Computer Graphics II
– Advanced Lighting

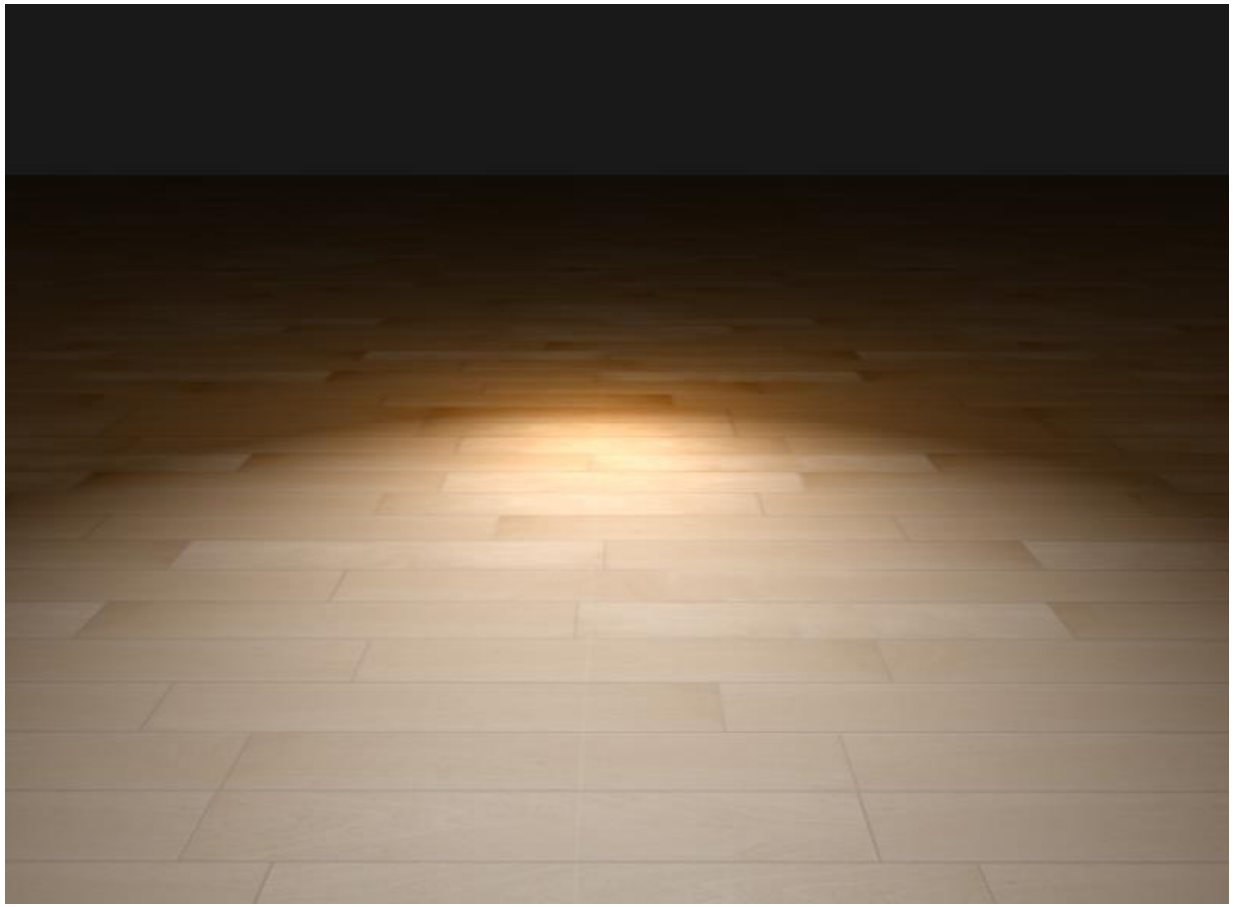
Kai Lawonn

Introduction

- In the lighting lecture, introduced the Phong lighting model to bring a basic amount of realism into scenes
- The Phong model looks quite nice, but has a few nuances we will focus on now

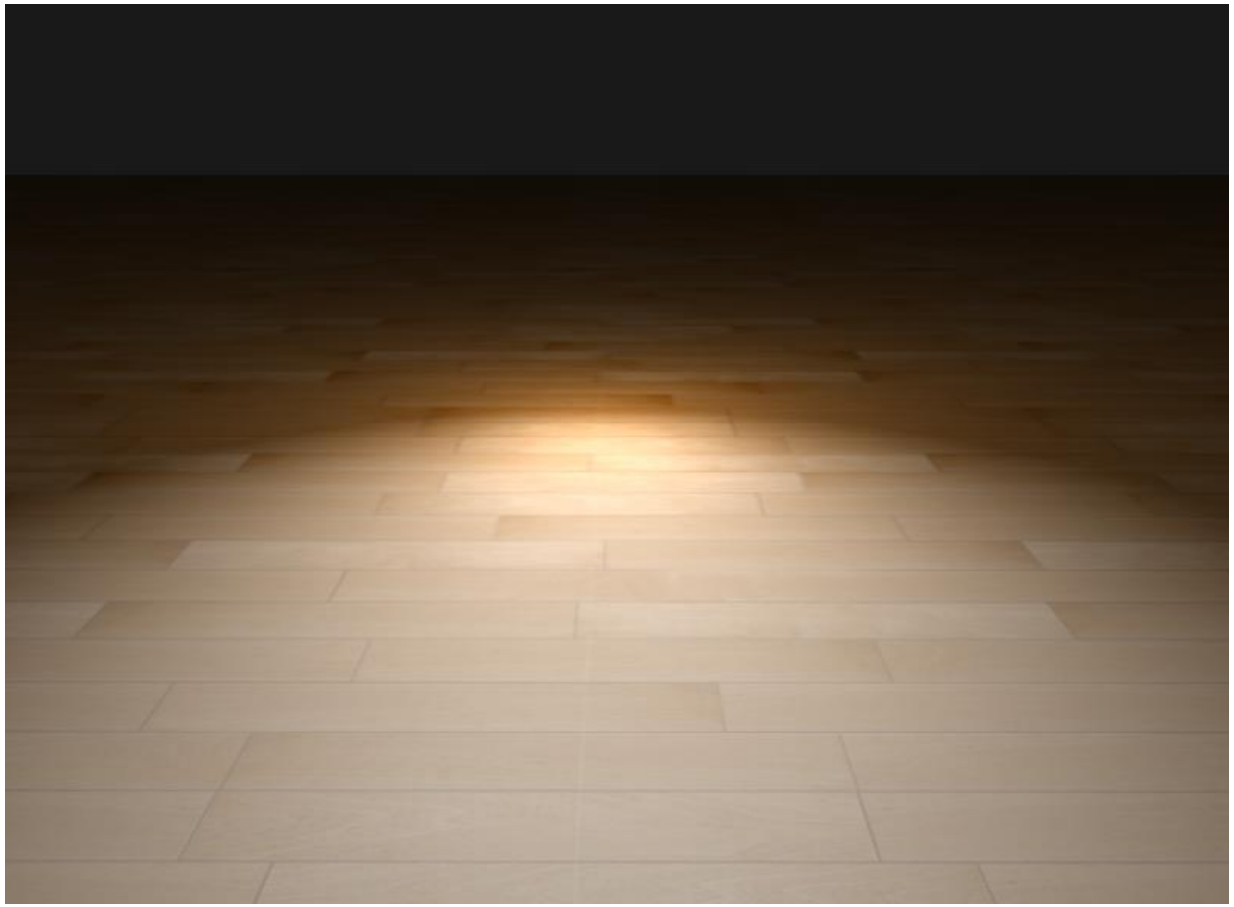
Blinn–Phong

- Phong lighting efficient approximation of lighting
- Specular reflections break down in certain conditions, when the shininess property is low resulting in a large (rough) specular area
- When we use a specular shininess exponent of 1.0 on a flat textured plane:



Blinn–Phong

- At the edges that the specular area is immediately cut off
- The reason is that the angle between the view vector and the reflection vector is not allowed to go higher than 90° degrees (if angle larger, dot product becomes negative resulting in specular exponents of 0.0)
- No light with angles higher than 90°?

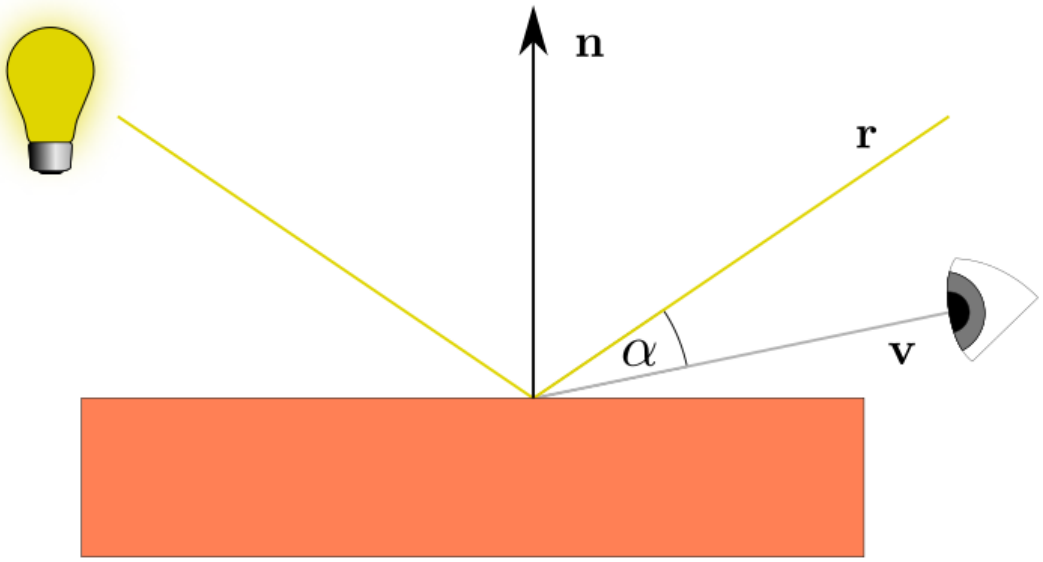


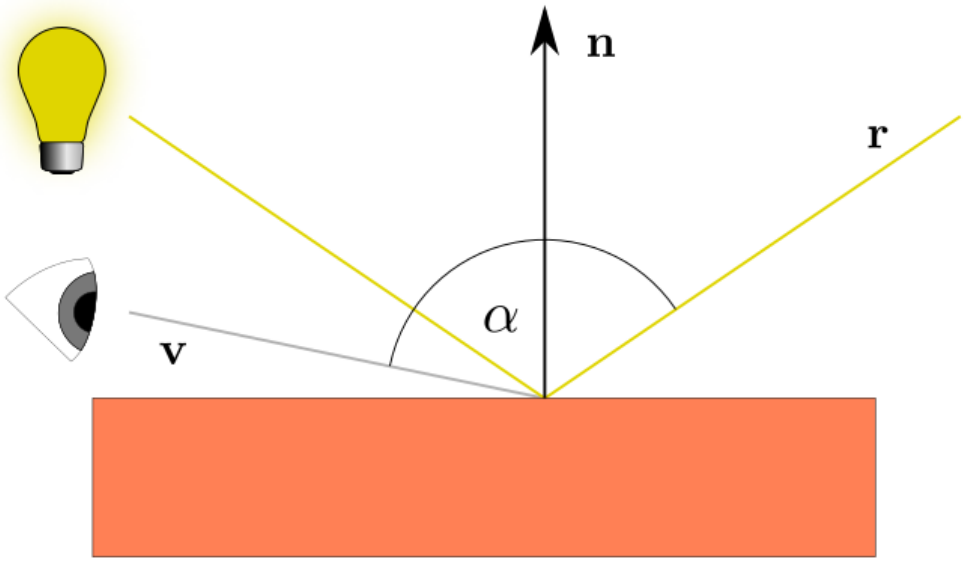
Blinn–Phong

- Only true with the diffuse component (angle higher than 90° between the normal and light source means light source is below the lighted surface) → light's diffuse contribution should equal 0.0

