CS 432 – Interactive Computer Graphics

Lecture 1 – Part 1

Course Introduction

Imaging Systems

Graphics Pipeline

Credits

- Material based on Ed Angel and Dave Shreiner's Interactive Computer Graphics book/notes
- Additional material from Prof. David Breen, Prof. George Kamberov.

Course Description

- This is an entry-level course to **interactive computer graphics**.
  - Not necessarily game programing, but the low-level work that goes into the graphics used in game programming
  - Also related to rendering other computer graphic images (movies, art, etc..)
- It covers:
  - The general graphics pipeline
  - The OpenGL API
    - Modern shader-based
  - Basics of 2D and 3D graphics
  - Object picking
  - Shading, lighting, etc..
    - Realistic effects

Prior Knowledge

- Programming (ideally C++)
- Data Structures
  - Linked Lists
  - Arrays
  - Stacks/Queues
  - Matrices and Vectors
- Geometry
  - Where to place vertices in 3D
  - Transformations
- Linear Algebra
  - Matrix Multiplication
  - Cross and Dot Products

Things You'll Think About

- Performance Considerations
  - Try to keep stuff on GPU (retention mode)
  - Use GPU to do as much as possible (shaders)
  - Data types (unsigned, bytes, etc.. when possible)
  - Indices to vertex lists if useful
  - Memory management
- Code Planning/Structure/Organization
  - Lots going on, want to keep concise, separated, clear..
- Using/reading/researching APIs
- Some physics

Course Info

- Instructor:
  - Matt Burlick
    - Contact: mjburlick@drexel.edu, UC137
    - Office Hours: W 3:00pm – 5:00pm, R 3:30pm-5:30pm
- TA:
  - Reza Moradinezhad
    - Contact: rm976@drexel.edu
    - CLC, Wednesday 10:00am – 12:00pm
- Blackboard
- Piazza

Course Blackboard Page

- Syllabus
- Additional resources
- Slides
- Assignments
- Discussion Groups/ Forums (Piazza)
- Grades

