Chapter 62

One Story, Two Rules, and a BSP Renderer

Chapter 62

Taking a Compiled BSP Tree from Logical to Visual Reality

As I've noted before, I'm working on Quake, id Software's follow-up to DOOM. A month or so back, we added page flipping to Quake, and made the startling discovery that the program ran nearly twice as fast with page flipping as it did with the alternative method of drawing the whole frame to system memory, then copying it to the screen. We were delighted by this, but baffled. I did a few tests and came up with several possible explanations, including slow writes through the external cache, poor main memory performance, and cache misses when copying the frame from system memory to video memory. Although each of these can indeed affect performance, none seemed to account for the magnitude of the speedup, so I assumed there was some hidden hardware interaction at work. Anyway, "why" was secondary; what really mattered was that we had a way to double performance, which meant I had a lot of work to do to support page flipping as widely as possible.

A few days ago, I was using the Pentium's built-in performance counters to seek out areas for improvement in Quake and, for no particular reason, checked the number of writes performed while copying the frame to the screen in non-page-flipped mode. The answer was 64,000. That seemed odd, since there were 64,000 byte-sized pixels to copy, and I was calling **memcpy()**, which of course performs copies a dword at a time whenever possible. I thought maybe the Pentium counters report the number of bytes written rather than the number of writes performed, but fortunately, this

time I tested my assumptions by writing an ASM routine to copy the frame a dword at a time, without the help of **memcpy()**. This time the Pentium counters reported 16,000 writes.

Whoops.

As it turns out, the **memcpy()** routine in the DOS version of our compiler (gcc) inexplicably copies memory a byte at a time. With my new routine, the non-page-flipped approach suddenly became slightly *faster* than page flipping.

The first relevant rule is pretty obvious: *Assume nothing.* Measure early and often. Know what's really going on when your program runs, if you catch my drift. To do otherwise is to risk looking mighty foolish.

The second rule: When you do look foolish (and trust me, it *will* happen if you do challenging work) have a good laugh at yourself, and use it as a reminder of Rule #1. I hadn't done any extra page-flipping work yet, so I didn't waste any time due to my faulty assumption that **memcpy()** performed a maximum-speed copy, but that was just luck. I should have done experiments until I was sure I knew what was going on before drawing any conclusions and acting on them.

In general, make it a point not to fall into a tightly focused rut; stay loose and think of alternative possibilities and new approaches, and always, always, always keep asking questions. It'll pay off big in the long run. If I hadn't indulged my curiosity by running the Pentium counter test on the copy to the screen, even though there was no specific reason to do so, I would never have discovered the ***memcpy()*** *problem—and by so doing I doubled the performance of the entire program in five minutes, a rare accomplishment indeed.*

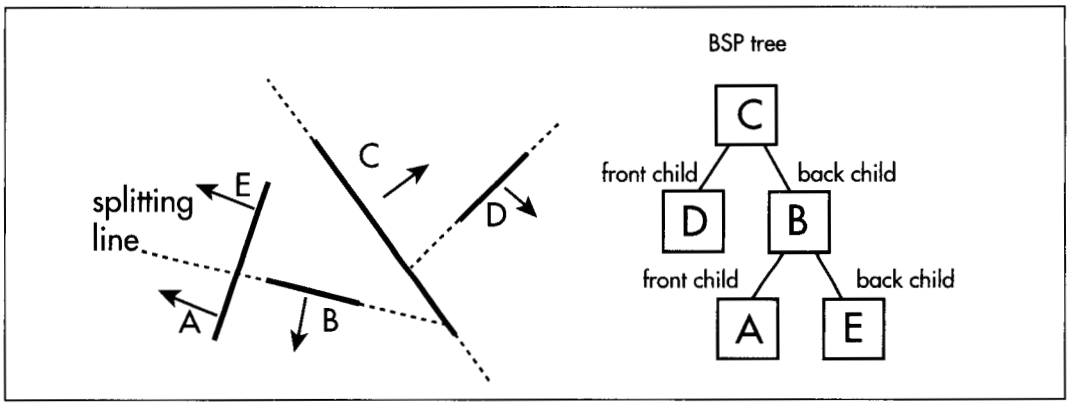
By the way, I have found the Pentium's performance counters to be very useful in figuring out what my code really does and where the cycles are going. One useful source of information on the performance counters and other aspects of the Pentium is Mike Schmit's book, *Pentium Processor Optimization Tools*, AP Professional, ISBN 0-12-627230-1.

Onward to rendering from a BSP tree.

BSP-based Rendering

For the last several chapters I've been discussing the nature of BSP (Binary Space Partitioning) trees, and in Chapter 60 I presented a compiler for 2-D BSP trees. Now we're ready to use those compiled BSP trees to do realtime rendering.

As you'll recall, the BSP compiler took a list of vertical walls and built a 2-D BSP tree from the walls, as viewed from above. The result is shown in Figure 62.1. The world is split into two pieces by the line of the root wall, and each half of the world is then split again by the root's children, and so on, until the world is carved into subspaces along the lines of all the walls.



