Object Recognition in Augmented Reality

Chester Pereira

Dept. of Computer Science,

University of Maine, Orono ME 04469

Project Advisor:

Dr. George Markowsky

Dept. of Computer Science,

University of Maine, Orono ME04469

Contents

1. Introduction
   1.1What is Augmented Reality?
   1.2 Augmented Reality vs. Virtual Reality
   1.3 Motivation
   1.4 Goals of this project
2. System Components
   2.1 Hardware components
   2.2 Software Used
   2.3 Working of the System
3. Calculating the user's view
   3.1 The viewing pipeline
   3.2 Transformation from World Co-ordinates to Eye Co-ordinates
   3.3 The Screen Co-ordinate system
   3.4 View-port Transformation
   3.5 Line Clipping
4. Implementation of the System
   4.1 Storing the known information of the scene
   4.2 Implementation of the viewing pipeline
   4.3 Video Capture operations
   4.4 Bitmap File formats
   4.5 The pciBIRD API
   4.6 Putting it all together
      4.6.1 Initializing required data and devices
      4.6.2 Processing each Frame: The *FrameProc()* function
   4.7 Experimental Results
5. Conclusion and Future Work
6. References

1. Introduction

1.1 What is Augmented Reality?

Augmented Reality (AR) is a growing research area that sprung out of Virtual Reality (VR). VR systems immerse the user in a completely synthetic environment thus blocking him out of the real world. In contrast AR systems aim at augmenting the user's perception of the world around him, enhancing his performance in the real world. This augmentation is done by superimposing synthetic elements such as sound or graphics over a real world environment in real time. Ideally, the user would not be able to distinguish between real and synthetic elements. Thus AR supplements reality instead of completely replacing it, as is the case with VR.

A typical AR system consists of the following components:

- ***Tracking System***
- ***Display System***
- ***Computing Device***

The tracking system is used to calculate the location of the user in reference to his surroundings. The display system is used to relay to the user the augmented view of the real world. The computing device does all the necessary computation to generate the augmented view from the information obtained by the tracking device and existing knowledge of the user's surroundings.

1.2 Augmented Reality vs. Virtual Reality

Virtual Reality systems tend to "fool" the user into believing that he is living in the virtual environment that is provided to him. In most cases this virtual world has very few features of the real world, if any at all. The user tolerates or even adapts to errors in the system. Features of the real world such as laws of physics that govern time, gravity and material properties may not hold good in such a system. On the other hand, an Augmented Reality system operates in the real world. Therefore, all the laws of physics must be taken care of: errors in the

system can be annoying to the user to the level of being distractive or even degrading his performance in the real world.

Milgram et. al. explain the difference between Augmented Reality and Virtual Reality by describing a Reality-Virtuality Continuum. (See figure1 below).

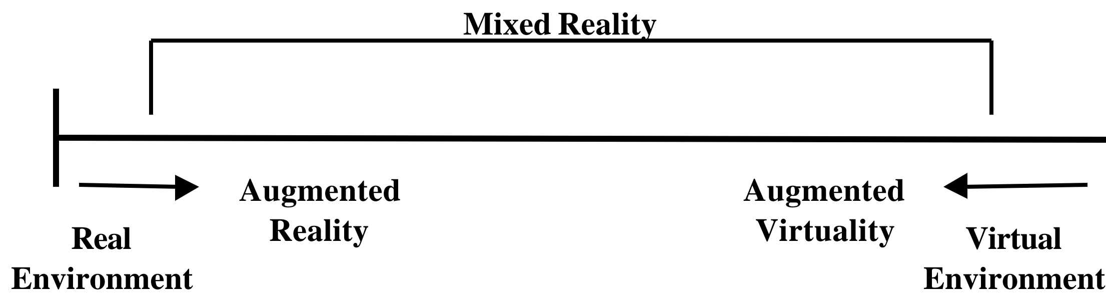

Figure1: The Reality – Virtuality Continuum

According to this concept, real and virtual environments lie on opposite ends of the continuum. The region in between is termed "Mixed Reality" to show that they contain both real and synthetic elements. Augmented Reality lies near the Real Environment end, signifying that the environment is mostly real, with a few synthetic elements overlaid. The authors use the term Augmented Virtuality to describe a system that is mostly synthetic, with a few features of the real world added on to the virtual objects. The ultimate aim of an Augmented Reality system would be to superimpose real and virtual elements in such a way that the user is unable to distinguish between them. We should bear in mind that this is not to done in order to fool the user, but to help him perform better in the real world.

1.3 Motivation

Augmented Reality is a relatively new research field and is fast growing. Computing devices are getting faster and more compact. For example, the **Microsoft Xbox** game console uses an NVidia 300Mhz 3-D Graphics Processing Unit that is capable of processing 150 Million polygons per second. Another

example is the **ASUS MyPal A600 Pocket PC** – It is a PDA that fits on the palm of one's hand, and it has a processor running at a speed of 400MHz.

While commercial vendors extensively sell VR systems, AR systems are primarily found in academic and industrial research laboratories. AR systems currently under development find applications in many realms including medicine, manufacturing, visualization, entertainment and military training. However, the systems developed are highly expensive and may cost up to a few tens of thousands of dollars. This project aims at developing a system that is relatively inexpensive. For example, ***MicroVision***'s Military HMD (Head Mounted Display) costs around $10,000; the display device chosen for this project costs a little less than $1,000.

1.4 Goals of this project

As stated in section **1.3**, the principle aim of this project is to develop a relatively inexpensive system that incorporates the concept of Augmented Reality. The system should be able to recognize simple objects (i.e., *objects that are simple to describe geometrically)* in a room, such as doors and windows. The objects are assumed to be static while the user may freely move around. The augmented view consists of drawing wire frames around the objects to be recognized, and text that explains to the user which object(s) he is currently viewing.

2. System Components

This section describes the hardware and software components used, and a general idea as to how the different components work together to form the system.

2.1 Hardware Components

The system comprises of the following hardware devices:

- **QuickCam Home Camera** (Logitech): A pc camera to capture the user's view. Video manipulation is carried out on each frame captured by the camera in order to generate the augmented view. The camera has a focusing range between 3 feet and infinity, and a field of view of 45^0.

