**What Color is that Pixel?**

When dealing with digital imagery, there is a lot of terminology that can confuse matters a great deal. Understanding a few key concepts though can make life a lot easier, and provide for more accurate and stable usage of image processing techniques.

This document describes a couple of key concepts related to digital image representation with pointers to more information.

There are two terms that need to be explicitly defined.

**Pixel** – A pixel is an in memory representation of color information.

Pixels are closely tied to a particular color space. I **Pixel layout** determines how many components there are, and how many bits are allocated per component. It also determines the exact memory locations of each of those bits in a single pixel. Familiar pixel representations might be ‘RGB32’, or ‘BGR24’.

In the case of RGB32, the designation is traditionally taken to mean that there are 3 color components in the pixel (R – Red, G-Green, B-Blue), and the data takes a total of 32 bits. It may be further implied by trial and error or actual definition that each of R, G, and B occupy 8 bits each, and there are 8 bits left over for no particular usage. This designation may also imply that the in memory representation is exactly as listed. That is, if we have bytes 0->3 in memory, they are occupied by the ‘R’, ‘G’, and ‘B’ components in that order. The 4th bit is allocated, but not occupied by anything in particular.

The BGR24 representation is similar, but the in memory layout is the opposite. The ‘B’ would come first, followed by the ‘G’ and finally the ‘R’. In this case, as there are only 24 bits, there is no extra byte to fill out to 32-bits.

Pixel layouts can come in many forms, and are tied to specific color spaces. Any number of bits can be allocated for each of the components, and they can be layed out in any order in memory, including having components interleaved.

**Color Space** – A color space is a descriptive mechanism used to describe a particular radio spectral phenomenon.

Although that definition is fairly accurate, it is a bit dry and leaves you scratching your head. In more practical terms, a color space is the way in which we describe colors. The most familiar color space for computer graphics is probably the RGB color space,that is, **R**ed, **G**reen, **B**lue. This particular color space matches the human vision system in that humans have what are known as ‘rods’ and ‘cones’ within the eye which are particularly sensitive to light values that are various intensities of Red, Green, and Blue. Thus, any color that can be seen by the human eye can probably be described by a mixture of Red, Green, and Blue.

Televisions, and computer monitors were constructed to follow this same system. In Cathode Ray Tubes (CRTs), there are phosphors coating the glass of the screen that correspond to the R, G, and B, and get excited variably by streams of electrons.

There are other uses for different color spaces though. One example is television. In the early days of television, there was the creation of YCrCb color space. This color space separates out the luminance (brightness) of the image (Y) from the Chrominance information (Cr, and Cb). This is convenient because when you want to display on a black and white television screen, having a separate color component that represents just that makes it a fairly easy task. Also, the human visual system is more sensitive to luminance than to color, so you can process them independently and have different sampling rates. But that’s a different story.

**Mixing and Matching**

The NewTOAPIA library contains a few interfaces, structures and classes that make it relatively easy to go between color spaces, and pixels. This is critical as in order to perform any image processing, you will often times have to switch between different domains, meaning from colors, to pixels, and between different color spaces.

The first and most important structure is the ColorRGBA. This is the basic structure that defines what it means to be a color. The structure is as follows:

public struct ColorRGBA

{

public static ColorRGBA Empty = new ColorRGBA();

public float r, g, b, a;

}

There are a few facts, assumptions, and limitations wrapped up in this particular representation of color information. First, let’s look at some of the facts:

Fact: This structure represents color from the RGB color model.

Fact: There are three components of actual color, and they are represented by single precision floating point numbers.

Fact: The structure takes up 16 bytes in memory. On many processors, including gaming consoles, 16-byte alignment is a good thing.

Now, for some assumptions and operational constraints

Assumption: Individual component values will always be in the range from 0.0 – 1.0. Negative numbers have no meaning, and numbers beyond 1.0 should be clamped to 1.0. So, the range for a single component of this color space is not the full range of numbers represented by float values.

There could be other choices made as to the full representation of this value. Each component could be represented by a single byte for example. This would make the storage of color information smaller, but it would severely limit the range of the base color system.

Assumption: All color models will be convertible to/from this data structure.

As there may be many different color spaces, there is an interface defined which allows for the conversion from one space to another:

public interface IColorModel

{

ColorRGBA GetRGB();

void SetRGB(ColorRGBA aColor);

}

Essentially what this interface says is that if you are to implement a specific color model, then it should inherit from this interface, and thus be able to transform itself to and from the RGB color model.

Colors are transient entities. They are used to assign pixel values, used in calculations, but are not typically stored en mass in large data structures. That’s where Pixels come in.

The next set of important interfaces are related to the storage of pixel information. First is the **IPixelInformaiton** interface:

public interface IPixelInformation

{

int BitsPerPixel { get; }

PixelLayout Layout { get; }

PixelComponentType ComponentType { get; }

}

This interface exists so that a pixel can be described. First is the number of bits that are used to represent the pixel. Next is the Layout. This is an enumeration that is currently limited. The layouts are:

Red, Green, Blue, Luminance, LuminanceAlpha, Rgb, Rgba, Bgr, and Bgra

There could be more layouts, like Yuv, and the like, but this is the current set. These are the most common in usage for basic image processing. If another layout is desired, it will have to be added to this enumeration.

