Chapter 65

3-D Clipping and Other Thoughts

Chapter 65

Determining What's Inside Your Field of View

Our part of the world is changing, and I'm concerned. By way of explanation, three anecdotes.

Anecdote the first: In the introduction to one of his books, Frank Herbert, author of *Dune,* told how he had once been approached by a friend who claimed he (the friend) had a killer idea for an SF story, and offered to tell it to Herbert. In return, Herbert had to agree that if he used the idea in a story, he'd split the money from the story with this fellow. Herbert's response was that ideas were a dime a dozen; he had more story ideas than he could ever write in a lifetime. The hard part was the writing, not the ideas.

Anecdote the second: I've been programming micros for 15 years, and writing about them for more than a decade and, until about a year ago, I had never—not once!—had anyone offer to sell me a technical idea. In the last year, it's happened multiple times, generally via unsolicited email along the lines of Herbert's tale.

This trend toward selling ideas is one symptom of an attitude that I've noticed more and more among programmers over the past few years—an attitude of which software patents are the most obvious manifestation—a desire to think something up without breaking a sweat, then let someone else's hard work make you money. It's an attitude that says, "I'm so smart that my ideas alone set me apart." Sorry, it doesn't work that way in the real world. Ideas are a dime a dozen in programming, too; I have a lifetime's worth of article and software ideas written neatly in a notebook, and

I know several truly original thinkers who have far more yet. Folks, it's not the ideas; it's design, implementation, and especially hard work that make the difference.

Virtually every idea I've encountered in 3-D graphics was invented decades ago. You think you have a clever graphics idea? Sutherland, Sproull, Schumacker, Catmull, Smith, Blinn, Glassner, Kajiya, Heckbert, or Teller probably thought of your idea years ago. (I'm serious—spend a few weeks reading through the literature on 3-D graphics, and you'll be amazed at what's already been invented and published.) If they thought it was important enough, they wrote a paper about it, or tried to commercialize it, but what they didn't do was try to charge people for the idea itself.

A closely related point is the astonishing lack of gratitude some programmers show for the hard work and sense of community that went into building the knowledge base with which they work. How about this? Anyone who thinks they have a unique idea that they want to "own" and milk for money can do so—but first they have to track down and appropriately compensate all the people who made possible the compilers, algorithms, programming courses, books, hardware, and so forth that put them in a position to have their brainstorm.

Put that way, it sounds like a silly idea, but the idea behind software patents is precisely that eventually everyone will own parts of our communal knowledge base, and that programming will become in large part a process of properly identifying and compensating each and every owner of the techniques you use. All I can say is that if we do go down that path, I guarantee that it will be a poorer profession for all of us—except the patent attorneys, I guess.

Anecdote the third: A while back, I had the good fortune to have lunch down by Seattle's waterfront with Neal Stephenson, the author of *Snow Crash* and *The Diamond Age* (one of the best SF books I've come across in a long time). As he talked about the nature of networked technology and what he hoped to see emerge, he mentioned that a couple of blocks down the street was the pawn shop where Jimi Hendrix bought his first guitar. His point was that if a cheap guitar hadn't been available, Hendrix's unique talent would never have emerged. Similarly, he views the networking of society as a way to get affordable creative tools to many people, so as much talent as possible can be unearthed and developed.

Extend that to programming. The way it should work is that a steady flow of information circulates, so that everyone can do the best work they're capable of. The idea is that I don't gain by intellectually impoverishing you, and vice-versa; as we both compete and (intentionally or otherwise) share ideas, both our products become better, so the market grows larger and everyone benefits.

That's the way things have worked with programming for a long time. So far as I can see it has worked remarkably well, and the recent signs of change make me concerned about the future of our profession.

Things aren't changing *everywhere*, though; over the past year, I've circulated a good bit of info about 3-D graphics, and plan to keep on doing it as long as I can. Next, we're going to take a look at 3-D clipping.

3-D Clipping Basics

Before I got deeply into 3-D, I kept hearing how difficult 3-D clipping was, so I was pleasantly surprised when I actually got around to doing it and found that it was quite straightforward, after all. At heart, 3-D clipping is nothing more than evaluating whether and where a line intersects a plane; in this context, the plane is considered to have an "inside" (a side on which points are to be kept) and an "outside" (a side on which points are to be removed or clipped). We can easily extend this single operation to polygon clipping, working with the line segments that form the edges of a polygon.