Vertical walls and a BSP tree to represent them.
Figure 62.1

Our objective is to draw the world so that whenever walls overlap we see the nearer wall at each overlapped pixel. The simplest way to do that is with the painter's algorithm; that is, drawing the walls in back-to-front order, assuming no polygons interpenetrate or form cycles. BSP trees guarantee that no polygons interpenetrate (such polygons are automatically split), and make it easy to walk the polygons in back-to-front (or front-to-back) order.

Given a BSP tree, in order to render a view of that tree, all we have to do is descend the tree, deciding at each node whether we're seeing the front or back of the wall at that node from the current viewpoint. We use that knowledge to first recursively descend and draw the farther subtree of that node, then draw that node, and finally draw the nearer subtree of that node. Applied recursively from the root of our BSP trees, this approach guarantees that overlapping polygons will always be drawn in back-to-front order. Listing 62.1 draws a BSP-based world in this fashion. (Because of the constraints of the printed page, Listing 62.1 is only the core of the BSP renderer, without the program framework, some math routines, and the polygon rasterizer; but, the entire program is on the CD-ROM as DDJBSP2.ZIP. Listing 62.1 is in a compressed format, with relatively little whitespace; the full version on the CD-ROM is formatted normally.)

LISTING 62.1 L62_1.C

```
/* Core renderer for Win32 program to demonstrate drawing from a 2-D
   BSP tree; illustrate the use of BSP trees for surface visibility.
   UpdateWorld() is the top-level function in this module.
   Full source code for the BSP-based renderer, and for the
   accompanying BSP compiler, may be downloaded from
   ftp.idsoftware.com/mikeab, in the file ddjbsp2.zip.
   Tested with VC++ 2.0 running on Windows NT 3.5. */
#define FIXEDPOINT(x)   ((FIXEDPOINT)(((long)x)*((long)0x10000)))
#define FIXTOINT(x)     ((int)(x >> 16))
```

```
#define ANGLE(x)        ((long)x)
#define STANDARD_SPEED  (FIXEDPOINT(20))
#define STANDARD_ROTATION (ANGLE(4))
#define MAX_NUM_NODES   2000
#define MAX_NUM_EXTRA_VERTICES   2000
#define WORLD_MIN_X  (FIXEDPOINT(-16000))
#define WORLD_MAX_X  (FIXEDPOINT(16000))
#define WORLD_MIN_Y  (FIXEDPOINT(-16000))
#define WORLD_MAX_Y  (FIXEDPOINT(16000))
#define WORLD_MIN_Z  (FIXEDPOINT(-16000))
#define WORLD_MAX_Z  (FIXEDPOINT(16000))
#define PROJECTION_RATIO (2.0/1.0)  // controls field of view; the
                // bigger this is, the narrower the field of view
typedef long FIXEDPOINT;
typedef struct _VERTEX {
   FIXEDPOINT x, z, viewx, viewz;
} VERTEX, *PVERTEX;
typedef struct _POINT2 { FIXEDPOINT x, z; } POINT2, *PPOINT2;
typedef struct _POINT2INT { int  x; int y; } POINT2INT, *PPOINT2INT;
typedef long ANGLE;     // angles are stored in degrees
typedef struct _NODE {
   VERTEX *pstartvertex, *pendvertex;
   FIXEDPOINT  walltop, wallbottom, tstart, tend;
   FIXEDPOINT  clippedtstart, clippedtend;
   struct _NODE *fronttree, *backtree;
   int         color, isVisible;
   FIXEDPOINT  screenxstart, screenxend;
   FIXEDPOINT  screenytopstart, screenybottomstart;
   FIXEDPOINT  screenytopend, screenybottomend;
} NODE, *PNODE;
char * pDIB;            // pointer to DIB section we'll draw into
HBITMAP hDIBSection;    // handle of DIB section
HPALETTE hpalDIB;
int iteration = 0, WorldIsRunning = 1;
HWND hwndOutput;
int DIBWidth, DIBHeight, DIBPitch, numvertices, numnodes;
FIXEDPOINT fxHalfDIBWidth, fxHalfDIBHeight;
VERTEX *pvertexlist, *pextravertexlist;
NODE *pnodelist;
POINT2 currentlocation, currentdirection, currentorientation;
ANGLE currentangle;
FIXEDPOINT currentspeed, fxViewerY, currentYSpeed;
FIXEDPOINT FrontClipPlane = FIXEDPOINT(10);
FIXEDPOINT FixedMul(FIXEDPOINT x, FIXEDPOINT y);
FIXEDPOINT FixedDiv(FIXEDPOINT x, FIXEDPOINT y);
FIXEDPOINT FixedSin(ANGLE angle), FixedCos(ANGLE angle);
extern int FillConvexPolygon(POINT2INT * VertexPtr, int Color);
// Returns nonzero if a wall is facing the viewer, 0 else.
int WallFacingViewer(NODE * pwall)
{
   FIXEDPOINT viewxstart = pwall->pstartvertex->viewx;
   FIXEDPOINT viewzstart = pwall->pstartvertex->viewz;
   FIXEDPOINT viewxend = pwall->pendvertex->viewx;
   FIXEDPOINT viewzend = pwall->pendvertex->viewz;
   int Temp;
/*  // equivalent C code
   if (( ((pwall->pstartvertex->viewx >> 16) *
         ((pwall->pendvertex->viewz -
          pwall->pstartvertex->viewz) >> 16)) +
