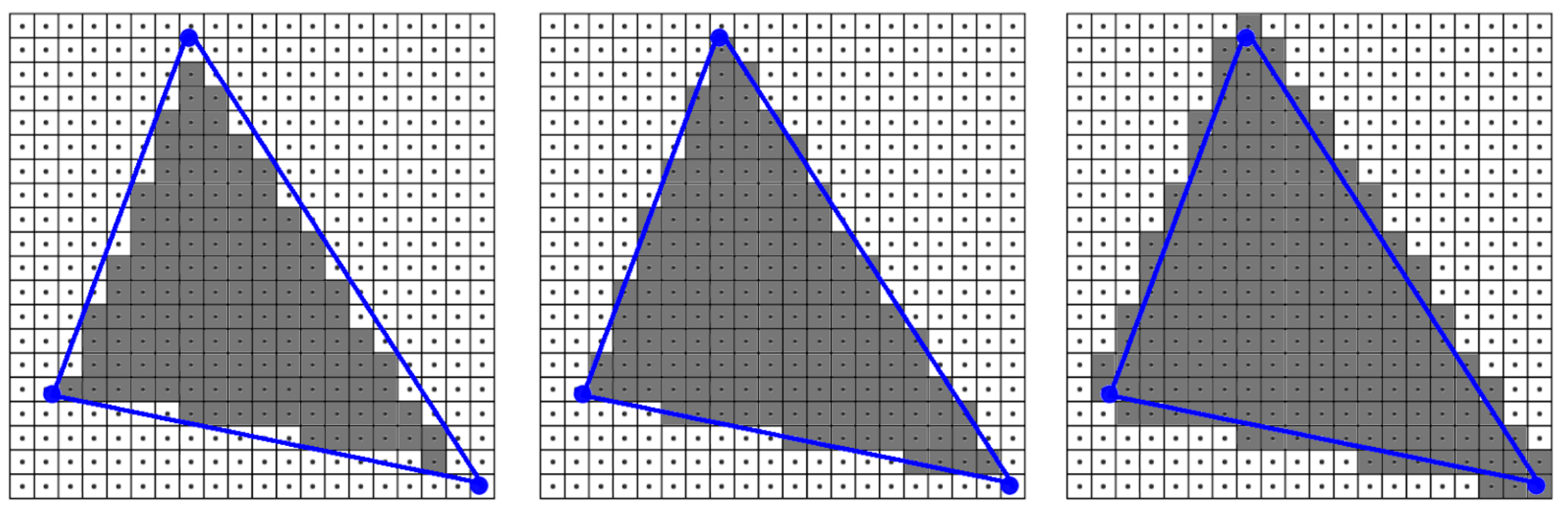


Geometry Processing

- A program called the *vertex shader* is applied to each vertex; each variable in the shader receives data from an associated buffer
- primary purpose of the vertex shader is to determine the final position of each point being rendered, calculated from transformations:
    - model transformation: translate, rotate, and scale the points defining a shape so it appears to have a location, orientation, and size within the world
    - view transformation: coordinates of each object converted to a frame of reference relative to the virtual camera
    - projection transformation: the set of visible points in the world must be aligned with the cubical region rendered by OpenGL