The most common application of 3-D clipping is as part of the process of hidden surface removal. In this application, the four planes that make up the view volume, or view frustum, are used to clip away parts of polygons that aren't visible. Sometimes this process includes clipping to near and far plane, to restrict the depth of the scene. Other applications include clipping to splitting planes while building BSP trees, and clipping moving objects to convex sectors such as BSP leaves. The clipping principles I'll cover apply to any sort of 3-D clipping task, but clipping to the frustum is the specific context in which I'll discuss clipping below.

In a commercial application, you wouldn't want to clip every single polygon in the scene database individually. As I mentioned in the last chapter, the use of bounding volumes to cull chunks of the scene database that fall entirely outside the frustum, without having to consider each polygon separately, is an important performance aspect of scene rendering. Once that's done, however, you're still left with a set of polygons that may be entirely inside, or partially or completely outside, the frustum. In this chapter, I'm going to talk about how to clip those remaining polygons. I'll focus on the basics of 3-D clipping, the stuff I wish I'd known when I started doing 3-D. There are plenty of ways to speed up clipping under various circumstances, some of which I'll mention, but the material covered below will give you the tools you need to implement functional 3-D clipping.

Intersecting a Line Segment with a Plane

The fundamental 3-D clipping operation is clipping a line segment to a plane. There are two parts to this operation: determining if the line is clipped by (intersects) the plane at all and, if it is clipped, calculating the point of intersection.

Before we can intersect a line segment with a plane, we must first define how we'll represent the line segment and the plane. The segment will be represented in the obvious way by the (x,y,z) coordinates of its two endpoints; this extends well to polygons,

where each vertex is an (x,y,z) point. Planes can be described in many ways, among them are three points on the plane, a point on the plane and a unit normal, or a unit normal and a distance from the origin along the normal; we'll use the latter definition. Further, we'll define the normal to point to the inside (unclipped side) of the plane. The structures for points, polygons, and planes are shown in Listing 65.1.

LISTING 65.1 L65_1.h

```
typedef struct {
    double v[3];
} point_t;

typedef struct {
    double   x, y;
} point2D_t;

typedef struct {
    int         color;
    int         numverts;
    point_t     verts[MAX_POLY_VERTS];
} polygon_t;

typedef struct {
    int         color;
    int         numverts;
    point2D_t   verts[MAX_POLY_VERTS];
} polygon2D_t;

typedef struct convexobject_s {
    struct convexobject_s   *pnext;
    point_t                 center;
    double                  vdist;
    int                     numpolys;
    polygon_t               *ppoly;
} convexobject_t;

typedef struct {
    double  distance;
    point_t normal;
} plane_t;
```

Given a line segment, and a plane to which to clip the segment, the first question is whether the segment is entirely on the inside or the outside of the plane, or intersects the plane. If the segment is on the inside, then the segment is not clipped by the plane, and we're done. If it's on the outside, then it's entirely clipped, and we're likewise done. If it intersects the plane, then we have to remove the clipped portion of the line by replacing the endpoint on the outside of the plane with the point of intersection between the line and the plane.

The way to answer this question is to find out which side of the plane each endpoint is on, and the dot product is the right tool for the job. As you may recall from Chapter 61, dotting any vector with a unit normal returns the length of the projection of that vector onto the normal. Therefore, if we take any point and dot it with the plane normal we'll find out how far from the origin the point is, as measured along the

plane normal. Another way to think of this is to say that the dot product of a point and the plane normal returns how far from the origin along the normal the plane would have to be in order to have the point lie within the plane, as if we slid the plane along the normal until it touched the point.

Now, remember that our definition of a plane is a unit normal and a distance along the normal. That means that we have a distance for the plane as part of the plane structure, and we can get the distance at which the plane would have to be to touch the point from the dot product of the point and the normal; a simple comparison of the two values suffices to tell us which side of the plane the point is on. If the dot product of the point and the plane normal is greater than the plane distance, then the point is in front of the plane (inside the volume being clipped to); if it's less, then the point is outside the volume and should be clipped.

