

Part I – Programming

Games of yesterday often had graphics that were flat and two-dimensional. This was because the hardware was either too expensive for many people to own or the technology didn't exist that could process the required calculations fast enough. Unlike the times gone by, 3D games are ubiquitous today and 3D applications can even now be found on cellphones (<http://www.ati.com/companyinfo/press/2004/4719.html> and http://www.n-gage.com/en-R1/gamedeck/ngage_qd/techspex/index.htm).

In the last program you developed the game breakout where the all of the elements were two-dimensional. For the final project, you will create a program that displays 3D information. You will create two cubes in 3D space, rotate them based on user input, map their projection to the 2D system of the screen, and finally save a screen shot of the rotated cubes just as the program exits.

Here is a breakdown of what you need to do:

1. Set video mode to VGA 320x200 with 256 colors
2. Project and display the points of the first cube
3. Connect the points with lines drawn on the screen for the first cube
4. Project and display the points the second cube
5. Connect the points with lines drawn on the screen for the second cube
6. Pause for a very short period of time, so the cubes can be seen
7. Remove cubes from screen
8. Take user input
9. Perform required operations based on user input
10. Update the parameters of the cubes (i.e. angles)
 - a. Rotate cube
 - b. Translate cube
11. If the user hasn't pressed the escape key, then go to step 2. If the escape key is pressed, then go to step 12
12. Perform a screen capture and save the screen to a 24 bitmap, grayscale bitmap.
13. Set video mode to text
14. End.

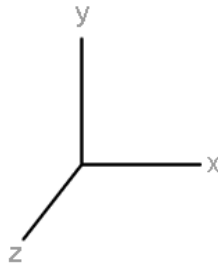
Projection

There are two types of projections systems: orthogonal, and perspective. In the perspective projection, the further away an object is from the camera, the closer it appears the horizontal horizon and its perceived size is smaller. In an orthogonal projection, this is not the case. Regardless of whether an object is close to or far away from the camera, its perceived size remains the same. We will be implementing the perspective projection so that the cubes look realistic.

We are going to use the Cartesian coordinate system, just like the coordinate system of the screen. The horizontal and vertical directions are defined as the x-axis and y-axis and run up and down the

screen. We are going to add a third axis, the z-axis. This axis will increase in value as it moves out of the screen in the direction of the viewer and will decrease in value going into the screen.

We want to project a point in 3D Cartesian space, at say point (x_1, y_1, z_1) to the camera that lies somewhere in space. For all practical purposes, our camera will lie along the z access in the positive direction at some point, $ZOFF$ where coordinates would be $0, 0, ZOFF$. 0 is the x coordinate, 0 is the y coordinate of the camera. For our purposes, $XOFF$ and $YOFF$ are exactly half of the horizontal and vertical sizes of the screen. The reason we will be using those values is so that the camera appears to be sitting in the middle of the screen. When we do the projection of a point, we will offset the x and y values by $XOFF$ and $YOFF$ so that the points also look like they are moved with respect to the center of the screen. So taking this in mind, are camera really sits at coordinates $XOFF, YOFF, ZOFF$ with respect to screen coordinates $0,0$. The purpose of this is so we have our primary objects of interested centered in the screen. Make sense?



In the general case you really would have to do a lot more math and more alignment is needed because there are actually 3 coordinate systems (all Cartesian) that you have to be concerned with: camera, world, and object. The camera will always be oriented with respect to the world system, etc. I thought I would just mention this for those of you who may not see the whole picture yet, but now that I have mentioned it you can see that there is more than meets the eye. For now, put that aside and lets continue.

Enough jibber jabber, let me get straight to the point. If I have a point at (x_{3D}, y_{3D}, z_{3D}) , how do I map those points to 2D screen points (x_{2D}, y_{2D}) ? Simple. Use the following formula:

$$x_{2D} = \frac{FOV * x_{3D}}{|ZOFF - z_{3D}|} + XOFF$$

$$y_{2D} = YOFF - \frac{FOV * y_{3D}}{|ZOFF - z_{3D}|}$$

Where FOV is the field of view. The following constants will be provided to you: FOV, XOFF, YOFF, and ZOFF.

