

Shaders



What is a shader ?

What is a shader ?

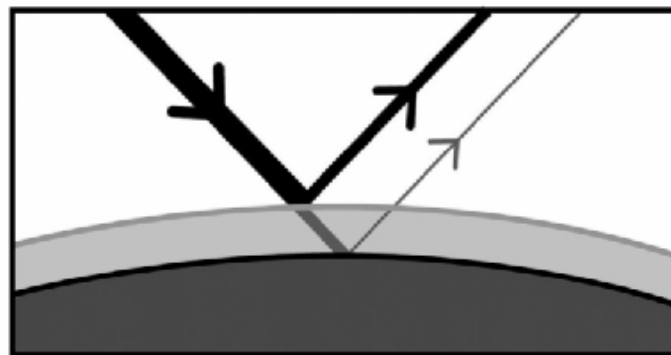
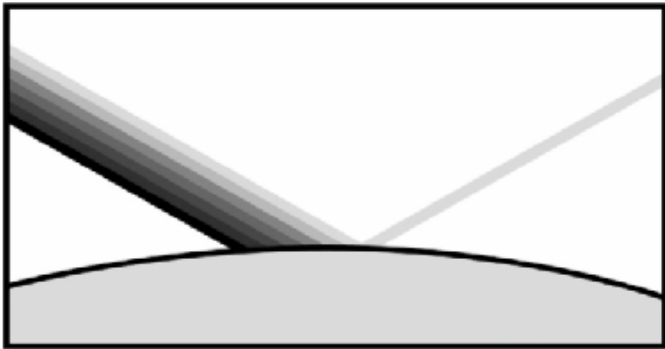
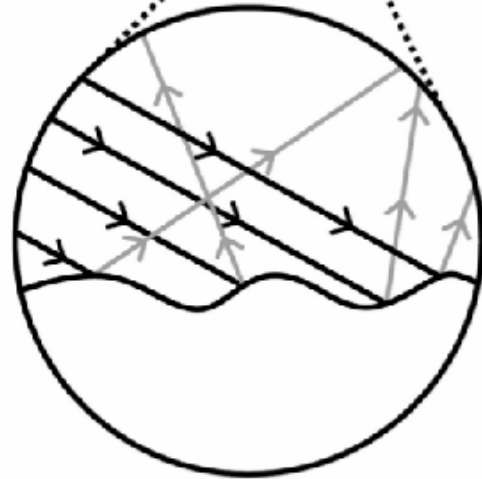
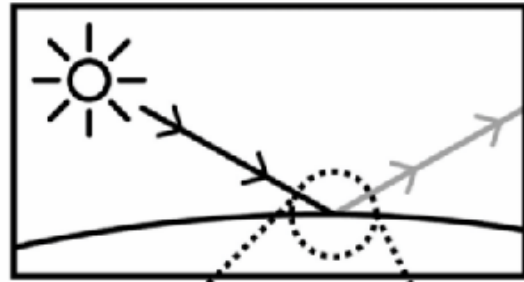
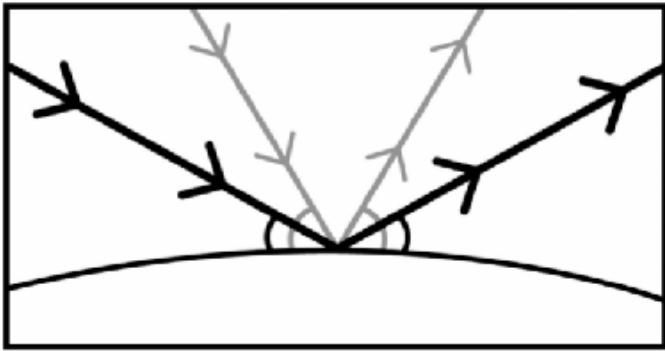
- A piece of code that runs on GPU
- ...which simulates some microscopic behaviour to create a photorealistic image



- In the physical world, light bounces (or is absorbed) on atoms with complex shapes
- This is not possible in graphics rendering: the computing would be too expensive.
- So we use simple shapes (triangles) and shaders to SIMULATE light behaviour on surfaces.
- There are two phases to render with shader : the outline phase and the painting phase.
 - The first choose to which triangle will belong a certain pixel of the screen
 - The second calculates the color of each pixel

Why a shader ?

