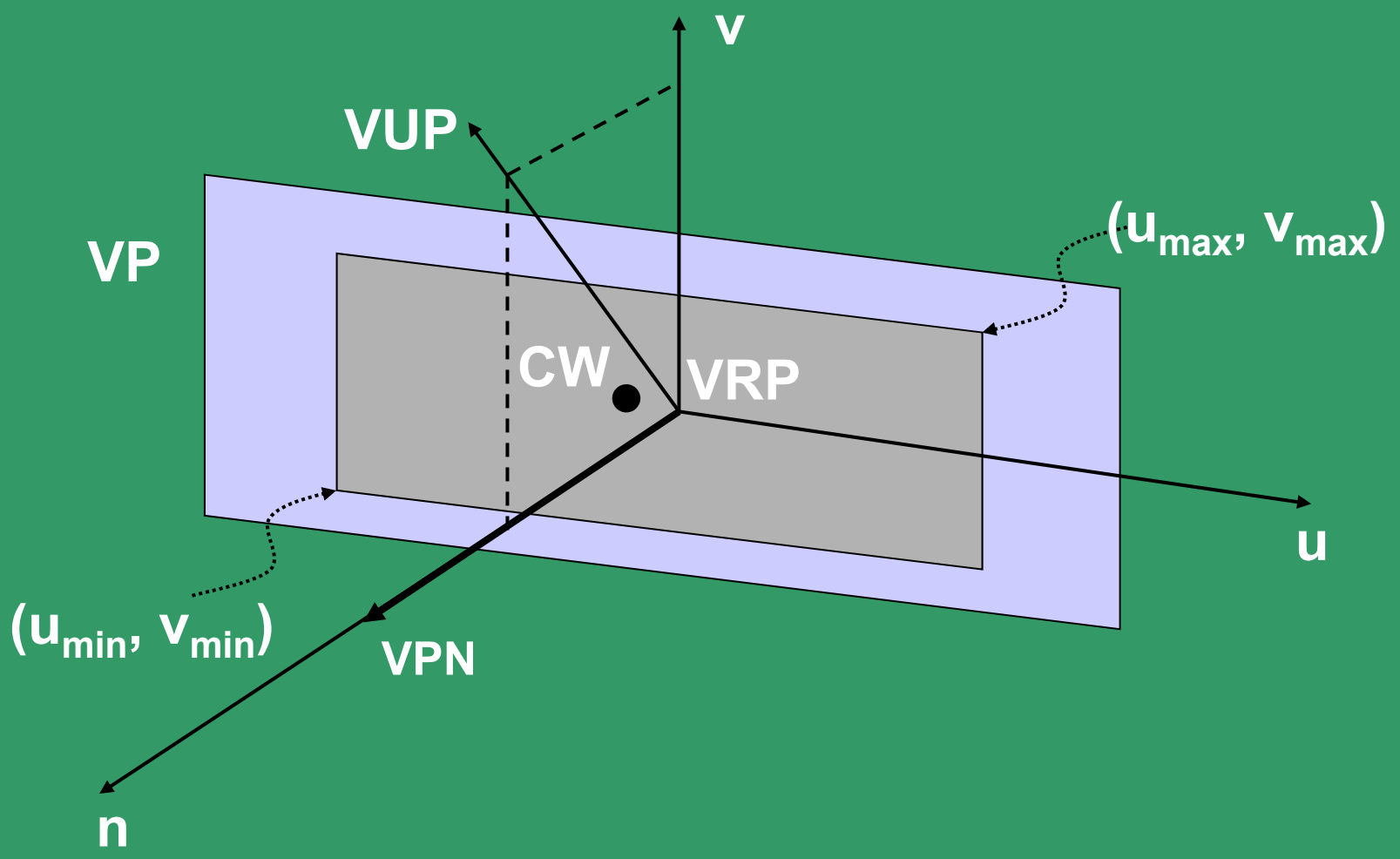
3D Viewing –

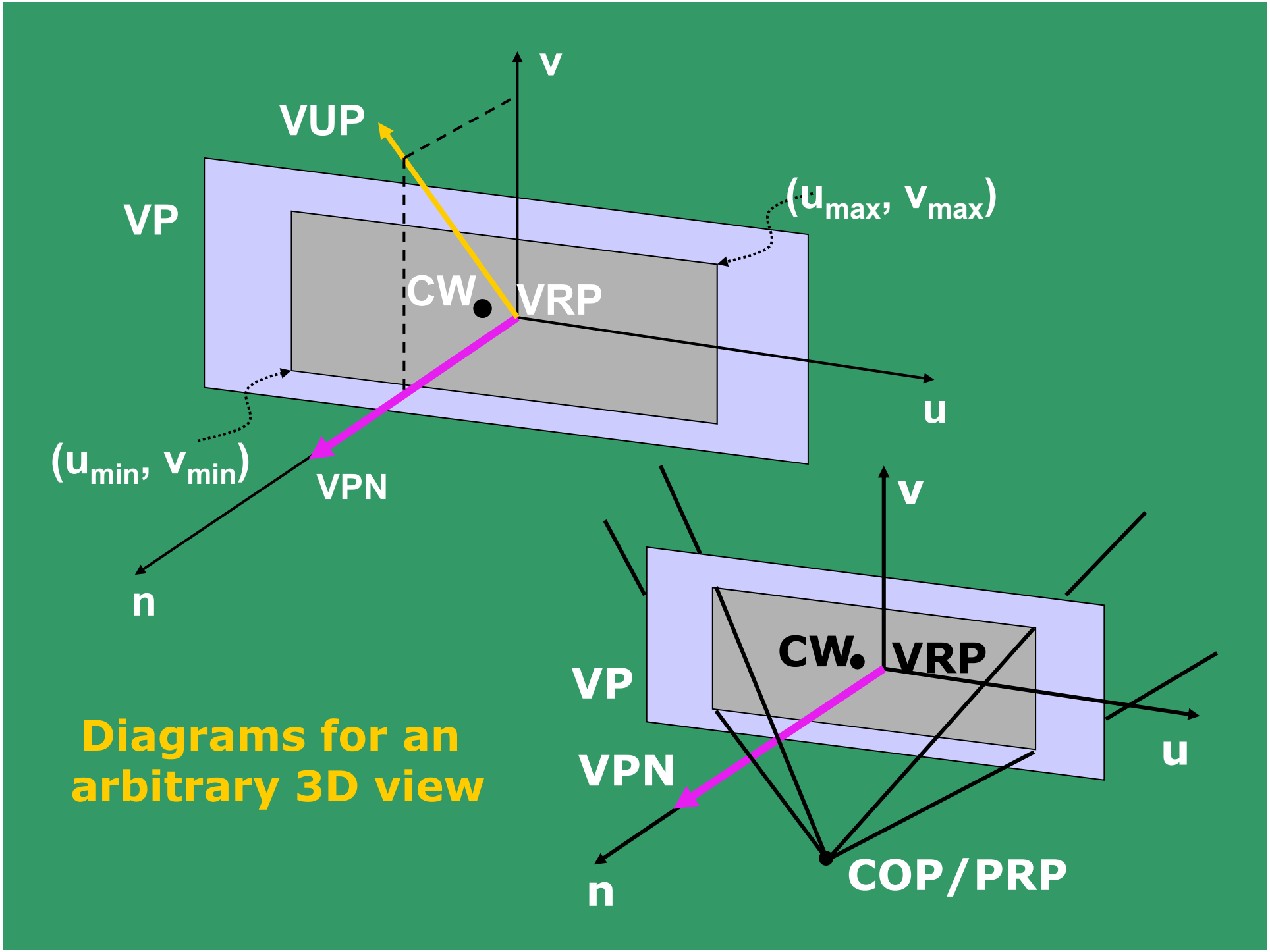
Projection Transformations

and

Viewing Pipeline

View Specifications:
VP, VRP, VUP, VPN, PRP, DOP, CW, VRC





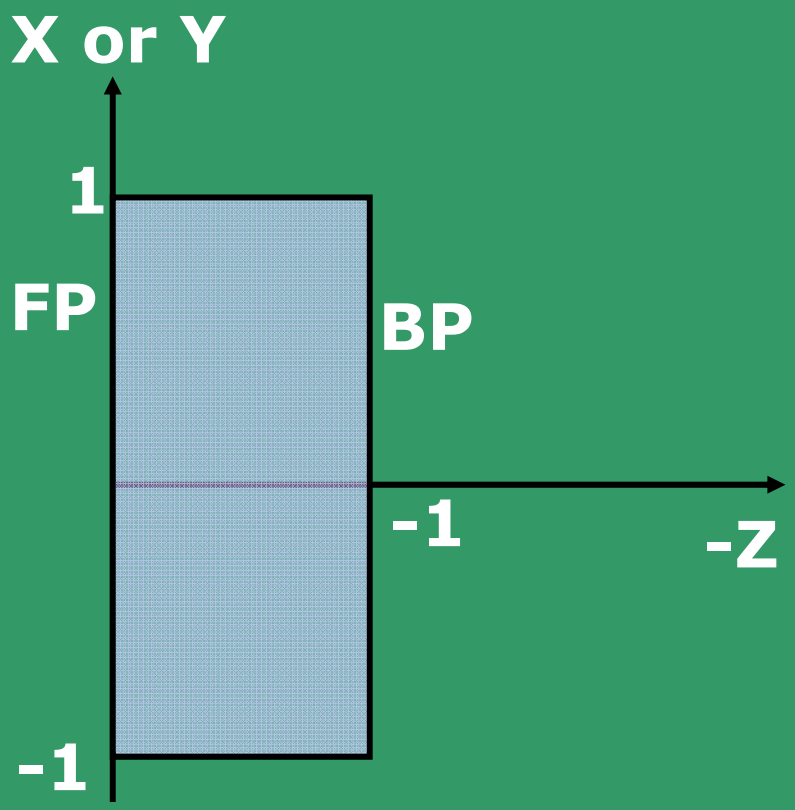
Diagrams for an arbitrary 3D view

Specifying an Arbitrary 3D View

Viewing Parameter	Example Values		
	Set 1	Set 2	Set 3
VRP (WC)	(0, 0, 54)	(16, 0, 54)	(0, 0, 0)
VPN (WC)	(0, 0, 1)	(0, 1, 0)	(0, 0, 1)
VUP (WC)	(0, 1, 0)	(-1, 0, 0)	(0, 1, 0)
PRP (VRC)	(8, 6, 30)	(12, 8, 30)	(8, 6, 84)
Window (VRC)	(-1, 17, -1, 17)	(-1, 25, -5, 21)	(-50, 50, -50, 50)
Projection Type	Perspective	Parallel	Perspective
F & B (VRC)	+1, -23	-	-

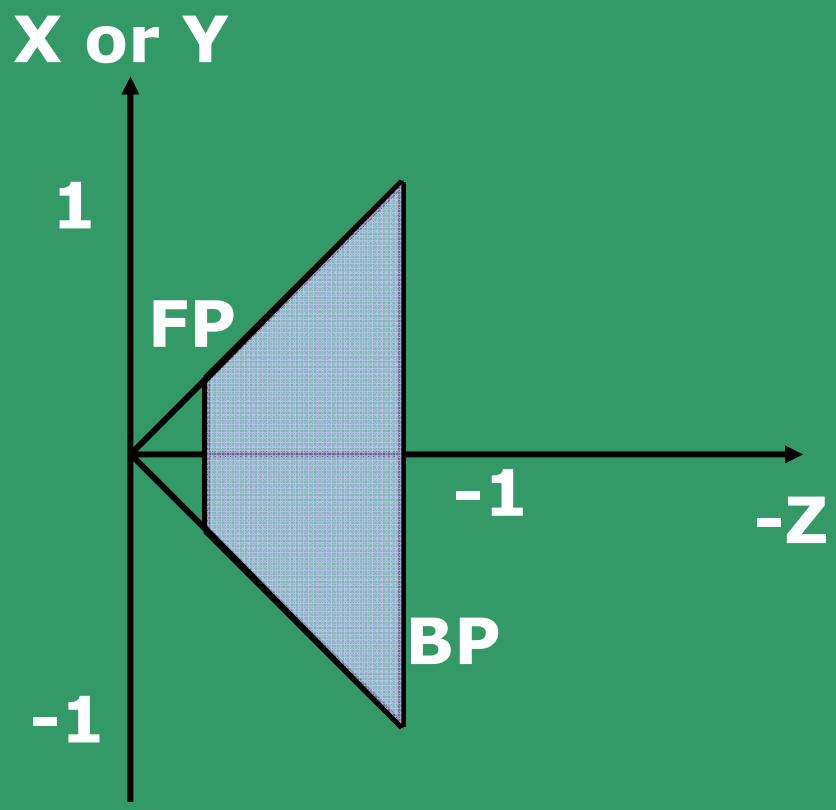
Canonical view volume for *parallel projection* is defined by six planes:

X = -1;	Y = -1;	Z = 0;
X = 1;	Y = 1;	Z = -1.

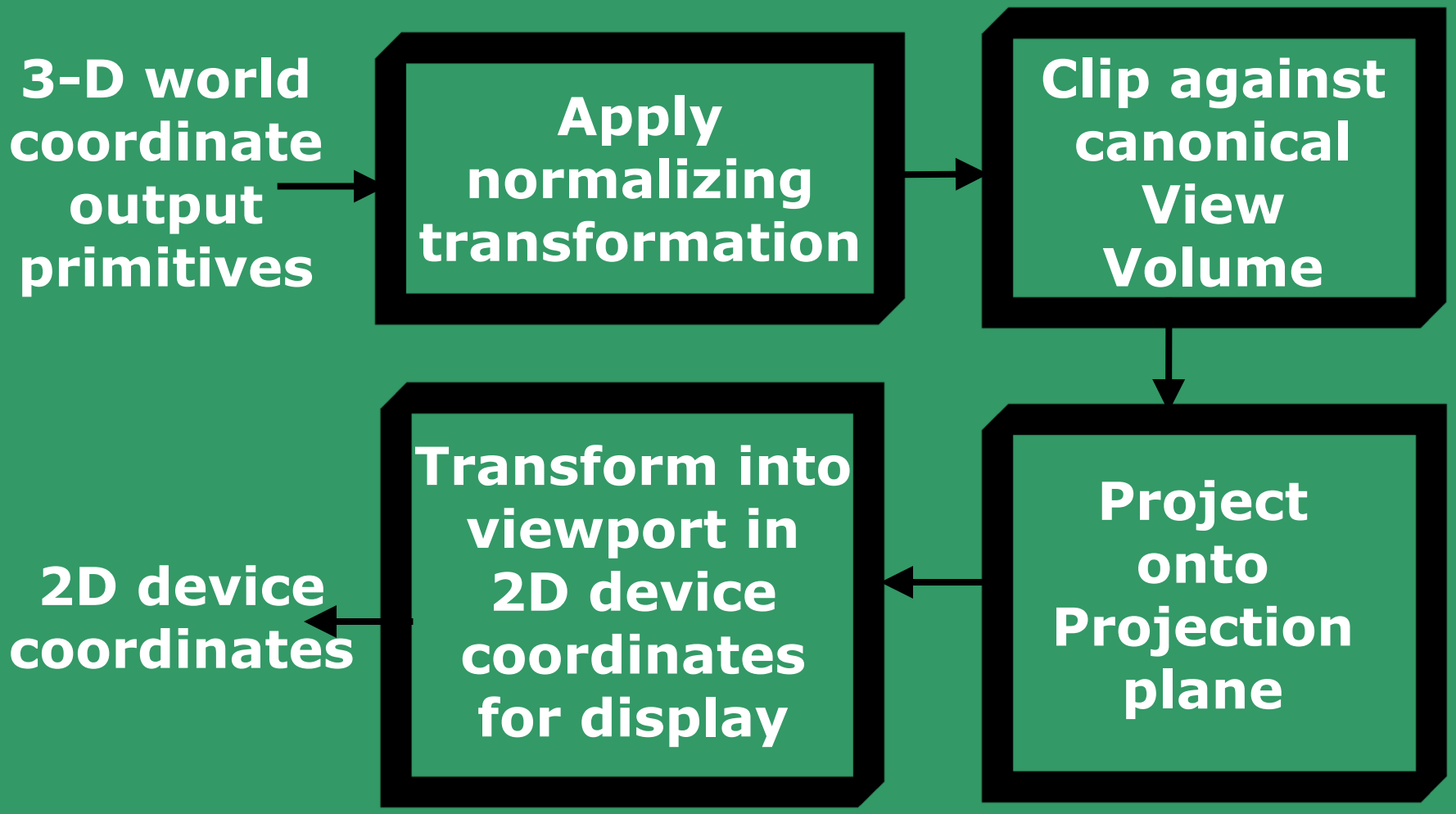


Canonical view volume for *perspective projection* is defined by six planes:

$X = Z;$	$Y = -Z;$	$Z = -Z_{min};$
$X = -Z;$	$Y = Z;$	$Z = -1.$



Implementation of 3D Viewing



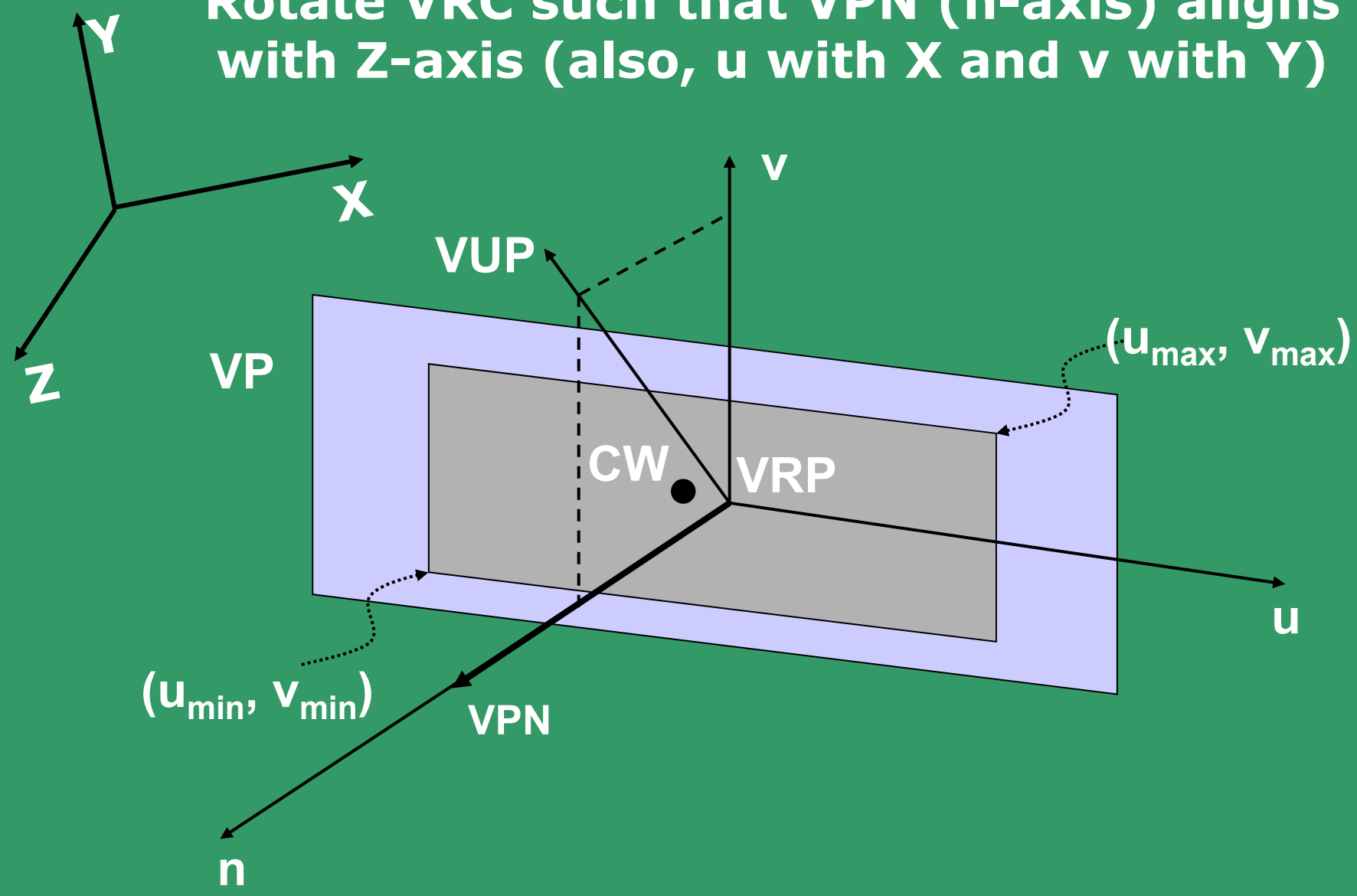
Steps for implementing normalizing transformation matrix for *parallel projection*

- Translate the VRP to origin
- Rotate VRC such that VPN (n-axis) aligns with Z-axis (also, u with X and v with Y-axis)
- Shear (not necessary for pure orthographic) such that DOP is parallel to the Z-axis
- Translate and scale into parallel-projection canonical view volume (CVV)

$$N_{par} = S_{par}\, T_{par}\, SH_{par}\, R\, T(-VRP)$$

Step 2 in normalizing transformations:

Rotate VRC such that VPN (n-axis) aligns with Z-axis (also, u with X and v with Y)



Expressions for Step 2 must be derived.