Rotation

You will perform rotation of the object with respect to the object coordinate system. That is you will rotate around the x-axis, y-axis, and z-axis of the object rather than the world. To rotate around the x-axis, your object will tilt it up or down. Rotating around the y-axis will pan the object form side to side. Rotating around the axis will roll the object around. Since you are rotating around the object axis, then the orientation of the of the axis may change with respect to the world coordinate system. You can rotate around the axis in any order, but variables must be updated accordingly before rotating around the remaining axis. The formulas for rotation are as follows:

x-axis rotation matrix representation

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\theta_x) & -\sin(\theta_x) \\ 0 & \sin(\theta_x) & \cos(\theta_x) \end{bmatrix} \times \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

x-axis rotation formulas

$$\begin{aligned} newx &= x \\ newy &= y \times \cos(\theta_x) - z \times \sin(\theta_x) \\ newz &= y \times \sin(\theta_x) + z \times \cos(\theta_x) \end{aligned}$$

z-axis rotation matrix representation

$$\begin{bmatrix} \cos(\theta_z) & -\sin(\theta_z) & 0 \\ \sin(\theta_z) & \cos(\theta_z) & 0 \\ 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

z-axis rotation formulas

$$\begin{aligned} newx &= x \times \cos(\theta_z) - y \times \sin(\theta_z) \\ newy &= x \times \sin(\theta_z) + y \times \cos(\theta_z) \\ newz &= z \end{aligned}$$

Translation (extra credit)

Translation is what happens when you move an object an object left, right, up, down, backwards, or forwards. So to translate an object along the x-axis you simply add the change in at direction to the x value. The same is true for the y-axis and z-axis. The formulas are as follows

$$\begin{aligned} newx &= x + x_translation_change \\ newy &= y + y_translation_change \\ newz &= z + z_translation_change \end{aligned}$$

We will translate the objects with respect to the world coordinate system. This means you will translate the objects after you have rotated them.

Clipping (extra credit)

If part of the cube is off of the screen or parts of it go beyond the screen dimensions, then don't draw that part so as to prevent mirroring on the other side of the screen. If the cubes are behind the camera, at a larger z value, then do not draw them to the screen either.

Line Drawing (extra credit)

Every vertex will have to be by a line so that the cubes look somewhat whole. To make things simpler, you can convert the code below into assembly to do this.

y-axis rotation matrix representation

$$\begin{bmatrix} \cos(\theta_y) & 0 & -\sin(\theta_y) \\ 0 & 1 & 0 \\ \sin(\theta_y) & 0 & \cos(\theta_y) \end{bmatrix} \times \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

y-axis rotation formulas

$$\begin{aligned} newx &= x \times \cos(\theta_y) - z \times \sin(\theta_y) \\ newy &= y \\ newz &= x \times \sin(\theta_y) + z \times \cos(\theta_y) \end{aligned}$$

```

// Extremely Fast Line Algorithm Var C (Addition)
// Copyright 2001-2, By Po-Han Lin
// Freely useable in non-commercial applications as long as credits
// to Po-Han Lin and link to http://www.edepot.com is provided in source
// code and can be seen in compiled executable. Commercial
// applications please inquire about licensing the algorithms.
//
// Lastest version at http://www.edepot.com/phl.html

// THE EXTREMELY FAST LINE ALGORITHM Variation C (Addition)
void myLine(SURFACE* surface, int x, int y, int x2, int y2)
{
    bool yLonger=false;
    int incrementVal, endVal;

    int shortLen=y2-y;
    int longLen=x2-x;
    if (abs(shortLen)>abs(longLen))
    {
        int swap=shortLen;
        shortLen=longLen;
        longLen=swap;
        yLonger=true;
    }

    endVal=longLen;
    if (longLen<0)
    {
        incrementVal=-1;
        longLen=-longLen;
    }
    else incrementVal=1;

    double declnc;
    if (longLen==0) declnc=(double)shortLen;
    else declnc=((double)shortLen/(double)longLen);
    double j=0.0;
    if (yLonger)
    {
        for (int i=0;i!=endVal;i+=incrementVal)
        {
            myPixel(surface,x+(int)j,y+i);
            j+=declnc;
        }
    }
    else {
        for (int i=0;i!=endVal;i+=incrementVal)
        {
            myPixel(surface,x+i,y+(int)j);
            j+=declnc;
        }
    }
}

```

You don't have to use the preceding code to draw lines, it is only provided for guidance. You may use any line drawing algorithm you wish.