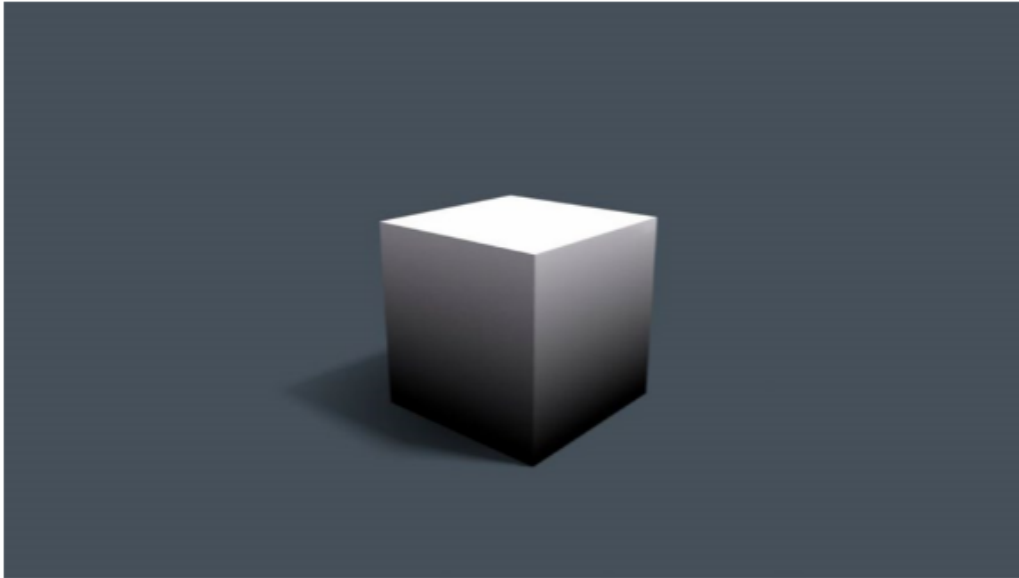
- In the physical world, light bounces (or is absorbed) on atoms with complex shapes



- This is not possible in graphics rendering: the computing would be too expensive.
- So we use simple shapes (triangles) and shaders to SIMULATE light behaviour on surfaces.

An exemple of rendering

- There are two phases to render with shader : the outline phase and the painting phase.
 - The first choose to which triangle will belong a certain pixel of the screen
 - The second calculates the color of each pixel



This scene has eight vertices, and it has been rendered to a 1920x1080 image (full HD resolution). What is happening exactly in the rendering process?

1. The scene's vertices and their respective data are passed to the vertex shader.
2. A vertex shader is executed on each of them.
3. The vertex shader produces an output data structure from each vertex, containing information such as color and position of the vertex on the final image.
4. Sequences of vertices are assembled into primitives, such as triangles, lines, points, and others. We'll assume triangles.
5. The rasterizer takes a primitive and transforms it into a list of pixels. For each potential pixel within that triangle, that structure's values are interpolated and passed to the pixel shader. (For example, if one vertex is green, and an adjacent vertex is red, the pixels between them will form a green to red gradient.) The rasterizer is part of the GPU; we can't customize it.
6. The fragment shader is run for any potential pixel. This is the phase that will be more interesting for us, as most lighting calculations happen in the fragment shader.
7. If the renderer is a forward render, for every light after the first, the fragment shader will be run again, with that light's data.
8. Each potential pixel (aka, fragment) is checked for whether there is another potential pixel nearer to the camera, therefore in front of the current pixel. If there is, the fragment will be rejected.
9. All the fragment shader light passes are blended together.
10. All pixel colors are written to a render target (could be the screen, or a texture, or a file, etc.)

Different kinds of shaders

- Vertex shader: Executed on every vertex.
- Fragment shader: Executed for every possible final pixel (known as a fragment).

In this course, we will use Unity. Unity has special shaders :

- Unlit shader: Unity-only, a shader that combines a vertex and pixel shader in one file.
- Surface shader: Unity-only, contains both vertex and fragment shader functionality, but takes advantage of the ShaderLab extensions to the Cg shading language to automate some of the code that's commonly used in lighting shaders.
- Image Effect shader: Unity-only, used to apply effects like Blur, Bloom, Depth of Field, Color Grading, etc. It is generally the last shader run on a render, because it's applied to a render of the geometry of the scene.

Unity uses two languages in its shaders : NVidia's CGLanguage, and ShaderLab, built in Unity.

There are also others shaders, for instance :

- Compute shader: Computes arbitrary calculations, not necessarily rendering, e.g., physics simulation, image processing, raytracing, and in general, any task that can be easily broken down into many independent tasks.

Programming your first shader

Most editors have visual programming tools to create shaders, but they restrict programming freedom, and sometimes generate bad code. So we will learn to program shaders by hand.

- Create a 3d object / sphere
- Create a Materials folder and a material that you will name RedMaterial
- Create a Shaders folder and create an Unlit shader inside
- Assign the RedShader to the RedMaterial by changing the "standard" shader in the material's dropdown menu. Then drag and drop the material on the sphere. It will become completely white.
- Open the shader code

Default unlit shader

Shader "Unlit/RedShader"

```
{
  Properties
  {
    _MainTex ("Texture", 2D) = "white" {}
  }
  SubShader
  {
    Tags { "RenderType"="Opaque" }
    LOD 100

    Pass
    {
      CGPROGRAM
      #pragma vertex vert
      #pragma fragment frag
      // make fog work
      #pragma multi_compile_fog

      #include "UnityCG.cginc"

      struct appdata
      {
        float4 vertex : POSITION;
        float2 uv : TEXCOORD0;
      };

      struct v2f
      {
        float2 uv : TEXCOORD0;
        UNITY_FOG_COORDS(1)
        float4 vertex : SV_POSITION;
      };

      sampler2D _MainTex;
      float4 MainTexST;

      v2f vert (appdata v)
      {
        v2f o;
        o.vertex = UnityObjectToClipPos(v.vertex);
        o.uv = TRANSFORM_TEX(v.uv, _MainTex);
        UNITY_TRANSFER_FOG(o,o.vertex);
        return o;
      }

      fixed4 frag (v2f i) : SV_Target
      {
        // sample the texture
        fixed4 col = tex2D(_MainTex, i.uv);
        // apply fog
        UNITY_APPLY_FOG(i.fogCoord, col);
        return col;
      }
    }
  }
}
```

