```
1  //  Copyright (c) 1994 Knowledge Adventure.  All Rights Reserved.
2
3  #include <math.h>
4
5  #include <ka/std.h>
6
7  #include "avplane.h"
8
9  #include KA_INC(stdmath.h)
10 #include KA_INC(frob.h)
11
12 //----------------------------------------------------------------------------
13 /*static*/ BOOL                AVPlane::testVisible(
14     Vista*          vista,
15     Point3*         vLeft,
16     Point3*         vRight,
17     const Point2&   size)
18 //
19 // Is the vplane visible at all?  To find out, transform the object corners
20 // into viewer coords.
21 {
22         // First a quick check: if the object is facing away nothing to do.
23         // Facing away if viewer in back halfplane.
24     if (vista->_viewpoint._at.y >= 0)
25         return FALSE;
26
27     const Viewpoint& vp = vista->_viewpoint;
28
29         // Convert the corners to Viewer coordinate space.
30     if (vista->_viewpoint._dir == Angle::_pi2)
31     {
32         vLeft->x = -vp._at.x;
33         vLeft->y = -vp._at.z;
34         vLeft->z = -vp._at.y;
35
36         vRight->x = size._x - vp._at.x;
37         vRight->y = size._y - vp._at.z;
38         vRight->z = -vp._at.y;
39     } else
40     {
41         int sinA = vp._dir.sin();
42         int cosA = vp._dir.cos();
43
44         vLeft->x = StdMath_mulAdd(-vp._at.x, sinA, vp._at.y, cosA,
45                                                 Angle_COS0/2, Angle_LOGCOS0);
46         vLeft->y = - vp._at.z;
47         vLeft->z = StdMath_mulAdd(-vp._at.x, cosA, -vp._at.y, sinA,
48                                                 Angle_COS0/2, Angle_LOGCOS0);
49
50         int x = size._x - vp._at.x;
51         vRight->x = StdMath_mulAdd(x, sinA, vp._at.y, cosA, Angle_COS0/2,
52                                                             Angle_LOGCOS0);
```

```
53          vRight->y = size._y - vp._at.z;
54          vRight->z = StdMath_mulAdd(x, cosA, -vp._at.y, sinA, Angle_COS0/2,
55                                                                  Angle_LOGCOS0);
56      }
57
58          //-------------------------------------------------------
59          // Crude determination of whether anything of the object is visible.
60          // Find clipped approximate endpoints in x to b8 precision.
61          // They are approximate in
62          // that they might be overlarge, but they are guaranteed not to
63          // be too restrictive.  That is, for objects whose image projects
64          // beyond the window edges, the xLeft and xRight fall between the
65          // window edge and the actual viewplane-projected object edge.
66          // For objects that fit within the window, the x values are the
67          // exact viewport positions of the endpoints.
68          //
69          // Since we know the plane faces the origin from the initial quick
70          // check, we can do the rest of the check quite simply.  We work in
71          // the x dimension only, and test that the left and right edges are
72          // within a field-of-view cone with sides with slope 1/power-of-two:
73          //
74          //            Viewport:
75          //       \___         l-------r        ___/ Wide FieldOfView
76          //           \___         | R      ___/
77          //               \___     |    ___/ <- Pretend this is a straight line
78          //          L        \___|___/
79          //   --------------------V-------------------- X-axis
80          //                      | V is Viewer at origin.
81          //
82          // Imagine that the object runs from a left at L to a right at R
83          // in the diagram above.  Then L would give an isLeftOffScreen of
84          // TRUE, since it falls outside the field-of-view cone.  R, on the
85          // other hand, falls within the cone so isRightOffScreen would be
86          // FALSE. Any point within the cone is guaranteed not to overflow
87          // the StdMath_div.
88          //
89          // The shift of 0 used here corresponds to a 90-deg FOV cone.  We
90          // could use a shift of up to 14 without overflowing the
91          // StdMath_div below (14 + 16 = 30 bits).  But we use a smaller
92          // shift in order to weed out invisible walls so as to avoid needing
93          // to do further computation on them.
94          //
95      const int sShift = 0;   // 90 deg FOV.
96
97      BOOL leftOut = abs(vLeft->x) >= (vLeft->z << sShift);
98      BOOL rightOut = abs(vRight->x) >= (vRight->z << sShift);
99
100     int xLeft;
101     if (!leftOut)
102         xLeft = max(vista->_viewport.p1._x, // 16 is for viewportDist<<8
103                         StdMath_div(vLeft->x, vLeft->z, 16));
104     else
```

```
105     {
106         if (rightOut)           // If both out of view, is invisible unless
107             return vLeft->x < 0 //  vLeft->x<0 && vRight->x>0, which means
108              && vRight->x > 0;  //  that it spans the screen on both sides
109 
110         xLeft = vista->_viewport.p1._x;
111     }
112 
113     int xRight;
114     if (!rightOut)
115         xRight = min(vista->_viewport.p2._x,    // 16 is for viewportDist<<8
116                         StdMath_div(vRight->x, vRight->z, 16));
117     else
118         xRight = vista->_viewport.p2._x;
119 
120     return xLeft < xRight;
121 }
122 
123 
124 //---------------------------------------------------------------------------
125 static int                  projectEndpoint(int x, int z)
126 //
127 // Project clipped ends of object to viewPlane.