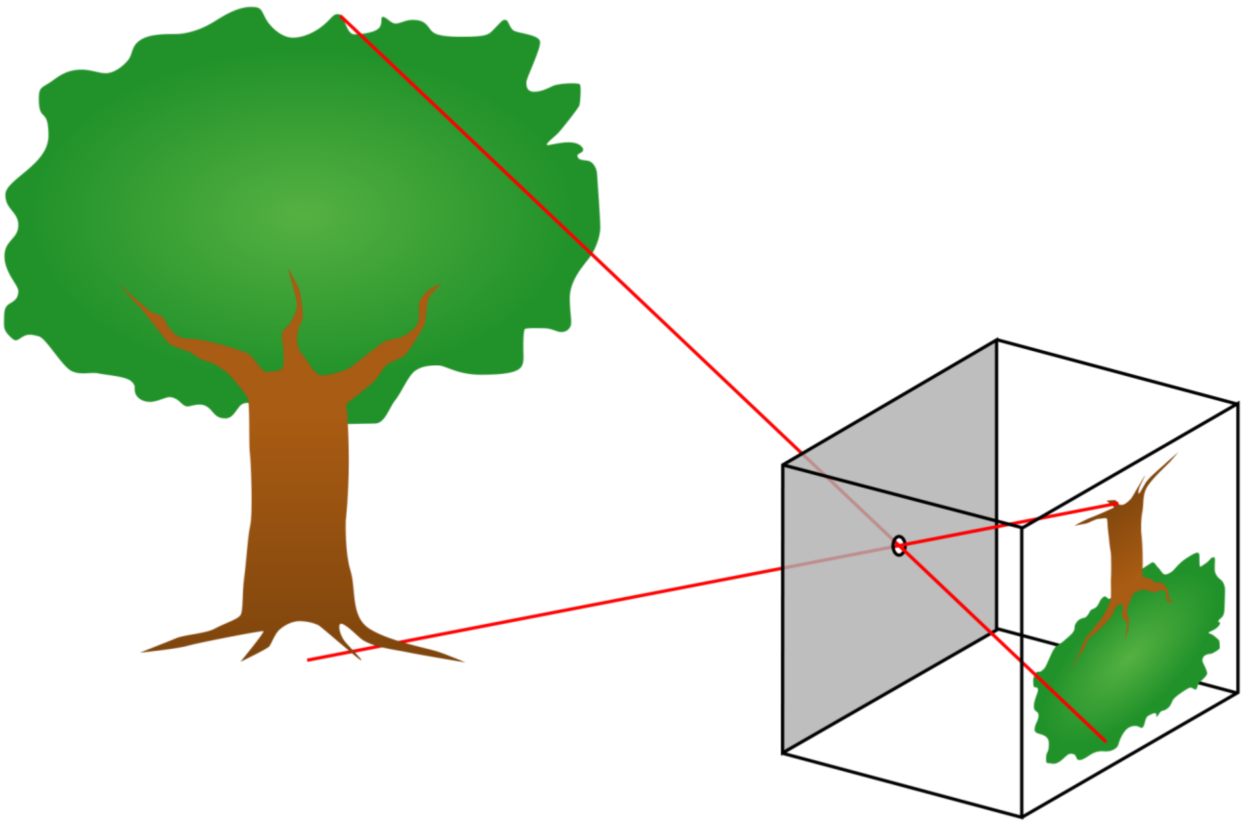
Policies

- Bound by Drexel Academic Integrity Code
  - We all are
  - You should discuss with each other *approaches* but not code!
    - Document any sources used for your projects
- If you use repository, keep it **private**.
- Assignments
  - You can use textbooks, notes
  - You cannot collaborate unless told otherwise
  - Late assignments will be penalized 1% per hour up to 48hrs (after which they will not be accepted).
  - (Almost) weekly assignments.
  - 70% of grade
- Final Project
  - Can collaborate with someone else if you like.
  - Worth 30% of grade

Course Resources

- Textbooks
  - Edward Angel and Dave Shreiner, Interactive Computer Graphics: a top down approach with shader-based OpenGL **(6th ed.),** Addison Wesley, ISBN-10: 0132545233, ISBN-13: 978-0132545235.
  - Dave Shreiner, *OpenGL Programming Guide* **(7th ed),** Addison Wesley, ISBN-10: 0321552628
- Software
  - We are programming in C++
  - Windows: Need freeGLUT and GLEW
    - Suggest using Microsoft Visual Studio
  - Mac: Need Xcode >=3.2
  - Linux: Need GL, GLUT, GLU, and GLEW
    - Must be able to compile and run on tux with X11 forwarding

Imaging Systems



Raster Graphics

- Modern graphics allow you to specify objects in 3D (including things like cameras, lighting sources, etc..).
- Then based on camera properties, a 2D image is created.
- This process of taking 3D geometrically specified objects (via lines, points, planes, etc..) and producing a 2D pixel image is called ***rasterization***
  - The resulting image is called the *raster image*

Modern Graphics

- Use GPU to do as many computations as possible
  - Really speeds things up!
- Can now do realistic stuff often in real-time
  - Texture mapping
  - Blending
- **Programmable pipelines**
  - Older code < OpenGL 3.0 uses a *fixed pipeline*. You **cannot** use this style of code.