         ((pwall->pstartvertex->viewz >> 16) *
```

```
          ((pwall->pstartvertex->viewx -
            pwall->pendvertex->viewx) >> 16)) )
               < 0)
      return(1);
   else
      return(0);
*/
   _asm {
      mov   eax,viewzend
      sub   eax,viewzstart
      imul  viewxstart
      mov   ecx,edx
      mov   ebx,eax
      mov   eax,viewxstart
      sub   eax,viewxend
      imul  viewzstart
      add   eax,ebx
      adc   edx,ecx
      mov   eax,0
      jns   short WFVDone
      inc   eax
WFVDone:
      mov   Temp,eax
   }
   return(Temp);
}
// Update the viewpoint position as needed.
void UpdateViewPos()
{
   if (currentspeed != 0) {
      currentlocation.x += FixedMul(currentdirection.x,
                                    currentspeed);
      if (currentlocation.x <= WORLD_MIN_X)
         currentlocation.x = WORLD_MIN_X;
      if (currentlocation.x >= WORLD_MAX_X)
         currentlocation.x = WORLD_MAX_X - 1;
      currentlocation.z += FixedMul(currentdirection.z,
                                    currentspeed);
      if (currentlocation.z <= WORLD_MIN_Z)
         currentlocation.z = WORLD_MIN_Z;
      if (currentlocation.z >= WORLD_MAX_Z)
         currentlocation.z = WORLD_MAX_Z - 1;
   }
   if (currentYSpeed != 0) {
      fxViewerY += currentYSpeed;
      if (fxViewerY <= WORLD_MIN_Y)
         fxViewerY = WORLD_MIN_Y;
      if (fxViewerY >= WORLD_MAX_Y)
         fxViewerY = WORLD_MAX_Y - 1;
   }
}
// Transform all vertices into viewspace.
void TransformVertices()
{
   VERTEX *pvertex;
   FIXEDPOINT tempx, tempz;
   int vertex;
   pvertex = pvertexlist;
   for (vertex = 0; vertex < numvertices; vertex++) {
      // Translate the vertex according to the viewpoint
```

```
      tempx = pvertex->x - currentlocation.x;
      tempz = pvertex->z - currentlocation.z;
      // Rotate the vertex so viewpoint is looking down z axis
      pvertex->viewx = FixedMul(FixedMul(tempx,
                                    currentorientation.z) +
                  FixedMul(tempz, -currentorientation.x),
                  FIXEDPOINT(PROJECTION_RATIO));
      pvertex->viewz = FixedMul(tempx, currentorientation.x) +
                  FixedMul(tempz, currentorientation.z);
      pvertex++;
   }
}
// 3-D clip all walls. If any part of each wall is still visible,
// transform to perspective viewspace.
void ClipWalls()
{
   NODE *pwall;
   int wall;
   FIXEDPOINT tempstartx, tempendx, tempstartz, tempendz;
   FIXEDPOINT tempstartwalltop, tempstartwallbottom;
   FIXEDPOINT tempendwalltop, tempendwallbottom;
   VERTEX *pstartvertex, *pendvertex;
   VERTEX *pextravertex = pextravertexlist;
   pwall = pnodelist;
   for (wall = 0; wall < numnodes; wall++) {
      // Assume the wall won't be visible
      pwall->isVisible = 0;
      // Generate the wall endpoints, accounting for t values and
      // clipping
      // Calculate the viewspace coordinates for this wall
      pstartvertex = pwall->pstartvertex;
      pendvertex = pwall->pendvertex;
      // Look for z clipping first
      // Calculate start and end z coordinates for this wall
      if (pwall->tstart == FIXEDPOINT(0))
         tempstartz = pstartvertex->viewz;
      else
         tempstartz = pstartvertex->viewz +
               FixedMul((pendvertex->viewz-pstartvertex->viewz),
               pwall->tstart);
      if (pwall->tend == FIXEDPOINT(1))
         tempendz = pendvertex->viewz;
      else
         tempendz = pstartvertex->viewz +
               FixedMul((pendvertex->viewz-pstartvertex->viewz),
               pwall->tend);
      // Clip to the front plane
      if (tempendz < FrontClipPlane) {
         if (tempstartz < FrontClipPlane) {
            // Fully front-clipped
            goto NextWall;
         } else {
            pwall->clippedtstart = pwall->tstart;
            // Clip the end point to the front clip plane
            pwall->clippedtend =
                  FixedDiv(pstartvertex->viewz - FrontClipPlane,
                        pstartvertex->viewz-pendvertex->viewz);
            tempendz = pstartvertex->viewz +
               FixedMul((pendvertex->viewz-pstartvertex->viewz),
               pwall->clippedtend);
         }
```

```
} else {
   pwall->clippedtend = pwall->tend;
   if (tempstartz < FrontClipPlane) {
      // Clip the start point to the front clip plane
      pwall->clippedtstart =
            FixedDiv(FrontClipPlane - pstartvertex->viewz,
                  pendvertex->viewz-pstartvertex->viewz);
