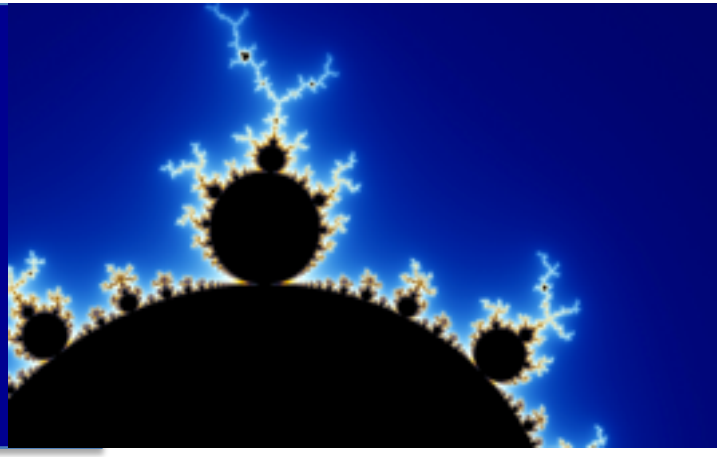
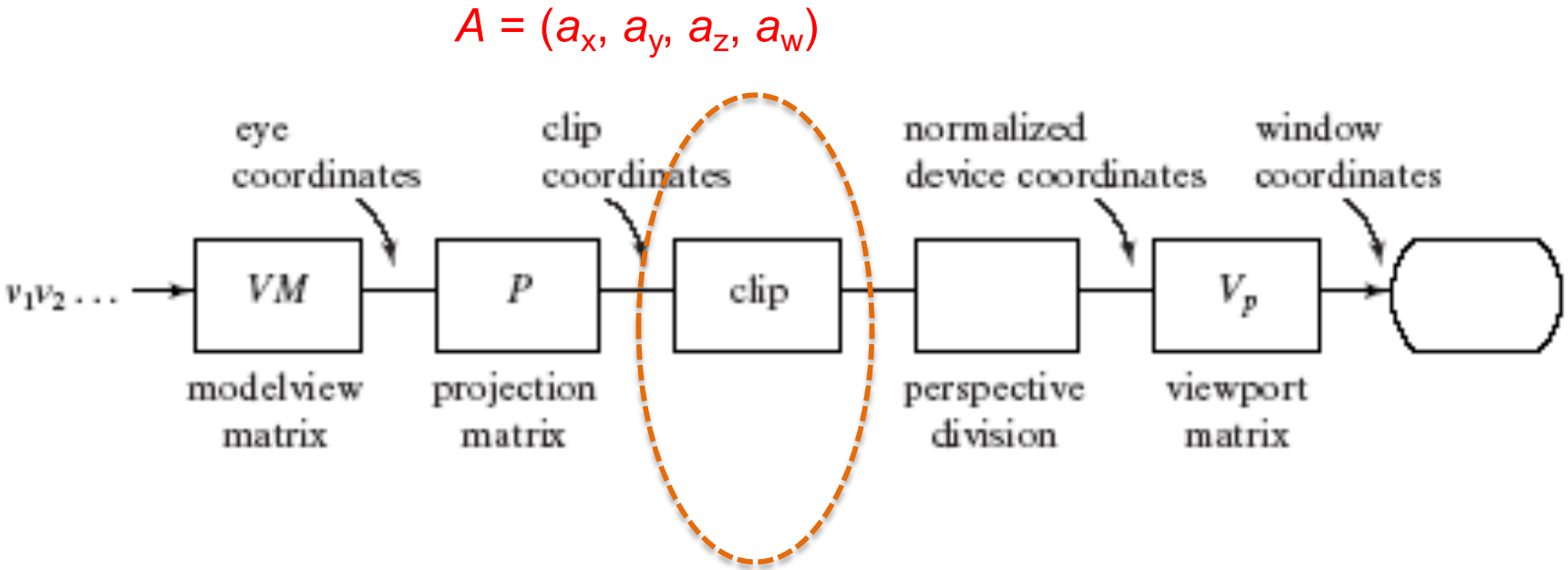
# Computer Graphics



## Clipping against the CVV

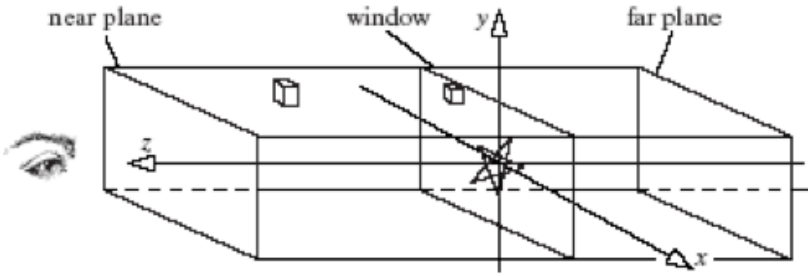
# Projections of 3-D Objects

- 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.

