

This document is a description of the Open Graphics Project's 3D rasterizer. It attempts to be compliant with most of the OpenGL 2.0 specification, insofar as it applies to those functions that are implemented.

1.0: Rasterize

Iterates over screen coordinates, shade colors (primary, secondary), texture coordinates (up to two textures), W, and F (used in fog). Iterates only once per fragment.

All rasterizer results are computed by initial values and linear deltas. As X and Y step, initial values are incremented by their corresponding deltas to produce the subsequent result value.

[When converting from world coordinates to screen coordinates, you have a 4-element vector with X,Y,Z,W in it against which you multiply a 4x4 transform matrix. That results in another 4-element vector which has X,Y,Z,W for screen coordinates. The W of the screen coordinates is the one referred to here.]

The rasterizer is actually broken up into two macro stages:

1.1: Vertical rasterize

Screen Y coordinates are integers, so the host must compute the appropriate trapezoids for each triangle with the correctly partially stepped deltas.

Vertical Rasterizer Parameters:

- uint16 **Y** and **H**, starting line and height of trapezoid
- float25 **X1** and **X2** initial left and right edges of trapezoid
- float25 **dX1dY** and **dX2dY** step of left and right edges
- float25 **W** initial value for perspective correction
- float25 **dWdY** for vertical step of W
- 4 of float25 ARGB for initial value of upper left primary shade
- 4 of float25 ARGB for vertical step of primary shade
- 4 of float25 ARGB for initial value of upper left secondary shade
- 4 of float25 ARGB for vertical step of secondary shade
- float25 **S1** and **T1** for primary texture initial coordinates
- float25 **dS1dY** and **dT1dY** for primary texture step
- float25 primary texture initial LOD
- float25 primary texture LOD vertical step
- float25 **S2** and **T2** for secondary texture initial coordinates
- float25 **dS2dY** and **dT2dY** for secondary texture step
- float25 secondary texture initial LOD
- float25 secondary texture LOD vertical step
- float25 **F** initial fog value
- float25 **dFdY** fog value vertical step

[need formulas to convert from world coordinates to rasterizer parameters.]

1.2: X adjust

Left and right X edges are adjusted to integer coordinates so that we generate fragments whose centers fall within the triangle. Since the adjustment causes a change in X, the corresponding adjustments have to be made to the initial values of all other parameters:

```
adjust = ceil(x) - x
new_param_initial = old_param_initial + param_delta * adjust
```

1.3: Horizontal rasterize

Iterate from left X to right X, generating other parameters by summing deltas.

Horizontal Rasterizer Parameters:

- uint16 **X1** and **X2**, left and right edges of scanline (computed)
- float25 **W** left (computed)
- float25 **dWdX** for horizontal step of W (user)
- 4 of float25 for left edge of primary shade (computed)
- 4 of float25 for horizontal step of primary shade (user)
- 4 of float25 for left edge of secondary shade (computed)
- 4 of float25 for horizontal step of secondary shade (user)
- float25 **S1** and **T1** for primary texture left coordinates (computed)
- float25 **dS1dX** and **dT1dX** for primary texture horizontal step (user)
- float25 primary texture left LOD (computed)
- float25 primary texture LOD horizontal step (user)
- float25 **S2** and **T2** for primary texture left coordinates (computed)
- float25 **dS2dX** and **dT2dX** for primary texture step (user)
- float25 secondary texture left LOD (computed)
- float25 secondary texture LOD horizontal step (user)
- float25 **F** initial fog value (computed)
- float25 **dFdX** fog value vertical step (user)

Other horizontal state:

- **draw_right_edge** – Usually, the pixel at X2 is not drawn. This parameter enabled drawing of the pixel at X2.
- **swap_xy** – For 3D graphics, slopes are computed Y-major. For some 2D operations, such as line segments, pixels must be computed in X-major order. This parameter exchanges the X and Y coordinates for the fragment.

Note: 1.1 through 1.3 may be combined into a single state machine. The reason is that we will only waste a few cycles for each scanline, it'll require less hardware, and the wasted cycles will generally be absorbed by some other delay further down in the pipeline anyway.

Question: Do we compute coverage mask (alpha for edge antialiasing) somewhere in 1.x for pixels which are partially covered by the triangle?