Implement using the concept of combined transformation (rotation).

Take $R_x =$

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\alpha) & -\sin(\alpha) & 0 \\ 0 & \sin(\alpha) & \cos(\alpha) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- Rows are unit vectors, when rotated by R_x, will align with the Y and Z axis respectively.
- When unit vectors along the principle axes are rotated by R_x, they form the column vectors.

Take $R_x =$

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\alpha) & -\sin(\alpha) & 0 \\ 0 & \sin(\alpha) & \cos(\alpha) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

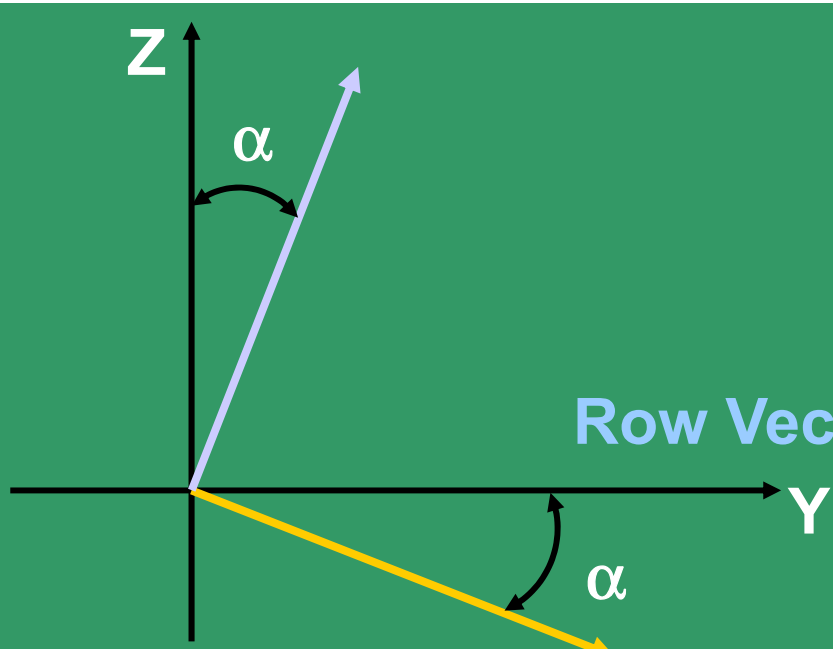
Consider R_x^{-1} as comprised
of row vectors of R_x;
Operation is Pre-Mult, with columns vectors

$$R_x R_x^{-1} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\alpha) & -\sin(\alpha) & 0 \\ 0 & \sin(\alpha) & \cos(\alpha) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\alpha) & \sin(\alpha) & 0 \\ 0 & -\sin(\alpha) & \cos(\alpha) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Now visualize the same,
as Post-multiplication – any issues ??

$$R_x I = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\alpha) & -\sin(\alpha) & 0 \\ 0 & \sin(\alpha) & \cos(\alpha) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

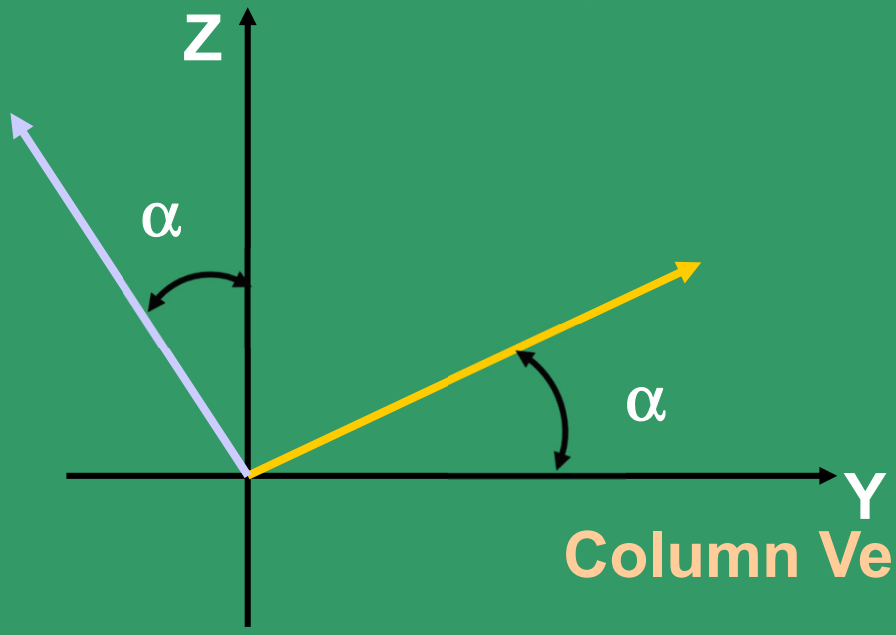
- Rows are unit vectors, when rotated by R_x, will align with the Y and Z axis respectively.
- When unit vectors along the principle axes are rotated by R_x, they form the column vectors.



$$R_x = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\alpha) & -\sin(\alpha) & 0 \\ 0 & \sin(\alpha) & \cos(\alpha) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Row Vectors:

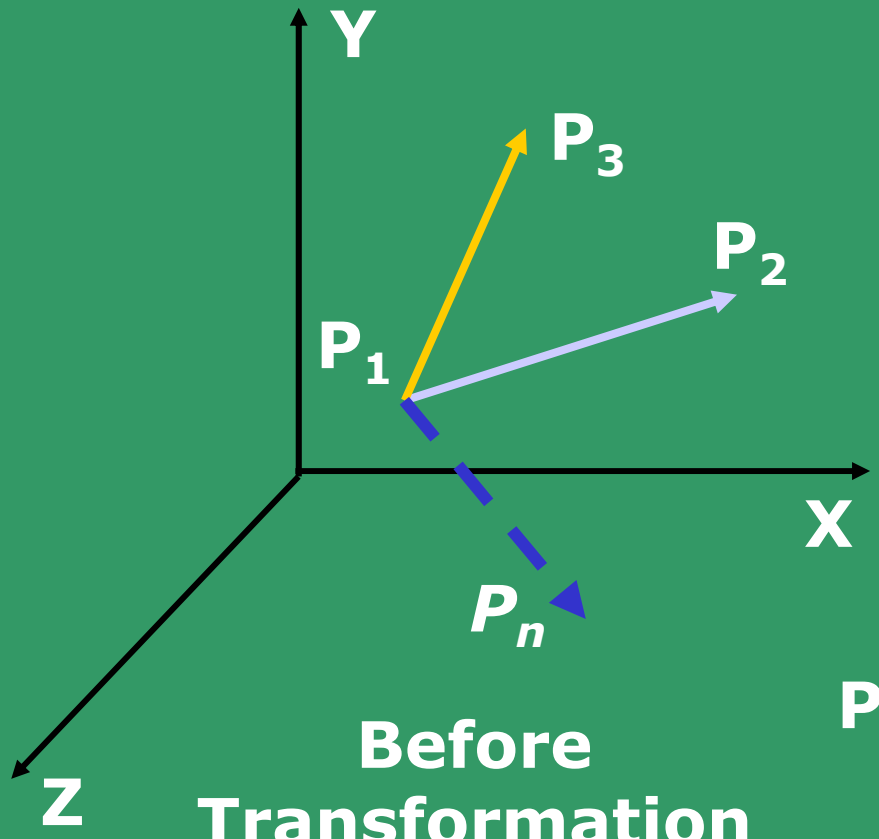
$$\begin{bmatrix} 1 & 0 & 0 \end{bmatrix}$$
$$\begin{bmatrix} 0 & \cos(\alpha) & -\sin(\alpha) \end{bmatrix}$$
$$\begin{bmatrix} 0 & \sin(\alpha) & \cos(\alpha) \end{bmatrix}$$



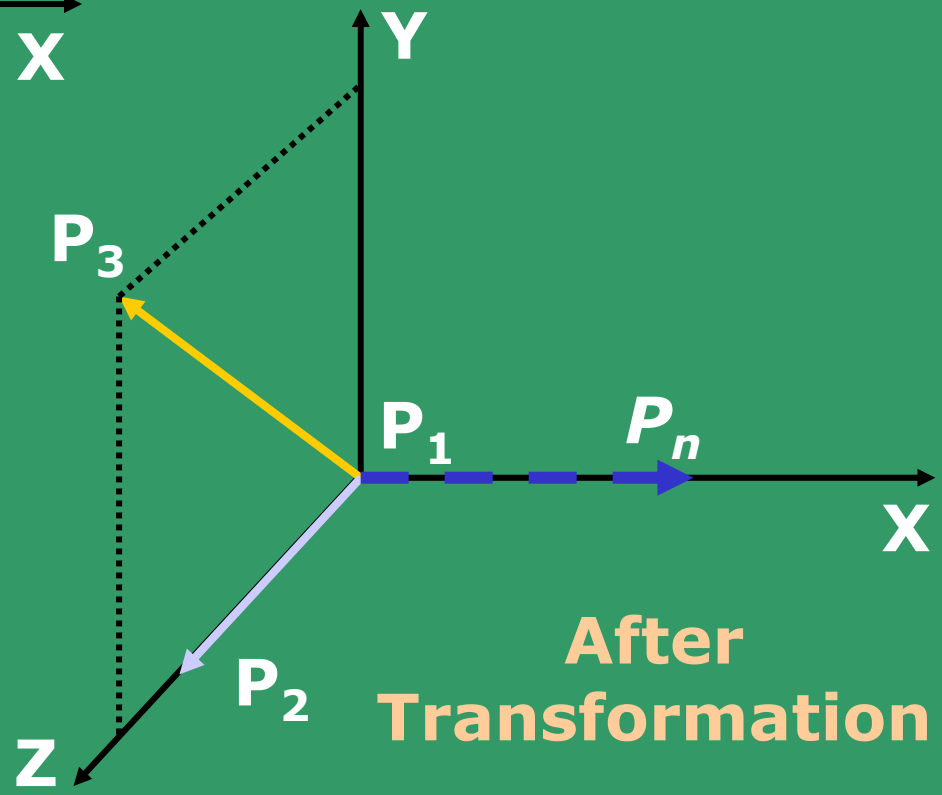
Column Vectors

$$\begin{bmatrix} 1 & 0 & 0 \end{bmatrix}^T$$
$$\begin{bmatrix} 0 & \cos(\alpha) & \sin(\alpha) \end{bmatrix}^T$$
$$\begin{bmatrix} 0 & -\sin(\alpha) & \cos(\alpha) \end{bmatrix}^T$$

Consider a general scenario of combined rotations and use the property derived based on the orthogonality of the R matrix.



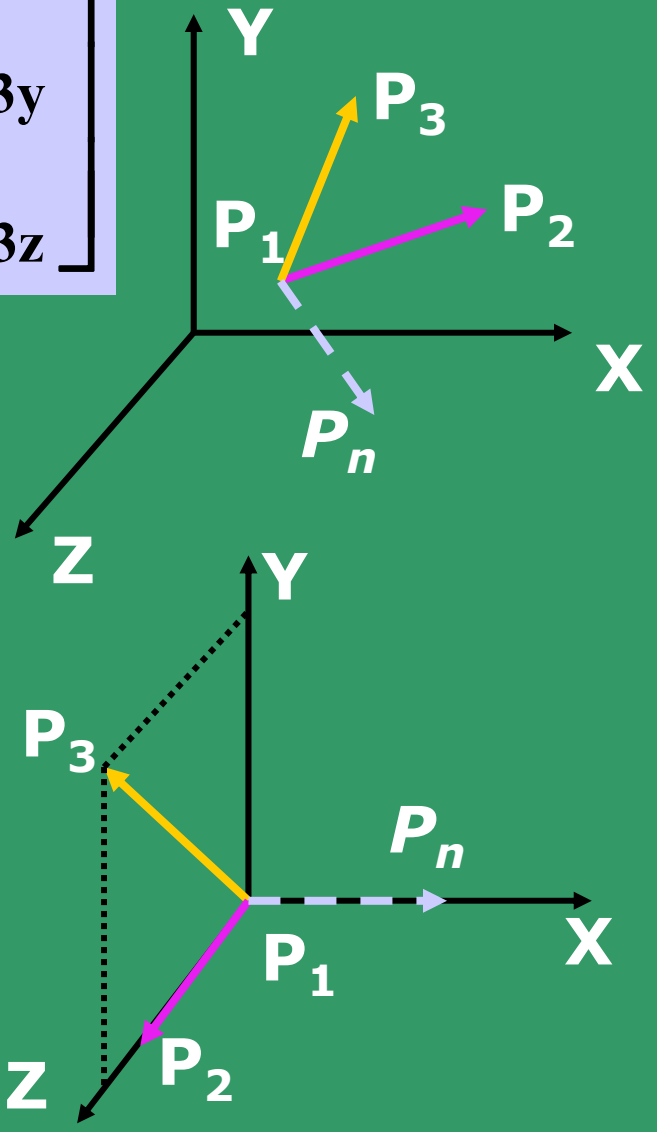
Before Transformation



After Transformation

Let the **effective rotation matrix** be a combination of three rows as:

$$\begin{bmatrix} r_{1x} & r_{2x} & r_{3x} \\ r_{1y} & r_{2y} & r_{3y} \\ r_{1z} & r_{2z} & r_{3z} \end{bmatrix}$$



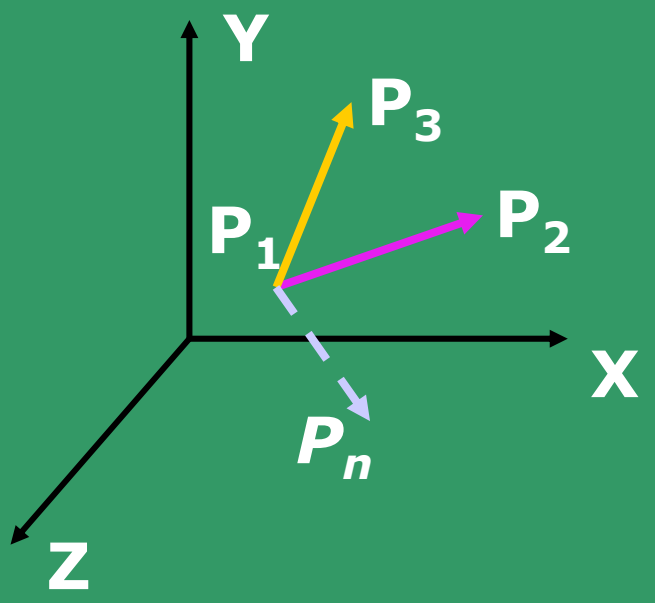
where,

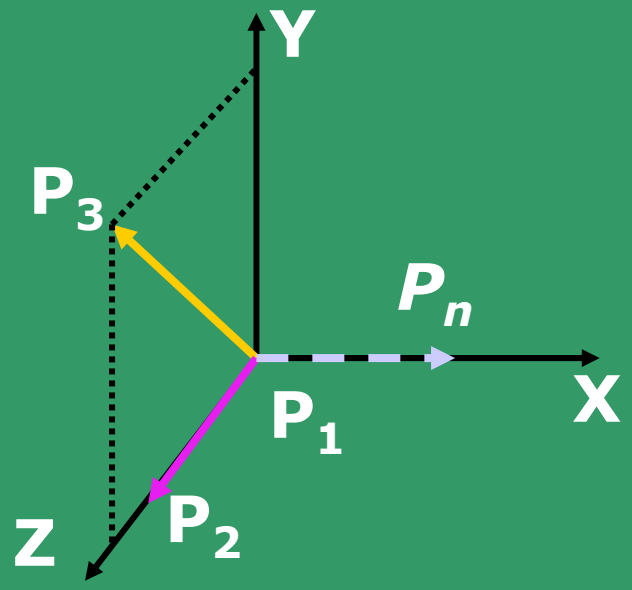
$$R_z = \begin{bmatrix} r_{1z} & r_{2z} & r_{3z} \end{bmatrix}^T = \frac{P_1P_2}{|P_1P_2|}$$

$$R_x = \begin{bmatrix} r_{1x} & r_{2x} & r_{3x} \end{bmatrix}^T = \frac{P_1P_2 \ X \ P_1P_3}{|P_1P_2 \ X \ P_1P_3|}$$

and

$$R_y = \begin{bmatrix} r_{1y} & r_{2y} & r_{3y} \end{bmatrix}^T = R_z \ X \ R_x$$



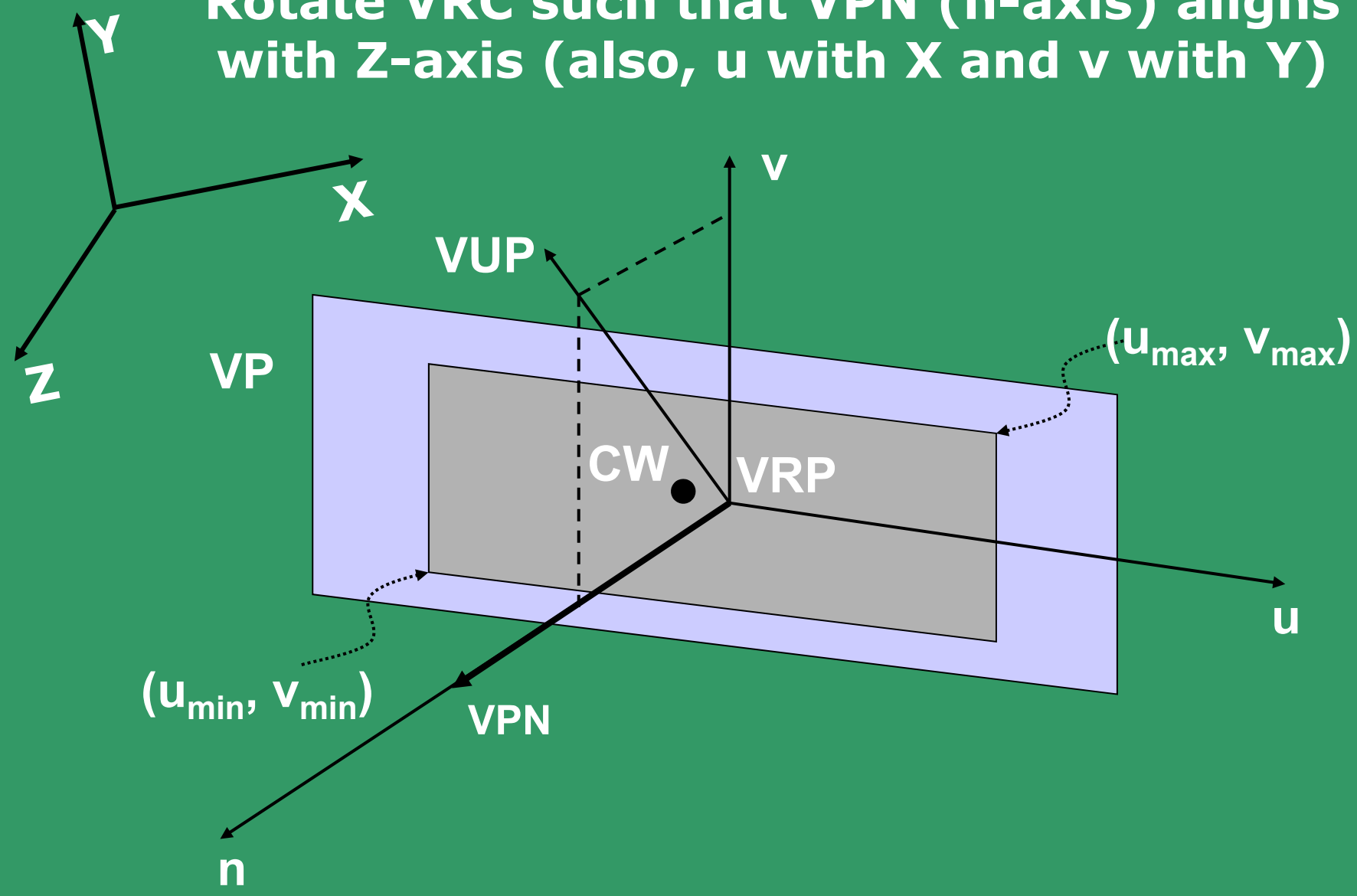


Thus the rotation matrix of step 2 in normalizing transformations, can be formulated as:

$$R = \begin{bmatrix} r_{1x} & r_{2x} & r_{3x} & 0 \\ r_{1y} & r_{2y} & r_{3y} & 0 \\ r_{1z} & r_{2z} & r_{3z} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Step 2 in normalizing transformations:

Rotate VRC such that VPN (n-axis) aligns with Z-axis (also, u with X and v with Y)



where,

$$R_z = \frac{VPN}{|VPN|};$$

$$R_x = \frac{VUP \times R_z}{|VUP \times R_z|};$$

$$and\ R_y = R_z \times R_x$$

The overall combined transformation matrix for parallel projection (*WCSVV -> PPCVV*), is:

$$N_{par} = S_{par}\ T_{par}\ SH_{par}\ R\ T(-VRP)$$

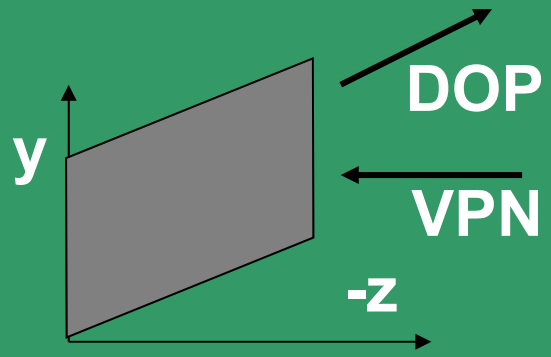
The overall combined transformation matrix for parallel projection (*WCSVV -> PPCVV*), is:

$$N_{par} = S_{par}\, T_{par}\, SH_{par}\, R\, T(-VRP)$$

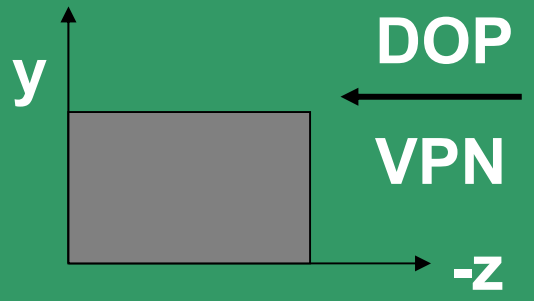
where,

$$SH_{par} = \begin{bmatrix} 1 & 0 & shx_{par} & 0 \\ 0 & 1 & shy_{par} & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$shx_{par} = -\frac{dop_x}{dop_z}; \qquad shy_{par} = -\frac{dop_y}{dop_z}$$



Side view of shearing of the VV



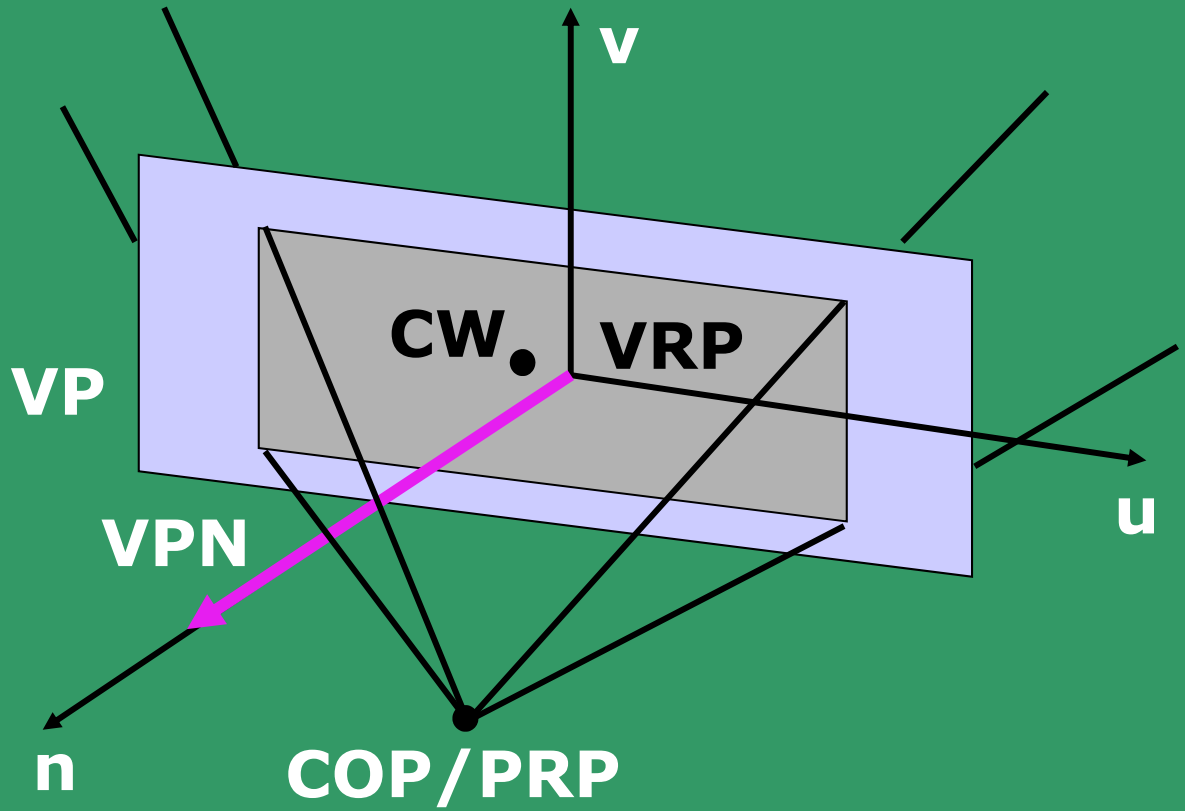
The overall combined transformation matrix for parallel projection (*WCSVV -> PPCVV*), is:

$$N_{par} = S_{par}\, T_{par}\, SH_{par}\, R\, T(-VRP)$$

$$T_{par} = \begin{bmatrix} -\frac{u_{\max} + u_{\min}}{2} & -\frac{v_{\max} + v_{\min}}{2} & FP \end{bmatrix};$$

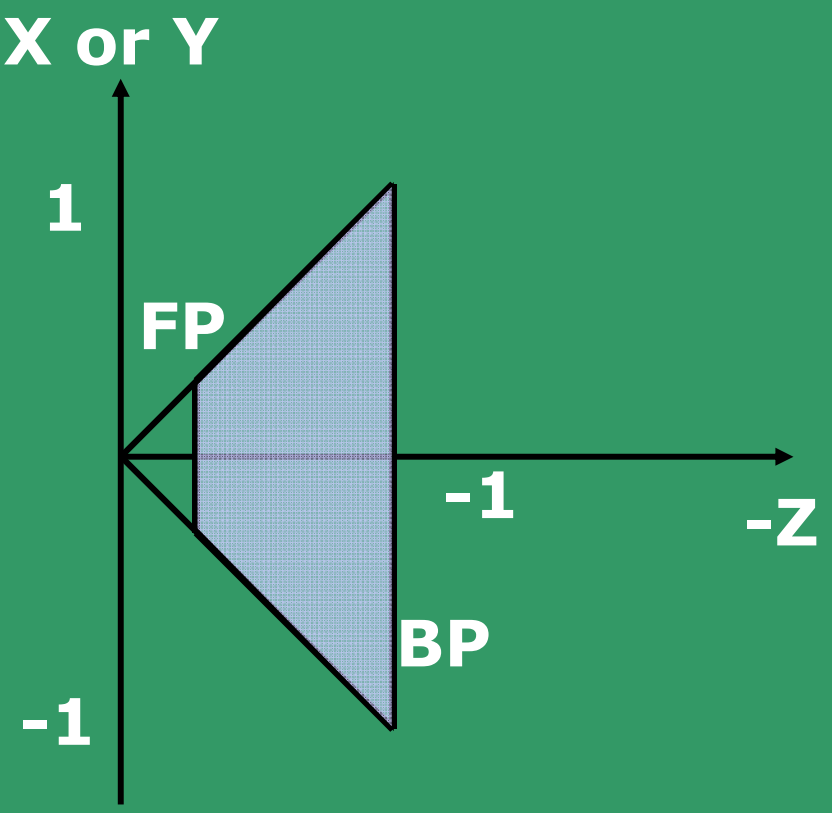
$$S_{par} = S\left(\frac{2}{u_{\max} - u_{\min}}, \frac{2}{v_{\max} - v_{\min}}, \frac{1}{FP - BP}\right)$$

Implementing normalizing transformation matrix for perspective projection



Canonical view volume for *perspective projection* is defined by six planes:

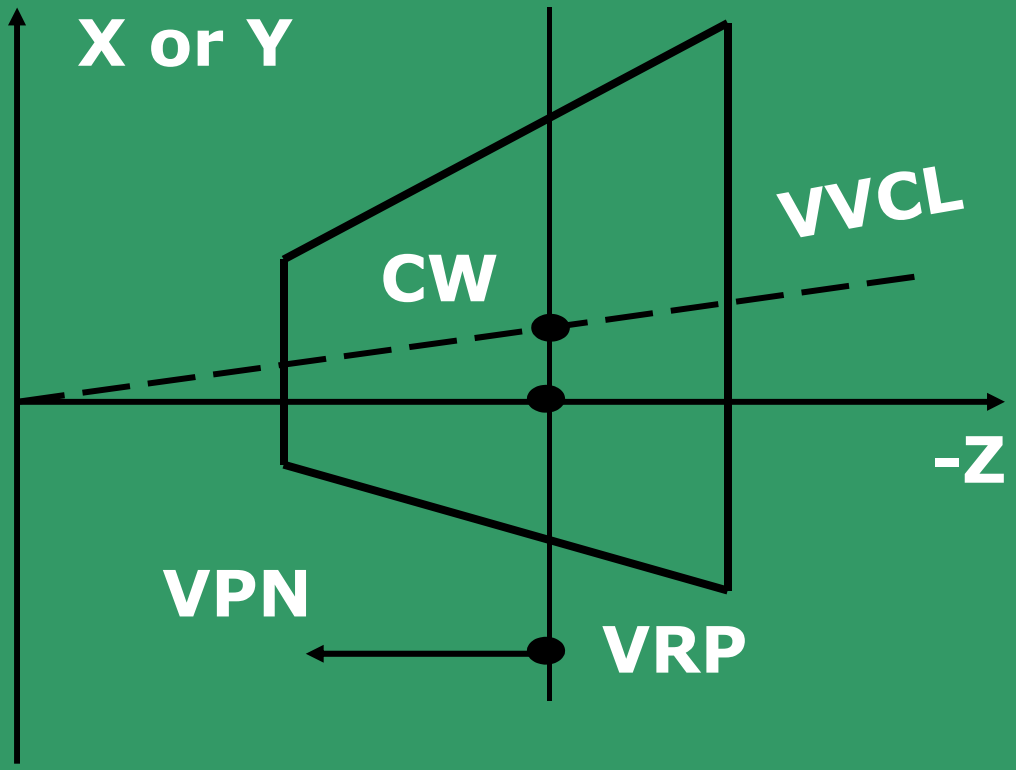
$X = Z;$	$Y = -Z;$	$Z = -Z_{min};$
$X = -Z;$	$Y = Z;$	$Z = -1.$



Steps for implementing normalizing transformation matrix for perspective projection

- Translate the VRP to origin
- Rotate VRC such that VPN (n-axis) aligns with Z-axis (also, u with X- and v with Y-axis)
- Translate such that COP (or PRP) is at the origin
- Shear such that center line of view volume (VVCL) becomes z-axis
- Scale such that VV becomes the canonical view volume (CVV)

Scenario of the cross-section of the VV after first three transformations.



$$N_{per} = S_{per}\, SH_{par}\, T(-PRP)\, R\, T(-VRP)$$

Comparison the overall combined transformation matrices for:

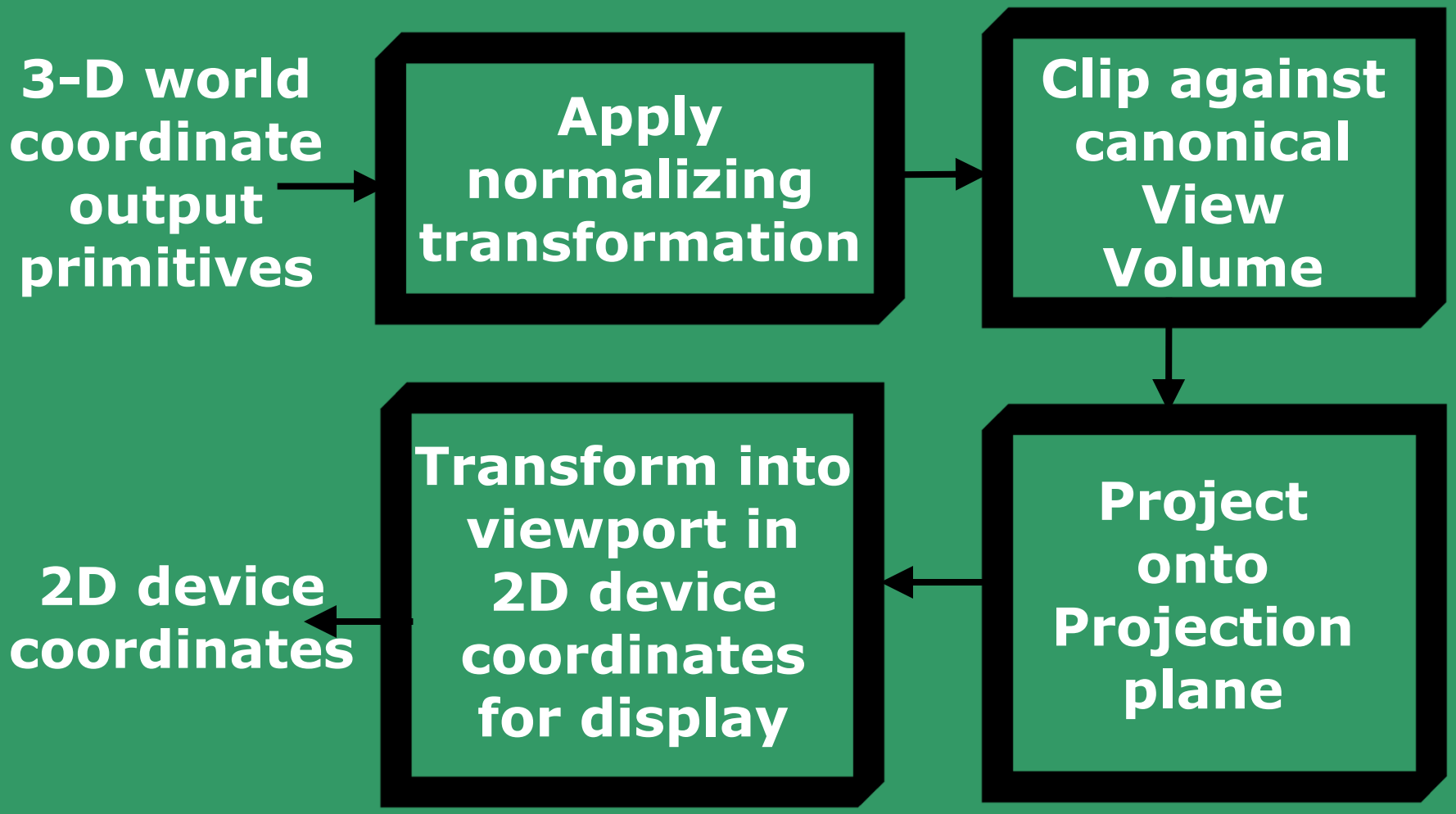
PARALLEL PROJECTION:

$$N_{par} = S_{par}\ T_{par}\ SH_{par}\ R\ T(-VRP)$$

PERSPECTIVE PROJECTION:

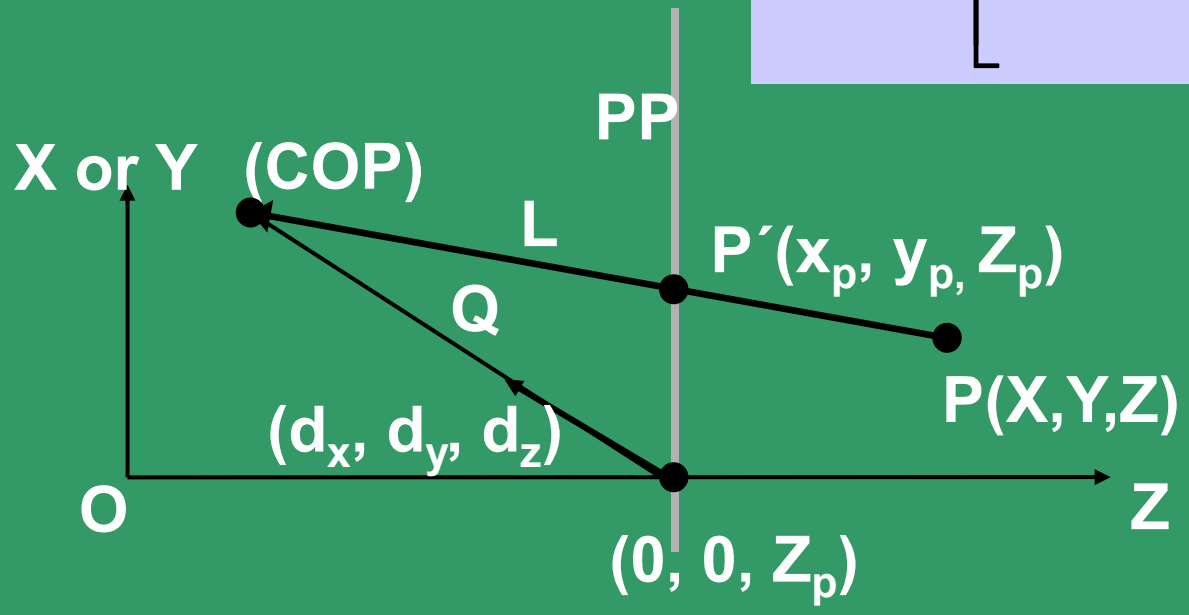
$$N_{per} = S_{per}\ SH_{par}\ T(-PRP)\ R\ T(-VRP)$$