      tempstartz = pstartvertex->viewz +
         FixedMul((pendvertex->viewz-pstartvertex->viewz),
         pwall->clippedtstart);
   } else {
      pwall->clippedtstart = pwall->tstart;
   }
}
// Calculate x coordinates
if (pwall->clippedtstart == FIXEDPOINT(0))
   tempstartx = pstartvertex->viewx;
else
   tempstartx = pstartvertex->viewx +
         FixedMul((pendvertex->viewx-pstartvertex->viewx),
         pwall->clippedtstart);
if (pwall->clippedtend == FIXEDPOINT(1))
   tempendx = pendvertex->viewx;
else
   tempendx = pstartvertex->viewx +
         FixedMul((pendvertex->viewx-pstartvertex->viewx),
         pwall->clippedtend);
// Clip in x as needed
if ((tempstartx > tempstartz) || (tempstartx < -tempstartz)) {
   // The start point is outside the view triangle in x;
   // perform a quick test for trivial rejection by seeing if
   // the end point is outside the view triangle on the same
   // side as the start point
   if (((tempstartx>tempstartz) && (tempendx>tempendz)) ||
      ((tempstartx<-tempstartz) && (tempendx<-tempendz)))
      // Fully clipped--trivially reject
      goto NextWall;
   // Clip the start point
   if (tempstartx > tempstartz) {
      // Clip the start point on the right side
      pwall->clippedtstart =
         FixedDiv(pstartvertex->viewx-pstartvertex->viewz,
               pendvertex->viewz-pstartvertex->viewz -
               pendvertex->viewx+pstartvertex->viewx);
      tempstartx = pstartvertex->viewx +
         FixedMul((pendvertex->viewx-pstartvertex->viewx),
                pwall->clippedtstart);
      tempstartz = tempstartx;
   } else {
      // Clip the start point on the left side
      pwall->clippedtstart =
         FixedDiv(-pstartvertex->viewx-pstartvertex->viewz,
               pendvertex->viewx+pendvertex->viewz -
               pstartvertex->viewz-pstartvertex->viewx);
      tempstartx = pstartvertex->viewx +
         FixedMul((pendvertex->viewx-pstartvertex->viewx),
                pwall->clippedtstart);
      tempstartz = -tempstartx;
   }
}
```

```
// See if the end point needs clipping
if ((tempendx > tempendz) || (tempendx < -tempendz)) {
   // Clip the end point
   if (tempendx > tempendz) {
      // Clip the end point on the right side
      pwall->clippedtend =
         FixedDiv(pstartvertex->viewx-pstartvertex->viewz,
                pendvertex->viewz-pstartvertex->viewz -
                pendvertex->viewx+pstartvertex->viewx);
      tempendx = pstartvertex->viewx +
         FixedMul((pendvertex->viewx-pstartvertex->viewx),
                 pwall->clippedtend);
      tempendz = tempendx;
   } else {
      // Clip the end point on the left side
      pwall->clippedtend =
         FixedDiv(-pstartvertex->viewx-pstartvertex->viewz,
                pendvertex->viewx+pendvertex->viewz -
                pstartvertex->viewz-pstartvertex->viewx);
      tempendx = pstartvertex->viewx +
         FixedMul((pendvertex->viewx-pstartvertex->viewx),
                 pwall->clippedtend);
      tempendz = -tempendx;
   }
}
tempstartwalltop = FixedMul((pwall->walltop - fxViewerY),
      FIXEDPOINT(PROJECTION_RATIO));
tempendwalltop = tempstartwalltop;
tempstartwallbottom = FixedMul((pwall->wallbottom-fxViewerY),
      FIXEDPOINT(PROJECTION_RATIO));
tempendwallbottom = tempstartwallbottom;
// Partially clip in y (the rest is done later in 2D)
// Check for trivial accept
if ((tempstartwalltop > tempstartz) ||
   (tempstartwallbottom < -tempstartz) ||
   (tempendwalltop > tempendz) ||
   (tempendwallbottom < -tempendz)) {
   // Not trivially unclipped; check for fully clipped
   if ((tempstartwallbottom > tempstartz) &&
      (tempstartwalltop < -tempstartz) &&
      (tempendwallbottom > tempendz) &&
      (tempendwalltop < -tempendz)) {
      // Outside view triangle, trivially clipped
      goto NextWall;
   }
   // Partially clipped in Y; we'll do Y clipping at
   // drawing time
}
// The wall is visible; mark it as such and project it.
// +1 on scaling because of bottom/right exclusive polygon
// filling
pwall->isVisible = 1;
pwall->screenxstart =
   (FixedMulDiv(tempstartx, fxHalfDIBWidth+FIXEDPOINT(0.5),
      tempstartz) + fxHalfDIBWidth + FIXEDPOINT(0.5));
pwall->screenytopstart =
      (FixedMulDiv(tempstartwalltop,
      fxHalfDIBHeight + FIXEDPOINT(0.5), tempstartz) +
      fxHalfDIBHeight + FIXEDPOINT(0.5));
pwall->screenybottomstart =
      (FixedMulDiv(tempstartwallbottom,
```

```