How to compute rasterizer parameters

MesaGL has all the necessary logic to convert from object space to world space to normalized device coordinates to screen coordinates. However, to be sure that quirks of this design do not cause headaches for programmers, I'll describe the math for computing rasterizer parameters from a set of hypothetical "scaled world coordinates". Scaled world coordinates are scaled to the viewport but not projected onto the viewport. I choose to do it this way because there are certain things about OpenGL viewport semantics which I do not yet understand. I am also assuming that X increases to the right, Y increases downward, and Z increases into the screen.

We begin with a scene which has been decomposed into triangles. A triangle is defined by three vertexes in space, and each vertex has a set of associated properties:

- Coordinate in space (X_w, Y_w, Z_w)
- Primary and secondary colors (A, R, G, B), which we'll collectively call C_w
- Two sets of texture coordinates U, V, and LOD_w
- Fog factor, F_w

In OpenGL, you can specify a W_w coordinate for each vertex, but I am assuming that a standard projective transform is being performed, where W_w is a function of Z_w .

The first thing to do is perform the projection which converts from world coordinates to screen coordinates. The projection involves dividing by:

$$W_w = 1 + Z_w/d,$$

where d is the distance from the eye to the screen (in pixels, since we're scaled already to the viewport). Each vertex property must be divided by W_w , yielding X_s, Y_s, C, S, T, LOD , and F .

The hardware needs to be able to reverse the projection. This is so that textures and color gradients have correct perspective foreshortening. For instance, the relationship between X and U is linear in world space but not in screen space. We, therefore, compute S in screen space and then reverse the projection by dividing by what we call W_s . For each fragment, W_s is simply the reciprocal of W_w at the corresponding point in space, and it turns out that W_s is a linear function of X_s and Y_s . Below, we refer to W_s as simply W .

Once everything has been projected, we need to split the triangle. The hardware can only handle flat-top or flat-bottom triangles. Sort the vertexes vertically and name the projected coordinates $X_a, Y_a, X_b, Y_b, X_c,$ and Y_c . The triangle will be split at Y_b . The point at X_b, Y_b is called the knee and it is either on the left or the right.

For right-knee triangles:

The vertical rasterizer needs to compute the left edge of the triangle.

In this case, we simply need to compute the gradients between Y_a and Y_c , as such:

- $dY = Y_c - Y_a$
- $dX1dY = (X_c - X_a) / dY$
- $dX2dY = (X_b - X_a) / (Y_b - Y_a)$
- $dWdY = (W_c - W_a) / dY$
- $dCdY = (C_c - C_a) / dY$
- $dSdY = (S_c - S_a) / dY$
- $dTdY = (T_c - T_a) / dY$
- $dFdY = (F_c - F_a) / dY$
- $dLODdY = (LOD_c - LOD_a) / dY$

Next, we need to compute initial values. You'd expect all parameters to start at the values from point a, but this is not the case. Y_a is probably not an integer, and the hardware can only handle integer Y coordinates. Software must make a small adjustment:

- $Y = \text{round}(Y_a)$
- $Y_{\text{adjust}} = \text{round}(Y_a) - Y_a + 0.5$
- $X1 = X_a + dX1dY * Y_{\text{adjust}}$
- $X2 = X_a + dX2dY * Y_{\text{adjust}}$
- $W = W_a + dWdY * Y_{\text{adjust}}$
- $C = C_a + dCdY * Y_{\text{adjust}}$
- $S = S_a + dSdY * Y_{\text{adjust}}$
- $T = T_a + dTdY * Y_{\text{adjust}}$
- $F = F_a + dFdY * Y_{\text{adjust}}$
- $LOD = LOD_a + dLODdY * Y_{\text{adjust}}$

In addition, we need to compute the height of the top portion of the triangle:

- $H = \text{round}(Y_b) - \text{round}(Y_a)$

To compute the horizontal parameters requires that we compute the gradient at the knee. Since the knee is on the right, we need to compute the parameters at the same row on the left, a point which we'll refer to as 'd':

- $dY = Y_b - Y_a$
- $X_d = X_a + dX1dY * dY$
- $C_d = C_a + dCdY * dY$
- $S_d = S_a + dSdY * dY$
- etc.

Now, we can compute the horizontal gradients:

- $dX = X_b - X_d$
- $dCdX = (C_b - C_d) / dX$
- $dSdX = (S_b - S_d) / dX$
- etc.

At this point, we instruct the hardware to rasterize the triangle and then we compute the parameters for the bottom half. It turns out that

everything's the same except for the slope, coordinate of the right edge, and the height. All other parameters are either the same or have iterated to the values they need to have for the bottom portion of the triangle.

