The Nintendo Reverse Engineering Project

Day 2

Things covered on day 2:

- What is a register?
- REG_DISPCNT and the setting of screen modes
- Plotting pixels in mode 3,4,5
- Input and the keypad
- Displaying a picture and using the back buffer
- Lines and other such raster creations
- Rotation and Matrices

What is a register?

Before going further a few concepts need to be ironed out. The first concept (and the only one that really matters at the moment) is the concept of memory mapped registers.

Now, I am sure you are aware that the GBA has several different chips inside responsible for creating the images and sounds that accompany most games. There is a sound processor responsible for producing annoying chip tunes, the video processor which puts all your convoluted data together in a nice and pretty display, the memory chips that hold the data for our programs, and the processor which is in overall control of the whole shebang (in all honestly many of these "chips" are actually just parts of one large integrated circuit and not really separate chips).

When you are writing c code to describe the events in your game you are directly controlling the GBA processor. But, the GBA does not work alone and generally we like to have some control over what the rest of the system is doing. The method by which this is accomplished is the use of hardware registers.

Beginning at the memory address of 0x4000000 and running for quite some many bytes is the memory mapped register space. What this means is that if I write some arbitrary value to the address 0x4000000 then I will be writing to a register that will have some effect on how the system renders (or fails to render) my video game. Understanding registers is key to console development.

This brings me to the next concept that you should already know (if you know any c at all) and that is how to write to memory address 0x4000000. Now, hopefully you remember the concept of a **pointer** but, if not, that's okay because I will cover it briefly. Recall that a pointer is a type of variable that holds not data but, instead, the *address* of some type of data. In this example, let us say we know (which we do) that the register at 0x4000000 was 16 bits in length and controlled the display and we hence call it REG_DISPCNT; we might use code like the following to write to this address:

```
unsigned short *REG_DISPCNT = (unsigned short *) 0x4000000;
*REG_DISPCNT = somevalue;
```

Now, this would work just fine but there are some issues with this code. First there is the fact that since these are hardware registers we may not be the only ones changing them. This is something the compiler needs to know else it will try to optimize our code and likely break it. The way we tell the compiler that variables change outside of the c code is to declare them volatile. The last issue is that we are using a variable (RAM) to store a constant. We would be much better served if we just used a #define…this also allows us to dereference the register in its declaration and makes writing to it a bit simpler. Here is the new, more proper code.

```
#define REG_DISPCNT (*(volatile unsigned short*)0x4000000)
REG_DISPCNT = somevalue;
```

Notice there is no longer any need to dereference the register prior to use as it is implicit in the definition. Also, this code uses no space in memory for the pointer as it is just a constant. I hope this concept is clear to you; about 30% of the following pages are nothing but descriptions and examples of how to use the hardware registers to control the many features of the GBA. If you look in gba_registers.h you will find all of the known registers defined similar to the manner described above.

REG_DISPCNT and setting the screen mode

And finally we are off to explore the GBA hardware. REG_DISPCNT is a 16-bit register that resides at the memory address 0x400:0000. It is responsible for the basic control of the display. Things such as turning on and off background layers, controlling sprite memory layout, setting the screen mode, and a few other items are in its domain. How can we get so much out of a single register? Well it turns out that each bit in a register has a specific purpose so that, for example, setting bit 8 of REG_DISPCNT would enable background 0. If setting a bit is an unclear concept please look at my number system primer (which I am sure I have not finished yet and therefore you will not actually be able to find and instead be forced to google about for it). Below is a table delineating the relationship between the 16 bits of the register and their corresponding control functions.

F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
OW	W1	W0	OBJ	BG3	BG2	BG1	BG0	B	OM	HB	DB	GB	MODE		

REG_DISPCNT @ 0x400:0000

Bits 0-2 (MODE): These bits control the screen mode. The screen mode can be 0, 1, 2, 3, 4, or 5.
Bit 3(GB): This is set by hardware if a GB or GBC cartridge is detected.
Bit 4(DB): This bit controls which buffer is the active buffer when in mode 4 and 5. (double buffering support)
Bit 5(HB): This bit, when set, allows OAM to be updated during the horizontal blank.

We will talk more about this when it is time to figure out sprites.
Bit 6(OM): This flag determines what mapping mode is used for sprite graphics: 0 = 2D 1 = 1D. We will talk more when we talk about sprites
Bit 7(B): This bit, when set, will put the screen in a forced blank causing the screen to go white.
Bits 8-C(BGx,OBJ): These bits, when set, enable the associated backgrounds and OBJ (sprites)
Bits D-F(WO,W1,W0): These bits enable the window displays. Windows will be covered later.

To make things simpler we will place all this new information into a header file so that it easily accessible. The definitions for display control register bits can be found in gba_video.h and are shown below.

```
#include "gba_types.h"
///// REG_DISPCNT defines

#define MODE_0       0
#define MODE_1       1
#define MODE_2       2
#define MODE_3       3
#define MODE_4       4
#define MODE_5       5

#define BACKBUFFER      BIT(0x4)
#define H_BLANK_OAM     BIT(0x5)

#define OBJ_MAP_2D      0
#define OBJ_MAP_1D      BIT(0x6)

#define FORCE_BLANK     BIT(0x7)

#define BG0_ENABLE      BIT(0x8)
#define BG1_ENABLE      BIT(0x9)
#define BG2_ENABLE      BIT(0xA)
#define BG3_ENABLE      BIT(0xB)
#define OBJ_ENABLE      BIT(0xC)

#define WIN1_ENABLE     BIT(0xD)
#define WIN2_ENABLE     BIT(0xE)
#define WINOBJ_ENABLE   BIT(0xF)

///////SetMode Macro
#define SetMode(mode) REG_DISPCNT = (mode)
```

Inside "gba_types.h" is a macro called BIT(n) that is defined as follows:

```
#define BIT(n) (1<<(n))
```

This macro simply takes a value and returns a number with the appropriate bit set using the left shift operator "<<".

Also included with the REG_DISPCNT definitions is a macro that allows you to more readably set the screen mode. To use this macro all you do is the following: Say you want mode 4, 1D OBJ mapping, and background 2 enabled.

```
SetMode(MODE_4 | OBJ_MAP_1D | BG2_ENABLE); //and that's it
```

You could also just as easily say

```
REG_DISPCNT = MODE_4 | OBJ_MAP_1D | BG2_ENABLE;
```

but the set mode option is more readable in my opinion.

