Shader: Custom/ToggleShader

Goal:

This shader lets us switch (“toggle”) between different types of lighting—Diffuse **(Lambert)**, **Ambient**, **Specular**, **Normal/Bump mapping**, and **Rim lighting**—using checkboxes in Unity. It’s designed for our **Part 2 – Illumination** section so we can visually show what each light type does on our models.

Properties Block—Material Settings

Everything inside Properties { } appears in Unity’s Inspector when we make a material using this shader.

1. Basic Surface Controls

```
// Base color tint for the surface
_BaseColor ("Base Color", Color) = (1, 1, 1, 1)

// Base texture (albedo)
_MainTex ("Base Texture", 2D) = "white" {}
```

- _BaseColor is a solid color we can tint our object with.
- _MainTex is the main **texture (albedo)** that defines how the surface looks (like metal, plastic, etc.).
  We multiply them together later, so changing the base color slightly brightens, darkens, or tints the texture.

2. Specular Reflection

```
// Specular reflection color
_SpecColor ("Specular Color", Color) = (1, 1, 1, 1)

// Controls highlight sharpness
_Shininess ("Shininess", Range(0.1, 100)) = 16
```

- _SpecColor controls what color our highlight will be (white = clean light, yellow = warm, etc.).

- _Shininess controls **how tight or soft** the highlight is.
  Higher value → smaller and sharper highlight (like polished metal).
  Lower value → bigger, blurry highlight (like plastic).

3. Normal/Bump Mapping

```
// Normal or bump map for surface detail
_myBump ("Bump Texture", 2D) = "bump" {}
_mySlider ("Bump Amount", Range(0,10)) = 1
```

These control our **normal map** (bump texture).

- _myBump stores the texture where the RGB colors encode fake depth information.
- _mySlider lets us scale the strength of that bump (how deep the dents look).
  It doesn't add real geometry—it just bends how light interacts with the surface, making it *look* 3D.

4. Rim Lighting

```
// Controls rim light falloff - higher = sharper edge
_RimPower ("Rim Power", Range(0.5, 8.0)) = 3.0

// Controls overall rim light brightness
_RimIntensity ("Rim Intensity", Range(0.0, 10.0)) = 0
```

These control how strong the glowing outline is around our object's edges.

- _RimPower decides how **soft or sharp** that edge glow is.
- _RimIntensity multiplies the brightness of that glow.

5. Toggles

```
// Toggles to enable or disable lighting components
[Toggle] _UseDiffuse ("Enable Diffuse", Float) = 1
[Toggle] _UseAmbient ("Enable Ambient", Float) = 1
[Toggle] _UseSpecular ("Enable Specular", Float) = 1
[Toggle] _UseBhong ("Enable Bump", Float) = 1
[Toggle] _UseRim ("Enable Rim", Float) = 1
```

Each toggle acts like a checkbox in the Inspector:

- We can switch each lighting effect on or off without touching the code.
  That's how we demonstrate "only ambient," "only specular," etc., which the **Part 2 rubric** asks for.

Attributes and Varyings

```
// Vertex Input (from the mesh)
struct Attributes
{
    float4 positionOS : POSITION; // Object-space vertex position
    float3 normalOS : NORMAL;     // Object-space surface normal
    float2 uv : TEXCOORD0;        // UV coordinates for textures
    float4 tangentOS : TANGENT;   // Tangent for normal mapping
};
```

These are per-vertex data from our 3D model in **Object space—before** it's transformed into the world.

- positionOS: the vertex position.
- normalOS: the vertex normal direction.
- uv: texture coordinates.
- tangentOS: used for bump mapping—tells how the texture is oriented on the surface.

Vertex → Fragment transfer:

```
// Data passed to the Fragment Shader
struct Varyings
{
    float4 positionHCS : SV_POSITION; // Homogeneous clip-space position
    float3 normalWS : TEXCOORD1;      // Normal in world space
    float3 tangentWS : TEXCOORD2;     // Tangent in world space
    float2 uv : TEXCOORD0;            // UV coordinates passed through
    float3 bitangentWS : TEXCOORD3;   // Bitangent, used to form TBN matrix
    float3 viewDirWS : TEXCOORD4;     // Direction from fragment to camera
};
```

These are what our vertex shader sends to the fragment shader.