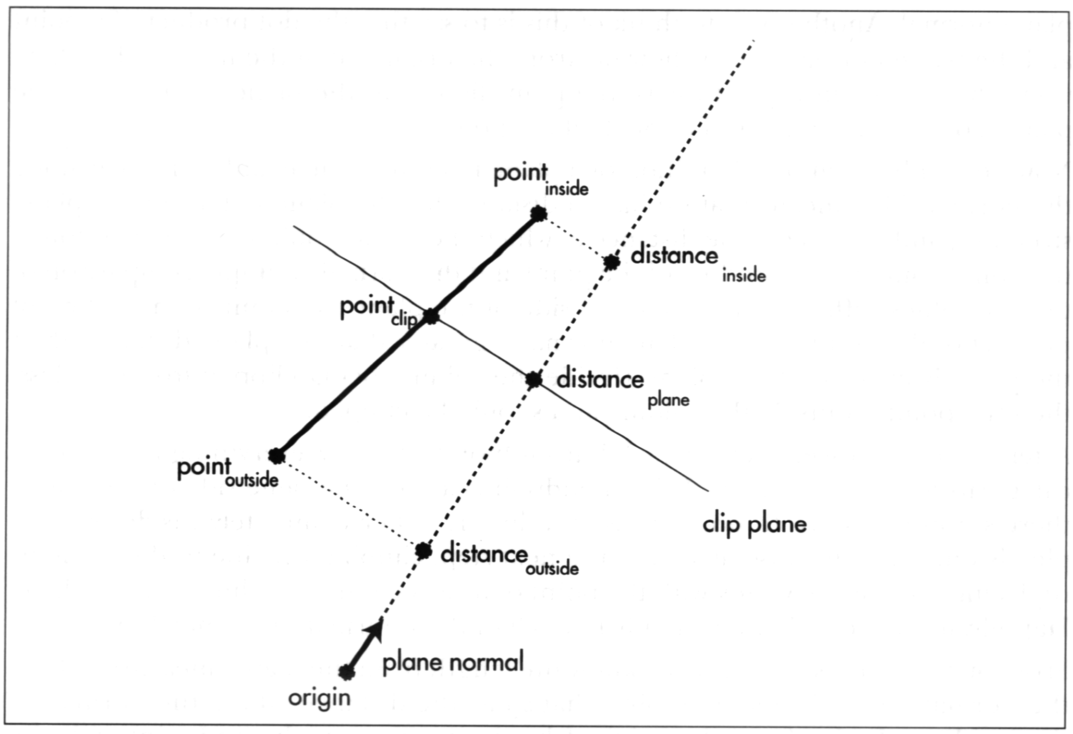
After we do this twice, once for each line endpoint, we know everything necessary to categorize our line segment. If both endpoints are on the same side of the plane, there's nothing more to do, because the line is either completely inside or completely outside; otherwise, it's on to the next step, clipping the line to the plane by replacing the outside vertex with the point of intersection of the line and the plane. Happily, it turns out that we already have all of the information we need to do this.

From our earlier tests, we already know the length from the plane, measured along the normal, to the inside endpoint; that's just the distance, along the normal, of the inside endpoint from the origin (the dot product of the endpoint with the normal), minus the plane distance, as shown in Figure 65.1. We also know the length of the line segment, again measured as projected onto the normal; that's the difference between the distances along the normal of the inside and outside endpoints from the origin. The ratio of these two lengths is the fraction of the segment that remains after clipping. If we scale the x, y, and z lengths of the line segment by that fraction, and add the results to the inside endpoint, we get a new, clipped endpoint at the point of intersection.

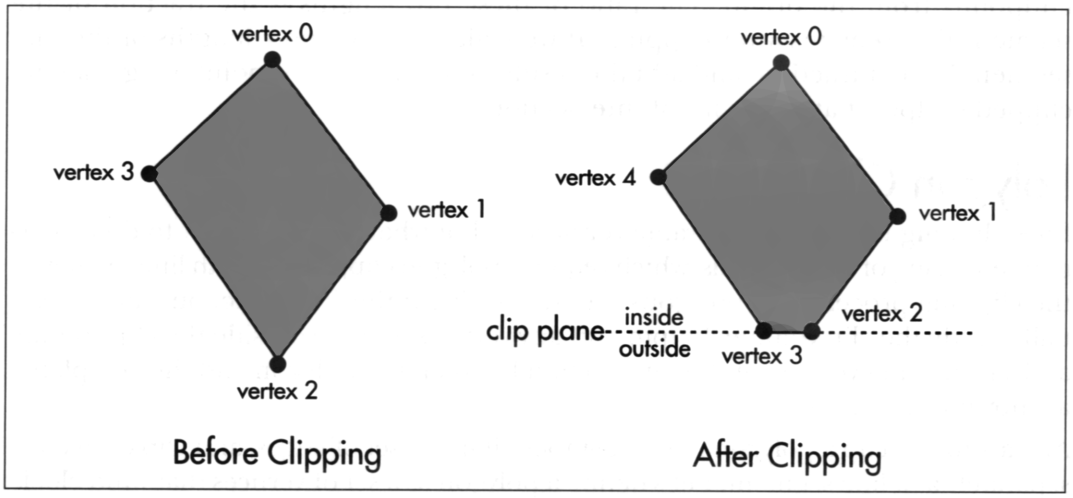
Polygon Clipping

Line clipping is fine for wireframe rendering, but what we really want to do is polygon rendering of solid models, which requires polygon clipping. As with line segments, the clipping process with polygons is to determine if they're inside, outside, or partially inside the clip volume, lopping off any vertices that are outside the clip volume and substituting vertices at the intersection between the polygon and the clip plane, as shown in Figure 65.2.

