Shaders

Gaëtan Blaise-Cazalet - Head of programming section

What is a shader ?

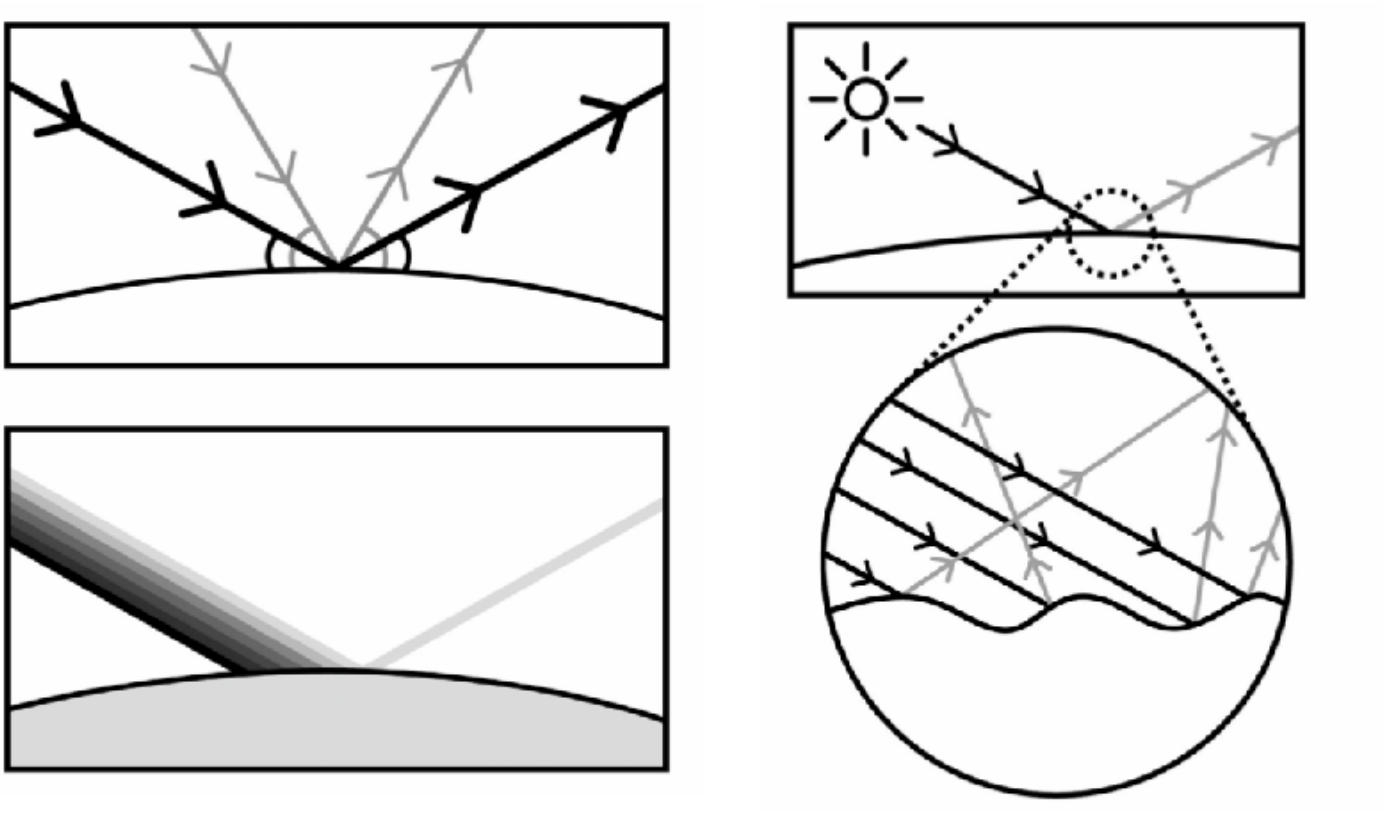
What is a shader ?

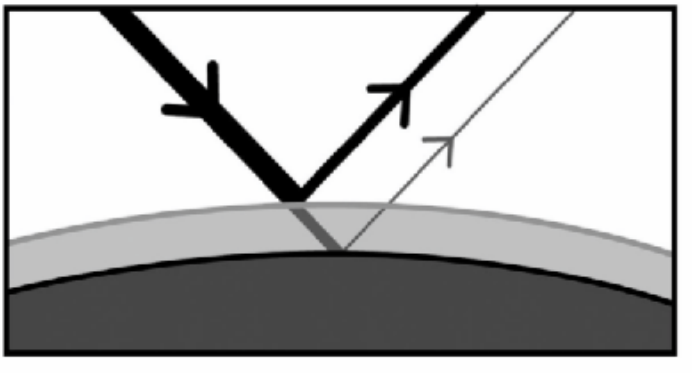
- A piece of code that runs on GPU
- ...which simulates some microscopic behaviour to create a photorealist image

- 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

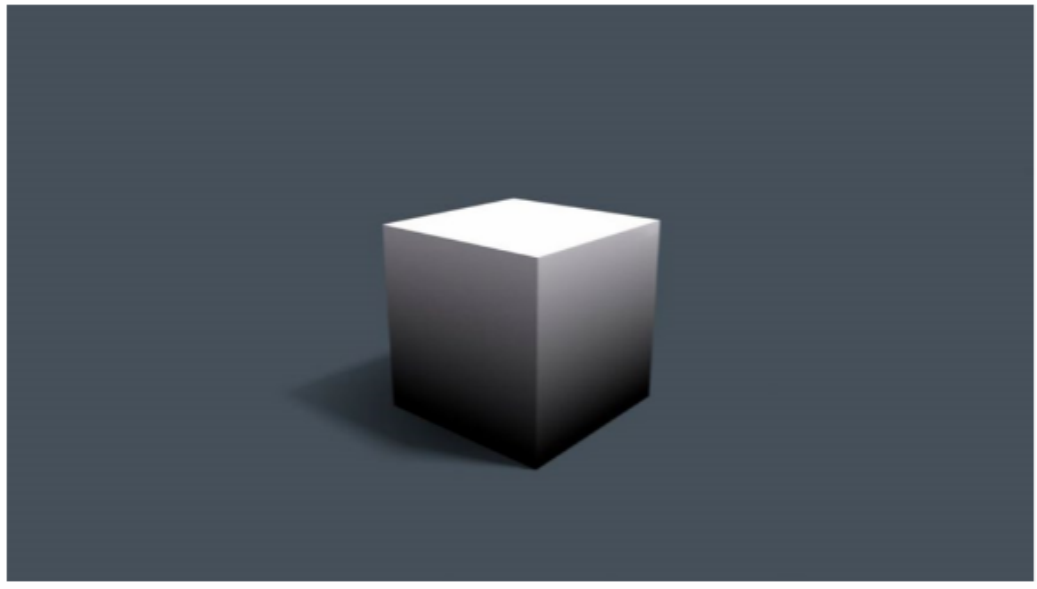




- This is not possible in graphics rendering: the computing would be too expensive.
- So we use simple shapes (triangles) and shaders to SIMULATE light behaviour on surfaces.

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 ohers 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;
			}
			ENDCG
		}
	}
}
```

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 semicolumns 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.

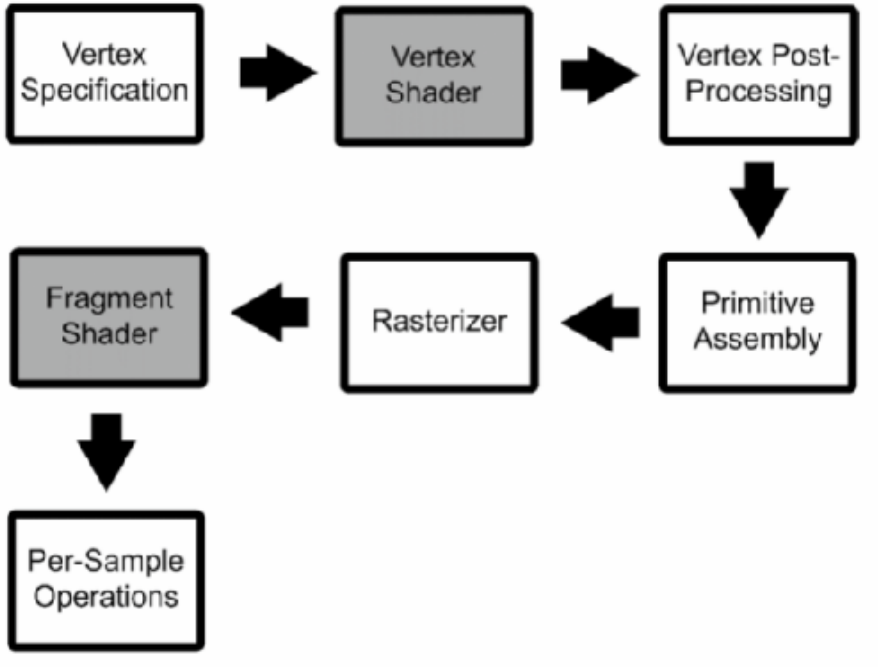
```
fixed4 _Color;
...