- positionHCS: final screen position (so GPU knows where to draw the pixel).
- normalWS, tangentWS, bitangentWS: all converted to **World Space**.
- uv: passed through for texture lookup.

- viewDirWS: direction from the pixel to the camera, used for specular and rim.

Vertex Shader — vert

```hlsl
//  Vertex Shader
Varyings vert (Attributes IN)
{
    Varyings OUT;

    // Convert vertex position to clip space for rasterization
    OUT.positionHCS = TransformObjectToHClip(IN.positionOS.xyz);

    // Convert normal and tangent to world space and normalize
    OUT.normalWS = normalize(TransformObjectToWorldNormal(IN.normalOS));
    OUT.tangentWS = normalize(TransformObjectToWorldNormal(IN.tangentOS.xyz));

    // Bitangent is calculated using cross product and tangent.w sign
    OUT.bitangentWS = cross(OUT.normalWS, OUT.tangentWS) * IN.tangentOS.w;

    // Calculate direction from vertex to camera
    float3 worldPosWS = TransformObjectToWorld(IN.positionOS.xyz);
    OUT.viewDirWS = normalize(GetCameraPositionWS() - worldPosWS);

    // Pass UVs for texture sampling
    OUT.uv = IN.uv;
    return OUT;
}
```

- Converts our model's vertex position from **Object Space** → **Clip Space**, so it appears on screen.
- Converts our normals and tangents into **World Space**, since all lighting happens there.
- Uses the **cross product** to build the **bitangent**, forming the complete **TBN matrix**.
  That TBN is later used to transform our normal map data correctly.
- Converts the vertex position into world coordinates.
- Finds the vector from this point **to the camera**, which is needed for calculating reflections and rim lighting.

So at this stage, every vertex now knows:

- Which way it faces (normalWS)
- Where the camera is
- How to sample the texture later
  Then Unity interpolates all that between vertices, so every pixel in between also gets its own values.

Fragment Shader — frag

This runs for every pixel drawn on the screen.

Step 1: Base color

```
// --- Step 1: Sample the base texture and apply color tint ---
half4 texColor = SAMPLE_TEXTURE2D(_MainTex, sampler_MainTex, IN.uv);
half3 base = texColor.rgb * _BaseColor.rgb; // combines texture + color
```

- Reads the color from our texture and multiplies it with our tint color.
  So if we want the same texture in red or blue, we just change _BaseColor.
  This is our **starting color** before lighting is added.

Step 2: Choose which normal to use

```
half3 normalWS;
// If Phong/Bump is enabled, use normal map
if (_UseBhong > 0.5)
{
    // Apply tiling and offset to UVs
    float2 bumpUV = IN.uv * _myBump_ST.xy + _myBump_ST.zw;

    // Unpack normal from bump texture (stored as RGB)
    half3 normalTS = UnpackNormal(SAMPLE_TEXTURE2D(_myBump, sampler_myBump, bumpUV));

    // Scale bump intensity using slider
    normalTS.xy *= _mySlider;

    // Build TBN matrix to convert tangent-space normal to world space
    half3x3 TBN = half3x3(IN.tangentWS, IN.bitangentWS, IN.normalWS);
    normalWS = normalize(mul(normalTS, TBN)); // transform to world space
}
else
{
    // If bump not used, just use the vertex normal
    normalWS = normalize(IN.normalWS);
}
```

When the checkbox for “Enable bhong” (_UseBhong) is ON:

- It takes our **bump texture**, applies Unity’s tiling and offset, and then samples it.
- UnpackNormal() converts that color into a **normal vector** (since normal maps store direction data in RGB).
- It multiplies the X and Y values by our _mySlider to control bump intensity.
- Then it builds a small **3×3 TBN matrix** (Tangent, Bitangent, Normal) and uses it to rotate our bump normal from **tangent space to world space**.

When it’s OFF, the shader just uses the mesh’s actual normal (flat surface).
This step makes our materials feel detailed without extra geometry.

