Computer Graphics

Three Dimensional Viewing
Clipping and Perspective Division

Perspective Transformation and Homogeneous Coordinates

- We have used homogeneous coordinates for points and vectors. Now we want to extend them to work with perspective transformations.
- Point P for any w != 0 is given by $P = \begin{pmatrix} wP_x \\ wP_y \\ wP_z \\ w \end{pmatrix}$
  - Before we display a point, we divide all 4 coordinates by w.
  - Affine transformations do not change w, since the last row of an affine transformation matrix is (0, 0, 0, 1).

Perspective Transformation and Homogeneous Coordinates (2)

- For example, the point (1, 2, 3) has the representations (1, 2, 3, 1), (2, 4, 6, 2), (0.003, 0.006, 0.009, 0.001), (-1, -2, -3, -1), etc.
- To convert a point from *ordinary coordinates* to *homogeneous coordinates*, append a 1.
- To convert a point from *homogeneous coordinates* to *ordinary coordinates*, divide all components by the last component and discard the fourth component.

Perspective Transformation and Homogeneous Coordinates (3)

- Suppose we multiply a point in this new form by a matrix with the last row (0, 0, -1, 0). (The perspective projection matrix will have this form.)

$$\begin{pmatrix} N & 0 & 0 & 0 \\ 0 & N & 0 & 0 \\ 0 & 0 & a & b \\ 0 & 0 & -1 & 0 \end{pmatrix} \begin{pmatrix} wP_x \\ wP_y \\ wP_z \\ w \end{pmatrix} = \begin{pmatrix} wNP_x \\ wNP_y \\ w(aP_z + b) \\ -wP_z \end{pmatrix}$$

Perspective Transformation and Homogeneous Coordinates (4)

- The resulting point corresponds (after dividing through by the 4th component) to
  (x*, y*, z*) = $(N\frac{P_x}{-P_z}, N\frac{P_y}{-P_z}, \frac{aP_z + b}{-P_z})$
- Using homogeneous coordinates allows us to capture perspective using a matrix multiplication.
- To make it work, we must always divide through by the fourth component, a step which is called **perspective division**.

Perspective Transformation and Homogeneous Coordinates (5)

- A matrix that has values other than (0,0,0,1) for its fourth row does not perform an affine transformation. It performs a more general class of transformation called a **perspective transformation**.
- It is a transformation, not a projection. A projection reduces the dimensionality of a point, e.g., from a 3D point to a 2-tuple point, whereas a perspective transformation takes a 4-tuple and produces a 4-tuple.

Perspective Transformation and Homogeneous Coordinates (6)

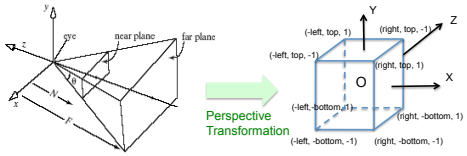
- Where does the projection part come into play?
  - The first two components of this point are used for drawing: to locate in screen coordinates the position of the point to be drawn.
  - The third component is used for depth testing.
- As far as locating the point on the screen is concerned, ignoring the third component is equivalent to replacing it by 0; this is the projection part.

Perspective Transformation and Homogeneous Coordinates (7)

- **(perspective projection) = (perspective transformation) + (orthographic projection)**
- OpenGL does the transformation step separately from the projection step.
- It inserts clipping, perspective division, and one additional mapping between them.

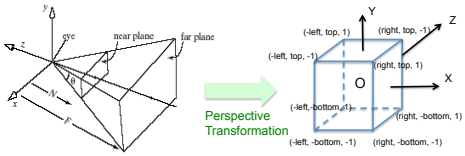
Geometry of Perspective Transformation

- The perspective transformation alters 3D point P into another 3D point, to prepare it for projection. It is useful to think of it as causing a warping of 3D space and to see how it warps one shape into another.
- Very importantly, it preserves straightness and flatness, so lines transform into lines, planes into planes, and polygonal faces into other polygonal faces.



