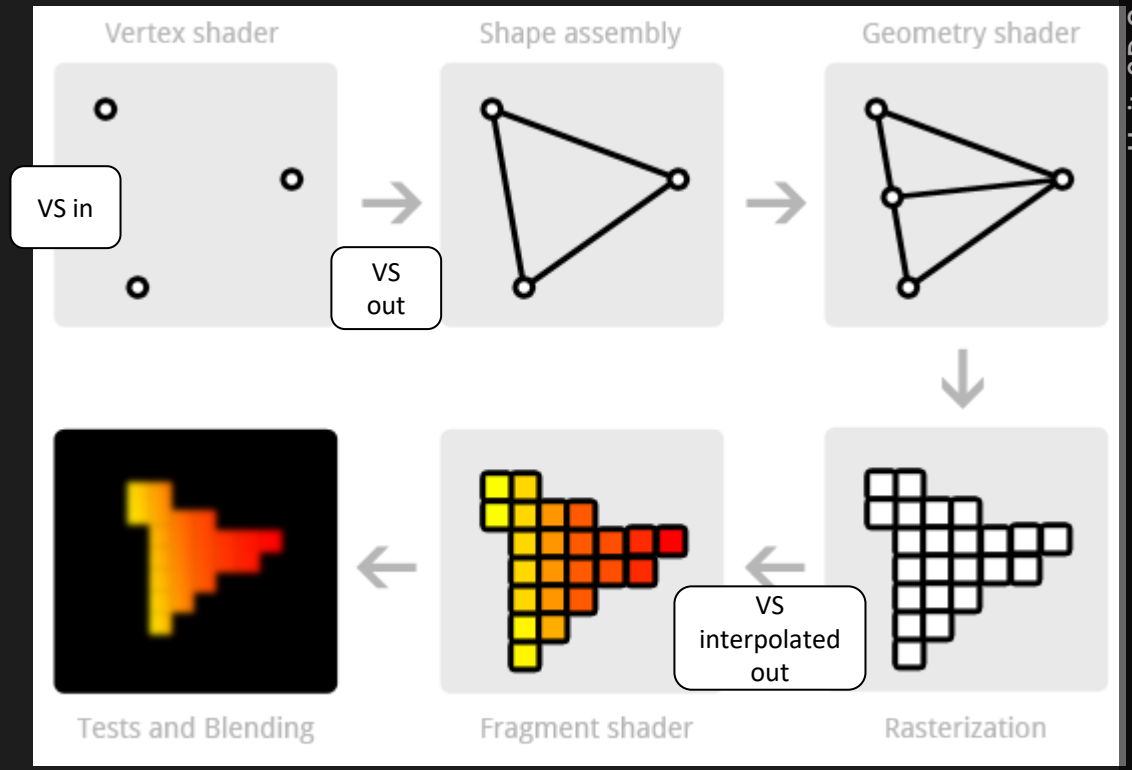
Shaders.. WT*?!

- To draw a mesh, we need primitives (Triangles, lines, dots)
- Shader types
  - Vertex
  - Pixel/Fragment
  - Geometry (primitive inputs: triangles, lines, dots)
  - Tessellation (OpenGL4 DX11)
  - Compute (General-purpose)

What happens if we want to draw a triangle?

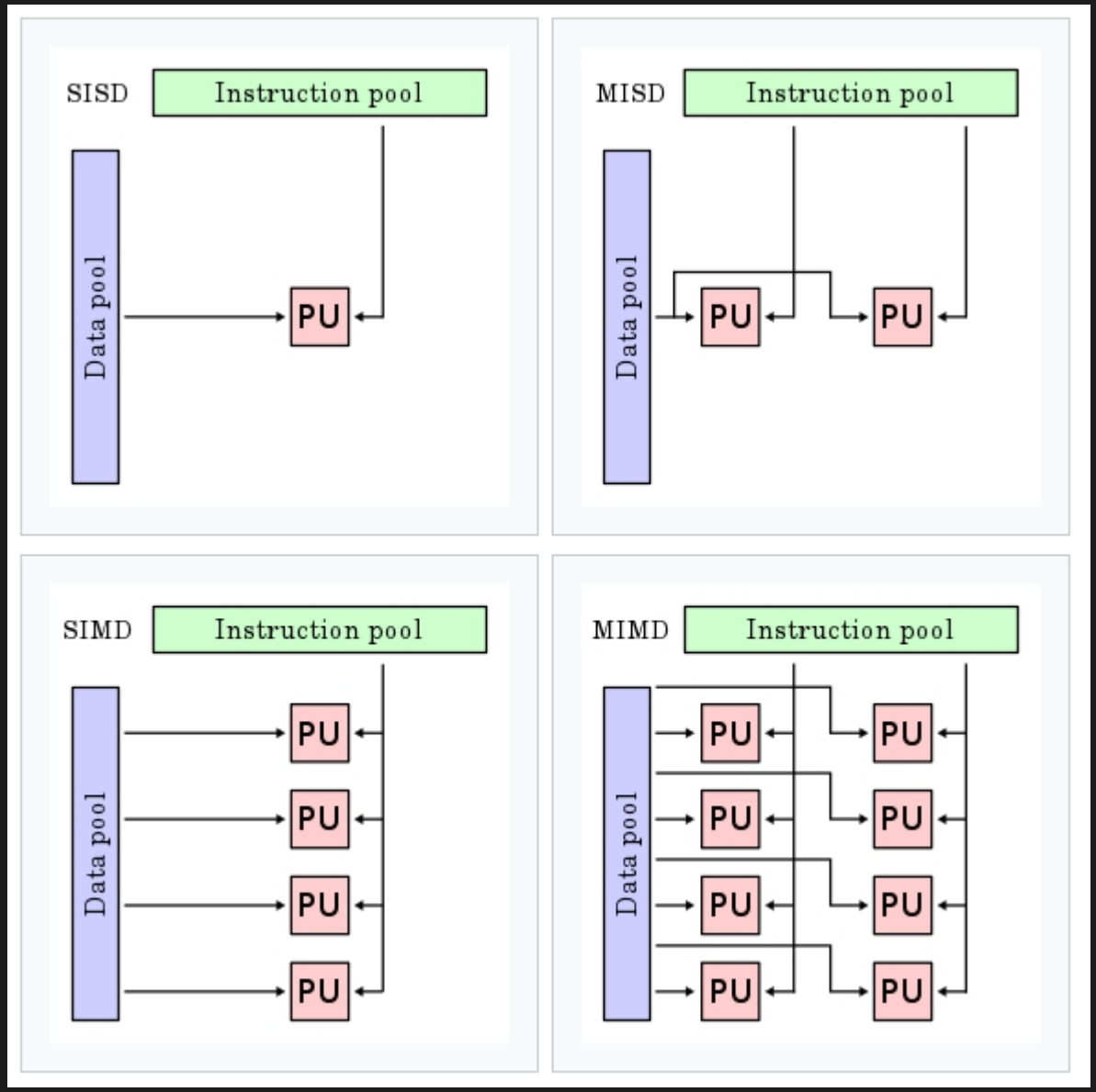
1. Send vertices one by one to VShader
   - Each Vertex has a number of info packed into a struct VSInput { position, normal, color, etc }
2. VShader passes an output struct VSOutput to the Rasterizer
   1. What are pixels covered by the primitive shape
   2. Interpolates VSOutput struct data for each pixel
3. Rasterizer passes these interpolated struct data to the PShader
4. PShader performs calculation on the pixel and ends up with a float4 value, the pixel color



Flynn Taxonomy

- GPU uses the SIMD paradigm: the same portion of code will be executed in parallel and applied to various elements of a data set
- SISD: older personal computers (PCs; by 2010, many PCs had multiple cores)
- SIMD: GPU
- MISD: generally used for fault tolerance. Heterogeneous systems operate on the same data stream and must agree on the result. Examples include the Space Shuttle flight control computer
- MIMD: MIMD architectures include multi-core superscalar processors, and distributed systems, using either one shared memory space or a distributed memory space

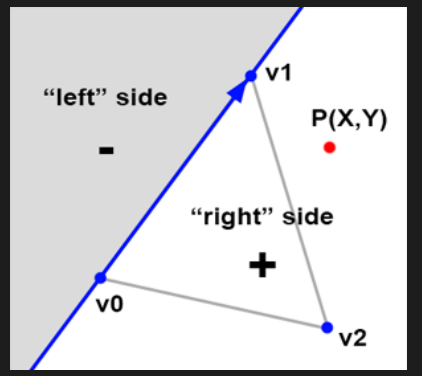
- Since the GPU has many different cores that run instructions for each vertex and pixel, they cannot communicate with each other, meaning every series of instructions can only handle the information for the vertex or pixel it has given

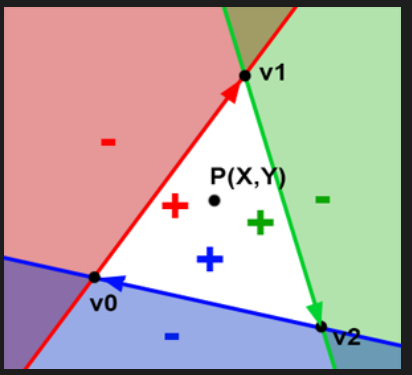


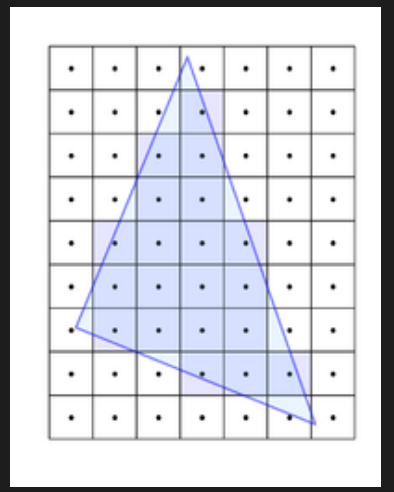
Rasterizer

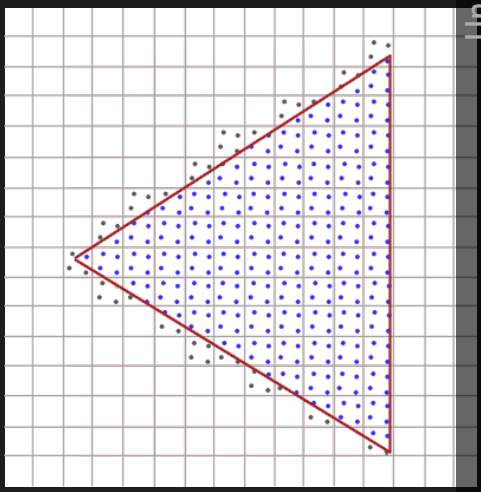
- How does the Rasterizer know what pixels will pass to the PShader?
  - Rasterizer input are triangles
  - Calculates the Geometric-area performing inside-out test (or coverage test)
    - Calculate edge function
  - Sample each pixel in the Geometric-area for AAliasing
  - NB: Even if Rasterizer sample N-times one fragment, PS executes only one time per fragment!

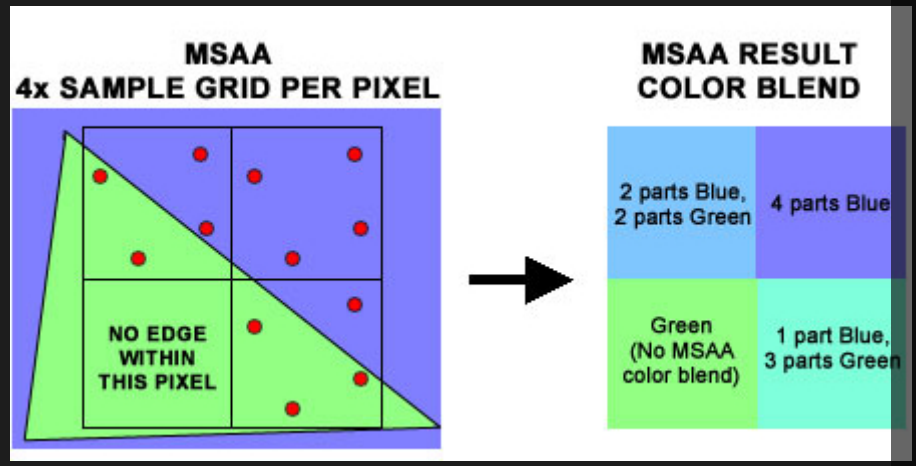
Eg. MSAA 4x (4 Samples)
- 2 samples inside, 2 samples outside triangle
  - PS final alpha will be multiplied by 2/4 (then will perform blending)
- 3 samples inside
  - PS final alpha will be multiplied by 3/4 (then will perform blending)
- 4 samples inside
  - PS final alpha will be preserved (then will perform blending)











Shader components

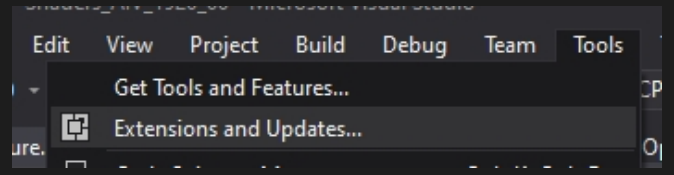
Types of files associated with shaders

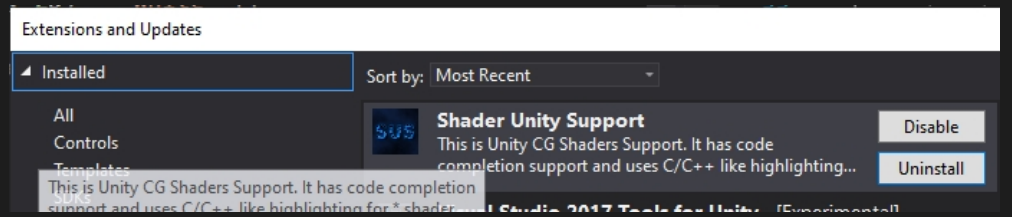
- .shader can compile in the different types of render pipelines
- .shadergraph can only compile in URP or HDRP
- .hlsl allow us to create customized functions; generally used within a node type called CustomFunction, found in Shader Graph
- .cginc it is a sort of include file for shaders (BIRP)

Shaders are written in Cg, HLSL (DirectX), GLSL (OpenGL)

- Unity shaders are written in Shaderlab + Cg or HLSL
- Shaderlab
  - Material Inspector properties
  - Blending Options
  - Render passes
  - Hardware fallback
- Cg/HLSL (within CGPROGRAM/ENDCG/HLSLPROGRAM/ENDHLSL snippets)
  - Surface shaders (BIRP)
    - Easy way, auto-generate Cg code for us after compiling
  - Vertex/Pixel shader

Shader coding syntax support



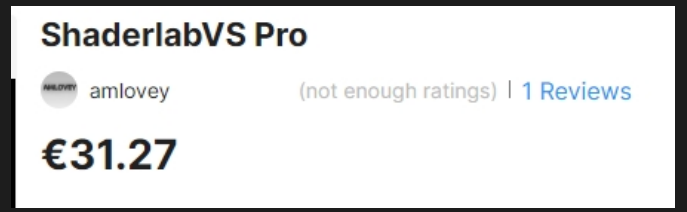


VS2017

- Go to Tools/ExtensionsAndUpdates
- Click on Online section in the left column
- Search for Shader Unity Support and install it
- After restarting VS, it should work

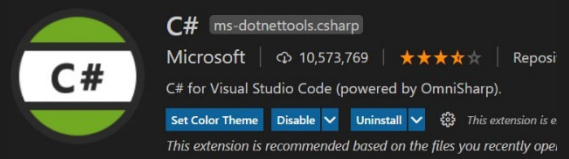
VS2019

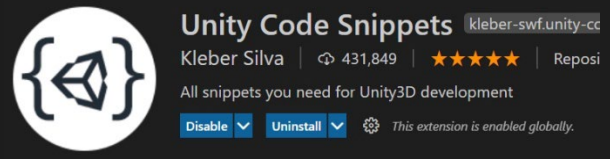
- Other resources: https://assetstore.unity.com/packages/tools/utilities/shaderlabvs-pro-186176?_ga=2.143044278.2086319534.1614242680-1515596986.1614242680&aid=1011lGoJ&utm_source=aff

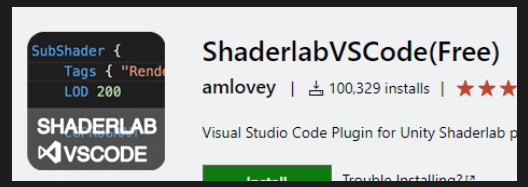


VSCODE

- https://marketplace.visualstudio.com/items?itemName=amlovey.shaderlabvscodefree#overview
- Press Shift+Alt+F to format the entire open shader. First time you do it, install the formatter

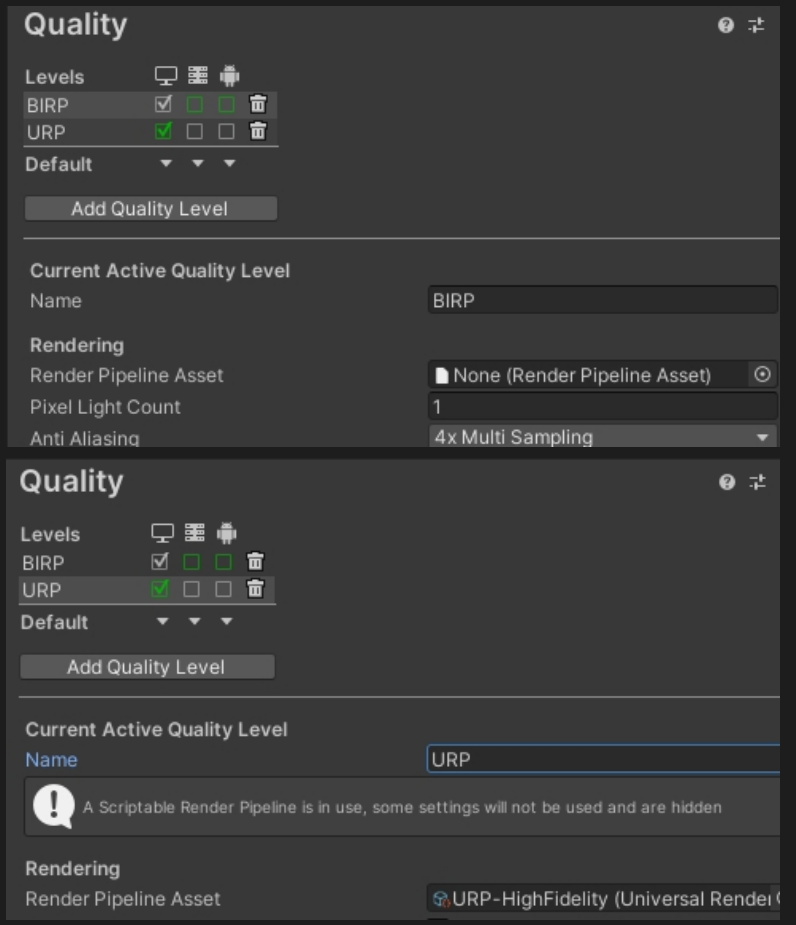






FAST BIRP/URP Switch

- In order to perform a fast switch between the 2 render pipelines, we'll use 2 configuration in ProjSettings/Quality
- BIRP will use no URP asset, URP will use a basic URPAsset
- Remove ProjSettings/Graphics ScriptableRenderPipelineSettings
- Remove
  - Static batching, Dynamic batching
  - [URP] SRP Batching, SSAO



Shader components

- Activate ShaderComponent and open _propertiesTest shader
- Test it in BIRP

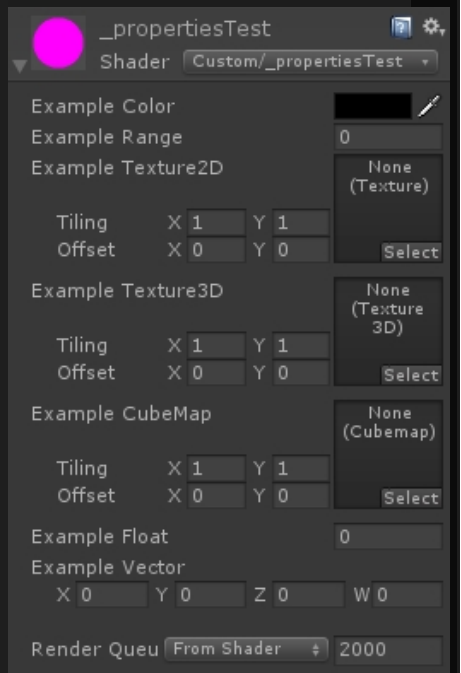
Main Shader Components

- Properties Public inspector variables
- Subshader Each shader may have more version, depending on the hardware supported features
- Pass Each subshader may perform multiple vertex/fragment passes. One pass is a DrawCall
- VS Input struct contains Vertex info like Position, Color, Normal
- Vertex Shader processes the vertexInput struct and fills the vertexOutput struct
- VS Output struct
- Fragment Shader its outputs are color and alpha of the fragment

- The sphere is lit by a flat shader, even if _propertiesTest shader is clearly incomplete. This is because #pragma only_renders ps4 says that this subshader (red squared in the img) has to be used iif playstation4 Graphics API is supported
- If not, the second SubShader (green squared in the img) is used: this is a void shader and the output is white
- If the second SubShader is not supported, the fallback shader is used (eg. Fallback "Mobile/Diffuse"). In this case there is no "fallbackShader", so if we comment out the green squared subshader, the shader compilation would notify an error
  - Fallback Off means: there is no Fallback shader

[Shaders_Start_A_01, _propertiesTest.shader]

```
Shader "Custom/_propertiesTest"
{
    Properties{
        _Color("Example Color", Color) = (1,1,1,1)
        _Range("Example Range", Range(0,1)) = 0.5
        _Tex2D("Example Texture2D", 2D) = "white" {}
        _Tex3D("Example Texture3D", 3D) = "white" {}
        _Cube("Example CubeMap", CUBE) = "" {}
        _Float("Example Float", Float) = 0.5
        _Vector("Example Vector", Vector) = (1,2,3,4)
    }

    Subshader
    {
        Pass
        {
            CGPROGRAM
            #pragma vertex vert
            #pragma fragment frag
            #pragma only_renderers ps4

            uniform sampler2D _Tex2D;
            uniform sampler3D _Tex3D;
            uniform samplerCUBE _Cube;

            struct vertexInput
            {
            };
            struct vertexOutput
            {
            };

            vertexOutput vert(vertexInput v)
            {
            }

            half4 frag(vertexOutput i): COLOR
            {
            }

            ENDCG
        }
    }

    Subshader
    {
        Pass
        {
        //...
        }
    }

    Fallback "fallbackShader"
}
```

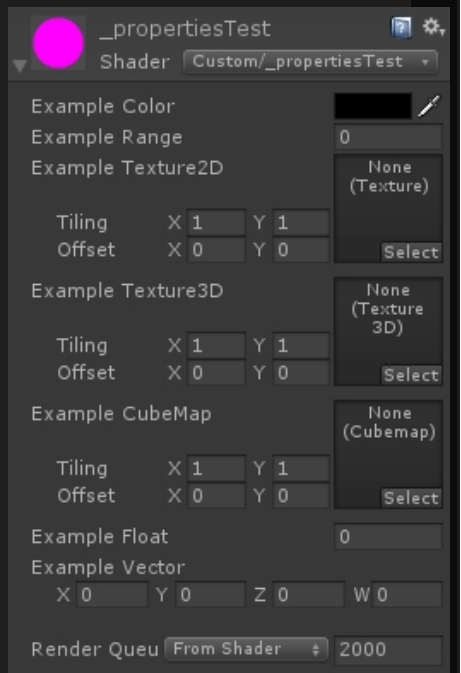


Shader components

- A structural factor that we must take into consideration is that the GPU will read the program from top to bottom linearly, therefore, if shortly we create a function and position it below the code block where it will be used, the GPU will not be able to read it, generating an error in the shader processing, therefore Fallback will assign a different shader so that graphics hardware can continue its process

[Shaders_Start_A_01, _propertiesTest.shader]

```
Shader "Custom/_propertiesTest"
{
    Properties{
        _Color("Example Color", Color) = (1,1,1,1)
        _Range("Example Range", Range(0,1)) = 0.5
        _Tex2D("Example Texture2D", 2D) = "white" {}
        _Tex3D("Example Texture3D", 3D) = "white" {}
        _Cube("Example CubeMap", CUBE) = "" {}
        _Float("Example Float", Float) = 0.5
        _Vector("Example Vector", Vector) = (1,2,3,4)
    }

    Subshader
    {
        Pass
        {
            CGPROGRAM
            #pragma vertex vert
            #pragma fragment frag
            #pragma only_renderers ps4

            uniform sampler2D _Tex2D;
            uniform sampler3D _Tex3D;
            uniform samplerCUBE _Cube;

            struct vertexInput
            {
            };
            struct vertexOutput
            {
            };

            vertexOutput vert(vertexInput v)
            {
            }

            half4 frag(vertexOutput i): COLOR
            {
            }

            ENDCG
        }
    }

    Subshader
    {
        Pass
        {
        //...
        }
    }

    Fallback "fallbackShader"
}
```



Test it

Try it
- Switch to BIRP

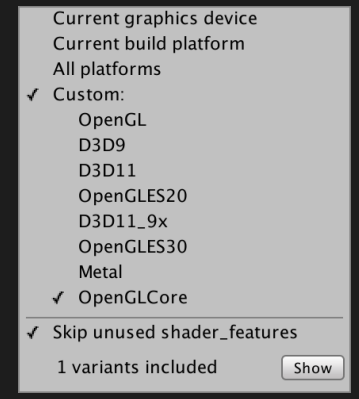
Pass
- Pass Each subshader may perform multiple vertex/fragment passes. One pass is a DrawCall
  - Duplicate the pass section the second subshader in _propertiesTest shader
  - See what happens in the FrameDebug: 2 drawcalls
  - [UPR] If you try this in URP, it recognizes that 2 passes are empty, and it performs only 1 pass

Fallback
- Remove the second Subshader
- Add "Mobile/Diffuse" Fallback
- Add a directional light
- Now properties in the inspector are from Shaderlab, but the shader is Mobile/Diffuse
- Change texture property name into _MainTex
- Now if you set a 2D Texture, the Mobile/Diffuse shader will use it!

[Shaders_Start, _propertiesTest.shader]

Target platform

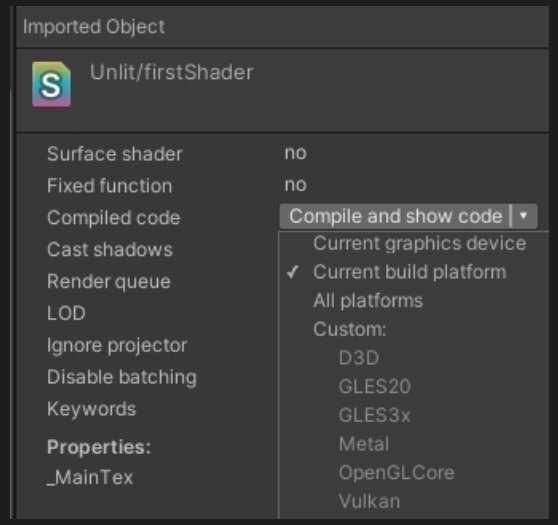
- You can select which platforms you manually compile the shader for, via the dropdown menu. The default is to compile for the graphics device that's used by your editor. You can manually compile for other platforms as well, either your current build platform, all platforms you have licenses for, or a custom selection
- This enables you to quickly make sure that your shader compiles on multiple platforms, without having to make complete builds



First simple shader

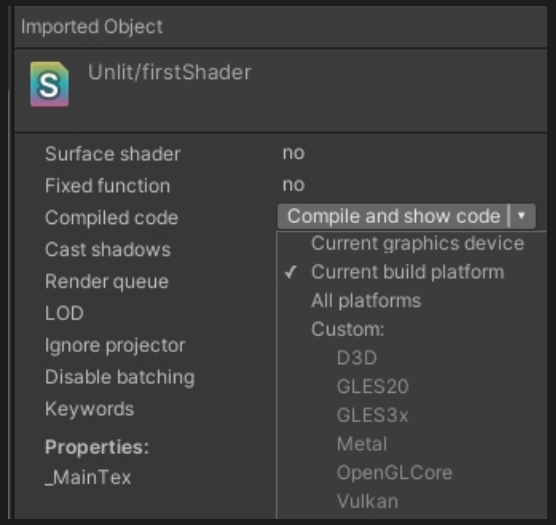
- Activate FirstShader_Start. The sphere has a defaultMat, and is lit only by the ambient color
- Create a new UnlitShader, name it firstShader
- Create a new Material starting from firstShader (Rclick on it/New material), name it firstShader
- Assign firstShader material to the sphere

[_firstShader.shader]




```
Shader "Custom/firstShader"
{
    Properties
    {
    }
    SubShader
    {
        Pass
        {
        }
    }
}
```

First simple shader




```
Shader "Custom/firstShader"
{
    Properties
    {
    }
    SubShader
    {
        Pass
        {
        }
    }
}
```

- Remove all shader content, leave only Shader/Properties/Subshader/Pass structure
- Add a Color Property. Each property has this syntax:
  - variable_name (label, data_type) default_value
  - _Color("Main color", Color) = (1,1,1,1)
- When declaring a property it is very important to consider that it will be written in ShaderLab declarative language while our program will be written in either Cg or HLSL language. As they are two different languages, we have to create “connection variables”. These variables are declared globally using the word uniform, however, this step (write uniform) can be skipped, because the program recognizes them as global variables. So, to add a property we must first declare the property in ShaderLab, then the global variable using the same name in Cg or HLSL, and then we can finally use it

[BIRP] We are going to write a Subshader with a single Pass

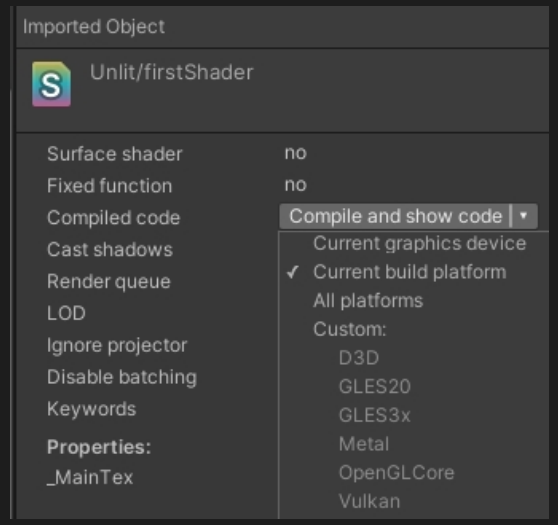
- Add CGPROGRAM and ENDCG inside the pass
- Define a vertexInput structure. Each vertex attribute has this syntax:
  - dataType varName : SemanticLabel a semantic is a chain connected to a shader input or output that transmits usage information of the intended use of a parameter
  - float4 vertex : POSITION
  - float4 is the same as Vector4 in Unity
  - Without POSITION semantic, Unity doesn’t know what information write inside that float4 vertex variable.
- Define a vertexOutput structure
  - float4 pos : SV_POSITION
    - SV_POSITION is the semantic for the final clip space position of a vertex, so that the GPU knows where on the screen to rasterize it, and at what depth. When fragment shader receives it in input, its value is transformed in ScreenSpace (screen resolution)
    - pos, not vertex, because this position will be interpolated: there will be N vertexOutput structures that the pixelShader will receive in input, depending on the fragments selected by the Rasterizer

[_firstShader.shader]

First simple shader

- Declare vertex and pixel shader names:
  - #pragma vertex VS_Name
  - #pragma fragment FS_Name
  - The word pragma comes from Greek and refers to an action, or something that needs to be done. It's used in many programming languages to issue special compiler directives
- We need another pragma: the final target. By default, Unity compiles shaders into almost the lowest supported target (2.5). Target depends on the hardware support for the shader functions. Eg.
  - Target 2.5 has 8 interpolators
  - Target 3.0 has 10 interpolators, texture LOD sampling, more math/textures instructions allowed
  - #pragma target 3.0
- Write vertex and fragment shader method declaration
- Fragment has
  - Semantic SV_Target specified directly in the function. This is the semantic used by DX10+ for fragment shader color output, its more compatible than COLOR semantic (which works only for DIRECTX9)
  - half4 instead of float4 data type
  - It is also possible to return a structure with the outputs (see the img on the right)
- Write v and f shader method body
  - Start outputting a constant Red color from the f shader
  - See if it works in Unity

[_firstShader.shader]




```
Shader "Custom/firstShader"
{
    Properties
    {
    }
    SubShader
    {
        Pass
        {
        }
    }
}
```

```
struct fragOutput {
    fixed4 color : SV_Target;
};
fragOutput frag (v2f i)
{
    fragOutput o;
    o.color = fixed4(i.uv, 0, 0);
    return o;
}
```

First simple shader

- Back to VS, you'll see that Unity automatically replaced 'mul(UNITY_MATRIX_MVP,*)' with 'UnityObjectToClipPos(*)' for optimization purpose. This function is defined in UnityShaderUtilities.cginc file which has been included as a dependency in UnityCG.cginc, which is automatically included in every shader
- Change f shader output using _Color Property. To use a property, we need to declare it as uniform variable. This means that
  - a variable has the same value for all vertices and fragments of a mesh. So it is uniform across all its vertices and fragments
  - We can modify it via script, using Material.SetFloat("_Color", newcolor);
    - Enable changeColor.cs on SphereFirstShader

[Shaders_Start, _firstShader.shader]

Test the fallback shader

- Add
  `#pragma only_renderers ps4`
- At the end of your shader, add
  `Fallback "Unlit/Texture“ //Or Fallback "Unlit/Color"`
- Now your shader is using the fallback shader!

```
            ENDCG
        }
    }
    Fallback "Unlit/Texture"
}
```

[Shaders_Start, _firstShader.shader]

URP First simple shader

- In order to have full URP compatibility, we have to switch to HLSL

- Use
  HLSLPROGRAM/ENDHLSL
  Instead of
  CGPROGRAM/ENDCG
- Add
  `#include "Packages/com.unity.render-pipelines.universal/ShaderLibrary/Core.hlsl"`
  - Unlike UnityCg.cginc which comes included in the Unity installation as such, Core.hlsl is added as a package once we install URP in our project, therefore, we will have to write the full path of its location in the project
- Fragment return type should be half4 or float4
  if fixed4, you must add `#include "HLSLSupport.cginc"`
- Add
  `Tags {"RenderPipeline"="UniversalRenderPipeline"}`
- Use
  `o.pos = TransformObjectToHClip(v.vertex); //included in the dependency “SpaceTransforms.hlsl”, which has been included in “Core.hlsl”`
  instead of
  `o.pos = UnityObjectToClipPos(v.vertex);`

[Shaders_Start, _firstShader.shader]

