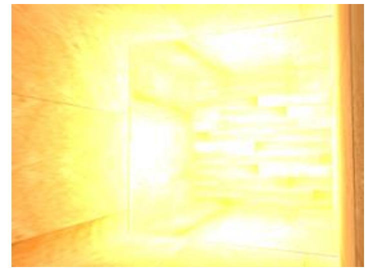
HDR



Brightness and color values, by default, are clamped between `0.0` and `1.0` when stored in a framebuffer.
- However, if fragments are being lit by multiple bright light sources at once, the sum of its color components may exceed `1.0`. When this happens, all fragments that have a brightness or color sum over `1.0` get clamped to `1.0`, which looks like the image to the right.
- Due to a large number of fragments' color values getting clamped to `1.0`, each of the bright fragments have the exact same white color value in large regions, losing a significant amount of detail and looking unrealistic.

One solution to the above problem is to reduce the strength of the light sources and ensure no area of fragments in your scene ends up brighter than `1.0`.
- This is not a good solution because it forces you to use unrealistic lighting parameters.

A better approach is to allow color values to temporarily exceed `1.0` and transform them back to the original range `[0.0, 1.0]` as a final step, but without losing detail.

Non-HDR monitors are limited to display colors in the range `[0.0, 1.0]`, but there is no such limitation in lighting equations. By allowing fragments to exceed `1.0`, we have a much higher range of color values available to work in, known as high dynamic range (HDR).

- With HDR, bright things can be really bright, dark things can be really dark, and details can be seen in both.
- HDR was originally only used for photography where a photographer takes multiple pictures of the same scene with varying exposure levels, capturing a large range of color values. Combining these forms an HDR image where a large range of details are visible based on the combined exposure levels, or a specific exposure it is viewed with.
  - The image to the right shows a lot of detail at brightly lit regions with a low exposure (look at the window), but these details are gone with a high exposure.
  - However, the high exposure image reveals a great amount of detail at darker regions that weren't previously visible.
  - This is also very similar to how the human eye works and the basis of HDR rendering.
    - When there is little light, the human eye adapts itself so the darker parts become more visible and similarly for bright areas.

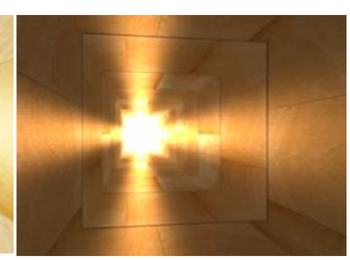
HDR works by allowing a much larger range of color values to render to, collecting a large range of dark and bright details of a scene, and, at the end, transforming all the HDR values back to the low dynamic range (LDR) of `[0.0, 1.0]`.
- This process of converting HDR values to LDR values is called **tone mapping**.
  - A large collection of tone mapping algorithms exist that aim to preserve most HDR details during the conversion process. These tone mapping algorithms often involve an exposure parameter that selectively favor dark or bright regions.

When it comes to real-time rendering, HDR allows us to not only exceed the LDR range of `[0.0, 1.0]` and preserve more detail, but also gives us the ability to specify a light source's intensity by their real intensities.
- Example: The sun has a much higher intensity than something like a flashlight, so why not configure the sun as such (e.g. a diffuse brightness of `100.0`). This wouldn't be possible with LDR rendering, since the values get clamped to `1.0`.

As (non-HDR) monitors only display colors in the range `[0.0, 1.0]`, we do need to transform the currently high dynamic range of color values back to the monitor's range.
- We can't just re-transform the colors back with a simple average because brighter areas then become a lot more dominant.
- What we can do, instead, is use different equations and/or curves to transform the HDR values back to LDR that give us complete control over the scene's brightness.
  - This is tone mapping, the final step of HDR rendering.

Floating Point Framebuffers

To implement HDR rendering, we need a way to prevent color values getting clamped after each fragment shader run.
- When framebuffers use a normalized fixed-point color format (like GL_RGB) as their color buffer's internal format, OpenGL automatically clamps the values between `0.0` and `1.0` before storing them in the framebuffer. This operation holds true for most types of framebuffer formats, except for floating point formats.
- When the internal format of a framebuffer's color buffer is specified as `GL_RGB16F`, `GL_RGBA16F`, `GL_RGB32F`, or `GL_RGBA32F`, the framebuffer is known as a **floating point framebuffer** that can store floating point values outside the default range of `[0.0, 1.0]`, perfect for HDR.
  - The default framebuffer of OpenGL (by default) only takes up 8 bits per color component.
  - With a floating point framebuffer with 32 bits per color component (when using `GL_RGB32F` or `GL_RGBA32F`), we're using 4 times more memory for storing colors values.
    - As 32 bits isn't really necessary (unless you need a high level of precision), using `GL_RGBA16F` (16 bits) will suffice.