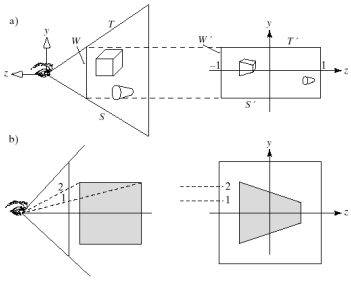
Geometry of Perspective Transformation

- It also preserves in-between-ness, so if point a is inside an object, the transformed point will also be inside the transformed object.
  - Our choice of a suitable pseudodepth function was guided by the need to preserve these properties.



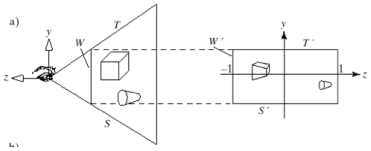
Geometry of Perspective Transformation

- How does it transform the camera view volume?
- We must clip against this volume.



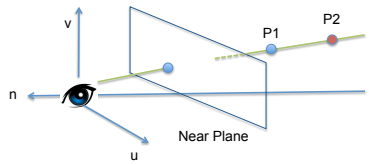
Geometry of Perspective Transformation

- The near plane at $z = -N$ maps into the plane at $z = -1$, and the far plane maps to the plane at $z = +1$.
- The top and bottom wall are tilted into the horizontal planes so that they are parallel to the z-axis.
- The two side walls become parallel to the z-axis.
- The camera's view volume is transformed into a parallelepiped.



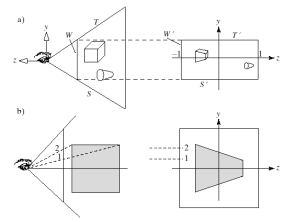
Geometry of Perspective Transformation

- Lines through the eye map into lines parallel to the z-axis.
  - **Proof**: All points of such a line project to a single point, say (x^*, y^*), on the viewplane. So all of the points along the line transform to all of the points (x, y, z) with $x = x^*$, $y = y^*$, and z taking on all pseudodepth values between -1 and 1.



Geometry of Perspective Transformation

- Lines perpendicular to the z-axis map to lines perpendicular to the z-axis.
  - **Proof:** All points along such a line have the same z-coordinate, so they all map to points with the same pseudodepth value.

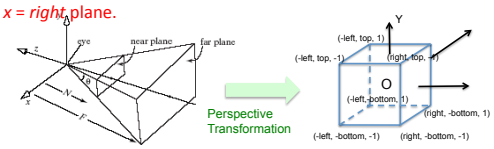


Geometry of Perspective Transformation

- The transformation also warps objects into new shapes.
- *The perspective transformation warps objects so that, when viewed with an orthographic projection, they appear the same as the original objects do when viewed with a perspective projection.*

Geometry of Perspective Transformation

- We want to put some numbers on the dimensions of the view volume before and after it is warped.
- Consider the top plane, and suppose it passes through the point (*left*, *top*, *-N*) at $z = -N$.
- It is composed of lines that pass through the eye and through points in the near plane all of which have a y-coordinate of *top, so* it must transform to the *y = top plane* . Similarly, the bottom plane transforms to the *y = bott* plane; the left plane transforms to the *x = left* plane; and the right plane transforms to the *x = right* plane.

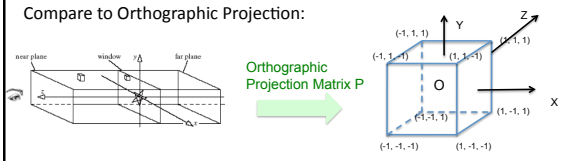


Geometry of Perspective Transformation (6)

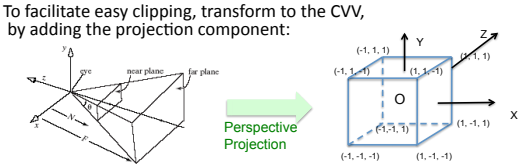
- We now know the transformed view volume precisely: a parallelepiped with dimensions that are related to the camera's properties in a very simple way.
- This is a splendid shape to clip against as we shall see, because its walls are parallel to the coordinate planes, but it would be even better for clipping if its dimensions didn't depend on the particular camera being used.
- OpenGL composes the perspective transformation with another mapping that scales and translates this parallelepiped into the **canonical view volume (CVV)**, a cube that extends from -1 to 1 in each dimension.
- Because this scales things differently in the x- and y- dimensions, it introduces some distortion, but the distortion will be eliminated in the final viewport transformation.

