Visualizing Change with Colour Gradient Approximations

by Ben Kenwright[1]

What is the problem?

Typically we have a scalar value ranging from a minimum to a maximum; these values are usually scaled to between 0 and 1 to make working with them more manageable. The values can represent a variety of possibilities, for example:

- Temperature change; showing the coldest to hottest temperature fluctuation (blue to red)
- Joint torque; showing the minimum and maximum torque being applied to a joint
- Height; showing the lowest and highest points on a map
- Flow; showing the fastest and slowest velocities

Colour gradients have been used for numerous years, and have been employed in numerous CAD packages to help identify 'hot spots'. But little focus has every been put on how to generate various colour patters, why we use particular patters and how they can be used in our everyday tools to emphasize details.

We use gradients instead of sharp on/off levels to help visualize the changing data over time.

What are the solutions?

While it is possible to use a single colour to visualize the range (e.g. a gradient from black to white) , it will be shown that using more colours helps emphasize more detail, and helps to emphasis problems at a glance. The main focus points are:

- Justify the advantages and disadvantages of colour gradients
- Give Examples of visual impact (Linear vs Non-Linear Visualization)
- Outline any approximations we can use for computational speed-ups

Single Colour Linear Blending

[1] *bkenwright@xbdev.net*

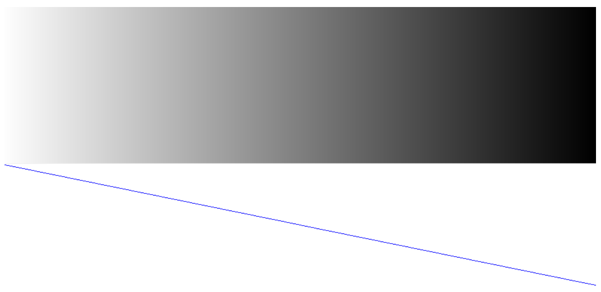

Figure 1. Linear interpolation single colour between white and black

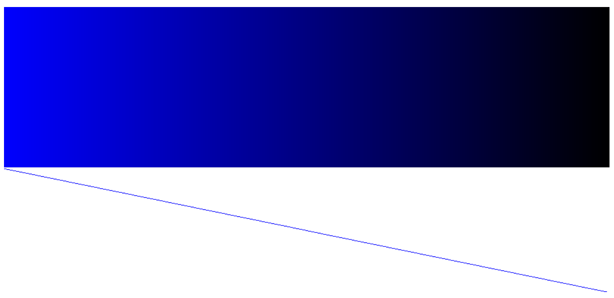

Figure 2. Linear interpolation between a single colour blue.

There are many downsides to a single colour gradient. Initially, you'll notice that t is difficult to identify the middle, when the gradient is 50%, you may also notice that you have no feel for what value each colour gradient has, for example if the colour was mapped onto a shape or surface, it would be difficult to identify which parts are closer to 1 or closer to 0. Then there is the issue of similar points with similar values, it's difficult to compare and contrast, since our eyes don't distinguish between brightness compare to differences between unique colours. And finally, if you need to point out a particular area, or problem, you can't just say the dark blue, or black sections, it would be easier if you could identify them more precisely.

Next we'll move onto using multiple colours to try and fix some of the problems we have identified with a single colour interpolation. The colours black and white are a combination of 'all' and 'none' of the other three primary colours (red, green and blue), hence we will avoid using them in our results, since our aim it to help distinguish particular values in the output.

Two Colour Linear Blending

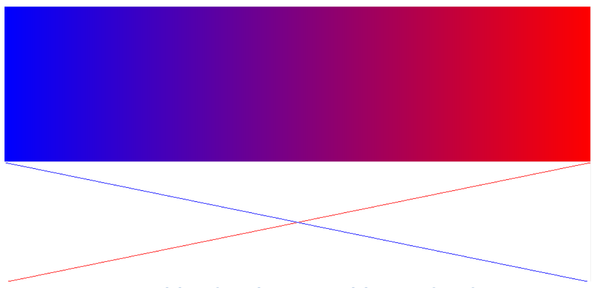

Figure 3. Linear blending between blue and red.