128 //     b8 = b8 * b8 / b8.  The 16 shift is viewportDist = 256<<8.
129 //
130 // Uses maximum s15b8 value on overflow.  This isn't correct, so the display
131 // will look wrong, but it works well for portals which might be right next
132 // to the camera but don't have anything of their own to display anyway so the
133 // exact values used here don't matter.
134 {
135     if (x < 0)
136     {
137         if (-x >= (z << 8))
138             return -32767 << 8;
139     } else
140     {
141         if (x >= (z << 8))
142             return 32767 << 8;
143     }
144 
145     return StdMath_div(x, z, 16);
146 }
147 
148 
149 //---------------------------------------------------------------------------
150 static void                     doProjectVertical(
151     const Point3&       left,
152     const Point3&       right,
153     AVPlane::Trapezoid* trap)
154 //
155 // Find the xs[12], ys[1234] project screen position values from the 3d
156 // left and right values.  Applies clipping to do this.
```

```
157 //
158 // Here is a drawing of a wall, used below to explain the notation used:
159 //
160 //  O__
161 //  |  4__       <-- ys4
162 //  |     \__
163 //  |   .....3__ <-- ys3
164 //  |   .   .   O       <--- right (3-d)
165 //  |   .   . __O
166 //  |   .....2   <-- ys2
167 //  |   __/
168 //  |__1         <-- ys1
169 //  O                  <--- left (3-d)
170 //     ^     ^
171 //    xs1   xs2
172 //
173 // In this diagram, the L,O,R symbols show the actual corners of the object.
174 // In practice they might even be behind the viewer, so that there is no way
175 // to represent them in a 2-d drawing, but for the purpose of this example
176 // they just stick out not too far beyond the viewwindow edges.
177 //
178 // The rectangle of periods represents the viewwindow.  The
179 // 1,2,3,4 symbols show the corners of the clipped wall, clipped so that it
180 // is bigger than the viewwindow, but small enough so that it can be
181 // guaranteed to project onto a finite part of the viewplane.  This is an
182 // x-dimension-only concept.  xs1 and xs2 are the results showing where the
183 // clipping points are.
184 //
185 // xs1 and xs2 are also chosen to clip out parts of the object that are too
186 // close to or are behind the viewer.  That is, there is a front clipping
187 // plane.  The position of the front clipping plane varies depending on the
188 // height of the image; it is always arranged so that the ys[1234] values fit
189 // in s15b8.
190 {
191     // Here's a top view:
192     //
193     //         R
194     //      __/____ <-- viewplane
195     //       / |
196     //      /  |   z
197     //     L   V   |_x
198     //
199     // Find the positions of the left and right endpoints of LR, clipped
200     // so their x components fall within an area of the viewplane
201     // representable as s15b8.  They must be clipped to a position outside
202     // the viewwindow to avoid letting the user see the clipping.
203     //
204     // A field of view with power-of-two sloping sides is used for the clip
205     // as in testVisible().  See the discussion there for more details.
206     //
207     // For the left endpoint, the
208     // clip position is the point on LR where x = -z.  LR is defined as:
```

```
209     //
210     //    (Lx + (Rx-Lx)*v, Ly + (Ry-Ly)*v, Lz + (Rz-Lz)*v)
211     //
212     // Setting x = -s*z gives:
213     //
214     //    Lx + (Rx-Lx)*v = -s*(Lz + (Rz-Lz)*v)
215     //    Lx+s*Lz = -v * ((Rx-Lx)+s*(Rz-Lz))
216     //    v = (Lx+s*Lz)/((Lx-Rx)+s*(Lz-Rz))
217     //
218     // Here, s is 1 for a 90 deg FOV, but can be set higher to
219     // widen the field of view if needed.
220     //
221     // We have first checked to verify that the line intersects the FOV
222     // edge, so it's guaranteed that v is in [0,1].  This allows the use of
223     // many bits of precision for v.
224     //
225     // If the wall passes so close to the viewer that the calcs below
226     // would overflow s15b8, the whole wall is clipped.  This case is
227     // triggered only for pathological situations.
228     //
229     // The v calc can't divide overflow due to approximation error since the
230     // computation is exact, and it can't overflow in the theoretical case
231     // of the line lying on the FOV edge because of the preceding "if".
232
233     Point3 clipLeft = left;
234     Point3 clipRight = right;
235
236     const int sShift = 0;   // 90 deg FOV.
237
238     if (left.x < (-left.z << sShift))
239     {
240         int v = StdMath_div(left.x + (left.z<<sShift),
241                         (left.x - right.x) + ((left.z - right.z)<<sShift),
242                         30);    // b30.
243         clipLeft.x += StdMath_mul(right.x - left.x, v, 0, 30);      // b8
244         clipLeft.z += StdMath_mul(right.z - left.z, v, 0, 30);      // b8
245     }
246     dAssert(clipLeft.z > 0);
247
248     if (right.x > (right.z << sShift))
249     {
250         // The right edge of the FOV is the same but with x=s instead of x=-s,
251         // and swapping left and right.
252         int v = StdMath_div(right.x - (right.z<<sShift),
253                         (right.x - left.x) - ((right.z - left.z)<<sShift),
254                         30);    // b30.
255         clipRight.x += StdMath_mul(left.x - right.x, v, 0, 30); // b8
256         clipRight.z += StdMath_mul(left.z - right.z, v, 0, 30); // b8
257     }
258     dAssert(clipRight.z > 0);
259
260         // Project the endpoints to the viewport.
```

