OpenGL ES 1.1 Compliant High Performance GPU

- **OpenGL ES**
  - → is a subset of the OpenGL computer graphics rendering API for rendering 2D and 3D.
  - → is a software interface to graphics hardware. The interface consists of a **set of procedures and functions that allow a programmer to specify the objects** and **operations involved in producing high-quality graphical images.**
  - → is designed for embedded systems like smartphones and tablets.
  - → is cross-language and cross-platform
  - → is free of charge.
  - → Versions :-
    - → For Programmable Hardware:- OpenGL ES 2.x
      OpenGL ES 3.x
    - → For Fixed-Function Hardware:- OpenGL ES 1.x

- **OpenGL ES version 1.1**
  - → is defined relative to the OpenGL 1.5 specification and emphasizes hardware acceleration of the API.
  - → is designed for **Fixed-Function Hardware**.
  - →Implementers (Examples) :-

Vendor	Product	Operating System
ARM	Mali T720, Mali T760	Linux 3.4
	Mali T678 Mali T658	Android 4
Intel	Intel® HD Graphics for Intel® Atom™ Processor Z3000 series	Android 4.2.2
	Intel® HD Graphics for Intel® Celeron™ Processor N and J series	
Apple	Apple iPhone 5s, Apple iPhone 4 Apple iPad Air, Apple iPad mini	iOS 7 "OpenGL ES-CM 1.1"
QUALCOMM	MSM 8610	Android 4.3
	MSM 8626 MSM 8974	Android 4.2
NVIDIA	Tegra 3	Android 4.1
	NVIDIA GeForce GTX 590 NVIDIA GeForce GTX 580	Windows

OpenGL ES 1.1 Compliant High Performance GPU

- **OpenGL ES 1.1 :-**

  → **Features :-**

Feature	Description
Draw Arrays	Draws (**Points, Line Strips, Line Loops, Separate Lines, Triangle strip, Triangle Fans, Separate Triangles**) from array vertices.
Enhanced point sprites and point sprite arrays	- specify texture coordinates that are interpolated across the point. - The Point Size Array extension permits an array of point sizes instead of a fixed input point size.
User-defined clip planes	permit for efficient early culling of non-visible polygons increasing performance and saving power
Enhanced texture processing	including a minimum of two multi-textures and texture combiner functionality for effects such as bump-mapping and per-pixel lighting
Draw Texture	defines a mechanism for writing pixel rectangles from one or more textures to a rectangular region of the screen which is useful for fast rendering of background paintings and 2D framing elements in games.
Auto mipmap generation	can offload the application from having to generate mip-levels.
Buffer objects	provide a mechanism that clients can use to allocate, initialize and render from memory. Buffer objects can be used to store vertex array and element index data.
State queries	This enables OpenGL ES to be used in a sophisticated, layered software environment
Direct Control	Provides direct control over the fundamental operations of 3D & 2D graphics such as transformation matrix, lighting equations, and antialiasing methods.
New Core Additions and Profile Extensions	There are core additions, required profile extensions, and optional profile extensions for the Common & Common-Lite profiles

OpenGL ES 1.1 Compliant High Performance GPU

- **OpenGL ES 1.1 :-**

  → **Profiles :-**

Common-Lite (CL) Profile	Common (CM) Profile
- **Supports only commands taking fixed-point arguments.** - **Examples:-** ClipPlanex, Color4x.	- **Supports commands taking floating-point arguments and commands taking fixed-point arguments.** - **Examples:-** ClipPlanef, Color4f ClipPlanex, Color4f
- does not support floating-point data (format FLOAT) in vertex arrays or images in client memory.	**- support floating-point data and fixed-point data in vertex arrays or images in client memory.**
- GL states are stored in fixed-point format. **- Applications must call the GetFixedv command, or the equivalent fixed-point versions of enumerated queries, such as GetLightxv, to query such state.**	**- GM states are stored in fixed-point format or floating-point format.** **- Applications use the call command according to the state types.**
- Computations are performed in fixed-point format, but the implementations are free to use floating-point computations if they wish.	**- Computations are performed in floating-point format**
- **Computations Specifications :-** **1) Individual results of Fixed Point operations are accurate within +/- 2^-15.** **2) The maximum representable magnitude of Fixed-Point number is at least** **- 2^15 for "Positional or normal coordinates"** **- 2^10 for color or texture coordinates** **- 2^15 for other types.** **- 0^0 =1, x.0 = 0.x = 0, 1.x = x.1 = x** **3) Transformation matrix uses the same type of data inputs.**	- **Computations Specifications :-** **1) Individual results of FP operations are accurate to about 1 part in 10^5.** **2) The maximum representable magnitude of FP number is at least** **- 2^32 for "Positional or normal coordinates"** **- 2^10 for color or texture coordinates** **- 2^32 for other types.** **- 0^0 =1, x.0 = 0.x = 0, 1.x = x.1 = x** **3) Transformation matrix uses the same type of data inputs.**

Notes:-

1) CL profile applications run without any modification on the GM

2) Apple products use OpenGL ES 1.1 GM

OpenGL ES 1.1 Compliant High Performance GPU

- **OpenGL ES 1.1 :-**
  - → **Core Additions and Extensions :-**
    - → some extended functionality that is drawn from a set of OpenGL ES–specific extensions to the full OpenGL specification.
    - → Each extension is added to the profile as either a **core addition** or a **profile extension.**
    - → **Core additions** commands and tokens do not include extension suffixes in their names while **profile extension.**
    - → **Profile extensions** are further divided into required (mandatory) and optional extensions

Extension Name	Common-Lite (CL) Profile	Common (CM) Profile
OES byte coordinates	core addition	core addition
OES fixed point	core addition	core addition
OES single precision	core addition	n/a
OES matrix get	core addition	core addition
OES read format	required extension	required extension
OES compressed paletted texture	required extension	required extension
OES point size array	required extension	required extension
OES point sprite	required extension	required extension
OES matrix palette	optional extension	optional extension
OES draw texture	optional extension	optional extension

→ **OpenGL Compliance:-**

-To label an implementation as an OpenGL ES compliant implementation, the implementation must pass a set of **conformance Tests.**

-**Conformance Testing** Adopters can download and run the conformance tests and if the implementation passes, they can advertise and promote the product as being compliant; using the OpenGL ES logos and trademarks under a royalty-free license.

OpenGL ES 1.1 Compliant High Performance GPU

→ Application Programming Interface(API):-

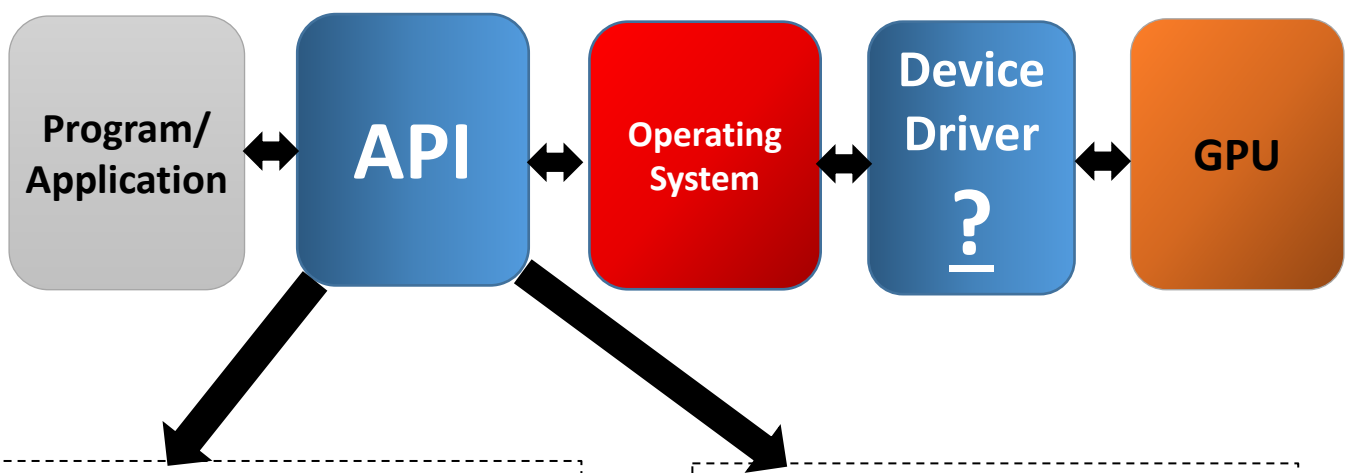
- consists of **data types/structures, constants and functions** that **other program or application** can use in your code to access the functionality of that external component.

→ Application Binary Interface(ABI):-

- is the compiled version of an API (or as an API on the machine-language level). When you write source code, you access the library though an API. Once the code is compiled, your application accesses the binary data in the library through the ABI. The ABI defines the structures and methods that your compiled application will use to access the external library (just like the API did), only on a lower level.
-**An ABI isn't necessarily something you will explicitly provide unless you are expecting people to interface with your code using assembly.**

→ Device Driver:- provides a **software interface** to hardware devices, enabling operating systems and other computer programs to access hardware functions without needing to know precise details of the hardware being used.

- When a calling program invokes a **routine** in the driver, the driver issues commands to the device. Once the device sends data back to the driver, the driver may invoke routines in the original calling program.
-Drivers are hardware-dependent and operating-system-specific.
- Extension (.DS)

