Affine mode graphics on the Nintendo Virtual Boy

Part one: A brief introduction

By David Tucker

http://www.goliathindustries.com/vb/

Revision 0.93, August 19th 2005

Table of Contents

Introduction

The affine mode transform is a powerful tool that can be used to generate pseudo 3D displays on the Virtual Boy. If you have ever played Mario Kart on a SNES or a GBA than you have seen affine mode at work. This is a brief tutorial intended to introduce the user to the world of affine.

Matrices

Introduction to Matrices

A matrix is a tool used to collect up the terms of an equation to simplify the manipulation and description of that equation, sort of like a `struct` in C. A matrix is a two dimensional array with

$$m \times n \quad \text{dimensions} \quad = \begin{bmatrix} 1 & \cdots & n \\ \vdots & & \\ m & & \end{bmatrix}$$

A vector is a subset of a matrix that fixes one dimension to one. For our purposes a vector can be defined as an $1 \times m$ matrix $= \begin{bmatrix} 1 \\ \vdots \\ m \end{bmatrix}$ and can be treated the same as a matrix. Given a series of equations:

$$a = x + y \cdot t$$
$$b = d^2 + k/2$$

We can use matrices and vectors to represent the common terms in the equations. If we assign

$$\boldsymbol{V}=\begin{bmatrix} a \\ b \end{bmatrix}$$

and

$$\boldsymbol{M}=\begin{bmatrix} x & y\cdot t \\ d^2 & k/2 \end{bmatrix}$$

than we can rewrite the above equation as $\boldsymbol{V}=\boldsymbol{M}$ much simpler. You could also rewrite it as

$$\begin{bmatrix} a \\ b \end{bmatrix}=\begin{bmatrix} x & y\cdot t \\ d^2 & k/2 \end{bmatrix}$$

if you want to expose the underlying detail of the matrix.

Matrix Addition

Matrices must have the same dimensions in order to add them together. For example we could add a 3×2 matrix $\boldsymbol{A}$ and a 3×2 matrix $\boldsymbol{B}$ together, but a 3×3 matrix $\boldsymbol{C}$ could not be added to either $\boldsymbol{A}$ or $\boldsymbol{B}$ In order to add $\boldsymbol{A}$ and $\boldsymbol{B}$ simply add the corresponding elements in $\boldsymbol{A}$ and $\boldsymbol{B}$ to each other.

$$\boldsymbol{A} = \begin{bmatrix} a & b \\ c & d \\ e & f \end{bmatrix}$$

$$\boldsymbol{B} = \begin{bmatrix} g & h \\ i & j \\ k & l \end{bmatrix}$$

$$\boldsymbol{A}+\boldsymbol{B} = \begin{bmatrix} a+g & b+h \\ c+i & d+j \\ e+k & f+l \end{bmatrix}$$

Matrix Addition is commutative, in other words $\boldsymbol{A}+\boldsymbol{B}=\boldsymbol{B}+\boldsymbol{A}$

Matrix Multiplication

Matrices can be multiplied together if the left hand matrix has the same number of columns as the right hand matrix has rows. For example, we could multiply a 1×2 matrix with a 2×2 matrix, which would result in a 1×2 matrix, but we can not multiply a 2×2 matrix with a 1×2 matrix. In order to multiply a matrix we multiply the rows of the first matrix with the columns in the second matrix and sum up the results. It is easier to comprehend with an example:

$$\begin{aligned}
\boldsymbol{A} &= \begin{bmatrix} a \\ b \end{bmatrix} \\
\boldsymbol{B} &= \begin{bmatrix} c & d \\ e & f \end{bmatrix} \\
\boldsymbol{C} &= \begin{bmatrix} g & h \\ i & j \end{bmatrix} \\
\boldsymbol{B}\cdot\boldsymbol{A} &= \begin{bmatrix} ac+bd & ae+bf \end{bmatrix} \\
\boldsymbol{B}\cdot\boldsymbol{C} &= \begin{bmatrix} cg+di & ch+dj \\ eg+fi & eh+fj \end{bmatrix}
\end{aligned}$$

Multiplication is not commutative $\boldsymbol{A}\cdot\boldsymbol{B}\neq\boldsymbol{B}\cdot\boldsymbol{A}$ and as a final note, you can multiply a matrix by a scaler (a single number). This results in every field in the matrix being multiplied by the scaler

$$\boldsymbol{A}\cdot s=\begin{bmatrix} as & bs \\ cs & ds \end{bmatrix}$$

Further Reading

This should cover all that we need to know about matrices for this problem. If you want to know more, I strongly urge you to pick up a book on Linear Algebra or take a course on it at your local college.

