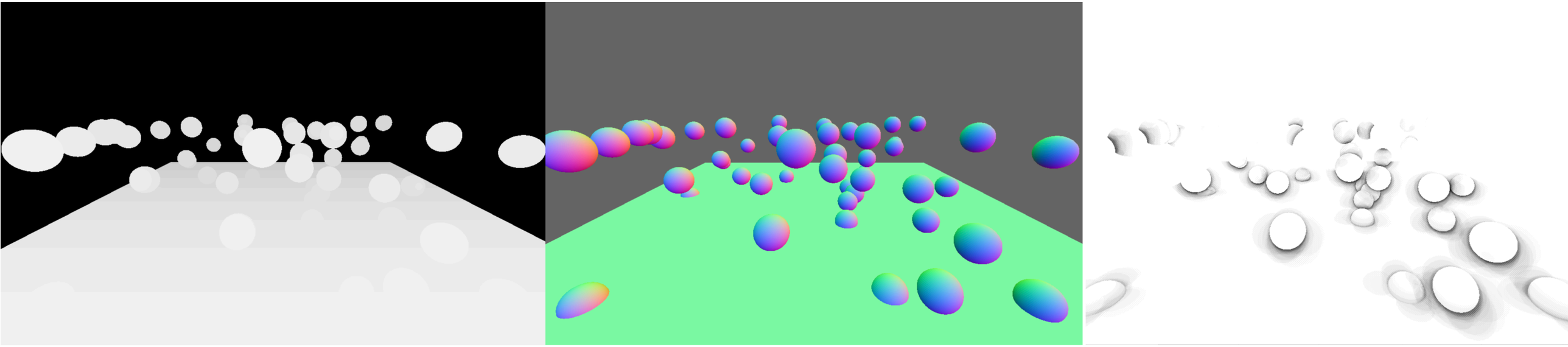
Visual Computing I:

Interactive Computer Graphics and Vision



Advanced Rendering Methods: Deferred Shading

Stefanie Zollmann and Tobias Langlotz

Deferred Shading

Forward Rendering

```
// Loop through all 'X' lights
for (int i = 0; i < actualLightCount; i++)
{
    // Calculate common light vectors
    vec3 lightDir = normalize(lights[i].position - FragPos);
    vec3 viewDir  = normalize(viewPos - FragPos);
    vec3 normal   = normalize(Normal);

    // --- Diffuse Lighting (Lambert) ---
    float diff = max(dot(normal, lightDir), 0.0);
    vec3 diffuse = diff * lights[i].color * lights[i].intensity;

    // --- Specular Lighting (Blinn-Phong) ---
    vec3 halfwayDir = normalize(lightDir + viewDir);
    float spec = pow(max(dot(normal, halfwayDir), 0.0), shininess);
    vec3 specular = spec * specularStrength *
          lights[i].color * lights[i].intensity;

    // --- Combine ---
    finalColor += (diffuse * material) + specular;
}

// Output the final, lit color
FragColor = vec4(finalColor, 1.0);
```

Forward Rendering

- So far, we did Forward Rendering
- "Classic" rendering pipeline
- Object-centric process: render one object, then the next, then the next
- For every fragment (pixel) of an object, the fragment shader:
  - Fetches material properties (diffuse colour, normals, etc.)
  - Loops through all relevant lights in the scene
  - Calculates the final colour
  - Final colour is written directly to the main framebuffer

```
// Loop through all 'X' lights
for (int i = 0; i < actualLightCount; i++)
{
    // Calculate common light vectors
    vec3 lightDir = normalize(lights[i].position - FragPos);
    vec3 viewDir  = normalize(viewPos - FragPos);
    vec3 normal   = normalize(Normal);

    // --- Diffuse Lighting (Lambert) ---
    float diff = max(dot(normal, lightDir), 0.0);
    vec3 diffuse = diff * lights[i].color * lights[i].intensity;

    // --- Specular Lighting (Blinn-Phong) ---
    vec3 halfwayDir = normalize(lightDir + viewDir);
    float spec = pow(max(dot(normal, halfwayDir), 0.0), shininess)
    vec3 specular = spec * specularStrength *
          lights[i].color * lights[i].intensity;

    // --- Combine ---
    finalColor += (diffuse * material) + specular;
}

// Output the final, lit color
FragColor = vec4(finalColor, 1.0);
```

Deferred Shading

- Bottleneck of Forward Rendering:
  - The lighting cost scales badly: O(Fragments×Lights)
- What if we want to have 1000 light sources?
- In addition: re-calculating lighting for pixels that might be hidden (overdrawn) later

```
// Loop through all 'X' lights
for (int i = 0; i < actualLightCount; i++)
{
    // Calculate common light vectors
    vec3 lightDir = normalize(lights[i].position - FragPos);
    vec3 viewDir  = normalize(viewPos - FragPos);
    vec3 normal   = normalize(Normal);

    // --- Diffuse Lighting (Lambert) ---
    float diff = max(dot(normal, lightDir), 0.0);
    vec3 diffuse = diff * lights[i].color * lights[i].intensity;

    // --- Specular Lighting (Blinn-Phong) ---
    vec3 halfwayDir = normalize(lightDir + viewDir);
    float spec = pow(max(dot(normal, halfwayDir), 0.0), shininess)
    vec3 specular = spec * specularStrength *
          lights[i].color * lights[i].intensity;

    // --- Combine ---
    finalColor += (diffuse * material) + specular;
}

// Output the final, lit color
FragColor = vec4(finalColor, 1.0);
```

Deferred Shading

- Could we decouple geometry from lighting?
- What if we first rendered all our geometry's data (Position, Normal, Colour) for the whole screen...
- ...and then ran the lighting calculations once per pixel, using that data?