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

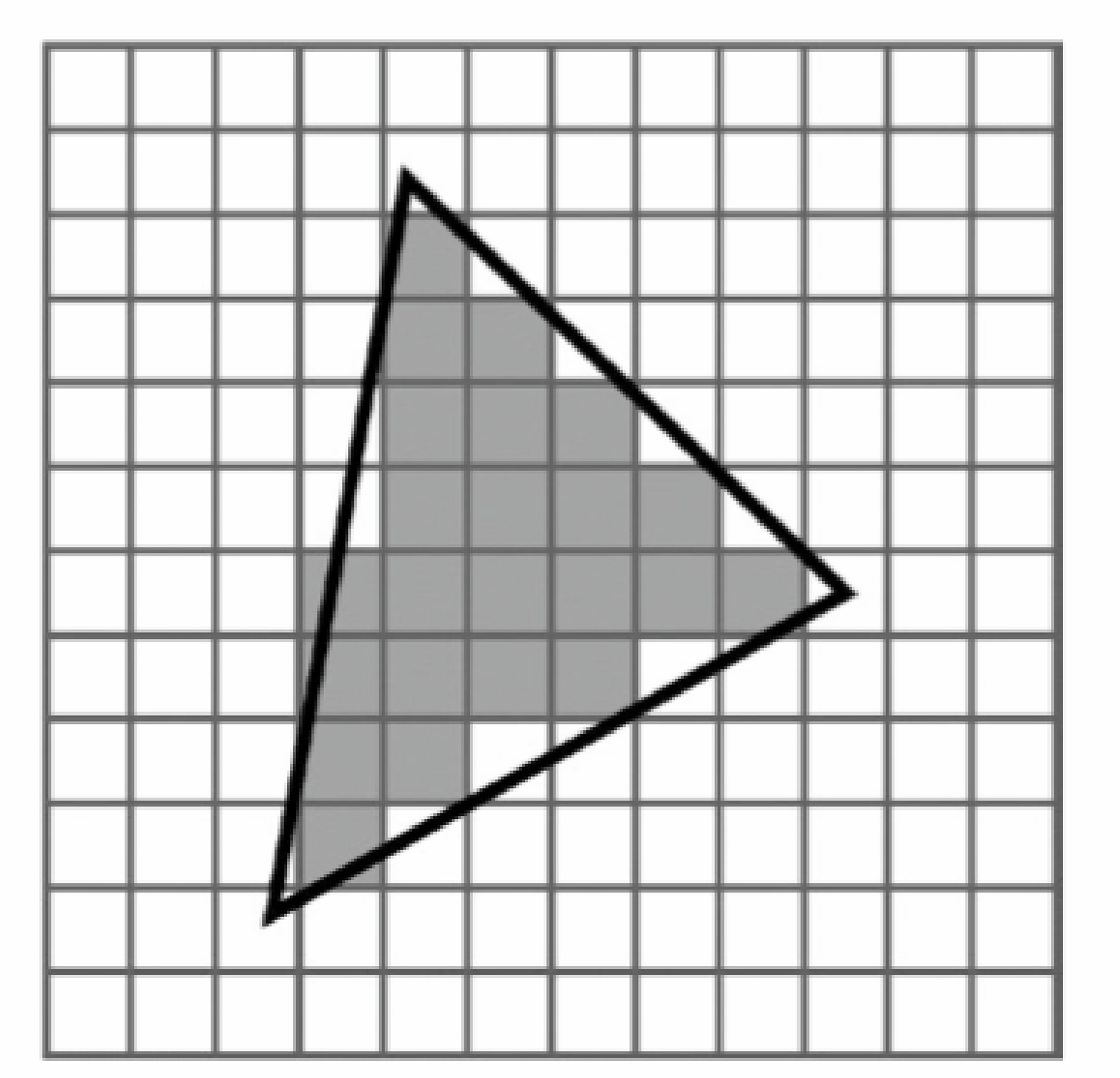
Stages



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.

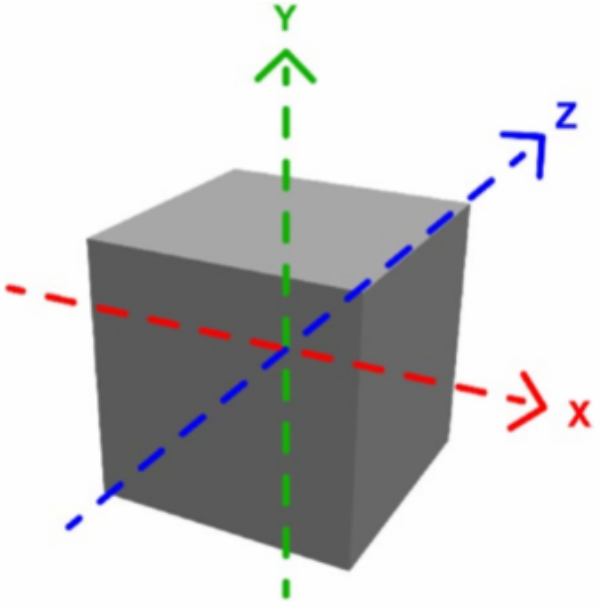
What the rasterizer does



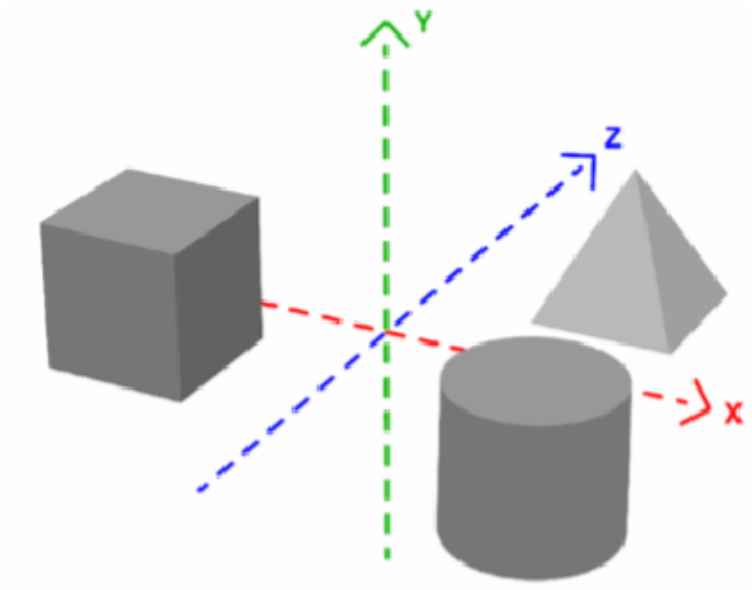
Transformations between spaces

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

Mathemaically, space transformations are 4x4 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

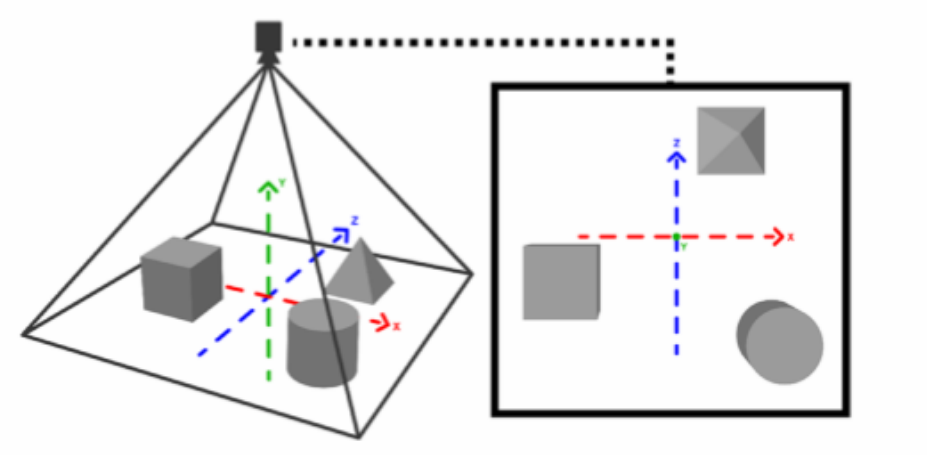
As an example, let's translate the vertex position from Object Space to World Space:

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

Spaces

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

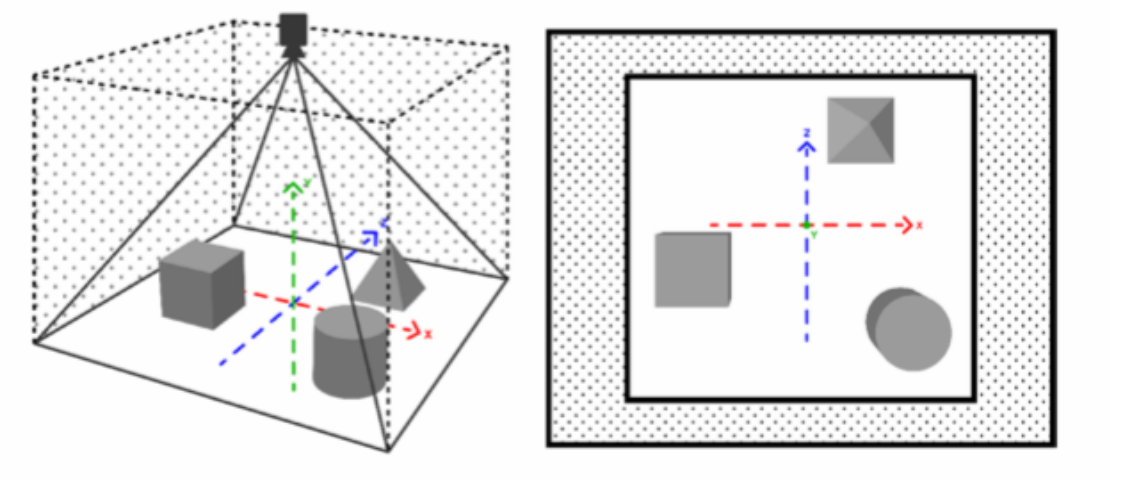
There is also one built-in function:

- 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 frustrum (cf. rendering basics).