Implementation of 3D Viewing

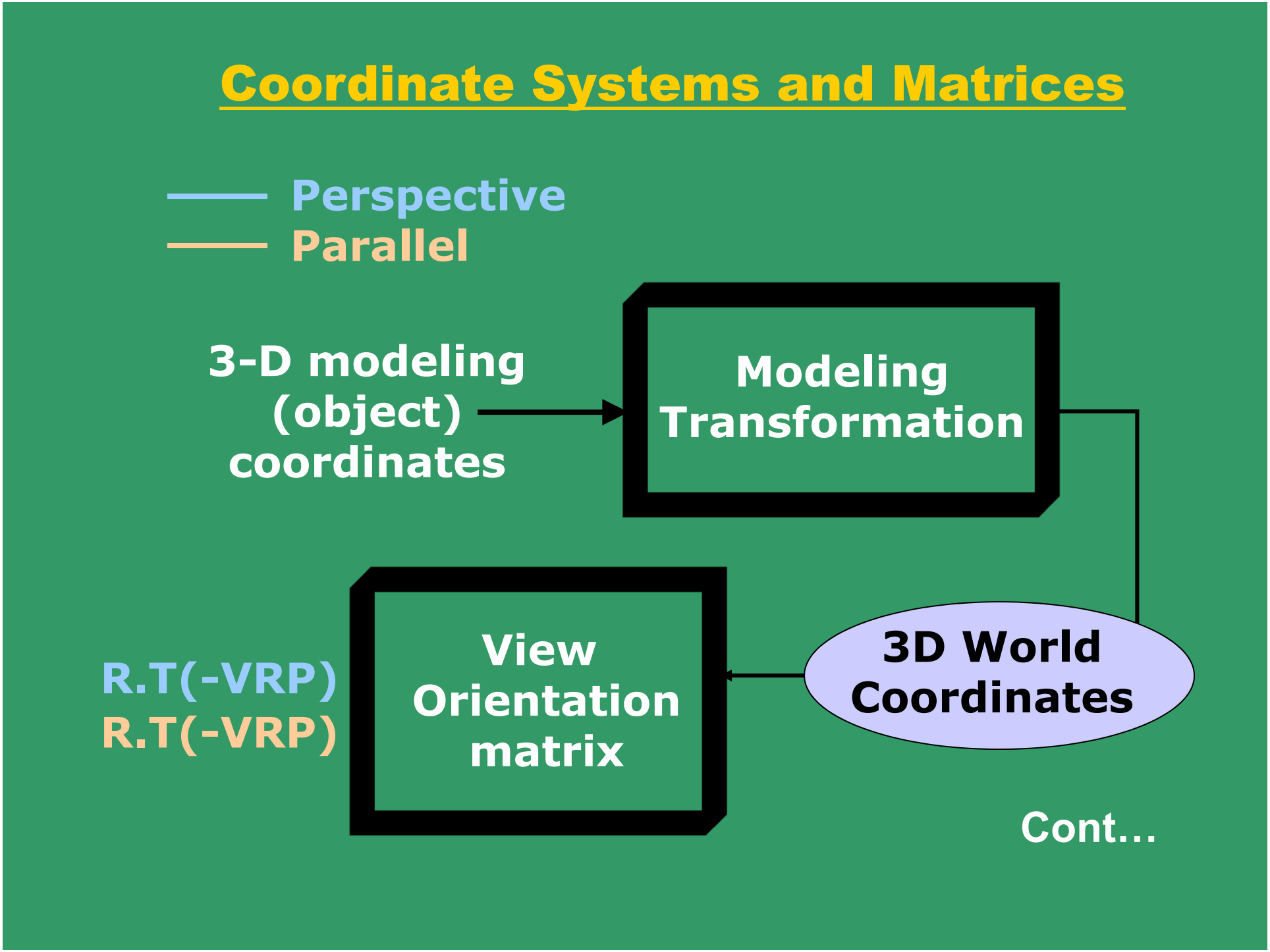


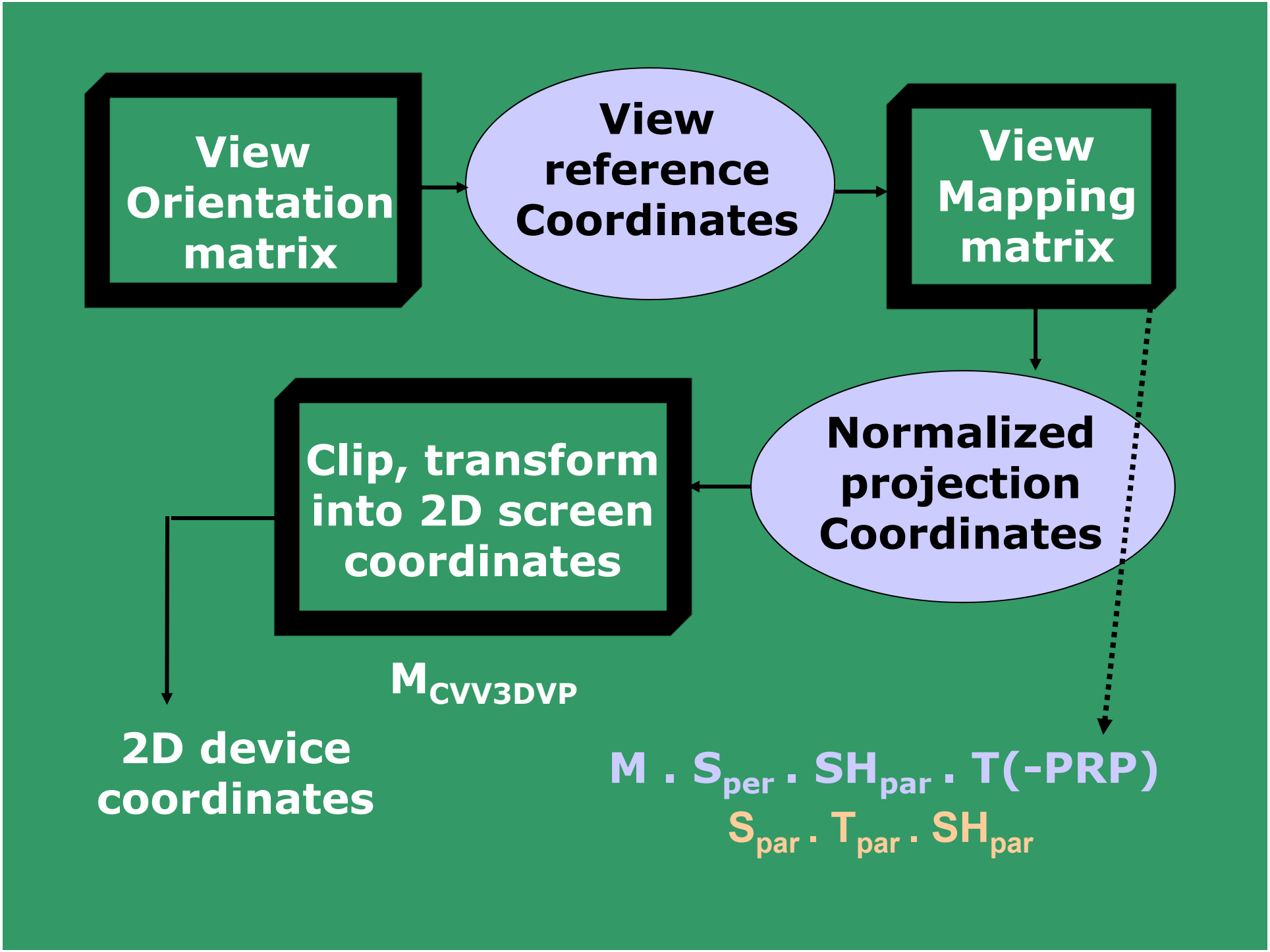
Generalized formula of perspective projection matrix:

$$M_{gen} = \begin{bmatrix} 1 & 0 & -\frac{d_x}{d_z} & z_p \frac{d_x}{d_z} \\ 0 & 1 & -\frac{d_y}{d_z} & z_p \frac{d_y}{d_z} \\ 0 & 0 & -\frac{Z_p}{Qd_z} & \frac{Z_p^2}{Qd_z} + Z_p \\ 0 & 0 & -\frac{1}{Qd_z} & \frac{Z_p}{Qd_z} + 1 \end{bmatrix}$$



Coordinate Systems and Matrices




View Orientation matrix
View reference Coordinates
View Mapping matrix
Clip, transform into 2D screen coordinates
Normalized projection Coordinates
$M_{CVV3DVP}$
2D device coordinates
$M . S_{per} . SH_{par} . T(-PRP)$
$S_{par} . T_{par} . SH_{par}$

where after clipping, use

$M_{CVV3DVP}$ =

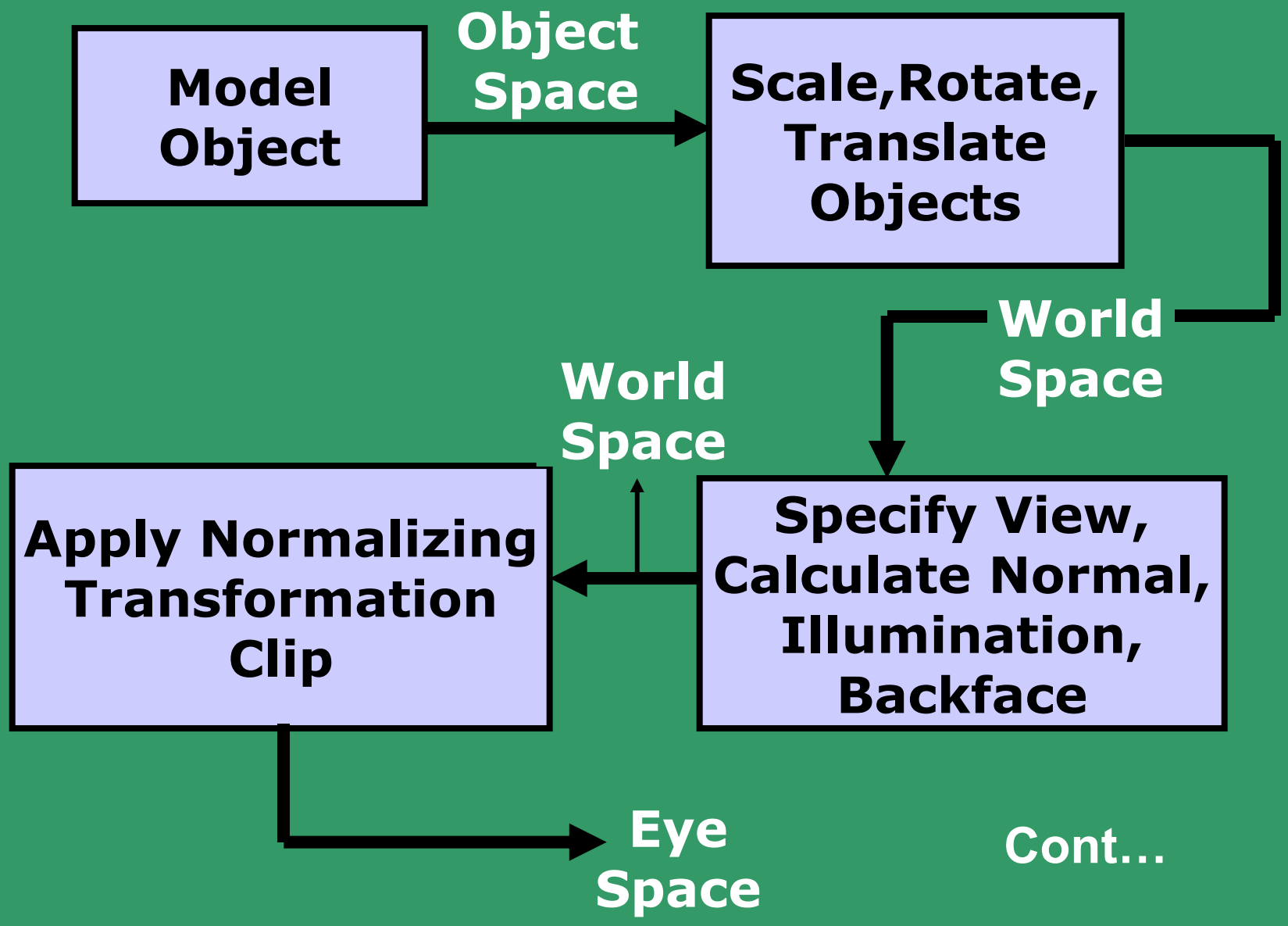
$$T(X_{vmin}, Y_{vmin}, Z_{vmin}) \cdot S\left(\frac{X_{vmax} - X_{vmin}}{2}, \frac{Y_{vmax} - Y_{vmin}}{2}, Z_{vmax} - Z_{vmin}\right) \cdot T(1,1,1)$$

The 3D Viewing Pipeline

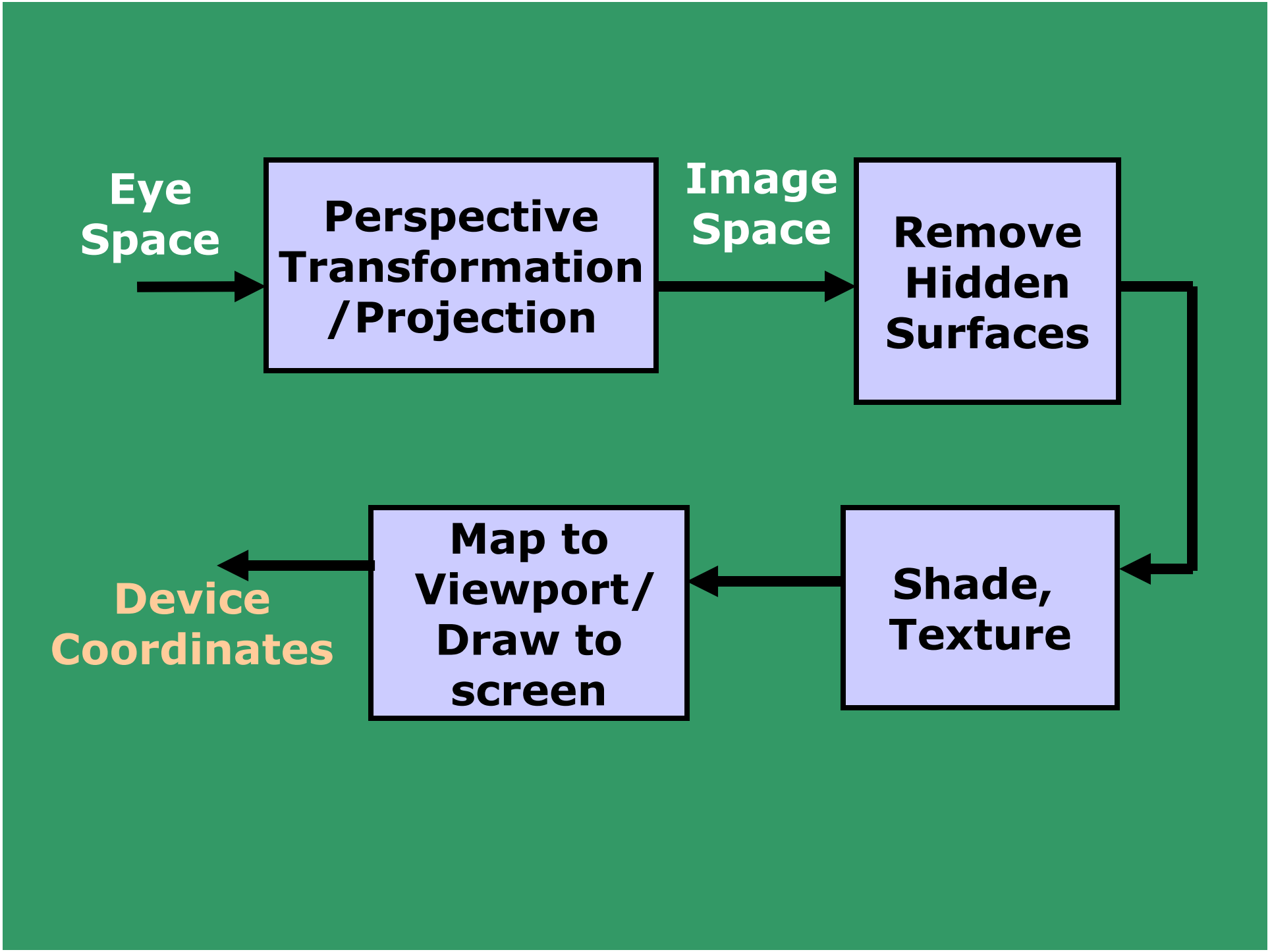
- Objects are modeled in object (modeling) space.
- Transformations are applied to the objects to position them in world space.
- View parameters are specified to define the view volume of the world, a projection plane, and the viewport on the screen.

- **Objects are clipped to this View volume.**
- **The results are projected onto the projection plane (window) and finally mapped into the 3D viewport.**
- **Hidden objects are then removed.**
- **The objects are scan converted and then shaded if necessary.**