Properties of the shader

There can be different subshaders with several passes

Information. Here, which rendering queue (opaque or transparent)

Starts program. Uses Nvidia CGLanguage

Shader compilation info

Library files needed to compile shaders

Information that comes from the app to the vertex shader

Information that will pass from vertex to fragment shader

Names after semicolons are Semantics, that will be stored in the struct

Define the properties types and variables used in the shader program

The names that were defined in the #pragma

Ends program

The red shader code

Simplify the shader so it will become :

```
Shader "Unlit/RedShader"
{
    SubShader
    {
        Tags { "RenderType"="Opaque" }

        Pass
        {
            CGPROGRAM
            #pragma vertex vert
            #pragma fragment frag

            #include "UnityCG.cginc"

            struct appdata
            {
                float4 vertex : POSITION;
            };

            struct v2f
            {
                float4 vertex : SV_POSITION;
            };

            v2f vert (appdata v)
            {
                v2f o;
                o.vertex = UnityObjectToClipPos(v.vertex);
                return o;
            }

            fixed4 frag (v2f i) : SV_Target
            {
                return fixed4(1, 0, 0, 1);
            }
            ENDCG
        }
    }
}
```

Add properties

Let's add a property. We don't want a hard coded color. Let's add a property :

```
Properties
{
    _Color ("Color", Color) = (1,0,0,1)
}
```

Add the variable in the program, and return the variable in the fragment shader.

```
fixed4 _Color;
...

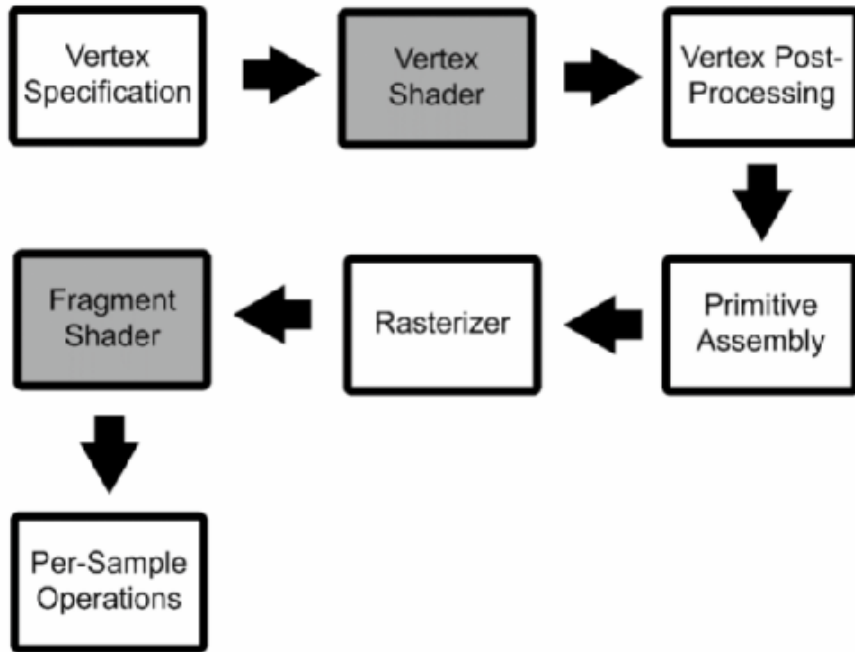
fixed4 frag (v2f i) : SV_Target
{
    return _Color;
}
```

Now you can modify the color in the material.

Rename the shader to MonochromeShader, and update the path.

The graphics pipeline in a real-time engine

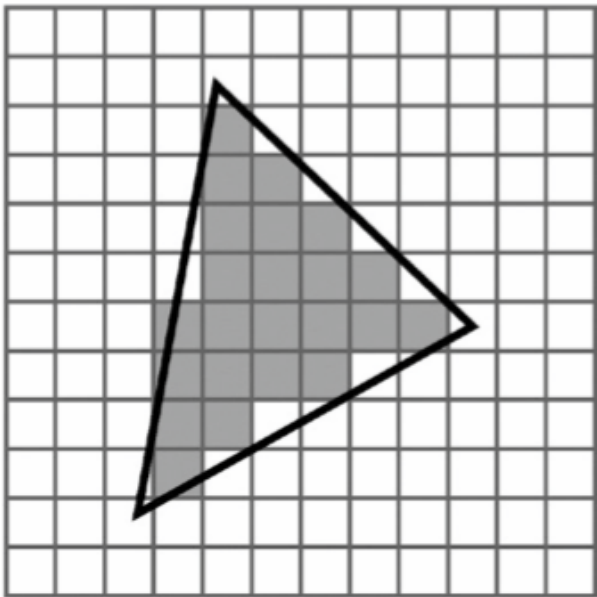
Stages



The stages of an example graphics pipeline are as follows:

- The input assembly stage gathers data from the scene (meshes, textures, and materials) and organizes it to be used in the pipeline.
- The vertex processing stage gets the vertices and their info from the previous stage and executes the vertex shader on each of them. The main objective of the vertex shader used to be obtaining 2D coordinates out of vertices. In more recent API versions, that is left to a different, later stage.
- The vertex post-processing stage includes transformations between coordinate spaces and the clipping of primitives that are not going to end up on the screen.
- The primitive assembly stage gathers the data output by the vertex processing stages in a primitive and prepares it to be sent to the next stage.
- The rasterizer is not a programmable stage. It takes a triangle (three vertices and their data) and creates potential pixels (fragments) out of it. It also produces an interpolated version of the vertex attributes data for each of the fragments and a depth value.
- The fragment shader stage runs the fragment shader on all the fragments that the rasterizer produces. In order to calculate the color of a pixel, multiple fragments may be necessary (e.g., antialiasing).
- The output merger performs the visibility test that determine whether a fragment will be overwritten by a fragment in front of it. It also does other tests, such as the blending needed for transparency, and more.

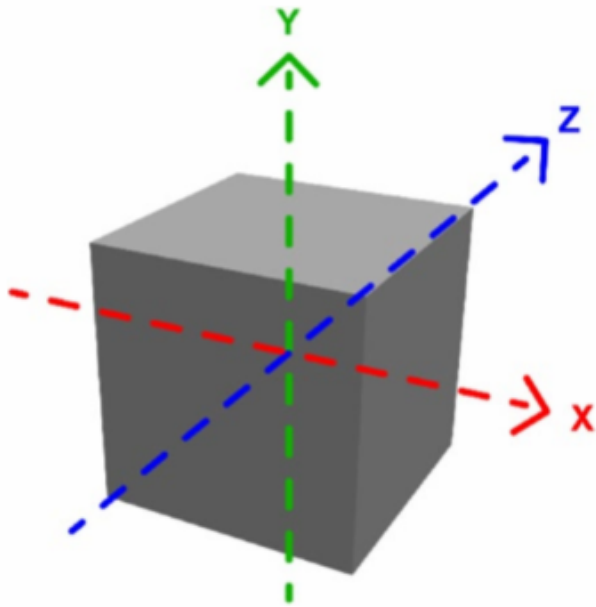
What the rasterizer does



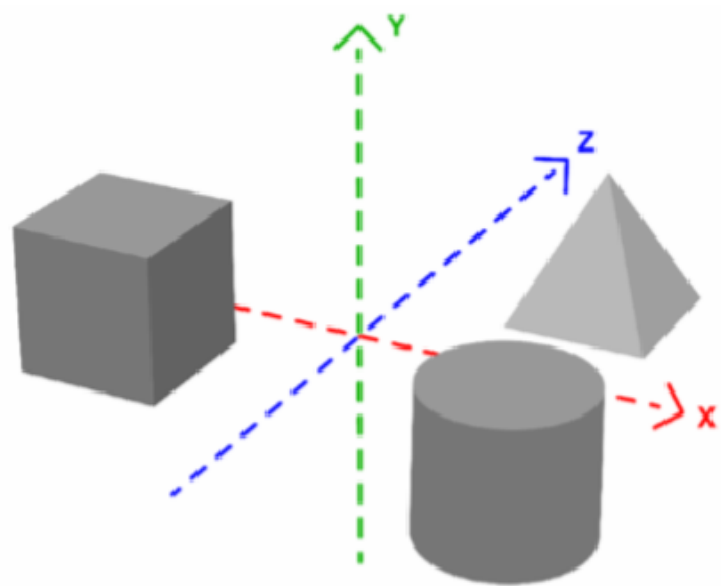
Transformations between spaces

Spaces

In real time graphics, coordinates are transformed between different spaces. Shaders computations are supposed to be done in specific coordinates spaces. This part of the lesson will presents the commonly used spaces.