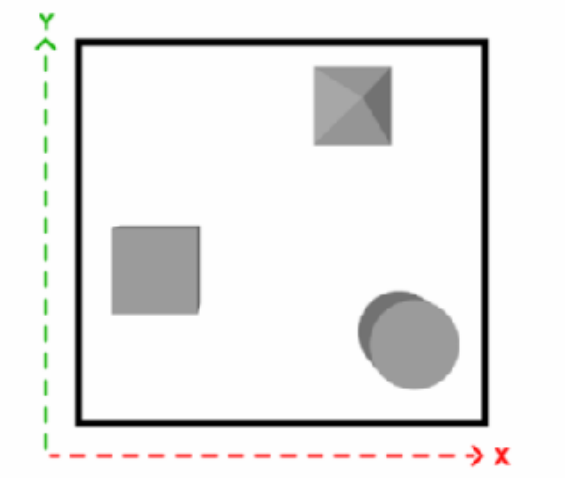
There are no built-in matrices for Clip Space, but there are some built-in functions for it:

- 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

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 diving 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



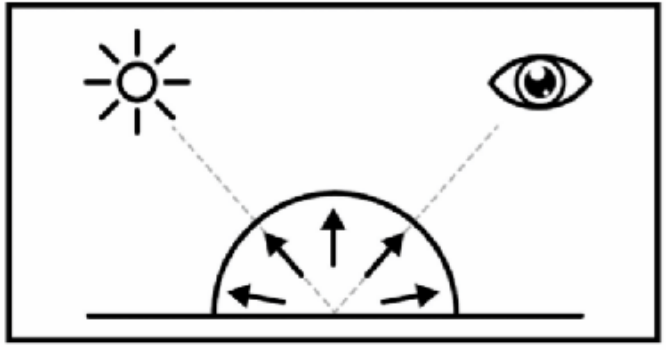
Screen Space is the coordinate space of the 2D render target. That may be a screen buffer, or a separate render target, or an image. It is obtained by transforming and scaling NDC into viewport resolution. Finally, these screen coordinates of the vertices are passed to the rasterizer, which will use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

First lighting shader

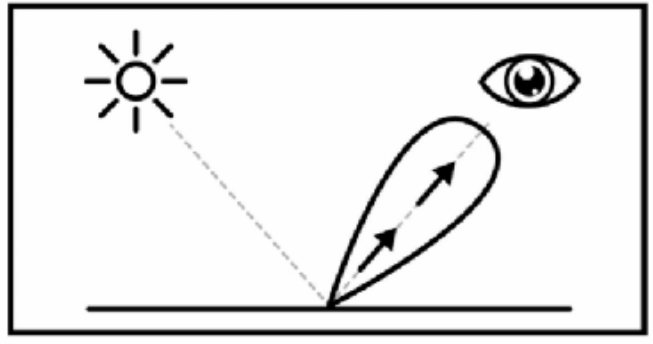
Before Physically Based Rendering

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

- Ambiant 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

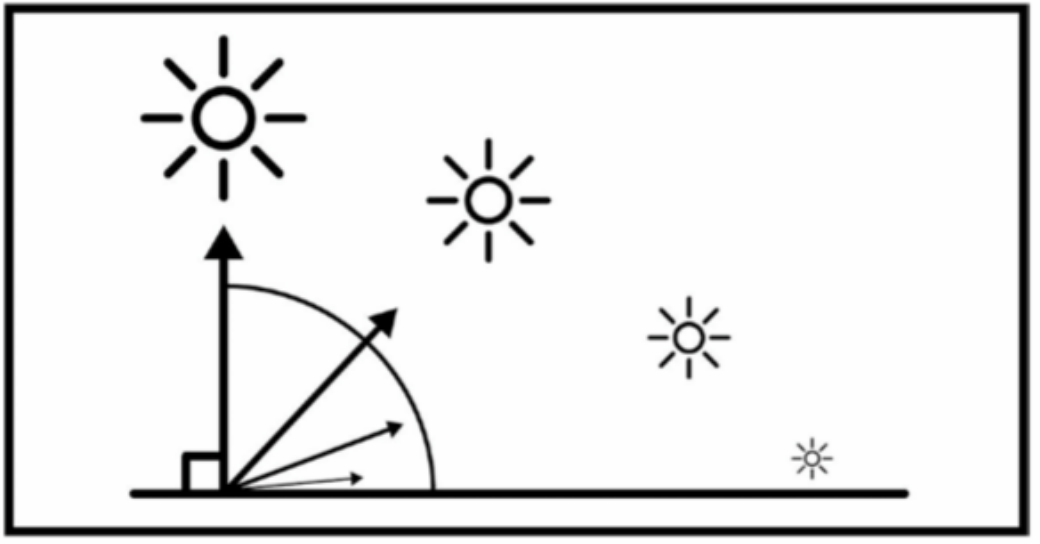


Specular

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 emiter is just above its normal. It will receive no light if the light emiter 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;
            }
            ENDCG
        }
    }
}
```

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
    {
        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 _DiffuseTex_ST;
            float4 _Color;

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

Add a texture property

Texture coordinates

Change to TEXCOORD1

Add UVs

Add variables for texture

Calculate the normal in world coordinates

Scale and offset texture coordinates

Use the texture with the diffuse calculation

Multiply by texture color

Add a texture in your unity assets. Chosse the texture in the material shader's properties.

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
    }
    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;

            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
        }
    }
}
```

Range property

Add variable for Ambient

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 . 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)
      _Shininess ("Shininess", Float) = 10
   }
   SubShader
   {
      Tags { "LightMode" = "ForwardBase" }
      LOD 100
      Pass
      {
         ...

         struct v2f
         {
            float4 vertex : SV_POSITION;
            float3 worldNormal : TEXCOORD2;
            float4 vertexWorld : TEXCOORD1;
            float2 uv : TEXCOORD0;
         };

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

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

            o.vertexWorld = mul(unity_ObjectToWorld, 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);
            float3 viewDirection = normalize(UnityWorldSpaceViewDir(i.vertexWorld));
            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));
            float3 specular = pow(specularDot, _Shininess);
            float4 specularTerm = float4(specular, 1) * _SpecColor * _LightColor0;

            // Add diffuse and specular
            float4 finalColor = diffuseTerm + specularTerm;
            return finalColor;
         }
         ENDCG
      }
   }
}
```

Color and intensity

Vertex world position
(update the numbers)

Do not forget the variable

Compute the vertexWorld

Needed for specular calculation

The computation we have seen before

The specular depends of the angle

Multiply the specular by the light color

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
   {
      LOD 100
      Pass
      {
         Tags { "LightMode" = "ForwardBase" }
         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

Surface shaders

Differences with previous shaders

Create a new Surface Shader.

Surface shaders hide a part fo the shader code. You won't have to copy and paste passes like we did just before, except if you need precise control on your shaders.

In surface shader, the vertex function is optional, and the fragment function is replaced by a surface function. You can also write optional lighting functions.

#pragma surface surf Standard fullforwardshadows

The pragma tells that the surface function will we surf, there will be a Standard lighting model and a fullforwardshadows option. To write your optional vertex function, you have to use this option : vertex:vert (vert function will be the vertex shader function)

Lighting model can be Standard, BlinnPhong, Lambert, or a custom one.

The surf function takes a Input parameter (with the texture UV inside) and a inout SurfaceOutputStandard, which will serves as a input AND a output parameter. This parameter will be sent to the lighting model. The function fills the SurfaceOutputStandard fields.

SurfaceOutputStandard type is :