Step 3: Get the scene's main light

```
Light mainLight = GetMainLight(); // Gets URP's main directional light
half3 lightDir = normalize(mainLight.direction);
```

- Unity provides the **main light** in our scene (the directional light, like the sun or a headlamp).
- .direction gives the vector pointing **from the light toward the surface**.
- normalize() makes sure that vector has a length of 1.
  This is important because lighting math depends on direction, not distance — longer vectors would make the dot product too bright or dark.
  we use this direction to figure out *how much light actually hits our surface.*

Step 4: Compute N·L (Lambert's Diffuse)

```
// N·L term for Lambert's cosine law; clamp negative values
half NdotL = saturate(dot(normalWS, lightDir));
```

- The **dot product** between our surface normal and the light direction measures how directly the surface faces the light.
    - If it faces straight at the light → dot ≈ 1 → fully lit.
    - If it's sideways → dot ≈ 0 → dim edge.
    - If it faces away → dot < 0 → no light.
- saturate() clamps that value to between 0 and 1 so it doesn't go negative — we don't want backfaces glowing.

This is what creates realistic light falloff across a curved surface — one side bright, one side dark.
That's the **Lambert Diffuse** term.

Step 5: Calculate each lighting component

```
// Diffuse: light scattered evenly depending on N·L
half3 diffuse = base * NdotL;
```

- Takes our base surface color and multiplies it by how much light hits the surface.
- This gives the classic smooth gradient lighting — bright where the light hits directly, dark where it's angled.

Only Diffuse On

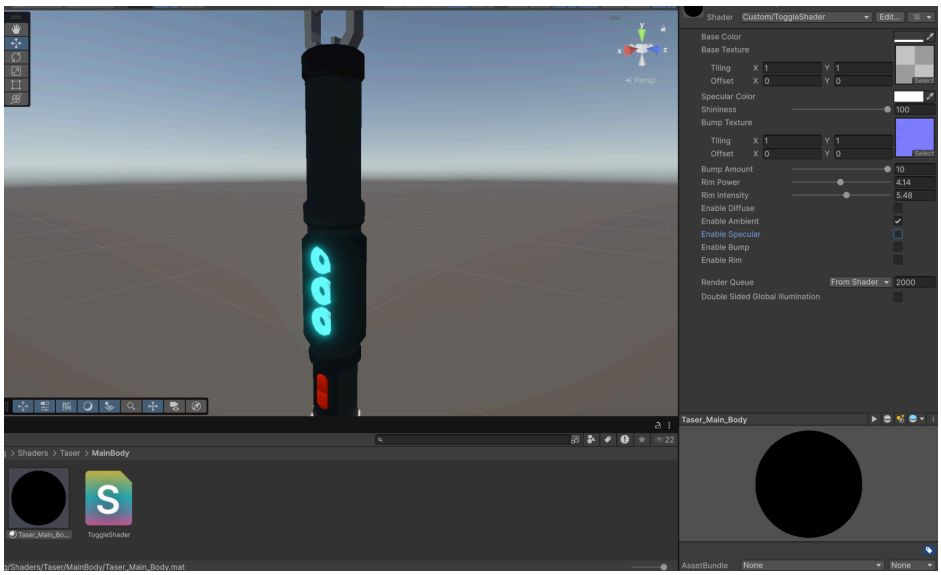
Ambient

```
// Ambient: environment light using Spherical Harmonics (approx global light)
half3 ambient = SampleSH(normalWS) * base;
```

- SampleSH(normalWS) uses **Spherical Harmonics** (Unity's built-in ambient light probe). It estimates the soft light that bounces around the scene.

- Multiplying by base tints that ambient light with our texture color.
  This ensures even areas facing away from the light aren't pitch black — they still get soft global illumination.

Only Ambient On



Specular

```
// Specular: reflection intensity depending on viewing angle
half3 reflectDir = reflect(-lightDir, normalWS); // reflection vector
half3 viewDir = normalize(IN.viewDirWS);
half specFactor = pow(saturate(dot(reflectDir, viewDir)), _Shininess);
half3 specular = _SpecColor.rgb * specFactor;
```

- reflect() calculates the mirror reflection direction from the light.
- viewDir is the direction toward the camera.
- The dot product between those tells how close our camera is to that reflection.
  - When aligned perfectly → bright spot.
  - When off-angle → fades out.
- Raising that to _Shininess makes the highlight sharper (smaller bright area).
- Multiplying by _SpecColor tints that highlight.
  This creates the shiny spot we see on metals, glass, or polished plastic.

