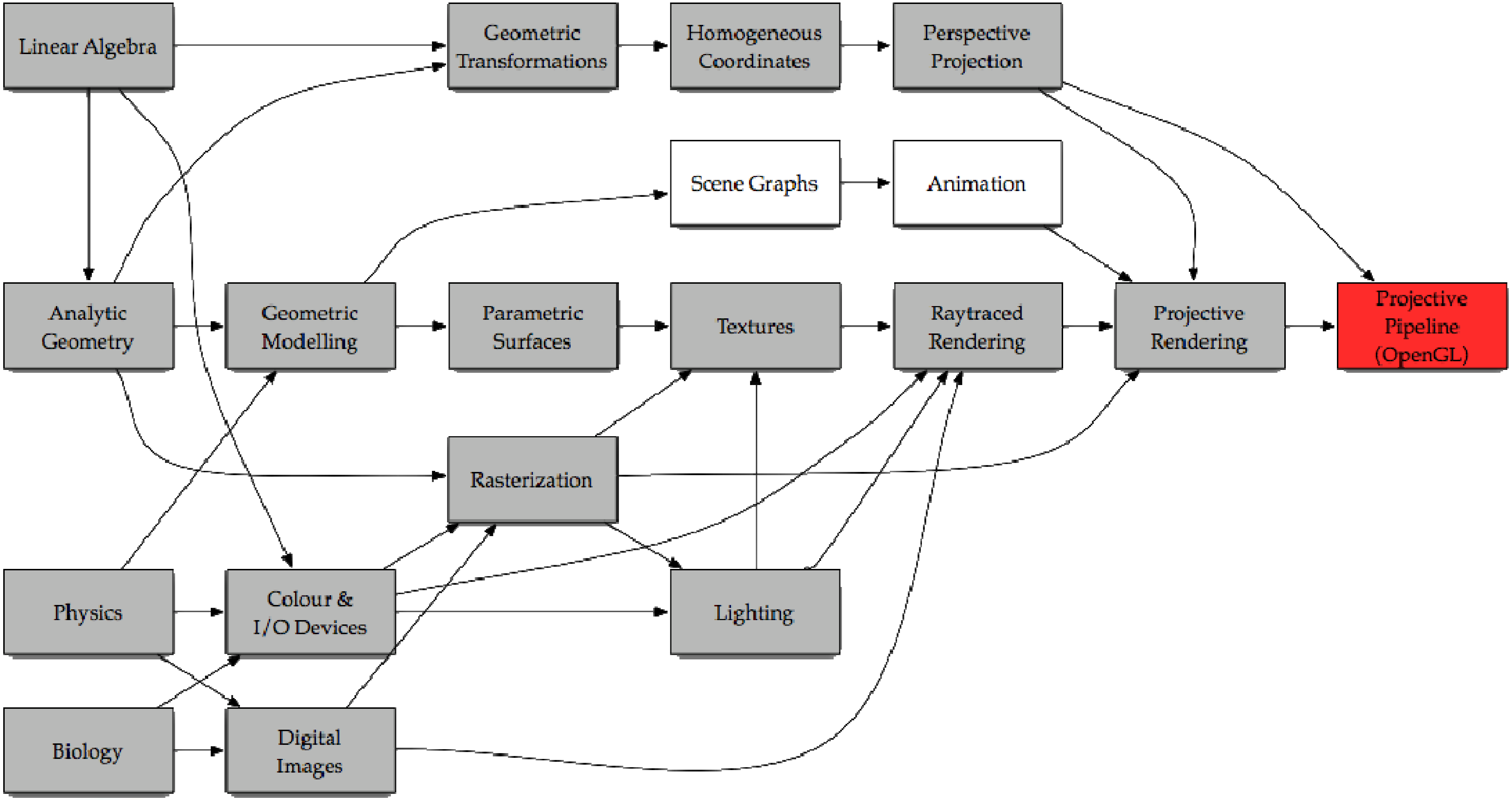
Coordinate Systems and Projective Rendering