An easy way to clip a polygon is to decompose it into a set of edges, and clip each edge separately as a line segment. Let's define a polygon as a set of vertices that wind clockwise around the outside of the polygonal area, as viewed from the front side of the polygon; the edges are implicitly defined by the order of the vertices. Thus, an edge is



The distance from the plane to the inside endpoint, measured along the normal.
Figure 65.1



Clipping a polygon.
Figure 65.2

the line segment described by the two adjacent vertices that form its endpoints. We'll clip a polygon by clipping each edge individually, emitting vertices for the resulting polygon as appropriate, depending on the clipping state of the edge. If the start point of the edge is inside, that point is added to the output polygon. Then, if the start and end points are in different states (one inside and one outside), we clip the edge to the plane, as described above, and add the point at which the line intersects the clip plane as the next polygon vertex, as shown in Figure 65.3. Listing 65.2 shows a polygon-clipping function.

LISTING 65.2 L65_2.c

```
int ClipToPlane(polygon_t *pin, plane_t *pplane, polygon_t *pout)
{
    int     i, j, nextvert, curin, nextin;
    double  curdot, nextdot, scale;
    point_t *pinvert, *poutvert;

    pinvert = pin->verts;
    poutvert = pout->verts;

    curdot = DotProduct(pinvert, &pplane->normal);
    curin = (curdot >= pplane->distance);

    for (i=0 ; i<pin->numverts ; i++)
    {
        nextvert = (i + 1) % pin->numverts;

        // Keep the current vertex if it's inside the plane
        if (curin)
            *poutvert++ = *pinvert;

        nextdot = DotProduct(&pin->verts[nextvert], &pplane->normal);
        nextin = (nextdot >= pplane->distance);

        // Add a clipped vertex if one end of the current edge is
        // inside the plane and the other is outside
        if (curin != nextin)
        {
            scale = (pplane->distance - curdot) /
                    (nextdot - curdot);
            for (j=0 ; j<3 ; j++)
            {
                poutvert->v[j] = pinvert->v[j] +
                    ((pin->verts[nextvert].v[j] - pinvert->v[j]) *
                     scale);
            }
            poutvert++;
        }

        curdot = nextdot;
        curin = nextin;
        pinvert++;
    }

    pout->numverts = poutvert - pout->verts;
    if (pout->numverts < 3)
        return 0;
```

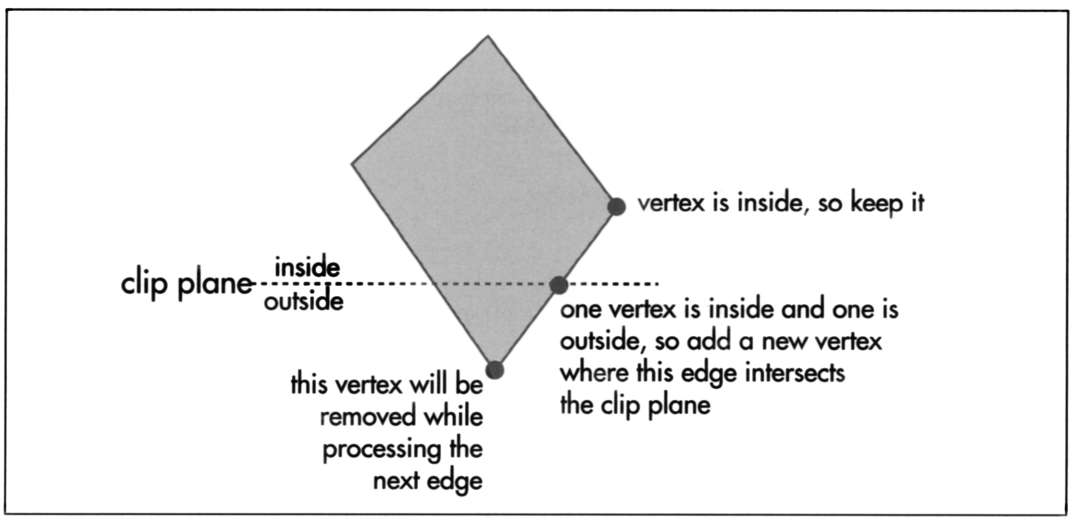
```
    pout->color = pin->color;
    return 1;
}
```

Believe it or not, this technique, applied in turn to each edge, is all that's needed to clip a polygon to a plane. Better yet, a polygon can be clipped to multiple planes by repeating the above process once for each clip plane, with each interation trimming away any part of the polygon that's clipped by that particular plane.