Geometry of Perspective Transformation (7)

Compare to Orthographic Projection:



To facilitate easy clipping, transform to the CVV, by adding the projection component:



Perspective Projection Matrix used by OpenGL

Perspective Projection Matrix
*= Perspective Transformation * Scale * Translation*

$$= \begin{pmatrix} N & 0 & 0 & 0 \\ 0 & N & 0 & 0 \\ 0 & 0 & a & b \\ 0 & 0 & -1 & 0 \end{pmatrix} * \begin{pmatrix} \frac{2}{right-left} & 0 & 0 & 0 \\ 0 & \frac{2}{top-bottom} & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} * \begin{pmatrix} 1 & 0 & 0 & -\frac{left+right}{2} \\ 0 & 1 & 0 & -\frac{top+bottom}{2} \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$= \begin{pmatrix} \frac{2N}{right-left} & 0 & \frac{right+left}{right-left} & 0 \\ 0 & \frac{2N}{top-bottom} & \frac{top+bottom}{top-bottom} & 0 \\ 0 & 0 & a & b \\ 0 & 0 & -1 & 0 \end{pmatrix}, \quad where\ a = \frac{-(F+N)}{F-N} \quad and\ b = \frac{-2FN}{F-N}$$

Compare against the Orthographic Project Matrix

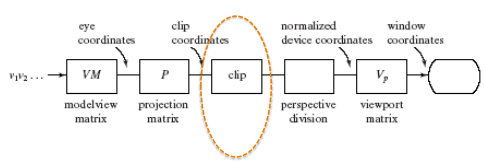
- Translate the view volume to center at the origin
- Scale the view volume to x:[-1, 1], y:[-1, 1], z: [-1:1]

$$P = ST = \begin{bmatrix} \frac{2}{right-left} & 0 & 0 & 0 \\ 0 & \frac{2}{top-bottom} & 0 & 0 \\ 0 & 0 & \frac{-2}{far-near} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & -\frac{left+right}{2} \\ 0 & 1 & 0 & -\frac{top+bottom}{2} \\ 0 & 0 & 1 & -\frac{far+near}{2} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$P = \begin{bmatrix} \frac{2}{right-left} & 0 & 0 & -\frac{right+left}{right-left} \\ 0 & \frac{2}{top-bottom} & 0 & -\frac{top+bottom}{top-bottom} \\ 0 & 0 & \frac{-2}{far-near} & -\frac{far+near}{far-near} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

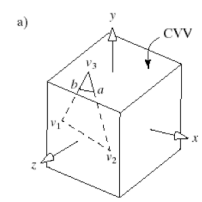
Projections of 3-D Objects

- The graphics pipeline: vertices start in world coordinates; after MV, in eye coordinates, after P, in clip coordinates; after perspective division, in normalized device coordinates; after V, in screen coordinates.



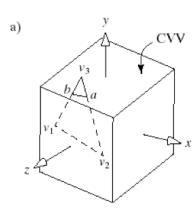
Clipping Against the View Volume

- The view volume is bounded by 6 infinite planes. We clip to each in turn.
- Example: A triangle has vertices v_1, v_2, and v_3. v_3 is outside the view volume, *CVV*.
- The clipper first clips edge v_1v_2, finding the entire edge is inside *CVV*.



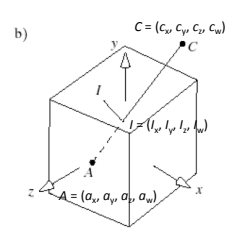
Clipping Against the View Volume (2)

- Then it clips edge v_2v_3, and records the new vertex *a* formed where the edge exits from the *CVV*.
- Finally it clips edge v_3v_1 and records the new vertex where the edge enters the *CVV*.
- The original triangle has become a quadrilateral with vertices $v_1v_2a\ b$.
- We actually clip in the 4D homogeneous coordinate space called "clip coordinates", which will distinguish between points in front of and behind the eye.