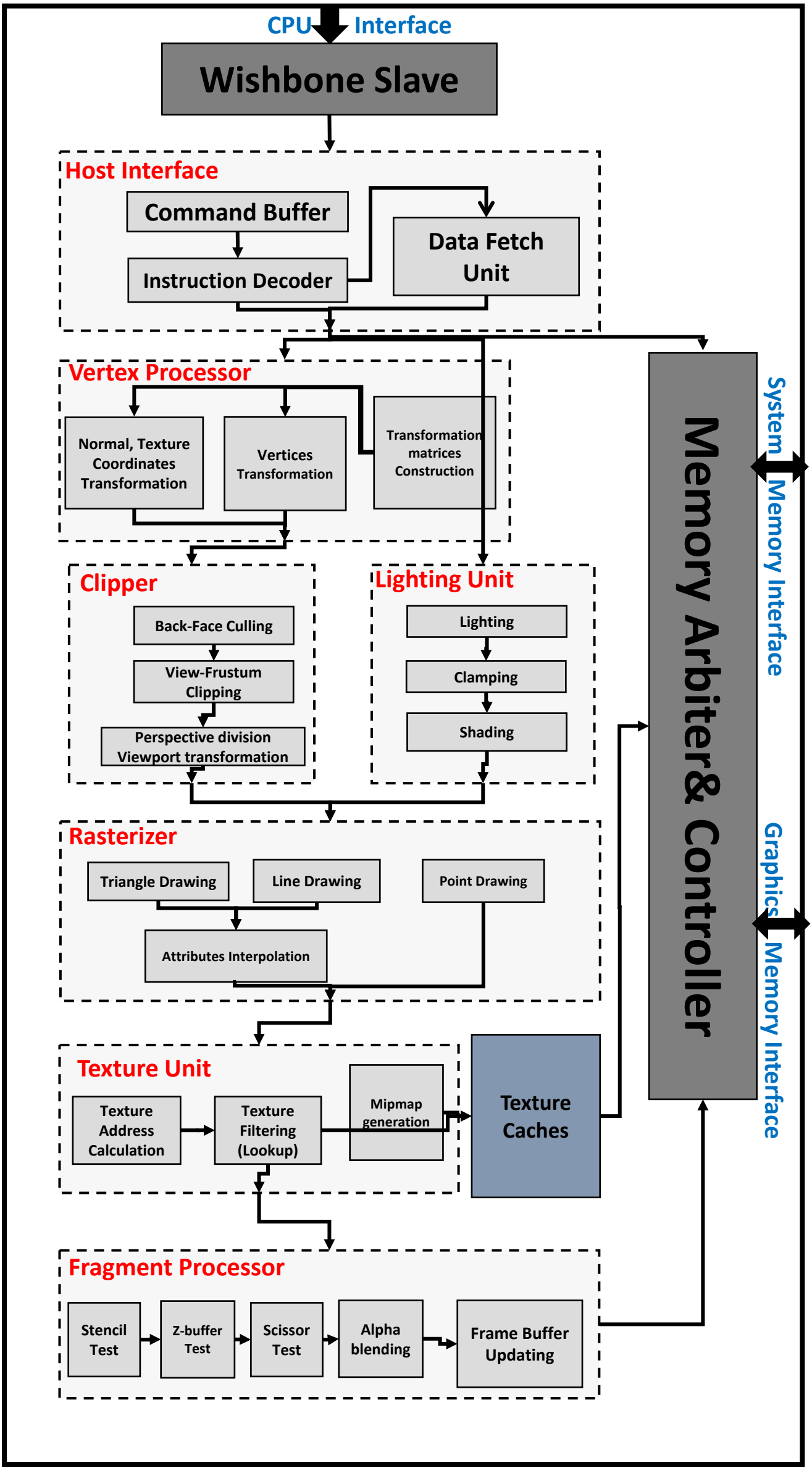


OpenGL ES 1.1 Headers

<GLES/gl.h> OpenGL ES 1.1 Header File.
<GLES/glext.h> OpenGL ES 1.1 Extension Header File.
<GLES/glplatform.h> OpenGL ES 1.1 Platform-Dependent Macros.
<GLES/egl.h> EGL Legacy Header File for OpenGL ES 1.1

OpenGL32 Dynamic Library (.DLL)

Function Na...	Address	Relative Address	Ordinal
glClearAccum	0x000007ff453...	0x000db790	18 (0x12)
glClearColor	0x000007ff453...	0x000db7f0	19 (0x13)
glClearDepth	0x000007ff453...	0x000db850	20 (0x14)
glClearIndex	0x000007ff453...	0x000db7c0	21 (0x15)
glClearStencil	0x000007ff453...	0x000db820	22 (0x16)
glClipPlane	0x000007ff453...	0x000dc090	23 (0x17)
glColor3b	0x000007ff453...	0x000dac80	24 (0x18)
glColor3bv	0x000007ff453...	0x000dac90	25 (0x19)


CPU Interface
Wishbone Slave
Host Interface
Command Buffer
Instruction Decoder
Data Fetch Unit
Vertex Processor
Normal, Texture Coordinates Transformation
Vertices Transformation
Transformation matrices Construction
Clipper
Back-Face Culling
View-Frustum Clipping
Perspective division Viewport transformation
Lighting Unit
Lighting
Clamping
Shading
Rasterizer
Triangle Drawing
Line Drawing
Point Drawing
Attributes Interpolation
Texture Unit
Texture Address Calculation
Texture Filtering (Lookup)
Mipmap generation
Texture Caches
Fragment Processor
Stencil Test
Z-buffer Test
Scissor Test
Alpha blending
Frame Buffer Updating
Memory Arbiter& Controller
System Memory Interface
Graphics Memory Interface

Registers

Name	Address	Size (bits)	FORMAT
CONTROL REGISTER	**0X 0000**	32	INTEGER
Vertices Coordinates :-			
POINT_1 - X – COORD	**0X000 4**	32	**Single-Precision Floating Point**
POINT_1 - Y – COORD	**0X 0008**	32	
POINT_1 - Z – COORD	**0X 000C**	32	
POINT_2 - X – COORD	**0X0010**	32	
POINT_2 - Y – COORD	**0X 0014**	32	
POINT_2 - Z – COORD	**0X0018**	32	
POINT_3 - X – COORD	**0X 001C**	32	
POINT_3- Y – COORD	**0X0020**	32	
POINT_3 - Z – COORD	**0X0024**	32	
Vertices Attributes (Color, Normal, Texture Coordinate) :-			
POINT_1_Color	**0X002 8**	32	**RGBA**
POINT_2_Color	**0X 002C**	32	
POINT_3_Color	**0X 0030**	32	
POINT_1_X_Normal	**0X00234**	32	**Floating point (we may use one normal per triangle)**
POINT_1_Y_Normal	**0X 0038**	32	
POINT_1_Z_Normal	**0X 003C**	32	
POINT_1_X_Normal	**0X00240**	32	
POINT_1_Y_Normal	**0X 0044**	32	
POINT_1_Z_Normal	**0X 0048**	32	
POINT_1_X_Normal	**0X004C**	32	
POINT_1_Y_Normal	**0X 0050**	32	
POINT_1_Z_Normal	**0X 0054**	32	
POINT_1_U_Coord.	**0X0058**	32	**Floating point**
POINT_1_V_Coord.	**0X 005C**	32	
POINT_2_U_Coord.	**0X 0060**	32	
POINT_2_V_Coord.	**0X0064**	32	
POINT_3_U_Coord.	**0X 0068**	32	
POINT_3_V_Coord.	**0X 006C**	32	

Registers

Name	Address	Size (bits)	FORMAT
Vertex processing stage uniforms:-			
FRAME_Buffer_Address (Final Colors Buffer)	**0X0070**	32	**Integer**
Screen Width (Nx)	**0X 0074**	32	
Screen Height (Ny)	**0X 0078**	32	
World_Space_left_side (L)	**0X007C**	32	**Floating-Point**
World_Space_right_side (R)	**0X 0080**	32	
World_Space_top_side (T)	**0X0084**	32	
World_Space_bottom_side (B)	**0X 0088**	32	
World_Space_near_side (n)	**0X008C**	32	
World_Space_far_side (f)	**0X0090**	32	
Camera_Position_X (**Ex)**	**0X0094**	32	**Floating-Point**
Camera_Position_Y (**Ey)**	**0X 0098**	32	
Camera_Position_Z (**Ez)**	**0X009C**	32	
Camera_Direction_X (**Gx)**	**0X00A0**	32	
Camera_Direction_Y (**Gy)**	**0X00A4**	32	
Camera_Direction_Z **(Gz)**	**0X00A8**	32	
Camera_Up_X (**Tx)**	**0X00AC**	32	
Camera_Up_Y (**Ty)**	**0X00B0**	32	
Camera_Up_Z **(Tz)**	**0X00B4**	32	

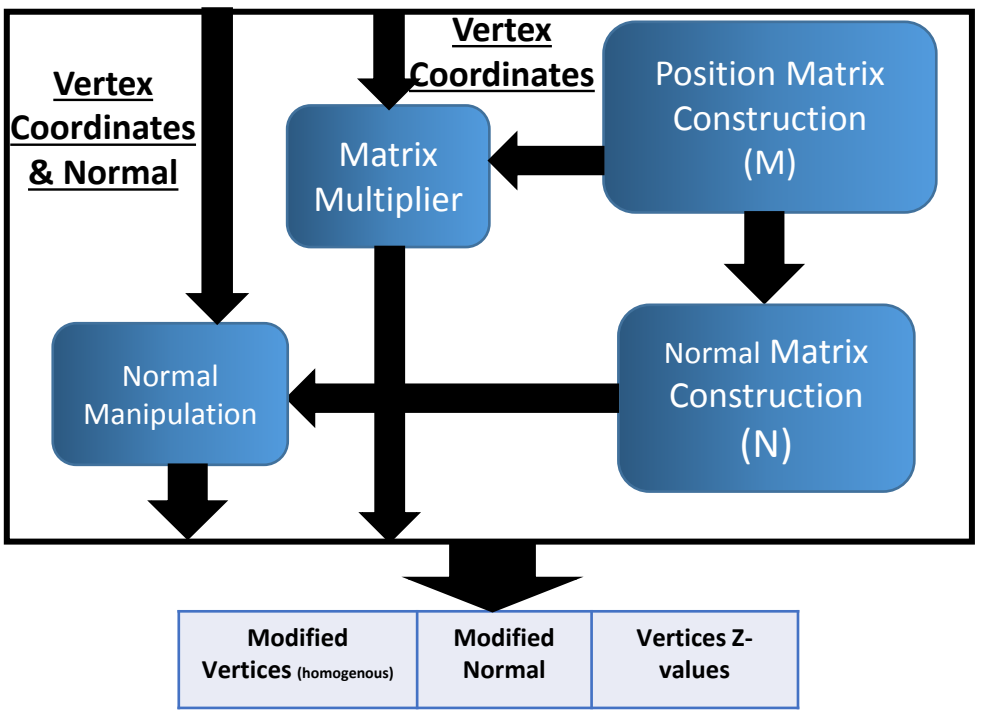
Registers

Name	Address	Size (bits)	FORMAT
Fragment processing stage uniforms:-			
Light Source Position_X	**0X00B8**	32	**Floating-Point**
Light Source Position_Y	**0X00BC**	32	
Light Source Position_Z	**0X00C0**	32	
Light Source Color	**0X00C4**	32	**RGBA**
Texture_Base_Address	**0X00C8**	32	**Integer**
Texture width	**0X 00CC**	32	
Texture height	**0X00D0**	32	
Stencil_Base_Address	**0X 00D4**	32	**Integer**
Stencil_reference_value	**0X00D8**	8	**Integer (the remaining 24-bits are zeros)**
Stencil_Input_value	**0X0DC**	8	
Z-Buffer-Base Address	**0X00E0**	32	**Integer**
Scissor-ref-X-Coord.	**0X00E4**	32	**Integer**
Scissor-ref-Y-Coord.	**0X00E8**	32	
Scissor-Frame-width	**0X00EC**	32	
Scissor-Frame-height	**0X00F0**	32	
Surface-Ambient-Coefficient (Ka)	**0X00F4**	8	**Fixed-Point**
Diffuse Coefficient (Kd)		8	
Specular Coefficient (Ks)		8	
Ambient-Light-Intensity (Ia)		8	
Shadow-Coefficient	**0X00F8**	32	