Basic Data Types

- float - Highest precision / 32 bit
  - World positions, Texture coords, scalar calculations involving complex functions such as trigonometry or exponentiation
- half – Half sized float / 16 bit
  - low magnitude vectors, object-space positions, Directions (normalized vectors), HDR HighDynamicRange colors
- [BIRP, Cg] fixed – Lowest precision / 11 bit
  - Regular colors, Color operations, simple operations
- int – Counter, Array indices

Why don't use only Floats? In practice this is possible, however, we must consider that float is a high-precision data type, which means that it has more decimals, therefore, the GPU will take longer to calculate it, increasing times and generating heat

They can be used in Packed Arrays (int2, float3, half4, fixed2)

- To access single Packed Arrays values, use .rgba or .xyzw
  - float4 mainColor;
  - mainColor.x == mainColor.r
  - mainColor.rb = float2(1,2);

- Swizzling
  - float2 vec2 = float2(1,2)
  - float3 vec3 = vec2.xxy

Packed Matrices

- float4x4 M;
- To access single matrix value, use M._m[row][col]
  - float val = M._m02;

Texture datatype

- Sampler2D - regular images
- SamplerCUBE - cubemaps

Coordinate System

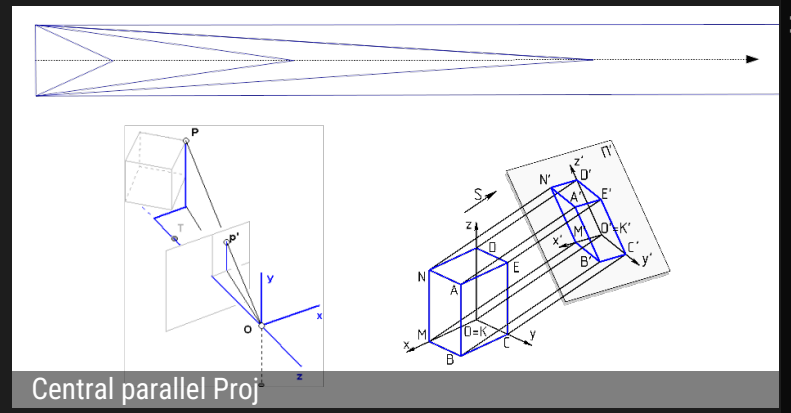
- Coordinate System
  - Cartesian
    - This system is insufficient when we have to deal with a projective space: 3D points projected onto 2D surfaces. For example, 2 parallel lines in a cartesian system doesn't intersect each other. In a Projective space, they intersect at an infinite distance
  - Homogeneous

Homogeneous

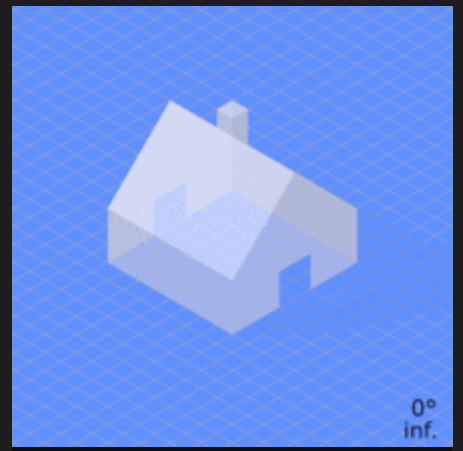
- Represent N-dimensional coordinates with N+1 numbers
- (X,Y) => (x,y,w), X = x/w, Y = y/w
- (2,3) => (2,3,1)
- Why "homogeneous"?
  - (1,2,3), (2,4,6) and (4,8,12) => (1/3, 2/3)
  - Scale invariant
- Translation toward infinity
  - (2,2,2) => (1,1)
  - (2,2,4) => (1/2,1/2)
  - (2,2,1/2) => (4,4)
  - (2,2,0) => (∞,∞)

Advantages

- We can express infinity value. Eg. Dir light position
- Central & Parallel projection are the same
- Better float precision
  - Float range is huge (32 bit), but has a 23 bit precision, which means about 8M. If we want a millimeter precision, 8M millimeters equals to an area of 8Km: good for FPS, not for Space simulations



Central parallel Proj



Central to parallel Proj moving vanishing point toward ∞

Transformation Matrices

- Scale, Rotation, Translation equations
- If I want to scale and then rotate a point, I should apply scale equation and then a rotation equation. There is a simpler way: Matrix calculation!
- Matrix and vectors multiplication formula
- Scale, Rotation, Translation Matrices
  - Translation matrix has to have 4 elements: to multiply also scalar and rotation matrices by translation matrix, we use 4x4 matrices also for scalar and rotation
  - Associativity property
- Order matters (no commutative operations): To scale and THEN rotate a point P:
  - P' = R x S x P

```
x * sx = x'
y * sy = y'
z * sz = z'
```

```
x * cos(angle) + y * -sin(angle) = x'
x * sin(angle) + y *  cos(angle) = y'
                                 z = z'
```

```
x + tx = x'
y + ty = y'
z + tz = z'
```

$$\begin{bmatrix} f1 & f2 & f3 \\ f4 & f5 & f6 \\ f7 & f8 & f9 \end{bmatrix} * \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} x' \\ y' \\ z' \end{bmatrix}$$

```
f1*x + f2*y + f3*z = x'
f4*x + f5*y + f6*z = y'
f7*x + f8*y + f9*z = z'
```

$$\begin{bmatrix} sx & 0 & 0 \\ 0 & sy & 0 \\ 0 & 0 & sz \end{bmatrix} * \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} x' \\ y' \\ z' \end{bmatrix}$$

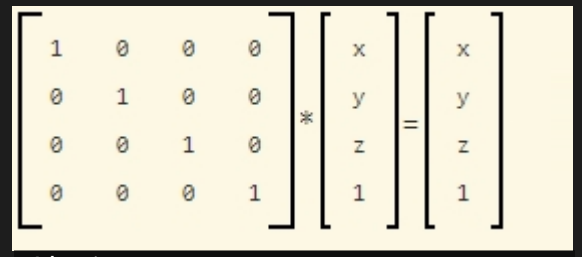
$$\begin{bmatrix} \cos(angle) & -\sin(angle) & 0 \\ \sin(angle) & \cos(angle) & 0 \\ 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} x' \\ y' \\ z' \end{bmatrix}$$

$$\begin{bmatrix} 1 & 0 & 0 & tx \\ 0 & 1 & 0 & ty \\ 0 & 0 & 1 & tz \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix}$$

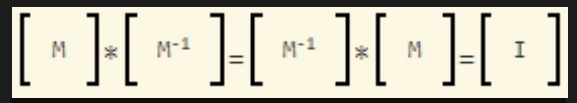
Transformation Matrices

- Identity (I)
  - I x P = P
- Inverse (M^{-1})
  - is the undo of matrix multiplications
  - if P' = M x P, then P = M^{-1} x P'
  - M x M^{-1} = M^{-1} x M = I
  - not always exist, but if M is obtained via a concatenation of Scale Translation Rotation, M^{-1} exists
    - The inverse matrix of local2World is world2Local
- Transpose (M^T)
  - M^T has M columns and rows inverted
  - M x P = P x M^T (see next slide)

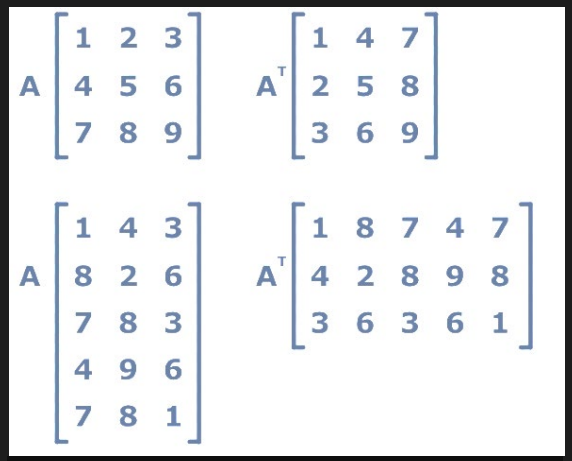
- In computer graphic, w value is often 0 or 1. General rule:
  - w = 0 for vectors, directions, directional lights
  - w = 1 for positions (vertices, other lights)
  - For normals, w is not a prefixed value (it depends on binormal value)



Identity



Inverse



Transpose

Images from Wikipedia

Unity3D Course – Luca Palmili

Shader matrices vector multiplication

Transforming Points

```
float4x4 matrix;
float4 point;
float4 transformed_point = mul(matrix, point);
```

If a transposed matrix is given

```
float4x4 matrix_transpose;
float4 point;
float4 transformed_point = mul(point, matrix_transpose);
```

Transforming directions

```
float3x3 matrix;
float3 direction;
float3 transformed_direction = mul(matrix, direction);
```

If the direction is a float3 vector and we have a 4x4 matrix

```
float4x4 matrix;
float3 direction;
float4 transformed_direction = mul(matrix, float4(direction, 0.0));
```

If the direction is a float4 vector (w=0) and we have a 3x3 matrix

```
float3x3 matrix;
float4 direction; // 4th component is 0
float4 transformed_direction = float4(mul(matrix, direction.xyz), 0.0);
```

Build a Matrix

- To build a NxN Matrix

```
float3x3 m = float3x3(
  1.1, 1.2, 1.3, // first row
  2.1, 2.2, 2.3, // second row
  3.1, 3.2, 3.3  // third row
);
```

Shader matrices vector multiplication

Transforming Normal Vectors. We need

- the inverse transpose matrix
- to preserve normal.w after multiplication
- to normalize again the multiplication result

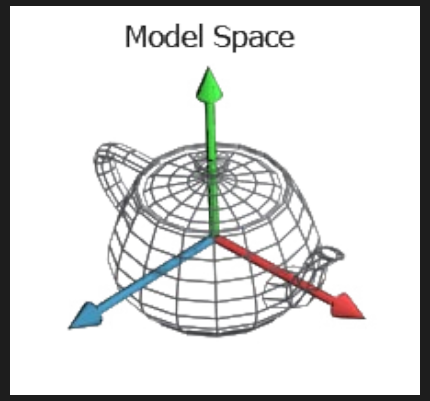
```
float3x3 matrix_inverse_transpose;
float4 normal;
float4 transformed_normal =
        normalize(float4(mul(matrix_inverse_transpose, normal.xyz),normal.w));
```

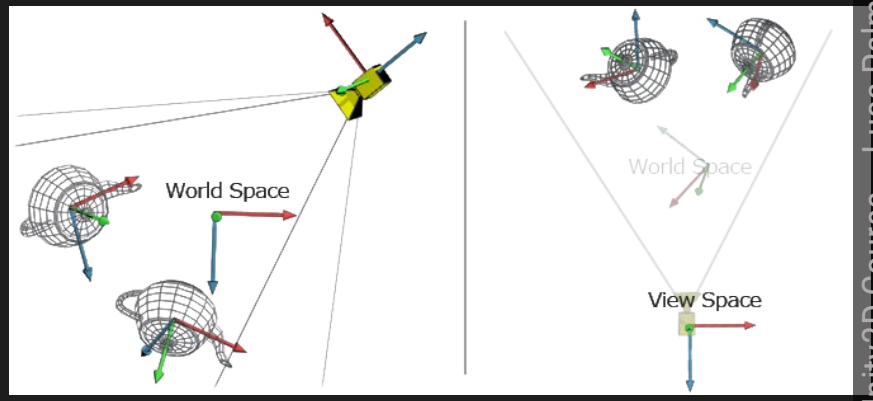
Let's say we want WorldNormal WN starting from ObjectNormal ON. If WN were a point, we would need Obj2World Matrix. WN is a normal => we need the inverse transpose of Obj2World Matrix. We don't have it: in these cases, there is a workaround: we'll use the inverse of Obj2World Matrix, which is World2Obj

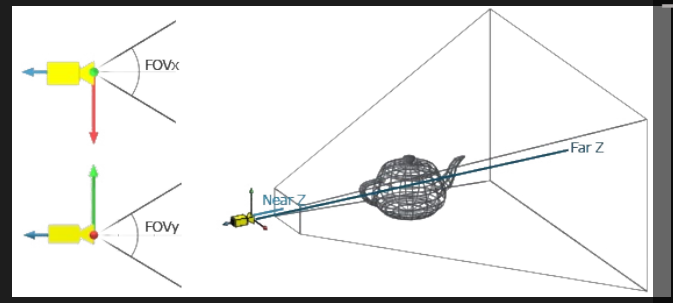
If we don't have the inverse transpose but we have the inverse, just invert the multiplication order between Normal and the Matrix:

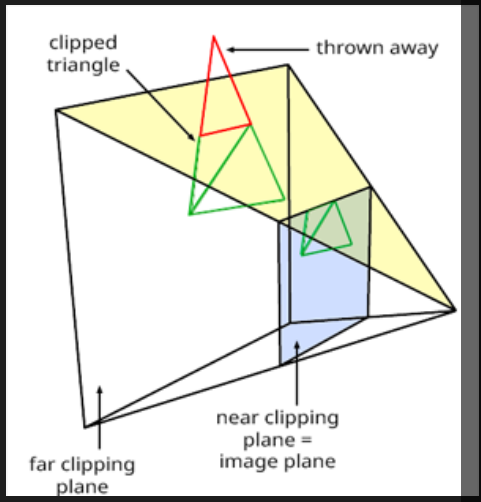
```
float3x3 matrix_inverse;
float4 normal;
float4 transformed_normal =
        normalize(float4(mul(normal.xyz, matrix_inverse), normal.w));
```

Coordinate Spaces









- Within a Coordinate System, we can have different origins => different Coordinate Spaces

Object space (origin: obj center)
- This is the C.Space provided to the VS

⇒ Multiply by ModelMatrix
  - C# transform.localToWorldMatrix
  - Shader built-in var unity_ObjectToWorld

World space (o: world center)

⇒ Multiply by ViewMatrix
  - C# camera.worldToCameraMatrix
  - Shader built-in var UNITY_MATRIX_V

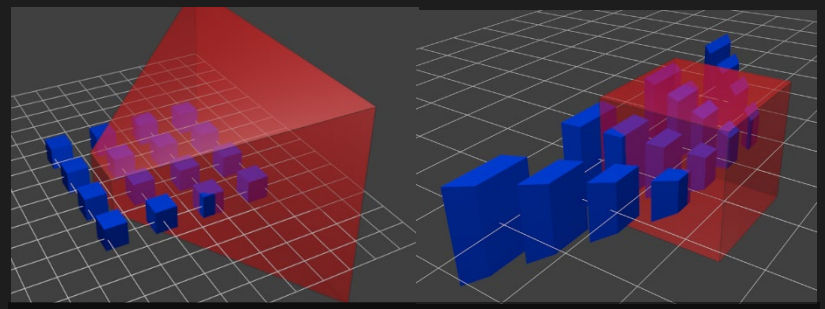
Camera/View/Eye space (o: camera center)

⇒ Multiply by ProjectionMatrix
  - C# camera.projectionMatrix
  - Shader built-in var UNITY_MATRIX_P
    - NB: If the point is 10 units away from the camera along z positive in Unity, its value will be -10 (camera z-axis is inverted respect the Unity-World)
    - Try it changing Camera Far plane
      - 10: ProjPosition is (0,0,10,10) [not clipped yet]
      - 9: ProjPosition is (0,0,10.1,10) [clipped!]

Projection/Clip space (inside View Frustom)

[CoordSpaces.cs]

Coordinate Spaces



Perspective divide

Projection/Clip space (inside View Frustom)

⇒ CLIPPING

- If at least one of (x,y,z) values are outside (-w,w) value, then discard vertex

⇒ Perspective Divide

- Divide (x,y,z) by Clip.w

NormalizedDeviceCoordinate/NDC space

- Inside [-1,1] cube

⇒ Remap [-1,1] range into [0,1]

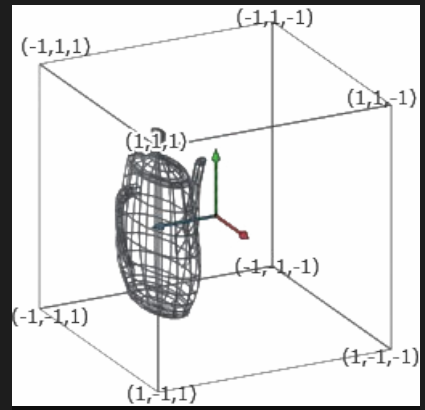
Texture space

⇒ Viewport transform

- Remap [0,1] range into ScreenSpace coords

Screen space

⇒ RASTERIZATION



NDC Space

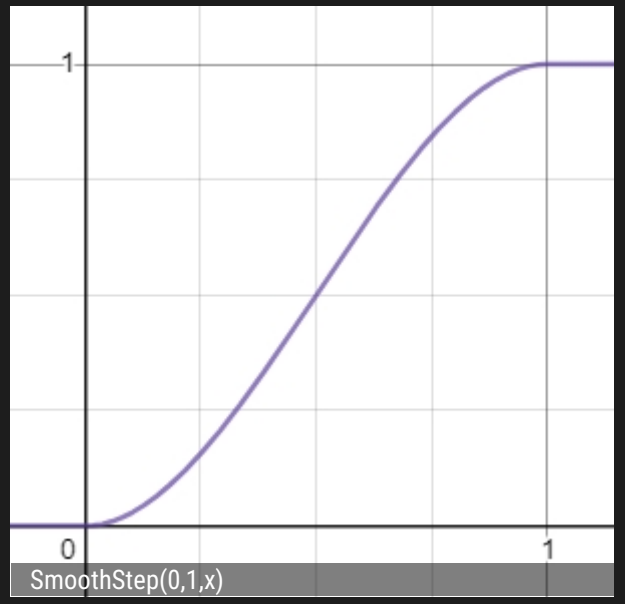
CoordSpaces.cs

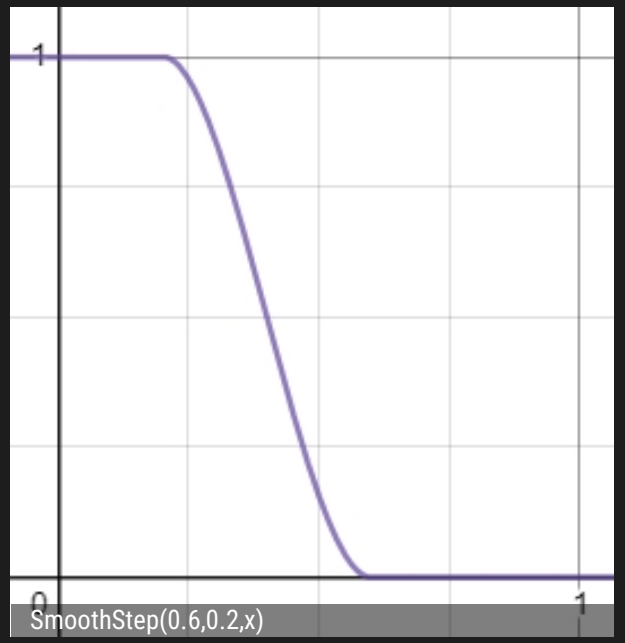
- (See also in-line script comments) To test the position of our transform in Worldspace, ViewSpace, ProjectionSpace, etc.. we start from ObjectSpace. The vertex we are going to transform is the object center, which has (0,0,0) vertex coords. For our first transformation we are multiplying a WorldSpaceMatrix for a column vector (0,0,0,1). This is why we can look directly at WS_m.m03, WS_m.m13, WS_m.m23) values to know the vertex position in WorldSpace

[CoordSpaces.cs]

Smoothstep

- Perform Hermite interpolation between two values
  - SmoothStep(minValx, maxValx) //minVal is 0, maxVal is 1
- Try it in https://www.desmos.com/calculator/xykhidbkbg





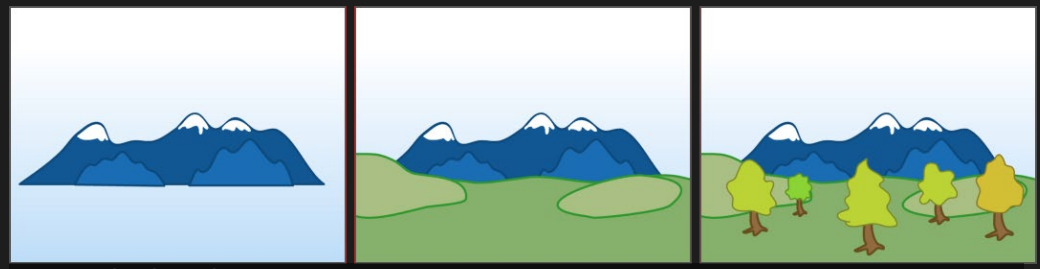
Rendering order

- Setup
  - Forward rendering
  - Camera HDR and MSAA OFF
  - Disable Dynamic & Static Batching (ProjSettings/Player)
  - Switch to BIRP
- Unity uses
  - Reverse Painter's Algorithm for Opaque Objects
  - Painter's algorithm for Transparent Objects
    - Front transparent objects need to calculate the correct blending with the transparent objects behind them. In case of reverse painter algorithm, this would be impossible to achieve: blending equation is order dependent
  - A special data structure RenderQueue, in which we can override this default rendering order
  - Stencil buffer techniques are based on rendering order values

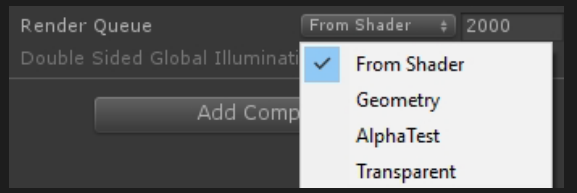
- RenderQueue (subshader scope tag, see later) allows to override Rendering order
  - Can be changed via Inspector or inside Shader code
  - Lower value are drawn first
  - Queue = "Geometry" = 2000
  - Queue = "Geometry+100" = 2100
  - Queue = 2900 = Transparent-100

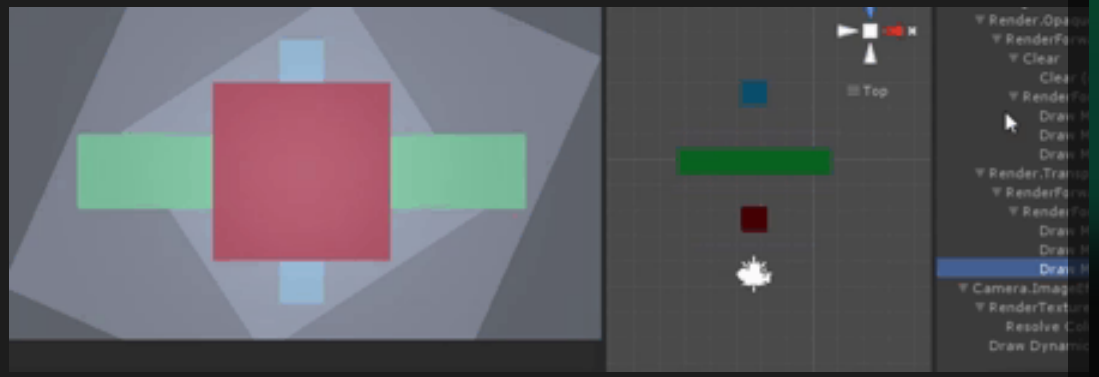
- Queues up to 2500 ("AlphaTest+50") are considered "opaque" and optimize the drawing order of the objects for best performance: an obj qith 2500 will still belong to the Render.OpaqueGeometry queue
- You can't have a value > 5000
- Overlay is dedicated to UI. Screen space UI will be always painted over scene objects

[DrawOrder_Sample]


Painter's algorithm

range	default	keyword
0/1499	1000	Background
1500/2399	2000	Geometry
2400/2699	2450	AlphaTest
2700/3599	3000	Transparent
3600/5000	4000	Overlay

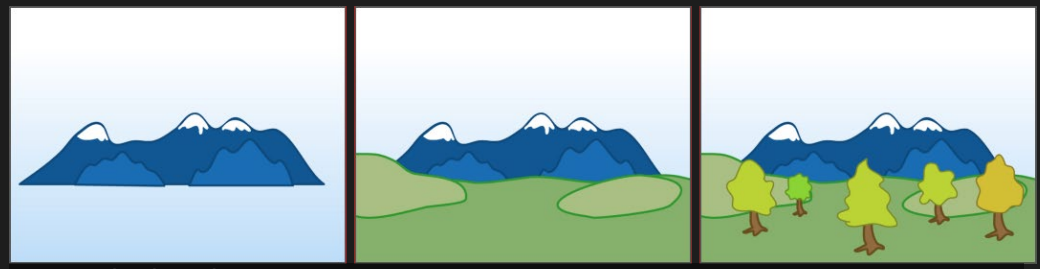




Rendering order

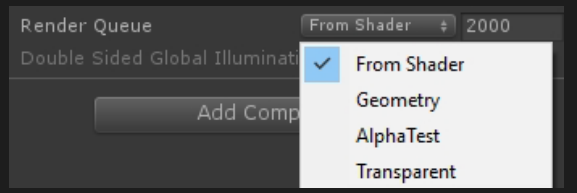
- AlphaTest: alpha tested geometry uses this queue. It's a separate queue from Geometry one since it's more efficient to render alpha-tested objects after all solid ones are drawn. These objects write on the depth buffer (alpha cutoff)
- Transparent: shaders that don't write to depth buffer
- Overlay: overlay effects. Anything rendered last should go here (e.g. lens flares)

[DrawOrder_Sample]



Painter's algorithm

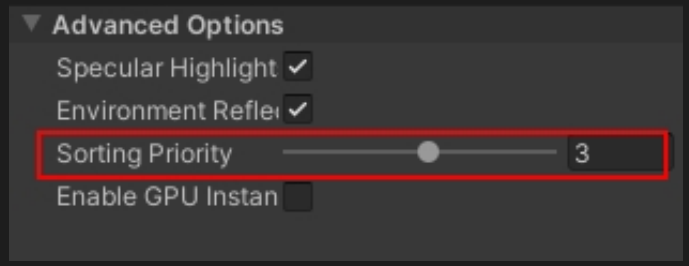
range	default	keyword
0/1499	1000	Background
1500/2399	2000	Geometry
2400/2699	2450	AlphaTest
2700/3599	3000	Transparent
3600/5000	4000	Overlay



Rendering order

Switch to URP

- Use FirstShaderH Material to reply the same effect you've seen in BIRP
- URP/Lit shader doesn't have Render Queue parameter, but has AdvancedOptions/SortingPriority

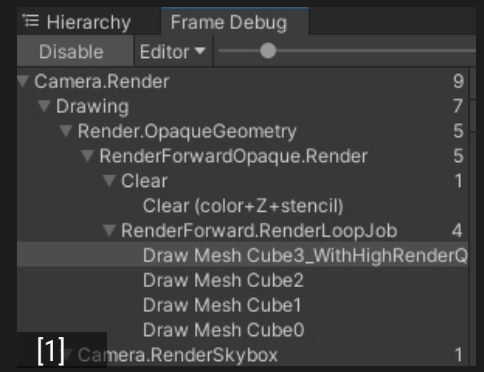


[DrawOrder_Sample]

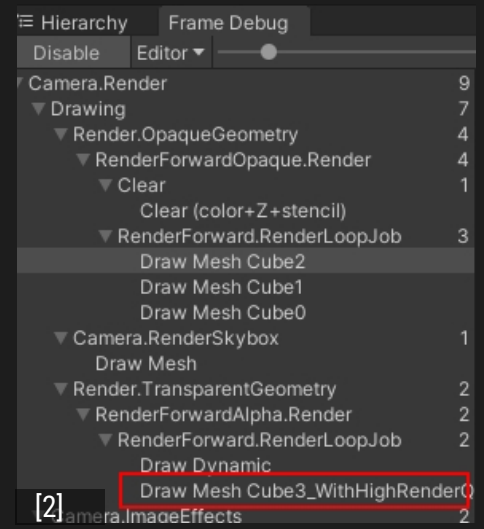
Rendering order

- Try it
  - Enable Cube3_WithHighRenderQueue
  - Use Mobile/Diffuse shader with RenderQueue set to 1500
    - It is rendered before the other cubes, even if it is the farthest one [1]
  - Use Mobile/Diffuse shader with RenderQueue set to 5000
    - This means that it will be the last object drawn (see FrameDebug drawcall order) inside the TransparendGeometry queue (even if it is opaque) [2]
    - Since Depth test is enabled and transparent objects don't write ZBuffer, it will be rendered behind the other opaque objects, but in front of the Transparent objects (see next slide)
  - If we add a UI/Image, it is rendered in Overlay queue
  - If we switch Canvas to WorldSpace and [2] is happening, we could generate an artifact like [3], because World space UI doesn't write Zbuffer, and AFTER that the Opaque object with RenderQueue 5000 is drawn!

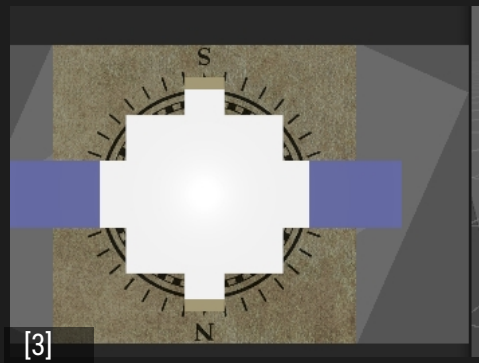
[DrawOrder_Sample]



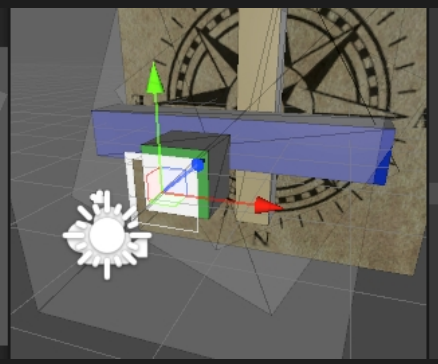
[1]

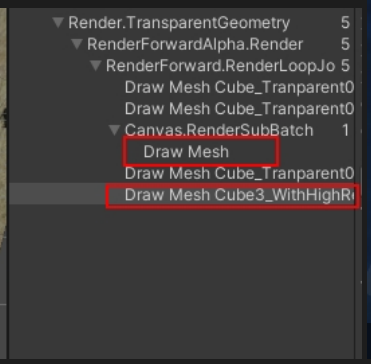


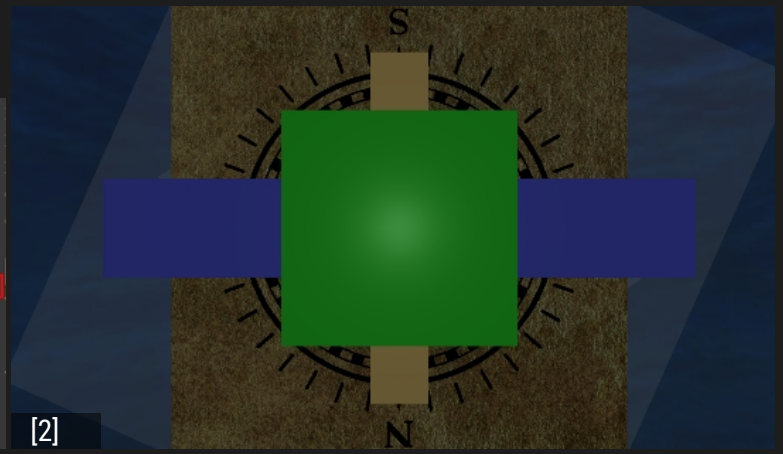
[2]



[3]







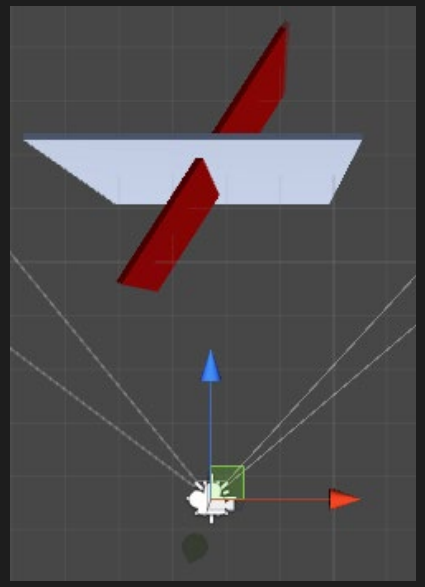
[2]

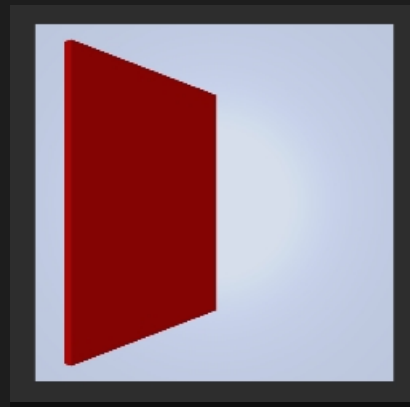
Sorting

Sorting, Z-Sorting, Depth-Sorting
Opaque objects are simple
- Z-Buffer: If Z-Test fails, PS doesn't write the ColorBuffer

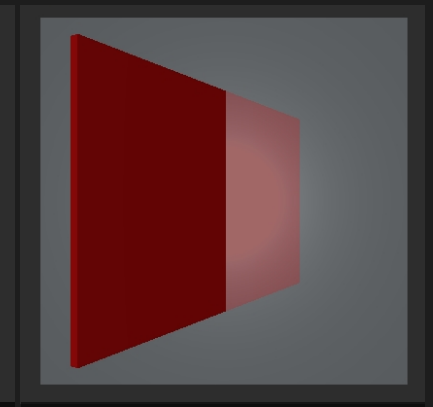
Semi-transparent objects
- Without intersection
  - Painter's algorithm
  - Render queue
- With intersection: custom solutions
  - It's very complex and complicated to render pixels from far to near
  - Object-center sorting is common, still can be time consuming
  - Object sorting doesn't guarantee pixel sorting: objects can intersect each other, or can be concave
  - Order Indipendent Transparency (OIT)
    - A-Buffer
      - Each transparent pixel write ZBuffer AND A-Buffer
      - If Z < Zbuffer, transparent pixel is above the other pixels, it maintains its alpha value
      - If Z >= ZBuffer value, take into account ABuffer like an "alpha mask": its alpha value is premultiplied by (1-Abuffer)
    - Depth Peeling
    - Weighted Blending

[Sorting_OpaqueTransparent_Sample]