Local/Object space



World space

- `float3 UnityObjectToWorldDir(in float3 dir)`
takes a direction in Object Space and transforms it to a direction in World Space
- `float3 UnityObjectToWorldNormal(in float3 norm)`
takes a normal in Object Space and transforms it to a normal in World Space; useful for lighting calculations
- `float3 UnityWorldSpaceViewDir(in float3 worldPos)`
takes a vertex position in World Space and returns the view direction in World Space; useful for lighting calculations
- `float3 UnityWorldSpaceLightDir(in float3 worldPos)`
takes a vertex position in World Space and returns the light direction in World Space; useful for lighting calculations

Mathematically, space transformations are 4×4 matrix multiplications. Here are some of the built-in Unity matrices for transformation from and to Object Space:

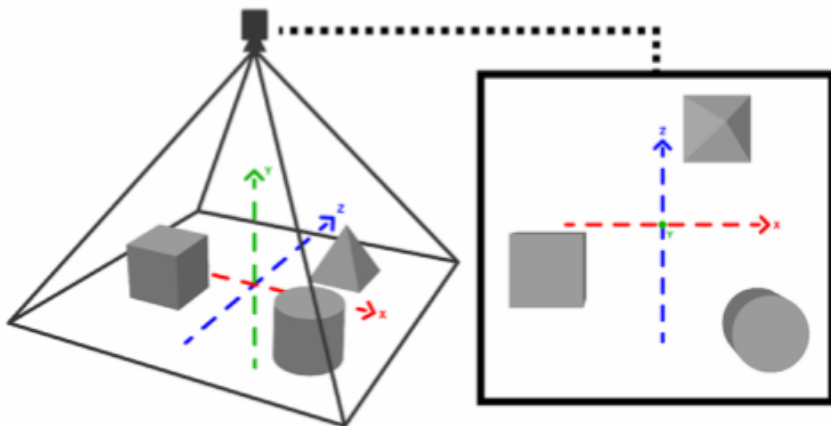
- `unity_ObjectToWorld`, which is a matrix that transforms from Object Space to World Space
- `unity_WorldToObject`, the inverse of the above, is a matrix that transforms from World Space to Object Space

As an example, let's translate the vertex position from Object Space to World Space:

```
float4 vertexWorld = mul(unity_ObjectToWorld, v.vertex);
```

Camera space

Also called View space :



There are a couple of built-in matrices for Camera Space:

- `unity_WorldToCamera`, which transforms from World Space to Camera Space
- `unity_CameraToWorld`, the inverse of the above, transforms from Camera Space to World Space

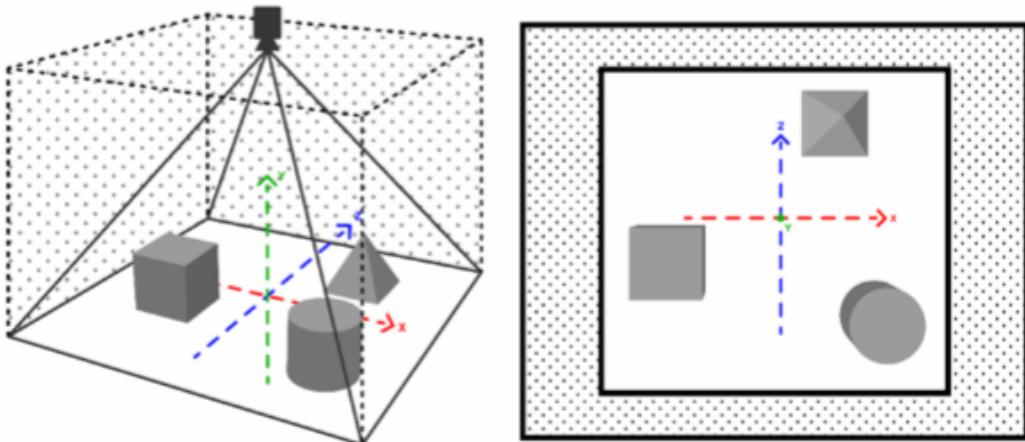
There is also one built-in function:

- `float4 UnityViewToClipPos(in float3 pos)` transforms a position from View Space to Clip Space

Spaces

Clip space

Range from -1 to 1 and remove all primitives outside the frustum (cf. rendering basics).



There are no built-in matrices for Clip Space, but there are some built-in functions for it:

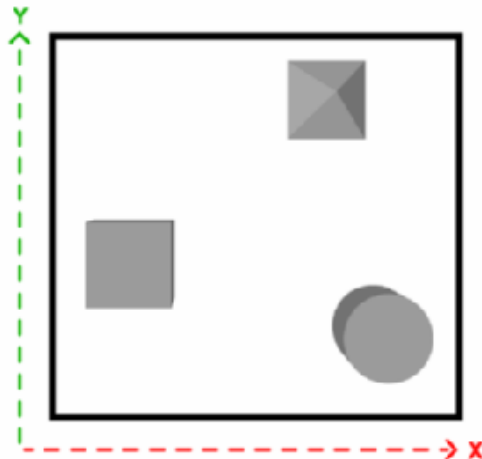
- `float4 UnityWorldToClipPos(in float3 pos)`, which transforms a position from World Space to Clip Space
- `float4 UnityViewToClipPos(in float3 pos)`, which transforms a position from View Space to Clip Space
- `float4 UnityObjectToClipPos(in float3 pos)`, which transforms a vertex position from Object Space to Clip Space

Spaces

Normalized Device Coordinates (NDC)

Next up are the Normalized Device Coordinates (NDC). This is a 2D space that is independent of the specific screen or image resolution. Coordinates in NDC are obtained by dividing Clip coordinates by the homogeneous coordinate (w , the 4th coordinate of the 4x4 transformation matrices), a process called perspective division. Again, NDC coordinates range from -1 to 1 in OpenGL. NDC uses three numbers instead of two, as you'd expect it to, but in this case the z coordinate is used for the depth buffer, rather than being a homogeneous coordinate.

