Pixel Manipulation

Contributed by Mark Thomas
Wednesday, February 07 2007
Last Updated Wednesday, February 07 2007

Textures in your game don't have to be pre-built in an image editing application such as Photoshop or The Gimp. You can directly manipulate pixel data within a Texture2D object using SetData and GetData. This tutorial takes a brief look at these methods.

Have you ever looked at a game and wondered just how all the textures were created? Games like Black and White feature shorelines effortlessly blending in with water, grass or rocks. It would be quite a job to create all those texture transitions. That's where procedurally generated textures come in. Procedural textures are generated in code, often with no source image data. This can be quite a chore, but the payoff can be huge in terms of time savings in asset creation, or just plain flexibility.

XNA can accomplish tricks like this as well. We won't be making Black and White XNA today, but we will take a look at how to directly manipulate textures in memory. You can use these techniques to accomplish all sorts of tricks with your textures.

Start by creating a new XNA game project. I called mine PixelManipulationTutorial. You can download the source at the end of this tutorial. We will be importing a single art asset, tilemap.png. It is included in the source download. It's just a simple graphic containing 2 48 x 48 textures:

Declare these variables so that they are accessible within our class: GraphicsDeviceManager graphics;
```
ContentManager content;
SpriteBatch spriteBatch;
Texture2D texTile;
Texture2D tileMap;
```

Replace the default LoadGraphicsContent() with this one: protected override void LoadGraphicsContent(bool loadAllContent)
```
{
 if (loadAllContent)
 {
  graphics.PreferredBackBufferFormat = SurfaceFormat.Color;
  graphics.PreferredBackBufferWidth = 144;
  graphics.PreferredBackBufferHeight = 48;
  graphics.IsFullScreen = false;

  //Lock Frame rate to vertical sync on monitor
  graphics.SynchronizeWithVerticalRetrace = false;

  graphics.PreferMultiSampling = true;

  //Commits the changes to the graphics Device
  graphics.ApplyChanges();

  tileMap = content.Load("tilemap");
  spriteBatch = new SpriteBatch(graphics.GraphicsDevice);

  //fill our dynamic texture with parts of 2 tiles.
  CreateDynamicTileTexture();

 }
}
```
Notice the call to CreateDynamicTileTexture(). We'll add that next.private void CreateDynamicTileTexture()
```
{
 //create a texture and fill it with some data
 texTile = new Texture2D(graphics.GraphicsDevice,
  48,
  48,
  0, ResourceUsage.Dynamic
```



```
 ,
 SurfaceFormat.Color, ResourceManagementMode.Manual);

//declare a uint array to hold the pixel data of our tilemap
uint[] tilesource = new uint[tileMap.Width * tileMap.Height];
//populate the array
tileMap.GetData(tilesource, 0, tileMap.Width * tileMap.Height);

//declare a uint array to hold the pixel data of our dynamic texture.
uint[] rgba = new uint[texTile.Width * texTile.Height];
//populate the array
//texTile.GetData(rgba); //this populates based on the current contents.... we dont want that
//we want to loop through our tilemap source array and grab desired elements.

for (int x = 0; x < texTile.Width; x++)
{
 for (int y = 0; y < texTile.Height / 2; y++)
 {
  //here we are grabbing the first 48 pixels on each row of the tilemap,
  //but only getting the top half of the texture.
  rgba[x + y * texTile.Width] = tilesource[x + y * tileMap.Width];
 }
}

//now we fill in the bottom half, laying down the second texture.
for (int x = 0; x < texTile.Width; x++)
{
 for (int y = texTile.Height / 2; y < texTile.Height; y++)
 {
  rgba[x + y * texTile.Width] = tilesource[(x + y * tileMap.Width + texTile.Width)];
 }
}

//apply grayscale to our texture.
//GrayScale(ref rgba);

//copy the source array into our dynamic texture....
texTile.SetData(rgba);
}
```

