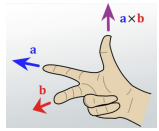
3GC3 Midterm - Solaris-3 Edition

Math

Dot Product:
$\mathbf{a}\cdot\mathbf{b} = x_a x_b + y_a y_b + z_a z_b = ||\mathbf{a}||||\mathbf{b}|| \cos\phi$

Cross Product: $\mathbf{a}\times\mathbf{b} = (y_a z_b - z_a y_b, z_a x_b - x_a z_b, x_a y_b - y_a x_b) = ||\mathbf{a}||||\mathbf{b}|| \sin(\theta)\mathbf{n}$



MatMul:

$$\begin{bmatrix}1&2&3\\4&5&6\end{bmatrix}\cdot\begin{bmatrix}7&8&9\\10&11&12\\13&14&15\end{bmatrix}=\begin{bmatrix}66&72&78\\156&171&186\end{bmatrix}$$

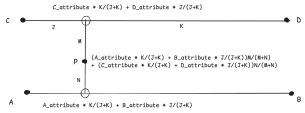
$C[1][1] = 1\cdot7 + 2\cdot10 + 3\cdot13 = 7 + 20 + 39 = 66$

Multiply each pair and add

Right Hand Rule: For axises, x is thumb, y is index, and z is middle finger.

Interpolation: t is how far along the line p_t is from p_0 to p_1 as a percentage between 0 and 1.
$t = (x_t - x_0)/(x_1 - x_0)$ or $t = (y_t - y_0)/(y_1 - y_0)$
$v_t = (1-t)v_0 + tv_1$

Bi-linear Interpolation



Barycentric Interpolation (Area):
$\alpha = A_a/A \quad \beta = A_b/A \quad \gamma = A_c/A \quad s.t.\ \alpha+\beta+\gamma = 1$
Now, $\mathbf{p}(\alpha,\beta,\gamma) = \alpha\mathbf{a} + \beta\mathbf{b} + \gamma\mathbf{c}$

Barycentric Interpolation (line function):
$f_{ij}(x,y) = (y_i - y_j)x + (x_j - x_i)y + x_i y_j - x_j y_i$
$\alpha = \frac{f_{bc}(x_P,y_P)}{f_{bc}(x_a,y_a)},\ \beta = \frac{f_{ac}(x_P,y_P)}{f_{bc}(x_b,y_b)},\ \gamma = \frac{f_{ab}(x_P,y_P)}{f_{ab}(x_c,y_c)}$

Transformations & Coordinate Systems

Linear Transformation:
$f(\begin{bmatrix}x\\y\end{bmatrix}) = \begin{bmatrix}a_{11}&a_{12}\\a_{21}&a_{22}\end{bmatrix}\begin{bmatrix}x\\y\end{bmatrix} = \begin{bmatrix}a_{11}x + a_{12}y\\a_{21}x + a_{22}y\end{bmatrix}$ satisfies:
$f(\mathbf{u}+\mathbf{v}) = f(\mathbf{u}) + f(\mathbf{v})$ and $f(c\mathbf{u}) = cf(\mathbf{u})$
In other words, origin is unchanged, straight lines remain straight lines.

Affine Transformation Straight lines remain lines.

2D Rotations

$$f(p) = x\begin{bmatrix}\cos\theta\\\sin\theta\end{bmatrix} + y\begin{bmatrix}-\sin\theta\\\cos\theta\end{bmatrix} = \begin{bmatrix}\cos\theta&-\sin\theta\\\sin\theta&\cos\theta\end{bmatrix}\begin{bmatrix}x\\y\end{bmatrix}$$

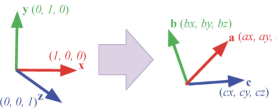
3D Rotations
Note: All orthonormal matrices are rotation matrices.