Only specular on

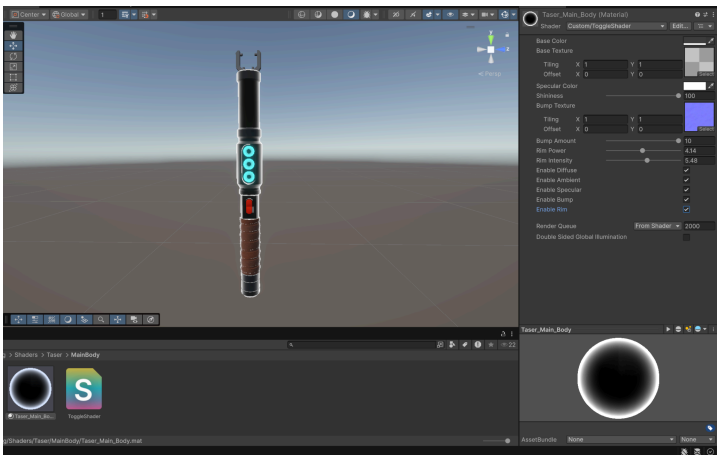
Rim Lighting

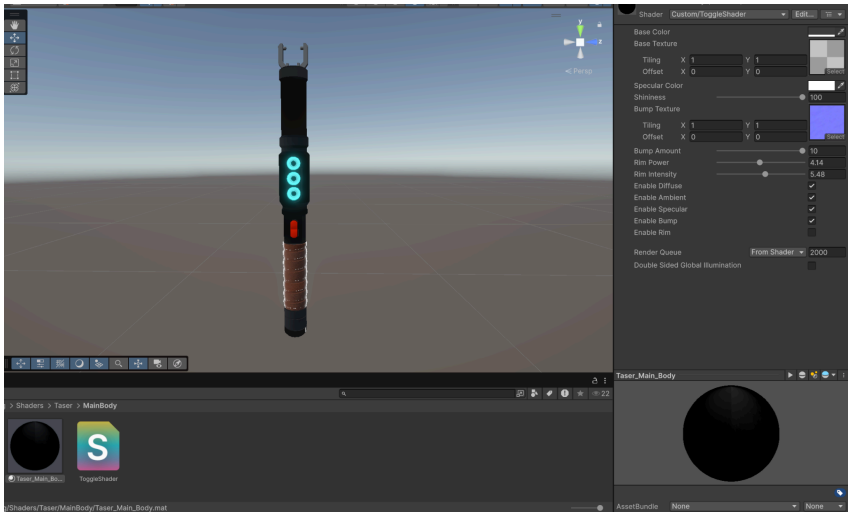
```
// Rim lighting: highlights edges facing away from light/view
half rimFactor = 1.0 - saturate(dot(viewDir, normalize(IN.normalWS)));
half rimLighting = pow(rimFactor, _RimPower);
half3 rimColor = _SpecColor.rgb * rimLighting * _RimIntensity;
```

- Rim lighting makes edges glow where the surface faces away from the camera.
- The dot product here checks how perpendicular the view is to the surface.
    - When facing straight → dot ≈ 1 → little or no rim.
    - When at a steep angle → dot ≈ 0 → strong rim.

- Subtracting from 1 inverts it so edges glow.
- pow(rimFactor, _RimPower) controls softness; _RimIntensity controls brightness.
  we use _SpecColor to color that rim—matching the material's highlight color.

This helps objects stand out from dark surroundings (for example, items on the floor).
Rim effect On vs Off





