Doing Math with RGB (and A)*

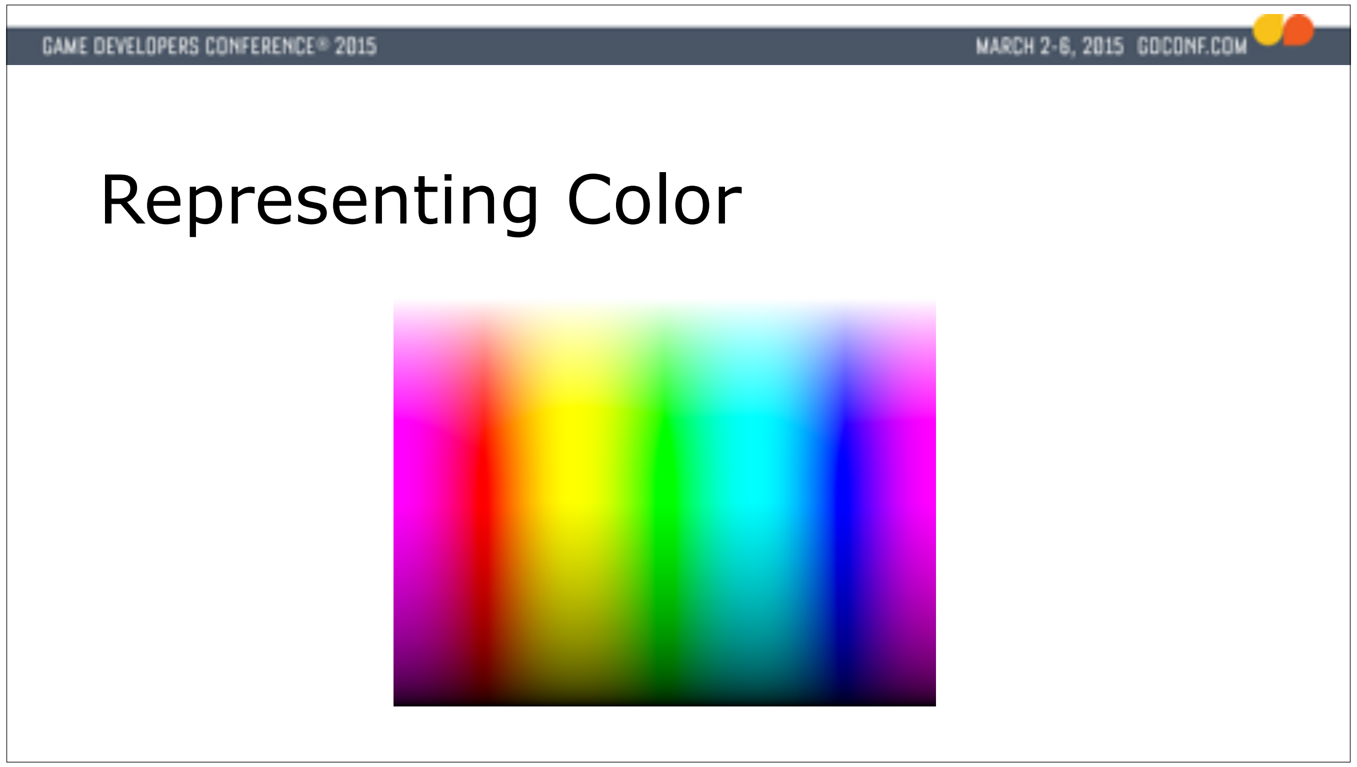
Jim Van Verth
Software Engineer, Google
www.essentialmath.com

*Correctly

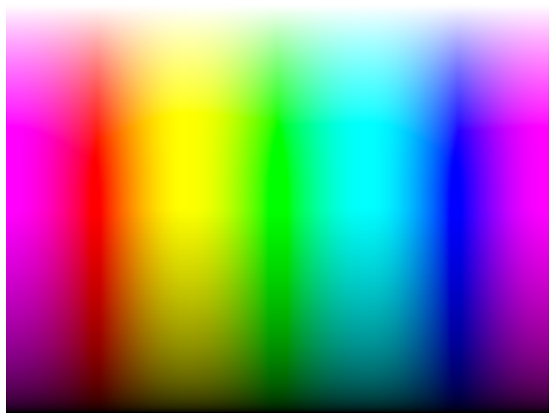
GAME DEVELOPERS CONFERENCE
MOSCONE CENTER · SAN FRANCISCO, CA
MARCH 2-6, 2015 · EXPO: MARCH 4-6, 2015

Topics

- Color Spaces and Gamut
- Gamma correction
- Premultiplied color



Representing Color

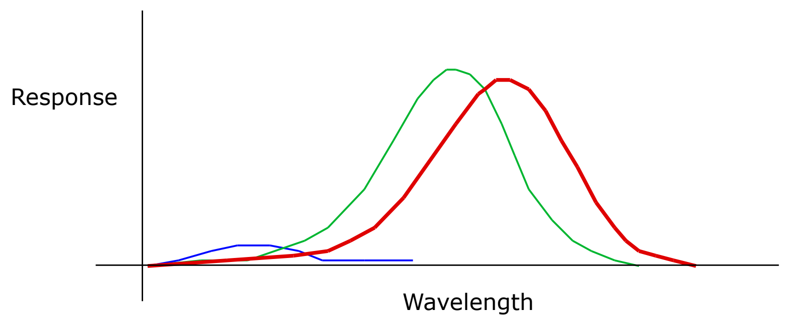


One possibility for representing color is to use a spectrum. High quality renderers use this, however, this is usually too much data for our purposes.

Representing Color

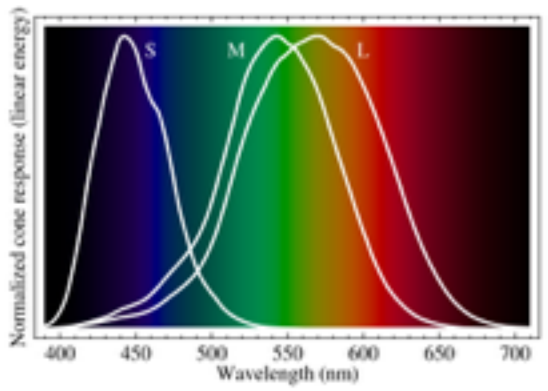
Rather, it’d be better to consider how our eyes perceive color. In particular, how the average observer perceives color.

Representing Color

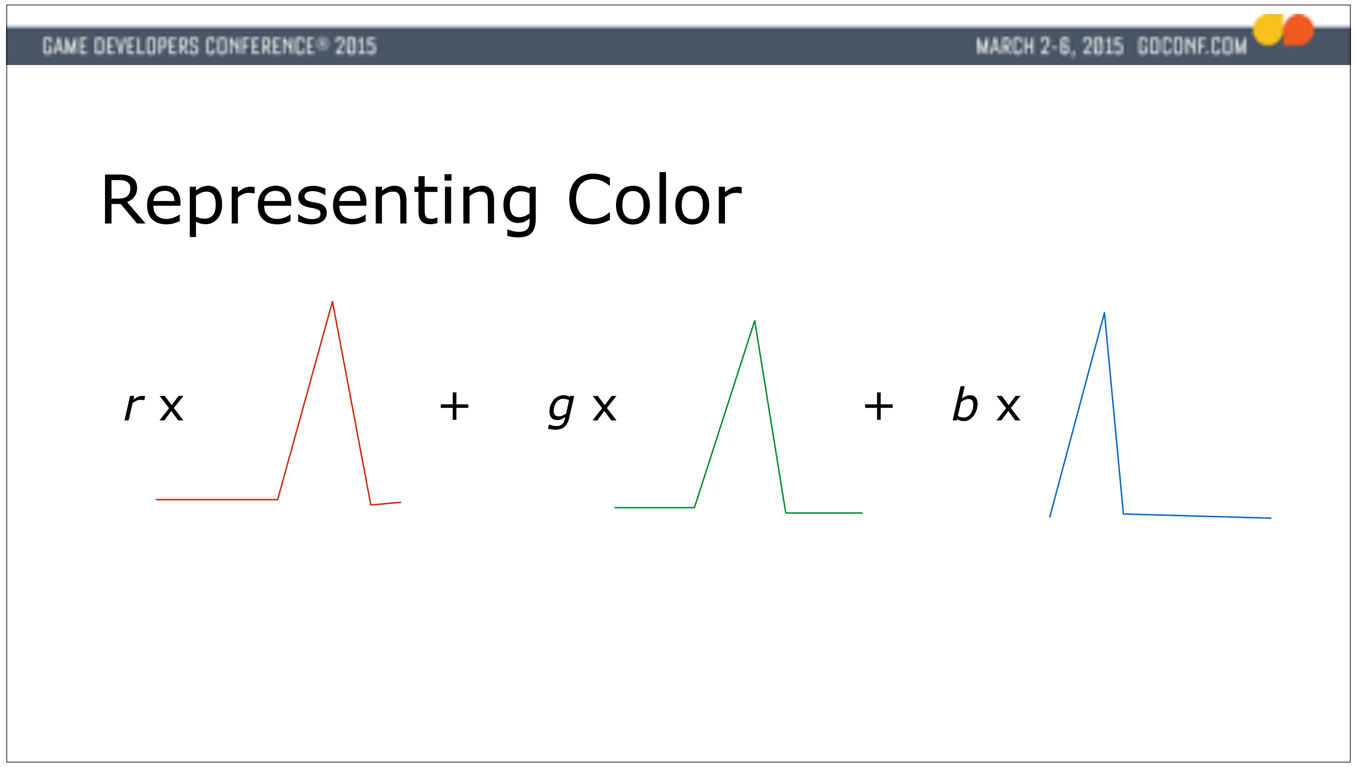


Because our eyes have three different color detecting cells or cones (though recently it was rediscovered that there may be a fourth type, which detects gradual changes in light and reacts to blue-violet), we can represent colors visible to an average observer using only three values. Here we can see the responses for the three cone types, which roughly correspond to red, green and blue.

Representing Color

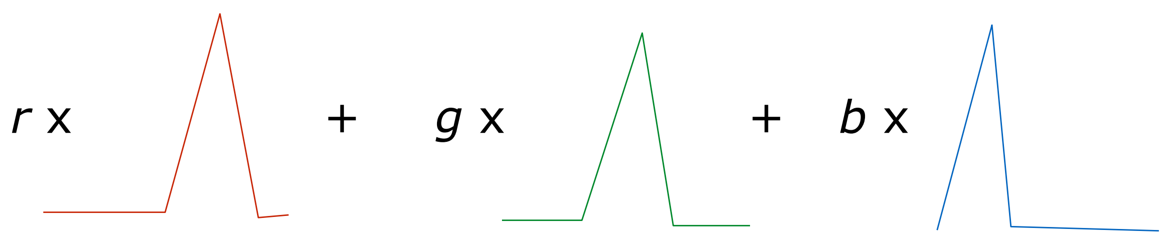


And this is the normalized version overlaid on a spectrum. As you can see, the blue (technically called S, for short) is a little more violet, and red (or L, for long) is actually a little more orange-yellow.



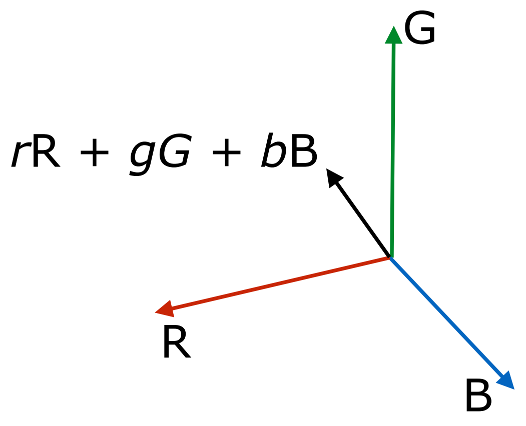
GAME DEVELOPERS CONFERENCE® 2015 MARCH 2-6, 2015 GDCONF.COM

Representing Color

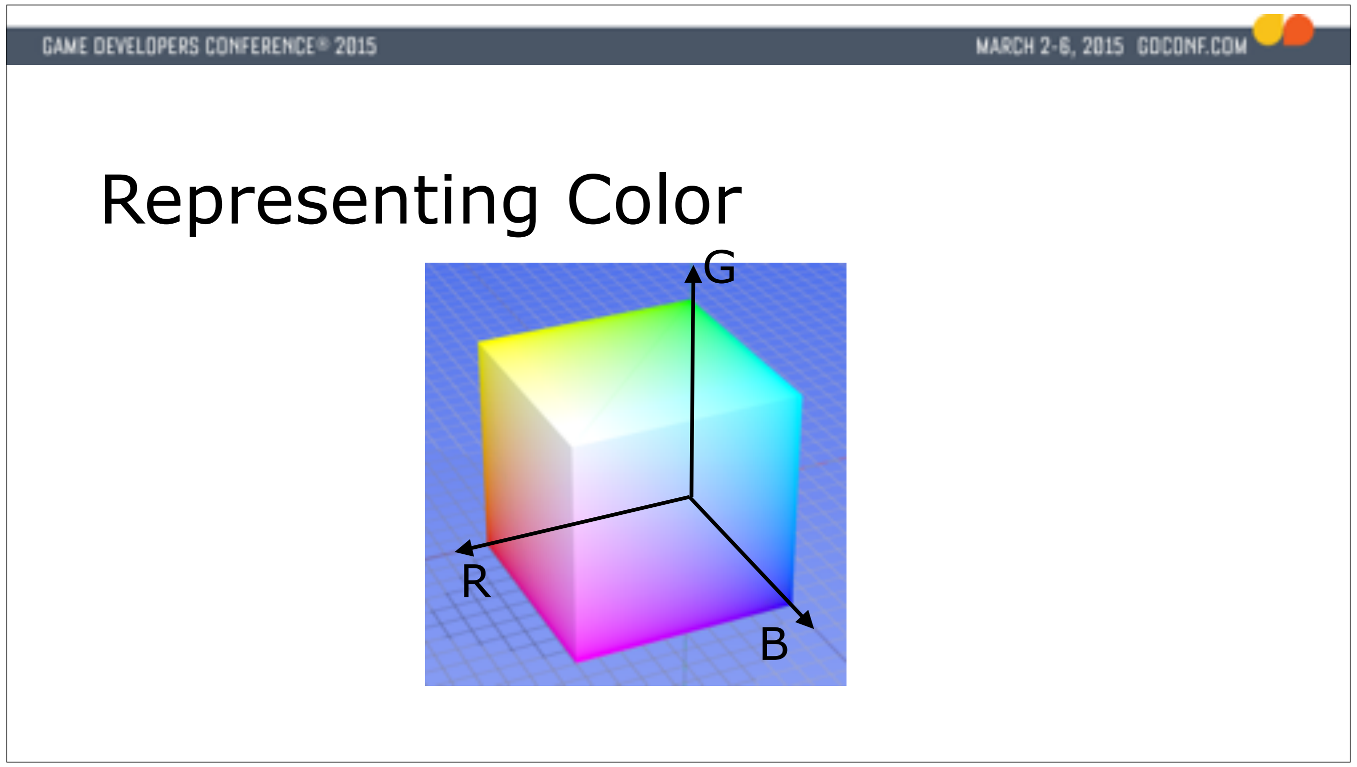


So the idea is that we will use the weighted sum of three primary values to represent color. This is a 3D frame, where the three primaries are our basis, black is the origin, and each point is a different color.