Blinn–Phong

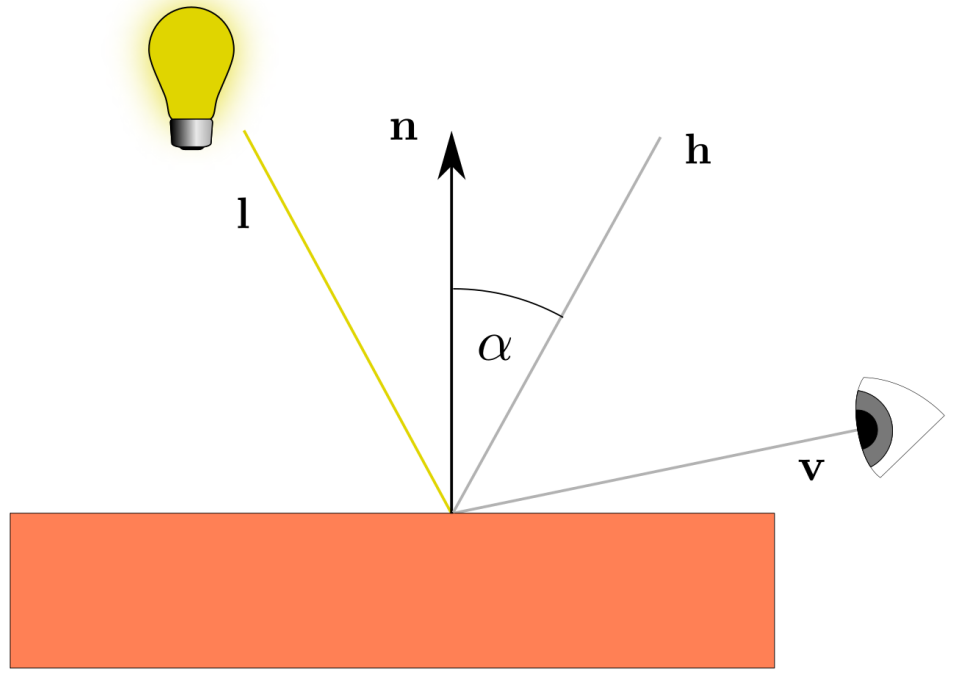
- Specular lighting measures between view and reflection direction vector





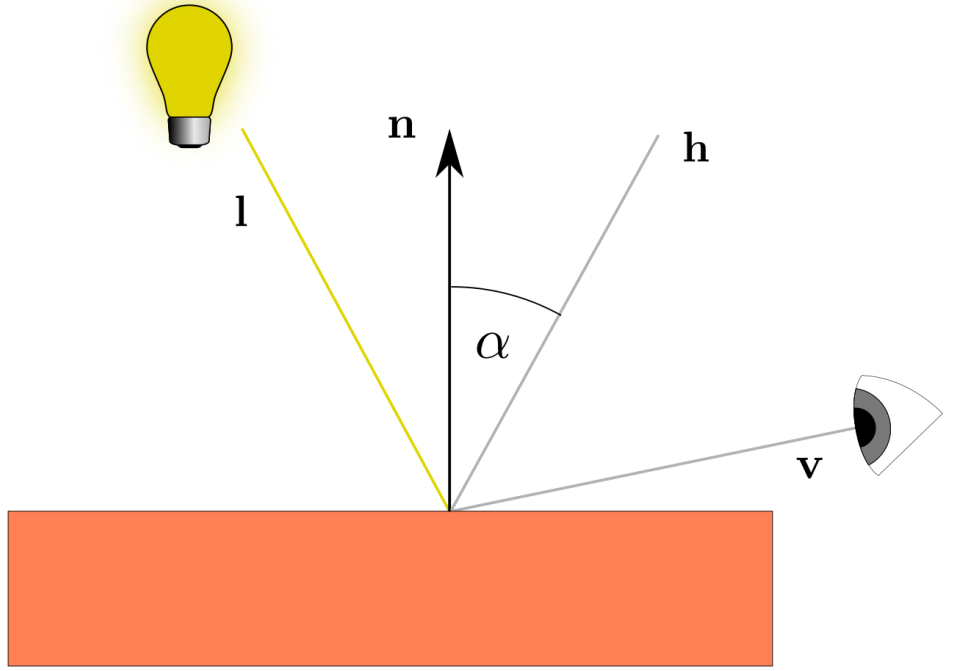
Blinn–Phong

- 1977 Blinn-Phong shading model was introduced by James F. Blinn as an extension to the Phong shading, which overcomes our problem
- Instead of using a reflection vector, it uses a halfway vector (unit vector halfway between the view direction and the light direction)

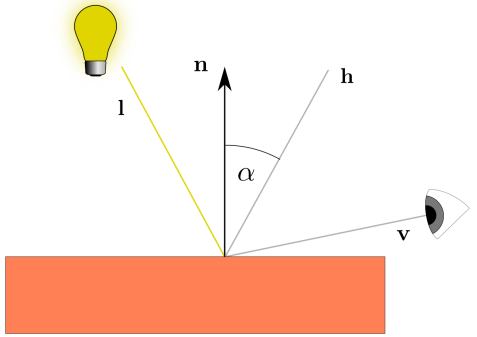


Blinn–Phong

- When view direction is perfectly aligned with the (now imaginary) reflection vector → halfway vector = normal vector
- The closer the viewer looks in the original reflection direction, the stronger the specular highlight
- Angle between the halfway vector and normal never exceeds 90° (unless the light is far below the surface)
- Produces slightly different results, but mostly looks slightly more visually plausible, especially with low specular exponents



Blinn–Phong



- Getting the halfway vector is easy, we add the light's direction vector and view vector together and normalize the result:

$$h = \frac{l + v}{\|l + v\|}$$

Blinn–Phong

$$h = \frac{l + v}{\|l + v\|}$$

- Getting the halfway vector is easy, we add the light's direction vector and view vector together and normalize the result:

```
vec3 lightDir = normalize(lightPos - FragPos);
vec3 viewDir = normalize(viewPos - FragPos);
vec3 halfwayDir = normalize(lightDir + viewDir);
```

Blinn–Phong

$$h = \frac{l + v}{\|l + v\|}$$

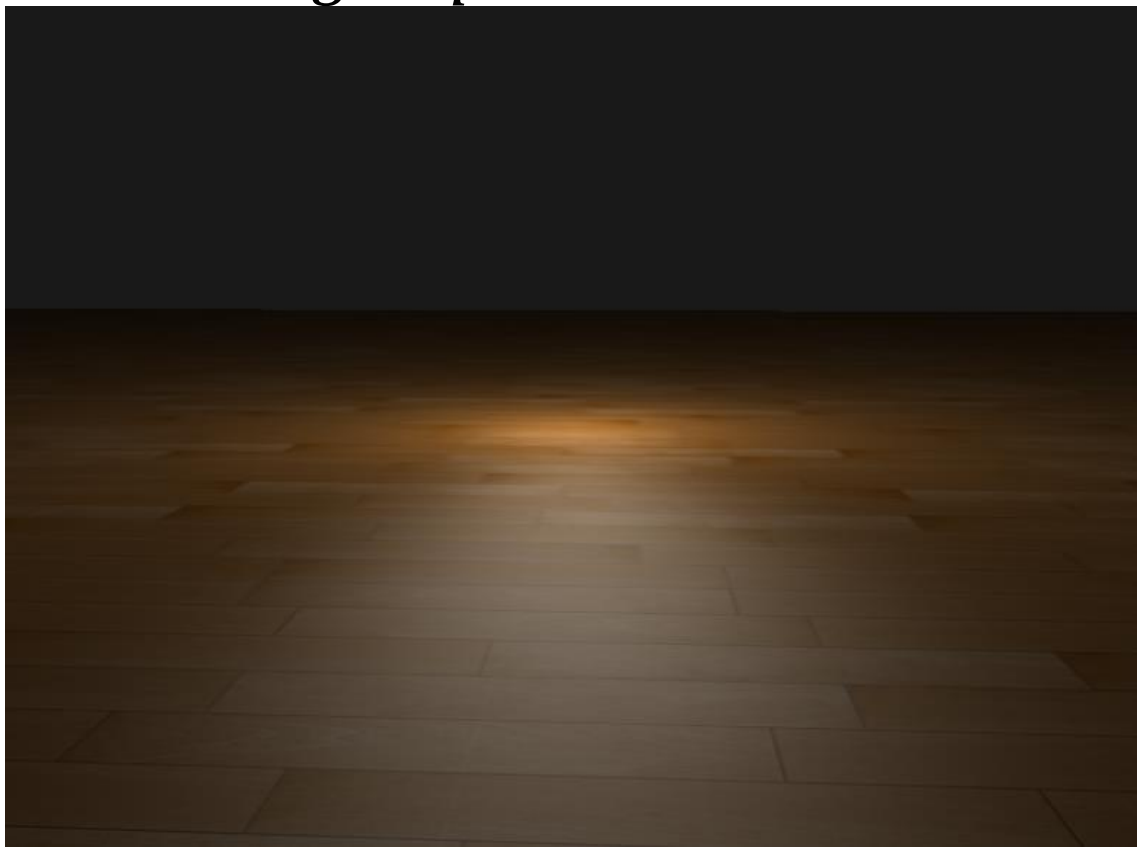
- Actual calculation of the specular term becomes a clamped dot product between the surface normal and the halfway vector to get the cosine angle between them that we again raise to a specular shininess exponent:

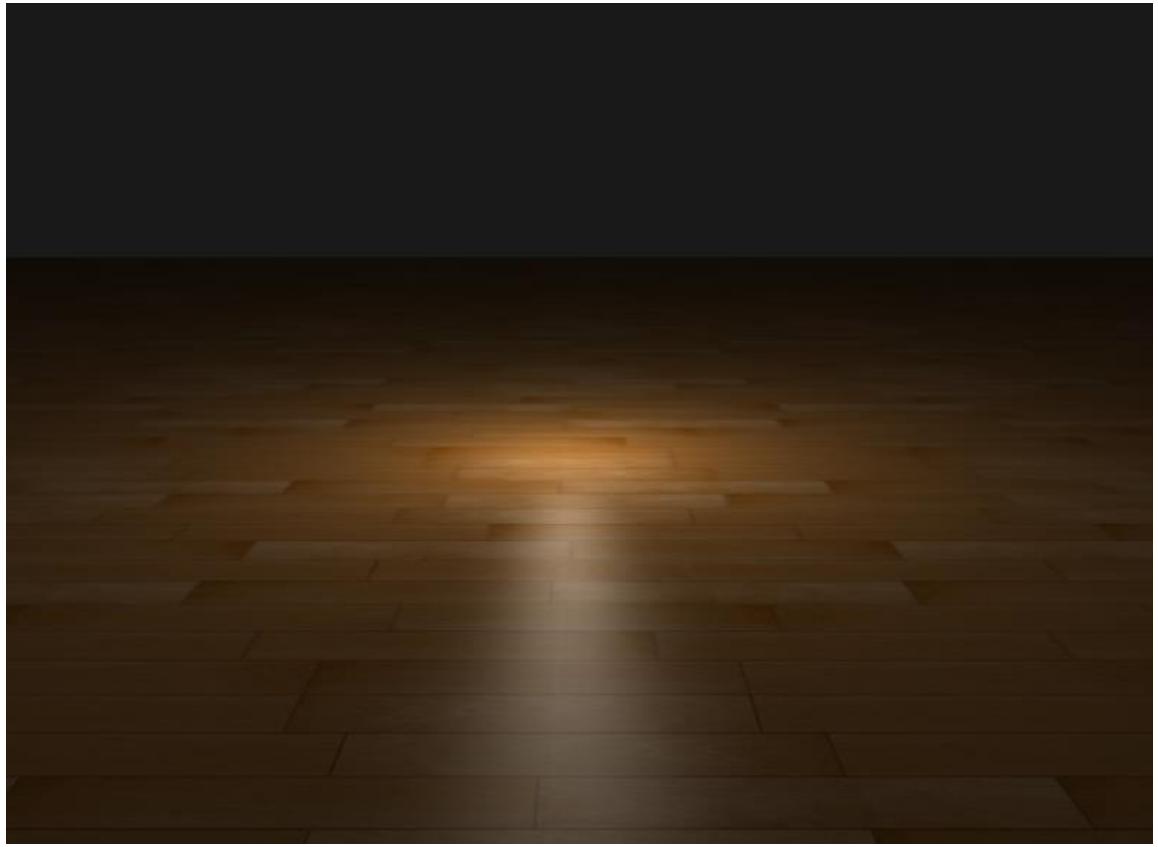
```
float spec = pow(max(dot(normal, halfwayDir), 0.0), shininess);
vec3 specular = lightColor * spec;
```

- And that’s it

F5…

- … left Phong - $Phong\ exponent\ =\ 8$, right Blinn-Phong - $\mathrm{Blinn} - Phong\ exponent\ = 32$





Blinn–Phong

- Simple fragment shader that switches between regular Phong reflections and Blinn-Phong reflections:

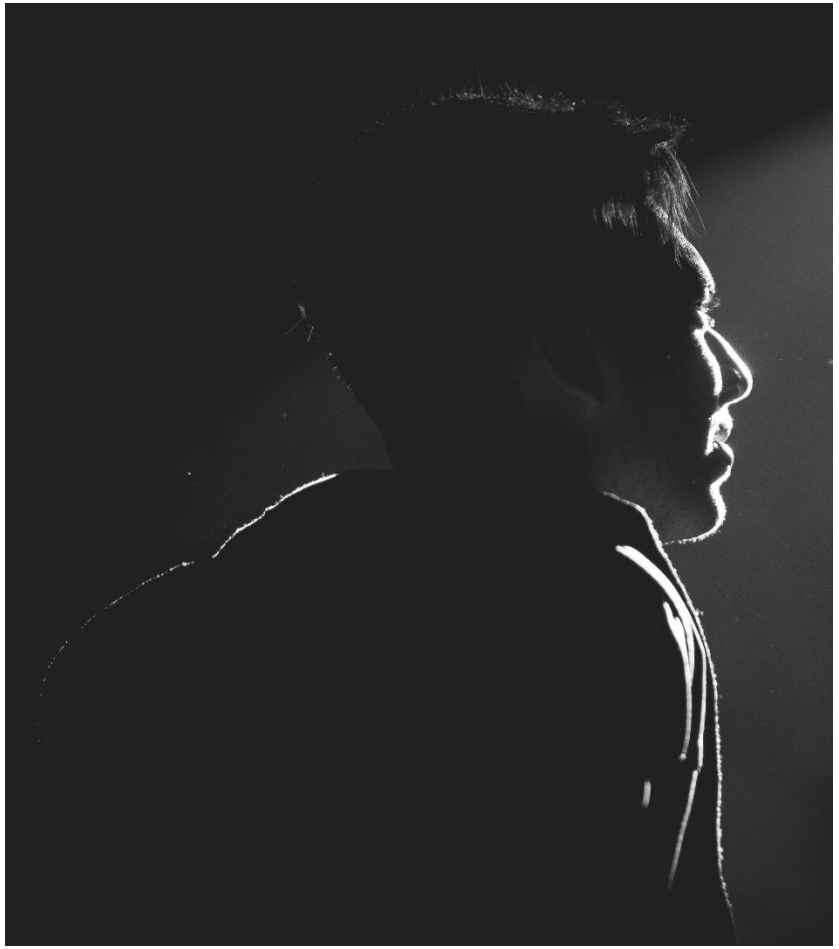
```
float spec = 0.0;
if(blinn)
    {
        vec3 halfwayDir = normalize(lightDir + viewDir);
        spec = pow(max(dot(normal, halfwayDir), 0.0), 32.0);
    }
else
    {
        vec3 reflectDir = reflect(-lightDir, normal);
        spec = pow(max(dot(viewDir, reflectDir), 0.0), 8.0);

    }
```

Rim Lighting*

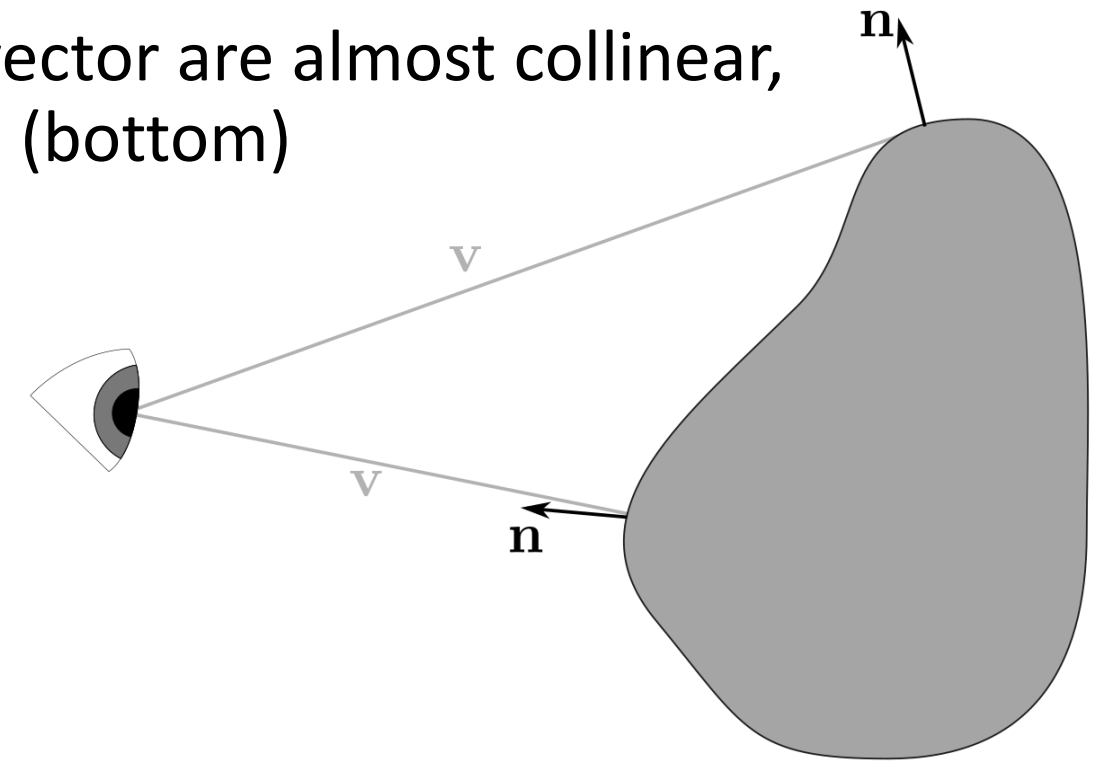
Introduction

- Rim lighting also known as back-lighting
- Effect that simulates light around an object, light source placed behind the object
- Produces a bright rim of light around the contours of the object
- Can simulate effect by determining how close view direction is at the contour

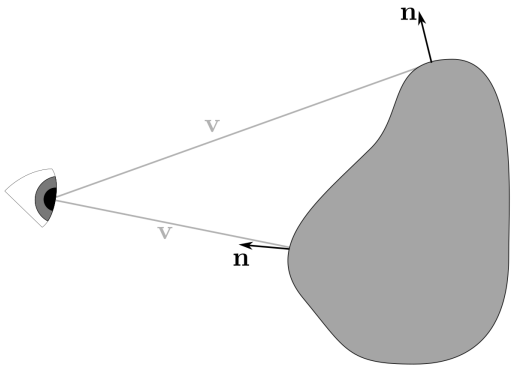


Introduction

- All we need is the view direction and the normal vector of the surface
- If the view direction is perpendicular to the normal, we are close at the contour, this has the greatest effect (top)
- If view direction and normal vector are almost collinear, rim lighting is least noticeable (bottom)