- **pciBIRD** (Ascension Technology Corporation): A 6 degree-of-freedom DC magnetic tracker on a pci card. The system consists of one transmitter and two sensors. The tracker works by having the transmitter generate magnetic fields by precisely known characteristics. Each sensor measures the transmitted field vectors at a point in space. Using these measured values, it is possible to deduce the position and orientation of the sensor relative to the transmitter. The device has a translation range of 76.2 cm along three directions (X, Y and Z axes) and is capable of taking up to 105 measurements per second.

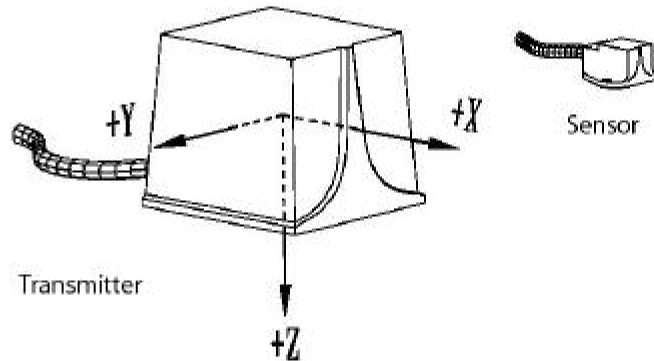

Figure 2: Magnetic tracker- sensor and transmitter

- **DELL Dimension 4100 Series Desktop PC:** Intel Pentium III processor @ 1000 MHz, to do the necessary computation.

- **"Clip-On" Model CO-1** (The MicroOptical Corporation): A small clip-on monocular LCD display monitor that can be clipped on to a pair of ordinary eyeglasses to relay the augmented view to the user. The image appears in front of the user and serves as a compact version of an ordinary desktop monitor. (Display format: 320 x 240, 16-bit color, 60Hz refresh rate).

Figure 3: Model CO-1 clip-on display monitor

The camera and sensors would typically be mounted on a hard hat: The camera and one of the sensors would be fitted to the front side of the hard hat, while the second sensor is attached to the behind of the hat. The first sensor indicates the position of the camera while the second sensor serves as a ***view reference point***. The line segment between these two points is the direction of view of the user.

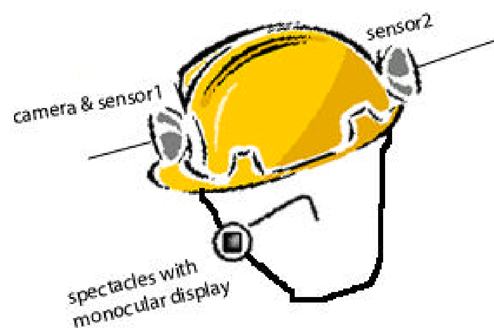

Figure 4: Components mounted on a hard hat

2.2 Software used

The code was written in Microsoft VC++ on the Windows2000 platform. The ***Video for Windows*** (VFW) API provided by Microsoft was used to carry out the necessary video capture and video manipulation operations. Interfacing of the tracking device to the software was done through the ***pciBIRD*** C++ API which contains all the required libraries to do the same.

2.3 Working of the System

We assume that the system has predefined knowledge of the objects to be recognized. Since we are dealing with simple rectangular objects, all we need to do is store the end points of the rectangle in 3-dimensional co-ordinates, and the textual description that needs to be displayed. These values may be stored in a file and read into appropriate data structures whenever necessary.

VFW provides for a preview of every frame captured. This allows for each frame to be modified before it is displayed. Whenever a frame is captured, the positions of the sensors are noted and the user's view is estimated. For each object, we check to see if it is in the user's view. If it is, the necessary perspective transformations are applied. The next section describes these computations in more detail. The wire-frames are then drawn around each object and the required text is overlaid. Once this is done, the frame is displayed on the screen.

3. Calculating the user's view

3.1 The Viewing Pipeline

The viewing pipeline describes the different steps involved in calculating the final view on the screen, given the co-ordinates of an object in the world co-ordinate system. The following sub-sections of this section describe each step in better detail.

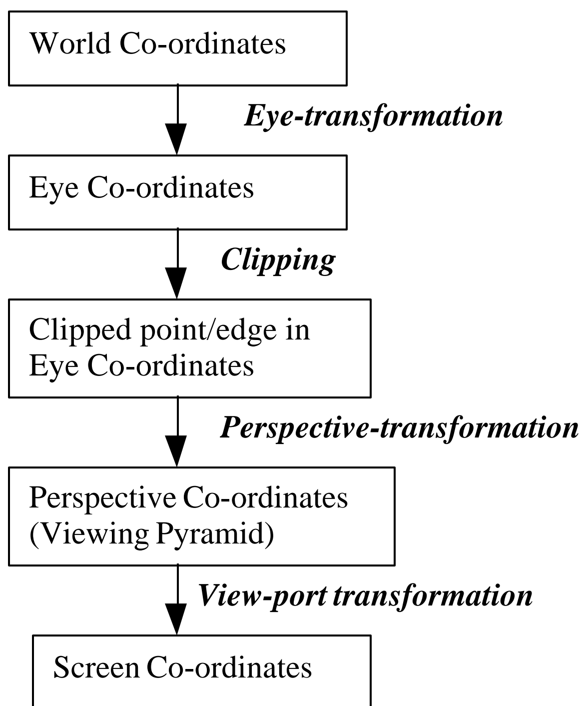

Figure 5: The viewing pipeline

3.2 Transformation from World Co-ordinates to Eye Co-ordinates

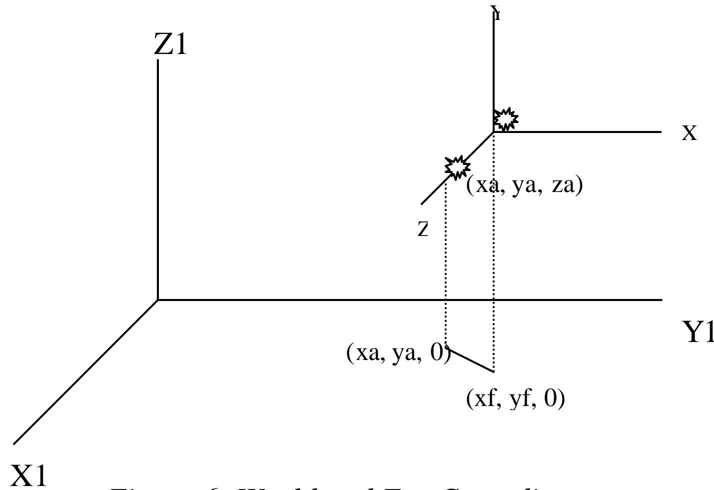

Figure 6: World and Eye Co-ordinate systems

In the diagram above X1, Y1 and Z1 represent the axes in the world co-ordinate system. The origin of the magnetic transmitter forms the origin of our world system and the sensor readings obtained are in this co-ordinate system. X, Y and Z represent the eye co-ordinate system. Note that the world system is a right-handed system while the eye system is left-handed. The eye is always at the center of projection (xf,yf,zf) and looks at the view reference point (xa,ya,za). The eye forms the origin of the eye system, and the line from the eye to the view reference point (vrp) forms the Z-axis of the eye system. The Z-axis of the eye system is always perpendicular to the XY plane in the eye system.