Rotate around X $f(p) = \begin{bmatrix}1&0&0\\0&\cos\theta&-\sin\theta\\0&\sin\theta&\cos\theta\end{bmatrix}\begin{bmatrix}x\\y\\z\end{bmatrix}$

Rotate around Y $f(p) = \begin{bmatrix}\cos\theta&0&\sin\theta\\0&1&0\\-\sin\theta&0&\cos\theta\end{bmatrix}\begin{bmatrix}x\\y\\z\end{bmatrix}$

Rotate around Z $f(p) = \begin{bmatrix}\cos\theta&-\sin\theta&0\\\sin\theta&\cos\theta&0\\0&0&1\end{bmatrix}\begin{bmatrix}x\\y\\z\end{bmatrix}$

For a generic 3D rotation: $R = \begin{bmatrix}ax&bx&cx\\ay&by&cy\\az&bz&cz\end{bmatrix}$ Where the $a*$ column is the coordinates of the object's x' vector, the $b*$ column is the y' vector, and the $c*$ column is the z' vector:

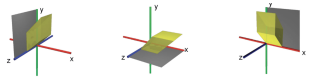


2D Reflection Reflect X: $\begin{bmatrix}-1&0\\0&1\end{bmatrix}$ Reflect Y: $\begin{bmatrix}1&0\\0&-1\end{bmatrix}$

3D Scaling $\begin{bmatrix}s_x&0&0\\0&s_y&0\\0&0&s_z\end{bmatrix}\begin{bmatrix}x\\y\\z\end{bmatrix} = \begin{bmatrix}s_x x\\s_y y\\s_z z\end{bmatrix}$

2D Shear Matrix Horizontal shear (top edge is shifted to the right (positive)): $\begin{bmatrix}1&s_x\\0&1\end{bmatrix}$ Vertical shear (right edge is shifted up (positive)): $\begin{bmatrix}1&0\\s_y&1\end{bmatrix}$

3D Shear Matrix



Shear on YZ Plane: $\begin{bmatrix}1&0&0\\s_y&1&0\\s_z&0&1\end{bmatrix}$

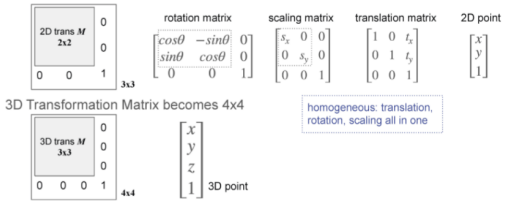
Shear on XZ Plane: $\begin{bmatrix}1&s_x&0\\0&1&0\\0&s_z&1\end{bmatrix}$

Shear on XY Plane: $\begin{bmatrix}1&0&s_x\\0&1&s_y\\0&0&1\end{bmatrix}$

Homogenous Coordinate 2D points represented by (x,y,z), 2D location is $(x/z, y/z) \mid z = 1$ or other value 3D points represented by (x,y,z,w), 3D location is $(x/w, y/w, z/w) \mid w = 1$ or other value Used for perspective projection, z becomes depth value.

Transformation Matrix

2D Transformation Matrix becomes 3x3



Composite Transformation Since scaling, rotation, etc is about origin, operations are not commutive!

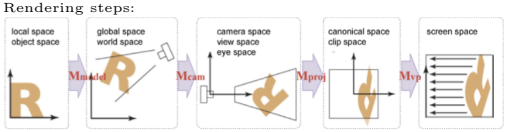
Examples To rotate around an object's centre, 1 translate object centre to origin, 2 rotate, 3 translate back.
To scale an object along a non uniform axis, 1 rotate the object to align with a canonical axis, 2 scale object, 3 rotate object back.

Inversions For rotation, scaling, and translating, the inverse of a matrix M^{-1} can be used. For rotation specifically, $M^{-1}_{rotate} = M^{T}_{rotate}$ For scaling, inversion is essentially $1/s$ for your scale factor. For translating, do $-x$.

Coordinate Systems *World/Global Coordinate* Only one, unique. Each model in scene goes through M_{model} to transform from model space to world space.

Camera / Viewing Transformations

Rendering steps:

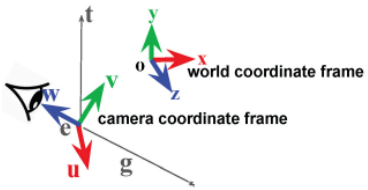


$$\begin{bmatrix}x_{screen}\\y_{screen}\\z\\1\end{bmatrix} = \mathbf{Mvp}\,\mathbf{Mproj}\,\mathbf{Mcam}\begin{bmatrix}x_{world}\\y_{world}\\z_{world}\\1\end{bmatrix}$$

to canonical space
to camera space

Viewing General Steps
1. Model 3D Objects in local space
2. Put 3D object at world coordinates
3. View scene in Camera space
4. Project camera space into cannonical space
5. Transform cannonical space to screen

