Overview of I, Robot 3D object format

Revision 0.1 – July 7 1998

(For the time being) you can reach me at lordfrito@comcast.net or john.manfreda@idselectronics.com)

Overview:

All static 3D objects are stored in the I, Robot ROMs numbered:

136029.101
136029.102
136029.103
136029.104

A total of 48 kilobytes of memory exists in these ROMs. However, the data stored in the ROMs is 16-bit, and hence a total of 24 kilowords of memory is specified in these ROMs.

The data in these ROMs is present in the 32 kiloword address space utilized by the mathbox. All addresses referenced here will refer to this address space. The address space is organized like the following:

`$0000-$0FFF` shared mathobox / CPU ram
`$1000-$1FFF` mathbox scratch RAM
`$1FFF-$7FFF` 3D object ROMs

NOTE: although the above mentioned ROMs hold static 3D information, the hardware is capable of and uses object information which is created dynamically in the RAM portion of the address space. The CPU is capable of generating dynamic 3D objects and placing them in the shared mathbox RAM area.

Objects:

In I, Robot, 3D objects can be drawn by specifying the base address of the object in the 32 kilword mathbox address space. Information at this address specifies information that can be used to generate 3D objects made of points, lines, and convex polygons.

At the base address of the object you will find:

- the base address of an array holding x/y/z vertex information (VERTEX structures)
- the object display list

Vertex base address:

The first value located at the objects base address is the base address of the VERTEX array associated with this object. At the VERTEX base address you will find an array of VERTEX structures specifying the 3D coordinates to be used in projecting the 3D image. Surfaces will specify vertices as relative offsets indices from the base address of the VERTEX array. See the section on Vertices for more information.

Object display list:

The object display list follows the vertex base address. The display list consists of a series of surface display instructions. This list is of variable length and is delimited by the value $8000.

Surface display instructions are either 2 or 3 WORD values in length. Currently there are two instructions that are known to exist:

Instruction	Length	Format
Display surface	2	`aaaaaaaaaaaaaaaa` **`0`**`-----tt-sccccc` `aaaaaaaaaaaaaaaa` = absolute address of surface display list `tt` = surface type `00` = polygon `01` = vector `10` = dot `s` = shading `0` = surface is not shaded (do not adjust color) `1` = surface is shaded (color index must be adjusted) `cccccc` = surface color, an index into the 64 entry RAM color palette
Cull surface	3	`aaaaaaaaaaaaaaaa` **`1`**`-ii--tt-sccccc` `rrrrrrrrrrrrrrr` `aaaaaaaaaaaaaaaa` = absolute address of surface display list `tt` = surface type `00` = polygon `01` = vector `10` = dot `11` = ??? `s` = shading `0` = surface is not shaded (do not adjust color) `1` = surface is shaded (color index must be adjusted) `cccccc` = surface color, an index into the 64 entry RAM color palette `ii` = culling instruction `00` = always take branch `01` = branch if this surface is visible `10` = branch if this surface is hidden `11` = ??? never draw ??? branch never ??? **NOTE** surface is not dawn if any bit in `ii` is set `rrrrrrrrrrrrrrr` = relative branch offset if culling branch is taken

Note that the size of the instructions is implicit – the instruction parser must determine the size of an instruction to determine the position of the next one. This is identical to the way a microprocessor would handle the fetching of instructions.

Hidden surface removal is achieved through the process of culling. The routine parsing the object display list may be instructed to branch around other display instructions, similar to branching on a microprocessor.

Any given surface in the display list may or may not be drawn, depending on whether the surface is visible (surface normal vector faces the camera) at the time. Culling uses this information to control parsing of the object display list. Certain instructions or groups of instructions may be completely bypassed if a particular surface is currently not visible.

The following is an example of an object display utilizing culling:

```
Object:
        1048                VERTEX structure array base address is $1048
        1137 000F           draw polygon surface at $1137 in color $0F
        113C 000F           draw polygon surface at $113C in color $0F
        1141 000F           draw polygon surface at $1141 in color $0F
        10E0 A007 000C      draw polygon surface at $10E0 in color $07
                            branch $000C WORDS to L1 if this surface is not visible
        10E0 0007           draw polygon surface at $10E0 in color $07
        10E5 0007           draw polygon surface at $10E5 in color $07
        10E9 0007           draw polygon surface at $10E9 in color $07
        10ED 0007           draw polygon surface at $10ED in color $07
        10F1 8007 000B      draw polygon surface at $10F1 in color $07
                            branch $000B words to L2 if this surface is visible
L1:     111D 0007           draw polygon surface at $111D in color $07
        1122 0007           draw polygon surface at $1122 in color $07
        1126 0007           draw polygon surface at $1126 in color $07
        112A 0007           draw polygon surface at $112A in color $07
        112E 0007           draw polygon surface at $112E in color $07
```

L2: 8000 end of object display list

Surfaces:

All surfaces are specified by a normal vector and a list coordinate indicies. Surfaces are specified as a list of indicies into the VERTEX array. The first index in the list points to a surface normal vector which can be used to perform hidden surface removal. All other indices point to coordinate information defining the edge of the polygon surface. The end of the list is delmited by an index which has the MSB set ($8000).