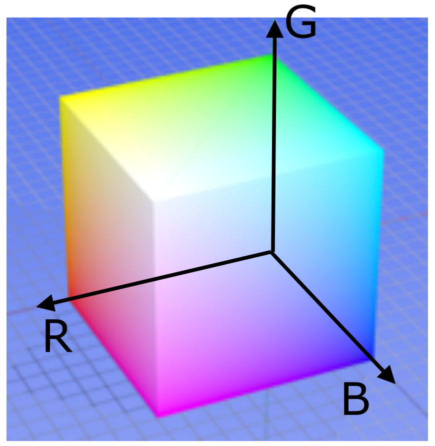
Representing Color



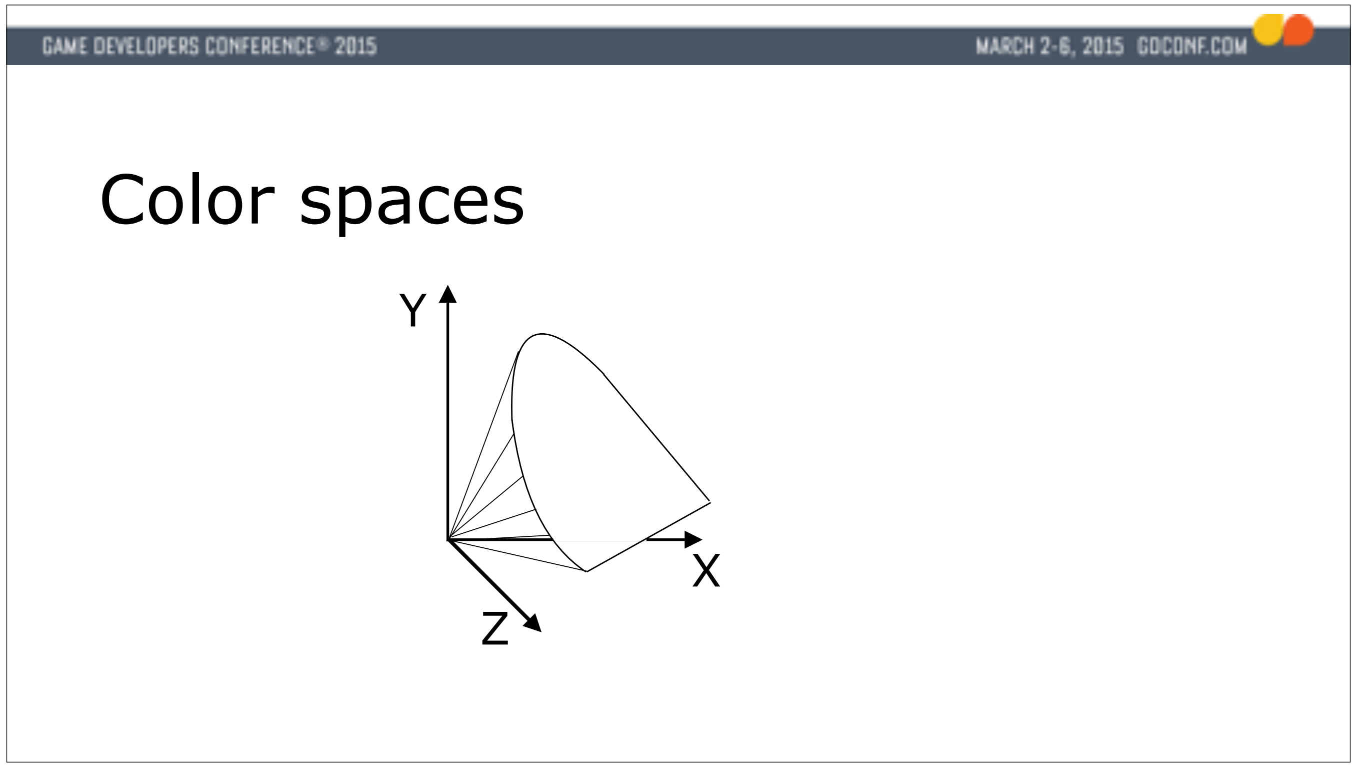
So here it is represented as a 3D frame. Note that on a physical device r, g, and b here can't be more than one — you can only get so bright. Similarly, your retina can only detect so much light before it becomes overstimulated.



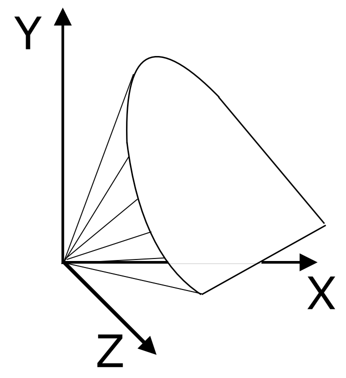
Representing Color



Since our coefficients are limited to the range 0 to 1, the colors we can represent with a given color space are limited to a 1 x 1 x 1 cube, called the color space's gamut. Now, there are a lot of choices we can make for our primaries. Depending on those choices, this gamut will be a certain subset of the average human visual range. So how can we represent that?



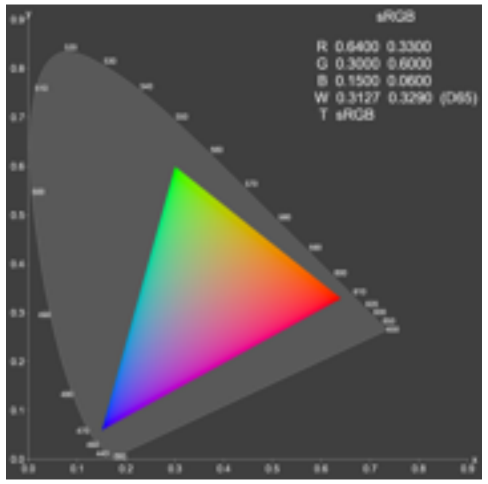
Color spaces



The CIE XYZ color space is another three primary system that encompasses the entire human visual system (again, for the average viewer), represented by this cone-like shape here. However, the X, Y, and Z primaries are not physically representable (though Y represent luminance, and Z is roughly the blue response). That said, it allows use to specify representable systems, and visualize how much of the average visual color space they cover.

Color spaces

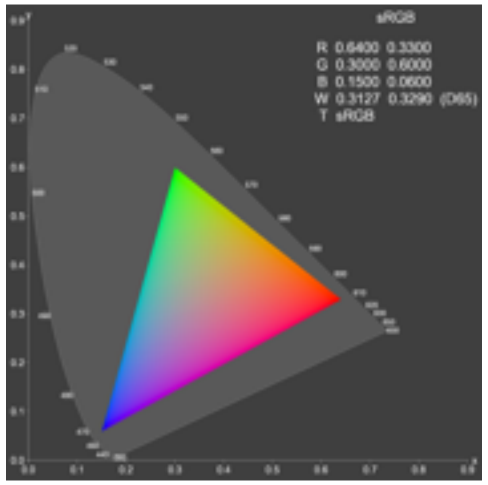
sRGB



So looking at a slice through this cone that contains the most saturated colors (which is where x + y + z = 1), we can show the colors (or at least, the fully saturated versions) that we generally can represent through games. These are the colors for the most common format today: sRGB. The corners of the triangle are the red, green and blue primary colors, and by doing a convex combination of those primaries, we get a broad range of color (though as you'll notice, not all the colors in the average visual space). Note that this image doesn't display any color outside of the triangle. That's because it's not possible! Gets my goat when you see these full color fake CIE images — I think it's deceptive.

Color spaces

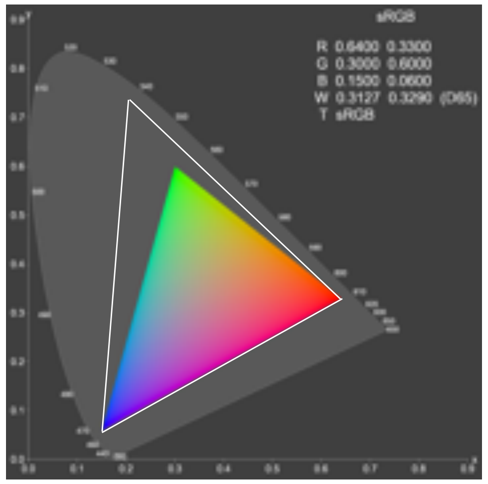
sRGB



History of sRGB? What happened was in the early 90s, there were a large number of color space standards out there — each monitor or operating system might have its own representation for color. So for every image you not only had to include the base color values, you also had to include all of the color space information that image was created in. With the growth of the web and the slow bandwidths at the time, reducing data for images became a big deal. HP (monitors) and Microsoft (operating systems) got together and devised a simple solution: every monitor and every computer would use a single representation for color, called sRGB ('s' stands for 'standard'). It took off, and it is now the de facto standard for everything, including HDTV (technically, sRGB is derived from HDTV's Rec. 709).

Color spaces

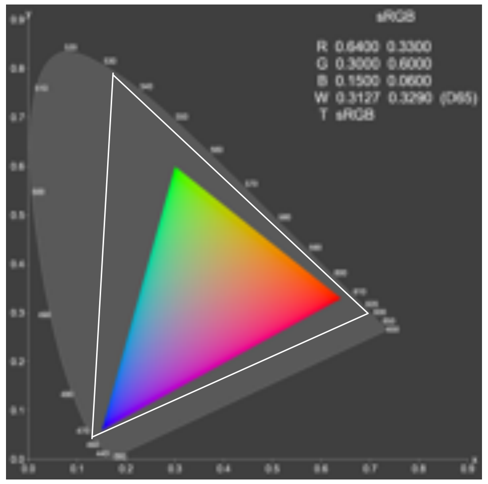
Adobe RGB



Then there’s the Adobe RGB system — this produces a broader range of color, but not quite as large as CIE RGB.

Color spaces

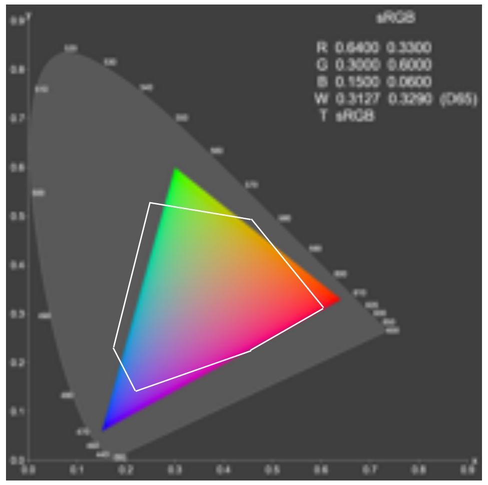
UHDTV



And this is the color space for Ultra HDTV, or Rec. 2020.

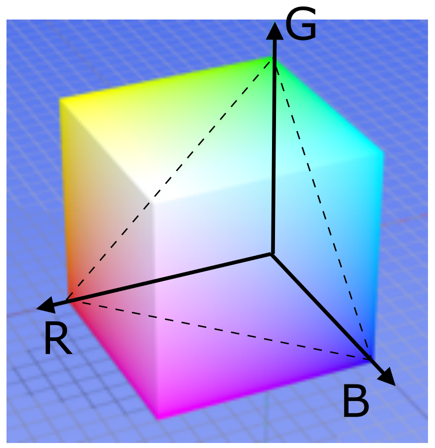
Color spaces

CMYK



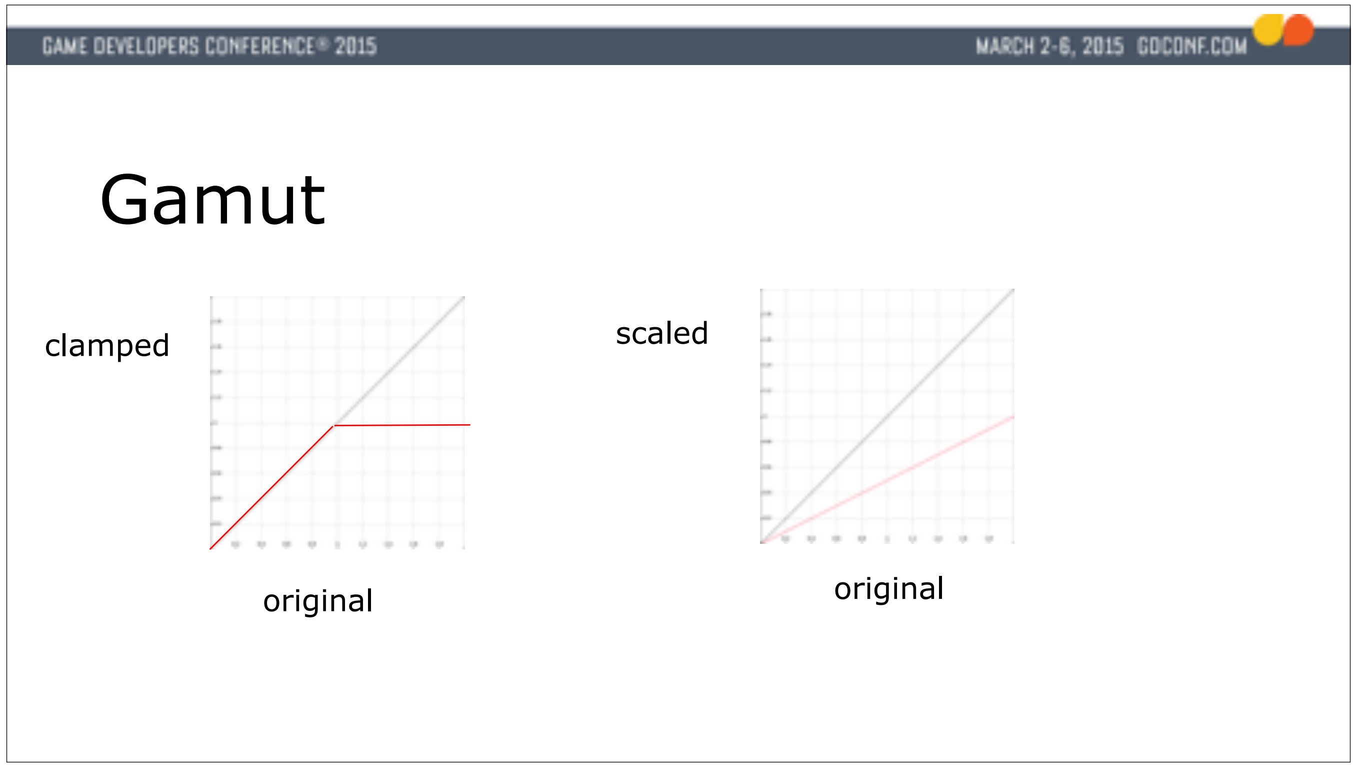
Another system is the CMYK system — this is a subtractive system used in printing, where we use cyan, magenta and yellow inks to produce color. The key, or black, is used only for areas where the values of cyan, magenta and yellow are equal. This gives better grayscale values than trying to perfectly align three dots. Not terrible useful to game developers, unless you’re planning a printer game.

Gamut

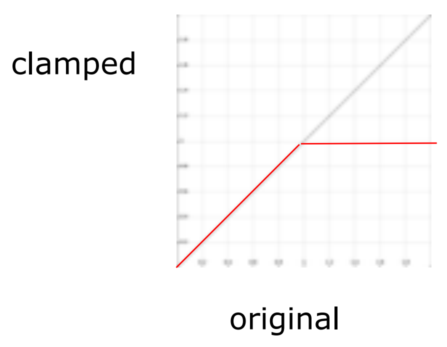


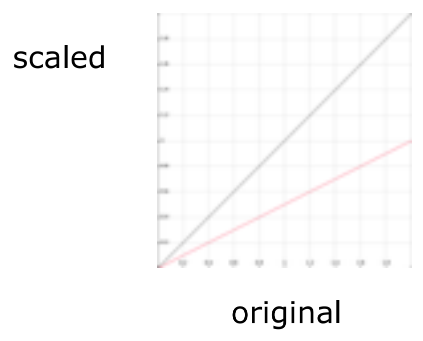
In any case, the triangle I showed for sRGB was just the fully saturated colors. To get the full color space, including lighter and darker colors and white and black, again, we represent that as a 3D space, where each point is a color.

So this raises one question: what happens if we end up with a coefficient that's greater than 1? This is not as crazy as it seems. Floating point error can end up with values slightly greater than one. But more commonly, when doing physically-based lighting calculations we can have ranges of lighting from starlight to bright sunlight (which I suppose is another kind of starlight, just much closer). This is broader than the gamut of our monitor, so we'll end up with values greater than 1, allowing us to brighten up the scene (otherwise we effectively end up darkening every scene, with some hacks).



Gamut

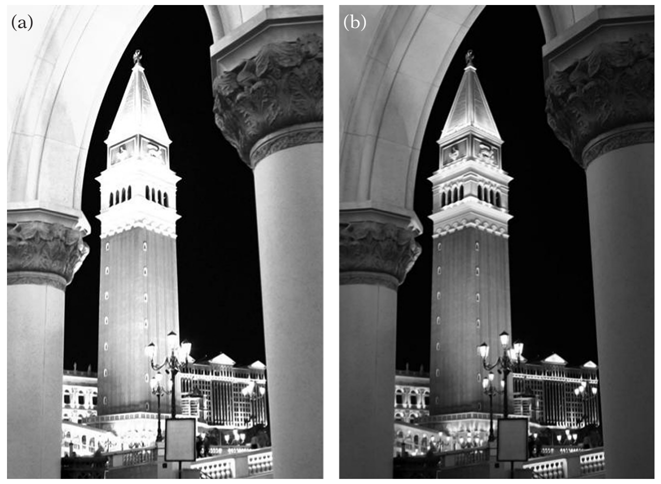




One solution is to clamp every value to 1 — this is pretty common if your results are fairly close to your gamut. This might happen if you have a little error in your calculations, for example. Another solution is to determine the maximum values across an entire range of colors and scale all of them by a common scale factor.

Gamut

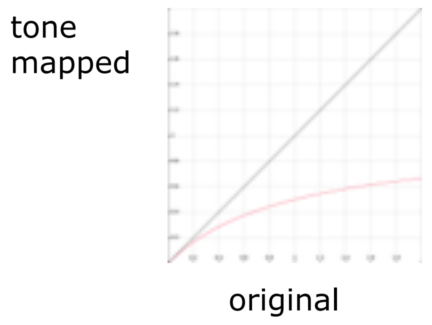
Clamp



Scale

So here, the left image shows a photograph clamped to a valid range, the right one rescaled to that range.