World Space → Camera Space: M_{cam} Given camera position e, gaze direction g and 'up' direction t, construct basis uvw for camera coordinate system.
$w = -\frac{g}{||g||},\ u = \frac{t\times w}{||t\times w||},\ v = w\times u$



Camera Space → World Space:

$$\begin{bmatrix}1&0&0&x_e\\0&1&0&y_e\\0&0&1&z_e\\0&0&0&1\end{bmatrix}\begin{bmatrix}x_u&x_v&x_w&0\\y_u&y_v&y_w&0\\z_u&z_v&z_w&0\\0&0&0&1\end{bmatrix}$$

World Space → Camera Space: $M_{cam} =$

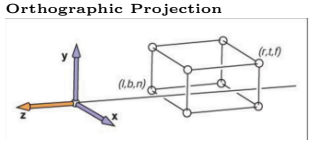
$$\begin{bmatrix}x_u&y_u&z_u&0\\x_v&y_v&z_v&0\\x_w&y_w&z_w&0\\0&0&0&1\end{bmatrix}\begin{bmatrix}1&0&0&-x_e\\0&1&0&-y_e\\0&0&1&-z_e\\0&0&0&1\end{bmatrix}$$

Step 1 (left matrix): Translates camera position e to origin e. Things originally at e are now at $\vec{0}$.
Step 2 (right matrix): Rotates, maps basis vectors: $u \mapsto (1,0,0), v \mapsto (0,1,0), w \mapsto (0,0,1)$

3×3 Camera Matrix $M_{cam} = [x_{cam}\quad y_{cam}\quad z_{cam}]$
Use column vectors
Step 1 (left matrix): Translates camera position e to origin e. Things originally at e are now at $\vec{0}$.
Step 2 (right matrix): Rotates, maps basis vectors: $u \mapsto (1,0,0), v \mapsto (0,1,0), w \mapsto (0,0,1)$

Cannonical Space
Cannonical space is the space (x,y,z) s.t. $x,y,z \in [-1,1]$.

Orthographic Projection



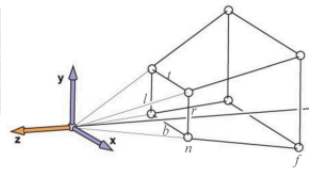
Given left plane x, l, right plane x, r, top plane y, t, bottom plane y b, near plane n, far plane f. *Recall that f, n are negative z values. See diagram.*
M_{orth} projects the view box defined above onto the cannonical space.

$$M_{orth} = \begin{bmatrix}\frac{2}{r-l}&0&0&0\\0&\frac{2}{t-b}&0&0\\0&0&\frac{2}{n-f}&0\\0&0&0&1\end{bmatrix}\begin{bmatrix}1&0&0&-\frac{r+l}{2}\\0&1&0&-\frac{t+b}{2}\\0&0&1&-\frac{n+f}{2}\\0&0&0&1\end{bmatrix}$$

$$= \begin{bmatrix}\frac{2}{r-l}&0&0&-\frac{r+l}{r-l}\\0&\frac{2}{t-b}&0&-\frac{t+b}{t-b}\\0&0&\frac{2}{n-f}&-\frac{f+n}{n-f}\\0&0&0&1\end{bmatrix}$$

Right: translate center to origin
Left: scales width of all dimensions to be 2, fitting inside [-1, 1].

Perspective Projection



l, r, b, t now define the near plane XY, with n being the Z. Far plane is only defined by f. **OR** use depth of field: θ = FOV on y-axis, $ratio = (r-l)/(t-b)$, n is near plane z, f is far plane z.

Frustrum to Box: Recall Homogenous Coordinates. We want frustrum $\mapsto$ box s.t. n stays near and f stays far.

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

$$M_{per} = M_{orth}P = \begin{bmatrix}\frac{2n}{r-l}&0&\frac{l+r}{l-r}&0\\0&\frac{2n}{t-b}&\frac{b+t}{b-t}&0\\0&0&\frac{f+n}{n-f}&\frac{2fn}{n-f}\\0&0&1&0\end{bmatrix}$$

Cannonical Space To View Port (VP)

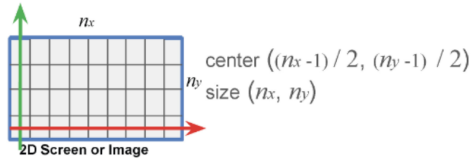


Diagram of viewport, which has no z depth.
1. Translate x,y center from $(0,0)$ to $((n_x-1)/2, (n_y-1)/2)$. 2. Scale x,y size from $(2,2)$ to (n_x, n_y).

$$M_{vp} = \begin{bmatrix}\frac{n_x}{2}&0&0&\frac{n_x-1}{2}\\0&\frac{n_y}{2}&0&\frac{n_y-1}{2}\\0&0&1&0\\0&0&0&1\end{bmatrix}$$

Rasterization

Definition: finding all pixels on the screen that are occupied by a geometric primitive.