- $dY = Y_c - Y_b$
- $dX2dY = (X_b - X_c) / dY$
- $adjust = round(Y_b) - Y_b + 0.5$
- $X2 = X_b + adjust * dX2dY$
- $H = round(Y_c) - round(Y_b)$

[Possible further discussion: triangles with already flat top and how to recompute all parameters to eliminate accumulated round-off error.]

For left-knee triangles:

Left-knee triangles are very similar, although now we have to deal with the fact that the gradient change occurs on the left. As such, everything is computed between Y_a and Y_b .

- $dY = Y_b - Y_a$
- $dX1dY = (X_b - X_a) / dY$
- $dX2dY = (X_c - X_a) / (Y_c - Y_a)$
- $dCdY = (C_b - C_a) / dY$
- $dSdY = (S_b - S_a) / dY$
- $dTdY = (T_b - T_a) / dY$
- $dFdY = (F_b - F_a) / dY$
- $dLODdY = (LOD_b - LOD_a) / dY$

Just as before, we have to adjust to integer Y :

- $Y = round(Y_a)$
- $Yadjust = ceil(Y_a) - Y_a + 0.5$
- $X1 = X_a + dX1dY * Yadjust$
- $X2 = X_a + dX2dY * Yadjust$
- $C = C_a + dCdY * Yadjust$
- $S = S_a + dSdY * Yadjust$
- $T = T_a + dTdY * Yadjust$
- $F = F_a + dFdY * Yadjust$
- $LOD = LOD_a + dLODdY * Yadjust$

The height is computed the same as before:

- $H = round(Y_b) - round(Y_a)$

Computing the knee is different since the knee is on the left and we now need to compute point d along the right edge:

- $X_d = X_a + dX2dY * (Y_b - Y_a)$
- $C_d = C_a + (C_c - C_a) / (Y_c - Y_a) * (Y_b - Y_a)$
- $S_d = S_a + (S_c - S_a) / (Y_c - Y_a) * (Y_b - Y_a)$
- etc.

Now, we can compute the horizontal gradients:

- $dX = X_d - X_b$
- $dCdX = (C_d - C_b) / dX$
- $dSdX = (S_d - S_b) / dX$
- etc.

At this point, instruct the GPU to rasterize. Next, we compute the parameters for the bottom portion. This time, things are a bit more complicated. Before, the left edge slope didn't change, so the vertical gradients didn't change. But now, the left edge slope changes. In addition, all of the initial values have to be recomputed since the current values are projected along the wrong slope. Therefore, we have to recompute all vertical gradients and initial values. Fortunately, the horizontal gradients remain the same.

- $dY = Y_c - Y_b$
- $dXldY = (X_c - X_b) / dY$
- $adjust = round(Y_b) - Y_b + 0.5$
- $X_l = X_b + dXldY * adjust$
- $dCdY = (C_c - C_b) / dY$
- $dSdY = (S_c - S_b) / dY$
- $dTdY = (T_c - T_b) / dY$
- $dFdY = (F_c - F_b) / dY$
- $dLODdY = (LOD_c - LOD_b) / dY$
- $H = round(Y_c) - round(Y_b)$
- $C = C_b + adjust * dCdY$
- $S = S_b + adjust * dSdY$
- etc.

2.0: Scissor

Screen coordinates are compared against scissor box. Fragments which fail test are dropped.

Note that scissor and ownership tests are out of order with respect to the OpenGL spec, but this has no effect on conformance.

Scissor parameters:

- `uint16 ScissorLeft, ScissorRight, ScissorTop, ScissorBot`
- `bool enable_scissor`

3.0: Ownership test

Fetch ownership values from a framebuffer store and compare to owner of fragment. Pass only those that match.

Ownership parameters:

- `uint32 OwnershipBufferBaseAddress`
- `uint16 OwnershipBufferPitch`
- `uint32 OwnershipReference`
- `bool enable_ownership_test`

4.0: Reciprocal and Perspective correction

Compute $1/W$, calling it M . Then multiply ALL fragment parameters by M .

[Notes: Some 2D functionality needs to go here, including 2D 32x32 stipple and 1D line stipple.]

5.0: Texture

Fetch texels from graphics memory and merge via filtering. This unit may make multiple texture fetches per fragment.

Primary (diffuse) shade is combined with textures starting from the beginning of the texture pipeline.

