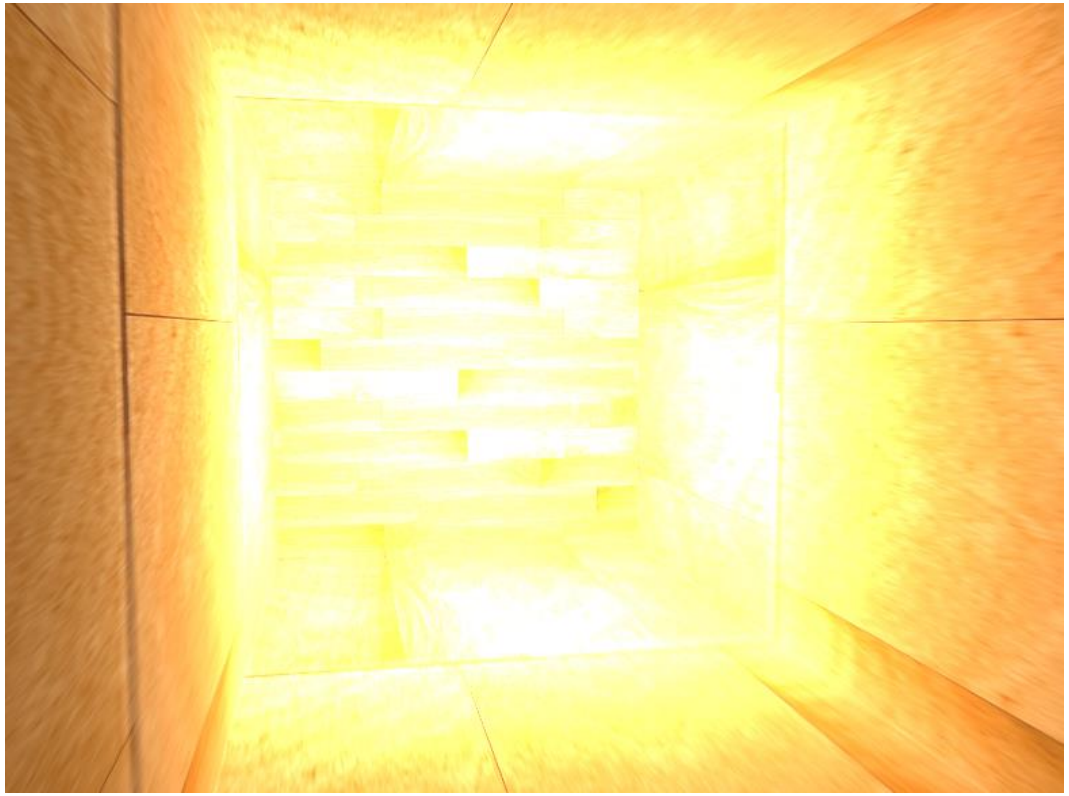
Computer Graphics II
– HDR

Kai Lawonn

Introduction

- Brightness and color values by default are clamped between $[0.0,1.0]$ when stored into a framebuffer
- Thus, always specify light and color values somewhere in this range
- Gives decent results, but what if we walk in an area with multiple bright light sources that as a total sum exceed 1.0?
- Answer: all fragments with a brightness or color sum over 1.0 get clamped to 1.0, is not pretty



Introduction

- Large number of fragments' color values clamped to 1.0 → bright fragments have the exact same white color in a large region → losing detail and it look fake
- Solution would be to reduce the strength of the light sources and ensure no area of fragments in your scene ends up brighter than 1.0 → not good it forces 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 of 0.0 and 1.0 as a final step, but without losing detail

Introduction

- Monitors are limited to display colors in the range of 0.0 and 1.0, but there is no such limitation in lighting equations
- Allowing fragment colors to exceed 1.0 → much higher range of color values available to work in known as high dynamic range (HDR)
- With high dynamic range bright things can be really bright, dark things can be really dark, and details can be seen in both

HDR

- HDR originally used for photography only (take multiple pictures with varying exposure levels, capturing a large range of color values)
- Combined images form an HDR image (large range of details are visible based on the combined exposure levels)