Image Formation

- In computer graphics, we form 2D images using a process analogous to how images are formed by physical imaging systems
  - Cameras
  - Human visual system
- These systems take into account
  - Objects
  - Viewer
  - Light source(s)

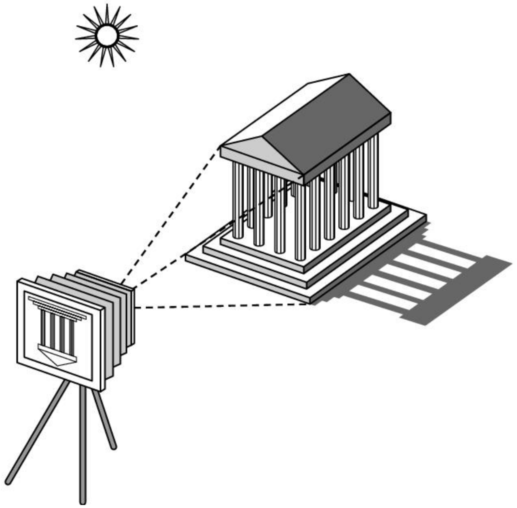
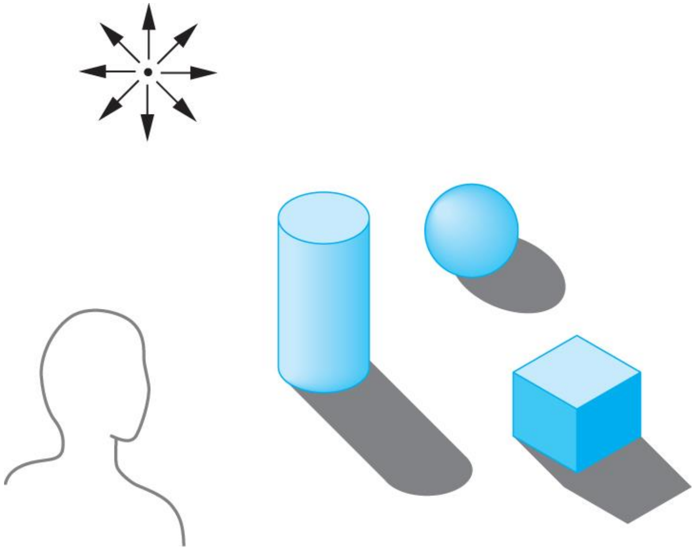


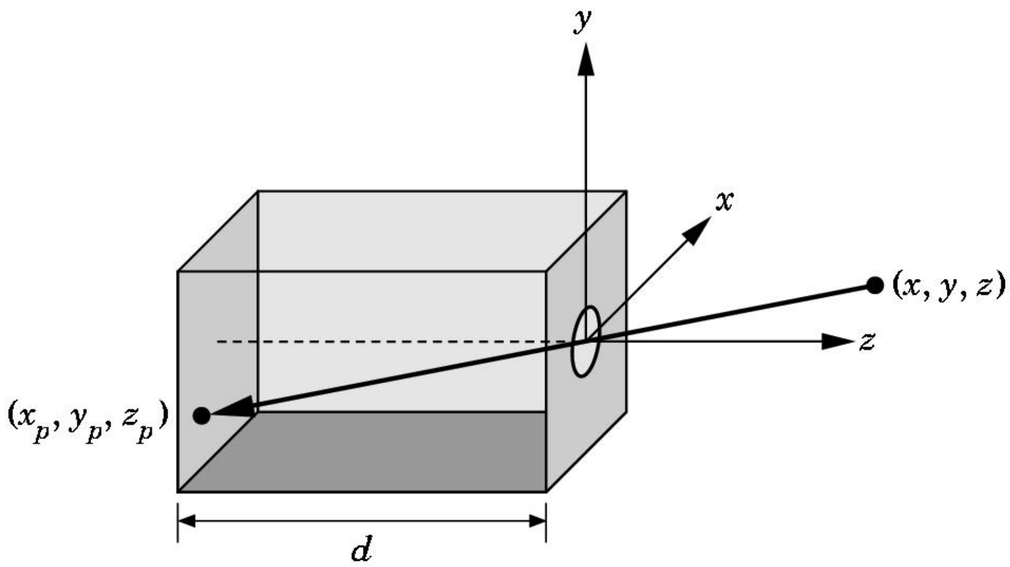
Image Formation

- Also can take into account how light interacts with materials.