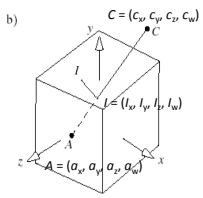
Clipping Method

- Suppose we want to clip the line segment *AC* against the CVV. This means we are given two points in homogeneous coordinates, $A = (a_x, a_y, a_z, a_w)$ and $C = (c_x, c_y, c_z, c_w)$, and we want to determine which part of the segment lies inside the *CVV*.
- If the segment intersects the boundary of the *CVV,* we will need to compute the intersection point $I = (I_x, I_y, I_z, I_w)$.



Clipping Method (2)

- We view the CVV as six infinite planes and consider where the given edge lies relative to each plane in turn.
- We can represent the edge parametrically as $A + (C\text{-}A)t$. It lies at A when t is 0 and at C when t is 1.
- For each wall of the *CVV,* we first test whether A and C lie on the same side of a wall; if they do, there is no need to compute the intersection of the edge with that wall.
- If they lie on opposite sides, we locate the intersection point and clip off the part of the edge that lies outside.

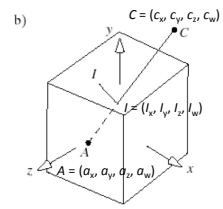


Clipping Method (3)

- So we must be able to test whether a point is on the *outside* or *inside* of a plane.
- For example, for plane $x = -1$. Point A lies to the right (on the inside) of this plane if

$a_x/a_w > -1$, or $a_x > -a_w$, or $a_w + a_x > 0$.

- Similarly A is inside the plane $x = 1$ if $a_x/a_w < 1$ or $a_w - a_x > 0$.
- We can use these to create boundary codes for the edge for each plane in the CVV.



Boundary Code Values

boundary coordinate	*homogeneous value*	*clip plane*
BC_0	$w + x$	$x = -1$
BC_1	$w - x$	$x = 1$
BC_2	$w + y$	$y = -1$
BC_3	$w - y$	$y = 1$
BC_4	$w + z$	$z = -1$
BC_5	$w - z$	$z = 1$

Boundary Code Values

Point inside CVV

boundary coordinate	*homogeneous value*		*clip plane*
BC_0	$w + x$		$x = -1$
BC_1	$w - x$		$x = 1$
BC_2	$w + y$	***> 0***	$y = -1$
BC_3	$w - y$		$y = 1$
BC_4	$w + z$		$z = -1$
BC_5	$w - z$		$z = 1$

Clipping Method (4)

- We calculate these six quantities for A and again for C.
- If all six are positive, the point lies inside the *CVV*.
- If any are negative, the point lies outside.
- If both points lie inside (outside), we have the same kind of "trivial accept (reject)" we had in the Cohen-Sutherland clipper.
  - *Trivial accept: both endpoints lie inside the CVV (all 12 BC's are positive).*
  - *Trivial reject: both endpoints lie outside the same plane of the CVV.*

Clipping Method (5)

- If neither condition holds, we must clip segment *AC* against each plane individually.
- Just as with the Cyrus-Beck clipper, we keep track of a **candidate interval** (*CI*), an interval of time during which the edge might still be inside the *CVV*.
- Basically we know the converse: if t is outside the *CI* we know for sure the edge is *not* inside the *CVV*. The *CI* extends from $t = t_{in}$ to t_{out}.

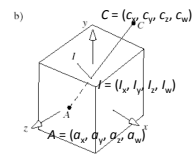
Clipping Method (6)

- We test the edge against each wall in turn.
- If the corresponding boundary codes have opposite signs, we know the edge hits the plane at some t_{hit}, which we then compute.
  - If the edge is entering (is moving into the inside of the plane as *t* increases), we update t_{in} = *max*(*old* t_{in}, t_{hit}).
  - Similarly, if the edge is exiting, we update t_{out} = *min*(*old* t_{out}, t_{hit}).
- If, at any time the *CI* is reduced to the empty interval ($t_{out} > t_{in}$), we know the entire edge is clipped off and we have an "early out", which saves unnecessary computation.