User Input

You will take in user input from the keyboard if it is present, modify parameters, change any parameters, and continue. If there is no user input available, you must update the parameters, and continue. This means that the cube will always rotate unless you specify for it stop or change the rotation rates to zero. The commands are as follows:

- A decrease x rotation angle step for cube 1
- S increase x rotation angle step for cube 1
- W increase y rotation angle step for cube 1

Z decrease y rotation angle step for cube 1
E increase z rotation angle step for cube 1
D decrease z rotation angle step for cube 1
Likewise for cube with keys: left arrow, right
arrow, up arrow, down arrow, greater than key, less than key.

Hitting the space key halts all movement for both cubes

Hitting the r key resets the cubes to their original position and resets angle steps to zero

Hitting 7 translate the box to the left, hitting 8 translates the box to the right

Hitting 9 translates the box down, hitting 0 translates the box up

Hitting '-' closes the distance between the camera and the box, hitting '+' increases the distance between the camera and the box.

Space bar stops all movement

As you can tell from the inputs, the rotations of either cube must move independently from each other; however, the translation will move both of them at the same time.

Video Mode

In order to display an image to the screen you will need to change to the appropriate video mode, which is mode 13h. You can do this by using the following two instructions:

```
mov ax, 0013h  
int 10h
```

You will need to restore the mode back to text mode when your program has finished doing the graphics processing which is done with the following two commands:

```
mov ax, 0003h  
int 10h
```

You may use whatever macros and interrupts you feel necessary; however, you may NOT use the video interrupts, pre made macros, etc., for accessing the video screen when displaying your graphics image. You MUST do this by MANUALLY writing to segment 0A000h after forcing video mode 13h as specified above.

Bitmap

The final part of the project is to save the screen to a 24-bit bitmap on program exit. See the specifications for a Windows bitmap for proper formatting and appropriate values. Remember that the 24-bit bitmap does not have a lookup table like that of the 8-bit bitmap. You will have to take each pixel value that you have put on the screen and scale them to fit back into the range 0 ~ 255 and store the result in the red, green, blue values. We will be making a pseudo-grayscale image by just taking the screen color value for a pixel and repeating for the red, green, and blue values in the bitmap file. You will save your grayscale image to a 24 bit bitmap defined by the equal directive "OFILENAME." You will need to include the file "constant.inc" in your program where these constants are defined

Grade distribution:

Part 1

X, Y, and Z rotation	20%
Projection	20%
User Input	10%
Bitmap	20%
Line Drawing (extra)	5%
Clipping (extra)	5%
Translation (extra)	5%
Following Instructions	10%
Comments	
Program Description	5%
Code Comments	5%
Misc./Efficiency	10%

Yes, that adds up to 115... you can get a maximum of 115 out of 100 because of clipping, translation, and line drawing being extra credit. You can get more extra credit, you just need to something extra that is significant and let me know about it, i.e. polygon filling, z-buffering, lighting, sound, etcetera.

In addition to the normal requirements (listed in the syllabus) for handing in the assignment, please include your email address in the comment header of your program. This is so I can get a hold of you if there are complications or problems with your final project.

Clean Coding Company

TITLE: <i>BMP Format</i>
SUBJECT: <i>Windows Bitmap File Format Specifications</i>
AUTHOR: <i>Wim Wouters</i>
VERSION: <i>V1.1</i>

Copyright ©2004, Last printed on Sunday 18 April 2004

Table of Contents

<u>TABLE OF CONTENTS</u>	<u>2</u>
<u>INTRODUCTION</u>	<u>3</u>
<u>BITMAP FILE FORMAT</u>	<u>4</u>
General	4
BMP Contents	4
<u>FIELD DETAILS</u>	<u>6</u>
Height Field	6
Bits Per Pixel Field	6
Compression Field	7
Colors Field	9
Important Colors Field	10

Introduction

This document describes the Microsoft Windows and IBM OS/2 picture bitmap files, called Bitmaps or BMP files. Most of the descriptions of the BMP file concentrate on the Microsoft Windows BMP structures like `BMPINFOHEADER` and `BMPCOREINFO` , but only a few describe the file contents on byte level. This information is therefor only intended to be used in applications where direct reading and writing of a BMP file is required.