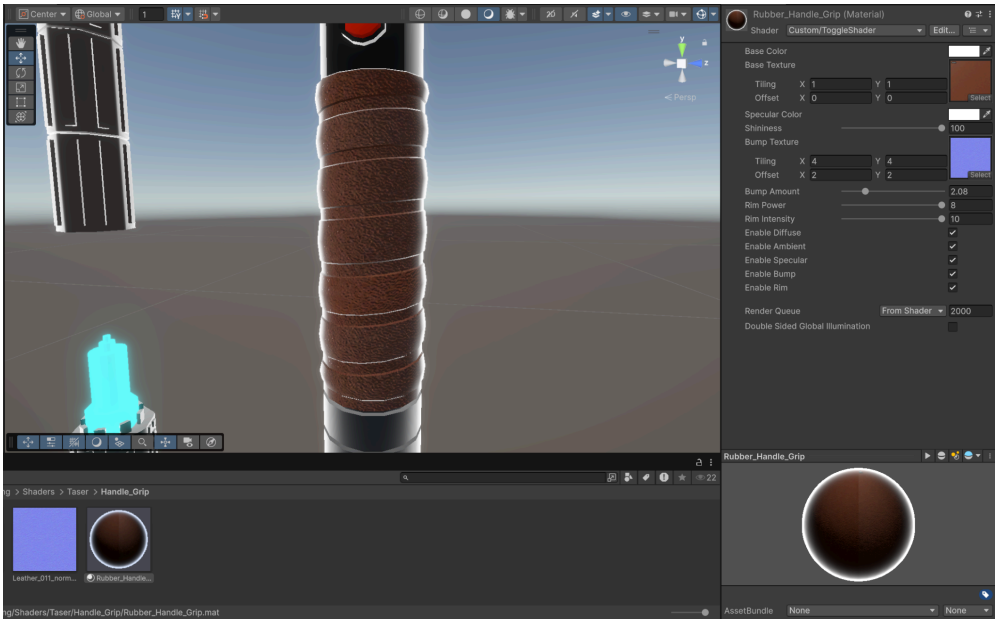
Step 6: Add them up depending on toggles

```
// Add each component if its toggle is enabled
if (_UseDiffuse > 0.5) finalColor += diffuse;    // Adds diffuse if enabled
if (_UseAmbient > 0.5) finalColor += ambient;    // Adds ambient light
if (_UseSpecular > 0.5) finalColor += specular; // Adds specular highlight
if (_UseRim > 0.5) finalColor += rimColor;       // Adds rim effect
```

Each toggle simply checks if it's turned on (>0.5) and adds that part to the final color.

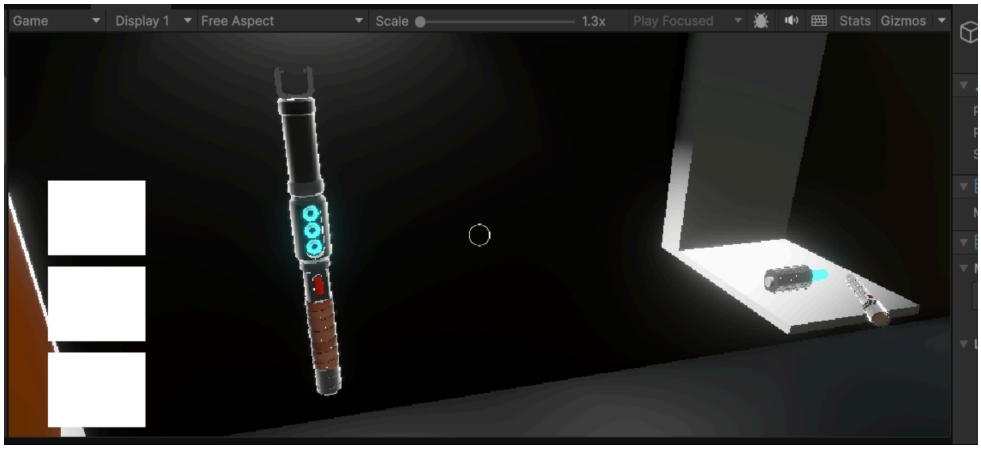
- This means we can see each lighting component separately or combined.
- The alpha (transparency) is set to 1.0 since this shader is opaque.