    -> Idea: Deferred Shading

Deferred Shading

- "Defer" (delay) all the expensive lighting work until second rendering pass
- Changes our pipeline:
  - Pass 1 (Geometry Pass): Render all objects, but instead of a final colour, we output their data (Position, Normal, Colour) into a set of textures
  - Pass 2 (Lighting Pass): Use those textures to compute all the lighting at once -> Decoupling lighting complexity from geometry complexity

Pass 1: Geometry Pass/G-Buffer

- Geometry Pass does not render to the screen:
  - Renders to a set of textures called the G-Buffer
- Use Multiple Render Targets (MRTs) In OpenGL: glDrawBuffers
- What's in the G-Buffer?
  - Position (World-space)
  - Normals (World-space)
  - Albedo (Diffuse Colour)
  - Material Properties (e.g., Specular, Roughness, Metallic)

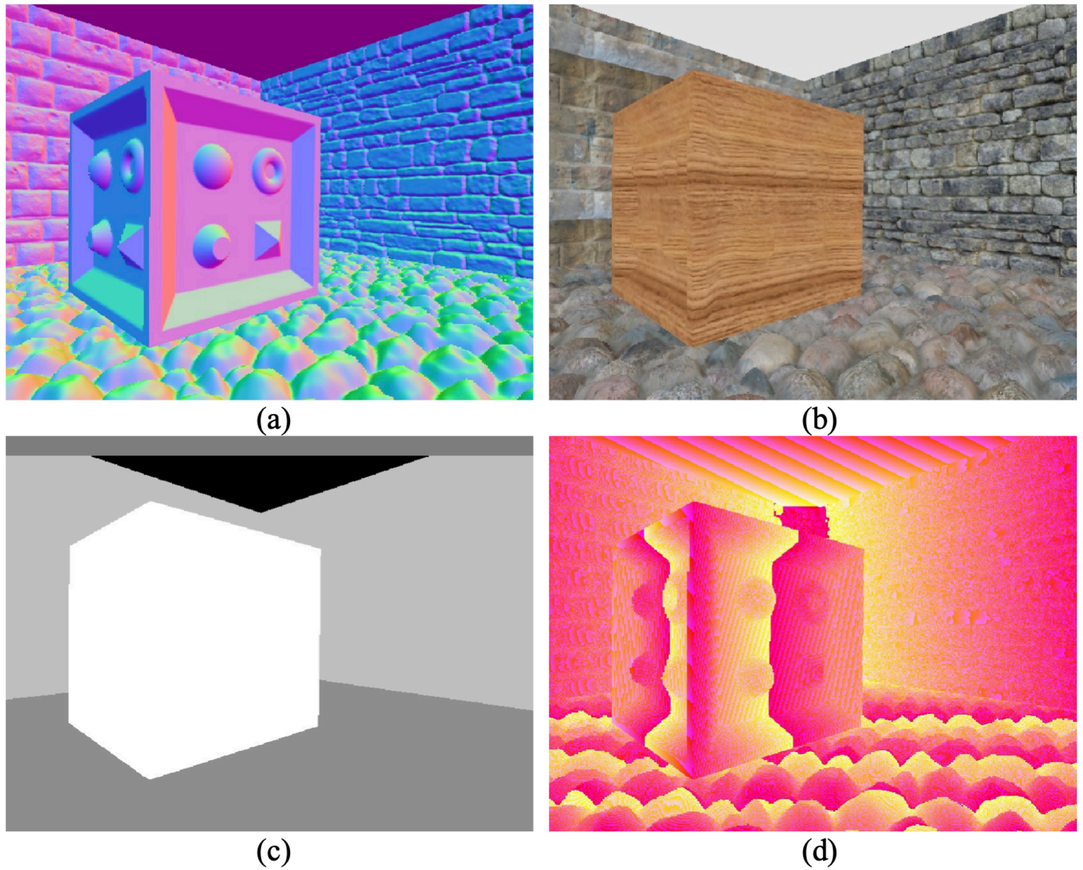


Figure 4: The four textures generated by the material pass: (a) normal, (b) diffuse, (c) specular and (d) encoded depth as color.

Policarpo and Fonseca, “Deferred Shading Tutorial”, 2005

Pass2: Lighting Pass -> Using the G-Buffer

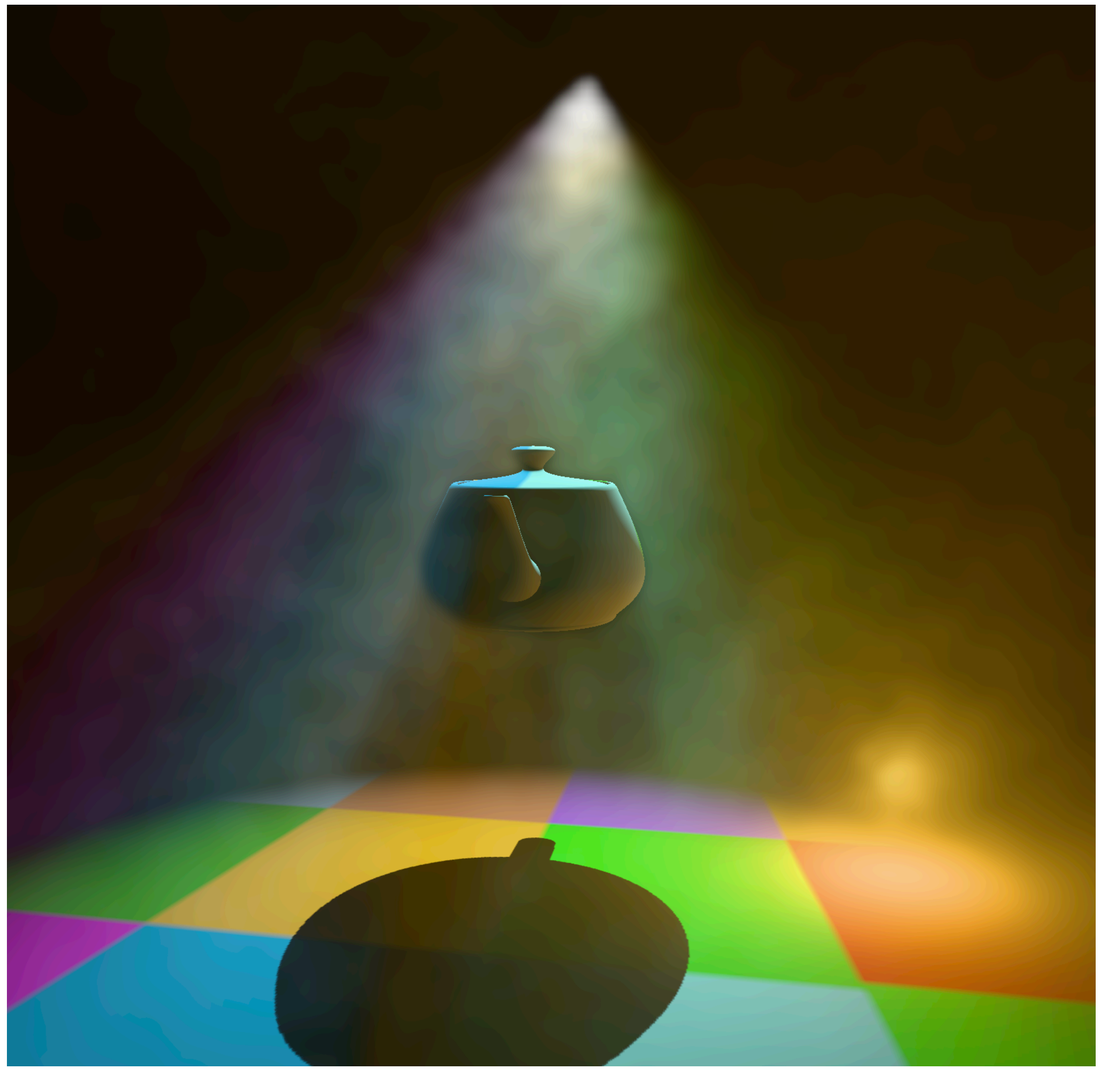
- Second pass:
  - Lighting is finally calculated
- How it works:
  - Render a single quad that covers the entire screen
  - Quad's fragment shader runs once for every pixel on the screen
  - For each pixel, the shader:
    - Gets its screen coordinate (gl_FragCoord).
    - Uses that coordinate to sample all the G-Buffer textures (Position, Normal, Albedo, etc.)
    - Reconstructs all the data for the pixel that's visible at that spot
    - Based on all the data, it performs the exact same lighting calculations as our old forward shader (looping through all the lights)

