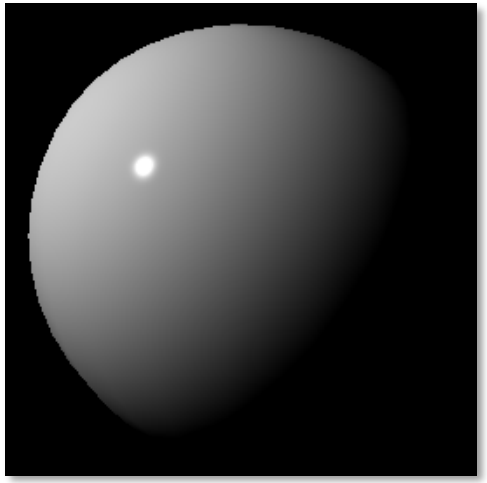
Gamma-Correct Lighting

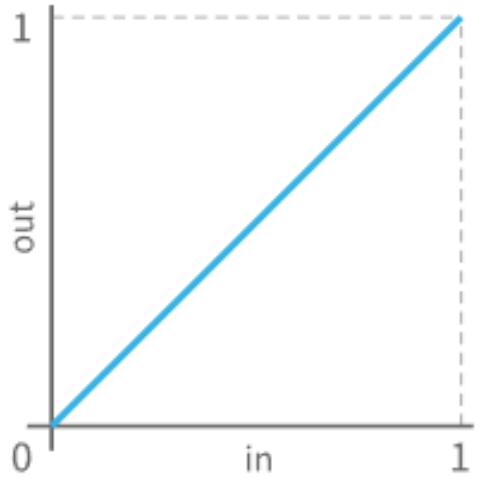
Or: The importance of being linear

Where can digital images go wrong?

- Do digital images captured by a camera accurately replicate light levels?
  - Does 2x light = 2x bright pixel?
- Does our renderer produce results that match real world lighting results?
- Do monitors accurately turn pixels into light?
  - Does 2x bright pixel = 2x light?