Plotting pixels in modes 3, 4, and 5

The GBA video hardware is, for the most part, very straight forward. In bitmap modes you have a linear map that controls every pixel on the screen. The screen is 240 pixels wide and 160 pixels high. First we will set up mode 3 which if you remember is a linear buffer of 16-bit pixels. Let us revisit our sample code from day one and see if we can understand now, how it works.

```
#include <gba.h> //everything you need for gba devving

////////////////// C code entry (main())//////////////////////
int main()
{
   unsigned char x,y;

   SetMode(MODE_3 | BG2_ENABLE);

   for(x = 0; x < SCREEN_WIDTH; x++)
       for(y = 0; y < SCREEN_HEIGHT; y++)
           VideoBuffer [x + y * SCREEN_WIDTH] = RGB16(31,0,0);

   while(1){}
 }//end main
```

The first line simply includes all the headers that are created during this tutorial series as well as some basic defines such as SCREEN_WIDTH and VideoBuffer.

VideoBuffer is defined as a pointer to the beginning of video ram (0x600:0000 if you will recall). Being a pointer we can use it as an array and since we have no silly OS telling us we are accessing memory out of range, or other such nonsense, using it is quite straightforward. Accessing each pixel is just a matter of storing that pixel's color into the array. The GBA stores these colors in BGR format meaning the low 5 bits are red the next 5 are green and the next 5 blue, the high bit is unused. 5 bits means 32 levels of each color for a total of 32*32*32 = 32,768 colors.

Bits	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
	/	B	B	B	B	B	G	G	G	G	G	R	R	R	R	R

A bit representation of a BGR triplet.

The RGB16 macro allows you to enter the red, green, and blue components separately and is defined in gba.h as follows.

```
#define RGB16(r,g,b)  (((b)<<10)+((g)<<5)+(r))
```

The first pixel resides at 0x600:0000 and the next at 0x600:0000 + 2 (or 0x600:0002) and so on until you get to the end of a scan line which is (239) * 2 away (or 0x600:01DD). The next line starts at 0x600:01F0.

To get the actual location of the pixel in terms of an x and y, you need to move down an entire line (240*2bytesPerPixel) for each y and over the correct number of x. Therefore, if you define an array, VideoBuffer, and make it equal to the start of video memory (u16* VideoMemory = (u16*) 0x600:0000) then all that is left is to multiply the y value by screen width and add the x value and you have the offset into the array. VideoBuffer[y * 240 + x]. If you store a color in that location you then have a pixel on the screen.

There is not much more to be said about Mode 3. We will come back to it later when we discuss raster graphics.

Mode 5 is operated in a very similar manner to Mode 3. The only changes are as follows:

```
SetMode(MODE_3 | BG2_ENABLE);
```

to

```
SetMode(MODE_5 | BG2_ENABLE); //set mode 5
```

and

```
VideoBuffer [x + y * SCREEN_WIDTH] = RGB16(31,0,0);
```

to

```
VideoBuffer [x + y * 160] = RGB16(31,0,0);
```

Although you have a smaller area to work with you gain the advantage of having enough room to place two screens in memory at once. This allows you to draw to one screen while displaying the other, a process referred to as "Double Buffering". Double buffering allows for a smooth update of the screen. You always draw to the off-screen buffer so your updates are not visible until you tell the video hardware to switch.

Mode 4 is a slightly uglier beast to deal with. Unfortunately there is a severe limitation imposed on mode 4. Even though mode 4 is 8 bits per pixel, video memory can only be accessed 16 bits at a time. This means that in order to write a single pixel in mode 4 you must read the two pixels that would be affected into ram, mask out the pixel you want to change then write the two pixels back into memory. The following is pseudocode that does just that:

```
unsigned short temp;
unsigned char pixel = color;

temp = VideoBuffer[(x + y * 240) / 2];

if(x & 1)
   temp = (temp & 0xFF00) + pixel;
else
   temp = (temp & 0xFF) + (pixel << 8);

VideoBuffer[(x + y * 240) / 2] = temp;
```

As you can probably guess, this causes a bottleneck, crippling mode 4's effectiveness. Because of this, most mode 4 engines just write 2 pixels at a time to video memory. Being forced to calculate two pixels at a time is not so bad, but at times, it is not possible; such as line or polygon drawing where the edges are only 1 pixel thick.

Besides the change to SetMode (which I will let you figure out on your own) there is also the added step of loading a palette into to palette memory so the video processor can decide what colors go with what indices. To demonstrate this we are going to do a simple program that loads and displays a PCX file.

Displaying a picture

We have nearly all the information we need to display a PCX picture. We are only lacking two things. First is the format of a PCX file. Second is how to get the PCX file into our code without an fopen() command. It turns out that the first is very simple and the second has many ways in which to go about it.

Let us start with getting the picture into our code then we will tackle the issue of decoding it. The first option, and arguably the simplest is to write (or borrow) a tool that converts the PCX file into a big array of color indexes and a palette and outputs them to a c file so we can just compile it and use it. This method also has the advantage of removing the step of decoding the PCX file on the GBA. Another method is to include the raw PCX file in our code by converting it into an object file or by using the .incbin command of the assembler. We will do both of the second two methods and decode the PCX file on the GBA. Also I will include a tool that runs on the PC and performs the first option. Creating PC tools may be covered later when we actually begin making a game.

To create an object file that can be linked directly with our code we can use the included objcopy tool that comes with gcc or we can use a third party tool that greatly simplifies this process…let's choose the latter. The following line, if added to the batch file, will create a linkable .o file from your PCX file. (you will need the bin2o tool that can be downloaded from darkfaders site. I recommend you place it in your devkitarm/bin dir).

Bin2o input.pcx output.o !input_data

Then just add output.o to the build.bat list of .o files. Bin2o takes an input file of any type and creates an output object file and allows you to reference it through the supplied identifier like this:

```
extern unsigned short input_data[];

main()
{
   int i;

   for(i = 0; i < 240*160; i++)
      VideoBuffer[i] = input_data[i];
}
```

The other simple way, is to use the assembler and its built-in .incbin statement. To do this we simply create a .s file (plain text) and place in it the following:

```
       .global picture
       .text
       .align 4

picture:
       .incbin "data/test.pcx"
```

You then use the assembler to assemble this code into a .o file.

"**.global picture**" simply ensures that the "picture" reference will be available to other files.

"**.align 4**" ensures that the following data will be on a 32-bit boundary which can be important for things such as Direct Memory Access.

"**.text**" simply causes the assembler to place the following data into the binary as apposed to allocating ram for it.

Finally we come to the label **"picture:"** and the **.incbin "your file"** statements. ".incbin" expands the data from the file in place so that to access the data you just look at the location tagged by picture. To add more files you simply add another global statement for each file, another align, and a new label. To access this data you use exactly the same method as above.

```
;This is how you would do it with 2 files that needed including
       .global picture
       .global picture2
       .text
       .align 4

picture:
```

```
        .incbin "data/test.pcx"

        .align 4

picture2:
        .incbin "data/test2.pcx"
```

Now that we know how to include data in our code let us take a close look at the PCX file structure and see if we can get an image onto the screen.