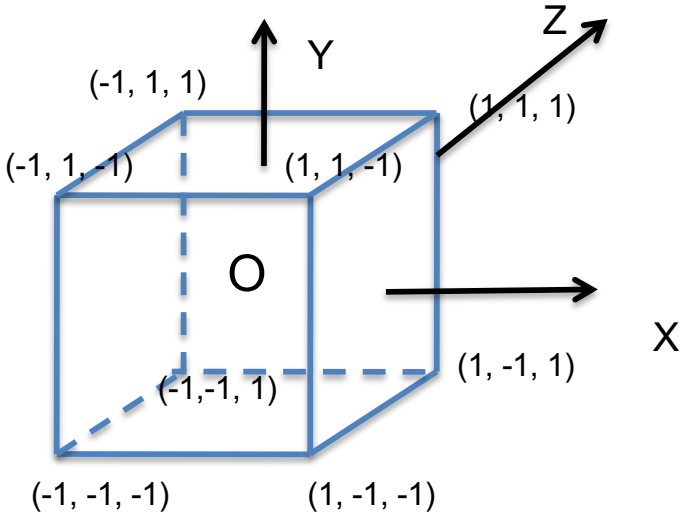


# Geometry of Perspective Transformation (7)

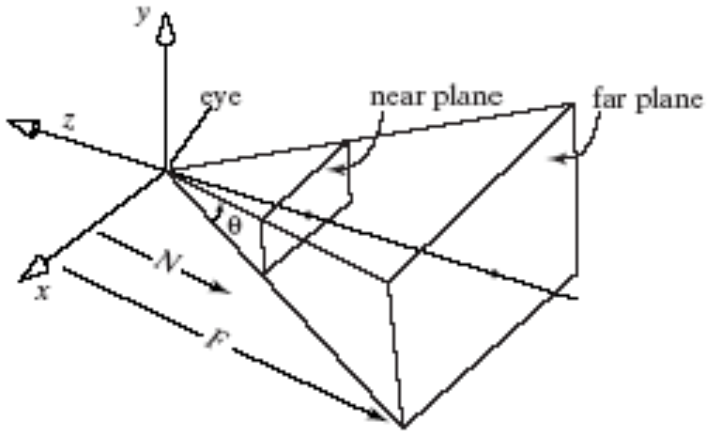
Compare to Orthographic Projection:



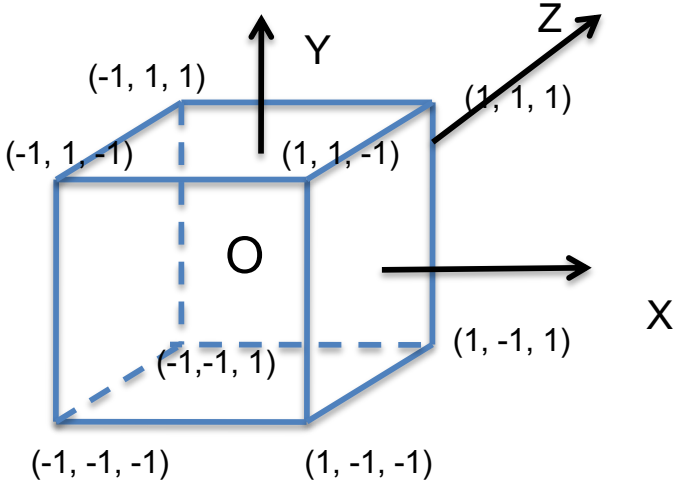

Orthographic Projection Matrix P




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

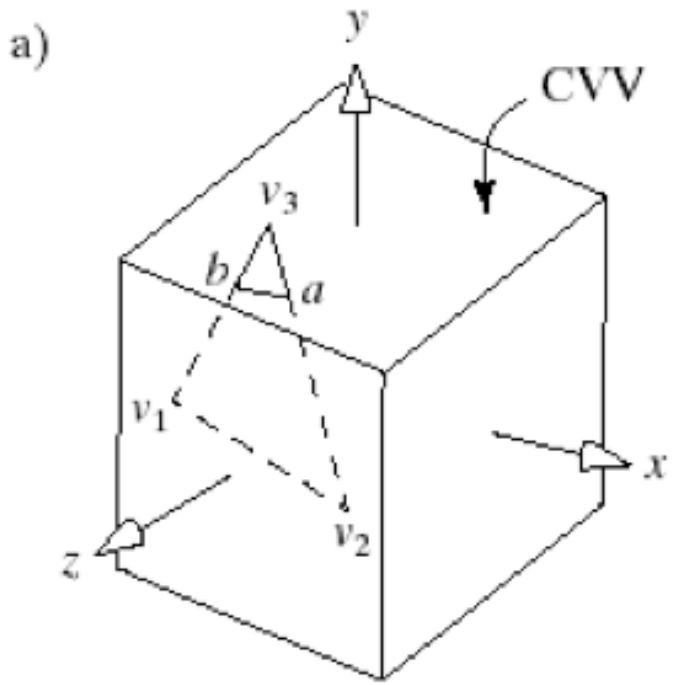



Perspective Projection



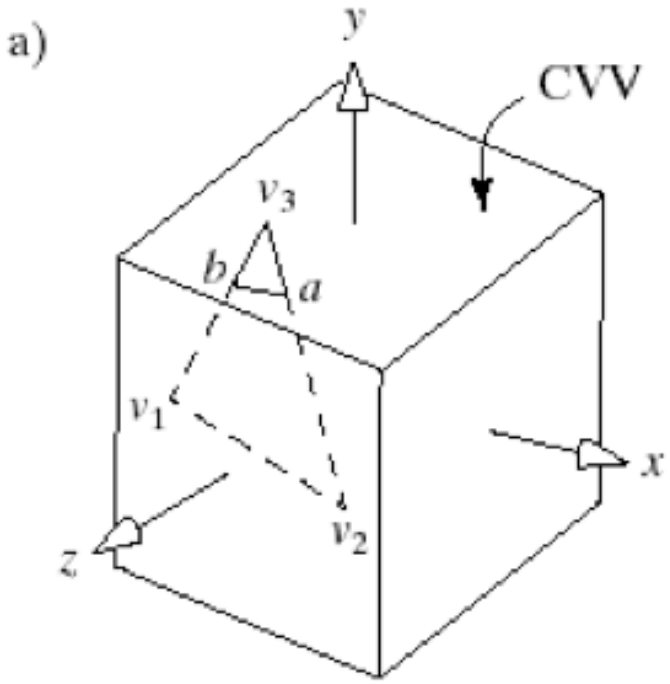
# Clipping Against the View Volume

- 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)

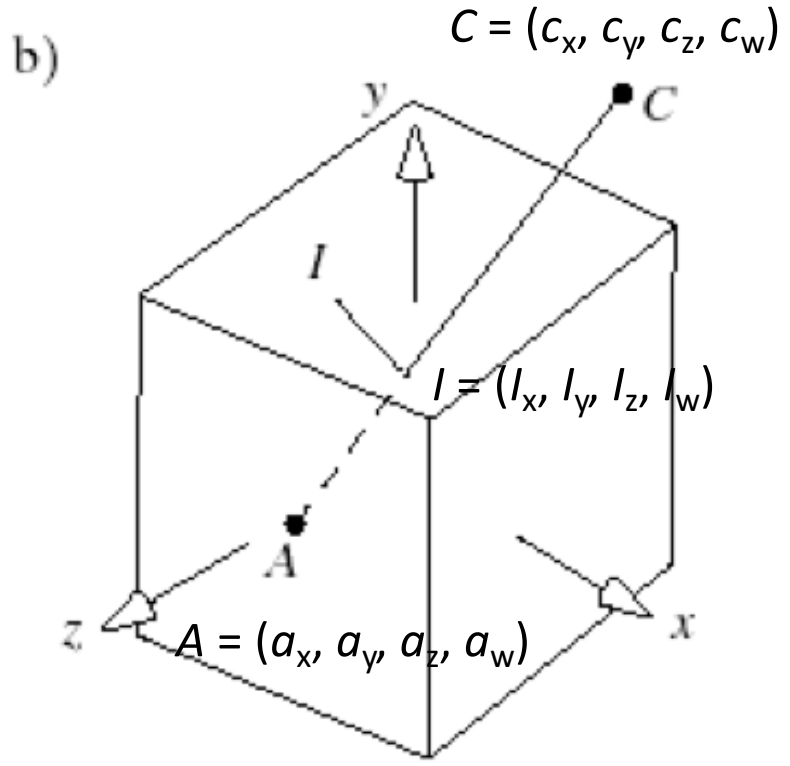
- Then it clips edge $v_2v_3$, and records the new vertex *a* formed where the edge exits from the *CVV*.
- Finally it clips edge $v_3v_1$ and records the new vertex b where the edge enters the *CVV*.
- The original triangle has become a quadrilateral with vertices $v_1v_2a\ b$.
- We actually clip in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



# Clipping Method

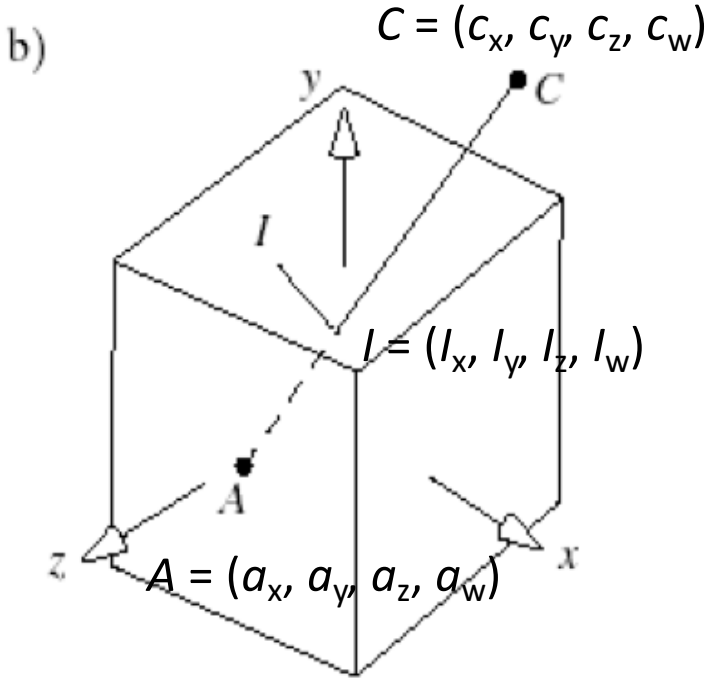
- Suppose we want to clip the line segment *AC* against the CVV. This means we are given two points in homogeneous coordinates, $A = (a_x, a_y, a_z, a_w)$ and $C = (c_x, c_y, c_z, c_w)$, and we want to determine which part of the segment lies inside the *CVV*.