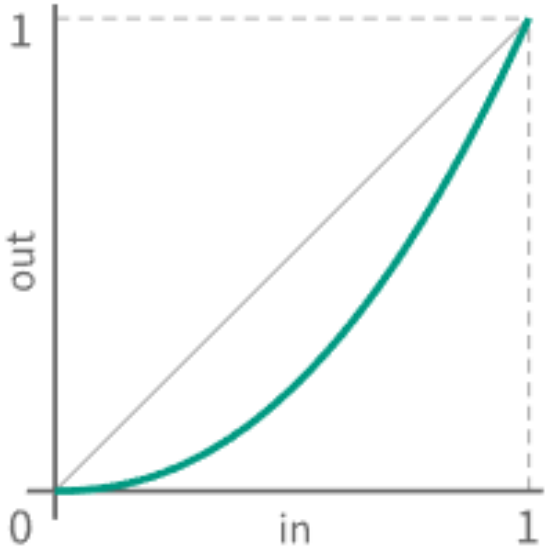
Let's start with monitors



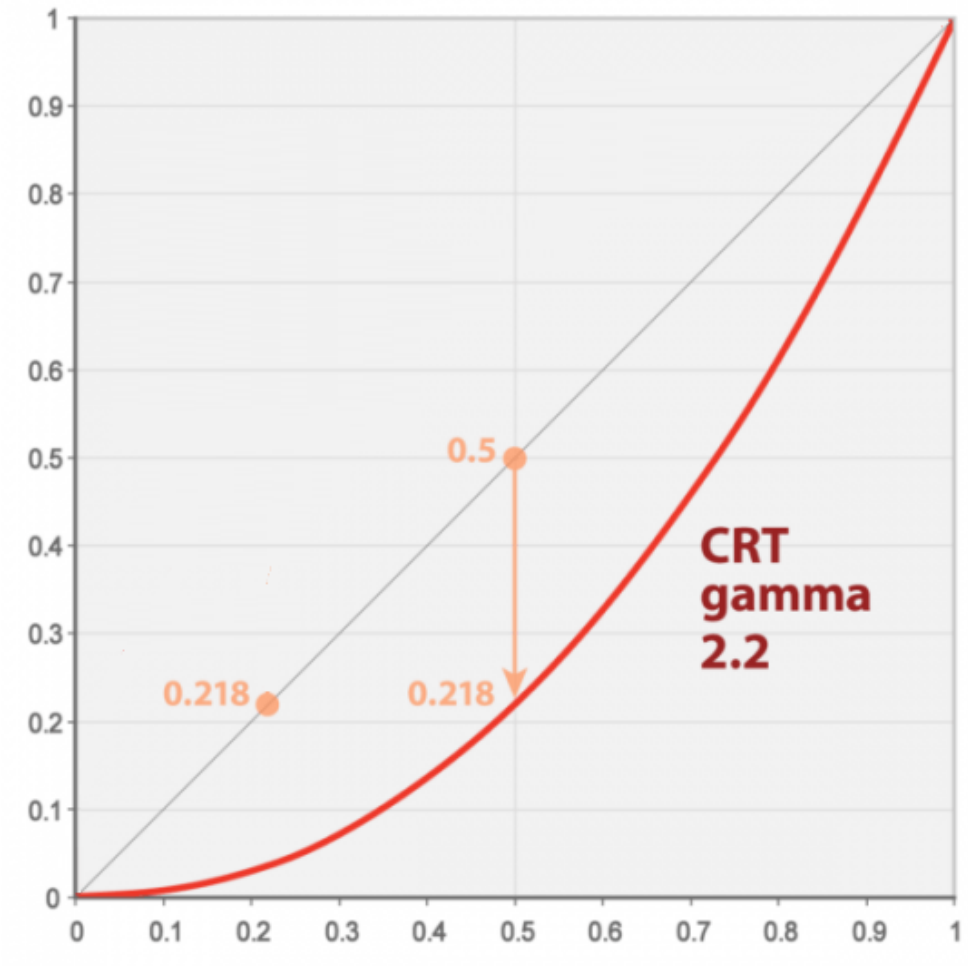
- Monitors (CRT, LCD, etc.) are not *linear*
- Amount of photons emitted does *not* increase linearly w/ the value you send to the monitor
- In other words:
  - A pixel value of .25 emits *n* photons
  - Does a value of .50 emit *2n* photons? Nope!

Monitor response: Gamma

- In general, displays are darker than expected
  - Referred to as gamma characteristics (or gamma)
- Given a set of linear *input values* (colors)
  - *Output* follows a power curve
  - Midtones appear darker
- Every device is different
  - On average, the curve is:
  - $out = in^{2.2}$



Gamma example



Monitor gamma example

- Given the image on the left
  - And no extra processing
- It looks darker on a monitor (right image)
  - Especially the mid-tones

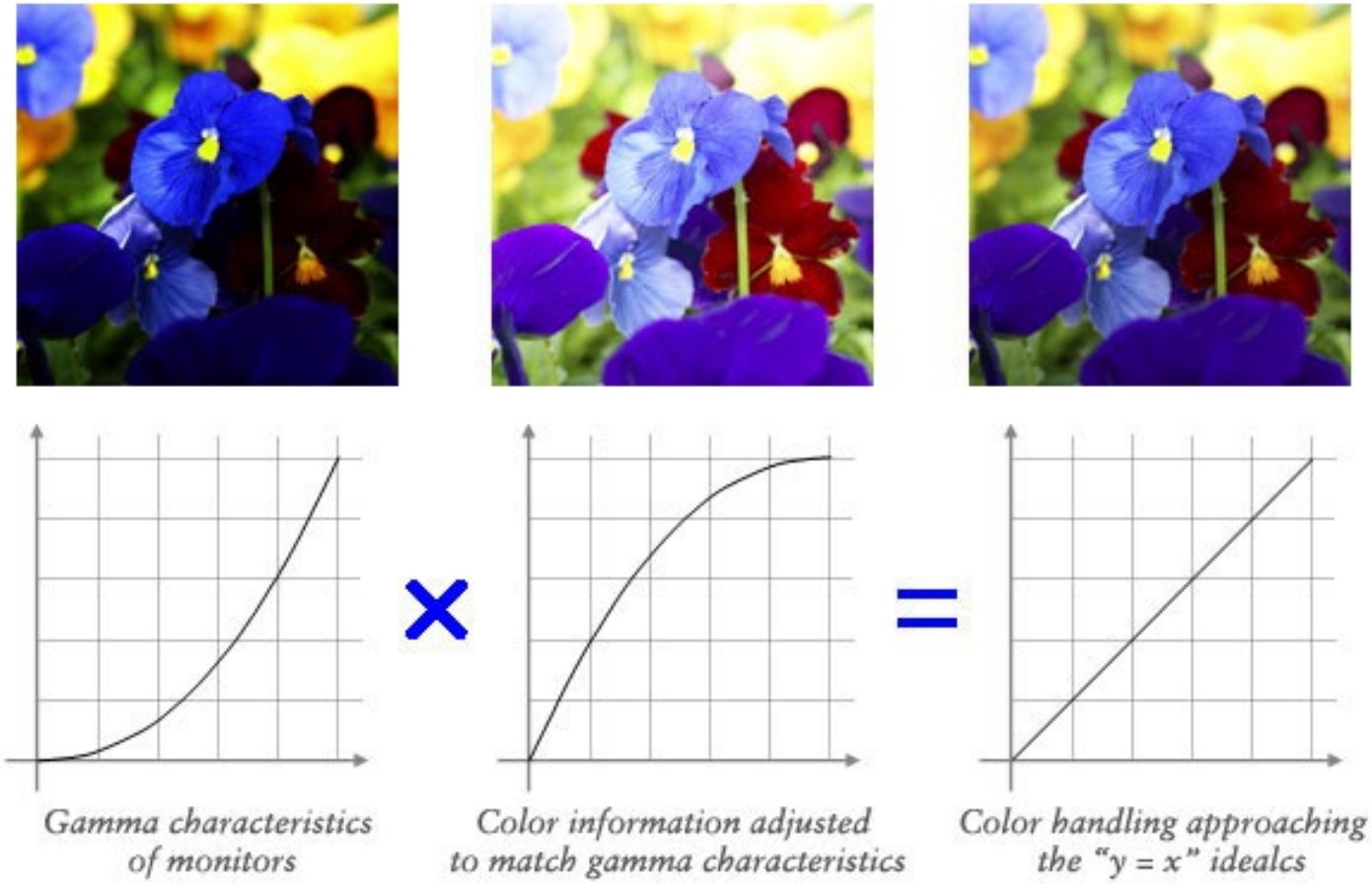
But…the image on the left was fine?