The PCX file structure consists of a header followed by run length encoded data followed by a 256 color palette of 24-bit RGB values. To decode is actually very simple. First we read in the header then use that to determine the size of the PCX and to ensure that it is a 256 color image and not true color. We then decode the run length encoded data and finally load the palette. Let's take a look at that PCX header.

```
#ifndef PCX_H
#define PCX_H

typedef struct
{
    char         manufacturer;   //should be 0
    char         version;        //should be 5
    char         encoding;       //should be 1
    char         bitsPerPixel; //should be 8 for 256 color images
    short int    xmin,ymin;      //coordinates for top left,bottom
right
    short int    xmax,ymax;
    short int    hres;           //resolution
    short int    vres;
    char         palette16[48];  //16 color palette if 16 color image
    char         reserved;       //ignore
    char         colorPlanes;   //ignore
    short int    bytesPerLine;
    short int    paletteType;   //should be 2
    char         filler[58];     //ignore
}PCXHeader, *pPCXHeader;
#endif
```

Now, to load the header is quite simple. Assuming we still have our data included as in the above example we don't even need to copy anything. All we need is to create a pointer to the begging of that data that is in the form of a PCX header.

```
PCX_header* header = (PCXHeader*)picture;
```

Access the header like any other struct pointer:

```
if(header->bitsPerPixel != 8)
{
   //something is wrong...exit gracefully
}
```

Run length encoding is one of the simplest forms of compression available. The results are good only on images with a lot of similar runs of colors. The way it works is as follows:

Say you have an image composed of red, green, orange, and blue that looked something like this:

R R R G G G O O O B B B R R G G G G G B B B

where each letter represented a color in the picture. The run length encoded version would look something like this:

3R 3G 3O 3B 2R 5G 3B

where each number represented the amount of the following color. As you may have noticed, if the run is only 2 pixels long then there is no compression at all and even worse, if it is only 1 pixel then it takes twice as much space for the same data … not so good. PCX, fortunately, does a bit better than this.

The PCX format is as follows: If the number you run into is greater then 192 then it represents a run of number minus 192. This means that most single runs will be represented as a single entry as long as they have a color index of less than 192. For single runs of color greater than 192 we still have the same issue of storing the run as 2 bytes.

All we need to do to decode the data portion of the PCX file is loop through the data and store the pixels as we go, ensuring to repeat the colors appropriately. First let us build a LoadPCX function that will take a pointer to the PCX data and provide a place for us to store the decompressed data and palette. Before that we need two additional structures: one to hold a RGB triplet from our PCX file and one to hold our image data.

```
#ifndef GBA_IMAGE_H
#define GBA_IMAGE_H

#include <pcx.h>

//holds a rgb triplet
typedef struct
{
   unsigned char r,g,b;
}__attribute__ ((packed)) RGB_24;

//holds a basic image type for loading image files
typedef struct
{
   short height,width;
   int bpp;
   unsigned short* palette;

   union
```

```
    {
       unsigned char* data8;
       unsigned short* data16;
       unsigned int* data32;
    };

} sImage, *psImage;

#endif
```

The RGB structure is self-explanatory with the exception of all that "__attribute__" crap. GCC has the annoying habit of trying to make its code faster and one of its tools to that end is to place structures on a 32-bit boundary by padding them to an even size. The "packed" attribute tells the compiler not to do this and allows us to overlay an array of our RGB triplets on top of the PCX data and ensures that they will line up.

The image structure is very basic. It stores height, width and bits per pixel of the image and also has a pointer to hold the palette and the pixel data. The pixel data pointer is unioned so we can use one pointer to reference different size chunks of data. In other words I can write to it in bytes and read it out in shorts or ints which will prove useful later on.

Here is tutorial 2 source code for demo 1 of day 2:

```
#include <gba.h> //everything you need for gba devving
#include <gba_image.h>//pcx support is not included by default in gba.h

#include <stdlib.h>

extern unsigned char picture[]; //imported using the assembler
extern unsigned char picture2[];//this one with bin2o

int LoadPCX(unsigned char* pcx, sImage* image)
{
   unsigned char c;
   int size;
   int count;
   int run;
   int i;
```

```
    RGB_24* pal; //struct rgb {unsigned char b,g,r;};

    PCXHeader* hdr = (PCXHeader*) pcx;

    pcx += sizeof(PCXHeader); //move past the header

    image->width  = hdr->xmax - hdr->xmin + 1 ;
    image->height = hdr->ymax - hdr->ymin + 1;

    size = image->width *image->height;

    if(hdr->bitsPerPixel != 8)
       return 0;

    image->data8 = (unsigned char*)malloc(size);
    image->palette = (unsigned short*)malloc(256 * 2));

    count = 0;

    while(count < size)
    {
       c = *pcx++;

       if(c < 192)
          image->data8[count++] = c;

       else
       {
          run = c - 192;

          c = *pcx++;

          for(i = 0; i < run; i++)
             image->data8[count++] = c;
       }
    }

    pal = (RGB_24*)(pcx + 1);

    for(i = 0; i < 256; i++)
       image->palette[i] = RGB16(pal[i].r >> 3 ,pal[i].g >> 3 , pal[i].b >>
3);

    return 1;
 }

 ////////////////// C code entry (main())//////////////////////
 int main()
 {
    int i;
    sImage image1,image2;

    if(LoadPCX(picture, &image1) && LoadPCX(picture2, &image2))
    {
       SetMode(MODE_4 | BG2_ENABLE);

       for(i = 0; i < SCREEN_WIDTH * SCREEN_HEIGHT / 2; i++)
```

```
         FrontBuffer[i] = image1.data16[i];

      for(i = 0; i < SCREEN_WIDTH * SCREEN_HEIGHT / 2; i++)
         BackBuffer[i] = image2.data16[i];

      for(i = 0; i < 256; i++)
         BGPaletteMem[i] = image1.palette[i];

      while(1)
      {
         if(!(REG_KEYS & KEY_A))
         {
            REG_DISPCNT ^= BACKBUFFER;
         }
      }
   }
   return 0;
}//end main
```

The first lines of this code include the appropriate headers, including the one we just built: "GBA_images.h". Next we import our data with the following two lines:

```
extern unsigned char picture[]; //imported using the assembler
extern unsigned char picture2[];//this one with bin2o
```

The fist one is created in include.s.

```
.global picture
   .text
   .align 4

picture:
   .incbin "data/test.pcx"
```

The second by adding the following line to the build.bat file:

```
bin2o data\test2.pcx test.o !picture2
```

Both of these PCX files were created in Paint Shop Pro from JASC software and saved as 256 color images. It is possible to save your PCX file as 16-color and as true-color, both of which will break the LoadPCX function. Before saving ensure you have reduced your images color palette to 256 colors.

