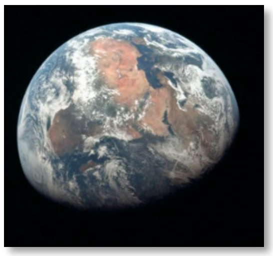
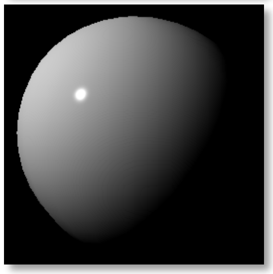
Gamma Correction

Ensuring our pixel colors are ready for display

When Digital Imagery Goes Wrong





We might assume that after shading, our pixel work is done: calculate lights, add them together, return the total. Seems easy, right? Unfortunately, our results are not ideal just yet, at least as far as our eyes are concerned.

The photo and render to the right each showcase light hitting a sphere. The two images have a subtle difference in their shading: specifically, the falloff or change in brightness (diffuse result) across the surface. The image of Earth is mostly the same brightness with a very quick falloff near the rim. The sphere render below it exhibits a smoother gradient, which doesn't match the photo.

Have we done something wrong in our calculations? Nope; they're perfectly suitable approximations. We'll be discussing more *physically accurate* approximations soon, but they'll suffer the same issue.

Are we combining multiple lights incorrectly? Nope; two lights *should* be twice as bright. This is a linear combination, which is appropriate for our purposes.

Is our input data inadequate? Nope; we've defined our input – surface colors, light colors and so forth – correctly. Our colors are also linear, where twice the color value means twice the brightness.

So, what's actually wrong? We've made a big assumption about how digital displays, like our monitors and cell phone screens, actually work: that the color we return from the pixel shader is the exact color of physical light that will eventually reach our users' eyeballs. As it turns out, this isn't the case! Meaning: we're not compensating for how digital display devices actually operate (at least not yet).

Digital Image Assumptions

Let's look at some of the assumptions we might make when it comes to digital images:

Assumption 1: When a digital camera captures an image, doubling the amount of incoming light will double the color value of the stored pixel. In other words:

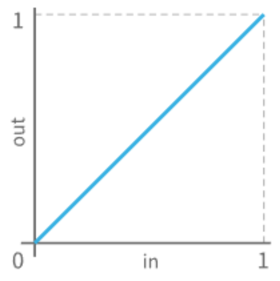
If an amount of light produces pixel color (0.5, 0.5, 0.5), then twice that light produces (1, 1, 1)

Assumption 2: When a digital screen displays an image, doubling a color value will cause the corresponding pixel of the monitor to produce twice as much light. In other words:

A pixel with color (1, 1, 1) will cause the monitor to produce twice as much light as (0.5, 0.5, 0.5)

Both of these assumptions are, generally, wrong! And herein lies our problem: our calculations are producing the correct values, but when viewed on a monitor, they don't *appear* correct. Now, you might be thinking "They look fine to me!" While they probably do look *fine*, they're not *correct*.

Linear Space



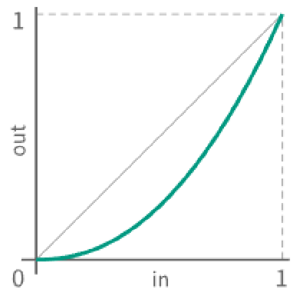
You'll be seeing the term "linear" a lot, so let's see it graphed before going further. If we were to lay out values of a linear set of data, we would see a diagonal line, similar to the graph on the right.

The equation of this graph is simply: $out = in$

In "linear space", doubling an input value doubles the corresponding output. An input of 0.1 gives an output of 0.1. An input of 0.2, therefore, gives an output of 0.2.

Display Devices & Gamma

In general, digital displays, whether they are old-school cathode-ray tubes (CRT), liquid crystal displays (LCD), organic light-emitting diodes (OLED) or another technology, exhibit a non-linear *response* to their input data. "Response" here means the actual output color. In other words: the color we send *to the monitor* is not the same as the color coming *out of the monitor*.