Pros/Cons

- Pros:
  - Handles Large Light Counts: Cost is no longer tied to geometry - > render hundreds or thousands of dynamic lights per frame
  - Decoupled Performance: Lighting and geometry complexity are separate -> Complex scene with many objects costs the same to light as a simple one (G-Pass is fast)
  - Shader Simplicity: Geometry pass shader is simple (just outputs data), and lighting pass shader is simple (just does lighting)
  - Free Data for Post-Processing: The G-Buffer (positions, normals, etc.) can be reused to add effects like SSAO (Screen-Space Ambient Occlusion), or screen-space reflections
- Cons
  - Transparency is a Challenge: How do you store a semi-transparent material (like glass) in the G-Buffer? Doesn't have one position or normal
  - Solution: Hybrid approach:
    - Render all opaque objects first with deferred shading
    - Then render all transparent objects on top using traditional forward rendering
  - High Memory Bandwidth: Writing and then reading 3-5 full-screen textures every single frame

Light Volumes

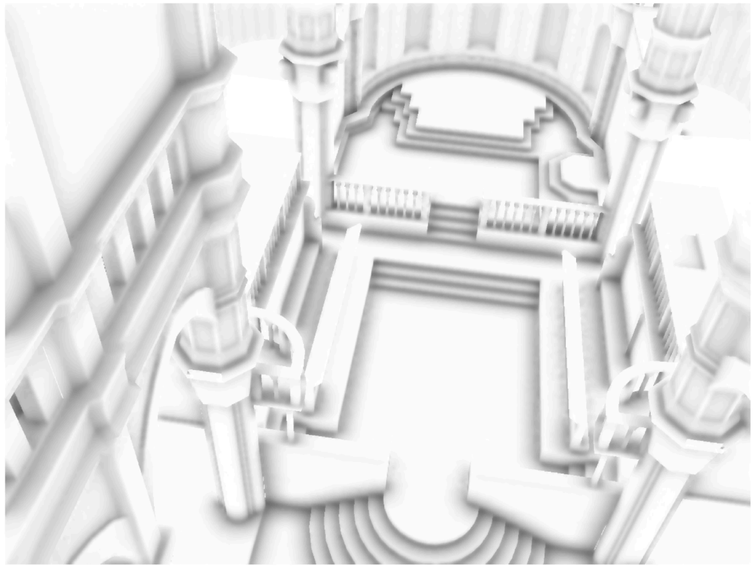
- Instead of drawing a full-screen quad for every light, we can draw the light's shape itself (e.g., a 3D sphere for a point light, a cone for a spot light).
- The lighting shader only runs for the pixels inside that shape
- A small point light in the corner of the room now only costs a few hundred pixels to calculate, not 2,000,000!
- This makes the true cost much closer to O(Pixels×Relevant_Lights).



