**Unity Technologies Korea**
**Technical Artist/Senior Advocate, SangYun Yi**

This document based on unity 2021.1, Universal RenderPipeline.

# URP Shader Basic term

- ## Shader란?

  셰이더, 쉐이더… 화면에 어떻게 그릴지 정의 된 코드(프로그래밍된 코드)
  ‘색의 농담, 색조, 명암효과 등을 주다 ' 라는 뜻을 가진 Shade에서 접미사 –er을 덧붙여서 Shader라고 부른다.

  결국 화면에 배열된 픽셀들이 어떤 색으로 그려질지를 정의된 코드로 각 픽셀별로 RGBA 값을 어떻게 지정할건지를 계산하는 과정
- Unity에서 셰이더 작성에 사용되는 건 셰이더 랩이라고 이야기 한다. URP에서는 CG와 HLSL 을 모두 지원하지만 HLSL을 기본으로 설명한다.

## Basic simple basic code

```
 // Shader 시작. 셰이더의 폴더와 이름을 여기서 결정합니다.
Shader "URPTraining/URPBasic"
{


  Properties
        {
// Properties Block : 셰이더에서 사용할 변수를 선언하고 이를 material inspector에 노출시킵니다
        }

        SubShader
        {

        Tags
       {
//Render type과 Render Queue를 여기서 결정합니다.
          "RenderPipeline"="UniversalPipeline"
          "RenderType"="Opaque"
          "Queue"="Geometry"
       }
        Pass
        {
         Name "Universal Forward"
         Tags { "LightMode" = "UniversalForward" }

        HLSLPROGRAM

        #pragma prefer_hlslcc gles
        #pragma exclude_renderers d3d11_9x
        #pragma vertex vert
        #pragma fragment frag

//cg shader는 .cginc를 hlsl shader는 .hlsl을 include하게 됩니다.
        #include "Packages/com.unity.render-pipelines.universal/ShaderLibrary/Lighting.hlsl"

//vertex buffer에서 읽어올 정보를 선언합니다.
        struct VertexInput
        {
        float4 vertex : POSITION;
        };

//보간기를 통해 버텍스 셰이더에서 픽셀 셰이더로 전달할 정보를 선언합니다.
        struct VertexOutput
        {
        float4 vertex   : SV_POSITION;
        };

//버텍스 셰이더
        VertexOutput vert(VertexInput v)
        {

        VertexOutput o;
        o.vertex = TransformObjectToHClip(v.vertex.xyz);
```

```
        return o;
        }

//픽셀 셰이더
        half4 frag(VertexOutput i) : SV_Target
        {

        return half4(0.5 , 0.5, 0.5, 1);

        }

        ENDHLSL
        }
    }
}
```

Shader “shader folder name / shader name”


- Shader “폴더이름 / Shader 이름“ 으로 인식하며, 같은 폴더/이름이 존재할 경우 둘중 하나는 안나옴

Properties에서 변수 설정하고 매터리얼에서 확인

- Material 창에 표시되는 항목. 변수[1]를 정의하고 타입을 정하게 되며 여기에서 선언된 변수명이 셰이더 계산에 사용된다. 아래와 같은 세가지 규칙이 적용된다.
    - 숫자로 시작하면 안됨
    - 예약어 안됨
    - 대소문자 구별함

| 셰이더 Properties에서는 아래와 같이 선언되며 | Cg/HLSL 코드로의 엑세스를 위해 다음과 같이 선언된다. |
|---|---|
| _MyColor ("Some Color", Color) = (1,1,1,1)<br>_MyVector ("Some Vector", Vector) = (0,0,0,0)<br>_MyFloat ("My float", Float) = 0.5<br>_MyTexture ("Texture", 2D) = "white" {}<br>_MyCubemap ("Cubemap", CUBE) = "" {} | fixed4 _MyColor;<br>float4 _MyVector;<br>float _MyFloat;<br>sampler2D _MyTexture;<br>samplerCUBE _MyCubemap;<br><br>Cg / HLSL은 uniform 키워드를 사용할 수도 있지만 반드시 필요한 것은 아님.<br><br>uniform float4 _MyColor; |

- **Color :** 컬러 팔레트를 변수로 사용한다.


    - _Name ( “display name”, Color ) = (기본값, 기본값, 기본값, 기본값)
    - ex) _TintColor(“Test Color”, color) = (1, 1, 1, 1)

```
_Intensity("Range Sample", Range(0, 1)) = 0.5
_Intensity( “Range Sample” , Range(0, 1)) = 0.5
```
[2]

- **Range :** 범위값을 변수로 사용한다.
    - _Name ( “display name”, Range(min, max) ) = 기본값
    - ex) _Intensity(“Range Sample”, Range(0, 1)) = 0.5


[1] 컴퓨터 프로그래밍에서 변수(變數) 또는 스칼라(scalar)는 아직 알려지지 않거나 어느 정도까지만 알려져 있는 양이나 정보에 대한 상징적인 이름이다. 컴퓨터 소스 코드에서의 변수 이름은 일반적으로 데이터 저장 위치와 그 안의 내용물과 관련되어 있으며 이러한 것들은 프로그램 실행 도중에 변경될 수 있다.