            fxHalfDIBHeight + FIXEDPOINT(0.5), tempstartz) +
            fxHalfDIBHeight + FIXEDPOINT(0.5));
      pwall->screenxend =
            (FixedMulDiv(tempendx, fxHalfDIBWidth+FIXEDPOINT(0.5),
            tempendz) + fxHalfDIBWidth + FIXEDPOINT(0.5));
      pwall->screenytopend =
            (FixedMulDiv(tempendwalltop,
            fxHalfDIBHeight + FIXEDPOINT(0.5), tempendz) +
            fxHalfDIBHeight + FIXEDPOINT(0.5));
      pwall->screenybottomend =
            (FixedMulDiv(tempendwallbottom,
            fxHalfDIBHeight + FIXEDPOINT(0.5), tempendz) +
            fxHalfDIBHeight + FIXEDPOINT(0.5));
NextWall:
      pwall++;
   }
}
// Walk the tree back to front; backface cull whenever possible,
// and draw front-facing walls in back-to-front order.
void DrawWallsBackToFront()
{
   NODE *pFarChildren, *pNearChildren, *pwall;
   NODE *pendingnodes[MAX_NUM_NODES];
   NODE **pendingstackptr;
   POINT2INT apoint[4];
   pwall = pnodelist;
   pendingnodes[0] = (NODE *)NULL;
   pendingstackptr = pendingnodes + 1;
   for (;;) {
      for (;;) {
         // Descend as far as possible toward the back,
         // remembering the nodes we pass through on the way.
         // Figure whether this wall is facing frontward or
         // backward; do in viewspace because non-visible walls
         // aren't projected into screenspace, and we need to
         // traverse all walls in the BSP tree, visible or not,
         // in order to find all the visible walls
         if (WallFacingViewer(pwall)) {
            // We're on the forward side of this wall, do the back
            // children first
            pFarChildren = pwall->backtree;
         } else {
            // We're on the back side of this wall, do the front
            // children first
            pFarChildren = pwall->fronttree;
         }
         if (pFarChildren == NULL)
            break;
         *pendingstackptr = pwall;
         pendingstackptr++;
         pwall = pFarChildren;
      }
      for (;;) {
         // See if the wall is even visible
         if (pwall->isVisible) {
            // See if we can backface cull this wall
            if (pwall->screenxstart < pwall->screenxend) {
               // Draw the wall
               apoint[0].x = FIXTOINT(pwall->screenxstart);
               apoint[1].x = FIXTOINT(pwall->screenxstart);
```

```
                apoint[2].x = FIXTOINT(pwall->screenxend);
                apoint[3].x = FIXTOINT(pwall->screenxend);
                apoint[0].y = FIXTOINT(pwall->screenytopstart);
                apoint[1].y = FIXTOINT(pwall->screenybottomstart);
                apoint[2].y = FIXTOINT(pwall->screenybottomend);
                apoint[3].y = FIXTOINT(pwall->screenytopend);
                FillConvexPolygon(apoint, pwall->color);
             }
          }
          // If there's a near tree from this node, draw it;
          // otherwise, work back up to the last-pushed parent
          // node of the branch we just finished; we're done if
          // there are no pending parent nodes.
          // Figure whether this wall is facing frontward or
          // backward; do in viewspace because non-visible walls
          // aren't projected into screenspace, and we need to
             // traverse all walls in the BSP tree, visible or not,
             // in order to find all the visible walls
          if (WallFacingViewer(pwall)) {
             // We're on the forward side of this wall, do the
             // front children now
             pNearChildren = pwall->fronttree;
          } else {
             // We're on the back side of this wall, do the back
             // children now
             pNearChildren = pwall->backtree;
          }
          // Walk the near subtree of this wall
          if (pNearChildren != NULL)
             goto WalkNearTree;
          // Pop the last-pushed wall
          pendingstackptr--;
          pwall = *pendingstackptr;
          if (pwall == NULL)
             goto NodesDone;
       }
WalkNearTree:
       pwall = pNearChildren;
    }
NodesDone:
;
}
// Render the current state of the world to the screen.
void UpdateWorld()
{
   HPALETTE holdpal;
   HDC hdcScreen, hdcDIBSection;
   HBITMAP holdbitmap;
   // Draw the frame
   UpdateViewPos();
   memset(pDIB, 0, DIBPitch*DIBHeight);    // clear frame
   TransformVertices();
   ClipWalls();
   DrawWallsBackToFront();
   // 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);
    iteration++;
}
```