Next is the **IPixel** interface:

public interface IPixel : IPixelInformation

{

ColorRGBA GetColor();

void SetColor(ColorRGBA color);

}

IPixel is meant to be implemented by any data structure that is capable of having its value set based on a ColorRGBA, as well as returning a ColorRGBA structure.

IPixel inherits from the IPixelInformation interface such that any pixel structure must also be self describing so that allocation decisions can be made at runtime.

To complete the story, there are typically higher level data structures and interfaces that allow you to treat pixels in a group. These are typically known as Bitmaps. Bitmaps are really limited to the black and white color space. Each ‘bit’ represents either black or white. The more general an accurate term would be PixelMap, or in our case, PixelArray. There is the **IPixelArray** interface that is used to define a data structure that holds a 2 dimensional array of pixels:

public interface IPixelArray

{

int BitsPerPixel { get; }

int BytesPerRow { get; }

PixelLayout Layout { get; }

PixelComponentType ComponentType { get; }

PixmapOrientation Orientation { get; }

int Width { get; }

int Height { get; }

IntPtr Pixels { get; }

IColorAccessor ColorAccessor { get; }

}

Most of these fields should be familiar or fairly self explanatory by now. There are two that need some explanation though. Orientation – this is whether the image is layed out top to bottom, or bottom to top. Pixels – This is a property that returns a ‘pointer’ to the actual pixel data when needed. For the most part, you will not need or want to access this property for pixel manipulation. If you’re familiar with C/C++ programming, it will seem tempting, because it is a direct pointer to the bits, but the better mechanism is to actually use the ColorAccessor property.

The ColorAccessor property gives access to an interface that has the ability to directly manipulate the pixels in the array. This is one of the interfaces that will be most commonly used to access the PixelArray using color information. This is probably not the fastest interface to use as you are translating between the color space and the pixel domain each time you set and get a pixel, but it is the most generic interface, and it is guaranteed to work on any Pixel Array, no matter what the pixel layout is, or what the individual pixels look like.

public interface IColorAccessor

{

void SetColor(int x, int y, ColorRGBA aColor);

ColorRGBA GetColor(int x, int y);

void PixBlt(IPixelArray pixArray, int x, int y);

void PixmapShardBlt(IPixelArray pixmap, Rectangle srcBoundary, Rectangle destinationRect);

void AlphaBlend(int x, int y, int width, int height,

IPixelArray pixArray, int srcX, int srcY, int srcWidth, int srcHeight,

byte alpha);

void FillWithColor(ColorRGBA aColor);

}

Having the ability to work with any layout of pixels, using a clearly defined color space is a great help in image processing. The best example is if you want to implement image enhancement filters, you can operate on the ColorAccessor level, dealing with the color values, and not worry at all about how those values are being represented in any given Pixel Array. As an example, the code for doing a ‘blur’ to an image can be applied to a grayscale image represented by floating point values, just as easily as with a ‘Red’ image that is represented with integer values. No special casing in the filter code.

The biggest drawback again is performance though as the translation goes to and from the color space for every pixel accessed. This may not be a bad thing if the processing would have performed such a transformation anyway.

Another benefit of using this interface is that the PixBlt functions are available here. PixBlt stands for Pixel Blit. A ‘Blt’ is nothing more than a “**Bl**ock **T**ransfer”. This is very old terminology from the earliest days of computer graphics.

This is very convenient for cases where you need to convert a PixelArray of a certain type to another in the most expedient fashion.

Now that we’ve gone through some of the most basic terminology, interfaces, and data structures, we will go through some of the concrete classes that pull this all together and make for a nice easy digital image processing base.

**Of Pixels and Bytes**

Let’s go back to pixel representations and examine a typical Pixel structure:

public interface IPixelBGR<T> : IPixel

where T : struct

{

T Red { get; set; }

T Green { get; set; }

T Blue { get; set; }

}

First, we want to be able to represent a pixel of the BGR layout, supporting many different base types. We want support for all the PixelComponent types. So, we create the IPixelBGR<T> generic interface which inherits from IPixel, and thus from IPixelInformation. This interface says that the pixel implementing this generic must return a Red, Green, and Blue component of the specified type. It must also allow the setting of the same.