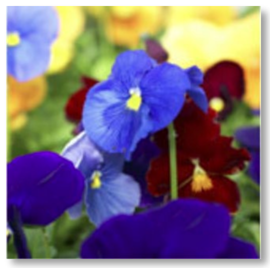
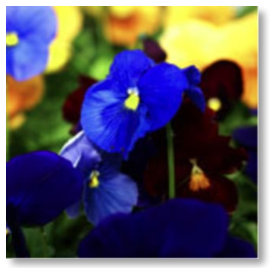
If we were to graph the input values (pixel colors) and output values (actual, physical light) of an average digital display, it would look a lot like the graph to the right.

In this graph, we can see that inputs of 0 and 1 (black and white) produce outputs of 0 and 1, respectively. However, all values in between are lower (darker) than they should be, especially the midtones.

Note that this is different than a simple brightness adjustment, which would equally darken all output (the line would remain straight, but be lower).

This curve is known as the *gamma curve* of a monitor. In this graph, the output is proportional to the input raised to a power: $out = in^{gamma}$ The power in this equation is known as the *gamma characteristic,* or just *gamma*, of the display device. While every device is slightly different, digital displays have, on average, a gamma of 2.2. In other words, we can fairly accurately describe how digital colors will look to our eyes by raising those colors to the 2.2 power: $out = in^{2.2}$

Gamma Example

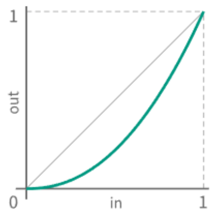
Let's say we took a raw digital picture of flowers (the left-most image). If we were to view this, unaltered, on a digital display device, it might look like the right-most image: darker than intended.

But wait, we ARE looking at this on a digital display. Why does the left image look fine? Because it's been *gamma corrected*.

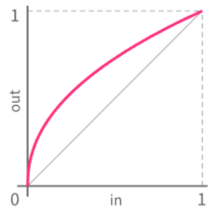
Gamma Correction

When a device, like a digital camera, or a photo-editing application, like Photoshop or even Paint, saves an image, they generally apply a correction to each and every pixel: they're brightened. Specifically, a power curve is applied to the color values to account for the fact that these digital images will most likely be shown on a digital display with a non-linear response.

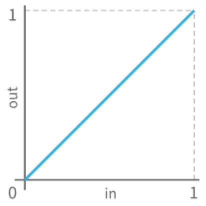
In other words, the image is artificially brightened when saved so that, when a monitor displays it (darker than intended, which we know it will do), the resulting colors look correct. How does the device or application account for monitor gamma characteristics? How much "brighter" should the colors be?



We know that monitors, on average, have a gamma of 2.2: $out = in^{2.2}$ This results in the graph and "darker than normal" image to the left. This will always happen on a digital display.



To combat this, we need to first brighten the pixels with a curve that will "cancel out" the gamma curve, as shown in the graph and "brighter than normal" image to the left. This happens automatically in digital cameras and photo-editing software (though there may be options to turn it off).

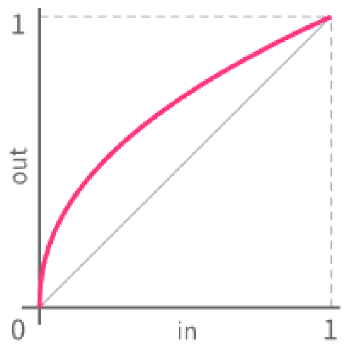


Ideally, when the monitor's gamma curve is applied to our artificially brightened curve, the resulting color values are once again linear. To put it another way, we need to first raise to a power that the 2.2 power will "undo":

$$out = \left(in^{???}\right)^{2.2}$$

What power will "undo" the 2.2 power? Let's look at a more obvious problem first: what power would undo a square? A square root. Squaring a number is raising it to the power of 2. Taking a square root is equivalent to raising to the ½ power (or 0.5).

$$x = (x^2)^{\left(\frac{1}{2}\right)} \quad \text{and} \quad x = \left(x^{\left(\frac{1}{2}\right)}\right)^2$$