Finding t_{hit}

- Write the edge parametrically using homogeneous coordinates:
- *edge*(*t*) = ($a_x + (c_x - a_x)\,t$, $a_y + (c_y - a_y)\,t$, $a_z + (c_z - a_z)\,t$, $a_w + (c_w - a_w)\,t$)
- Using the *X* = 1 plane, for instance, when the *x*-coordinate of *A* + (*C*-*A*)*t* is 1:

$$\frac{a_x + (c_x - a_x)t}{a_w + (c_w - a_w)t} = 1$$



Finding t_{hit} (2)

- This is easily solved for *t*, yielding

$$t = \frac{a_w - a_x}{(a_w - a_x) - (c_w - c_x)}$$

boundary coordinate	*homogeneous value*	*clip plane*
BC_0	$w + x$	$w = -1$
BC_1	$w - x$	$x = 1$
BC_2	$w + y$	$y = -1$
BC_3	$w - y$	$y = 1$
BC_4	$w + z$	$z = -1$
BC_5	$w - z$	$z = 1$

- Note that t_{hit} depends on only two boundary coordinates:

$$t = \frac{aBC_1}{aBC_1 - cBC_1}$$

- Intersection point *I* can be computed as *I*=*A*+*Ct*
- Intersection with other planes yield similar formulas.

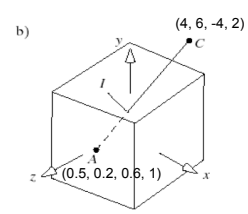
Clipping Method (7)

- This is the **Liang Barsky** algorithm with some refinements suggested by Blinn.
- clipEdge(Point4 A, Point4 C) takes two points in homogeneous coordinates (having fields *x*, *y*, *z*, and *w*) and returns 0 if no part of *AC* lies in the *CVV*, and 1 otherwise.
- It also alters *A* and C so that when the routine is finished *A* and *C* are the endpoints of the clipped edge.

Clipping Method (8)

- The routine finds the six boundary coordinates for each endpoint and stores them in aBC[ ] and cBC[ ].
- For efficiency, it also builds an **outcode** for each point, which holds the *signs* of the six boundary codes (left, right, bottom, top, front, back) for that point.
  - Bit *i* of *A*' s outcode holds a 0 if aBC[i] > 0 (*A* is inside the *i*-th wall) and a 1 otherwise.
  - For example: 101000 (outside, in the left bottom corner)
- Using these, a trivial accept occurs when both aOutcode and cOutcode are 0.
- A trivial reject occurs when the bit-wise AND of the two outcodes is nonzero, e.g., they are outside of at least one common plane.

Practice Question



Clip line AC against CVV

Practice Question

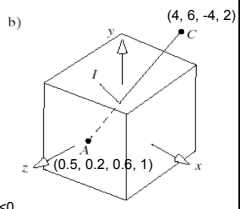
1.) aBC[0] =w+x=1.5 cBC[0] =w+x=6
aBC[1]=w-x=0.5 cBC[1]=w-x=-2
aBC[2]=w+y=1.2 cBC[2]=w+y=8
aBC[3]=w-y=0.8 cBC[3]=w-y=-4
aBC[4]=w+z=1.6 cBC[4]=w+z=-2
aBC[5]=w-z=0.4 cBC[5]=w-z=6

2.) Outcode for A: 000000
Outcode for C: 010110

3.) aOutcode & cOutcode ==0
aOutcode | cOutcode !=0

4.) Clip against each plane where aBC<0 or cBC<0
for i=1: t_{out}=aBC[1]/(aBC[1]-cBC[1]) = 0.5/(0.5-(-2))= 0.2
i=3: t_{out}=aBC[3]/(aBC[3]-cBC[3]) = 0.8/(0.8-(-4))= 0.167
i=4: t_{out}=aBC[4]/(aBC[4]-cBC[4]) = 1.6/(1.6-(-2))= 0.444

5.) Update C: C.x=A.x+t_{out}*(C.x-A.x)= 1.0845; C.y=A.y+t_{out}*(C.y-A.y)= 1.167;
C.z=A.z+t_{out}*(C.z-A.z)= -0.2682; C.w=A.w+t_{out}*(C.w-A.w)= 1.167.