One particularly useful aspect of 3-D clipping is that if you're drawing texture mapped polygons, texture coordinates can be clipped in exactly the same way as (x,y,z) coordinates. In fact, the very same fraction that's used to advance x, y, and z from the inside point to the point of intersection with the clip plane can be used to advance the texture coordinates as well, so only one extra multiply and one extra add are required for each texture coordinate.

Clipping to the Frustum

Given a polygon-clipping function, it's easy to clip to the frustum: set up the four planes for the sides of the frustum, with another one or two planes for near and far clipping, if desired; next, clip each potentially visible polygon to each plane in turn; then draw whatever polygons emerge from the clipping process. Listing 65.3 is the core code for a simple 3-D clipping example that allows you to move around and look at polygonal models from any angle. The full code for this program is available on the CD-ROM in the file DDJCLIP.ZIP.



Clipping a polygon edge.
Figure 65.3

LISTING 65.3 L65_3.c

```
int DIBWidth, DIBHeight;
int DIBPitch;
double  roll, pitch, yaw;
double  currentspeed;
point_t currentpos;
double  fieldofview, xcenter, ycenter;
double  xscreenscale, yscreenscale, maxscale;
int     numobjects;
double  speedscale = 1.0;
plane_t frustumplanes[NUM_FRUSTUM_PLANES];
double  mroll[3][3] = {{1, 0, 0}, {0, 1, 0}, {0, 0, 1}};
double  mpitch[3][3] = {{1, 0, 0}, {0, 1, 0}, {0, 0, 1}};
double  myaw[3][3] =  {{1, 0, 0}, {0, 1, 0}, {0, 0, 1}};
point_t vpn, vright, vup;
point_t xaxis = {1, 0, 0};
point_t zaxis = {0, 0, 1};
convexobject_t objecthead = {NULL, {0,0,0}, -999999.0};

// Project viewspace polygon vertices into screen coordinates.
// Note that the y axis goes up in worldspace and viewspace, but
// goes down in screenspace.
void ProjectPolygon (polygon_t *ppoly, polygon2D_t *ppoly2D)
{
    int     i;
    double  zrecip;

    for (i=0 ; i<ppoly->numverts ; i++)
    {
        zrecip = 1.0 / ppoly->verts[i].v[2];
        ppoly2D->verts[i].x =
               ppoly->verts[i].v[0] * zrecip * maxscale + xcenter;
        ppoly2D->verts[i].y = DIBHeight -
             (ppoly->verts[i].v[1] * zrecip * maxscale + ycenter);
    }
    ppoly2D->color = ppoly->color;
    ppoly2D->numverts = ppoly->numverts;
}

// Sort the objects according to z distance from viewpoint.
void ZSortObjects(void)
{
    int             i, j;
    double          vdist;
    convexobject_t  *pobject;
    point_t         dist;

    objecthead.pnext = &objecthead;
    for (i=0 ; i<numobjects ; i++)
    {
        for (j=0 ; j<3 ; j++)
            dist.v[j] = objects[i].center.v[j] - currentpos.v[j];
        objects[i].vdist = sqrt(dist.v[0] * dist.v[0] +
                                dist.v[1] * dist.v[1] +
                                dist.v[2] * dist.v[2]);
        pobject = &objecthead;
        vdist = objects[i].vdist;
        // Viewspace-distance-sort this object into the others.
        // Guaranteed to terminate because of sentinel
        while (vdist < pobject->pnext->vdist)
            pobject = pobject->pnext;
```

```
            objects[i].pnext = pobject->pnext;
            pobject->pnext = &objects[i];
        }
    }

    // Move the view position and set the world->view transform.
    void UpdateViewPos()
    {
        int     i;
        point_t motionvec;
        double  s, c, mtemp1[3][3], mtemp2[3][3];

        // Move in the view direction, across the x-y plane, as if
        // walking. This approach moves slower when looking up or
        // down at more of an angle
        motionvec.v[0] = DotProduct(&vpn, &xaxis);
