NDS/Tutorials Day 3

< NDS

Contents

[edit] What is a register

The first concept we must get under our belts (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 DS has several different chips inside responsible for creating the images and sounds that accompany most games. There is sound hardware responsible for producing annoying chip tunes, the video hardware 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 CPUs which are 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 DS processors. But, the CPUs do 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 vital in understanding console development.

Using memory mapped registers requires some knowledge I hope you already possess (if you know any c at all) and that is: how to write data to any specific address.

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 32 bits in length and controls the display and we hence call it DISPLAY_CR; we might use code like the following to write to this address:

```
unsigned int  *DISPLAY_CR = (unsigned int *) 0x4000000;
*DISPLAY_CR = somevalue;
```

Now, this would work just fine but there are some issues with this code. First, because these are hardware registers it is possible (even likely) that the values stored at these addresses will be changed by the hardware directly. This is something the compiler needs to know else it will try to optimize our

code and we would miss these changes. The way we tell the compiler that variables change outside of the c code is to declare them volatile.

The other issue is 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 DISPLAY_CR  (*(volatile unsigned int *) 0x4000000)
DISPLAY_CR = 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 (of course the compiler being as smart as it is even had you used a variable it likely would have been smart enough to optimize and the result would have been the same).

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 DS

[edit] Twiddling Bits

It will rapidly become apparent that controlling hardware via registers will require an understanding of how to target specific bits inside the register. That is, we must be able to set or clear some bits in a register while leaving the rest untouched. Even though this is rather simple and talked about in many other places, it is important enough that we must waste the time of the 90% of readers who already know it in order to ensure the 10% who have no clue are not left in the dust.

[edit] Numbering Systems

Most of you can begin at 0 and count all the way to 9 (a feat by anyone's reckoning). If so, you probably realize there are in fact 10 unique digits you will come across in this endeavor. Oddly enough, the numbering system we use on a daily basis is called base 10 (in academic elitist societies you may also hear it referred to as Decimal).

First let us review an interesting detail about base 10 that you already know: the significance of the placement of the digits in any number. If the digit is on the right-hand side of a number it is generally weighted less than those digits appearing on the left to such a degree one might call it exponential. For instance consider the following examples of decimal numbers:

```
1) 1    = 1 * 10^0
2) 10   = 1 * 10^1
3) 100  = 1 * 10^2
4) 1000 = 1 * 10^3
```

This ringing any bells? You realize quite readily that the weight of the digit in question is equal to 10 to the power of the place of that digit in the number.

Unfortunately for us, computers do not use the same numbering system we do. The reason for this is a simple one. They only know 2 digits. Why is this unfortunate for us? It turns out that 90 percent of the time we can ignore the fact that computers use Base 2 because the compilers and tools we use automatically convert our base 10 numbers to binary for us. But, sometimes an understanding of the computer numbering system is crucial to coding.

The binary system works much in the same way as does our own decimal system with the exception that it has fewer digits and instead of weighting the value of a digit by 10 to the power of the place we instead weight it by 2 to the power of the place. For instance here are the same numbers as above in base 2 and their decimal equivalents.

```
1) 1    = 1 * 2^0 = 1
2) 10   = 1 * 2^1 = 2
3) 100  = 1 * 2^2 = 4
4) 1000 = 1 * 2^3 = 8
```

You might notice writing the value 8 in binary requires 4 digits, this may not seem an issue off hand but as numbers increase writing binary rapidly becomes cumbersome. A string of ones and zeros is prone to error and very difficult to read. To combat this, a new base was developed making manipulation of numbers on computers easier to handle. That numbering system is base 16; often referred to as Hexadecimal or Hex.

Hex numbering uses the digits 0 – 9 and A – F and you end up with numbers which look like the following:

```
4d6172696f
4b69726279
4c75696769
5a656c6461
```

At first you may be wondering why the hell you would ever go through such seeming pain to write numbers in such a way. Well it turns out the conversion between Hex and Binary is very simple. Because there are 16 digits in hex (a power of 2 mind you) you can represent each hex digit with 4 binary numbers. All you need do is become familiar with counting in binary from 0 to 15 to convert back and forth.

A few examples might be in order:

Take the binary number 11010010010101010101010001 To convert this number to decimal would require you to look at each bit and add 2 to the power of the place if there is a one in that position. This number in decimal becomes:

```
1 + 16 + 64 + 256 + 1024 + ….. and then you give up and put it in your calculator and get: 220
```

To do the same conversion to Hex you break the number into 4 bit nibbles and convert so it becomes:

```
1101 0010 0101 0101 0101 0101 0001
 D    2    5    5    5    5    1      = D255551
```

Now that you believe that hex to binary might be simpler than binary to decimal you may still be a bit unclear on why we don't just write everything in decimal and let the compiler figure it out (because it will). As we said before computers are binary in nature and as such hardware is controlled by specific bits at specific addresses. Because we need to set specific bits in the binary number we must at some point think of the number in binary. This will become more apparent as we actually set those bits.

Another good reason is bit boundaries play a significant role in memory addressing on most systems. For instance DS video memory for one of the graphics units begins on a boundry. The address of this memory can be written in hex as 6000000, that same address in decimal is 100,663,296. Although you could store the value as a pointer and write to video memory using either numbering system the hex value is much easier to remember and much easier on the eyes.

As a final note in C hexadecimal numbers are denoted with an 0x at the beginning of the number.

```
0x3FF
0x3ff
0x60000000
```

Notice case is not an issue.

[edit] Bitwise Operations

Being able to 'and' and 'or' bits together is important when attempting to enable certain features of hardware. Below is a summation of how bitwise operators work in C and how you might use them.

AND operator: '&'

Different than the logical and '&&' the single ampersand denotes two operands should be 'anded' together. Each number will be compared against the other bit by bit and if either number has a 0 in that bit position a 0 will be stored in the result.

AND is useful for checking the status of bits. For instance to see if the first bit of a register is set simply AND the register with the value 1. The result can only be 1 if the first bit of the register is also 1.

```
Register value = 1101 0100 1101 1101
               & 0000 0000 0000 0001
-------------------------------------
        Result = 0000 0000 0000 0001
```

It is also good for clearing bits. If you want the lower 8 bits of a number cleared to 0 simply AND the value with a number that has all the other bits set to 1. This is where hex comes in very handy.

```
Original Number            = 1101 0100 1101 1101
                           & 1111 1111 0000 0000  = 0xFF00
-----------------------------------------------
Result with low 8 bits clear 1101 0100 0000 0000
```

OR operator: '|'

A bit by bit comparison which causes the result bits to be set if either of the bits in the arguments are set.

OR is good for setting bits. To set a bit, simply OR the register with a value that has that bit and that bit only set. Here is an example of setting bit 9 of a 16 bit number (bits are numbered right to left begining at 0)

```
Old Value =              1101 0100 1101 1101
                       | 0000 0010 0000 0000
-------------------------------------------
New value with bit 9 set 1101 0110 1101 1101
```

XOR Operator: '^'

XOR is great for flipping between states. In the above example, if an XOR was used in the place of an

OR then the bit would be set if it was clear and it would be cleared if it were set. This is very useful anytime you need certain bits to alternate states every frame.

NOT Operator: '~'

NOT inverts the bits in a number rendering all 1s to 0s and all 0s to 1s. This is very useful for clearing bits. For instance if you know the main graphics engine will render to the top LCD when Bit 15 of the power control register is set to 1 and on the bottom when set to 0. We can 'or' the power control register with bit 15 to set it and we can 'and' the register with bit 15 inverted to clear it. Here is a snippet from libnds.

```
//! Forces the main core to display on the top.
static inline void lcdMainOnTop(void) { POWER_CR |= POWER_SWAP_LCDS; }

//! Forces the main core to display on the bottom.
static inline void lcdMainOnBottom(void) { POWER_CR &= ~POWER_SWAP_LCDS; }
```

In this case POWER_CR is defined as a pointer to the power control register and POWER_SWAP_LCDS is defined as bit 15.

The final bitwise operations to talk about are the shift operations '>>' and '<<'. When used these operators cause the binary number to be shifted to the left or right the specified number of places:

```
0101011 << 1 = 1010110
0101011 << 2 = 0101100
0101011 << 3 = 1011000
```

```
0101011 >> 1 = 0010101
0101011 >> 2 = 0001010
0101011 >> 3 = 0000101
```

If you will recall the weight of a digit is proportional to the base raised to the power of its position in the number. When we shift numbers to the right '>>' we are reducing the weight of the digits effectively dividing the number by 2^n where n is the amount we shifted. Similarly a left shift '<<' will multiply by a power of 2.

Often it is beneficial to use shift operators when division and multiplication are required as they execute more quickly. Do not get carried away though as they are less readable and the compiler will convert multiplications to shifts when ever possible for you.