- If the segment intersects the boundary of the *CVV,* we will need to compute the intersection point $I = (I_x, I_y, I_z, I_w)$.

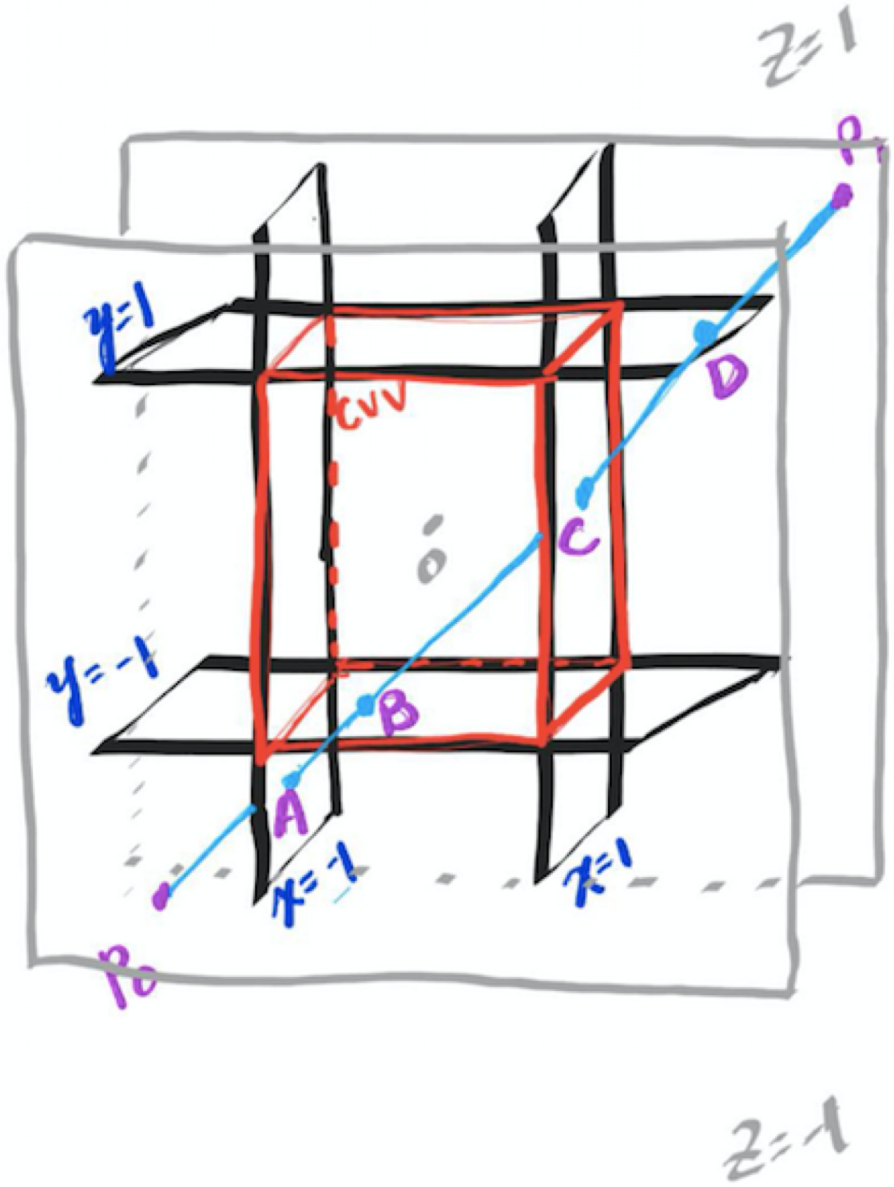


# Clipping Method (2)

- We view the CVV as six infinite planes (x=1, x=-1, y=1, y=-1, z=1, z=-1) and consider where the given edge lies relative to each plane in turn.
- We can represent the edge parametrically as $A + (C-A)t$. 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.