Pinhole Camera

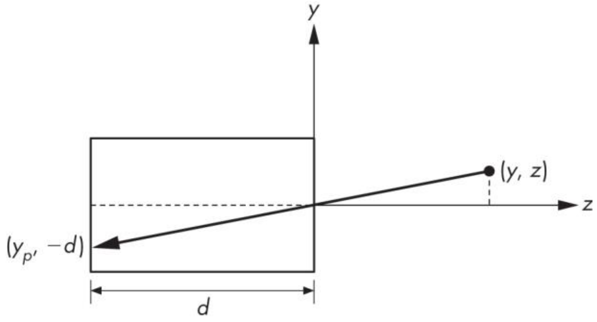
- To model the camera we usually us a simple *pinhole camera*.
- Use trigonometry (or the idea of similar triangles) to find projection of point at (x, y, z)
- Let d be the *focal distance* of our camera, then:
    - $x_p = \frac{x}{\frac{z}{d}}$
    - $y_p = \frac{y}{\frac{z}{d}}$
    - $z_p = d$

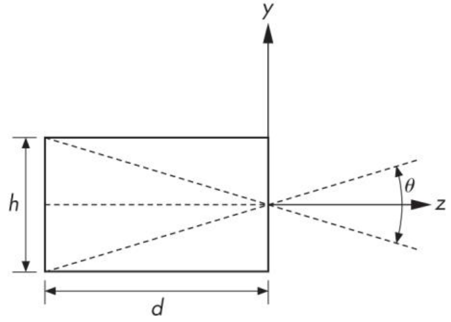


Pinhole Camera

- The point $(x_p, y_p, -d)$ is called the *projection* of point (x, y, z).
- We can also calculate the *field of view* of our camera.
  - Given the height of the camera, *h*:

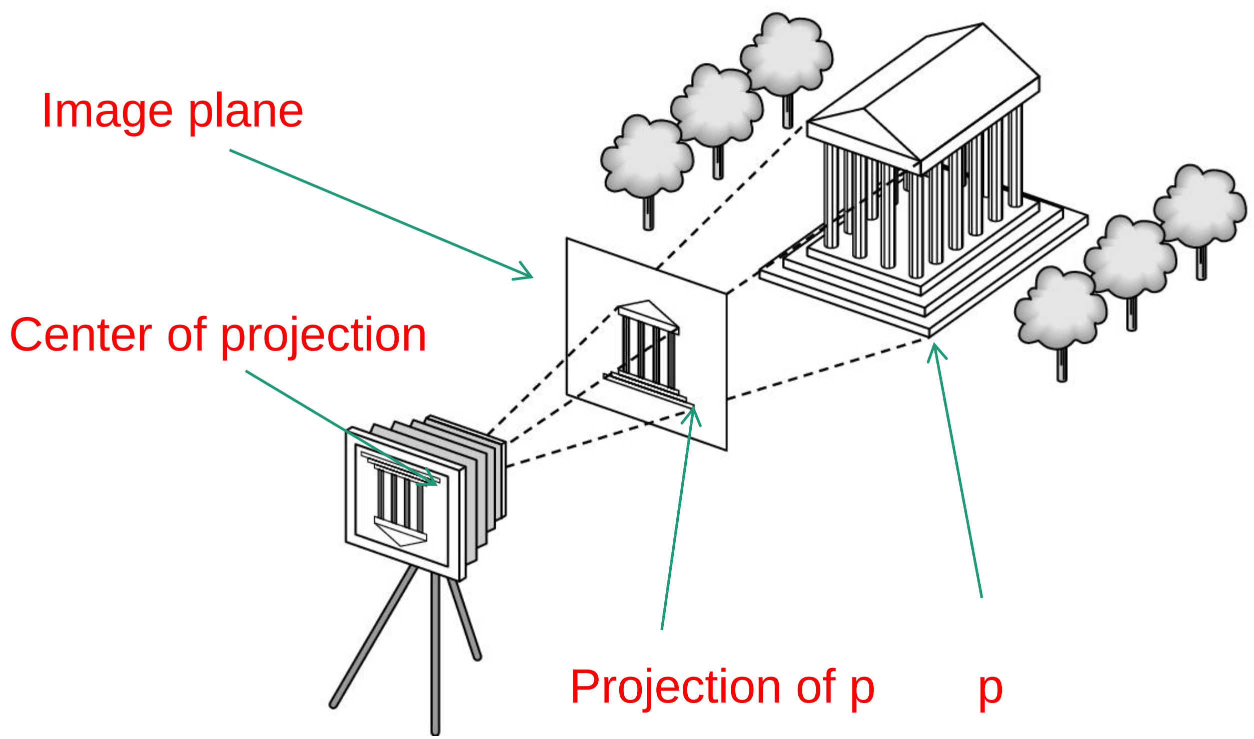
$$\theta = 2\tan^{-1}\left(\frac{h}{2d}\right)$$