The shift operator has other uses and plays a big role in fixed point arithmetic which we will cover shortly.

[edit] Talking to the keypad

It is difficult to do any interesting yet simple demo programs without understanding how to read user input. Fortunately for us, getting the state of the DS keys is exceedingly simple (if you understood the above discussion that is).

The state of each button is stored as a bit in memory mapped register space. To know if a key is pressed or released we just read the state of a specific bit. All we need to process the keys is the knowledge of where these values are stored.

Let us write our first real demo that checks for key presses and prints their state on the screen. Before we get to the code let us look at the main register used for key state on the DS.

(insert key pad register description here).

One thing you might note is the glaring absence of the X and Y keys. A bit further down we will demonstrate a more refined approach to handling input and introduce the functionality built into libnds and see if we can't find those missing buttons. For now let us get our first real demo out of the way.

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

int main(void)
{
 consoleDemoInit();

 while(1)
 {
  if(REG_KEYINPUT & KEY_A)
   printf("Key A is released");
  else
   printf("Key A is pressed");

  swiWaitForVBlank();

  consoleClear();
 }

 return 0;
}
```

Much of this code is as you have seen before. We initialize the print console so printf prints to the sub

screen using default settings. The next two lines initialize the libnds interrupt handler and enable the vblank interrupt. This is necessary for something we do in the main loop but it is a bit out of scope for this first day. We will talk at much greater length about interrupts (IRQs) on a later day.

The main loop just checks the KEY_A bit of the input register. This happens to be bit 0 and when the key is pressed that bit will be clear. This is all there is to checking for key presses on the DS.

The next two lines of code force the DS to wait until the screen is done drawing and then clears the screen. This prevents some nasty looking text flickering. Again, understanding this bit of code requires some knowledge of interrupts which will have to wait.

Now that simple reading of the key presses has been covered it is time to consider a bit more advanced needs…such as what about those X and Y buttons?

Unfortunately the designers of the DS were a bit lazy and stole the GBA input hardware; it seems our wonderful GBA input was a bit lacking in the number of buttons available to the user. What this resulted in is that we can read the A, B, Up, Down, Left, Right, Start, Select, and the Left and Right shoulder buttons from one place but the X, and Y buttons are in a different register…and can't even be read at all by the main CPU!. It seems in our very first foray into DS programming we must face the complexities of a dual processor system.

The solution is to read the keypad from the ARM 7 (which can read the X and Y buttons plus the hinge "button" on the DS lid and the pen state for the touch pad) and put the results someplace readable by the ARM 9.

If you are like me then this code seems a bit awkward. It would be nice if we had some way of wrapping all these bits into a single location to simplify the reading of key presses. Libnds provides just such a wrapper. Let us see how we would do the same code using the libnds wrapper then take a closer look at what the wrapper is doing.

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

int main(void)
{
 consoleDemoInit();

 while(1)
 {
  scanKeys();
  int held = keysHeld();

  if( held & KEY_A)
   printf("Key A is pressed\n");
  else
   printf("Key A is released\n");

  if( held & KEY_X)
   printf("Key X is pressed\n");
  else
   printf("Key X is released\n");

  if( held & KEY_TOUCH)
   printf("Touch pad is touched\n");
  else
   printf("Touch pad is not touched\n");

  swiWaitForVBlank();

  consoleClear();
 }

 return 0;
}
```

Two things to note in this new demo is the use of a scanKeys() call every frame and the change to positive logic: Now the bits are set if the key is pressed and clear if they are released. Along with keysHeld() is a keysDown() which will only be true if the key was pressed since the last time you checked (ie it will return true once but unless the player releases the key and presses it again it will return false).

Some other useful functions are keysUp() which returns the released keys and keysDownRepeate which returns true after a certain delay (measured in number of scanKey() calls) even if the keys have been held down. Check the documentation for input.h for more information on how to use these other functions.

Basically the way scanKeys works is to combine the bits from REG_KEYINPUT and IPC->buttons and apply a little bit of state tracking to determine which have been pressed since the last call. Here is an excerpt from the key handling code in libnds:

```
#define KEYS_CUR (( ((~REG_KEYINPUT)&0x3ff) | (((~IPC->buttons)&3)<<10) | (((~IPC->buttons)<<6

void scanKeys(void) {
 keysold = keys;
 keys = KEYS_CUR;

    ///..some code for handling key repeats
}

uint32 keysHeld(void) {
 return keys;
}

uint32 keysDown(void) {
 return (keys ^ keysold) & keys;
}
```

The statement at the beginning does most of the work by negating the register state and masking out / recombining the two sources of key state. If you have not noticed by now you will need to have a decent understanding of bit operations to work with register controlled hardware.

[edit] Frame buffer...finally

It is nice to finally get to graphics programming…I don't know about you but two full days of fluff is about all I can take.

If you have actually followed along with the subjects and code presented so far you will find doing frame buffer graphics on the DS is surprisingly simple. All we need do is put the DS into frame buffer mode and begin writing images to the screen.

If you recall from yesterday's topic the DS supports many graphics modes with the main screen supporting a simple frame buffer. It is this mode we will turn to first as it is very easy to set up and even easier to use.

I figured I would start this chapter with some code and use that to explain the frame buffer mode.

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

int main(void)
{
 int i;

 //initialize the DS Dos-like functionality
 consoleDemoInit();

 //set frame buffer mode 0
 videoSetMode(MODE_FB0);

 //enable VRAM A for writing by the cpu and use
 //as a framebuffer by video hardware
 vramSetBankA(VRAM_A_LCD);

 while(1)
 {
  u16 color = RGB15(31,0,0); //red

  scanKeys();
  int held = keysHeld();

  if(held & KEY_A)
   color = RGB15(0,31,0); //green

  if (held & KEY_X)
   color = RGB15(0,0,31); //blue

  swiWaitForVBlank();

  //fill video memory with the chosen color
  for(i = 0; i < 256*192; i++)
   VRAM_A[i] = color;
 }

 return 0;
}
```

This code is very similar to the code for the day 1 introduction demo. As before we enable interrupts and turn on the vblank interrupt. Next we set the video mode to a frame buffer mode which uses the first video ram bank. We then set the first VRAM bank to be writable by the CPU and to act as a buffer for the LCD (recall the first VRAM bank is VRAM_A).

The main loop creates a color as a 16 bit unsigned short integer and sets its value to red. If A or X are pressed then the color is altered to be green or blue respectively. Finally we wait for the screen draw to finish and fill VRAM_A with the selected color.

Now that we have a base understanding of the code we need to get to the details, namely:

- How do videoSetMode and vramSetBankx work?
- How does the DS treat color?

- What the hell is a frame buffer and how do I write pixels to it?

These are the questions we will now explore.

[edit] Display Control

```
(todo: add description of display control register here)
```

[edit] VRAM Control

```
(todo: add some guidance on VRAM control...or delete this section and move it to the next chap
```

[edit] DS Color Formats

The DS has several ways in which it represents color. These ways generally fall into two categories: Paletted and Direct.

Direct color means the value directly controls the intensity of red, green, and blue that is fed to the pixel. There are technically two direct color formats used by the DS but you will see the variance between the two is minimal.

Direct color uses 5 bits to represent how bright each color component can be (red, green, and blue). We refer to this format as 555 or sometimes 15 bit color. If you recall from our discussion on binary numbers, 5 bits amounts to 32 levels of intensity for each of the three colors.

To describe a color in this format we need to combine our desired values of red, green, and blue to form one 15-bit number. The color components are stored as follows:

xBBBBBGGGGGRRRRR

This can be translated as the least significant 5 bits hold the red component, the next 5 bits hold the green and the remaining 5 hold the blue. We refer to this as BGR format. This would be a good time to flex our bit twiddling muscles and see if we cant define some colors.

To do this we specify an intensity for each of the 3 components and then shift the values into the correct place, finally we OR them all together to get our 15 bit value.