# CVV and the 6 planes for clipping

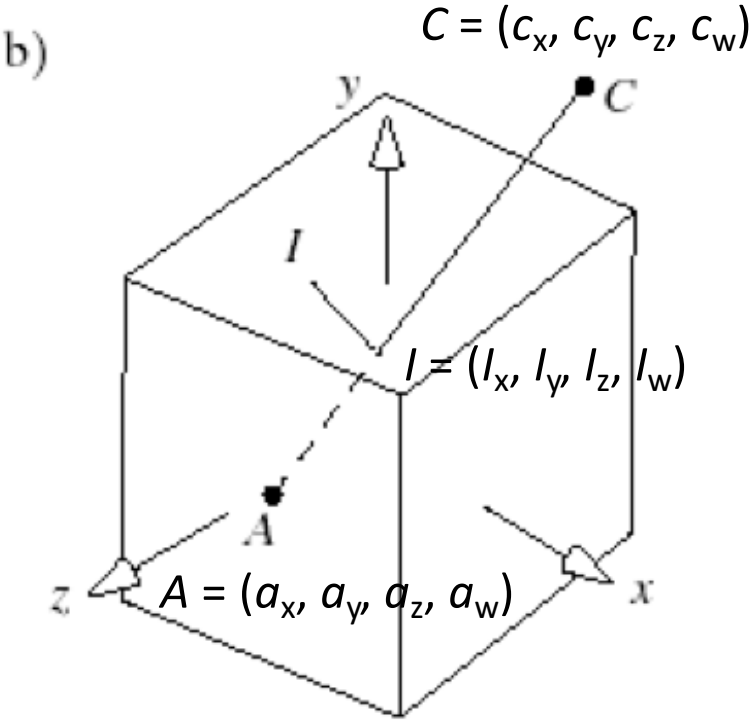


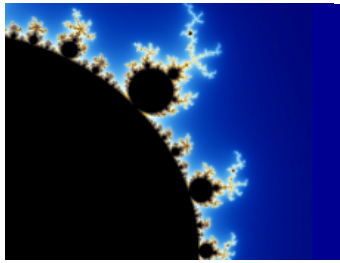
# Clipping Method (3)

- 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 $a_w + a_x > 0$.

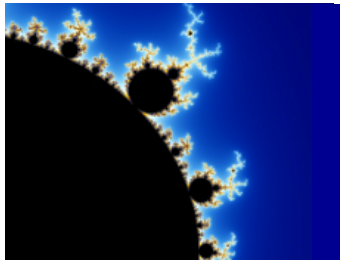
- Similarly *A* is inside the plane $x = 1$ if $a_x/a_w < 1$ or $a_w - a_x > 0$.
- We can use these to create boundary codes for the edge for each plane in the CVV.





# Boundary Code Values

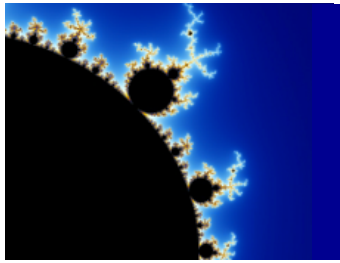
| *boundary coordinate* | *homogeneous value* | *clip plane* |
|---|---|---|
| $BC_0$ | $w + x$ | $x = -1$ |
| $BC_1$ | $w - x$ | $x = 1$ |
| $BC_2$ | $w + y$ | $y = -1$ |
| $BC_3$ | $w - y$ | $y = 1$ |
| $BC_4$ | $w + z$ | $z = -1$ |
| $BC_5$ | $w - z$ | $z = 1$ |



# 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*** | $y = -1$ |
| $BC_3$ | $w - y$ | | $y = 1$ |
| $BC_4$ | $w + z$ | | $z = -1$ |
| $BC_5$ | $w - z$ | | $z = 1$ |

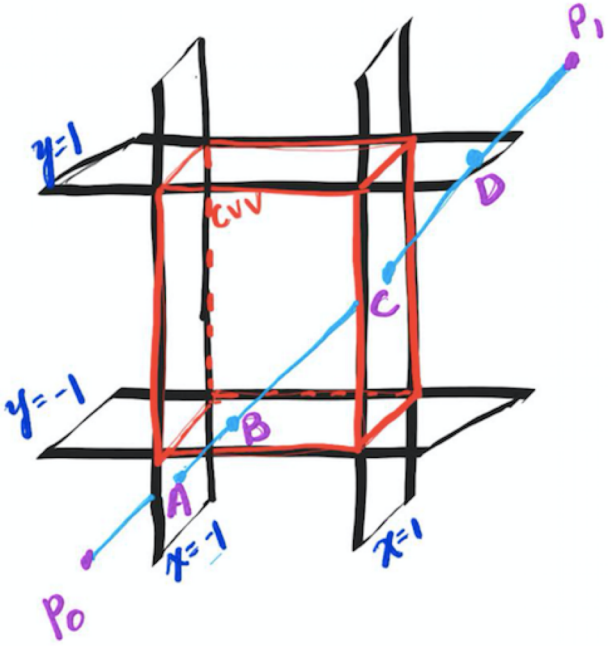


# Clipping Method (4)

- We calculate these six quantities for *A* and again for *C*.
- If all six are positive, the point lies inside the *CVV*.
- If any is 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 (6)