Registers

Control Register		
Field-Name	**Bits-number**	**Description**
Operation	**[0]→[4]**	[0]→ Draw Line [1] → Draw Triangle [2] → Draw Bezier curve [3] → Draw Circle [4] → Draw Ellipse
Dimension	**[5]**	0→ 2D 1→ 3D
Color-Key Enable	**[6]**	Rendering image with transparency
Coloring Method	**[7]**	Flat or Gradient
Bezier Fill	**[8]**	0→ inside 1→ outside
Texture Enable	**[9]**	--
Blending Enable	**[10]**	--
Scissor Enable	**[11]**	--
Stencil Enable	**[12]**	--
Shading-Model	**[13]→[14]**	--
Normal type	**[15]**	0→ per-vertex 1→ per-triangle
Number of light sources	**[16]→[18]**	--
Reserved-bits	**[19]→[23]**	--
FIFO Size	**[24] →[31]**	--

Vertex Processing stage



Modified Vertices (homogenous)	Modified Normal	Vertices Z-values

Attrite Registers

Vertices Coordinates (V_1,V_2, V_3)

Vertices Normal (N_1, N_2, N_3)

Uniform Registers

- **Eye Location** (Ex, Ey, Ez)
- **Eye Direction** (Gx, Gy, Gz)
- **Up-vector** (Tx, Ty, Tz)

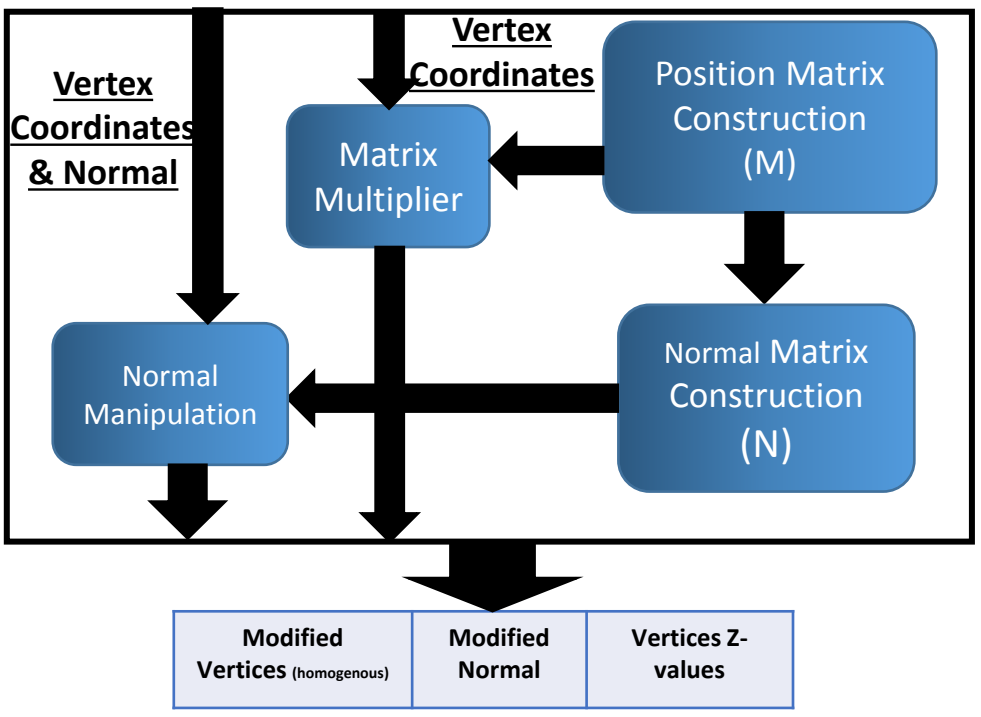
World Space [l, r] , [t, b], [n, f]

Screen Space [N_x, N_y]

Notes :-

1) we support 2D & 3D.
2) input vertices are in world-space
3) till now, we not support draw objects (meshes or stripes).
4) we support single -normal/triangle & normal/vertex.
5) we use “perspective projection”
6) the view-port transformation & division are delayed after the culling/clipping stage.
- to have:- make all transformations before division
7) camera looking in the –z direction with his head pointing in the y-direction.
8) the view volume is bounded by **[l, r] , [t, b], [n, f]**
9) the screen space: X € [-0.5 , n_x-0.5]
Y € [-0.5 , n_y- 0.5]
10) The project plane is the near plane (Z=n).

Vertex Processing stage



Attrite Registers

Vertices Coordinates (V_1,V_2, V_3)

Vertices Normal (N_1, N_2, N_3)

Uniform Registers

-Eye Location (Ex, Ey, Ez)
-Eye Direction (Gx, Gy, Gz)
-Up-vector (Tx, Ty, Tz)

World Space
[l, r] , [t, b], [n, f]

Screen Space
[N_x, N_y]

Coordinates Matrix Construction: (M)-

If (3D object)

$$M = M_{cam} \times M_{per}$$

Else

$$M = M_{cam.}$$

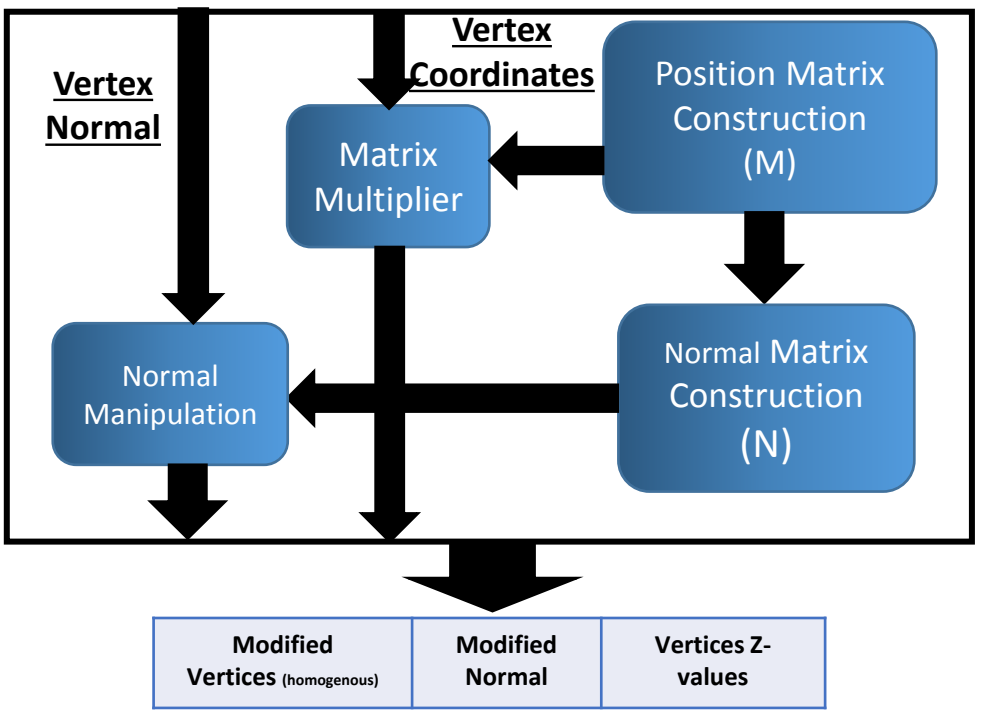
$$\text{- Mcam (3D)} = \begin{bmatrix} X_u & Y_u & Z_u & 0 \\ X_v & Y_v & Z_v & 0 \\ X_w & Y_w & Z_w & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \text{X} \begin{bmatrix} 1 & 0 & 0 & -Xe \\ 0 & 1 & 0 & -Ye \\ 0 & 0 & 1 & -Ze \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$where\ (\ W = \frac{-g}{||g||}\ ,u = \frac{t \times w}{||t \times w||}\ , v = w \times u\ )$$

$$\text{-Mcam (2D)} = \begin{matrix} 1 & 0 & -Xe \\ 0 & 1 & -Ye \\ 0 & 0 & 1 \end{matrix}$$

$$\text{Mper} = \begin{bmatrix} 2*n/r-l & 0 & (l+r)/(l-r) & 0 \\ 0 & 2*n/t-b & (b+t)/(b-t) & 0 \\ 0 & 0 & (f+n)/(f-n) & (2*n*f)/(f-n) \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

Vertex Processing stage



Attrite Registers

Vertices Coordinates (V_1,V_2, V_3)

Vertices Normal (N_1, N_2, N_3)

Uniform Registers

-Eye Location (Ex, Ey, Ez)
-Eye Direction (Gx, Gy, Gz)
-Up-vector (Tx, Ty, Tz)

World Space
[l, r] , [t, b], [n, f]

Screen Space
[N_x, N_y]

Normal Matrix Construction (N):-

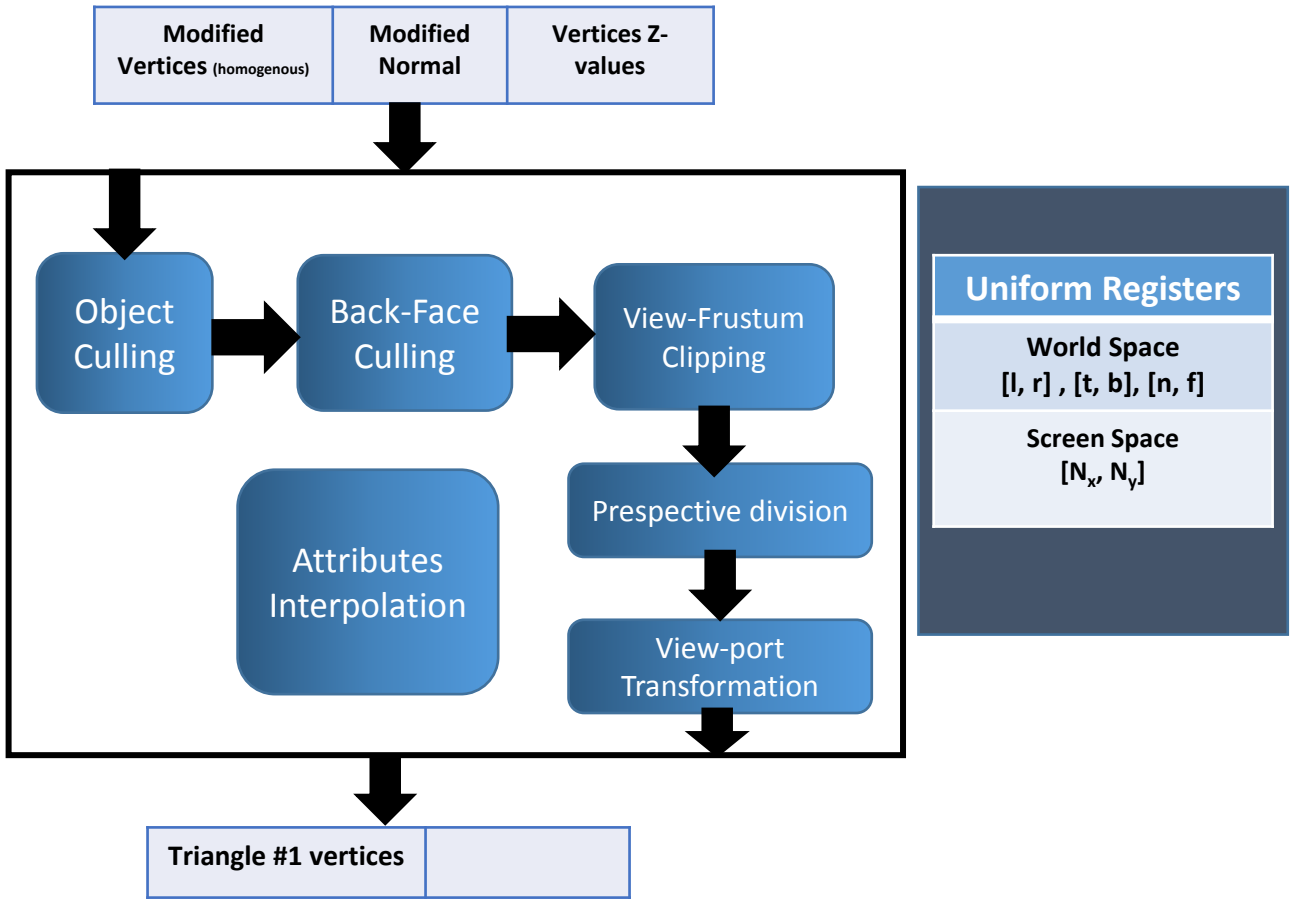
$$\mathbf{N} = \begin{bmatrix} m_{11}^c & m_{12}^c & m_{13}^c \\ m_{21}^c & m_{22}^c & m_{23}^c \\ m_{31}^c & m_{32}^c & m_{33}^c \end{bmatrix}$$