User Parameters:

- Two texture base addresses
- Two texture pitch (pixel width) values
- Two texture sizes, which are the total memory size of the level zero MIPmap.
- Two sets of X and Y bitmasks for texture repeat dimensions
- Two filter modes
- Two sets of MIP level boundaries
- Texture color and alpha selectors and functions
- Texture color constants

Real texture coordinates U and V are computed by this formula:

$$U = S/W, \text{ and likewise} \\ V = T/W$$

MIPmap level is computed as:

$$\text{TAU} = \log_2(\text{LOD}/(W*W)), \text{ where LOD is computed by the rasterizer}$$

Texture filter modes are:

- None
- Nearest
- Linear – bilinear interpolation
- NEAREST_MIPMAP_NEAREST
- NEAREST_MIPMAP_LINEAR – Linearly interpolate between two MIPmaps. The MIPmap fetches themselves are not interpolated.
- LINEAR_MIPMAP_NEAREST – Pick the nearest MIPmap and then perform bilinear interpolation
- LINEAR_MIPMAP_LINEAR – Full bilinear interpolation

There are two texture stages. At each stage, a fragment color is combined with a texture color and a constant color. These four colors are referred to in selectable order as C_1 , C_2 , C_3 , and C_4 .

The selectable color sources are:

- From previous texture context
- Primary

- Constant
- From any preceding texture context
- Alpha channel from any of the above, replicated as RGB

Alpha source are alpha channels from the same sources. Finally, sources can be inverted (1-X).

Color combine functions are:

- Replace – $C_{out} = C_0$
- Modulate – $C_{out} = C_0 * C_1$
- Add – $C_{out} = C_0 + C_1$
- Subtract – $C_{out} = C_0 - C_1$
- Add signed – $C_{out} = C_0 + C_1 - 0.5$
- Blend/Interpolate/Decal – $C_{out} = C_0 * C_1 + C_2 * C_3$
- Dot product – $R = G = B = 4.0 * ((R_0 - 0.5) * (R_1 - 0.5) + (G_0 - 0.5) * (G_1 - 0.5) + (B_0 - 0.5) * (B_1 - 0.5))$

Alpha combine functions are:

- Replace – $A_{out} = A_0$
- Modulate – $A_{out} = A_0 * A_1$
- Add – $A_{out} = A_0 * A_1$
- Subtract – $A_{out} = A_0 - A_1$
- Add signed – $A_{out} = A_0 + A_1 - 0.5$
- Blend – $A_{out} = A_0 * A_1 + A_2 * A_3$
- Dot product – (Same value as for color)

6.0: Color sum

Combine texels with shades.

Secondary (specular) shade is combined with final texture color. This is simple addition.

Color sum parameters:

- bool enable_color_sum

7.0: Fog

Interpolate between fragment and fog value based on depth.

Input: fragment data and float25 F

Compute: $F_2 = \text{clamp}(F)$

Compute: $C_{new_frag} = (1 - F_2) * C_{old_frag} + F_2 * C_{fog_color}$

- A fog depth F, which was computed in the rasterizer, is divided by W, producing F₂, which is a linear function of world Z.
- For nonlinear fog, some math or lookup will need to occur.
- It appears that F may exceed the range [0..1], so I will have to clamp it.
- Dividing by W can be omitted if necessary.

Fog parameters:

- Fog color constant
- bool enable_fog

8.0: Alpha test

Alpha value is compared against constant. Fragments which fail test are dropped.

For the purposes of this test, Alpha is clamped, converted to fixed-point, and rounded.

Alpha test comparison modes:

- Equal
- Not equal
- Greater than
- Less than
- Greater or equal
- Less than or equal

Alpha test parameters:

- float25 AlphaReference
- AlphaTestFunc
- bool enable_alpha_test

9.0: Stencil and Depth read

Read framebuffer store of stencil and depth values. Stencil is an 8-bit unsigned integer. Depth is stored as a 24-bit float. Depth values aren't really Z but rather W.

9.1: Stencil and depth test

First, the stencil test is performed. The comparisons that can be made are:

- $(\text{stencil_val} \ \& \ \text{mask}) == (\text{StencilRef} \ \& \ \text{mask})$
- $(\text{stencil_val} \ \& \ \text{mask}) != (\text{StencilRef} \ \& \ \text{mask})$
- $(\text{stencil_val} \ \& \ \text{mask}) < (\text{StencilRef} \ \& \ \text{mask})$
- $(\text{stencil_val} \ \& \ \text{mask}) > (\text{StencilRef} \ \& \ \text{mask})$
- $(\text{stencil_val} \ \& \ \text{mask}) \leq (\text{StencilRef} \ \& \ \text{mask})$
- $(\text{stencil_val} \ \& \ \text{mask}) \geq (\text{StencilRef} \ \& \ \text{mask})$
- Always pass
- Always fail

If the stencil test fails, the StencilOpFail operator is used on the stencil buffer, the depth test is skipped, and the fragment is discarded.