Screen Space



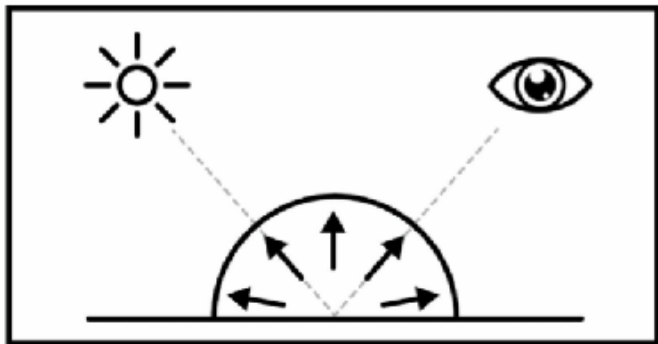
Screen Space is the coordinate space of the 2D render target. That may be a screen buffer, or a separate render target, or an image. It is obtained by transforming and scaling NDC into viewport resolution. Finally, these screen coordinates of the vertices are passed to the rasterizer, which will use them to produce fragments.

First lighting shader

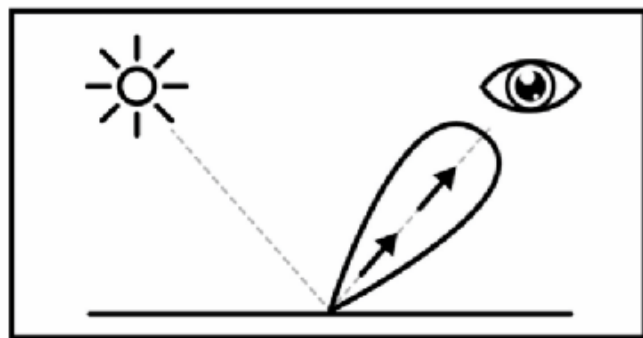
Before Physically Based Rendering

Before PBR, real time, rendering used rough approximations to simulate light behaviour. The shading used three components :

- Ambient lighting, a color that gives the scene a non-black tone
- Diffuse lighting, where light bounces on surfaces in every possible directions
- Specular lighting, where light bounces on surfaces only in a few directions



Diffuse

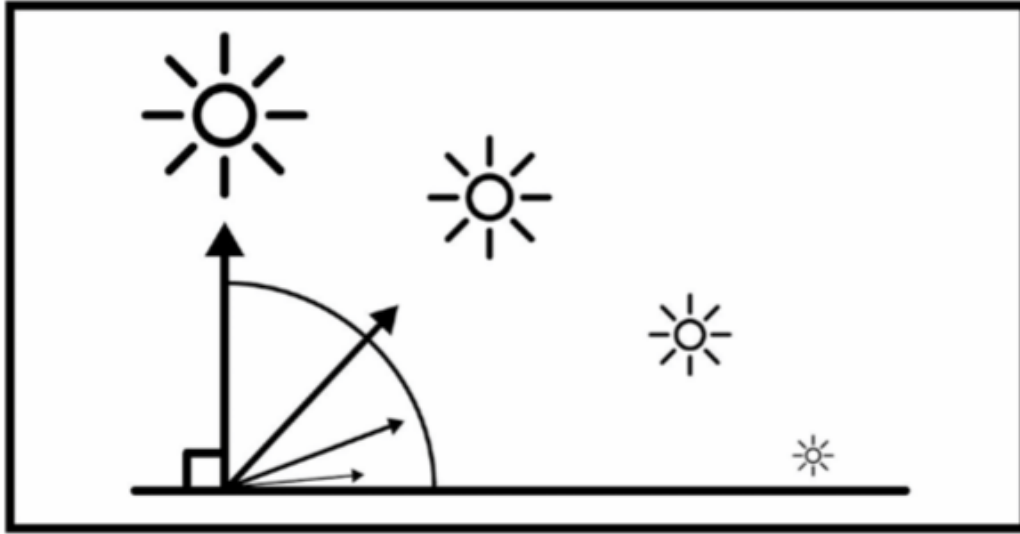


Specular

Angle of incidence and Lambert reflectance

To know how much light a surface receives, we can use the angle of incidence.

A surface will receive maximum light if the light emitter is just above its normal. It will receive no light if the light emitter is perpendicular to the normal. We can use the dot product of the light vector and the normal vector to compute brightness.



```
float brightness = dot( normal, lightDir ) // brightness calculated from the Normal and Light directions  
float3 pixelColor = brightness * lightColor * surfaceColor // final value of the surface color
```

This is the lambert approximation we have seen in the first lesson about rendering. Let's implement it in a shader.