Ambient Occlusion

Ambient Occlusion

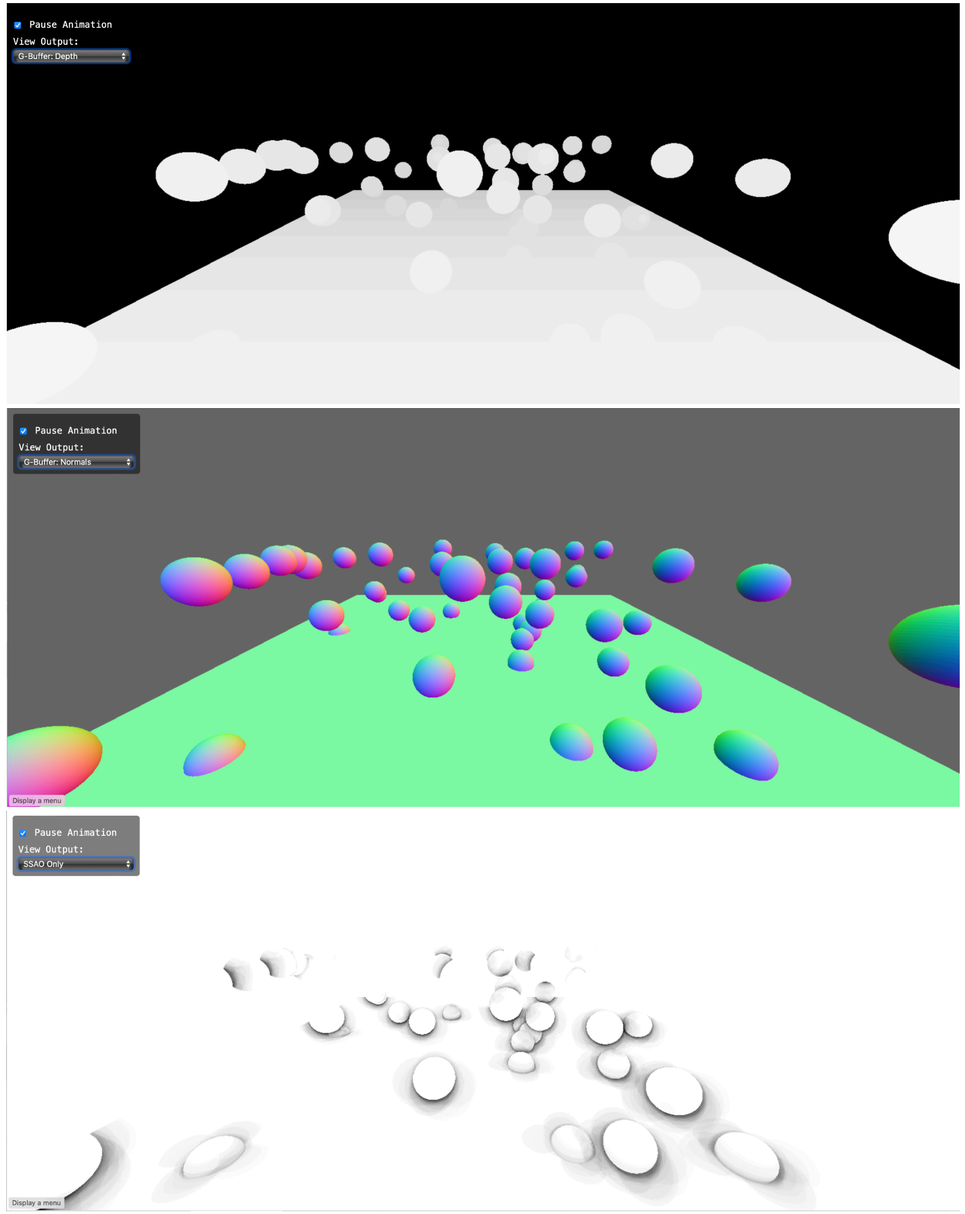
- Coarse approximation of global illumination
- Often referred to as "sky light"
- Global method: Illumination at each point is a function of other geometry in the scene
- Soft, realistic shadows that appear in corners, creases, and where objects get close to each other