Affine Transforms

Introducing the affine transform

An affine transform is a way to translate points from one coordinate space to another. There are several properties to affine transforms: 1 to 1 mapping, parallel lines remain parallel, etc., but none of that matters to us. For our purposes, an affine transform is a way to map a point on the display back into a texture buffer, or the BGMap, in such a way that we can perform simple translations on the source bitmap. For example, we can perform rotations, scales, shears, reflections, and, given some fancy trickery, we can even generate pseudo 3D environments.

General Affine transform

If we define a transformation matrix

$$\boldsymbol{P}=\begin{bmatrix} p_a & p_b \\ p_c & p_d \end{bmatrix}$$

and a displacement vector

$$\boldsymbol{Dx}=\begin{bmatrix} d_x \\ d_y \end{bmatrix}$$

we can define the general affine transform as

$$\begin{bmatrix} x' \\ y' \end{bmatrix}=\boldsymbol{P}\cdot\begin{bmatrix} x \\ y \end{bmatrix}+\boldsymbol{Dx}$$

where

$\begin{bmatrix} x' \\ y' \end{bmatrix}$ is a point on the BGMap

and

$\begin{bmatrix} x \\ y \end{bmatrix}$ is a point on the display

filling in $\boldsymbol{P}$ and $\boldsymbol{Dx}$ results in

$$\begin{bmatrix} x' \\ y' \end{bmatrix}=\begin{bmatrix} p_a & p_b \\ p_c & p_d \end{bmatrix}\cdot\begin{bmatrix} x \\ y \end{bmatrix}+\begin{bmatrix} d_x \\ d_y \end{bmatrix}$$

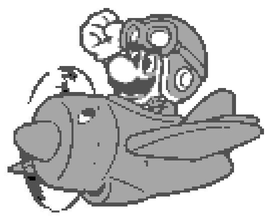
if we multiply out our equation we end up with

$$x'=p_a\cdot x+p_b\cdot y+d_x$$
$$y'=p_c\cdot x+p_d\cdot y+d_y$$

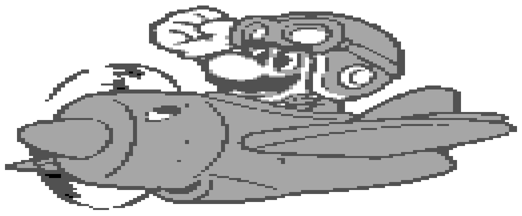
General forms of P

The following are some general forms of $\boldsymbol{P}$ that we can place into the above equation, along with a simple example of the result produced.

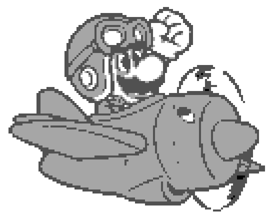
$$\boldsymbol{P}_{normal}=\begin{bmatrix}1 & 0\\ 0 & 1\end{bmatrix}$$



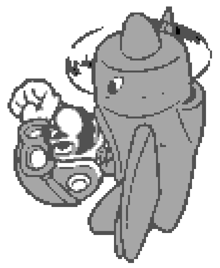
$$\boldsymbol{P}_{scale}=\begin{bmatrix}s_x & 0\\ 0 & s_y\end{bmatrix}$$



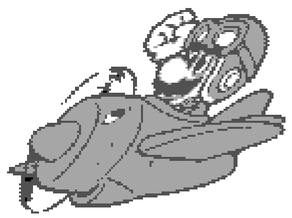
$$\boldsymbol{P}_{reflectY}=\begin{bmatrix}-1 & 0\\ 0 & 1\end{bmatrix}$$



$$\boldsymbol{P}_{reflectD}=\begin{bmatrix}0 & 1\\ 1 & 0\end{bmatrix}$$



$$\boldsymbol{P}_{shear}=\begin{bmatrix}1 & sh_x\\ sh_y & 1\end{bmatrix}$$



$$\boldsymbol{P}_{\boldsymbol{rotate}}=\begin{bmatrix}\cos(\alpha) & -\sin(\alpha)\\ \sin(\alpha) & \cos(\alpha)\end{bmatrix}$$