We then move on to the LoadPCX function which takes a pointer to the PCX data and an image pointer as arguments. The first few lines instantiate some variable that will be needed further on.

```
//struct rgb {unsigned char b,g,r;};
   RGB_24* pal;

   PCXHeader* hdr = (PCXHeader*) pcx;

   pcx += sizeof(PCXHeader); //move past the header
```

These lines set up our palette pointer then point a PCXHeader structure to the beginning of the PCX data. Then the PCX pointer is advanced past the header so we can begin work on decoding.

```
   image->width  = hdr->xmax - hdr->xmin + 1 ;
   image->height = hdr->ymax - hdr->ymin + 1;

   size = image->width *image->height;

   if(hdr->bitsPerPixel != 8)
      return 0;
```

These lines load in the height and width of the image from the header and then calculate the size of the data. Also we verify that the image is a 256-color image because that is all our loader can handle.

```
   image->data8 = (unsigned char*)malloc(size);
   image->palette = (unsigned short*)malloc(256 * 2));
```

Next the memory is allocated to hold the data and the palette. And then the decoding process begins.

```
   while(count < size)
   {
      c = *pcx++;

      if(c < 192)
         image->data8[count++] = c;

      else
      {
         run = c - 192;

         c = *pcx++;

         for(i = 0; i < run; i++)
            image->data8[count++] = c;
      }
   }
```

This loop does the bulk of LoadPCX's work. We grab the first data byte (c = *pcx++) and determine if it is a run by checking its size. If it is not a run (size < 192) then it is just a color index and its value gets stored in the image data. If it is a run the run length is calculated by subtracting 192. The next byte is the color of the run so we grab it and the for loop places that color into image data the appropriate number of times. We keep track of the amount of decoded data by the count variable which is incremented every time we write to image data. When count reaches our precalculated size then we know we have decoded all the data.

Finally, we point our RGB_24 palette to the PCX palette and convert it to the 16 bit palette needed by the GBA.

```
   pal = (RGB_24*)(pcx + 1);

   for(i = 0; i < 256; i++)
      image->palette[i] = RGB16(pal[i].r >> 3 ,pal[i].g >> 3 , pal[i].b
>> 3);
```

The only tricky part is the actual conversion. Since the PCX palette components are 8 bit they can be a maximum of 255 whereas the GBA can only use up to 31. TO compensate we divide each component by 8 (365 / 8 = 32) which is the same as right shifting by 3. We use our RGB16 macro from before to convert the separate components into a 16-bit color.

Input

Before moving on to main() there is a topic that needs to be discussed, and that is input. Getting input from the keypad is extremely simple. There is a single register that stores the state of the key presses. It is defined in "gba_registers.h" as REG_KEYS and resides at 0x40000130. When a key is pressed the corresponding bit in the register is cleared. To check for a key press you just AND the register with the appropriate bit and if it is set then the key is NOT pressed. Here is "gba_keys.h":

```
#ifndef GBA_KEYPAD_H
#define GBA_KEYPAD_H

#include "GBA_types.h"

#define KEY_A                   BIT0
#define KEY_B                   BIT1
#define KEY_SELECT              BIT2
#define KEY_START               BIT3
#define KEY_RIGHT               BIT4
#define KEY_LEFT                BIT5
#define KEY_UP                  BIT6
#define KEY_DOWN                BIT7
#define KEY_R                   BIT8
#define KEY_L                   BIT9
#endif
```

Now we can move on to main.

```
if(LoadPCX(picture, &image1) && LoadPCX(picture2, &image2))
```

This line loads the PCX files into the two images. If either image are not of the correct format LoadPCX returns 0 and the rest of the program will not be carried out.

```
      SetMode(MODE_4 | BG2_ENABLE);

      for(i = 0; i < SCREEN_WIDTH * SCREEN_HEIGHT / 2; i++)
         FrontBuffer[i] = image1.data16[i];

      for(i = 0; i < SCREEN_WIDTH * SCREEN_HEIGHT / 2; i++)
         BackBuffer[i] = image2.data16[i];

      for(i = 0; i < 256; i++)
         BGPaletteMem[i] = image1.palette[i];
```

First the mode is set to 4 and background 2 is enabled. Background 2 must be enabled for all bitmap modes (mode 3, 4, 5) in order for anything to appear on the screen. Next we load our images into video memory. The first picture is loaded into the FrontBuffer (defined in "gba.h" as (u16*)0x600:0000) and the second is loaded into the BackBuffer ((u16*)0x600:A000). You will notice that the image is loaded using it's 16-bit data pointer. All three data pointers point to the same location due to the union but as we can not access the video hardware one byte at a time we must use data16 or data32.

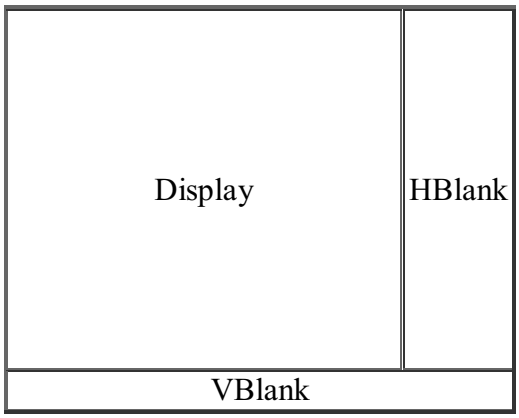
Finally we load the palette. There is only room for one background palette and since both images use the same palette we just load in the first pictures palette. If your PCX images where designed using different palettes then you will see some funky coloring when you switch to the other image.

```
    while(1)
    {
        if(!(REG_KEYS & KEY_A))
        {
            REG_DISPCNT ^= BACKBUFFER;
        }
    }
```

This last little bit puts the code into an infinite loop that monitors the A key. If the key press is detected it simple flops the back buffer bit with an xor ('^'). Since we have put no proper delay into our code this switch is extremely rapid which you will notice if you hold the A key down.

Blanking periods

Before moving on to raster graphics we need to talk a bit more about how the GBA renders the display. Much like a normal CRT, the LCD on the GBA goes line by line. The time in-between each scan line is known as the "Horizontal Blanking Period" or Hblank for short. During this time nothing is being rendered on screen and data can be altered to cause some interesting effects (Fzero racetracks for instance). After the last scan line is complete and before starting the process over there is a time of rest known as the "Vertical Blanking Period" or Vblank for short. It is during this time that changes are normally made to the video hardware to ensure that the screen is not changed mid-draw; an issue called tearing which, when unintentional, causes very ugly artifacting.