```
int red = 31;
int blue = 0;
int green = 0;

unsigned short int color_red = (blue << 10) | (green << 5) | red;
```

Simple enough? Since we don't normally want to concern ourselves with this detail we use the macro provided by libnds (or write our own) which is depicted below:

```
#define RGB15(r,g,b)  ((r)|((g)<<5)|((b)<<10))

//to use:

unsigned short int color = RGB15(red, green, blue);
```

Taking a short break from theory you should now glance up to the demo we just wrote. You will notice I used this macro to paint the screen red. It should be apparent at this point that the frame buffer utilizes 15 bit Direct color format.

The other Direct color format is nearly identical to the one just discussed. The only difference is in the most significant bit (depicted as the 'x' in xBBBBBGGGGGRRRRR above). When utilizing this format this bit is known as the 'alpha' bit and when set to 0 will prevent the color from appearing onscreen. Most 16 bit graphics operations on the DS utilize this "alpha" bit to determine transperency of the rendered pixel.

When we move on to Direct color bitmap modes you will quickly discover not setting this alpha bit will result in nothing on screen. Recall to set this bit requires some more bit operations:

```
//set alpha
color = color | (1<<15);

//clear alpha
color = color & ~(1<<15);
```

Although direct color formats provide a wide range of colors they have a serious drawback: They take up 16 bits for each pixel. Although the DS has an abundant amount of memory and CPU power compared to early 2D systems it still pales in comparison to most modern machines. You will be surprised how rapidly you will fill memory with a 16 bit image or how much stress you will place on the hardware if you attempt to blit large 16 bit images.

To alleviate this the DS utilizes many space saving tricks. The most prevalent is the use of paletted colors. Instead of specifying color components directly we instead build a table of colors and specify an index into this table.

Let us say we have a 256 color table (we will refer to this table as the "palette") which contains 256 direct color values. We can then set pixels onscreen to these values by specifying an index. Because the table is small we would only need an 8 bit index to describe the pixel color…a savings of 50%!.

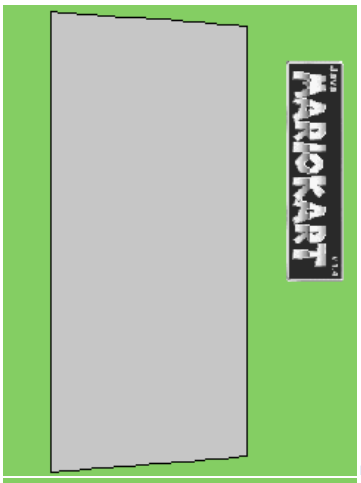
The DS supports 8 bit indexed palettes as well as 4 bit and we will figure out the mechanics of their use as we proceed.

[edit] Frame buffer 101

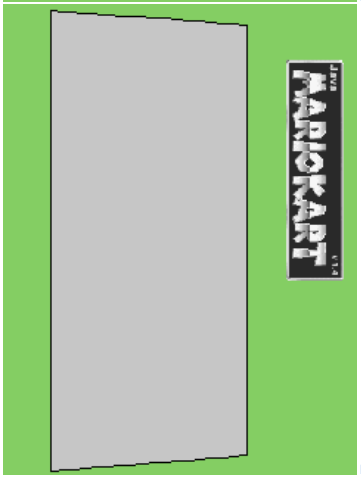
A frame buffer can be described as a direct map of memory values to onscreen colors. By simply writing the correct color value into memory we can set a pixel as we see fit.

Framebuffer memory can be completely described by three things: The address at which it begins, the color format of the pixels, and the number of pixels per horizontal line.

The memory for a frame buffer is a single linear map such that the first W entries correspond to the top row of pixels on the screen (W in this case is the "width" of the buffer). The next row of pixels follows and occupies entries W to 2*W – 1.



(image of linear memory)



(image of memory as it represents 2D space)

[edit] Pixels and things

To accurately place pixels onscreen we must have some idea how to specify location. This is normally done using a modified Cartesian coordinate system where we specify how many pixels from the left and how many pixels from the top we wish our value to be placed.

Image:Coordinate sys.png

The distance from the left hand side of the screen is usually referred to as the X coordinate of the pixel and the distance from the top is the Y coordinate. (Those of you who are math whizzes might see the disparity between Cartesian coordinates as Y usually is measured from the bottom up…live with it).

To figure out the offset into framebuffer memory we need to perform a simple calculation based on the X and Y coordinates we wish to affect. Because memory is arranged linearly in the buffer to get to the correct horizontal line we simply multiply the number of pixels on a line by the value of the Y coordinate. We then add the value of X and we have our offset.

Image:Pixel offseting

```
unsigned short* frame_buffer = address_of_the_buffer;

frame_buffer[y * width_in_pixels + x] = color;
```

Let us translate this new knowledge of pixel plotting and color formats and see if we can produce an interesting (sort of) demo.

For our first pixel demo we will do a starfield with little floating dots. Each dot will make its way across the screen at a random speed. When it reaches the end we will move it back to the beginning and give it a new random height and new random speed. This should give us a nice star-trekie feeling demo of a moving star field.

Here is the source in its entirety which we will pick apart below; you can cut and paste this code into your main.c (or template.c if it is so named) from our first few demos. You can then build and run the demo.

```
#include <nds.h>
#include <stdlib.h>

#define NUM_STARS 40

typedef struct
{
 int x;
 int y;
 int speed;
 unsigned short color;

}Star;


Star stars[NUM_STARS];

void MoveStar(Star* star)
{
 star->x += star->speed;

 if(star->x >= SCREEN_WIDTH)
 {
  star->color = RGB15(31,31,31);
  star->x = 0;
  star->y = rand() % 192;
  star->speed = rand() % 4 + 1;
 }
}

void ClearScreen(void)
{
     int i;

     for(i = 0; i < 256 * 192; i++)
           VRAM_A[i] = RGB15(0,0,0);
}

void InitStars(void)
{
 int i;

 for(i = 0; i < NUM_STARS; i++)
 {
  stars[i].color = RGB15(31,31,31);
  stars[i].x = rand() % 256;
  stars[i].y = rand() % 192;
  stars[i].speed = rand() % 4 + 1;
 }
}
void DrawStar(Star* star)
{
 VRAM_A[star->x + star->y * SCREEN_WIDTH] = star->color;
}

void EraseStar(Star* star)
{
 VRAM_A[star->x + star->y * SCREEN_WIDTH] = RGB15(0,0,0);
}

int main(void)
{
 int i;

 irqInit();
 irqEnable(IRQ_VBLANK);

 videoSetMode(MODE_FB0);
 vramSetBankA(VRAM_A_LCD);

       ClearScreen();
 InitStars();

 //we like infinite loops in console dev!
 while(1)
 {
  swiWaitForVBlank();

  for(i = 0; i < NUM_STARS; i++)
  {
   EraseStar(&stars[i]);

   MoveStar(&stars[i]);

   DrawStar(&stars[i]);
  }
 }

 return 0;
}
```

We begin with a structure to define our star. It needs a location in the form of an X and Y coordinate, it needs speed, and finally it needs color.

```
typedef struct
{
 int x;
 int y;
 int speed;
 unsigned short color;

}Star;
```

We then need an array of stars we can track across the screen:

```
#define NUM_STARS 40

Star stars[NUM_STARS];
```

Before we start the demo, we need to clear the pixels of any color information they currently have. In other words, we are making sure we start with a black screen.

```
void ClearScreen(void)
{
     int i;

     for(i = 0; i < 256 * 192; i++)
           VRAM_A[i] = RGB15(0,0,0);
}
```

To start the demo off it would be nice if we could arrange our stars randomly about the screen. We do this with an initialize function.

```
void InitStars(void)
{
 int i;

 for(i = 0; i < NUM_STARS; i++)
 {
  stars[i].color = RGB15(31,31,31);
  stars[i].x = rand() % 256;
  stars[i].y = rand() % 192;
  stars[i].speed = rand() % 4 + 1;
 }
}
```

This function loops through all stars and sets the color to white, the speed to a random value between 1 and 4 and the X and Y to some random location on screen. If the '%' is unfamiliar to you I will give a brief explanation.

'%' performs a division and returns the remainder of that division.

Rand() returns a random short integer so moding the value with a number will result in a value which is between zero and that number. To generate a random number in any range between MIN and MAX is simply:

```
Num = rand() % (MAX-MIN) + MIN;
```

Next we need some function to move, draw, and erase the star. Let us begin with erase.