Rim Lighting



- Calculate this effect with the dot product
- Perpendicular → highest effect
- Collinear → smallest effect

$$L_{rim} = C_{rim} \cdot (1 - \langle n, v \rangle)^r$$

- C_{rim} color of the rim light, e.g., white $(1,1,1)$
- r power of the rim lighting
- $< n, v >$ dot product of normal and view vector

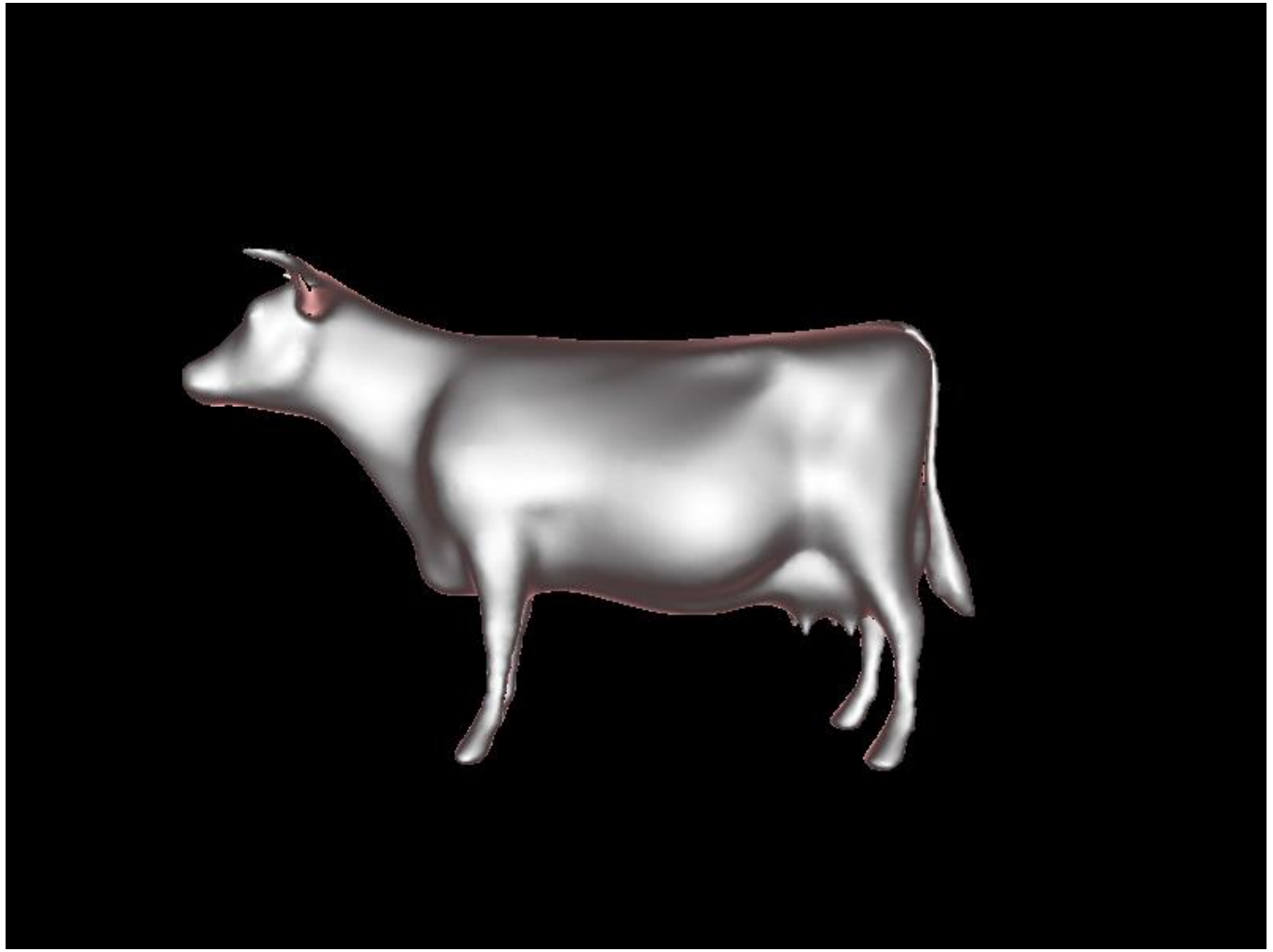
Rim Lighting

- Add function in the fragment shader:

```
vec3 rimLighting(vec3 normal, vec3 view, vec3 rimColor, float rimPower)
{
   float res = 1.0 - dot(normal, view);
   // Clamp it to the range 0 to 1
   res = clamp(res, 0.0, 1.0);
   res = pow(res, rimPower);
   return res * rimColor;
}
…
void main()
{
…
FragColor+=vec4(rimLighting(normal, view, vec3(1,0.7,0.7), 2),0);
}
```

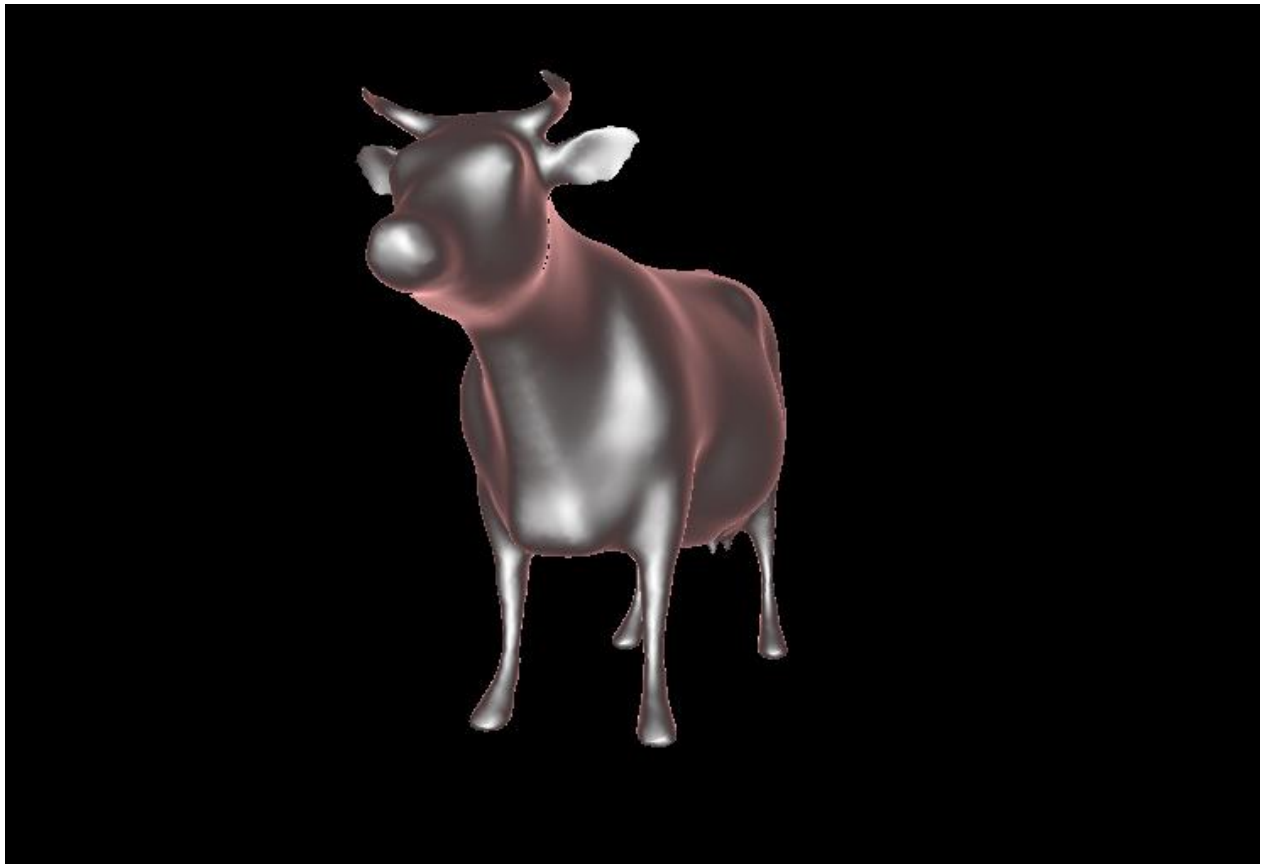
F5…

- … rim lighting!



Notes

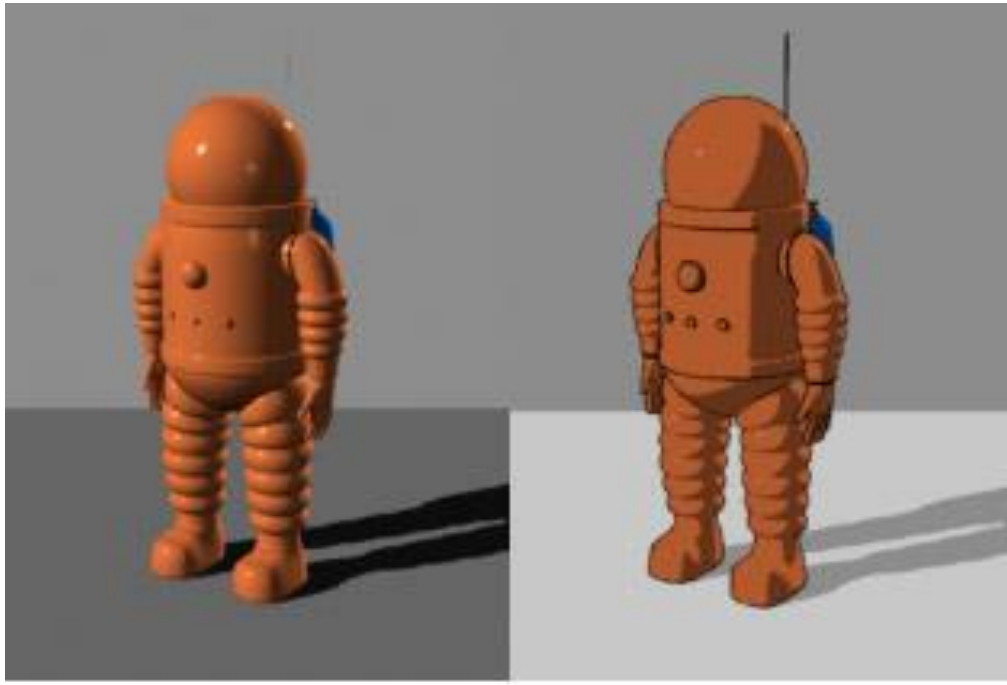
- Lighting at the contours
- Not realistic as it should only be placed at the outline