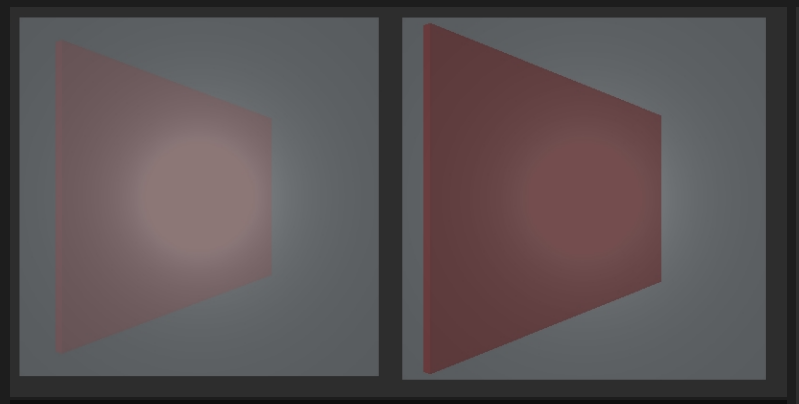




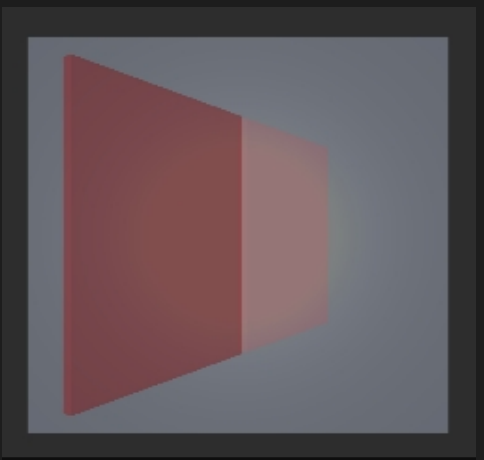
Opaque/Opaque
Result is correct



Opaque/Fade
Result is correct



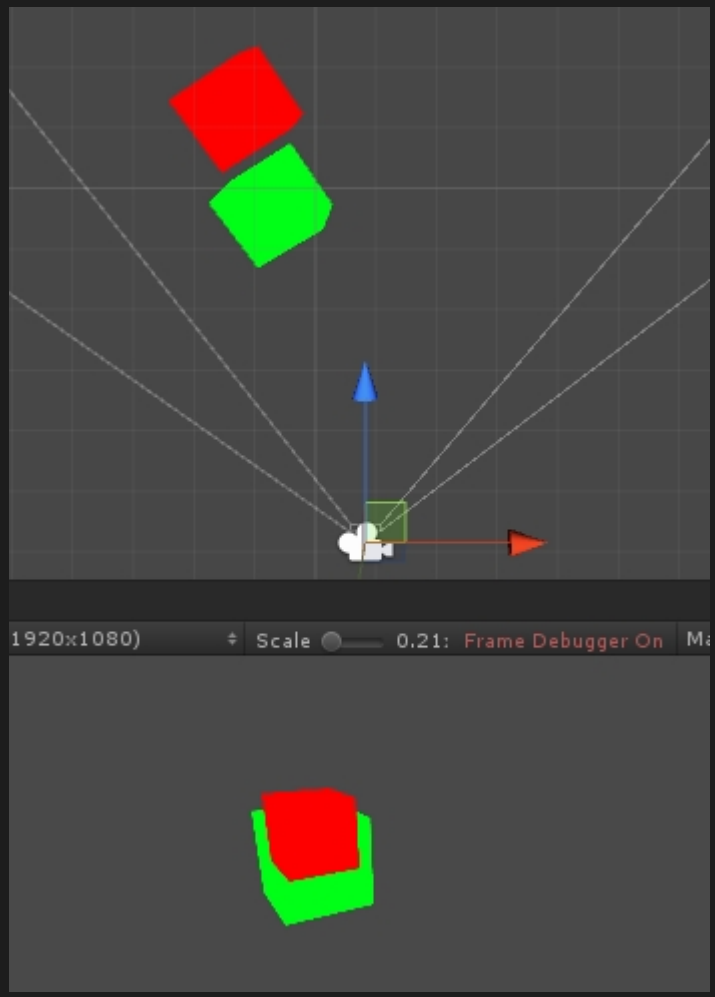
Fade/Fade
Red plane pivot (if closer or not to the camera vs the white one) determines the rendering order, but the entire object is rendered over the other



Fade/Fade - Splitted mesh
Rendering order based on Z values lead to a correct result

Z-Test, ColorMask

- Activate ZWrite_ZTest_ColorMask_Sample
- ZWrite [Off | On]
  - Off: If this fragment passes the Z-Test, don't update the Z-Buffer
  - On: If this fragment passes the Z-Test, update the Z-Buffer
    - General rule: ZWrite must be On for Opaque Objects
- ZTest [Less | Greater | LEqual | GEqual | Equal | NotEqual | Always]
  - Specify what is the function to use for the Z-Test
    - NB: The Z-Buffer is initialized with the lowest Z value
- ColorMask [R | G | B | A]
  - Specify what color channel to write in the ColorBuffer
  - ColorMask R means that only the R channel is written
  - Try it using a white color and masking R,G,B channels separately



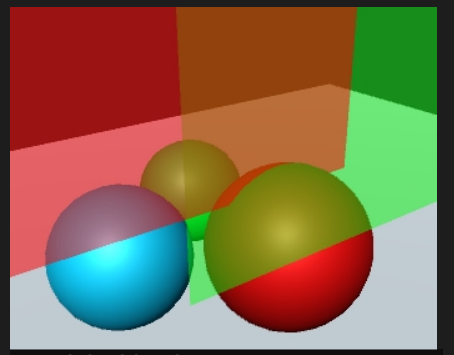
- Let's say that the ColorBuffer has just rendered a flat blue background, and for the next object I want only channels RGA to pass. What is the final obj color if we use frag1 or frag2 PShaders?

```
ColorMask RGA //Ignore B channel
half4 frag1(vertexOutput i): SV_Target
{
   half4 finalColor = half4(0,0,1,1);
   …
   return finalColor;
}
```
Result: BLUE Color

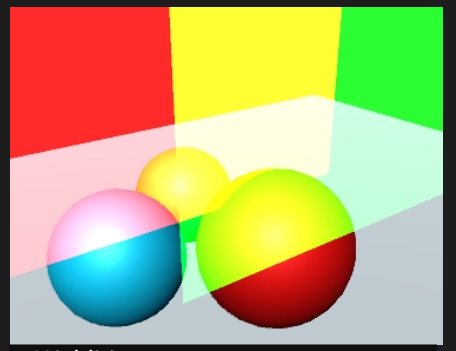
```
ColorMask RGBA //Write all
half4 frag2(vertexOutput i): SV_Target
{
   half4 finalColor = half4(0,0,1,1);
   …
   return half4(finalColor.r, finalColor.g, 0, 1);
}
```
Result: BLACK Color (We overwrite the existent blue color with a 0)

Blending

- [URP] Setup scene
  - Game view fixed resolution (Eg. HD)
  - Assign Spheres and Floor an URP/Lit Material shader
  - Create transparentShaderH from firstShaderH, create 2 materials that use this shader, assign them to Transparent_quadTop and Transparent_quadBottom
  - Activate AlphaBlendingOnRT and check if a RT image appear on the top left of the monitor



```
//Alpha blending
BlendOp Add
Blend SrcAlpha OneMinusSrcAlpha
```



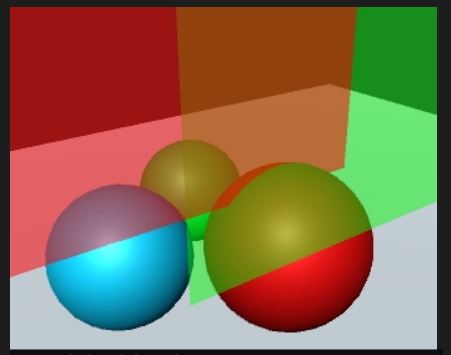
```
//Additive
BlendOp Add
Blend One One
```

Blending

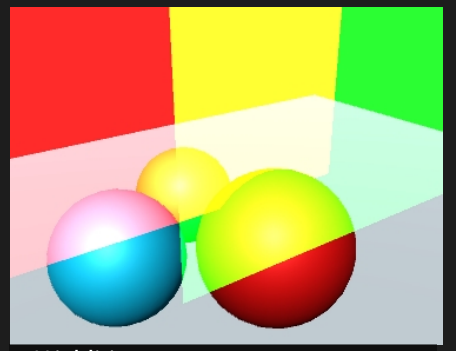
- Blend SrcFactor DstFactor, SrcFactorA DstFactorA
  - Factor values are: One | Zero | Src[Alpha/Color] | OneMinusSrcAlpha | Dst[Alpha/Color]
- BlendOp OpColor, OpAlpha
  - Min | Max | Add | Sub
- If SrcFactorA DstFactorA and OpAlpha are not specified, they have the same value of SrcFactor DstFactor and OpColor
- FinalVal = [SrcColor x SrcFactor] [BlendingOp] [DstColor x DstFactor]
  - SrcColor: PS output color
  - DstColor: ColorBuffer color
- NB
  - To work with Blend and BlendOp for the Alpha channel, you need to work with RGBA RenderTexture, because ColorBuffer doesn't have Alpha channel!
  - To have alpha blending you have to use "Queue" = "Transparent" subshader tag
- Complete list of Blend operations

Try it

- Create transparentShader from firstShader, create 2 materials that use this shader, assign them to Transparent_quadTop and Transparent_quadBottom
- Change their RenderQueue Order to Transparent
  - From the inspector
  - Or by adding in the Tags with subshader scope, not pass scope `"Queue" = "Transparent"`
- Don't write the Zbuffer: ZWrite Off
- [Additive] Start trying a BlendOp Add and Blend One One:
  - Red plus Green is yellow. Your result is: [SrcCol x 1] + [DestCol x 1]



```
//Alpha blending
BlendOp Add
Blend SrcAlpha OneMinusSrcAlpha
```



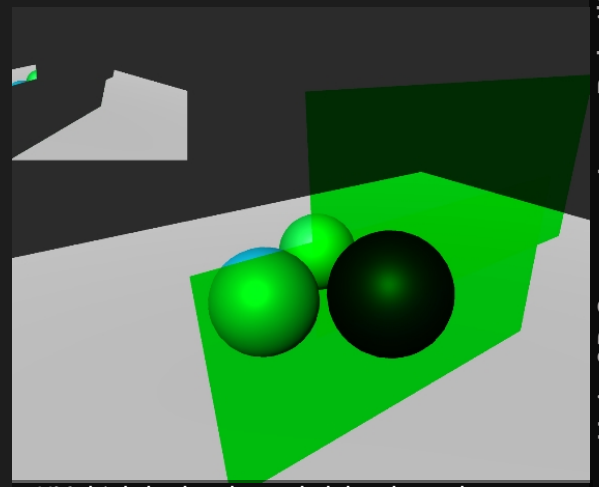
```
//Additive
BlendOp Add
Blend One One
```

[Blending_Start, _transparentShader.shader]

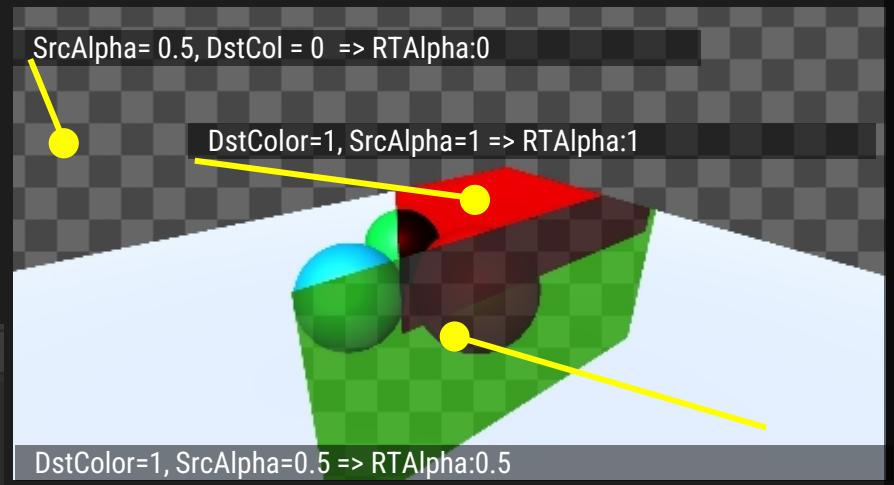
Blending

Try it

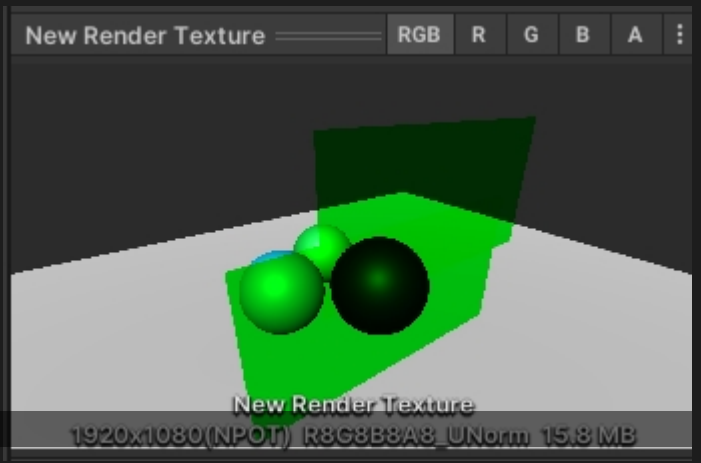
- [Standard Alpha blending] BlendOp Add and Blend SrcAlpha OneMinusSrcAlpha:
  - Your result is: [SrcCol x SrcAlpha] + [DestCol x OneMinusSrcAlpha]
- [Multiply] BlendOp Add and Blend DstColor Zero:
  - Your result is: [SrcCol x DstColor] + 0
- Try to change alpha while using Multiply Blending: you have no effect, because Blend operator is multiplying only colors. We need a RTarget image to see the final effect
- Activate AlphaBlendingOnRT
  - It is the same as: BlendOp Add, Add and Blend DstColor Zero, DstAlpha Zero
  - Set Alpha to 0: the final alpha should be 0 x DstColor = 0. But how can we see 0 alpha value if the final Color buffer doesn't have alpha?
  - Select the Camera Rtexture
    - RGB values shows the exact result of the colorBuffer (no A channel)
    - A channel shows our 0 alpha value!
- What if we want to have an Add in the alpha channel and maintain a Multiply in the color channel? Use
  - BlendOp Add, Add and Blend DstColor Zero, One One



//Multiply both color and alpha channels
BlendOp Add, Add
Blend DstColor Zero, DstColor Zero

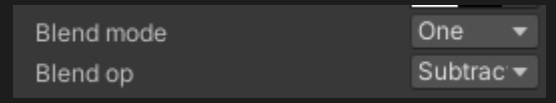


Alpha channel (select RT in the inspector to see it)



[Blending_Start, _transparentShader.shader]

Blending



You can use:

```
[Enum(UnityEngine.Rendering.BlendMode)] _Blend ("Blend mode", Float) = 1
[Enum(UnityEngine.Rendering.BlendOp)] _BlendOp ("Blend op", Float) = 1
```

to expose Blending enums in your inspectors, and to use those values:

```
BlendOp [_BlendOp]//1 - Additive with Enum
Blend [_Blend] [_Blend] //1 - Additive with Enum
```

Blending

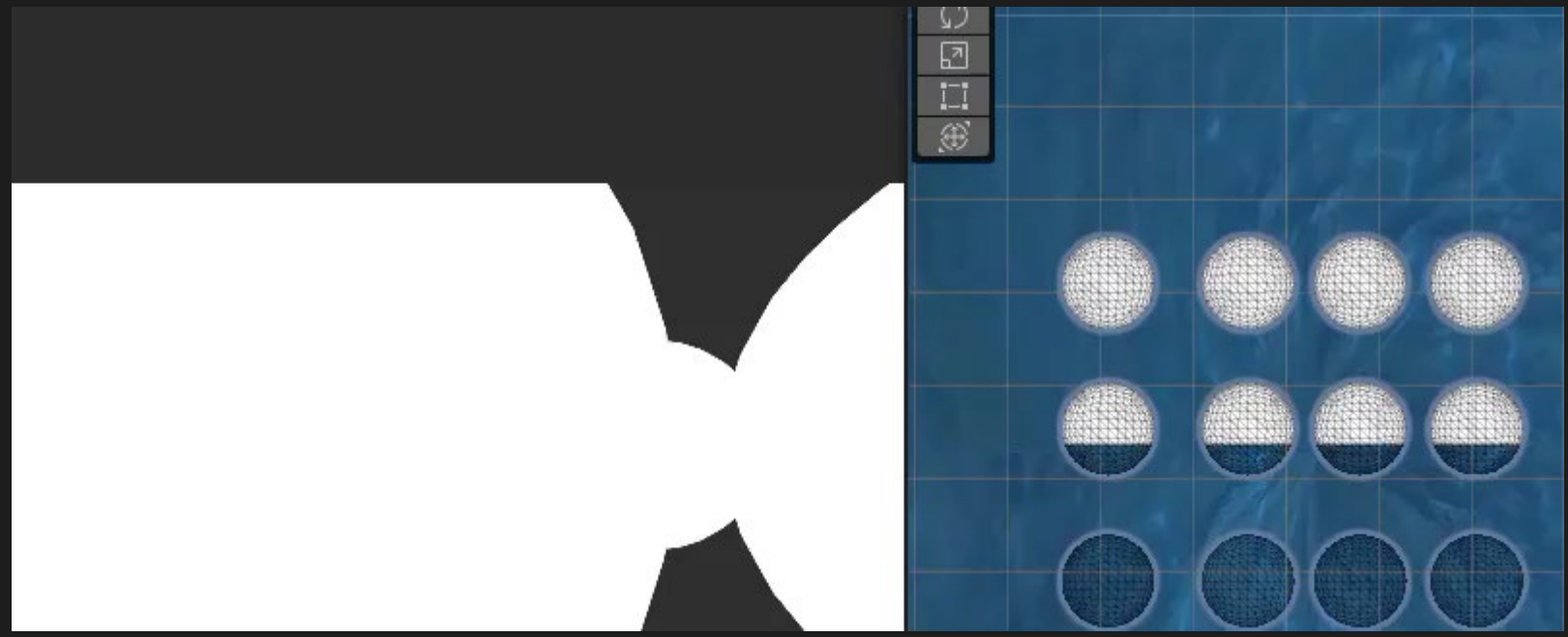
The most common types of blending are the following:

- Blend SrcAlpha OneMinusSrcAlpha Common transparent blending
- Blend One One Additive blending color
- Blend OneMinusDstColor One Mild additive blending color
- Blend DstColor Zero Multiplicative blending color
- Blend DstColor SrcColor Multiplicative blending x2
- Blend SrcColor One Blending overlay
- Blend OneMinusSrcColor One Soft light blending
- Blend Zero OneMinusSrcColor Negative color blending

Near clip plane Blending

- _ProjectionParams.y is the camera's near plane
- We can use the distance from the camera (in view space) and _ProjectionParams.y to know if we should fade out the object, avoiding near clipping effect!
- Write [nearClipPlaneAlphaH] from [transparentShader]

[nearClipPlaneAlphaH]



AlphaToMask

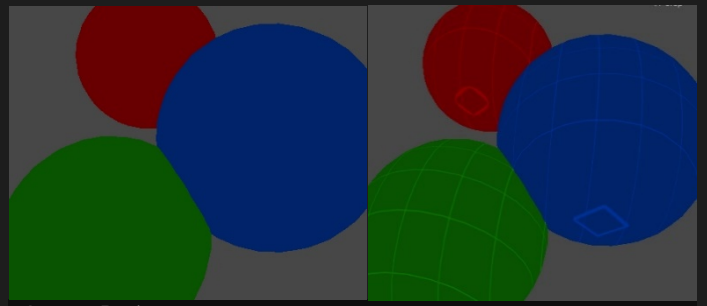
- There are some types of Blending that are very easy to control, e.g., the SrcAlpha OneMinusSrcAlpha Blend, which adds a transparent effect with the Alpha channel included, but there are other cases where Blending is not capable of generating transparency for our shader. In this case, the AlphaToMask property is used, which applies a mask over the Alpha channel and is a technique compatible with both BIRP and URP
- Unlike Blending, a mask can only assign the values 1/0 to the Alpha channel
- While the Blending can generate different levels of transparency; levels from 0.0f to 1.0f, AlphaToMask can only generate integers. This translates to a harsher type of transparency, which works in specific cases, e.g., it is very useful for vegetation in general and to create space portal effects. To activate this command, we can declare it in both the SubShader scope and the pass scope. It only has two values: On/Off, and it is declared in the following way:

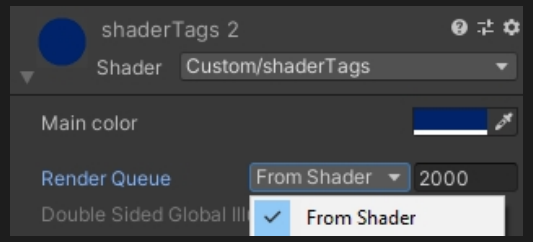
```
Shader "InspectorPath/shaderName"
{
    Properties { … }
    SubShader
    {
        Tags { "RenderType"="Opaque" }
        AlphaToMask On
    }
}
```

- In this case, it is not necessary to add Transparency tags or other commands. We just add AlphaToMask and automatically the alpha channel acquires the qualities of the mask in our program (<0.5 transparent, opaque in the other cases)
- Try it
  - Create alphaToMask from firstShader, create 1 material and assign it to a new Quad
  - Change its alpha

[AlphaToMask]

Subshader/Pass Tags


Ignore Projector



- Tags can have Subshader or Pass scope

- They are defined in Tag block, separated by spaces
  - Tags{ “TagName” = “TagValue” “TagName” = “TagValue” … }
  - **valid only in Subshader scope**:
    - Queue – Rendering Order
    - [BIRP] IgnoreProjector
    - RenderType – [BIRP] Used by Replacement shader
  - For Pass Tags, see here
  - Custom Tags

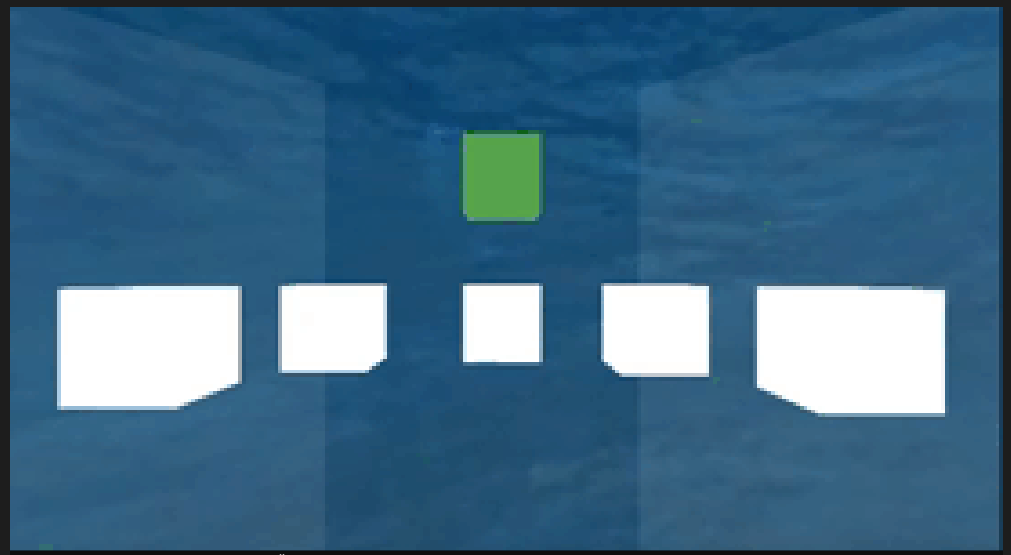
Try it
- Starting from firstShader create firstShaderTags and 3 materials and assign them to the 3 spheres. Use a dark color for them
- Add Subshader block Tags{"Queue" = "Geometry"}
- Change RenderQueue value in the material inspector to FromShader
- Change Tags subtracting a value of 2000: Tags{"Queue" = "Geometry-2000"}
- This change will reflect in the inspector
- Changing this value from the inspector allows you to change rendering order!
- [BIRP] Add "IgnoreProjector" = “True“: now projector has no effect!

[Tags_Start, _firstShaderTags.shader]

[BIRP] RenderType

- In the Built-in Render Pipeline, you can swap SubShaders at runtime using a technique called shader replacement. This technique works by identifying SubShaders that have matching RenderType tag values
- Some rendering effects require rendering a scene with a different set of shaders. So if all shaders would have, for example, a RenderType tag with values like Opaque, Transparent, Background, Overlay, you could write a replacement shader that only renders solid objects by using one subshader with RenderType=Solid tag
- All BIRP Unity shaders have a RenderType tag set

- Camera.SetReplacementShader(Shader shaderFx, string replacementTag);
- Camera.ResetReplacementShader();
    - If replacementTag is empty, then all objects in the scene are rendered with the given replacement shader
    - If replacementTag is not empty, then for each object that would be rendered:
        - The real object's shader is queried for the tag value
            - If it does not have that tag, object is not rendered
            - If it has that tag, a subshader is found in the replacement shader that has a given tag with the found value. If no such subshader is found, object is not rendered
                - Otherwise, that subshader is used to render the object

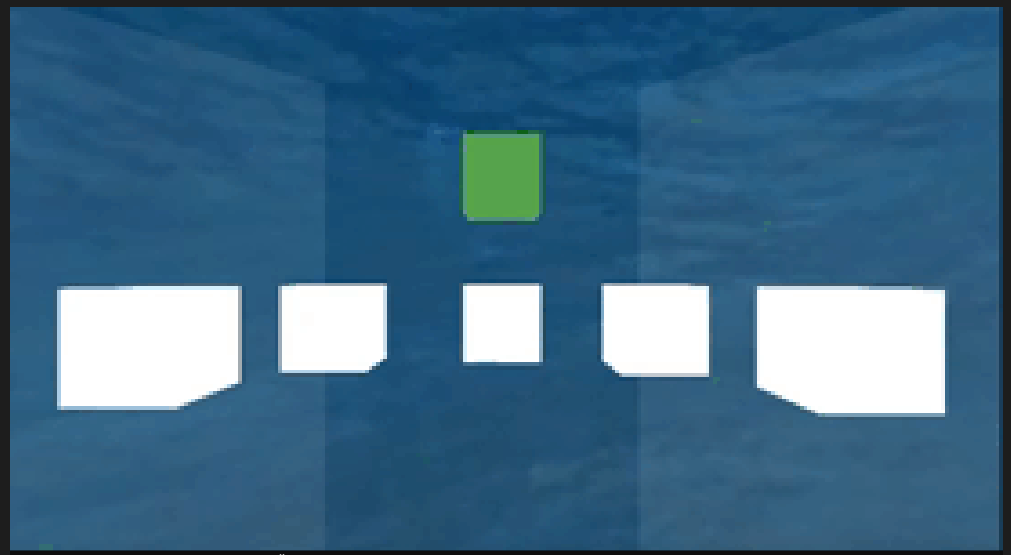


SetReplacementShader()

[BIRP] RenderType

Try it

- Activate RenderType_SetReplacement
- Cubes have _RenderTypeOpaque shader, with “CustomTag” = “CustomTagValue_1”
- Planes have _RenderTypeTransparent shader, with “CustomTag” = “CustomTagValue_2”
- When you turn ON ShaderReplacer.ApplyFx, _RenderTypeFx shader is used to render Cubes and Planes:
  - Cubes will use its opaque subshader
  - Planes will use its transparent subshader, and will change their RED channel value using _Time Built-In Shader Variable (x,y,z,w) = (t/20, t, t*2, t*3)
- Try to add a cube with StandardShader
  - If you perform a replace with Tag RenderType it works, if you use CustomTag it isn’t rendered!



SetReplacementShader()

[Shaders_Start_A_01, Tags_Start, _RenderTypeOpaque, _RenderTypeTransparent, _RenderTypeFx]

Textures

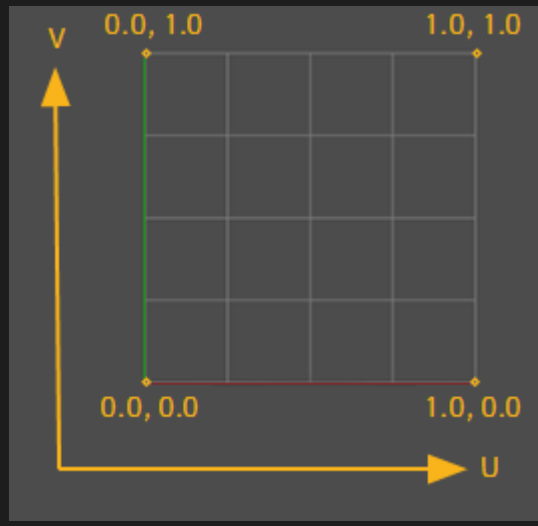
- Every Mesh has an associated UVSet, a 2DSpace within [0,1] range
- UV vertex coords are imported into Unity, and provided to the VS

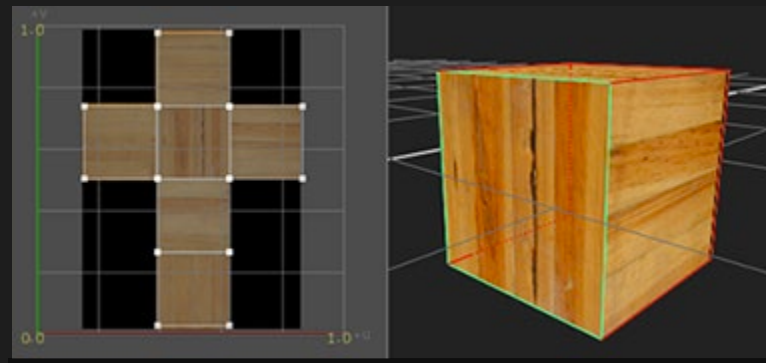
Try it

- Starting from transparentShader create texture shader and 1 material and assign it to the quad

To read a texture in a shader:

- Inspector texture property
  - _MainTex("Main Texture", 2D) = "white" {}
- Global variable in order to access the texture inside shader code
  - sampler2D _MainTex;
  - float4 _MainTex_ST; //Inspector Tiling (xy) & Offset (zw) values
- UVCoords for each vertex in the vertexInput structure
  - float4 texcoord : TEXCOORD0;
- VShader will pass vertexInput.texcoord to the PShader, and then they will be interpolated from the rasterizer. In the vertexOutput structure, we need:
  - float4 texcoord : TEXCOORD0;
- NB: the semantic TEXCOORDN as vertex output data is not strictly related to texture coordinates. Every time we need to interpolate a float4 value, we can use this semantic





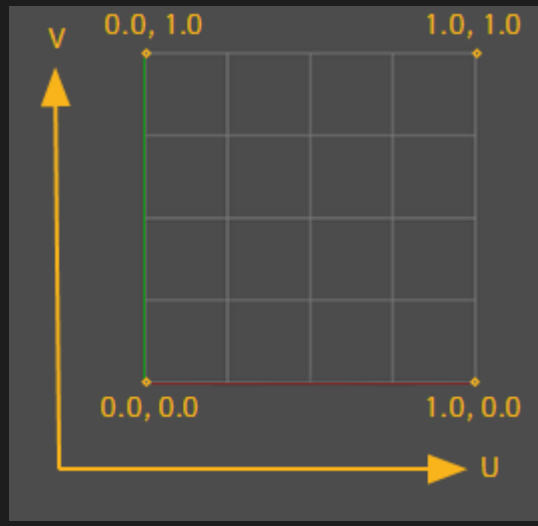
UVMapping

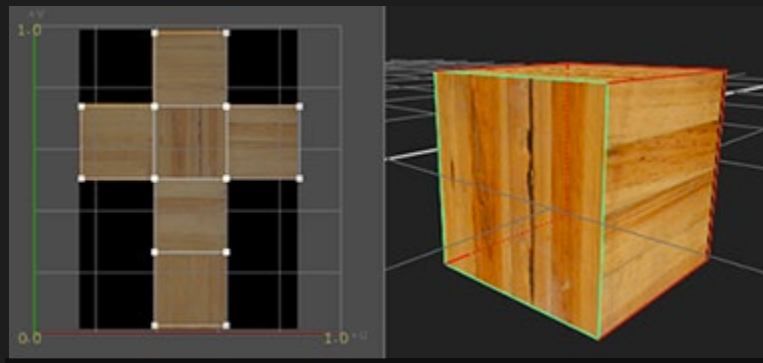
[Texture_Start, _texture.shader]

Textures

To read a texture in a shader (continue)

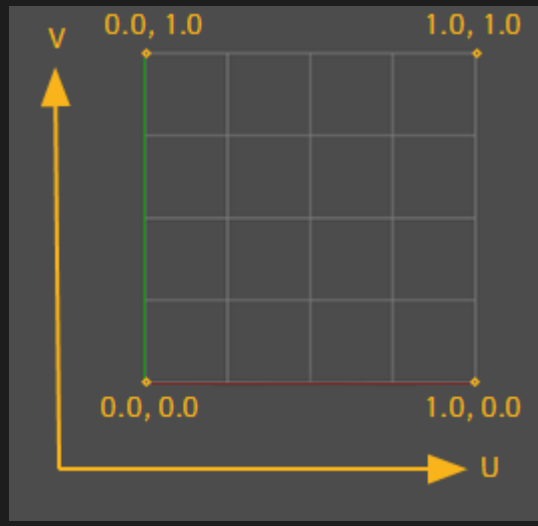
- VShader
  - Just copy vertexInput.texcoord to vertexOutput.texcoord
  - To take into account Tiling and Offset:
    o.texcoord.xy = (v.texcoord.xy * _MainTexture_ST.xy + _MainTexture_ST.zw);
    - Or o.texcoord.xy = TRANSFORM_TEX(v.texcoord.xy, _MainTexture);
      - In BIRP you have to include UnityCG.cginc
    - Which is a macro:
      #define TRANSFORM_TEX(tex,name) (tex.xy * name##_ST.xy + name##_ST.zw);
- PShader
  - Sample the texture _MainTex at point vertexOutput.texcoord
  - half4 textureColor = tex2D(_MainTex, vertexOutput.texcoord);
  - Sample the texture _MainTex using vertexOutput.texcoord.w MipMap index
  - half4 textureColor = tex2Dlod(_MainTex, vertexOutput.texcoord);
- Properties attributes
  - [NoScaleOffset] Material inspector will not show texture tiling/offset fields for texture properties with this attribute
  - [Normal] If this texture is not imported as NormalTexture, material inspector will show a warning
  - [MainTexture] Indicates that a property is the main texture for a Material. By default, Unity considers a texture with the property name _MainTex as the main texture. Use this attribute if your texture has a different property name, but you want Unity to consider it the main texture
  - [MainColor] Indicates that a property is the main color for a Material. By default, Unity considers a color with the property name name _Color as the main color. Use this attribute if your color has a different property name, but you want Unity to consider it the main color