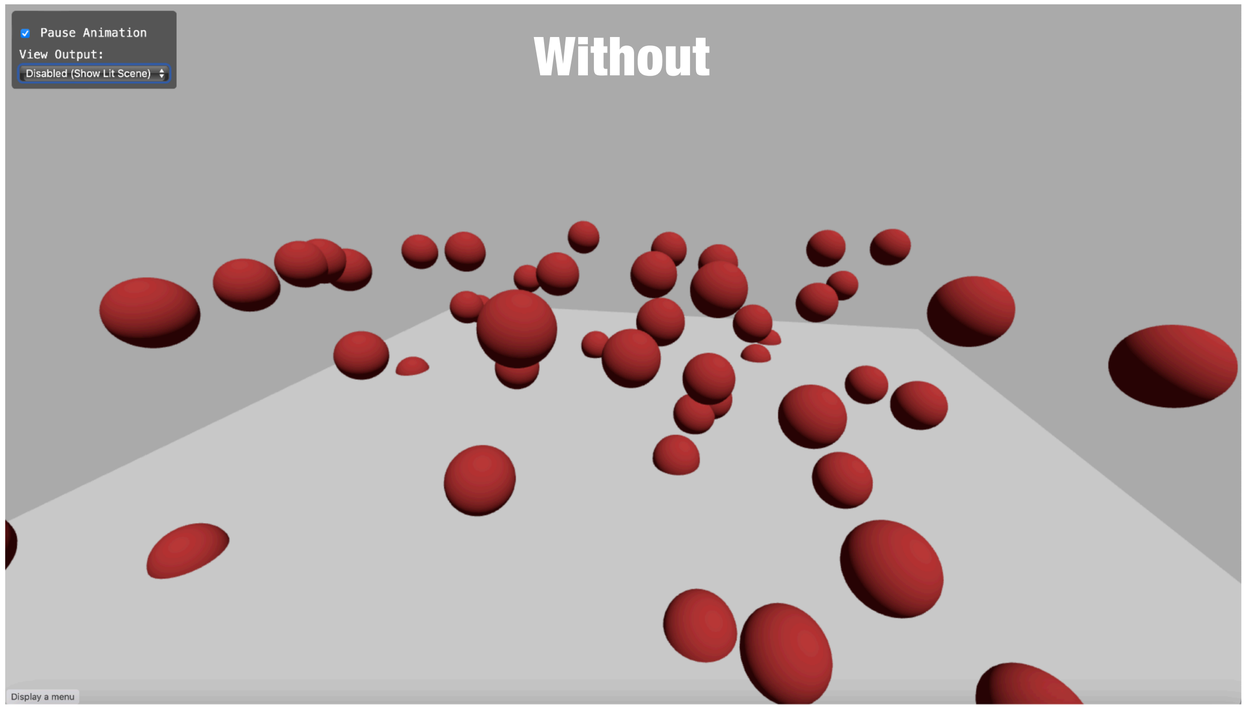
Nvidia: Screen Space Ambient Occlusion

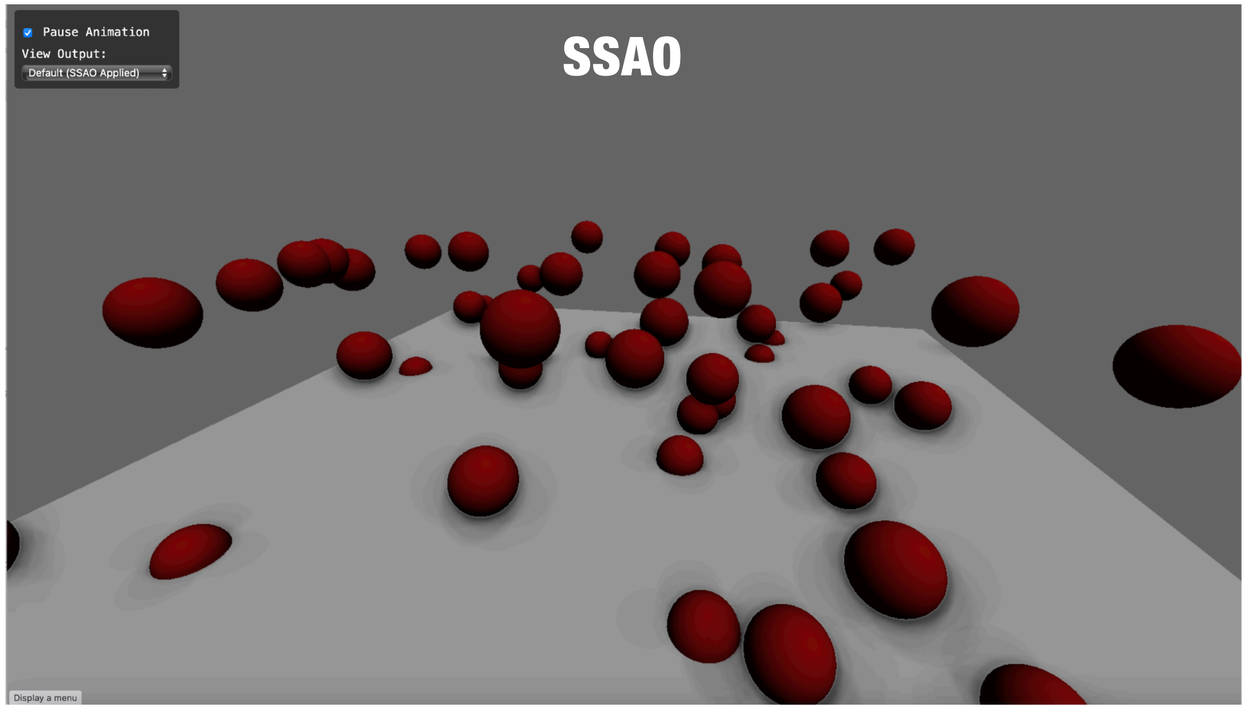
Screen-Space Ambient Occlusion (SSAO)

- Post-processing effect that approximates ambient occlusion
- Runs as another pass after the G-Pass
- Render a full-screen quad
- For every pixel, the fragment shader:
  - Reads Position and Normal from G-Buffer
  - Takes several random samples from the Position texture in small radius around pixel
  - Checks how many of those samples are behind original pixel's surface ("occluding" it).
- The more samples that are occluded, the darker that pixel becomes
- This creates a grayscale "occlusion map," which is then multiplied with final lit scene to apply the soft shadows
- This is fast: All done in screen-space using the G-Buffer data (already created before)

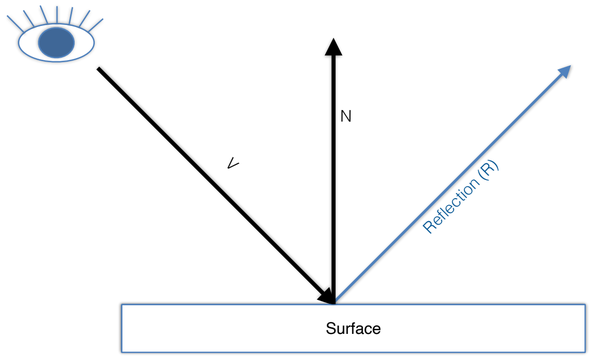


Screen-Space Ambient Occlusion (SSAO)

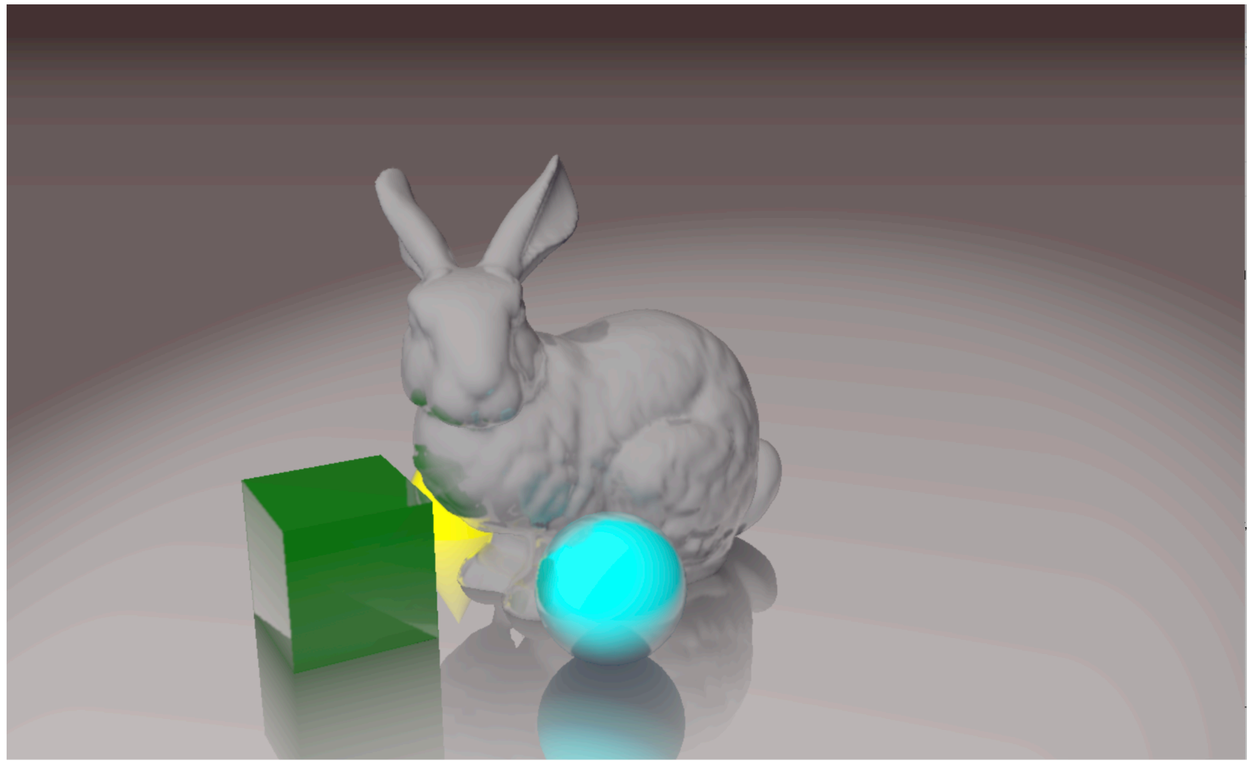




Screen-Space Reflections (SSR)