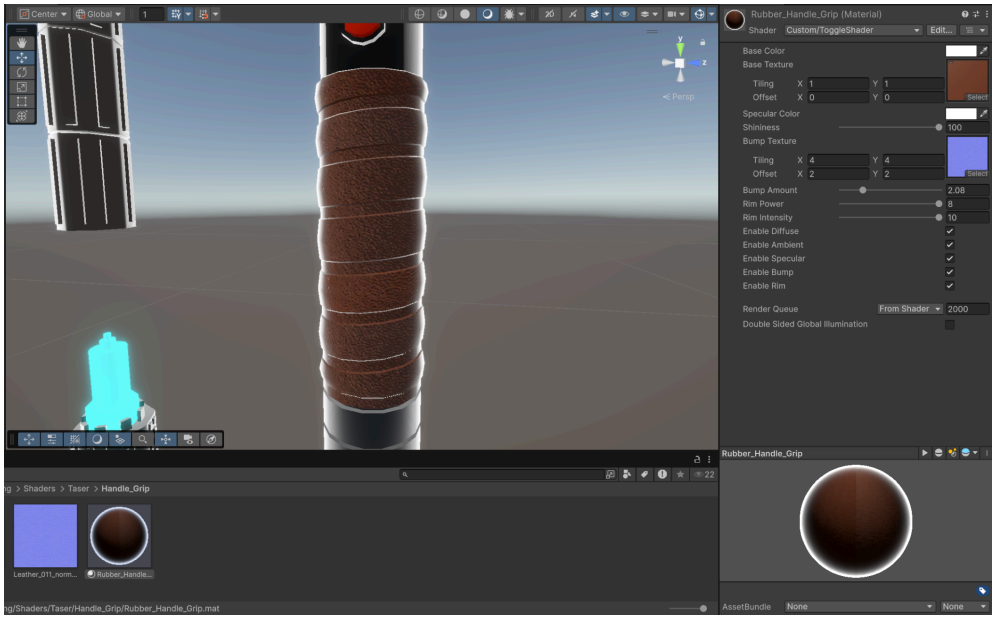
Everything On



Why we’re using this shader

- It meets the **Illumination Part 2** goal perfectly by letting us **demonstrate each type of light individually**.
- It’s a clean, all-in-one shader that works in URP and can be reused for many materials.
- The toggles make it easy to test and explain — no multiple shaders needed.
- Adding rim lighting helps our models pop out in our **spaceship environment**, especially in low-light areas.
- The bump map toggle makes parts like the **taser grip or flashlight grip** look realistic with fake depth.





Part 3 — Shaders and Effects

in this part our group implemented **three main shaders**, each serving a different purpose in the scene:

1. **rim lighting shader** — to make small, dark objects visible and reactive.
2. **ambient + specular + lambert shader (no toggle)** — for fine-detail close-ups such as buttons.
3. **world reflection shader** — to simulate reflective, mirror-like surfaces.

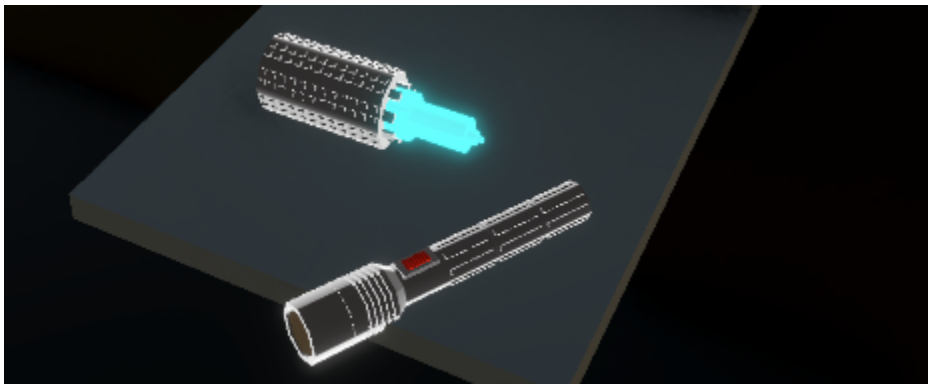
these satisfy the *"groups of three → four shader implementations"* rule from the assignment. they all build on the illumination work from part 2, but each focuses on a special surface effect our spaceship environment needs.

Rim Lighting Shader

Why we used it

small pickup items (taser batteries, small tools, flashlight heads) sit in dark areas of the ship. normal lambert + ambient lighting made them blend into the floor. rim lighting adds a soft glow around their edges, so players instantly notice interactive objects even when global light is low.

We also extended it with an **intensity control** to create a **glow that can change over time**, for example showing a taser battery running low.