The Rendering Pipeline

Conceptually rendering from a BSP tree really is that simple, but the implementation is a bit more complicated. The full rendering pipeline, as coordinated by **UpdateWorld()**, is this:

- Update the current location.
- Transform all wall endpoints into viewspace (the world as seen from the current location with the current viewing angle).
- Clip all walls to the view pyramid.
- Project wall vertices to screen coordinates.
- Walk the walls back to front, and for each wall that lies at least partially in the view pyramid, perform backface culling (skip walls facing away from the viewer), and draw the wall if it's not culled.

Next, we'll look at each part of the pipeline more closely. The pipeline is too complex for me to be able to discuss each part in complete detail. Some sources for further reading are *Computer Graphics*, by Foley and van Dam (ISBN 0-201-12110-7), and the *DDJ Essential Books on Graphics Programming* CD.

Moving the Viewer

The sample BSP program performs first-person rendering; that is, it renders the world as seen from your eyes as you move about. The rate of movement is controlled by key-handling code that's not shown in Listing 62.1; however, the variables set by the key-handling code are used in **UpdateViewPos()** to bring the current location up to date.

Note that the view position can change not only in x and z (movement around the plane upon which the walls are set), but also in y (vertically). However, the view direction is always horizontal; that is, the code in Listing 62.1 supports moving to any 3-D point, but only viewing horizontally. Although the BSP tree is only 2-D, it is quite possible to support looking up and down to at least some extent, particularly if the world dataset is restricted so that, for example, there are never two rooms stacked on top of each other, or any tilted walls. For simplicity's sake, I have chosen not to implement this in Listing 62.1, but you may find it educational to add it to the program yourself.

Transformation into Viewspace

The viewing angle (which controls direction of movement as well as view direction) can sweep through the full 360 degrees around the viewpoint, so long as it remains horizontal. The viewing angle is controlled by the key handler, and is used to define a unit vector stored in **currentorientation** that explicitly defines the view direction (the z axis of viewspace), and implicitly defines the x axis of viewspace, because that axis is at right angles to the z axis, where x increases to the right of the viewer.

As I discussed in the previous chapter, rotation to a new coordinate system can be performed by using the dot product to project points onto the axes of the new coordinate system, and that's what **TransformVertices()** does, after first translating (moving) the coordinate system to have its origin at the viewpoint. (It's necessary to perform the translation first so that the viewing rotation is around the viewpoint.) Note that this operation can equivalently be viewed as a matrix math operation, and that this is in fact the more common way to handle transformations.

At the same time, the points are scaled in x according to **PROJECTION_RATIO** to provide the desired field of view. Larger scale values result in narrower fields of view.

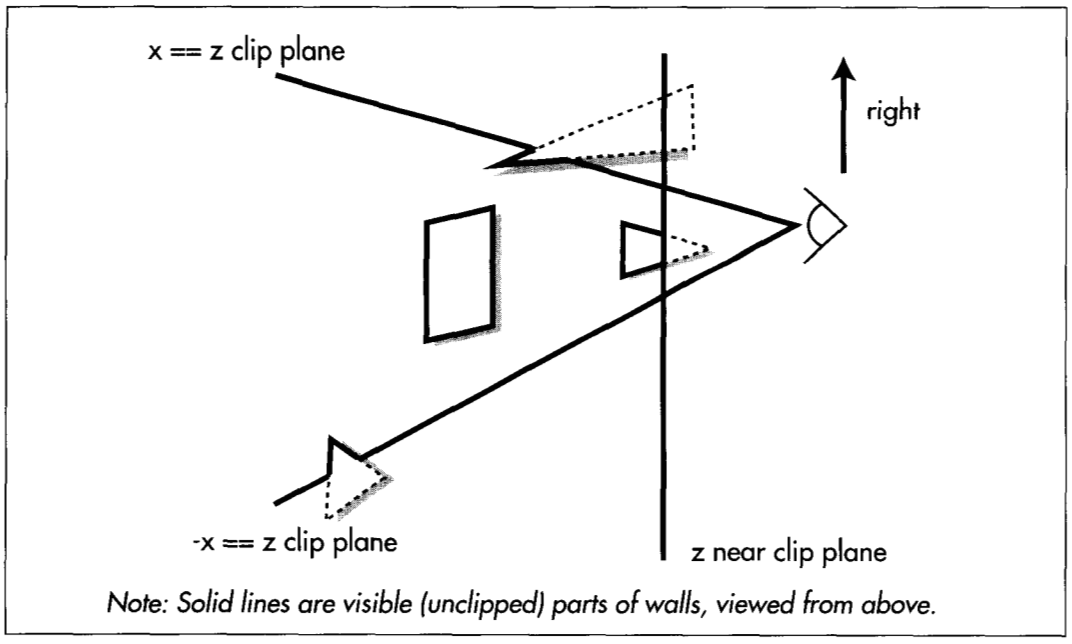
When this is done the walls are in viewspace, ready to be clipped.

Clipping

In viewspace, the walls may be anywhere relative to the viewpoint: in front, behind, off to the side. We only want to draw those parts of walls that properly belong on the screen; that is, those parts that lie in the view pyramid (view frustum), as shown in Figure 62.2. Unclipped walls—walls that lie entirely in the frustum—should be drawn in their entirety, fully clipped walls should not be drawn, and partially clipped walls must be trimmed before being drawn.