```
void EraseStar(Star* star)
{
 VRAM_A[star->x + star->y * SCREEN_WIDTH] = RGB15(0,0,0);
}
```

To erase just set the location of the star in the framebuffer to the background color (black in our case).

Similarly to draw the star we set the location of the star in the frame buffer to the color of the star:

```
void DrawStar(Star* star)
{
 VRAM_A[star->x + star->y * SCREEN_WIDTH] = star->color;
}
```

The final step is to move the star to its new location:

```
void MoveStar(Star* star)
{
 star->x += star->speed;

 if(star->x >= SCREEN_WIDTH)
 {
  star->color = RGB15(31,31,31);
  star->x = 0;
  star->y = rand() % 192;
  star->speed = rand() % 4 + 1;
 }
}
```

Moving a star is simple, we just add its speed to its current x position. The caviate is we must then check if the star has gone off screen. To do that we compare its x location to the width of the screen. If it is greater we know we are off the screen and we can take appropriate action.

When a sprite goes off screen we move it back to the left by settings its X value to 0. We then give it another random speed and random Y value making it look like a new star has come on screen.

The main loop which controls the demo consists of looping through each star and first erasing it from its old position, then moving it to its new position, and finally redrawing it at its new location:

```
//we like infinite loops in console dev!
 while(1)
 {
  swiWaitForVBlank();

  for(i = 0; i < NUM_STARS; i++)
  {
   EraseStar(&stars[i]);

   MoveStar(&stars[i]);

   DrawStar(&stars[i]);
  }
 }
```

And so ends our pixel plotting demo. Below are a few more demos explained in the same excruciating manner as above.

Color Bar Demo

[edit] Touching things

The touch pad is an amazing addition to a handheld video game system that not only makes for an interesting gameplay experience but so too does it add a new level of fun to game programming.

This section will introduce the touch pad, show you how it works, and give a quick demo on its use.

The DS touchpad utilizes a resistive coating which changes conduction depending on the area of the contacting object. This change is measured by some analogue to digital converts on a special chip inside the DS and translated to an X and Y location. These measurements can also be used to determine the area of the contact point which, to some degree, can be translated into pressure.

To get to this raw data we must communicate with this chip via a serial interface which is only accessible via the ARM 7. Currently I do not have the stomach to go into serial comms in this tutorial but if you have a mind to explore such things the source code is in the arm7 code base of libnds.

For now I am just going to do a bit of hand waving and tell you there is code running on the arm7 in the default arm7 template which reads out the state of the touch pad. This data is unformed and does not correlate exactly to the dimensions of the screen meaning some processing must be done to convert this raw location data to useful pixel data.

The libnds arm7 stub does the appropriate conversion and communicates the result to the ARM9. These values are then read using the touchRead function which writes the raw coordinates and the transformed pixel coordinates into the pointer parameter.

You can also get a go - nogo test of the pen by using the scanKeys() macro we discussed as above. This tells you if the pen is up or down so you know when to read the touch pad.

Now for a simple demo. We will make the simplest of art programs possible. It will render random colored dots wherever you touch the screen.

```
#include<nds.h>
#include<stdlib.h>

int main(void)
{
 touchPosition touch;

 videoSetMode(MODE_FB0);
 vramSetBankA(VRAM_A_LCD);

        //notice we make sure the main graphics engine renders
        //to the lower lcd screen as it would be hard to draw if the
        //pixels did not show up directly beneath the pen
```

```
    lcdMainOnBottom();

    while(1)
    {
     scanKeys();

     if(keysHeld() & KEY_TOUCH)
     {
      // write the touchscreen coordinates in the touch variable
      touchRead(&touch);

      VRAM_A[touch.px + touch.py * 256] = rand();
     }
    }

    return 0;
   }
```

There is not too much to say about this demo…but of course I am going to say it anyway.

You should notice I skipped the interrupt setup for this demo. This is for two reasons….there is no real animation cycle and we only render one pixel at a time. This means even if we draw while the screen is rendering our pixels there wont be anything to tear.

Notice also the use of scanKeys(); we use the keysHeld() macro instead of the keysDown() because keysDown() would only return true the first time the pen touches while keysHeld() returns true until the pen is lifted up again. This allows us to draw our dots without lifting the pen.

When using the touchRead function, we need to pass in the pointer to the variable touch rather than the variable itself. Putting an ampersand, &, in front of a variable replaces it with the address to that data.

Color is selected at random using rand() –recall rand() returns a random 16 bit value which is convenient as color is also 16 bit.

When drawing rapidly you probably noticed big gaps between your dots as apposed to nice smooth curves. This is because the touch coordinates are only updated once per frame and you can move the pen a lot faster than that. We will make this demo a little prettier when we learn how to draw lines a few pages hence.

[edit] Bitmap Graphics Modes

I talked a bit before about the DS graphics modes and alluded to being able to compose a scene from layers of graphics. Normally all rendering is done to one of these layers and this section will be the first real use of the 2D engine. Before we talk about the specifics let us look again at the possible graphics modes and what each layer can do in these modes.

Graphics Modes

Main 2D Engine				
Mode	BG0	BG1	BG2	BG3
Mode 0	Text/3D	Text	Text	Text
Mode 1	Text/3D	Text	Text	Rotation
Mode 2	Text/3D	Text	Rotation	Rotation
Mode 3	Text/3D	Text	Text	Extended
Mode 4	Text/3D	Text	Rotation	Extended
Mode 5	Text/3D	Text	Extended	Extended
Mode 6	3D	-	Large Bitmap	-
Frame Buffer	Direct VRAM display as a bitmap			
Sub 2D Engine				
Mode	BG0	BG1	BG2	BG3
Mode 0	Text	Text	Text	Text
Mode 1	Text	Text	Text	Rotation
Mode 2	Text	Text	Rotation	Rotation
Mode 3	Text	Text	Text	Extended
Mode 4	Text	Text	Rotation	Extended
Mode 5	Text	Text	Extended	Extended

You will notice each engine has several modes of operation with each mode having different background layer configurations. The background configurations we are interested in this case are the ones marked “extended” graphics layer.

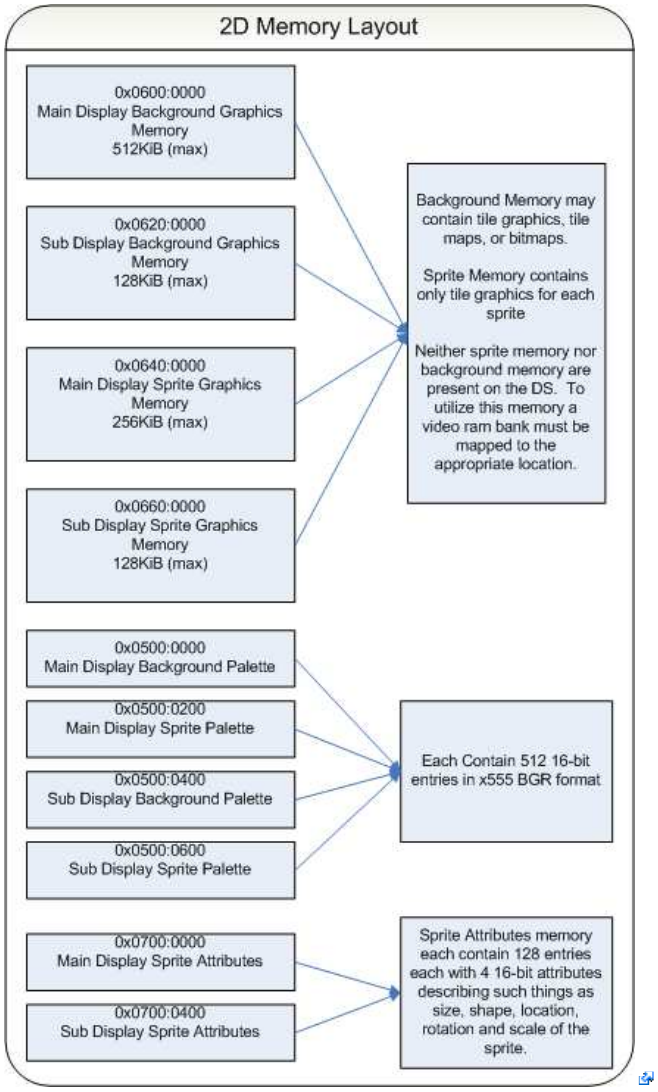
Extended rotation background layers can be configured as linear frame buffers, much like we have been using in the examples above.

When we talk tomorrow about tile based graphics we will cover in detail the capabilities of the “extended” backgrounds as well as the text and rotation backgrounds. For now we need only to understand a few things. First is how to put the display into a mode which supports an extended background, next is how to turn on the correct background, and finally we need to know how to

initialize the background properly.

The first thing to remember when using the 2D engine is the lack of memory available. In fact, the DS has no memory assigned to its 2D units by default other than Sprite attributes and base palettes.

In order for us to do anything we must map video memory somewhere the 2D engine can find it. To do this we must know where the engine is going to expect memory to be and we need to know what video memory can be mapped to these regions. What follows is the layout of 2D graphics memory.



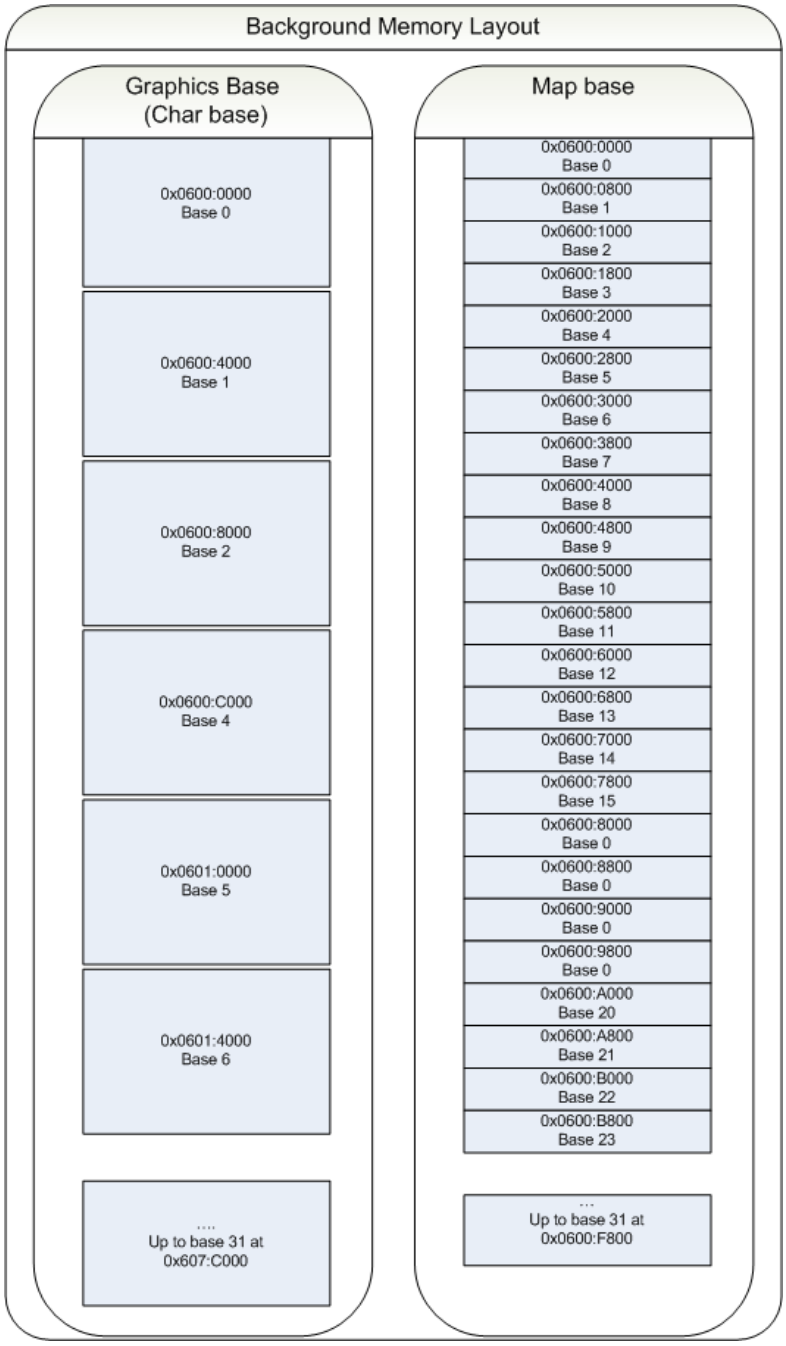
The areas we are concerned with now are the background graphics memories. To use background layers we must map video memory to at least one of these regions.

Let us start with a small example which paints the screen red and see how using a background layer differs from direct frame buffer access.