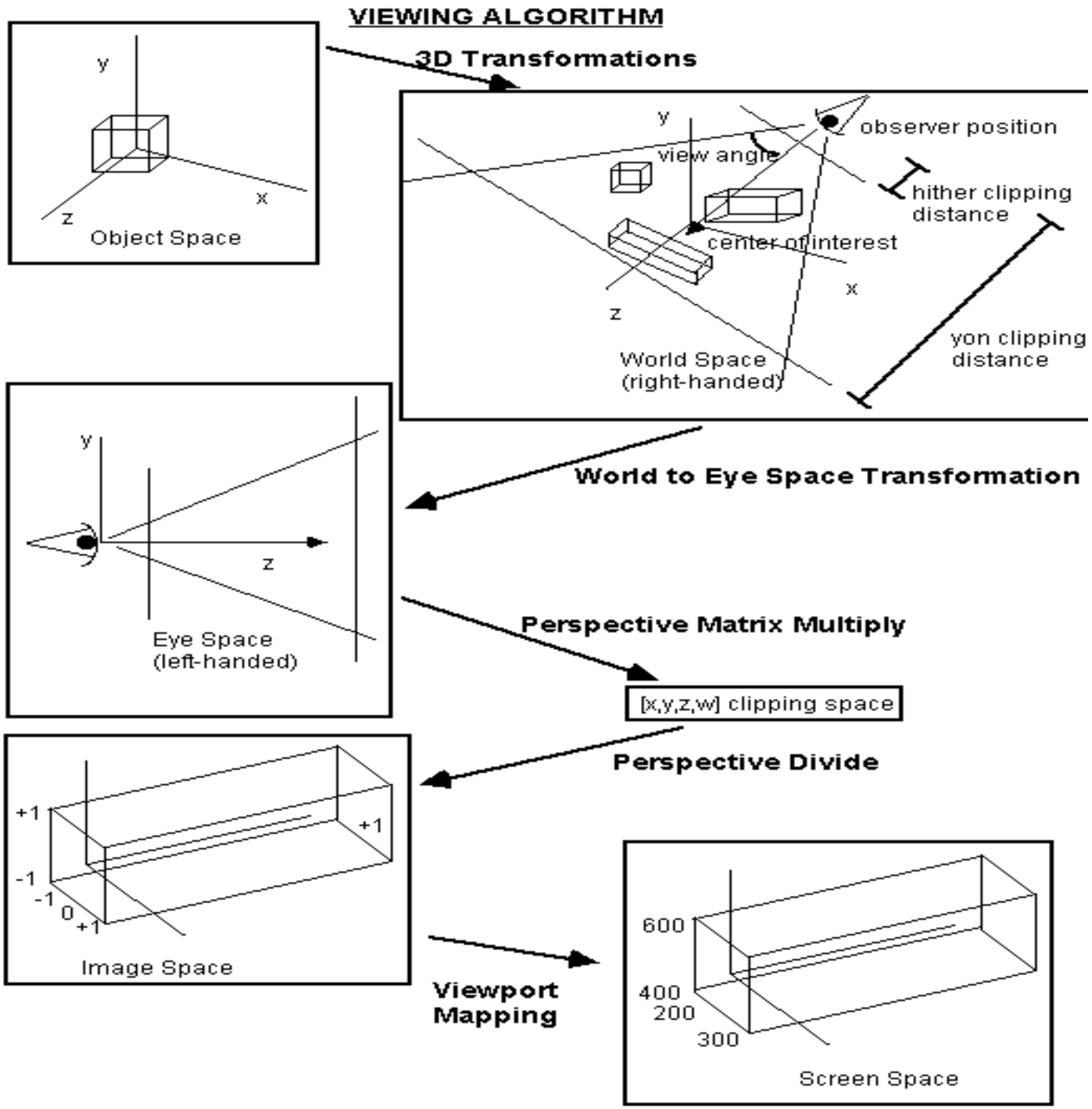
Flowchart of the 3D Viewing Pipeline



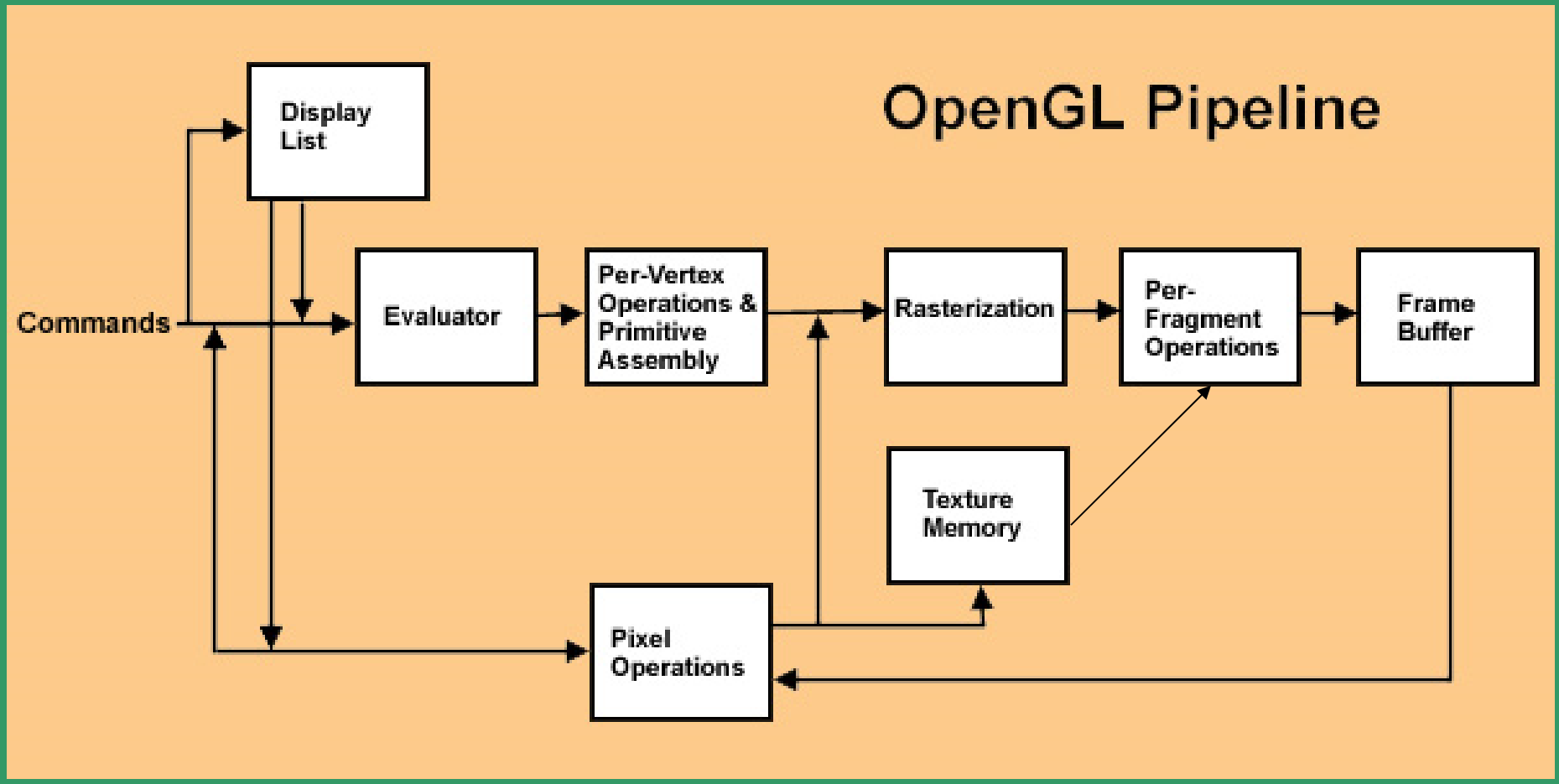
Cont…

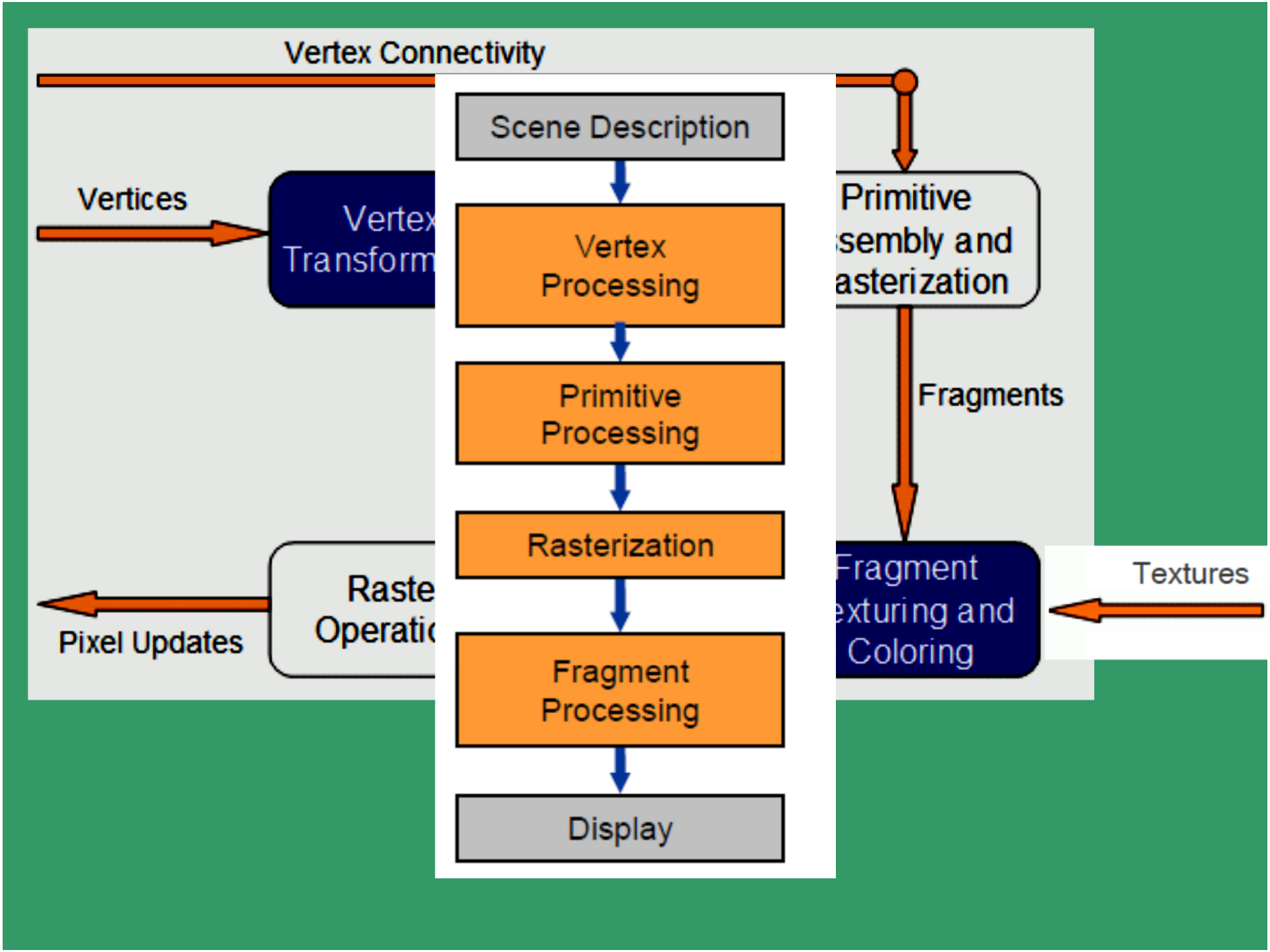

Eye Space
Perspective Transformation /Projection
Image Space
Remove Hidden Surfaces
Shade, Texture
Map to Viewport/ Draw to screen
Device Coordinates

The Computer Graphics Pipeline Viewing Process



The OPENGL PIPELINE




Vertex Connectivity
Scene Description
Vertices
Vertex Transform
Vertex Processing
Primitive Processing
Rasterization
Fragment Processing
Display
Primitive ssembly and asterization
Fragments
Fragment exturing and Coloring
Textures
Raste Operati
Pixel Updates

Application Stage: Here, the programmer (you!) talks to OpenGL. In this way, you have control over the following steps. Also in this stage, you give the GPU some triangles to draw.

Vertex Stage: Those triangles are defined by three vertices each. Here, the GPU arranges the vertices so that they are in the right locations on the screen.

Rasterization Stage: Here, the triangles' vertices are connected by pixels.

Fragment Stage: Here, the pixels are colored/textured.

Framebuffer Stage: Here, the pixels are written into graphics memory. Tests are applied to make sure they should go into memory (for example: if triangle A is behind triangle B, then we shouldn't draw any of triangle A's pixels!).

Screen Stage: Here, the memory on the GPU is displayed on the screen. It is only here that we get to see what happened.

All program statements/executions falls under the application stage. Some sophisticated calls let you program some stages (such as the vertex and fragment stages). These calls load shader programs to be executed instead of the fixed function pipeline.

Calls like "glEnable" and "glColor3f" alter the GPU's internal state. Calls like "glGetFloatv" and "glTexImage2D" get and retrieve data, and calls like "glVertex3f" and "glDrawArrays" draw data using the GPU's current state.

When you tell OpenGL to draw a triangle, the graphics pipeline begins with the vertex stage. OpenGL also provides functionality to draw quadrilaterals and some other stuff. Ultimately, the graphics driver breaks these down into individual triangles, a process called triangulation.

The application stage also handles setting up an OpenGL context, which tells the GPU that a given area of screen should be rendered to. The OpenGL context encompasses information about models, textures, lights, as well as data for the framebuffer(s).

Vertex Stage: - Transformation Matrices:

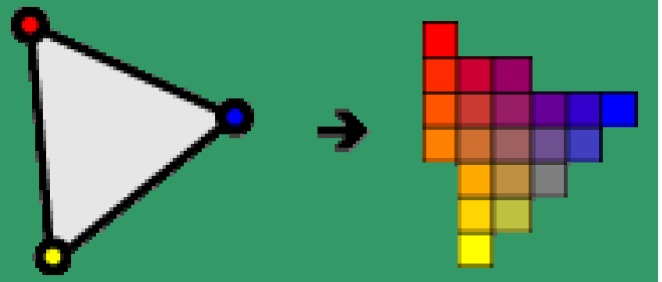
OpenGL concatenates the model matrix and the view matrix into the modelview matrix. Multiplying vertices by the modelview matrix transforms them directly from object space to eye space.

You use the model matrix to move locally defined objects around to globally defined positions. At the end of the model transform, the objects are defined in relation to a common origin that all objects share.

To transform to eye space, we multiply by the view matrix. At the end of the view transform, everything in the entire world is defined in relation to you, the observer/camera.

Next transform the vertices into a clip space, for clipping to be performed. To do this, multiply the transformed vertices by the projection matrix. OpenGL defines a volume called a frustum (rectangular prism) which defines everything you can see. The projection matrix typically maps this volume to a cube of side length 2.0, centered at the origin. This is called clip space, defined relative to the eye, and is in perspective.

After the perspective divide (1/w), we say that the vertices are in normalized device coordinates (NDC), or, inside the canonical view volume. The viewport transformation (2-D) is applied here or postponed.



Rasterization Stage :

The process of converting the three points (vertices) into a set of fragments, (which can be imagined as pixels with an arbitrary amount of extra information attached to them). A pixel has a position and a color; fragments can store, among other things, depth, normals, and texture coordinates, in addition to their position and color.

The rasterizer first finds the set of pixels the polygon covers, and creates a fragment at each (or often for a set of close) pixel's location. The data each fragment contains (color, depth, etc.) must be interpolated from the three vertices' values. This process is often done using barycentric coordinates.

From here, the fragments go on to the fragment stage, where their final attribute(s) are computed.

Fragment Stage:

Here, the fragments are prepared for entry into the framebuffer. This involves setting the final values for the fragment's final attributes. The most common fragment stage operation is to determine a color for the fragment.

It is almost always in the fragment stage that texturing, lighting, and other effects are done. The output of this stage is a fragment with more limited information.

In the fixed function pipeline, a pixel's final color depends on lights, OpenGL colors, and texture. If texturing is enabled, the fragment's color is multiplied by a per-pixel lookup from a texture (basically an image).

The fixed function pipeline supports a number of lights (at a minimum, and typically, 8), material properties and multi-texturing. When using shaders, far more advanced shading methods are possible!

Framebuffer Stage:

The new fragment must now determine whether its information should be written to the framebuffer. If a fragment passes a sequence of tests, its data is written into the framebuffer.

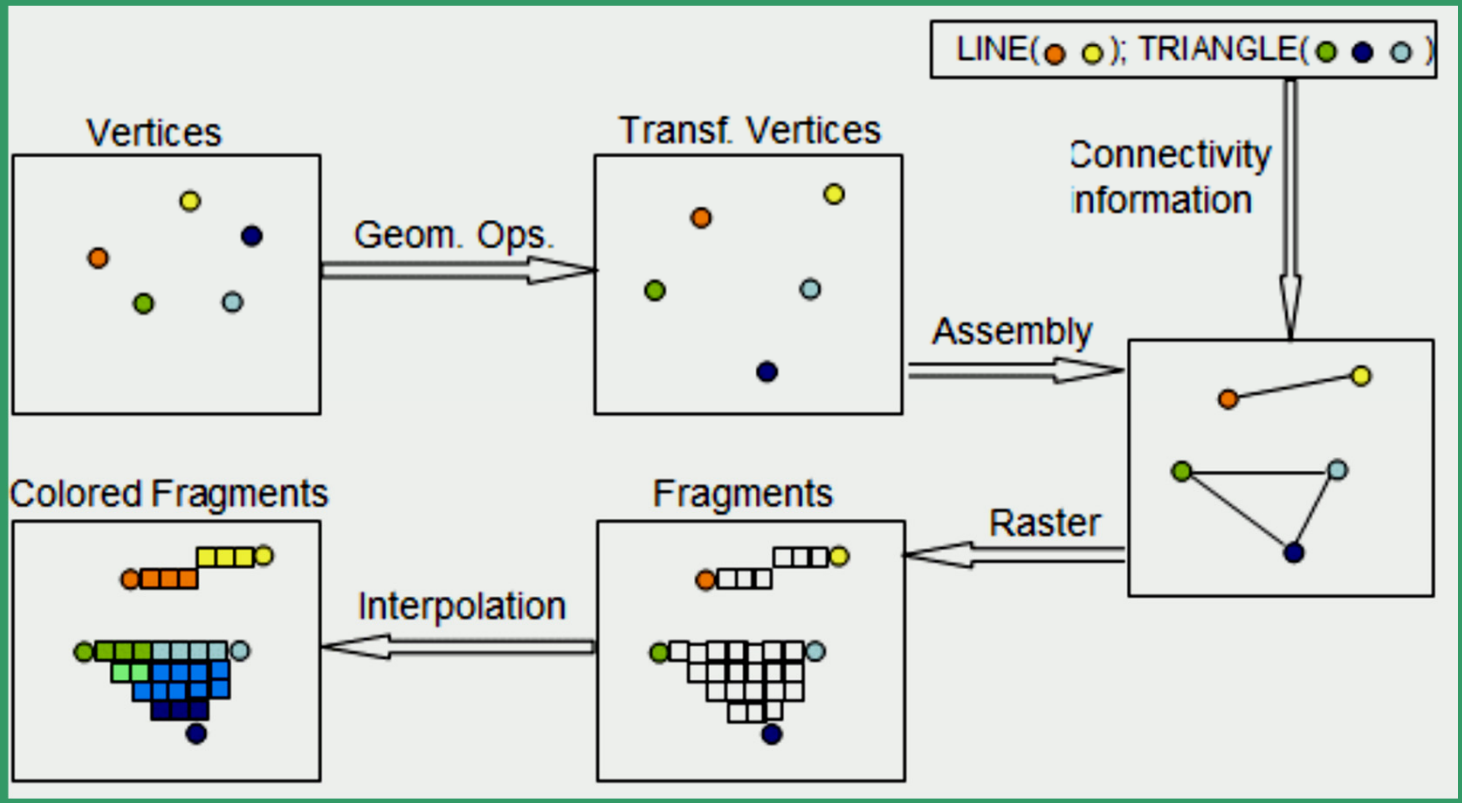
Colorbuffer stores the colors of each pixel. The depthbuffer (sometimes, z-buffer) stores depths.

In addition to the depth test, the fragment may need to pass the stencil, alpha, and/or other test.

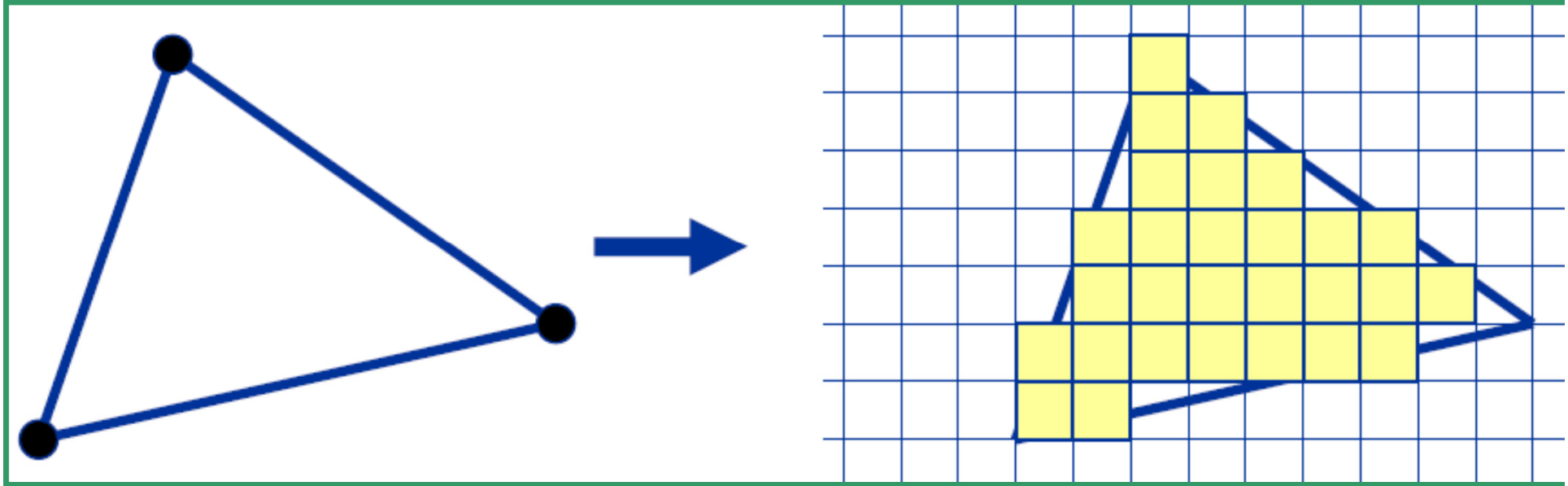
Screen Stage:

This is where we finally get to see what happened. The framebuffer contains color values, and these are drawn to the screen.

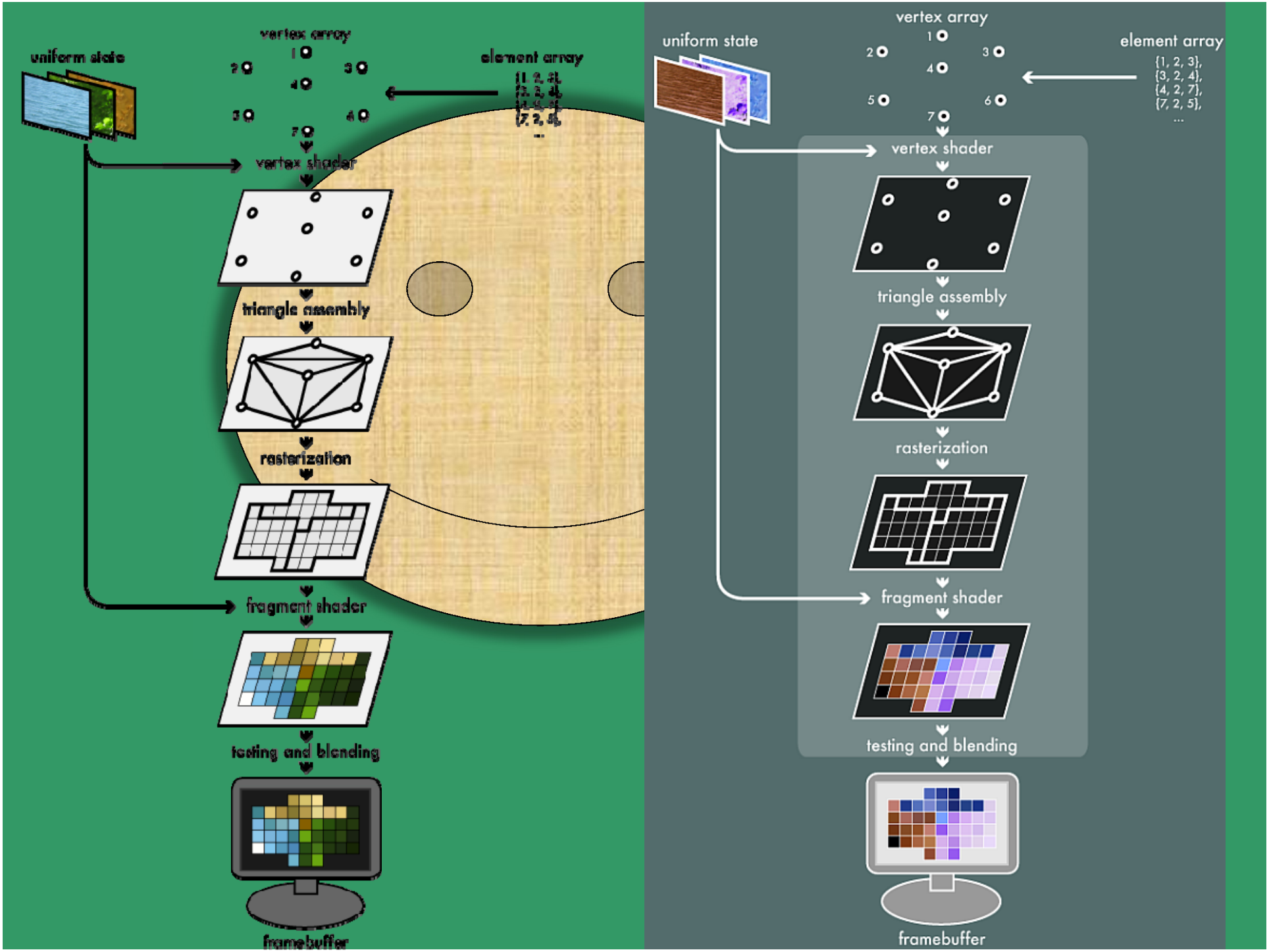
Overwhelmingly, most programs set up double-buffering, which uses two framebuffers, with only one displayed at a time. In a double-buffered context, the GPU draws to one framebuffer while the other framebuffer is being displayed. When the new framebuffer is ready, the screen flips, swapping the framebuffers. The GPU will now render into the other framebuffer while the new one is rendering. The cycle repeats.