- Yes – because it's been *gamma corrected*
  - The image is actually stored brighter than normal
  - So it looks correct when displayed

Gamma corrected
(extra bright)

Correctly displayed

Gamma correction example

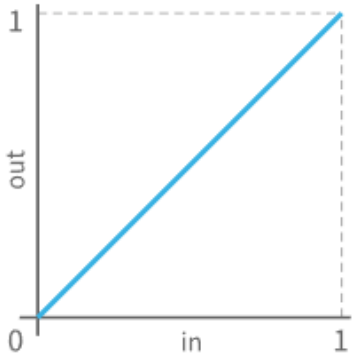


Should we *gamma correct* images?

- It turns out they already are!
  - Digital cameras do this for us already
  - Unless you're shooting in *raw* mode
- Pretty much every image file is already gamma encoded for us
  - Since it'll ultimately be displayed on a screen
  - And that screen will have a gamma curve
- Great…so what's the catch?

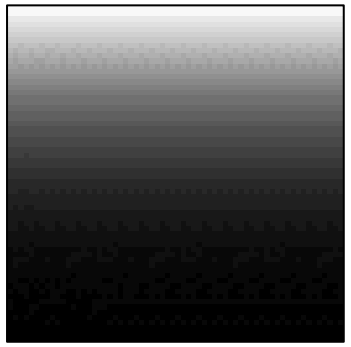
What does this have to do with us?

- Our lighting calculations are linear
  - We assume 2x bright = 2x color
  - [0.6, 0.4, 0.2] is twice as bright as [0.3, 0.2, 0.1]

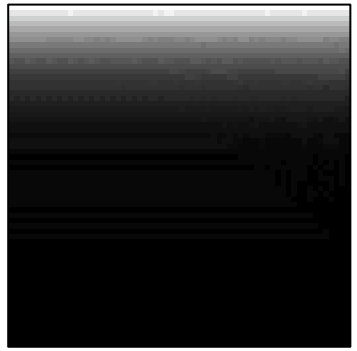


- But they won't be displayed that way!
  - The monitor has that pesky gamma curve