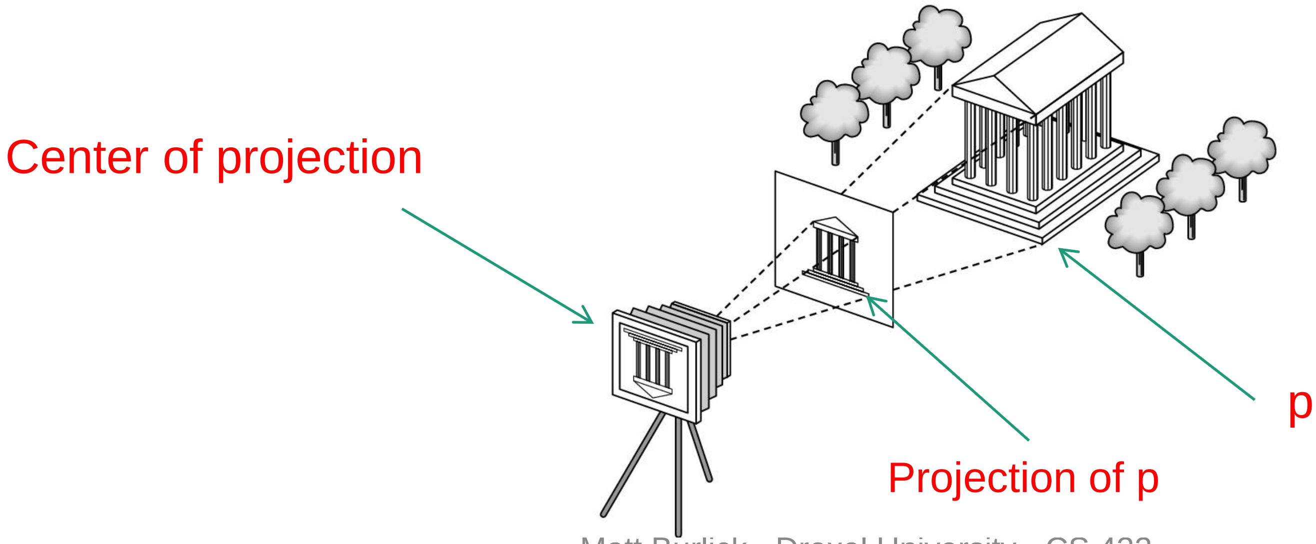
Synthetic Camera Model

- To aid in conceptualizing the image formation, we use the *synthetic camera model*

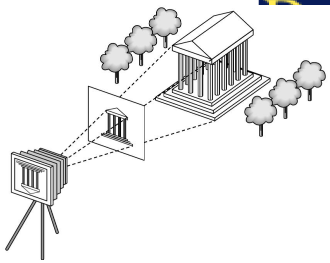


Synthetic-Camera Model

- We find the image of a point on the object on the virtual image plane by drawing a line, called a *projector* from the point to the center of the lens
  - Called the *center of projection (COP)*
  - Note: All projectors come from the COP