Gamut



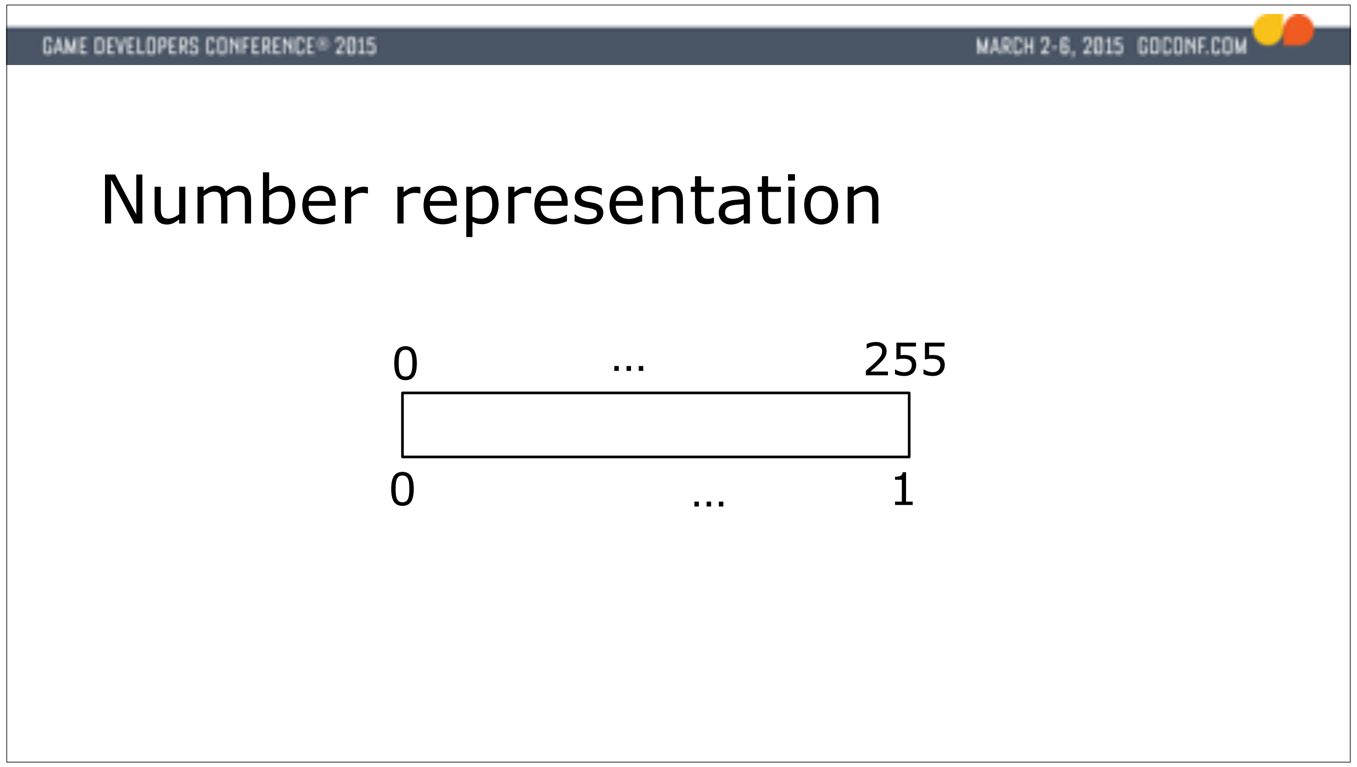
The standard solution these days is to use tone mapping. The scene is analyzed to determine a log-average luminance (or perceived brightness). This can either be done in small areas or globally across the entire image. Then each region is scaled based on that average luminance to place that luminance at a medium value. Finally, this curve is applied to get all the values into range. This mimics what the eye does when we move from a bright outside area to a dark room, or vice versa. It's effectively a virtual iris.

Gamut

Tone mapped

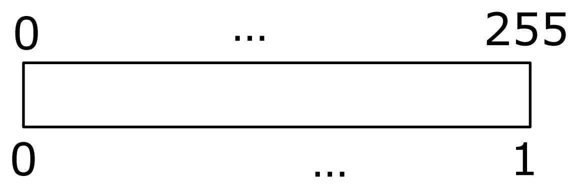
And here we see a rather extreme version of tone mapping. In general, most engines these days support tone mapping, so worrying about it is probably either above or below your pay grade.

So to sum up, in order to handle values outside our gamut, we need to make decision on how shift things back into range. For 2D games or simple 3D games, this is not much of an issue, but it's still good practice to be aware of it.

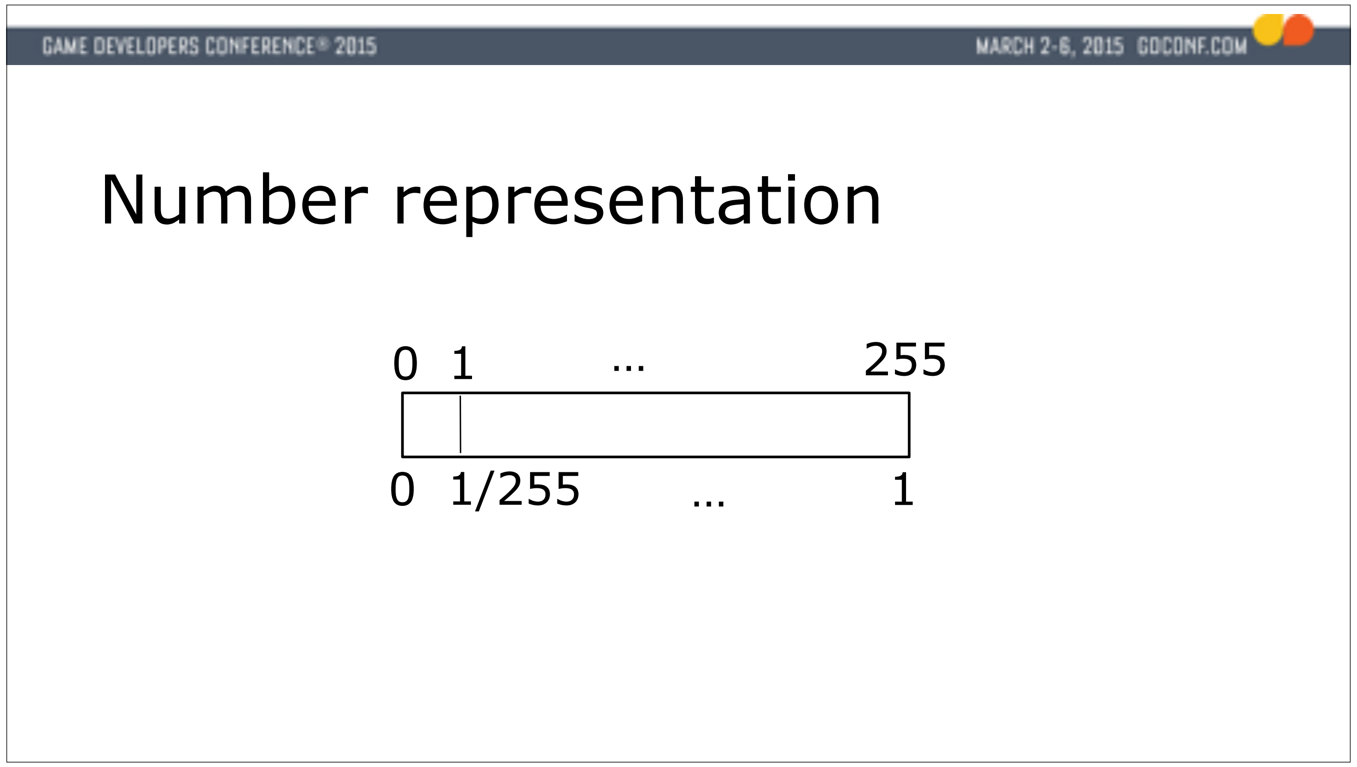

GAME DEVELOPERS CONFERENCE® 2015

MARCH 2-6, 2015 GDCONF.COM

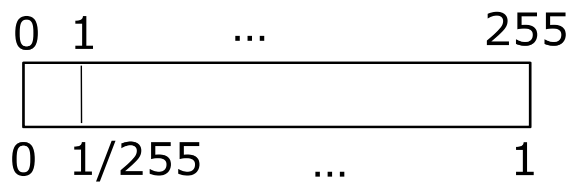
Number representation



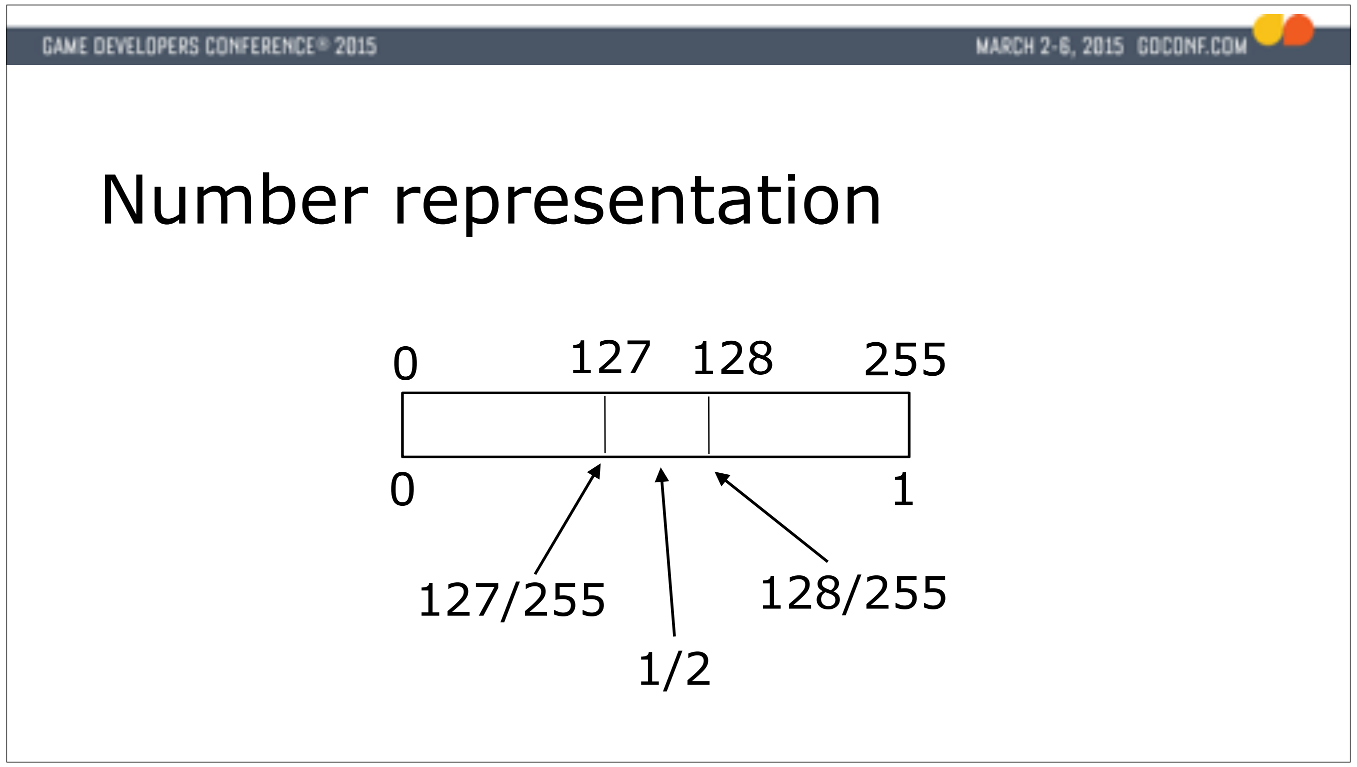
Another question when representing color is how to store the values for each color channel. The standard method is to represent them as an eight bit value, with 0 mapping to 0 and 255 mapping to 1. This is known as 24-bit color or sometimes true color. “True color” is a misnomer. Despite the fact that you can represent billions of values, and the human visual system can distinguish between billions of colors, we’re only representing billions of colors within the sRGB gamut, which we’ve seen does not represent the full color perception of the average viewer.



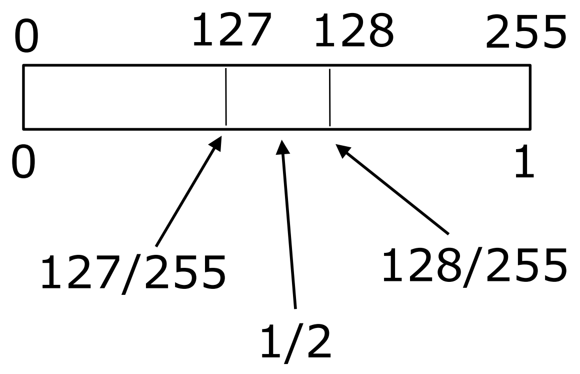
Number representation



We don’t really think about this much, but particularly when working with binary numbers, this is a pretty odd system. The difference between each neighboring values is 1/255, which can’t be represented exactly in floating point, or in any fixed point system.



Number representation

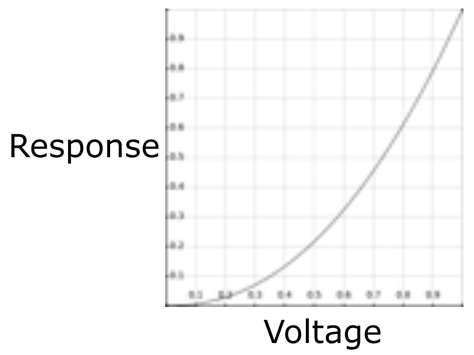


Another issue with this is that we can’t represent standard values like 1/2 (prime factors are 17, 5 and 3). I ran into this myself with creating distance field textures — I wanted to use the value 1/2 to represent the border between the interior and exterior of a shape, and couldn't do it exactly. Finally, we only get to distinguish between 256 values of one primary, which for some applications (gradients, e.g.) is not enough. We’ll talk a bit later about this limitation later when we cover gamma.

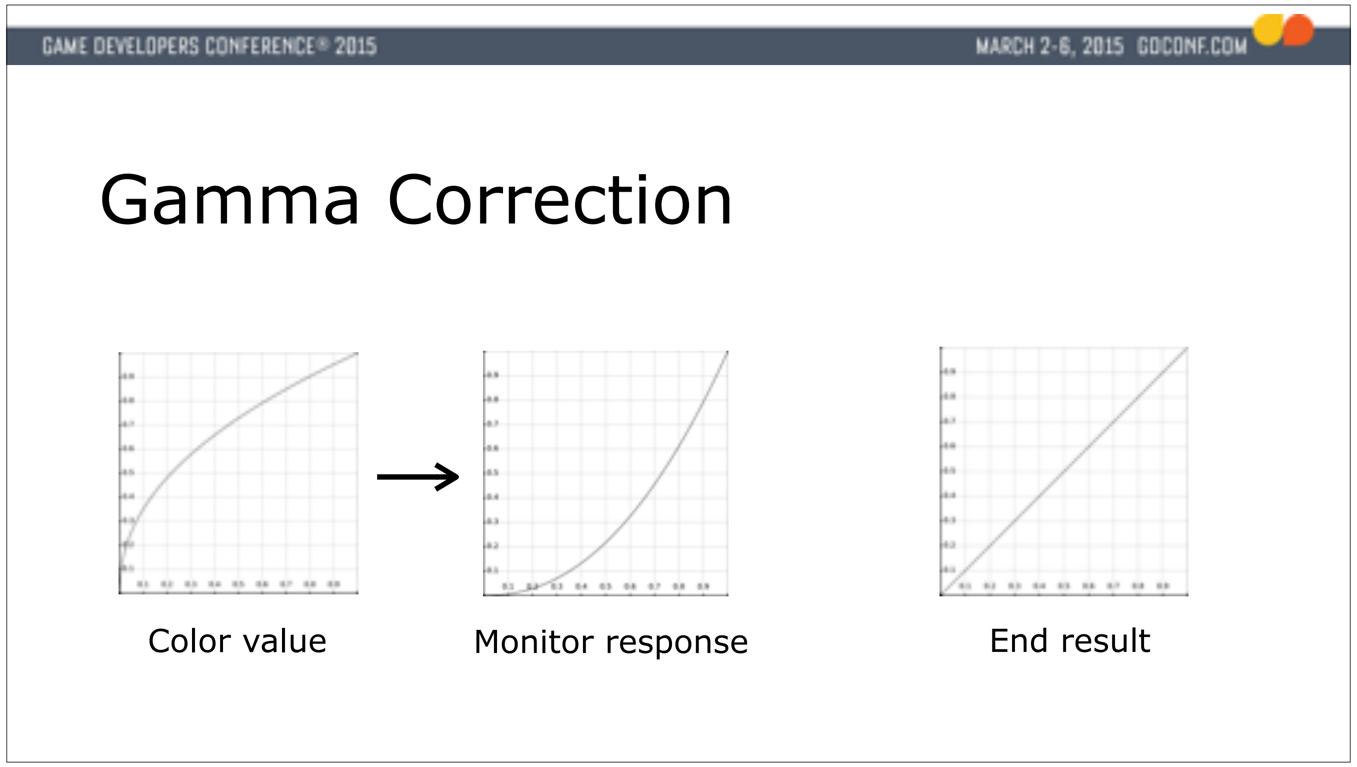
Number representation

One solution is to use some kind of floating point format, either single precision floating point (32 bits), or half precision (16 bits). This does broaden our range considerably. The downside, of course, is that they use more space and hence more memory traffic, may be slower on some platforms, and not all features may be supported (for example, writing to floating point images). But if you’ve got ‘em and can afford the space and time, this is a great solution.

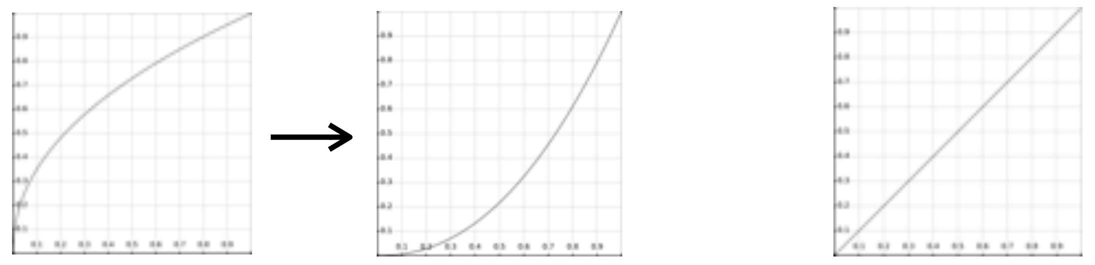
Gamma Correction



In order to explain the next topic, we again need to go back in history. So put on your flannels and drink your lattes — we are going way back to the 1990s. Well, truthfully before that. In those days, the primary displays were CRTs. And they have an odd response to voltages. At small amounts of voltage, a slight change in voltage produces a relatively small change in brightness on the screen. And conversely, at larger amounts of voltage, a small change in voltage produces relatively large change in brightness. The curve looks like this, and is an exponential curve which is roughly quadratic. Each monitor had a different exponential response, known as that monitor's "gamma".

