2d Stationary Sprites

Intro

If you are unfamiliar with the term sprites then you might have a really peculiar look on your face! They can be defined as moveable objects on a video game field. They can be a vial of health potion, a machine gun or even the monster holding the machine gun! Sprites have many applications and can apply to a wide variety of objects and characters.

Starting Out

We are first going to start out with normal 2 dimensional stationary sprites. Although the we define them as moveable objects, they don't really have to move, they can appear to move through a tricky use of animation. A very useful tutorial that goes along with this one is the one on Reading the Windows .BMP Graphic File Format. If you haven't read it it's here. We need to read this because we are going to assume that you know how to read a .BMP file so we can make some functions that load up animation frames from one.

A Brief Look at the Sprite Class

Lets make the .H file for our sprite class just so we have a rough idea of what we want our class to do. It isn't mandatory that all of these functions be encapsulated within a class, but I'm the teacher and I use C++ so that's what I'll use!!! It is also a really nice way of containing alike data. Here's a rough view of it!

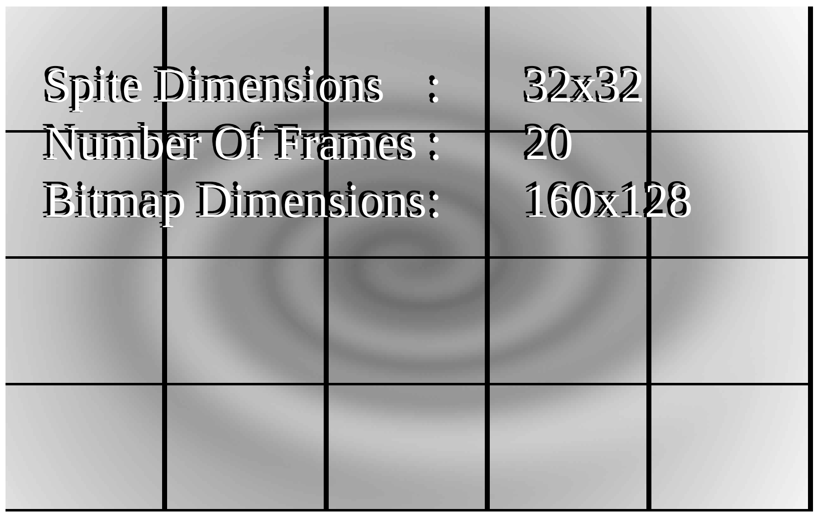
```
typedef struct frameptr
{ uchar* Image;
}Frame;

class Sprite_Container
{ public:
  Sprite_Container(Video*);
  ~Sprite_Container();
  void DetectDimensions(ulong,ulong,uchar*);
  void Load(ulong,ulong,ulong,uchar *);
  void Cut(uchar*,ulong,ulong,ulong);
  void Display(ulong,ulong,ulong);
  void ERROR(int);
  ulong NumberOfFrames,CellWidth,CellHeight,BitmapWidth;
  Video* video;
  Frame* frames;
};
```

I usually make all functions and data members public by default whenever I'm creating a new class simply because I'm not sure who needs access to what or who needs to communicate with what, so instead of dealing with a communication hassle, I'll just change things to private and protected later on when the class is more developed and utilized! We need a default constructor and destructor along with a function to load the bitmap, one to display it, a char pointer to hold the actual frames and two variables for the sprite's dimensions. These will create our foundation for a really cool sprite class :)

Creating the Sprite Frames

After you've decided what you want to animate, you must decide how many frames you want to use in the animation. This is a big issue for game developers because although more frames will look smoother, they will also take up a LOT of space over all. Use more frames on those animations that you consider vital and those that are going to be very visible. Open up your favorite graphics editing program and create a new bitmap who's width is the sprite width * the number of columns you want in the bitmap and who's height is sprite height * total number of frames/frames per row.