HDR

- Very similar to how the human eye works and the basis of HDR rendering
- Little light → eye adapts so darker parts are better visible (bright areas similar)
- Like human eye has an automatic exposure slider based on the scene's brightness

HDR

- HDR rendering works a bit like that
- Allow for a much larger range of color values to render (collecting a large range of dark and bright details of a scene), at the end transform HDR values back to the low dynamic range (LDR) $[0.0, 1.0]$
- Converting HDR to LDR values is called tone mapping (a lot of tone mapping algorithms aim to preserve most HDR details)
- These tone mapping algorithms often involve an exposure parameter that selectively favors dark or bright regions

HDR

- In real-time rendering HDR allows not only to exceed LDR of $[0,1]$, but gives ability to specify light source's intensity by their real intensities
- E.g., sun higher intensity than, e.g., flashlight so configure sun as such (like a diffuse brightness of 10.0)
- Allows to more properly configure a scene's lighting with more realistic lighting parameters (would not be possible with LDR)

HDR

- Monitors only display colors in $[0,1]$ → transform the currently HDR color values back to the monitor's range
- Re-transforming colors back with an average not good, brighter areas then become a lot more dominant
- Use different equations/curves to transform HDR values back to LDR -> → complete control over the scene's brightness
- This process (tone mapping) is the final step of HDR rendering

Floating Point Framebuffers

Introduction

- To implement HDR rendering need a way to prevent color values getting clamped after each fragment shader run
- Framebuffers use a normalized fixed-point color format (like GL_RGB) as their colorbuffer's internal format OpenGL automatically clamps the values $[0,1]$ before storing them in the framebuffer
- Holds for most types of framebuffer formats, except for floating point formats that are used for their extended range of values

Floating Point Framebuffers

- Internal format of framebuffer's colorbuffer is specified as: GL_RGB16F, GL_RGBA16F, GL_RGB32F or GL_RGBA32F framebuffer is known as floating point framebuffer → can store floating point values outside the default range of 0.0 and 1.0
- Perfect for HDR rendering
- To create a floating point framebuffer, only change its internal format:

```
glBindTexture(GL_TEXTURE_2D, colorBuffer);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA16F, SCR_WIDTH, SCR_HEIGHT, 0, GL_RGBA,
        GL_FLOAT, NULL);
```

Floating Point Framebuffers

- Default framebuffer of OpenGL takes 8 bits per color component
- Floating point framebuffer 32 bits per color component (GL_RGB32F or GL_RGBA32F) → 4 times more memory for storing color values
- 32 bits not necessary unless high level of precision (GL_RGBA16F will suffice)
- With floating point colorbuffer attached to framebuffer → render the scene into this framebuffer and colors will not clamped to $[0,1]$

Floating Point Framebuffers

- First render a lighted scene into the floating point framebuffer and then display the framebuffer's colorbuffer on a screen-filled quad:

```
glBindFramebuffer(GL_FRAMEBUFFER, hdrFBO);
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
// [...] render (lighted) scene
glBindFramebuffer(GL_FRAMEBUFFER, 0);

// now render hdr colorbuffer to 2D screen-filling quad with different shader
hdrShader.use();
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, hdrColorBufferTexture);
RenderQuad();
```

Floating Point Framebuffers

- Simple demo scene with a large stretched cube acting as a tunnel with four point lights, one being extremely bright positioned at the tunnel's end:

```
// colors
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));
```