The co-ordinates of the end points are stored as 3-dimensional co-ordinates (x,y,z). Using homogeneous matrix representation, this point may be represented as:

$$P = [x\ y\ z\ 1].$$

In order to convert this point into an equivalent point in the eye system co-ordinates, we take a look at transforming the world co-ordinate system to the eye co-ordinate system:

- Translate the world system's origin to the origin of the eye system.
- Shift to a left handed system.
- Rotate by 90^0 around the X-axis.

- Rotate about the Y-axis until the Z-axis is above the Z-axis of the eye system.
- Rotate about the X-axis so that the Z-axis drops down to the Z-axis of the eye system.

Representing each of these transformations by a 4x 4 matrix, and multiplying them to obtain he transformation matrix gives us:

$$EYE = \begin{vmatrix} -\cos\theta & -\sin\theta * \sin\psi & -\sin\theta * \cos\psi & 0 \\ \sin\theta & -\cos\theta * \sin\psi & -\cos\theta * \cos\psi & 0 \\ 0 & \cos\psi & -\sin\psi & 0 \\ xf*\cos\theta - yf*\sin\psi & -zf*\cos\psi + xf*\sin\theta*\sin\psi + yf*\cos\theta*\sin\psi & zf*\sin\psi + xf*\sin\theta*\cos\psi + yf*\cos\theta*\cos\psi & 1 \end{vmatrix}$$

Here,

$\sin\theta = (xf-xa)/d1$ and $\cos\theta = (yf-ya)/d1$.

Also,

$\sin\psi = (zf-za)/d2$ and $\cos\psi = d1/d2$.

Where

$d1 = \sqrt{((xf-xa)^2+(yf-ya)^2)}$ and $d2 = \sqrt{((xf-xa)^2+(yf-ya)^2+(zf-za)^2)}$

In other words, we need to multiply P by the matrix EYE to obtain the equivalent co-ordinates in the eye system.

3.3 The Screen Co-ordinate system

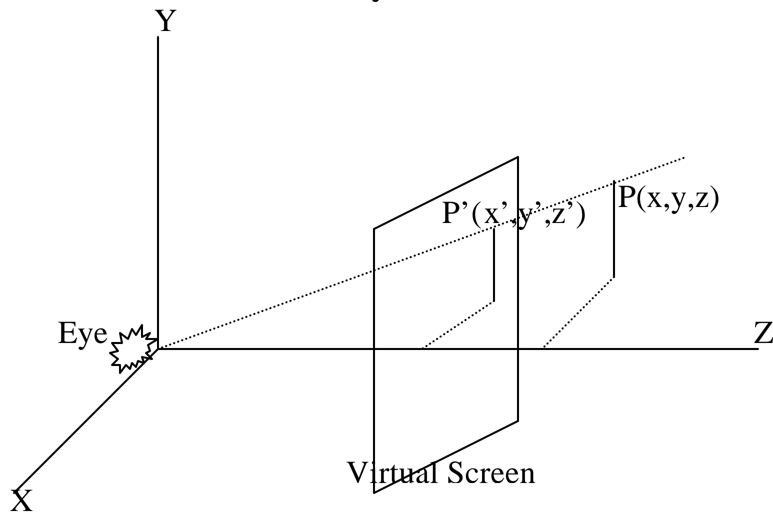

Figure 7: Eye and virtual-screen co-ordinates

In converting the object co-ordinates from world to eye co-ordinates, we have obtained the user's view of the object. We now need to convert these co-ordinates to the screen system for display purposes.

A point in space is projected along a line to the eye where it meets the virtual screen. The eye and screen boundaries determine the visible region of space. This is called the **viewing pyramid**.(See Figure 8 below)

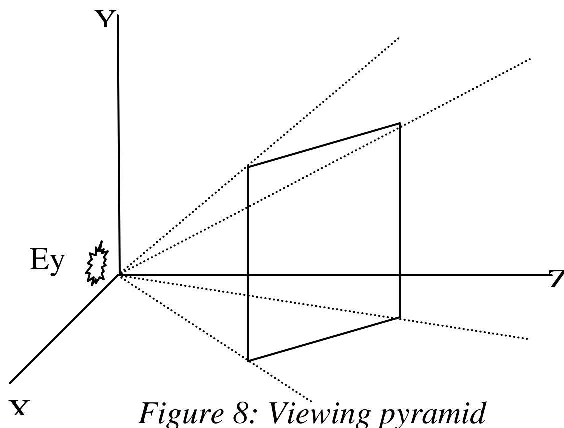

Figure 8: Viewing pyramid

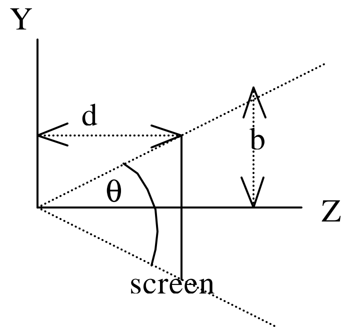

Figure 9: Aperture and field of view

The ratio **b/d** (see figure 9) is called the **viewing aperture** or just aperture. The angle **q** is the field of view. Thus, **aperture = tan (q/2)**. [In our case, $\theta = 45^0$. So, the aperture would be **tan (45/2)** = **0.4142** (approximately).]

Normalizing the co-ordinates of the point P that is under consideration (*i.e.,* ***0<=x', y'<=1)***, we get

x'=x/(z*a) and **y'=y/(z*a)**, where **a** is the aperture.

At this point, we have converted the point P from 3-dimensional eye co-ordinates 2-dimensional virtual screen co-ordinates.

