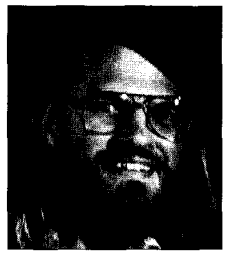
Jim Blinn's Corner



All the published clipping algorithms I could find were—how shall I put this diplomatically—really terrible. They're way too complicated and have too many special cases. There is a good way to do clipping.

A Trip Down the Graphics Pipeline: Line Clipping

James F. Blinn, Caltech

Teaching an introductory graphics class at Caltech has two benefits: It encourages me to read up on recent developments, and it gives me an excuse to sit and think about better ways of doing things. Recently I've been lecturing about the standard computer graphics transform-clip-draw pipeline, so I diligently researched all the published clipping algorithms I could find and came to the conclusion that they all were—how shall I put this diplomatically—really terrible. They're way too complicated and have too many special cases. There *is* a good way to do clipping. I learned it at the University of Utah, but now I realize that it hasn't made it into the published literature.

So this is the first of a series of columns on the graphics pipeline. It's a bit of a personal view, since I will concentrate only on techniques that I have found useful. This time out I am going to concentrate on the algorithmic aspects of the line-clipping part of the pipeline. I will have to defer some juicy issues about how clipping interacts with transformations for later, so some of the things I say here will have to be taken on faith. Trust me.

Definitions of type styles used

matrices	capital letters in bold roman type
vectors	capital letters in roman type
vector elements	italic type
homogeneous	lowercase italic type
nonhomogeneous	capital italic type
variables	initial capital letter, roman type

The pipeline

The classic computer graphics pipeline is an assembly-line-like process that geometric objects must experience on their journey to becoming pixels on the screen. Since we are all adults, I won't mess around with 2D versions of these routines; I'll deal directly with 3D homogeneous coordinates. I'll also follow my typical convention that homogeneous coordinates are in lowercase and real coordinates (the ones with the w divided out) are in uppercase. We begin with some objects defined as collections of (x, y, z, w) points (often with $w = 1$), connected by line segments, and perform three operations on them (one of which is performed twice):

- transformation
- clipping
- w division

There are many ways to do this, but let's look at the four stages in Figure 1:

Transform takes (x, y, z, w) points and multiplies them by a 4×4 matrix to get them into a special clipping space.

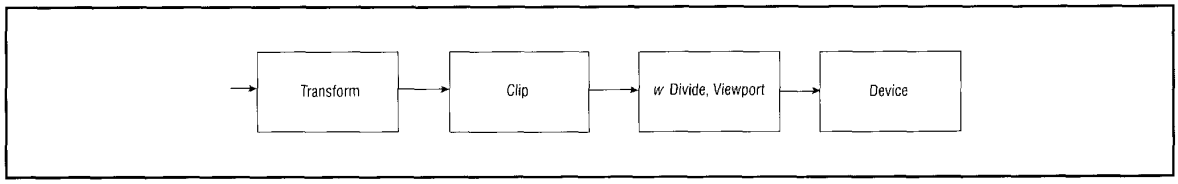


Figure 1.

Clip takes (x, y, z, w) points in clipping space and clips lines against some set of boundary planes.

Viewport takes (x, y, z, w) in clipping space, divides out the w, and transforms to hardware pixel space.

Device takes (X, Y, Z) in the coordinate space of the hardware and does whatever is necessary to display lines.

Transformations

I'll talk in detail about transformations in a later column, but we need to cover enough to see how they affect the clipper arithmetic.

There is conceptually one big transformation that takes coordinate points straight from their definitional space all the way to integer hardware pixel space. We want to minimize redundant transformation as much as possible. After all, that's why mathematicians invented associativity, isn't it? So, this whole thing could (if it weren't for clipping) be represented as multiplication by one 4×4 matrix.

But we do have to do clipping, so we define an intermediate coordinate system—clipping space. The big transformation is broken into two pieces. The first maps definitional space to clip space (I'll call this $\mathbf{T}_1$), and the second maps clip space to pixel space (I'll call this $\mathbf{T}_2$). The $\mathbf{T}_1$ transformation typically contains modeling and viewing transformations and perspective. The $\mathbf{T}_2$ transformation is typically just a scale and translation. The product of the two should still equal the original complete transformation.