```
#include <nds.h>

//---------------------------------------------------------------------------------
int main(void) {
//---------------------------------------------------------------------------------

 int i;

 //set video mode to mode 5
 videoSetMode(MODE_5_2D);

 //map vram a to start of background graphics memory
 vramSetBankA(VRAM_A_MAIN_BG);

 //initialize the background
 bgInit(3, BgType_Bmp16, BgSize_B16_256x256, 0,0);

 //write the color red into the background
 for(i=0; i<256*256; i++){
   BG_GFX[i]=RGB15(0,31,0) | BIT(15);
 }
 while(1) {
  swiWaitForVBlank();
 }
 return 0;
}
```

To better understand the memory layout for 2d backgrounds, we must first look at the image below. It

is logically divided into 32 blocks of memory for graphics (also 32 blocks for map data but that is a topic for tomorrow). We will revisit this organization of graphics memory tomorrow but for today it is enough to know that the background will pull data starting at one of these blocks and which block it pulls from is controlled via the background control register.



First, we set the video mode. If you look back at the video mode table, you will realize mode 5 allows for background layers 2 and 3 to be extended rotation backgrounds. We chose background 3 for this demo although background 2 would have worked just as well.

```
//set video mode to mode 5
videoSetMode(MODE_5_2D);
```

Next we map vram bank A to main background memory. The vram table shows we could have mapped other vram banks to this region as well. Picking which vram bank to map takes a bit of planning, but for our simple demos it will not be too difficult.

```
vramSetBankA(VRAM_A_MAIN_BG);
```

Next, we initialize the background. bgInit follows the header from the nds/arm9/Background.h: bgInit(int layer, BgType type, BgSize size, int mapBase, int tileBase). In this demo, we wanted a bitmap mode and since we have not really discussed palettes yet we are going to stick with 16 bit color. We choose a size of 256x256 to fill the entire screen. Once this code is completed, BG_GFX will be set to the address of the initialized layer; in this case, it is layer 3.

```
//initialize the background
bgInit(3, BgType_Bmp16, BgSize_B16_256x256, 0,0);
```

The final step is to paint the screen red. Take a look at the following lines and see if you can note the difference between the straight framebuffer code.

```
 //paint the screen red
 for(i=0; i<256*256; i++){
  BG_GFX[i]=RGB15(0,31,0) | BIT(15);
 }
```

Hopefully you noticed the setting of bit 15 of the color value. Remembering back to a previous talk about color formats on the DS you might recall there is a 16-bit color mode with the normal 5 bits of red green and blue along with one bit for alpha. 16-bit bitmap backgrounds use this color format.

The one bit of alpha tells the DS to render that pixel. If you leave this bit clear the pixel will not be drawn and anything behind it will show through. This is a very useful feature when you have two layers of background and want part of the top layer to be transparent.

Next we will look at paletted bitmaps through the somewhat practical exercise of decoding graphics files.

[edit] Bitmap on the sub display

For completeness sake here is an example which puts both the main and sub engines in mode 5 and renders to bitmap backgrounds. Notice the alternative register access using the background struct, instead of the bgInit function.

```
#include <nds.h>

//---------------------------------------------------------------------------------
int main(void) {
//---------------------------------------------------------------------------------

 int i;

 //point our video buffer to the start of bitmap background video
 u16* video_buffer_main = (u16*)BG_BMP_RAM(0);
 u16* video_buffer_sub =  (u16*)BG_BMP_RAM_SUB(0);

 //set video mode to mode 5 with background 3 enabled
 videoSetMode(MODE_5_2D | DISPLAY_BG3_ACTIVE);
 videoSetModeSub(MODE_5_2D | DISPLAY_BG3_ACTIVE);

 //map vram a to start of main background graphics memory
 vramSetBankA(VRAM_A_MAIN_BG_0x06000000);
 vramSetBankC(VRAM_C_SUB_BG_0x06200000);

 //initialize the background
 BACKGROUND.control[3] = BG_BMP16_256x256 | BG_BMP_BASE(0);

 BACKGROUND.bg3_rotation.hdy = 0;
 BACKGROUND.bg3_rotation.hdx = 1 << 8;
 BACKGROUND.bg3_rotation.vdx = 0;
 BACKGROUND.bg3_rotation.vdy = 1 << 8;

 //initialize the sub background
 BACKGROUND_SUB.control[3] = BG_BMP16_256x256 | BG_BMP_BASE(0);

 BACKGROUND_SUB.bg3_rotation.hdy = 0;
 BACKGROUND_SUB.bg3_rotation.hdx = 1 << 8;
 BACKGROUND_SUB.bg3_rotation.vdx = 0;
 BACKGROUND_SUB.bg3_rotation.vdy = 1 << 8;

 //paint the main screen red
 for(i = 0; i < 256 * 256; i++)
  video_buffer_main[i] = RGB15(31,0,0) | BIT(15);

  //paint the sub screen blue
 for(i = 0; i < 256 * 256; i++)
  video_buffer_sub[i] = RGB15(0,0,31) | BIT(15);
 while(1) {
  swiWaitForVBlank();
 }
 return 0;
}
```

[edit] Background struct

libnds defines a struct for easy access of the background registers.