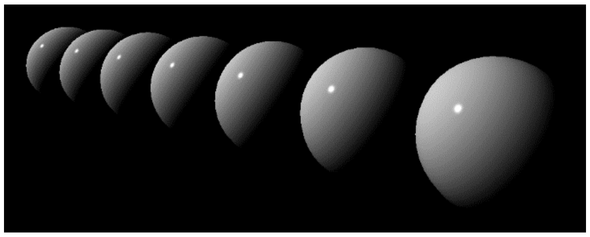
Thus, to undo the monitor's effect of raising to the 2.2 power, each pixel's color value is raised to the $\left(\frac{1}{2.2}\right)$ power before being saved.

$$out = in^{\left(\frac{1}{2.2}\right)}$$

This is a standard, but non-obvious, process that is taken care of for us across all of our devices and applications. However, since we're in charge of our exact rendering output, we've got to do it ourselves.

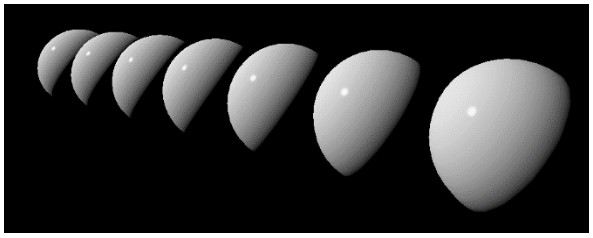
Gamma & Rendering Output

Okay, so monitors can't be trusted. What does this have to do with us? Let's look at our current output:



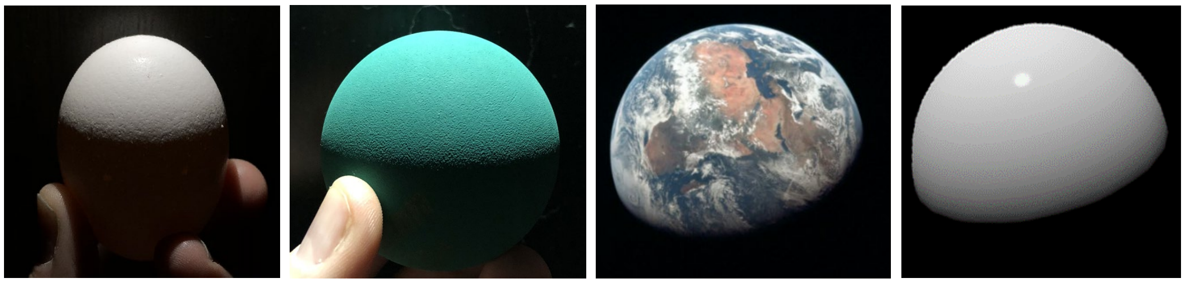
This looks fine, right? Unfortunately, it's wrong. Not the math, but the results on our monitors.

Here is the corrected version. It certainly looks *different*, but is it truly *correct*? Is it photorealistic? How would we know?



Let's compare to actual photographs.

Want Photorealism? Look at Some Photos!



The images from left to right: an egg, a ball, Earth and our gamma-corrected sphere. Each is lit by a single light source (or as close as I could get to one). The top of each shape is brightest. The brightness diminishes very slowly until we reach the edge, where it very quickly darkens.

Gamma Correction in Shaders

How do we handle gamma correction in our shaders? We need to apply the exact same compensation (artificial brightening) to the colors we intend to show to the user, since our monitors will absolutely make them darker than intended. Note that this only needs to be done to the RGB channels.

In code, we simply raise our final color to the proper power before returning it from the pixel shader:

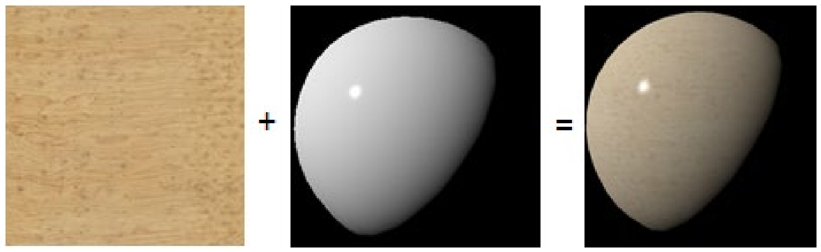
```
	float3 gammaAdjustedColor = pow(totalColor, 1.0f / 2.2f);
```

Do this at the very end of your pixel shader, after all lighting calculations. That's it*!