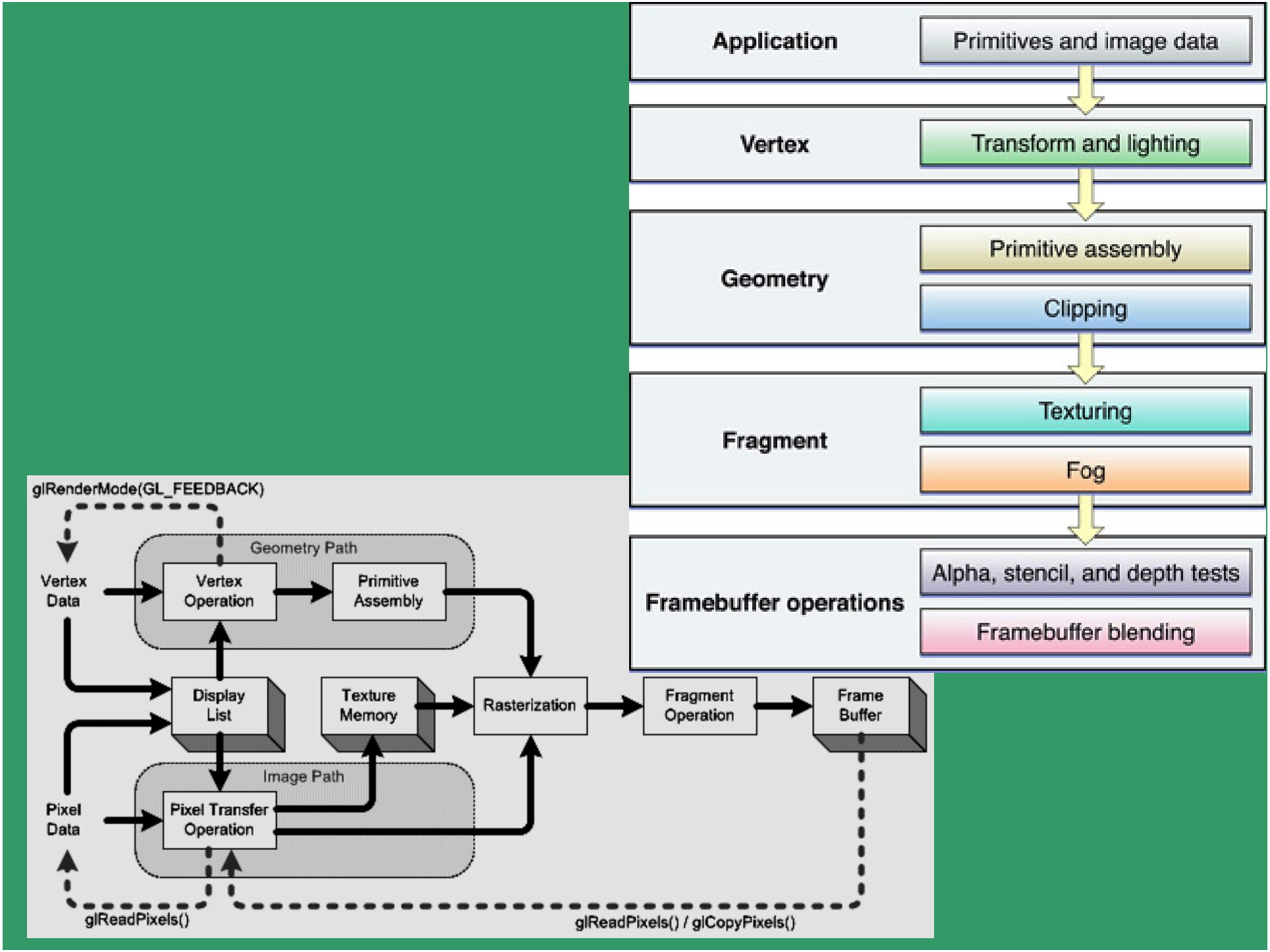


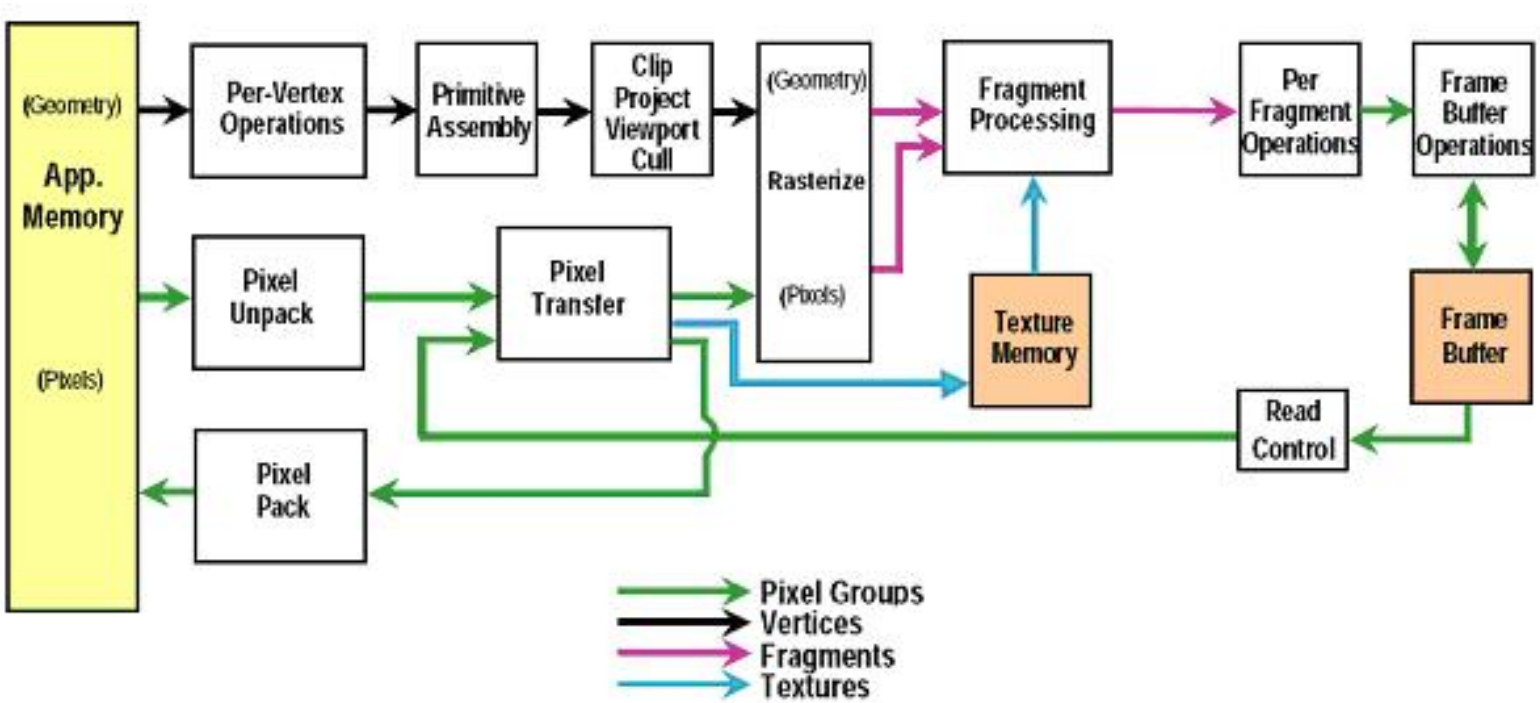
Rendering pipeline example

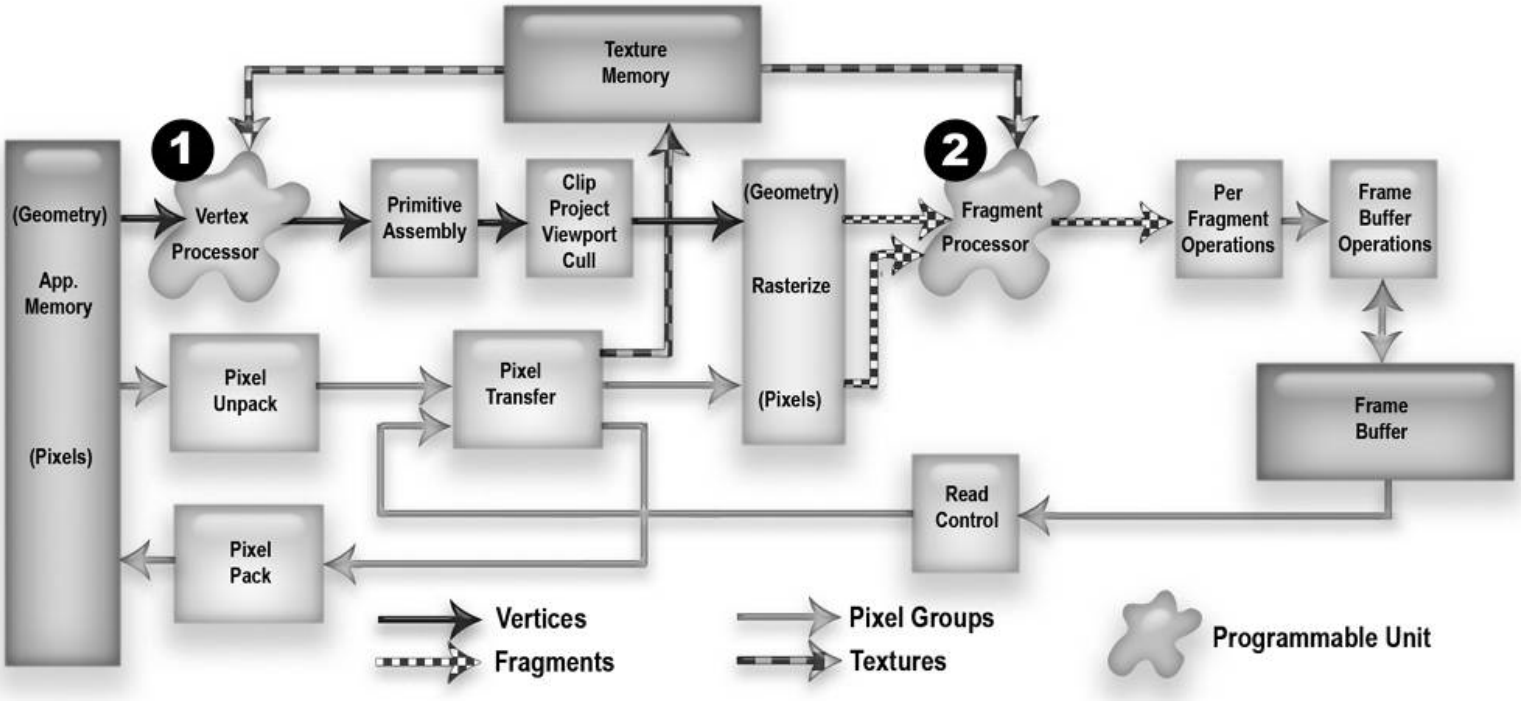


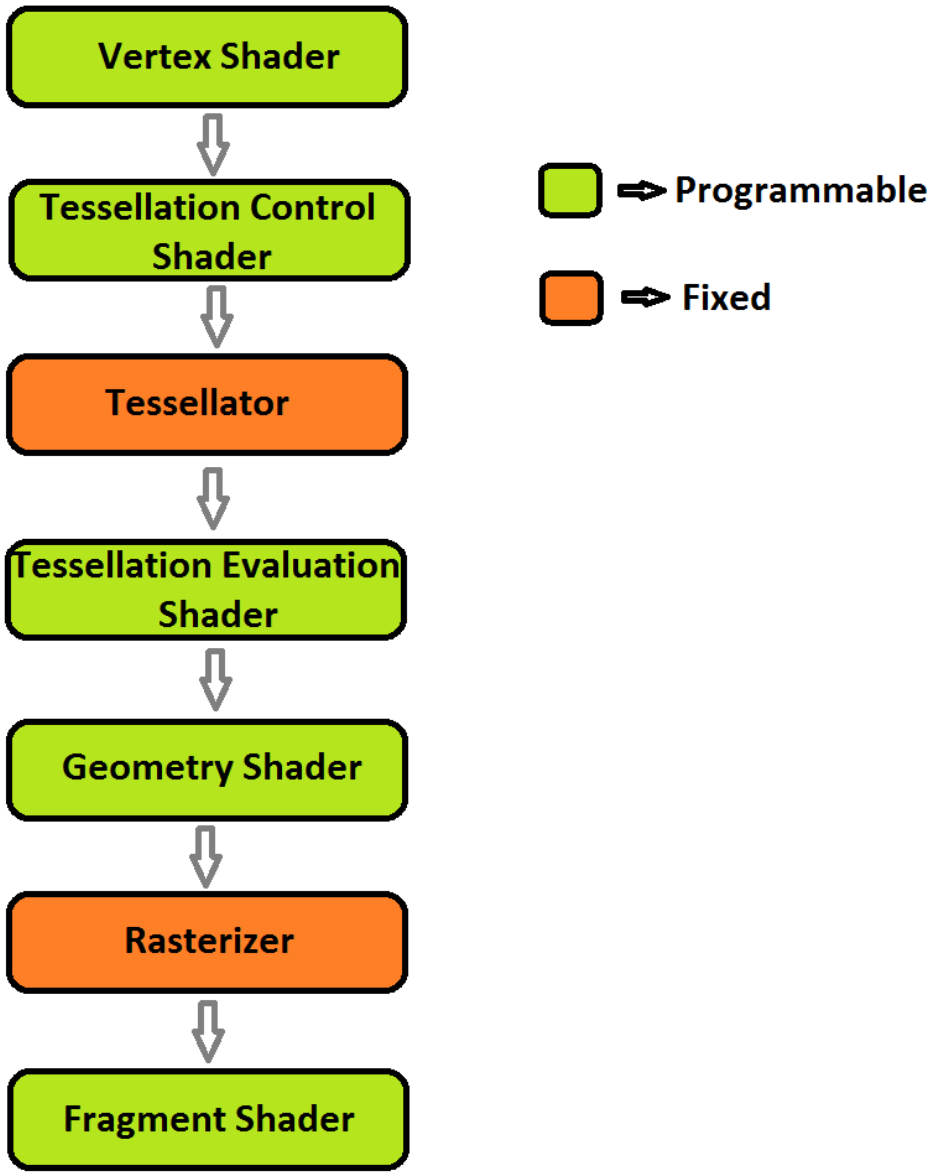
RASTERIZATION

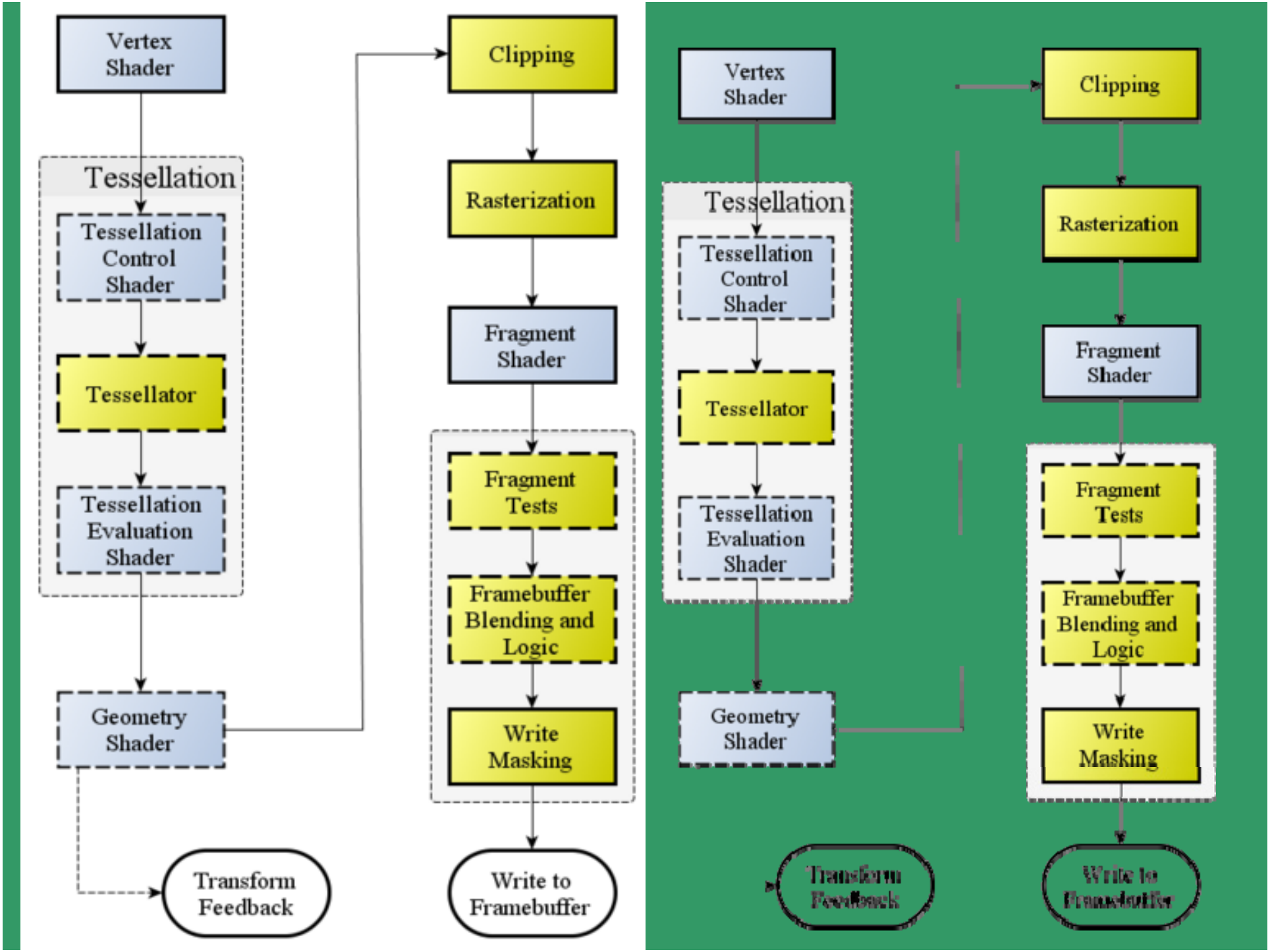

uniform state
vertex array
element array
vertex shader
triangle assembly
rasterization
fragment shader
testing and blending
framebuffer
uniform state
vertex array
element array
{1, 2, 3},
{3, 2, 4},
{4, 2, 7},
{7, 2, 5},
...
vertex shader
triangle assembly
rasterization
fragment shader
testing and blending
framebuffer