There are two reasons for using a separate clipping space. First, it can make the clipper arithmetic simpler. Second, it separates knowledge of the location of the boundaries from the code of the clipper. The idea is to make a clipper that clips to a simple built-in set of boundaries. Then the transforms are responsible for mapping the actually desired boundaries to the clipper's boundaries. In fact, the algorithm I describe here will work perfectly well for any boundaries, but the basic version wires in some simple ones.

To make the clipper and its arithmetic as simple as possible, we pick a particularly convenient clipping space. The usual choice is to clip to the rectangular region $-1 \le X \le 1$, $-1 \le Y \le 1$, and $0 \le Z \le 1$. But wait a minute. The reason for using a special clipping space is to make the clipping arithmetic simple. This isn't as simple as it could be. Let's face it, you're going to have to do the transformation anyway. Why not make it go to a *really* convenient clip space: the region $0 \le X \le 1$, $0 \le Y \le 1$, and $0 \le Z \le 1$. This single change made an almost 20 percent difference in the speed of my clipper.

Let me reassure you that a clipping region of (0...1) does not imply that, for perspective, the eye is looking at a corner of the screen. The actual clipping region is set up in the viewport initialization routine. It effectively appends an extra scale and translate onto the end of T_1, and its inverse at the beginning of T_2. This makes the location of the boundaries used internally by the clipper invisible to the caller of the pipeline. The punch line is that a clipper with built-in boundaries of (0...1) in X, Y, Z can, when spliced between two transforms, serve for all rectangular or perspective pyramid boundaries.

Connectivity

A typical database poured into this pipe consists of a bunch of points connected by line segments: a sequence of MoveTo, DrawTo, DrawTo... operations. All the clippers I have seen in the literature immediately break this down into a series of unconnected segments with calls such as Clip(endpoint0,endpoint1). I think this is a mistake for two reasons. First, you are destroying information about connectivity. Some output devices (like physical pen plotters—these still do exist) or output routines (like Postscript postprocessors) can use this connectivity. Second, some intermediate calculations (the boundary coordinates and outcodes described below) need be done only once per point if you maintain the connectivity.

So we've established that we want MoveTo(point) and DrawTo(point) calls. We could provide a separate MoveTo and DrawTo routine at each stage of the pipeline, but this is a bit of a nuisance. The Transform and Viewport stages of the pipe do the same operation on their parameters, regardless of whether the call is MoveTo or DrawTo. For this reason each stage of the pipeline consists of a single routine with two parameters: the new point and a flag that indicates MoveTo versus DrawTo. The Clip and Device stages keep an internal "current point" for joining up segments.

Clipper overview

A clipper is an example of a common type of computer graphics problem: One that does a lot of work to accomplish nothing. By this I mean that clippers are unnecessary a lot of the time (for example, if the object is completely visible). Any time required to figure this out is wasted. So our main

goal is for the clipper to quickly determine when it isn't needed and step out of the way.

History (?)

If you look in the literature at published line clippers, you will typically find the following three: Cohen-Sutherland (described in Newman and Sproull,[1] 1979), Cyrus-Beck[2] (1978), and Liang-Barsky[3] (1984). The algorithm I'm going to talk about doesn't present any stunning new concepts in clipping; it just contains all the best features of these algorithms. In addition I think it's a lot more general and more straightforward.

The interesting thing is that I learned the essence of this algorithm from Martin Newell in 1975! And I don't think *he* invented it either—it was just presented at the University of Utah as a reasonable way to do clipping. I've been teaching it to my graphics classes for years but never realized that it hadn't been formally published.

Over the years I've tinkered with the algorithm, refining and simplifying it. Even while writing this column I've come up with some new simplifications. Simplification is something we need more of in this world. I don't really remember how much of it is mine and how much I got from others, so I won't try to accurately reproduce any particular historical algorithm here. I'll just present the most highly refined clipper that I've been able to come up with.

Boundary coordinates

We define our convex clipping volume as a set of bounding planes. In homogeneous coordinates a plane is a column vector. The dot product of a point with this vector gives zero if the point is on the plane. By picking the signs of the column vector properly, we can arrange for the dot product to be positive if the point is on the visible side of the plane, and negative if it's on the invisible side.