```
struct SurfaceOutputStandard {
    fixed3 Albedo;      // base (diffuse or specular) color
    fixed3 Normal;      // tangent space normal, if written
    half3 Emission;
    half Metallic;      // 0=non-metal, 1=metal
    half Smoothness;    // 0=rough, 1=smooth
    half Occlusion;     // occlusion (default 1)
    fixed Alpha;        // alpha for transparencies
};
```

Lighting models

The Standard Lighting model is complex, so we will look at the Lambert lighting model :*

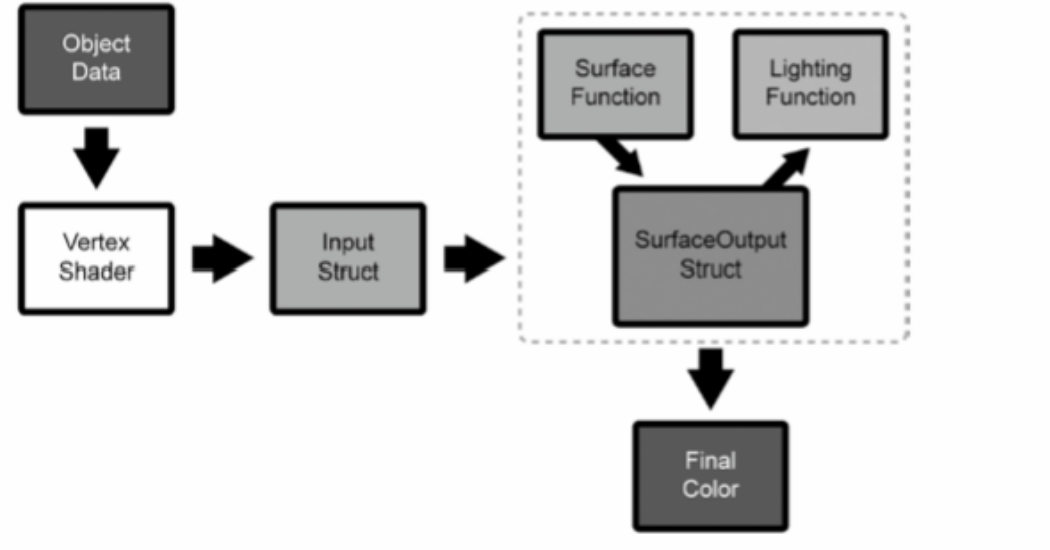
```
inline fixed4 LightingLambert (SurfaceOutput s, UnityGI gi) {
    fixed4 c;
    UnityLight light = gi.light;
    fixed diff = max (0, dot (s.Normal, light.dir));

    c.rgb = s.Albedo * light.color * diff;
    c.a = s.Alpha;

    #ifdef UNITY_LIGHT_FUNCTION_APPLY_INDIRECT
        c.rgb += s.Albedo * gi.indirect.diffuse;
    #endif
    return c;
}
```

A lighting model take a SurfaceOutput as a parameter and returns a fixed4. The UnityGI is a parameter that is passed by the Global Illumination system, which is a way to manage indirect lights. Here we only use gi.light.

You recognize the algorithm we used for Lambert's shader. If we would like to use the point of view, we would have called :

```
half4 Lighting<Name> (SurfaceOutput s, half3 viewDir, UnityGI gi)     // note the viewDir parameter
```

Surface shader pipeline



Add a second albedo

We want to add a second texture and a slider that will determine the proportion of each texture display.

```
Shader "Custom/SurfaceShader2Tex" {
	Properties {
		_Color ("Color", Color) = (1,1,1,1)
		_MainTex ("Albedo (RGB)", 2D) = "white" {}
		_SecondAlbedo ("Second Albedo (RGB)", 2D) = "white" {}
		_AlbedoLerp ("Albedo Lerp", Range(0,1)) = 0.5
		_Glossiness ("Smoothness", Range(0,1)) = 0.5
		_Metallic ("Metallic", Range(0,1)) = 0.0
	}
	SubShader {
		Tags { "RenderType"="Opaque" }
		LOD 200

		CGPROGRAM
		// Physically based Standard lighting model, and enable shadows on all light types
		#pragma surface surf Standard fullforwardshadows

		// Use shader model 3.0 target, to get nicer looking lighting
		#pragma target 3.0

		sampler2D _MainTex;
		sampler2D _SecondAlbedo;
		half _AlbedoLerp;

		struct Input {
			float2 uv_MainTex;
		};

		half _Glossiness;
		half _Metallic;
		fixed4 _Color;

		// Add instancing support for this shader. You need to check 'Enable Instancing' on
materials that use the shader.
		// See https://docs.unity3d.com/Manual/GPUInstancing.html for more information about
instancing.
		// #pragma instancing_options assumeuniformscaling
		UNITY_INSTANCING_BUFFER_START(Props)
			// put more per-instance properties here
		UNITY_INSTANCING_BUFFER_END(Props)

		void surf (Input IN, inout SurfaceOutputStandard o) {
			// Albedo comes from a texture tinted by color
			fixed4 c = tex2D (MainTex, IN.uvMainTex); //* _Color;
			fixed4 secondAlbedo = tex2D (_SecondAlbedo, IN.uv_MainTex);
			o.Albedo = lerp(c, secondAlbedo, _AlbedoLerp)  * _Color;
			// Metallic and smoothness come from slider variables
			o.Metallic = _Metallic;
			o.Smoothness = _Glossiness;
			o.Alpha = c.a;
		}
		ENDCG
	}
	FallBack "Diffuse"
}
```

New properties

New variables

Load the second texture color

Mix with the first texture

Add a normal map

Note that once you add your new material, the shadow will cast on the nearby duck.

```
Shader "Custom/SurfaceShaderNormal" {
	Properties {
		_Color ("Color", Color) = (1,1,1,1)
		_MainTex ("Albedo (RGB)", 2D) = "white" {}
		_NormalMap("Normal Map", 2D) = "bump" {}
		_Glossiness ("Smoothness", Range(0,1)) = 0.5
		_Metallic ("Metallic", Range(0,1)) = 0.0
	}
	SubShader {
		Tags { "RenderType"="Opaque" }
		LOD 200

		CGPROGRAM
		// Physically based Standard lighting model, and enable shadows on all light types
		#pragma surface surf Standard fullforwardshadows

		// Use shader model 3.0 target, to get nicer looking lighting
		#pragma target 3.0

		sampler2D _MainTex;
		sampler2D _NormalMap;

		struct Input {
			float2 uv_MainTex;
		};

		half _Glossiness;
		half _Metallic;
		fixed4 _Color;

		// Add instancing support for this shader. You need to check 'Enable Instancing' on
materials that use the shader.
		// See https://docs.unity3d.com/Manual/GPUInstancing.html for more information about
instancing.
		// #pragma instancing_options assumeuniformscaling
		UNITY_INSTANCING_BUFFER_START(Props)
			// put more per-instance properties here
		UNITY_INSTANCING_BUFFER_END(Props)

		void surf (Input IN, inout SurfaceOutputStandard o) {
			// Albedo comes from a texture tinted by color
			fixed4 c = tex2D (_MainTex, IN.uv_MainTex)* _Color;
			o.Normal = UnpackNormal (tex2D (_NormalMap, IN.uv_MainTex));
			o.Albedo = c.rgb;
			// Metallic and smoothness come from slider variables
			o.Metallic = _Metallic;
			o.Smoothness = _Glossiness;
			o.Alpha = c.a;
		}
		ENDCG
	}
	FallBack "Diffuse"
}
```

New property

New variable

Simplified compared to a vertex/fragment shader code

Change the light model

```
Shader "Custom/SurfaceShaderBlinnPhong" {
	Properties {
		_Color ("Color", Color) = (1,1,1,1)
		_MainTex ("Albedo (RGB)", 2D) = "white" {}
		_SpecColor ("Specular Material Color", Color) = (1,1,1,1)
		_Shininess ("Shininess", Range (0.03, 1)) = 0.078125
	}
	SubShader {
		Tags { "RenderType"="Opaque" }
		LOD 200

		CGPROGRAM
		// Physically based Standard lighting model, and enable shadows on all light types
		#pragma surface surf BlinnPhong fullforwardshadows

		// Use shader model 3.0 target, to get nicer looking lighting
		#pragma target 3.0

		sampler2D _MainTex;

		struct Input {
			float2 uv_MainTex;
		};

		fixed4 _Color;
		float _Shininess;

		UNITY_INSTANCING_BUFFER_START(Props)
			// put more per-instance properties here
		UNITY_INSTANCING_BUFFER_END(Props)

		void surf (Input IN, inout SurfaceOutput o) {
			// Albedo comes from a texture tinted by color
			fixed4 c = tex2D (_MainTex, IN.uv_MainTex) * _Color;
			o.Albedo = c.rgb;
			o.Specular = _Shininess;
			o.Gloss = c.a;
			o.Alpha = 1.0f;
		}
		ENDCG
	}
	FallBack "Diffuse"
}
```

New property

New property

BlinnPhong light model

No metalness nor smoothness

A custom light model

To write a custom light model with forward rendering, we can use those functions' signatures :

half4 Lighting<Name> (SurfaceOutput s, UnityGI gi); (view independant)

half4 Lighting<Name> (SurfaceOutput s, half3 viewDir, UnityGI gi); (view dependent)

If you would prefer to use deferred rendering, you would use :

half4 Lighting<Name>_Deferred (SurfaceOutput s, UnityGI gi, out half4 outDiffuseOcclusion, out half4 outSpecSmoothness, out half4 outNormal);

half4 Lighting<Name>_PrePass (SurfaceOutput s, half4 light);

Difference between defered rendering and forward rendering :
https://gamedevelopment.tutsplus.com/articles/forward-rendering-vs-deferred-rendering--gamedev-12342

TLDR When to choose forward, when to choose deferred :
https://unity3d.com/fr/learn/tutorials/topics/graphics/choosing-rendering-path

We will reimplement phong lighting, but this time with a custom lignt model.

Custom light model : Phong