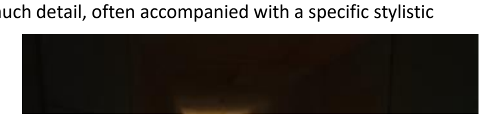
Create a floating point framebuffer that uses the GL_RGBA16F internal format for the color buffer.
- Make sure to also add a depth attachment to the floating point framebuffer.
- Floating Point Framebuffer

From here, we first render a lit scene into the floating point framebuffer and then display the framebuffer's color buffer on a screen-filled quad, which will look somewhat like this.
- Here, a scene's color values are filled into a floating point color buffer that can contain any arbitrary color value, possibly exceeding `1.0`.

For this chapter, a simple demo scene was created with a large stretched cube acting as a tunnel with four point lights, one being extremely bright (positioned at the end of the tunnel). The code for setting up this scene can be found here.
- It becomes clear, as seen in the images above, that the intense light values at the end of the tunnel are clamped to `1.0` since a large portion of it is completely white, losing a lot of lighting details.
  - As we directly write HDR values to an LDR output buffer, it is as if we don't have HDR enabled at all.
  - We need to transform all the floating poit color values into the `[0.0, 1.0]` range without losing any of its detail. We can do so using tone mapping.

Tone Mapping

Tone mapping is the process of transforming floating point color values to the expected [0.0, 1.0] range (LDR) without losing too much detail, often accompanied with a specific stylistic color balance.



One of the more simple tone mapping algorithms is Reinhard tone mapping.
- This method involves dividing the entire HDR color values to LDR color values, evenly balancing out all brightness values onto

Tone mapping is the process of transforming floating point color values to the expected [0.0, 1.0] range (LDR) without losing too much detail, often accompanied with a specific stylistic color balance.

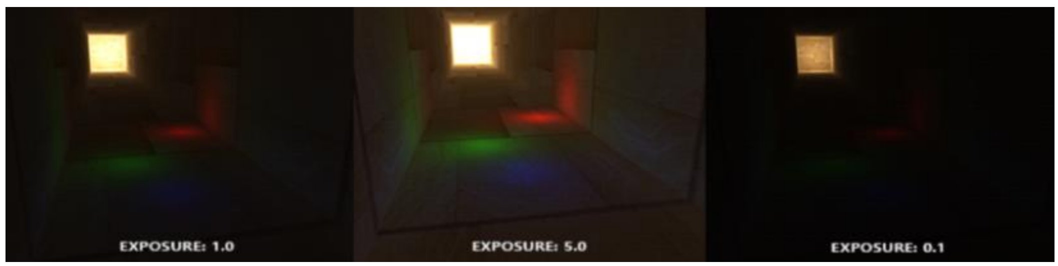
One of the more simple tone mapping algorithms is Reinhard tone mapping.
- This method involves dividing the entire HDR color values to LDR color values, evenly balancing out all brightness values onto LDR.

Implement Reinhard tone mapping in the HDR fragment shader with gamma correction.
- Don't forget to remove the gamma correction from the vertex shader so that you're not gamma correcting twice.
- Reinhard Tone Mapping

As you can see in the image to the right, with Reinhard tone mapping, we no longer lose any detail at the bright areas of our scene; we can see the wood texture pattern at the end of the tunnel. It does tend to slightly favor bright areas, making darker regi ons seem less detailed and distinct.
- **NOTE:** I disabled specular highlights when taking the screenshot of the image to the right for the sake of viewing the colored lights. As it stands, the ambient, diffuse, and specular calculations all get multiplied with the light color, which (if it exceeds `1.0` in any color component) are still very bright and make it hard to see the other lights.
- We can now properly see the entire range of HDR values stored in the floating point framebuffer, giving us precise control over the scene's lighting without losing details.
  - **NOTE:** We could also directly tone map at the end of our lighting shader, not needing any floating point framebuffer at all. However, as scenes get more complex, you'll frequently find the need to store intermediate HDR results as floating point buffers, so this is a good exercise.

You can also use tone mapping to allow the use of an exposure parameter.
- If you remember from the beginning of this chapter, HDR images contain lots of details visible at different exposure levels.
  - Example: If we have a scene that features a day and night cycle, it makes sense to use a lower exposure at daylight and a higher exposure at night time, similar to how the human eye adapts.

We can slightly modify our current HDR fragment shader to implement exposure tone mapping like so.
- With higher exposure values, the darker areas of the tunnel show significantly more detail.
- In contrast, a low exposure largely removes the dark region details, but allows us to see more detail in the bright areas of a scene.