Cel/Toon Shading*

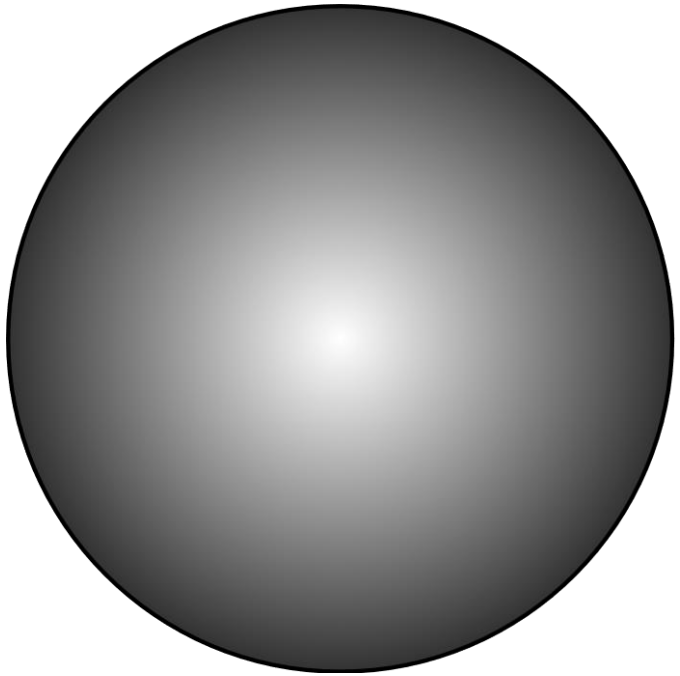
Introduction

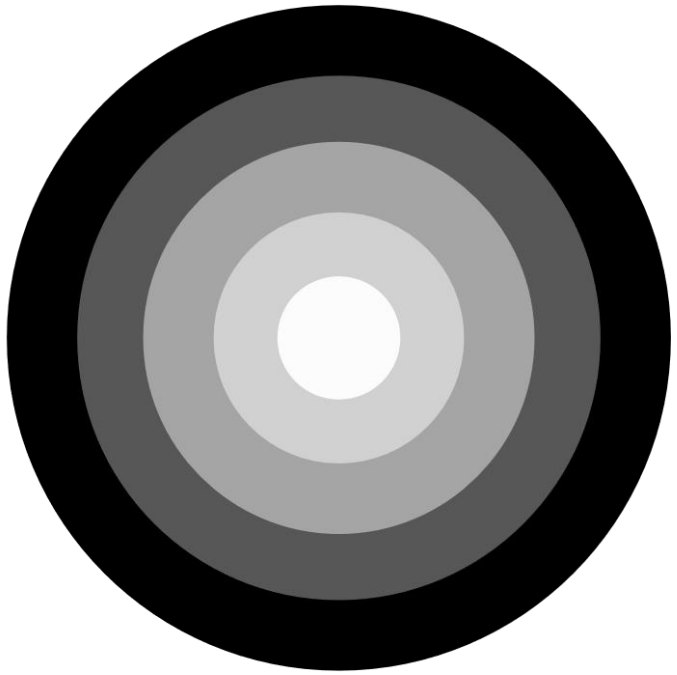
- Cel/toon shading is a non-photorealistic rendering technique
- Try to appear flat by using less shading



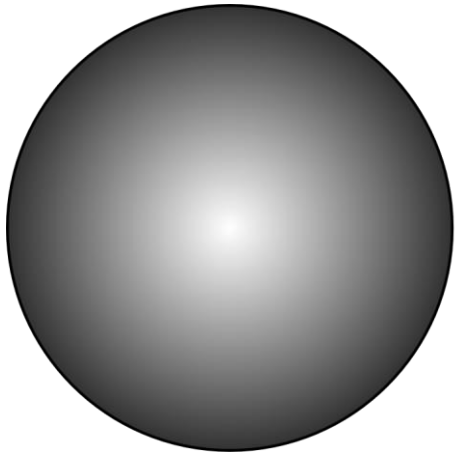
Introduction

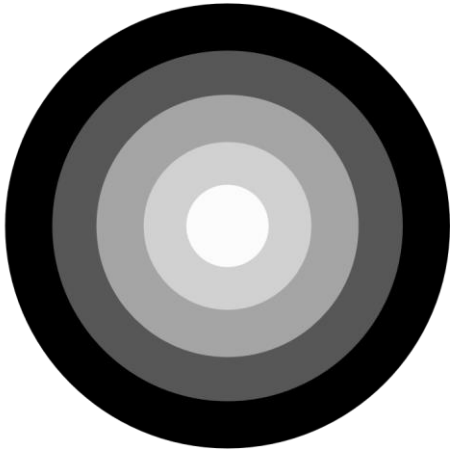
- Normal shading is quantized with discrete colors
- Shading in a range is mapped to a constant color





Cel/Toon Shading





- Calculate this effect with the ceil function
- Shading is in the range $[0,1]$, ceil finds the nearest (greater) integer

$$L_{cell} = C_{cell} \cdot \frac{ceil(num \cdot \langle n, v \rangle)}{num}$$

- C_{cell} color of the cell shading, e.g., white $(1,1,1)$
- num number of colors used
- $< n, v >$ dot product of normal and view vector

Cel/Toon Shading

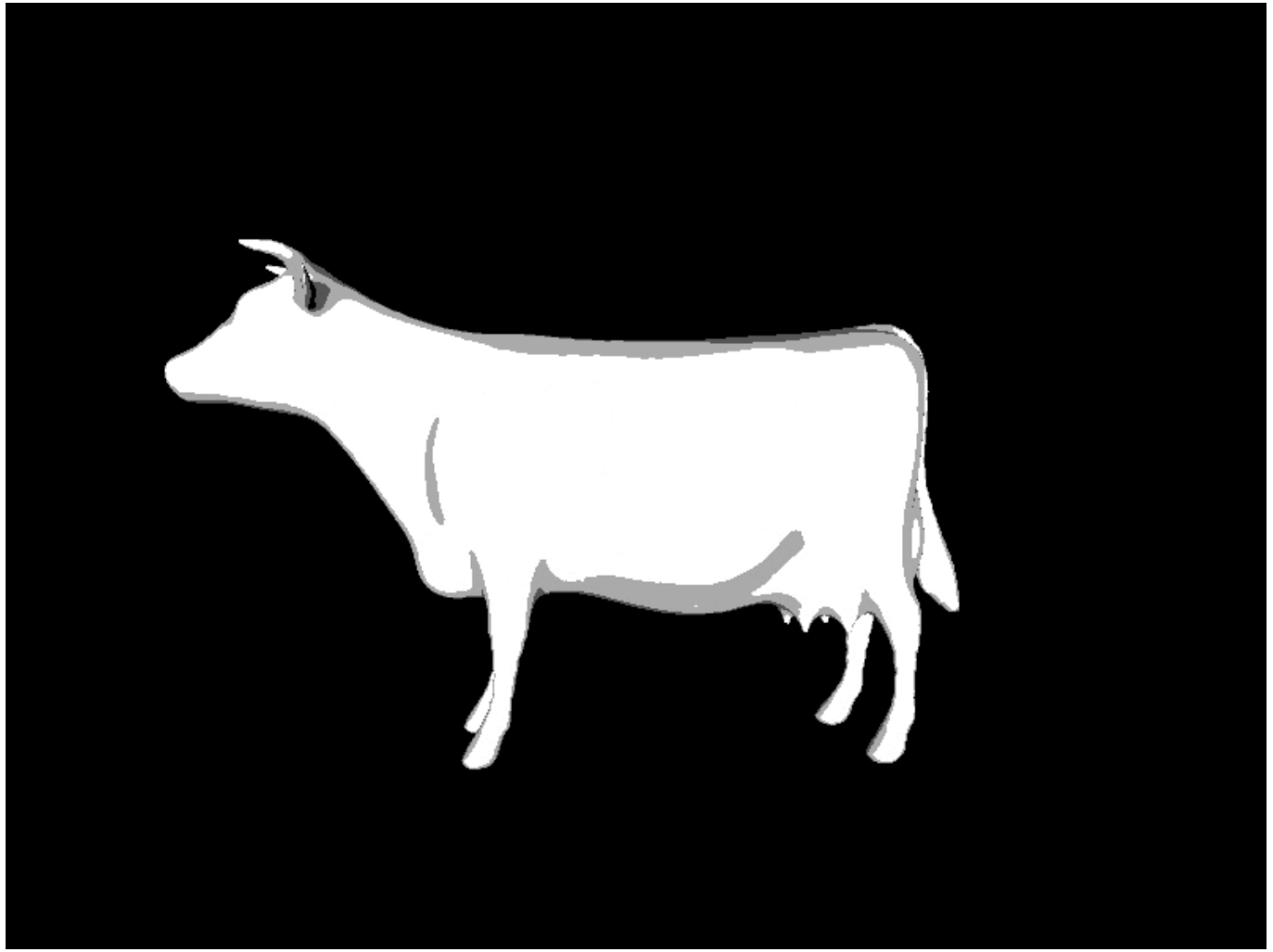
- Add function in the fragment shader:

```
vec3 celShading(vec3 normal, vec3 view, vec3 cellColor, float numberOfColors)
{
   float res = max(dot(normal, view),0.0);
   res = ceil(res * numberOfColors) / numberOfColors;
   return cellColor*vec3(res);
}

void main()
{
   …
   FragColor=vec4(celShading(normal,view,vec3(1),3),1);
}
```