It would be nice if we could alter our PCX display program to wait for this vblank period before switching between the buffers. Let's do that. First we need to look at another register called REG_DISPSTAT

F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
Y Trigger								/	/	Ytri	Hirq	Yirq	YT	HB	YB

REG_DISPSTAT @ 0x400:0004

Bit 0 (YB): This bit is set if in Yblank period
Bit 1 (HB): This bit set if in Hblank period
Bit 2 (YT): This bit is set if past the scan line indicated by Y trigger
Bit 3 (Yirq): This bit, when set, will cause Yblank to generate an interrupt
Bit 4 (Hirq): This bit, when set, will cause Hblank to generate an interrupt
Bit 5 (Ytri): This bit, when set, will cause Y Trigger to generate an interrupt
Bit 8-F(Y Trigger): When the Ytri bit is set the 8-bit value stored in Y trigger corresponds to a scan line. When that scan line is reached an interrupt is generated

Let us build a WaitForVblank function that sits in a loop until the beginning of the Vblank.

```
void WaitForVblank(void)
{
   while(! (REG_DISPSTAT & DISPSTAT_VB));
   while(REG_DISPSTAT & DISPSTAT_VB);
}
```

First the function ensures that we are not already in the Vblank because we only want it to return at the *start* of the Vblank. It then returns once the Vblank bit is set. This is where the volatile keyword is crucial. The compiler otherwise would realize we are checking a variable repeatedly and store it in a CPU register to make things work faster, the volatile ensures that it is loaded each time it is checked otherwise the changes to the bits made by the video hardware would go unnoticed. Later when we learn of interrupts we will find a much more power friendly way to wait for the VBlank. Finally, we place this function in our tutor2_1.c and make the following change:

```
      while(1)
      {
         WaitForVblank();

         if(!(REG_KEYS & KEY_A))
         {
            if(REG_DISPCNT & BACKBUFFER)
               REG_DISPCNT &= ~BACKBUFFER;
            else
               REG_DISPCNT |= BACKBUFFER;
         }
      }
```

Hopefully this will prevent the tearing you noticed before. Keep in mind that if you hold down the A key, the image will still change 60 times per second. If you were building this file as we went now would be a good time to put it all together and compile it and test it out on your hardware. If you have not built an Xboo cable yet or have not purchased a Flash cart yet you will have to be content looking at your new creation in the emulator.

Raster Graphics

Raster graphics are the means by which all the early games were created. It simply means to render to a display on a per-pixel basis. We are going to let the GBA hardware do most of our rendering for us. But, it is sometimes fun to do things in software so a few simple raster graphics techniques will be discussed in this section.

Although software rasterization is little more than a novelty on the GBA there is a reason why I am including it here. The only topics covered will be Lines and line based polygons. This sections main benefit is not the raster graphics techniques; instead, its purpose is to serve as an introduction to matrix math and rotation which are easily demonstrated using lines and polygons.

Computer line drawing has been the focus of many books. There are a myriad of ways to go about rendering a line on the screen. For our purposes we are going to look at only one. It is relatively fast and easy to understand, and has the added benefit of being the most popular line drawing algorithm out there.

The algorithm we are going to concern ourselves with is known as Bresenham (after some guy likely named Bresenham). The premise of the algorithm is as follows: First you figure out which direction the line is changing the fastest in. In other words if it is taller then it is wide it is changing fastest in the y direction. You then loop through all the values of x or y in that direction and check to see if you need to change in the other direction each time through.

Normally to calculate the y position of a line when you know two points on that line you would plug it into the equation of the form:

Y – y = m(X – x)

Where x and y are any point on the line and the slope is calculated by dividing the difference in y values of any two points on the line by the difference in those two points corresponding x values.

Because this requires both a divide (which the GBA can only do in software) and a series of multiplies of fractional values, this method suffers in the speed area.

Bresenham found a way to do it with only integer math, addition, and subtraction. The method involves keeping track of an error term. After you have figured out whether you are tracing the line in the x or the y direction you check, every iteration, to see if you need to step in the other direction. This is done using an error term. For instance if you decided the x difference is greater you would start at x1 and increment until you reached x2 each time adding y difference to the error term. If the error term reaches x difference in size then you know it is time to increment the y value. The closer the difference in x values is to the difference in y values the more often the error term will overflow causing you to step in the y direction.

For example; if you had a horizontal line then the x difference would be the length of the line and the y difference would be zero. You step through the x values incrementing the error term by the y difference (0 in this case). Since the error term is always less then the x difference you will never increase the y value and the line will be plotted as a strait horizontal.

Another example would be a 45 degree line were x difference and y difference are equal. In this case you would loop through the x values adding y difference to the error term. Every iteration would cause the error term to reach the x difference value causing you to add one to the y value.

For a final example consider a case were the slop is ½. This means that the line moves twice the distance in the x direction as it does in the y. If you loop through the x's adding y difference to the error term then every *other* iteration the error term will reach x difference and you will add 1 to y which is exactly what you want.

It is now time to see the code for a simple Bresenham line drawing algorithm.

```
void DrawLine(int x1,int y1,int x2,int y2,unsigned short color)
{
   int yStep = 240;
   int xStep = 1;
   int xDiff = x2 - x1;
   int yDiff = y2 - y1;

   int errorTerm = 0;
   int offset = y1 * 240 + x1;
   int i;

   if (yDiff < 0)
   {
      yDiff = -yDiff;
      yStep = -yStep;
   }

   if (xDiff < 0)
   {
      xDiff = -xDiff;
      xStep = -xStep;
   }


   if (xDiff > yDiff)
   {
      for (i = 0; i < xDiff + 1; i++)
      {
         VideoBuffer[offset] = color;

         offset += xStep;

         errorTerm += yDiff;

         if (errorTerm > xDiff)
         {
```

```
            errorTerm -= xDiff;
            offset += yStep;
         }
      }
   }//end if xdiff > ydiff
   else
   {
      for (i = 0; i < yDiff + 1; i++)
      {
         VideoBuffer[offset] = color;

         offset += yStep;

         errorTerm += xDiff;

         if (errorTerm > yDiff)
         {
            errorTerm -= yDiff;
            offset += xStep;

         }
      }
   }
}
```

Analyzing the line drawing code

The first thing we do is declare the variables we will be using.

```
   int yStep = 240;
   int xStep = 1;
   int xDiff = x2 - x1;
   int yDiff = y2 - y1;

   int errorTerm = 0;
   int offset = y1 * 240 + x1;

   int i;
```

The step variables are necessary because we access our video buffer as an array and if you will recall, adding one to that array will move one pixel in the x direction but to move one in the y direction we must move an entire scan line (or 240 pixels). xDiff, yDiff, and errorTerm are exactly as described above and i is simply an indexing variable.