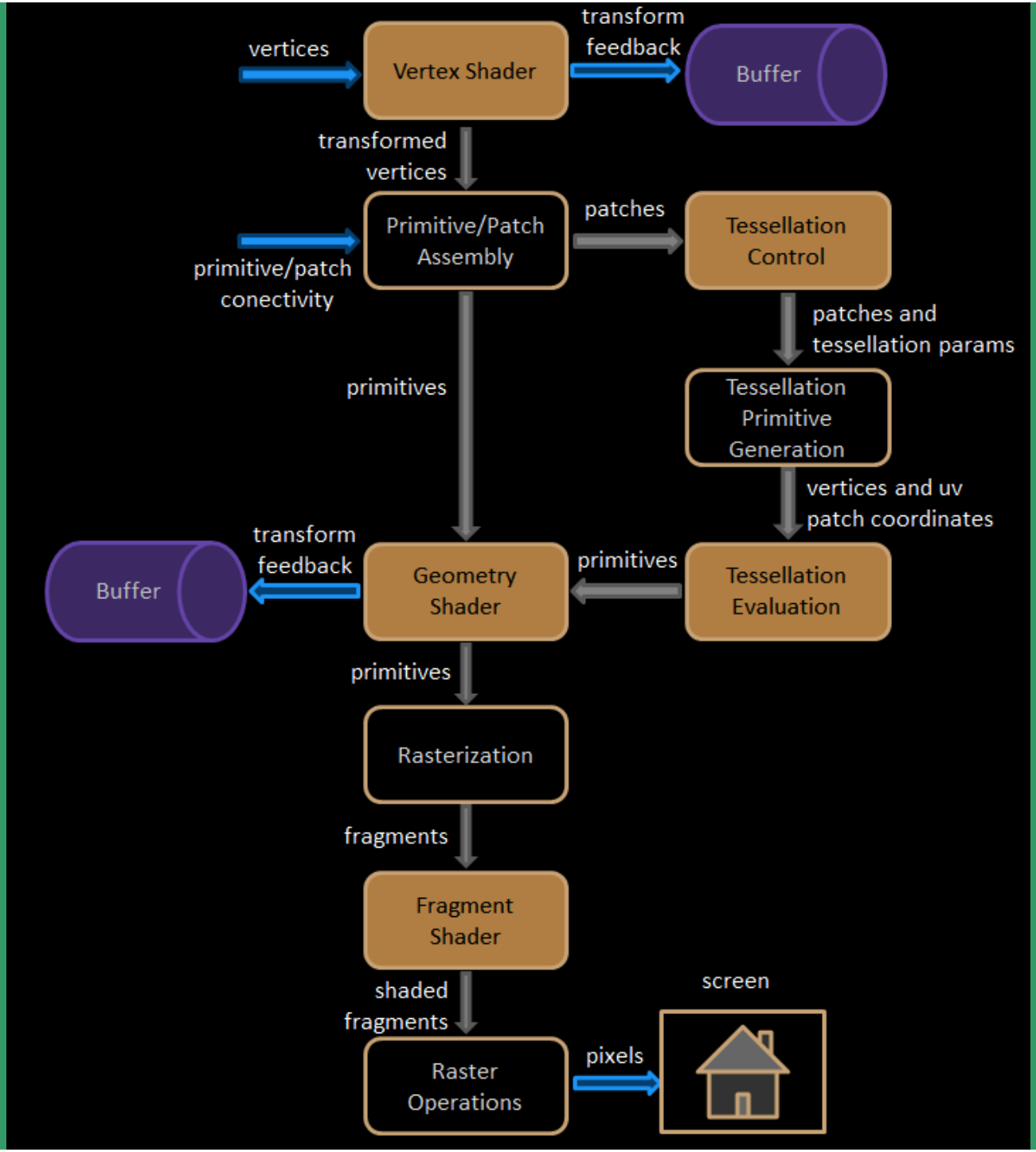

Application
Primitives and image data
Vertex
Transform and lighting
Geometry
Primitive assembly
Clipping
Fragment
Texturing
Fog
Framebuffer operations
Alpha, stencil, and depth tests
Framebuffer blending
glRenderMode(GL_FEEDBACK)
Geometry Path
Vertex Data
Vertex Operation
Primitive Assembly
Display List
Texture Memory
Rasterization
Fragment Operation
Frame Buffer
Image Path
Pixel Data
Pixel Transfer Operation
glReadPixels()
glReadPixels() / glCopyPixels()

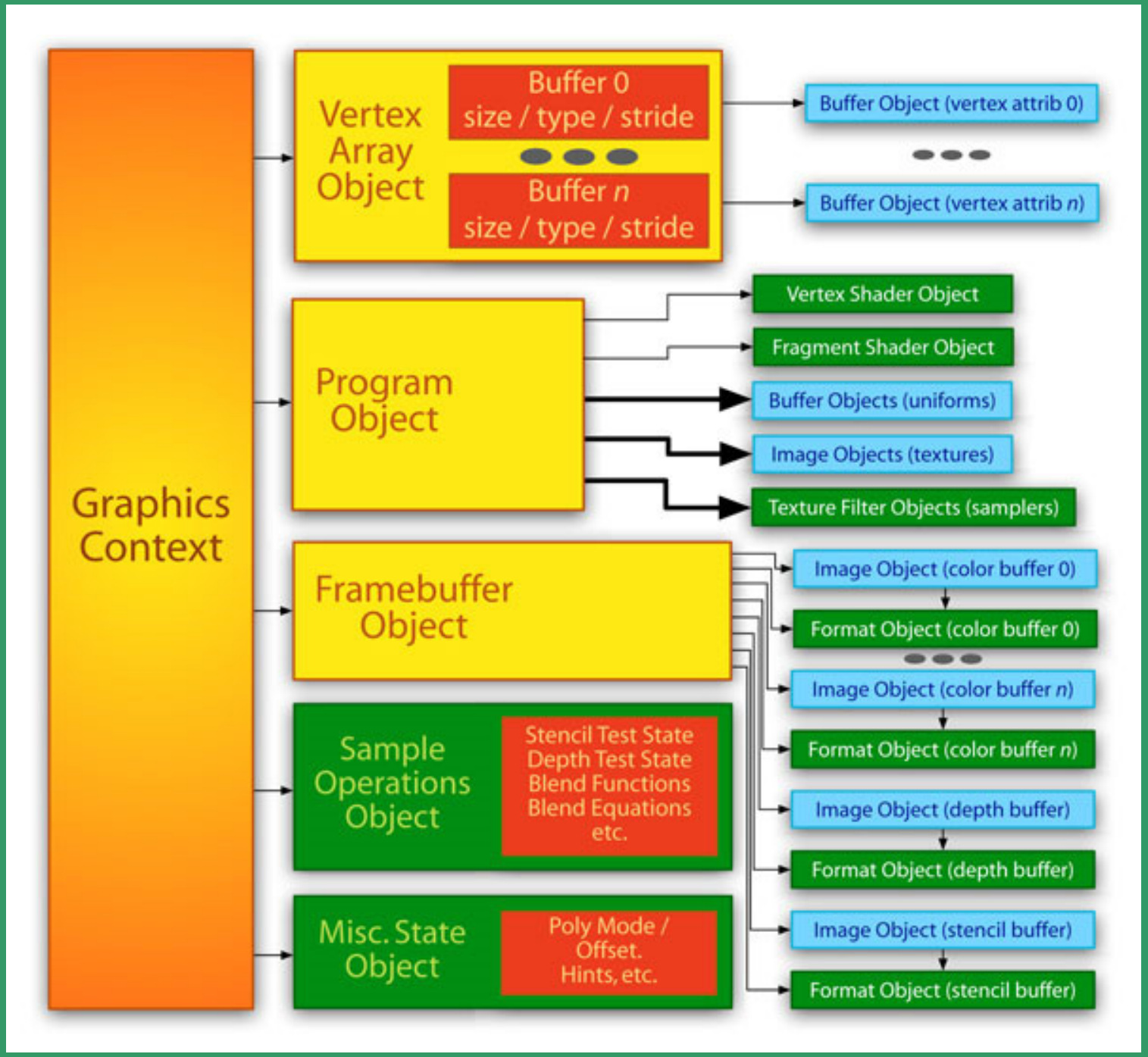

(Geometry)
App. Memory
(Pixels)
Per-Vertex Operations
Primitive Assembly
Clip Project Viewport Cull
(Geometry)
Rasterize
(Pixels)
Fragment Processing
Per Fragment Operations
Frame Buffer Operations
Pixel Unpack
Pixel Transfer
Texture Memory
Frame Buffer
Read Control
Pixel Pack
Pixel Groups
Vertices
Fragments
Textures


Texture Memory
1
2
(Geometry)
App. Memory
(Pixels)
Vertex Processor
Primitive Assembly
Clip Project Viewport Cull
(Geometry)
Rasterize
(Pixels)
Fragment Processor
Per Fragment Operations
Frame Buffer Operations
Pixel Unpack
Pixel Transfer
Frame Buffer
Read Control
Pixel Pack
Vertices
Fragments
Pixel Groups
Textures
Programmable Unit

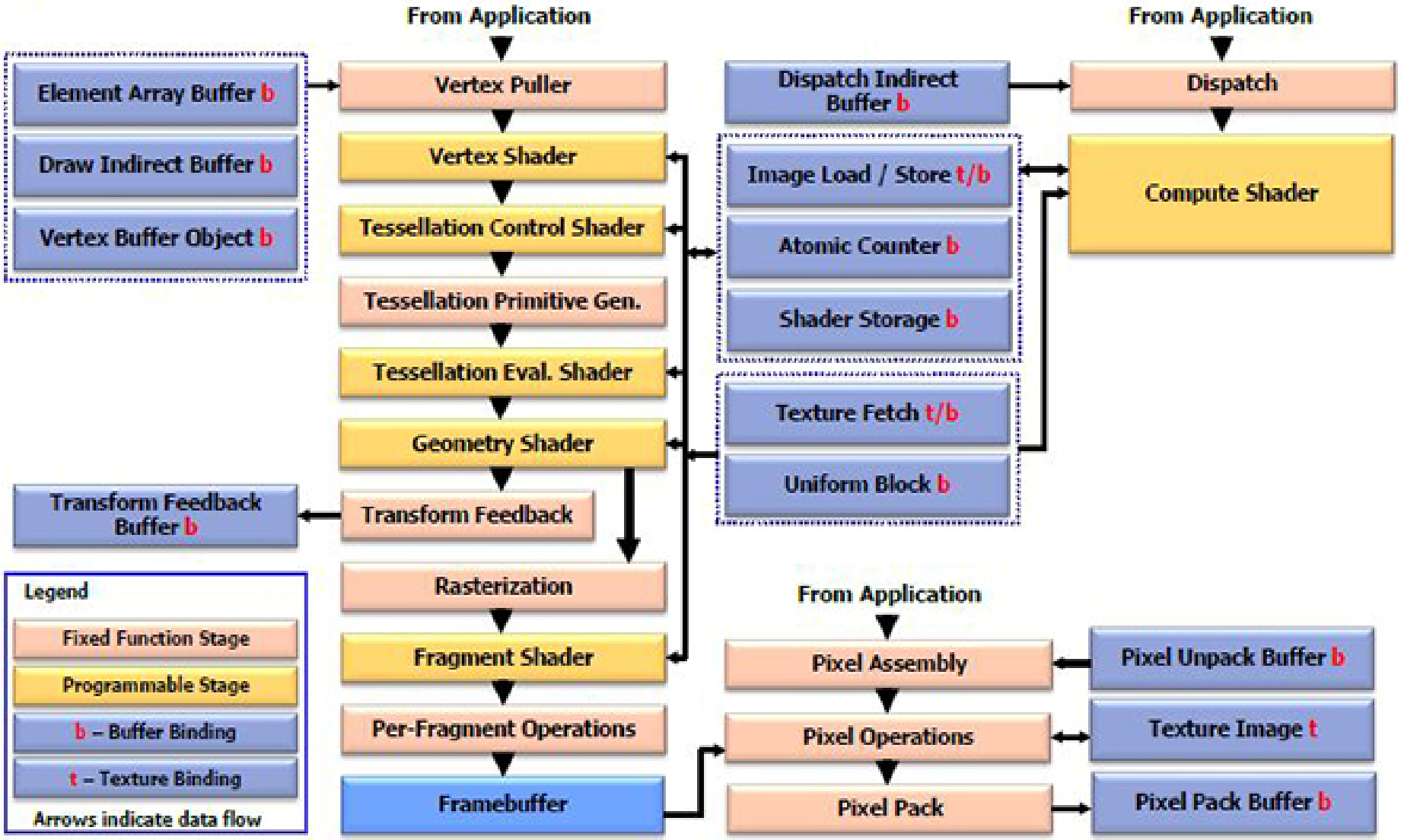

Vertex Shader
Tessellation Control Shader
Tessellator
Tessellation Evaluation Shader
Geometry Shader
Rasterizer
Fragment Shader
Programmable
Fixed


Vertex Shader
Tessellation
Tessellation Control Shader
Tessellator
Tessellation Evaluation Shader
Geometry Shader
Transform Feedback
Clipping
Rasterization
Fragment Shader
Fragment Tests
Framebuffer Blending and Logic
Write Masking
Write to Framebuffer
Vertex Shader
Tessellation
Tessellation Control Shader
Tessellator
Tessellation Evaluation Shader
Geometry Shader
Clipping
Rasterization
Fragment Shader
Fragment Tests
Framebuffer Blending and Logic
Write Masking


vertices
Vertex Shader
transform feedback
Buffer
transformed vertices
Primitive/Patch Assembly
primitive/patch conectivity
patches
Tessellation Control
patches and tessellation params
Tessellation Primitive Generation
vertices and uv patch coordinates
Tessellation Evaluation
primitives
Geometry Shader
transform feedback
Buffer
primitives
Rasterization
fragments
Fragment Shader
shaded fragments
Raster Operations
pixels
screen


Graphics Context
Vertex Array Object
Buffer 0 size / type / stride
Buffer n size / type / stride
Buffer Object (vertex attrib 0)
Buffer Object (vertex attrib n)
Program Object
Vertex Shader Object
Fragment Shader Object
Buffer Objects (uniforms)
Image Objects (textures)
Texture Filter Objects (samplers)
Framebuffer Object
Image Object (color buffer 0)
Format Object (color buffer 0)
Image Object (color buffer n)
Format Object (color buffer n)
Image Object (depth buffer)
Format Object (depth buffer)
Image Object (stencil buffer)
Format Object (stencil buffer)
Sample Operations Object
Stencil Test State
Depth Test State
Blend Functions
Blend Equations
etc.
Misc. State Object
Poly Mode / Offset.
Hints, etc.

OpenGL 4.3 with Compute Shaders



Things to know –
GLUT, GLU, GLX, GLEW, and WGL:

- Current Specifications (OpenGL 4.5)
- OpenGL 4.5 API Specification (updated February 2, 2015);
-
  - OpenGL 4.4
  - OpenGL 4.4 API Specification (updated March 19, 2014)

The OpenGL API is defined as a state machine. Almost all of the OpenGL functions set or retrieve some state in OpenGL. The only functions that do not change state are functions that use the currently set state to cause rendering to happen.

The state machine has a very large struct with many different fields. This struct is called the OpenGL context, and each field in the context represents some information necessary for rendering.

Objects in OpenGL are defined as a list of fields in this struct, that can be saved and restored. Binding an object to a target within the context causes the data in this object to replace some of the context's state.

The Camera Model

We specify our initial camera model by identifying the following parameters.

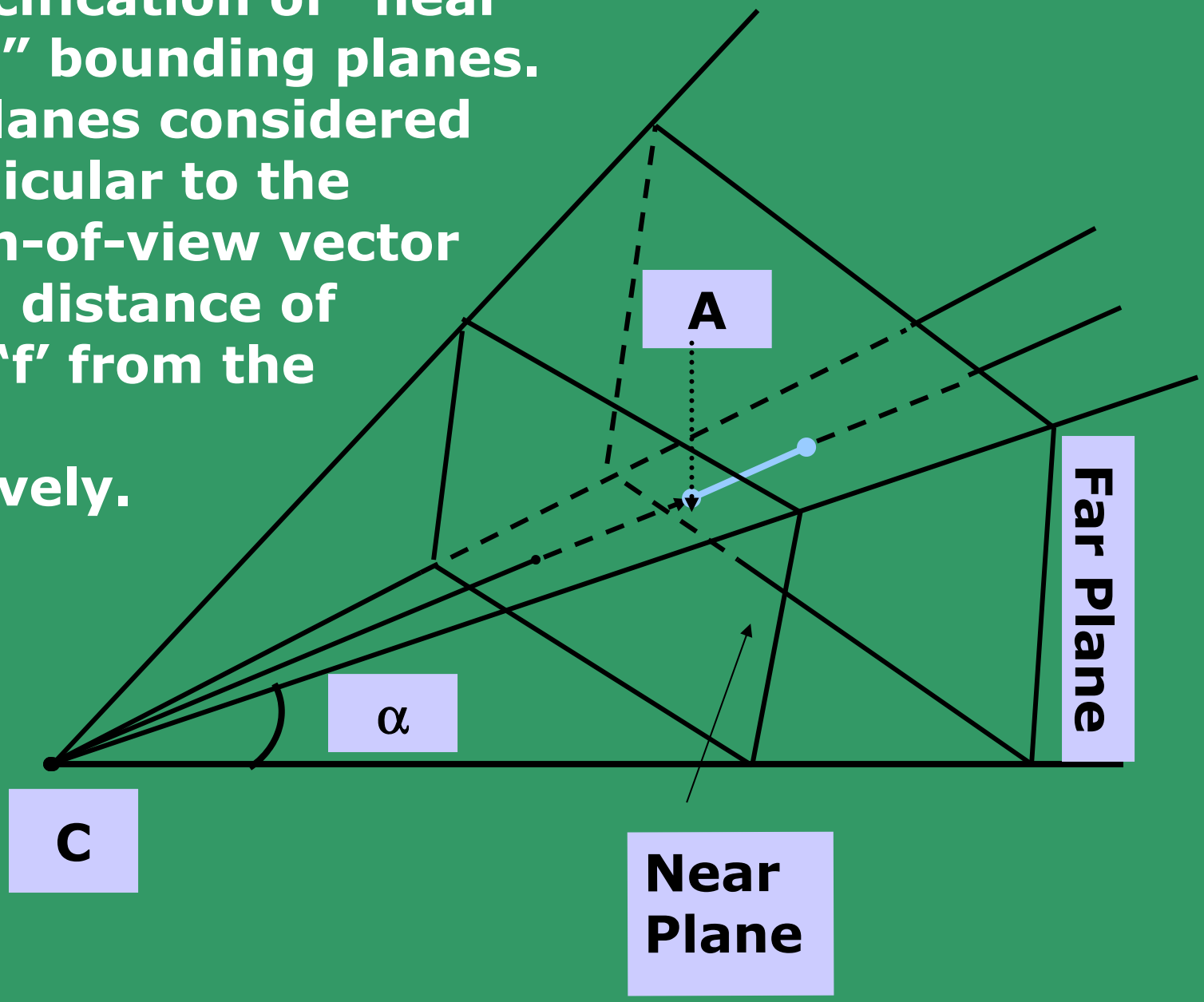
1. A scene, consisting of polygonal elements each represented by their vertices;

2. A point that represents the camera position: $C = [C_x, C_y, C_z]$;

3. A point that represents the "center-of attention" of the camera (i.e. where the camera is looking): $A = [A_x, A_y, A_z]$;

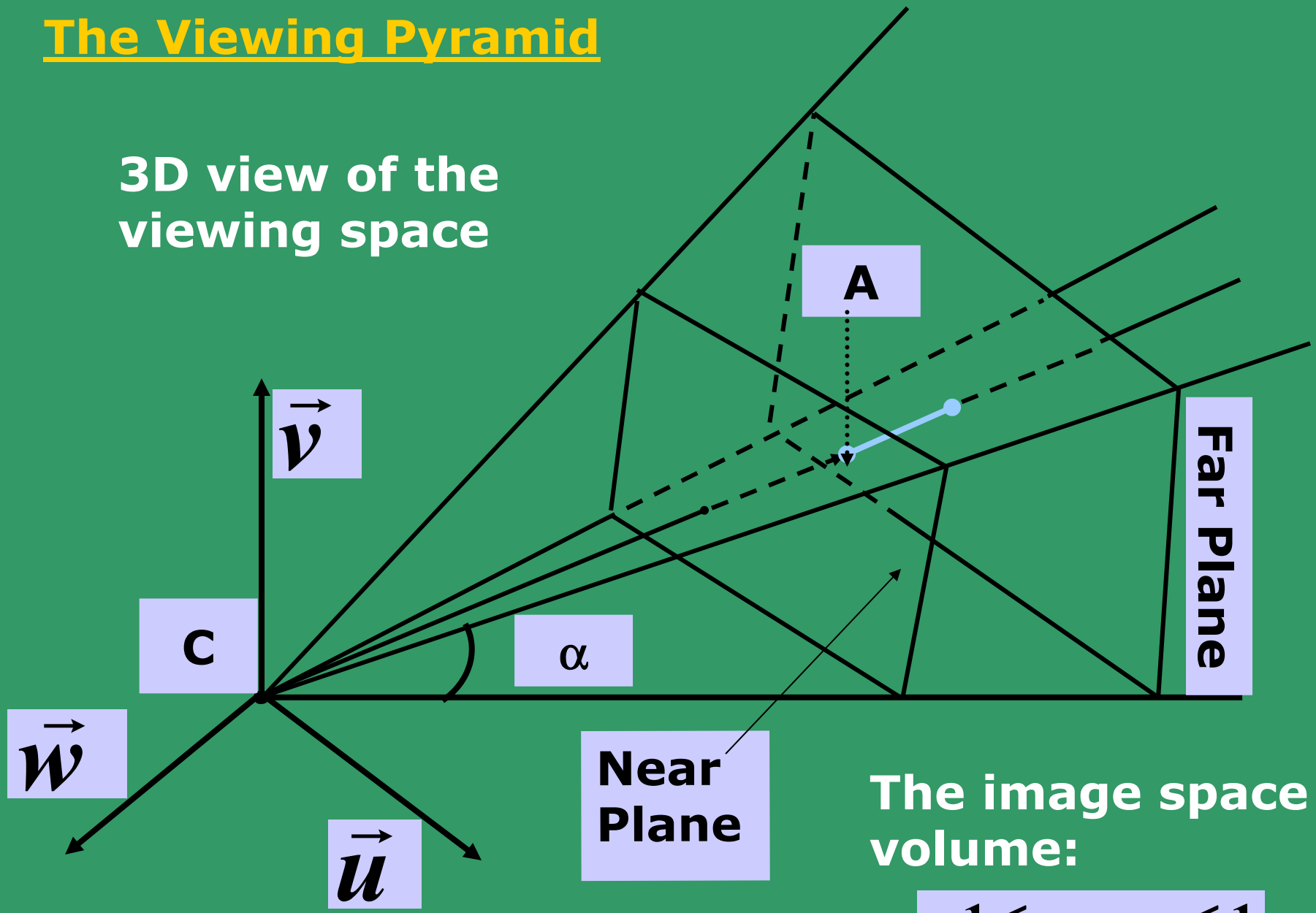
4. A field-of-view angle, α ,representing the angle subtended at the apex of the viewing pyramid.