Bitmap File Format

The following chapters contain the detailed information on the contents of the BMP file. First more general information will be given regarding the byte order and file alignment. The second chapter will concentrate on the byte-level contents of a BMP file. The third chapter will elaborate on this chapter and explain some of the concepts - like compression - and/or values in detail.

General

The BMP file has been created by Microsoft and IBM and is therefore very strictly bound to the architecture of the main hardware platform that both companies support: the IBM compatible PC. This means that all values stored in the BMP file are in the Intel format, sometimes also called the Little Endian format because of the byte order that an Intel processor uses internally to store values.

The BMP files are the way, Windows stores bit mapped images. The BMP image data is bit packed but every line must end on a dword boundary - if that's not the case, it must be padded with zeroes. BMP files are stored bottom-up, that means that the first scan line is the bottom line.

The BMP format has four incarnations, two under Windows (new and old) and two under OS/2, all are described here.

BMP Contents

The following table contains a description of the contents of the BMP file. For every field, the file offset, the length and the contents will be given. For a more detailed discussion, see the following chapters.

Offset	Field	Size	Contents
0000h	Identifier	2 bytes	The characters identifying the bitmap. The following entries are possible: 'BM' - Windows 3.1x, 95, NT, ... 'BA' - OS/2 Bitmap Array 'CI' - OS/2 Color Icon 'CP' - OS/2 Color Pointer 'IC' - OS/2 Icon 'PT' - OS/2 Pointer
0002h	File Size	1 dword	Complete file size in bytes.
0006h	Reserved	1 dword	Reserved for later use.
000Ah	Bitmap Data Offset	1 dword	Offset from beginning of file to the beginning of the bitmap data.
000Eh	Bitmap Header Size	1 dword	Length of the Bitmap Info Header used to describe the bitmap colors, compression, ... The following sizes are possible:

			28h - Windows 3.1x, 95, NT, ... 0Ch - OS/2 1.x F0h - OS/2 2.x
0012h	Width	1 dword	Horizontal width of bitmap in pixels.
0016h	Height	1 dword	Vertical height of bitmap in pixels.
001Ah	Planes	1 word	Number of planes in this bitmap.
001Ch	Bits Per Pixel	1 word	Bits per pixel used to store palette entry information. This also identifies in an indirect way the number of possible colors. Possible values are: 1 - Monochrome bitmap 4 - 16 color bitmap 8 - 256 color bitmap 16 - 16bit (high color) bitmap 24 - 24bit (true color) bitmap 32 - 32bit (true color) bitmap
001Eh	Compression	1 dword	Compression specifications. The following values are possible: 0 - none (Also identified by BI_RGB) 1 - RLE 8-bit / pixel (Also identified by BI_RLE4) 2 - RLE 4-bit / pixel (Also identified by BI_RLE8) 3 - Bitfields (Also identified by BI_BITFIELDS)
0022h	Bitmap Data Size	1 dword	Size of the bitmap data in bytes. This number must be rounded to the next 4 byte boundary.
0026h	HResolution	1 dword	Horizontal resolution expressed in pixel per meter.
002Ah	VResolution	1 dword	Vertical resolution expressed in pixels per meter.
002Eh	Colors	1 dword	Number of colors used by this bitmap. For a 8-bit / pixel bitmap this will be 100h or 256.
0032h	Important Colors	1 dword	Number of important colors. This number will be equal to the number of colors when every color is important.
0036h	Palette	N * 4 byte	The palette specification. For every entry in the palette four bytes are used to describe the RGB values of the color in the following way: 1 byte for blue component 1 byte for green component 1 byte for red component 1 byte filler which is set to 0 (zero)
0436h	Bitmap Data	x bytes	Depending on the compression specifications, this field contains all the bitmap data bytes which represent indices in the color palette.