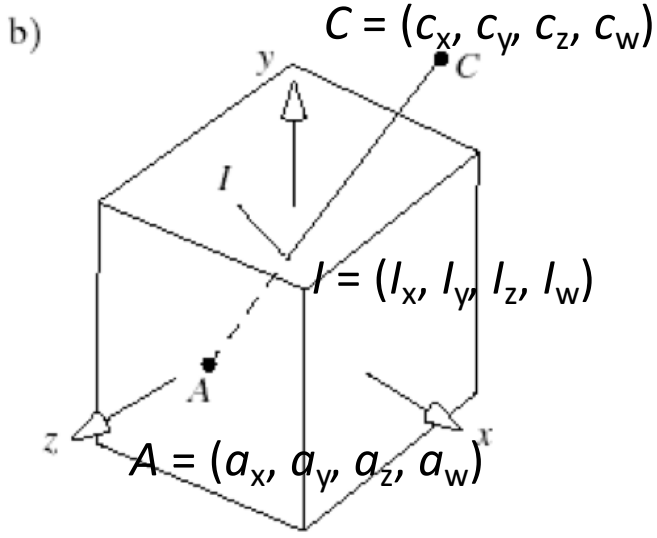
- We test the edge against each wall in turn.
- If the corresponding boundary codes have opposite signs, we know the edge hits the plane at some $t_{hit}$, which we then compute.
  - If the edge is entering (is moving into the inside of the plane as $t$ increases), we update $t_{in} = max(old\ t_{in}, t_{hit})$.
  - Similarly, if the edge is exiting, we update $t_{out} = min(old\ t_{out}, t_{hit})$.
- Candidate Interval (CI) is used for keep track of the calculation

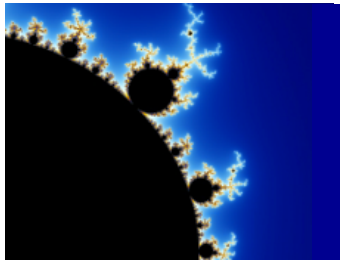


# Finding $t_{hit}$

- 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

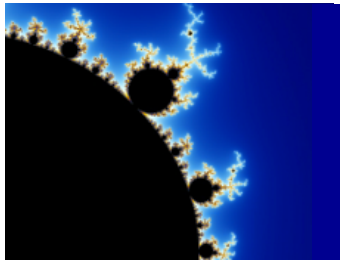
$$t = \frac{a_w - a_x}{(a_w - a_x) - (c_w - c_x)}$$

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

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

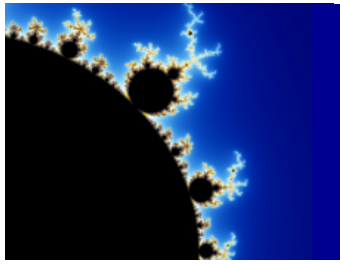
$$t = \frac{aBC_1}{aBC_1 - cBC_1}$$

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



# Clipping Method (7)

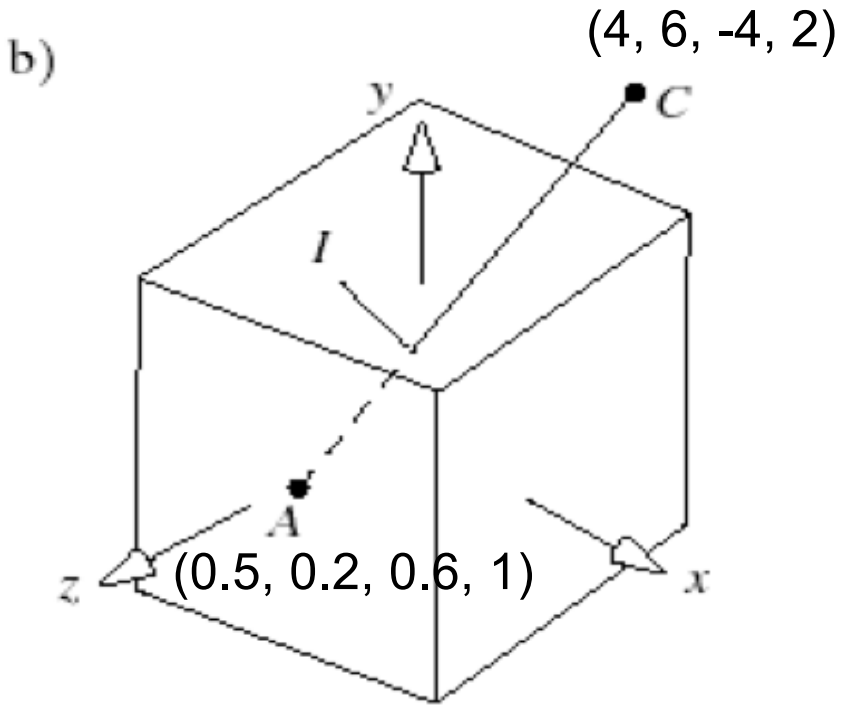
- This is the **Liang Barsky** algorithm with some refinements suggested by Blinn.
- clipEdge(Point4 A, Point4 C) takes two points in homogeneous coordinates (having fields *x*, *y*, *z*, and *w*) and returns 0 if no part of *AC* lies in the *CVV*, and 1 otherwise.
- It also alters *A* and C so that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