Where we Are

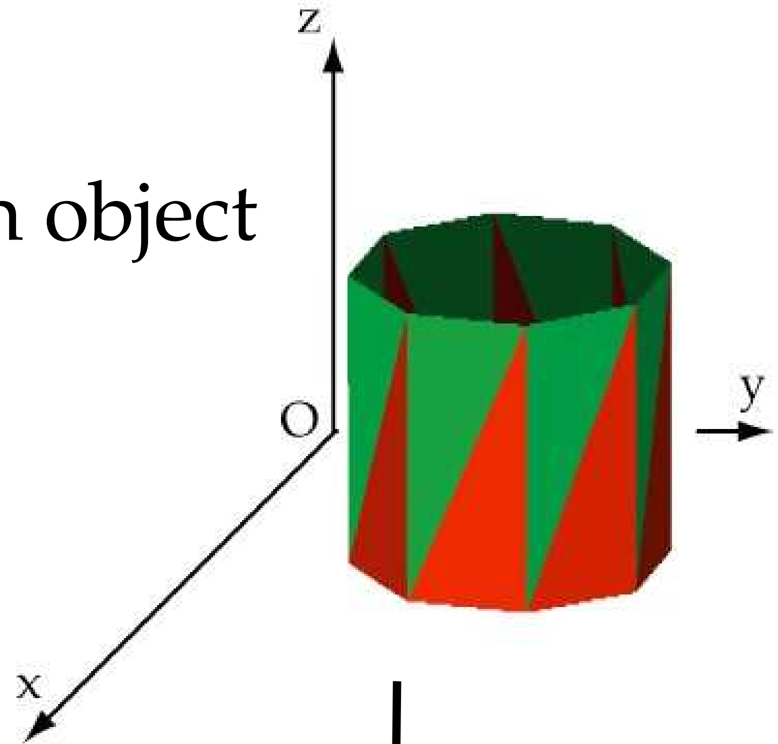

Linear Algebra
Geometric Transformations
Homogeneous Coordinates
Perspective Projection
Scene Graphs
Animation
Analytic Geometry
Geometric Modelling
Parametric Surfaces
Textures
Raytraced Rendering
Projective Rendering
Projective Pipeline (OpenGL)
Rasterization
Physics
Colour & I/O Devices
Lighting
Biology
Digital Images