$$\mathbf{N} = \begin{bmatrix} m_{22}m_{33} - m_{23}m_{32} & m_{23}m_{31} - m_{21}m_{33} & m_{21}m_{32} - m_{22}m_{31} \\ m_{13}m_{32} - m_{12}m_{33} & m_{11}m_{33} - m_{13}m_{31} & m_{12}m_{31} - m_{11}m_{32} \\ m_{12}m_{23} - m_{13}m_{22} & m_{13}m_{21} - m_{11}m_{23} & m_{11}m_{22} - m_{12}m_{21} \end{bmatrix}$$

Normal Manipulation (N):-

- For Line or per-vertex normal
  - New_normal = N x i/p_Normal
- We can obtain per-triangle normal as
  - $(V_2 - V_0) \times (V_1 - V_0)$

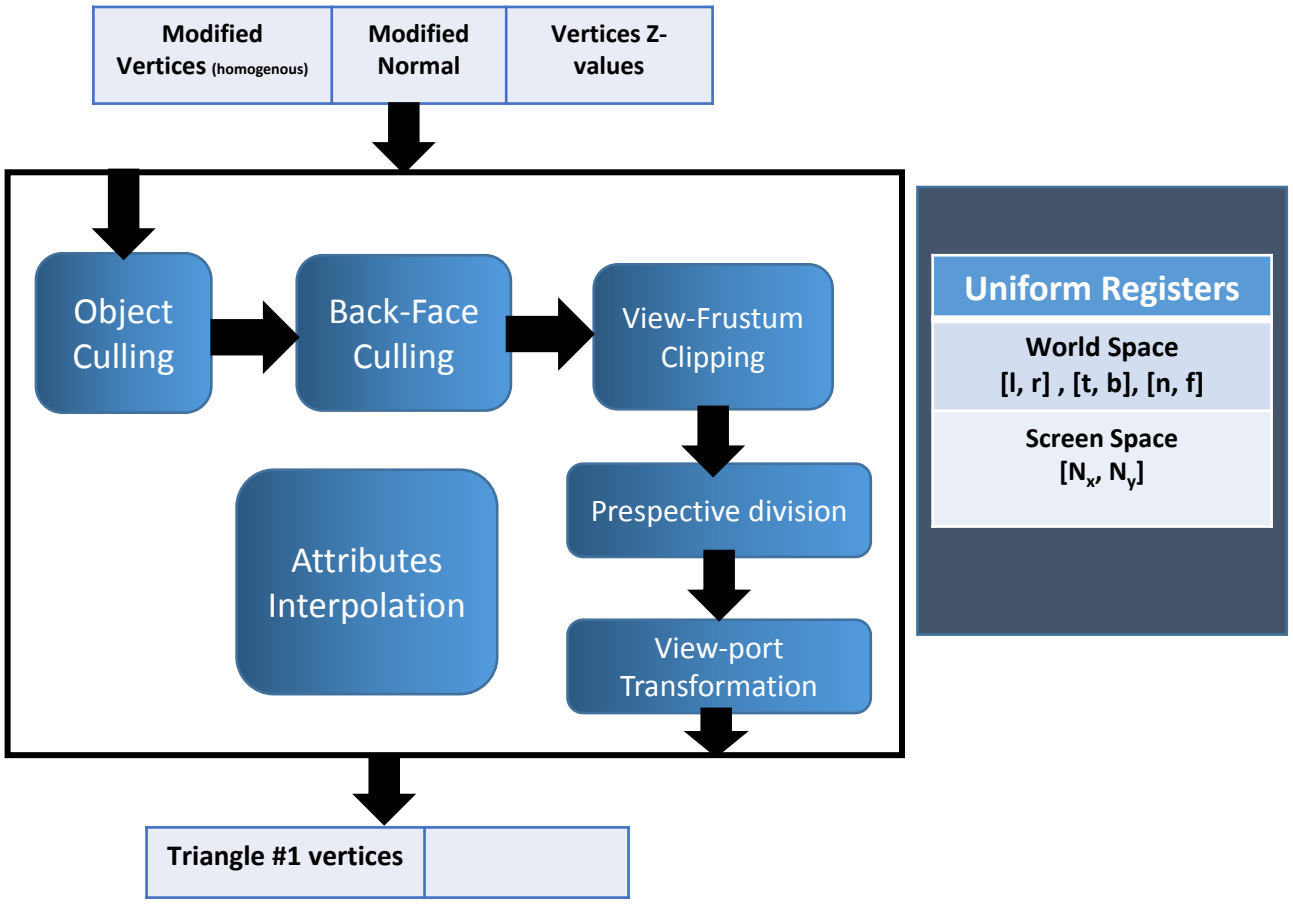
Culling & Clipping



Notes :-

1) till now, we not support “draw objects”. So, the object culling is bypassed.
2) back-facing culling:- if the triangle is oriented away from the eye point, then this triangle is not visible.
3) View-Frustum Clipping: we use Cohen-Sutherland Line Clipping Algorithm.

Culling & Clipping



Back-Face Culling :-

- The plane of the triangle is N.X =d
  - where N is the normal of the plane, and X is any vertex of the triangle.
- The triangle is oriented away from the eye point if N.E <d where E is the Eye direction

---

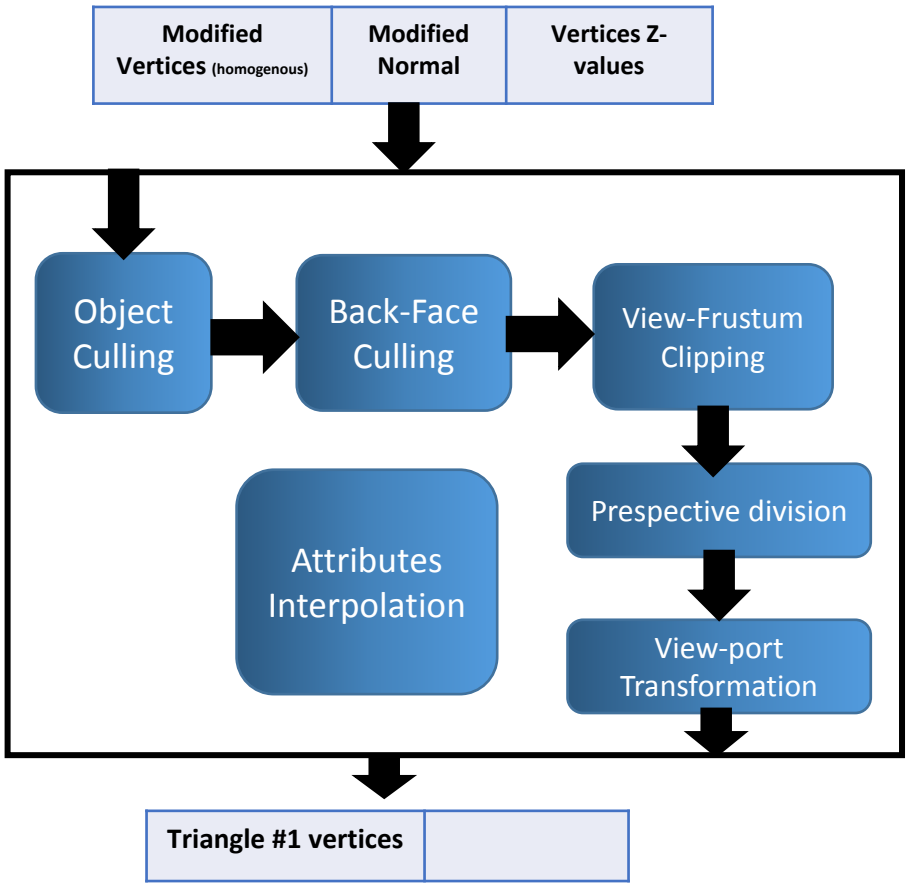
Algorithm:-

- Calculate (P- V_0). N by

$$\text{Det}\begin{pmatrix} -x_0 & x_1 - x_0 & x_2 - x_0 \\ -y_0 & y_1 - y_0 & y_2 - y_0 \\ -z_0 & z_1 - z_0 & z_2 - z_0 \end{pmatrix}$$

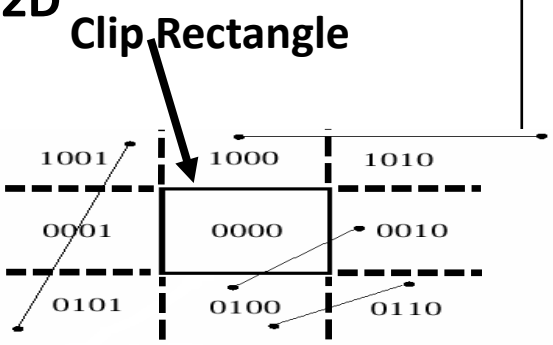
- If (P-$V_{0)}$. N <0 then the triangle is back-facing (cull it)

Culling & Clipping



View-Frustum Clipping :-

Cohen-Sutherland Line Clipping in 2D



- Divide plane into 9 regions
- Compute the sign bit of 4 comparisons between a vertex and an edge

$$y_{max} - y;\ \ y - y_{min};\ \ x_{max} - x;\ \ x - x_{min}$$

(point lies inside only if all four sign bits are) 0, otherwise exceeds edge

- 4 bit outcode records results of four bounds tests:

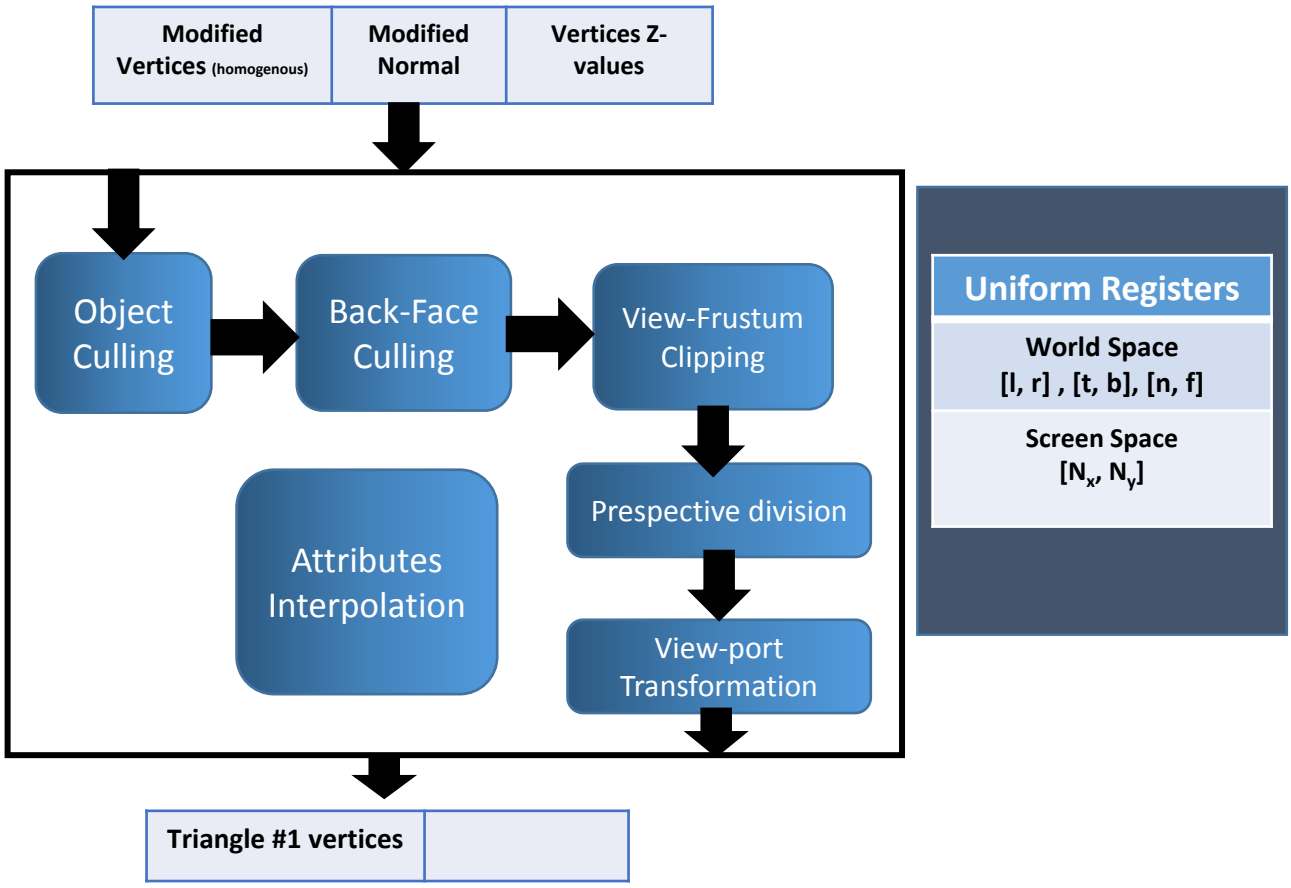
First bit: above top edge

Second bit: below bottom edge

Third bit: to the right of right edge

Fourth bit: to the left of left edge

Culling & Clipping



View-Frustum Clipping :-

Cohen-Sutherland Line Clipping in 2D

- Compute outcodes for both vertices of each line (denoted OC_0 and OC_1)

→Lines with $OC_0 = 0$ and $OC_1 = 0$ can be *trivially accepted.*

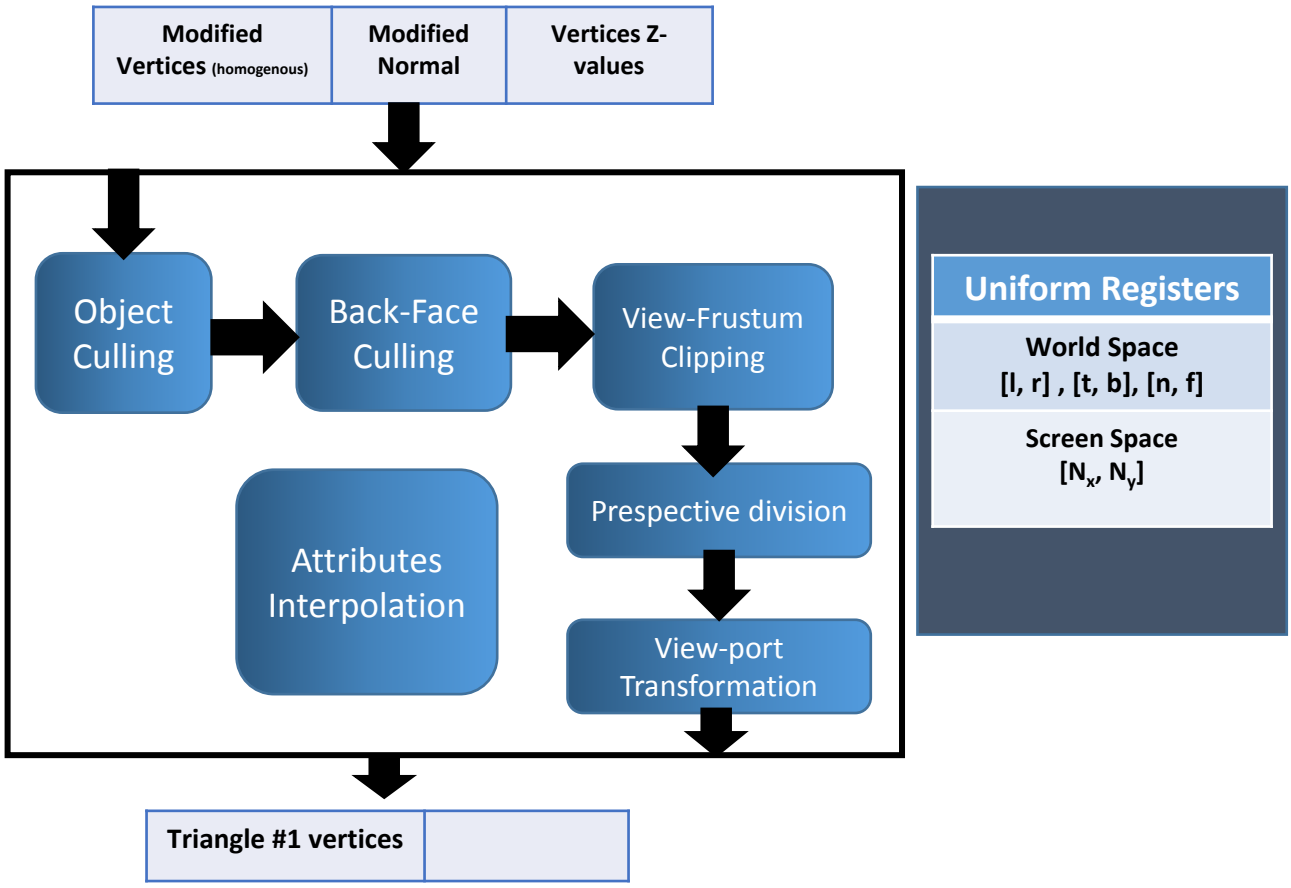
→ Lines lying entirely in a half plane outside an edge can be *trivially rejected* if (OC_0 **AND** OC_1) $\neq 0$

→If we can neither trivially accept/reject (T/A, T/R), divide and conquer

-Subdivide line into two segments; then T/A or T/R one or both segments:

- use a clip edge to cut line
- use outcodes to choose the edges that are crossed.
- pick an order for checking edges: top – bottom – right – left
- compute the intersection point
  - -the clip edge fixes either *x* or *y*
  - -can substitute into the line equation
- iterate for the newly shortened line, “extra” clips may happen (e.g., E-I at H)