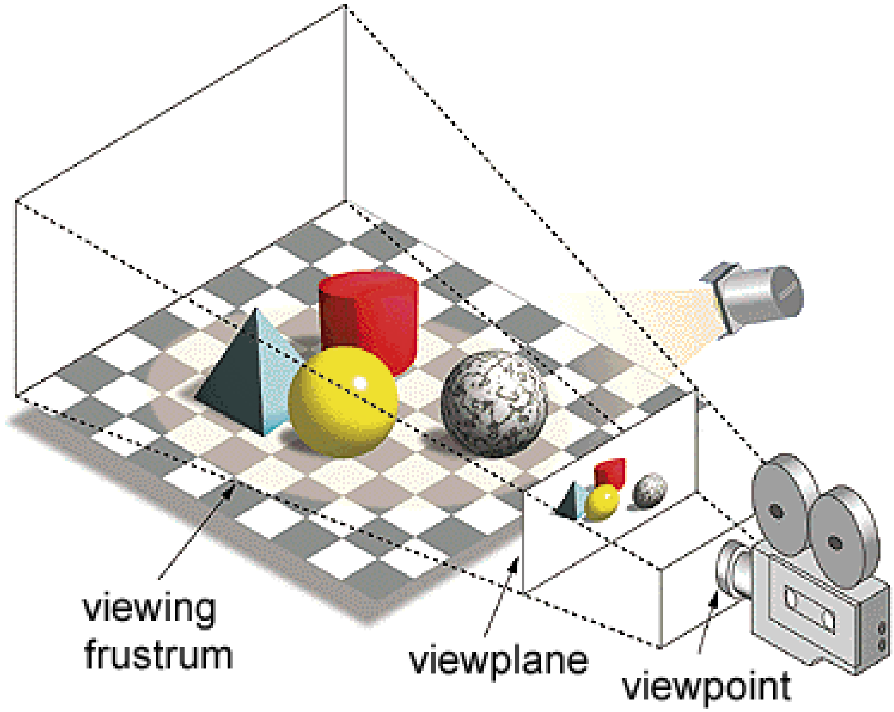
Synthetic-Camera Model



- The image plane is finite
  - Not all object will be projected onto it
  - This is limited by the *field/angle of view*
- So we must *clip* out stuff that is not in the field-of-view
  - We do this by placing a *clipping volume* around the camera.
- We can then determine which objects appear given:
  - The location of the COP
  - The location and orientation of the projection plane
  - The size of the clipping volume
  - The location of the objects

Graphics Pipeline

From Computer Desktop Encyclopedia
Reprinted with permission.
© 1998 Intergraph Computer Systems