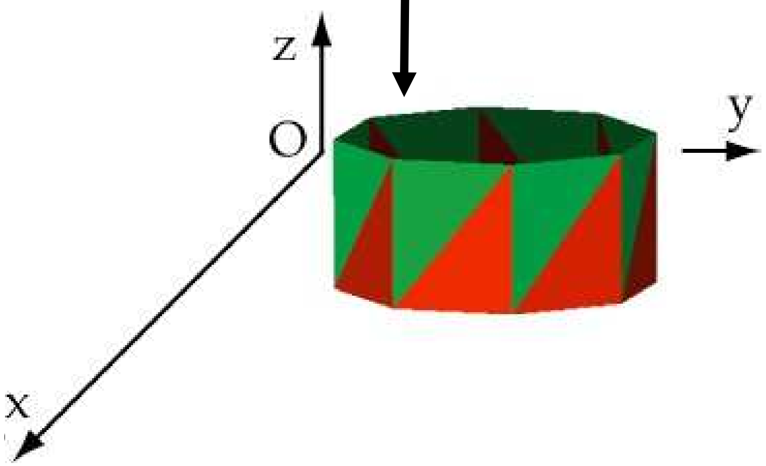
Coordinate Systems

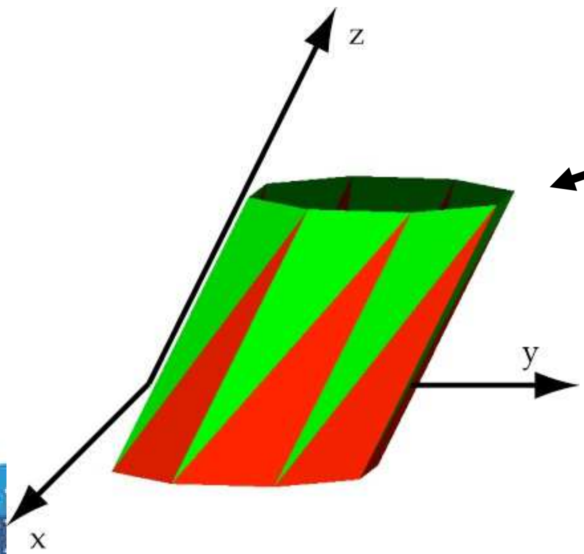
- *OCS* is the *Object Coordinate System*
- *WCS* is the *World Coordinate System*
- *VCS* is the *View Coordinate System*
- *CCS* is the *Clipping Coordinate System*
- *NDCS* is the *Normalized DCS*
- *DCS* is the *Device Coordinate System*

Object Coordinates

- Coordinates defining an object
    - e.g. the cylinder
    - travel with object







View Matrix

- Second most commonly changed matrix
- Converts from *WCS* to *VCS*
- *Or* moves *VCS* relative to *WCS*
- Gives position / orientation of *eye* in scene

World Coordinates

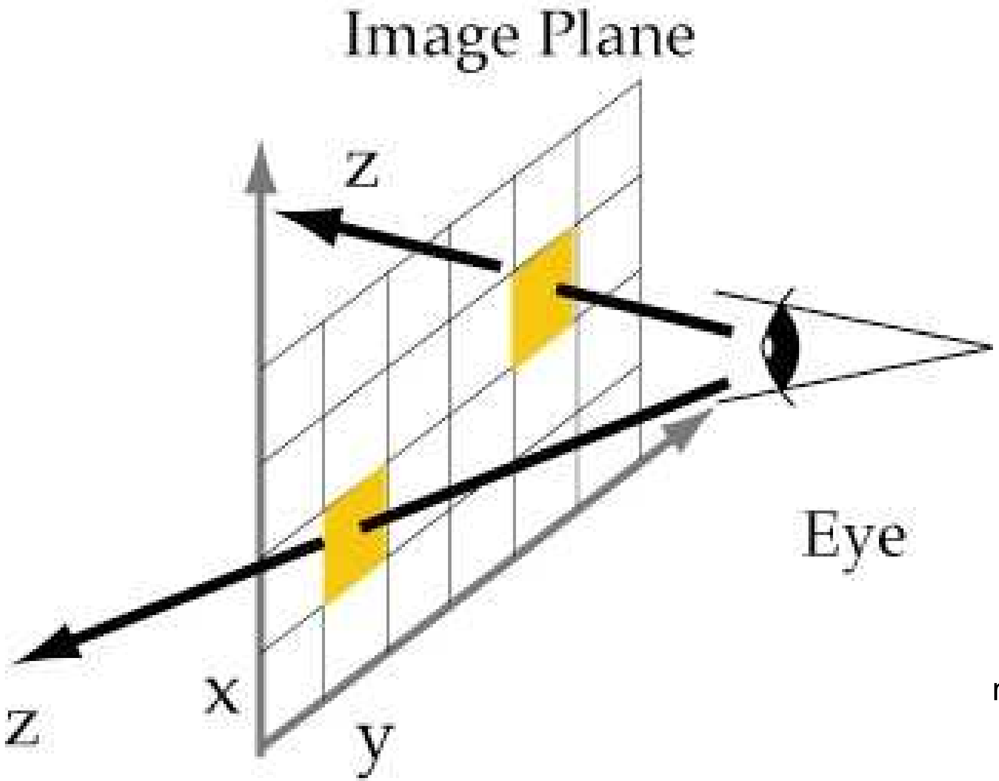
- Arbitrary coordinate system
- Where is the origin?
  - The Earth?
  - The Sun?
  - Greenwich meridian?
- Used to keep track of other systems