More HDR

The two tone mapping algorithms we used are only a few of a large collection of (more advanced) tone mapping algorithms of which each has their own strengths and weaknesses.
- Some tone mapping algorithms favor certain colors/intensities above others.
- Some algorithms display both the low and high exposure colors at the same time to create more colorful and detailed images.
- There is also a collection of techniques known as **automatic exposure adjustment** or **eye adaptation** techniques that determine the brightness of the scene in the previous frame and (slowly) adapt the exposure parameter such that the scene gets brighter in dark areas or darker in bright areas, mimicking the human eye.

The real benefit of HDR rendering really shows itself in large and complex scenes with heavy lighting algorithms.
HDR also makes several other interesting effects more feasible and realistic; one of these effects is bloom, which we'll discuss in the next chapter.

Floating Point Framebuffer

Sunday, June 12, 2022 4:39 PM

```
unsigned int hdrTexture;
glGenTextures(1, &hdrTexture);
glBindTexture(GL_TEXTURE_2D, hdrTexture);
// This texture supports HDR due to using the GL_RGBA16F internal format. Any framebuffers that have this texture attached are thus
// considered floating point framebuffers.
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA16F, WIDTH, HEIGHT, 0, GL_RGBA, GL_FLOAT, NULL);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);

unsigned int depthStencilRBO;
glGenRenderbuffers(1, &depthStencilRBO);
glBindRenderbuffer(GL_RENDERBUFFER, depthStencilRBO);
glRenderbufferStorage(GL_RENDERBUFFER, GL_DEPTH24_STENCIL8, WIDTH, HEIGHT);

unsigned int hdrFBO;
glGenFramebuffers(1, &hdrFBO);
glBindFramebuffer(GL_FRAMEBUFFER, hdrFBO);

glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0, GL_TEXTURE_2D, hdrTexture, 0);
glFramebufferRenderbuffer(GL_FRAMEBUFFER, GL_DEPTH_STENCIL_ATTACHMENT, GL_DEPTH24_STENCIL8, depthStencilRBO);

if (glCheckFramebufferStatus(GL_FRAMEBUFFER) != GL_FRAMEBUFFER_COMPLETE) {
   std::cout << "ERROR::FRAMEBUFFER:: Framebuffer is not complete!" << std::endl;
}
```

Example HDR Rendering

Sunday, June 12, 2022 5:00 PM

```
glBindFramebuffer(GL_FRAMEBUFFER, hdrFBO);
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
// Render the (lit) scene
[...]
glBindFramebuffer(GL_FRAMEBUFFER, 0);

// Render the HDR color buffer to a 2D screen-filling quad using the tone mapping shader
hdrShader.use();
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, hdrColorBufferTexture);
RenderQuad();
```

Scene Setup

Sunday, June 12, 2022 7:25 PM

<u>hdr.vert</u>

```
#version 330 core

layout (location = 0) in vec2 aPos;
layout (location = 1) in vec2 aTexCoords;

out vec2 texCoords;

void main() {
   texCoords = aTexCoords;

   gl_Position = vec4(aPos, 0.0, 1.0);
}
```

<u>hdr.frag</u>

```
#version 330 core

out vec4 FragColor;

in vec2 texCoords;

uniform sampler2D tex;

void main() {
   vec3 hdrColor = texture(tex, texCoords).rgb;
   FragColor = vec4(hdrColor, 1.0);
}
```

<u>main()</u>

```
std::vector<glm::vec3> lightColors;
lightColors.push_back(glm::vec3(200.0f, 200.0f, 200.0f));
lightColors.push_back(glm::vec3(0.1f, 0.0f, 0.0f));
lightColors.push_back(glm::vec3(0.0f, 0.0f, 0.2f));
lightColors.push_back(glm::vec3(0.0f, 0.1f, 0.0f));

std::vector<glm::vec3> lightPositions;
lightPositions.push_back(glm::vec3(0.0f, 0.0f, 49.5f)); // back light
lightPositions.push_back(glm::vec3(-1.4f, -1.9f, 9.0f));
lightPositions.push_back(glm::vec3(0.0f, -1.8f, 4.0f));
lightPositions.push_back(glm::vec3(0.8f, -1.7f, 6.0f));
```