General form of Dx

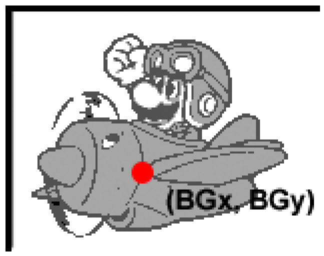
The $\boldsymbol{Dx}$ vector is used to set the offset of the object being transformed, without it an object's center of transformation is the upper left hand corner, and it is placed at the upper left hand corner of the display as well.

$$\boldsymbol{Dx}=\boldsymbol{Bg}-\boldsymbol{P}\cdot\boldsymbol{Dsp}$$

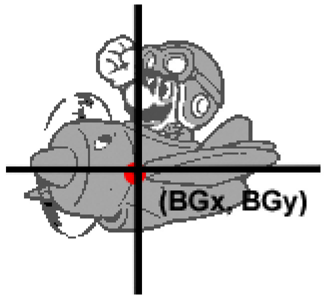
or expanded as

$$\boldsymbol{Dx}=\begin{bmatrix}bg_x-(p_a\cdot dsp_x+p_b\cdot dsp_y)\\ bg_y-(p_c\cdot dsp_x+p_d\cdot dsp_y)\end{bmatrix}$$

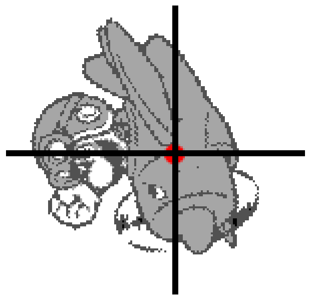
where $\boldsymbol{Bg}=\begin{bmatrix}bg_x\\ bg_y\end{bmatrix}$ is the center point on the BGMap image and $\boldsymbol{Dsp}=\begin{bmatrix}dsp_x\\ dsp_y\end{bmatrix}$ is the center point on the display.



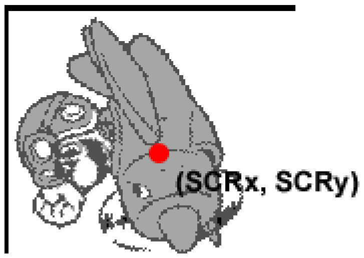
Define the center point on your BG image



Translate that point to the origin



Apply your P transformatoin



Translate the image back to a screen center point

Conclusion

In order to pull off more complicated results, such as a rotation followed by a scale of an image, we can combine the $\boldsymbol{P}$ matrices together via multiplication. Note, however, that the order that you place the transforms in matters. First because a translate followed by a rotation does not produce the same result as a rotation followed by a translation. Second because the image quality will change based on what order you perform the operations. Basically, you want to perform the least destructive transformation first in order to preserve the highest quality image. This concludes our discussion of theory, from here on out it is all hardcore examples.

The Virtual Boy and Affine Transforms

Introduction

The virtual boy pulls some tricks with the affine equation in order to make things like pseudo 3D environments easier to implement. First off, the VB computes a separate affine transform equation for each line on the display bitmap so that you can easily change the parameters to the $\boldsymbol{P}$ and $\boldsymbol{Dx}$ matrices on a line by line basis; this is stored in the param table on the Virtual Boy. Secondly, the VB combined the $\boldsymbol{P}$ and $\boldsymbol{Dx}$ matrices together in order to save space in memory and to move some of the multiplication out of the display render loop. I have defined the Virtual Boy's modified $\boldsymbol{P}$ matrix as

$$\boldsymbol{P}_{\boldsymbol{vb}}=\begin{bmatrix} p_a & y\cdot p_b+d_x \\ p_c & y\cdot p_d+d_y \end{bmatrix}$$

filling in $\boldsymbol{Dx}$ gives us

$$\boldsymbol{P}_{\boldsymbol{vb}}=\begin{bmatrix} p_a & y\cdot p_b+bg_x-(p_a\cdot dsp_x+p_b\cdot dsp_y) \\ p_c & y\cdot p_d+bg_y-(p_c\cdot dsp_x+p_d\cdot dsp_y) \end{bmatrix}$$

Notice how the $\boldsymbol{P}_{\boldsymbol{vb}}$ precomputes the line for each affine transform. This makes sense since we have one affine transform per line and it allows us to move the $\boldsymbol{Dx}$ vector into the $\boldsymbol{P}$ matrix. Otherwise, we would need a separate $\boldsymbol{Dx}$ vector for each line or we would need to modify the $\boldsymbol{Dx}$ vector for each line. This is how the Game Boy Advanced does things.