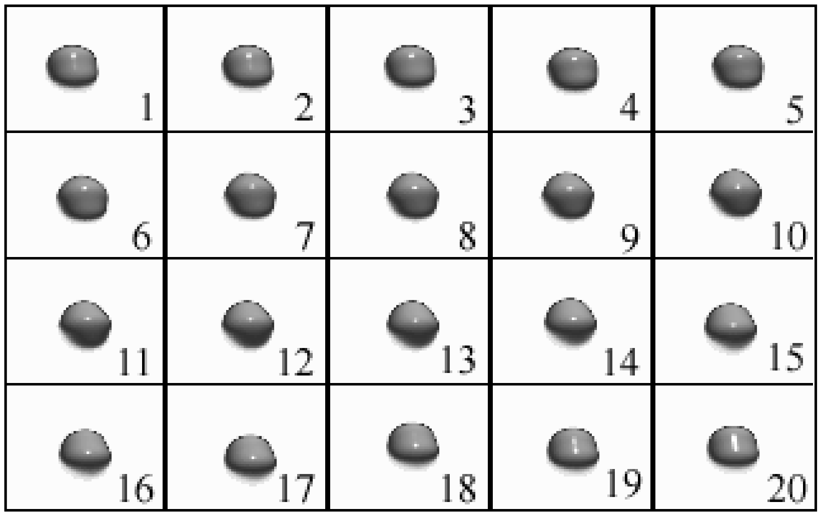


There, now that is what I mean. When you are creating the frames, also remember to line up each item within its own square, unless it's meant to move. Even when we want a sprite to move within its own square, we have to do it very, very carefully so that it doesn't seem to jump around. For instance to make our Mr. Blob sprite, we will have to center each frame exactly in the middle, otherwise we will get a jumping effect which makes a good animation horrible!

I suggest that you create a couple of test bitmaps to test your source code out before spending too much time on any sprite in any detail. I really like to create sprites in 3D Studio Max and export an .AVI file. Using another program I can convert the .AVI frames into .BMP files. I then inport the .BMP files into one master file from which I can do all of my loading.

Loading the Sprite Frames

In order to load our new sprite frames into our program, we only need to slightly modify our existing code that loads a .BMP file. We will actually be adding on a couple of functions that will rip the frames from the original bitmap file. I pulled in the individual graphics files that I was using and created this bitmap. They appear pretty close to being lined up, but I didn't spend too much time on it. I just wanted you to get a feel for what our master .BMP file is going to look like, I also added frame numbers to make it crystal clear! Also, when you are making your own sprite frames, don't waste any horizontal or vertical space, this will slow down our displaying routine and waste valuable ram and HD space!



What we really need now is a function that will take a single char array and have it spit out individual frames, each contained in its own char array. We will start by creating a function that will step through the accumulating vertical and horizontal values by stepping positions, with each step position being equal to the equivalent dimension of the frames. Vertical step = cellheight and Horizontal step = cellwidth. This is actually very simple as we will soon find out! Let's go ahead and define our Load function Mr.Blob in the making!

```
void Sprite_Container::Load(ulong bitmapwidth,ulong bitmapheight,ulong numframes,uchar* bmp)
{
 ulong CellsAcross,CellsDown,XPosition,YPosition,XCount,YCount;
 ulong FrameNumber=0;

 DetectDimensions(bitmapwidth,bitmapheight,bmp);
 frames = new Frame[numframes];
 if(frames ==NULL)
  {ERROR(3);
  }

 NumberOfFrames=numframes;

 CellsAcross=bitmapwidth/CellWidth;
 CellsDown  =bitmapheight/CellHeight;

for( YPosition=0,YCount=0;YCount<CellsDown;  YCount++,YPosition+=CellHeight)
{for(XPosition=0,XCount=0;XCount<CellsAcross;XCount++,XPosition+=CellWidth,FrameNumber++)
  {Cut(bmp,FrameNumber,XPosition,YPosition);
  }
 }
}
```