Here's where our special clipping space pays off. The two clip planes in the X direction, for example, are $X = 0$ and $X = 1$. These correspond to the column vectors

$$B_1 = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} \text{ and } B_2 = \begin{bmatrix} -1 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

Taking the dot product of (x, y, z, w) with these is easy; it's just x and $w - x$. We will call these values the *boundary coordinates*, BC. The expressions for each of the six boundary coordinates for the clipping boundaries are then

	Homogeneous value	Real space plane
BC_1	x	$X = 0$
BC_2	$w - x$	$X = 1$
BC_3	y	$Y = 0$
BC_4	$w - y$	$Y = 1$
BC_5	z	$Z = 0$
BC_6	$w - z$	$Z = 1$

The first thing the clipper does to a new point is to calculate this vector of BCs. Because of our clever choice of boundaries, this takes only three subtractions. We will use these BC values at several places in the algorithm. A point is visible if all its BCs are positive.

Outcodes

We use the sign bits of the BCs to find where we need to do work. Assume we have the list of boundary coordinates for point P0, called BC0, and the list for point P1, called BC1. To see if the segment from P0 to P1 straddles boundary i, look at the sign bits of $BC0_i$ and $BC1_i$. Remember, a 1 bit means that the point is outside the boundary:

Sign ($BC0_i$)	Sign ($BC1_i$)	Meaning
0	0	Whole segment visible with respect to this boundary
1	0	Straddles boundary, P0 is out
0	1	Straddles boundary, P1 is out
1	1	Whole segment outside boundary

To make the best use of these, we construct a flag word for each point called an "outcode." Each bit in the outcode is the sign bit of an element of the BC array. If the whole outcode is zero, the point is visible. But the really nifty thing about outcodes is that they enable us to test all clip planes in parallel using bitwise logical operations.

If both ends of a segment lie outside at least one of the boundaries, the whole segment can be skipped. We can detect this "trivial reject" by taking the logical AND of the outcodes of the two endpoints. If the bit position of any boundary is 1 for both outcodes, the result will be nonzero. Next, if both endpoints lie inside all boundaries, the whole segment is visible. We can detect this "trivial accept" by taking the logical OR of the outcodes and getting a result of zero.

A lot of high-level languages don't seem to encourage this type of shenanigans. But it's so useful that it's worth coercing your program to do it. After all, if your computer can't do ANDs and ORs, who can?

I actually construct the outcode with a little assembly-language routine that looks at the sign bits of the floating-point numbers. It requires no floating-point instructions at all, just six repetitions of load and shift instructions. This is the ideal application for assembly language—small routines that do the bit shuffling that compilers are crummy at. This type of solution may not appeal to everyone, but profiling tests showed me that outcode generation is where a lot of time was spent. You do have to be careful though; under certain weird circumstances, I have had problems—getting a value of –0 for an element of BC, resulting in a bad outcode bit.

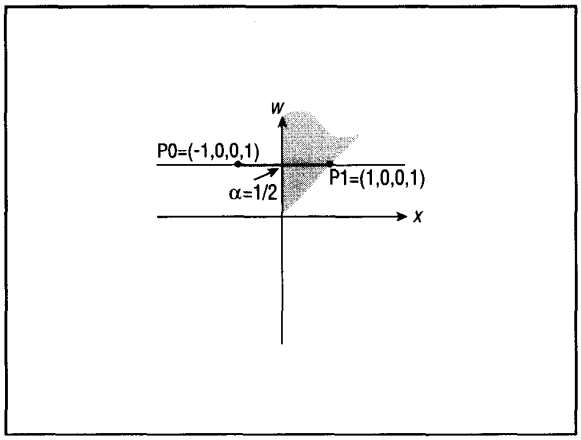


Figure 2a.