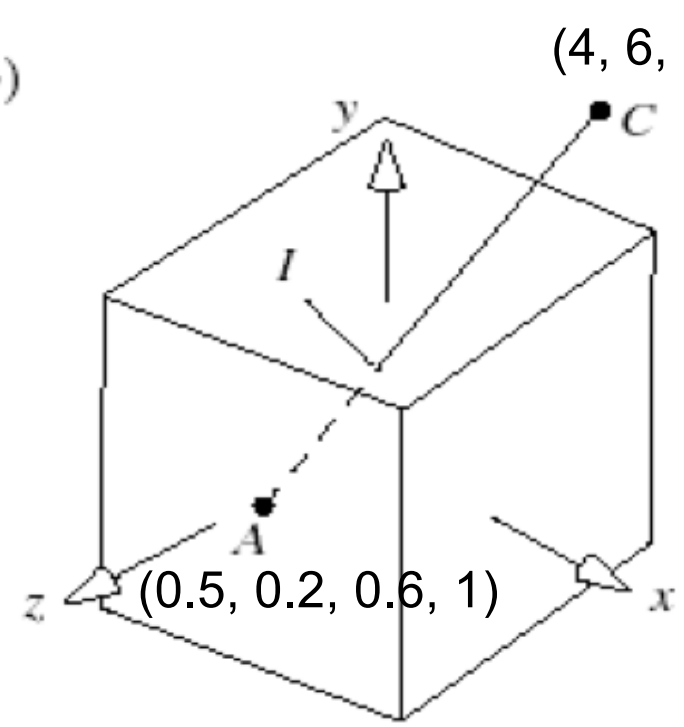
# Practice Question

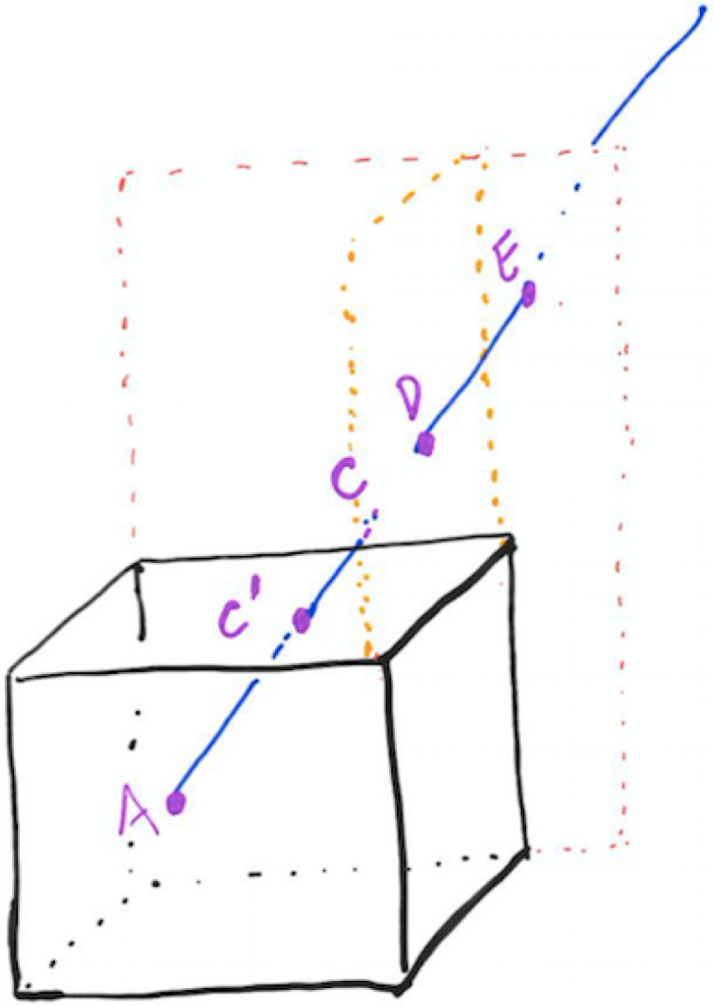



Clip line AC against CVV

# Practice Question - Clip line AC against CVV

b)