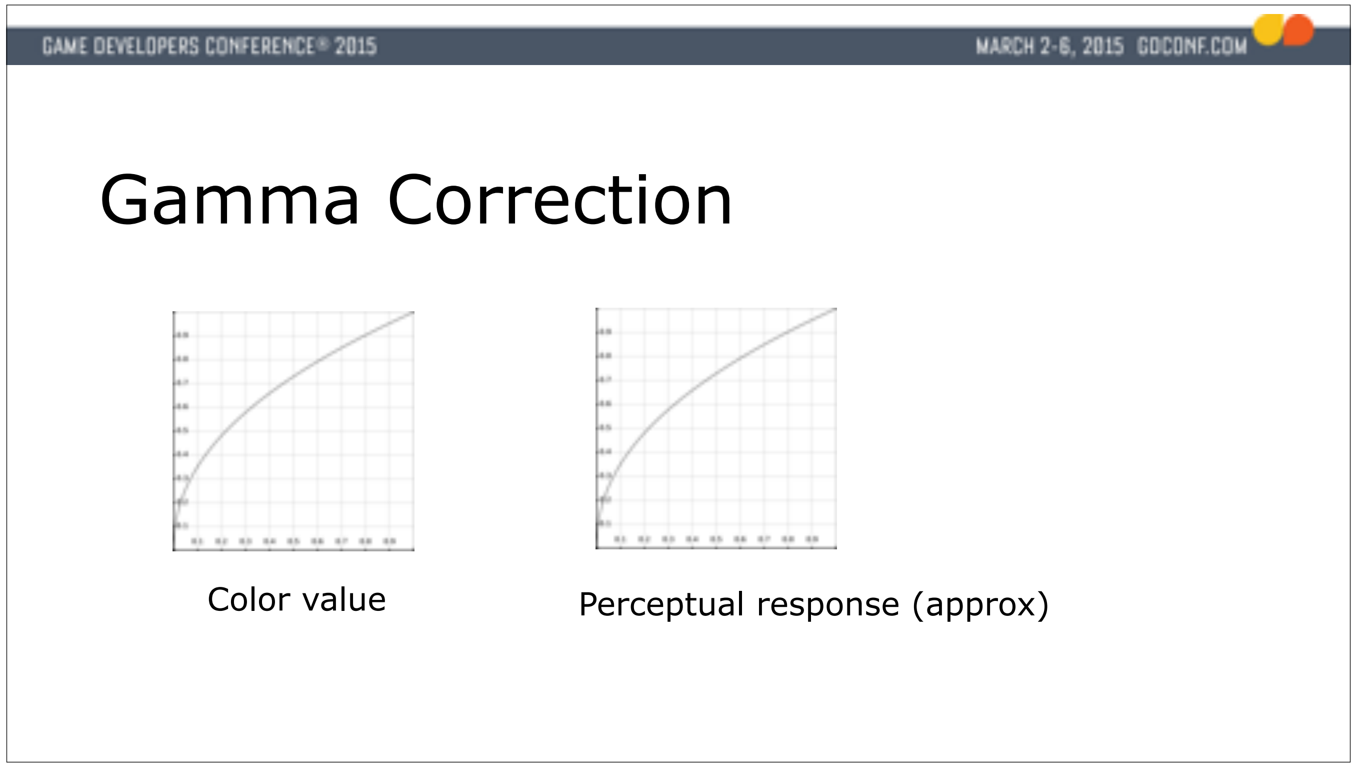


Gamma Correction

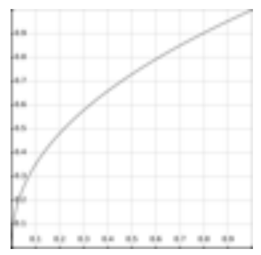


The problem with this is that we want our color values to behave linearly — if we double a color value, we expect double the response on the screen. The solution is to remap our color values using gamma correction. This runs the color value through the inverse of the gamma function, so when both are applied, we get a linear response.

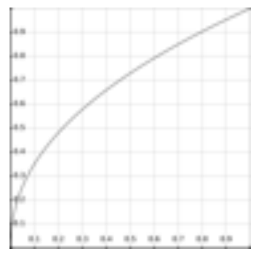
That said, why are we doing this at all? We don't use CRTs anymore, and LCD and OLED displays have a completely different physical response curve. The standard gamma response for LCDs is produced by correcting hardware — why not just correct it to produce a linear response? Two reasons: first, it's hard to kill the standard.



Gamma Correction

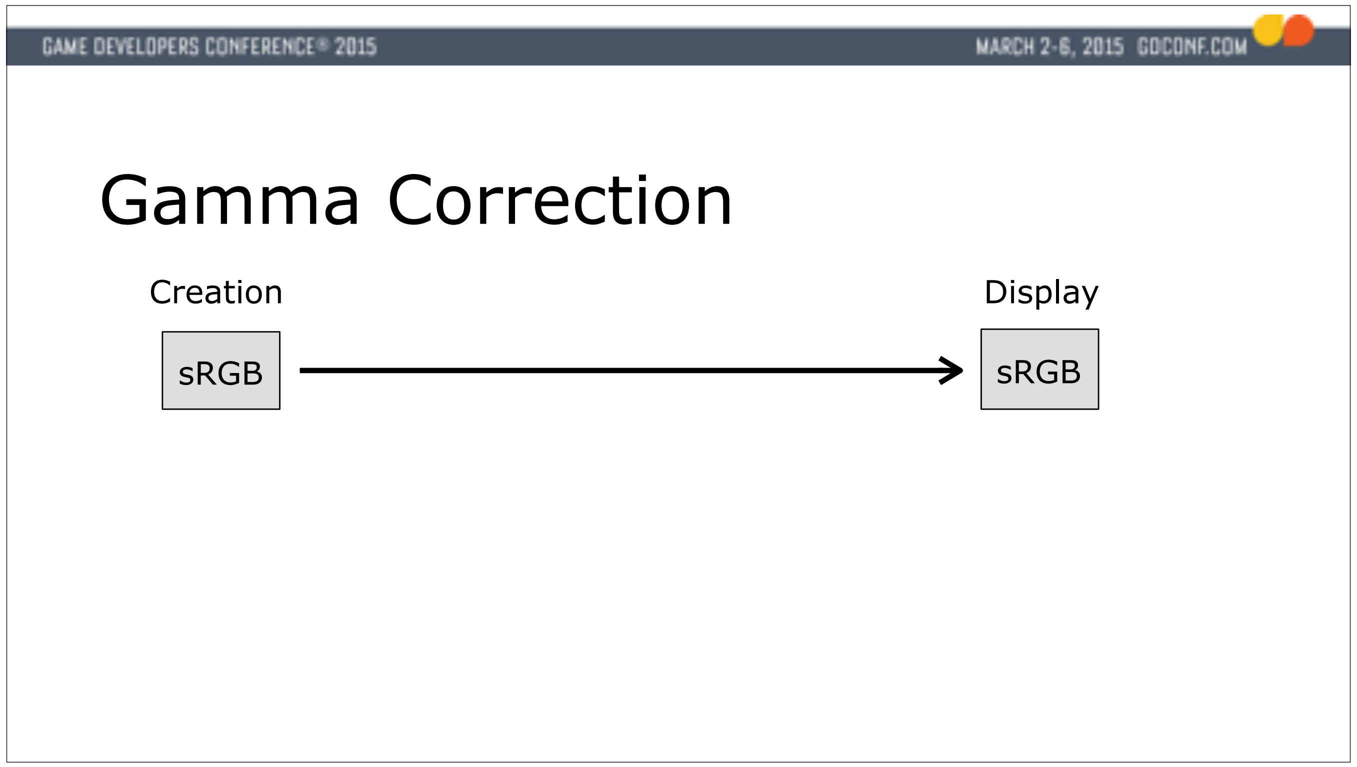


Color value



Perceptual response (approx)

But the other reason is that the gamma corrected curve is very close to our eyes' perceptual response. Our eyes are very sensitive to changes in dark values, and not as sensitive to changes in light values. By storing our color values in this way this we are effectively compressing them to match how we'll perceive them.

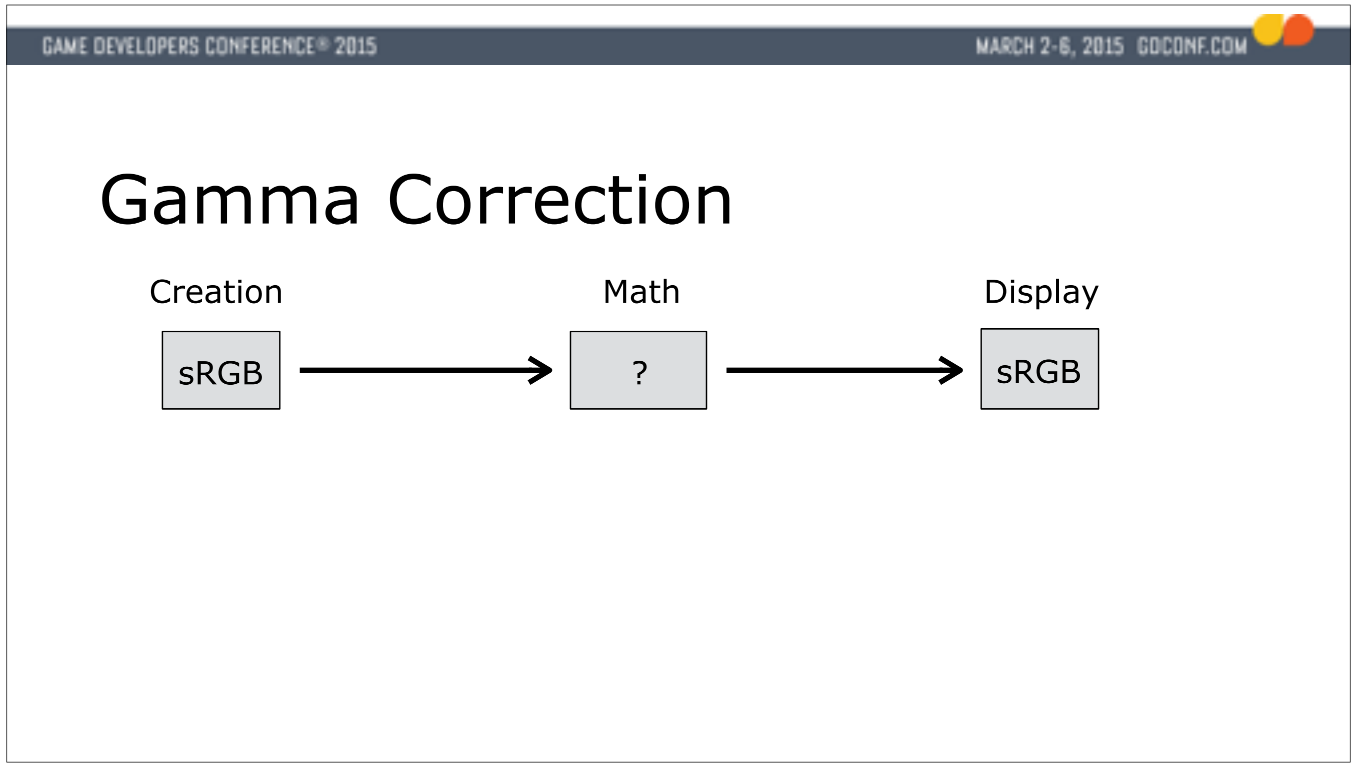


Gamma Correction

Creation Display

$$\begin{tikzcd}\boxed{\text{sRGB}} \arrow[r] & \boxed{\text{sRGB}}\end{tikzcd}$$

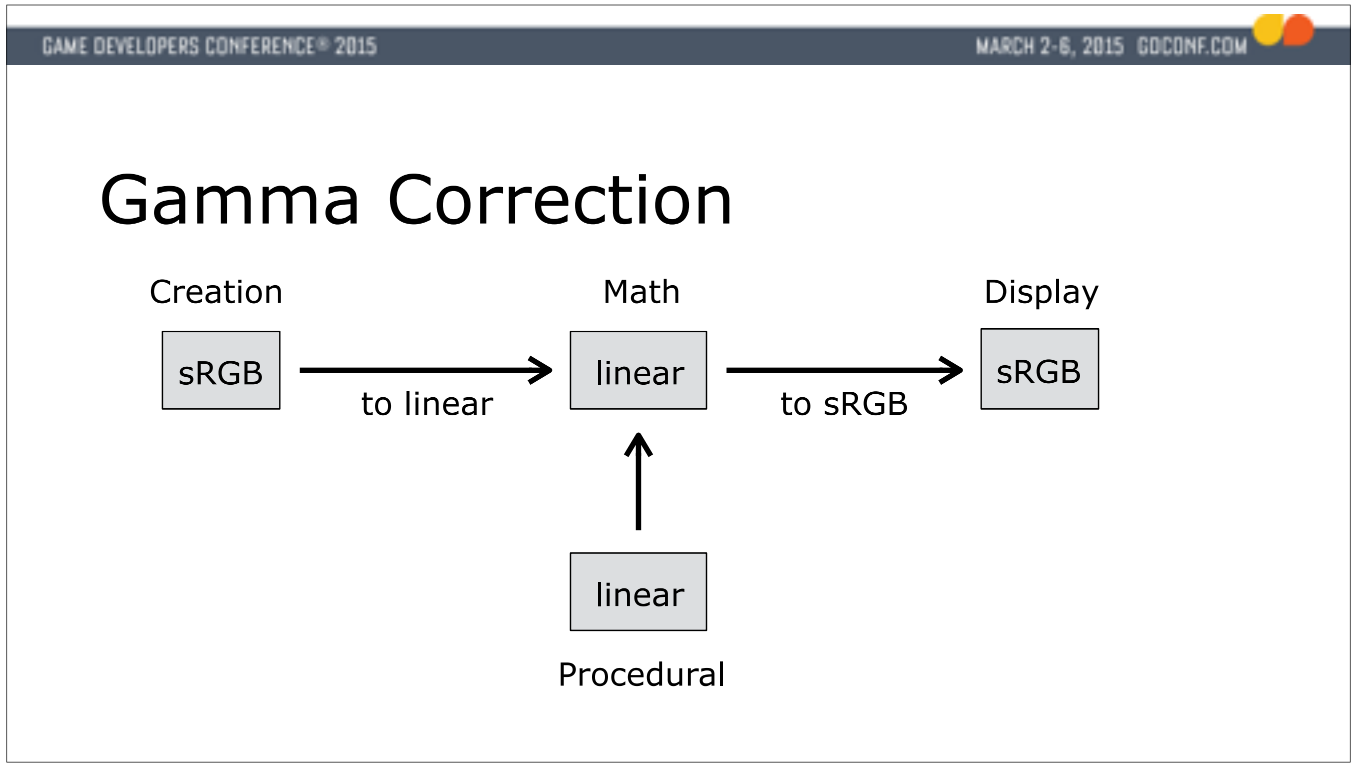
The sRGB color space, in addition to the primaries, defines a gamma curve which is roughly an exponent of 2.2 (it's actually a short linear curve plus an exponent of 2.6). The assumption is that any files created in a standard art package are using sRGB color, so they have built-in gamma correction. When we pass those to our display, the display applies the sRGB gamma and end result is that we see the same colors as the when the artist created it on her monitor.