3.4 View-port Transformation

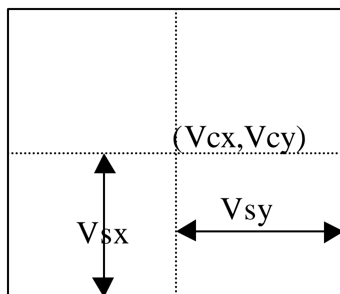

Figure 10: Viewing region of actual screen

In order to calculate the actual screen co-ordinates **(xs,ys)** of the point **(x',y')**, we define the rectangular viewing region by specifying the center **(Vcx,Vcy)** and distances to the top and side of the screen **Vsx** and **Vsy** respectively.

Hence we obtain:

xs = Vcx + x'*Vsx and

ys = Vcy + y'*Vsy.

3.5 Line Clipping

Before converting the scene from the eye-system to the screen system, we need to obtain those portions of the lines (which form the borders of the objects to be recognized) that lie within the viewing pyramid. In other words, we need to clip these lines against each of the four faces of the viewing pyramid. These faces form the planes:

x = z , y = z, x = - z and y = - z.

Figure 11 below depicts the algorithm to clip a line **(x_1, y1, z1) - (x_1, y1, z1)** against the **x=z** plane. Similar procedures are carried out on the rest of the faces of the viewing pyramid to determine the visible portion(s) of the line.

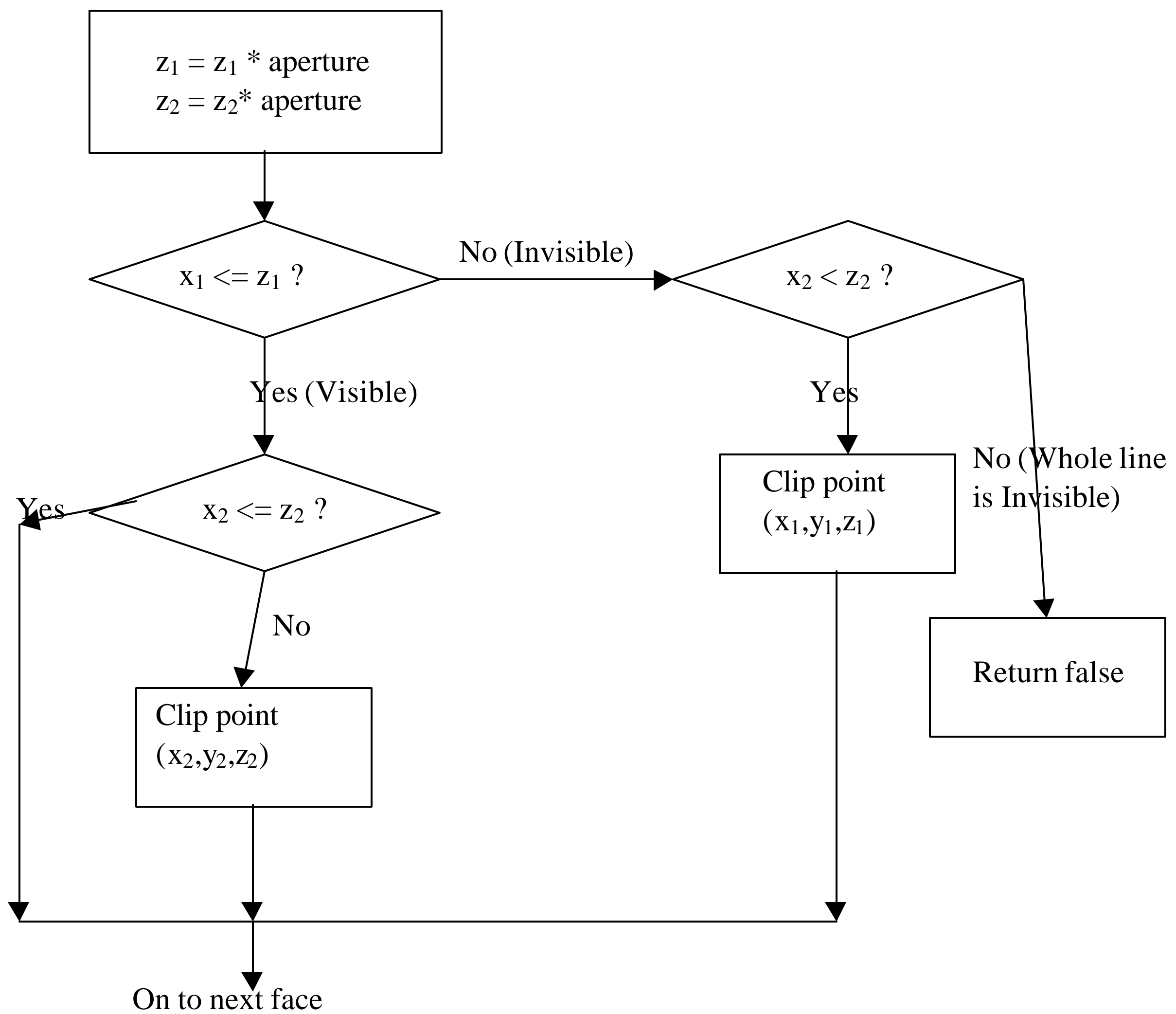

Figure 11: The Line Clipping Algorithm

4. Implementation of the System

This section describes the software implementation of the system. Some sample code is provided, and operations regarding video capture, video manipulation and position tracking are explained. The last sub-section presents a sample scenario in which the system was tested and the results obtained.

4.1 Storing the known information of the scene

As discussed earlier, we need to store the co-ordinates of the objects under consideration. These co-ordinates were stored in a text file called *"Obs.txt"*.

The file contains the following information:

- ***Number of objects in the scene.***
- For each object in the scene, the following information should be present:
  - ***An integer representing the type of object: 1=>Rectangle, 2=>Triangle*** (The system was tested only on rectangular objects, but having an option for the type of the object would allow the system to be expanded to include other geometric shapes.)
  - ***The 3-dimensional co-ordinates (x, y, z) of each of the vertices*** (which are measured using the magnetic tracking device.)
  - ***A description of the object that is to be displayed.***

The co-ordinates are stored in clockwise order. For example, with respect to the rectangle in ***figure 12*** below, the co-ordinates may be stored in any of the following orders: <ABCD>, <BCDA>, <CDAB>, <DABC>. The same holds for triangular objects.