**Unless we're using textures…*

Gamma Correction with Textures

Our quick and easy fix for monitor gamma works great when we're applying it to solid-colored objects. Unfortunately, there's an issue if we're using textures as part of our shading. For example, notice how the texture looks too bright or washed out on the sphere below. What's going on here?



Textures are Already Gamma Corrected

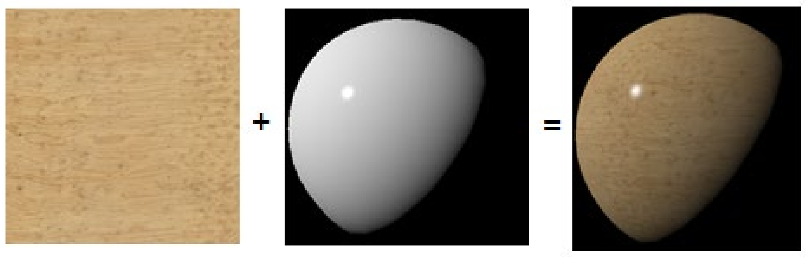
Remember that our textures have already been gamma corrected by the software that created them! They've already had their colors raised to the (1/2.2) power, meaning those colors are no longer in linear space. However, our math assumes linear colors (twice the value = twice as bright). We're raising our results to the (1/2.2) power, we're essentially brightening an already-partially-brightened image.

What's the fix? Get those texture colors back into linear space *before* using them as part of our lighting calculations by raising them to the 2.2 power. We need to *un-correct* their gamma. It was assumed this would be darkened by the monitor; we need to approximate that darkening ourselves!

This should be done as soon as you sample the texture in the shader so that no other changes are necessary throughout our lighting code. Again, only apply this to the RGB channels, as the alpha channel is probably already linear. (Though we aren't using it yet).

```
        float3 textureColorLinear = pow(textureColor, 2.2f);
```

As we can see below, this results in a much more accurate texture color on our objects.



Which Textures Should be Un-Corrected?

Only textures that represent *surface colors* (colors used in our lighting calculations). For instance, you generally shouldn't gamma correct normal maps, as the software that creates them specifically keeps the colors linear on purpose. What about specular maps? It depends on how they were created. If you got them as part of a set of textures online, they're potentially already linear.

All Together Now

To properly account for the gamma characteristics of our digital display devices, you'll need to follow these steps in any pixel shaders that perform lighting calculations:

- Apply a 2.2 power curve to any texture samples that represent surface colors
- Perform your lighting calculations as normal
- Apply a 1/2.2 power curve to the final color returned from the shader

Note that shaders which simply sample and return a texture color as-is, such as our skybox pixel shader, do not need to perform any gamma correction. They're simply passing the color through.

Automatic Gamma Handling with sRGB

If these gamma issues are so common, why can't GPUs just handle it for us? As it turns out, they can!

GPUs support a special texture color format called sRGB, or "Standard Red Green Blue". When a GPU samples from an sRGB texture, it automatically applies the 2.2 power curve to get the colors back into linear space. There are even advanced texture loading functions in the DirectX Toolkit that can force the resulting texture to be sRGB.

sRGB was originally developed by Hewlett-Packard & Microsoft in the mid 1990's and standardized by the International Electrotechnical Commission in 1999.

On the flip side, if our render target (like the back buffer) uses the sRGB color format, the GPU will automatically apply the 1/2.2 power curve to any colors it writes into that texture. Our starter code is not explicitly using sRGB, though DXCore could be tweaked to create an sRGB back buffer if we wanted.

Further reading

If you're interested in diving deeper into this topic, below are a few links that you may find useful.

- The Importance of Being Linear from GPU Gems 3
- Understanding Gamma Correction for photographers from Cambridge in Colour
- Linear-Space Lighting from John Hable, rendering programmer at Unity (formerly Naughty Dog)