Finally, we are set to create a specific pixel. In this case, we see the code for the BGRb pixel, wich is a BGR pixel using bytes as the individual component type. This would be a 24-bit data structure.

unsafe public struct BGRb : IPixelBGR<byte>

{

public static BGRb Empty = new BGRb();

fixed byte data[3];

public BGRb(byte red, byte green, byte blue)

{

fixed (byte\* dataPtr = data)

{

dataPtr[2] = red;

dataPtr[1] = green;

dataPtr[0] = blue;

}

}

public ColorRGBA GetColor()

{

return new ColorRGBA((float)Red / 0xff, (float)Green / 0xff, (float)Blue / 0xff);

}

public void SetColor(ColorRGBA aColor)

{

Red = (byte)Math.Floor((aColor.Red \* 0xff) + 0.5f);

Green = (byte)Math.Floor((aColor.Green \* 0xff) + 0.5f);

Blue = (byte)Math.Floor((aColor.Blue \* 0xff) + 0.5f);

}

public int BitsPerPixel { get { return sizeof(byte) \* 3; } }

public PixelLayout Layout { get { return PixelLayout.Bgr; } }

public PixelComponentType ComponentType

{ get { return PixelComponentType.Byte; } }

public byte Red

{

get {

fixed (byte\* dataPtr = data)

{return dataPtr[2]; }

}

set {fixed (byte\* dataPtr = data)

{dataPtr[2] = value; }

}

}

public byte Green…

public byte Blue…

}

There are a couple of things of note here. First, the data is stored in an interesting way where you see the ‘unsafe’ and ‘fixed’ keywords being used. This is not particularly relevant, and causes certain behaviors. It guarantees that the components are tightly packed in memory, but it also forces you to compile code using the ‘safe’ flag.

Besides that, we see that Pixel information is available, and the component properties are accessible. Now this pixel can be used anywhere a generic IPixel is needed and in some more specific cases where a IPixelBGR<byte> is needed.

There are more definitions for BGRs, BGRi, BGRf, BGRd. You could implement a BGR<mycomponenttype>, but you would have to be clear about the number of bits, the component type, and the other pixel information. This may not be useful in a generic case.

Now that we have a basic pixel representation, we are now finally ready to put it into a PixelArray. Some of the more basic stuff has been left out of this listing, but the most salient features are here.

unsafe public class PixelArray<T> : IPixelArray

where T : IPixel, new()

{

public PixelArray(int width, int height)

: this(width, height, PixmapOrientation.TopToBottom, new T())

{

}

public PixelArray(int width, int height, PixmapOrientation orientation, IPixelInformation pInfo)

{

fWidth = width;

fHeight = height;

fPixmapOrientation = orientation;

fNumberOfPixels = width \* height;

fPixelInformation = new PixelInformation(pInfo);

// Allocate the array of the specific pixel type

fTypedPixelArray = new T[fNumberOfPixels];

// Pin the array in mememory, and get a data pointer to it

IntPtr dataPtr;

unsafe

{

GCHandle dataHandle = GCHandle.Alloc(fTypedPixelArray, GCHandleType.Pinned);

dataPtr = (IntPtr)dataHandle.AddrOfPinnedObject();

}

Pixels = dataPtr;

}

public int CalculateOffset(int x, int y)

{

int offset = 0;

if (Orientation == PixmapOrientation.TopToBottom)

{

offset = x + (Width \* y);

}

else

{

offset = x + (Width \* (Height - 1 - y));

}

return offset;

}

public virtual void AssignPixel(int x, int y, T aPixel)

{

if ((x >= 0 && x < Width) && (y >= 0 && y < Height))

fTypedPixelArray[CalculateOffset(x, y)] = aPixel;

}

public virtual T RetrievePixel(int x, int y)

{

// If it's in range, retrieve the actual pixel

if ((x >= 0 && x < Width) && (y >= 0 && y < Height))

return fTypedPixelArray[CalculateOffset(x, y)];

else

return new T();

}

public T this[int index]

{

get

{

return fTypedPixelArray[index];

}

set

{

fTypedPixelArray[index] = value;

}

}

}

As a generic class, this object accepts a pixel type, but it must conform to the IPixel interface. Furthermore, it must have a default constructor. An array of the appropriate size is allocated. The elements of this array are of the Type specified for the generic, ‘T’.

There are two very important functions beyond the constructors. AssignPixel, and RetrievePixel. AssignPixel is will take the pixel value, and assign it to the right place in the pixel array. RetrievePixel does the same. At the moment, neither of them will throw an exception. AssignPixel will fail silently, and RetrievePixel will give the default pixel value.

These routines are the fastest access to the data. In most cases they will perform faster than direct pointer manipulation as they don’t have to go through “fixed” or GC Handle creation.

Subclasses can implement more specific forms of these accessors if they deal with their own memory allocation. This may be the case where you have a GLTexture object, who’s memory is actually resident on the GPU, and you have specialized access mechanisms.

**Summary**

Digital image representation is the bread and butter of many computer applications. Understanding the difference between color spaces and pixels is a critical part of being able to create performant and stable digital imaging applications. This document has given an overview of the basic interfaces and structures which are at the heart of the NewTOAPIA imaging capabilities. We have clearly defined the differences between colors, and pixels, and how the two are related to each other.

Given the support provided, exchanging images, converting between color spaces and pixel representations become fairly straight forward tasks. More extensive operations and data structures are formed upon this base. This forms the basis for operations ranging from video image display, to graphics rendering and 3D interaction.

**References**

<http://lurkertech.com/lg/packings/index.html>

<http://codebox/TOAPIToppers>