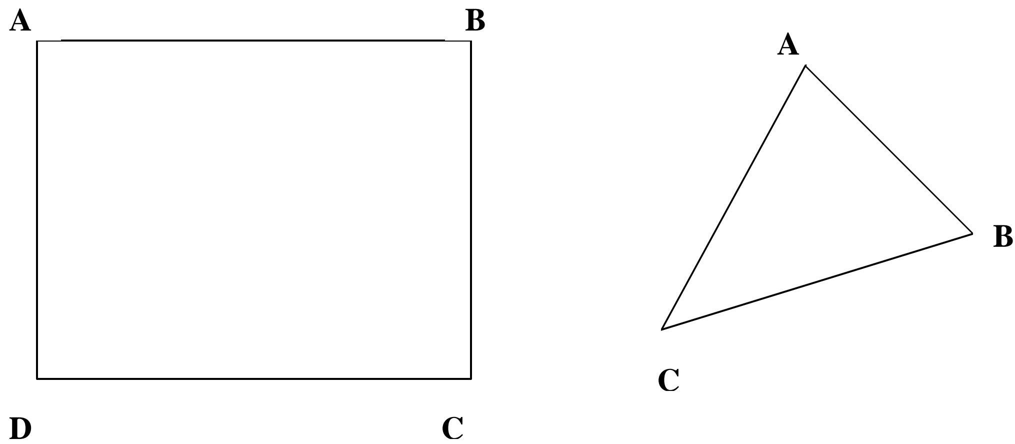

Figure 12: Order in which the co-ordinates of the vertices are stored.

This same idea may be expanded to include cuboid objects, as shown below. The numbers indicate one of the possible ways of the order in which the co-ordinates of the vertices may be stored.

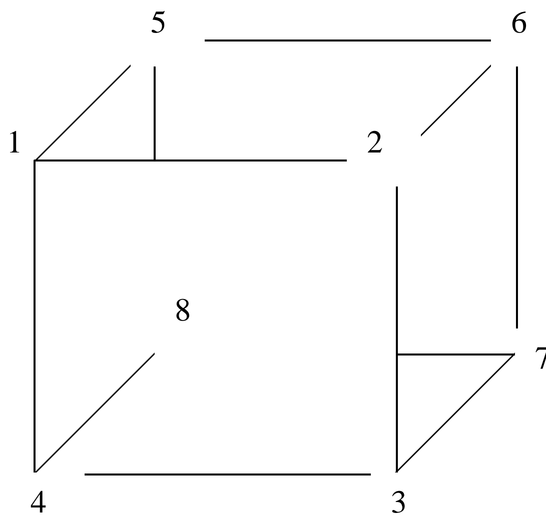

Figure13: Possible representation of a cuboid object

This information is read into an appropriate structure:

```
struct { int type; //type of object
        point3d thePoints[MAX_POINTS]; //object co-ordinates
           char *description; // textual description that needs to be displayed
}theObjects[MAX_OBJECTS];
```

4.2 Implementation of the viewing pipeline

The class "EyeStuff" implements the viewing pipeline: It contains all the functions needed to carry out the required geometrical transformations.

```
class EyeStuff
{ private:
     double EYE[4][4]; //The EYE matrix
      point3d eye_lo ; //Eye location
      point3d vrp;//View Reference Point
     double vcx, vcy;// Co-ordinates of the center of the screen
     double vsx, vsy;
     double apert;// viewing aperture

 public:
     EyeStuff();
      void calc_eye(); // Calculate EYE matrix
      void set_eye_loc(point3d);
      point3d get_eye_loc();
      point3d get_vrp();
     void set_vrp(point3d);
     void set_virtualcenter(double cx, double cy);
     void set_vs(double sx, double sy);
     void perspec(point3d &prev, point3d &curr); //Perspective transformation
     void vu_port(point3d &p ,point3d &curr); //View-port transformation (line)
     void vu_port(point3d &p);
     void perspec(point3d &p); //View-port transformation (point)
     bool clip3d(point3d &p1,point3d &p2); //Clip line segment p1-p2
      point3d trans_EYE(point3d p1); //Eye tranformation

};
```

4.3 Video Capture operations

Microsoft Video for Windows (VFW) provides a number of functions for video capture. The captured frames are stored in **.avi** format; each frame is captured as a bitmap, whose format may be selected by the user.

- ***CapCreateCaptureWindow*** => Creates a new capture window.
- ***capGetDriverDescription*** => Returns the description of a capture driver.
- ***capDriverConnect*** =>Connects a capture window to a capture driver.
- ***capSetCallbackOnFrame*** =>lets the user to set the function to be called whenever a frame is captured.
- ***capSetCallbackOnError*** =>lets the user to set the function to be called in case an error was encountered.
- ***capCaptureSetSetup*** =>Brings up a common dialog to let the user select capture settings.
- ***capPreviewRate*** =>Sets the preview rate, normally set to 15 frames per sec, as the human eye cannot process more than 15 frames per second.
- ***capDlgVideoFormat*** => Brings up a common dialog that lets the user select the format of the captured frame, depending on what formats are provided by the driver.

4.4 Bitmap File formats

A bitmap **(.bmp)** file is a Device Independent Bitmap (DIB). DIB files have a standard header that identifies the format, size, color palette (if applicable) of the bitmapped image. The header is a BITMAPINFO structure.

```
typedef struct tagBITMAPINFO {
   BITMAPINFOHEADER    bmiHeader;
   RGBQUAD             bmiColors[1];
} BITMAPINFO;
```

BITMAPINFOHEADER is a structure of the form:

typedef struct tagBITMAPINFOHEADER{
  DWORD biSize; // bytes required by structure
  LONG biWidth; //width of bitmap (in pixels)
  LONG biHeight; //height of bitmap (in pixels)
  WORD biPlanes; // No of planes for target device (Always =1)
  WORD biBitCount; //No of bytes per pixel
  DWORD biCompression; //type of compression used
  DWORD biSizeImage;
  LONG biXPelsPerMeter;
  LONG biYPelsPerMeter;
  DWORD biClrUsed;
  DWORD biClrImportant;
} BITMAPINFOHEADER;

bmiColors [1] is the first entry in an optional color palette or color table of RGBQUAD data structures. True color (24 bit RGB) images do not need a color table. 4 and 8 bit color images use a color table.

typedef struct tagRGBQUAD {
  BYTE rgbBlue; // Intensity of Blue
  BYTE rgbGreen; // Intensity of Green
  BYTE rgbRed; // Intensity of Red
  BYTE rgbReserved; // Always zero
} RGBQUAD;

When ***biBitCount*** = 32, the bitmap has a maximum of 2^32 colors. If the biCompression member of the BITMAPINFOHEADER is BI_RGB, the bmiColors member is NULL. Each DWORD in the bitmap array represents the relative intensities of blue, green, and red, respectively, for a pixel. The high byte in each DWORD is not used. In our case, we use uncompressed bottom-up bitmaps, and so the ***biCompression*** member is set to **BI_RGB**.