UVMapping

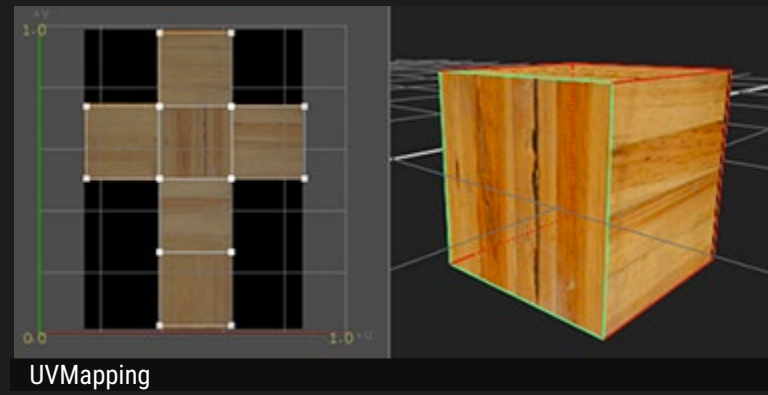
Textures



Try it

- Try to use first tex2D(), then text2Dlod(), passing _MipMap level in the inspector

- NB
  - Texture sampling will reflect import Filtering (Point/Bilinear)
  - Texture Mipmap sampling value is a float: if you have bilinear filtering on that texture, you can sample Mipmap “3.5”, that is in between Mipmaps 3 and 4, like if you are using Trilinear filtering!
  - Vertex input cords are: float4 texcoord : TEXCOORD0; Why text cords are a float4?
    - x,y: uv
    - z: used in texture3D sampling
    - w: contains the right Mipmap index, according to camera distance. In this way, tex2D() sample the texture with the correct mipmap



UVMapping

Is it possible to perform a texture fetch in the Vshader? (VTF, VertexTextureFetch)

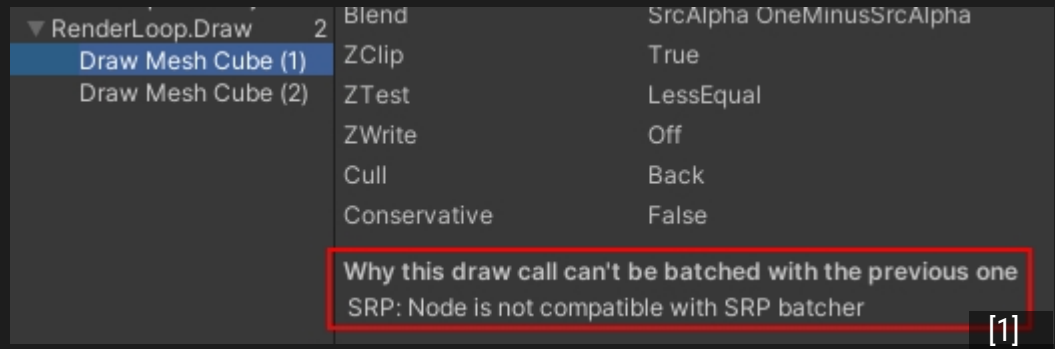
- From shaderModel3 it is possible, but we have to use tex2Dlod(). This is because tex2D() is really a shortcut that says "figure out the right mip level to sample automatically" - in a fragment shader this is done using implicit derivatives, but those aren't available at the vertex stage

[Texture_Start, _texture.shader]

Enabling SRPBatch

- Try to duplicate the cube with your textureH shader and open the Frame debug: you'll see warning [1], or something similar
- Shaders created using Shader Graph automatically support the SRP Batcher
- Our written-by-hand shader need to define a UnityPerMaterial CBUFFER [2]
- Put inside the CBUFFER block all the per-material properties except for textures. Every pass in the shader needs to use the same CBUFFER values, so it would be useful putting it inside the SubShader in HLSLINCLUDE tags (we'll see them later)

[_textureSRPBatchH]



[1]

```
uniform sampler2D _MainTex;
CBUFFER_START(UnityPerMaterial)
    uniform half4 _Color;
    uniform float4 _MainTex_ST; //used for Tiling and offset
    float _MipMap;
CBUFFER_END
```
[2]

Positional color Texture Space

- We can use
  float4 textureScreenPos = ComputeScreenPos(projSpaceVertexPos)
  to know its position in Texture Space
  - (0,0) bottom left, (1,1) top right
- You should be dividing .xy / .w in the pixel shader. ComputeScreenPos() will just transform input from clip coordinate vertex position [-w,w] into [0,w]

[_positionalColorTextureH.shader]

(0,0) bottom left, (1,1) top right

Positional color Texture Space (Sampling)

- We can use
  float4 textureScreenPos = ComputeScreenPos(projSpaceVertexPos)
  to know its position in Texture Space
  - (0,0) bottom left, (1,1) top right
- You should be dividing .xy / .w in the pixel shader

- ComputeScreenPos() will not divide input's xy by w because this method expect you sample the texture in fragment shader using tex2Dproj(float4)
- tex2Dproj() is similar to tex2D(), it just divide input's xy by w in hardware before sampling, which in much faster than user code division in fragment shader(result always correct but slow), or vertex shader(result will not correct if polygon not facing directly to camera)

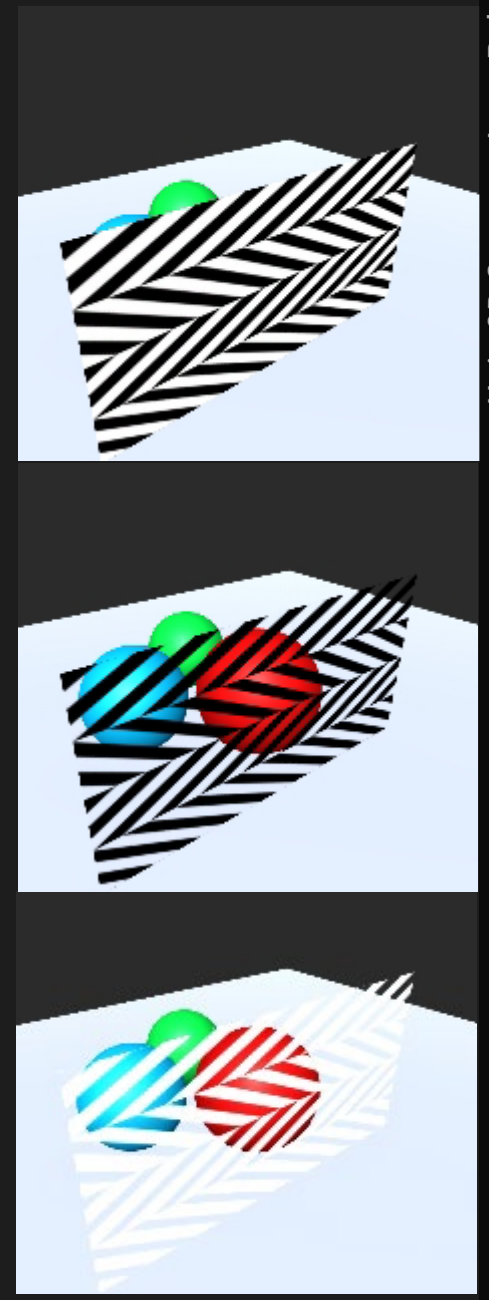
- ComputeScreenPos() will just transform input from clip coordinate vertex position [-w,w] into [0,w]
- then calling tex2DProj() will transform [0,w] into [0,1], which is a valid texture sampling value

[_positionalColorTextureSamplingH.shader]

Blending with textures

- Activate Blending_WTexture_End
- Try to change _textureTransparent Blend modes : Add/Multiply/AlphaBlending

[_textureTransparent]



Vertex Animation

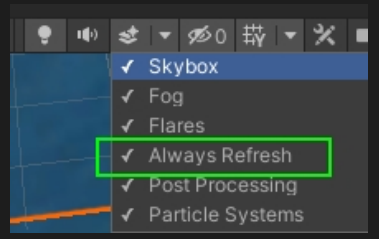
- Move vertices along a texture coordinate axis using sin() function
- Use www.desmos.com/calculator to help you with the final equation
  - To animate the sin() function, we can use _Time built-in variable
  - We need Frequency, Speed and Amplitude to tweak this kind of deformation
    - offset = sin(x – (_Time*Speed) * Frequency) * Amplitude
- sin() distortion shouldn't start from UV coord U=0, but from U=val, where val is the position of the flagpole
  - We can use smoothstep() function to achieve this

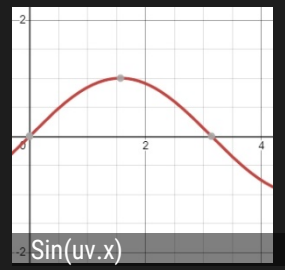
Vertex Animation

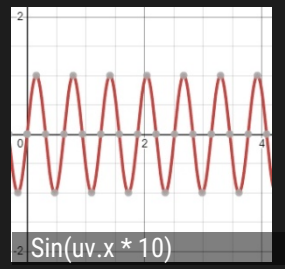
Try it

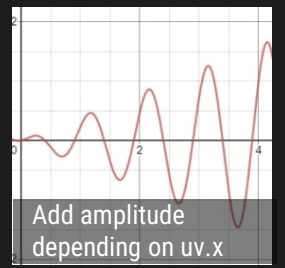
- Create TextureFlag shader and material from texture shader, and assign it to Plane_Flag object
- Change textureFlagStart material in order to use TextureFlag shader, and move the start of the flag near the pole
- Declare a flagMovement method that changes vertex position in object space depending on its uv (before multiplying it by MVP matrix)
- Activate AnimatedMaterials in SceneView Toggles
- Try sin(uv.x), sin(uv.x*10), sin(uv.x*10-t) in the flagMovement calculation. Use _Time.w as time value
  - float4 _Time Time since level load (t/20, t, t*2, t*3), use to animate things inside the shaders
  - float4 unity_DeltaTime Delta time: (dt, 1/dt, smoothDt, 1/smoothDt)
- Add Frequency/Amplitude/Speed Float parameters. Use them to change the flagMovement calculation (try it in the graph simulation)
- The movement should start from the pole, not at the beginning. We can
  - multiply by uv.x to obtain this
  - using smoothstep function to move vertices only from one specific uv.x value
- Try to calculate 2 displacement values, using different Frequency/Amplitude/Speed values

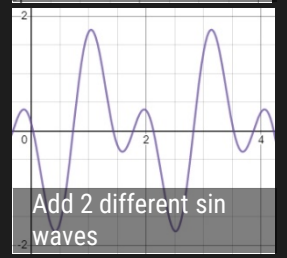
[Shaders_Start_B_01, Texture_Flag_Start, _textureFlag.shader]











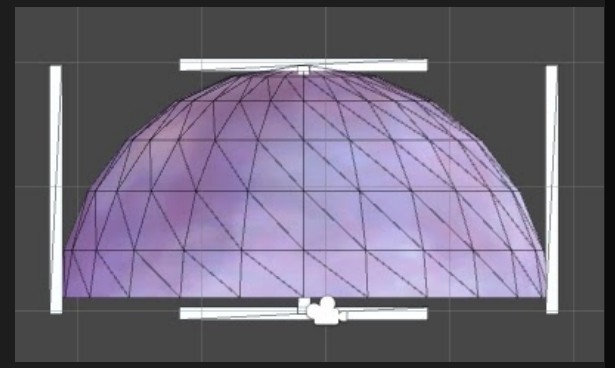
Vertex Normals

- We can displace vertices using their normal values
- To receive normal values for each vertex
  - Add float4 normal : NORMAL; to Vshader input
  - normal.xyz will be a normalized vector in 3D
  - Since it is a direction, normal.w will be 0

Try it

- Create textureNormals shader and material from textureFlag shader, and assign it to Jelly_Up_Start object, with jellyUp texture
- Duplicate multiple times Plane obj under Tentacles, and assign to them textureFlag material with tentacle_02 texture
- We don't need StartFrom property remove it
- Add NORMAL input to vertexInput struct. We'll use the normal in objectSpace: no need to pass it to the Pshader
- Rename flagMovement function to normalMovement, adding a float4 vNormal input. Let's start with an X offset:
  - vIn.x += (_Amplitude * vNormal.x);
- With an amplitude of 1, the vertices will reach cubes position (try to expand also in y and z axes)
- Use also _Speed, _Frequency to achieve this deformation:
  - vIn += (sin((vNormal - (_Time.w*_Speed))*_Frequency)) * (_Amplitude * vNormal);
- Frequency 4, Amplitude 0.1, Speed 0.3 is a good start. Use Speed 1 to see the deformation

[_Shaders_Start_B_01, _textureNormals.shader]

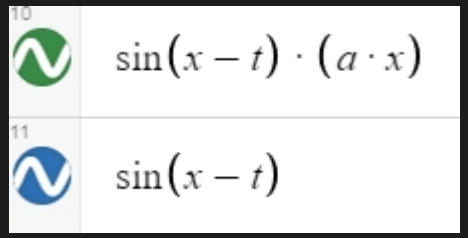


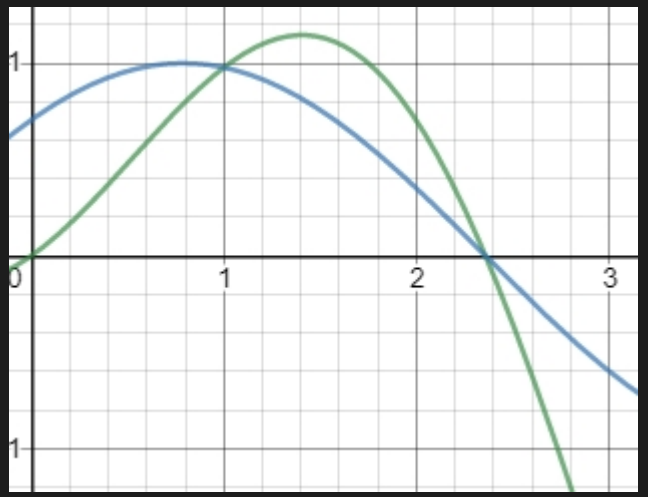
Vertex Normals

Try it

- //6 If vNoxrmal.x == 0, depending on time it could move +1/-1
- //7 the equation this time pass through the origin: If vNoxrmal.x == 0, it doesn't move
- NB: If we use the same fx on the lion, we'll see some spaces between polygons. This is because some vertices are doubled to achieve hard edge effect. Try to tweak Smoothing angle import: 0 or 180 values will result in Separated quad/merged quads

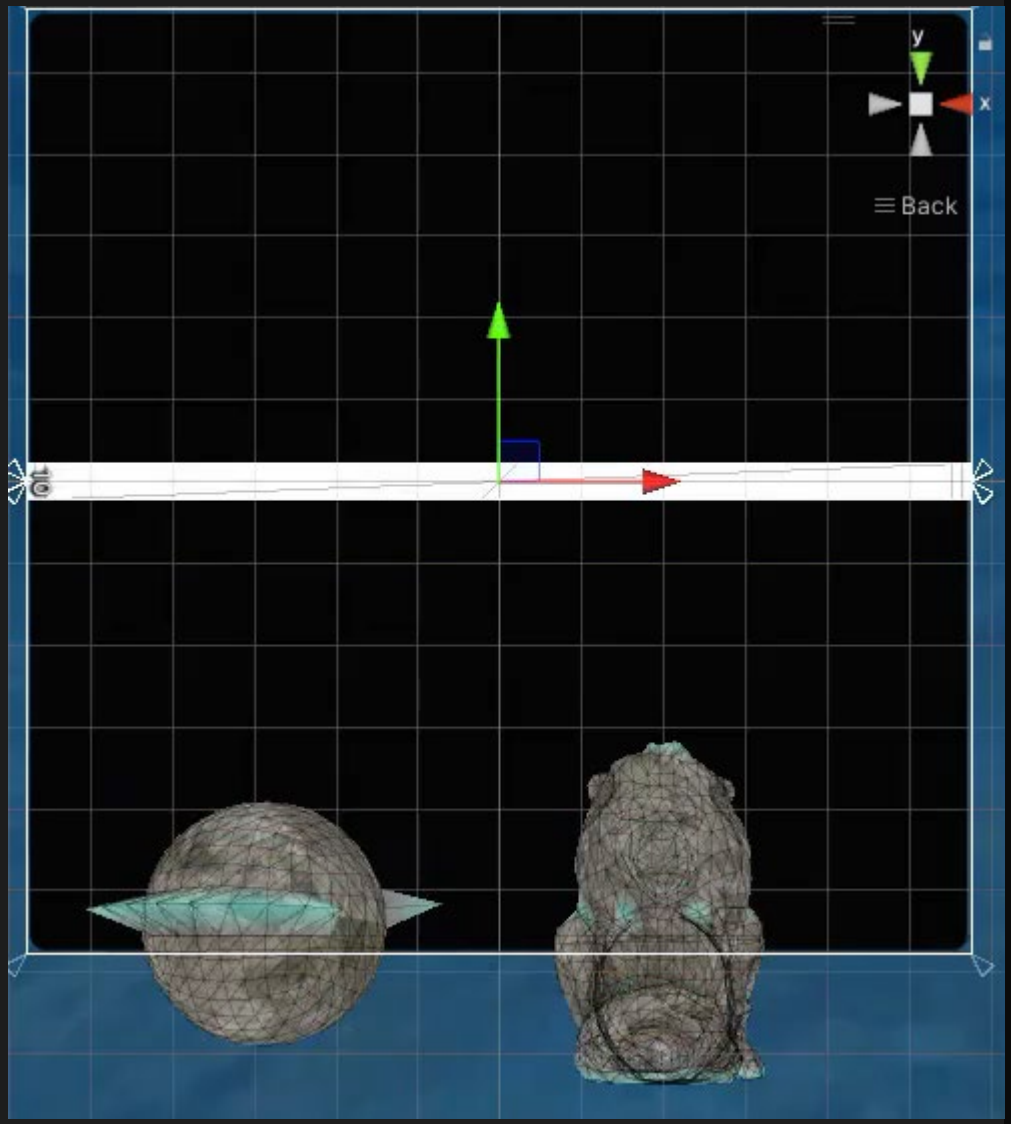
[_textureNormals.shader]





Displacement in World space

- Create textureNormalsTextureDisplacementH shader and material from textureNormals shader, and assign it to Lion_statue object and a sphere obect
- Change it in an Opaque shader
- Use these properties
  - _DispColor
  - _DispTex
  - _Amplitude, Range(1,5)
  - _Speed, Range(1,5)
  - _EndFade, Range(1,5)
- We need a Black/White texture that give us a displacement amount
  - Online gradient editor: https://angrytools.com/gradient/
- Try to change your shader in order to pick the amount of displacement from _DispTex texture, sampled in the vertex shader
- Now use worldSpace obj coordinates to sample the texture (u coord is not important in our example that has a vertical gradient, the texture could have 1 pixel width): take into account that worldPos.y changes very fast, and you need a [0,1] range all over the vertical object space in order to see something: divide worldPos by a factor (10, 20, etc)
- You can even use a RTexture as Displacement texture



[textureNormalsTextureDisplacementH_End_00, _textureNormalsTextureDisplacementH]

Normal Map 1/2

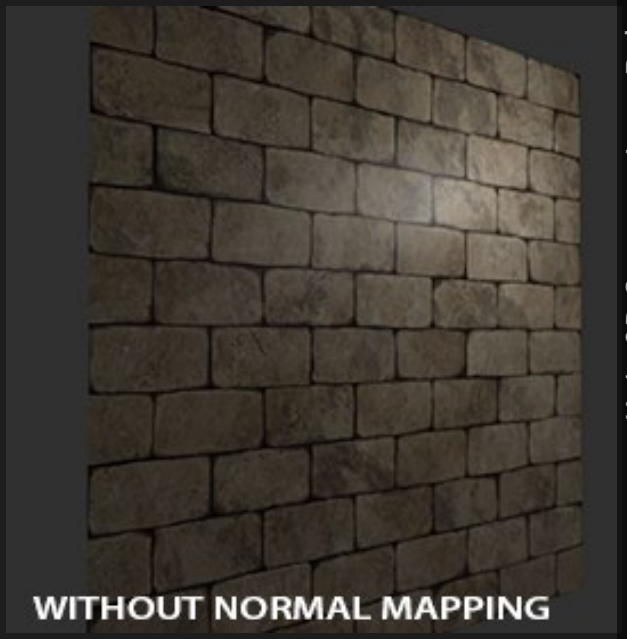
- Normal Mapping allows to add details to a lowpoly mesh
- Normals are encoded into an image, the Normal Map, using RGBA channels (A channel is used in some compressed Normal maps formats)
- Normal mapping is useless if there is no light: Normal map additional information is all about how light should be reflected on the mesh surface
- We can have a normal map encoded in WorldSpace or ObjectSpace

WorldSpace NormalMap

- NormalMap encodes normals baked in the 3D software world space
- When the mesh is imported in our game engine, it will exist in the game engine world space
    - The mesh position/orientation in our game engine will probably differ from the position/orientation it had in the 3D software during the normal map bake process
    - Even if we place the object at the same position/orientation, as soon as the object moves its normals will differ from those encoded in the normal map
        - There is no way to update those normals to the right value

ObjectSpace NormalMap

- NormalMap encodes normals baked in object space
- When the mesh is imported in our game engine, we can transform object space normals into world space normals using a transformation matrix (like Obj2World Matrix for vertices). This is the theory, but... There is a problem (see next slide)



Normal Map 2/2

ObjectSpace NormalMap (Theory)

- PShader
  1. Read ObjectSpace normal from the normal map
  2. Transform ObjectSpaceNormal to WorldSpaceNormal
  3. Use WorldSpaceNormal for shading calculations

ObjectSpace NormalMap (Practice)

- There is a problem with skinned mesh and non-uniform scaling
  - During triangle deformation, surface normal should deform itself [1], but we can't calculate this new normal if we start from ObjectSpaceNormal

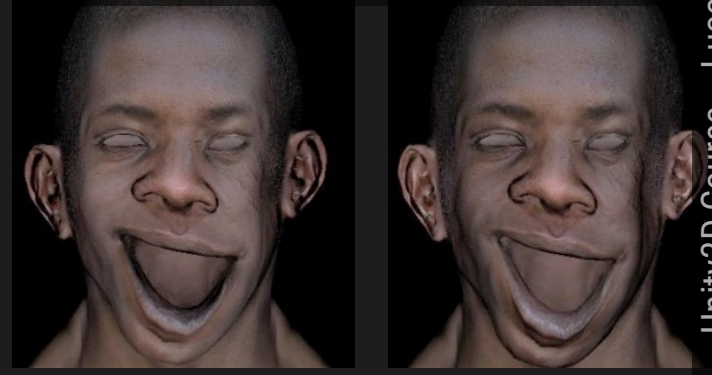
Solution: TangentSpaceNormal

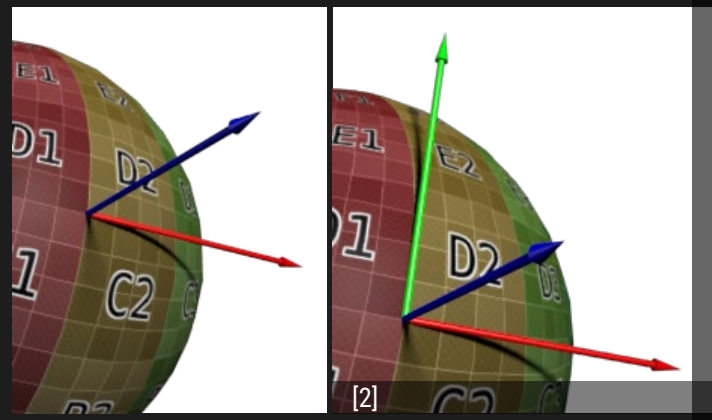
- Tangent Space is similar to UVSpace, but with 3 dimensions
  - Normal
  - Tangent, orthogonal to Normal, parallel to U axis
  - Binormal, orthogonal to Normal, parallel to V axis

- Activate TangentSpace
- If you use tangentSpaceVisualizer.cs to view these vectors, you'll notice that the 3 axis origin is on the darker corner of the cube (UV 0,0)
  - The script uses tangent.w to know binormal sign. In figure [2], if we have normal (blue) and tangent (red), binormal could orient towards North or South. Since V is growing towards North (see UV tiling C2, D2, E2, etc..), tangent.w will tell us the correct orientation
  - NB: we are not taking into account negative Transform scale values, this will be taken into account using the built-in var unity_WorldTransformParams.w in shader calculations

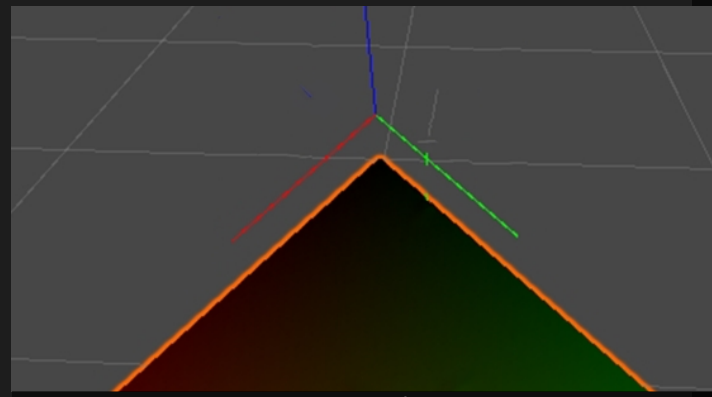
[Shaders_Start_C_01, tangentSpaceVisualizer.cs]

[1] Mesh is deformed using a blendshape for facial expression. Left: wrong normals: normals aren't updated. Right: normals are correct





[2]



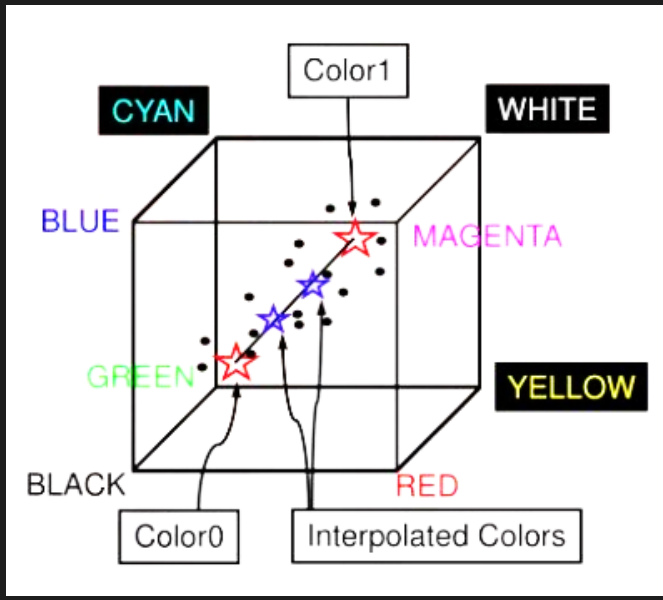
Tangent and binormal are oriented from the black corner towards red (U - Tangent) and green (V - Binormal) corner

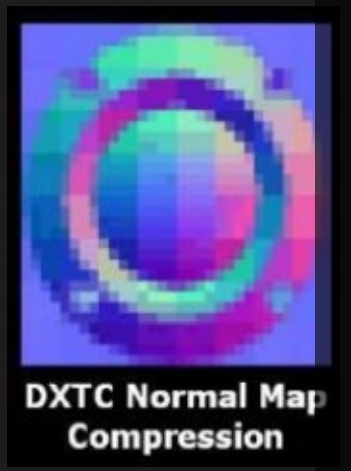
Normal DXT Compression

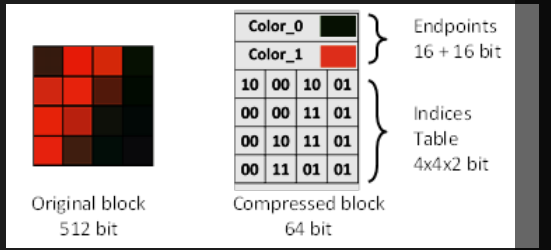
- Normals are encoded in RGB
- Colors range is [0,1]
- Normalized normals range is [-1,1]
- How to Map [0,1] to [-1,1]?
  - normal.xyz = (normalFromNormalMap.rgb * 2) - 1
- Usually NormalMaps have great details, and RGB pixel values change a lot between them
  - How to compress the NormalMap?

DXT Algorithm

- Must be taken into account for Normal decompression in shader code
  1. Divide the NormalMap in 4x4 texel blocks
  2. For each block, find 2 most present colors and save only them using 16 bit (RGB encoded in 5,6,5 bit)
  3. For each texel, save the distance from these 2 main colors encoding it in 2 bits
     - 16 + 16 + (4x4x2) = 64bit per 4x4 block //6x compressed ratio vs uncompressed img (which is 4x4x24 bits – uncompressed stores RGB using 8 bit per channel)
- We need to throw away one channel in order to improve compression quality (e.g. Save only 2 RG colors, encoding them in 8,8 bit)
  - By discarding the blue channel of the texture, we collapse the three dimensional RGB colorspace into a two dimensional red/green space. This increases the odds of a single line being a good fit for all the colors in a block, and thus reduces compression artifacts
- Let's say we save XY in RG, how to calculate Z?
  - Remember that normals are normalized: x*x + y*y + z*z = 1
  - From x,y values, our shader can compute: z = sqrt(1 - (x*x + y*y))







From TSNormal to WSNormal

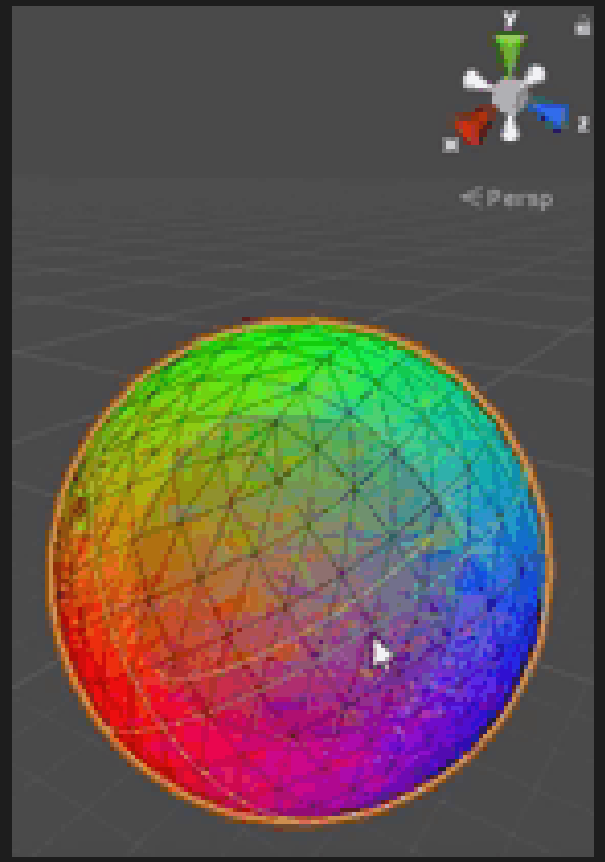
- VShader NORMAL and TANGENT inputs are in ObjectSpace, but we won't use VShader NORMAL input as the final normal value for that vertex: we need to sample the normal value from the NormalMap
- The Normal encoded in the NormalMap is in TangentSpace: to convert TangentSpace normals to WorldSpace normals we need the TBN Matrix
- We can build TBN Matrix from Vshader NORMAL and TANGENT inputs

Two ways to go:

- [1] If TBN Matrix is built from Normal and Tangent in ObjectSpace, then we'll be able to convert NormalMap TangentSpace normal into an ObjectSpace normal
  - ObjectSpace_normal = TBN x TangentSpace_normal
- [2] If TBN Matrix is built from Normal and Tangent in WorldSpace, then we'll be able to convert NormalMap TangentSpace normal into a WorldSpace normal
  - WorldSpace_normal = TBN x TangentSpace_normal

- We need a WorldSpaceNormal in the fragment shader, so we need to transform VShader NORMAL and TANGENT inputs (Object space) into World space [2]

$$\begin{pmatrix} T_x & B_x & N_x \\ T_y & B_y & N_y \\ T_z & B_z & N_z \end{pmatrix}$$

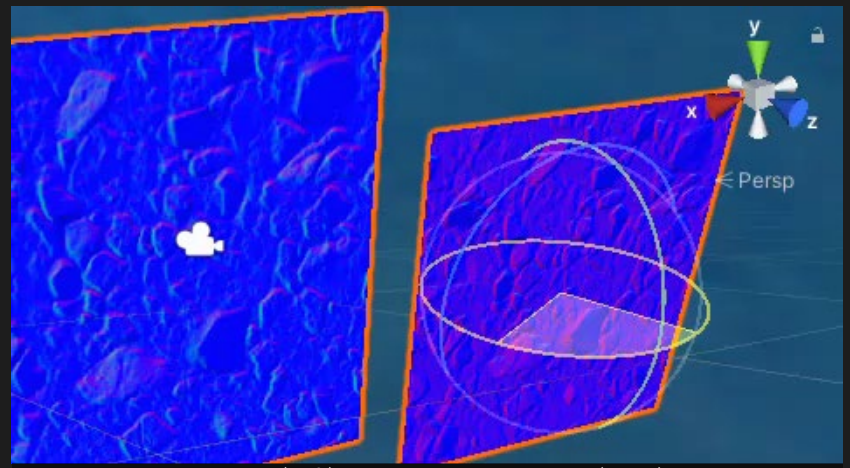


From TSNormal to WSNormal

VShader

1. start from Vertex ObjectSpaceNormal and ObjectSpaceTangent passed to the Vshader via VertexInput structure
   - float4 normal: NORMAL
   - float4 tangent: TANGENT
2. Transform them into WorldSpaceNormal and WorldSpaceTangent.
   Problem: For normals, to take into account non uniform scaling, we have to
   - Use the inverse transpose of unity_ObjectToWorld for normal
   - Use unity_ObjectToWorld matrix for tangent
3. Calculate WorldSpaceBinormal via cross product
4. Pass WorldSpaceNormal/Tangent/Binormal to the Pshader via VertexOutput structure
   - float4 normalWorld : TEXCOORD1;
   - float4 tangentWorld : TEXCOORD2;
   - float3 binormalWorld : TEXCOORD3; //Cross product is defined for 3D vectors

$$\begin{pmatrix} T_x & B_x & N_x \\ T_y & B_y & N_y \\ T_z & B_z & N_z \end{pmatrix}$$



TangentSpace normals (Left) vs WorldSpace normals (Right)

PShader (we have Normal, Tangent and Binormal in WorldSpace interpolated from the Rasterizer)

1. Build TBN Matrix
2. Sample TangentSpaceNormalMap to get TangentSpaceNormal
3. WorldSpaceNormal = TBN x TangentSpaceNormal //See [2] previous slide
4. Perform shading calculations

Shader matrices vector multiplication

Transforming directions

```
float3x3 mymatrix;
float3 direction;
float3 transformed_direction = mul(mymatrix, direction);
```

Transforming Normal Vectors

- We need the inverse transpose matrix
- We need to preserve normal.w after multiplication
- We need to normalize again the multiplication result

```
float3x3 matrix_inverse_transpose;
float4 normal;
float4 transformed_normal =
            normalize(float4(mul(matrix_inverse_transpose,
normal.xyz),normal.w));
```