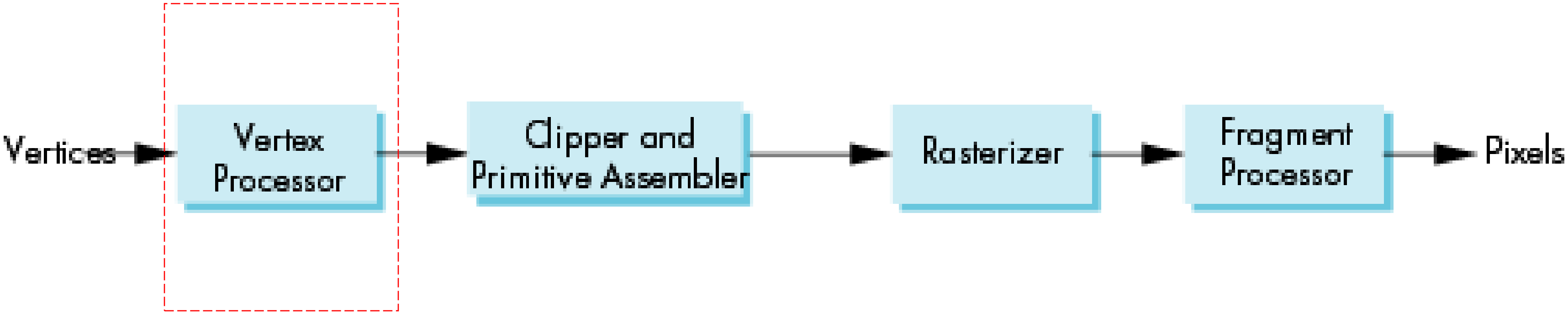


The Graphics Pipeline

- The Graphics Pipeline provides a way to go from vertices used to specify shapes/primitives (maybe millions of them!) to pixels in our raster image
- Each part can be done quickly using hardware

Vertex Processing

- Here, each vertex is processed *independently*
- Two major functions:
  - Coordinate transformations
  - Color computation



Vertex Processing: Coordinate Transformations

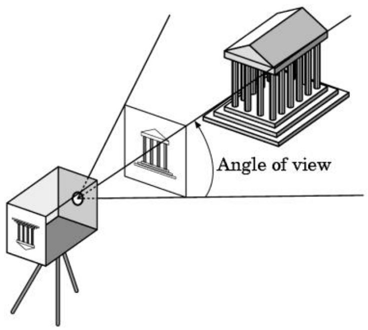
- There are many types of coordinate transformations
  - Remember going from (x, y, z) to $(x_p, y_p, -d)$? I.e. world coordinate to image plane coordinates?
  - We also need to consider
    - Going from object coordinates to world coordinates
    - Going from image plane coordinate to screen coordinates
- We represent each transformation using a *matrix* and each transformation is done via a matrix multiplication

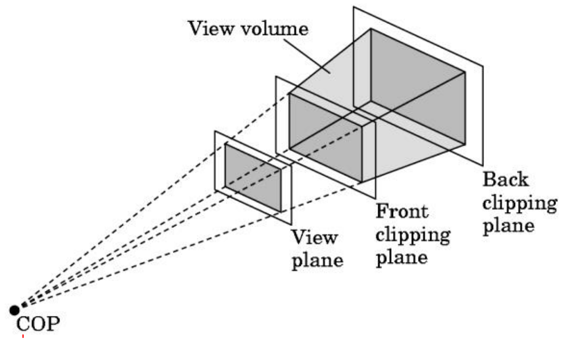
Vertex Processing: Color Computation

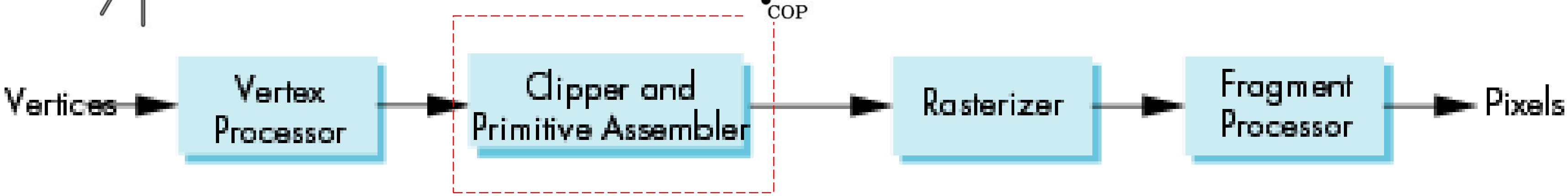
- Several ways to assign a color to each vertex:
  - Let the program specify the color
  - Compute from lighting model
    - Incorporates surface properties of the object and the light sources in the scene.