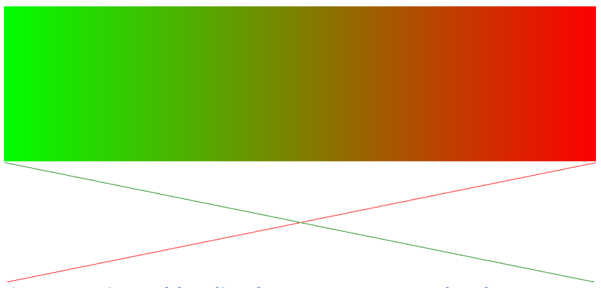

Figure 4. Linear blending between green and red

The results are an improvement over single colour linear blending, and enable us to identify 'hot spots' which might be of interest. Again its easier to point out red or blue areas, but it still leaves an area of ambiguity when describing values, for example in the blue to red blending, if you wanted to point of a value between 0.25, what would you say? Would you say, "A darkish blue with a hint of purple?" Or if you had to distinguish between two colours of similar value, maybe 0.25 and 0.3, it would be almost impossible, unless they where right next to each other, and monitor displays are never the best, so distinguishing between small colour changes doesn't always stand out.

Three Colours

Three colours should allow us to clearly identify unique region without ambiguity. For example, if I where to say the colour, red, green, blue, yellow or purple, it should be very easy to locate it. The challenge is to try and the spread out the three primary colours to make it possible to visualize 'hot spots' at a glance.

Sine's and Cosine's

The first approach is to take advantage of trigonometric functions (Sin and Cos) and see how they perform at giving us a good colour mix.

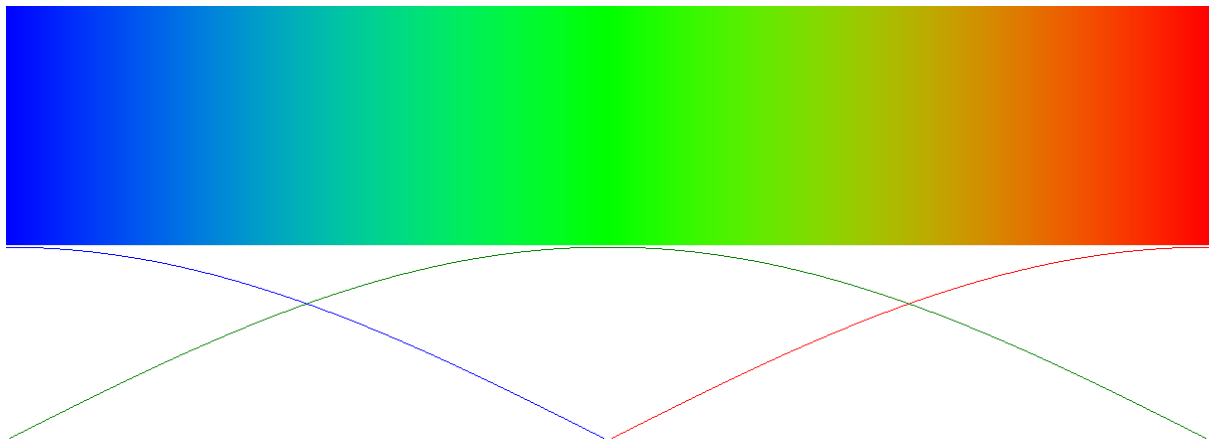

Figure 5. Sine and Cosine to generate the colour gradient.

First glance you might think, "hey, this looks good", but after careful scrutiny, you start to notice problems with it. You can identify the three primary colours, at 0, 0.5 and 1, which are blue, green and red, but what about the secondary colours, such as yellow, orange, aqua? Most people say, they can 'sort' of see them, but that's not good enough.