View Coordinates

- Tilt your head 90 degrees right
  - “Up” (y) is now to the right
  - “Backward” (z) hasn’t changed
  - We can use cross-product to get x
- Each camera / eye has coordinates

Clipping Coordinates

- coordinates in perspective projection
- z is distance from the eye



Clipping Coords

- Projection to *CCS* models the eye's lens
- In *CCS*, *z* is the distance from the eye
  - *not always in same direction*
  - i.e. Cartesian coordinates fail
  - homogeneous coordinates work
- View frustum is a box in these coords

Normalized DCS

- *Normalized Device Coordinate System*
- Big mouthful, but simple idea
- Divide *CCS* through by w
  - converts homogeneous coords to Cartesian
- Independent of screen coordinates

Device Coordinates

- Finally, we convert into *device* coords
  - (x, y): position on image plane
  - z: distance in front of image plane
- I.e. pixel position & object depth

Why so many?

Object Model | World Model | View Model

$$\begin{bmatrix}x\\y\\z\\w\end{bmatrix}_{OCS} \xrightarrow{\textit{Model Matrix}} \begin{bmatrix}x\\y\\z\\w\end{bmatrix}_{WCS} \xrightarrow{\textit{View Matrix}} \begin{bmatrix}x\\y\\z\\w\end{bmatrix}_{VCS}$$

Projection Matrix

$$\begin{bmatrix}x\\y\\z\\w\end{bmatrix}_{CCS} \xrightarrow{\textit{Perspective Division}} \begin{bmatrix}x\\y\\z\end{bmatrix}_{NDCS} \xrightarrow{\textit{Viewport Transformation}} \begin{bmatrix}x\\y\\z\end{bmatrix}_{DCS}$$

Clipping | Image Plane | Screen

Transformations

- Model Matrix
- View Matrix
- Projection Matrix
- Perspective Division
- Viewport Transformation

Model Matrix

- Most commonly changed matrix
- Converts vertices from *OCS* to *WCS*
- *Or* moves *OCS* around in *WCS*
- Used to position & orient objects in scene

Projection Matrix

- Less commonly changed
- Converts *VCS* to *CCS*
- Used to specify characteristics of *camera*
  - field of view, depth of field, symmetry
- Objects are usually *clipped* when in *CCS*
  - i.e. cut off at boundaries of frustum

Perspective Division

- Converts from homogeneous coordinates
- Gets rid of scale factor w
- Necessary for rasterization

Viewport Transformation

- Converts from *NDCS* to *DCS*
- *NDCS* is independent of window size
- *DCS* is not
- Used to separate rendering from O/S
- Change whenever window resized

Projective Rendering

- Raytracing is hard to accelerate
  - you don’t know what the ray will hit
  - processing gets *complicated*
- Projective rendering is much easier
  - just throw *all* the objects at the screen
  - broken down into simpler steps

Basic Idea

- Each object is modelled geometrically
  - as *primitives* built from *vertices*
- Vertices are then *projected* to screen
  - i.e. *geometric* computations
- Primitives are *rasterized*
  - i.e. *pixel (fragment)* computations