Shader returns

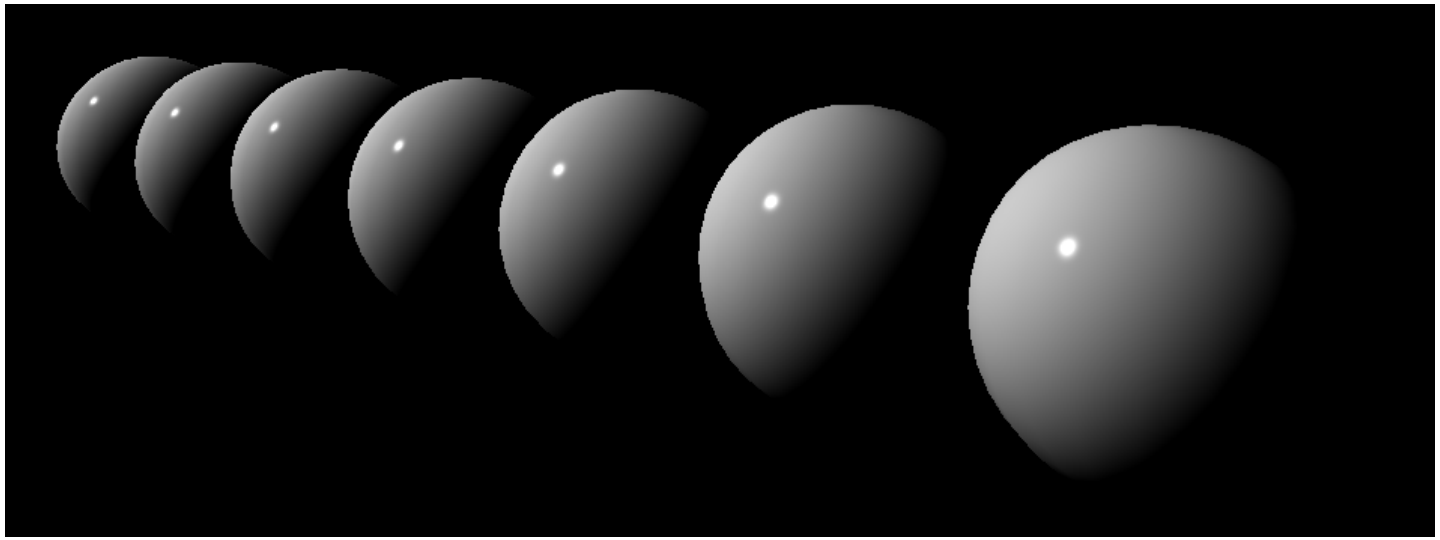


Monitor displays



Our current diffuse lighting

- Here's the standard Lambert (N dot L)