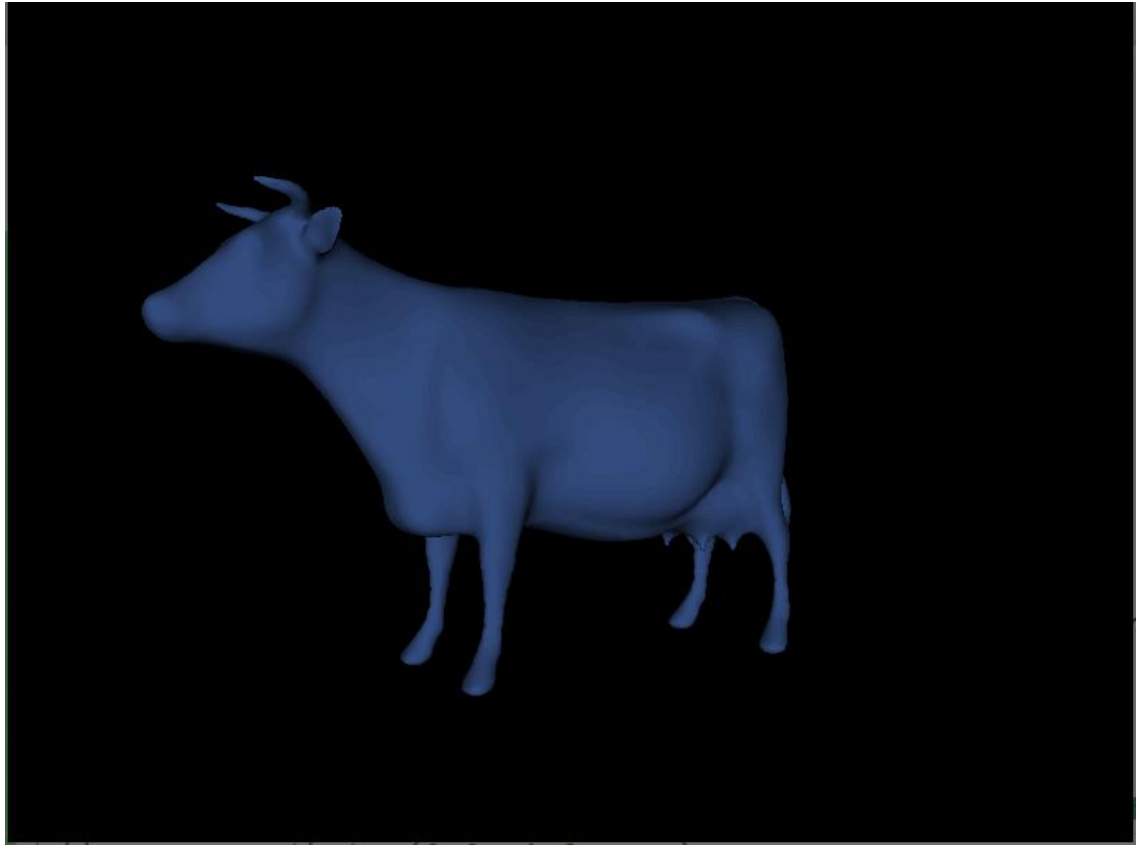
F5…

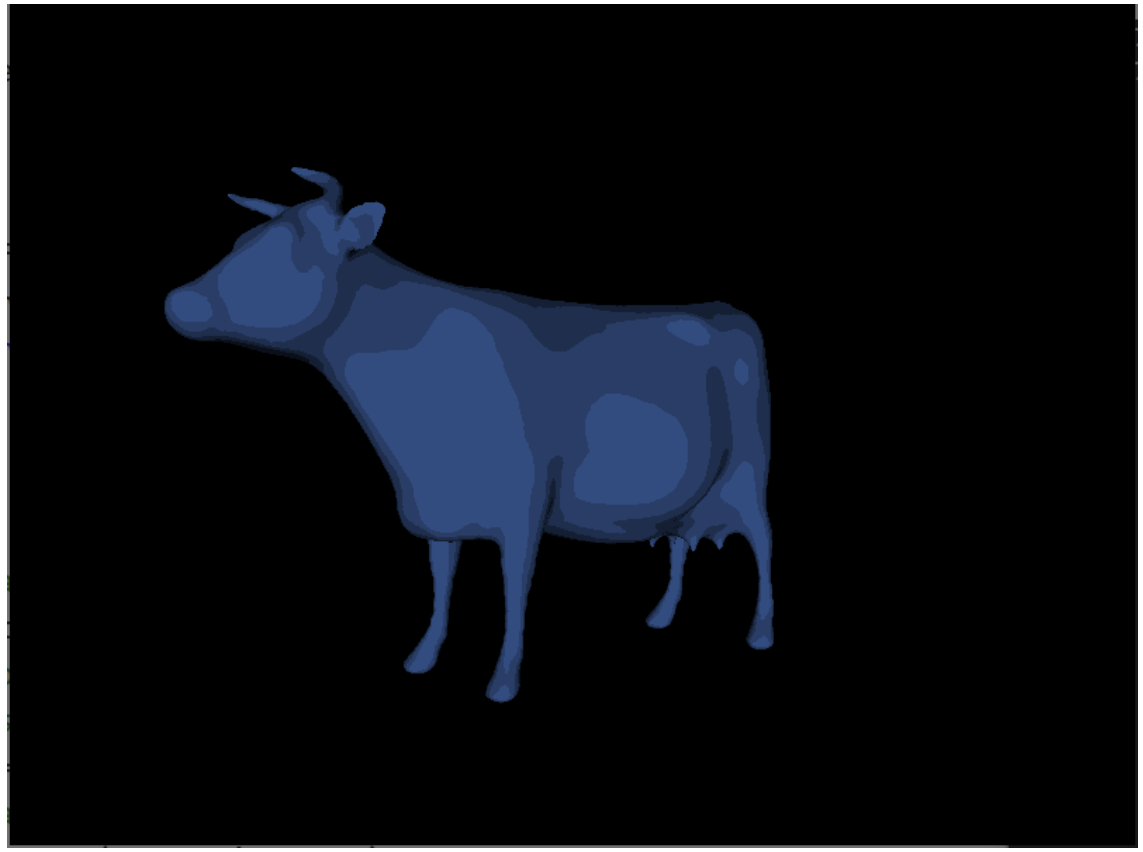
- … cel shading!



Notes

- The higher num the more shading (right 20; bottom 5)





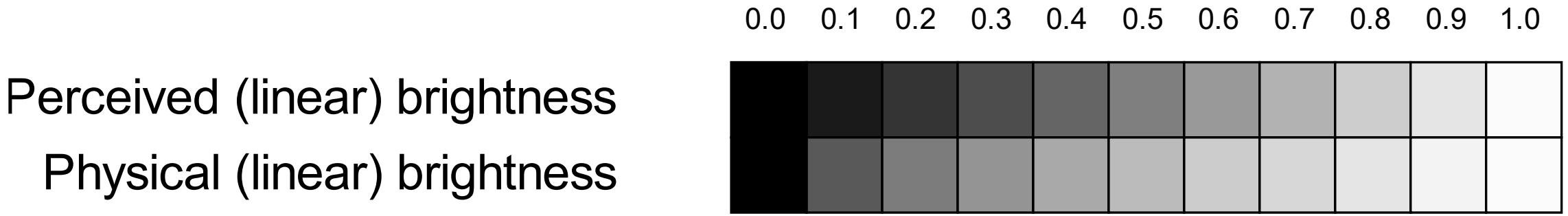
Gamma Correction

Introduction

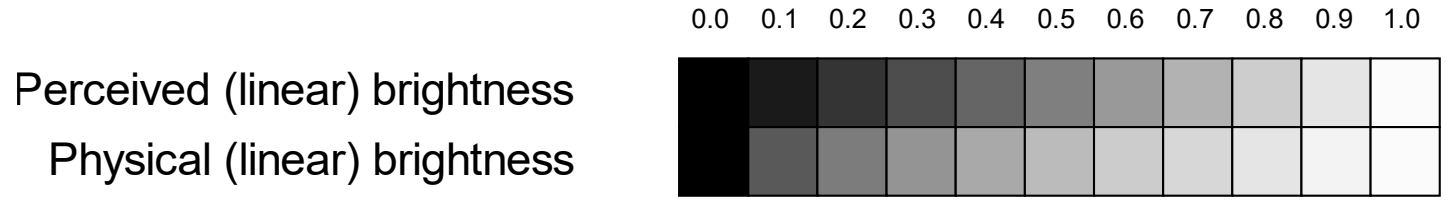
- Computed final pixel colors have to displayed on a monitor
- In the old days, most monitors were cathode-ray tube (CRT) monitors
- These had the physical property that twice the input voltage did not result in twice the amount of brightness
- Doubling the input voltage resulted in a brightness equal to a power relationship of roughly 2.2 also known as the gamma of a monitor

Introduction

- This, closely match how human beings measure brightness (brightness is also displayed with a similar (inverse) power relationship)

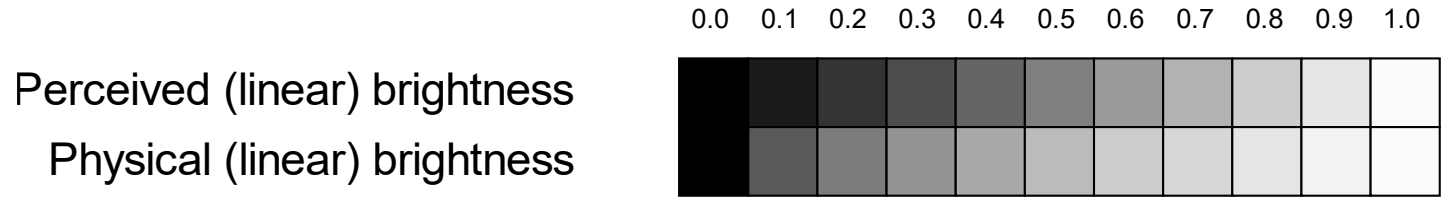


Introduction



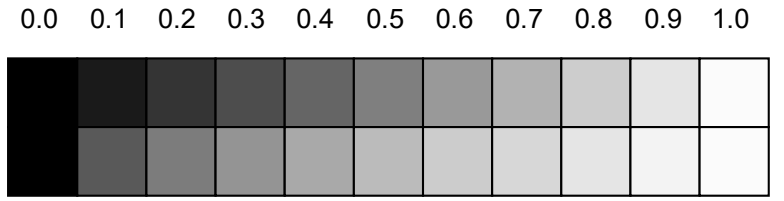
- Human eyes prefer to see brightness colors according to the top scale
- Monitors (still today) use a power relationship for displaying output colors so that the original physical brightness colors are mapped to the non-linear brightness colors in the top scale

Introduction

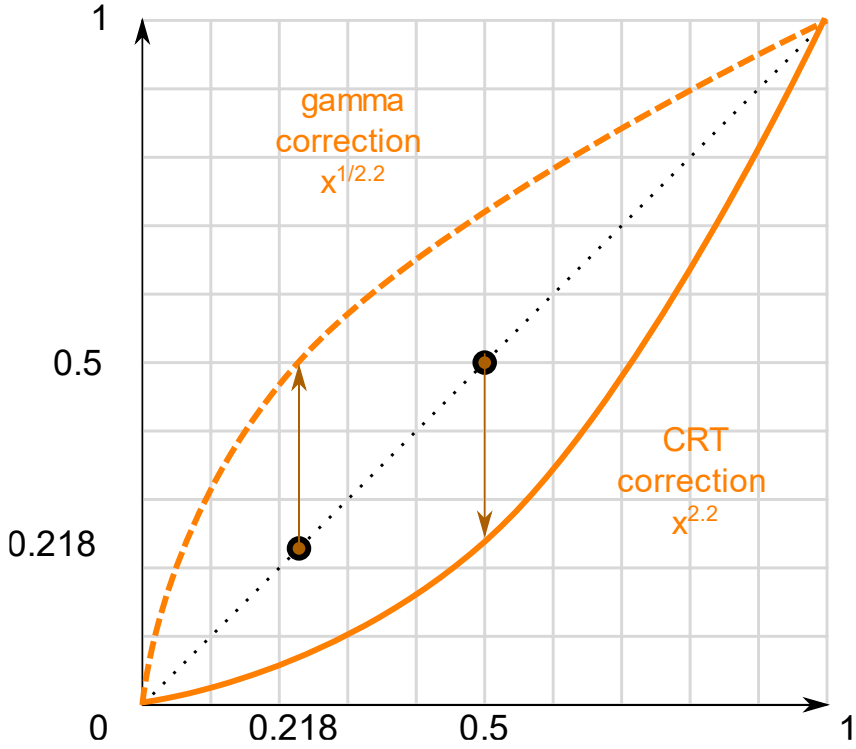


- Non-linear mapping of monitors make the brightness look better, but there is one issue: color and brightness options are based on what we perceive from the monitor and thus all the options are actually non-linear brightness/color options
- Take a look at the graph on the next slide:

Introduction



- Dotted line: color/light values in linear space; solid line: color space that monitors the display
- Double a color in linear space results in a double value
- E.g., double light's color vector $l = (0.5, 0.0, 0.0)$ in linear space become $(1.0, 0.0, 0.0)$
- Colors still have to output to the monitor display, the original color gets displayed on the monitor as $(0.218, 0.0, 0.0)$
- Issue: double the dark-red light in linear space, it becomes more than 4.5 times as bright on the monitor!