```
typedef struct {
 u16 x;
 u16 y;
} bg_scroll;

typedef struct {
    u16 hdx;
    u16 hdy;
    u16 vdx;
    u16 vdy;
```

```
    u32 dx;
    u32 dy;
} bg_transform;

typedef struct {
    u16 control[4];
    bg_scroll scroll[4];
    bg_transform bg2_rotation;
    bg_transform bg3_rotation;
} bg_attribute;

#define BACKGROUND          (*((bg_attribute *)0x04000008))
#define BACKGROUND_SUB      (*((bg_attribute *)0x04001008))
```

[edit] Working With Graphics Files

Being able to decode graphics files is a useful skill and although this is usually done on the PC we are going to do it directly on the DS just for kicks. There are a lot of different graphics files out there, and each file has advantages and disadvantages. For our purposes we need one that is easy to decode and is supported by many graphics applications. Some good choices would be: GIF, BMP and PCX.

I am going to tackle the BMP for this example. It is simple and supported by just about every graphics application on Earth.

To decode a file format you must first seek out its spec. A quick search on Google gives me the following information for bitmap files.

There is a short bitmap header followed by a variable length image header (this variable length header turns out to be 40 bytes in length almost always). Next comes the palette (if there is one) and finally the pixel data. Here is a bit more detail:

Bitmap File

Bitmap Header		
Offset	Size in bytes	Description
0	2	The characters "BM"
2	4	Filesize in bytes
6	4	Reserved (usually set to 0)
10	4	Offset to data
Image Header		
0	4	Size of image header (normally 40)
4	4	width of the image
8	4	Height of the image
12	2	Number of planes (normally 1)
14	2	Color depth (bits per pixel)
16	24	The rest is not interesting and hardly ever used
Palette Data		
The color palette stored as Red, Green, Blue bytes (with an extra byte of padding)		
Graphics Data		
The pixel data. This will either be indexes into the palette or raw Red, Green, Blue color data.		

BMPs support a number of bitmaps types and although this example will assume 8 bit 256 color bitmaps it could very easily be extended for other color depths (an excellent exercise for the reader if you are of a mind for those sorts of endeavors).

For a first step let us write a short demo which looks at the header, checks if it is bitmap file by reading the signature, and prints out the bits per color, height, and width.

Image:Nds day3 bmp show.png Bitmap Header Decode

We are going to use a trick called overlay to read the header. Instead of parsing each byte in and figuring it out we are going to define a structure which is the same size as the header. We can then pretend the start of the bitmap is the start of this structure and access all the attributes like normal structure members. Let's start with the bitmap header struct.

```
typedef struct
{
 char signature[2];
 unsigned int fileSize;
 unsigned int reserved;
 unsigned int offset;
}__attribute__ ((packed)) BmpHeader;
```

The idea is to look at the layout of the header and design a struct to match it. The first two characters are the signature so we add a character array of length 2 to line up with this. We do this for each element in the header.

Unfortunately for us the C language does not describe how exactly a compiler treats the memory assigned to a structure and often a compiler will pad a structure with empty bytes to make it a multiple of 32 bits in length. It does this because processing structs aligned so is generally more efficient. For us, we need the structures to be packed together so we can lay them on top of our bitmap data with no padding throwing us off. To ensure this is the case, we add the packed attribute to the structure definition…how you do this varies between compilers but for gcc this works like a charm.

Next we need a struct to hold the image header which we will just assume is 40 bytes, even though it could technically be variable.

```
typedef struct
{
 unsigned int headerSize;
 unsigned int width;
 unsigned int height;
 unsigned short planeCount;
 unsigned short bitDepth;
 unsigned int compression;
 unsigned int compressedImageSize;
 unsigned int horizontalResolution;
 unsigned int verticalResolution;
 unsigned int numColors;
 unsigned int importantColors;

}__attribute__ ((packed)) BmpImageInfo;
```

Notice again the use of the packed attribute.

Finally we define a structure to hold the entire bitmap and image header.

```
typedef struct
{
 unsigned char blue;
 unsigned char green;
 unsigned char red;
 unsigned char reserved;
}__attribute__ ((packed)) Rgb;

typedef struct
{
 BmpHeader header;
 BmpImageInfo info;
 Rgb colors[256];
 unsigned short image[1];
}__attribute__ ((packed)) BmpFile;
```

A bitmap file just consists of the two headers back to back followed by palette data and finally the image. This is where forgetting the packed attribute would byte you in the ass as gcc would stick in a few bytes of padding in-between the structs and things would not align (go ahead…try it).

The palette colors are stored as blue, green, red bytes followed by one empty byte. Since we are only going to decode 256 color bitmaps we are going to hardcode this into our bitmap structure. If you want to extend this you will need to do a small amount of parsing and first read in the bits per pixel and the number of colors before you do anything with the palette or image data.

The final entry in the bitmap file might seem a bit odd as it is an array of length one. Since we don't know how big the image array needs to be in advanced we give it a length of one, since it is an overlay we can just keep reading as far as we like.

Finally the code to decode the bitmap header:

```
//---------------------------------------------------------------------------------
int main(void) {
//---------------------------------------------------------------------------------

 BmpFile* bmp = (BmpFile*)beerguy_bin;

 consoleDemoInit();

 printf("%c%c\n", bmp->header.signature[0], bmp->header.signature[1]);
 printf("bit depth: %i\n", bmp->info.bitDepth);
 printf("width:     %i\n", bmp->info.width);
 printf("height:    %i\n", bmp->info.height);

 return 0;
}
```

This demo (if it were complete) would overlay our bitmap structure onto a bitmap file and print out the signature, bit depth and dimensions of the file. Hopefully, after all that discussion, there should only be one question remaining: How did I get a bitmap file into my DS application?

It turns out there are a lot of ways to get data into your application. You can use a file system and read it in from your compact flash or SD card. You can read it in from a wifi source like the internet or

a file share. You can run it through a converter and output the data as a big c array and compile it in or you can use object copy and create an object file you can link in.

The simplest way (thanks to wintermutes wonderful make file) is the object copy method. To include data in our project we just create a folder called "data" in the project folder (right next to source and include) and drop in a file. If we add a ".bin" to the end of the file name the make file will pick it up, run it through object copy, and create a header file with some ease of use variables declared.

For instance in the above demo I dropped in a bmp file called beerguy.bmp into the data folder. I then renamed it to beerguy.bin and typed make. A header file was created called beerguy_bin.h which contained the following:

```
extern const u8 beerguy_bin_end[];
extern const u8 beerguy_bin[];
extern const u32 beerguy_bin_size;
```

To access the data I just include the header file. I recommend you use this method as it works on any media and is simple. If you need file access because the 4MB limit is too much then the built in fatlib included with devkit pro is a great option.

I think we are finally ready to display this bitmap. All we have to do is copy the palette to the backgrounds palette memory and copy the image data to the backgrounds bitmap memory. Each of these steps have a small quirk.

The palette entries from the bmp file are in 8 bits per color and we need 5. To fix this we need to chop off the lower 3 bits of each element. If you remember our conversation on bit operations this is a simple matter of shifting the number to the right by 3. Copying in the palette is done as follows:

```
//copy the palette
for(i = 0; i < 256; i++)
{
Background_palette[i] = RGB15(bmp->colors[i].red >> 3, bmp->colors[i].green >> 3, bmp->colors[
}
```

The quirk for the image data is that the image is (for some strange reason) stored upside down. We have to flip the image by reading the bottom of the bmp into the top of the video buffer.

```
//copy the image
for(iy = 0; iy < bmp->info.height; iy++)
{
 for(ix = 0; ix < bmp->info.width / 2; ix++)
  video_buffer[iy * 128 + ix] = bmp->image[(bmp->info.height - 1 - iy) * ((bmp->info.width + 3
}
```

We do this by starting at line height minus 1 and subtracting the y index. Because video memory only accepts 16 or 32 bit writes we copy two bytes at a time (this is why you see the width divided by 2 in several locations and why video memory and bitmap image memory are declared as pointers to short instead of char).

Demo Source

Image:Nds day3 bmp decode.png Displaying the Bitmap

[edit] The Double Buffer

Double buffering is a common technique used to address a common problem with raster graphics. Usually you do not want the user to see what you are rendering until you are done rendering it. To ensure this happens you have one of two options.

First, you can only render during the time the display is not rendering—vblank and hblank. This method works well and in fact is an often used mode on the DS as the hardware does much of the rendering work for us. But, often you just need more time to get your scene in order.

This brings us to method number two: The double buffer. For this we simply render to an off screen buffer. When we are done we make the off screen buffer visible and whatever was visible before becomes our new off screen buffer. This gives us as much time as we need to compose the scene with the slight drawback of needing a copy of visible video memory. Fortunately the DS is more than roomy enough to accommodate.

There are actually several ways to go about implementing a double buffer but the easiest is to use a single background's video memory and allocate one half to the visible buffer and one half to the non visible buffer (the "back buffer").

To demonstrate the use of double buffers let us extend our bitmap demo and animate a series of 10 frames. A friend was kind enough to render me up some full screen graphics and for this first go we will do it without a double buffer.