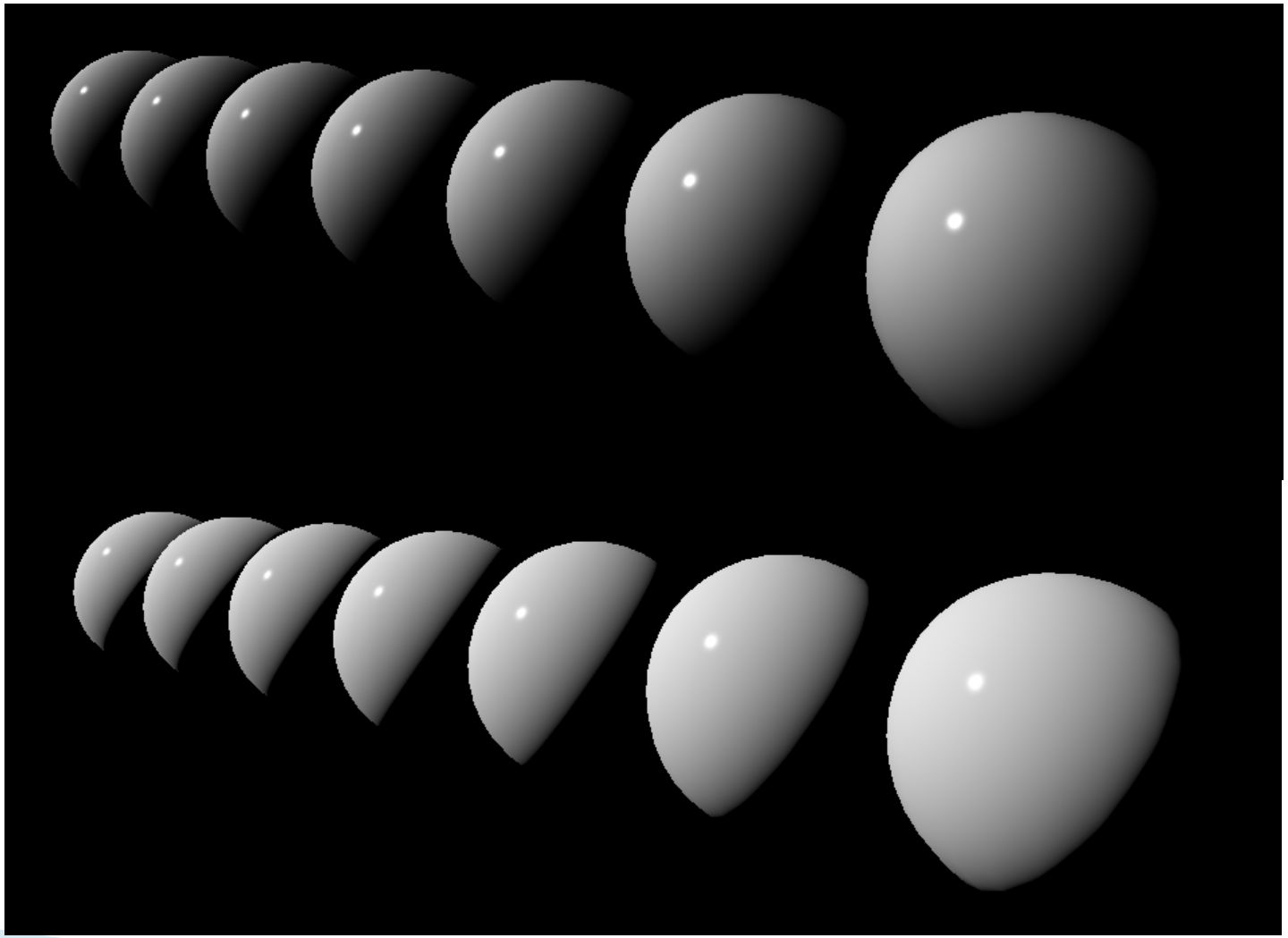


- Looks pretty good! It's also incorrect

Lambert with gamma correction

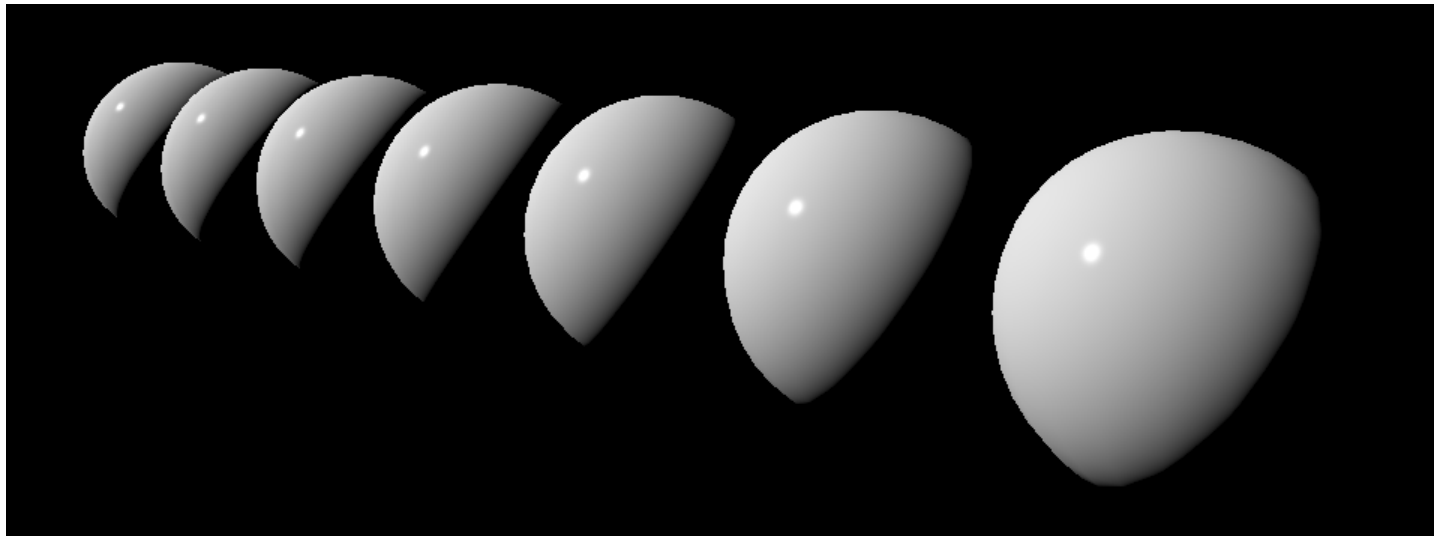
No correction

Corrected



Is this actually correct?

- How would we know?