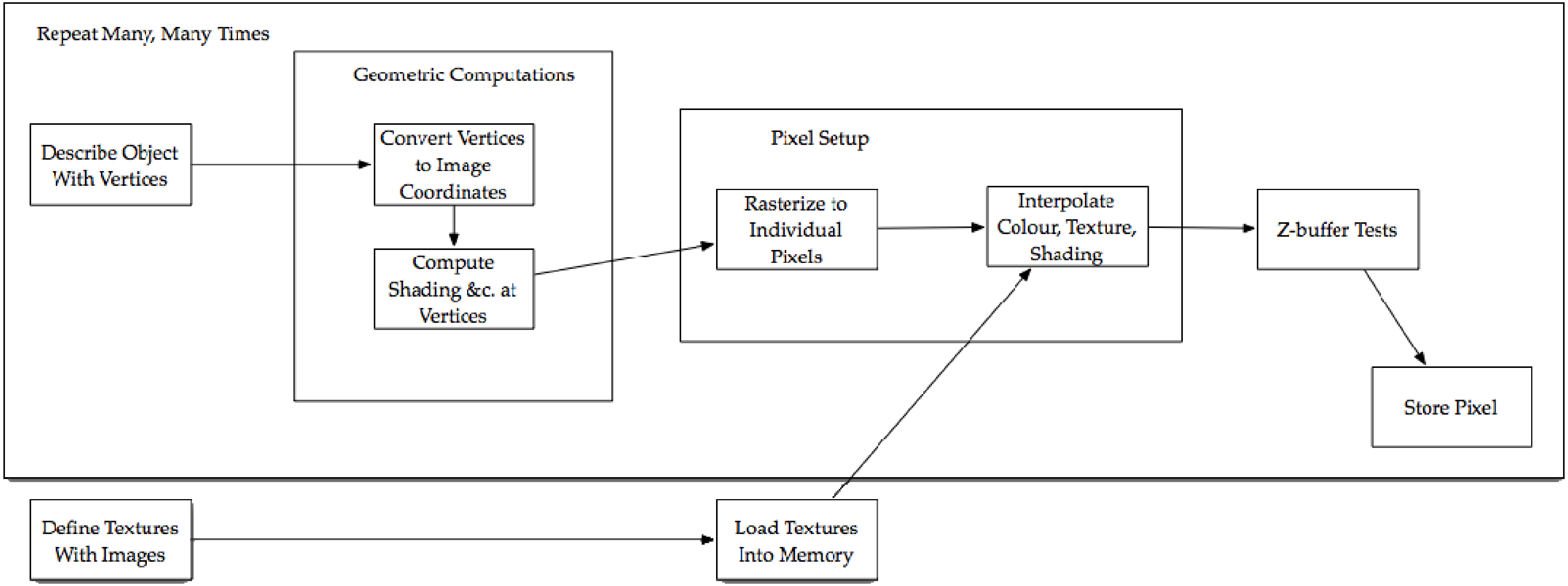
Advantages

- Simple programming model
  - no complicated intersection tests
- Abstracts object definition
- Optimizes matrix multiplications
- Well-suited to parallelization
  - especially in dedicated hardware

Disadvantages

- Lots of setup required
- Opaque black-box computation
  - debugging is *hard*
  - but bugs are pretty standard
- Hard to keep track of *meaning* of matrices

Projective Rendering



I: Setup

- Define *model, view, projection* matrices
- Define *viewport* transformation
- Define *lights* & lighting properties
- Define *shading & lighting* model
- Load *textures* as images
- Set *texture* properties

II: Define Objects

- Describe object with *primitives*
  - points, lines, triangles
  - made up of *vertices*
- Define *colour*, *material properties*, *textures*
  - for each vertex

III: Geometric Operations

- *Transform* vertices to image coordinates
  - apply *model, view, projection* matrices
  - perform *perspective* division
  - apply *viewport* transformation
- Also transform *normal* vectors
- *Clip* primitives to view volume

IV: Rasterization

- Rasterize primitives to *fragments*
  - i.e. pixels which may or may not render
- Use barycentric coordinates to interpolate
  - colour
  - material properties
  - texture coordinates

