GAMMA CORRECTION

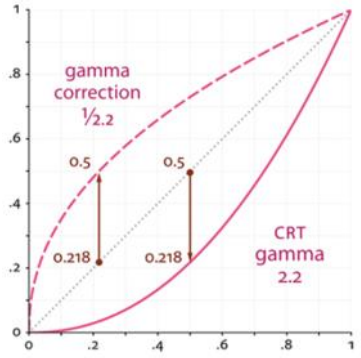
How humans measure brightness is close to an inverse power relationship of 2.2 $\left(\frac{1}{2.2}\right)$. This can be observed in the image to the right.

- The top line looks more like how brightness should scale to the human eye. Doubling the brightness (e.g. from 0.1 to 0.2) looks like it's twice as bright with consistent differences.
- However, in regards to the physical brightness of light (e.g. the amount of photons leaving a light source), the bottom scale actually displays the correct brightness.
  - Since our eyes perceive brightness differently (because they're more susceptible to changes in dark colors), the bottom line does not appear consistent.
- Because human eyes prefer to see brightness colors according to the top scale, monitors use a power relationship for displaying output colors so that the original physical brightness colors are mapped to the non-linear brightness colors in the top scale.

The non-linear mapping of monitors does output more pleasing brightness results for our eyes, but there's one particular issue that occurs when rendering graphics:

- All the color and brightness options we configure in our applications are based on what we perceive from the monitor and thus all the options are actually non-linear brightness/color options.

To better understand the issue above, let's take a look at the graph to the right.



- The dotted line represents color/light values in linear space and the solid line represents the color space that monitors display.
- If we double a color in linear space, its result is exactly double the value.
  - Example: If we take a light's color vector `(0.5, 0.0, 0.0)`, which represents a semi-dark red light, and double it in linear space, it would be `(1.0, 0.0, 0.0)`.
    - However, the original color gets displayed on the monitor as `(0.218, 0.0, 0.0)`, as you can see from the graph.
    - This is where the issues start to rise. Once we double the dark-red light in linear space, it actually becomes more than 4.5 times as bright on the monitor (since it's technically going from a red component of `0.218` to `1.0`.)

Since we've been under the assumption that we were working with colors in linear space, the colors we've been outputting through the monitor haven't been physically correct. Because of this, we (and artists) generally set lighting values way brighter than they should be (since the monitor darkens them), which, as a result, makes most linear space calculations incorrect.

- **NOTE:** The monitor (CRT) and linear graph both start and end at the same position; it's the intermediate values that are darkened by the display.
- The discrepancy of linearly-correct and physically-correct lighting calculations becomes more apparent as more advanced lighting algoriths are in place, as you can see in the images from the game, Overgrowth, to the right.
  - Notice how the gamma-corrected color values work more nicely together and darker areas show more details. Overall, the image quality is better.
  - Without properly correcting the monitor gamma, the lighting looks wrong and artists have a hard time getting realistic and good-looking results.

The solution to these lighting issues is to apply **gamma correction**.

Gamma Correction

The idea of gamma correction is to apply the inverse of the monitor's gamma to the final output color before displaying to the monitor, essentially cancelling out the monitor's gamma. This result is the color adjustments becoming linear.

- Example: If we look at the same dark-red color vector `(0.5, 0.0, 0.0)` from before, before displaying this color to the monitor we apply the gamma correction curve to the color value. Linear colors displayed by a monitor are roughly scaled to a power of `2.2`, so the inverse requires scaling the colors by a power of `1/2.2`. When the corrected colors are output by the monitor, the result then becomes that gamma-corrected color to a power of `2.2`, which brings the visible color back to `(0.5, 0.0, 0.0)`.
  - Original Color: $(0.73, 0.0, 0.0)$ ===> Gamma-Corrected Color: $(0.5, 0.0, 0.0)^{\frac{1}{2.2}}$ --> $(0.73, 0.0, 0.0)$ ===> Displayed Color: $(0.73, 0.0, 0.0)^{2.2}$ --> $(0.5, 0.0, 0.0)$

NOTE: A gamma value of 2.2 is a default value used in gamma correction that roughly estimates the average gamma of most displays. The color space as a result of this gamma of 2.2 is called the **sRGB** color space (not 100% exact, but close). Each monitor has their own gamma curves, but a gamma value of 2.2 gives good results on most monitors. For this reason, games often allow players to change the game's gamma setting as it varies slightly per monitor.

There are two ways to apply gamma correction to your scene:

1. By using OpenGL's built-in sRGB framebuffer support.
2. By doing the gamma correction ourselves in the fragment shader(s).

NOTE: Keep in mind with the following approaches that gamma correction also transforms the colors from linear space to non-linear space, so it is very important that you only do gamma correction at the last and final step. If you gamma-correct your colors before the final output, all subsequent operations on those colors will operate on incorrect values and, thus, yield incorrect results.