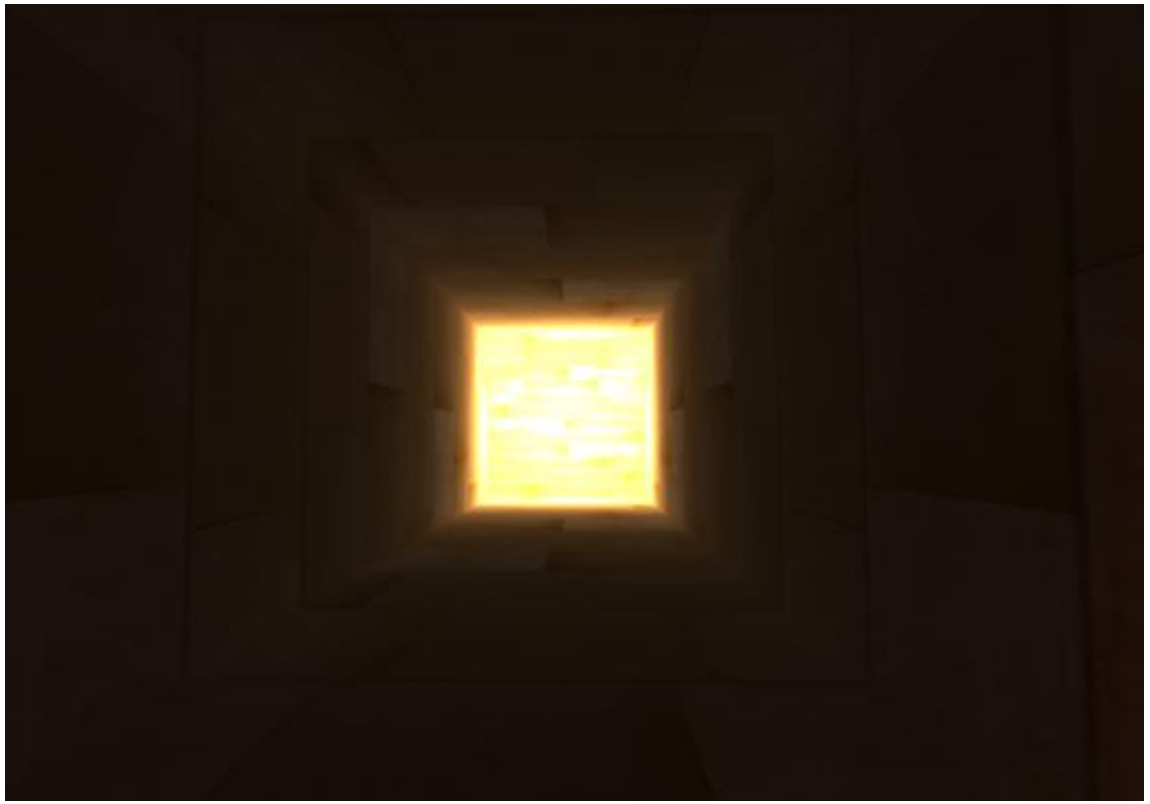
Floating Point Framebuffers

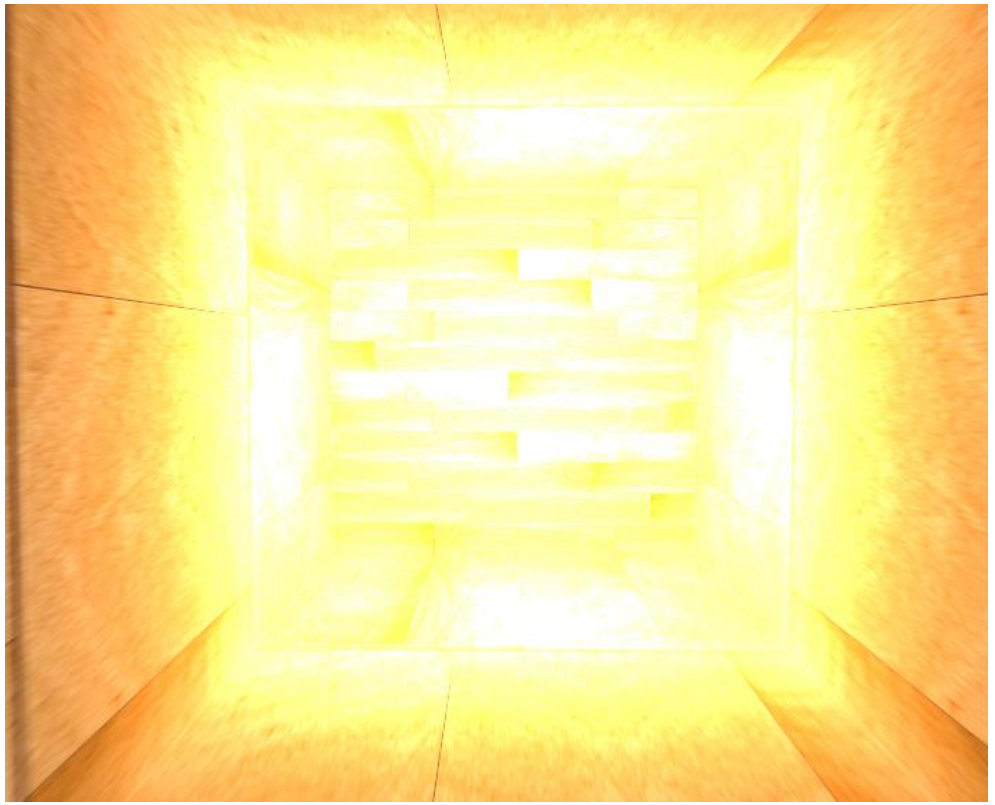
- Rendering into the floating point framebuffer exactly the same as we would normally render into a framebuffer
- New is hdrShader's fragment shader (renders final 2D quad with the floating point colorbuffer texture attached)
- Define a simple pass-through fragment shader first:

```
#version 330 core
out vec4 FragColor;
in vec2 TexCoords;
uniform sampler2D hdrBuffer;
void main()
{
   vec3 hdrColor = texture(hdrBuffer, TexCoords).rgb;
   FragColor = vec4(hdrColor, 1.0);
}
```