The offset value is what we will use to index our array. For now VideoBuffer[offset] points to the first point on our line.

```
   if (yDiff < 0)
   {
      yDiff = -yDiff;
      yStep = -yStep;
    }

   if (xDiff < 0)
```

```
    {
        xDiff = -xDiff;
        xStep = -xStep;
    }
```

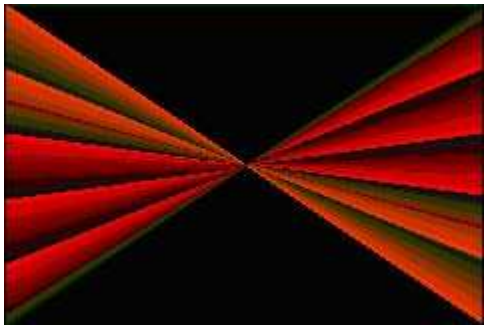
Next we must ensure that our xDiff and yDiff are positive values because the user may not enter the points like we would prefer. If they are backwards that is fine. We will just draw our line backwards by negating the value of the steps.

```
    if (xDiff > yDiff)
    {
        for (i = 0; i < xDiff + 1; i++)
        {
            VideoBuffer[offset] = color;

            offset += xStep;

            errorTerm += yDiff;

            if (errorTerm > xDiff)
            {
                errorTerm -= xDiff;
                offset += yStep;
            }
        }
    }//end if xdiff > ydiff
```

If the x difference is larger, we step through all the x values from point 1 to point 2. We increase our offset by the xStep each iteration. Also, the error term is incremented by y difference until the error term is greater than or equal to the x difference. We then reset the error term by subtracting x difference (instead of setting it to zero like you may be tempted to do).

If the y difference is only slightly less then x difference, resetting the error term to zero would cause the y value to only be incremented every other iteration when it needs to be incremented much more often.

After resetting the error term, we increase the offset by one y (240 pixels) and then begin a new iteration. The process for if yDiff is greater is nearly identical to that above.



Here is a tutorial 2 of day 2 which draws some lines. Only the main() source is posted since the rest of the code is already spread about this chapter.

```
////////////////// C code entry (main())//////////////////////
int main()
{
   SetMode(MODE_3 | BG2_ENABLE);
   u16 color = 0;
   u8 x1 = 0;
   u8 x2 = 0;
   u8 y1 = 0;
   u8 y2 = 0;

   while(1)
   {

      WaitForVblank();

      DrawLine(x1,y1,x2,y2,color);

      if(x1 < 239)
         x1++;
      else
         x1 = 0;

      if(x2 > 0)
         x1--;
      else x2 = 239;

      if(y2 < 159)
         y2++;
      else
         y2 = 0;

      if(y1 > 0)
         y1--;
      else
         y1 = 159;

      if(color < 32*32*32 - 1)
         color++;
      else
         color = 0;
   }

   return 0;
}//end main
```

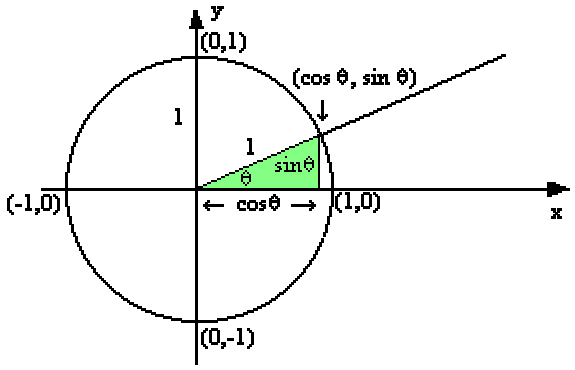
This code just declares a few variables to keep track of x and y and increments or decrements them along with the color to plot a changing array of lines on the screen.

That is it for line drawing and lines will be the only primitive we talk about as far as raster graphics go. If you are dying to create your own lighting fast 3D triangle texture mapper or, perhaps, just want to know a bit more then check out the recommended reading section that is at the end of this chapter.

Rotation and Matrices

You may be asking yourself why we went to all the trouble to learn about lines. The real reason is so I could show you how to deal with rotation and matrices. In order to understand how to rotate a point you must have a basic understanding of the Sin and Cos functions. If you will recall from geometry class Sin and Cos are functions that describe circular behavior.

Sin and Cos are best understood through the use of the "Unit Circle". The unit circle is simply a circle of radius 1.0 that is centered on the origin.



The nice thing about the Unit Circle is that any point on the circle can be calculated using sin and cos and by looking at the angle formed between the positive X axis and the line that extends from the origin to the point.

To rotate a point by an angle of theta about the origin the following equations are needed:

newX= cos(theta) * x – sin(theta) * y;

newY= sin(theta) * x + cos(theta) * y;

Of course, it is usually the case that you don't want to rotate the point around the origin. To rotate it about an arbitrary point you simply translate the x, y in such a manner as to make the arbitrary point the origin. How? If you need to rotate the point (x, y) about the point (a, b) you simply translate the point by subtracting (a, b), do the rotation, and then translate back by the same amount.

x = x – a;
y = y – b;

newX= cos(theta) * x – sin(theta) * y;
newY= sin(theta) * x + cos(theta) * y;

newX = newX + a;
newY = newY+ b;

With this is mind let us write a demo that will rotate a square about a point on the screen.

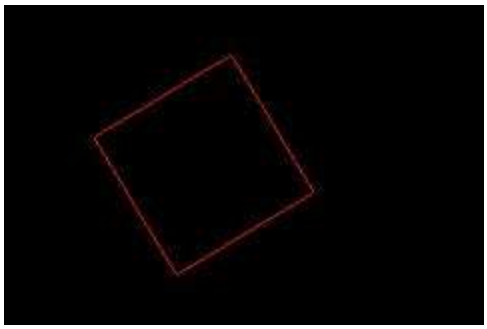
Before we begin the code for this demo there is one outstanding issue. Cos() and Sin() functions are about as slow as a function can get…and worse they deal in doubles and Radians. In order to speed things up we are going introduce our first Look Up Table (LUT). A LUT is a pre-calculated array of values that eliminates the need to do real-time calculations.

There are two ways to generate the LUTs. The first is to use the math library and build them when our program first runs. This works, but it takes up quite a bit of RAM not to mention adds an annoying pause to the beginning of your demo. We will be using the second method which involves simply creating the LUT in a windows program and outputting that data to a binary file. Here is the source code that creates the LUT:

It fills the array with Fixed point values of the form 2.14 (see Appendix A). It then writes these values to a binary file that we can include in our source.

```
#include <math.h>
#include <stdio.h>

#define PI 3.14159
#define RADIAN(n)  ((((double)n)/180.0)*PI)

int main(int argc, char* argv[])
{
   FILE* f;
   int angle;

   short int COS[360];
   short int SIN[360];

   for(angle = 0; angle < 360; angle++)
   {
      COS[angle] = (short int)(cos(RADIAN(angle)) * (double)(1<<14));
      SIN[angle] = (short int)(sin(RADIAN(angle)) * (double)(1<<14));
   }

   f = fopen("lut.bin","wb");

   fwrite(COS,360 * 2,1,f);
   fwrite(SIN,360 * 2,1,f);

   fclose(f);

   return 1;
}
```

Now that we have a binary file with our LUT stored inside we can just include it by one of the methods discussed in the previous chapter. For this demo I chose bin2o.



