Fast Color Threshold Calculation

If you want to allow certain colors in a picture to be visible, and others to turn black, you can use color thresholding. This technique sets two thresholds (borders) around a picked color, to describe a range, within which the colors are shown. Everything outside of this range is turned to black.

This sounds easy, but colors are described by multiple components. The most common is RGB (Red, Green, Blue).

How do we set a threshold for three values?

Range

The most basic way is to see if each component of a pixel color is within the given range. A color would be within range if

$$R > threshold_{lower}$$
$$R < threshold_{upper}$$
$$G > threshold_{lower}$$
$$G < threshold_{upper}$$
$$B > threshold_{lower}$$
$$B < threshold_{upper}$$

As you can see this is quite an expensive computation, if you keep in mind that this must be calculated for every pixel.
This approach has another disadvantage. Imagine the full range of colors formed by RGB as a three-dimensional graph. The red values are placed along the first axis, the green values along the second axis, and the blue values along the third axis. If we pick a color, and we specify a range that will be the same for each component, for each value, we will get a cube as range around it.
Now we take a color with the lowest values possible, such that it is just within the range, in one of the corners of our cube, and name it color a.
We take a second color, named b, and pick it from the center of one of the edges of the cube. This value is also just in range, but physically, it's closer to the center color than color a is.

Radius

To create a true range of colors, in which each color that is just within this range, is equally distant from the center, you will need a sphere around the picked color. With a sphere, it's easier to speak of radius in stead of range. Every color within this radius is visible.
But how do you know if a color is within a certain radius?

We have to calculate the distance from the center, and compare this with the radius. As you can see, we have reduced the comparisons from six to only one. But the calculation of this distance is a bit more extended:

$$\sqrt{(R_c - R_n)^2 + (G_c - G_n)^2 + (B_c - B_n)^2} < r$$

Where c is the center color, n is the color to be evaluated, and r is the radius.

As this article is actually about efficient programming, let us look at this equation from a programmatic perspective. R, G and B of the evaluated color are variable, since we evaluate a lot of colors (pixels in an image). On the other hand, r is fixed, and nothing happens with it. To reduce the number of computations, we transform this equation:

$$(R_c - R_n)^2 + (G_c - G_n)^2 + (B_c - B_n)^2 < r^2$$
$$r_{squared} = r^2$$
$$(R_c - R_n)^2 + (G_c - G_n)^2 + (B_c - B_n)^2 < r_{squared}$$

We can square the radius outside of the loop that evaluates every pixel color, and with that we can remove the square root.

This is a fast method, but it can be even faster.

Table Lookup

A quick way to see if a value is valid, is a lookup table. Just give an index, and you get a value. If we make a table with cells for each color, and in each cell we describe if the color is visible, we just have to compute an index.
One disadvantage though, because modern images can contain more than 16 million colors (3 components with 8 bit range). Luckily we don't see that much difference, so it won't harm us if we decrease the color range to not more than a few hundred thousand (6 bit range).

$$2^{(3\times 6bit)} = 2^{18}$$

We cut off the lowest bits of each color, because these describe minor differences:

```
111111XX
```

If we pick a color, we don't have to calculate the visibility for every color. Just pick the cube from the first approach, reaching from *-r* to *+r* for each component, and calculate for every color within it.

Index

To get the index of a color, we have to create a unique number for every color.
With our color range of 18 bit, we have that amount of unique indexes. Let's look at the bits of the highest index:

```
111111 111111 111111
```

We have to get each color component in 6 of these bits:

```
RRRRRR GGGGGG BBBBBB
```

The range of each component is now:

R	4096	262143
G	64	4095
B	0	63

Let's start with the red component:

```
000000 000011 1111XX
```

The two unused bits are still present, but that does not matter. Now how do we get these bits to the 6 most left bits?
If you know something about bits, you know that the highest bit of our component is exactly 1024 times smaller than the highest bit in the whole range.

```
000000 000010 000000 = 128
100000 000000 000000 = 131072
```

Shifted 10 bits to the left, a value is 1024 times (2^{10}) higher.
So we only have to multiply the red component with 1024, or we can use a thing called bitshift, an operator present in most modern programming languages. To shift a value 10 bits to the left, you can do

```
R << 10;
```

Now the red component is in place.

```
RRRRRR XX0000 000000
```

The green component must only be shifted 4 bits to the left. This will then overlap the two unused bits of the red component, and leave two unused green bits to be overlapped by the blue component.

```
RRRRRR GGGGGG XX0000
```

The blue component is more to the left than wanted, so we have to shift it to the right. 2 bits to the right equals dividing by 4.

```
B >> 2;
```

Finally, add these three outcomes together.
So what do we have now? An index for the lookup table, but not a unique index for every color. This technique maps 24bit colors to 18bit colors, meaning that it maps 4 colors in one index. (4 is the two bits we didn't use.)