F5…

- … light values clamped.





Tone Mapping

Tone Mapping

- It transforms floating point color values to $[0, 1]$ (LDR) without losing too much detail (often accompanied with stylistic color balance)
- Simplest algorithm is known as Reinhard tone mapping: dividing HDR color values to LDR color values evenly balancing them all out
- The Reinhard tone mapping algorithm evenly spreads out all brightness values onto LDR

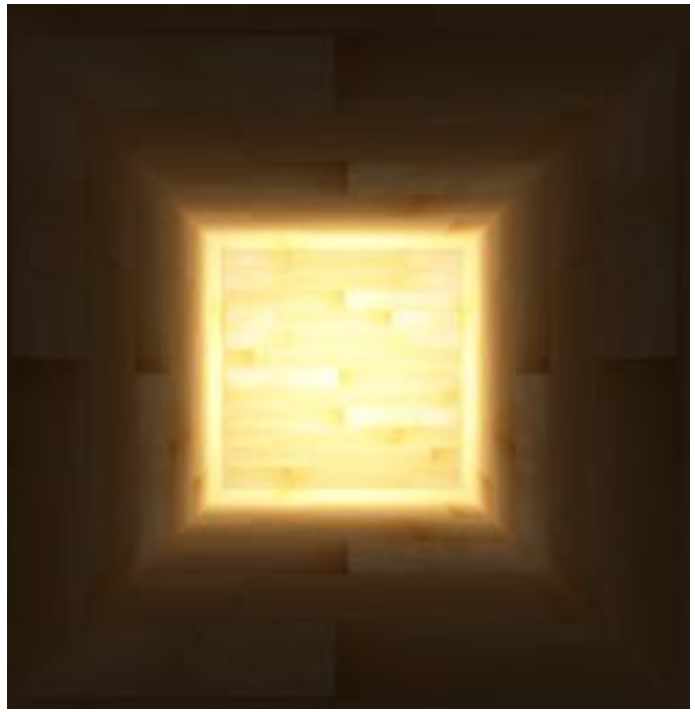
Tone Mapping

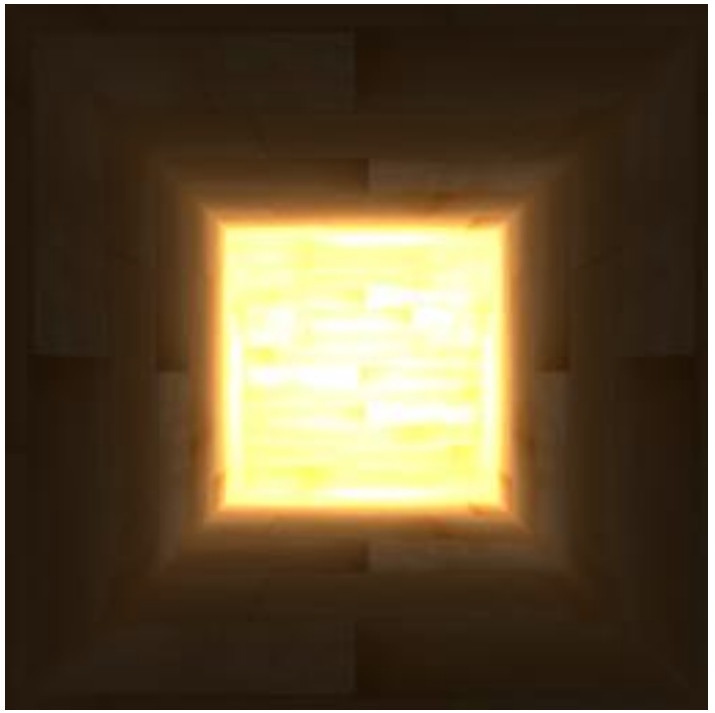