```
Shader "Custom/SurfaceShaderPhong" {
	Properties {
		_Color ("Color", Color) = (1,1,1,1)
		_MainTex ("Albedo (RGB)", 2D) = "white" {}
		_SpecColor ("Specular Material Color", Color) = (1,1,1,1)
		_Shininess ("Shininess", Range (0.03, 128)) = 0.078125
	}
	SubShader {
		Tags { "RenderType"="Opaque" }
		LOD 200

		CGPROGRAM
		// Physically based Standard lighting model, and enable shadows on all light types
		#pragma surface surf Phong fullforwardshadows

		// Use shader model 3.0 target, to get nicer looking lighting
		#pragma target 3.0

		sampler2D _MainTex;
		float _Shininess;
		fixed4 _Color;

		struct Input {
			float2 uv_MainTex;
		};


		UNITY_INSTANCING_BUFFER_START(Props)
			// put more per-instance properties here
		UNITY_INSTANCING_BUFFER_END(Props)

		inline void LightingPhong_GI (SurfaceOutput s, UnityGIInput data, inout UnityGI gi)
		{
			gi = UnityGlobalIllumination (data, 1.0, s.Normal);
		}

		inline fixed4 LightingPhong (SurfaceOutput s, half3 viewDir, UnityGI gi)
		{
			UnityLight light = gi.light;
			float nl = max(0.0f, dot(s.Normal, light.dir));
			float3 diffuseTerm = nl * s.Albedo.rgb * light.color;
			float3 reflectionDirection = reflect(-light.dir, s.Normal);
			float3 specularDot = max(0.0, dot(viewDir, reflectionDirection));
			float3 specular = pow(specularDot, _Shininess);
			float3 specularTerm = specular * _SpecColor.rgb * light.color.rgb;
			float3 finalColor = diffuseTerm.rgb + specularTerm;

			fixed4 c;
			c.rgb = finalColor;
			c.a = s.Alpha;
#ifdef UNITY_LIGHT_FUNCTION_APPLY_INDIRECT
			c.rgb += s.Albedo * gi.indirect.diffuse;
#endif
			return c;
		}