Culling & Clipping



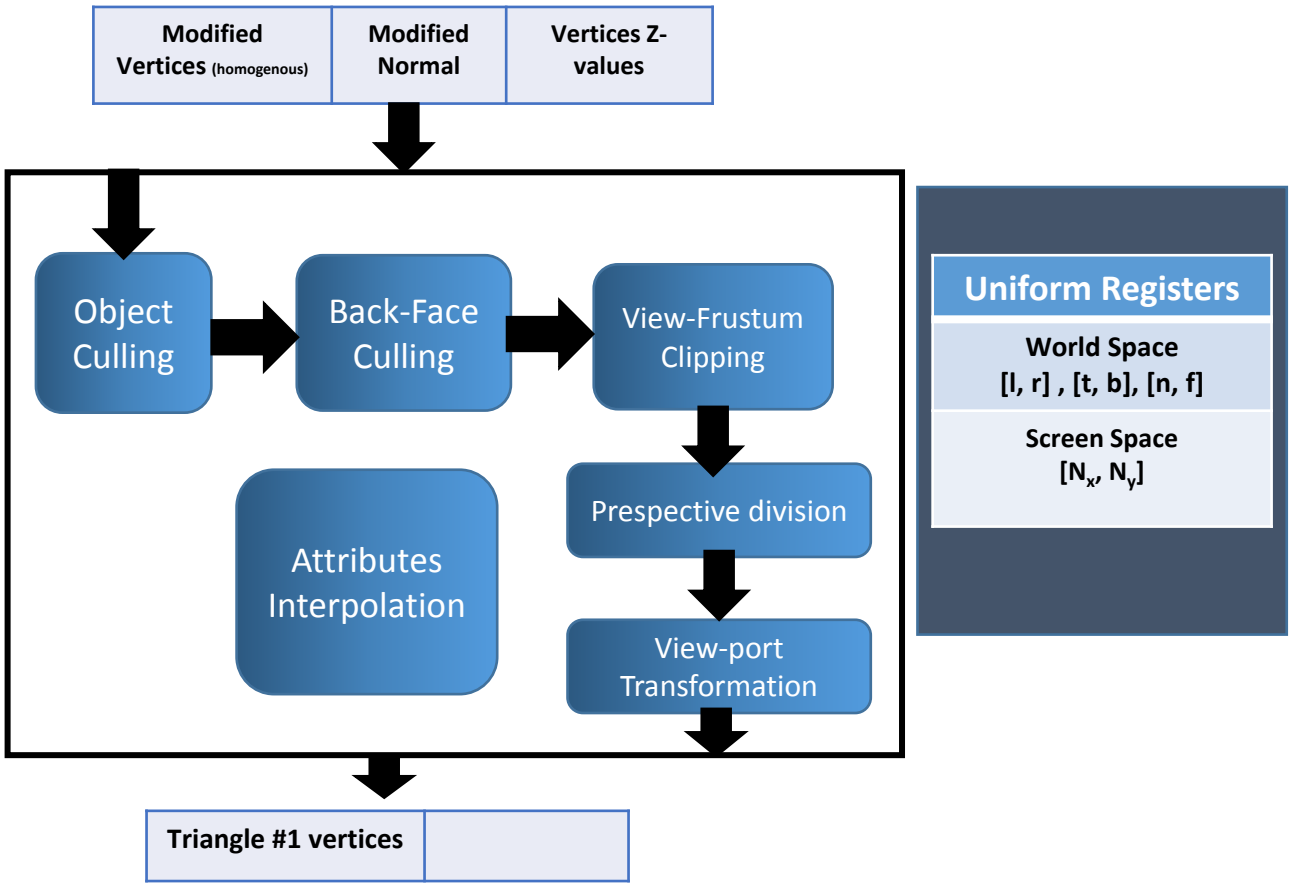
View-Frustum Clipping :-

Cohen-Sutherland Line Clipping in 2D

- Algorithm :-

```
ComputeOutCode(x0, y0, outcode0);
ComputeOutCode(x1, y1, outcode1);
repeat
check for trivial reject or trivial accept
pick the point that is outside the clip rectangle
if TOP then
    x = x0 + (x1 – x0) * (ymax – y0) / (y1 – y0);
  y = ymax;
else if BOTTOM then
    x = x0 + (x1 – x0) * (ymin – y0) / (y1 – y0);
    y = ymin;
else if RIGHT then
   y = y0 + (y1 – y0) * (xmax – x0) / (x1 – x0);
  x = xmax;
else if LEFT then
  y = y0 + (y1 – y0) * (xmin – x0) / (x1 – x0);
  x = xmin;
  if (x0, y0 is the outer point) then
     x0 = x; y0 = y; ComputeOutCode(x0, y0,  outcode0)
  else
      x1 = x; y1 = y; ComputeOutCode(x1, y1, outcode1)
until done
```

Culling & Clipping



View-Frustum Clipping :-

Cohen-Sutherland Line Clipping in 3D

Back plane
000000 (in front)
100000 (behind)

Front plane
010000 (in front)
000000 (behind)

Top plane
001000 (above)
000000 (below)

Bottom plane
000000 (above)
000100 (below)

Right plane
000000 (to left of)
000010 (to right of)

Left plane
000001 (to left of)
000000 (to right of)

- very similar to 2D
- Divide volume into 27 regions
- 6-bit outcode records results of 6 bounds tests
  - First bit: behind back plane
  - Second bit: in front of front plane
  - Third bit: above top plane
  - Fourth bit: below bottom plane
  - Fifth bit: to the right of right plane
  - Sixth bit: to the left of left plane
- note, outcodes may be calculated by **D = H.P**

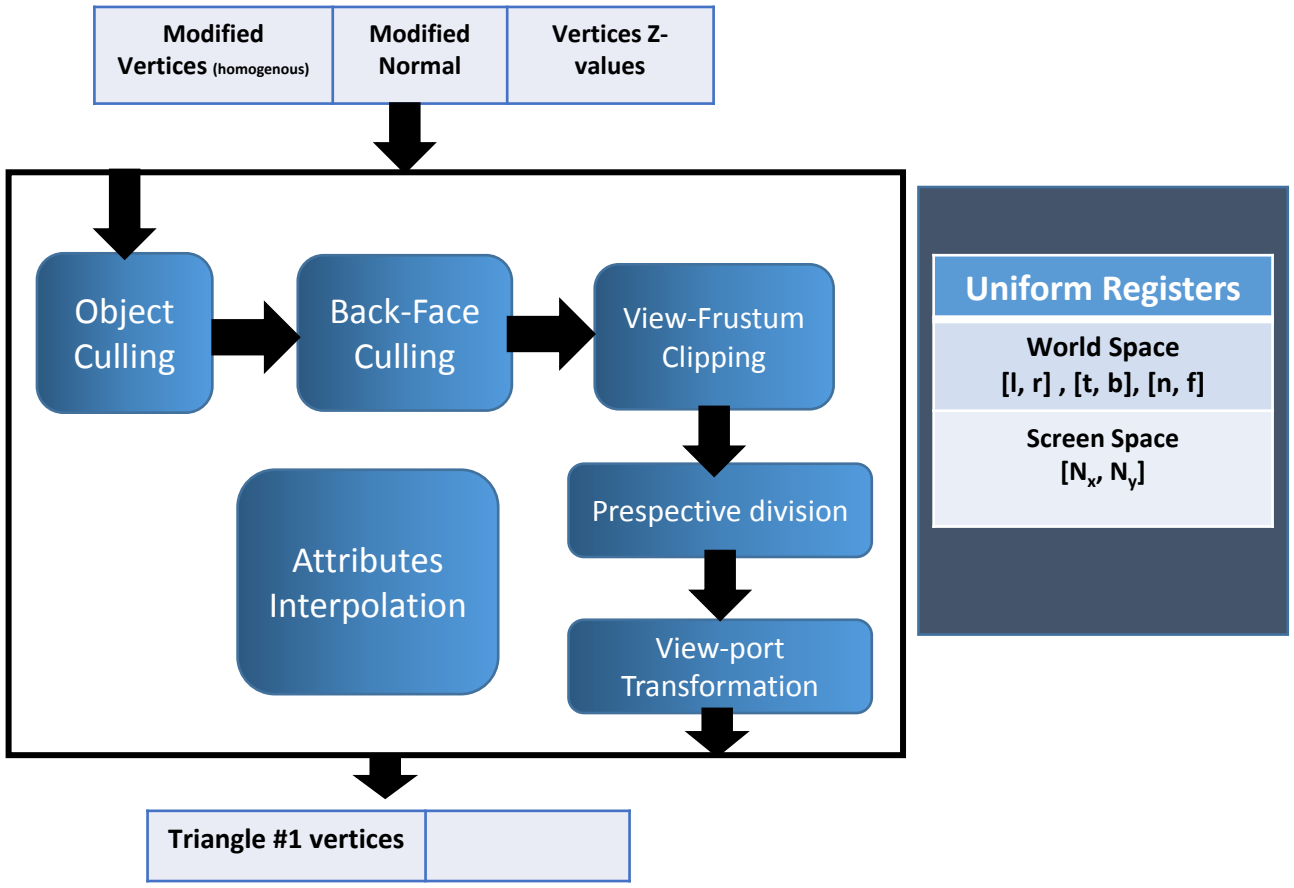
→ D >= 0 (pass through)
→ D < 0 (cull or reject)

$$H_{near} = (\ 0 \quad 0 \quad -1 \quad -near)$$
$$H_{far} = (\ 0 \quad 0 \quad 1 \quad far\ )$$
$$H_{bottom} = (\ 0 \quad near \quad bottom \quad 0\ )$$
$$H_{top} = (\ 0 \quad -near \quad -top \quad 0\ )$$
$$H_{left} = (\ left \quad near \quad 0 \quad 0\ )$$
$$H_{right} = (-right \quad -near \quad 0 \quad 0\ )$$

- Intersection Calculation: Insert explicit equation of line into implicit equation of plane

L(t) = P0 + *t* * (P1 – P0)