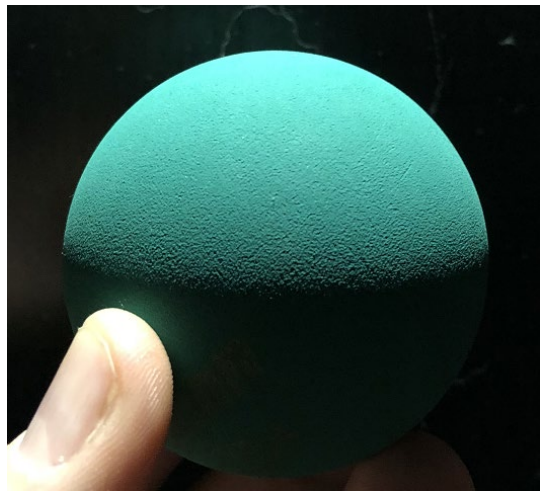


- Let's look at the real world to find out!

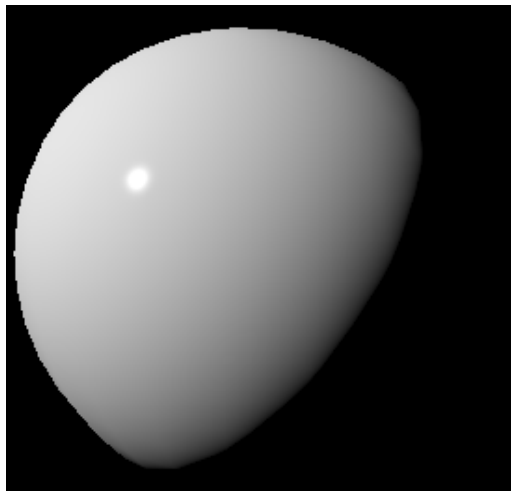
Real world references

- If you want photo-realism, take some photos

Egg



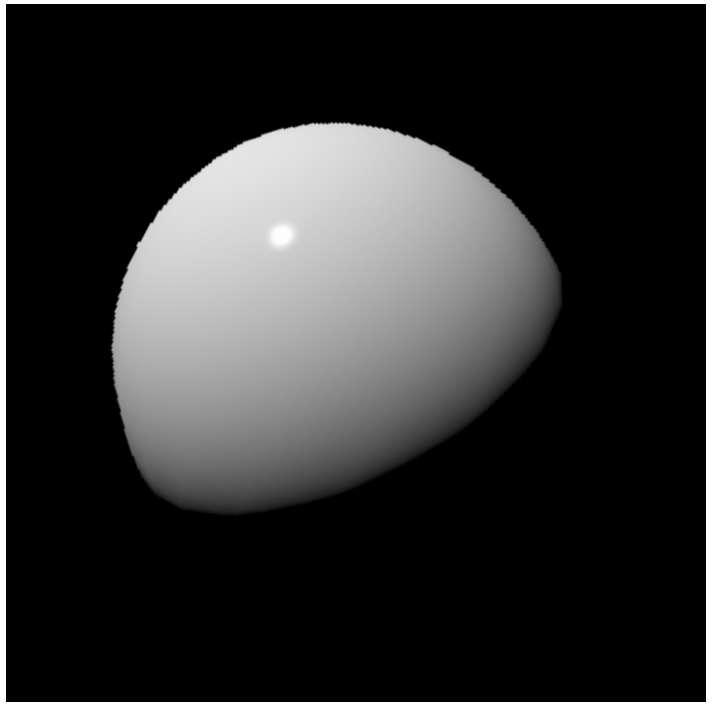
Racquetball



Gamma corrected
Lambert sphere

Another real world reference

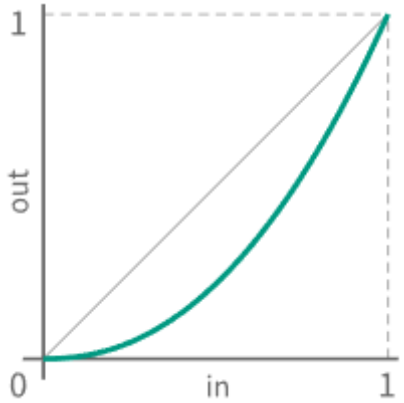
Actual photo of
Earth from space



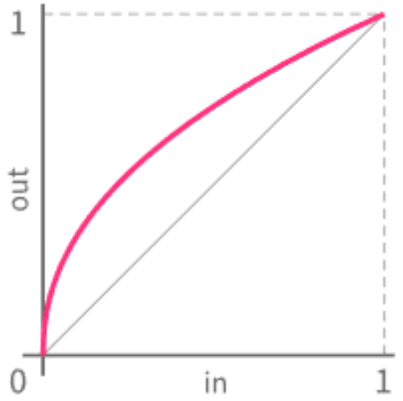
Gamma corrected
Lambert sphere
(rotated to match)