		void surf (Input IN, inout SurfaceOutput o) {
			// Albedo comes from a texture tinted by color
			fixed4 c = tex2D (_MainTex, IN.uv_MainTex) * _Color;
			o.Albedo = c.rgb;
			o.Alpha = 1.0f;
		}
		ENDCG
	}
	FallBack "Diffuse"
}
```

New property

New property

Phong light model

Lighting + light model name + _GI

Lighting + light model name

Physically Based Shading Theory

PBR Theory 1 - Refraction

Microfacet theory

Useful statistical approximation when facets are bigger than wavelength.

Diffuse is an approximation when the microfacets of a surface are orientated in different directions, specular is an approximation when they are orientated in roughly the same direction.

Refraction / Transmission

Until now, we have spoken about reflection, absorption, but not about refraction. Each material has a index of refraction, that represents how fast the light go through a material.

When light travels through a material it goes in straight line. If the index of refraction changes slightly, the light ray will bend. If it changes (because the material changes for instance), the light is scattered into several directions.

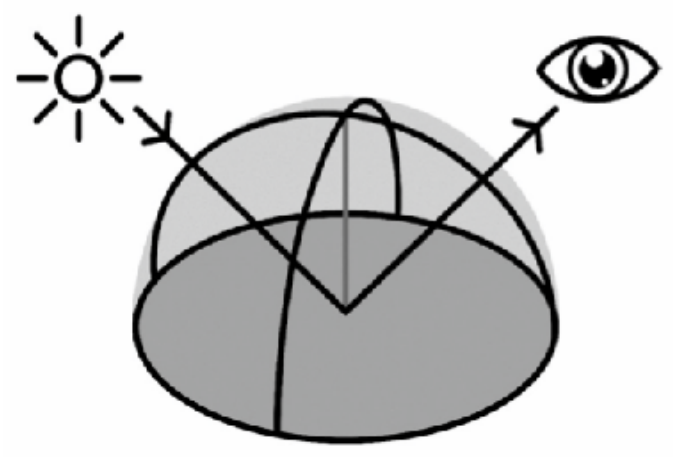
When a certain quantity of light hit a surface, a part is reflected, a part is absorbed and a part is refracted. Keeping track of how much go to each part is important.

Incoming light

To calculate how much light comes to a point of a surface, we consider the hemisphere around this point.

Within the rendering equation, we add all the incoming light coming through the hemisphere. This light can be direct or indirect. This is approximated with global illumination.

Sometimes you need to reflect the scene around. In real time simulations, this is approximated with reflection probes (e.g. in Unity or Unreal).



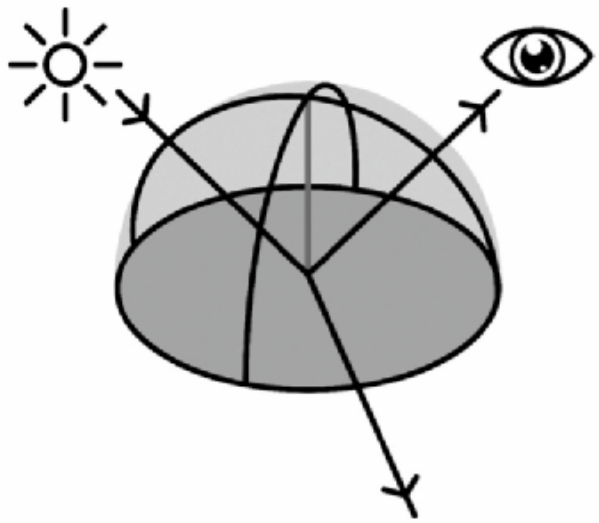
Refraction

When light changes material, it bends. When it changes from a faster to a slower material, it bends toward the normal.

Refraction is computed with the Snell's law. For our simulations, we only need to compute one light ray per pixel.

Considering refraction, we have to distinguish metal and non-metal (di-electric) materials. Metals absorb all the refracted light, that's why they have a lower diffuse component. Non-metal scatters light around, until it is absorbed, or until it exits away from where it enters.

This is the origin of the diffuse approximation. We can use diffuse when the light gets out from the pixel where is enters. If it gets out further away, we have to use subsurface scattering.



Distance to the camera is an important factor when you want to know if something should be shaded as diffuse or as subsurface scattering, because it determines the "size" of a pixel.

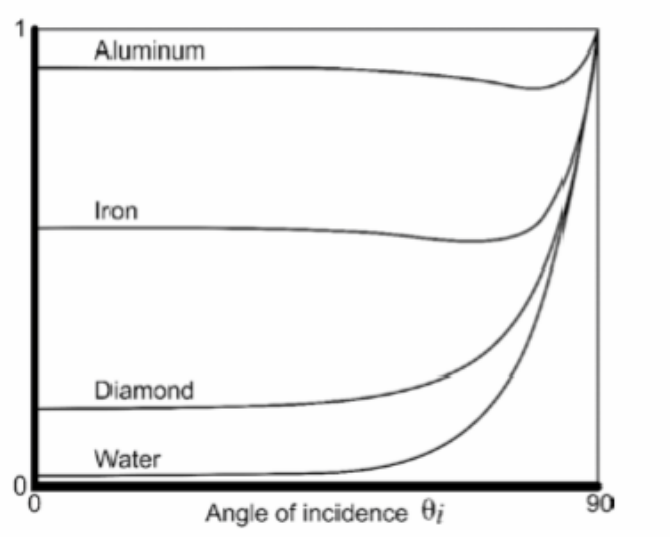
PBR Theory 2 - Resnel Reflectance

Fresnel Reflectance

Phong is unrealistic because the surface can send more light than it receives. The ratio of light reflected by the surface depends on the incidence angle with the normal, the view angle with the normal. Fresnel Reflectance gives the reflectance of the material in function of the view angle.

You can use the color of the material as the specular color. Specular colors are mostly necessary for metals. Dielectrics tend to have less precise, darker, and monochromatic specular colors. Since metals don't have any subsurface scattering (they absorb all the light that's not reflected) if they didn't have specular color, they wouldn't have any color.

We commonly calculate this split between reflection and refraction using the Fresnel equations, commonly implemented in code through the Schlick approximation. This approximation is supposed to behave similarly to those real physical world values in the reflectance graph.



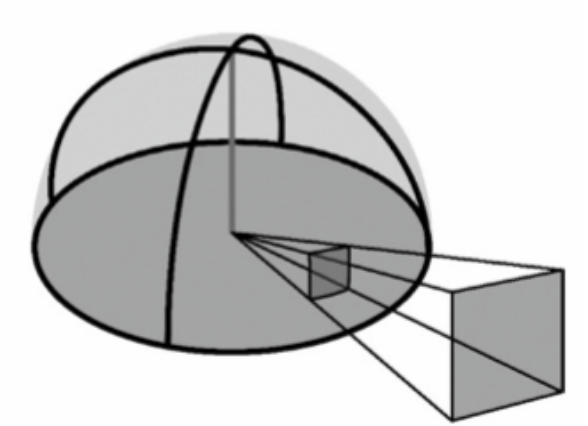
$$F_{schlick}(F_0,l,h) = F_0 + (1-F_0)(1-(l\cdot h))^5$$

Fo is the specular color, meaning the color at 0 degrees of the angle of incidence. The original Fresnel formula use instead of specular color, spectral indices of refraction, which are much less convenient to deal with.

Now, we will see how to mesure light, thanks to a physics field called radiometry.

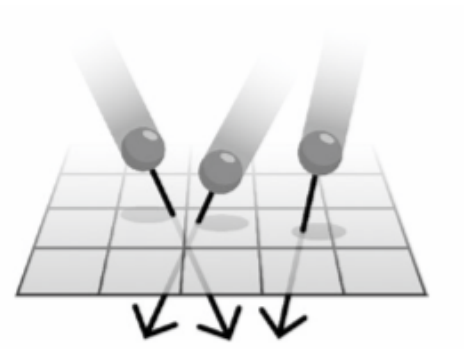
PBR Theory 3 - Radiometry

Solid angle



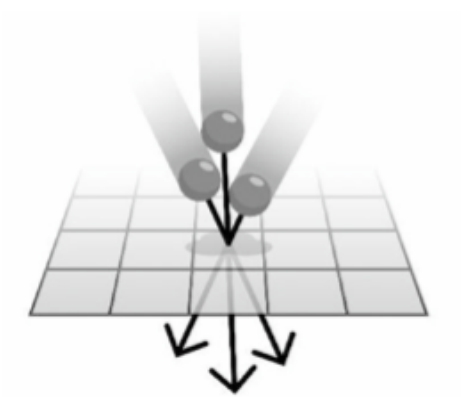
A solid angle is expressed in a unit called a steradian, for which the symbol is sr. It's the projection of a shape onto a unit sphere.

Power



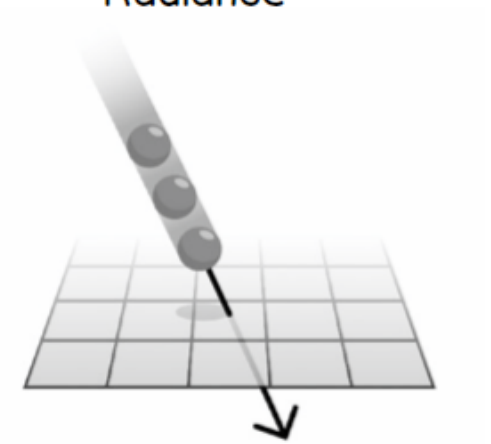
The rate of transmission of energy coming from any direction, and going through a surface, is power. You'll find power represented as W.

Irradiance



Irradiance is power coming from all directions, going through a point. In formulas, it's normally represented as E. You'll find it represented as W/m^2.

Radiance

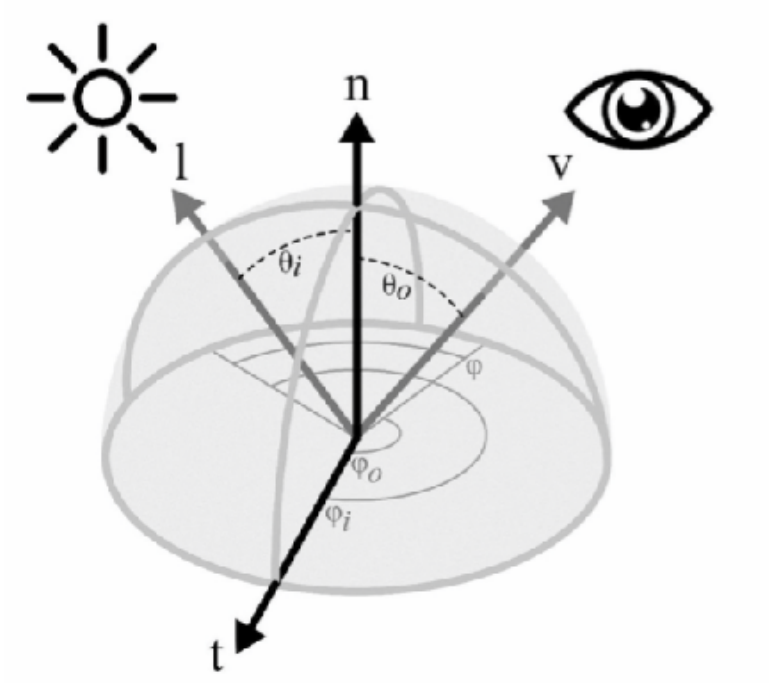


Radiance is the magnitude of light along a single ray. It is normally represented by L in formulas, with Li being the incoming radiance and L0 the outgoing radiance. You'll find it represented as W/(m^2.sr).

PBR Theory 4 - BRDF

For Bidirectional Reflectance Distribution Function. It describes how light is REFLECTED on a surface. Our previous lighting models were BRDF. A BRDF cannot represent subsurface scattering or translucency.

BRDF take as argument the angle of incident light and the view angle, in POLAR coordinates. (in the left schema, phi angle starts from the tangent vector t.



Original BRDF :

$$f_r(\omega_i,\omega_r) = \frac{dL_r(\omega_r)}{dE_i(\omega_i)} = \frac{dL_r(\omega_r)}{L_i(\omega_i)\cos\theta_i\omega_i}$$

L is radiance and E is irradiance. ωi and ωr are the incoming and outgoing (reflected) light directions. cosθi is the angle between the incoming light direction and the surface normal.

BRDF has 3 properties :

- Positivity : It means that the BRDF cannot return a negative number, there is no negative light.
- Reciprocity : The BRDF must have the same value when ωi and ωr are swapped.
- Energy Conservation : In the simplest terms, the outgoing light cannot exceed the incoming light, unless the excess of light has been emitted from the object.

There are other functions than BRDF, that can represent ss scattering or translucency.

- BSDF (Bidirectional Scattering Distribution Function)
- BSSRDF (Bidirectional scattering-surface reflectance distribution function)

They are not used in real time rendering, so we will see them in a future lesson !

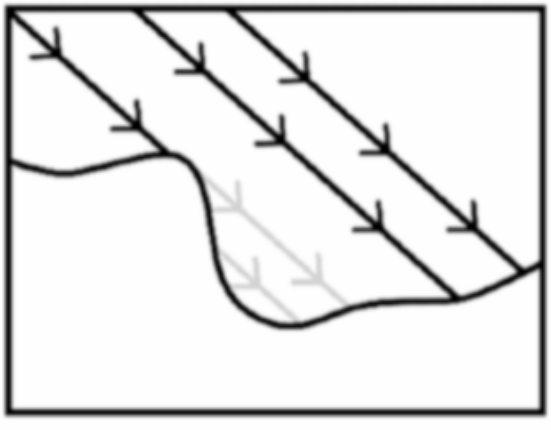
PBR Theory 5 - Microfacets return

Shadowing and masking

Microfacets are considered perfect mirrors : they reflect light without refraction. For this light to get into our viewpoint, these microfacets need to be oriented so that they reflect into the direction of view.

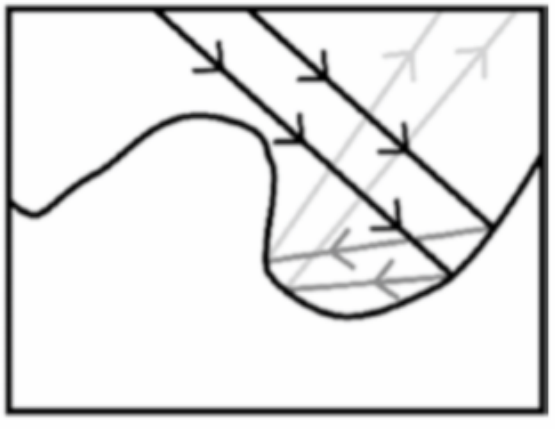
To achieve that the direction of their normal has to be the half vector. The half vector is halfway between the direction of the light and the direction of the view. The half vector is obtained by adding lightDir and viewDir, and then normalizing the result.

Shadowing and Masking forbid certain microfacets to contibute to light reflection. Shadowing is when irregularity creates a peak, which shadows some microfacets. Masking is when the reflection going out is blocked by a peak. In real world, this blocked light will continue to bounce, and might make it out of the surface eventually.



Shadowing

Masking



Real world

Microfacets specular BRDF

$$f(l,v)=\frac{F(l,h)G(l,v,h)D(h)}{4(n\cdot l)(n\cdot v)}$$

F(l,h) : Fresnel. This is the amount of light that's reflected rather than refracted, for a given substance, based on the light angle and normal.

D(h) : Normal Distribution Function. It tells us how many of the microfacets are pointing in the half vector direction, as those are the only ones we care about, because they contribute outgoing light. The distribution function determines the size and shape of the highlights. Some NDFs that you're likely to encounter are GGX, BlinnPhong, and Beckmann.

G(l,v,h) : Geometry Function. This tells us how many facets are not rendered useless by shadowing and masking. It is necessary for the BRDF to be energy conserving. Often, Smith's shadowing function is used as a geometry function.

Microfacet theory is very useful, but it can't simulate all phenomena. It ignores diffraction and interference, which depend on the nature of light as a wave.

PBR Theory 6 - the rendering equation

$$L_0(x,\omega_0) = L_e(x,\omega_0) + \int_\Omega f(x,\omega_i \rightarrow \omega_0) L_i(x,\omega_i)(\omega_i \cdot n) dw$$

$L_0(x,\omega_0)$ First, we have outgoing light. This is what we want to obtain from the equation.

$L_e(x,\omega_0)$ Then we have emitted light. Which is the light emitted by the surface we are shading. It's out of the BRDF, because it's not reflected or refracted light, and out of the integral, so it's only added once.

$\int_\Omega f(x,\omega_i \rightarrow \omega_0) L_i(x,\omega_i)(\omega_i \cdot n) dw$ Then we have the integral. We repeat and add up everything within it, for each solid angle in the hemisphere (dw).

$f(x,\omega_i \rightarrow \omega_0)$ Within the integral, we have the BRDF. It takes the incoming light ωi and returns the reflected and refracted light ω0.

$L_i(x,\omega_i)$ This is the incoming light to the point on the surface that we're shading.

$(\omega_i \cdot n)$ Then we have the normal attenuation which you know as n . l from earlier lighting model implementations.

The integral part of the rendering equation cannot yet be executed in real-time. As mentioned multiple times, indirect light is simulated with many techniques—spherical harmonics, global illumination, lightmapping, and image-based lighting (reflection probes, cubemaps, etc.).

One important point to know is that if you implement a new lighting model, you may need to change how global illumination is calculated and how cubemaps are processed. With reflection probes, a cubemap is captured from the scene, and to use image based lighting, the cubemap needs to be processed according to the BRDF that you're using. In practice, whatever processing is built-in within Unity may be sufficiently compatible for it not to be worth changing with a formulation for your BRDF.

PBR Theory 7 - Linear Color Space

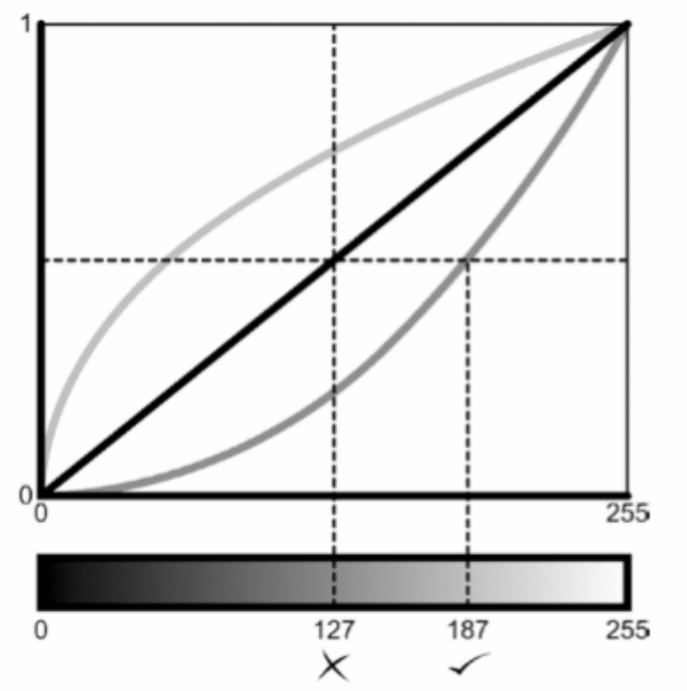
HDR / LDR

High Dynamic Range and Low Dynamic Range gives the ratio between the brightest and the darkest part of the scene.

You may convert your image to a LDR screen, with a tone mapping post processing effect. Choose well.

LDR and Linear Color Space

Colors on LDR screens are in Gamma space, which is non linear. The midpoint between 0 and 255 is 187, not 127. PBR uses Linear Space, so take care.



Making Phong Physically Based

Modified Phong

Phong