Culling & Clipping



View-Frustum Clipping :-

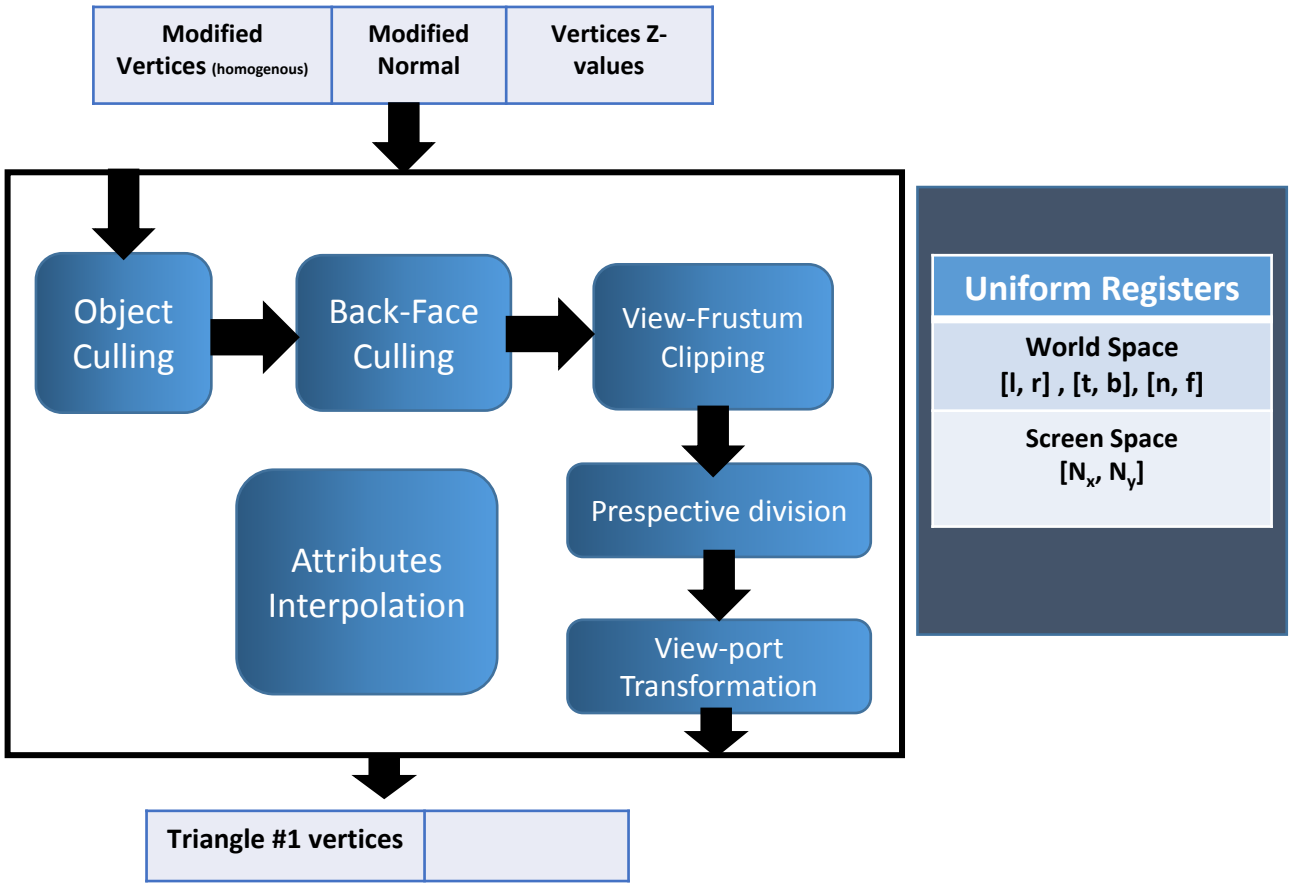
Cohen-Sutherland Line Clipping in 3D

- Algorithm:-

```
ComputeOutCode(x0, y0, z0, outcode0);
ComputeOutCode(x1, y1, z1, outcode1);
repeat
    check for trivial reject or trivial accept
    pick the point that is outside the clip rectangle
    if TOP then
        x = x0 + (x1 – x0) * (ymax – y0) / (y1 – y0);
        z = z0 + (z1 – z0) * (ymax – y0) / (y1 – y0);
        y = ymax;
    else if BOTTOM then
        x = x0 + (x1 – x0) * (ymin – y0) / (y1 – y0);
        z = z0 + (z1 – z0) * (ymin – y0) / (y1 – y0);
        y = ymin;
    else if RIGHT then
        y = y0 + (y1 – y0) * (xmax – x0) / (x1 – x0);
        z = z0 + (z1 – z0) * (xmax – x0) / (x1 – x0);
        x = xmax;
    else if LEFT then
        y = y0 + (y1 – y0) * (xmin – x0) / (x1 – x0);
        z = z0 + (z1 – z0) * (xmin – x0) / (x1 – x0);
        x = xmin;
    else if NEAR then
        x = x0 + (x1 – x0) * (zmax – z0) / (z1 – z0);
        y = y0 + (y1 – y0) * (zmax – z0) / (z1 – z0);
        z = zmax;
    else if FAR then
        x = x0 + (x1 – x0) * (zmin – z0) / (z1 – z0);
        y = y0 + (y1 – y0) * (zmin – z0) / (z1 – z0);
        z = zmin;

    if (x0, y0, z0 is the outer point) then
        x0 = x; y0 = y; z0 = z;
        ComputeOutCode(x0, y0, z0, outcode0)
    else
        x1 = x; y1 = y; z1 = z;
        ComputeOutCode(x1, y1, z1, outcode1)
until done
```

Culling & Clipping



Perspective Division:-

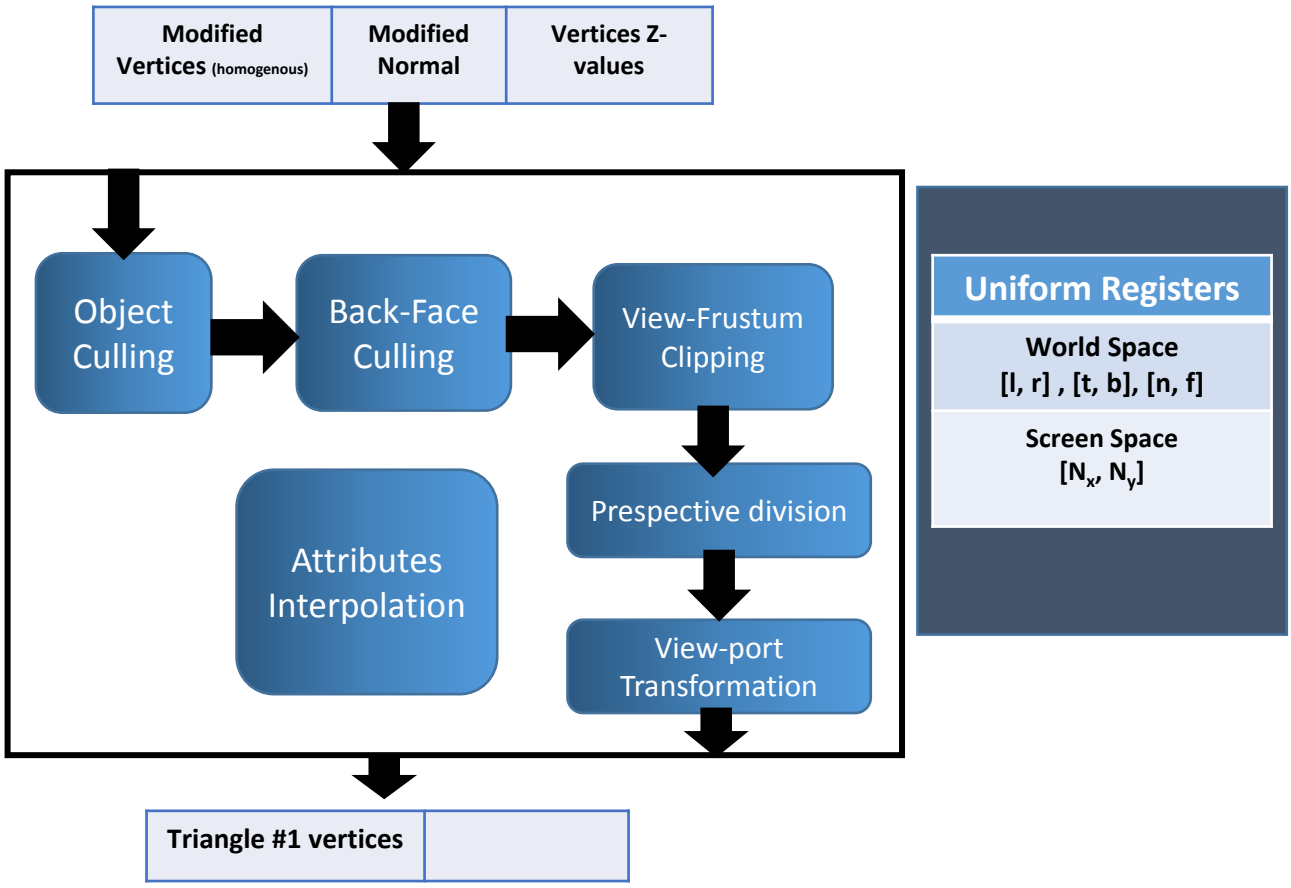
- [x y z w] → [x/w y/w z/w]

homogeneous clip space → normalized device coordinates (NDC)

The View-port Transformation :-

$$\text{- Mvp} = 0.5 * \begin{bmatrix} nx/1 & 0 & 0 & nx-1/1 \\ 0 & ny/1 & 0 & ny-1/1 \\ 0 & 0 & 1/2 & 0 \\ 0 & 0 & 0 & 1/2 \end{bmatrix}$$

Culling & Clipping



Attributes Interpolation:-

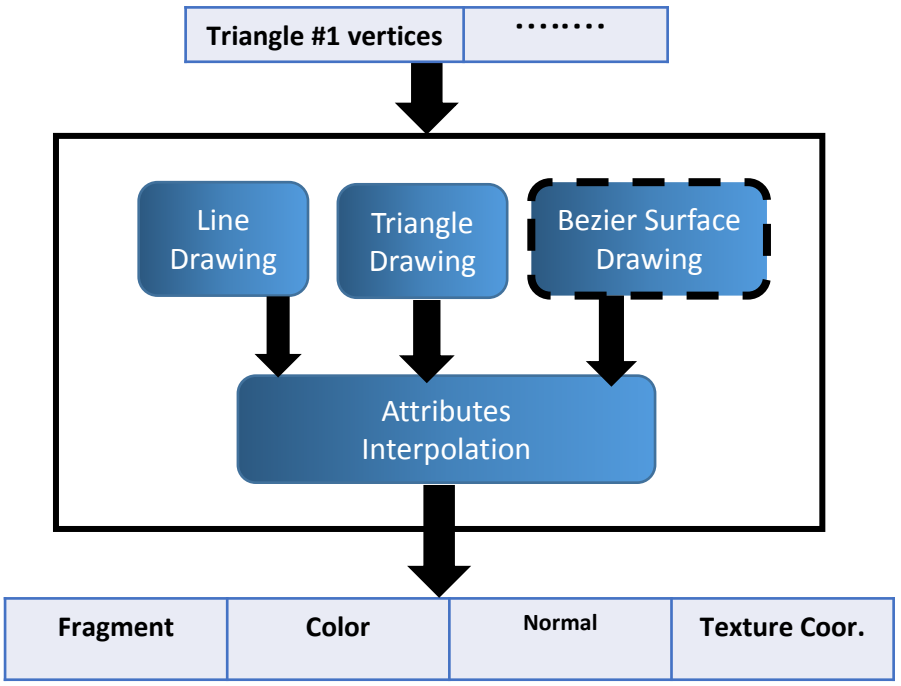
- We have to interpolate the vertex attributes such as (color**(c)**, normal**(n)**, texture**(t)**) to determine the attributes of the new vertices.
- we are going to ignore the Z-coordinate in the interpolation for simplification.
- we uses the Ground interpolation for the barcentric coordinates.
- **Algorithm** :-
- For point P(x, y) between two endpoints $P_1(X_1,Y_1)$ and $P_2(X_2, Y_2)$:-

→ $f_{01}(x, y) = (y_0 - y_1)x + (x_1 - x_0)y + x_0y_1 - x_1y_0$,
→ $f_{12}(x, y) = (y_1 - y_2)x + (x_2 - x_1)y + x_1y_2 - x_2y_1$
→ $\alpha = f_{12}(x, y)/f_{12}(x_0, y_0)$
→ $\beta = f_{20}(x, y)/f_{20}(x_1, y_1)$

Then,

$$c = \alpha * c_0 + \beta * c_1$$
$$t = \alpha * t_0 + \beta * t_1$$
$$n = \alpha * n_0 + \beta * n_1$$

Rasterization



Uniform Registers

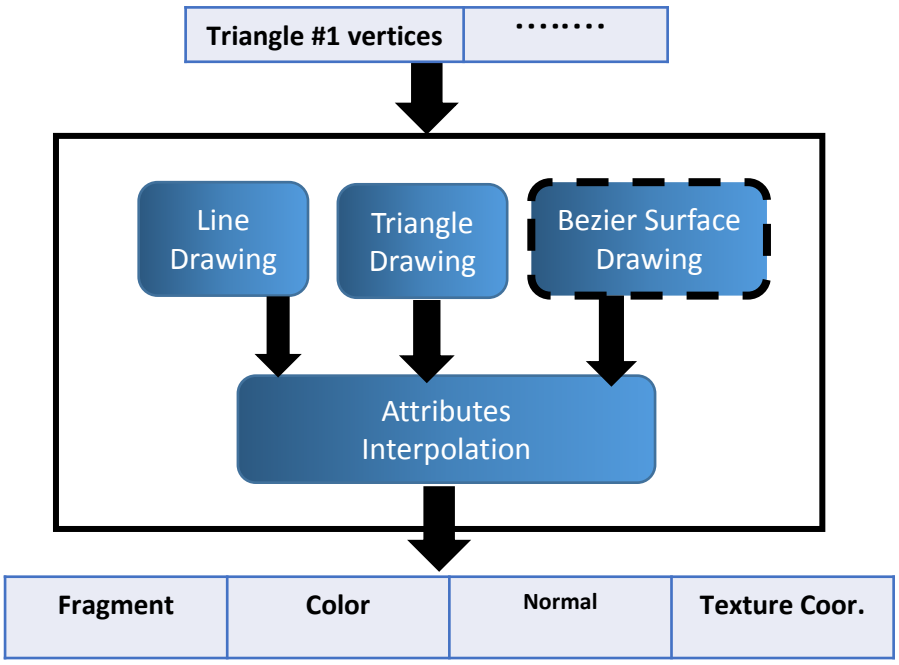
World Space
[l, r] , [t, b], [n, f]

Screen Space
[N_x, N_y]

Notes :-

1) we use the Incremental mid-point Algorithm for the line Drawing.
2) For triangles, we uses the Ground interpolation for the barcentric coordinates.
3) Till now, we have not design a bezier surface drawing.
4) what about Circle & Ellipse drawing ?