If we don't have the inverse transpose but we have the inverse, just invert the multiplication order between Normal and the Matrix:

```
float4x4 matrix_inverse;
float4 normal;
float4 transformed_normal =
      normalize(float4(mul(normal.xyz, matrix_inverse.xyz), normal.w));
```

Build a Matrix

- To build a NxN Matrix

```
float3x3 m = float3x3(
  1.1, 1.2, 1.3, // first row
  2.1, 2.2, 2.3, // second row
  3.1, 3.2, 3.3  // third row
);
```

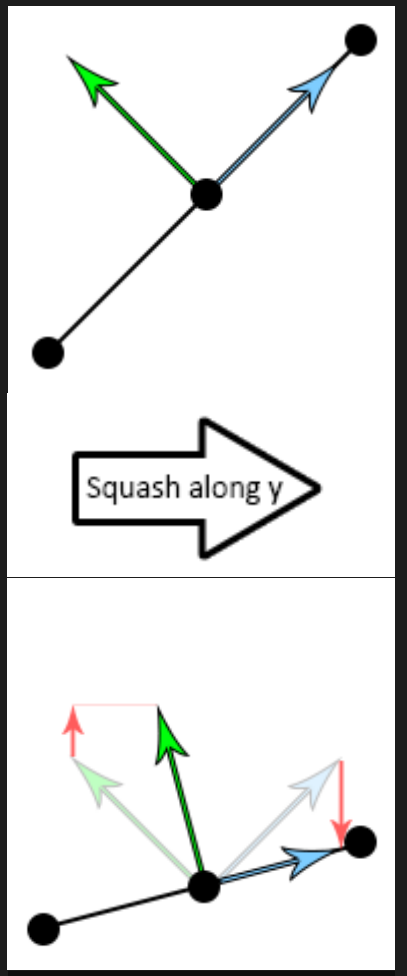
From ObjSpaceNormal/Tangent to WSNormal/Tangent

From ObjSpaceTangent To WorldSpaceTangent

- It is a simple transformation of a direction vector: mul(M,v)
- tangent.w is -1 or 1, if object has mirrored scale or not. We want to preserve tangent.w, because it will be used in binormal calculation
- WorldSpaceTangent = float4(normalize(mul((float3x3)unity_ObjectToWorld, v.tangent.xyz)), ObjSpaceTangent.w);
- [BIRP] To simplify, we can #include "UnityCG.cginc" and use:
  - WorldSpaceTangent = float4(UnityObjectToWorldDir(ObjSpaceTangent.xyz), ObjSpaceTangent.w);

From ObjSpaceNormal To WorldSpaceNormal

- Since normals are orthogonal to surface and non-uniform scaling might produce incorrect orthogonal values, we have to use the inverse transpose of unity_ObjectToWorld for normal
- We don't have the inverse transpose of that matrix, but the inverse of unity_ObjectToWorld is unity_WorldToObject
  - Hence we need to invert the multiplication order between Normal and the Matrix: mul(v,M)
- We want to preserve ObjSpaceNormal.w
- WorldSpaceNormal = float4(normalize(mul(normalize(v.normal.xyz), (float3x3)unity_WorldToObject)), ObjSpaceNormal.w);
- [BIRP] To simplify, we can #include "UnityCG.cginc" and use:
  - WorldSpaceNormal = float4(UnityObjectToWorldNormal(ObjSpaceNormal.xyz), ObjSpaceNormal.w);
- [URP] To simplify, we can use:
  - WorldSpaceNormal = float4(TransformObjectToWorldNormal(ObjSpaceNormal.xyz), ObjSpaceNormal.w);



The tangent is a dir on the surface and so if the y axis is squashed it does the same to follow the surface (hence we use obj2w matrix). The normal is a dir perpendicular to the surface and so if the y axis is squashed it moves the inverse way to stay perpendicular (hence we use inverse transpose matrix)

Binormal calculation

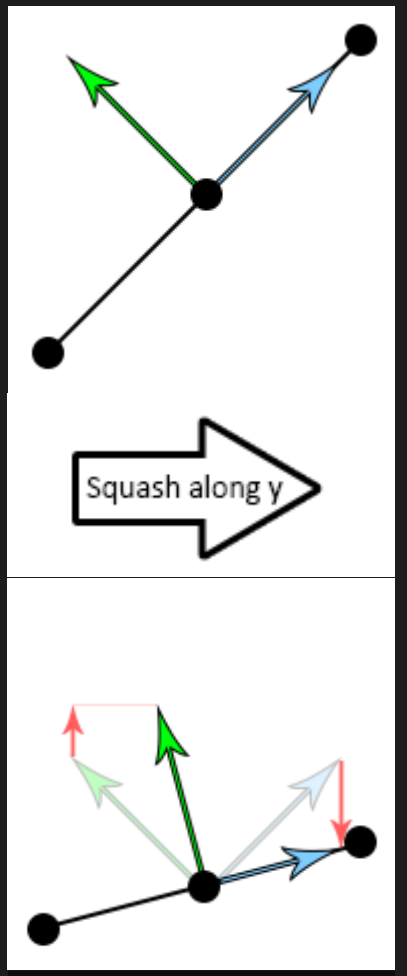
WorldSpaceBinormal calculation

- We use cross product between WorldSpaceTangent and WorldSpaceNormal
- Then, we need to perform two final multiplications in order to build a correct tangent space:
  - Tangent, binormal & normal are calculated using right-hand rule. If in the 3D software vertex UVs are flipped for any reason, handness of the tangent is flipped as well. To address this issue, we have to multiply by ObjSpaceTangent.w
  - If the obj is mirrored (has negative scale), we have to flip the binormal to correctly mirror the tangent space as well. We'll multiply the final result by unity_WorldTransformParams.w
- WorldSpaceBinormal =

normalize(cross(WorldSpaceNormal.xyz, WorldSpaceTangent.xyz) * ObjSpaceTangent.w) * unity_WorldTransformParams.w;

- unity_WorldTransformParams.w Suppose an object has its scale set to (-1, 1, 1). That means that it is mirrored. We have to flip the binormal in this case, to correctly mirror the tangent space as well. In fact, we have to do this when an odd number of dimensions are negative UnityShaderVariables helps us with this, by defining the float4 unity_WorldTransformParams variable. Its fourth component contains −1 when we need to flip the binormal, and 1 otherwise
- ObjSpaceTangent.w For OpenGL and DirectX this value is inverted. The binormal is the V of the UV, which is different in OpenGL and DirectX. OpenGL is bottom to top, and DirectX is top to bottom

Let's write our NormalMap shader!

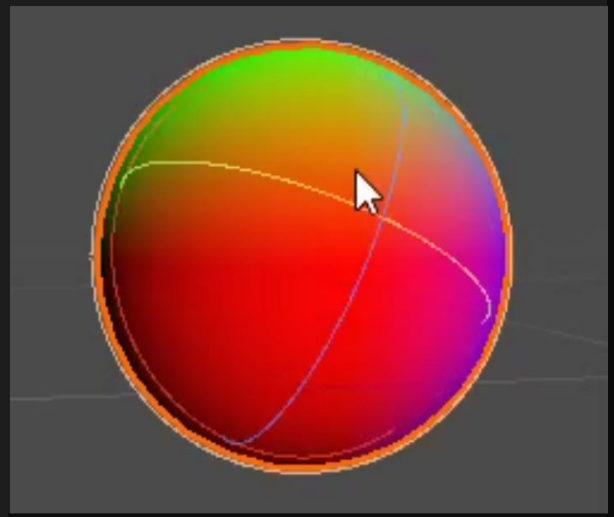


The tangent is a dir on the surface and so if the y axis is squashed it does the same to follow the surface (hence we use obj2w matrix). The normal is a dir perpendicular to the surface and so if the y axis is squashed it moves the inverse way to stay perpendicular (hence we use inverse transpose matrix)

NormalMap shader

Try it

- Activate NormalWorld_Start
- Create normal shader and material from texture shader, and assign it to Sphere
- We don't need _MipMap property: remove it
- Add _NormalMap Texture Input Property
- We need NORMAL and TANGENT input for Vshader, we don't have NORMAL and TANGENT semantic for Pshader, so we'll use TEXCOORD1,2,3 for interpolating normalWorld, tangentWorld, binormalWorld
- Add also TEXCOORD4 for normalTexCoord, in case you want to use _NormalMap_ST values
- Follow previous 'From ObjSpaceNormal To WorldSpaceNormal' slide to obtain o.normalWorld
- Follow previous 'From ObjSpaceTangent To WorldSpaceTangent' slide to obtain o.tangentWorld
- Follow previous 'WorldSpaceBinormal calculation' slide to obtain o.binormalWorld
- Calculate normalTexCoord
- Try to output the color of normalWorld in the Pshader. You should obtain the same color even if you rotate the sphere



Final color = normalWorld

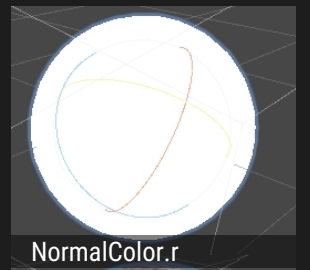
[Shaders_Start_C_01, _normal.shader]

NormalMap shader

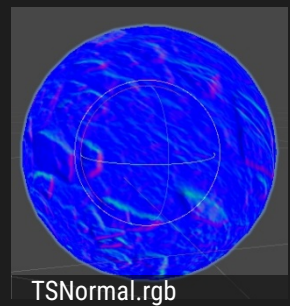
Try it (continue)
PShader

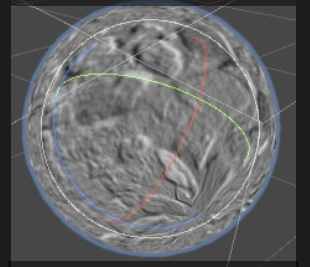
- Sample NormalColor using normalTexCoord
- If we use this value as the finalColor, we'll obtain a RedTexture. This is because in DXT5 compression we have:
  - R channel not used (filled with 1, hence the high red color component)
  - G used to encode Y
  - B not used (may contain same value of G channel)
  - A used to encode X
- From NormalColor calculate the correct value TSNormal of the Normal in TangentSpace using a custom function
  float3 normalFromColor(float4 Color)
  - You have to take into account if there is DXT compression or not. Use preprocessor keyword UNITY_NO_DXT5nm
  - In case of compression, you have TangentSpaceNormal.xy values encoded into AG NormalMap channels. Then you can calculate TangentSpaceNormal.z
- Build the TBNWorld matrix
- Calculate normalWorldAtPixel using
  - normalize(mul(TBNWorld, TSNormal));
- If you want to introduce a NormalStrenght value, add Property, variable and a multiply calculation:
  - _NrmScale ("Normal Scale", Float) = 1
  - normalDecompressed.xy *= _NrmScale;

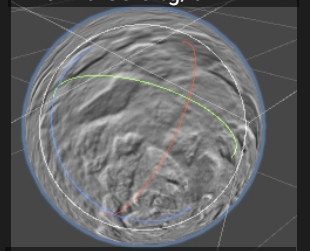
[_normal.shader]


NormalColor.r

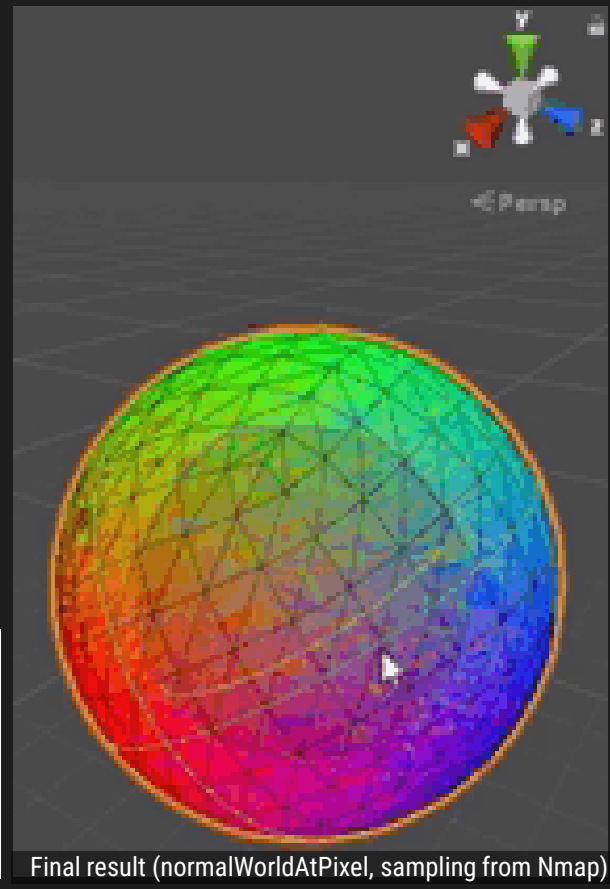

NormalColor.rgb


TSNormal.rgb


NormalColor.g/b


NormalColor.a

$$\begin{pmatrix} T_x & B_x & N_x \\ T_y & B_y & N_y \\ T_z & B_z & N_z \end{pmatrix}$$
TBN Matrix


Final result (normalWorldAtPixel, sampling from Nmap)

Adding simple Diffuse lighting

- Activate SimpleDiffuseWithNMapping
- Create _normalDiffuse.shader from _normal.shader

BIRP

```
Tags{"LightMode" = "ForwardBase"}
uniform float4 _LightColor0; //_LightColor0 is a built-in Unity variable. Since it is defined in UnityLightingcommon.cginc, we have to declare it as a uniform
float3 lightDir = normalize(_WorldSpaceLightPos0.xyz);
float3 lightColor = _LightColor0.xyz;
```

URP

```
#include "Packages/com.unity.render-pipelines.universal/ShaderLibrary/Lighting.hlsl"
Tags{"LightMode" = "UniversalForward"} //Pass scope
Light light = GetMainLight();
float3 lightDir = normalize(light.direction.xyz);
float3 lightColor = light.color;
```

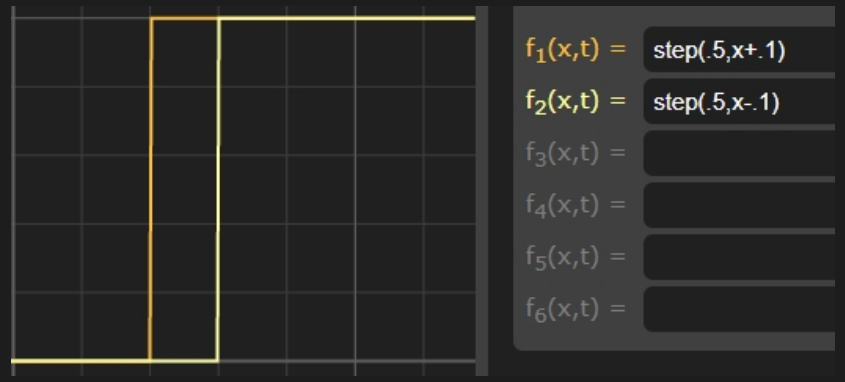
GetMainLight() in RealtimeLights.hlsl: direction is its position!

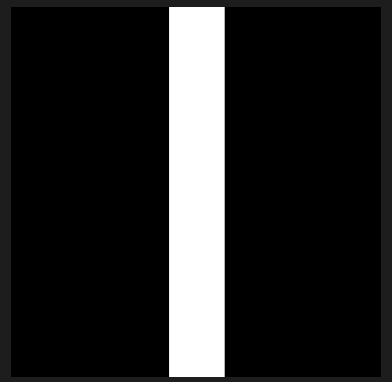
```
////////////////////////////////////////////////////////////////////////////////
//                      Light Abstraction                                     //
////////////////////////////////////////////////////////////////////////////////

Light GetMainLight()
{
    Light light;
    light.direction = half3(_MainLightPosition.xyz);
```

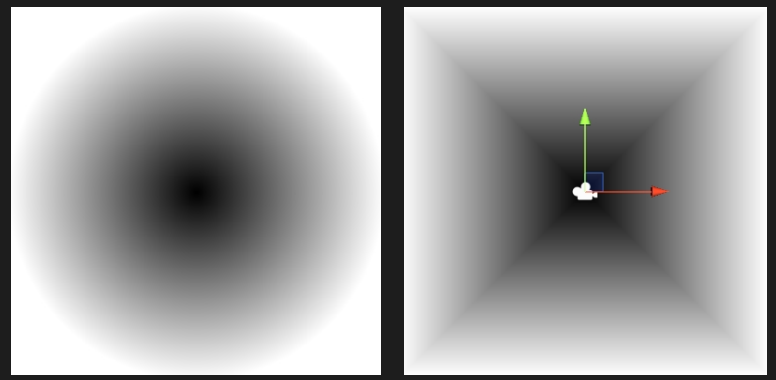
[_normalDiffuse.shader]

Intro to SDF Functions

- Stroke
  - Graphtoy formulas
- Circle
  - Try to return circle(), which is a SDF function
    - `(st-0.5)*2.0 // change range [0,1] > [-1,1]`
  - Then we stroke
- Rect
  - Try to return rect() SDF function
    - `st = st*2-1; // change range [0,1] > [-1,1]`
    - `max(abs(st.x/s.x), abs(st.y/s.y)) // Takes into account x distance from the center on the left/right, y d from the center on the top/bottom`
  - Then we stroke







[Shaders_Elements_Start_01, _elements.shader]

[BIRP] Culling

- Activate Culling
- What if we want to override standard Culling rule?
- Try to change _culling Cull modes
  - Cull
    - Back Back-Faces are culled
    - Front Front-Faces are culled
    - Off Render both
- NB: The cube is transparent on the edges because we have:
  - BlendOp Add
  - Blend SrcAlpha OneMinusSrcAlpha
  - This means: SrcColor * SrcAlpha + DstColor * OneMinusSrcAlpha
  - Source Alpha is 0 or 1. See import texture settings: on A channel there is black value on the edges the final result is that they are transparent, because
    - SrcColor * 0 + DstColor * 1

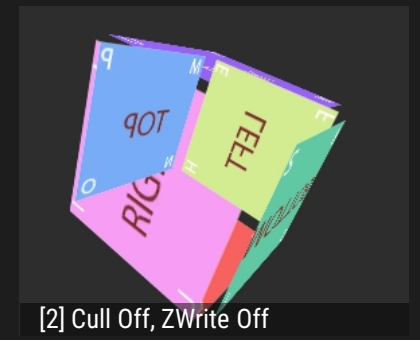
- Notice that Cull Off doesn't render the cube in the right way: faces rendering order doesn't allow it. In [1], the faces rendering order is (we can confirm this using Zwrite Off):
  - Red
  - Pink,Purple
  - Blue: when it renders, it finds the Zbuffer from the previous faces, and doesn't render on them (even if they have alpha=0 on borders)
  - Yellow: same as Blue
  - Green

[Shaders_Start_B_01, _culling, _cullingFrontBackH]

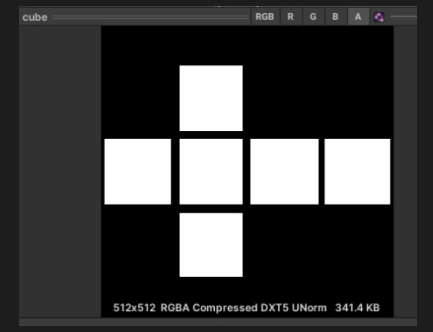
Cull Front

Cull Back

[1] Cull Off



[2] Cull Off, ZWrite Off



[BIRP] Outline Shader

Render.TransparentGeometry 4
RenderForwardAlpha.Render 4
RenderForward.RenderLoopJob 4
Draw Mesh Cube
Draw Mesh Cube
Draw Mesh Sphere
Draw Mesh Sphere

Try it

- Create textureOutline shader and material from texture shader, and assign it to the objects
- We don't need _MipMap property: remove it
- Add _OutlineColor and _OutlineBorder Properties
- First pass
  - Don't write ZBuffer
  - Vshader
    - Build the Scale matrix scaleM and calculate the scaled vertex pos scaledObjPos

```
float4x4 scaleMatrix = float4x4(
  sx, 0, 0, 0,        // first row
  0, sy, 0, 0,        // second row
  0, 0, sz, 0,        // third row
  0, 0, 0, 1          // fourth row
);
```

  - PShader
    - Color output is _OutlineColor
- Second pass
  - You can use Vshader & Pshader from texture shader

$$\begin{bmatrix} sx & 0 & 0 \\ 0 & sy & 0 \\ 0 & 0 & sz \end{bmatrix} * \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} x' \\ y' \\ z' \end{bmatrix}$$

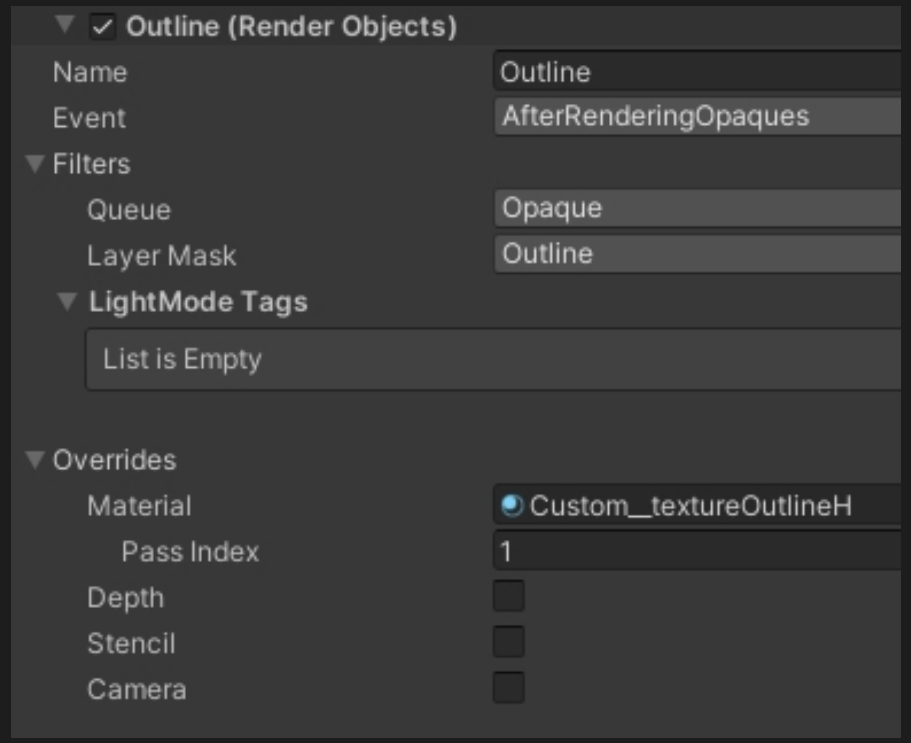
- This is the simplest way to achieve this effect. For convex geometry, it is better to inflate the mesh using vertices normals
- NB: This shader could have some problems with Tags Geometry/Opaque. Use Transparent queue instead

[Shaders_Start_B_01, _textureOutline.shader]

[URP] Outline Shader

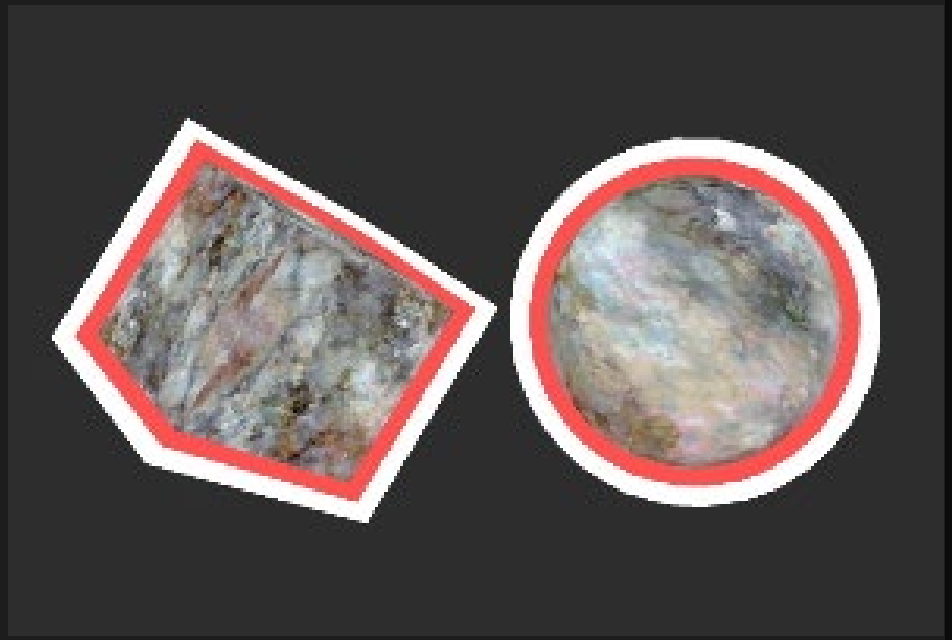
- Use the same technique used for Double culling cube [BIRP]
- Add a RenderObjects feature to the UniversalRenderData asset, use a different PassID
  - Use the same material textureOutlineH , add Objs to a new Outline layer, and specify PassIndex 1 in the UniversalRenderData
- Notice that this pass is performed AFTER the first passes, so we don't have Z fighting between objects that you can see in BIRP!

[Shaders_Start_B_01, _textureOutline.shader]



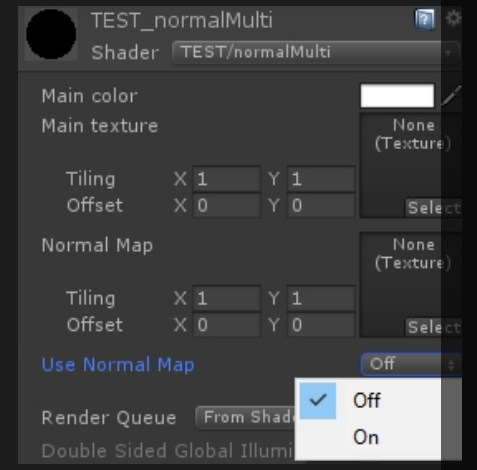
HLSLINCLUDE/ENDHLSL

- Inside the SubShader we can also use HLSLINCLUDE to include the code in every Pass inside that SubShader
- This is very useful for writing shaders in URP as every pass needs to use the same UnityPerMaterial CBUFFER to have compatibility with the SRP Batcher and this helps us reuse the same code for every pass instead of needing to define it separately. We could alternatively use a separate include file instead too
- The 2 outline assignment has 3 passes, every pass is very similar to the other, so we can make use of this. See [TextureOutlineH]



Multivariant shader, include files

- Multivariant shader are capable of compiling different portions of code depending on inspector input
- To create a MVariantShader
  1. Define one or more properties KeywordEnum, Toggle
  2. Use #pragma [multi_compile | shader_feature]
  3. Use #if #else #endif to isolate feature-dependent code
- To create inspector menus or toggles
  - [KeywordEnum(option1Name, option2Name, ...)]] _varName ("varLabel", Float) = 0 //No more than 9 options
  - [Toggle] _varName ("varLabel", Float) = 0 //0: Toggle OFF, 1: Toggle ON
- To use multivariant properties
  - #pragma [multi_compile | shader_feature] VARNAME_OPTION1NAME VARNAME_OPTION2NAME
    - multi_compile Shader is compiled N times, 1 for each multivariant option
      - If we have 2 Toggles, 4 shaders will be compiled (one for each combination)
      - Each shader is included in the final build, even if not used by the scene assets
    - shader_feature
      - Only Multivariant inspector values are compiled, resulting in a single shader
- To use .cginc/.hlsl files [BIRP/URP]
  - #include "filename.cginc/hlsl"
  - "UnityCG.cginc" already includes:
    - "UnityInstancing" instancing support to reduce draw calls
    - "UnityShaderVariables" variables that are necessary for rendering, like transformation, camera, and light data, which are all set by Unity when needed
    - "HLSLSupport" allows you not to worry about using platform-specific data types



Multivariant shader, .cginc files

Try it

- Start from normal shader and create normalMulti shader and material
- Add this property
  - [KeywordEnum(Off,On)] _UseNormal("Use Normal Map", Float) = 0
- Add #pragma shader_feature _USENORMAL_ON _USENORMAL_OFF
- Add #if _USENORMAL_ON #else #endif to isolate feature-dependent code
  - What shader code lines do we need to calculate NormalMap?

- Start from normalMulti and create normalMultiInc
- Create CGLighting.cginc in the same folder of your shaders
- Search for general-use functions (NormalMap calculations, etc):
  - normalFromColor
  - If you want, create and use float3 WorldNormalFromNormalMap(sampler2D normalMap, float2 normalTexCoord, float3 tangentWorld, float3 binormalWorld, float3 normalWorld)
  - These 2 functions will be easily reused in our shaders: cut/paste them into CGLighting.cginc
  - Add #include "Lighting.cginc/hlsl"

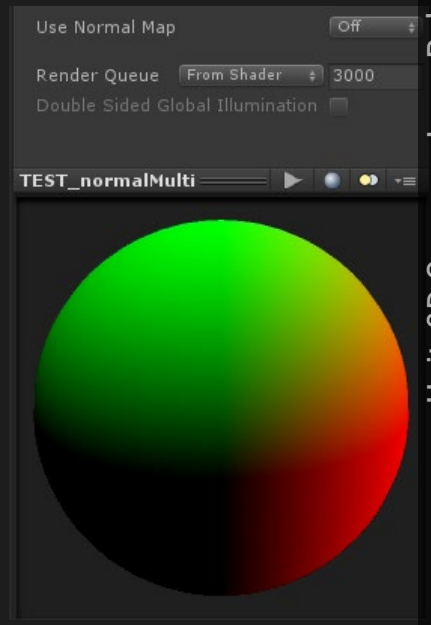
- Try to duplicate 2 times the line #include "CGLighting.cginc": this will result in a redefinition of 'functionName' error. To avoid this, add at the beginning of the .cginc/hlsl file:

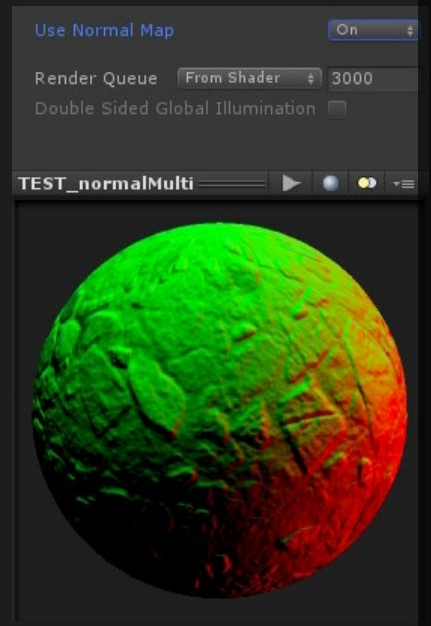
```
#ifndef LIGHTING
#define LIGHTING
```

- And at the end:

```
#endif
```

[Shaders_Start_D_01, NormalMulti_Start, normalMulti.shader, _normalMultiInc.shader, CGLighting.cginc]





Material property drawers

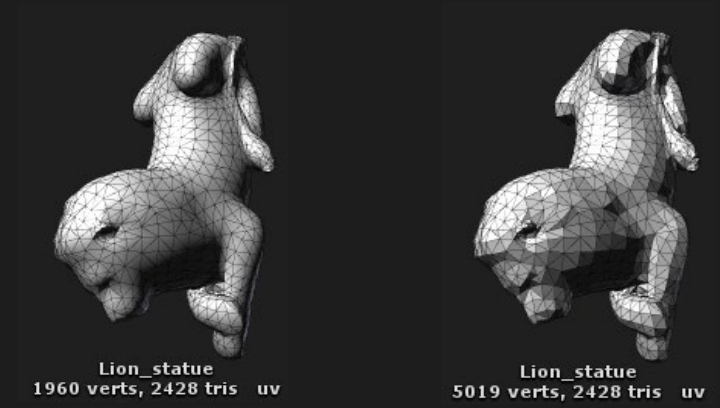
- Toggle, Enum, KeywordEnum
- PowerSlider These drawers are quite useful when working with numerical ranges and precision. On the one hand, we have the PowerSlider, which allows us to generate a non-linear slider with curve control
- IntRange
- Space allows us to add space between one property and another. If we wish that our properties appear separate in the material inspector, we can add space between them
- Header adds a header in the Unity Inspector. This is very useful when generating categories in our properties

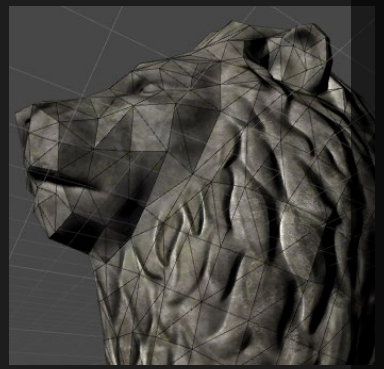
Basic Lighting model

- Objects – light interaction is modeled from the Lighting Model, and is based on
  - Light properties
  - Object material properties

- Basic Lighting model
  - surfaceColor = emissive + ambient + diffuse + specular

- Gouraud shading
  - Calculate shading only on vertices, and then rasterizer will interpolate shading values for each fragment
- Phong shading
  - Calculate shading for each fragment
- Flat Shading
  - Use only surface normal to shade the entire triangle
    - Use a geometry shader to calculate triangle surface normal
    - Use a script to analyze the mesh and duplicate vertices to simulate hard edges
    - Use Unity Inspector Import settings to create hard edges (new vertices are created)





Forward lighting

- N objects, M lights = N x M DrawCalls
- Each Draw call is a shading pass on fragment object, they are blended together for the final result
- We saw in outline shader that each Pass is a draw call
- Forward lighting uses 2 Passes
  - [BIRP/URP] ForwardBase/UniversalForward (performed once)
    - Perform lighting calculations for: ambient, main directional light, vertex/SH lights and lightmaps
  - [BIRP] ForwardAdd (performed N times, if this object is lit by N lights)
- During lighting calculations we'll use Unity built-in variables. If we don't specify Pass Tag variables in the correct way, these variables will not be correctly initialized
  - Pass{ Tags{"LightMode" = "[ForwardBase | ForwardAdd | UniversalForward]"} …}

Adding simple Diffuse lighting

BIRP

- `Tags{"LightMode" = "ForwardBase"}`
- `uniform float4 _LightColor0; //_LightColor0 is a built-in Unity variable. Since it is defined in UnityLightingcommon.cginc, we have to declare it as a uniform`
- `float3 lightDir = normalize(_WorldSpaceLightPos0.xyz);`
- `float3 lightColor = _LightColor0.xyz;`

URP

- `#include "Packages/com.unity.render-pipelines.universal/ShaderLibrary/Lighting.hlsl"`
- `Tags{"LightMode" = "UniversalForward"} //Pass scope`
- `Light light = GetMainLight();`
- `float3 lightDir = normalize(light.direction.xyz);`
- `float3 lightColor = light.color;`