- Example: If you use multiple framebuffers, you probably want intermediate results passed in between framebuffers to remain in linear space and only have the last framebuffer apply gamma correction before being sent to the monitor.

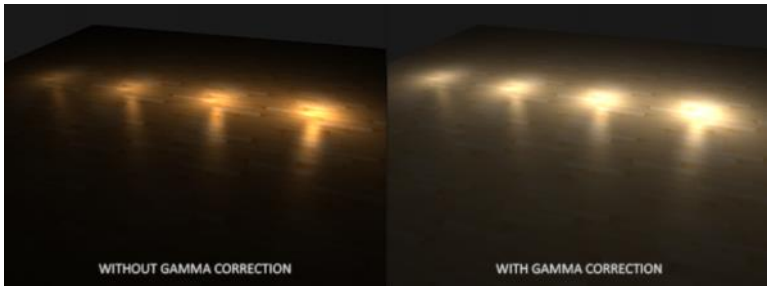
The first approach is the easiest, but gives you less control.

- You can enable sRGB framebuffers with `glEnable(GL_FRAMEBUFFER_SRGB)`.
- When enable sRGB framebuffers, you are telling OpenGL that each subsequent drawing command should first gamma correct colors (from the sRGB color space) before storing them in color buffer(s). This means that OpenGL will automatically perform gamma correction after each fragment shader run to all subsequent framebuffers, including the default framebuffer.
- As this is handled by the hardware, it is completely free.

The second approach requires more work, but also gives you complete control over the gamma operations.

- We apply gamma correction at the end of each relevant fragment shader run so the final colors end up gamma-corrected before being sent out to the monitor.
- Here is an example of how you would apply gamma correction in the fragment shader.
- An issue with this approach is that, in order to remain consistent, you have to apply gamma correction to each fragment shader that contributes to the final output. If you have a dozen fragment shaders for multiple objects, you have to add the gamma correction code to each of these shaders.
  - An easier solution would be to introduce a post-processing stage in your render loop and apply gamma correction on the post-processed quad as a final step, which you'd only have to do once.

sRGB Textures

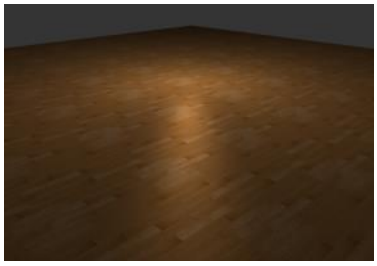


Because monitors display colors with gamma applied, whenever you draw, edit, or paint a picture on your computer, you are picking colors based on what you see on the monitor.

- This effectively means that all the picture you create or edit are not in linear space, but in sRGB space (e.g. doubling a dark-red color on your screen based on perceived brightness does not equal double the red component).
- As a result, when texture artists create art by eye, all the textures' values are in sRGB space.
  - This is why the textures themselves looked fine in our application up to this point. However, if we look at them after having applied gamma correction, the texture colors will be off.

In the image to the right, you'll notice that the gamma-corrected texture is way too bright. That is because it is actually being gamma-corrected twice: once when the texture is created by the artist, and again when it is rendered with gamma correction.

- To solve this issue, we can't rely on texture artists working in linear space, so we must transform these sRGB textures to linear space before doing any calculations on their color values.
  - Technically, you could do this manually, like so, but you would have to do this for each texture in sRGB space, which can be incredibly cumbersome.
  - However, OpenGL provides us with the GL_SRGB and GL_SRGB_ALPHA internal texture formats, which we can sub in to solve this issue automatically.
    - **NOTE:** You must be careful when specifying your textures in sRGB space, as not all textures will actually be in sRGB space.
      - Example: Diffuse textures are almost always in sRGB space, but textures used for retrieving lighting parameters (e.g. specular maps, n ormal maps), are almost always in linear space.

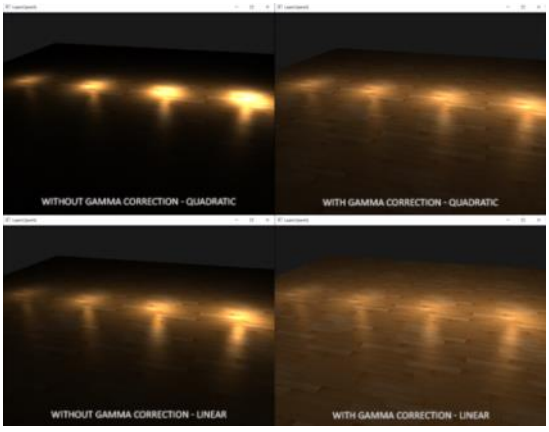


Implement gamma correction in your scene. Then, fix the overcorrection of texture colors.