This might look confusing at first, but have no fear! We will prevail! We just want this function to move stepping variables through the bitmap and not do anything else! The Cut function actually does the cutting. First off we declare some variables that we are going to need. Please take a look and see that we are using a dynamically allocated array of structures that for now only have 1 item which is Image. I decided to encapsulate that in a structure so if we wanted to hold unique information for 1 frame we would already have the structures to do so. We call the DetectDimensions function giving it our bitmap pointer and its dimensions. That function is defined below. All we need to know is that after that function is called, the CellWidth, CellHeight, and BitmapWidth variables are set correctly. Knowing that we set the CellsAcross and CellsDown variables. This division can have a remainder although it gets truncated. This is nice when you need to pad the width and height to make them divisible by 4, which is what I have to do. We soon enter the big bad loop in which you should see a definite pattern. We use X and YPosition as our permanent variables that represent the beginning x and y coordinates of each cell. They are incremented by surprisingly enough the CellWidth and CellHeight variables that as you remember where set when we called DetectDimensions. We want each inner iteration to have a unique cell in our master list, so lets use FrameNumber and increment it so. Now we are stepping through the bitmap every cell calling the Cut function. Before we find out how that works let’s take a quick look at the DetectDimensions function!

```
void Sprite_Container::DetectDimensions(ulong bitmapwidth,ulong bitmapheight,uchar *bmp)
{//detect 1st white markers on top and left side
  uint xcount,ycount,flag=0;

  for(xcount=0;xcount<bitmapwidth&&!flag;xcount++)
   { if(bmp[xcount]==255)
      {flag=1;bmp[xcount]=0;
      }
   }
   flag=0;

  for(ycount=0;ycount<bitmapheight&&!flag;ycount++)
   { if(bmp[ycount*bitmapwidth]==255)
      {flag=1;bmp[ycount*bitmapwidth]=0;
      }
   }

 CellWidth=xcount;
 CellHeight=ycount;
 BitmapWidth=bitmapwidth;
}
```

Here we are just checking the top row for the 1st white pixel as well as the 1st white pixel on the left hand side. This allows us to plot 2 pixels in our master bitmap that represent the cellwidth and cellheight dimensions without having to stare at stupid grids all day! I really like this method because I never make any frames by hand, I use a rendering program that spits out the frames for me, I just have to stick them into 1 and line them up! This function also sets the to white pixels to black after it detects them so they won’t show up in the display. Without further delay, let’s take a look at the Cut function!

```
void Sprite_Container::Cut(uchar* bitmap,ulong framenumber,ulong x,ulong y)
{
 ulong height;
 uchar *bptr,*fptr;
 frames[framenumber].Image = new uchar[CellWidth*CellHeight+1];

 if(frames[framenumber].Image ==NULL)
  {ERROR(2);
  }
 fptr=frames[framenumber].Image;
 bptr=bitmap+((BitmapWidth*y)+x);

 for(height=0;height<CellHeight;height++,fptr+=CellWidth,bptr+=BitmapWidth)
  { memmove(fptr,bptr,CellWidth);
  }
}
```

Here’s where the magic takes place :) We declare 3 variables to keep track of our accumulating height in our inner loop, and 2 dummy pointers that will move through the bitmap. We finally allocate memory for our Image pointer that is contained inside a structure. We then initialize our dummy pointers, setting fptr to the array where the frame is going to be place and bptr a pointer somewhere inside our big bitmap. Next we enter our magical loop! Inside it advances the fptr(frame pointer) by CellWidth and advances bptr(Bitmap Pointer) by its width. We use the memmove function to copy exactly CellWidth number of pixels from out big bitmap to an array exactly 1 cell in size. Wasn’t that magical :) Now that we have this darn thing loaded up, what do we do with it next!?