Rasterization



Uniform Registers

World Space
[l, r] , [t, b], [n, f]

Screen Space
[N_x, N_y]

Line Drawing & interpolating Algorithm:-

Sort(x_0,x_1);
Sort(y_0,y_1);
M = (y_1-y_0)/(x_1-x_0);
If (m > -1 && m < 1)
{ y = y_0;
If(m > 0) {D = f(x_0,y_0-0.5);}
If(m < 0) {D = f(x_0,y_0+0.5);}
For x= x_0 to x_1
{Draw (x,y) pixel with c, n, t
If (m > 0){ D = D + (x_1 – x_0) + (y_0 – y_1);
If(D > 0) y = y+1;}
if(m < 0){ D = D + (x1 – x0) – (y0 – y1);
if (D < 0) y = y-1;}

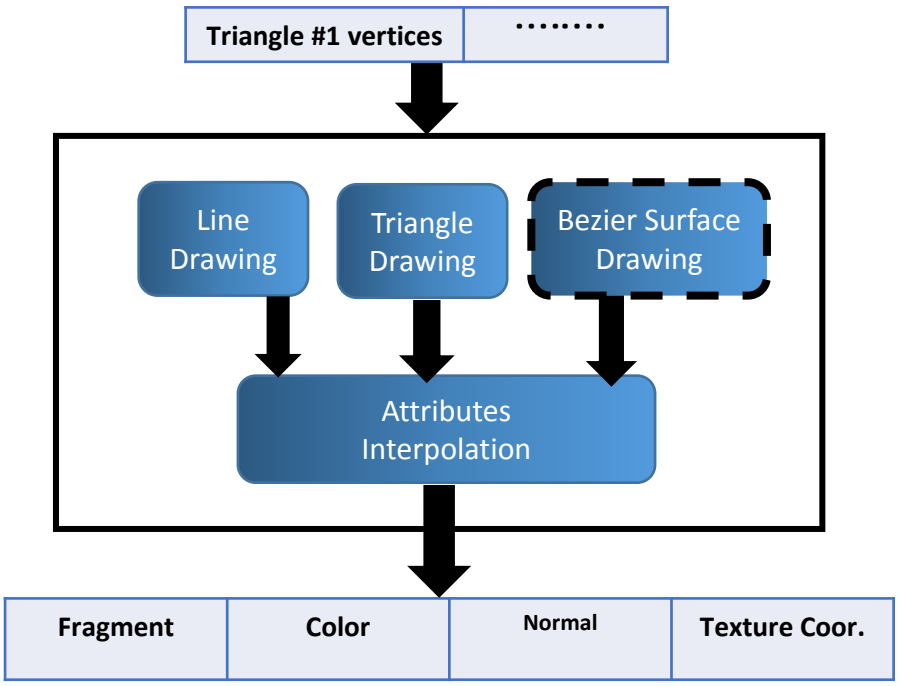
If (m < -1 || m > 1)
{ x = x0;
If(m > 0) {D = f(x0-0.5,y0);}
If(m < 0) {D = f(x0+0.5,y0);}
For y= yo to y1
{Draw (x,y)
If (m > 0){ D = D - (x1 – x0) + (y0 – y1);
If(D > 0) x = x+1;}
if(m < 0){ D = D - (x1 – xo) + (yo – y1)
if (D < 0) x = x-1;}

Draw (x,y) with c, n, t
{
$f_{01}(x, y) = (y_0 - y_1)x + (x_1 - x_0)y + x_0y_1 - x_1y_0$,
$f_{12}(x, y) = (y_1 - y_2)x + (x_2 - x_1)y + x_1y_2 - x_2y_1$
$\alpha = f_{12}(x, y)/f_{12}(x_0, y_0)$;
$\beta = f_{20}(x, y)/f_{20}(x_1, y_1)$;
Then,

$c = \alpha * c_0 + \beta * c_1$
$t = \alpha * t_0 + \beta * t_1$
$n = \alpha * n_0 + \beta * n_1$

update buffer
}

Rasterization



Uniform Registers

World Space
[l, r] , [t, b], [n, f]

Screen Space
[N_x, N_y]

Triangle Drawing & Interpolation Algorithm:-

- $x_{min} = floor\ (x_i)$
  $x_{max} = ceiling\ (x_i)$
  $y_{min} = floor\ (y_i)$
  $y_{max} = ceiling\ (y_i)$

$f_{01}(x, y) = (y_0 - y_1)x + (x_1 - x_0)y + x_0y_1 - x_1y_0,$
$f_{12}(x, y) = (y_1 - y_2)x + (x_2 - x_1)y + x_1y_2 - x_2y_1,$
$f_{20}(x, y) = (y_2 - y_0)x + (x_0 - x_2)y + x_2y_0 - x_0y_2.$

for $y = y_{min}$ to y_{max} do
for $x = x_{min}$ to x_{max} do
$\alpha = f_{12}(x, y)/f_{12}(x_0, y_0)$
$\beta = f_{20}(x, y)/f_{20}(x_1, y_1)$
$\gamma = f_{01}(x, y)/f_{01}(x_2, y_2)$
if ($\alpha > 0$ and $\beta > 0$ and $\gamma > 0$) then
$c = \alpha * c_0 + \beta * c_1 + \gamma * c_2$
$n = \alpha * n_0 + \beta * n_1 + \gamma * n_2$
$t = \alpha * t_0 + \beta * t_1 + \gamma * t_2$
update buffer

Fragment Processing Algorithm

1. Getting the shading parameters as inputs and the fragment position & color from varying registers
2. Calculate the pixel's color from direct equation
3. Check if the pixel is in shadow or not
   - Using a comparator , if the two z-values are equal then it's in light otherwise it's in shadow
   - .
   - If it's in shadow , we get the new color for the pixel which is more darker than usual color
   - Multiplying the original color with the shadowing coeff
   - Note: : enabling shadows & shading is similar to enabling 2D/3D
   - Some control registers to determine the shading model and the number of light sources
   - 5 comparator, 2 adders, 4 multipliers,SFU. (expected to increase)

Applying direct equation :

$C_{R\ or\ B\ or\ G} = K_a I_a + K_d i \max(0, n.i) + K_s i \max(0, n.h)^p$

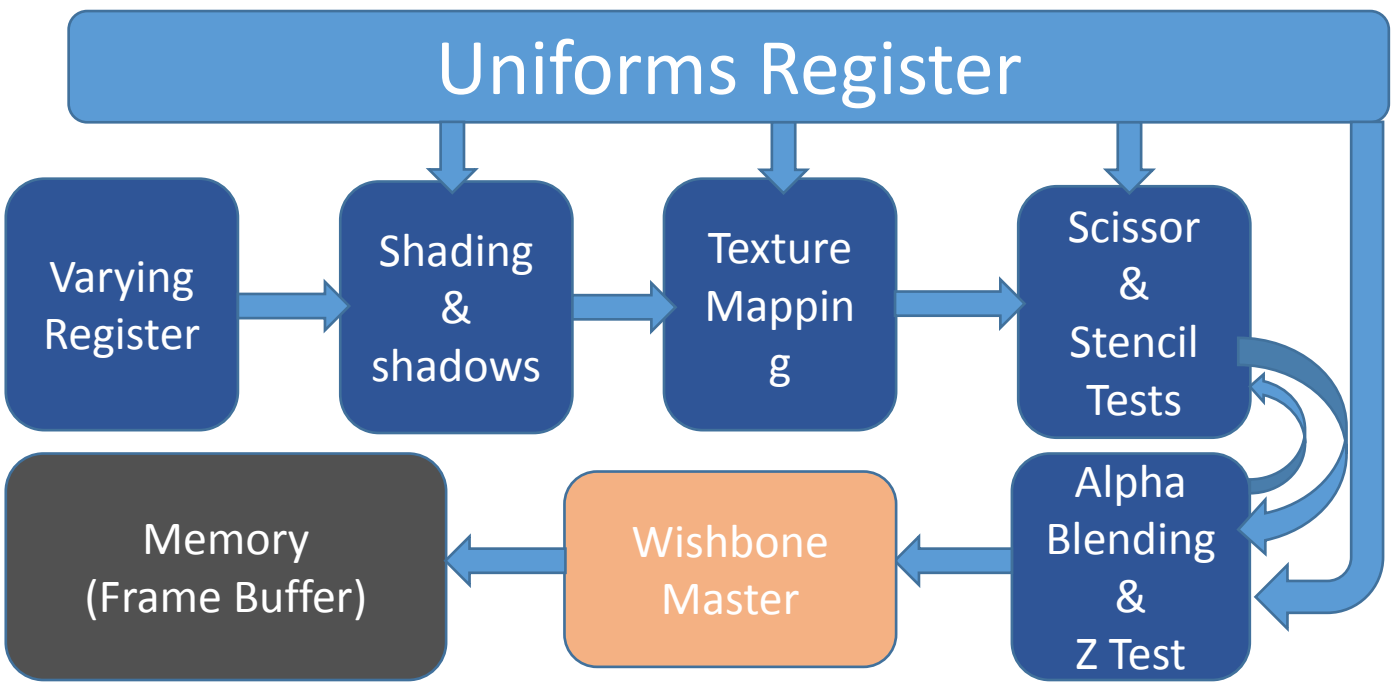
For more light sources :

$C_{R\ or\ B\ or\ G} = K_a I_a + \sum_{k=1}^{N} K_d\, i_k \max(0, n.i_k) + K_s i_k \max(0, n.h)^p$

For shadows :

$C_{new} = S * C_{old}$

Per-fragment Processor Diagram



Tests Algorithm

1. If scissor test is enabled check whether your pixel is located in it or not using adders & comparators

- If not the pixel is discarded

1. If stencil test is enabled check with the stencil buffer is going to be updated or not

2. If two pixels have the same position (x,y) check with the depth test which is foreground and which is background

3. Determinig the final color if we have transparent objects with Alpha Blending test

4. Finally , Updating the frame buffer with the ready-to-display pixel.

- 6 comparator, 3 adders,1 subtract, 2 multipliers. (expected to increase)

For scissor :

$left \leq (x)\ also\ (x+1) < left + width$

$bottom \leq (y)\ also\ (y+1), bottom + height$

For Alpha Blending :

$C = \alpha C_f + (255 - \alpha)\ C_b$