- For the time being, you can just specify the internal format of the `glTexImage2D` calls that use `GL_RGB` and `GL_RGBA` as their internal formats to use `GL_SRGB` and `GL_SRGB_ALPHA` instead, respectively. Since our specular map is fully white, it will be unaffected, but remember the note above.
- Your result should look like the image to the right.
  - **NOTE:** I increased the shininess of the material from `0.5` to `32.0` and lowered the light's y-position to `1.0`.
- Adding Gamma Correction Manually

Now, everything is gamma corrected only once.

Attenuation



Something else that's different with gamma correction is lighting attenuation.

- In the real world, lighting attenuates closely inversely proportional to the squared distance from a light source. In other words, the light strength is reduced over the distance to the light source squared, as we've been calculating it.
- However, when using the equation we have in the past, the attenuation effect is usually way too strong, giving lights a small radius that doesn't look physically right.
  - For this reason, other attenuation functions were used (as discussed in the basic lighting chapter) that give more control, or the linear equivalent: $attenuation = \frac{1.0}{distance}$
  - Take a look at the image to the right. While the linear equivalent (without gamma correction) yields more plausible results compared to the quadratic variant, it looks too weak when gamma correction is enabled, compared to the physically correct quadratic attenuation, which looks better.
    - This is because, if you think about it, the linear attenuation without gamma correction is equivalent to $\left(\frac{1.0}{distance}\right)^{2.2}$ on the screen, which is very similar to its physical equivalent. However, when the color is gamma-corrected, the squared attenuation calculation remains as $\left(\frac{1.0}{distance}\right)^{2}$, which is why they look similar.

NOTE: The more advanced attenuation function we discussed in the basic lighting chapter still has its place in gamma-corrected scenes as it gives more control over the exact attenuation, but it will require different parameters.

To summarize, gamma correction allows us to do all our shader/lighting calculations in linear space. Because linear space makes sense in the physical world, most physical equations now actually give good results (like real light attenuation $\left(\frac{1.0}{distance}\right)^{2}$). The more advanced your lighting becomes, the easier it is to get good looking (and realistic) results with gamma correction. That is also why it's advised to only really tweak your lighting parameters as soon as you have gamma correction in place.

Simple Gamma Correction in Fragment Shader

Wednesday, May 25, 2022 5:26 PM

```
void main() {
   // Lighting calculations
   [...]

   // Gamma correction
   float gamma = 2.2;
   FragColor.rgb = pow(FragColor.rgb, vec3(1.0 / gamma));
}
```

The implementation is not that impressive, but there are other things you have to consider when doing gamma correction, as we discuss later in this chapter.

Manual Texture Color Re-Correction Example

Wednesday, May 25, 2022 5:58 PM

```
float gamma = 2.2;
vec3 diffuseColor = pow(texture(diffuse, texCoords).rgb, vec3(gamma));
```

This method or texture color correction when applying gamma correction is NOT recommended.

Adding Gamma Correction Manually

Wednesday, May 25, 2022 6:27 PM

Texture2D.h

```
class Texture2D {
public:
   ...
   Texture2D(std::string dir, int unit, GLenum sWrapping, GLenum tWrapping, bool gammaCorrect = false);
   ...
};
```

Texture2D.cpp

```
if (texData) {
   switch (numChannels) {
      ...
      case 3:
         gammaCorrect ? glTexImage2D(GL_TEXTURE_2D, 0, GL_SRGB, width, height, 0, GL_RGB, GL_UNSIGNED_BYTE, texData)
             : glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width, height, 0, GL_RGB, GL_UNSIGNED_BYTE, texData);
         break;
      case 4:
         gammaCorrect ? glTexImage2D(GL_TEXTURE_2D, 0, GL_SRGB_ALPHA, width, height, 0, GL_RGBA, GL_UNSIGNED_BYTE, texData)
             : glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, width, height, 0, GL_RGBA, GL_UNSIGNED_BYTE, texData);
         break;
      ...
   }
   ...
}
```

NOTE: You will need to load your diffuse textures with GL_SRGB or GL_SRGB_ALPHA regardless of whether you use the built-in sRGB framebuffers or your own gamma correction implementation.

main()

```
Texture2D planksDiffuseTexture{ "Textures/planks.png", 0, GL_REPEAT, GL_REPEAT, true };
Texture2D fullSpecularTexture{ "Textures/full_specular.png", 1, GL_REPEAT, GL_REPEAT };
```

NOTE: With how the `Texture2D` constructor is set up in the header file, you don't need to pass false for the `gammaCorrect` parameter since it is false by default.

blinn-phong.frag

```
void main() {
   ...

   FragColor = vec4(color, 1.0);

   // Gamma correction
   float gamma = 2.2;
   FragColor.rgb = pow(FragColor.rgb, vec3(1.0 / gamma));
}
```