Linear

As an alternative to Sine and Cosine we can try and generate the blending using simple linear approximations.

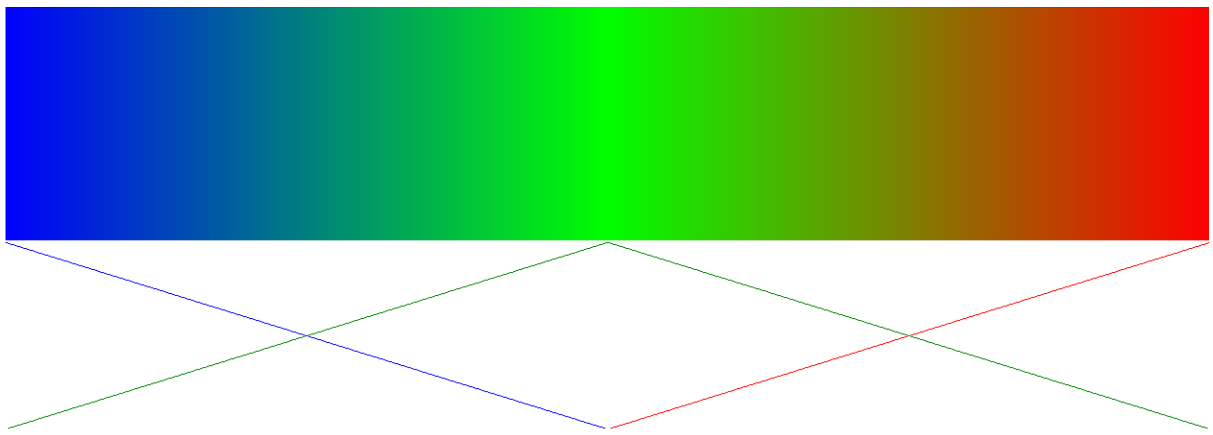

Figure 6. Linear blending between the three primary colours.

The linear method is computationally less expensive than the sine and cosine version, since we are only need to calculate a gradient. If you compare the output from both the linear and sine-cosine version the only noticeable difference is the slight loss of smoothness during transitions.

Neither result gives us what we are looking for, a blend between the various colours with identifiable sections.

Linear Stepping

Keeping specific colours fixed for sections of time, we can help produce more colour sections.