<u>Render Loop</u>

```
glBindFramebuffer(GL_FRAMEBUFFER, hdrFBO);
glClearColor(0.2f, 0.2f, 0.2f, 1.0f);
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

blinnPhongShader.use();

planksDiffuseTexture.bind();
fullSpecularTexture.bind();

// ============================================== VERTEX SHADER UNIFORMS ============================================== //
glm::mat4 objModel{ 1.0f };
objModel = glm::translate(objModel, glm::vec3(0.0f, 0.0f, 25.0f));
objModel = glm::scale(objModel, glm::vec3(2.5f, 2.5f, 27.5f));

glUniformMatrix4fv(glGetUniformLocation(blinnPhongShader.id, "model"), 1, GL_FALSE, glm::value_ptr(objModel));
glUniformMatrix4fv(glGetUniformLocation(blinnPhongShader.id, "view"), 1, GL_FALSE, glm::value_ptr(view));
glUniformMatrix4fv(glGetUniformLocation(blinnPhongShader.id, "projection"), 1, GL_FALSE, glm::value_ptr(projection));

glUniform3fv(glGetUniformLocation(blinnPhongShader.id, "viewPos"), 1, glm::value_ptr(camera.position));
glUniform1i(glGetUniformLocation(blinnPhongShader.id, "inverseNormals"), true);

// ============================================= FRAGMENT SHADER UNIFORMS ============================================= //
// Material uniforms
glUniform1i(glGetUniformLocation(blinnPhongShader.id, "material.diffuse"), planksDiffuseTexture.getTexUnit());
glUniform1i(glGetUniformLocation(blinnPhongShader.id, "material.specular"), fullSpecularTexture.getTexUnit());
glUniform1f(glGetUniformLocation(blinnPhongShader.id, "material.shininess"), 32.0f);
// PointLight uniforms
for (int i = 0; i < 4; ++i) {
   glUniform3fv(glGetUniformLocation(blinnPhongShader.id, ("pointLights[" + std::to_string(i) + "].position").c_str()),
       1, glm::value_ptr(lightPositions[i]));
```

```cpp
    glUniform1f(glGetUniformLocation(blinnPhongShader.id, ("pointLights[" + std::to_string(i) + "].constant").c_str()), 1.0f);
    glUniform1f(glGetUniformLocation(blinnPhongShader.id, ("pointLights[" + std::to_string(i) + "].linear").c_str()), 0.7f);
    glUniform1f(glGetUniformLocation(blinnPhongShader.id, ("pointLights[" + std::to_string(i) + "].quadratic").c_str()), 1.8f);
    glUniform3fv(glGetUniformLocation(blinnPhongShader.id, ("pointLights[" + std::to_string(i) + "].ambient").c_str()),
        1, glm::value_ptr(glm::vec3(0.05f) * lightColors[i]));
    glUniform3fv(glGetUniformLocation(blinnPhongShader.id, ("pointLights[" + std::to_string(i) + "].diffuse").c_str()),
        1, glm::value_ptr(glm::vec3(1.0f) * lightColors[i]));
    glUniform3fv(glGetUniformLocation(blinnPhongShader.id, ("pointLights[" + std::to_string(i) + "].specular").c_str()),
        1, glm::value_ptr(glm::vec3(1.0f) * lightColors[i]));
}
// ============================================================================================================== //

glBindVertexArray(megaCubeVAO);
glDrawArrays(GL_TRIANGLES, 0, 36);

glBindFramebuffer(GL_FRAMEBUFFER, 0);
glClearColor(1.0f, 1.0f, 1.0f, 1.0f);
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

hdrShader.use();
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, hdrTexture);

glUniform1i(glGetUniformLocation(hdrShader.id, "tex"), 0);

glBindVertexArray(screenQuadVAO);
glDrawArrays(GL_TRIANGLES, 0, 6);
```

screenVAO and megaCubeVAO are frequently used VAOs from past chapters, and blinnPhongShader is the shader we made in the Advanced Lighting chapter. If you do not have them, I believe you can figure out how to set them up :)

Reinhard Tone Mapping

Sunday, June 12, 2022 7:51 PM

hdr.frag

```
void main() {
   const float gamma = 2.2;
   vec3 hdrColor = texture(tex, texCoords).rgb;

   // Reinhard Tone Mapping
   vec3 mapped = hdrColor / (hdrColor + vec3(1.0));
   // Gamma Correction
   mapped = pow(mapped, vec3(1.0 / gamma));

   FragColor = vec4(mapped, 1.0);
}
```

Exposure Tone Mapping

Sunday, June 12, 2022 8:19 PM

```
...
uniform float exposure;

void main() {
   const float gamma = 2.2;
   vec3 hdrColor = texture(tex, texCoords).rgb;

   // Exposure Tone Mapping
   vec3 mapped = vec3(1.0) - exp(-hdrColor * exposure);
   // Gamma Correction
   mapped = pow(mapped, vec3(1.0 / gamma));

   FragColor = vec4(mapped, 1.0);
}
```