This function fills our Texture2D object by using its SetData() method. This method accepts an array of unsigned integers representing the pixel data in the texture. What this means is that if we can write code to manipulate the integers representing those pixels, we can create any texture we want! It's easier said than done though... Rather than just creating some integers and stuffing them in our texture, we are going to fill it with parts of two different textures. First we use the GetData() method of our tilemap Texture2D object. This populates an array of uint variables that we can access to get at the pixel information. At this point we loop through the first half of our dynamic texture, filling each pixel we encounter with pixel data from the first half of our source texture (the first tile only). This can get confusing, since ideally, we would want to think of our pixels in a two-dimensional array, much like the X Y coordinates on our screen. Unfortunately, that's not how it works. Our array can be thought of as one long row of integers. For example, if our texture is 100 pixels wide and 10 pixels high, our row is 1000 pixels long. The first integer represents the first pixel in the first row, and the 101st integer represents the first pixel in the second row. We use the width of our texture to keep things straight. After we fill the top half of our texture, we loop through the bottom half, filling it with the bottom half of our second 48 x 48 tile texture. Notice that we have a call to a function called GrayScale() commented out. We'll come back to that. Copy this code over the Draw() function: protected override void Draw(GameTime gameTime)

```
 {
  graphics.GraphicsDevice.Clear(Color.CornflowerBlue);

  spriteBatch.Begin();
  //draw our first tile
  spriteBatch.Draw(tileMap,new Rectangle(0,0,48,48),new Rectangle(0,0,48,48),  Color.White);
  //draw our second tile
 spriteBatch.Draw(tileMap, new Rectangle(48, 0, 48, 48), new Rectangle(48, 0, 48, 48),      Color.White);
  //draw our dynamic tile
  spriteBatch.Draw(texTile, new Rectangle(96, 0, 48, 48), Color.White);
  spriteBatch.End();
```



```
   base.Draw(gameTime);
  }
```

If you've worked with textures before, this should look pretty ordinary to you. We are simply using the Draw() method of our SpriteBatch object to output our textures onto the screen. If we run this now, we should see something like this:

Finally, we'll go a step further, turning our dynamic texture into a greyscale version of itself. Add the following function to your code:

```
    private void GrayScale(ref uint[] rgba)
  {
   for (int x = 0; x < texTile.Width; x++)
   {
    for (int y = 0; y < texTile.Height; y++)
    {
     uint f = rgba[x + y * texTile.Width];
     //bit shifting and other voodoo magiks are applied
     uint r = ((((uint)f) & 0x000000FF) << 16);//red
     if (r > 255) r = 255;
     if (r < 0) r = 0;
     uint g = ((((uint)f) & 0x000000FF) << 8);//green
     if (g > 255) g = 255;
     if (g < 0) g = 0;
     uint b = ((((uint)f) & 0x000000FF) << 0);//blue
     if (b > 255) b = 255;
     if (b < 0) b = 0;
     //uint a = (((uint)0xFF) << 24);//alpha, which I am throwing out.

     uint gray = (r + g + b) / 3;
     if (gray > 255) gray = 255;
     if (gray < 0) gray = 0;

     Microsoft.Xna.Framework.Graphics.Color color;

     //grayscale the pixel. this tends to wash things out.
     color = new Color((byte)gray, (byte)gray, (byte)gray);
     //put our pixel back in the array
     rgba[x + y * texTile.Width] = color.PackedValue;
    }
   }
  }
```

This function accepts the array of uint variables that holds our dynamic texture data. It loops through the array, using bitshift operators to access the individual red, green, and blue components that are ultimately being held by the integers. I won't attempt explaining this, because there are many places on the web that can explain it better than I can. I will explain the grayscale part though; basically, the way to produce gray is to take the average of the red, blue, and green components. To do this, once we have each part collected, we add them together and divide by three. Then we create a new Color object and populate it with this data, and finally shove the integer back in the array from whence it came. After uncommenting the call to this function in your CreateDynamicTileTexture() method, run the program. You should now see something like this:

I had intended to demonstrate some cool effects such as magnification, or randomly blending textures together. Unfortunately, time got away from me, but hopefully this gives you some ideas on how you can use similar techniques in your own games. Download the source for this tutorial here .