```
#include <nds.h>

#include "beerguy0010_bmp.h"
```

```
#include "beerguy0020_bmp.h"
#include "beerguy0030_bmp.h"
#include "beerguy0040_bmp.h"
#include "beerguy0050_bmp.h"
#include "beerguy0060_bmp.h"
#include "beerguy0070_bmp.h"
#include "beerguy0080_bmp.h"
#include "beerguy0090_bmp.h"
#include "beerguy0001_bmp.h"


extern void DisplayBmp(unsigned short* video_buffer, unsigned short* pal, unsigned char* bmp_d


u8* bmps[10] =
{
 (u8*)beerguy0010_bmp,
 (u8*)beerguy0020_bmp,
 (u8*)beerguy0030_bmp,
 (u8*)beerguy0040_bmp,
 (u8*)beerguy0050_bmp,
 (u8*)beerguy0060_bmp,
 (u8*)beerguy0070_bmp,
 (u8*)beerguy0080_bmp,
 (u8*)beerguy0090_bmp,
 (u8*)beerguy0001_bmp
};

//---------------------------------------------------------------------------------
int main(void) {
//---------------------------------------------------------------------------------

 int frame = 0;

 BG3_CR = BG_BMP8_256x256 | BG_BMP_BASE(0);
 u16* video_buffer = (u16*)BG_BMP_RAM(0);

 irqInit();
 irqSet(IRQ_VBLANK, 0);

 videoSetMode(MODE_5_2D | DISPLAY_BG3_ACTIVE);
 //map vram a to start of background graphics memory
 vramSetBankA(VRAM_A_MAIN_BG_0x06000000);

 //initialize the background

 BG3_XDY = 0;
 BG3_XDX = 1 << 8;
 BG3_YDX = 0;
 BG3_YDY = 1 << 8;


 while(1)
 {
  DisplayBmp(video_buffer, BG_PALETTE, bmps[frame]);

  if(++frame > 9)
   frame = 0;

  swiWaitForVBlank();
 }

 return 0;
}
```

Compile and run this demo and you will see a bit of tearing and a lot of ugliness. The reason for this is my bitmap decode function is very slow and cannot complete in the vblank time frame.

The only thing different between this demo and the one before is the inclusion of 10 bitmap files which we cycle through once per vblank.

Let us use a double buffer and fix the problem.

```
#include <nds.h>

#include "beerguy0010_bmp.h"
#include "beerguy0020_bmp.h"
#include "beerguy0030_bmp.h"
#include "beerguy0040_bmp.h"
#include "beerguy0050_bmp.h"
#include "beerguy0060_bmp.h"
#include "beerguy0070_bmp.h"
#include "beerguy0080_bmp.h"
#include "beerguy0090_bmp.h"
#include "beerguy0001_bmp.h"


extern void DecodeBmp(unsigned short* video_buffer, unsigned short* pal, unsigned char* bmp_da


u8* bmps[10] =
{
 (u8*)beerguy0010_bmp,
 (u8*)beerguy0020_bmp,
 (u8*)beerguy0030_bmp,
 (u8*)beerguy0040_bmp,
 (u8*)beerguy0050_bmp,
```

```
  (u8*)beerguy0060_bmp,
  (u8*)beerguy0070_bmp,
  (u8*)beerguy0080_bmp,
  (u8*)beerguy0090_bmp,
  (u8*)beerguy0001_bmp
};

//---------------------------------------------------------------------------------
int main(void) {
//---------------------------------------------------------------------------------

 int frame = 0;
 int swaped = 1;
 int i = 0;

 unsigned short old_palette[256];

 BG3_CR = BG_BMP8_256x256 | BG_BMP_BASE(3); //<--changed to (3)
 u16* back_buffer = (u16*)BG_BMP_RAM(0);

 irqInit();
 irqSet(IRQ_VBLANK, 0);

 videoSetMode(MODE_5_2D | DISPLAY_BG3_ACTIVE);

 vramSetBankA(VRAM_A_MAIN_BG_0x06000000);

 BG3_XDY = 0;
 BG3_XDX = 1 << 8;
 BG3_YDX = 0;
 BG3_YDY = 1 << 8;



 while(1)
 {

  swiWaitForVBlank();

  if(swaped)
  {
   swaped = 0;
   BG3_CR = BG_BMP8_256x256 | BG_BMP_BASE(0);
   back_buffer = (u16*)BG_BMP_RAM(3);
  }
  else
  {
   swaped = 1;
   BG3_CR = BG_BMP8_256x256 | BG_BMP_BASE(3);
   back_buffer = (u16*)BG_BMP_RAM(0);
  }

  for(i = 0; i < 256; i++)
   BG_PALETTE[i] = old_palette[i];

  //decodes a bmp to the backbuffer
  DecodeBmp(back_buffer, old_palette, bmps[frame]);

  if(++frame > 9)
   frame = 0;

 }

 return 0;
}
```

In this demo we set up much as before only this time we set the starting point of background graphics to base 3. If you recall each base represents 16 KB of memory and if you are quick with math you will notice that a 256x192 image takes up exactly 3 blocks of memory. This means we set aside the first three blocks as visible and the next three as our render buffer.

```
BG3_CR = BG_BMP8_256x256 | BG_BMP_BASE(3);
u16* back_buffer = (u16*)BG_BMP_RAM(0);
```

We create a Boolean value which tracks which buffer is visible and alternate each frame. To swap the buffers we simply tell background 3 to render from one base and set the back buffer to the other base. This ensures we are always rendering to the off screen buffer and displaying the on screen buffer.

```
  if(swaped)
  {
   swaped = 0;
   BG3_CR = BG_BMP8_256x256 | BG_BMP_BASE(0);
   back_buffer = (u16*)BG_BMP_RAM(3);
  }
  else
  {
   swaped = 1;
   BG3_CR = BG_BMP8_256x256 | BG_BMP_BASE(3);
   back_buffer = (u16*)BG_BMP_RAM(0);
  }
```

The only other difference to note might seem a bit odd. It turns out that all my bitmaps use a separate palette so I must double buffer the palette as well! For this I use an in memory buffer.

The decode bitmap copies the palette to my local palette array (old_palette), and when I swap the visible buffer I also swap in this palette so the right palette is loaded with the right buffer.

```
for(i = 0; i < 256; i++)
  BG_PALETTE[i] = old_palette[i];

//decodes a bmp to the backbuffer
DecodeBmp(back_buffer, old_palette, bmps[frame]);
```

That is about all I am going to say on double buffers. They are useful and fairly simple to implement and can greatly enhance the look and feel of your program.

[edit] Raster 101

Raster graphics are the means by which most early games were rendered. It simply means to draw to a display on a per-pixel basis. We are going to let the DS hardware do most of our rendering for us, but there is something to be said for doing things the hard way every once in a while. We will cover line, circle, and polygon raster graphics and throw in a bit of optimization discussion along the way.

[edit] Bresenham Lines

We will begin our raster discussion with line drawing. The deceptively simple task of connecting two points on a 2D display by a series of pixels has been the subject of much research and countless papers. Perhaps we should start by defining a line.

In the majority of mathematical realms a line is an infinite, straight, one dimensional projection through space. To define a line all you need is the location of one point on that line and some indication of its direction.

Because one dimensionality is tough to achieve on a computer display and an infinite line might take too long to render we will have to restrict this definition a bit. For us all lines will lie on the plane of the screen, the length will definitely be finite, and that one dimensional thing will be very loosely applied. Let us take a close look at what a line on a computer display looks like to get a feel for what we need to accomplish.

<image of line here>

As you can see the line can only move in discrete steps of pixels. To render a line we just iterate through one dimension and plug the other into the equation for a line. Let us remember way back to geometry class and recal that a line can be defined as follows:

Y = mX + b;

Where Y is the axis we are trying to calculate, X is the axis we are iterating through, b is the value of Y when the line crosses the X axis and m is the slope of the line defined as change in Y divided by change in X (rise over run). I don't know about you but when I want to render a line I plan on just picking the two ends points and the color and having the algorithm do the rest. We can certainly calculate these values from two points but there is a slightly less common equation for a line that will be easier to work with:

Y – y = m (X – x)

In this case X Y and m are as before but x and y are the coordinates of any point on the line. As you might notice we don't have to worry about the intercept (previously 'b') in this form which simplifies things a bit.

So let us see if we can translate this equation into a line on the screen. First we will need to calculate the slope. Let us define our function as DrawLine(x1, y1, x2, y2, color) where the x's and y's are coordinates of the two points we wish to connect.