Gamma Correction

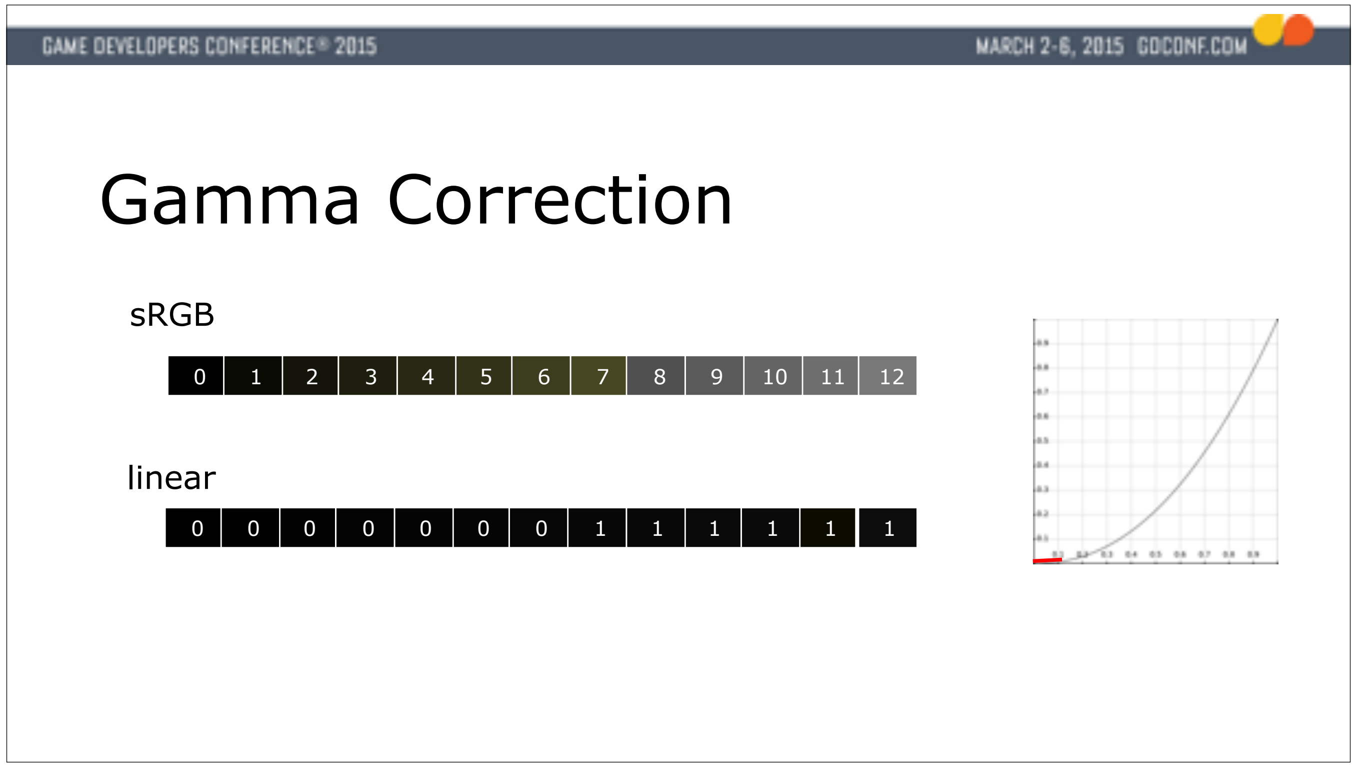
$$\begin{tikzcd} \text{Creation: } \boxed{\text{sRGB}} \arrow[r] & \text{Math: } \boxed{?} \arrow[r] & \text{Display: } \boxed{\text{sRGB}} \end{tikzcd}$$

The problem comes in when we actually want to do some calculations on those colors. For example, suppose we apply a texture to an object, and then light that object. The lighting values are presumably physical and in a linear space, but the sRGB colors are in a non-linear space. Or more simply, suppose we have two textures and want to blend between them. Blending is usually a linear operation, but again our textures are in a non-linear space.



Gamma Correction

The solution is to convert our sRGB gamma corrected colors to a linear RGB space, perform our operations, and then convert them back. Raises a question — why not just convert on load and store linear values, use linear values, then only convert to sRGB when ready to display?



GAME DEVELOPERS CONFERENCE® 2015 MARCH 2-6, 2015 GDCONF.COM

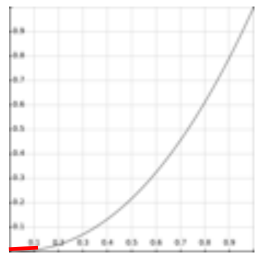
Gamma Correction

sRGB

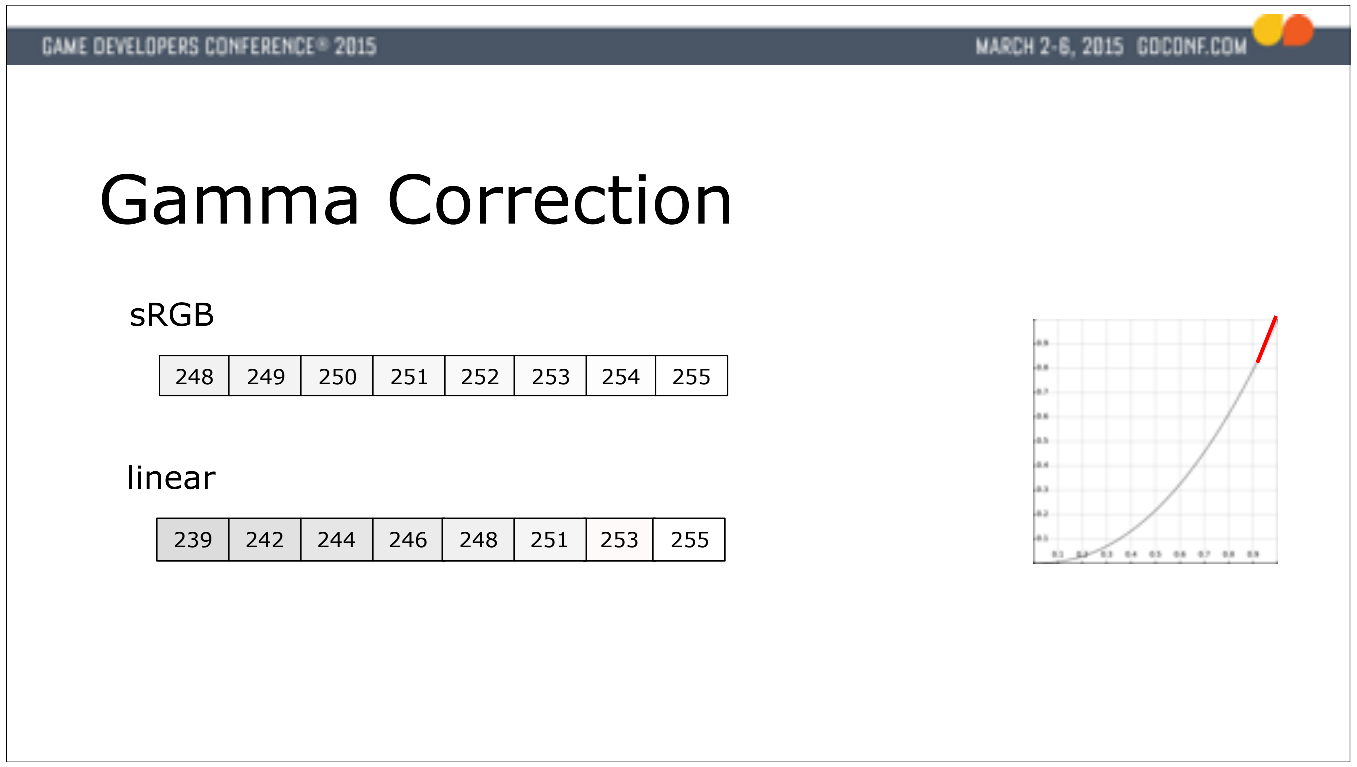
0	1	2	3	4	5	6	7	8	9	10	11	12

linear

0	0	0	0	0	0	0	1	1	1	1	1	1



This could work, but with 8-bit values, at the low end we lose a lot of precision. We would ideally need 13 bits of precision to end up with corresponding dark values that are as distinguishable in linear as they are in sRGB.



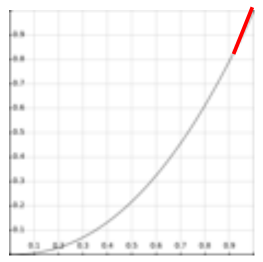
Gamma Correction

sRGB

248	249	250	251	252	253	254	255

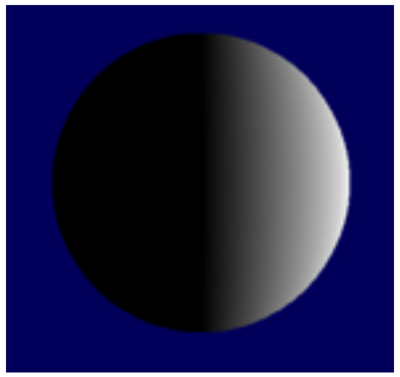
linear

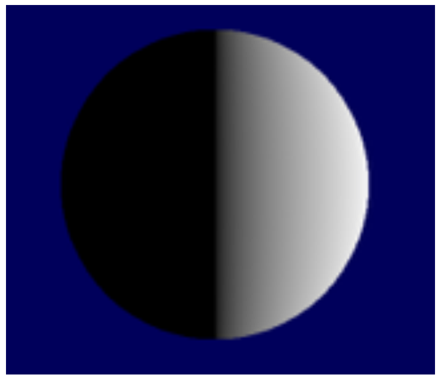
239	242	244	246	248	251	253	255



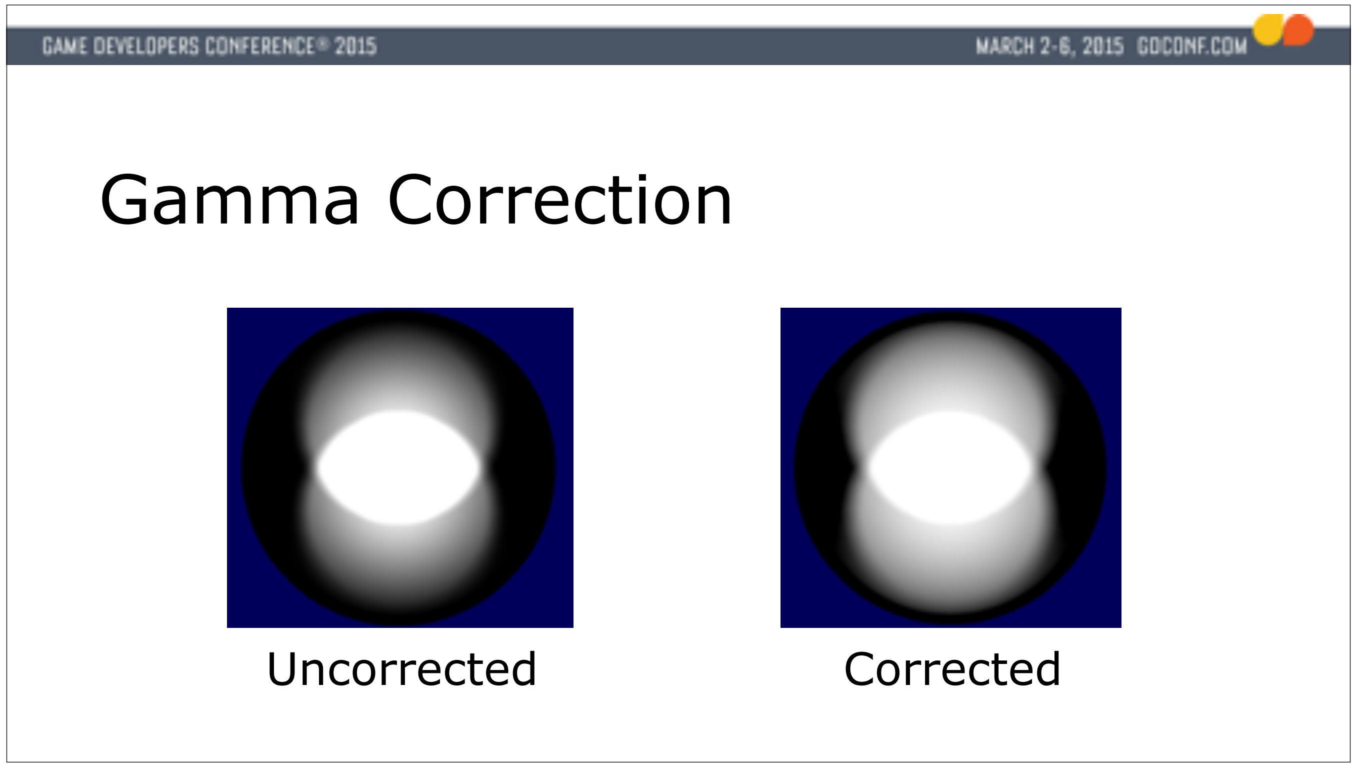
So okay, let’s store our color values using half floats (which gives us at least 13 bits of precision). But then at the brighter end, linear has a sharper curve, so we end up wasting those bits representing those values. So if you’ve got the space, using a 16-bit per-channel format could work. But if you need to use 8-bit color, better to just keep our storage format as sRGB.

Gamma Correction

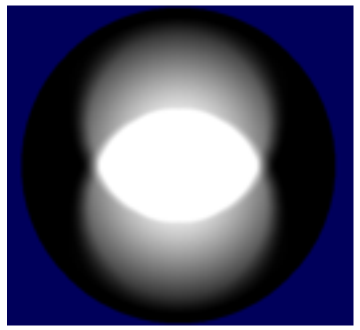

Uncorrected

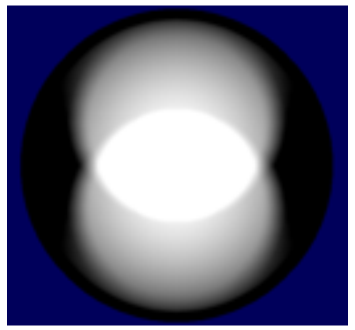

Corrected

Lighting example — uncorrected image has a very fuzzy shadow line. Corrected image has clear sharp shadow at halfway point.

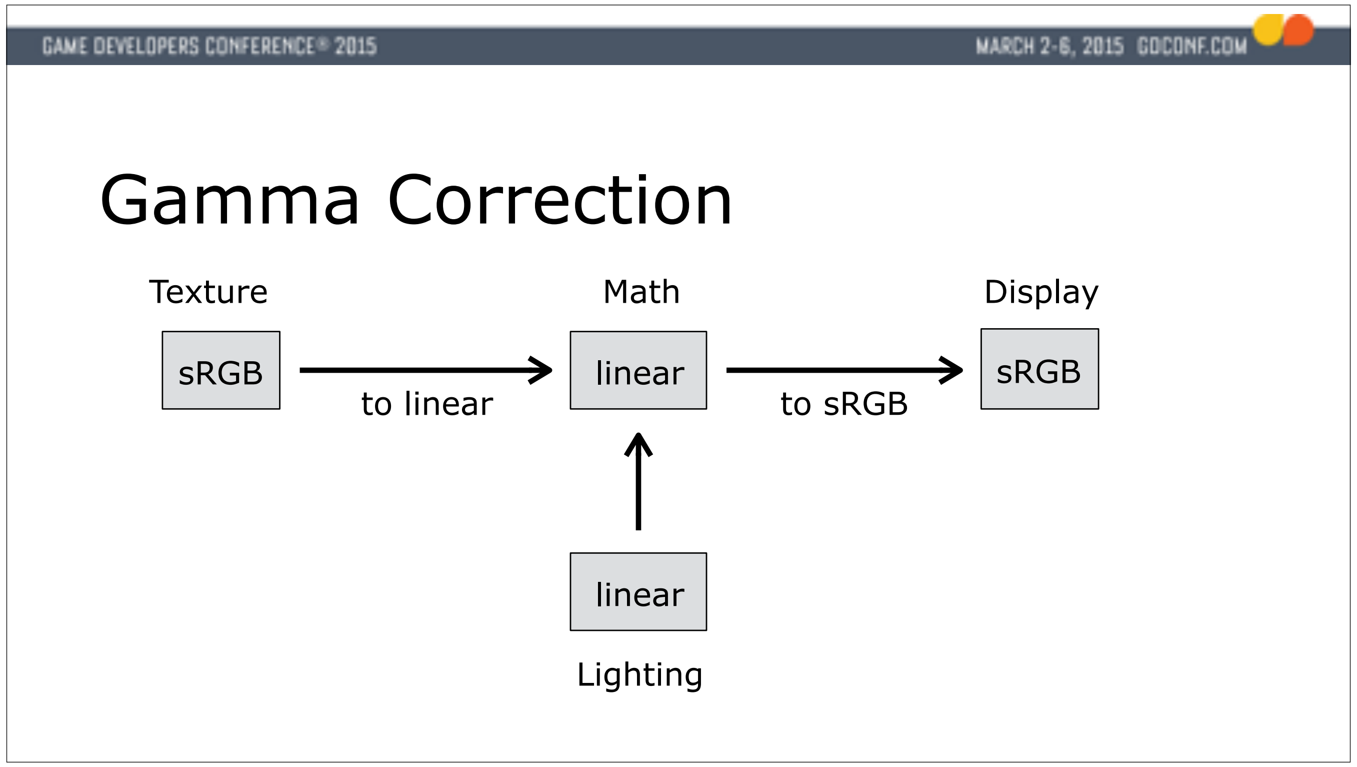


Gamma Correction


Uncorrected


Corrected

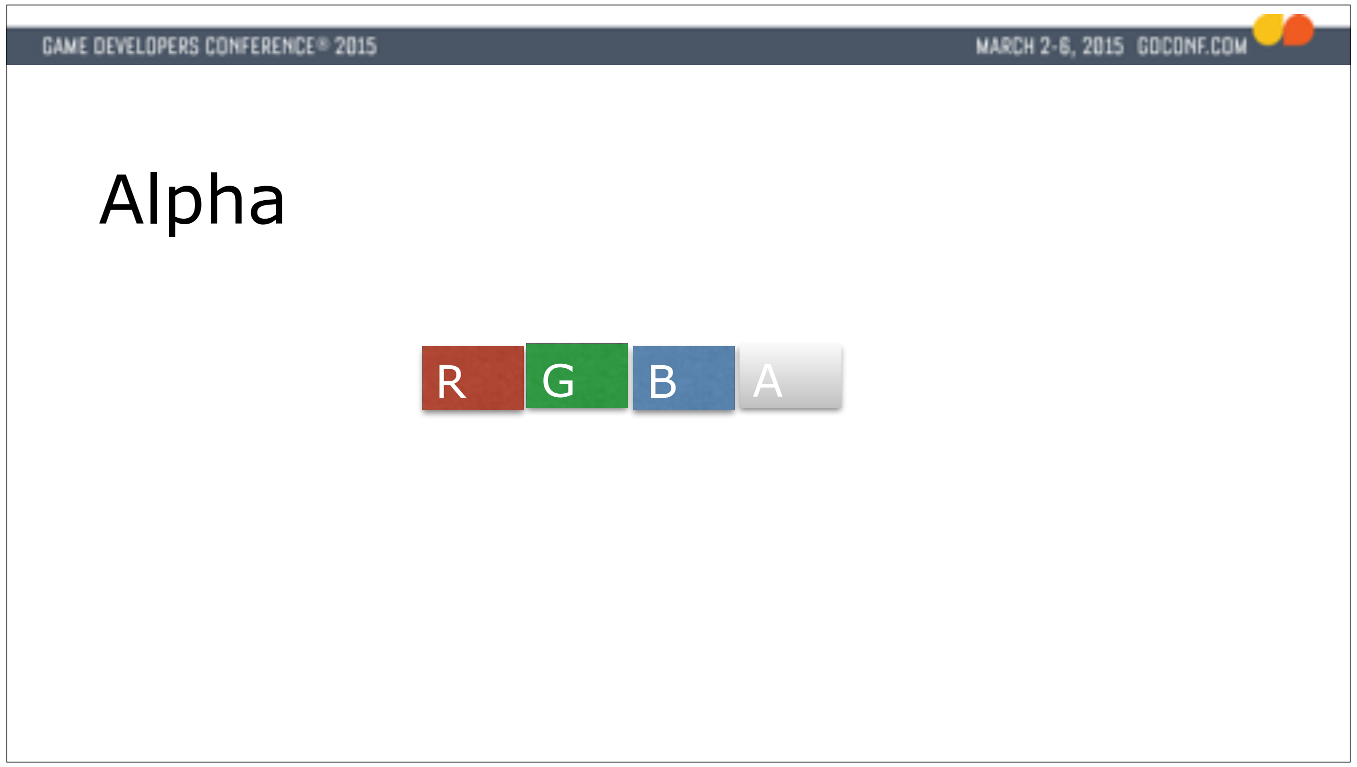
Another example: two spotlights on a sphere. Uncorrected version looks too bright in the overlap and dim in the other areas. Corrected looks like the right amount of blending. Note that the bright section has been clamped to 1 — probably should have used tone mapping on this image to get even better results.