Introduction

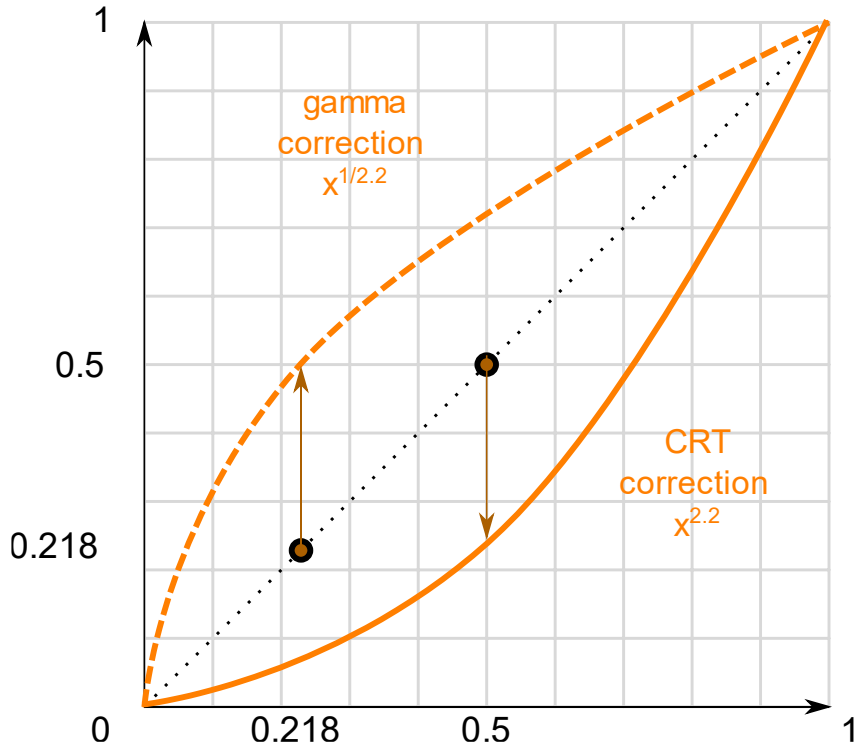
- Up until now, assumed we were working in linear space
- Working in the color space defined by the monitor's output color space → colors and lighting weren't physically correct
- Thus, we (and artists) generally set lighting values way brighter than they should be (because monitor darkens them) → makes most linear-space calculations incorrect
- Note, the monitor graph and the linear graph both start and end up at the same position; intermediate colors get darkened by the display

Introduction

- Colors configured based on the monitor's display → all intermediate (lighting) calculations in linear-space are physically incorrect
- Becomes more and more obvious as more advanced lighting algorithms are used, as you can see in the image below:

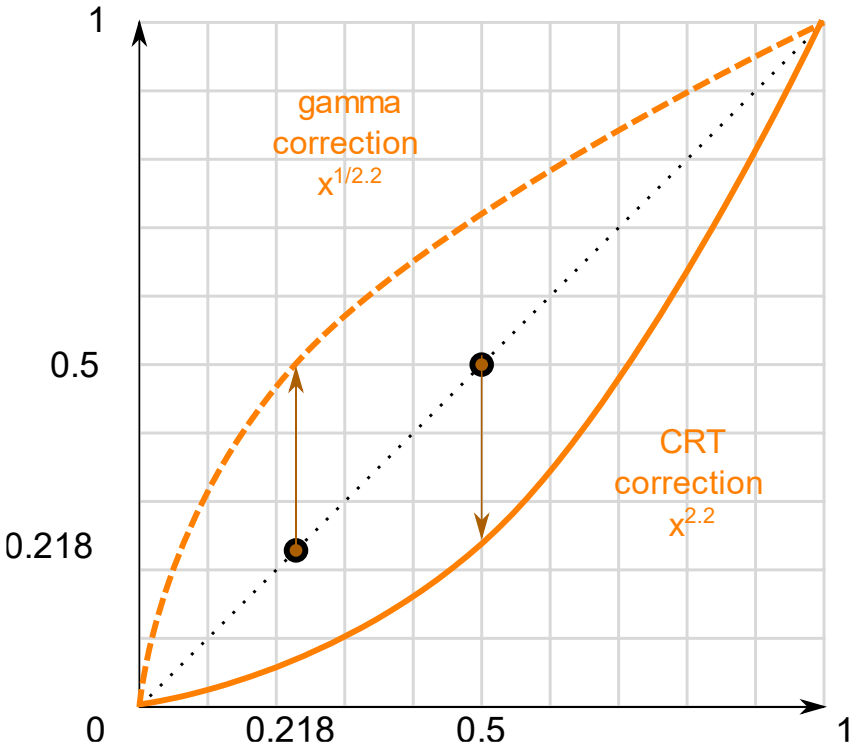
Gamma Correction

- Idea of gamma correction is to apply the inverse of the monitor's gamma to the final output color
- See another dashed line that is the inverse of the monitor's gamma curve
- Multiply linear output colors by this inverse gamma curve (brighter), colors displayed on the monitor, gamma curve is applied → colors become linear
- Basically we make the intermediate colors brighter so that as soon as the monitor darkens them, it balances all out



Gamma Correction

- E.g., dark-red color $(0.5, 0.0, 0.0)$, apply the gamma correction curve to the color value
- Linear colors displayed by a monitor roughly scaled to a power of 2.2, inverse is a scaling by a power of 1/2.2
- The gamma-corrected dark-red color thus becomes $(0.5, 0.0, 0.0)^{1/2.2} \approx (0.73, 0.0, 0.0)$
- Resulting color is displayed on monitor as $(0.73, 0.0, 0.0)^{2.2} = (0.5, 0.0, 0.0)$



Gamma Correction

2.2 is a default gamma value 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

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

Gamma Correction

- Two ways to apply gamma correction:
  - Using OpenGL's built-in sRGB framebuffer support
  - Doing the gamma correction manually in the fragment shaders
- First option easiest, but less control
- By enabling GL_FRAMEBUFFER_SRGB, subsequent drawing commands gamma correct colors from the sRGB color space (before store color buffer)
- sRGB color space roughly corresponds to gamma of 2.2
- After enabling perform gamma correction after each fragment shader run to all subsequent framebuffers, including the default framebuffer

Gamma Correction

- Enabling GL_FRAMEBUFFER_SRGB:

```
glEnable(GL_FRAMEBUFFER_SRGB);
```

- Rendered images will be gamma corrected
- Note, with these approaches gamma correction (also) transforms the colors from linear space to non-linear space → important to do gamma correction at the last and final step
- Gamma-correct colors before the final output → all subsequent operations on those colors will operate on incorrect values
- E.g., 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

Gamma Correction

- Second approach more work, but control over the gamma operations
- Apply gamma correction at the end of each relevant fragment shader, colors gamma corrected before being sent out to the monitor:

```
void main()
{
    …
    float gamma = 2.2;
    FragColor.rgb = pow(FragColor.rgb, vec3(1.0/gamma));
}
```

Gamma Correction

- Issue: to be consistent have to apply gamma correction to each relevant fragment shader (a dozen fragment shaders for multiple objects → add the gamma correction to each of these shaders)
- Easier solution: post-processing stage and apply gamma correction on the post-processed quad as a final step (do once)
- These one-liners represent the technical implementation of gamma correction
- Not all too impressive, but there are a few extra things you have to consider when doing gamma correction