The demo source code follows. The code that was already covered has been removed to conserve space.

```
#include <gba.h>

extern short MATH_LUT[];

short* COS = &MATH_LUT[0];
short* SIN = &MATH_LUT[360];

typedef struct
{
   int x, y;
}sPoint2D;

typedef struct
{
   int x, y;
   sPoint2D p[4];
   unsigned short color;
}sBox;

//WaitForVblank and Draw line not shown...code is same as before

void DrawBox(sBox* box, int angle)
{
   sPoint2D p[4];

   p[0].x = ((COS[angle] * box->p[0].x - SIN[angle] * box->p[0].y)
>> 14) + box->x;
   p[0].y = ((SIN[angle] * box->p[0].x + COS[angle] * box->p[0].y)
>> 14) + box->y;

   p[1].x = ((COS[angle] * box->p[1].x - SIN[angle] * box->p[1].y)
>> 14) + box->x;
   p[1].y = ((SIN[angle] * box->p[1].x + COS[angle] * box->p[1].y)
>> 14) + box->y;

   p[2].x = ((COS[angle] * box->p[2].x - SIN[angle] * box->p[2].y)
>> 14) + box->x;
   p[2].y = ((SIN[angle] * box->p[2].x + COS[angle] * box->p[2].y)
>> 14) + box->y;

   p[3].x = ((COS[angle] * box->p[3].x - SIN[angle] * box->p[3].y)
```

```
>> 14) + box->x;
   p[3].y = ((SIN[angle] * box->p[3].x + COS[angle] * box->p[3].y)
>> 14) + box->y;

   DrawLine(p[0].x,p[0].y,p[1].x,p[1].y,box->color);
   DrawLine(p[1].x,p[1].y,p[2].x,p[2].y,box->color);
   DrawLine(p[2].x,p[2].y,p[3].x,p[3].y,box->color);
   DrawLine(p[3].x,p[3].y,p[0].x,p[0].y,box->color);
}

void ClrScreen(void)
{
   int i;
   for(i = 0; i < SCREEN_HEIGHT * SCREEN_WIDTH; i++)
      VideoBuffer[i] = 0;
}

int main()
{
   int angle = 0;

   sBox box =  {
            100, 80, //x,y
            {
            {-40,40}, //points
            {40,40},
            {40,-40},
            {-40,-40}
            },
            RGB16(31,0,0) //color
            };

   SetMode(MODE_3 | BG2_ENABLE);

   while(1)
   {
      WaitForVblank();
      ClrScreen();
      DrawBox(&box, angle % 360);
      angle++;
    }
   return 0;
 }//end main
```

First, you will notice that the identifier I sent to bin2o was MATH_LUT. This array will allow me access both tables since I know the COS table is 360 entries long and that the SIN table immediately follows.

```
extern short MATH_LUT[];
short* COS = &MATH_LUT[0];
short* SIN = &MATH_LUT[360];
```

I just create a reference array called SIN and COS that begin at the appropriate place in my LUT. Next we have two structures that simplify the data organization.

One defines a point as two integers (x, y) the other defines a box as being a set of points, a color, and a center point.

```
typedef struct
{
   int x, y;
}sPoint2D;

typedef struct
{
   int x, y;
   sPoint2D p[4];
   unsigned short color;
}sBox;
```

Then follows my WaitForVblank code as well as the DrawLine code we saw before. You will have to refer to earlier sections for a description of that code. Next comes the interesting part: DrawBox().

```
void DrawBox(sBox* box, int angle)
{
   sPoint2D p[4];

   p[0].x = ((COS[angle] * box->p[0].x - SIN[angle] * box->p[0].y) >>
14) + box->x;
   p[0].y = ((SIN[angle] * box->p[0].x + COS[angle] * box->p[0].y) >>
14) + box->y;

   p[1].x = ((COS[angle] * box->p[1].x - SIN[angle] * box->p[1].y) >>
14) + box->x;
   p[1].y = ((SIN[angle] * box->p[1].x + COS[angle] * box->p[1].y) >>
14) + box->y;

   p[2].x = ((COS[angle] * box->p[2].x - SIN[angle] * box->p[2].y) >>
14) + box->x;
   p[2].y = ((SIN[angle] * box->p[2].x + COS[angle] * box->p[2].y) >>
14) + box->y;

   p[3].x = ((COS[angle] * box->p[3].x - SIN[angle] * box->p[3].y) >>
14) + box->x;
   p[3].y = ((SIN[angle] * box->p[3].x + COS[angle] * box->p[3].y) >>
14) + box->y;

   DrawLine(p[0].x,p[0].y,p[1].x,p[1].y,box->color);
   DrawLine(p[1].x,p[1].y,p[2].x,p[2].y,box->color);
   DrawLine(p[2].x,p[2].y,p[3].x,p[3].y,box->color);
   DrawLine(p[3].x,p[3].y,p[0].x,p[0].y,box->color);
}
```

This code creates a temporary array of points to hold the rotated box. It then uses the formula from above to rotate each point. By giving the box a center and defining its points as relative to that center the rotation is quite simple… we just rotate it about the center (which is the box's origin) and then translate it out to the location of the center with a simple addition. The shift right 14 that you see is a conversion from the fixed point LUT to integer. If this is unclear be sure and reference Appendix

A. We then connect the points with 4 calls to our DrawLine function from earlier. This next function clears the screen and is necessary else we wind up with tracers of our rotating box that would quickly fill the screen. I believe the code speaks for itself.

```
void ClrScreen(void)
{
   int i;
   for(i = 0; i < SCREEN_HEIGHT * SCREEN_WIDTH; i++)
      VideoBuffer[i] = 0;
}
```

Finally we move to the main entry point of our program. The demo enters an infinite loop where it draws the box at an incrementing angle. The angle is modded with 360 to ensure we stay within the bounds of our SIN and COS arrays.

```
int main()
{
   int angle = 0;

   sBox box =  {
            100, 80, //x,y
            {
            {-40,40}, //points
            {40,40},
            {40,-40},
            {-40,-40}
            },
            RGB16(31,0,0) //color
            };



   SetMode(MODE_3 | BG2_ENABLE);

   while(1)
   {
      WaitForVblank();

      ClrScreen();

      DrawBox(&box, angle % 360);

      angle++;

   }
   return 0;
 }//end main
```

The only tricky portion might be the declaration of the box. If the notation is unfamiliar now would be a good time to refresh your memory on that sort of thing. Feel free to look at Appendix B which covers a bit of C and some of the less common things that you may have missed in your CS 101 class or in your C for dummies book.

3D Rotation
It turns out that to rotate a 3D point about an axis is something we already know how to do. In fact we just spent the last few pages doing it. In our case we were rotating a point about the z axis and all the z components where 0 but it just so happens that to rotate about the z axis you leave the z components as is anyway. Let's look at the equation to rotate a point about the x, y, and z axis.