The following function was used to set a specified pixel to a specified color, for a 32-bit uncompressed bitmap. (In the implementation of this project, only 32-bit format at a resolution of 320 x 240 is implemented. This can be extended to include other formats without much difficulty.)

```
void mySetPixel(unsigned char *VideoData,int col, int row, int width,int color)
{        //if point is out of bounds, ignore it
  if(col>320 || row>240|| col<0 || row<0) return;
   if(color==1) //RED
        { VideoData[4*(row*width+col)]=0x00;
          VideoData[4*(row*width+col)+1]=0x00;
          VideoData[4*(row*width+col)+2]=0xFF;
          VideoData[4*(row*width+col)+3]=0x00;
        }
        if (color==2) //WHITE
   { VideoData[4*(row*width+col)]=0xFF;
          VideoData[4*(row*width+col)+1]=0xFF;
          VideoData[4*(row*width+col)+2]=0xFF;
          VideoData[4*(row*width+col)+3]=0x00;
        }
   if (color==3)      // BLUE
   { VideoData[4*(row*width+col)]=0xFF;
          VideoData[4*(row*width+col)+1]=0x00;
          VideoData[4*(row*width+col)+2]=0x00;
          VideoData[4*(row*width+col)+3]=0x00;
        }
}
```

Here, VideoData is a structure containing the RGB values of the pixels of a captured frame. This is accessible through the frame preview function provided by VFW.

4.5 The pciBIRD API

Commands are sent and responses are obtained from the tracking device through the pciBIRD API provided by Ascension Technology Corporation. Following is a list of functions that are of significance:

- ***InitializeBIRDSystem*** – Causes the pciBIRD driver to reset all PCIBIRD boards in the system, obtain and build a database of information containing number of sensors, transmitters etc.
- ***SetSensorParameter*** – Used to set various parameters of the sensor, including the data format type, frequency of measurement,etc. The device is capable of presenting data in the form of position, or angles or both. For this project, the data format was set to ***"position only"***.
- ***GetAsynchronousRecord*** – When invoked reads and returns the last data record from the last computation cycle of the sensor.
- ***GetBIRDError*** – Each of the above mentioned functions returns an integer value representing an error code. If the return value is 0, it means the function was successful. If any other return value is encountered, the ***GetBIRDError*** method can be used to get an integer that returns the oldest error message in the error queue. The ***GetErrorText*** method can then be used to obtain a textual description of the error.

4.6 Putting it all together

4.6.1 Initializing required data and devices:

- The first step is to read the number of objects in the scene, their co-ordinates and description as explained in section **4.1** above.
- Create a capture window, and connect to a capture driver.
- Initialize the pciBIRD system and set the sensor data format to ***"position only"***.
- Set the callback functions: Set the callback in the case of a frame capture to ***FrameProc()***. This means that all frame manipulation will have to be handled by this function.
- Initialize the settings of the video capture device (*i.e.,* encoding, resolution, etc)

4.6.2 Processing each Frame: The *FrameProc()* function

```
LRESULT CALLBACK FrameProc(HWND hcap, LPVIDEOHDR
lpVHdr)
{
        HDC hdc;

        RECT *lpRect=(RECT*)malloc(sizeof(RECT));
        lpRect->left=0;
        lpRect->top=250;
        lpRect->right=320;
        lpRect->bottom=300;
        count++;

      BITMAPINFO bm;
        unsigned char *VideoData;
        VideoData=(unsigned char*)malloc(sizeof(unsigned char));
        VideoData=lpVHdr->lpData;

   int bmsize=capGetVideoFormatSize(hcap);

if(capGetVideoFormat(hcap,&bm,bmsize)==0)
   MessageBox(hcap,"bmsize=0!","Error",MB_OK);
 BITMAPINFOHEADER bmH=bm.bmiHeader;
 double bpp=bmH.biBitCount /8.0; //bytes per pixel;
 int img_size=(int)bmH.biHeight * bmH.biWidth * bpp;
 if(img_size==0 || bmH.biCompression != BI_RGB)
{  MessageBox(hcap,"Image Size =0; RLE or JPEG compression possibly
used","ERROR",MB_OK);
 return (LRESULT) true;
        }

  if(img_size!=lpVHdr->dwBytesUsed)
{ MessageBox(hcap,"Image Size and Buffer Size do not
match","ERROR",MB_OK);
 return (LRESULT) true;
        }

        //bmH.biWidth=320 [-> 320 columns(x) -> 240 rows(y)]
         // char myZeroRGBDword[4];
        int i=0,j=0,k=0,ln=0;
         int err=0;
```

```
double dbuff[3];
point3d sensor1;
double temp;
 switch (bmH.biBitCount)
{ case 32 : // bmiColors=NULL, each DWORD in the bitmap array holds
          // relative intensities of R,G,B for each pixel
          E.set_virtualcenter(bmH.biWidth/2 , bmH.biHeight/2);
          E.set_vs(bmH.biWidth , bmH.biHeight);

err=GetAsynchronousRecord(0,dbuff,sizeof(SHORT_POSITION_ANGL
ES_RECORD));

err=GetAsynchronousRecord(0,dbuff,sizeof(DOUBLE_POSITION_REC
ORD));
 for(i=0;i<3;i++)
 {   temp=dbuff[i];
     switch(i)
     { case 0: sensor1.x=temp;
               break;
       case 1: sensor1.y=temp;
               break;
       case 2: sensor1.z=temp;
               break;
       }
 }
          E.set_eye_loc(sensor1);
          E.calc_eye();

         // Cross hair to help calibrate view-reference point
         Drawline(VideoData,bmH.biWidth*3/8,bmH.biHeight/2,bmH.biWi
         dth*5/8,bmH.biHeight/2,bmH.biWidth,2);
         Drawline(VideoData,bmH.biWidth/2,bmH.biHeight*3/8,bmH.biWi
         dth/2,bmH.biHeight*5/8,bmH.biWidth,2);

         ProcessObjects(hcap,VideoData,bmH.biWidth);
 break;

  default : MessageBox(hcap,"Sorry.This format is not supported at this
time.","YAAAAAAAAAARGH!!!",MB_OK);
               exit(0);
               break;

}
```

```
        InvalidateRect(GetParent(hcap),lpRect,true); // Required in order
to post the WM_PAINT message to modify the video frame
        hdc=GetDC(hcap);

        SetDIBitsToDevice(hdc,0,0,bmH.biWidth,bmH.biHeight,0,bmH.bi
Height,0,bmH.biHeight,VideoData,&bm,DIB_RGB_COLORS);

 ReleaseDC(hcap,hdc);
return (LRESULT) true ;}
```