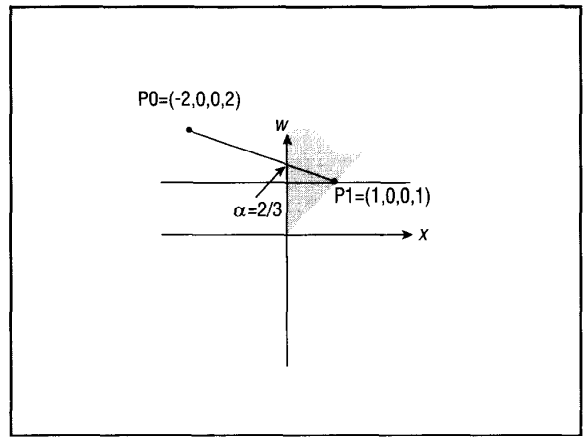


Figure 2b.

Interpolation

If a line segment crosses a clipping plane, we must calculate the intersection. You can do this easily with parametric interpolation. The parameterized segment from point P0 to P1 is

$$P(\alpha) = P0 + \alpha(P1 - P0)$$

When $\alpha = 0$ we are at P0 and when $\alpha = 1$ we are at P1.

How to calculate α? Dot the above equation with the column vector B_i of the straddled boundary plane:

$$P(\alpha) \cdot B_i = P0 \cdot B_i + \alpha(P1 - P0) \cdot B_i$$

Since we want $P(\alpha)$ to be on the boundary, its dot product should be zero. The dot products with P0 and P1 are just the boundary coordinates. This gives

$$0 = BC0_i + \alpha(BC1_i - BC0_i)$$

Then just solve for α:

$$\alpha = \frac{BC0_i}{BC0_i - BC1_i}$$

Note that this expression is only in the range (0...1) if $BC0_i$ and $BC1_i$ have different signs. We will endeavor to calculate α only when actually necessary to do an interpolation, that is, when we have previously determined that the *BC* values for the endpoints of the segment have opposite signs.

This works in homogeneous coordinates for perhaps a somewhat subtle reason. Look at Figure 2a. Here we are clipping P0 = (–1, 0, 0, 1) to P1 = (1, 0, 0, 1). The left edge *BC* is just *x*, so $BC0_1 = -1$ and $BC1_1 = +1$. The different signs indicate that the line crosses the left boundary, so $\alpha = -1/(-1 - (+1)) = 1/2$, as you would expect.

Now what if P0 = (–2, 0, 0, 2)? It's the same geometric location in space but represented homogeneously differently. Here the *BC*s are –2 and 1 and the $\alpha = -2/(-2 - (+1)) = 2/3$. You would think that the interpolated point would be wrong. But we are interpolating in *homogeneous* space, not real space. If you look at Figure 2b you see that the point two thirds of the way from P0 to P1 in homogeneous space does indeed project onto the boundary, $X = 1$, when we (later) divide by *w*.

The algorithm

First a few conventions: The variables P0, P1, and P are homogeneous points (four elements: *x, y, z, w*). We assume that any arithmetic done on them is done for each element. The clip routine outputs its results to the next stage of the pipeline by calling the routine ViewPt.

I have separated the code into bite-size chunks with what appear to be subroutine calls connecting them. I don't suggest that you make separate subroutines out of them; the subroutine call overhead would probably be excessive. I just split the pieces out to make the listing more understandable, all nice and top down.

Outer shell

The outer shell of the clip routine basically just takes care of connectivity. It maintains information about the "current point" in P0, BC0(1...6), and Kode0. (You'll have to excuse me for some of the variable names, but I get uneasy when I see integer type variables with names that don't start with the letters I, J, K, L, M, or N.) Note that the BC and Kode values are calculated only once per endpoint and are saved from one call to another for DrawTo operations that are chained together:

```
Clip(P1,Flag)
    calculate BC1(1...6), Kode1
    if Flag='Move'
        do MoveStuff
    else
        do DrawStuff

    copy P1,BC1,Kode1 to P0,BC0,Kode0
```

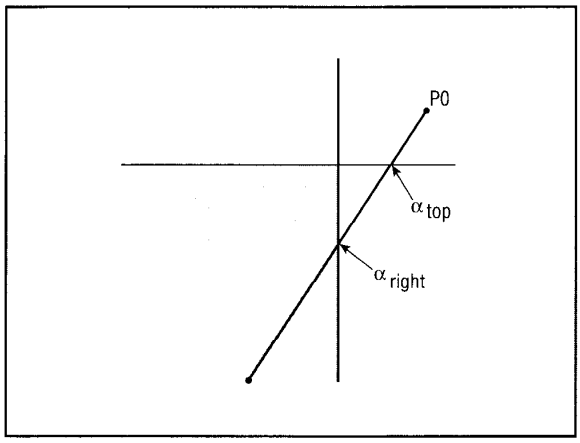


Figure 3a.