Lambert shader

Shader "Custom/DiffuseShader"

```
{
  Properties
  {
    _Color ("Color", Color) = (1,0,0,1)
  }
  SubShader
  {
    Tags { "LightMode" = "ForwardBase" }
    LOD 100
    Pass
    {
      CGPROGRAM
      #pragma vertex vert
      #pragma fragment frag
      #include "UnityCG.cginc"
      #include "UnityLightingCommon.cginc"

      struct appdata
      {
        float4 vertex : POSITION;
        float3 normal : NORMAL;
      };

      struct v2f
      {
        float4 vertex : SV_POSITION;
        float3 worldNormal : TEXCOORD0;
      };

      float4 _Color;

      v2f vert (appdata v)
      {
        v2f o;
        o.vertex = UnityObjectToClipPos(v.vertex);
        float3 worldNormal = UnityObjectToWorldNormal(v.normal);
        o.worldNormal = worldNormal;
        return o;
      }

      float4 frag (v2f i) : SV_Target
      {
        float3 normalDirection = normalize(i.worldNormal);
        float nl = max(0.0, dot(normalDirection, _WorldSpaceLightPos0.xyz));
        float4 diffuseTerm = nl * Color * LightColor0;
        return diffuseTerm;
      }
    }
  }
}
```

← This pass will be used for the first light pass of the forward renderer

← Contains functions about lighting

← We need to ask the renderer what is the normal, so we add it in appdata

← Output info

← Calculate the normal in world coordinates

← Compute the incidence, then multiply by color of the surface and color of the light

Add a texture

Shader "Custom/DiffuseShader"

```
{
  Properties
  {
    __Color ("Color", Color) = (1,0,0,1)
    __DiffuseTex ("Texture", 2D) = "white" {}
  }
  SubShader
```

Add a texture property

```
{
  Tags { "LightMode" = "ForwardBase" }
  LOD 100
  Pass
  {
    CGPROGRAM
    #pragma vertex vert
    #pragma fragment frag
    #include "UnityCG.cginc"
    #include "UnityLightingCommon.cginc"
```

```
    struct appdata
    {
      float4 vertex : POSITION;
      float3 normal : NORMAL;
      float2 uv : TEXCOORD0;
    };
```

Texture coordinates

```
    struct v2f
    {
      float4 vertex : SV_POSITION;
      float3 worldNormal : TEXCOORD1;
      float2 uv : TEXCOORD0;
    };
```

Change to TEXCOORD1

Add UVs

```
    sampler2D _DiffuseTex;
    float4 _DiffuseTexST;
    float4 _Color;
```

Add variables for texture

```
    v2f vert (appdata v)
    {
      v2f o;
      o.vertex = UnityObjectToClipPos(v.vertex);

      o.uv = TRANSFORM_TEX(v.uv, _DiffuseTex);

      float3 worldNormal = UnityObjectToWorldNormal(v.normal);
      o.worldNormal = worldNormal;
      return o;
    }
```

Calculate the normal in world coordinates

```
    float4 frag (v2f i) : SV_Target
    {
      float3 normalDirection = normalize(i.worldNormal);

      float4 tex = tex2D(_DiffuseTex, i.uv);

      float nl = max(0.0, dot(normalDirection, _WorldSpaceLightPos0.xyz));
      float4 diffuseTerm = nl * _Color * _LightColor0;
      return diffuseTerm;
    }
  }
  ENDCG
}
```

Use the texture with the diffuse calculation

Multiply by texture color

Ambient light

Ambient light is a value under which the diffuse cannot drop.

Shader "Custom/DiffuseShader"

```
{
  Properties
  {
    _Color ("Color", Color) = (1,0,0,1)
    _DiffuseTex ("Texture", 2D) = "white" {}
    _Ambient ("Ambient", Range (0, 1)) = 0.25
```

← Range property

SubShader

```
{
  Tags { "LightMode" = "ForwardBase" }
  LOD 100
  Pass
  {
    CGPROGRAM
    #pragma vertex vert
    #pragma fragment frag
    #include "UnityCG.cginc"
    #include "UnityLightingCommon.cginc"
```

```
    struct appdata
    {
      float4 vertex : POSITION;
      float3 normal : NORMAL;
      float2 uv : TEXCOORD0;
    };

```

```
    struct v2f
    {
      float4 vertex : SV_POSITION;
      float3 worldNormal : TEXCOORD1;
      float2 uv : TEXCOORD0;
    };

```

```
    sampler2D _DiffuseTex;
    float4 _DiffuseTexST;
    float4 _Color;
    float _Ambient;
```

← Add variable for Ambient

```
    v2f vert (appdata v)
    {
      v2f o;
      o.vertex = UnityObjectToClipPos(v.vertex);

      o.uv = TRANSFORM_TEX(v.uv, _DiffuseTex);

      float3 worldNormal = UnityObjectToWorldNormal(v.normal);
      o.worldNormal = worldNormal;
      return o;
    }

```

```
    float4 frag (v2f i) : SV_Target
    {
      float3 normalDirection = normalize(i.worldNormal);

      float4 tex = tex2D(_DiffuseTex, i.uv);

      float nl = max(_Ambient, dot(normalDirection, WorldSpaceLightPos0.xyz));
      float4 diffuseTerm = nl * _Color * _LightColor0;
      return diffuseTerm;
    }
    ENDCG
  }
}
```

← Replace 0 by _Ambient

Speculars with Phong

Now that we know about the diffuse approximation, we will talk about the specular approximation. Phong approximation is one of the simplest :

Reflection direction = $2 * (N \cdot L) * N - L$ with N the normal to the triangle and L the light direction

The specular (= mirror) light that bounces on the surface depends of your point of view. This fonction is usually implemented in shader languages with the word "reflect".