Line Rasterization Given pixels $P_0 = (x_0, y_0), P_1 = (x_1, y_1)$, fill the pixels on the screen between them.
$f(x,y) : y = mx + c$
given $m = (y_1 - y_0)/(x_1 - x_0), c = (x_1 y_0 - x_0 y_1)/(x_1 - x_0)$

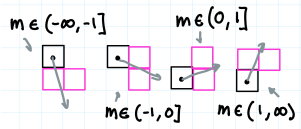
Implicit line function:: $f(x,y) : Ax + By + C = 0$.
Checking sign of $f(x_P, y_P)$ can determine point position relative to line. If $A > 0 \wedge B < 0$, then $f(x_P, y_P) < 0$ means point P is above the line.

Naive Implementation: Use the function $y = mx + c$ and iterate x from x_0 to x_1. No vertical lines!

Digital Difference Analyser DDA: Precomputes step size for both Y and X from line function. Marches in direction based on which value changes slower: if y changes slower march in the x direction and vice versa.

```
drawDDA(p0, p1):
    ranges = p1-p0
    numSteps = (ranges.x > ranges.y)
        ? ranges.x
        : ranges.y
    delta = ranges / numSteps
    p = p0
    for i in 0..numSteps:
        fill pixel (round(p.x), round(p.y))
        p += delta
```

Midpoint Algorithm: Assume $x_0 < x_1$. Otherwise, swap P_0, P_1. Seperate cases based on m either $\in \{(-\infty,-1], (-1,0], (0,1], (1,\infty)\}$. The case will affect which pixels are the candidate pixels.



For following cases, assume $m \in [0,1)$.
Assume (x,y) is drawn. Compute the midpoint of the next candidates, in this case $(x+1, y+0.5)$
To check if point is above line, calculate if $f(x+1, y+0.5) > 0$. Otherwise, count it as below. Select the candidate and draw it.

```
drawMidpointTraditional(x0, y0, x1, y1):
    y = y0
    for x=x0; x<=x1; x++:
        fill pixel (x,y)
        float f = (y0-y1)*(x+1)
                + (x1-x0)*(y+0.5)
                + x0*y1
                - x1*y0
        if (f < 0):
            y++;
```

Midpoint Incremental Algorithm: We don't need to evaluate f at every case, since m is constant. evaluate f at the first midpoint. We know that our next candidates are either on the same level or above the last candidate. If we increase diagonal, then $f(x+1, y+1) = f(x,y) + (y_0 - y_1) + (x_1 - x_0)$. If we increase horizontal, then $f(x+1, y) = f(x,y) + (y_1 - y_0)$

```
drawMidpointIter(x0, y0, x1, x0):
    p = p0
    f = (y0 - y1)*(x0 + 1)
        + (x1 - x0)*(y0+0.5)
        + x0*y1
        - x1*y0
    y = y0
    for x = x0; x <= x1; x++:
        fill pixels[x][y]
        if (f < 0): //line above M
            f += (x1-x0) + (y0-y1)
            y++;
        else: //line below M
            f += y0-y1
```

Bresenham Algorithm:

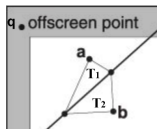
```
drawBresenham(x0, y0, x1, y1):
    deltaX = x1-x0
    deltaY = y1-y0
    D = 2*deltaY - deltaX
    fill pixel (x0,y0)
    y = y0
```

```
    for x=x0+1; x<=x1; x++:
        //Diagonal Increment
        if D > 0:
            y++
            D += 2*(deltaY-deltaX)
        //H Increment
        else:
            D += 2*deltaY
        fill pixel (x,y)
```