Note: The following sizes were used in the specification above:

Size	# bytes	Sign
char	1	signed
word	2	unsigned
dword	4	unsigned

Field Details

Some of the fields require some more information. The following chapters will try to provide this information:

Height Field

The *Height* field identifies the height of the bitmap in pixels. In other words, it describes the number of scan lines of the bitmap. If this field is negative, indicating a top-down DIB, the *Compression* field must be either BI_RGB or BI_BITFIELDS. Top-down DIBs cannot be compressed.

Bits Per Pixel Field

The *Bits Per Pixel* (BBP) field of the bitmap file determines the number of bits that define each pixel and the maximum number of colors in the bitmap.

- **When this field is equal to 1.**

The bitmap is monochrome, and the palette contains two entries. Each bit in the bitmap array represents a pixel. If the bit is clear, the pixel is displayed with the color of the first entry in the palette; if the bit is set, the pixel has the color of the second entry in the table.

- **When this field is equal to 4.**

The bitmap has a maximum of 16 colors, and the palette contains up to 16 entries. Each pixel in the bitmap is represented by a 4-bit index into the palette. For example, if the first byte in the bitmap is 1Fh, the byte represents two pixels. The first pixel contains the color in the second palette entry, and the second pixel contains the color in the sixteenth palette entry.

- **When this field is equal to 8.**

The bitmap has a maximum of 256 colors, and the palette contains up to 256 entries. In this case, each byte in the array represents a single pixel.

- **When this field is equal to 16.**

The bitmap has a maximum of 2^{16} colors. If the *Compression* field of the bitmap file is set to BI_RGB, the *Palette* field does not contain any entries. Each word in the bitmap array represents a single pixel. The relative intensities of red, green, and blue are represented with 5 bits for each color component. The value for blue is in the least significant 5 bits, followed by 5 bits each for green and red, respectively. The most significant bit is not used.

If the *Compression* field of the bitmap file is set to `BI_BITFIELDS`, the *Palette* field contains three dword color masks that specify the red, green, and blue components, respectively, of each pixel. Each word in the bitmap array represents a single pixel.

Windows NT specific: When the *Compression* field is set to `BI_BITFIELDS`, bits set in each dword mask must be contiguous and should not overlap the bits of another mask. All the bits in the pixel do not have to be used.

Windows 95 specific: When the *Compression* field is set to `BI_BITFIELDS`, Windows 95 supports only the following 16bpp color masks: A 5-5-5 16-bit image, where the blue mask is `0x001F`, the green mask is `0x03E0`, and the red mask is `0x7C00`; and a 5-6-5 16-bit image, where the blue mask is `0x001F`, the green mask is `0x07E0`, and the red mask is `0xF800`.

- **When this field is equal to 24.**

The bitmap has a maximum of 2^{24} colors, and the *Palette* field does not contain any entries. Each 3-byte triplet in the bitmap array represents the relative intensities of blue, green, and red, respectively, for a pixel.

- **When this field is equal to 32.**

The bitmap has a maximum of 2^{32} colors. If the *Compression* field of the bitmap is set to `BI_RGB`, the *Palette* field does not contain any entries. 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.

If the *Compression* field of the bitmap is set to `BI_BITFIELDS`, the *Palette* field contains three dword color masks that specify the red, green, and blue components, respectively, of each pixel. Each dword in the bitmap array represents a single pixel.

Windows NT specific: When the *Compression* field is set to `BI_BITFIELDS`, bits set in each dword mask must be contiguous and should not overlap the bits of another mask. All the bits in the pixel do not have to be used.

Windows 95 specific: When the *Compression* field is set to `BI_BITFIELDS`, Windows 95 supports only the following 32bpp color mask: The blue mask is `0x000000FF`, the green mask is `0x0000FF00`, and the red mask is `0x00FF0000`.

Compression Field

The *Compression* field specifies the way the bitmap data is stored in the file. This information together with the *Bits Per Pixel (BPP)* field identifies the compression algorithm to follow.

The following values are possible in this field:

Value	Meaning
<code>BI_RGB</code>	An uncompressed format.
<code>BI_RLE4</code>	An RLE format for bitmaps with 4 bits per pixel. The compression format is a

	two-byte format consisting of a count byte followed by two word-length color indices. For more information, see the following Remarks section.
BI_RLE8	A run-length encoded (RLE) format for bitmaps with 8 bits per pixel. The compression format is a two-byte format consisting of a count byte followed by a byte containing a color index. For more information, see the following Remarks section.
BI_BITFIELDS	Specifies that the bitmap is not compressed and that the color table consists of three double word color masks that specify the red, green, and blue components, respectively, of each pixel. This is valid when used with 16- and 32- bits-per-pixel bitmaps.