With reflect, Phong implementation is :

```
float3 reflectionVector = reflect (-lightDir, normal);  
float specDot = max(dot(reflectionVector, eyeDir), 0.0);  
float spec = pow(specDot, specExponent);
```

specExponent controls the specular intensity.

Now duplicate and rename to SpecularShader your Lambert shader, create a new SpecularMaterial and assign the new shader. Don't forget to update the shader path.

Implement Phong

```
Shader "Custom/SpecularShader"
```

```
{
```

```
    Properties
```

```
    {
```

```
        ...  
        _SpecColor ("Specular Material Color", Color) = (1,1,1,1) ← Color and intensity
```

```
        _Shininess ("Shininess", Float) = 10
```

```
    }
```

```
    SubShader
```

```
    {
```

```
        Tags { "LightMode" = "ForwardBase" }
```

```
        LOD 100
```

```
        Pass
```

```
        {
```

```
            ...
```

```
            struct v2f
```

```
            {
```

```
                float4 vertex : SV_POSITION;
```

```
                float3 worldNormal : TEXCOORD2;
```

```
                float4 vertexWorld : TEXCOORD1; ← Vertex world position
```

```
                float2 uv : TEXCOORD0; (update the numbers)
```

```
            };
```

```
            sampler2D _DiffuseTex;
```

```
            float4 _DiffuseTexST;
```

```
            float4 _Color;
```

```
            float _Ambient;
```

```
            float _Shininess; ← Do not forget the variable
```

```
            v2f vert (appdata v)
```

```
            {
```

```
                v2f o;
```

```
                o.vertex = UnityObjectToClipPos(v.vertex);
```

```
                o.vertexWorld = mul(unity_ObjectToWorld, v.vertex); ← Compute the vertexWorld
```

```
                o.uv = TRANSFORM_TEX(v.uv, _DiffuseTex);
```

```
                float3 worldNormal = UnityObjectToWorldNormal(v.normal);
```

```
                o.worldNormal = worldNormal;
```

```
                return o;
```

```
            }
```

```
            float4 frag (v2f i) : SV_Target
```

```
            {
```

```
                float3 normalDirection = normalize(i.worldNormal);
```

```
                float3 viewDirection = normalize(UnityWorldSpaceViewDir(i.vertexWorld)); ← Needed for specular calculation
```

```
                float3 lightDirection = normalize(UnityWorldSpaceLightDir(i.vertexWorld));
```

```
                float4 tex = tex2D(_DiffuseTex, i.uv);
```

```
                // Diffuse
```

```
                float nl = max(_Ambient, dot(normalDirection, WorldSpaceLightPos0.xyz));
```

```
                float4 diffuseTerm = nl * _Color * tex * _LightColor0;
```

```
                //Specular implementation (Phong)
```

```
                float3 reflectionDirection = reflect(-lightDirection, normalDirection);
```

```
                float3 specularDot = max(0.0, dot(viewDirection, reflectionDirection)); ← The specular depends of the angle
```

```
                float3 specular = pow(specularDot, _Shininess);
```

```
                float4 specularTerm = float4(specular, 1) * _SpecColor * _LightColor0; ← Multiply the specular by the light color
```

```
                // Add diffuse and specular
```

```
                float4 finalColor = diffuseTerm + specularTerm;
```

```
                return finalColor;
```

```
            }
```

```
        } ENDCG
```

```
    }
```

```
}
```

```
}
```

Phong with multiple lights

Now duplicate and rename your SpecularShader to , create a new SpecularMaterial and assign the new shader. Don't forget to update the shader path.

```
Shader "Custom/SpecularShaderForwardAdd"
```

```
{
  Properties
  {
    ...
  }
  SubShader
  {
    Tags { "LightMode" = "ForwardBase" }
    LOD 100
    Pass
    {
      CGPROGRAM
      #pragma vertex vert
      #pragma fragment frag
      #pragma multi_compile_fwdbase
      ...
    }
    Pass
    {
      Tags { "LightMode" = "ForwardAdd" }
      Blend One One
      CGPROGRAM
      #pragma vertex vert
      #pragma fragment frag
      #pragma multi_compile_fwdadd
      ...

      float4 frag (v2f i) : SV_Target
      {
        ...
        // Diffuse
        float nl = max(0.0, dot(normalDirection, _WorldSpaceLightPos0.xyz));
        ...
      }
    }
  }
}
```



For each light after the first

Blend colors additively

Don't add _Ambient several times