Code for Clipper

```
int clipEdge(Point4& A, Point4& C)
{
    double tIn = 0.0, tOut = 1.0, tHit;
    double aBC[6], cBC[6]; // points A and C against 6 planes
    int aOutcode = 0, cOutcode = 0;    // 6 bits for each outcode

    <.. find BC's for A and C ..>
    <.. form outcodes for A and C ..>

    if((aOutcode & cOutcode) != 0) // trivial reject
        return 0;
    if((aOutcode | cOutcode) == 0) // trivial accept
        return 1;
```


Code for Clipper (2)

```
for (int i = 0; i < 6; i++) // clip against each plane
{
    if(cBC[i] < 0)  // exits: C is outside
    {
        tHit = aBC[i]/(aBC[i] - cBC[i]);
        tOut = MIN(tOut,tHit);
    }
    else if(aBC[i] < 0) //enters: A is outside
    {
        tHit = aBC[i]/(aBC[i] - cBC[i]);
        tIn = MAX(tIn, tHit);
    }
    if(tIn > tOut) return 0; //CI is empty early out
}   // end for
```

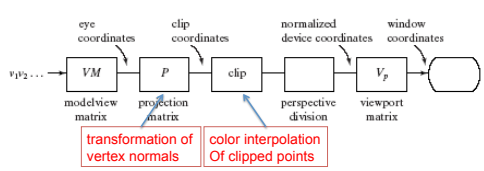

Code for Clipper (3)

```
    // update the end points as necessary
    Point4    tmp;
    if(aOutcode != 0) // A is out: tIn has changed; find updated A, don't change it yet)
    {
        tmp.x = A.x+ tIn*(C.x - A.x);      tmp.y = A.y + tIn*(C.y - A.y);
        tmp.z = A.z+ tIn*(C.z - A.z);      tmp.w = A.w+ tIn*(C.w - A.w);
    }

    if(cOutcode != 0)      // C is out: tOut has changed; update C (using original value of A)
    {
        C.x = A.x+ tOut *(C.x - A.x);     C.y = A.y + tOut*(C.y - A.y);
        C.z = A.z+ tOut*(C.z - A.z);      C.w = A.w+ tOut*(C.w - A.w);}
    }

    A = tmp;    // now update A
    return 1;    // some of the edge lies inside the CVV
} // end function clipEdge
```

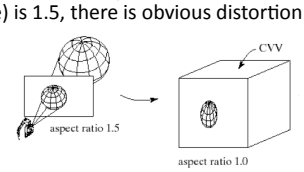

The Graphics Pipeline in OpenGL



- model in world coordinates → ModelView matrix (eye coordinates) → projection matrix (canonical view volume coordinates) → clipper → perspective division → viewport matrix →display (screen coordinates)

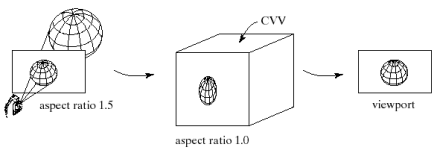
The Graphics Pipeline in OpenGL (2)

- Following clipping **perspective division** is finally done and the 3-tuple (*x*, *y*, *z*) is passed through the viewport transformation.
- The perspective transformation squashes the scene into the canonical cube. If the aspect ratio of the camera's view volume (that is, the aspect ratio of the window on the near plane) is 1.5, there is obvious distortion introduced.



Distortion and Its Removal

- The viewport transformation can undo this distortion by mapping a square into a viewport of aspect ratio 1.5. We normally set the aspect ratio of the viewport to be the same as that of the view volume.



- glViewport(x, y, width, height) specifies that the viewport will have lower left corner (x,y) in screen coordinates and will be width pixels wide and height pixels high. It thus specifies a viewport with aspect ratio width/height.

Steps in the Pipeline: Each Vertex P Undergoes Operations Below

- *P* is **extended** to a homogeneous 4-tuple by appending a 1, and this 4-tuple is multiplied by the **modelview matrix**, producing a 4-tuple giving the position in eye coordinates.
- The point is then multiplied by the **projection matrix**, producing a 4-tuple in clip coordinates.
- The edge having this point as an endpoint is **clipped**.
- **Perspective division** is performed, returning a 3-tuple.
- The **viewport transformation** multiplies the 3-tuple by a matrix; the result (s_x, s_y, d_z) is used for drawing and depth calculations. (s_x, s_y) is the point in screen coordinates to be displayed; d_z is a measure of the depth of the original point from the eye of the camera.
  - The viewport transformation also maps pseudodepth from the range -1 to 1 into the range 0 to 1.