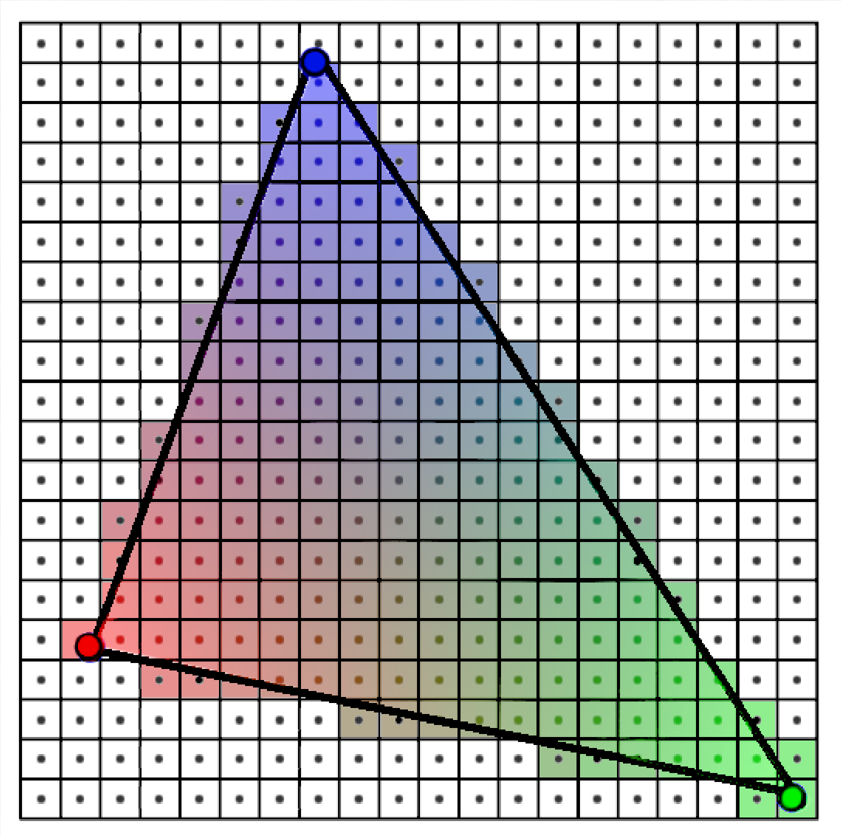
Rasterization

- geometric primitives: points, lines, or triangles
- primitive assembly: grouping points into primitives
- determine which pixels correspond to the interior of each primitive
  - the entire pixel is contained within the shape
  - the center point of the pixel is contained within the shape
  - any part of the pixel is contained within the shape



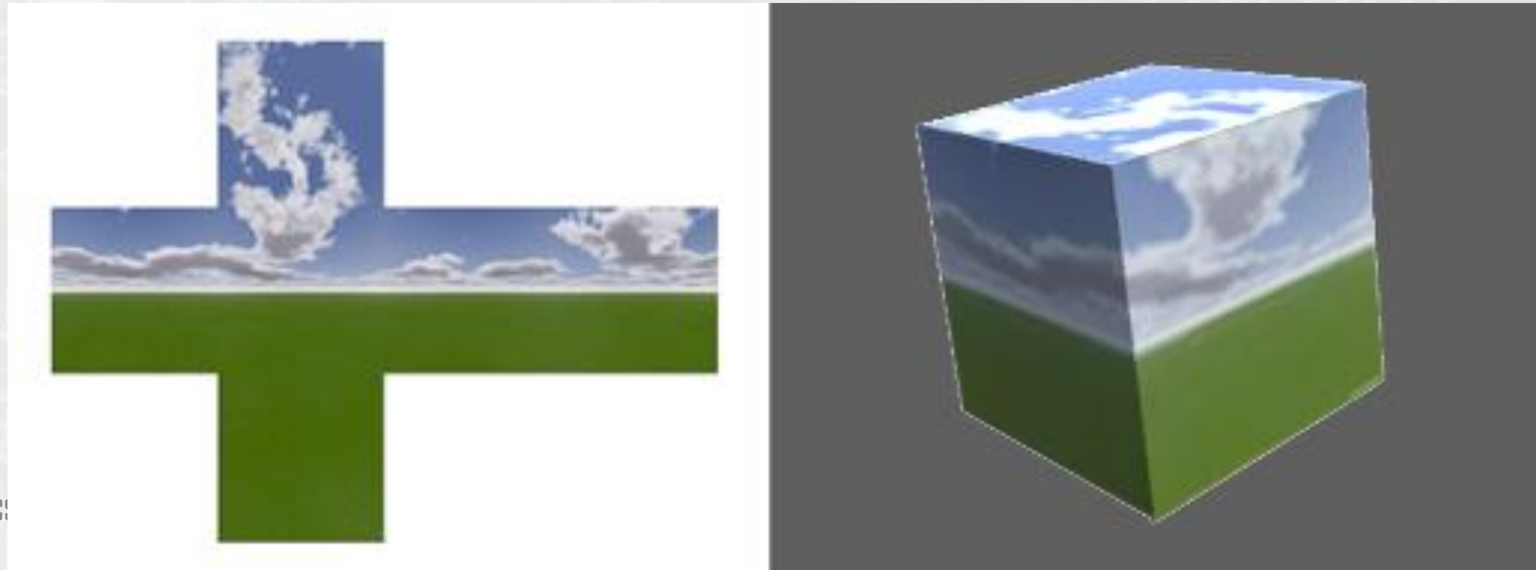
Rasterization

- fragment: collection of data used to determine pixel color
- fragment data includes the *raster position / pixel coordinates*
- in three-dimensional scenes, fragments store a *depth value*
- additional data for each vertex, such as color, passed along from the vertex shader to the fragment shader will be *interpolated*



Pixel Processing

- determine the final color of each pixel; store data in frame buffer
- a program called the *fragment shader* is applied to each of the fragments to calculate the final color; may involve data such as:
  - a base color applied to the entire shaper
  - colors stored in each fragment (interpolated from vertex colors)
  - textures, where colors are sampled from locations by texture coordinates
  - light sources, which may lighten or darken the color based on light position and normal vectors



Pixel Processing

- *depth* and *transparency* are automatically handled by the GPU using values stored in each fragment
  - depth values determine which parts of objects are blocked from view by other objects
  - alpha values determine how much of one pixel color should be blended with another pixel color