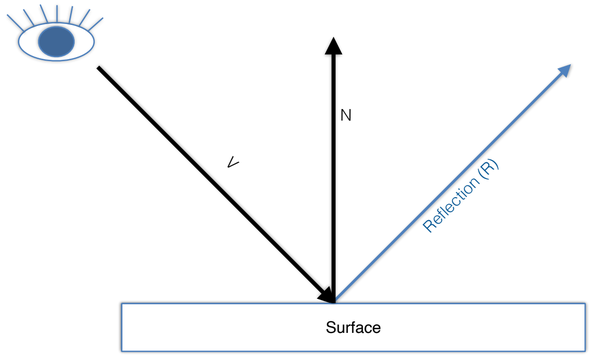


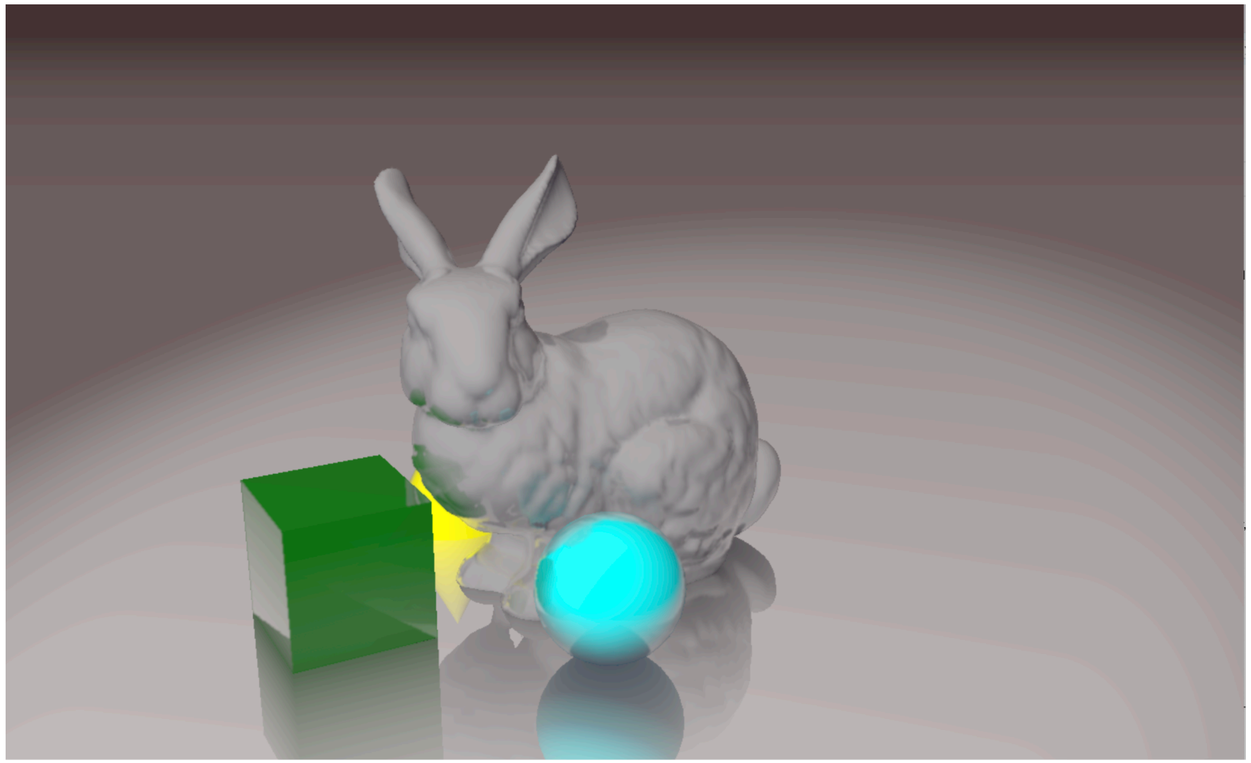
- Use the G-Buffer to find reflections
- How it Works:
  - For each pixel on a reflective surface (like a wet floor or a mirror)
    - Calculate the reflection vector (just like in Blinn-Phong)
    - Instead of sampling a cubemap-> “march" a ray from that pixel in the reflection direction in screen-space
    - Use the Position/Depth Buffer to see if that ray "hits" any other pixel on the screen
    - If it does, sample the colour of the pixel it hit and use that as the reflection
- **Advantage**: Gives dynamic, real-time reflections of other objects in the scene for a very low cost
- **Disadvantage**: It can only reflect what is already visible on the screen -> If reflected object is off-screen or hidden, the reflection disappears