```
inline fixed4 LightingPhong (SurfaceOutput s, half3 viewDir, UnityGI gi)
{
    UnityLight light = gi.light;

    float nl = max(0.0f, dot(s.Normal, light.dir));
    float3 diffuseTerm = nl * s.Albedo.rgb * light.color;

    float3 reflectionDirection = reflect(-light.dir, s.Normal);
    float3 specularDot = max(0.0, dot(viewDir, reflectionDirection));
    float3 specular = pow(specularDot, _Shininess);
     float3 specularTerm = specular * _SpecColor.rgb * light.color.rgb;

    float3 finalColor = diffuseTerm.rgb + specularTerm;

    fixed4 c;
    c.rgb = finalColor;
    c.a = s.Alpha;
#ifdef UNITY_LIGHT_FUNCTION_APPLY_INDIRECT
      c.rgb += s.Albedo * gi.indirect.diffuse;
#endif
    return c;
}
```

PBR Criteria

Checking for Positivity

You can make sure that the BRDF output is always positive, with the little trick of putting a max function starting from 0 in front of the final color. Generally it's better if the BRDF mathematically guarantees that.

Checking for Reciprocity

Reciprocity is not easy to check from the shader code. In general, any BRDF published as physically based should have this property. Dig into the research papers on the BRDF if you want to check that, or alternatively if you're handy with programs such as Mathematica, or even with hand calculations, you can do the math and check for yourself. Phong is not reciprocal, so this needs some work.

Checking for Energy Conservation

PB Phong

Phong normalized

The work of making Phong physically based was done by Lafortune and Willems, in their paper from 1994. The normalization factor we want is:

(n + 2) / (2 * PI)

```
inline fixed4 LightingPhong (SurfaceOutput s, half3 viewDir, UnityGI gi)
{
    UnityLight light = gi.light;

    float nl = max(0.0f, dot(s.Normal, light.dir));
    float3 diffuseTerm = nl * s.Albedo.rgb * light.color;

    float norm = (_Shininess + 2) / (2 * 3.14159);
    float3 reflectionDirection = reflect(-light.dir, s.Normal);
    float3 specularDot = max(0.0, dot(viewDir, reflectionDirection));
    float3 specular = norm * pow(specularDot, _Shininess);
     float3 specularTerm = specular * _SpecColor.rgb * light.color.rgb;

    float3 finalColor = diffuseTerm.rgb + specularTerm;

    fixed4 c;
    c.rgb = finalColor;
    c.a = s.Alpha;
#ifdef UNITY_LIGHT_FUNCTION_APPLY_INDIRECT
      c.rgb += s.Albedo * gi.indirect.diffuse;
#endif
    return c;
}
```

We have implemented a physically based BRDF. It boiled down to multiplying the specular for the normalization factor.

The normalization factor prevents the BRDF from returning more light than it received in the first place.

What you are looking for is a reduction of specular brightness when the specular is spread out, and an increase of specular brightness when it's concentrated on a small surface. Since we have a pre-normalization version of Phong lying around, we can check that easily. As you can see, the original Phong keeps the same brightness, while the normalized Phong is less bright when the specular occupies a bigger area and brighter when it occupies a smaller one, as energy conservation should do.

Post processing

What is Post Processing ?

Definition

A post-processing effect is a basically an Image Effect shader, which is applied to every pixel in the current screen. Imagine you have rendered your scene, not to the screen but to a separate buffer, which in Unity is called a RenderTexture.

Now you can either send that to be visualized by the screen or manipulate it. To manipulate it, you can access it as a texture within the post-processing shader. This shader is executed separately from the other ones in the scene, because it needs to be executed when the scene has just been rendered. To trigger that, there is a function signature you can use in a script attached to the camera you want to apply the post effect to.

Post processing in PBR

The most important use of post effects for physically based shading is for tone mapping. You should always use an HDR camera for better realism, and then it would need to be tone-mapped back to LDR before being shown on an LDR screen. Another effect that does a lot for realism, especially for skin and other subsurfacebased materials, is depth of field.

For platforms where Unity doesn't support a linear color space, post effects are a great way to implement that with a minimum of fuss. Unity added support for a linear color space for mobile, but before, if you wanted to execute your calculations in linear space, you would need to first convert any color input to your shader (color picker, textures, etc.) to the linear space, do the shading calculations, and then convert back the entire rendered image to gamma as the last action in your post effects.

Setup

You need three things to set up a post effect:

- A camera in the scene to apply it to
- A script to add to that camera as a component
- A shader that the component will execute

HDR and Linear + Script Setup

HDR and Linear Setup

On your camera, check Allow HDR

In Build Settings / Player Settings / Other Settings, set the color space to Linear. The light will change.

Script Setup

Add a PostEffects script on the Camera.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

[RequireComponent(typeof(Camera))]
[ExecuteInEditMode]
public class PostEffects : MonoBehaviour
{

    public Shader currentShader;
    private Material currentMaterial;
    Material material
    {
        get
        {
            if (currentMaterial == null)
            {
                currentMaterial = new Material(currentShader);
                currentMaterial.hideFlags = HideFlags.HideAndDontSave;
            }
            return currentMaterial;
        }
    }

    void Start()
    {
        currentShader = Shader.Find("Hidden/PostEffects");
        GetComponent<Camera>().allowHDR = true;
        if (!SystemInfo.supportsImageEffects)
        {
            enabled = false;
            Debug.Log("not supported");
            return;
        }
        if (!currentShader && !currentShader.isSupported)
        {
            enabled = false;
            Debug.Log("not supported");
        }
        GetComponent<Camera>().depthTextureMode = DepthTextureMode.Depth;
    }

    void Update()
    {
        if (!GetComponent<Camera>().enabled)
        {
            return;
        }
    }

    void OnDisable()
    {
        if (currentMaterial)
        {
            DestroyImmediate(currentMaterial);
        }
    }
...
```