        motionvec.v[1] = 0.0;
        motionvec.v[2] = DotProduct(&vpn, &zaxis);
        for (i=0 ; i<3 ; i++)
        {
            currentpos.v[i] += motionvec.v[i] * currentspeed;
            if (currentpos.v[i] > MAX_COORD)
                currentpos.v[i] = MAX_COORD;
            if (currentpos.v[i] < -MAX_COORD)
                currentpos.v[i] = -MAX_COORD;
        }
        // Set up the world-to-view rotation.
        // Note: much of the work done in concatenating these matrices
        // can be factored out, since it contributes nothing to the
        // final result; multiply the three matrices together on paper
        // to generate a minimum equation for each of the 9 final elements
        s = sin(roll);
        c = cos(roll);
        mroll[0][0] = c;
        mroll[0][1] = s;
        mroll[1][0] = -s;
        mroll[1][1] = c;
        s = sin(pitch);
        c = cos(pitch);
        mpitch[1][1] = c;
        mpitch[1][2] = s;
        mpitch[2][1] = -s;
        mpitch[2][2] = c;
        s = sin(yaw);
        c = cos(yaw);
        myaw[0][0] = c;
        myaw[0][2] = -s;
        myaw[2][0] = s;
        myaw[2][2] = c;
        MConcat(mroll, myaw, mtemp1);
        MConcat(mpitch, mtemp1, mtemp2);
        // Break out the rotation matrix into vright, vup, and vpn.
        // We could work directly with the matrix; breaking it out
        // into three vectors is just to make things clearer
        for (i=0 ; i<3 ; i++)
        {
            vright.v[i] = mtemp2[0][i];
            vup.v[i] = mtemp2[1][i];
            vpn.v[i] = mtemp2[2][i];
        }
```

```
    // Simulate crude friction
    if (currentspeed > (MOVEMENT_SPEED * speedscale / 2.0))
        currentspeed -= MOVEMENT_SPEED * speedscale / 2.0;
    else if (currentspeed < -(MOVEMENT_SPEED * speedscale / 2.0))
        currentspeed += MOVEMENT_SPEED * speedscale / 2.0;
    else
        currentspeed = 0.0;
}

// Rotate a vector from viewspace to worldspace.
void BackRotateVector(point_t *pin, point_t *pout)
{
    int     i;

    // Rotate into the world orientation
    for (i=0 ; i<3 ; i++)
        pout->v[i] = pin->v[0] * vright.v[i] +
                     pin->v[1] * vup.v[i] +
                     pin->v[2] * vpn.v[i];
}

// Transform a point from worldspace to viewspace.
void TransformPoint(point_t *pin, point_t *pout)
{
    int     i;
    point_t tvert;

    // Translate into a viewpoint-relative coordinate
    for (i=0 ; i<3 ; i++)
        tvert.v[i] = pin->v[i] - currentpos.v[i];
    // Rotate into the view orientation
    pout->v[0] = DotProduct(&tvert, &vright);
    pout->v[1] = DotProduct(&tvert, &vup);
    pout->v[2] = DotProduct(&tvert, &vpn);
}

// Transform a polygon from worldspace to viewspace.
void TransformPolygon(polygon_t *pinpoly, polygon_t *poutpoly)
{
    int         i;

    for (i=0 ; i<pinpoly->numverts ; i++)
        TransformPoint(&pinpoly->verts[i], &poutpoly->verts[i]);
    poutpoly->color = pinpoly->color;
    poutpoly->numverts = pinpoly->numverts;
}

// Returns true if polygon faces the viewpoint, assuming a clockwise
// winding of vertices as seen from the front.
int PolyFacesViewer(polygon_t *ppoly)
{
    int     i;
    point_t viewvec, edge1, edge2, normal;

    for (i=0 ; i<3 ; i++)
    {
        viewvec.v[i] = ppoly->verts[0].v[i] - currentpos.v[i];
        edge1.v[i] = ppoly->verts[0].v[i] - ppoly->verts[1].v[i];
        edge2.v[i] = ppoly->verts[2].v[i] - ppoly->verts[1].v[i];
    }
```

```
    CrossProduct(&edge1, &edge2, &normal);
    if (DotProduct(&viewvec, &normal) > 0)
        return 1;
    else
        return 0;
}

// Set up a clip plane with the specified normal.