Gamma Correction

$$\begin{tikzcd}
\text{Texture: sRGB} \arrow[r, "\text{to linear}"'] & \text{Math: linear} \arrow[r, "\text{to sRGB}"'] & \text{Display: sRGB} \\
& \text{Lighting: linear} \arrow[u] &
\end{tikzcd}$$

So this is the process we're stuck with. Fortunately, most APIs/hardware support automatic conversion — so in general don't have to worry about it. Just make sure that it's enabled.

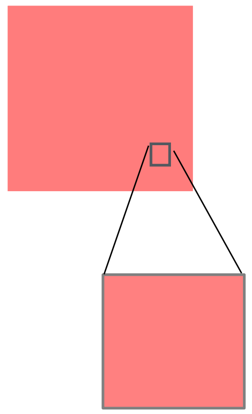


Alpha

R G B A

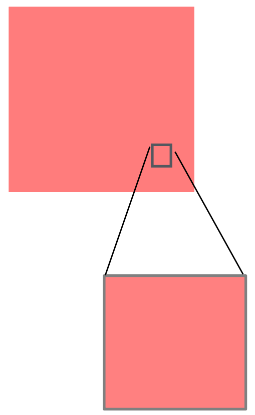
So we’ve talked about RGB colors. Let’s finish by discussing the final element used with color — alpha.

Alpha



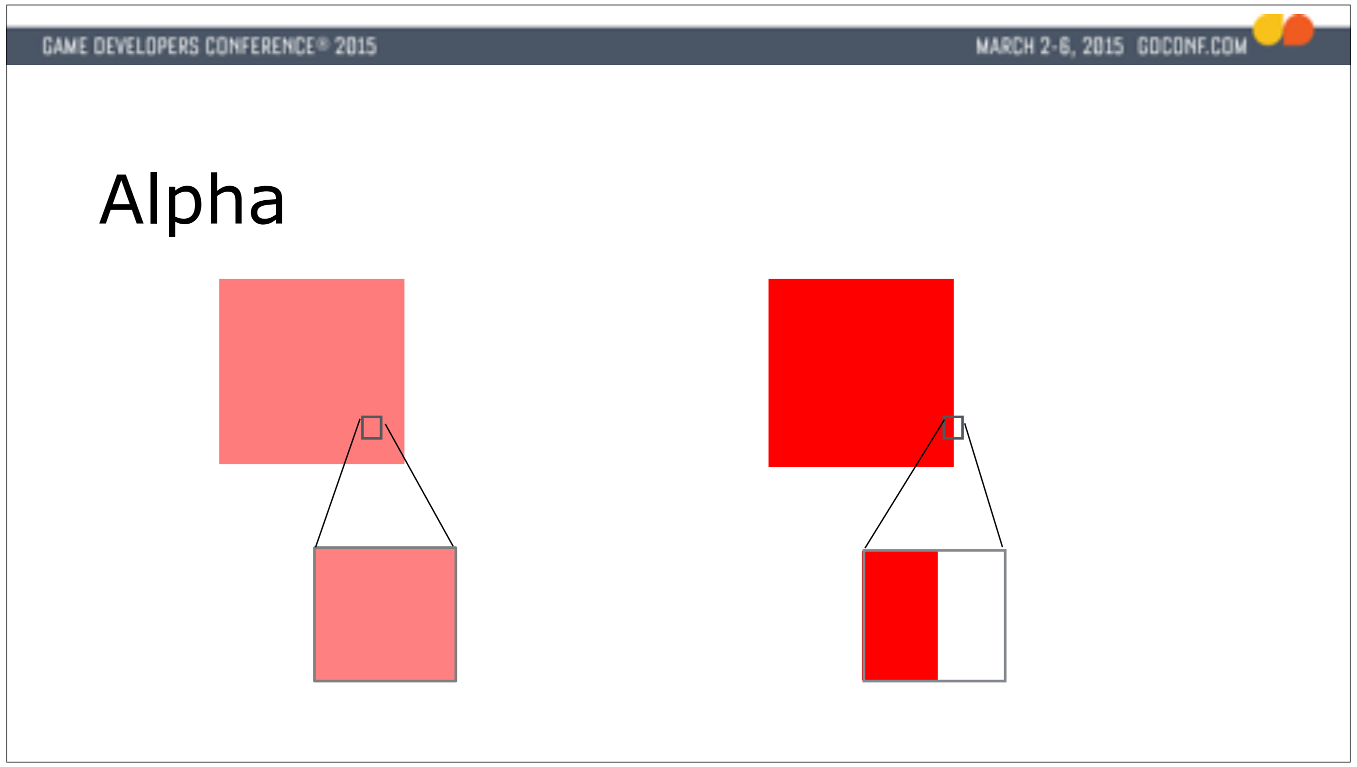
Traditionally we think of alpha as meaning transparency — or perhaps more accurately, opacity. In this interpretation, looking at a single pixel, the color fills the pixel with an opacity value of, for example, 50%. (This looks like a light salmon square, but it’s really a red square with opacity of 50% covering the white background).

Alpha

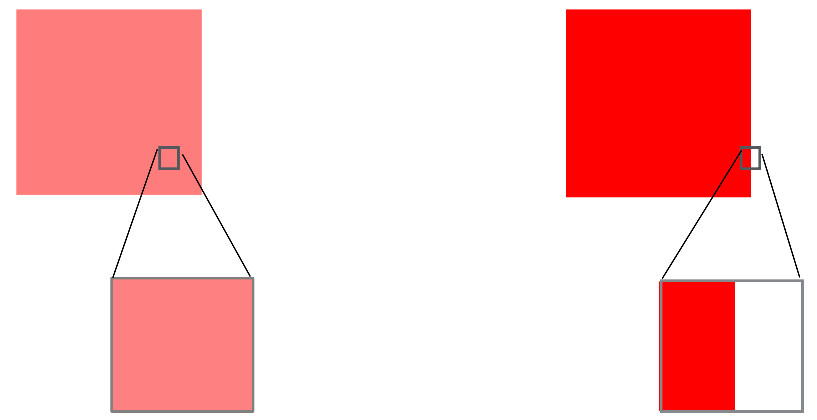


$$(1-\alpha_s)C_d + \alpha_s C_s$$

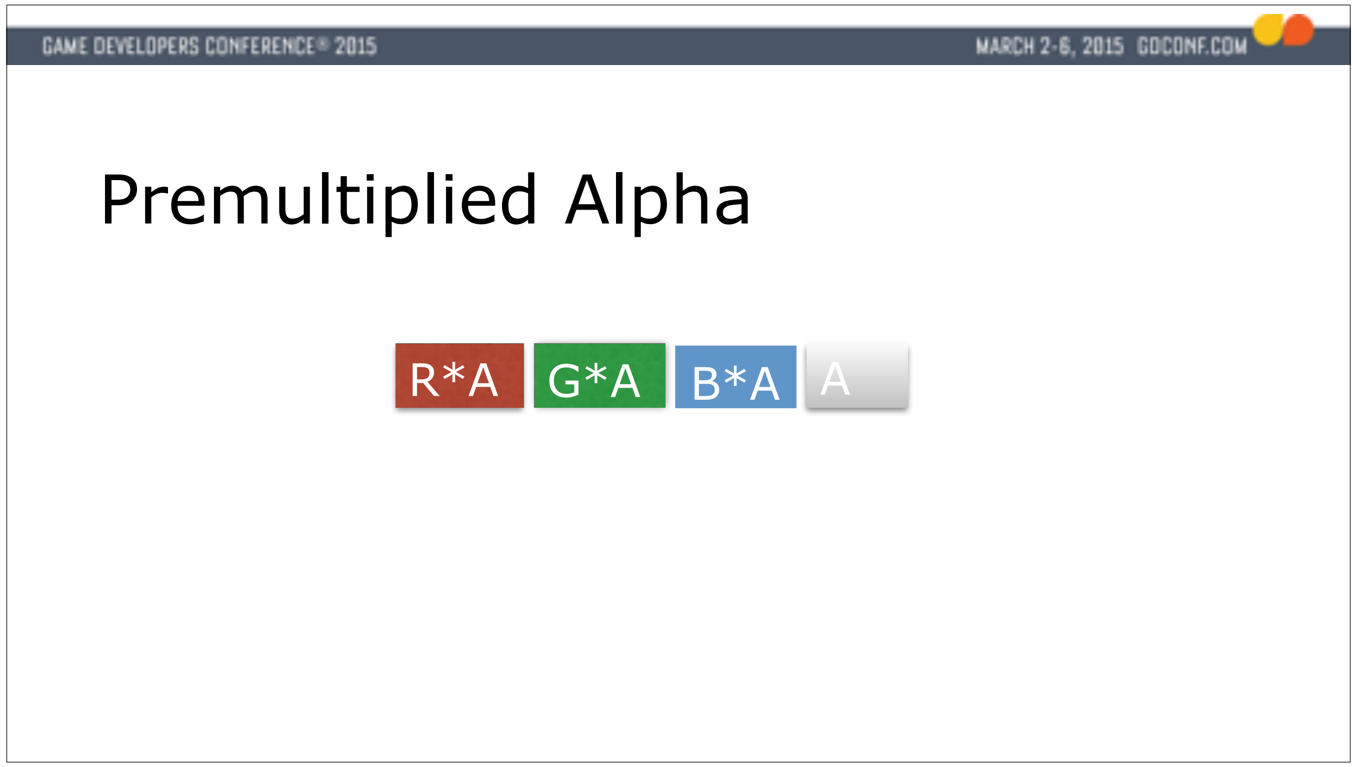
The standard formula for alpha transparency is this. Here the capital Cs mean the RGB values, and the subscripts ’s’ and ‘d’ mean ‘source’ and ‘destination’ respectively. We are drawing the source on top of the destination.



Alpha



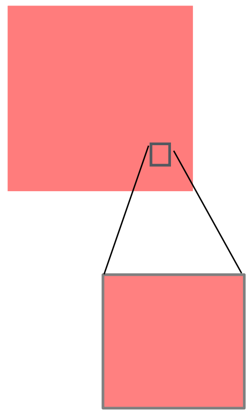
However, we can also consider alpha as coverage, for example for antialiasing. In this case, the square contributes 50% red to the value of the pixel. So a more accurate way of thinking about alpha is as contribution to a final pixel value. Whether we think of it as opacity or coverage, the final result is the same.



Premultiplied Alpha

The way alpha is most often used in games — particularly with alpha transparency — is just an additional channel, stuck onto the RGB values without modifying them. However, it is far better to premultiply the RGB values by the alpha — this is called pre-multiplied alpha.

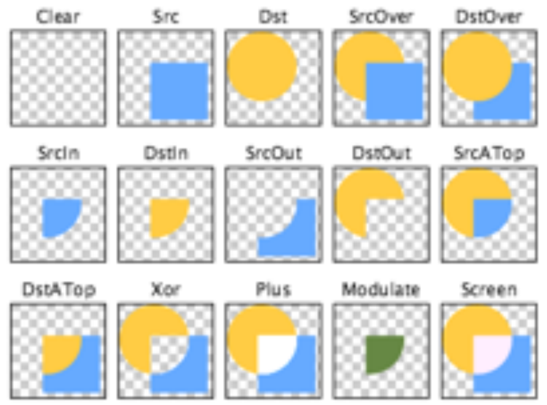
Alpha



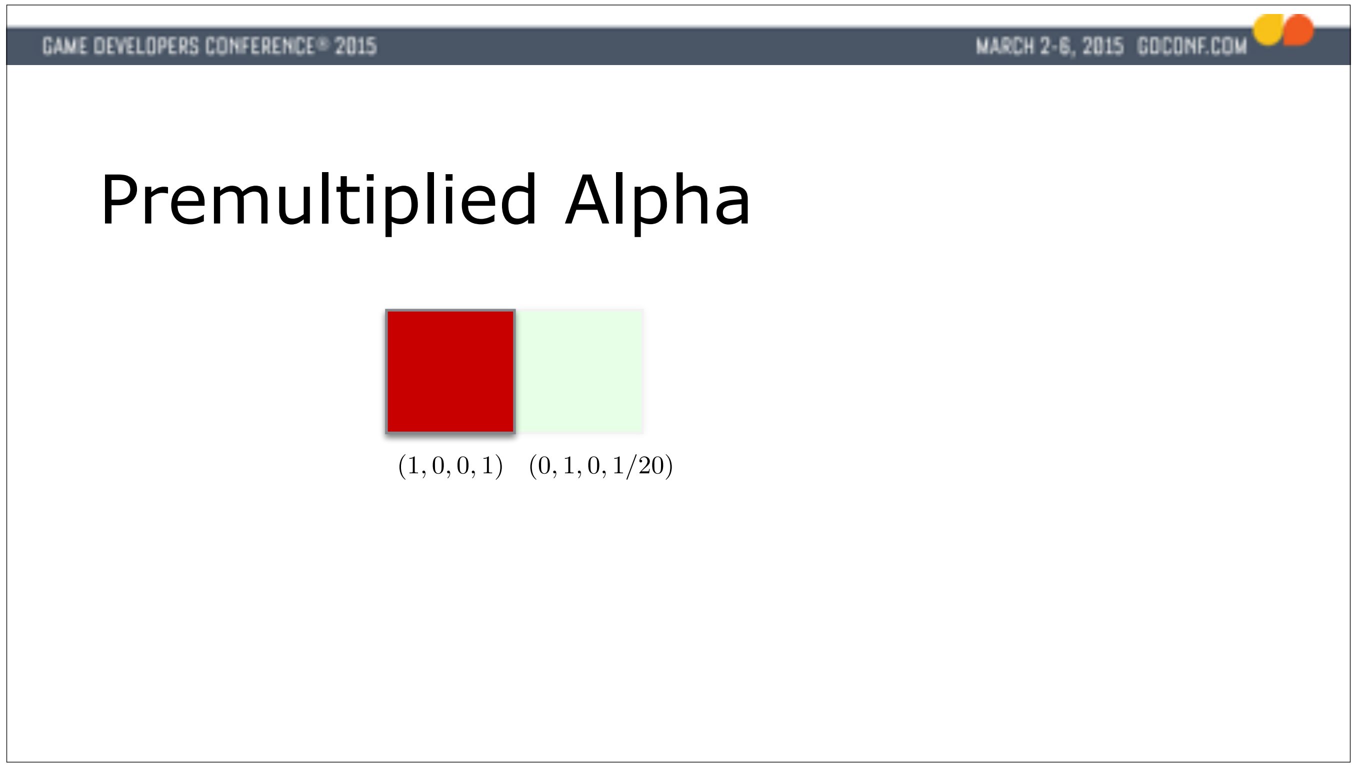
$$(1 - \alpha_s)(\alpha_d C_d) + (\alpha_s C_s)$$

Using premultiplied alpha, our formula for alpha transparency is now this. This is known as source over, and using that name makes a little more intuitive sense. We are taking a source — transparent or not — and placing it over the destination. Note that both Cd and Cs have been premultiplied by their alphas. If Cd is transparent, this will produce a slightly different result than the traditional method — but the result will be more correct.