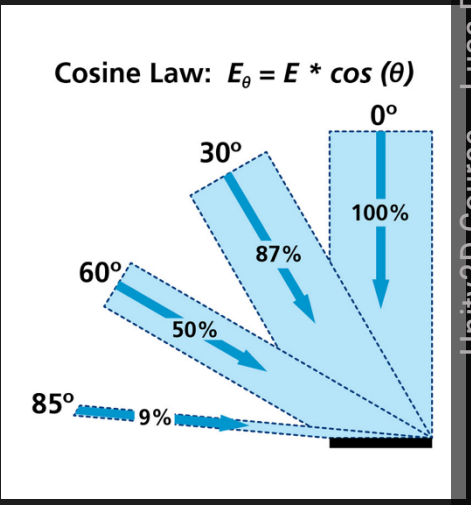
```
////////////////////////////////////////////////////////////////////////////////
//                      Light Abstraction                                     //
////////////////////////////////////////////////////////////////////////////////

Light GetMainLight()
{
    Light light;
    light.direction = half3(_MainLightPosition.xyz);
```

[_normalDiffuse.shader]

Diffuse reflection

- Why we are able to see objects?
  - Light is emitted, it reflects on the obj surface, and then reaches our eyes
- Diffuse reflection
  - Light is reflected uniformly in every direction
  - Lambert cosine law: the diffuse reflection is directly proportional to the cosine of the angle between the dir of light L and the surface normal N
    - This is exactly what the result of the dot product dot(N,L). If N and L are normalized, it will become simply dot(N,L) = cos(α)
    - Since opposite values have negative cos(), our equation becomes
      - diffuseReflection = max(0,dot(N,L))
- Surface Diffuse Factor
  - Besides diffuse reflection, we have to take into account also
    - light intensity
    - diffuse material property
    - light attenuation
  - Diffuse = light_intensity * material_diffuse_factor * light_Attenuation * diffuse_reflection

- [BIRP] In our shader code we'll use these unity built-in variables
  - float4 _LightColor0 Light color
  - float4 _WorldSpaceLightPos0 Directional lights: (world space direction, 0). Other lights: (world space position, 1)
- For [URP], see prev slide



$$\mathbf{a} \cdot \mathbf{b} = \|\mathbf{a}\| \, \|\mathbf{b}\| \cos(\theta)$$

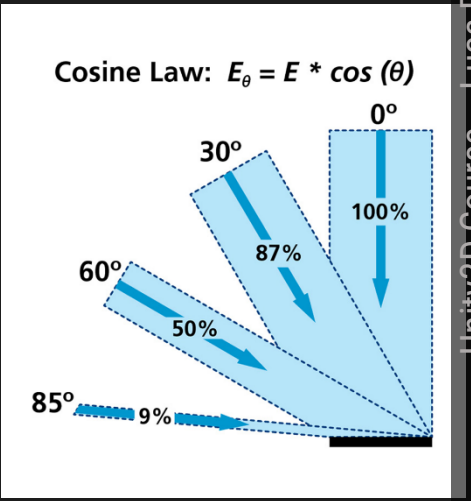
[Shaders_Start_D_01, LightingDiffuse_Start, lightingDiffuse.shader]

Diffuse reflection

Try it

- Start from normalMulti shader and create lightingDiffuse shader and material
- Add Tags{"LightMode" = "ForwardBase"} inside the FIRST pass. Now we have _LightColor0 and _WorldSpaceLightPos0 variables correctly initialized:
  - _LightColor0 this is a Property in the auto-included UnityLightingcommon.cginc – we must use it as uniform var
  - _WorldSpaceLightPos0 – do NOT use this as uniform. Its w value is 0 for dir light, 1 otherwise
- Add a _Diffuse float parameter and a [KeywordEnum(Off, Vert, Frag)] _Lighting for Lighting mode
  - _LIGHTING_VERT
    - Add float4 surfaceColor : COLOR0; to Vshader output
    - Add lighting calculations in Vshader
    - In Fshader, use surfaceColor as output
  - _LIGHTING_FRAG
    - Add lighting calculations in Fshader
    - _USENORMAL_ON
      - Use normalWorldAtPixel as normal
    - _USENORMAL_OFF
      - Use Vshader output normalWorld as normal
- Create float3 DiffuseLambert(float3 normalVal, float3 lightDir, float3 lightColor, float diffuseFactor, float attenuation) function for lighting calculations, and move it into CGLighting.cginc

[Shaders_Start_D_01, LightingDiffuse_Start, _lightingDiffuse.shader]

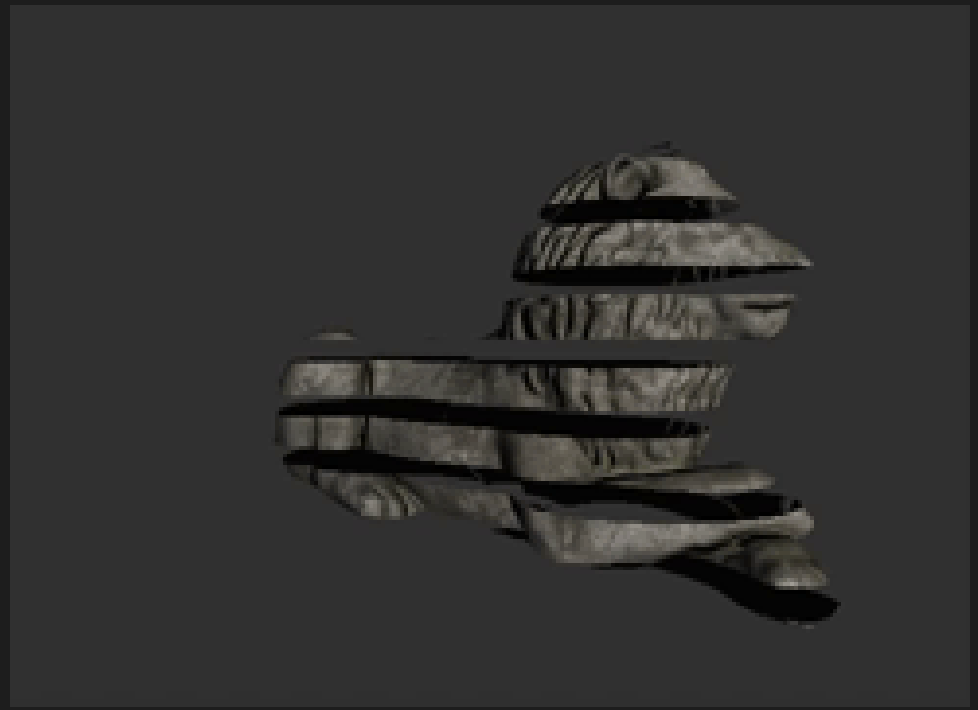


$$\mathbf{a} \cdot \mathbf{b} = \|\mathbf{a}\| \, \|\mathbf{b}\| \cos(\theta)$$

Ex // Alpha Stripes

- Start from lightingAmbient.shader
- Add two float properties: _Stripes and _StripeCutOff
  - _Stripes is linked to the amount of horizontal stripe subdivisions
  - _StripeCutOff is linked to the height of each stripe
- Pay attention to face culling for the lion internal faces!

- Reference [bit.ly/Ex_ALPHA_STRIPES]

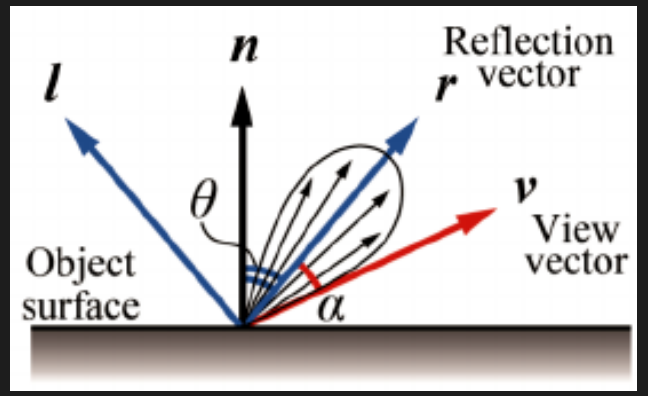


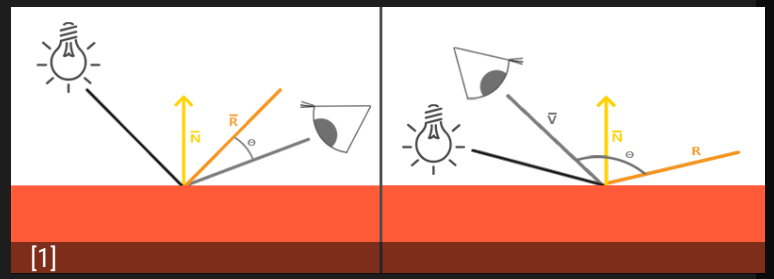
[Shaders_Start_D_01, AlphaStripes_Ex_Start, _alphaStripes.shader]

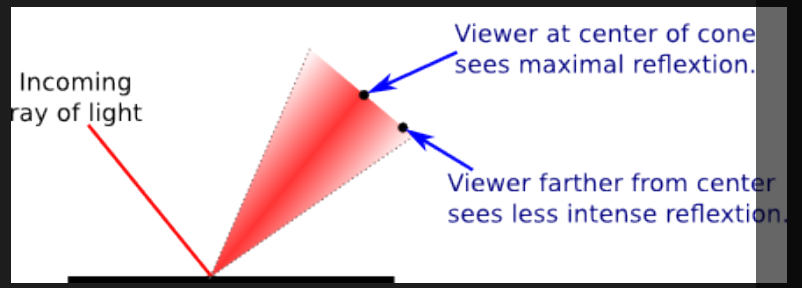
Specular reflection

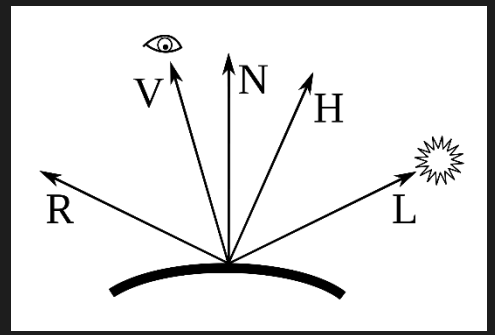
- Law of reflection The angle of incident light is equal to the angle of reflected light
  - Specular reflection is view-point dependent
  - We need
    - L light vector – from surface to light //calculate from _WorldSpaceLightPos0
    - N normal vector //We have this from VShader input struct
    - V view vector – from surface to viewer //Calculate from _WorldSpaceCameraPos
  - R reflection vector
- Specular reflection cone View vector V farther from Reflection vector R sees less intense reflection
  - Specular reflection = max(0, dot(R,V)) //Same as Diffuse reflection (for now)
  - Phong observed that there is a rapid falloff inside the specular reflection cone, proportional to $\cos(\alpha)^2$
  - Specular reflection = max(0, dot(R ,V))^2 //Phong reflection model
  - We don't have R, but we could calculate it using
    Specular reflection = max(0, dot(2*dot(N,L)-L ,V))^2
  - There is a problem: there might be cases in which dot(R,V) is 0 (see [1]), but if specular value is low, we should get a specular component > 0. In these cases, Phong reflection model doesn't work
  - Solution: Bling-Phong model, which is also simpler, using H, an half-way vector between L and V:
    H = L+V
  - Specular reflection = max(0, dot(N,H))^2 //Blinn-Phong reflection model
- Specular = light_intensity (specular map) * material_specular_factor (0,1) * light_Attenuation * specular_reflection

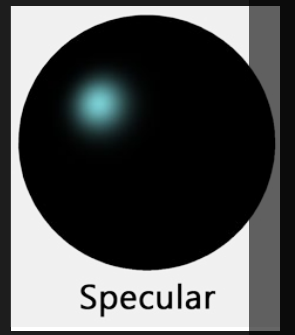
[Shaders_Start_D_01, LightingSpecular_Start, lightingSpecular.shader]

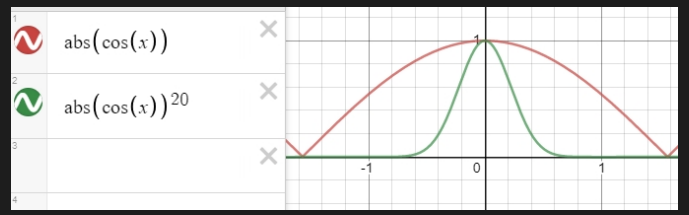












Specular reflection

Try it

- Add _SpecularMap, _SpecularFactor, _SpecularPower Properties
  - _SpecularMap default value should be Black
- Create float3 SpecularBlinnPhong(float3 N, float3 L, float3 V, float3 specularColor, float specularFactor, float attenuation, float specularPower) function for lighting calculations, and move it into CGLighting.cginc

- _LIGHTING_VERT
  - Sample _SpecularMap using tex2Dlod()
  - Calculate V vector from
    - transforming v.vertex in worldSpace
    - _WorldSpaceCameraPos.xyz
  - Save DiffuseLambert method return value in diffuseColor, and SpecularBlinnPhong return value in specularColor
    - o.surfaceColor will be the sum of this 2 values
- _LIGHTING_FRAG
  - Output posWorld, the position of v.vertex in worldSpace
  - Sample _SpecularMap using tex2D()
  - Calculate V vector from
    - posWorld
    - _WorldSpaceCameraPos.xyz
  - Save DiffuseLambert method return value in diffuseColor, and SpecularBlinnPhong return value in specularColor
    - Return the sum of this 2 values
  - You can use Lion Occlusion texture as the SpecularMap, to better appreciate the specular effect
  - You can multiply the diffuse color by the albedo color (use Lion_Statue_Albedo texture)
    - finalColor = texture_albedo_color * diffuse + specular

[Shaders_Start_D_01, LightingSpecular_Start, lightingSpecular.shader]

Metallic workflow

- 3 components: Color + Microsurface + Metalness

Color BaseMap (RGB) Defines the color of the object, no matter the type of material. Usually match perceived obj color

Metalness MetallicMap (R) Defines metal (1) or non metal (0) areas

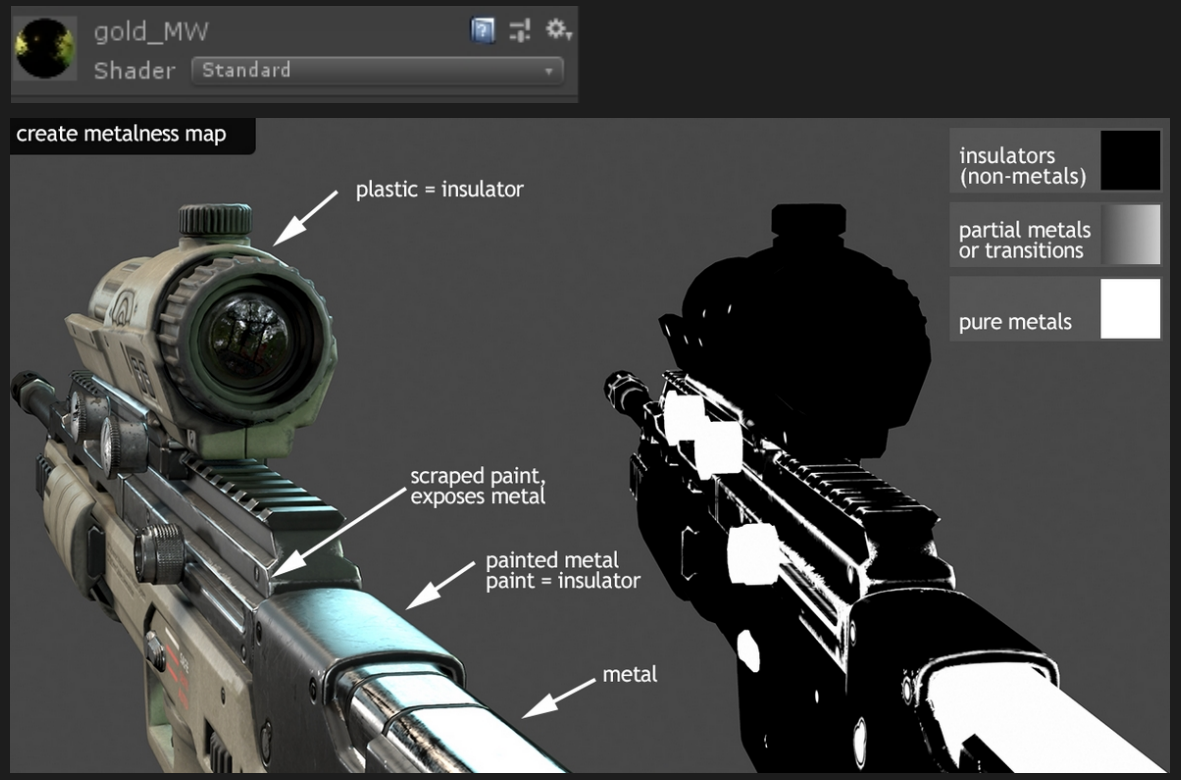
- Metal areas reflect 60/90% of the incoming light, while Non-metal areas reflect about 20% of the incoming light
- On metal areas, Skybox contribute is higher than Albedo
- Usually Metalness Map color is black or white. Grayscale areas are used only in transition between Metal & Non-metal

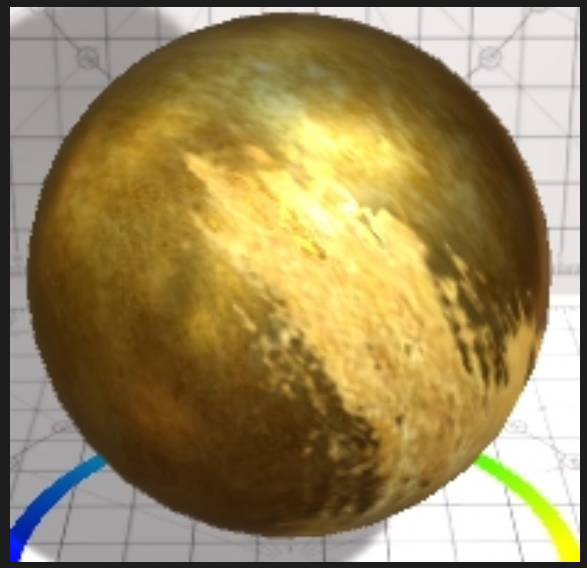
Microsurface Smoothness (A) (Gloss Map) defines gloss (1) or rough (0) areas

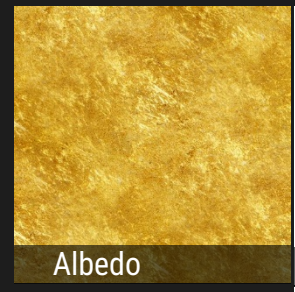
Reflectivity Specular reflection color is given by Albedo texture. In fact, metal surfaces may appear as tinted (gold = yellow)

- Use less texture memory than Specular Workflow (Metalness is B&W)

[Materials_01/Metallic/Specular Workflow]







Specular workflow URP

- 3 components: Color + Microsurface + Specularity

Color BaseMap (RGB) Perceived obj color for Non-metal areas, black for Metal areas

Specularity SpecularMap (RGB) Defines metal color, dark grey for Non-metal areas

- Brighter areas have high reflection

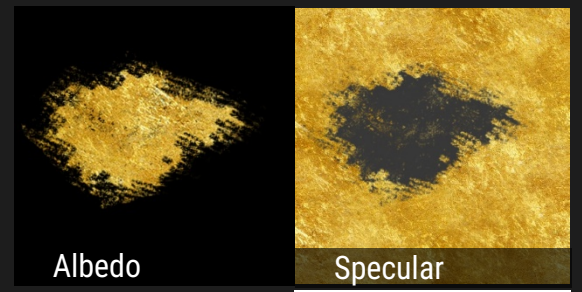
Microsurface Smoothness (A) (Gloss Map) defines gloss (1) or rough (0) areas

Reflectivity Specular reflection color is given by Specular Map

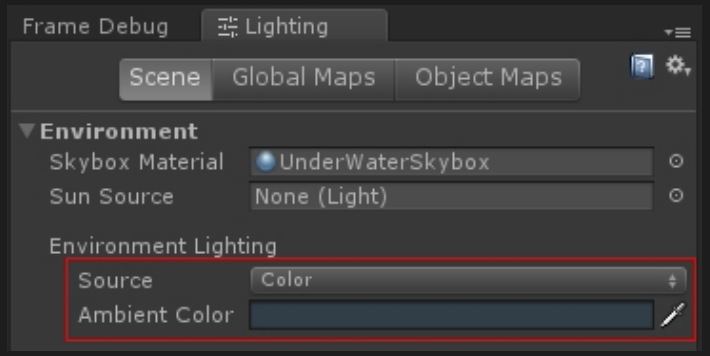
An easy way to know which workflow someone is using is to look at the Albedo texture to see if it contains metallic color or not

- Use more texture memory than Metallic Workflow (Specular is RGB)

[Materials_01/Metallic/-Specular Workflow]

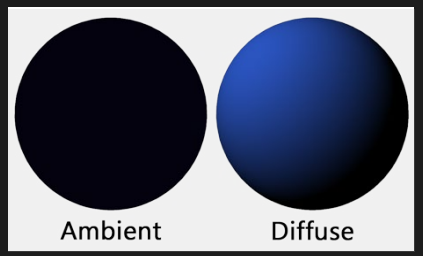


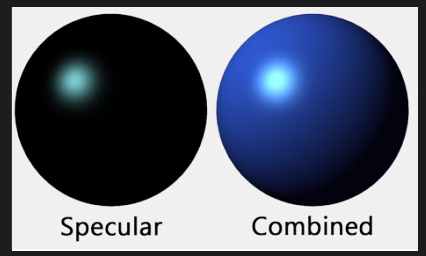
Ambient illumination + Albedo



- Simulates Global Illumination
- We can change the Ambient lighting in the scene in Lighting/Scene/EnvironmentLighting/Color/AmbientColor

- [BIRP] ambient = material_ambient_factor * UNITY_LIGHTMODEL_AMBIENT
- [URP] ambient = material_ambient_factor * half3(unity_SHAr.w, unity_SHAg.w, unity_SHAb.w)
  - Spherical harmonics coefficients (used by ambient and light probes) are set up for UniversalForward pass,. They contain 3rd order SH. The variables are all half4 type, unity_SHAr/g/b. Plain ambient light contain in the L0 coefficient (.w) of spherical harmonic



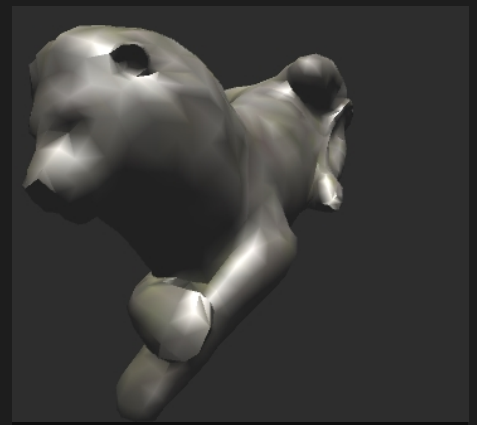


Try it
- Add [Toggle] _AmbientMode, _AmbientFactor Properties, along with its shader_features
- Add Ambient lighting calculation in Vshader or FShader

Finally, we get:
- finalColor = texture_albedo_color * diffuse + specular + ambient

If we want to add a texture albedo also in case of _LIGHTING_VERT, we have to add this albedoColor contributionin the Vshader:
- half4 albedoColor = tex2Dlod(_MainTexture, float4(o.texcoord.xyz, 0));

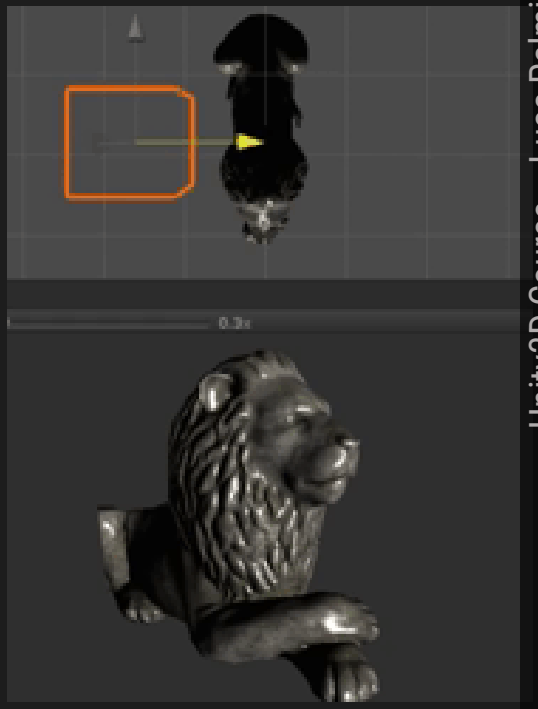
Try to add also the use of normal mapping in case of _LIGHTING_VERT_ON



Vertex lighting with albedo sampled per-vertex, and diffuse, specular and ambient contribution

[Shaders_Start_D_01, LightingAmbient_Start, _lightingAmbient.shader]

StencilBuffer



Using Stencil buffer we are able to discard fragments before they arrive to the frame buffer, like we do with Z-Buffer

To make use of the Stencil buffer

- Think about rendering order – Use “Queue” Tag
- Use Stencil block

```
Stencil {
        Ref ref_value
        Comp comparison_func [Greater | GEqual | Less | LEqual | Equal | NotEqual | Always | Never]
        Pass pass_action [Keep | Zero | Replace | IncrSat | DecrSat | Invert | IncrWrap | DecrWrap]
        Fail fail_action
        ZFail zfail_action
}
```

- Ref The value to be compared against and the value to be written to the buffer for replace action
- Comp Function used to compare ref_value to the current contents of the buffer. Default: always
- Pass/Fail/ZFail What to do with the contents of the buffer if
  - the stencil test (and the depth test) passes. Default: keep
  - the stencil test fails. Default: keep
  - the stencil test passes, but the depth test fails. Default: keep
- IncrSat/DecrSat +1/-1, clamp to [0,255]
- IncrWrap/DecrWrap +1/-1 loops: if IncrWrap on 255, becomes 0
- Invert Negate all the bits

StencilBuffer

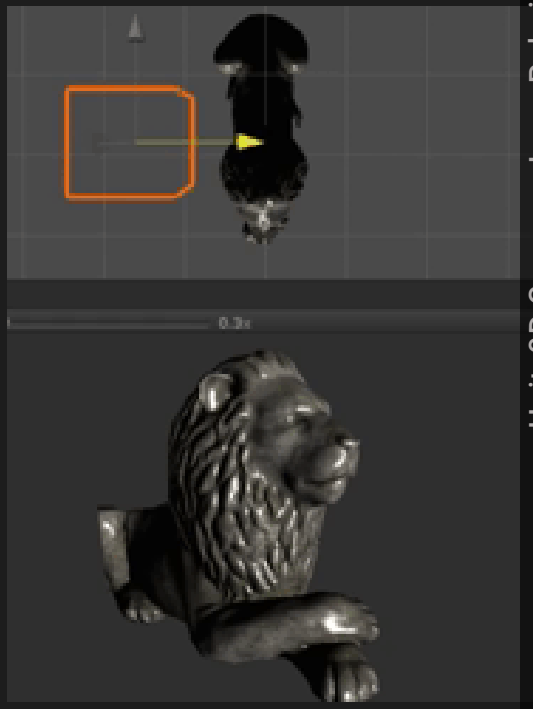
- Create lightingStencil.shader from lightingAmbient.shader, and assign the material to the Lion
  - Add Stencil block: render only if notequal to 1
- Create writeStencil.shader from firstShader.shader, and assign the material to the Cube
  - It is rendered before the lion ("Queue"="Geometry-1“)
  - [A] It writes 1 to the stencil buffer
  - [B] ColorMask 0 avoids that its fragments arrive to the frame buffer
  - [C] ZWrite Off avoids that the sphere fails to draw its fragments because of ZTest

- When the Lion is drawn
  - Its fragments arrive to the frame buffer only if stencil buffer is != 1

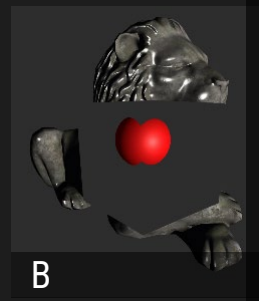
- The Sphere is drawn with standard shader: it is inside the lion, we can’t see it unless we use the cube to see inside the Lion

- What if we want to use the cube as “make the lion transparent” filter?
  - Simply add a pass in lightingStencil.shader
    - the lion renders only if the stencil test is equal
    - The final alpha value is 0.3
- [URP] remember that in order to perform a multipass with a material, you need to create a ne RenderObjects URPAsset RenderFeature!

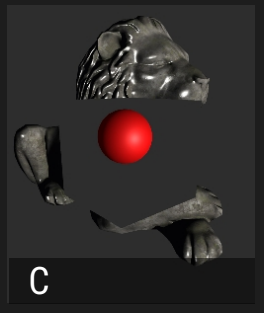
[Shaders_Start_E_03, StencilBuffer_Start, _lightStencil.shader, _writeStencil.shader]



A



B



C

Useful resources

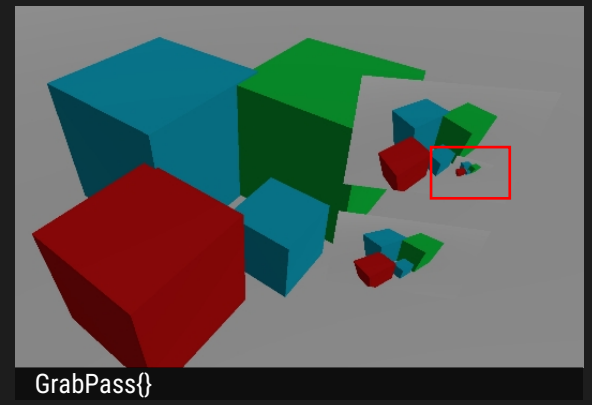
- http://tobyschachman.com/Shadershop/
- https://graphtoy.com/
- https://www.shadertoy.com/
- The Cg Tutorial

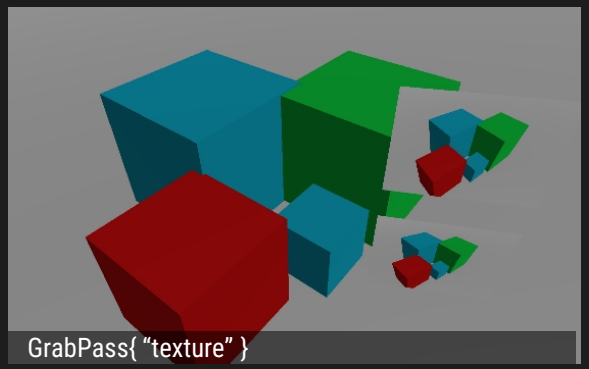
[BIRP] GrabPass

- This Pass grabs the contents of the screen where the object is about to be drawn into a texture
- Create grabScreen.shader from _texture

To make use of GrabPass
- Add the Pass
  - GrabPass { “TextureName” }
- Add the texture global variable
  - Sampler2D TextureName
- Sample TextureName in your shader code

- NB
  - It is a shader pass: it is performed when the subshader Grabpass is performed (try with a Queue = Geometry value)
  - GrabPass{} grabs the current screen contents into a texture. The texture can be accessed in further passes by _GrabTexture name. Note: this form of grab pass will do the time-consuming screen grabbing operation for each object that uses it
  - GrabPass { "TextureName" } grabs the current screen contents into a texture, but will only do that once per frame for the first object that uses the given texture name. The texture can be accessed in further passes by the given texture name. This is a more performant method when you have multiple objects using GrabPass in the scene (try to duplicate the GrabQuad and use the 2 variant: with texture name and without)





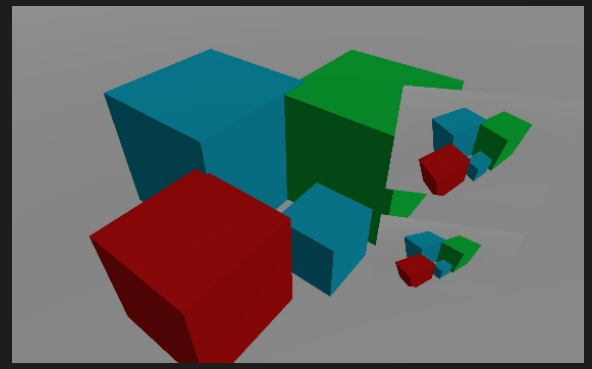
[Shaders_Start_E_03, GrabPass_Start, grabScreen.shader]

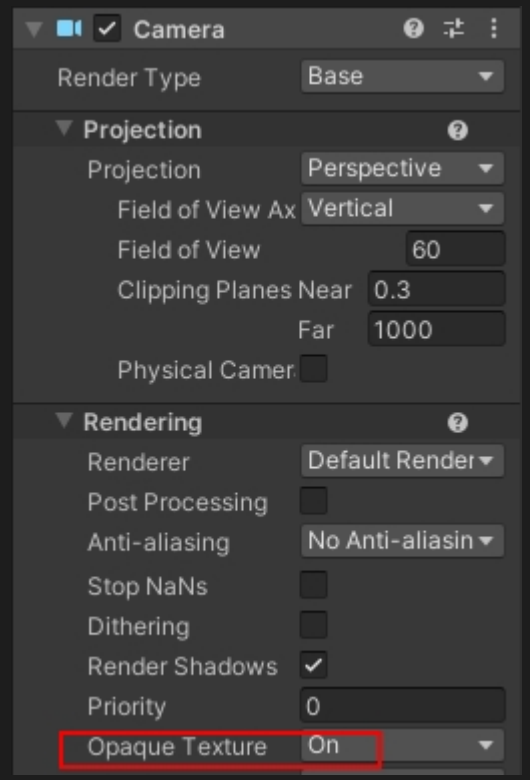
[URP] GrabPass

- GrabPass is not available in URP, but we have a camera property that can render Opaque objects into a texture
- Create grabScreen.shader from _texture

- Make sure that our quad is Transparent
- We need to set Camera/Rendering/OpaqueTexture ON in order to sample
  uniform sampler2D _CameraOpaqueTexture;
- Then we can use _CameraOpaqueTexture as the albedo texture

[Shaders_Start_E_03, GrabPass_Start, GrabPassH.shader]



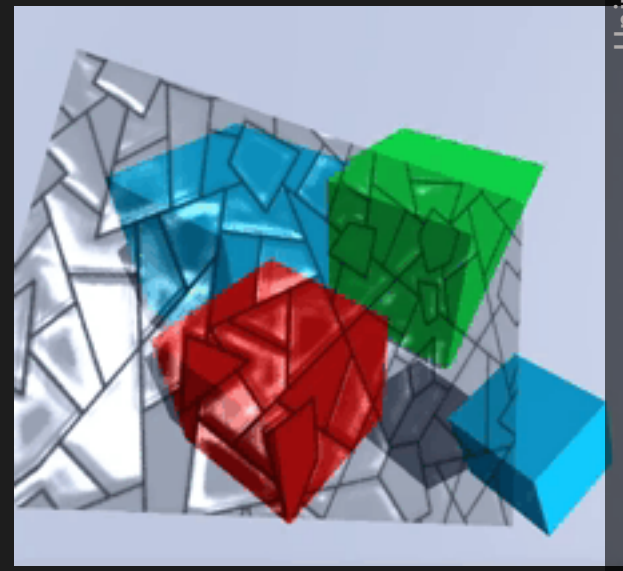


[BIRP] GrabPass: GlassEffect

How can we simulate glass?