Storage

What values must the table store? Actually only true or false, a color is visible or not. We need a bit.
Unfortunately, most programming languages do not support a type that's just one bit. The smallest type is one byte, that is 8 bits. No problem, because we were already doing magic with bits. Assuming we use a type of just one byte, and we will store 8 values in each byte, our table size must be

$$\frac{2^{18}}{8} = 2^{15}$$

times the size of a byte. That will be 32 kilobyte.
We now have an index range of 15 bit, and per index another 8 indexes. Remember that with 3 bits we can make 8 different values.
With the 18 bits we have, we can find these indexes, but we have to cut it in two parts, one for the Table index (15bit), and one for the Byte index (3bit):

```
TTTTTT TTTTTT TTTBBB
```

We can get the Table index just by shifting the total 18 bits three bits to the right, so that the Byte index will fall off.
To get the Byte index, we have to get rid of the other 15 bits. You can best do this by masking the index with another value. Every bit that is present in the index as well as in the masking value, gets through:

```
0 & 0 = 0
0 & 1 = 0
1 & 0 = 0
1 & 1 = 1
```

As you can see, this operator is displayed in programming languages as & (Logic AND).
We need to have the last three bits, so these have to be masked with

```
000000 000000 000111 = 7
```

```
ByteIndex = Index & 7;
```

If our Byte index is, for example 5, we have to switch the fifth bit in our byte to 1.
This can be done with another operator, the OR operator, displayed as | . To start easy, take a byte with value 1, and an empty storage byte.

```
0 0 0 0 0 0 0 1
0 0 0 0 0 0 0 0
```

If one of the bits is 1, the output will be 1.
To directly put it in our storage byte, we can do one of the following:

```
Byte = Byte | 1;
Byte |= 1;
```

Both do the same. But this always sets the bit to 1. If you want to switch it, that is, if the value is already 1, than set it to 0, you can use the Exclusive OR, displayed as ^ .

Now if we have to switch the fifth bit, we have to shift the 1 in our byte 4 bits to the right:

```
0 0 0 0 0 0 0 1 : first bit
0 0 0 1 0 0 0 0 : fifth bit
```

That means that we need to decrement our Byte index with 1. Then we have to shift a byte with value 1 to the right this much bits, and then OR it with our storage Byte:

```
Byte |= 1 << (ByteIndex - 1);
```

And we are finished. We can actually store the visibility information of 32 colors in one single byte.

Read out the table

The operations we used are directly manipulating the bits, and therefore much faster than other programmatic functions. We still have to use a few of these to read out the table, but they are not computational expensive, so that does not matter.

The readout is almost the same as the writing of the table. The index of the storage byte is retrieved the same way:

```
Index = (R<<10) + (G<<4) + (B>>2);
```

Getting the byte index and then the bit as described before, but then we have to see if this bit is true or false.
Take the storage byte, and make a byte with just one bit (the bit we want to get) set to 1, as we did with the OR operation. Now we do not use OR, but AND, to see if this bit is set to 1:

```
Byte & (1 << (ByteIndex - 1));
```

What happens if the bit is true, is:

```
0 1 0 0 1 0 1 1 : storage
0 0 0 0 1 0 0 0 : AND with byte
0 0 0 0 1 0 0 0 : outcome
```

So if the outcome is not 0 (is *true*), the color is visible.

I used this technique for realtime video, and it works well. It isn't said though, that this technique works best, or as well on other kinds of image data. Fact is, that by trying to find the most effective way to do color thresholding, I learned a great deal about working with bits and bytes.

Some people who helped me create this technique (thank you):

Pieter Suurmond
Lecturer and Software Developer at the Utrecht School of the Arts

Simon Asselbergs
Student at the Utrecht School of the Arts

February 8 / 9, 2005
Toine Heuvelmans
Student Audio Design
Utrecht School of the Arts

This article was written during my STEIM internship.