The above function is invoked by Windows whenever a frame is captured; the frame is not displayed unless this function states that the frame is to be displayed. (This is done using the ***SetDIBitsToDevice()*** function.)

The following steps are executed by the ***FrameProc()*** function:

- ***Obtain information about the type of encoding used***
- ***Note readings from the sensors and set the eye location and the vrp values to these readings.***
- ***Invoke the ProcessObjects() function, which checks if each object is in the scene, and if it is, calculate the user's view of that object.***

The ***ProcessObjects()*** function is as follows:

```
void ProcessObjects(HWND hcap,unsigned char*VideoData,int width)
{ int i=0;
  bool visible;
 for(;i<NUM_OBJECTS;i++)
  {
        visible=false;
        switch(theObjects[i].type)
        { case 0 : break;
            case1:ProcessEdge(hcap,VideoData,TheObjects[i].thePoints[0],
theObjects[i].thePoints[1],width,visible,theObjects[i].description);

ProcessEdge(hcap,VideoData,theObjects[i].thePoints[1],theObjects[i].the
Points[2],width,visible,theObjects[i].description);
```

```
ProcessEdge(hcap,VideoData,theObjects[i].thePoints[2],theObjects[i].the
Points[3],width,visible,theObjects[i].description);

ProcessEdge(hcap,VideoData,theObjects[i].thePoints[3],
theObjects[i].thePoints[0],width,visible,theObjects[i].description);
                    break;
         case2:ProcessEdge(hcap,VideoData,     theObjects[i].thePoints[0],
theObjects[i].thePoints[1],width,visible,theObjects[i].description);

 ProcessEdge(hcap,VideoData,theObjects[i].thePoints[1],
theObjects[i].thePoints[2],width,visible,theObjects[i].description);

ProcessEdge(hcap,VideoData,theObjects[i].thePoints[2],
theObjects[i].thePoints[0],width,visible,theObjects[i].description);
 break;
        }
 }
}
```

The ***ProcessEdge()*** function calculates if each edge of the object is visible and if so, invokes the functions for processing of the viewing pipeline which was described in section **3.** It invokes the ***ProcessEdge()*** function for each edge of the object.

```
void ProcessEdge(HWND hcap,unsigned char *VideoData, point3d p1,
point3d p2,int width, bool &visible, char desc[100])
{  bool now=false;
        p1=E.trans_EYE(p1); // Convert p1 to Eye System
        p2=E.trans_EYE(p2); // Convert p2 to Eye System
        if(E.clip3d(p1,p2)); // Clip edge (p1,p2)
        { E.perspec(p1,p2); // Perspective transformation
          E.vu_port(p1,p2); // View port transformation
          Drawline(VideoData,p1.x,p1.y,p2.x,p2.y,width,1);
          now=true;
        }
   if ( visible==false && now==true)
        { WriteText(p1,p2,desc);
          visible=now;
        }
  }
```

Drawline() is a function that implements a simple line drawing algorithm between two 2-dimensional points p1 and p2, by setting each pixel on the line via the ***mySetPixel()*** function.

4.7 Experimental Results

Though the proposed system was intended to have two sensors, only one was used for test purposes. This sensor was used to return the position of the eye. The system was tested by keeping the camera stationary and noting the sensor measurement that corresponded to the point that was projected on to the center of the screen. This point was set as the view reference point. This was carried out for different positions of the camera, noting and setting the view reference point in each case. One of the sample scenarios consisted of just one object: A poster against a blue background. The information was stored in the text file as follows:

1
1
9.3 20.8 27.2
15.7 19.6 25.8
17.1 20 17.
9.5 19.7 16.7
< A Poster>

This may be interpreted as an environment having one recognizable object, which is a rectangle and is described as ***"A Poster"***. Figures 14 and 15 below depict two screen shots that portray the results obtained for two different positions of the camera. The white cross-hairs were drawn as a matter of convenience to measure the position of the view reference point, as it is a known fact that the view reference point is always projected on to the center of the screen.