Clipping and Primitive Assembly

- As we mentioned in the Synthetic-Camera model, our image plane cannot see the whole world.
- So we provide a *clipping volume*, often a pyramid in front of the lens.
  - Object in this volume are projected into the image





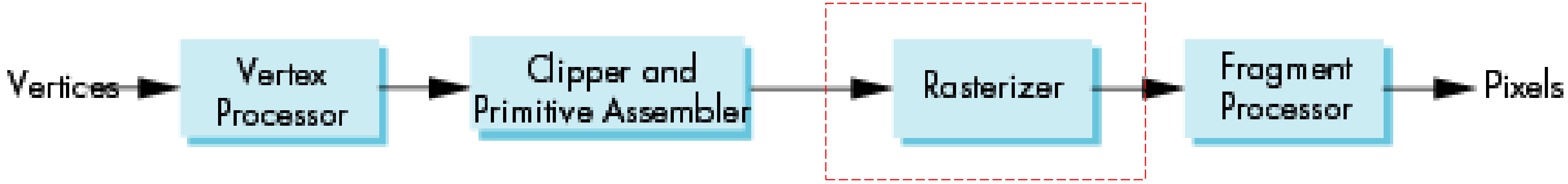


Clipping and Primitive Assembly

- Clipping is done on a primitive-to-primitive bases rather than vertex-by-vertex
  - We're considering if "objects" are in view, not vertices
- So in this stage we take vertices and make primitives
  - Line segments
  - Polygons
- The output of this stage is the set of visible primitives

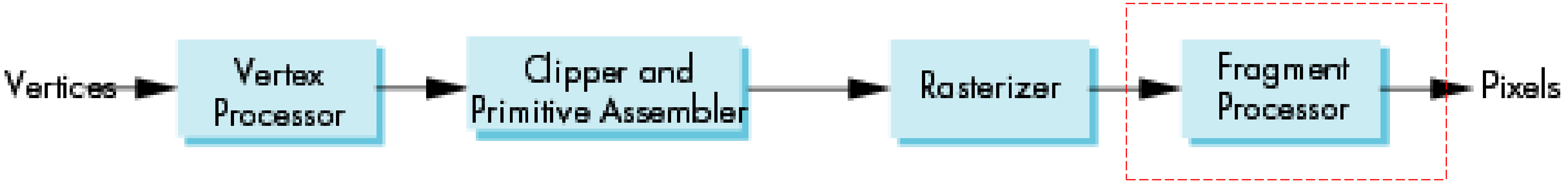
Rasterizer

- Primitives that come out of the primitive assembler are still represented in terms of vertices and not pixels.
- The rasterizer must determine which pixels belong to which primitive
  - Vertex attributes are **interpolated** over primitives by the rasterizer
- Rasterizer produces a set of *fragments* for each primitive
- Fragments are "potential pixels"
  - Have a location in frame buffer
  - Color and depth attributes



Fragment Processing

- Fragments are processed to determine the color of the corresponding pixel in the frame buffer
- Colors can be determined by texture mapping or interpolation of vertex colors
- Fragments may be blocked by other fragments closer to the camera
  - Hidden-surface removal



Programmable Pipelines

- Using just a standard CPU it is difficult, if not impossible, to do many graphics thing is real time
- As graphics cards and their GPUs have become more popular and powerful, much computation has been offloaded to them
- These GPUs have built-in pipelines
  - For years this pipeline was fixed, couldn't change in the GPU what's happening to the vertices and/or fragments
  - But now we're allowed to program both the vertex and fragment processors!

Programmable Pipelines

- Vertex programs can alter the location or color of each vertex as it flows through the pipeline
  - So we can do complex light-material models and projection effects
- Fragment programs allow us to use textures and implement other parts of the pipeline, like lighting, on a per-fragment bases rather than per-vertex

Additional (Optional) Stages

- Tessellation Shading Stage
  - Comes after the vertex shader
  - Processes *patches* (order list) of vertices
  - Can add/remove vertices on the fly from patches as well as move them
    - Typically involves subdividing the patch, computing new vertex attributes for new vertices.
- Geometric Shading Stage
  - Comes right before the primitive assembly
  - Input is collection of vertices
  - Operates on geometric primitives (vertices of a triangle, etc..)
  - May spawn new geometry
  - May alter primitive type (from lines to triangles)
  - May discard geometry