Triangle Rasterization
First, we find the bounding box (rectangle) of the triangle using the min and max positions of all 3 vertexes. Then, we iterate over every pixel (n^2), checking if the barycentric coordinate of the pixel lies inside the triangle.
We can tell if it's outside if all coordinates are ≥ 0. If one is not, then it is outside, and we skip drawing.
The rest of the following code interpolates the colour of the pixels (p0 is red, p1 is green and p2 is blue).

```
void DrawTriangle(int x0, int y0,
        int x1, int y1,
        int x2, int y2,
        std::vector<std::vector<std::string>> &pixels) {

// Bounding box
int minx = std::min({x0, x1, x2});
int maxx = std::max({x0, x1, x2});
int miny = std::min({y0, y1, y2});
int maxy = std::max({y0, y1, y2});

// Precompute terms for barycentric coordinates
float F01 = (y0-y1)*x2 + (x1-x0)*y2 + x0*y1 - x1*y0;
float F12 = (y1-y2)*x0 + (x2-x1)*y0 + x1*y2 - x2*y1;
float F20 = (y2-y0)*x1 + (x0-x2)*y1 + x2*y0 - x0*y2;

// Vertex colors: p0 - Red, p1 - Green, p2 - Blue
int R0 = 255, G0 = 0, B0 = 0;  // p0 - Red
int R1 = 0, G1 = 255, B1 = 0;  // p1 - Green
int R2 = 0, G2 = 0, B2 = 255;  // p2 - Blue

// Loop over bounding box pixels
for (int x = minx; x <= maxx; ++x) {
    for (int y = miny; y <= maxy; ++y) {
        // Calculate barycentric coordinates
        float f01 = (y0-y1)*x + (x1-x0)*y + x0*y1 - x1*y0;
        float f12 = (y1-y2)*x + (x2-x1)*y + x1*y2 - x2*y1;
        float f20 = (y2-y0)*x + (x0-x2)*y + x2*y0 - x0*y2;
        float b0 = f12 / F12, b1 = f20 / F20, b2 = f01 / F01;
        // Check if inside the triangle
        if (b0 >= 0 && b1 >= 0 && b2 >= 0) {
            // Interpolate color
            int R = b0*R0 + b1*R1 + b2*R2;
            int G = b0*G0 + b1*G1 + b2*G2;
            int B = b0*B0 + b1*B1 + b2*B2;
            // Set pixel color
            pixels[y][x] = std::to_string(R) + ',' +
                           std::to_string(G) + ',' +
                           std::to_string(B);
}}}}
```

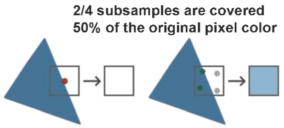
Non-overlapping Triangles
sharing edges: We want to ensure no-double drawing. Assume T_1, T_2 share one edge. Let a be the vertex of T_1 not along this edge. Let b be the coresponding vertex in T_2. Choose offscreen point q. T_1 should be responsible of drawing the edge if q falls on the same side of the edge as a. Likewise for T_2 and b.



Proper Perspective
Attribute Attribution: How we set up the above code leads to incorect attribute interpolation when taking perspective into account. This is because we are interpolating on the 2d projection of the triangle and not considering the distace of the 3d space. We can interpolate using the following code, which scales based on a scaling metric.

```
float Rs = 60*(R0/w0) + b1*(R1/w1) + b2*(R2/w2);
float Gs = b0*(G0/w0) + b1*(G1/w1) + b2*(G2/w2);
float Bs = 60*(B0/w0) + b1*(B1/w1) + b2*(B2/w2);
float Is = b0*(1/w0) + b1*(1/w1) + b2*(1/w2);
float R = Rs / Is;
float G = Gs / Is;
float B = Bs / Is;
```

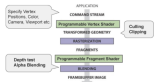
Anti-Aliasing
Supersample: Screen Goal is 256x256. We instead render x4, meaning we actually render a 1024x1024 image. Each highresolution pixel is considered a **fragment**. Then, on scale down, we can average the 16 virtual pixels into 1 screen pixel. Can use box filter or a gaussian filter.
Multisample: Screen goal is 256x256. We still rasterize for a higher resolution, but fragment (triangle) colour computation is only calculated once. Then, we sample n times from the pixel area, averaging the samples to get the true pixel colour.



Fragments here are each dot on a pixel.