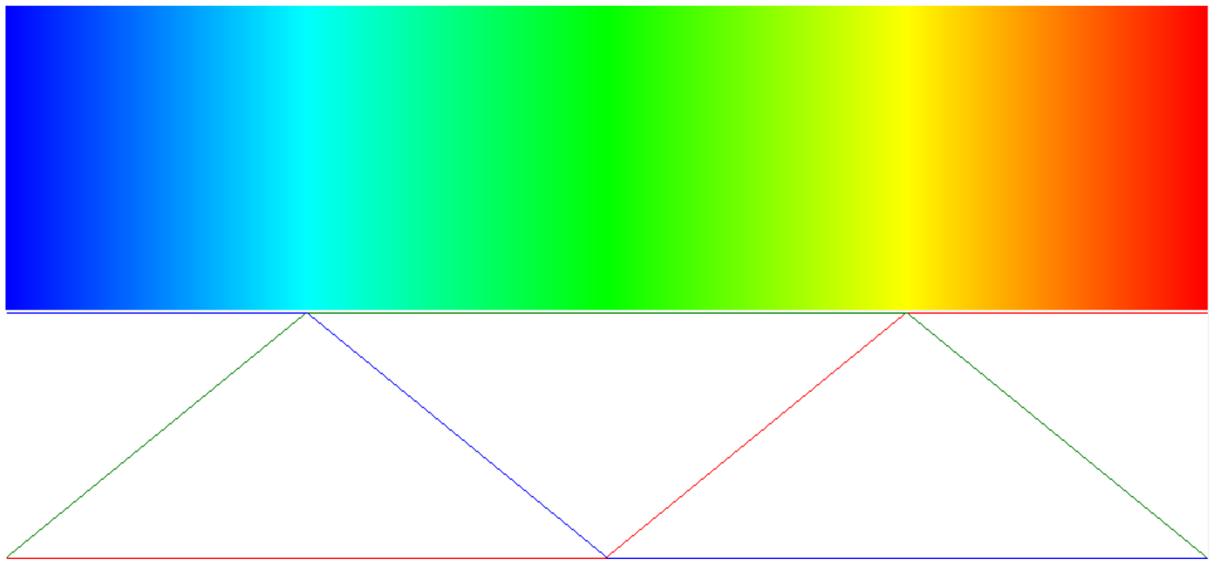

Figure 7. Keeping colours fixed for half the range, then linearly interpolating.

Instead of just interpolated between 1 and 0 over the specified colour range, this time we fix the magnitude to the maximum for half the time and double its gradient decrease down to zero. This has the added effect of making the different regions more prominent. As you can see from the figure above, the colours aqua and yellow are now more identifiable.

Non-Linear

What we have is good, but by introducing some linear and non-linear scaling with clamping, we can emphasize the various regions further.

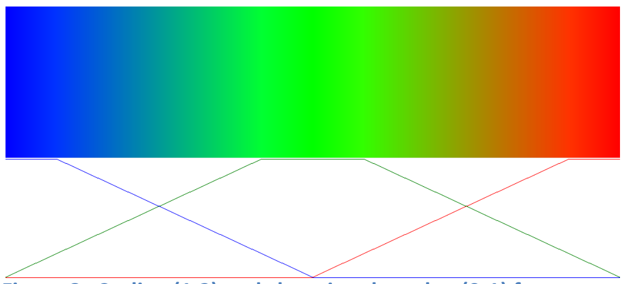

Figure 8. Scaling (1.2) and clamping the value (0-1) for to the linear-rgb model.

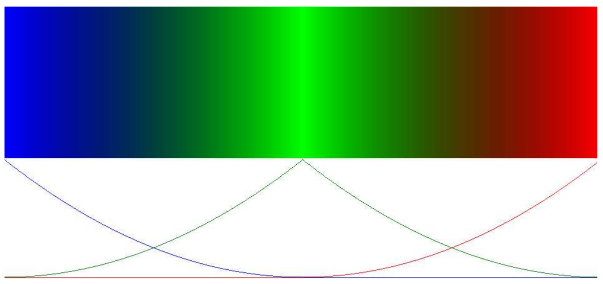

Figure 9. Squaring the result and clamping the value (0-1) for the linear-rgb model

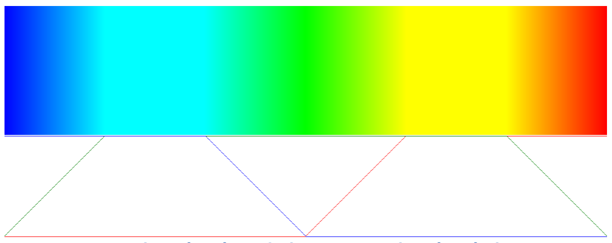
Figure 10. Scaling (1.5) and clamping value (0-1) the linear-rgb-stepping model.

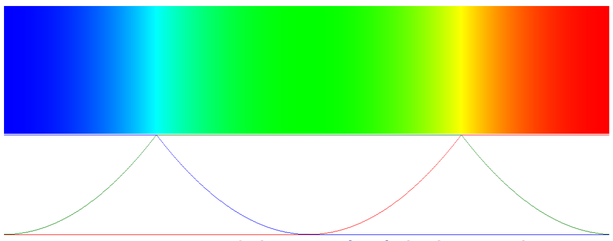
Figure 11. Squaring and clamping (0-1) the linear-rgb-stepping model.