Affine structures in the Virtual Boy

The virtual boy defines a world entry as follows:

Table 1 - World entry format

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
LON	RON	BGM		SCX		SCY		OVR	END	0	0	BGMAP_BASE (0xD)			
GX (-0xFFFF - 0x17F)															
GP (-0x100 - 0x0FF)															
GY (-0xFFFF - 0x0DF)															
MX (-0xFFFF - 0xFFF)															
MP (-0x100 - 0x0FF)															
MY (-0xFFFF - 0xFFF)															
W (0x000 - 0xFFF)															
H (0x000 - 0xFFF)															
PARAM_BASE (0x000 - 0xEBF)															
OVERPLANE_CHARACTER															
WRITING FORBIDDEN															
WRITING FORBIDDEN															
WRITING FORBIDDEN															
WRITING FORBIDDEN															
WRITING FORBIDDEN															

There are three entries in the world entry that are specific to the Affine display mode.

First we must set the BGM type to Affine mode.

BGM - Sets the mode of the world

0	*Normal BGMap*
1	*H-bias BGMap*
2	*Affine*
3	*OBJ*

Next we need to set a pointer to the param table where we will store our $\boldsymbol{P_{vb}}$ matrices. The param table can be located anywhere between 0x00020000 and 0x0003D7FF. Notice however that this space is shared with the world table, so you are probably better off locating the param table after 0x0003C000 if you do not want to limit the number of worlds available to you.

PARAM_BASE - Parameter Table Base pointer, the last 4 bits must be zero.

```
True_base = (Param_Base & 0xFFF0) * 2 + 0x00020000
```

The last setting is the **OVR** flag. This flag controls the wrapping of a BGMap. Basically, if you try to index off of the end of your BGMap, you can either wrap to the beginning of the BGMap or you can fill in the missing bits with zero. So, if we set the OVR flag, then we disable BGMap wrapping, and, if we clear it, we enable the wrapping.

OVR - Turns off the display wrapping. If you retrieve a pixel from (515,32) on a single BGMap, it would be retrieved from (3,32), if over was not enabled.

Once we have defined the **PARAM_BASE**, we need to fill in our param table. There is one param table entry for every line of the image, so there are **GY** param table entries in total. Obviously, the pa, pb_y, pc and pd_y entries correspond to the elements in $\boldsymbol{P}_{vb} = \begin{bmatrix} p_a & y \cdot p_b + d_x \\ p_c & y \cdot p_d + d_y \end{bmatrix}$ and parallax is a horizontal shift that is applied to the display to generate a 3D effect just like GP and MP in the world structure. At this time it is not known whether parallax is applied to X before multiplication with $\boldsymbol{P}_{vb}$ or after. The purpose of the last three entries in the param table are not known at this time.

Table 2 - Affine Param table entry

15	0
pb_y (fixed point 13.3)	
paralax	
pd_y (fixed point 13.3)	
pa (fixed point 7.9)	
pc (fixed point 7.9)	
Unknown	
Unknown	
Unknown	

Fixed Point Math

When representing fractional numbers, such as $1/3 = 0.3333\bar{3}$ in a computer, you have two options: floating point and fixed point. Floating point (`float` and `double` in C) is the standard way to represent a fraction, but floating point math can be quite slow and, in video games, speed can be everything. So, some bright individual came up with the idea of fixed point math. Basically, we take a 16 bit or 32 bit number and declare that some number of the bits represent the fractional portion of the number. To convert from an int or float to a fixed point number, we would multiply the int by 2^x where X is the number of bits given over to the fractional part in our fixed point number.