Gamma correction

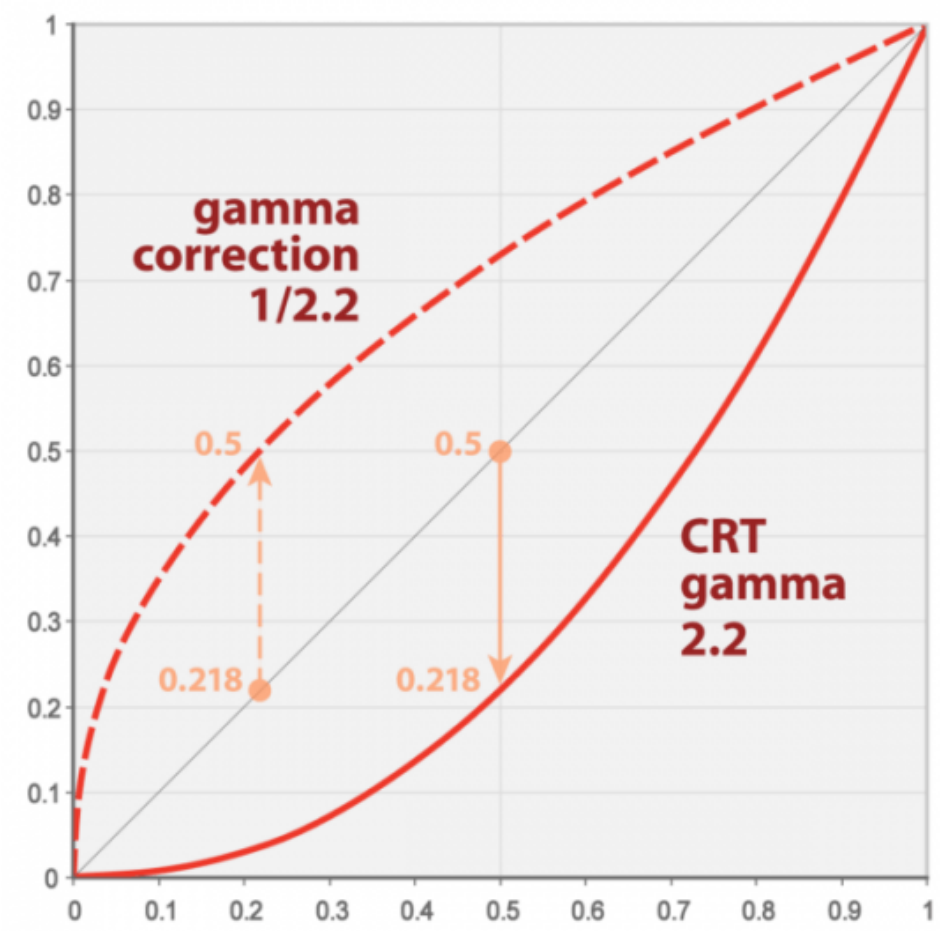
- Monitors essentially apply a gamma of 2.2
  - $color^{2.2}$
  - (on average)



- We need to account for that by applying an opposite curve ourselves
  - $color^{(1/2.2)}$
  - In our shaders



Gamma correction example



Gamma correction shader

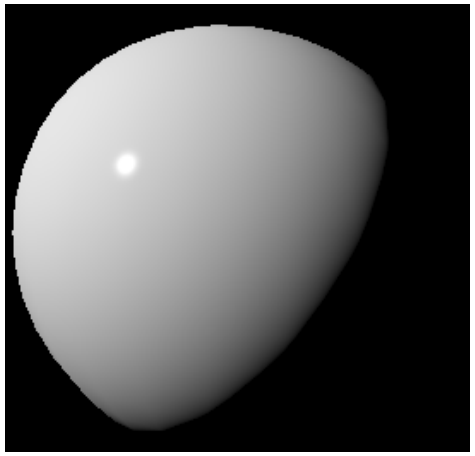
- Raise the final color to a power of 1/2.2
  - Do this at the very end of the shader
  - After all lighting is calculated

```
// ‘color’ is a float3 in this example
// (don’t adjust the alpha value)
float3 gammaAdjustedColor = pow(totalColor, 1.0f / 2.2f);
```