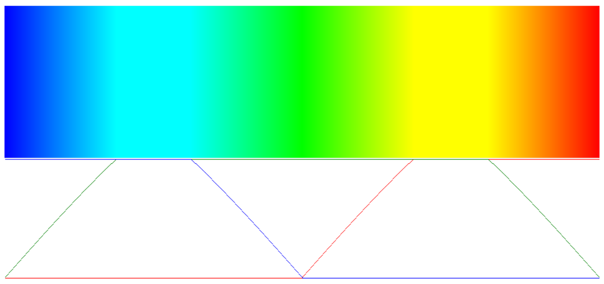
Figure 12. Scaling (1.8) and clamping (0-1) the sin-cos model.

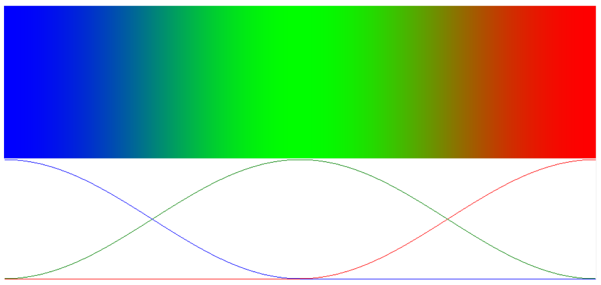
Figure 13. Squaring and clamping (0-1) the sin-cos model.

Conclusion

We have demonstrated three main methods for generating the colour information to illustrate information. After comparing and contrasting each method, the overall best method was found to be the linear stepping model with scaling, offering a very computationally algorithm and producing a very clear set of gradient colours.

References

weblink :
http://www.xbdev.net/misc_demos/demos/colour_heat_gradient_blue_to_red/index.html