Z axis (what we already know)
newX = cos(angle) * x – sin(angle) * y;
newY = sin(angle) * x + cos(angle) * y;
newZ = z;

X axis
newX = x;
newX = cos(angle) * y – sin(angle) * z;
newZ = sin(angle) * y + cos(angle) * z;

Y axis
newX = cos(angle) * x + sin(angle) * z;
newY = y;
newZ = -sin(angle) * x + cos(angle) * z;

To rotate a point by an angle about each axis we must perform each operation specified above. Generally speaking when you are doing 3D rotations you will need to rotate about each axis every time. This may seem like a lot of calculations to you and that is because it is…there is a much more efficient means of combining all these calculations and if you have not guessed already it is Matrices.

Matrices
Most of you have probably seen matrices before so this section should not be all that new. Matrices are generally used to represent equations in an organized manner making manipulation of multiple equations at the same time a simple manner.

If we can find a way to stick our rotation code in matrices the math not only becomes simpler to deal with but also much faster. Here is why it is simpler: Let us say we have a point (x,y,z) and we were to somehow represent this point in a matrix and also represent the rotation formulas for each axis in three other matrices. We will call our point [P] and our three rotation matrices [x], [y], and [z]. To rotate our point about the three axes we simply do the following.

[p] = [p] * [x] * [y] * [z]

We just multiply the 4 matrices together. Now, once you understand how to multiply matrices, you will find that this method results in the exact same number of calculations. So how is it faster? One of the great properties of matrices is that they are associative for multiplication. What this means is that:

[p] * [x] * [y] * [z] = [p] * ([x] * [y] * [z])

This does not change much when dealing with a single point, but when we have many points to rotate by the same rotation matrices then we can pre-compute [x] *

[y] * [z] and multiply [P] by a single matrix, greatly reducing the number of calculations required.

[R] = [x] * [y] * [z];
[P1] = [P1] * [R];
[P2] = [P2] * [R];

That is not all! Besides rotation we also often need to translate and scale our objects. By storing the translation and scaling equations into a matrix we can concatenate 5 calculations into a single matrix and multiply that matrix by all our points.

You should understand now why matrix math is crucial to video game graphics but you are probably feeling a bit clueless about the details of actually using the matrices. First let's figure out how to multiply two matrices together and then we will figure out how to place all this data into our matrices.

Let us say you had two matrices with the following elements:

[A]={1, 2, 3,
4, 5, 6,
7, 8, 9 }

[B]={1, 2,
3, 4,
5, 6 }

The mathematical formula for multiplying these together would be:

$[\mathbf{AB}]_{ij} = \mathbf{SIGMA}\ [\mathbf{A}]_{ik}[\mathbf{B}]_{kj}$

Where sigma represents the sum of the products of the elements in [a] times the elements in [B] according to the indices specified. In English multiplication is done as follows.

We look at the first row of [A] which is 1,2,3 and the first column of [B] 1,3,5

These must be the same length which means that to multiply two matrices together the number of columns in [A] must equal the number of rows in [B].

We then multiply the first element in the first row of [A] times the first in [B] and add it to the second and then to the third. This becomes the first element in [AB].

1*1 + 2*3 + 3*5 = 22

This makes [AB] =
22 ?
? ?
? ?

you then take the first row of [A] and the second column of [B]

1*2 + 2*4 + 3*6 = 22
[AB] =
22 28
? ?
? ?

Then the second row of [A] and the first column of [B]

4*1 + 5*3 + 6*5 = 49
4*2 + 5*4 + 6*6 = 66

[AB] =
22 28
49 66
? ?

And finally the last row and first and second column:

7*1 + 8*3 + 9*5 = 76
7*2 + 8*4 + 9*6 = 10

[AB] =
22 28
49 66
76 100

Okay now that we know how to multiply matrices we can set up our point as a 3x1 matrix and try to find the correct matrix to create our rotation.

[P] = {x, y, z}

First let us do the rotation about the z axis.

[z]={ cos(theta) , sin(theta), 0,
-sin(theta), cos(theta), 0,
0, 0, 1}

[P] * [Z] becomes:

{x*cos(theta) + y* -sin(theta) + 0, x*sin(theta) + y*cos(theta) + 0,
0+0+z}

These entries are exactly of the form we had above for rotation about the z axis.

[Y]={cos(theta) , 0, -sin(theta),
0, 1, 0,
sin(theta), 0, cos(theta, 1)}

[X]={1, 0, 0,
0, cos(theta) , sin(theta),
0, -sin(theta), cos(theta) }

To scale a point simply means to multily the x,y,z by some value. To do this in matrix form is quite simple

[P] = {x,y,z} * { zx, 0, 0,
0, zy, 0,
0, 0, zz} = { x*zx, y*yz, z*zz}

In this case zx, zy, and zz represent the scaling factors for each dimension.

Translation by matrices requires a bit of trickery. We must expand our point matrix to be equal to the following:

[P] = {x, y, z, 1}

Now, our translation matrix [T] can be written:

[T]={1, 0, 0, 0,
0, 1, 0, 0,
0, 0, 1, 0,
dx, dy, dz, 1}

This makes **[P] * [T] = {x + dx, y+ dy, z+dz, 1}**

Neglecting the 1 at the end and assuming that dx, dy, and dz are the translation amounts then we have exactly what we need. In order to concatenate out translation matrix with the others they must be expanded to 4x4 as well. For example [Z] will now be:

[z]={cos(theta) , sin(theta), 0, 0
-sin(theta), cos(theta), 0, 0,
0, 0, 1, 0
0, 0, 0, 1}

The others are expanded similarly. Keep in mind that although this looks like it takes up more space this is in fact an abstraction and when we implement these multiplies in code we will only store the needed data and insert the 1's and 0's were needed.

As you may have guessed we are not going to be doing to much 3D on the GBA, but there are certain tasks that the hardware performs that simulate 3D and those tasks are based on the theory above.

Although now would be an excellent time to do a 3D wire cube demo I just don't have the energy. This chapter has already reached the 35 page mark and that is a bit too large for me. If anyone would like to do a cube demo that uses my code and my headers I will post it in this tutorial. Please do not sacrifice any readability for performance unless necessary. For more on raster graphics I recommend the following:

Books:
Abrash, Michael. Zen of Graphics Programming.
LaMothe, Andre. Tricks of The 3D Game Programming Gurus.

Internet:
http://www.byte.com/abrash/ This is Abrash's complete black book of programming in an online format

This finally sums up day 2. We learned a lot today, but tomorrow things really start to move along. If you had any trouble compiling the demos be sure and check your copy paste and compare it to the downloadable versions of the source. Be sure and test your demos on hardware because the emulator is just downright dull when it comes down to it. So build an xboo or buy a flash cart!