For an example lets take an 16 bit number and let 8 bits represent the fractional portion and 8 bits for the integer portion, lets call this new type a FIXED_8_8. Now to convert an integer to a FIXED_8_8 we simply multiply the integer by 2^8 or 256, and to convert the FIXED_8_8 to a integer we divide by 256. We can speed up these operations by taking advantage of the fact that a multiplication by 2^x is the same as a binary shift to the left by x, and division by 2^x is the same as a binary shift to the right by x. To convert a floating point number to a FIXED_8_8 we perform the same operations only using a floating point constant so that the compiler does not truncate the fractional portion of our number in an effort to be more efficient. When converting from a float to an integer the least significant bit is always rounded down, we can offset this by adding 0.5 to the floating point number before assigning it to our FIXED_8_8 in order to force the number to round up or down as appropriate.

```
      #define FLOAT_TO_FIXED_8_8 (n)      (FIXED_8_8)((n)*256.0f+0.5f)
      #define FIXED_8_8 _TO_FLOAT(n)      (float)    ((n)/256.0f)
      #define INT_TO_FIXED_8_8(n)         (FIXED_8_8)((n)<<8) // n*256
      #define FIXED_8_8 _TO_INT(n)        (int)      ((n)>>8) // n/256
```

Addition and subtraction between two fixed point numbers operate the same as with integers. Multiplication and devision on the other hand bring about some small challenges that must be dealt with. Given two numbers A and B that we converted to fixed pint by multiplying with a scale factor S, multiplying the two numbers together results in the following equation: $(A \cdot S)(B \cdot S) = (A \cdot B) \cdot S^2$ As you can see the result is scaled by S^2 not S as we had intended. This can easily be corrected by either dividing A and B by the $\sqrt{S}$ before the multiplication step or by dividing the result by S after the multiplication. In the first case we loose half of our bits in the floating point portion of our number and in the later case we loose resolution in the real portion of the number. Notice that if you set S to be 1/2 of the bits available in your integer storage type that you potentially could loose all of the real portion of your number with a multiplication. There are many tricks that can work around this, most involve performing two multiplications one with the whole number and one with the fractional portion. However we could just as simply use a integer storage type that has more bits than we are looking for. For example if your final fixed point number is a FIXED_8_8 we can perform all of our math operations by using a FIXED_16_16, or a 32 bit number and convert the result back to a FIXED_8_8, or a 16 bit number, after all of our multiplications have been performed. Division is the opposite, when we are done dividing A by B we end up canceling out the S factor all together, and thus removing the fractional component. So in order to combat this we scale A by S before the division.

```
      //This fails because we loose the integer portion of our number
      #define FIXED_8_8_MUL_1(a,b) (((a)*(b))>>8)
      #define FIXED_8_8_DIV_1(a,b) (((a)<<8)/(b))

      //alternative, sacrifice accuracy for extra bits
      //this gives us 6:5 bits of resolution without truncation.
      #define FIXED_8_8_MUL_2(a,b) ((((a)>>3)*((b)>>3))>>2)
      #define FIXED_8_8_DIV_2(a,b) ((((a)<<3)/((b)>>3))>>2)

      //If we convert to a larger data type we preserve the lost bits
      #define FIXED_8_8_MUL_3(a,b) (FIXED_8_8)(((long)(a)*(long)(b))>>8)
      #define FIXED_8_8_DIV_3(a,b) (FIXED_8_8)(((long)(a)<<8)/(long)(b))
```

`Look into this:`

$$\frac{a}{10}\frac{b}{10} + \left(a'\frac{b}{10}\right)\Big/10 + \left(\frac{a}{10}b'\right)\Big/10 + \frac{a'b'}{100}$$

Scaling and Rotation

Here are some simple examples of how you might combine the $\boldsymbol{P}$ and $\boldsymbol{Dx}$ variables from before into a general $\boldsymbol{P_{vb}}$ equation on the Virtual boy.