```
float m = (y1 - y2) / (x1 - x2)
```

One thing you should note right away is that slope will most likely be fractional in nature requiring us to use floating point math. You may also remember the DS has no floating point hardware and if you are particularly astute you will very shortly realize all this discussion is going to lead to some more interesting way of drawing a line.

Some pseudo code for drawing the line using this point slope equation would be as follows:

```
void DrawLine(float x1, float y1, float x2, float y2, short int color)
{
 Float m = (x1 - x2) / (y1 - y2);

 for( ; x1 < x2; x1++)
 {
  Float y = m * (x1 - x2) + y2;

  Buffer[x + y * BUFFER_WIDTH] = color;
}
}
```

I don't recommend trying to compile this as there are several flaws in the algorithm. First we are kind

of assuming the line changes more in x than it does in y, if that is not the case we are going to be jumping more than 1 pixel in y each time we iterate through the loop and leave large gaps in the line. Second we are assuming that x1 is smaller than x2 when really we could have specified any two points for the function. You could of course modify the above and use this method to draw lines on the DS but it is certainly not the best way to go about.

This brings us to a more realistic way of rendering lines. It involves only integer math, no division, and no multiplication.

Bressenham lines:

Let us again look at how a line ends up looking on screen and see if we cant find a way to iterate through X and change Y accordingly without calculating the slope. First look at a line that changes more in the X direction than it does in the Y:

<image of a small slope closup line>

From inspection you can probably note the slope on this line is 1 / 3. If we zoom in a bit we see this 1 / 3 slope holds as the line drops down 1 pixel in the Y direction every 3 pixels in the X.

To calculate the slope we would normally do something like this:

```
M = (y1 - y2) / (x1 - x2)
```

But instead let us treat the difference in y and the difference in x seperatly and call then ydiff and xdiff respectively.

For this case:

```
Xdiff = x1 - x2;
Ydiff = y1 - y2;
```

It might be helpful at this point to consider how we would draw the above line if we had infinite resolution.

We would first plot a pixel at x1, y1 then move one X to the right. If this were an infinite display we would also move 1/3 of a pixel down and plot the pixel. Because our real display is finite the best I can do is move one pixel in X and 0 in the Y. If I were to continue I would move another pixel in X and another 1/3 of the way down in Y. Again, because we have a finite display and 2/3 of a pixel is pretty meaningless we settle for no change in Y. Finally, as I move for the third time in X I reach a point where I should be a full pixel in Y further down and I can render at the correct location.

This process is the basis of Bressenham's algorithm.

We keep track of this difference between the line we SHOULD draw and the line we CAN draw in an error term; when that error reaches a threshold we know we can correct by adjusting our Y value by one pixel up or down (down in this case). But what is that threshold and how much do we adjust the error term each time? Well, the easy answer would be to use 1.0 as the threshold and add the slope to the error each time. If we did this things would move along smoothly…unfortunately calculating slope requires a floating point division and tracking the error term would require even more floating point operations. Fortunately there is a simpler way.

We store the difference between x1, and x2 (xdiff) as well as the difference between y1 and y2 (ydiff). These two values will be proportional to the numerator and denominator of the slope. For instance, in the line depicted above the coordinates are (0,10) and (30, 0). This amounts to an xdiff of 30 and a ydiff of 10 (and a slope of 1/3 incase you have forgotten). To render the line we iterate in the X direction (because it changes more than the Y direction) and keep track of how far the line we are drawing is from the line we would like to draw.

We can use a threshold of xdiff and increment the error term by the ydiff (or the other way around for a line that changes more in Y than it does in X). In this case we add 10 to our error term each time we move in X and when it reaches 30 we move one step down in Y. We then reset the error term by subtracting 30 and continue on (notice we don't set it to zero as in most cases the ydiff and xdiff will not align so well). If we continue we will draw a line at the correct slope from the two points.

```
threshold = xdiff;

for(x = x1; x < x2; x++)
{
 //increment the error term
error += ydiff;

 //if the error gets big enough we correct by moving down one
 //y increment
 if(error > threshold)
 {
  y++;
  error = error - threshold;
 }

 buffer[x + y * BUFFER_WIDTH] = color;
}
```

That is the basis of the bressenham algorithm. We just keep going one direction building up an error term until the error term reaches a certain point, then we correct by incrementing the other direction and reset the error term by the threshold.

Unfortunately there is a bit more to it. As before we only handle the case where the second point is

above and to the left of the first point and the line changes more in X than it does in Y. Fortunately handling these other cases turns out to be pretty easy.

Here is the final line algorithm, following will be a short explanation of the changes needed to make the code handle the other cases.

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

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

    //need to adjust if y1 > y2
    if (yDiff < 0)
    {
       yDiff = -yDiff;   //absolute value
       yStep = -yStep;   //step up instead of down
    }

    //same for x
    if (xDiff < 0)
    {
       xDiff = -xDiff;
       xStep = -xStep;
    }

    //case for changes more in X than in Y
    if (xDiff > yDiff)
    {
       for (i = 0; i < xDiff + 1; i++)
       {
          VRAM_A[offset] = color;

          offset += xStep;

          errorTerm += yDiff;

          if (errorTerm > xDiff)
          {
             errorTerm -= xDiff;
             offset += yStep;
          }
       }
    }//end if xdiff > ydiff
    //case for changes more in Y than in X
    else
    {
       for (i = 0; i < yDiff + 1; i++)
       {
          VRAM_A[offset] = color;

          offset += yStep;

          errorTerm += xDiff;

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

          }
       }
    }

}
```

Notice we broke the cases where change in Y is greater and change in X is greater so we could loop through either X or Y accordingly. Also if the x and y values for the points were opposite from expected we just take the absolute value of the difference and change the x and y step so we step the other way through the line.

Now that we have an okay understanding of line drawing let us modify our drawing demo from before to connect the pixels we were drawing with lines. This should fill the gaps and make a nice smooth line as we trace around.

```
int main(void)
{
 touchPosition touch;

 int oldX = 0;
 int oldY = 0;

 videoSetMode(MODE_FB0);
 vramSetBankA(VRAM_A_LCD);

 lcdMainOnBottom();

 while(1)
 {
  scanKeys();
```

```
    touchRead(&touch);

    if(!(keysDown() & KEY_TOUCH) && (keysHeld() & KEY_TOUCH))
    {
     DrawLine(oldX, oldY, touch.px, touch.py, rand());
    }

    oldX = touch.px;
    oldY = touch.py;

    swiWaitForVBlank();
   }

   return 0;
  }
```

We make an old X and Y value to hold the previous position, grab the new position, and draw a line between them. Finally we update the old x and y and repeat. This code only draws a line if the pen is down for more than two frames because we need two points to draw a line. This is done by not drawing the line if the keysDown() touch is set.

When putting the code together, do not forget to #include <nds.h> and either copy and paste the DrawLine function above the main function or #include a header file with it defined.

[edit] Bliting Things

```
(todo: add a section on software blitting...not that important all things considered)
```

[edit] Drawing Pictures

[edit] Polygons

```
(todo: some software 3D)
```

[edit] Circles

This code is in pascal, but some changes will turn it easily in c++ :

```
procedure DrawPixel(X,Y:Integer;Color:Integer);
var
i : Integer;
begin
  i:=X + Y * SCREEN_WIDTH;
 if Color=0 then
     VRAM_A[i] := RGB15(0,0,0)
 else if Color=1 then
        VRAM_A[i] := RGB15(31,31,31);
end;
```

To draw a Circle

```
procedure DrawCircle (Rayon,X_Centre,Y_Centre : Integer; Color : Integer);
        var
        x, y, m : Integer;
        begin
 x :=0 ;
        y := rayon ;        // Place on the top of the circle
        m := 5-4*rayon ;            // initialisation
        while x <= y  do begin    // while we are in the second half
                DrawPixel( x+x_centre, y+y_centre, Color ) ;
                DrawPixel( y+x_centre, x+y_centre, Color ) ;
                DrawPixel( -x+x_centre, y+y_centre, Color ) ;
                DrawPixel( -y+x_centre, x+y_centre, Color ) ;
                DrawPixel( x+x_centre, -y+y_centre, Color ) ;
                DrawPixel( y+x_centre, -x+y_centre, Color ) ;
                DrawPixel( -x+x_centre, -y+y_centre, Color ) ;
                DrawPixel( -y+x_centre, -x+y_centre, Color ) ;
```

Fill the circle or remove this 4 drawing lines if empty circle

```
LineBresenham( x+x_centre, y+y_centre, -x+x_centre, -y+y_centre, Color ) ;
LineBresenham( y+x_centre, x+y_centre, -y+x_centre, -x+y_centre, Color ) ;
LineBresenham( -x+x_centre, y+y_centre, x+x_centre, -y+y_centre, Color ) ;
LineBresenham( -y+x_centre, x+y_centre, y+x_centre, -x+y_centre, Color ) ;
```

Don't miss the end of the code

```
                if m > 0 then begin      //choix du point F
                        y := y - 1 ;
                        m := m-8*y ;
```

```
                end ;
                x := x+1 ;
                m := m + 8*x+4 ;
        end ;
end ;
```

Dev-Scene (c)