Pipeline

Application



Main Program
Runs on CPU Defines geometry vertex positions, normals, texture coords, colours, etc. Sets up camera position, orientation, projection volume Sets screen size
Copies data to GPU

Vertex Shader

Per-vertex
Computation No transformation: Simply assigns input to output (pass-through shader) Transforming vertex: Apply $M_{proj} M_{cam} M_{model}$ to input Shading: determines vertex color (Gouraud)
GPU
performs parallel processing on each vertex

Culling

Backface Culling

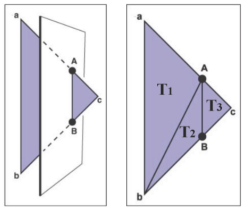
Removes
primitives facing away from camera
Look at face normal / right hand rule, face normal points in same direction as face.

View Frustrum Culling

Removes
geometries
outside
view
volume. 6 planes: near, far, left, right, top, bottom. Plane function is $f(p) = n \cdot (p - a) = n \cdot p + n \cdot a = n \cdot p + D = 0$ **Test if outside view volume** Take bounding box of object, e.g. a sphere with centre c, radius r. Check $f(c)$, c's signed distance to plane, see if it intersects or is within frustrum.

Clipping

View volume cuts
primitive to avoid drawing out of bounds
Clipping a Line *Plane Function*:
$f(p) = n \cdot p + D$ or $f(p) = n \cdot (p - c)$,
p is some point, n is the normal, D is a known const, c is a known point on the plane. If $f(p) = 0$, then p is on the plane.
Line Function: $p(t) = \mathbf{a} + t(\mathbf{b} - \mathbf{a})$
Intersection Point Plug p into plane function $f(p) = n \cdot (\mathbf{a} + t(\mathbf{b} - \mathbf{a})) + D = 0$
Solve for $t = \frac{n \cdot \mathbf{a} + D}{n \cdot (\mathbf{a} - \mathbf{b})}$
Clipping a Triangle
Plane Function: Same as above
Intersection:
Assume a, b is on one side, c is on the other side Compute intersection points $\mathbf{A}, \mathbf{B}$ using line clipping method.
Split Triangle
$\mathbf{T_1} = \triangle abA$, $\mathbf{T_2} = \triangle bBA$, $\mathbf{T_3}$
Throw Away If $f(c) \geq 0$, keep T_3; if $f(c) < 0$, keep $\mathbf{T_1}, \mathbf{T_2}$
Special Case
Handle zero-area triangles



Depth Testing

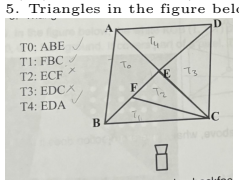
We need to order object rendering so things closer to camera appear 'ontop' of things futher away. Multiple primities can occupy the same fragment.
Painter's Algorithm: Sort primitive by their depths, draw primitives far to near. *Drawbacks:* Sorting is slow, many writes to buffer Occlusion cycle: cases where no correct order appears correct
Color and Z Buffer Two buffers, one for colour and one for depth. Draw primitives as they come in (no sorting), check z buffer (inited with ∞). If the primitive's depth is closer to the camera than what is there (smaller), update the z buffer and override the colour buffer.
Z Fighting: is caused by two primitives sharing the same z value. There are 2^n distinct values that z can be, where n is the number of bits for the depth value. *Precision Formula*: precision $= (z_{far} - z_{near})/2^b$ We want precision < max difference between z values *Mitigation Tactices*: Properly set the near and far planes increase bit count for depth value, Don't put objects too close to each other in scene.
Transparency / Alpha We can define a primitive's Transparency as $\alpha \in [0, 1]$, color now is RGBA.
src is the colour we want to write, $dest$ is the colour existing in the buffer.
Over Operation: is defined as $\alpha_{src} C_{src} + (1 - \alpha_{src}) C_{dest}$. This keeps buffer alpha after the operation=1.
Post-multiplication Set dest rgb using the over operation. $C_{src} = (R, G, B)$.
Pre-multiplication Premultiplied alpha has C_{src} already multiplied together with α_{src} when calculating the blending. Thus, we set dest rgb. $C_{src} = (R\alpha_{src}, G\alpha_{src}, B\alpha_{src})$.
$C_{dest} = C_{src} + (1 - \alpha_{src}) C_{dest}$.
Alpha and Depth Test
Zbuffer does not care if the fragment has Transparency – Fragments are not ordered. However- **ORDER matters when dealing with transparency!**. Thus, We draw all opaque objects first using the depth buffer, then use painters algorithm to draw transparent objects.