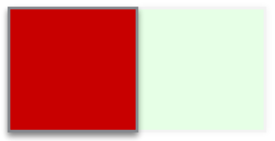
Premultiplied Alpha



One advantage of premultiplied alpha is that by combining it with the built-in blending modes in your graphics API, you get a broad range of possible blending modes known as the Porter-Duff blending modes (so named after the authors of the original paper on alpha compositing). This will make your UI artists happy as it will allow them many interesting effects.

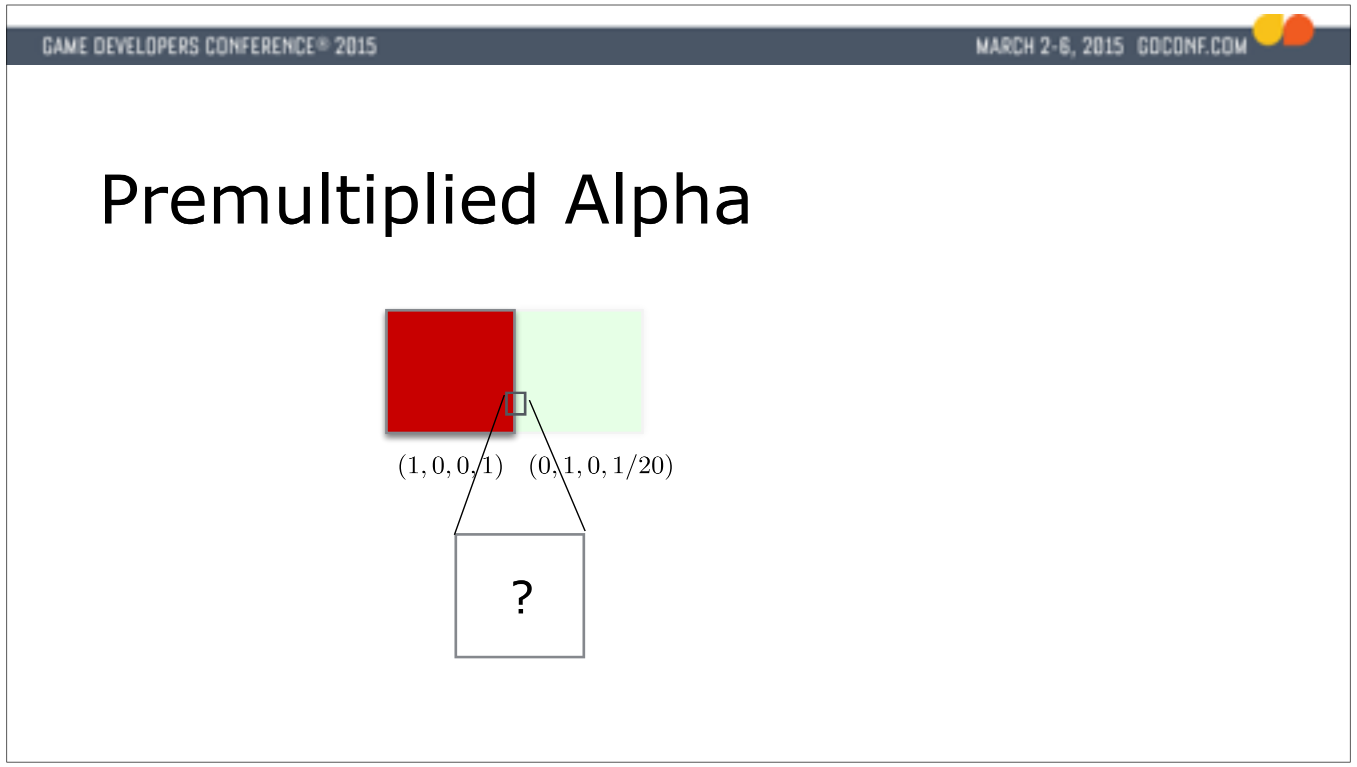


Premultiplied Alpha

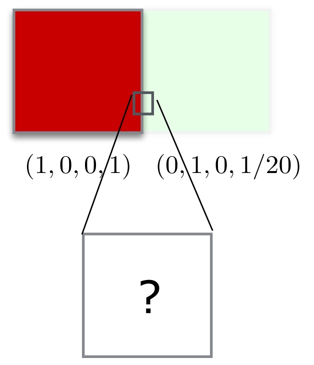


$(1, 0, 0, 1)$ $(0, 1, 0, 1/20)$

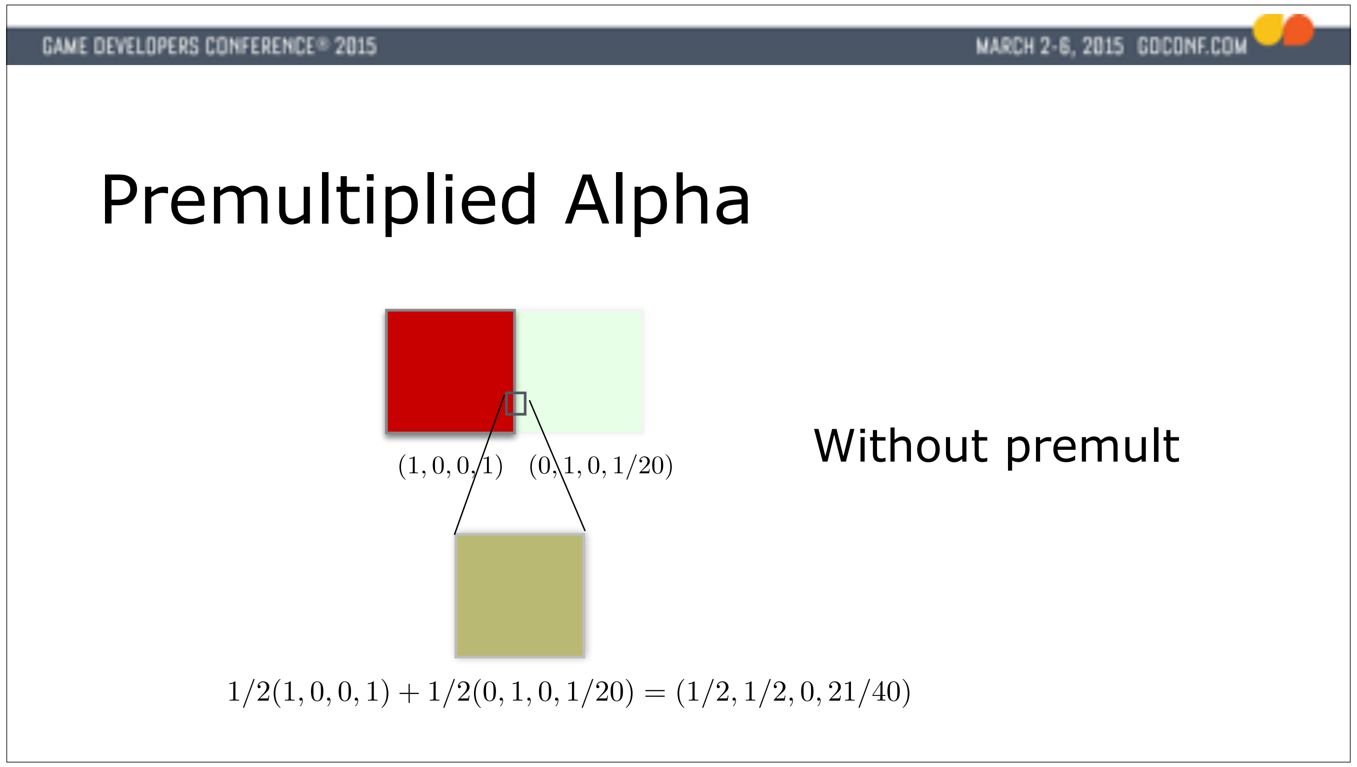
But even if all you’re doing is standard alpha transparency, premultiplied alpha is a win. Suppose we have a texture with a solid red pixel next to a transparent green pixel. This is an unlikely case, but bear with me.



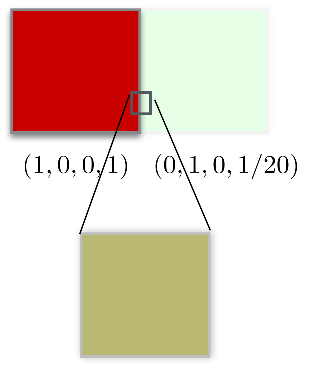
Premultiplied Alpha



Suppose our texture is placed so that a pixel lies across the border between the colors, splitting the pixel between solid red and transparent green. With bilinear filtering, we'd just take an average of the two texels. Using normal alpha transparency…



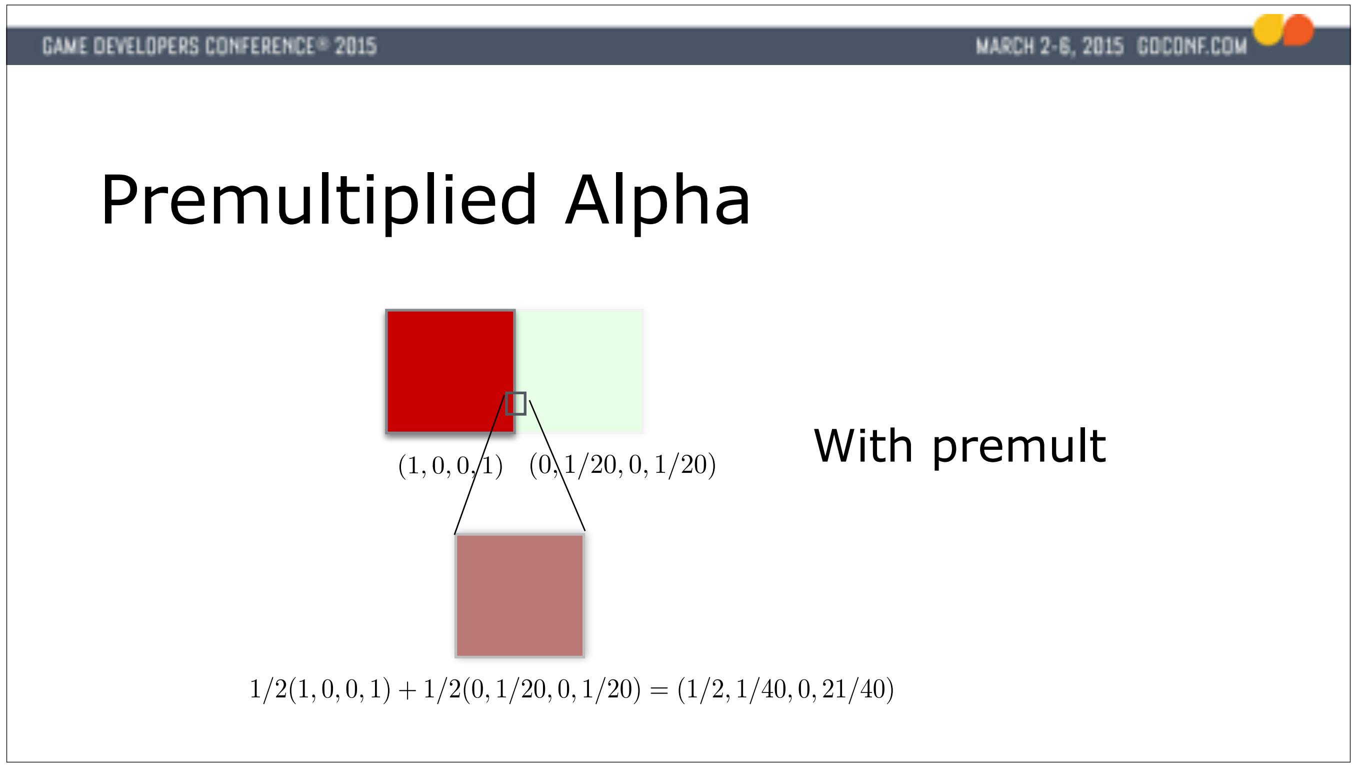
Premultiplied Alpha



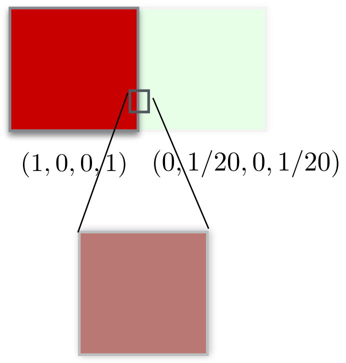
Without premult

$$1/2(1,0,0,1) + 1/2(0,1,0,1/20) = (1/2,1/2,0,21/40)$$

… we'd end up with this olive transparent color, which is not what we want.



Premultiplied Alpha

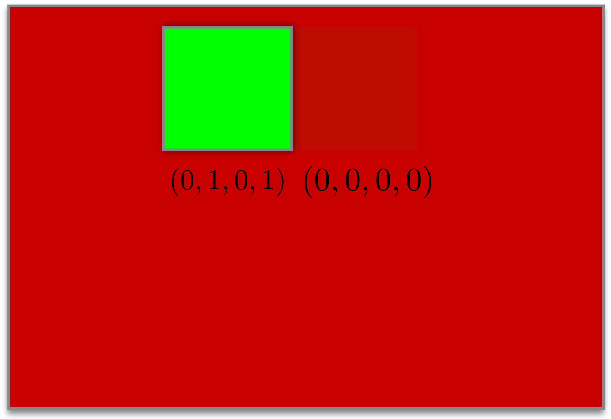


With premult

$1/2(1,0,0,1)+1/2(0,1/20,0,1/20)=(1/2,1/40,0,21/40)$

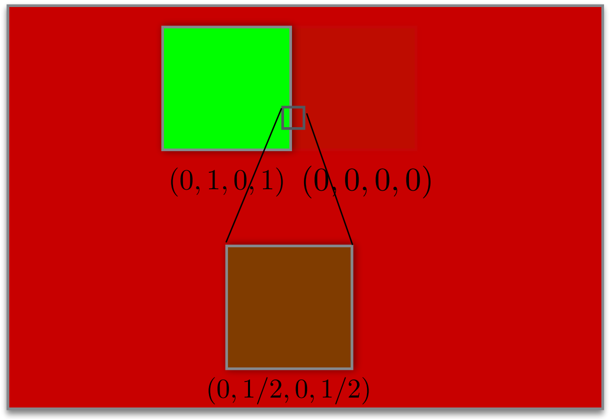
If we instead use premultiplied alpha, then we considerably reduce the contribution of the green pixel to the blend, and we end up with a much more reasonable result. Now, this is an extreme example, but even with more reasonable colors, you won’t quite get the result you want unless you use premultiplied alpha.

Premultiplied Alpha



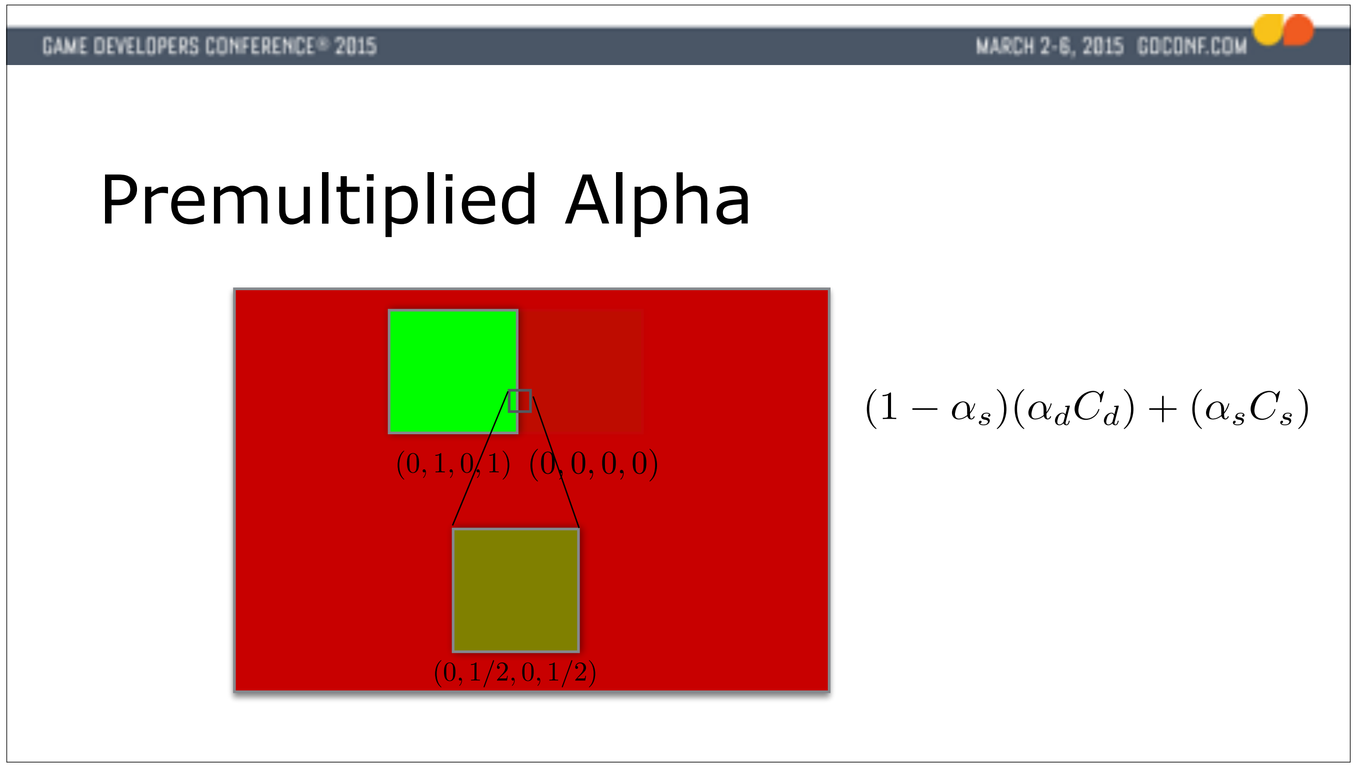
So our previous example was a bit ridiculous — why would we place red next to green in our texture and make the green transparent? Okay, let’s say we have a texture with opaque green and transparent black — say a leaf texture. A reasonable choice, we’d think. We want to overlay this on a red background.