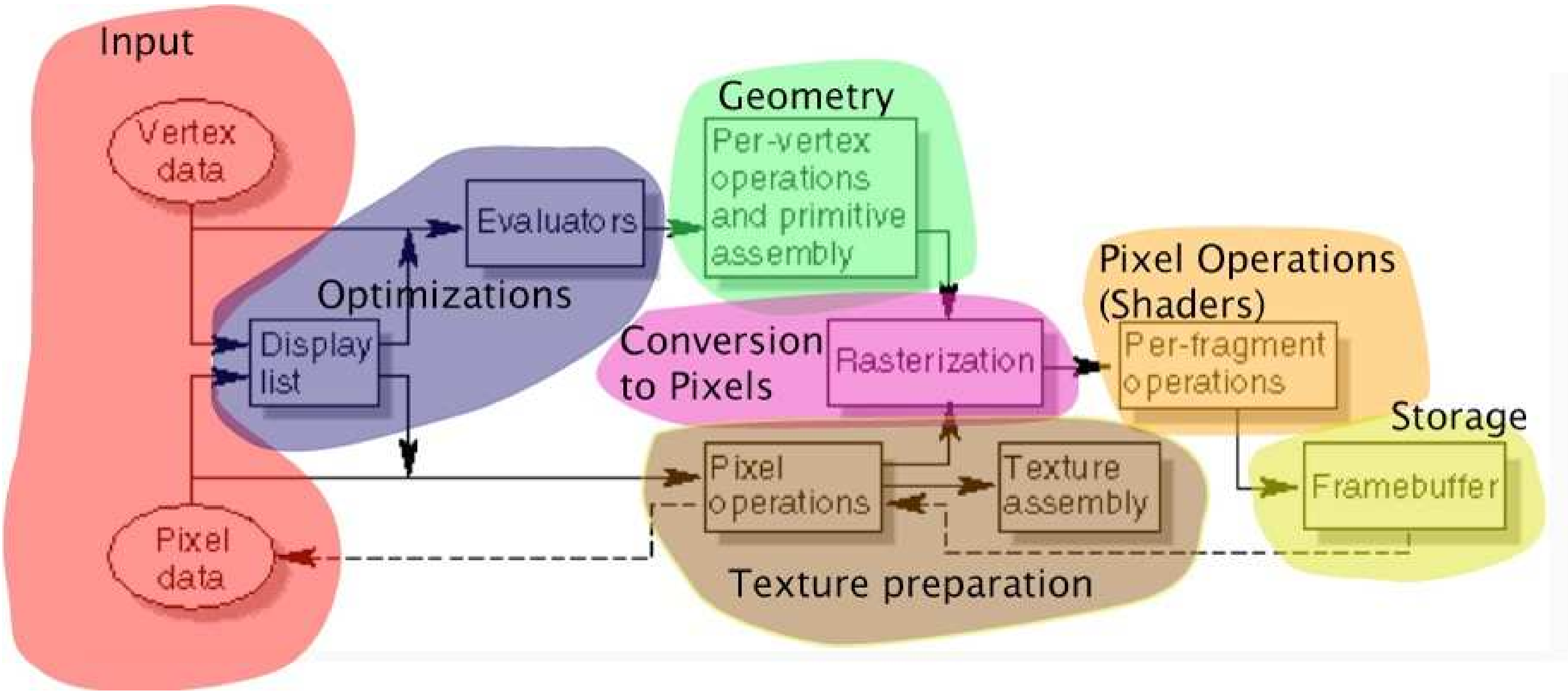
V: Pixel Operations

- For each *fragment*
  - find *Texel* colour
    - with bilinear or NN interpolation
  - *combine* texture & shading
    - *replace* or *modulate*
  - perform *fragment* tests
    - Stencil (aids shadows) , depth buffer (is one object in front of another, &c.

VI: Framebuffer

- Write pixel to framebuffer
- Keep two framebuffers (at least)
  - for *double-buffering*
- Hardware transfers framebuffer to screen
  - or to an image
  - or to a texture

OpenGL 2 Pipeline



- From the OpenGL Red Book