Example Questions

2. What are the limitations of painter's algorithm?

1. **Cyclic Overlap**: It fails with overlapping or cyclically overlapping polygons (e.g., A in front of B, B in front of C, and C in front of A).
2. **Intersecting Polygons**: It struggles with intersecting polygons, as ordering becomes ambiguous.
3. **High Sorting Cost**: Sorting polygons by depth is computationally expensive, slowing performance in complex scenes.
4. **Overdraw and No Depth Information**: Without per-pixel depth tracking, it may redraw pixels unnecessarily, reducing efficiency.
5. **Inaccurate Transparency**: It handles transparency poorly, as color compositing requires depth per pixel.

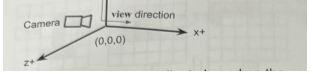
1. Given triangle ABC, Barycentric coord given by (1, 0, 0), (0, 1, 0), (0, 0, 1). Point P is (0.13, 0.17, 0.7) Since barycentric, the larger the number is to 1, the closer it is to that point in the triangle.
4. What is the composite transformation matrix to **revert** the following sequence of transformations to a 3D object?
First, the object is rotated around x-axis by rx; then it is scaled by s; then it is translated by (x,y,z) to origin; then it is rotated around y-axis by ry; lastly, the object is translated back to its original position;
$\mathbf{M^T}rotate - x(rx), \mathbf{M^{-1}}scale(s), \mathbf{M}translate(-x, -y, -z), \mathbf{M^T}rotate - y(ry), \mathbf{M^{-1}}translate(-x, -y, -z)$
5. Triangles in the figure below are described as:



Following counter-clockwise order, backface culling will remove which faces? (Use right hand rule where fingers curl and thumb points away from you to remove the faces).
Remove $T2, T3$
6. What is the purpose of the model matrix in the graphics pipeline?
Transform objects from local space to global world space.
The Digital Differential Analyzer (DDA) algorithm improves efficiency by incrementally calculating pixel positions using floating-point arithmetic, reducing redundant calculations. Unlike the naive approach, DDA avoids costly multiplication operations, making it faster for line rasterization. Additionally, it produces smoother lines by directly handling fractional increments.
8b. What is the main benefit of the Bresenham algorithm compared to the Midpoint Incremental line rasterization method?
The Bresenham algorithm optimizes line drawing by using only integer arithmetic, which is faster than the floating-point calculations of the Midpoint method. It also requires fewer computations per pixel, enhancing performance, and provides more accurate pixel positions for steeper line slopes.
9a. What is computed color of pixels without antialiasing?
Without antialiasing, the color of each pixel is typically computed based on the color at the center of the pixel.
9b. What is computed colors of pixels, using mutlisample antialiasing?
The final pixel color is the average of the sampled colors. This averaging smooths out jagged edges by blending colors from the edges of objects with the background, giving a smoother appearance without fully rendering the entire scene at a higher resolution.
11. Given line function: $f(x, y) = 3y - 2x - 1$, rasterize line segment from $(1, 1)$ to $(7, 5)$ using the midpoint algorithm. At the first step, pixel $(1, 1)$ is drawn. Mark the pixels that should be drawn and the midpoint sample positions evaluated: Midpoint can be between the two squares, either same height or above, based on line. Input the midpoint between the two points:
$M_1 = (2, 1.5)$, $f(M_1) = 3(1.5) - 2(2) - 1 = -0.5$ Since its negative, draw the above square in. Draw $(2, 2)$. Follow these steps.
12. Triangle ABC, where vertex A, B, C's barycentric coordinates are represented in (α, β, γ) are $(1, 0, 0)$, $(0, 1, 0)$, $(0, 0, 1)$. A,B,C's (x,y,z) positions in 3D space are $(0, 0, 0)$, $(10, 0, 0)$, $(0, 20, 0)$. Point P's barycentric coordinates are $(0.5, 0.15, \gamma_p)$. What is Point P position?
$\gamma_p = 1 - 0.5 - 0.15 = 0.35$, $P(0.5, 0.15, 0.35) = \alpha A + \beta B + \gamma C$. $P(0.5, 0.15, 0.35)$.
What are the areas of triangle ABC, PBC, PCA and PAB?
$ABC = bh/2 = 100$.
$\alpha = PBC/ABC, 0.5 = PBC/100, PBC = 50$.
$\beta = PCA/ABC, 0.15 = PCA/100, PBC = 15$.
$\gamma = PAB/ABC, 0.35 = PAB/100, PAB = 35$.
Transformation function $f(\begin{bmatrix} x \\ y \end{bmatrix})$ rotates input $\begin{bmatrix} x \\ y \end{bmatrix}$ clockwise by θ. Thus $f(\begin{bmatrix} 1 \\ 0 \end{bmatrix}) = \begin{bmatrix} cos(-\theta) \\ sin(-\theta) \end{bmatrix}$ i.e transform (1,0) to red point and $f(\begin{bmatrix} 1 \\ 0 \end{bmatrix}) = \begin{bmatrix} -sin(-\theta) \\ cos(-\theta) \end{bmatrix}$ i.e transform (0,1) to green point.
For any point p: $p = [x\ y] = x \begin{bmatrix} 1 \\ 0 \end{bmatrix} + y \begin{bmatrix} 0 \\ 1 \end{bmatrix}$. This $f(p) = f(\begin{bmatrix} x \\ y \end{bmatrix}) = x_p \begin{bmatrix} cos(-\theta) \\ sin(-\theta) \end{bmatrix} + y_p \begin{bmatrix} -sin(-\theta) \\ cos(-\theta) \end{bmatrix}$. Thus the clockwise rotation matrix is $\begin{bmatrix} cos(-\theta) - sin(-\theta) \\ sin(-\theta) cos(-\theta) \end{bmatrix} \begin{bmatrix} x_p \\ y_p \end{bmatrix}$.
14. Orthographic projection, view volume is defined as Left $l = -5$, Right $r = 5$, Top $t = 6$, Bottom $b = 1$, Near $n = -8$, Far $f = -28$. To convert this view volume to canonical space box $[-1, 1]$, what is the M_{orth} matrix?
$\frac{s_x}{r - l} = \frac{-1}{10}$, $\frac{s_y}{t - b} = \frac{1}{5}$, $\frac{s_z}{n - f} = \frac{1}{20}$.
$-(r + l)/s_x = 0$, $-(t + b)/s_y = 7$, $-(n + f)/s_z = 36$.