Premultiplied Alpha

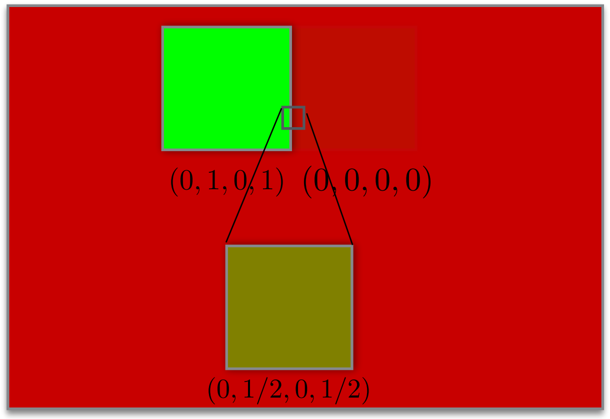


$$(1-\alpha_s)C_d + \alpha_s C_s$$

Now again our pixel contains half of each color. So we average the two texel values and use our standard alpha transparency blending and end up with this — much darker than we expected.

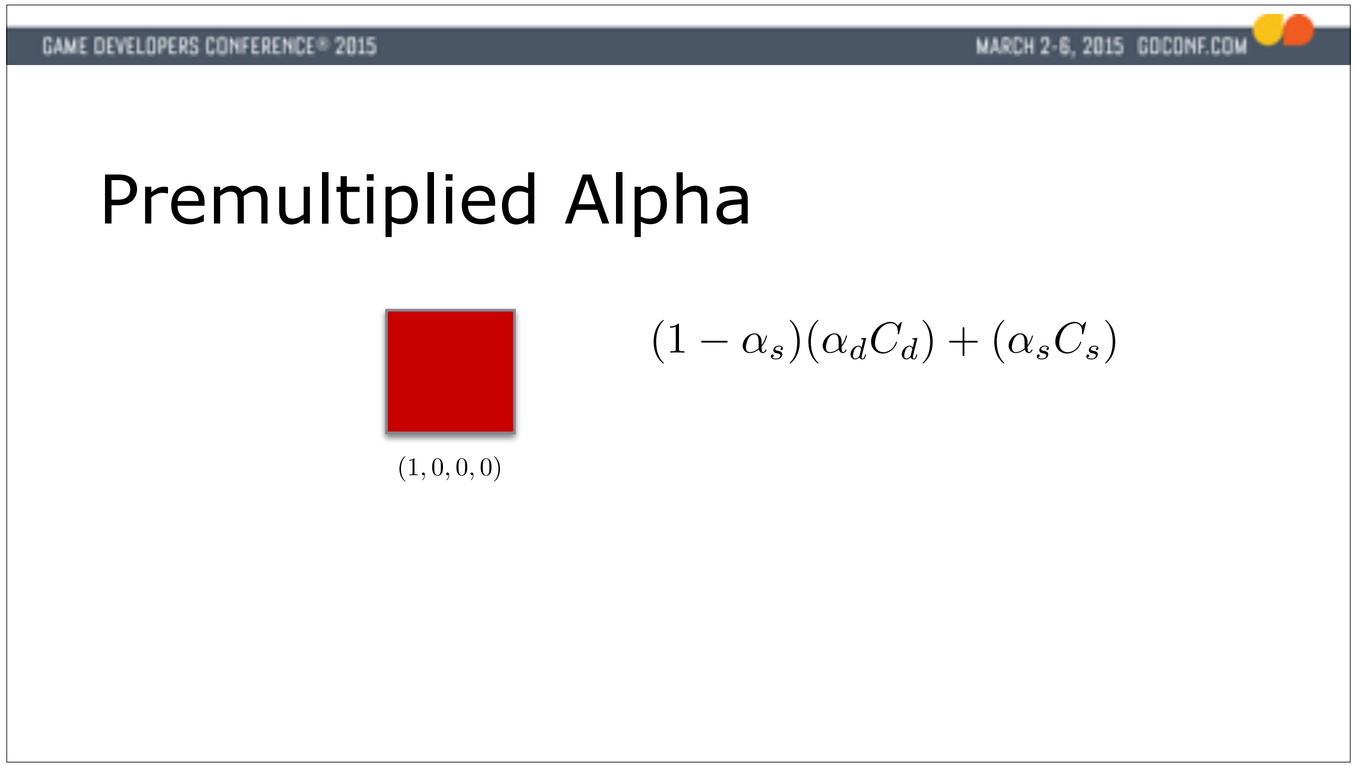


Premultiplied Alpha



$$(1-\alpha_s)(\alpha_d C_d)+(\alpha_s C_s)$$

However, if we treat them as premultiplied alpha colors and use the SrcOver equation, we get the correct result.

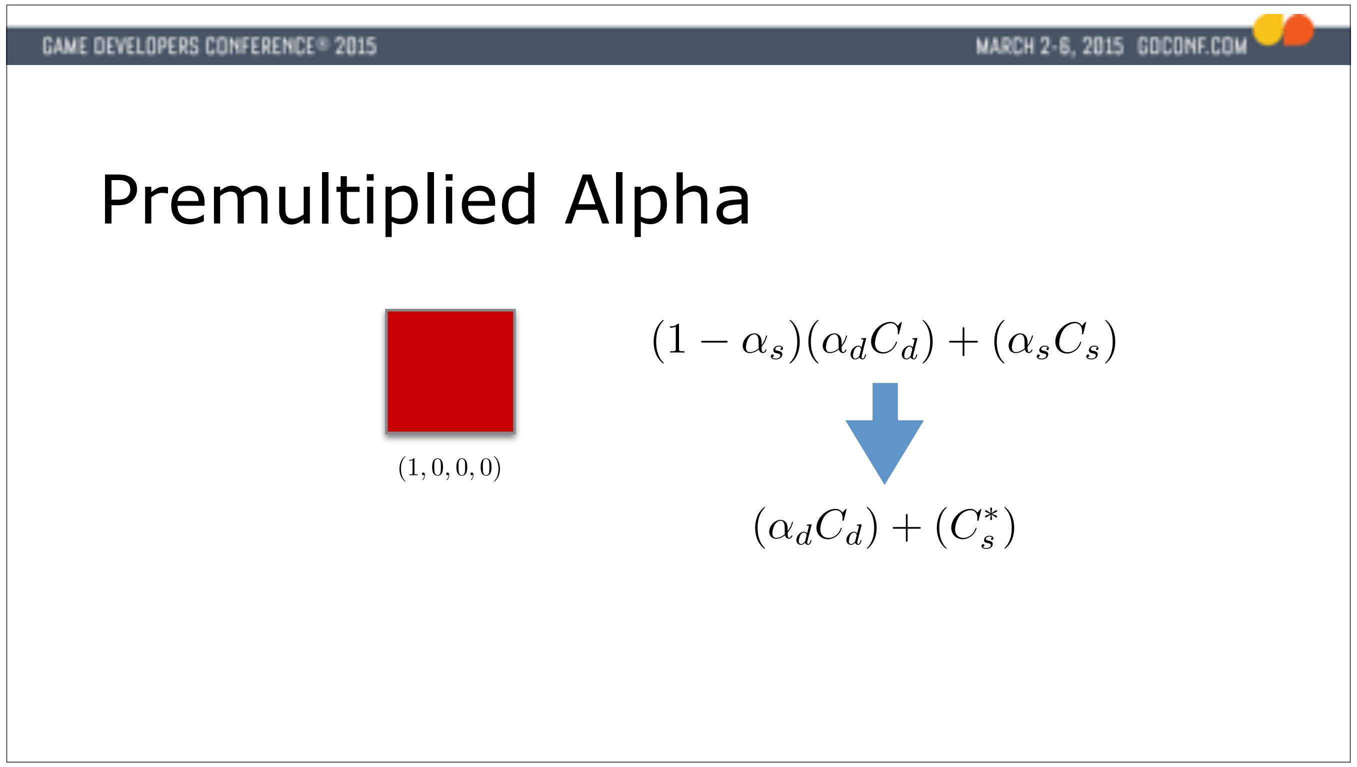


Premultiplied Alpha

$(1, 0, 0, 0)$

$$(1 - \alpha_s)(\alpha_d C_d) + (\alpha_s C_s)$$

This next little trick comes from Tom Forsythe’s blog. Now suppose, we take a color with a zero alpha channel. Normally with premultiplied alpha any color with a zero alpha would be all zeros, or transparent black. But if we set the RGB values to something non-zero and apply our SrcOver equation….



Premultiplied Alpha

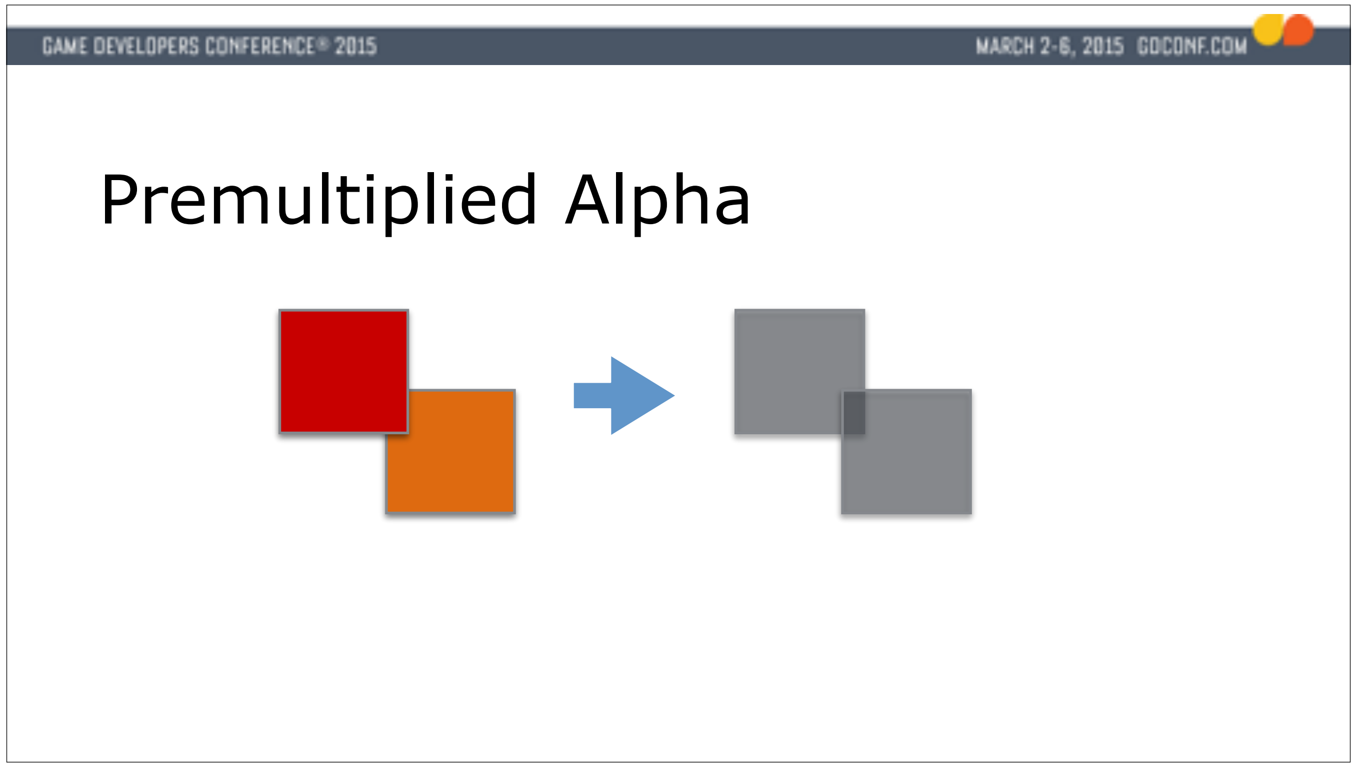
$(1, 0, 0, 0)$

$$(1 - \alpha_s)(\alpha_d C_d) + (\alpha_s C_s)$$

$$\downarrow$$

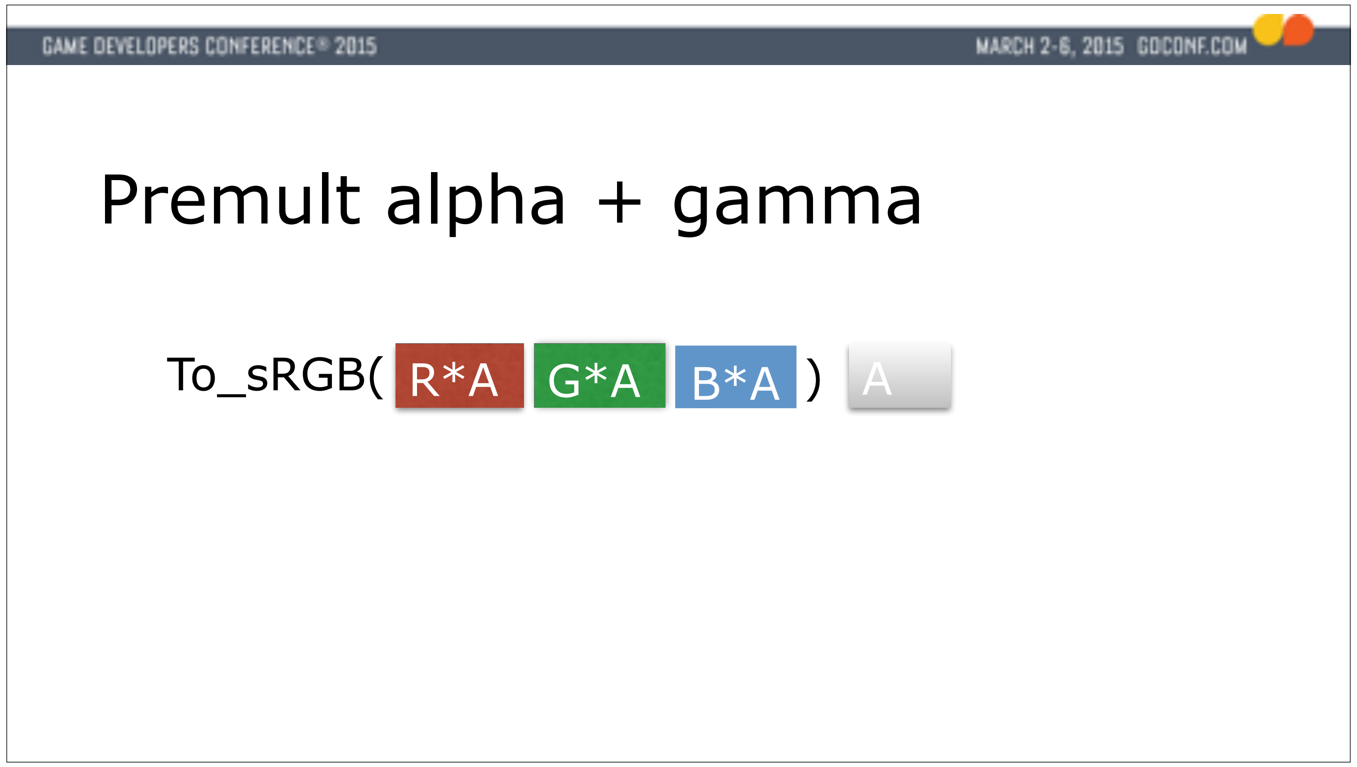
$$(\alpha_d C_d) + (C_s^*)$$

we end up with additive blending. What this means is that we can mix colors that are ended to have transparency with colors that are intended to have additive blending, and use the same shader and draw call.



Premultiplied Alpha

So for example, we could have particles that represent sparks or fire, and have them start out with additive colors, and then as they decay, have them change to transparent colors. No need to sort out which is which at rendering time — just rendering them all in one call.

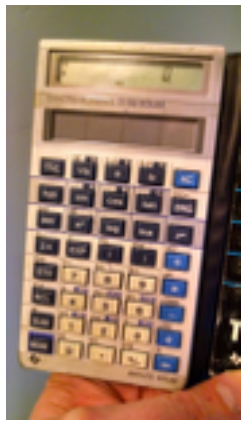


Premult alpha + gamma

To_sRGB(R*A G*A B*A) A

One note on using the sRGB or any other gamma correction curve with premultiplied alpha — be sure to gamma correct your colors after the premultiplication. Otherwise you won't get the correct result when you convert back to linear for blending or other linear operations. Also, alpha is a linear variable — so don't apply gamma correction to it.

Summing up



Be careful of out-of-gamut issues
Gamma correct and handle conversions
Use pre-multiplied alpha!

References

- Charles Poynton Color FAQ
- Porter & Duff, “Compositing Digital Images”
- Tom Forsythe’s blog

?

jim@essentialmath.com
Twitter: @cthulhim
G+: vintagejim