void SetWorldspaceClipPlane(point_t *normal, plane_t *plane)
{

    // Rotate the plane normal into worldspace
    BackRotateVector(normal, &plane->normal);
    plane->distance = DotProduct(&currentpos, &plane->normal) +
            CLIP_PLANE_EPSILON;
}

// Set up the planes of the frustum, in worldspace coordinates.
void SetUpFrustum(void)
{
    double  angle, s, c;
    point_t normal;

    angle = atan(2.0 / fieldofview * maxscale / xscreenscale);
    s = sin(angle);
    c = cos(angle);
    // Left clip plane
    normal.v[0] = s;
    normal.v[1] = 0;
    normal.v[2] = c;
    SetWorldspaceClipPlane(&normal, &frustumplanes[0]);
    // Right clip plane
    normal.v[0] = -s;
    SetWorldspaceClipPlane(&normal, &frustumplanes[1]);
    angle = atan(2.0 / fieldofview * maxscale / yscreenscale);
    s = sin(angle);
    c = cos(angle);
    // Bottom clip plane
    normal.v[0] = 0;
    normal.v[1] = s;
    normal.v[2] = c;
    SetWorldspaceClipPlane(&normal, &frustumplanes[2]);
    // Top clip plane
    normal.v[1] = -s;
    SetWorldspaceClipPlane(&normal, &frustumplanes[3]);
}

// Clip a polygon to the frustum.
int ClipToFrustum(polygon_t *pin, polygon_t *pout)
{
    int         i, curpoly;
    polygon_t   tpoly[2], *ppoly;

    curpoly = 0;
    ppoly = pin;
    for (i=0 ; i<(NUM_FRUSTUM_PLANES-1); i++)
    {
        if (!ClipToPlane(ppoly,
                         &frustumplanes[i],
                         &tpoly[curpoly]))
            return 0;
```

```
        ppoly = &tpoly[curpoly];
        curpoly ^= 1;
    }
    return ClipToPlane(ppoly,
                       &frustumplanes[NUM_FRUSTUM_PLANES-1],
                       pout);
}

// Render the current state of the world to the screen.
void UpdateWorld()
{
    HPALETTE        holdpal;
    HDC             hdcScreen, hdcDIBSection;
    HBITMAP         holdbitmap;
    polygon2D_t     screenpoly;
    polygon_t       *ppoly, tpoly0, tpoly1, tpoly2;
    convexobject_t  *pobject;
    int             i, j, k;

    UpdateViewPos();
    memset(pDIBBase, 0, DIBWidth*DIBHeight);    // clear frame
    SetUpFrustum();
    ZSortObjects();
    // Draw all visible faces in all objects
    pobject = objecthead.pnext;
    while (pobject != &objecthead)
    {
        ppoly = pobject->ppoly;
        for (i=0 ; i<pobject->numpolys ; i++)
        {
            // Move the polygon relative to the object center
            tpoly0.color = ppoly->color;
            tpoly0.numverts = ppoly->numverts;
            for (j=0 ; j<tpoly0.numverts ; j++)
            {
                for (k=0 ; k<3 ; k++)
                    tpoly0.verts[j].v[k] = ppoly->verts[j].v[k] +
                            pobject->center.v[k];
            }
            if (PolyFacesViewer(&tpoly0))
            {
                if (ClipToFrustum(&tpoly0, &tpoly1))
                {
                    TransformPolygon (&tpoly1, &tpoly2);
                    ProjectPolygon (&tpoly2, &screenpoly);
                    FillPolygon2D (&screenpoly);
                }
            }
            ppoly++;
        }
        pobject = pobject->pnext;
    }
    // We've drawn the frame; copy it to the screen
    hdcScreen = GetDC(hwndOutput);
    holdpal = SelectPalette(hdcScreen, hpalDIB, FALSE);
    RealizePalette(hdcScreen);
    hdcDIBSection = CreateCompatibleDC(hdcScreen);
    holdbitmap = SelectObject(hdcDIBSection, hDIBSection);
    BitBlt(hdcScreen, 0, 0, DIBWidth, DIBHeight, hdcDIBSection,
           0, 0, SRCCOPY);
```

```
    SelectPalette(hdcScreen, holdpal, FALSE);
    ReleaseDC(hwndOutput, hdcScreen);
    SelectObject(hdcDIBSection, holdbitmap);
    ReleaseDC(hwndOutput, hdcDIBSection);
}
```