DOCKET ALARM

Explore Litigation Insights

Docket Alarm provides insights to develop a more informed litigation strategy and the peace of mind of knowing you're on top of things.

Real-Time Litigation Alerts

Keep your litigation team up-to-date with **real-time alerts** and advanced team management tools built for the enterprise, all while greatly reducing PACER spend.

Our comprehensive service means we can handle Federal, State, and Administrative courts across the country.

Advanced Docket Research

With over 230 million records, Docket Alarm's cloud-native docket research platform finds what other services can't. Coverage includes Federal, State, plus PTAB, TTAB, ITC and NLRB decisions, all in one place.

Identify arguments that have been successful in the past with full text, pinpoint searching. Link to case law cited within any court document via Fastcase.

Analytics At Your Fingertips

Learn what happened the last time a particular judge, opposing counsel or company faced cases similar to yours.

Advanced out-of-the-box PTAB and TTAB analytics are always at your fingertips.

API

Docket Alarm offers a powerful API (application programming interface) to developers that want to integrate case filings into their apps.

LAW FIRMS
Build custom dashboards for your attorneys and clients with live data direct from the court.

Automate many repetitive legal tasks like conflict checks, document management, and marketing.

FINANCIAL INSTITUTIONS
Litigation and bankruptcy checks for companies and debtors.

E-DISCOVERY AND LEGAL VENDORS
Sync your system to PACER to automate legal marketing.