- We need to apply a distortion effect to the portion of the screen covered by the glass
- Create glass.shader from lightingAmbient.shader

- We should perform a GrabPass not from the screen corners but from the ScreenUVs according to the 4 corners of the quad glass. There is already a Unity built-in macro for us:
  - Float4 texcoordGrab = ComputeGrabScreenPos(ClipSpaceVertex);
    //Unity macro that returns Screen UVs from ClipSpaceVertex pos
- UV returned from texcoordGrab could be outside [0,1] range. To ensure that they fit into [0,1] range, we should divide texcoordGrab.xy by texcoordGrab.w. This is automatically done if we use tex2Dproj() instead of tex2D():
  - tex2Dproj(_BackgroundTexture, texcoordGrab);
    - ComputeGrabScreenPos() will just transform input from clip coordinate vertex position [-w,w] into [0,w] then calling tex2Dproj() will transform [0,w] into [0,1], which is a valid texture sampling value
    - ComputeGrabScreenPos() doesn't perform automatically a division by w because we need interpolation for the fragment. And a value already divided by w won't interpolate the way you'd expect: you have to perform this division in the fragment shader

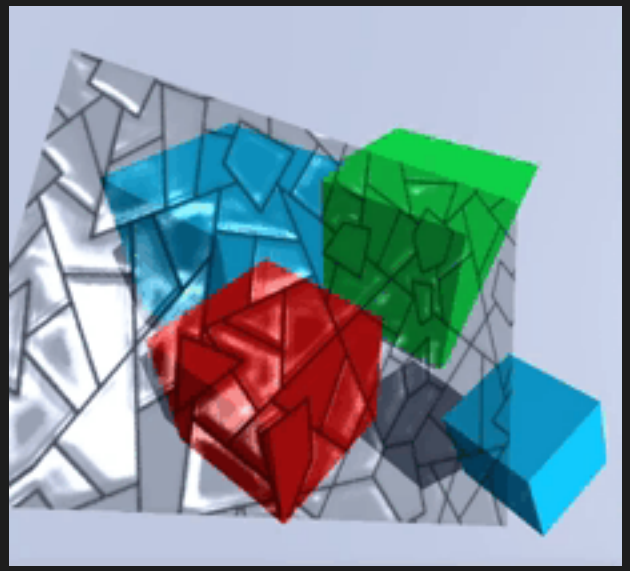


How can we simulate glass distortion?

- We need to apply a distortion effect to the portion of the screen covered by the glass
- We can introduce a color offset of the screen grabbed portion based on the glass normal at that fragment
- NB: We can't use WorldSpaceNormals, because they change according to glass orientation! We need a Normal in a Space that remains the same independently from glass orientation. What about TangentSpaceNormal?

[Shaders_Start_E_03, Glass_Start, glass.shader]

[URP] GrabPass: GlassEffect



How can we simulate glass?

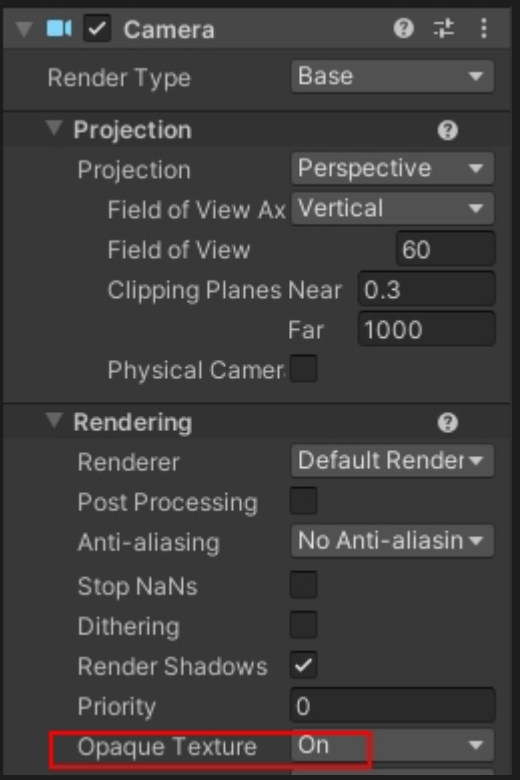
- We need to apply a distortion effect to the portion of the screen covered by the glass
- Create glass.shader from lightingAmbient.shader

- Make sure that our glass panel is Transparent
- We need to set Camera/Rendering/OpaqueTexture ON in order to sample
  uniform sampler2D _CameraOpaqueTexture;
- Then we need to retrieve the screen Texture Space UVs of the 4 corners of the quad glass
- float4 textureScreenPos = ComputeScreenPos(projSpaceVertexPos) //Vertex shader
  - ComputeScreenPos() will not divide input's xy by w because this method expect you sample the texture in fragment shader using tex2Dproj(float4)
  - ComputeScreenPos() will just transform input from clip coordinate vertex position [-w,w] into [0,w]
  - then calling tex2DProj() will transform [0,w] into [0,1], which is a valid texture sampling value

How can we simulate glass distortion?

- We need to apply a distortion effect to the portion of the screen covered by the glass
- We can introduce a color offset of the screen grabbed portion based on the glass normal at that fragment
- NB: We can't use WorldSpaceNormals, because they change according to glass orientation! We need a Normal in a Space that remains the same independently from glass orientation. What about TangentSpaceNormal?

[Shaders_Start_E_03, Glass_Start, lightingGlassH.shader]



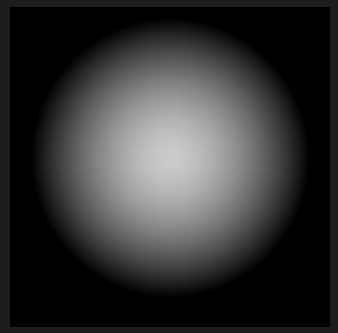
[BIRP] Ex // GrabPass: Heat distortion effect

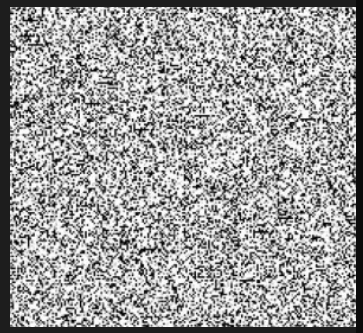
How can we simulate heating distortion? Very similar to previous glass FX

- Create heatDistortion.shader from texture.shader
- Start from a quad: HeatQuad
- Ensure it is in transparent queue (grab pass after all opaque objs rendered): Trasparent queue, Blending
- Add GrabPass on _BackgroundTexture. Sample it using ComputeGrabScreenPos() and test it. Use the MainColor.a as the final alpha
- Add a noise texture to sample for a distortion effect on vertex position in Vshader. Use a Speed to tweak the noise
    - What kind of distortion do you notice?
        - We need more vertices! Change Quad with a Plane
- Add a radial mask to suppress the distortion on the quad edges

Final Effect [bit.ly/EX_HEATINGFX]

[Shaders_Start_E_03, Heat_Start, heatDistortion.shader]





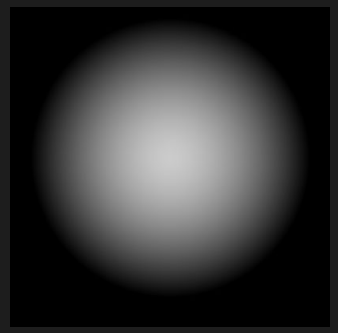
[URP] Ex // GrabPass: Heat distortion effect

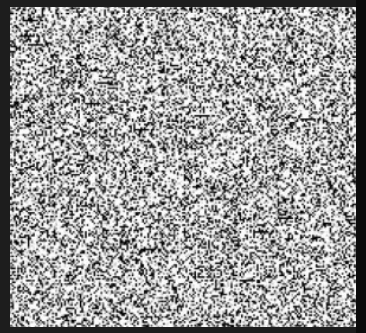
How can we simulate heating distortion? Very similar to previous glass FX

- Create heatDistortion.shader from texture.shader
- Start from a quad: HeatQuad
- Ensure it is in transparent queue
- Add a noise texture to sample for a distortion effect on vertex position in Vshader. Use a Speed to tweak the noise
    - What kind of distortion do you notice? We need more vertices! Change Quad with a Plane
- Add a radial mask to suppress the distortion on the quad edges

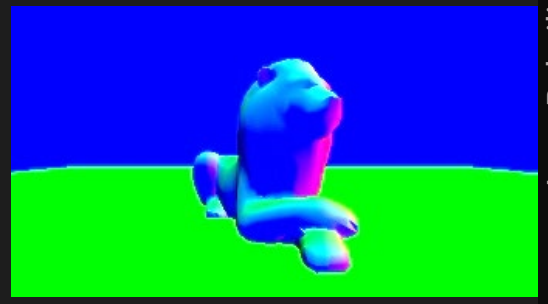
Final Effect [bit.ly/EX_HEATINGFX]

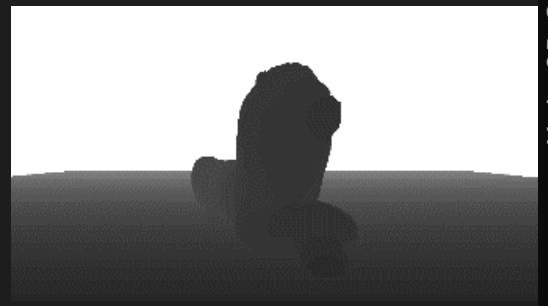
[Shaders_Start_E_03, Heat_Start, heatDistortion.shader]





[BIRP] Cam depth/Normals/MotionVectors





- A Camera can generate a depth, depth+normals, or motion vector Texture
  - Cam.depthTextureMode = DepthTextureMode.[None | Depth | DepthNormals | MotionVectors];
- With this directive, Unity uses a SetReplacementShader() method to render the scene with a Hidden/Camera-DepthTexture shader

- These textures will be available inside your shader by declaring uniform Sampler2D with these names:
  - _CameraDepthTexture

```
float depth = Linear01Depth (tex2D(_CameraDepthTexture, i.texcoord).r);
//Linear01Depth decodes depth in [0,1] range (nearPlane/FarPlane)
```

  - _CameraDepthNormalsTexture

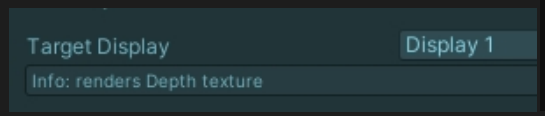
```
float depthValue; float3 normalValues;
DecodeDepthNormal(tex2D(_CameraDepthNormalsTexture, i.texcoord), depthValue, normalValues);
//It takes 8 bit per channel RGBA texture and extracts the 16 bit float depth and view space normal (with a
//        reconstructed Z)
```

  - _CameraMotionVectorsTexture

```
float2 motionXY = tex2D(_CameraMotionVectorsTexture, i.texcoord).rg;
```

- Press play
- DEBUGDEPTH/Canvas/Image has a depthCam material with depthCam.shader: this will display the camera Depth/Normal/MotionVectors textures (the mesh is a quad, and there is no need to use an image texture)
  - depthCam has a vshader input structure appdata_base, witch is a predefined structure in UnityCG.cginc (see img on the right)

```
struct appdata_base
{
    float4 vertex   : POSITION;
    float3 normal   : NORMAL;
    float4 texcoord : TEXCOORD0;
};
```

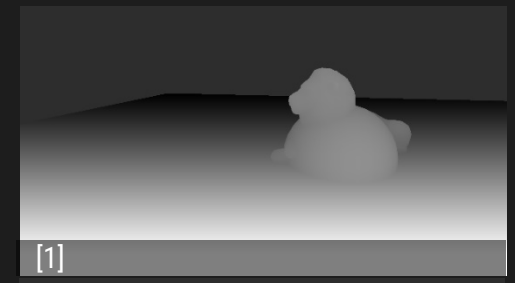


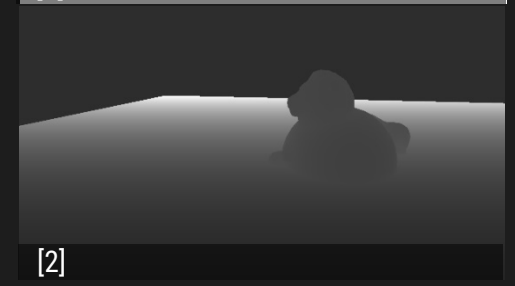
X Translation > Red color
Y Translation > Green color
Z Translation > Red/Green/Yellow colors, depending on the screen-XY direction of the pixels

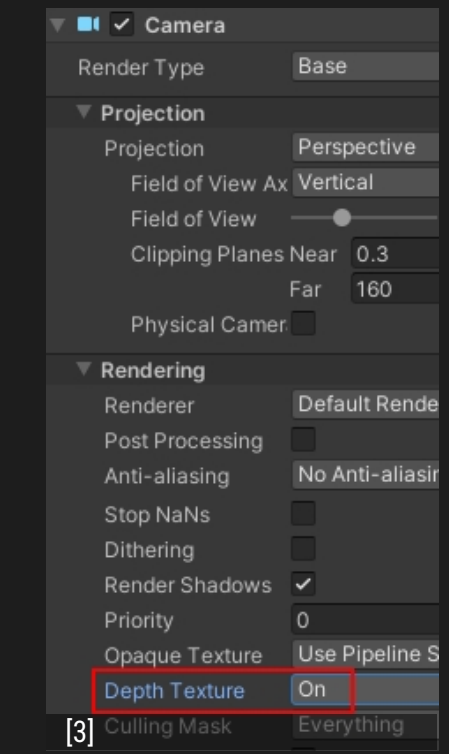
[Shaders_Start_E_03, _DepthCam.shader, CameraDepth_FX_Start, DepthCamSwitcher.cs]

Depth values

- For URP version, enable Camera Depth writing [3]
- Activate DepthViewer and press Play
- Every obj in the scene has _depthViewer shader
- Choose DepthSource parameter to visualize: depth value from the selected source / DebugDiv (use DebugDiv == 1 to see the real value)
  - CamDepthTextureRaw [1] shows _CameraDepthTexture value. Range [1,0] means [near, far]. 0.5 = very close to the camera
    _DebugDiv = 0.1
  - Linear01 [2] shows Linear01Depth() value. Range [0,1] = [near, far]. 0.5 means half way between near and far
    _DebugDiv = 1
  - EyeLinear [2] shows LinearEyeDepth() value. Range [0,far] = [near, far]. 0.5 means 0.5 meters from the camera (depth, not distance)
    _DebugDiv = far
  - ScreenPos_W [2] shows ComputeScreenPos().w interpolated value. Range [0,far] = [near, far]. 0.5 means 0.5 meters from the camera (depth, not distance). When you use a perspective matrix, it is the view depth value, the same value of EyeLinear
    _DebugDiv = far


[1]


[2]



[3]

[Shaders_Start_E_03, CameraDepthFX/DepthViewer, _depthViewer]

Depth values

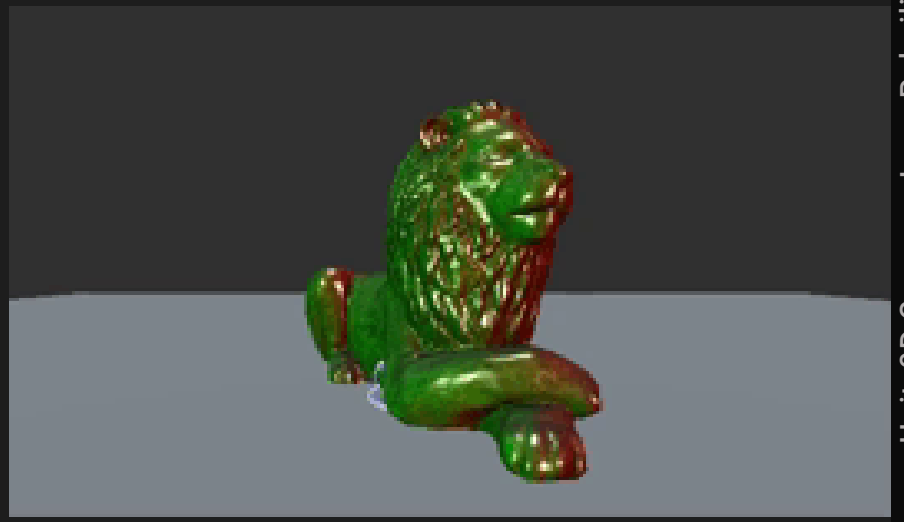
- In order to be rendered in the Depth texture, a shader should [1]
  - Have Tags “RenderType” = “Opaque” “Queue” = “Geometry”
  - Write the Zbuffer Zwrite On
  - If not, that object won’t display in the depth texture. You can use a Fallback that supports these features (ex. Fallback “Diffuse”, Fallback “Lit” [BIRP/URP])

Try it

- From lightingAmbient shader create lightingAmbientDepth shader. Assign it to the Lion
  - Add RenderType, Queue values, Fallback described in [1]
    - Add FallBack
      - now the Lion is rendered with _Color.a alpha
      - Depth frame doesn’t show Lion: Fallback “Diffuse” shader doesn’t help, because our shader is correct: is transparent, and Depth values can’t be recorded
    - Add “RenderType” = “Opaque” “Queue” = “Geometry”
      - Now the Lion is rendered with alpha 1
      - Depth frame shows Lion: our “Opaque” shader doesn’t support shadows, so Fallback “Diffuse” is used instead

[Shaders_Start_E_03, CameraDepthFX/DepthViewer, _depthViewer.shader]

Play with camera depth



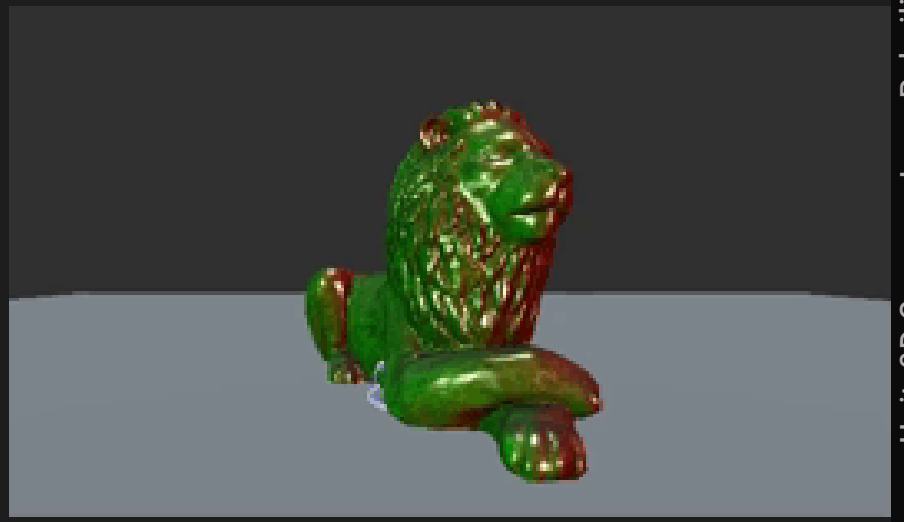
Intersection fragment FX

Try it

- Create intersectionHighlight.shader from textureStart.shader
- We need
  - _RegularColor, _HighlightColor colors and
  - _HighlightThresholdMax, a Range(0,1) witch is the max depth difference for intersections
- VS
  - ComputeScreenPos() Computes texture coordinate for doing a screenspace-mapped texture sample. Input is clip space position
  - It is similar to ComputeGrabScreenPos(), but ComputeScreenPos doesn't take into account OPENGL/DIRECTX differences (because we don't need it for the screen)
    - OPENGL texture coordinate 0 is at the bottom, and grows upward
    - DIRECTX texture coordinate 0 is at the top, and grows downward
- Render SceneDepthTexture from camera
- Compute the ScreenSpaceFragment position range from ObjSpaceVertex
  - Use it to sample SceneDepthTexture sceneZ
  - Depth is not stored in linear values: more precision is for nearest objects: use LinearEyeDepth() to process depth info. The new range wil be [0,farPlane]: a value of n is a surface that is n unit from the camera's pivot along the camera's z axis

[CameraDepth_FX_Start, _intersectionHighlight2.shader, _intersectionHighlightH.shader]

Play with camera depth



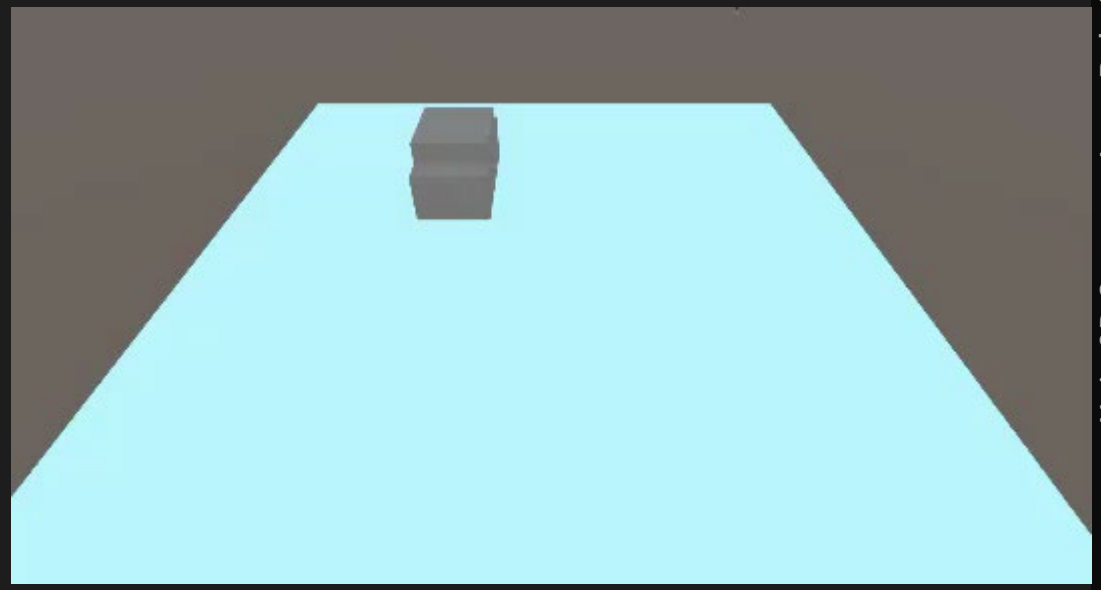
Try it

- Compute Fragment distance from the camera in worldSpace dFromCam
  - We have _WorldSpaceCameraPos for camera, we need fragment in WorldSpace o.wpos
- At this point you can debug outputting them ad dividing their value by a _DebugDiv float val Range(0,100)
- If abs(sceneZ – dFromCam) < threshold, the obj is intersecting something
- NB:
  - This FX is working because the sphere is NOT rendered in the camera depth buffer

[Shaders_Start_E_03, _intersectionHighlight2.shader, _intersectionHighlightH.shader]

SnowShader

- Car emits 2 particleSystems
- Particle systems are additive and in Snow Layer
- Snow_Plane_Black is a black plane in Snow Layer
- Snow_RTCamera is an orthogonal Camera which renders into RT Snow. The rendertexture is a square texture (hence the RTCamera should have Viewport 0,1)
- Particles leave an additive trail on Snow_Plane_Black when the car moves
- Snow_RTCamera can see only Snow Layer
- MainCamera can't see Snow Layer
- Create _snowH shader and add it to SnowPlane_Real
    - Read from RT Snow the height offset in object space.y of the vertices



[snowShader_Start, snowShader_End,_snowH.shader]

Toon water shader

Try it

- Create waterIntersection.shader from intersectionHighlight.shader, assign it to WaterPlane
- If you use simple difference between sceneZ and dFromCam [0], where (fragment shader)

  sceneZ = LinearEyeDepth (tex2Dproj(_CameraDepthTexture, currprojPos)).r

  dFromCam = i.projPos.w;

  and (vertex shader)

  o.projPos = ComputeScreenPos(o.pos);

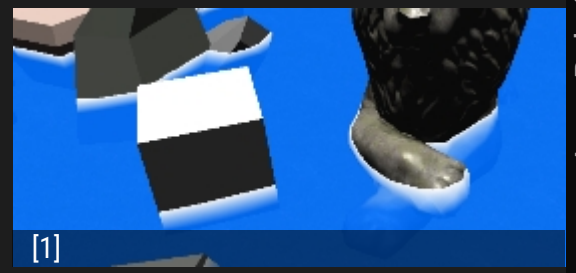
  you'll get [1] result
- If you use a threshold value, you'll get [2] result
- If you use a ramp texture, where uv coords are

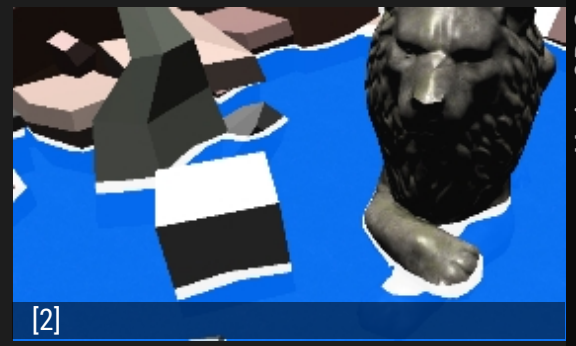
  u – difference [0]

  v – 0.5

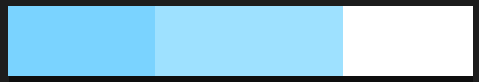
  you'll get [3] result
- Properties
  - A FoamMode enum to get [1],[2],or [3] final color
  - A FoamRamp texture to get the toon effect
  - Speed, amplitude, noise texture to move water
- vIn structure different from app_data
  - Noise texture texcoords: pay attention to TEXCOORD interpolator numbers: if noise texture is the second texture in the parameter list, you have to use TEXCOORD1 for it

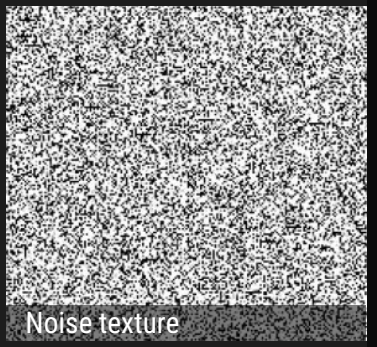
[Shaders_Start_E_03, _waterIntersection.shader]


[1]


[2]


[3]


Ramp texture


Noise texture

Toon water shader

- This foam approach based on depth has an Edge problem [4]. This is a good compromise between shader complexity and final result. To get a better result, you can use a multisampling of _CameraDepthTexture: use an offset to sample the depth texture on the right/left of the current fragment, and save the min value between sampling [5]
- A better approach to have edge foam is with procedural meshes.

[4]

[5]

[Shaders_Start_E_03, _waterIntersection.shader]

Screen shaders

- We can write scripts that alter camera output and write Image Fx
- The shader core struct is simple
  - #include “UnityCG.cginc”
  - _MainTex Texture 2D property which will be filled with the current camera frame
  - Standard built-in VShader vert_img
    - minimal vertex shader that is defined in UnityCG.cginc
  - Standard built-in output VShader structure v2f_img

```
v2f_img vert_img( appdata_img v )
{
    v2f_img o;
    o.pos = mul (UNITY_MATRIX_MVP, v.vertex);
    o.uv = v.texcoord;
    return o;
}
```

```
struct v2f_img
{
    float4 pos : SV_POSITION;
    half2 uv   : TEXCOORD0;
};
```

- Attach a script on the rendering camera with:

```
void OnRenderImage(RenderTexture source, RenderTexture destination)
{
    if (!ApplyScreenFx)
        //Write the current frame to the frameBuffer
        Graphics.Blit(source, destination);
    else
        //Process curr frame with material shader and then pass the result img to the frameBuffer
        Graphics.Blit(source, destination, material);
}
```

- OnRenderImage is called after all rendering is complete to render image. It allows you to modify final image by processing it with shader based filters

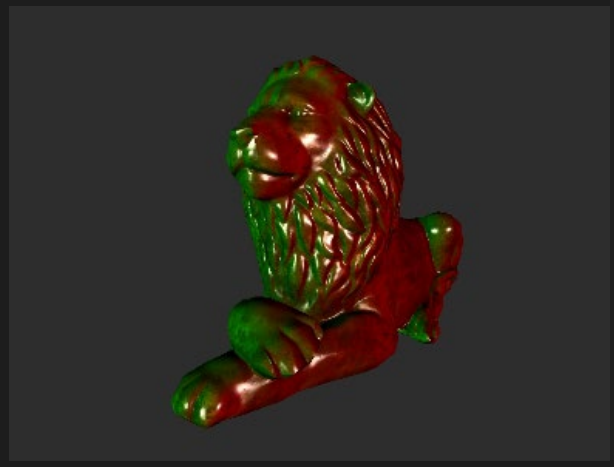
[Shaders_Start_E_03/ScreenShader_Start]

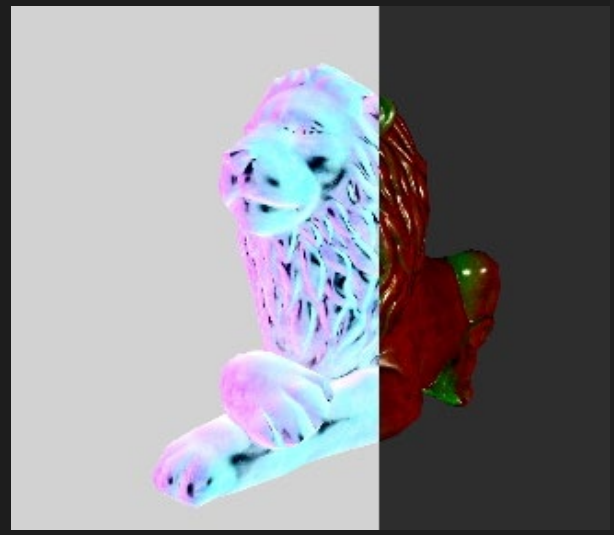
Screen shaders

Try it

- Try to write a shader that invert color on the left _Treshold portion of the screen (0.5 = half screen)
- Start from texture.shader and create a screen shader IMG_HalfInv.shader, using standard built-in VShader vert_img and standard built-in output VShader structure v2f_img

[Shaders_Start_E_03/ScreenShader_Start, _IMG_HalfInv.shader, ApplyScreenShaderFx.cs]

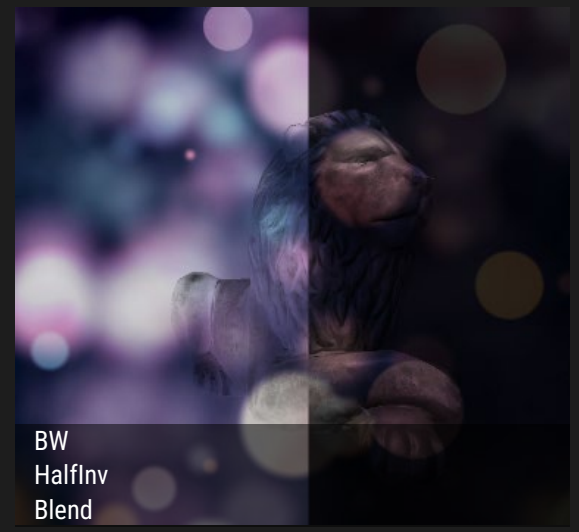


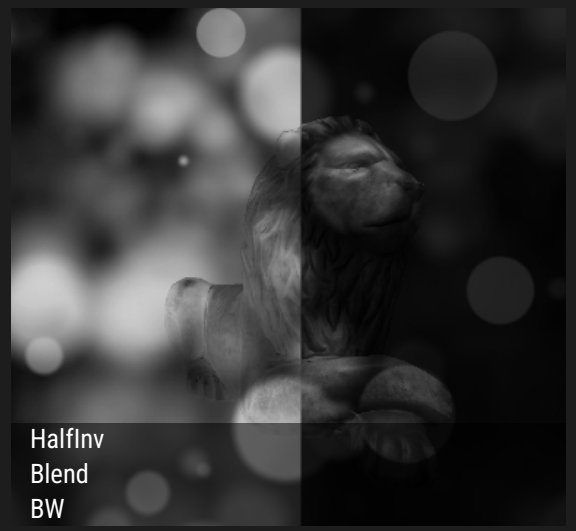


Multiple Image filters

OnRenderImage(RenderTexture source, RenderTexture destination)

- The incoming image is source render texture. The result should end up in destination render texture. You must always issue a Graphics.Blit or render a fullscreen quad if you override this method
- When there are multiple image filters attached to the camera, they process image sequentially, by passing first filter's destination as the source to the next filter
- This message is sent to all scripts attached to the camera





Particle system vertex streams

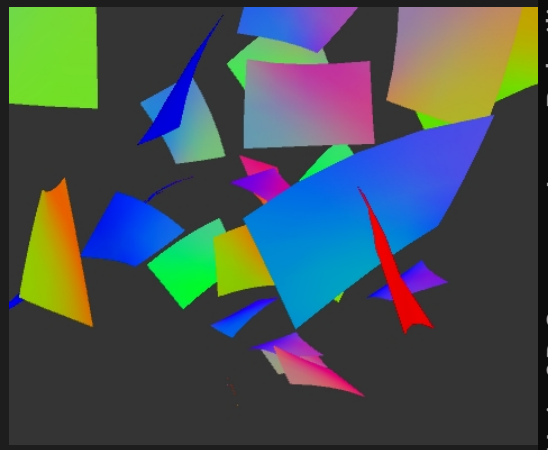
Let's see how to apply Shaders to Particle Systems

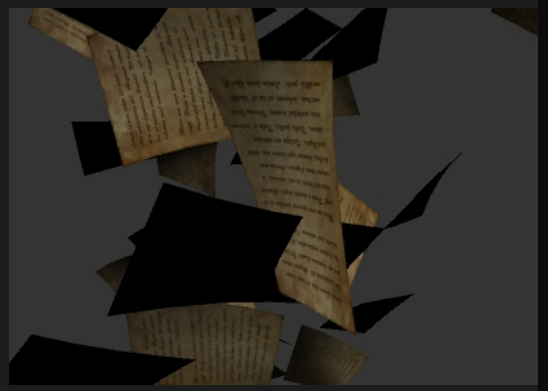
- Open [ShaderParticles_00] and start from BurningPage_Start
- Create lightingParticles shader and material from lightingAmbient
- ParticleSystem/RendererModule/CustomVertexStream allows to control what data to send to the VShader. We can use it to send
  - TANGENT for NormalMapping
  - CustomData values
    - Once we decide what data to send, Unity suggest us how to read that data via data mapping (e.g. UV => TEXCOORD0.xy)
- ParticleSystem/CustomDataModule allow to specify a value that could change over particle lifetime