In Listing 62.1, **ClipWalls()** does this in three steps for each wall in turn. First, the z coordinates of the two ends of the wall are calculated. (Remember, walls are vertical and their ends go straight up and down, so the top and bottom of each end have the same x and z coordinates.) If both ends are on the near side of the front clip plane, then the polygon is fully clipped, and we're done with it. If both ends are on the far side, then the polygon isn't z-clipped, and we leave it unchanged. If the polygon straddles the near clip plane, then the wall is trimmed to stop at the near clip plane by adjusting the t value of the nearest endpoint appropriately; this calculation is a simple matter of scaling by z, because the near clip plane is at a constant z distance. (The use of t values for parametric lines was discussed in Chapter 60.) The process is further simplified because the walls can be treated as lines viewed from above, so we can perform 2-D clipping in z; this would not be the case if walls sloped or had sloping edges.

After clipping in z, we clip by viewspace x coordinate, to ensure that we draw only wall portions that lie between the left and right edges of the screen. Like z-clipping, x-clipping can be done as a 2-D clip, because the walls and the left and right sides of



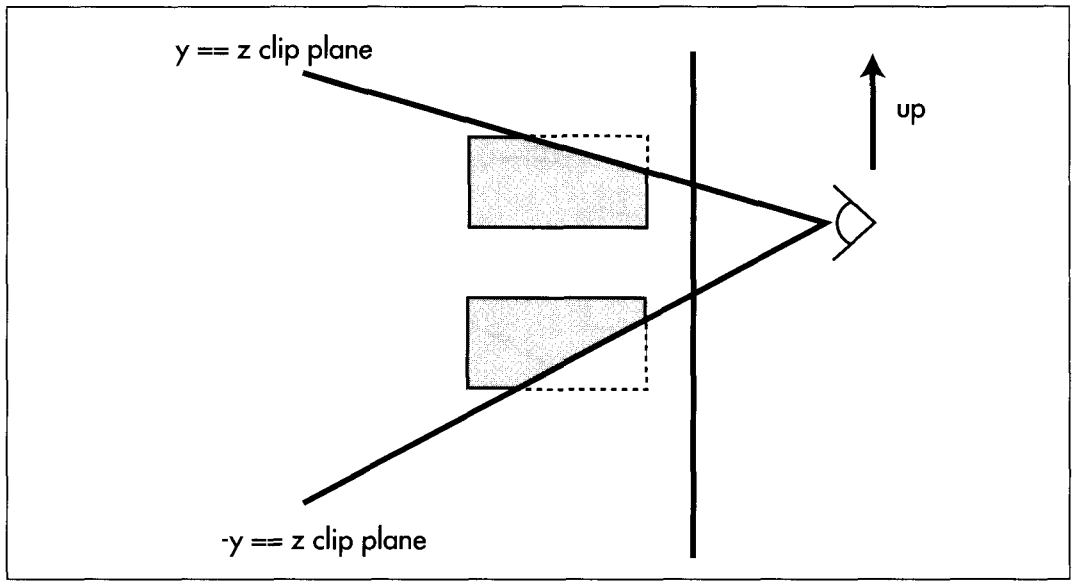
Clipping to the view pyramid.
Figure 62.2

the frustum are all vertical. We compare both the start and endpoint of each wall to the left and right sides of the frustum, and reject, accept, or clip each wall's t values accordingly. The test for x clipping is very simple, because the edges of the frustum are defined as the planes where x==z and -x==z.

The final clip stage is clipping by y coordinate, and this is the most complicated, because vertical walls can be clipped at an angle in y, as shown in Figure 62.3, so true 3-D clipping of all four wall vertices is involved. We handle this in **ClipWalls**() by detecting trivial rejection in y, using y==z and -y==z as the y boundaries of the frustum. However, we leave partial clipping to be handled as a 2-D clipping problem; we are able to do this only because our earlier z-clip to the near clip plane guarantees that no remaining polygon point can have z<=0, ensuring that when we project we'll always pass valid, y-clippable screenspace vertices to the polygon filler.

Projection to Screenspace

At this point, we have viewspace vertices for each wall that's at least partially visible. All we have to do is project these vertices according to z distance—that is, perform perspective projection—and scale the results to the width of the screen, then we'll be ready to draw. Although this step is logically separate from clipping, it is performed as the last step for visible walls in **ClipWalls**().



Why y clipping is more complex than x or z clipping.
Figure 62.3

Walking the Tree, Backface Culling and Drawing

Now that we have all the walls clipped to the frustum, with vertices projected into screen coordinates, all we have to do is draw them back to front; that's the job of **DrawWallsBackToFront()**. Basically, this routine walks the BSP tree, descending recursively from each node to draw the farther children of each node first, then the wall at the node, then the nearer children. In the interests of efficiency, this particular implementation performs a data-recursive walk of the tree, rather than the more familiar code recursion. Interestingly, the performance speedup from data recursion turned out to be more modest than I had expected, based on past experience; see Chapter 59 for further details.

As it comes to each wall, **DrawWallsBackToFront()** first descends to draw the farther subtree. Next, if the wall is both visible and pointing toward the viewer, it is drawn as a solid polygon. The polygon filler (not shown in Listing 62.1) is a modification of the polygon filler I presented in Chapters 38 and 39.