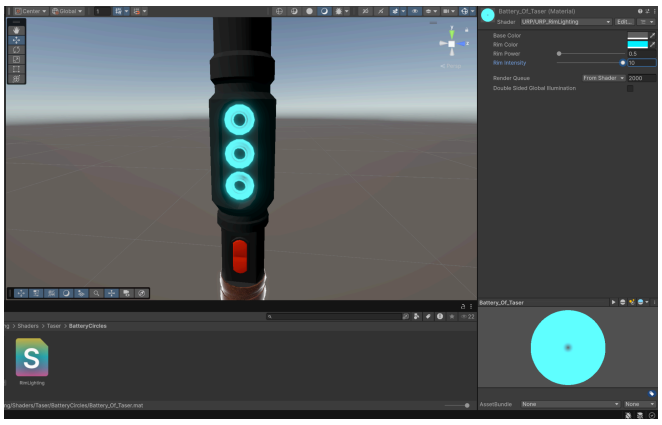


How the code works

```
// Fragment Shader
half4 frag (Varyings IN) : SV_Target
{
    // --- Normalize inputs ---
    half3 normalWS = normalize(IN.normalWS);
    half3 viewDirWS = normalize(IN.viewDirWS);

    // --- Rim lighting ---
    half rimFactor = 1.0 - saturate(dot(viewDirWS, normalWS));
    half rimLighting = pow(rimFactor, _RimPower);

    // --- Combine rim with base ---
    half3 finalColor = _BaseColor.rgb + (_RimColor.rgb * rimLighting) * _RimIntensity;

    return half4(finalColor, _BaseColor.a);
}

ENDHLSL
```

- dot(viewDirWS, normalWS)
  gives 1 when the surface faces us directly and 0 when it's at 90 degrees.
- subtracting from 1 flips it: now edges (perpendicular to the camera) give higher values.
- pow(rimFactor, _RimPower) sharpens or softens that falloff.
- _RimIntensity is the new variable we added—multiplying by it boosts brightness, effectively turning a faint halo into a **visible glow**.

in-game, we can animate _RimIntensity:



Why it matters

- guides the player's eye to pickups.
- visually indicates **object state** (battery charge).
- inexpensive: computed per-pixel, no extra geometry or post-processing.

---

2) Ambient + Specular + Lambert Shader (no toggle)

Why we used it

this version removes all toggles so the lighting is **always combined**.
it's ideal for **small components close to the camera**, like flashlight buttons or taser switches, where we want smooth shading and a visible highlight without manual enabling.

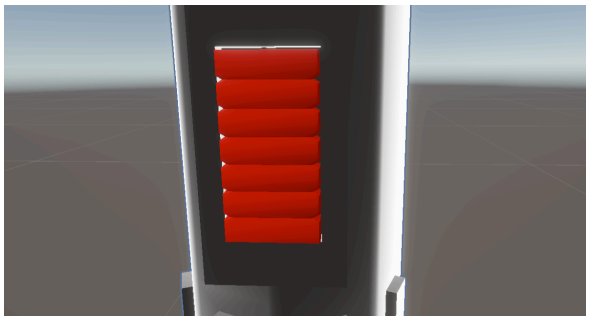
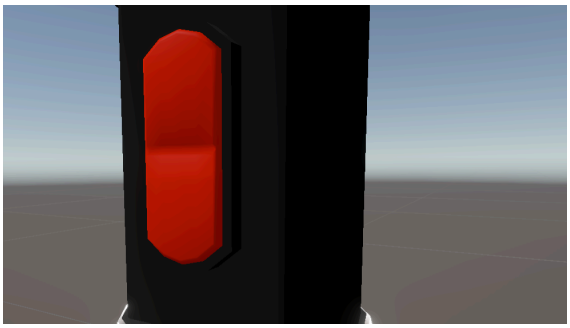
What happens inside

```
// Fragment Shader
half4 frag (Varyings IN) : SV_Target
{
    // --- Sample texture and prepare lighting vectors ---
    half4 texColor = SAMPLE_TEXTURE2D(_MainTex, sampler_MainTex, IN.uv);
    Light mainLight = GetMainLight();
    half3 lightDir = normalize(mainLight.direction);
    half3 normalWS = normalize(IN.normalWS);

    // --- Diffuse (Lambert) ---
    half NdotL = saturate(dot(normalWS, lightDir));
    half3 diffuse = texColor.rgb * _BaseColor.rgb * NdotL;

    // --- Ambient (soft indirect light) ---
    half3 ambient = SampleSH(normalWS) * texColor.rgb * _BaseColor.rgb;

    // --- Specular (Phong reflection) ---
    half3 reflectDir = reflect(-lightDir, normalWS);
    half3 viewDir = normalize(IN.viewDirWS);
    half specFactor = pow(saturate(dot(reflectDir, viewDir)), _Shininess);
    half3 specular = _SpecColor.rgb * specFactor;

    // --- Combine final result ---
    half3 finalColor = diffuse + ambient + specular;
    return half4(finalColor, 1.0);
}

ENDHLSL
```

- **lambert diffuse** handles direct light from the main source.
- **ambient (SampleSH)**—adds soft bounce light from the environment.
- **specular** – produces the shiny highlight where the light reflects toward the camera.
  Combining them produces realistic lighting even for small objects.