scale
$$\begin{aligned} p_a &= scale_x \\ p_b &= 0 \\ p_c &= 0 \\ p_d &= scale_y \end{aligned}$$

rotation
$$\begin{aligned} p_a &= \cos(\alpha) \\ p_b &= -\sin(\alpha) \\ p_c &= \sin(\alpha) \\ p_d &= \cos(\alpha) \end{aligned}$$

scale and rotation
$$\begin{aligned} p_a &= \cos(\alpha)\cdot scale_x \\ p_b &= -\sin(\alpha)\cdot scale_x \\ p_c &= \sin(\alpha)\cdot scale_y \\ p_d &= \cos(\alpha)\cdot scale_y \end{aligned}$$

where
$$\boldsymbol{P}_{\boldsymbol{vb}} = \begin{bmatrix} p_a & y\cdot p_b + bg_x - (p_a\cdot dsp_x + p_b\cdot dsp_y) \\ p_c & y\cdot p_d + bg_y - (p_c\cdot dsp_x + p_d\cdot dsp_y) \end{bmatrix}$$

If we do not provide the ***Dx*** vector, then the scale and rotate functions use the upper left hand of the BGMap as the center point of the image. You should verify this for yourself by using an empty ***Dx*** vector in order to see what is going on.

In order to set things up we must first load up our Char and BGMap tables and define an affine mode world entry.

```
//memory base of the param table!
u16* const param = (u16*)0x0003C000;

void initialize() {
      //initiate world as affine mode
      WA[31].head = WRLD_ON | WRLD_AFFINE | WRLD_OVR;
      WA[31].gx = 0;
      WA[31].gy = 0;
      WA[31].gz = 0;
      WA[31].w = screenW;
      WA[31].h = screenH;
      WA[31].param = (u32)param;
      WA[31].over = 0x90;
      WA[30].head = WRLD_END;

      //move image to char table and bgmap
      copymem((BYTE*)CharSeg0, mario_char, mario_char_len);
      copymem((BYTE*)BGMap(0), mario_map, mario_map_len);
}
```

Next we iterate through the lines on the display, filling in the affine table as needed.

```
void affine_set_all(u8 world, PDx_ST * pdx) {
      s16 i, max;
      AFFINE_ST *affine;

      affine = (AFFINE_ST*)((WA[world].param<<1)+0x00020000);
      max = WA[world].h;

      for (i = 0; i < max; i++) {
            affine[i].pb_y    = FTOFIX13_3(i*pdx->pb+pdx->dx);
            affine[i].paralax = pdx->paralax;
            affine[i].pd_y    = FTOFIX13_3(i*pdx->pd+pdx->dy);
            affine[i].pa      = FTOFIX7_9(pdx->pa);
            affine[i].pc      = FTOFIX7_9(pdx->pc);
      }
}

void affine_rotscl(u8 world,s16 alpha, float zoom,
                              s16 bg_x, s16 bg_y, s16 fg_x, s16 fg_y) {
      PDx_ST pdx;
      pdx.pb  = 0.0f;

      pdx.pa  = COSF(alpha)*(1.0f/zoom);
      pdx.pb -= SINF(alpha)*(1.0f/zoom);
      pdx.pc  = SINF(alpha)*(1.0f/zoom);
      pdx.pd  = pdx.pa;

      pdx.dx = bg_x-(pdx.pa*fg_x + pdx.pb*fg_y);
      pdx.dy = bg_y-(pdx.pc*fg_x + pdx.pd*fg_y);

      pdx.paralax = 0;

      affine_set_all(world,&pdx);
}
```

If you place the above code in a loop you can change the affine parameters on every screen refresh, thus creating some simple animations.

```
while (1) {
      //Zoom by a multiplicative constant so
      //  closer objects move faster....
      zoom *= sclstep;

      if((zoom>maxscale)||(zoom<minscale)) {
            //reverse direction of scale
            sclstep = 1.0f/sclstep;
      }

      //increment (rotation)
      if(alpha++ > maxalpha) alpha = 0;

      affine_rotscl(31,alpha,zoom,
             imageW_Center, imageH_Center,
             screenW_Center, screenH_Center);

      //delay 1/50th of a second
      vbWaitFrame(1);
}
```

Conclusion

This concludes our introduction to Affine mode transforms on the Virtual boy. At this point you should have no trouble implementing simple scaling and rotation effects. In the next document we will discuss how to combine the affine mode transforms with some simple 3D projections to generate pseudo 3D environments.

References

- TONC GBA Affine Mode Documentation, http://user.chem.tue.nl/jakvijn/tonc/
- 2D Transformations - Introduction to Computer Graphics,

  Arizona State University, Dianne Hansford, February 2, 2005
- Fixed Point Math Tutorial
- 3D Graphics Math Book