If the stencil test is disabled or it passes, then DepthOpPass is assumed and then the depth test is performed.

The depth test comparisons are the same set as for stencil. The comparison is between the depth value from the buffer and the W of the fragment.

If the depth test fails, the fragment is discarded, and the DepthOpFail operator is used on the stencil buffer.

If the depth test passes or the depth test is disabled, the depth field of the stencil buffer is updated with the fragment W and the DepthOpPass operator is used.

These operators are available for modifying the stencil field of the stencil/depth buffer:

- Zero
- Replace – stencil_val = StencilRef
- Increment with clamp
- Decrement with clamp
- Invert
- Increment with wrap
- Decrement with wrap
- Keep original value

Stencil/Depth parameters:

- uint32 Stencil buffer base address
- uint16 Stencil pitch
- StencilFunc, DepthFunc
- StencilOpFail, DepthOpFail, DepthOpPass
- enable_stencil_read, enable_stencil_write
- enable_stencil_test, enable_depth_test
- enable_depth_write – affects replacement of stencil W with fragment W
- StencilRef
- StencilMask
- StencilConst – what is “read” from stencil buffer if read is disabled

9.2: Stencil and depth write

Write updates to buffer.

10.0: Destination read

Fetch target pixels from framebuffer.

Parameters to this unit:

- Constant for when read is disabled
- uint32 Dest base address
- uint16 Dest pitch (width in pixels)
- Logic ROP
- bool enable_dest_read, enable_dest_write
- Planemask
- BlendSourceColorFunc
- BlendSourceAlphaFunc
- BlendDestColorFunc
- BlendDestAlphaFunc
- BlendEquationMode
- Blend constant color

Compute: $\text{Fragment_address} = \text{Base_address} + X + \text{Pitch} * Y$

Converting X and Y to an address will probably be done higher up in the pipeline. Past that point, only the 32-bit address will be forwarded.

10.1: Merge

To blend source and dest pixels, a BlendModeEquation is chosen. The operands to the equation are also selectable.

Blend color factors:

- Zero
- Source color (from stencil unit)
- Destination color (read from framebuffer)
- Source alpha, replicated to RGB
- Destination alpha, replicated to RGB
- Constant color
- Constant alpha, replicated to RGB
- $\min(\text{SourceAlpha}, 1.0 - \text{DestAlpha})$

Optionally, any of those factors can be inverted ($1.0 - X$).

Blend alpha factors:

- Zero
- Source alpha
- Destination alpha
- Constant alpha

Any of those factors can be inverted.

The blend factors are used in only some of the blend equations. The blend equations are:

- Source only
- $\text{Source} * \text{SourceFactor} + \text{Dest} * \text{DestFactor}$
- $\text{Source} * \text{SourceFactor} - \text{Dest} * \text{DestFactor}$
- $\text{Dest} * \text{DestFactor} - \text{Source} * \text{SourceFactor}$
- $\min(\text{Source}, \text{Dest})$
- $\max(\text{Source}, \text{Dest})$

The same functions are available independently for both color and alpha.

After blending, the blended result can be combined again with the original destination value according to a raster op and a plane mask. The ROPs are the same as in X11. The following bit of Verilog code describes the operation of the raster op and planemask.

```
module apply_logic_rop(clock, src, dst, rop, pmask, result);
input clock;
input [31:0] src, dst, pmask;
input [3:0] rop;
output [31:0] result;
reg [31:0] result;
```

```
integer i;
always @(posedge clock) begin
    for (i=0; i<32; i=i+1) begin
        result[i] <= pmask[i] ? rop[{src[i], dst[i]}] : src[i];
    end
end
endmodule
```

Finally, since the DRAM chips support byte masks, we will take advantage of them. They allow us to apply byte-oriented planemasks without requiring a framebuffer read.

10.2: Dest write

Store pixels in framebuffer.
[Note: need discussion on sync tokens.]

Note to self: Go back and be sure to add in missing 2D features.