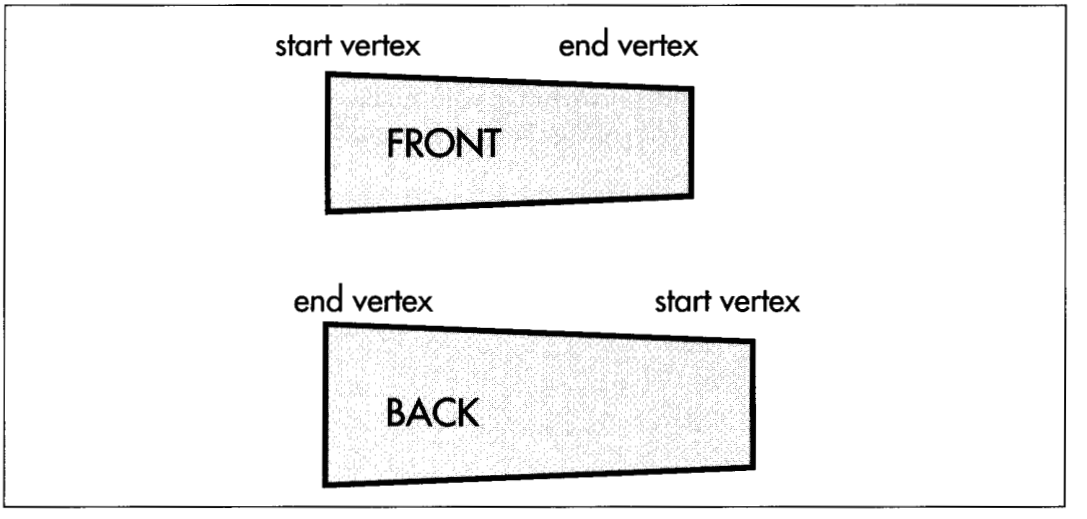
It's worth noting how backface culling and front/back wall orientation testing are performed. (Note that walls are always one-sided, visible only from the front.) I discussed backface culling in general in the previous chapter, and mentioned two possible approaches: generating a screenspace normal (perpendicular vector) to the polygon and seeing which way that points, or taking the world or screenspace dot product

between the vector from the viewpoint to any polygon point and the polygon's normal and checking the sign. Listing 62.1 does both, but because our BSP tree is 2-D and the viewer is always upright, we can save some work.

Consider this: Walls are stored so that the left end, as viewed from the front side of the wall, is the start vertex, and the right end is the end vertex. There are only two possible ways that a wall can be positioned in screenspace, then: viewed from the front, in which case the start vertex is to the left of the end vertex, or viewed from the back, in which case the start vertex is to the right of the end vertex, as shown in Figure 62.4. So we can tell which side of a wall we're seeing, and thus backface cull, simply by comparing the screenspace x coordinates of the start and end vertices, a simple 2-D version of checking the direction of the screenspace normal.

The wall orientation test used for walking the BSP tree, performed in **WallFacingViewer()**, takes the other approach, and checks the viewspace sign of the dot product of the wall's normal with a vector from the viewpoint to the wall. Again, this code takes advantage of the 2-D nature of the tree to generate the wall normal by swapping x and z and altering signs. We can't use the quicker screenspace x test here that we used for backface culling, because not all walls can be projected into screenspace; for example, trying to project a wall at z==0 would result in division by zero.

All the visible, front-facing walls are drawn into a buffer by **DrawWallsBackToFront()**, then **UpdateWorld()** calls Win32 to copy the new frame to the screen. The frame of animation is complete.



Fast backspace culling test in screenspace.
Figure 62.4

Notes on the BSP Renderer

Listing 62.1 is far from complete or optimal. There is no such thing as a tiny BSP rendering demo, because 3D rendering, even when based on a 2-D BSP tree, requires a substantial amount of code and complexity. Listing 62.1 is reasonably close to a minimum rendering engine, and is specifically intended to illuminate basic BSP principles, given the space limitations of one chapter in a book that's already larger than it should be. Think of Listing 62.1 as a learning tool and a starting point.

The most obvious lack in Listing 62.1 is that there is no support for floors and ceilings; the walls float in space, unsupported. Is it necessary to go to 3-D BSP trees to get a normal-looking world?

No. Although 3-D BSP trees offer many advantages in that they allow arbitrary datasets with viewing in any arbitrary direction and, in truth, aren't much more complicated than 2-D BSP trees for back-to-front drawing, they do tend to be larger and more difficult to debug, and they aren't necessary for floors and ceilings. One way to get floors and ceilings out of a 2-D BSP tree is to change the nature of the BSP tree so that polygons are no longer stored in the splitting nodes. Instead, each leaf of the tree—that is, each subspace carved out by the tree—would store the polygons for the walls, floors, and ceilings that lie on the boundaries of that space and face into that space. The subspace would be convex, because all BSP subspaces are automatically convex, so the polygons in that subspace can be drawn in any order. Thus, the subspaces in the BSP tree would each be drawn in turn as convex sets, back to front, just as Listing 62.1 draws polygons back to front.

This sort of BSP tree, organized around volumes rather than polygons, has some additional interesting advantages in simulating physics, detecting collisions, doing line-of-sight determination, and performing volume-based operations such as dynamic illumination and event triggering. However, that discussion will have to wait until another day.