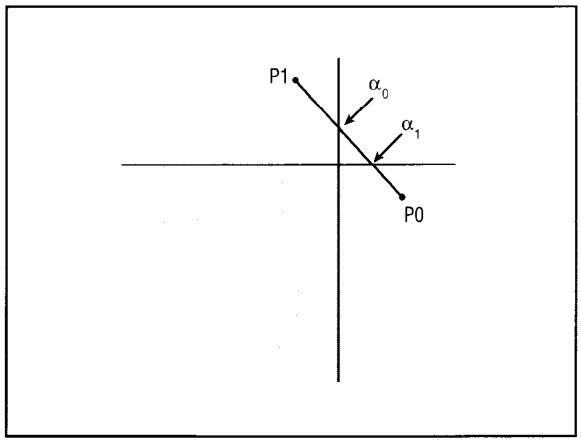


Figure 3b.

Move

This part is simple; it just passes the Move operation down the pipe if the point is visible:

```
MoveStuff
    if Kode1=0
        call ViewPt(P1,'Move')
```

Draw

First we do trivial reject and trivial accept tests using bitwise logical operations on the outcodes. If the logical AND of the outcodes is nonzero we have trivial reject—or stated another way, a zero result means no trivial reject. Next, if the logical OR is zero, we have trivial accept:

```
DrawStuff
  if (Kode0 AND Kode1)=0
      if (Kode0 OR Kode1)=0
          call ViewPt(P1,'Draw')
      else
          do NontrivialStuff
```

Almost all line segments will be trivial accepts or trivial rejects, so the above covers most cases. Once you have the outcodes, you only need one AND and one OR and you're outa there.

Another way

Modern computers have prefetch queues operating in parallel with their instruction execution. This queue must be flushed every time there is a branch instruction. So, to really streamline things, we want to avoid branching. Let's reorganize this code to make the most commonly occurring situation (the trivial accept) fall through with straight line code. (If you assume trivial reject is the most common, a slightly different rearrangement would be necessary.) This is not so "structured," requiring horrible Goto statements, but I think it is actually just as understandable.

```
Clip(P1,Flag)
    calculate BC1(1...6), Kode1
    if Flag='Move' goto move
    if (Kode0 AND Kode1) ≠ 0 goto finish
    if (Kode0 OR Kode1) ≠ 0 goto nontriv
    call ViewPt(P1,'Draw')
    finish: copy P1,BC1,Kode1 to P0,BC0,Kode0
    exit

    move: if Kode1 ≠ 0 goto finish
    call ViewPt(P1,'Move')
    goto finish

    nontriv: do NontrivialStuff
    goto finish
```

Nontrivial stuff

You get to NontrivialStuff only if the segment straddles at least one boundary. The individual bits of (Kode0 OR Kode1) give a mask for the boundaries that the segment straddles, so we recalculate this value and store it as Klip. It might seem strange to do this a second time, but remember, this is the rarely executed part of the code. We just avoided storing Klip in the more often encountered case where we never use it.

While examining straddled boundaries we keep track of the still-visible part of the segment with the two values α_0 and α_1. For each straddled boundary we calculate the α value for the intersection and update whichever endpoint the outcodes tell us is the outside one. The loop variable Mask

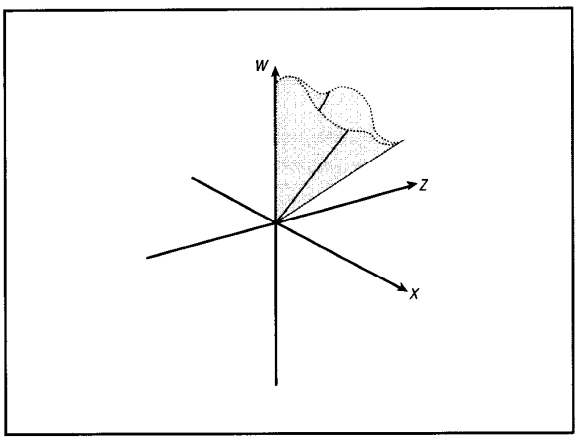


Figure 4a.