In general the surface list contains WORD sized values containing the following information:

```
dnooooooooooooo
```

where:

```
ooooooooooooo = unsigned offset specifying location of VERTEX array
            d = delimiter (set to 1 if this is the last surface index)
            n = no normal vector (set to 1 if the surface has no normal vector specified)
```

NOTE: this bit will only be set in first value of the surface display list

VERTEX lookup:

The index offset value is used to lookup the appropriate VERTEX structure from the VERTEX array specified for the object. However, it should be noted that the value is not an index into the array, but rather specifies the physical address offset from the base of the array.

Incorrect:

```
VERTEX *pVertex = VertexArray[offset];
```

Correct:

```
VERTEX *pVertex = (VERTEX *) (((WORD *) &VertexArray[0]) + offset)
```

Correct:

```
VERTEX *pVertex = VertexArray[offset / 4];
```

Normal vector:

Normally the first VERTEX in the surface display list specified the surface normal vector that will be used to perform hidden surface removal.

The x/y/z values of the normal vector are themselves fixed-precision integers, and all vectors encountered in the software can be assumed to be unit-vectors (vectors whose length is one unit). This makes the vectors suitable for shading. The vectors are 16-bits and signed, with 1 bit of integer precision and 14 bits of fractional precision. Conversion from fixed point to floating-point would be:

```
floating point = fixed_point / 2^14
               = fixed_point / 16384.0
```

Please note that certain surfaces do not have any normal vector specified. These surfaces cannot be shaded, and never have the hidden surface removal algorithm performed on them – they are always drawn. These surfaces can be detected by checking bit 14 of the first value in the surface display list, if this bit is set then no surface normal exists and the index should be treated as a coordinate.

Hidden surface removal:

Hidden surface removal can easily be performed by calculating the dot product of the surface normal vector and the object vector (the vector pointing from the camera to the object). The algorithm for determining if a surface is visible is the following:

```
BOOL IsSurfaceVisible( int16 x, int16 y, int16 z, int16 nx, int16 ny, int16 nz )
{
      // x = x position of object, relative to camera
      // y = z position of object, relative to camera
      // z = z position of object, relative to camera
      // nx = x component of surface normal vector
      // ny = y component of surface normal vector
      // nz = z component of surface normal vector

      int dot_product = nx*x + ny*y + nz*z;

      if (dot_product > 0)
            return TRUE; // surface is visible
```

```
        else
                return FALSE; //surface is hidden
    }
```

Surface shading:

A typical method for performing flat shading is by calculating the angle between the surface and a light source. The angle specifies how directly the light source is illuminating the surface. If the light source is directly hitting the surface, then the angle between the light source vector and the surface normal vector is 0º. If the light is hitting the surface edge-on then the angle will be 90º.

If the angle that the light source is hitting the surface is at is known, then a shade offset can be calculated. A simple shading method calculates 'percent illumination' as the cosine of the angle between the two vectors. The cosine of a head-on light source (0º) is 1.0 or 100% illumination. The cosine of an edge-on light source (90º) is 0.0 or 0% illumination. Light sources behind the surface will naturally calculate negative percentages, specifying a percentage of darkness.

I, Robot was designed to make surface shading simple because the specified surface normal vector is also a unit vector. The cosine of the angle between the surface and the light source is simply the dot product of the two unit vectors.

The dynamic RAM palette created by the I Robot program further simplifies the shading problem. The program creates palette bins containing 8 shades of a particular color, all in consecutive color indices. The color of a shaded surface is always specified to be the first color of any particular bin. A shade offset into the pallete can then be calculated by multiplying the percent illumination by a factor of 8. The following algorithm demonstrates this principle at work.

```
int ShadeSurface( int base_color, int16 lx, int16 ly, int16 lz, int16 nx, int16 ny, int16 nz)
{
        // lx = x component of lighting vector
        // ly = y component of lighting vector
        // lz = z component of lighting vector
        // nx = x component of surface normal vector
        // ny = y component of surface normal vector
        // nz = z component of surface normal vector

        // calculate the dot product
        int dot_product = nx*lx + ny*ly + nz*lz;

        // normalize the dot product to an index
        // (divide by 2^14 twice and multiply by 8)
        dot_product = dot_product >> 25;

        // use the normalized dot product as a color index
        color += min( 7, max( 0, dot_product ));

        return color;
}
```

Surface display list example:

The following are examples of surface display lists:

Surface 1:

0094 normal vector is at &VertexArray[0] + $94
0024 first coordinate of surface is at &VertexArray[0] + $24
0020 second coordinate of surface is at &VertexArray[0] + $20
8054 third and last coordinate of surface is at &VertexArray[0] + $54

Surface 2:

4000 no normal vector
0010 first coordinate of surface is at &VertexArray[0] + $10
0004 second coordinate of surface is at &VertexArray[0] + $04
8008 third and last coordinate of surface is at &VertexArray[0] + $08

Vertices:

Coordinate vertex information is stored in an array of VERTEX structures. Each structure is of the form:

```
typedef struct {
```

```
            int16 x;
            int16 y;
            int16 z;
            int16 type;
        } VERTEX;
```

where:

`x` = x value (signed)
`y` = y value (signed)
`z` = z value (signed)
`type` = `$xxx0` = vector
`$xxx1` = coordinate
`$8xxx` = end of VERTEX array

Every object has an associated VERTEX array, the base address of which is specified in the object display list. The mathbox address space and 3D object ROMs do not specify the number of VERTEX structures in a given array -- the only way to determine the end of the VERTEX array is by checking the 'type' value from each structure in the array.

The x/y/z values stored in the structure are fixed-precision integers. The information is 16-bits and signed, with 1 bit of integer precision and 14 bits of fractional precision. Conversion from fixed point to floating-point would be:

```
floating point = fixed_point / 2^14
               = fixed_point / 16384.0
```

Display algorithm:

The following is an example of an algorithm that could be used to display objects from these ROMs.

```
uint16 ObjectROM[0x8000];

VOID RasterizeObject( uint16 ObjectBaseAddress, int16 ObjX, int16 ObjY, int16 ObjZ, VERTEX* pLight )
{
   uint16* pVertexArray = ObjectROM[ObjectBaseAddress];
   uint16* pObjectDisplayList = ObjectROM[ObjectBaseAddress+1];

   // while object display list terminator not reached
   while (*pObjectDisplayList < 0x8000)
   {
      // load information pertaining to next surface in display list
      uint16* pSurfaceDisplayList = ObjectROM[*(pObjectDisplayList++)];
      uint16  SurfaceInstruction = *(pObjectDisplayList++);
      uint16  SurfaceColor = SurfaceInstruction & 0x003F;
      uint16  SurfaceType = SurfaceInstruction & 0x0300;
      uint16  CullInstruction = SurfaceInstruction & 0xB000;
      uint16  PerformShading = SurfaceInstruction & 0x0040;
      BOOL    SurfaceIsVisible;

      // check first index of the surface array for a normal vector
      if (!(*pSurfaceDisplayList & 0x4000))
      {
         // get the normal vector for the surface
         VERTEX* pNormal = (VERTEX*) &pVertexArray[*pSurfaceDisplayList & 0x3FFF];

         // use dot product to determine if the surface is visible
         int16 DotProduct = pNormal->x * ObjX + pNormal->y * ObjY + pNormal->z * ObjZ;
         SurfaceIsVisible = (DotProduct > 0);

         if (PerformShading)
         {
            // use dot produt to shade the surface
            DotProduct = (pNormal->x * pLight->x
                          + pNormal->y * pLight->y
                          + pNormal->z * pLight->z)
                          >> 25;
            SurfaceColor += min( 7, max( 0, DotProduct ));
         }
      }
      else
         SurfaceIsVisible = TRUE;
      pSurfaceDisplayList++;
```

```
      // rasterize surface only if it is visible
      if (SurfaceIsVisible)
      {
         VERTEX Temp[100];
         int n = 0;

         // locate all surface coordinates and pass them to rasterizer
         do
         {
            // get the coordinate from the surface display list
            VERTEX* pCoordinate = (VERTEX*) pVertexArray[*pSurfaceDisplayList & 0x3FFF];
            Temp[n].x = pCoordinate->x;
            Temp[n].y = pCoordinate->y;
            Temp[n].z = pCoordinate->z;
            n++;
         }
         // until until delimiter is reached
         while (!(*(pSurfaceDisplayList++) & 0x8000));

         // kick off the rasterizer now that we are done
         switch(SurfaceType)
         {
         case 0x0000: RasterizePolygon( Temp, n, SurfaceColor, POLYGON ); break;
         case 0x0100: RasterizeVector( Temp, n, SurfaceColor, VECTOR ); break;
         case 0x0200: RasterizeDot( Temp, n, SurfaceColor, DOT ); break;
         }
      }

      // resolve branch if this is a branch instruction
      if ( CullInstruction & 0x8000)
      {
            if (CullInstruction & 0x2000 && SurfaceIsVisible)
               pObjectDisplayList++;
            else if (CullInstruction & 0x1000 && !SurfaceIsVisible)
               pObjectDisplayList++;
            else
               pObjectDisplayList+= (int16) * pObjectDisplayList;
      }
   }
}
```