- Include Reinhard tone mapping into the previous fragment shader and add a gamma correction filter for good measure (including the use of SRGB textures):

```
void main()
{
    const float gamma = 2.2;
    vec3 hdrColor = texture(hdrBuffer, 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);
}
```

F5…

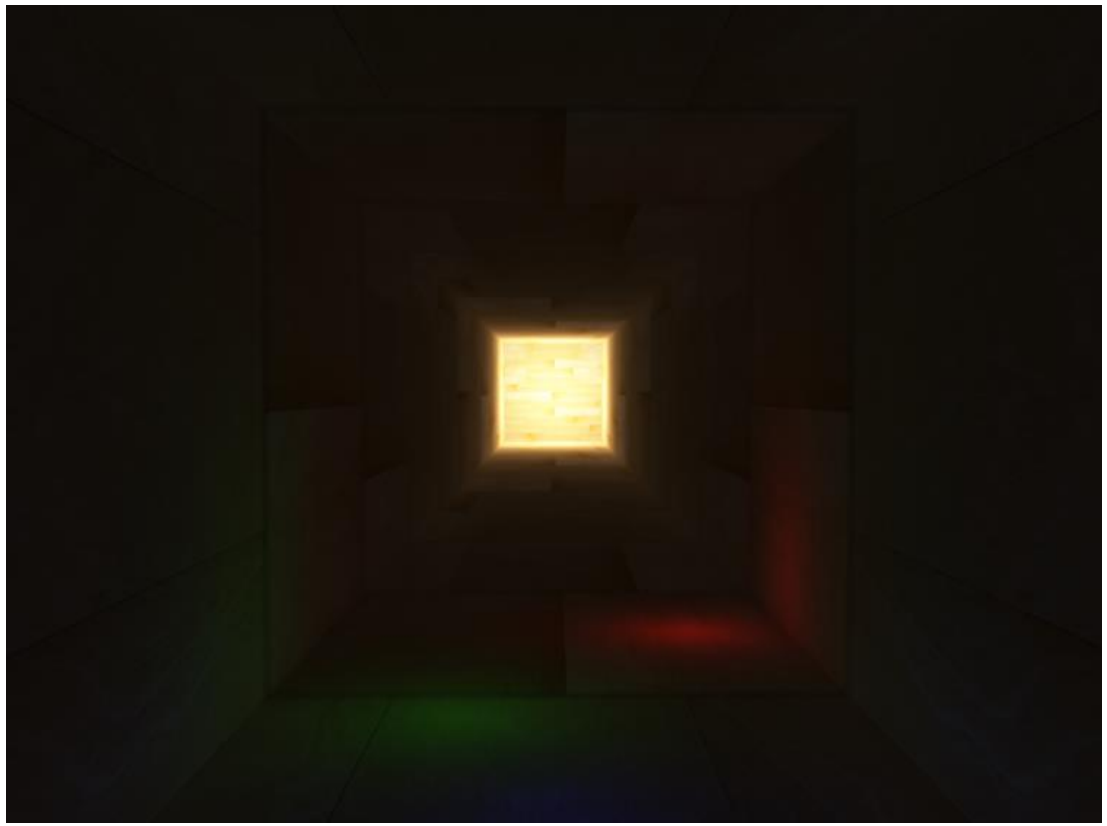
- … left: with Reinhard, right without





F5…

- Tend to slightly favor brighter areas, making darker regions seem less detailed and distinct