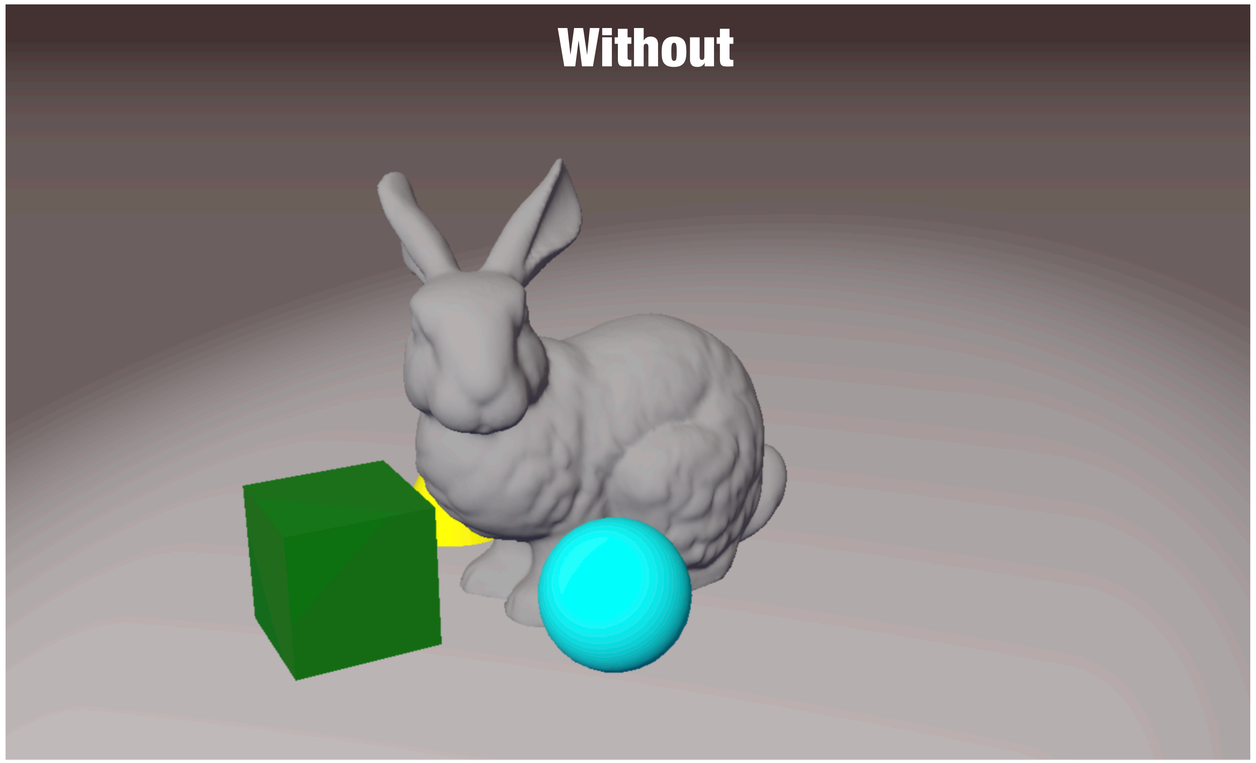
Screen-Space Reflections (SSR)

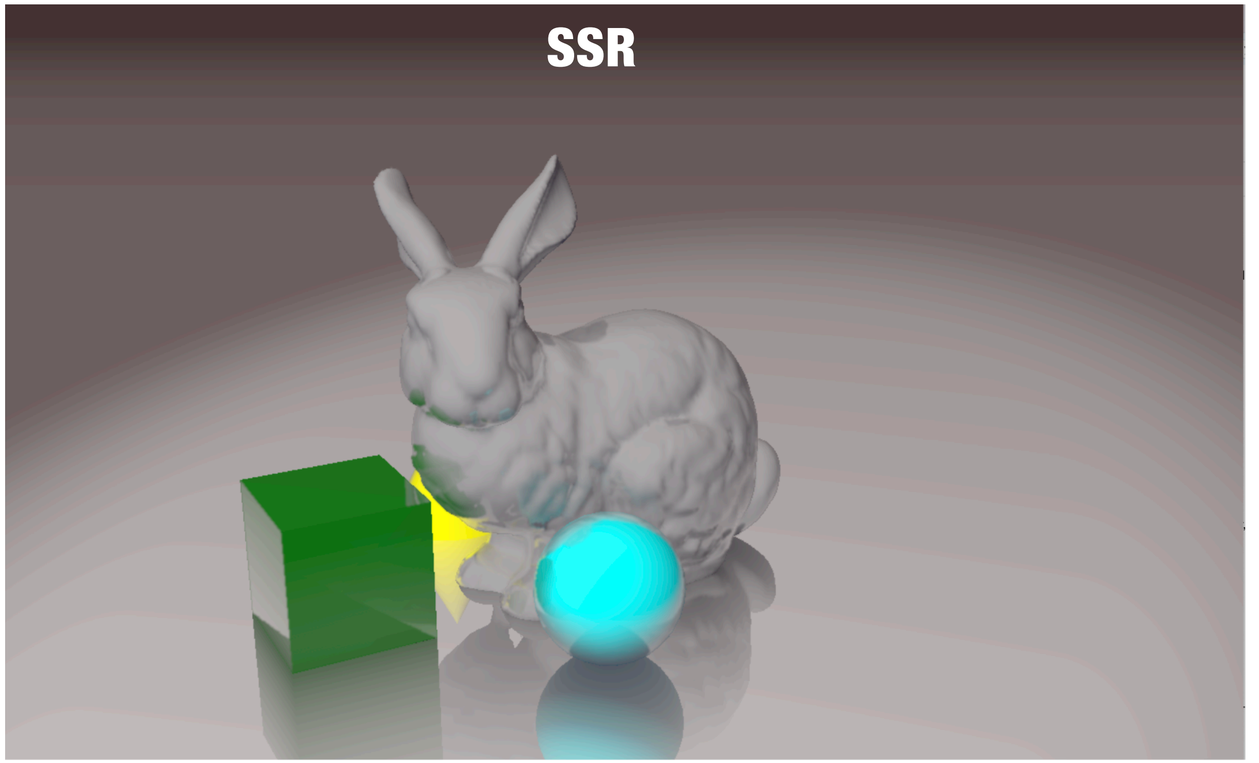
```
void main() {
  // Read G-buffer data
  float d = texture(sceneDepth, vUV).r;
  vec3 N = texture(sceneNormal, vUV).xyz;
  vec3 P = viewPos(vUV, d);

  // View and reflection directions
  vec3 V = normalize(-P);
  vec3 R = reflect(V, N);

  // Raymarch along reflection
  vec2 hitUV = vUV;
  bool hit = false;
  for (int i=0; i<40; ++i) {
      P += R * 0.1;                          // step forward
      vec4 q = proj * vec4(P,1.0);           // project to screen
      vec2 uv = q.xy/q.w*0.5 + 0.5;
      float z = texture(sceneDepth, uv).r; // compare depth
      if (abs(P.z - viewPos(uv, z).z) < 0.01) {
          hitUV = uv; hit = true; break;
      }
  }

  // Blend base color with reflection
  vec3 base = texture(sceneColor, vUV).rgb;
  vec3 refl = hit ? texture(sceneColor, hitUV).rgb : vec3(0);
  fragColor = vec4(mix(base, refl, 0.5), 1.0);
}
```