Why it matters

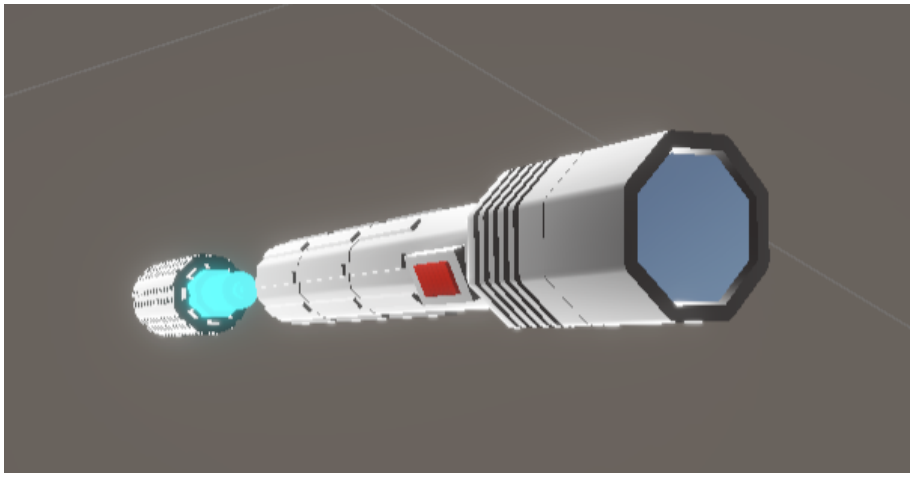
- small parts with curves or bevels catch the highlight correctly.
- when viewed close-up, these reflections make the model look physically believable.
- simpler to manage than the toggle shader for fixed objects.

3) World Reflection Shader

Why we used it

certain surfaces in the level—mirrors on walls and the **front glass of the flashlight**—need to look reflective, as if showing the environment around them. regular diffuse/specular lighting doesn't create real reflections. this shader samples the **scene's reflection probe or cubemap** to fake that mirror look.



How the code works conceptually

```
// Fragment Shader
half4 frag (Varyings IN) : SV_Target
{
    // --- Sample base texture ---
    half3 baseCol = SAMPLE_TEXTURE2D(_BaseMap, sampler_BaseMap, IN.uv).rgb * _BaseColor.rgb;

    // --- Compute view direction ---
    float3 V = SafeNormalize(GetWorldSpaceViewDir(IN.positionWS));

    // --- Surface normal ---
    float3 N = SafeNormalize(IN.normalWS);

    // --- Reflection direction ---
    float3 R = reflect(-V, N);

    // --- Environment reflection sample ---
    half3 envCol = SAMPLE_TEXTURECUBE(_EnvCube, sampler_EnvCube, R).rgb * _EnvIntensity;

    // --- Fresnel effect for edge highlights ---
    float ndotv = saturate(dot(N, V));
    float fresnel = pow(1.0 - ndotv, _FresnelPow) * _FresnelBoost;
    envCol *= (1.0 + fresnel);

    // --- Blend between texture and reflection ---
    half3 finalCol = lerp(baseCol, envCol, _EnvBlend);

    return half4(finalCol, 1.0);
}

ENDHLSL
```

- reflect(-viewDirWS, normalWS)
  finds the direction that a ray from the camera would bounce off the surface.
- sampling the cubemap (_ReflectionTex) in that direction gives us the color of whatever the surface “sees.”
- _ReflectionStrength blends between our base color and that reflection so we can make it fully mirror or just slightly glossy.

in the flashlight front glass, we keep a **medium strength** so it looks reflective but still shows the glass tint. on mirrors in the environment, the strength is higher for a clear reflection.

Why it matters

- creates believable reflective materials without real-time ray tracing.
- adds contrast in the scene (bright reflections vs dark corridors).
- helps communicate material differences — metal vs glass vs plastic