- Fairly easy!
- I wonder if we forgot something 🤔

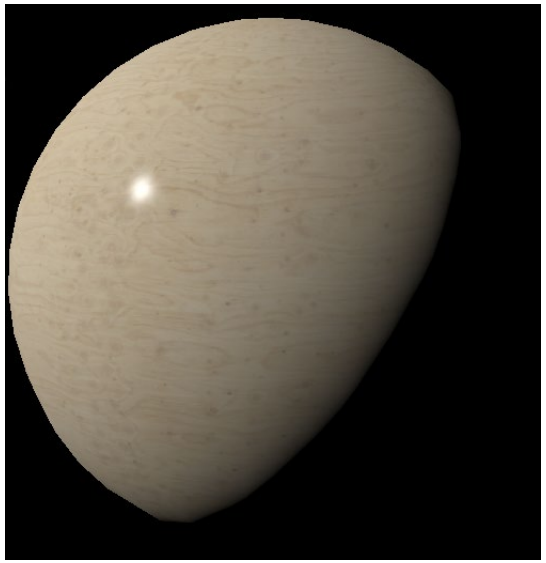
Let's add a texture

- We have our gamma corrected shader output



- We have our fancy texture

- We put them together…
  - Aaaand it's too bright!
  - What went wrong?



Textures & gamma

- Remember this?

Gamma corrected image

Correctly displayed

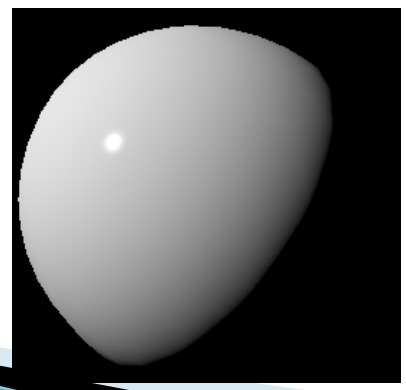
- Images already have gamma correction applied!
- But our math assumes linear colors
- We need to un-correct the colors we get from textures before we use them for shading

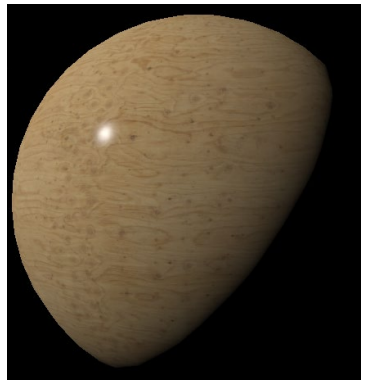
Un-correcting texture colors

- Apply the same curve your monitor would

```
// Just do this to the RGB channels
float3 textureColorLinear = pow(textureColor, 2.2);
```

- Now it's ready to be used for shading





Which textures to correct?

- Only perform on textures that represent ***surface colors!***

- In general, don't apply the gamma correction to:
  - Normal maps
  - Specular maps
  - Metalness/roughness maps (for PBR)

All together now

- Overall steps for a gamma correct shader:

1. Sample textures
   Apply 2.2 power curve to surface colors:
   $texColor = texColor^{2.2}$

2. Do all lighting, texturing, etc.

3. Apply a 1/2.2 power curve to final color:
   return $finalColor^{(1/2.2)}$

This seems like a common issue…

- …why doesn't our GPU handle it for us?
- As it turns out, it CAN!
- We need to use a special texture format: ***sRGB***
  - "Standard Red Green Blue"
  - Created by HP & Microsoft in the 90's
  - Specifically for this kind of issue

Using sRGB textures

- If we create a texture as an sRGB texture:
  - Sampling from it applies the gamma correction
  - We'll get the colors in linear space
  - Let the hardware do it – it'll be faster

- Fancy version of our texture loading function:
  - `CreateWICTextureFromFileEx`
  - Has a *loadFlags* param that can force sRGB
  - Can return an sRGB version of the texture
  - `DXGI_FORMAT_R8G8B8A8_UNORM_SRGB`

Using sRGB render targets

- If we write to an sRGB render target
  - The pipeline will apply the gamma correction
  - It'll store the colors in gamma space

- We could set our back buffer render target up as an sRGB texture
  - Requires editing DXCore
  - Swap chain description takes a format
  - Use `DXGI_FORMAT_R8G8B8A8_UNORM_SRGB`

References

- The Importance of Being Linear [Link]
- Understanding Gamma Correction [Link]
- Linear Space Lighting [Link]