Screen-Space Reflections (SSR)





The Final Polish: HDR, Tone Mapping & Bloom

- How to create cinematic look?
- Problem (HDR):
  - Real world is "High Dynamic Range" (HDR) -> But displays can only show "Low Dynamic Range" (LDR)
- Solution (Tone Mapping):
  - Render entire scene to an HDR texture (framebuffer with a format like GL_RGB16F)
  - Then, draw a final full-screen quad -> Fragment shader for this quad runs a Tone Mapping algorithm
  - Algorithm intelligently "squashes" the HDR values back into the 0-1 range, preserving detail in both bright and dark areas (e.g., you can still see the bright sky and the dark shadows)

Standard (Low) Dynamic Range

High Dynamic Range

Mantiuk et al. High Dynamic Range Imaging

The Final Polish: HDR, Tone Mapping & Bloom

- How to create cinematic look?
- Problem (HDR):
  - Real world is "High Dynamic Range" (HDR) -> But displays can only show "Low Dynamic Range" (LDR)
- Solution (Tone Mapping):
  - Render entire scene to an HDR texture (framebuffer with a format like GL_RGB16F)
  - Then, draw a final full-screen quad -> Fragment shader for this quad runs a Tone Mapping algorithm
  - Algorithm intelligently "squashes" the HDR values back into the 0-1 range, preserving detail in both bright and dark areas (e.g., you can still see the bright sky and the dark shadows)

Tone Mapping (Reinhard)

$$L_{\text{out}} = \frac{L_{\text{in}}}{L_{\text{in}} + 1}$$

When L_in is small, output ≈ input (dark areas stay dark),
When L_in is large, the result levels off near 1 (bright areas stop growing).
-> Everything fits nicely into 0–1

The Final Polish: HDR, Tone Mapping & Bloom

- How to create cinematic look?
- Problem (HDR):
  - Real world is "High Dynamic Range" (HDR) -> But displays can only show "Low Dynamic Range" (LDR)
- Solution (Tone Mapping):
  - Render entire scene to an HDR texture (framebuffer with a format like GL_RGB16F)
  - Then, draw a final full-screen quad -> Fragment shader for this quad runs a Tone Mapping algorithm
  - Algorithm intelligently "squashes" the HDR values back into the 0-1 range, preserving detail in both bright and dark areas (e.g., you can still see the bright sky and the dark shadows)

Tone Mapping (Reinhard)

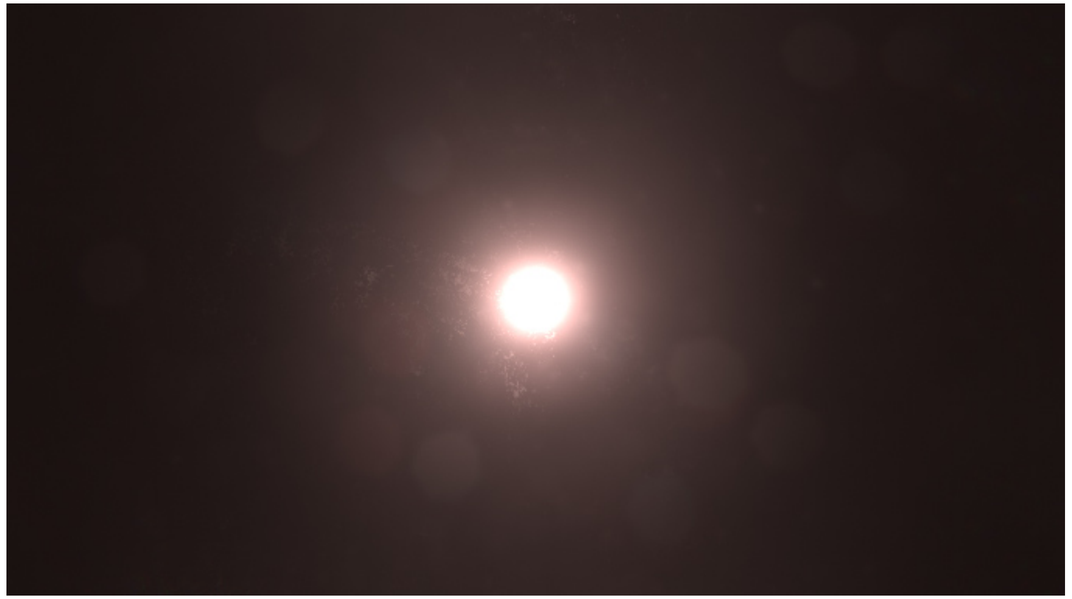
$$L_{\text{out}} = \frac{L_{\text{in}}}{L_{\text{in}} + 1}$$

When L_in is small, output ≈ input (dark areas stay dark),
When L_in is large, the result levels off near 1 (bright areas stop growing).
-> Everything fits nicely into 0–1

```
  vec3 tonemap(vec3 x) { return x / (x + 1.0); }

  void main() {
    vec3 hdr = texture(hdrColor, vUV).rgb;
    vec3 mapped = tonemap(hdr);   // compress high values
    fragColor = vec4(mapped, 1.0); // output
}
```

The Final Polish: HDR, Tone Mapping & Bloom

- "Bonus" (Bloom):
  - While we have that HDR texture, we can create Bloom (a realistic glow)
  - Extract only the super-bright parts of the image (colors > 1.0)
  - Put them in a new texture, and blur it
    - Add this blurred "glow" texture back to the final image after tone mapping.
  - Makes all lights and bright spots feel intense and realistic



https://docs.unity3d.com/Packages/com.unity.postprocessing@3.2/manual/Bloom.html

```
uniform sampler2D toneMappedScene;
uniform sampler2D bloomBlur;

void main() {
    vec3 scene = texture(toneMappedScene, vUV).rgb;
    vec3 bloom = texture(bloomBlur, vUV).rgb;
    fragColor = vec4(scene + bloom, 1.0); // add glow back in
}
```

The end!