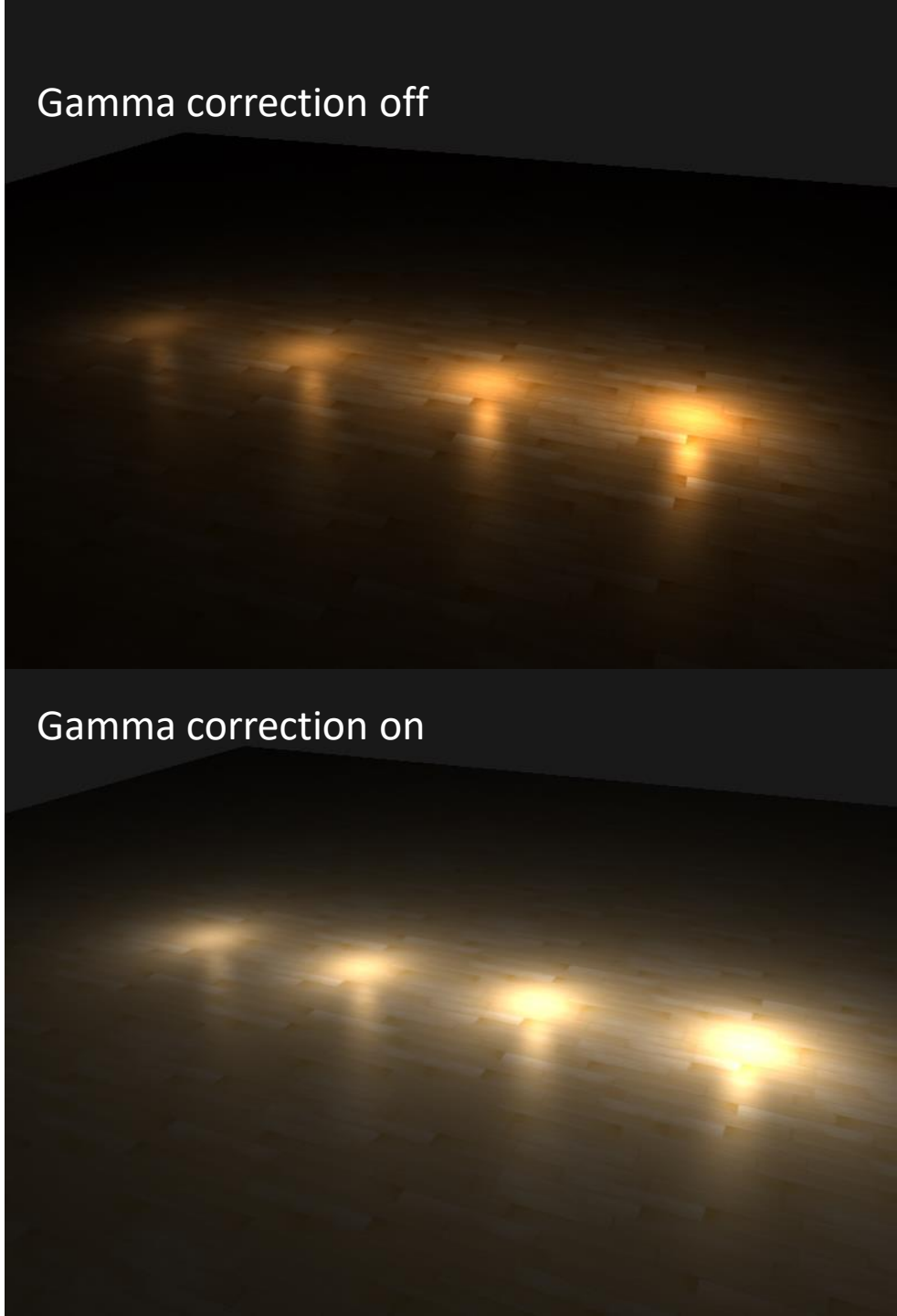
sRGB Textures

Introduction

- Monitors always display colors with gamma applied in sRGB space, 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 all the pictures 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 your perceived brightness, does not equal double the red component

Introduction

- As a result, texture artists create textures in sRGB (if we use those textures in the rendering, we have to take this into account)
- Before we applied gamma correction this was not an issue (textures looked good in sRGB, without gamma correction, also worked in sRGB → textures displayed exactly as they are which was fine)
- Now, displaying everything in linear space → texture colors will be off



sRGB Textures

- To fix this, make sure texture artists work in linear space
- Easier to work in sRGB this is probably not the preferred solution
- The other solution: re-correct or transform these sRGB textures back to linear space:

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

sRGB Textures

- For each texture in sRGB is troublesome
- OpenGL gives us yet another solution to our problems by giving us the GL_SRGB and GL_SRGB_ALPHA internal texture formats.

sRGB Textures

- If creating a texture with any of these two sRGB texture formats, it will automatically correct the colors to linear-space as soon as we use them
- We can specify a texture as an sRGB texture as follows:

```
glTexImage2D(GL_TEXTURE_2D, 0, GL_SRGB, width, height, 0, GL_RGB,
                                                    GL_UNSIGNED_BYTE, image);
```

- To include alpha components specify the texture's internal format as GL_SRGB_ALPHA

sRGB Textures

- Careful when specifying textures in sRGB (not all textures in sRGB)
- Textures for coloring objects (diffuse textures) mostly in sRGB
- Textures for retrieving lighting parameters, e.g., specular and normal maps mostly in linear space (configure these as sRGB textures → lighting will break down)
- Diffuse textures specified as sRGB textures get the expected visual output, but this time everything is gamma corrected only once

Attenuation

Attenuation

- Lighting attenuates closely inversely proportional to the squared distance from a light source:

```
float attenuation = 1.0 / (distance * distance);
```

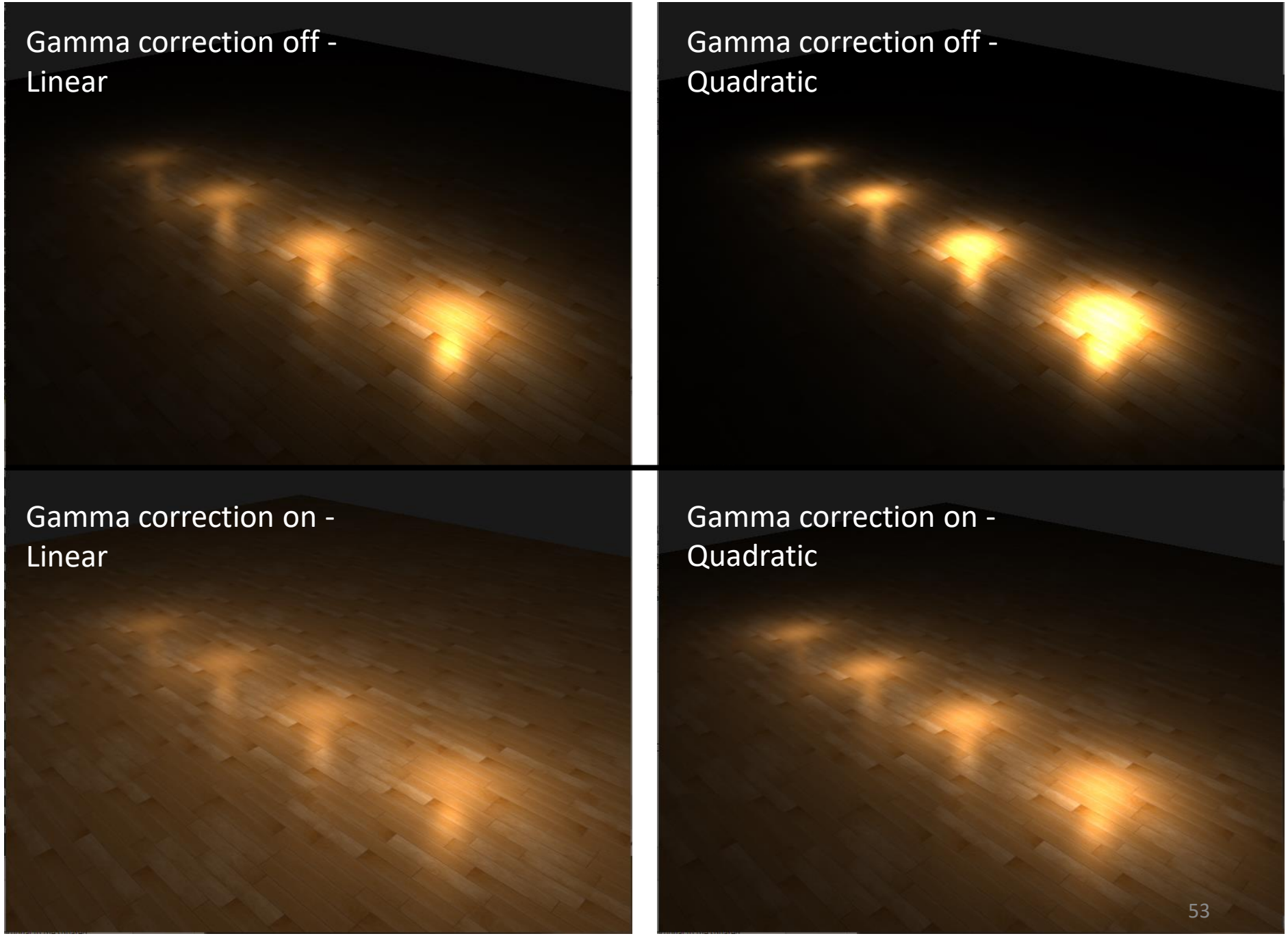
Attenuation

- This attenuation effect is always way too strong (giving lights a small radius that didn’t look physically right)
- Thus, other attenuation functions were used (see Lighting lecture):

```
float attenuation = 1.0 / distance;
```

F5…

- … nice!


Gamma correction off - Linear
Gamma correction off - Quadratic
Gamma correction on - Linear
Gamma correction on - Quadratic

Attenuation

- Cause: light attenuation change brightness, (not visualizing scene in linear space → chose the attenuation functions that looked best on our monitor, but weren't physically correct)
- Squared attenuation function without gamma correction effectively becomes: $(1/distance^2)^{2.2}$ (displayed on a monitor)
- Creates larger attenuation effect without gamma correction
- Linear equivalent makes much more sense without gamma correction becomes: $(1/distance)^{2.2}$ (physical equivalent a lot more)

Attenuation

The more advanced attenuation function, we discussed in the lighting lecture is still useful in gamma corrected scenes as it gives much more control over the exact attenuation (but of course requires different parameters in a gamma corrected scene).

Attenuation

- Let’s create the scene where we can change between gamma correction, and the linear and quadratic function
- Add bools for those (change with key pressed)

```
bool gammaEnabled = false;
bool quaLin = false;
bool gammaKeyPressed = false;
bool quaLinKeyPressed = false;
```

Attenuation

- We will load one floor texture in sRGB and one in linear space:

```
// sRGB
glTexImage2D(GL_TEXTURE_2D, 0, GL_SRGB, width, height, 0, GL_RGB,
GL_UNSIGNED_BYTE, data);
…
// RGB
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width, height, 0, GL_RGB,
GL_UNSIGNED_BYTE, data);
```

Attenuation

- Check if space or L key was pressed:

```
void processInput(GLFWwindow *window)
{
   …
    if (glfwGetKey(window, GLFW_KEY_SPACE) == GLFW_PRESS && !gammaKeyPressed)
    {
        gammaEnabled = !gammaEnabled;
        gammaKeyPressed = true;
    }
    if (glfwGetKey(window, GLFW_KEY_SPACE) == GLFW_RELEASE)
    {
        gammaKeyPressed = false;
    }

    if (glfwGetKey(window, GLFW_KEY_L) == GLFW_PRESS && !quaLinKeyPressed)
    {
        quaLin = !quaLin;
        quaLinKeyPressed = true;
    }
    if (glfwGetKey(window, GLFW_KEY_L) == GLFW_RELEASE)
    {
        quaLinKeyPressed = false;
    }
}
```

Attenuation

- Set options as uniforms to the shader:

```
shader.setInt("gamma", gammaEnabled);
shader.setInt("quaLin", quaLin);
```

Attenuation

- Change attenuation and gamma option (fragment shader):

```
…
float attenuation = 1.0/ (quaLin ? distance * distance : distance);
diffuse *= attenuation;
specular *= attenuation;
…
if(gamma)
   color = pow(color, vec3(1.0/2.2));
FragColor = vec4(color, 1.0);
```

Summary

- Gamma correction allows to work/visualize renders in linear space
- Linear space makes sense in the physical world, most physical equations now actually give good results like real light attenuation
- The more advanced 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

Questions???