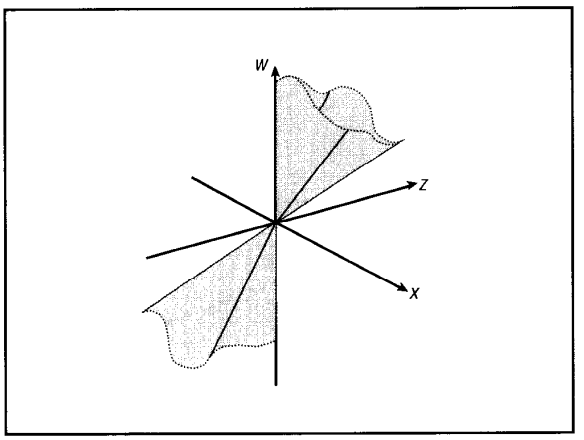


Figure 4b.

selects one bit at a time, so the innards of this loop are executed only for those bit positions of Klip that are 1.

There are two subtleties. First, a point might be outside more than one boundary. An α value must be updated only if the new intersection shaves more off the segment than has been shaved off so far. For example, in Figure 3a we first test the right boundary and then the top boundary. The latter test generates an α that is closer to the endpoint, so no update happens. Figure 3b shows the second subtlety: If α_0 ever gets greater than α_1, we have a nontrivial reject:

```
NontrivialStuff
    Klip ← Kode0 OR Kode1
    α0 ← 0.0; α1 ← 1.0; Mask ← 1
    for I ← 1 to NbrClipPlanes
        if(Klip AND Mask) ≠ 0
            α ← BC0(I)/(BC0(I) –BC1(I))
            if (Kode0 AND Mask) ≠ 0
                α0 ← max(α0, α)
            else
                α1 ← min(α1, α)
            if (α1 < α0) goto NontrivReject
        shift Mask left one bit


    if Kode0 ≠ 0
        P ← P0 + α0(P0 – P1)
        call ViewPt(P,'Move')

    if Kode1 ≠ 0
        P ← P0 + α1(P0 – P1)
        call ViewPt(P,'Draw')
    else
        call ViewPt(P1,'Draw')


    NontrivReject:
```

Note that we actually do the interpolation arithmetic only if an endpoint is out. Also, we never need to do an explicit ViewPt(P0,'Move') here because we are guaranteed that this was already done by a previous call ('Move' or 'Draw') if P0 is visible.

Homogeneous clipping

There often seems to be some confusion about how homogeneous coordinates affect a clipper, especially after points are passed through a homogeneous perspective transformation. A paper I cowrote with Martin Newell[4] might have scared some people, but it's really not that big a deal. This algorithm works just dandy for all but the most perverse cases. In particular, stuff behind the eye gets clipped properly, with no special cases required.

What are the perverse cases that cause problems? (I of course keep stumbling onto them.) Well, sometimes some lines will get clipped that should, in fact, be visible. In Figure 4a you see the region we are clipping to in homogeneous space, a sort of inverted pyramid. In Figure 4b you see the region we should be clipping to, a double pyramid. All points in the bottom pyramid do indeed, when projected to $w = 1$, wind up in the visible region.

So what sorts of things reside in the bottom pyramid with negative w's? If you start with good honest positive w's, the homogeneous perspective transform will not put anything there. You need to worry only when you have negative w values in your original model (not usually necessary), or when the original model has infinite segments (those that connect their endpoints through infinity rather than directly) indicated by a positive w on one end and a negative w on the other. And sometimes not even then. For example, in an earlier column[5] about perspective shadows we ran into this problem, but the clipper "bug" worked to our advantage. Some stuff got generated in the bottom pyramid, but we actually wanted it clipped because it just generated the "antishadow."