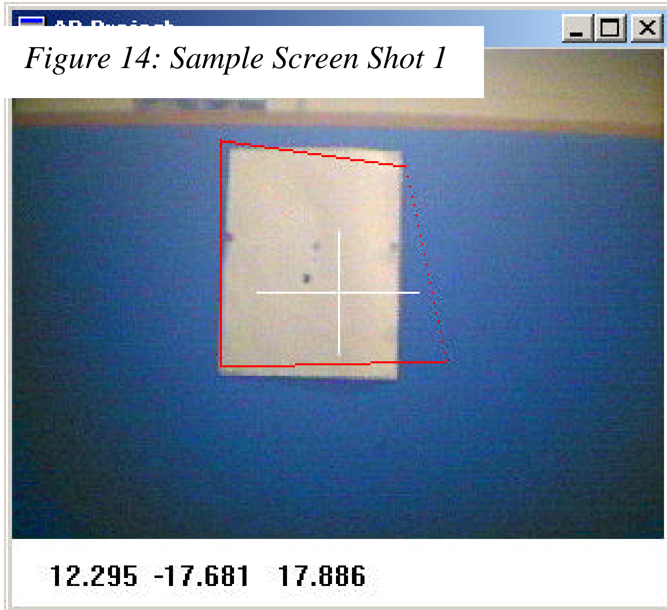

Figure 14: Sample Screen Shot 1

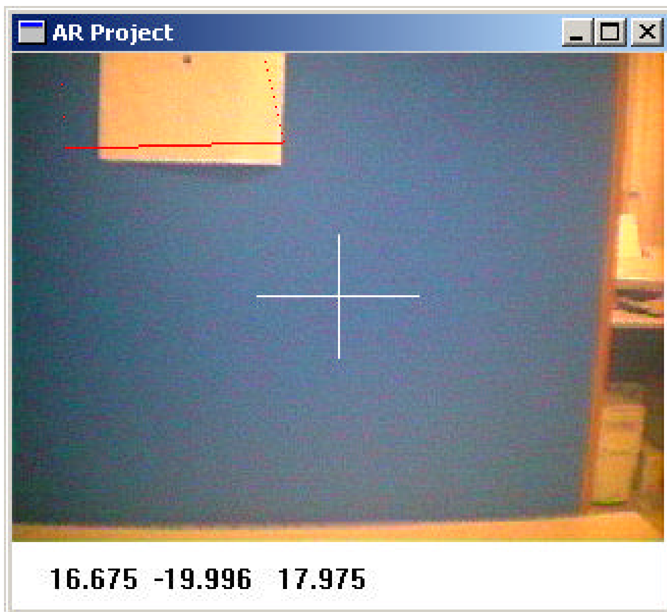

Figure 15: Sample Screen Shot 2

As obvious from the screen shots, the results are erroneous - The wireframes do not match with the boundaries of the images of the object. The following factors may have contributed to such results:

- ***Effect of electrical/magnetic devices in the vicinity:*** Tracking is based on the measurement of a magnetic field generated by the transmitter. This field is affected by the presence of electrical and magnetic devices such as the pc-monitor, wires carrying electricity etc. This produces errors in the measurement of the sensor position. For example, when the sensor was moved along a line parallel to the X-axis, one would expect the Y co-ordinate readings to be the same. However, this was not what happened when the system was tested. The readings for the Y co-ordinates seemed to change. When the transmitter was placed within 1ft. of the pc-monitor, the error observed was much greater: The difference of the Y co-ordinate readings of two points at a physical distance of roughly 6 inches from each other would sometimes be as bad as 10 inches, when we would have liked it to be close to zero.
- ***Tracking device does not accurately convey direction (sign) of co-ordinates:*** The tracking device calculates the magnitude of the magnetic field at the position of the sensor; this field is symmetrical around the X, Y, and Z-axes, and hence there is no guarantee that the device will return accurate signs on the co-ordinates. In most cases, it was observed that the readings of the X co-ordinate values were positive on both sides of the X-axis.
- ***Assumptions in our calculation of the user's view:*** The one assumption that is made in the theory of the viewing pipeline is that the user cannot tilt his head from side to side. Since it is hard to keep the camera from tilting, this is hard to achieve.
- ***Field of view of the camera:*** The vendors of the camera claim that the field of view of the camera is 45^0 (or **p/4** radians). Using this value within the code produced a larger error in generating the wireframes around the objects. Decreasing the angle (up to a certain

point) seemed to improve the results obtained. The actual value of the aperture used in the above mentioned scenario was **p/11** radians.

5. Conclusion and Future Work

This project explored the task of developing a cost-effective system that could be used to recognize simple objects and provide some information regarding those objects to the user. Giving higher priority to cost effectiveness led to the selection of a magnetic tracking device for this project. This turned out to be the cause of a lot of problems. The readings obtained were not accurate and would be distorted by the presence of electro-magnetic devices and metallic objects in the vicinity. This would definitely prove to be impractical in a real-life situation, as we cannot expect a real world environment to be devoid of objects that are sensitive to magnetic or electrical fields. Besides, the range in which the device works is roughly a sphere of 3-feet radius. Tracking devices based on Infra-red are available. These are not affected by electromagnetic signals or sound waves. E.g.: The HiBall-3100, developed at UNC is available commercially nowadays. It has a resolution of better than 0.2mm (angular resolution is better than 0.01 degrees) and has an operating range varying from as low as 144 sq. ft. and is scalable to over 1600 sq. ft.

The display device used was not a see-through device - it was just a very small PC monitor that could be mounted on a pair of eyeglasses. This proved to be a mild distraction whenever used, as some parts of the real world scene would be partly obstructed. See-through displays that are commercially available nowadays seem to be very expensive. We can only hope that they get cheaper in the future.

Augmented Reality is a fast growing field. Though it is not yet ready for the commercial world yet, a lot of research is being done in this field. As the years advance, we may expect commercial tracking and display devices to be available more easily in the future, and we may expect them to perform better and cost

lesser than they do now. We may expect to see many devices based on Augmented Reality used extensively in day-to-day life.

6. References

- ***http://www.se.rit.edu/~jrv/research/ar/*** - Presents an introduction to the field of Augmented Reality and has a number of links to Augmented Reality work available on the web.

- ***http://www.howstuffworks.com/augmented-reality.htm*** - Presents a simple introduction to Augmented Reality.

- ***"Tracking Requirements for Augmented Reality"***, Ronald T. Azuma, *Communications of the ACM, 36*, 7 (July 1993), 50-51.

- ***"A Survey of Augmented Reality"***, Ronald T. Azuma, *Presence: Teleoperators and Virtual Environments 6*, 4 (August 1997), 355 - 385.

- ***"Augmented Reality: A class of displays on the Reality-Virtuality Continuum",*** Paul Milgram, Haruo Takemura, Akira Utsumi, Fumio Kishino.

- ***http://www.augmented-reality.org/*** - Has links to technology, research groups, projects, products, and resources for more information, related to Augmented Reality.

- ***http://www.cs.unc.edu/~tracker/*** - The UNC tracker project.

- ***http://www.3rdtech.com/HiBall.htm*** - Contains information of the commercially available version of UNC's HiBall Tracker.

- ***"Table-Top Spatially-Augmented Reality: Bringing Physical Models to Life with Projected Surgery",*** Ramesh Raskar, Greg Welch, Wei-Chao Chen, University of North Carolina at Chapel Hill, First International Workshop on Augmented Reality, San Francisco CA, Nov 1998.

- ***"Stereoscopic Vision and Augmented Reality",*** David Drascic, *Scientific Computing and Automation*, 9(7), 31-34, June 1993.

- ***http://www.stereo3d.com/hmd.htm*** - Contains a comparison of commercially available Head Mounted Displays.

- ***http://www.mvis.com/prod_mil_hmd.htm*** - MicroVision's Military HMD that uses the ***Retinal Scanning Display*** Technology.

- ***http://www.ascension-tech.com*** – Website of Ascension Technology Corporation, manufacturers of various tracking devices.