When the Compression field is BI_RLE8, the bitmap is compressed by using a run-length encoding (RLE) format for an 8-bit bitmap. This format can be compressed in encoded or absolute modes. Both modes can occur anywhere in the same bitmap.

- **Encoded mode consists of two bytes:**

The first byte specifies the number of consecutive pixels to be drawn using the color index contained in the second byte. In addition, the first byte of the pair can be set to zero to indicate an escape that denotes an end of line, end of bitmap, or delta. The interpretation of the escape depends on the value of the second byte of the pair, which can be one of the following:

- 0 End of line.
- 1 End of bitmap.
- 2 Delta. The two bytes following the escape contain unsigned values indicating the horizontal and vertical offsets of the next pixel from the current position.

- **Absolute mode.**

The first byte is zero and the second byte is a value in the range 03H through FFH. The second byte represents the number of bytes that follow, each of which contains the color index of a single pixel. When the second byte is 2 or less, the escape has the same meaning as in encoded mode. In absolute mode, each run must be aligned on a word boundary.

The following example shows the hexadecimal values of an 8-bit compressed bitmap.

```
03 04 05 06 00 03 45 56 67 00 02 78 00 02 05 01 02 78 00 00 09 1E 00 01
```

This bitmap would expand as follows (two-digit values represent a color index for a single pixel):

```
04 04 04
06 06 06 06 06
45 56 67
78 78
move current position 5 right and 1 down
78 78
end of line
1E 1E 1E 1E 1E 1E 1E 1E 1E
end of RLE bitmap
```

When the Compression field is BI_RLE4, the bitmap is compressed by using a run-length encoding format for a 4-bit bitmap, which also uses encoded and absolute modes:

- **In encoded mode.**

The first byte of the pair contains the number of pixels to be drawn using the color indices in the second byte. The second byte contains two color indices, one in its high-order four bits and one in its low-order four bits. The first of the pixels is drawn using the color specified by the high-order four bits, the second is drawn using the color in the low-order four bits, the third is drawn using the color in the high-order four bits, and so on, until all the pixels specified by the first byte have been drawn.

- **In absolute mode.**

The first byte is zero, the second byte contains the number of color indices that follow, and subsequent bytes contain color indices in their high- and low-order four bits, one color index for each pixel. In absolute mode, each run must be aligned on a word boundary. The end-of-line, end-of-bitmap, and delta escapes described for BI_RLE8 also apply to BI_RLE4 compression.

The following example shows the hexadecimal values of a 4-bit compressed bitmap.

```
03 04 05 06 00 06 45 56 67 00 04 78 00 02 05 01 04 78 00 00 09 1E 00 01
```

This bitmap would expand as follows (single-digit values represent a color index for a single pixel):

```
0 4 0
0 6 0 6 0
4 5 5 6 6 7
7 8 7 8
move current position 5 right and 1 down
7 8 7 8
end of line
1 E 1 E 1 E 1 E 1
end of RLE bitmap
```

Colors Field

The *Colors* field specifies the number of color indices in the color table that are actually used by the bitmap. If this value is zero, the bitmap uses the maximum number of colors corresponding to the value of the *BBP* field for the compression mode specified by the *Compression* field.

If the *Colors* field is nonzero and the *BBP* field less than 16, the *Colors* field specifies the actual number of colors the graphics engine or device driver accesses.

If the *BBP* field is 16 or greater, then *Colors* field specifies the size of the color table used to optimize performance of Windows color palettes.

If *BBP* equals 16 or 32, the optimal color palette starts immediately following the three double word masks.

If the bitmap is a packed bitmap (a bitmap in which the bitmap array immediately follows the bitmap header and which is referenced by a single pointer), the *Colors* field must be either 0 or the actual size of the color table.

Important Colors Field

The *Important Colors* field specifies the number of color indices that are considered important for displaying the bitmap. If this value is zero, all colors are important.