Displaying the Sprite Frames

By now we have our master bitmap split up into individual frames that are elements of an array of structures. Now we have to create a function that we pass x,y coordinates with a frame number to display the frame onto the screen or offscreen buffer.

```
void Sprite_Container::Display(ulong x,ulong y,ulong frame)
{
  unsigned int ypos;
  uchar byte,*buff,*vbuff,*vbuffstep;
   buff=frames[frame].Image;
  vbuff=video->video_buffer+((video->Screen_Width*y)+x);

   for(ypos=0;ypos<CellHeight;ypos++,vbuff+=video->Screen_Width,buff+=CellWidth)
     { memmove(vbuff,buff,CellWidth);
     }

}
```

This function looks a lot like our Cut function doesn't it!? I should first note that we use some functions from my Video class which we were given access to through a pointer in our class constructor. The variables are pretty self explanatory: Screen_Width is the screen width, video_buffer is the char pointer to an offscreen buffer. We declare ypos which does the same thing as Height did in the cut function; It keeps track of the height. We declare our trio of dummy pointers buff for our bitmap (the size of 1 cell), vbuff for a pointer to video memory or in this case our offscreen buffer, and finally vbuffstep which will be moving top to bottom on the screen or in the buffer. Inside the loop we are doing the same thing as the Cut function did. We make sure that we are cycling exactly CellHeight times. Every iteration we move buff to the beginning of the next line in our little cell array and move vbuff to the next position down as well. One problem with this function is that it doesn't take into account transparency. The memove function will move every pixel from destination to its source without picking out pixels that don't belong. The alternative would be to have an inner loop that moves from left to right, checking to see that the pixel's color is greater than 0, otherwise don't plot it and move to the next position. We will run into this problem later on where we will show how much of a pain in the butt it is and how to really fix it! There's a final executable that shows it CAN be done at www.inversereality.org/files/sprite1.zip

Sprite Attributes

Our Sprite class, should we be using C++, can contain private variables that give the Sprite Object characteristics according to what it will be used for. If we really wanted, we could create several structures based on attributes of different objects. For instance, one structure could contain attributes for a living person, while another contained information on a plant. Here's what I mean :

```
struct persondata
{ unsigned int Life,Money;
  unsigned char Allignment,Class;
  BOOL UsesWeapons;

}HumanStruct;
```

Or for a Plant it could be

```
struct plantdata
{ unsigned int Color,Life,Flamable;
  BOOL HasIntelligence;
}PlantStruct;
```

Now if we really wanted to create some detailed characteristics of deferent sprite types, our structures could be great in not only number, but also size! The way to combat this, is to use structure pointers that have to be dynamically allocated, that way we won't be wasting space with variables we will never use!

Well now we can create a bitmap that holds frames of some object in motion and now we can correctly display it! This type of animation is great for objects that just move in their own little space like a medicine vial or a power-up. Thanks for taking time to read! If you have any questions, comments, rude remarks please give me some feedback!

I guess since I've got some spare space on this page, that maybe I should thank some people that helped me along the way.

Hecktarzuli	: Thanks for hosting my site and getting all my DNS and domain crap working! He has a gigantic MOD site with *hand picked* music. He's literally gone through thousands and kept only those that he liked, check it out! www.alteredperception.com!!
Intruder	: Thanks for listening to my compu-babble!
Heatseeker	: Thanks for the awesome music!! I would have gone insane if it wasn't for you!
Nutty	: Thanks for being a good internet bud, and for the help when I needed it!
Dad	: Thanks for our 1st 8088 computer and my 1st compiler TC++!!

Contact Information

I just wanted to mention that everything here is copyrighted, feel free to distribute this document to anyone you want, just don't modify it! You can get a hold of me through my website or direct email. Please feel free to email me about anything. I can't guarantee that I'll be of ANY help, but I'll sure give it a try :-)

Email : deltener@mindtremors.com
Webpage : http://www.inversereality.org