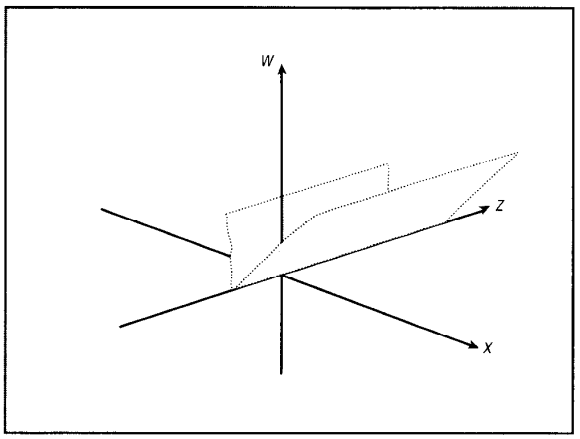


Figure 5.

The easiest solution if you want negative pyramid stuff is to draw your object, multiply the whole $\mathbf{T}_1$ matrix by –1, and draw the object again.

Z clipping

Z clipping is very confusing to a lot of users of graphics systems; things near the viewer keep disappearing. *X* and *Y* clipping make sense—you don't expect to see things that are off the screen. But try to explain the near and far clipping planes to an artist, and you are stuck with the question of why you'd ever want to do something like that.

Well, *Z* clipping is sort of optional. The trouble with eliminating it has to do with the homogeneous division. In the pipeline of Figure 1 the homogeneous division occurs after the clipping stage. (Contrary to popular belief, with a few reasonable assumptions it *is* possible to clip after doing the homogeneous division. It's just a real algorithmic nuisance. Juicy item for later.) Anyway, what happens if we get $w = 0$ and try to divide? Well, with Z clipping enabled (see Figure 4a) there is only one point that can generate $w = 0$; that is (0, 0, 0, 0). It's real hard to get this point out of the clipper. If, on the other hand, you disable *Z* clipping, you get the region in Figure 5. For the most part, this still works. Lines that come toward you in real space project into lines that puncture the sides of the v-shaped trough. The only lines that generate $w = 0$ after clipping are those that pass through the *z* axis, the valley of the trough. These are lines that pass exactly through the eyepoint—after which you are dead and don't care about divide-by-zero errors. But seriously folks, no matter how unlikely it is to get $w = 0$, you should still test for it before dividing. Having a program die because of a zero-divide error is a most unpleasant experience.

Global clipping

Of course, the best way to speed up clipping is not to do it at all. Our outcode machinery allows us to do this in a very straightforward manner. We just apply trivial accept and reject tests to the convex hull of a whole object. What is a convex hull? Well, imagine blowing up a balloon. Insert your object inside and then let the air out. The resulting shape is the convex hull.

Why is this nice? If all the points of the convex hull are trivially visible, the whole object is trivially visible. And if all the points of the convex hull are trivial rejects, the whole object is a trivial reject.

The actual convex hull is sometimes hard to come by, but you don't really have to use it. I have tried related shapes that work just as well:

1. Control points for Bezier curves and surfaces.
2. Corners of the bounding volume formed from max/min (x, y, z) of the object (8 points).
3. Max/min (x, y) for 2D objects (4 points). This is good for explicitly modeled text.

To do global clipping, before you draw an object you calculate the cumulative logical OR and logical AND of the outcodes of the bounding points:

```
Global Clip Setup
    Ocumulate ← 0; Acumulate ← –1
    for I ← 1 to NbrBoundingPoints
        transform bounding point to clip space
        calculate Kode
        Ocumulate ← Ocumulate OR Kode
        Acumulate ← Acumulate AND Kode
```

Then outside the drawing loop you can do a trivial reject/accept test that looks a whole lot like the test inside the DrawStuff part of the clipper. Furthermore, if you detect a global trivial accept, you can merge the general 4×4 matrix $\mathbf{T}_1$ with the viewport transform $\mathbf{T}_2$ and avoid doing two transformations inside the drawing loop:

```
Global Clip/Draw
    if (Acumulate = 0)
        if (Ocumulate = 0)
            T ← T1T2
            transform points of object by T
            divide by w
            pass them directly to Device
        else
            transform points of object by T1
            pass them through normal clipper
```

Global clipping is a gamble. Suppose you have an object with 100 points, and the bounding volume has eight points. If you do the global test and there's a trivial accept or reject, you win. If it's a nontrivial case, you have had to do a total of 108 transform, BC, and Kode calculations. Not a bad risk. This is especially effective for Bezier curve primitives: The global clipping stage can be built into the primitive curve drawing routine, making it invisible to the caller of the routine.