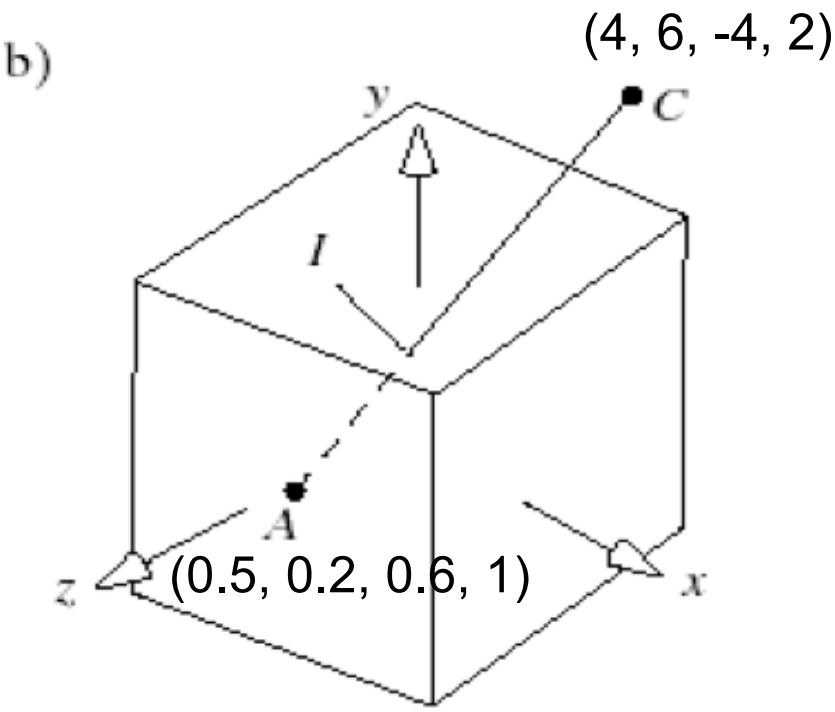



| *boundary coordinate* | *homogeneous value* | *clip plane* |
|---|---|---|
| $BC_0$ | $w+x$ | $x=-1$ |
| $BC_1$ | $w-x$ | $x=1$ |
| $BC_2$ | $w+y$ | $y=-1$ |
| $BC_3$ | $w-y$ | $y=1$ |
| $BC_4$ | $w+z$ | $z=-1$ |
| $BC_5$ | $w-z$ | $z=1$ |

# 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



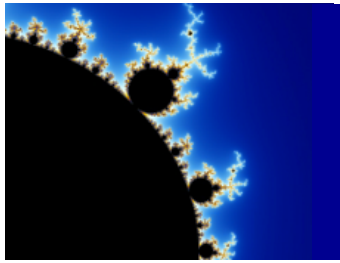

2.) Outcode for A: 000000
Outcode for C: 010110

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

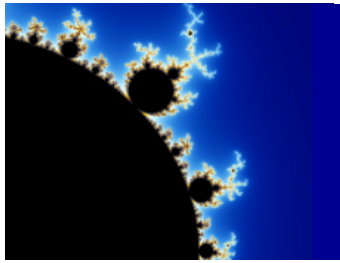
5.) Update C to I: C.x=A.x + $t_{out}$*(C.x - A.x)= 1.0845; C.y=A.y + $t_{out}$*(C.y - A.y)= 1.167
C.z=A.z + $t_{out}$*(C.z - A.z)= -0.2682; C.w=A.w + $t_{out}$*(C.w - A.w)= 1.167

6) Perspective division



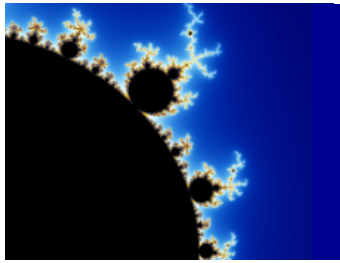
# Code for Clipper

```
function clipEdge(A, C)
{
    var tIn = 0.0, tOut = 1.0, tHit;
    var aBC, cBC; // points A and C against 6 planes
    var aOutcode = 0, cOutcode = 0;// 6 bits for each outcode

    <.. find BCs' for A and C: aBC0, aBC1, .. aBC5, cBC0, cBC1,… cBC5>
    <.. form outcodes for A and C ..>

    if((aOutcode & cOutcode) != 0) // trivial reject
        return 0;

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



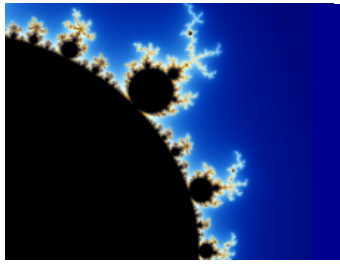
# Code for Clipper (2)

```
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
```



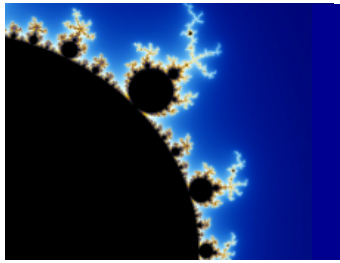
# Code for Clipper (3)

```
    // 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
```



# Clipper

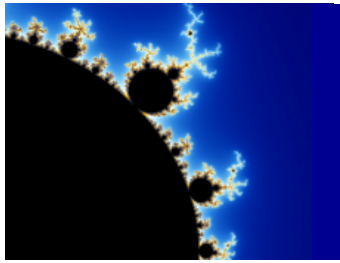
- *A* is updated to *A* + (C - *A*) $t_{In}$ if $t_{In}$ has changed, and *C* is updated to *A* + (*C* - *A*) $t_{Out}$ if $t_{Out}$ has changed.
- Note that using the canonical view volume for clipping allows clipping with no parameters.
- Note that using homogeneous coordinates results in fewer tests in the clipper, which means it is more efficient.



# Why Use the CVV?

- It is parameter-free: the algorithm needs no extra information to describe the clipping volume. It uses only the values -1 and 1. So the code itself can be highly tuned for maximum efficiency.

- Its planes are aligned with the coordinate axes (after the perspective transformation is performed). This means that we can determine which side of a plane a point lies on using a single coordinate, as in $\alpha_x > -1$. If the planes were not aligned, an expensive dot product would be needed.



# Why Use Homogeneous Coordinates?

- Doing the perspective divide step destroys information; if you have the values $a_x$ and $a_w$ explicitly you know the signs of both of them.
- Given only the ratio $a_x/a_w$ (that you would have after perspective division), you can tell only whether $a_x$ and $a_w$ have the same or opposite signs.
- Keeping values in homogeneous coordinates and clipping points closer to the eye than the near plane automatically removes points that lie behind the eye.