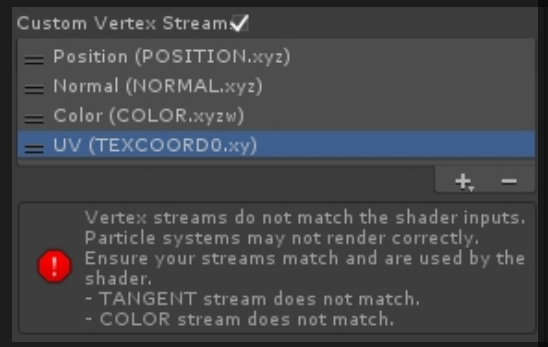
Try it

- Add Position, Normal, Color, UV, Tangent to CustomVertexStreams
- Add lightingParticles as Render module material. You should see particles colored using their normal
- Add page texture as albedo for the material, and set LightingMode shader variant to Frag
- Add glass texture as normal map for the material, and set UseNormalMaps shader variant to On
- We are sending COLOR from the Psystem, but we don't use this info in our shader (this will cause an error)
- Fix these 2 issues by adding COLOR as Vshader input (half4 color : COLOR) and output (half4 vcolor : COLOR1)
  - The Vshader should pass the Particle color: o.vcolor = v.color.
  - The Fragment finalColor should be multiplied by i.vColor
- Check that ColorOverLifeTime Psystem module works: activate it and see if colors are passed to the shader. Then, deactivate this module

[ShaderParticles_Start_01, _lightingParticles2.shader]







Particle system vertex streams

Try it (continue)

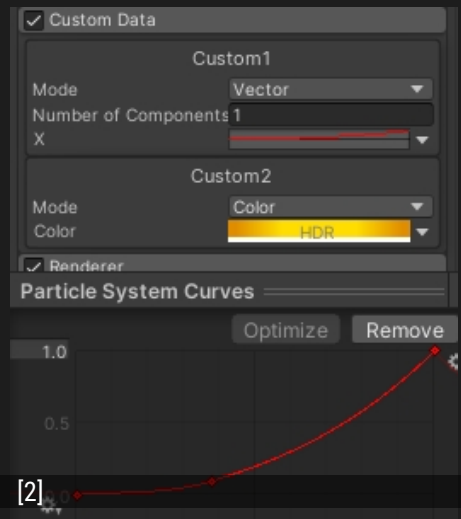
- Page texture has an alpha channel. We can use it to fade (with a burn effect) the page [1]
  - The final alpha should be: [pageTexture.alpha] – [a grow-in-time value] (from 0 to 1)
  - We'll use a CustomData float growing value (Vector with 1 component) as the [grow-in-time value] [2]
  - We'll use also a CustomData Color for the burn color [2]
- Start by simply using pageTexture.alpha value as finalAlpha. You'll obtain something too transparent [3]
- We need a threshold value for the Alpha. Add _BurnTolerance float value, and multiply finalAlpha by this value
  - finalAlpha = saturate(finalAlpha * _BurnTolerance) //0 if x<0, 1 if x>1, x otherwise
- Add CustomData inputs to the shader
  - Vector1 with an exponential curve / Add Custom1.x to Render.CustomVertexStreams / it will be passed in TEXCOORD1.x to the Vshader input
  - Color / Add Custom2.xyz / it will be passed in TEXCOORD1.yzw to the Vshader input [*]
- Use TEXCOORD1 as CustomData semantic for the Vshader, TEXCOORD6 as CustomData semantic for the Fragment
  - TEXCOORD0,1,2,3,4,5 are already used by tangentWorld,binormalWorld, etc.
- In Vshader, o.customData = v.customData;

- Even if final alpha is 0, we are writing framebuffer RGB values and ZBuffer
  - clip() discards a pixel if any of its component values is less than 0
  - We can also use if(finalAlpha < 0) discard;

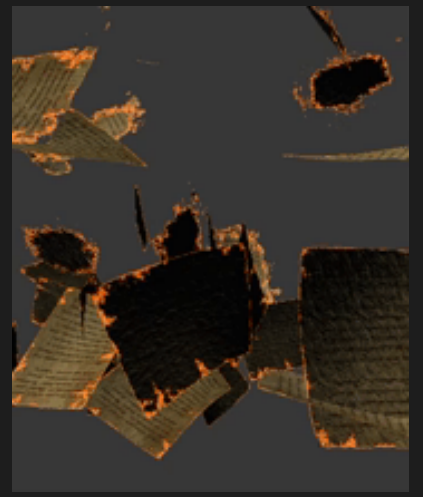
[*]Custom2 float3 data is mapped into TEXTCOORD0.w|xy, The (w|xy) indicates that the xy belong to the next texcoord, which would be TEXCOORD1, but since we already use TEXCOORD1 for other purposes, we need to add 1 to the others TEXCOORD interpolator

[Particles_00.scene, BurningPage_Start, lightingParticles2.shader]

[1]



[2]

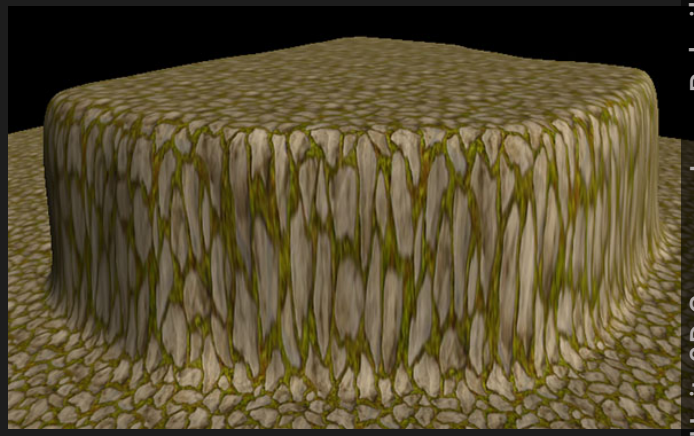


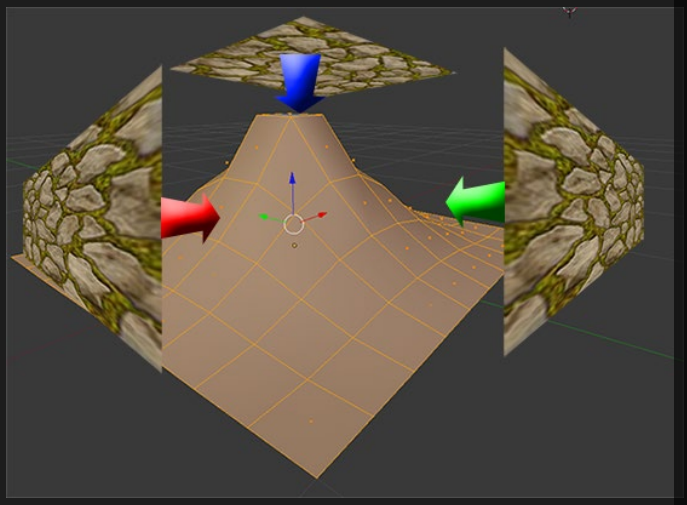

[3]

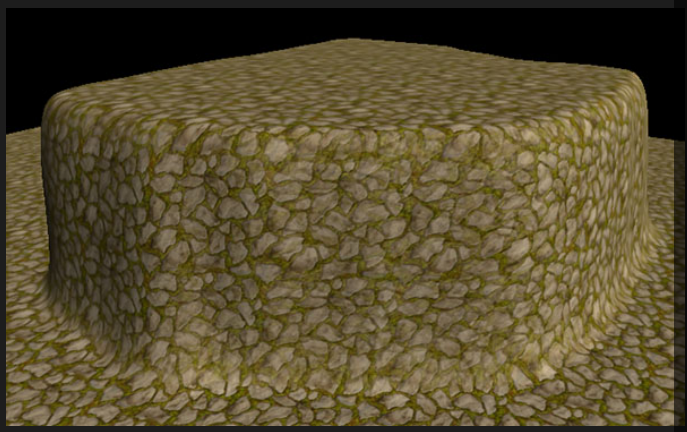
Triplanar texturing

- We'll have 3 projection-textures, one for each plane: xz, zy, xy
  - Add 3 2DTextures properties: _TexturePlaneX,y,Z
- Find 2D point: WorldSpace_Vertex_Projection on each plane: xz, zy, xy
- Scale that projection by a ScaleFactorX, ScaleFactorY, ScaleFactorZ
  - Add 3 Float properties: ScaleFactorX,Y,Z
- In the PShader
  - Use that projection as Texture coordinate to sample the 3 projection-textures
  - Use the normal x,y,z components to blend the 3 textures
    - blendWeight = pow (abs(i.normalWorld), _TriplanarBlendSharpness);
    - Change blendWeight so that (blendWeight.x + blendWeight.y + blendWeight.z) = 1
    - Use blendWeight.x,y,z to blend the 3 textures

[TextureTriPlanar_Ex_Start, _textureTriplanar.shader]



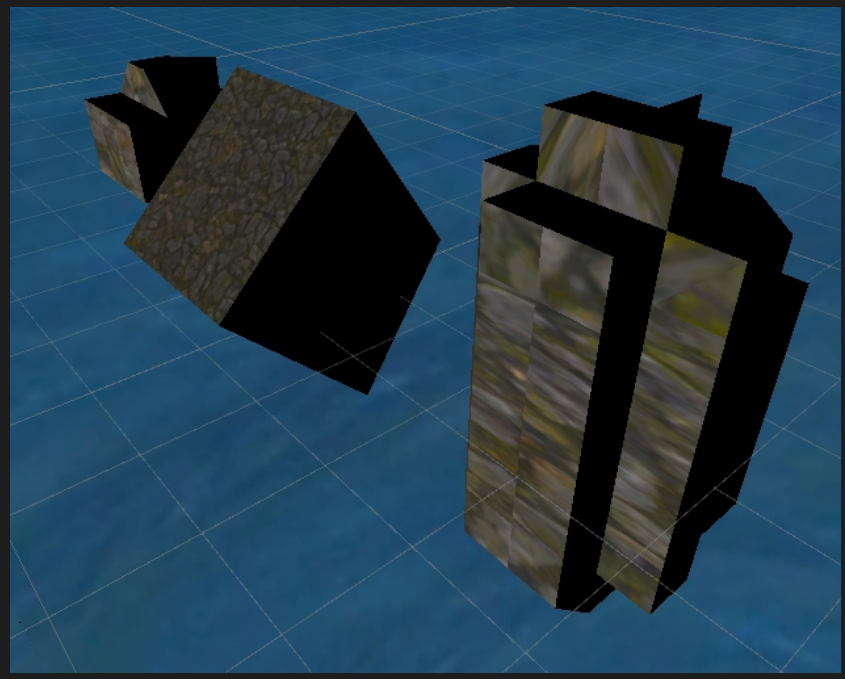




Geometry shaders: GridSnap

- Geometry shader

[_textureGridSnap.shader]



Add more lights

Remember the ForwardBase Tag?
Forward rendering path has two different code written passes that we can use in our shader: base pass and the additional pass. In the base pass, we can define the ForwardBase light mode and in the additional pass, we can define the ForwardAdd light mode. Both are characteristic functions of a shader with lighting calculation. The base pass can process

- directional light per-pixel and will use the brightest light if there are multiple directional lights in the scene
- light probes
- global illumination
- ambient illumination (skylight)

As its name says, the additional pass can process “additional lights” per-pixel or also shadows that affect the object, what does this mean? If we have two lights in the scene, our object will be influenced only by one of them, however, if we have defined an additional pass for this configuration, then it will be influenced by both. One point that we must take into consideration is that each illuminated pass will generate a separate draw call.

- Only for the brightest Directional light. If no DirLightExist, this will render only AmbientLight
- Let’s add another Pass, this time with ForwardAdd Tag, to handle other lights. This Pass will share a lot of code with the ForwardBase one. Create a CGLihtShader.cginc file to avoid repetitions
    - In the cginc file you can include from “uniform” variables declaration to fragment function
    - In your lighting shader you should leave:
        - 1 Pass with ForwardBase
            - Blending AlphaBlending
            - CGPROGRAM snippet
        - 1 Pass with ForwardAdd
            - Blending Multiply
            - Zwrite Off
            - CGPROGRAM snippet
        - Inside each CGPROGRAM snippet
            - All #pragma directives
            - #include CGLightShader.cginc

[MoreLights_Start, MoreLights.shader]

```
Subshader
{
    Tags{"Queue" = "Transparent" "RenderType" = "Transparent" "IgnoreProjector" = "true"}
    Pass
    {
        Tags{"LightMode" = "ForwardBase"}

        Blend SrcAlpha OneMinusSrcAlpha
        BlendOp Add
        CGPROGRAM

            #pragma vertex vert
            #pragma fragment frag
            #pragma shader_feature _LIGHTING_OFF _LIGHTING_VERT _LIGHTING_FRAG
            #pragma shader_feature _USENORMAL_ON _USENORMAL_OFF
            #pragma shader_feature _AMBIENTMODE_OFF _AMBIENTMODE_ON


            #include "CGLightShader.cginc"
        ENDCG

    }
    Pass
    {
        Tags{"LightMode" = "ForwardAdd"}

        Blend One One
        BlendOp Add
        ZWrite Off

        CGPROGRAM
            #pragma vertex vert
            #pragma fragment frag
            #pragma shader_feature _LIGHTING_OFF _LIGHTING_VERT _LIGHTING_FRAG
            #pragma shader_feature _USENORMAL_ON _USENORMAL_OFF
            #pragma shader_feature _AMBIENTMODE_OFF _AMBIENTMODE_ON


            #include "CGLightShader.cginc"
        ENDCG

    }
}
```

Add more lights

- Now our shader support more than one Directional Light
- Try it with 2 Directional lights, different colors

[MoreLights_Start, _MoreLights.shader]

One Draw call for each directional light

Add more lights

- Now our shader support more than one Directional Light
- Try it with 2 Directional lights, different colors

What about Point lights?

- Point Lights VS Directional Lights
  - Dir is different for each vertex
    lightDir = normalize(_WorldSpaceLightPos0.xyz – vertex_WorldSpacePos);
  - We need to calculate a lightVector which magnitude will attenuate the light intensity
    lightVec = _WorldSpaceLightPos0.xyz - vertex_WorldSpacePos;
  - We need to calculate the attenuation, which should be something like $1/(1+d^2)$, where d = | lightVec |
    - Since dot(a,b) = |a||b| cos(alpha), if a = b we got |a||a|, then
      float attenuation = 1 / (1+dot(lightVec, lightVec));
- If we are processing a Directional light, DIRECTIONAL constant will be defined by Unity. In case of a Point light, POINT will be used. We add this variants using:
  - #pragma multi_compile DIRECTIONAL POINT
- Finally, we have to take into account Light Range value. The easiest way to do this in PShader is to use Built-in UNITY_LIGHT_ATTENUATION macro:
  #include "AutoLight.cginc"
  UNITY_LIGHT_ATTENUATION(attenuation, 0, vertex_WorldSpacePos);

[MoreLights_Start, _MoreLights.shader]

One Draw call for each light

Point Light Range BBox

ShaderGraph

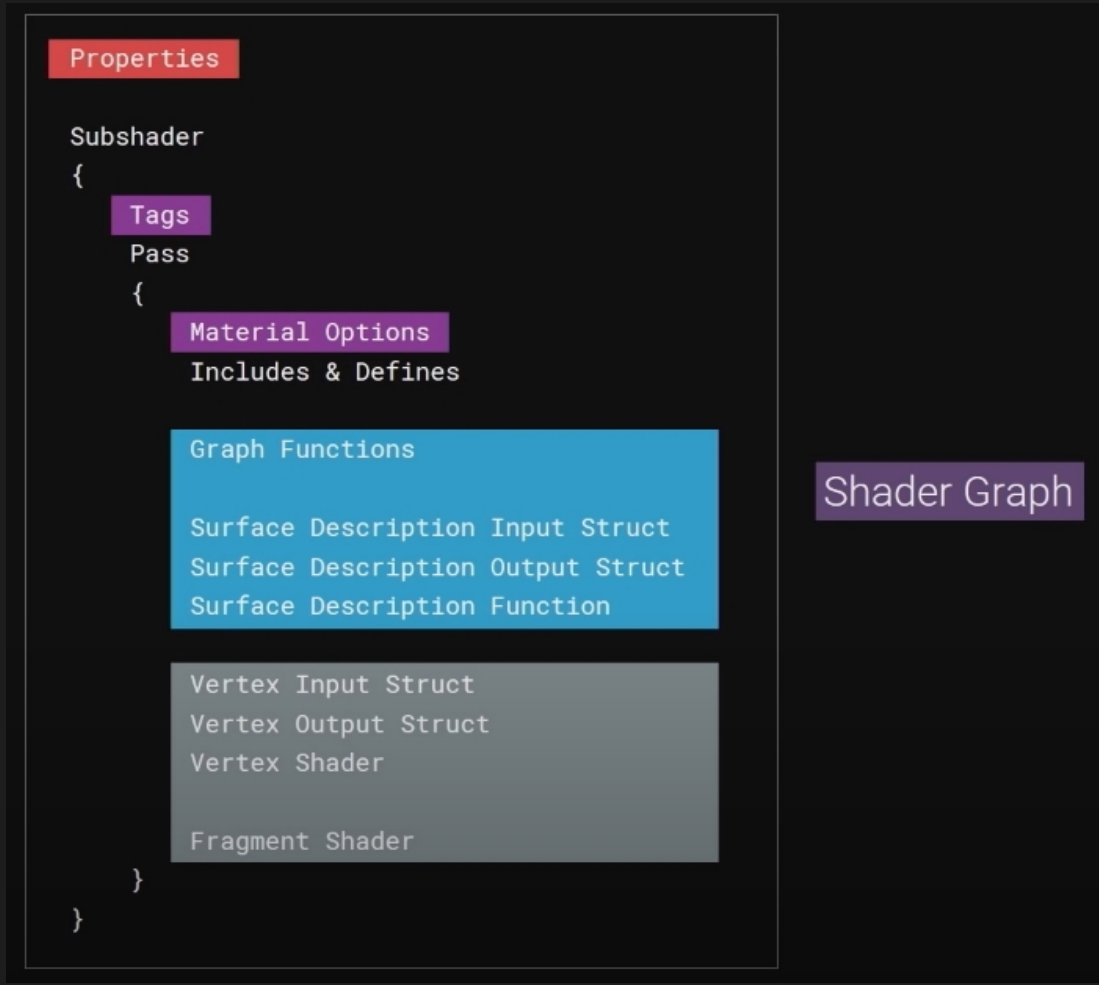
- We need to specify a SRPipeline in order to use ShaderGraph, because each RPipeline has a shader backend, which includes subshader generators for supported Master Nodes
- Each Master Node has a per-pipeline template

Properties
- are exposed in the material inspector, and can be modified per material instance
- They are specified in ShaderGraph Blackboard, and most input nodes can be converted to properties
    - Just click “+” in the blackboard, and drag the exposed property you created into the black canvas

Tag & Material Options
- Are defined in the settings and various properties of the masternode itself (gear icon on masternode UI)

Function used in your shader graph
- If you use multiply or noise function, that code will be included in your shader
- We need to provide the input and output data structures for the surface shaders unity uses is to tell the GPU exactly how data will be transferred in and out of your shader functions
- the surfacefunction itself ties all of your shadergraph operations together

[https://youtu.be/Y6WfgFI5H90?t=1510]



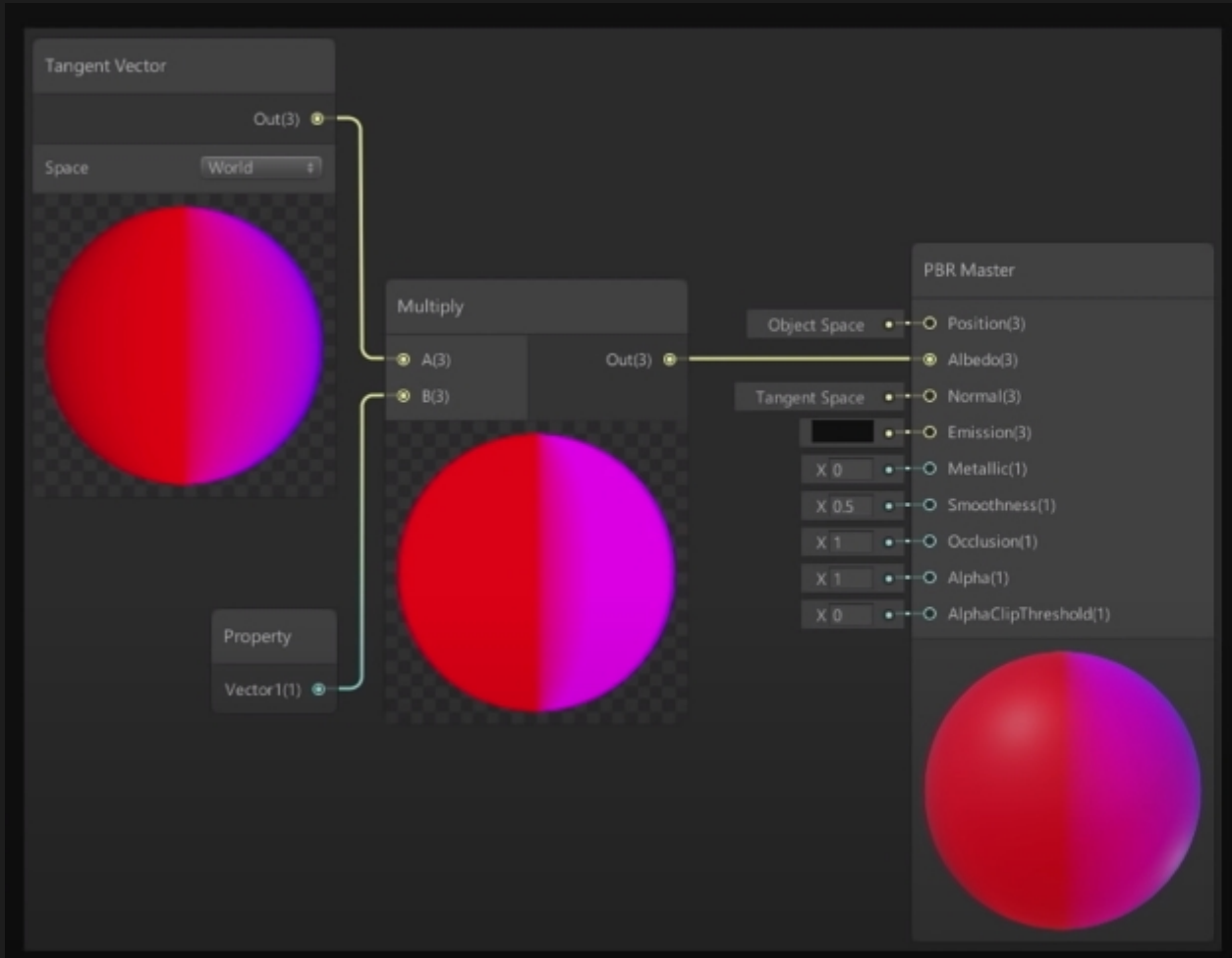
Update Materials to URP

- Some materials can’t be converted into URP materials, simply because there is not a provided shader converter from the start shader to a compatible URP shader
- You can try to hack this conversion when you got an error like this:
  - “xxx material was not upgraded. There's no upgrader to convert xxx shader to selected pipeline UnityEditor.Rendering.Universal.UniversalRenderPipelineMaterialUpgrader:UpgradeSelectedMaterials() (at Library/PackageCache/com.unity.render-pipelines.universal@x.y.z/Editor/UniversalRenderPipelineMaterialUpgrader.cs)”
  - If the error was a Custom/ToonShader material, try to add something like upgraders.Add(new StandardUpgrader("Custom/ToonShader"));
    In UniversalRenderPipelineMaterialUpgrader.cs GetUpgraders() method

ShaderGraph

Function used in your shader graph

- see what kind of code Unity generates from this shader graph




```
void Unity_Multiply_float (float3 A, float3 B, out float3 Out)
{
    Out = A * B;
}

struct SurfaceDescriptionInputs
{
    float3 WorldSpaceTangent;
}


struct SurfaceDescription
{
    float3 Albedo;
}


SurfaceDescription PopulateSurfaceData (SurfaceDescriptionInputs IN)
{
    SurfaceDescription surface = (SurfaceDescription)0;
    float _Vector1_Out = 2;
    float3 _Multiply_Out;

    Unity_Multiply_float(IN.WorldSpaceTangent,
        _Vector1_Out.xxx,
        _Multiply_Out)

    surface.Albedo = _Multiply_Out;
    return surface;
}
```

ShaderGraph

Function used in your shader graph

- We need to include the multiply function
- Define input and output structures for surface shaders (we need WorldTangent as input)
- The surface shader function
  - this is where we call the graph functions in the way that you set up in your shader graph using data supplied from the description input data structure
  - at the end of the function we simply return the calculated value in the output data structure
  - in this particular example the code is doing exactly what the graph is defining which is multiplying the tangent vector with the scalar then setting that as a surface color

Vertex/Fragment shader

- We need to inject code into the vertex input and output structures as well as the vertex and fragment shaders themselves
- this is done because in order to perform operations using the mesh data we need to pass that data along from the vertex to the fragment pipeline
- this includes space transformations and other various properties depending on what the shader graph actually needs

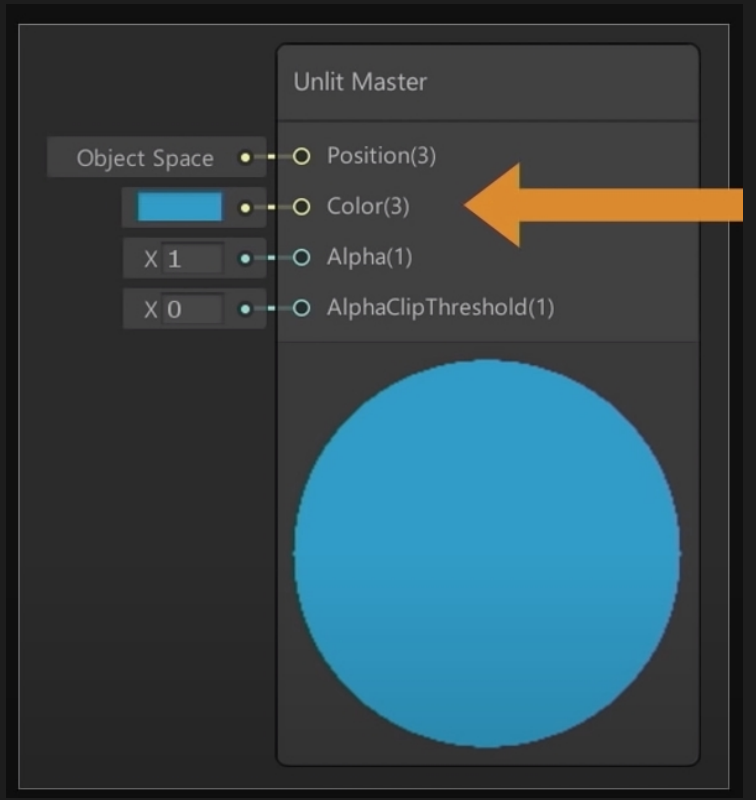
```
void Unity_Multiply_float (float3 A, float3 B, out float3 Out)
{
    Out = A * B;
}

struct SurfaceDescriptionInputs
{
    float3 WorldSpaceTangent;
}

struct SurfaceDescription
{
    float3 Albedo;
}

SurfaceDescription PopulateSurfaceData (SurfaceDescriptionInputs IN)
{
    SurfaceDescription surface = (SurfaceDescription)0;
    float _Vector1_Out = 2;
    float3 _Multiply_Out;

    Unity_Multiply_float(IN.WorldSpaceTangent,
        _Vector1_Out.xxx,
        _Multiply_Out)

    surface.Albedo = _Multiply_Out;
    return surface;
}
```

```
        Vertex Input Struct
        Vertex Output Struct
        Vertex Shader

        Fragment Shader
    }
}
```

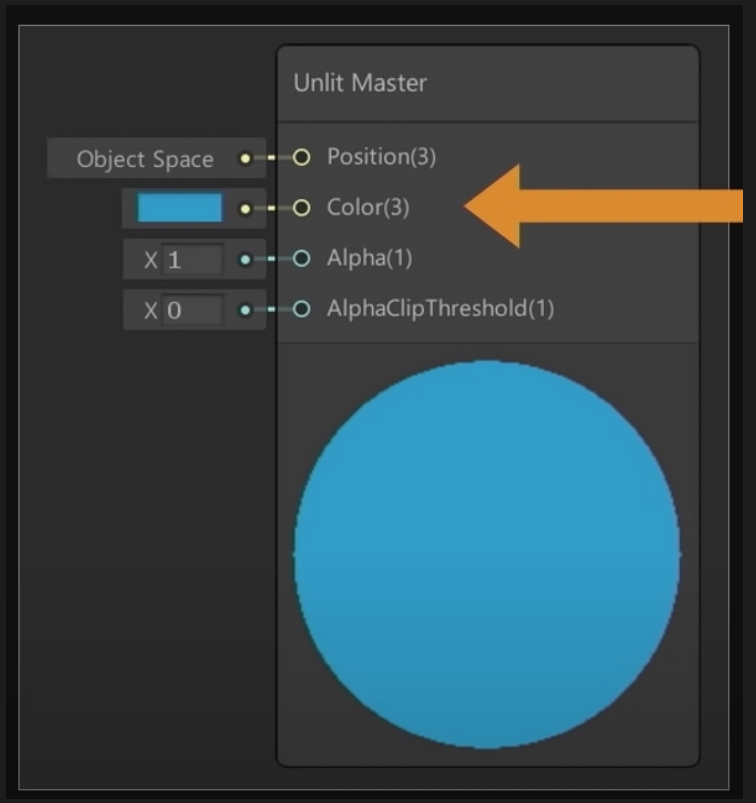
ShaderGraph

Choosing a master node

- it's actually an important decision that has profound implications on overall performance
- PBR Master node works with all kind of SRP: good candidate for prototyping
- HDLit Master
  - Can’t use it for everything: too expensive
- Unlit
  - works with all kind of SRP
  - Good for Fx and UI
  - Can perform math operations in nodes for lighting (e.g. Blinn-Phong) and then apply the result to the Color input of the Unlit Master node
  - Lightweight way to do some simple lighting



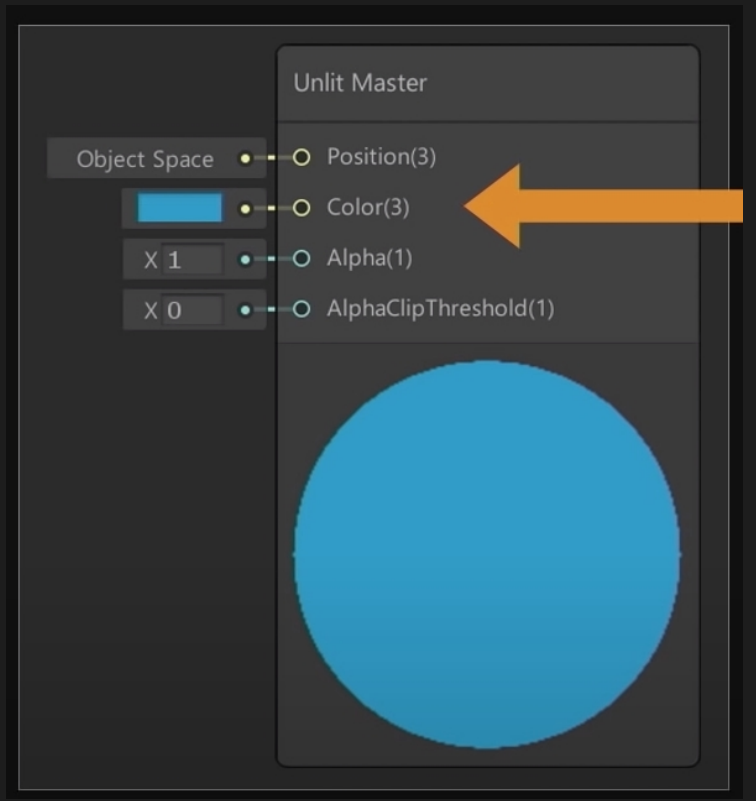
SG Performance



The master node that you select and how you use it affects how unity will try to optimize your shader performance

- any feature in your master node that you don't use won't be added to your output shader
  - there are a few exceptions to this like the albedo color
  - if I were to change the default value for a particular feature, that feature will get marked as active and code for that will get added to your output shader
- what the shader compiler will do for you
  - It takes your high-level shader lab code then compile that down to low-level by code that could be sent over to the GPU
  - This doesn't apply just to share the graph output, but to all shaders that go through unity's shader compiler and as part of the byte code generation process the shader compiler will also attempt to optimize your shader code
    - by removing no-opcode (mul by 1)
    - If you leave emission node to black and unchecked in the inspector, emission code will be compiled out (removed)
    - If you convert a node to a property, this eliminates the possibility for the shader compiler to optimize your shader function

SG Performance



- GPU instancing is supported in shaders (https://answers.unity.com/questions/1427045/whats-the-difference-between-using-perrendererdata.html)
  - Needs to be enabled on the material
  - Requires hardware support
- Dynamic batching is disabled in the SRP
  - Even if it did work, limits you to no more than 900 vertex attributes per mesh, which isn't very suitable for most modern 3D Games today anyway
- There is a new kind of batching technique: SRP batcher, automatically supported by every shader created with SGraph
  - Replaces dynamic batching in the SRP
  - Works both for Mesh renderers and for Skinned Mesh renderers

- If you just need a Vector2, don't use a Vector4
- If you want to hand-optimize a shader, this would likely interfere with the injection based templated approach that shader graph uses
  - Rclick on MasterNode/CopyShader/Paste code in a new shader file
- Nodes not connected to MasterNode aren't avaluated
  - Can keep them in view for quick iterations
  - Create a PreviewNode that isn't attached to the MasterNode
    - Editing these nodes will be mush faster

SG Performance

SGraph optimization Workflow summary

- Prototype with PBR MasterNode
- Start optimizations
  - Optimize nodes
  - Convert properties to inline values
  - Near the end of the project, start hand optimizing shaders

SG Setup

- PackageManager / ShaderGraph
- Install URP: PackageManager / Universal RP
- Create a PipelineAsset
  - This will create 2 assets in your projects: URPAsset and URPAsset_Renderer
- Drag ProjectSettings/Graphics/SRPipelineSettings