The specification of “near” and “far” bounding planes. These planes considered perpendicular to the direction-of-view vector are at a distance of ‘n’ and ‘f’ from the camera, respectively.



The Viewing Pyramid

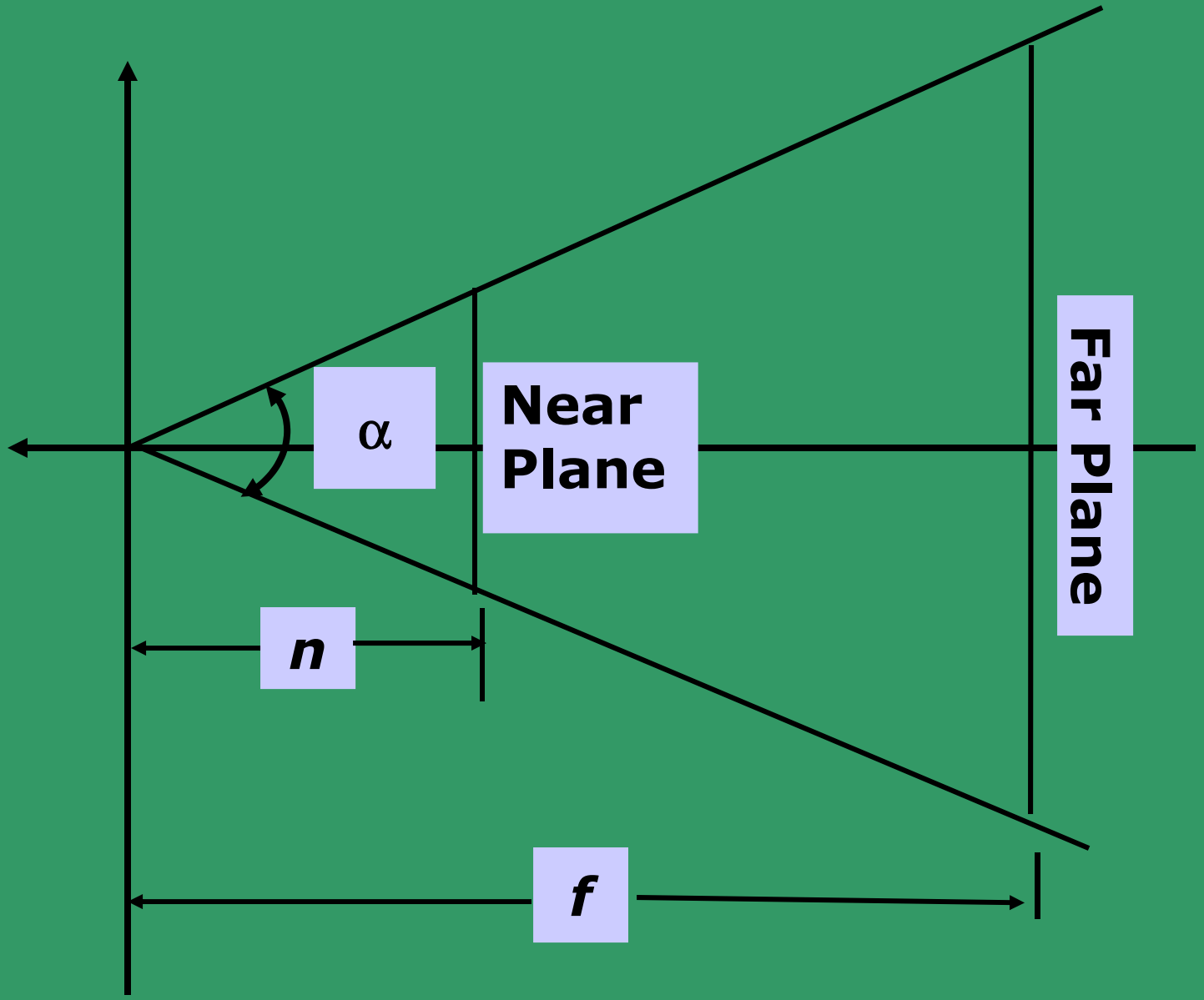
3D view of the viewing space



The image space volume:

$$-1 \le u, v, w \le 1$$

Side view of the viewing space

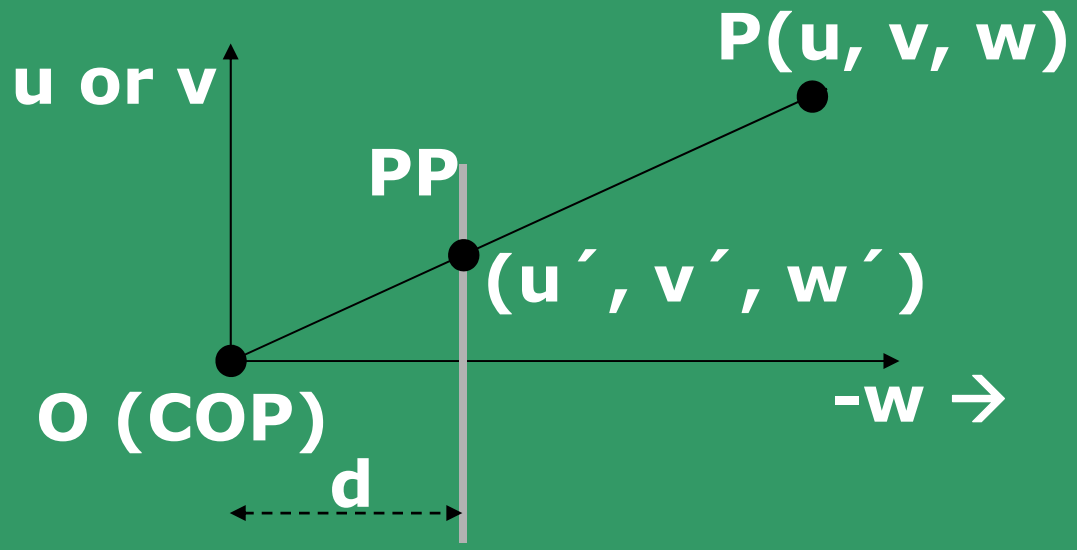


Derivation of the viewing transformation matrix, in terms of camera parameters:

$$(u,v,w) \rightarrow \left(\frac{d.u}{-w}, \frac{d.v}{-w}, -d\right) = \left(\frac{d.u}{-w}, \frac{d.v}{-w}, \frac{d.w}{-w}\right)$$

Thus,

$$(u, v, w, 1) \rightarrow (d.u, d.v, d.w, -w)$$



Express as transformation:

$$P_d = \begin{bmatrix} d & 0 & 0 & 0 \\ 0 & d & 0 & 0 \\ 0 & 0 & d & -1 \\ 0 & 0 & 0 & 0 \end{bmatrix} ;(\quad or \quad \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & -\frac{1}{d} \\ 0 & 0 & 0 & 0 \end{bmatrix})$$

$$\begin{bmatrix} u & v & w & 1 \end{bmatrix} \begin{bmatrix} d & 0 & 0 & 0 \\ 0 & d & 0 & 0 \\ 0 & 0 & d & -1 \\ 0 & 0 & 0 & 0 \end{bmatrix} = \begin{bmatrix} d.u & d.v & d.w & -w \end{bmatrix}$$

Transformation of the finite (truncated) viewing pyramid to the cube (CVV), -1 ≤ u, v, w ≤ 1.

The image space volume:

$$-1 \le u, v, w \le 1$$

Let us first analyze w-axis only.
Use the transformation matrix:

$$P = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & a & -1 \\ 0 & 0 & b & 0 \end{bmatrix};$$

such that,
(0, 0, -n)P → (0, 0, 1)
and
(0, 0, -f)P → (0, 0, -1)

Solve for parameters *a and b*, using the above equations:

From the constraints of the above two equations:

$$-an + b = n$$

$$and$$

$$-a.f + b = -f$$

The solution:

$$a = \frac{f + n}{f - n};$$

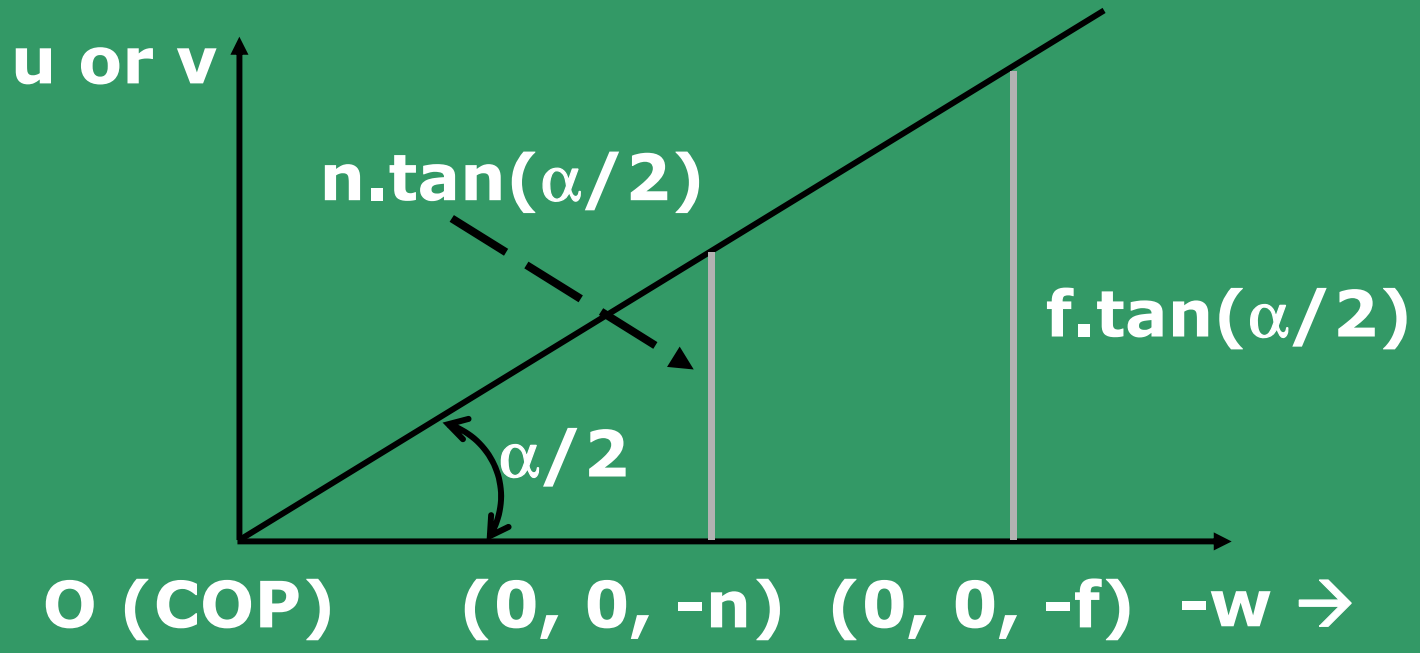
$$b = \frac{2f.n}{f - n}$$

Hence the transformation is:

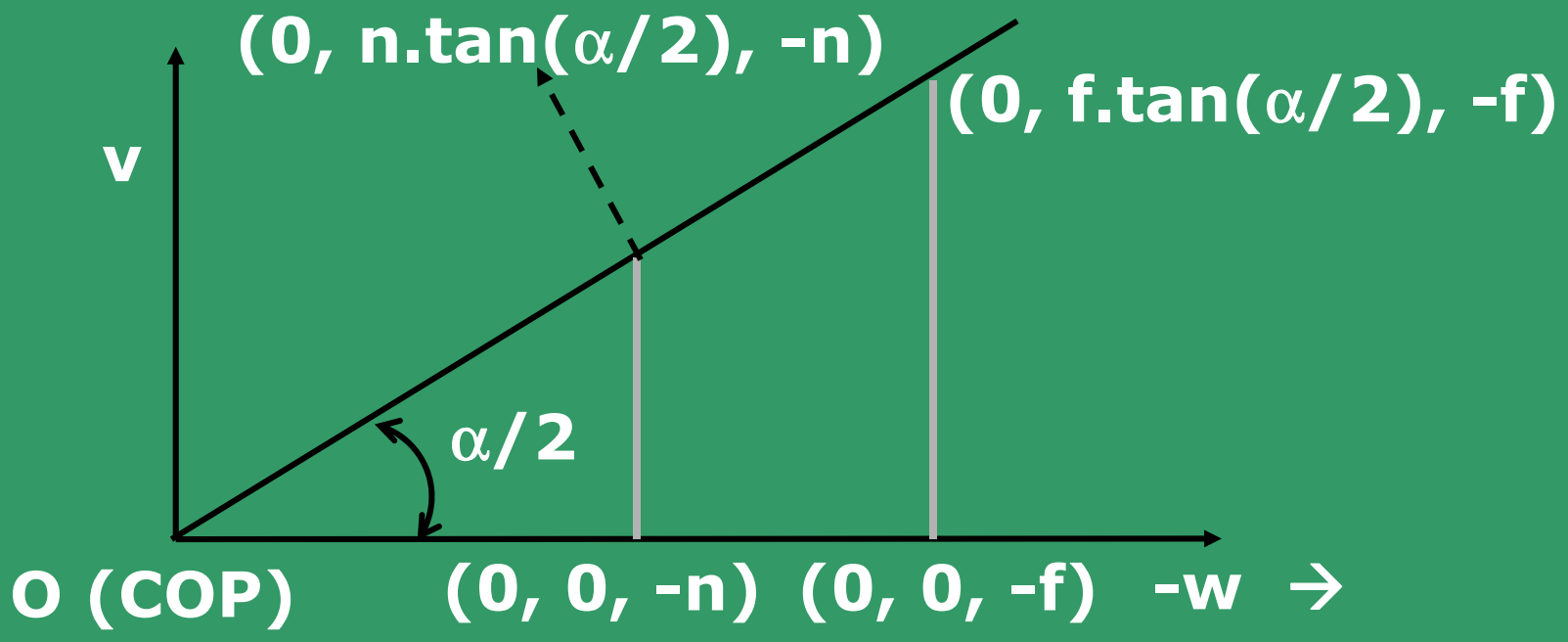
$$P = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \dfrac{f+n}{f-n} & -1 \\ 0 & 0 & \dfrac{2f.n}{f-n} & 0 \end{bmatrix}$$

What about u and v-axis transformations in the pyramid ?

u and v-axis transformations in the pyramid



Transformations for the two points are as follows:



$$\begin{bmatrix} 0 & \text{n.tan}(\alpha/2) & -\text{n} & 1 \end{bmatrix}.P$$

$$= \begin{bmatrix} 0 & \text{n.tan}(\alpha/2) & [-\text{n}\dfrac{\text{f}+\text{n}}{\text{f}-\text{n}} + \dfrac{2\text{nf}}{\text{f}-\text{n}}] & \text{n} \end{bmatrix}$$

$$= \begin{bmatrix} 0 & \text{n.tan}(\alpha/2) & \text{n} & \text{n} \end{bmatrix}$$

$$P = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \dfrac{f+n}{f-n} & -1 \\ 0 & 0 & \dfrac{2f.n}{f-n} & 0 \end{bmatrix}$$

Desired normalized 3-D coordinates for both the points: [0, 1, +/-1, 1].

$$\begin{bmatrix} 0 & \text{f.tan}(\alpha/2) & -\text{f} & 1 \end{bmatrix}.P$$

$$= \begin{bmatrix} 0 & \text{f.tan}(\alpha/2) & -\text{f}\dfrac{\text{f}+\text{n}}{\text{f}-\text{n}} + \dfrac{2\text{nf}}{\text{f}-\text{n}} & \text{f} \end{bmatrix}$$

$$= \begin{bmatrix} 0 & \text{f.tan}(\alpha/2) & -\text{f} & \text{f} \end{bmatrix}$$

Thus modify P to be:

$$P' = \begin{bmatrix} \cot(\alpha/2) & 0 & 0 & 0 \\ 0 & \cot(\alpha/2) & 0 & 0 \\ 0 & 0 & \frac{f+n}{f-n} & -1 \\ 0 & 0 & \frac{2f.n}{f-n} & 0 \end{bmatrix}$$

$$\begin{bmatrix} 0 & n.\tan(\alpha/2) & -n & 1 \end{bmatrix}.P'$$

$$= \begin{bmatrix} 0 & n & -n\frac{f+n}{f-n} + \frac{2nf}{f-n} & n \end{bmatrix}$$

$$= \begin{bmatrix} 0 & 1 & 1 & \end{bmatrix}$$

Its inverse has the form:

$$P^{-1} = \begin{bmatrix} \tan(\alpha/2) & 0 & 0 & 0 \\ 0 & \tan(\alpha/2) & 0 & 0 \\ 0 & 0 & 0 & \frac{f-n}{2fn} \\ 0 & 0 & -1 & \frac{f+n}{2fn} \end{bmatrix}$$

The Viewing Transformation Matrix

$$P_f = P_d . P$$

$$= \begin{bmatrix} d.\cot(\alpha/2) & 0 & 0 & 0 \\ 0 & d.\cot(\alpha/2) & 0 & 0 \\ 0 & 0 & \dfrac{d(f+n)-2fn}{f-n} & -d \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

$$P_d = \begin{bmatrix} d & 0 & 0 & 0 \\ 0 & d & 0 & 0 \\ 0 & 0 & d & -1 \\ 0 & 0 & 0 & 0 \end{bmatrix} ;(\ or \ \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & -\dfrac{1}{d} \\ 0 & 0 & 0 & 0 \end{bmatrix}) \quad P = \begin{bmatrix} \cot(\alpha/2) & 0 & 0 & 0 \\ 0 & \cot(\alpha/2) & 0 & 0 \\ 0 & 0 & \dfrac{f+n}{f-n} & -1 \\ 0 & 0 & \dfrac{2f.n}{f-n} & 0 \end{bmatrix}$$

$$\begin{bmatrix} u & v & w & 1 \end{bmatrix} \begin{bmatrix} d & 0 & 0 & 0 \\ 0 & d & 0 & 0 \\ 0 & 0 & d & -1 \\ 0 & 0 & 0 & 0 \end{bmatrix} = \begin{bmatrix} d.u & d.v & d.w & -w \end{bmatrix}$$

or using the regular expression of P_d

$$= \begin{bmatrix} \cot(\alpha/2) & 0 & 0 & 0 \\ 0 & \cot(\alpha/2) & 0 & 0 \\ 0 & 0 & \dfrac{d(f+n)-2fn}{d(f-n)} & -1 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

$$P_d = \begin{bmatrix} d & 0 & 0 & 0 \\ 0 & d & 0 & 0 \\ 0 & 0 & d & -1 \\ 0 & 0 & 0 & 0 \end{bmatrix} ;\left(\text{or} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & -\frac{1}{d} \\ 0 & 0 & 0 & 0 \end{bmatrix} \right) \quad P = \begin{bmatrix} \cot(\alpha/2) & 0 & 0 & 0 \\ 0 & \cot(\alpha/2) & 0 & 0 \\ 0 & 0 & \dfrac{f+n}{f-n} & -1 \\ 0 & 0 & \dfrac{2f.n}{f-n} & 0 \end{bmatrix}$$

$$[u \quad v \quad w \quad 1] \begin{bmatrix} d & 0 & 0 & 0 \\ 0 & d & 0 & 0 \\ 0 & 0 & d & -1 \\ 0 & 0 & 0 & 0 \end{bmatrix} = [d.u \quad d.v \quad d.w \quad -w]$$

End of Lectures on

3D Viewing – Projection Transformations and Viewing Pipeline