$$M_{orth} = \begin{bmatrix} \frac{-1}{10} & 0 & 0 & 0 \\ 0 & \frac{1}{5} & 0 & -0 \\ 0 & 0 & \frac{1}{20} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 6 \\ 0 & 1 & 0 & 7 \\ 0 & 0 & 1 & 36 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

15. Z-buffer and Color buffer, the closer z values (closer to 0) are closer to camera, so they overlap the farther colours, fill in the colours based on their distance from camera then away from camera. For color buffer it should be the updated colours based on the z-buffer.
16. A camera is put in world space (-1,0,0) looking toward (1,0,0) direction. The up direction is (0,1,0).



Contruct the camera coordinate based on the up and view direction. Specify the x,y,z axis and the origin of the camera coordinate in the world.
$w = -1 \cdot \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} -1 \\ 0 \\ 0 \end{bmatrix}$. $u = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} \cdot \begin{bmatrix} -1 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$.
$v = \begin{bmatrix} -1 \\ 0 \\ 0 \end{bmatrix} \cdot \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}$.
Based on the camera coordinate in world space, write out 4×4 matrix transforming from camera space to world space?

$$M_{cam} = \begin{bmatrix} 100 & -1 & & \\ & 0100 & & \\ & 0010 & & \\ & 0001 & & \end{bmatrix} \cdot \begin{bmatrix} 1001 & & & \\ & 0100 & & \\ & 0010 & & \\ & 0001 & & \end{bmatrix} = \begin{bmatrix} & 0010 & & \\ & 0100 & & \\ & -100 & -1 & \\ & 0001 & & \end{bmatrix}$$

For point p at world space $(4, 5, 6)$, what is p's coordinate in the camera space?

$$p = \begin{bmatrix} & 0010 & & \\ & 0100 & & \\ & -100 & -1 & \\ & 0001 & & \end{bmatrix} \cdot \begin{bmatrix} 4 \\ 5 \\ 6 \\ 1 \end{bmatrix} = \begin{bmatrix} 6 \\ 5 \\ -5 \\ 1 \end{bmatrix} = (6, 5, -5, 1)$$