The Lessons of Listing 65.3

There are several interesting points to Listing 65.3. First, floating-point arithmetic is used throughout the clipping process. While it is possible to use fixed-point, doing so requires considerable care regarding range and precision. Floating-point is much easier—and, with the Pentium generation of processors, is generally comparable in speed. In fact, for some operations, such as multiplication in general and division when the floating-point unit is in single-precision mode, floating-point is much faster. Check out Chris Hecker's column in the February 1996 *Game Developer* for an interesting discussion along these lines.

Second, the planes that form the frustum are shifted ever so slightly inward from their proper positions at the edge of the field of view. This guarantees that it's never possible to generate a visible vertex exactly at the eyepoint, averting the divide-by-zero error that such a vertex would cause when projected and at no performance cost.

Third, the orientation of the viewer relative to the world is specified via yaw, pitch, and roll angles, successively applied in that order. These angles are accumulated from frame to frame according to user input, and for each frame are used to rotate the view up, view right, and viewplane normal vectors, which define the world coordinate system, into the viewspace coordinate system; those transformed vectors in turn define the rotation from worldspace to viewspace. (See Chapter 61 for a discussion of coordinate systems and rotation, and take a look at Chapters 5 and 6 of *Computer Graphics*, by Foley and van Dam, for a broader overview.) One attractive aspect of accumulating angular rotations that are then applied to the coordinate system vectors is that there is no deterioration of the rotation matrix over time. This is in contrast to my X-Sharp package, in which I accumulated rotations by keeping a cumulative matrix of all the rotations ever performed; unfortunately, that approach caused roundoff error to accumulate, so objects began to warp visibly after many rotations.

Fourth, Listing 65.3 processes each input polygon into a clipped polygon, one line segment at a time. It would be more efficient to process all the vertices, categorizing whether and how they're clipped, and then perform a test such as the Cohen-Sutherland outcode test to detect trivial acceptance (the polygon is entirely inside) and sometimes trivial rejection (the polygon is fully outside) without ever dealing with the edges, and to identify which planes actually need to be clipped against, as discussed in "Line-Segment Clipping Revisited," *Dr. Dobb's Journal*, January 1996. Some clipping approaches also minimize the number of intersection calculations when a segment is clipped by multiple planes. Further, Listing 65.3 clips a polygon against each plane in turn, generating a new output polygon for each plane; it is possible

and can be more efficient to generate the final, clipped polygon without any intermediate representations. For further reading on advanced clipping techniques, see the discussion starting on page 271 of Foley and van Dam.

Finally, clipping in Listing 65.3 is performed in worldspace, rather than in viewspace. The frustum is backtransformed from viewspace (where it is defined, since it exists relative to the viewer) to worldspace for this purpose. Worldspace clipping allows us to transform only those vertices that are visible, rather than transforming all vertices into viewspace, then clipping them. However, the decision whether to clip in worldspace or viewspace is not clear-cut and is affected by several factors.

Advantages of Viewspace Clipping

Although viewspace clipping requires transforming vertices that may not be drawn, it has potential performance advantages. For example, in worldspace, near and far clip planes are just additional planes that have to be tested and clipped to, using dot products. In viewspace, near and far clip planes are typically planes with constant z coordinates, so testing whether a vertex is near or far-clipped can be performed with a single z compare, and the fractional distance along a line segment to a near or far clip intersection can be calculated with a couple of z subtractions and a divide; no dot products are needed.

Similarly, if the field of view is exactly 90 degrees, so the frustum planes go out at 45 degree angles relative to the viewplane, then x==z and y==z along the clip planes. This means that the clipping status of a vertex can be determined with a simple comparison, far more quickly than the standard dot-product test. This lends itself particularly well to outcode-based clipping algorithms, since each compare can set one outcode bit.

For a game, 90 degrees is a pretty good field of view, but can we get the same sort of efficient clipping if we need some other field of view? Sure. All we have to do is scale the x and y results of the world-to-view transformation to account for the field of view, so that the coordinates lie in a viewspace that's normalized such that the frustum planes extend along lines of x==z and y==z. The resulting visible projected points span the range -1 to 1 (before scaling up to get pixel coordinates), just as with a 90-degree field of view, so the rest of the drawing pipeline remains unchanged. Better yet, there is no cost in performance because the adjustment can be added to the transformation matrix.

I didn't implement normalized clipping in Listing 65.3 because I wanted to illustrate the general 3-D clipping mechanism without additional complications, and because for many applications the dot product (which, after all, takes only 10-20 cycles on a Pentium) is sufficient. However, the more frustum clipping you're doing, especially if most of the polygons are trivially visible, the more attractive the performance advantages of normalized clipping become.

Further Reading

You now have the basics of 3-D clipping, but because fast clipping is central to high-performance 3-D, there's a lot more to be learned. One good place for further reading is Foley and van Dam; another is *Procedural Elements of Computer Graphics*, by David F. Rogers. Read and understand either of these books, and you'll know everything you need for world-class clipping.

And, as you read, you might take a moment to consider how wonderful it is that anyone who's interested can tap into so much expert knowledge for the price of a book—or, on the Internet, for free—with no strings attached. Our part of the world is a pretty good place right now, isn't it?