Note

Could also directly tone map at the end of lighting shader (no need of any floating point framebuffer)

However, as scenes get more complex, the need to store intermediate HDR results as floating point buffers might be a good choice

Tone Mapping

- Another use of tone mapping is the use of an exposure parameter
- HDR images contain a lot of details visible at different exposure levels
- With a scene featuring day and night cycle, could use a lower exposure at daylight and a higher exposure at night time (human eye)
- Exposure parameter allows to configure lighting parameters that work at day and night under different lighting conditions (change the exposure parameter only)

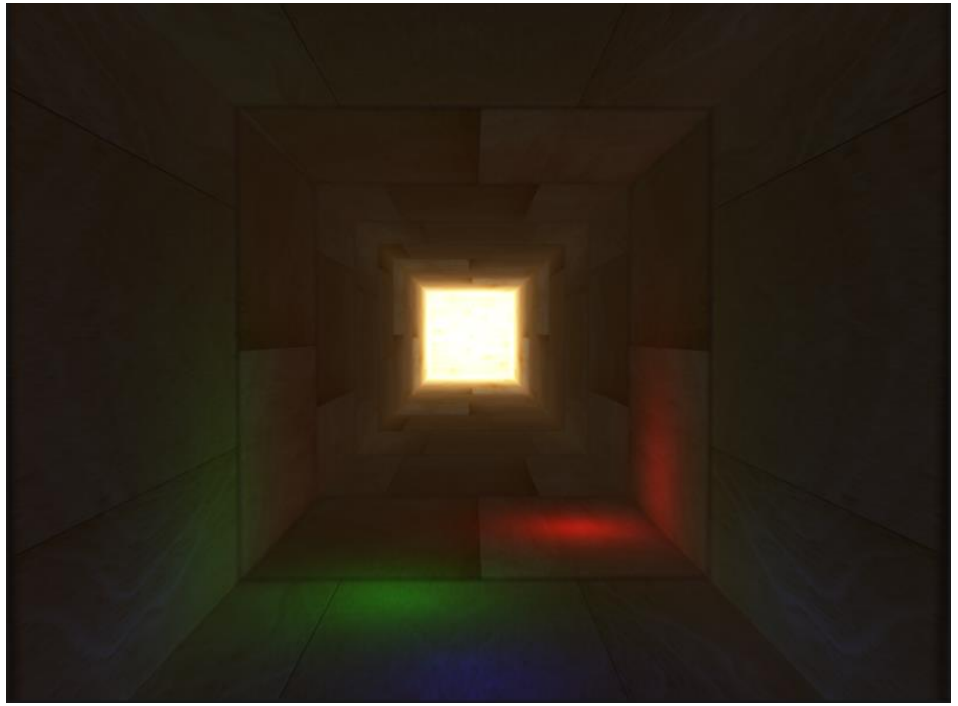
Tone Mapping

- Simple exposure tone mapping algorithm looks as follows:

```
uniform float exposure;

void main()
{
    const float gamma = 2.2;
    vec3 hdrColor = texture(hdrBuffer, 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);
}
```

F5…

- 0.2, 1.0, 4.0



Remark

- The two tone mapping algorithms are only a few of a large collection
- Some favor certain colors/intensities above others
- Some display both low and high exposure colors at the same time to create more colorful and detailed images
- Also collection of techniques known as automatic exposure adjustment or eye adaptation techniques (determine brightness of the scene in the previous frame and slowly adapt the exposure parameter → scene gets brighter in dark areas or darker in bright areas, mimicking the human) eye

Questions???