Auto load the shader

We force the camera to compute a depth texture. A depth texture is useful for many effects, such as depth of field.

Coding the Effect

Method

To apply our effects, we need another of those functions, OnRenderImage:

void OnRenderImage(RenderTexture source, RenderTexture destination)

As you can see, it takes two arguments, a source RenderTexture and a destination one.

You're supposed to put the code that actually applies the effect in this method. There is also another way, using two similar methods—void OnPreRender() and void OnPostRender(). In this case, you need to create the source render texture yourself, which depending on your platform, could be more efficient. We'll cover both methods. Let's start with OnRenderImage. There are a series of steps needed:

- Get the current rendered scene in a RenderTexture (RenderTexture source does that for us).
- Use Graphics.Blit to apply the image effect shader to the source texture. Blit means copying all pixels from an origin surface to a destination surface, while optionally applying some transformation.
- That also includes a destination RenderTexture. If that is null, Blit will send the result directly to the screen.

PostEffects shader

Create a Image Effect Shader named PostEffects. Let's study the default shader.

It declared Cull and Zwrite off, and ZTest Always. They are needed because we're not shading a model, we're processing a 2D image that's going to be shown on-screen as two triangles, forming a quad. We mustn't cull the back face, and Zwrite doesn't serve any purpose, because there is no depth.

The main texture is actually our rendered scene. Apply this shader in OnRenderImage and take a look at the result. We already have the source texture and a destination texture (which is the screen), so we only need a few lines of code to apply the effect.

```
void OnRenderImage(RenderTexture sourceTexture, RenderTexture destTexture)
{
    if (currentShader != null)
    {
        Graphics.Blit(sourceTexture, destTexture, material, 0);
    }
}
```

Graphics.Blit takes the source texture, the destination texture, the material that contains the shader to apply, and which pass to use, starting from 0.

This shader inverts the images colors.

Coding the Effect

Getting the depth texture of the camera

Copy and paste the Pass. Change the code of the Fragment shader.

```
name "DebugDepth"
Cull Off ZWrite Off ZTest Always

Pass
{
    ...
    fixed4 frag (v2f i) : SV_Target
    {
        fixed depth = UNITY_SAMPLE_DEPTH( tex2D(_CameraDepthTexture, i.uv) );
        fixed4 col = fixed4(depth,depth,depth, 1.0);
        return col;
    }
    ENDCG
}
```

To use this pass, change the Blit function :

```
Graphics.Blit(sourceTexture, destTexture, material, 1);
```

Now, let's add an option to switch between effects :

```
void OnRenderImage(RenderTexture sourceTexture, RenderTexture destTexture)
{
    if (InvertEffect)
    {
        Graphics.Blit(sourceTexture, destTexture, material, 0);
    }
    else if (DepthEffect)
    {
        Graphics.Blit(sourceTexture, destTexture, material, 1);
    }
    else
    {
        Graphics.Blit(sourceTexture, destTexture);
    }
}
```

Conversion to linear space

We used unity to convert from gamma to linear color space. We could also use a shader.

We would add another pass to the image effect shader. Within this pass, the _MainTex should be sampled withing a power to 2.2, then the invert is applied, then we get the color elevated back to a power of 1/2.2

```
fixed4 frag (v2f i) : SV_Target
{
    fixed4 col = pow(tex2D( _MainTex, i.uv ), 2.2);
    col = 1 - col;
    return pow(col, 1/2.2);
}
```

RenderTextures

Quick introduction

RenderTextures are textures you can write to. You can set one as the target of a camera.

You can create them from the UI, as assets, or you can create them programmatically. When you do create them programmatically, it's important to remember to release them. One way you can do that is by getting a temporary RenderTexture:

```
RenderTexture.GetTemporary(512,512,24,RenderTextureFormat.DefaultHDR);
```

After you use it, call:

```
RenderTexture.ReleaseTemporary(someRenderTex);
```

Multiple effects on RenderTextures

Up to now, we have only considered post effects that are not consecutive, as they don't use the result of the previous pass.

Some effects that use it are blur and depth of field. In that case, you need to use temporary textures to pass to the next shader pass as arguments. Nevertheless, you'll most likely use the Unity post-processing stacks (a ready made way to apply multiple effects on the same image), anyway, to do this kind of effect.

Here is an alternative way to use Post Processing Effects. This technique may be quicker, depending on your platform:

```
RenderTexture aRenderTex;

void OnPreRender() {
    aRenderTex = RenderTexture.GetTemporary(width,height,bitDepth, textureFormat);
    camera.targetTexture = myRenderTexture;
}

void OnPostRender() {
    camera.targetTexture = null;
    Graphics.Blit(aRenderTex,null as RenderTexture, material, passNumber);
    RenderTexture.ReleaseTemporary(aRenderTex);
}
```

Tone Mapping

Tone mapping is a way to gracefully convert an HDR buffer to an LDR buffer. The idea is to map the HDR values to LDR values in aesthetically pleasing ways, instead of just clipping them.

There are many tone-mapping operators that you can use, but we'll stick to a relatively simple one, the one that John Hable invented for Uncharted 2. First, we need to add a new boolean, a new if statement, and a new shader pass. We also need to add another property, an exposure for the camera.

To obtain a slider in the inspector, you should use this code:

```
[Range(1.0f, 10.0f)] public float ToneMapperExposure = 2.0f;
```

After declaring it, we want to send the value to the shader when the effect is active. For that we should use material.setFloat. We're going to put that within the if chain:

```
...
} else if (ToneMappingEffect) {
  material.SetFloat("_ToneMapperExposure", ToneMapperExposure);
  Graphics.Blit(sourceTexture, destTexture, material, 3);
} else {
...
```

Then we need to declare the property within the shader and declare the variable only within the pass where we're going to use it. Then, we implement Hable's operator in the fragment shader:

```
float _ToneMapperExposure;

float3 hableOperator(float3 col) {
   float A = 0.15;
   float B = 0.50;
   float C = 0.10;
   float D = 0.20;
   float E = 0.02;
   float F = 0.30;
   return ((col * (col * A + B * C) + D * E) / (col * (col * A + B) + D * F)) - E / F;
}

fixed4 frag (v2f i) : SV_Target {
   float4 col = tex2D(_MainTex, i.uv);
   float3 toneMapped = col * _ToneMapperExposure * 4;
   toneMapped = hableOperator(toneMapped) / hableOperator(11.2);
   return float4(toneMapped, 1.0);
}
```

At this point, if you check the boolean that activates the tone mapper pass, you should be able to change the exposure slider in the inspector and see it change the result. Turning on and off the tone mapper should show you how it influences the final result.

Now you have some idea about how tone mapping is implemented. There are various tone-mapping operators documented online, although you'll most likely going to want to use the powerful post-processing stack for this, which we are going to briefly introduce next.

The Post-Processing Stack

The post-processing stack is an über effect that combines a complete set of effects into a single post-processing pipeline. This has a few advantages:

- Effects are always configured in the correct order
- It allows a combination of many effects into a single pass
- All effects are grouped together in the UIfor a better user experience

The post-processing stack also includes a collection of monitors and debug views to help you set up your effects correctly and debug problems in the output.

To use post-processing, download the post-processing stack from the Asset Store.

Manual : https://docs.unity3d.com/Manual/PostProcessing-Stack.html

At this point, if you check the boolean that activates the tone mapper pass, you should be able to change the exposure slider in the inspector and see it change the result. Turning on and off the tone mapper should show you how it influences the final result.

Now you have some idea about how tone mapping is implemented. There are various tone-mapping operators documented online, although you'll most likely going to want to use the powerful post-processing stack for this, which we are going to briefly introduce next.

Implementing BRDF

If you want to go further

More fun with non PBR Shaders