Observations

Part of the goal here was to minimize the amount of floating-point arithmetic. I would hazard a guess that integer-bit testing is always faster than floating-point arithmetic. Even a floating-point comparison counts as one unit of floating-point arithmetic, so I've made almost all tests into logical or integer comparisons. The only floating-point operations (besides *BC*s) are in the rarely executed part of the code.

We purposely made the clipping planes as simple as possible, but this algorithm can clip against *any* set of planes. Just supply the proper expressions for (any number of) boundary coordinates. The location of the clipping planes is completely encapsulated in the calculation of the *BC* values. The normal clipper and the global clipper just use the *BC*s (whatever they happen to be) to calculate the outcodes and the alphas. My implementation of this clipper has the six simple planes built in, but optionally it can include a couple more general ones for some special purposes.

A lot of the streamlining of this algorithm resulted from profiling tests of various earlier versions. You can theorize and fantasize about where you think the time is going, but actual experimental data are the only reasonable things to base optimization on. I've found that most of the execution time is still in the calculations of *BC*s, even after making them as simple as I have done. Why is this? Because that is the only code that is executed *every* time the clipper is called. The trivial accept and reject tests allow us to skip most of the rest. Performance of the algorithm depends mostly on how fast it can execute for trivial accept or reject cases, since that is what happens most of the time.

What is publishable?

Now admittedly I haven't compared this algorithm instruction for instruction with all the "historical" clippers mentioned earlier, but it sure seems a lot simpler to me. This is not meant as an insult to Cyrus, Beck, Liang, Barsky, et al. They didn't build on this algorithm, probably because they didn't know about it. After all, it has never been published. And why? *I* never published it before because for the most part it wasn't my idea. And also it didn't seem like a big enough idea to publish.

This raises an interesting question: "What is publishable?" (That is, in terms of being new or different enough to publish.) This question becomes doubly interesting in light of the current flap about software patents. A lot of good ideas don't get published because nobody knows who invented them; they are just classed as "common knowledge." There are channels for publishing new research results, but there are fewer places to put more minor ideas or ideas that are simply refinements of standard techniques—or just to archive common knowledge.

I have an ideal avenue for describing such things in these columns. I can write anything I want, because it's not a formal technical paper. All things are not expected to be major research results. Often they are the results of my own tinkering, combined with common knowledge that might not be well publicized. Others of you have a similar opportunity via a book called *Graphics Gems*, edited by Andrew Glassner (Academic Press, Cambridge, Mass., 1990). This is a collection of short ideas by a variety of people. I believe he is working on a sequel; you can contact him at Xerox PARC.

After all, from a historical perspective, if you don't publish it, you may as well not have done it.

The experiment

Oh, yes. The gray wedge I published last time will be described next time. Basically, I am working on a column about the transfer function from gray levels requested by Postscript to actual visible output. The only real way to do this is to measure it. That requires running a test wedge completely through the printing process to the finished printed page. I will then measure the reflectivity of the wedge and generate the transfer function. Not everyone has the luxury of doing this, but since I get a few pages to play with every issue, I do. ❑

References

1. W. Newman and R. Sproull, *Principles of Interactive Computer Graphics*, 2nd ed., McGraw-Hill, New York, 1979.
2. M. Cyrus and J. Beck, "Generalized Two- and Three-Dimensional Clipping," *Computers and Graphics*, Vol. 3, 1978, pp. 23-28.
3. Y.-D. Liang and B. Barsky, "A New Concept and Method for Line Clipping," *ACM Trans. Graphics*, Vol. 3, No. 1, Jan. 1984, pp. 1-22.
4. J. Blinn and M. Newell, "Clipping Using Homogeneous Coordinates," *Computer Graphics* (Proc. Siggraph), Vol. 12, No. 3, Aug. 1978, pp. 245-251.
5. J. Blinn, "Me and My (Fake) Shadow," *IEEE CG&A*, Vol. 8, No. 1, Jan. 1988, pp. 82-86.