Appendix

Code Fragments

```
// *** Linear 'single colour' blending blue-black
// t : 0 to 1
// returns argb colour
Colour GetColour(float t)
{
       float r = 0;
       float g = 0;
       float b = 1 - t;
       return Color.FromArgb(255, (int)(r*255.0f), (int)(g*255.0f), (int)(b*255.0f) );
}

// *** Linear 'two colour' blending blue-red
// t : 0 to 1
// returns argb colour
Colour GetColour(float t)
{
       float b = 1 - t;
       float r = t;
       float g = 0;
       return Color.FromArgb(255, (int)(r*255.0f), (int)(g*255.0f), (int)(b*255.0f) );
}

// *** Trig 'three colour' blending blue-green-red
// t : 0 to 1
// returns argb colour
Colour GetColour(float t)
{
       b =  (float)Math.Cos( Math.PI * t );
       r = -(float)Math.Cos( Math.PI * t );
       g =  (float)Math.Sin( Math.PI * t );

       r = MathHelper.Clamp(r, 0.0f, 1.0f);
       g = MathHelper.Clamp(g, 0.0f, 1.0f);
       b = MathHelper.Clamp(b, 0.0f, 1.0f);

       return Color.FromArgb(255, (int)(r*255.0f), (int)(g*255.0f), (int)(b*255.0f) );
}

// *** Linear 'three colour' blending blue-green-red
// t : 0 to 1
// returns argb colour
Colour GetColour(float t)
{
       float b              =        MathHelper.Clamp(  1 - t*2, 0, 1);
       float r              =        MathHelper.Clamp( -1 + t*2, 0, 1);
       float g0      =        MathHelper.Clamp(      t*2, 0, 1);
       float g1      = 1 - MathHelper.Clamp(  2 - t*2, 0, 1);
       float g              = g0 - g1;
       return Color.FromArgb(255, (int)(r*255.0f), (int)(g*255.0f), (int)(b*255.0f) );
}

// *** Linear-Stepping 'three colour' blending blue-green-red
// t : 0 to 1
// returns argb colour
Colour GetColour(float t)
{
       float b              = MathHelper.Clamp( 2.0f - 4.0f * t,     0, 1);
       float r              = 1.0f - MathHelper.Clamp( 3.0f - 4.0f * t,     0, 1);
       float g0      = MathHelper.Clamp( 4.0f * t,     0, 1);
       float g1      = MathHelper.Clamp( -3 + 4.0f * t, 0, 1);
       float g              = g0 - g1;
       return Color.FromArgb(255, (int)(r*255.0f), (int)(g*255.0f), (int)(b*255.0f) );
}

// *** Linear-Stepping AND linear-scaling 'three colour' blending blue-green-red
// t : 0 to 1
// returns argb colour
Colour GetColour(float t)
{
       float b              = MathHelper.Clamp( 2.0f - 4.0f * t,     0, 1);
       float r              = 1.0f - MathHelper.Clamp( 3.0f - 4.0f * t,     0, 1);
       float g0      = MathHelper.Clamp( 4.0f * t,     0, 1);
       float g1      = MathHelper.Clamp( -3 + 4.0f * t, 0, 1);
       float g              = g0 - g1;
```

```
        b *= 1.5f;
        r *= 1.5f;
        g *= 1.5f;

        r = MathHelper.Clamp(r, 0.0f, 1.0f);
        g = MathHelper.Clamp(g, 0.0f, 1.0f);
        b = MathHelper.Clamp(b, 0.0f, 1.0f);

        return Color.FromArgb(255, (int)(r*255.0f), (int)(g*255.0f), (int)(b*255.0f) );
}

// *** Linear AND scaling 'three colour' blending blue-green-red
// t : 0 to 1
// returns argb colour
Colour GetColour(float t)
{
        float b             =       MathHelper.Clamp(  1 - t*2, 0, 1);
        float r             =       MathHelper.Clamp( -1 + t*2, 0, 1);
        float g0      =       MathHelper.Clamp(      t*2, 0, 1);
        float g1      = 1 - MathHelper.Clamp(  2 - t*2, 0, 1);
        float g             = g0 - g1;

        b *= 1.2f;
        r *= 1.2f;
        g *= 1.2f;

        r = MathHelper.Clamp(r, 0.0f, 1.0f);
        g = MathHelper.Clamp(g, 0.0f, 1.0f);
        b = MathHelper.Clamp(b, 0.0f, 1.0f);
        *
        return Color.FromArgb(255, (int)(r*255.0f), (int)(g*255.0f), (int)(b*255.0f) );
}

// *** Linear-Stepping AND squaring 'three colour' blending blue-green-red
// t : 0 to 1
// returns argb colour
Colour GetColour(float t)
{
        float b             = MathHelper.Clamp( 2.0f - 4.0f * t,     0, 1);
        float r             = 1.0f - MathHelper.Clamp( 3.0f - 4.0f * t,     0, 1);
        float g0      = MathHelper.Clamp( 4.0f * t,     0, 1);
        float g1      = MathHelper.Clamp( -3 + 4.0f * t, 0, 1);
        float g             = g0 - g1;

        b *= b;
        r *= r;
        g *= g;

        r = MathHelper.Clamp(r, 0.0f, 1.0f);
        g = MathHelper.Clamp(g, 0.0f, 1.0f);
        b = MathHelper.Clamp(b, 0.0f, 1.0f);

        return Color.FromArgb(255, (int)(r*255.0f), (int)(g*255.0f), (int)(b*255.0f) );
}

// *** Trig AND squaring the'three colour' blending blue-green-red
// t : 0 to 1
// returns argb colour
Colour GetColour(float t)
{
        b =  (float)Math.Cos( Math.PI * t );
        r = -(float)Math.Cos( Math.PI * t );
        g =  (float)Math.Sin( Math.PI * t );

        r = MathHelper.Clamp(r, 0.0f, 1.0f);
        g = MathHelper.Clamp(g, 0.0f, 1.0f);
        b = MathHelper.Clamp(b, 0.0f, 1.0f);

        r *= r;
        g *= g;
        b *= b;

        return Color.FromArgb(255, (int)(r*255.0f), (int)(g*255.0f), (int)(b*255.0f) );
}
```