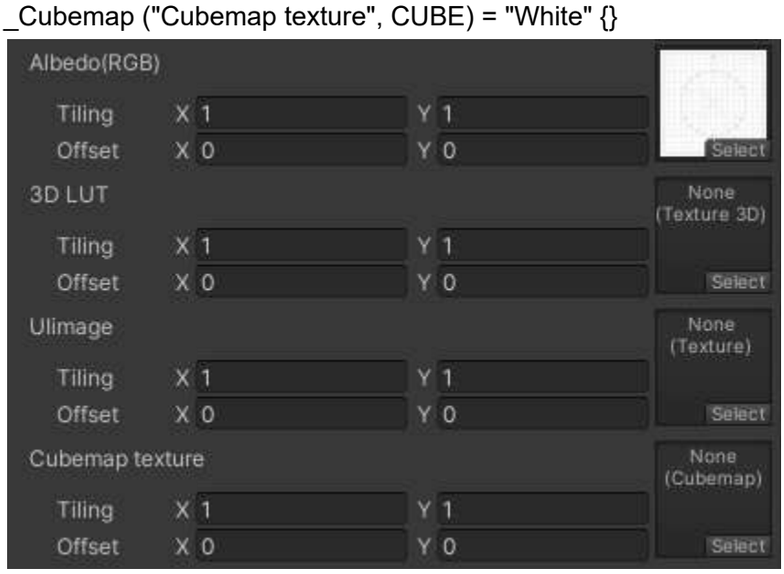
[2] 문서에서 복사/붙여넣기를 하는경우 셰이더에서 변수명에 사용되는 세미콜론이 다른게 들어가서 에러가 나는 경우가 있다. VS 인텔리전스에서는 시각적으로 확인이 가능하니 유심히 살펴보자.

- **Vector :** xyzw 4개의 값(벡터)을 가지는 변수를 사용한다.
  - _Name ( “display name”, Vector ) = (기본값, 기본값, 기본값, 기본값)
  - ex) _Vector("Vector Sample", vector) = (1,1,1,1)


- **Int, float :** 각각의 정밀도를 가진 변수를 사용한다.
  - Int :  _Name ( “display name”, Int ) = 기본값
  - float : _Name ( “display name”, Float ) = 기본값
  - ex) _Int("Int value", Int) = 1
  - _Float("Float value", Float) = 1


- **Texture Sampler 타입의 properties 변수**. 텍스쳐를 입력 받지 않을때 기본 색상을 지정해야 한다. 지정할 수 있는 색상은 white, black, gray, bump
  - 2D : _Name ( “display name”, 2D ) = “White” {}
  - 3D : _Name ( “display name”, 3D ) = “White” {}
  - Rect : _Name ( “display name”, Rect ) = “White” {}
  - CUBE : _Name ( “display name”, CUBE ) = “White” {}

  - ex)_MainTex ("Albedo(RGB)", 2D ) = "white" {}
  - _3DTex  ("3D LUT", 3D) = "White" {}
  - _UIimage ("UIimage", Rect) =  "White" {}
  - _Cubemap ("Cubemap texture", CUBE) = "White" {}




- **2DArray :** 2D 텍스쳐 어레이 프로퍼티를 사용할때 쓰는 변수타입
  - 2D Array : _Name ( “display name”, 2DArray ) = “ ” {}
  - ex) _MyArr ("Tex", 2DArray) = "" {}


SubShader

메시를 렌더링 할 때 Unity는 대상 장치의 GPU와 호환되는 첫 번째 SubShader 블록을 선택하는데 SubShader 블록은 SubShader Tags 블록을 선택적으로 포함 할 수 있다. Tags 키워드를 사용하여 SubShader Tags 블록을 아래와 같이 선언하게 된다.

| Tag {"RenderPipeline"= "UniversalPipeline"  "RenderType"= "Opaque" “Queue”=”Geometry”} |
|---|

RenderPipeline 은 SubShader를 사용할 렌더 파이프 라인을 Unity에 선언하게 된다. UniversalPipeline은 URP를 사용하는 것을 선언한다. 만약, 다른 렌더 파이프 라인에서 동일한 셰이더를 사용하려면 다른 RenderPipeline 태그 값으로 여러 SubShader 블록을 만들어서 사용하면 된다.(bulit-in 렌더 파이프 라인에서 실행하려면 RenderPipeline을 빈 값으로 설정하면 된다.).

### Shader Tag

SubShader 내부의 tag는 렌더링 순서 및 기타 매개 변수를 결정하는데 사용하게 된다. 이 tag는 pass가 아닌 subshader섹션에 위치해야 한다.[3]

Rendering Order - Queue Tag
Queue tag를 사용해 오브젝트를 그리는 순서를 결정할 수 있으며 이는 오브젝트가 속한 렌더 대기열을 결정하게 된다. 유니티는 내부에서 해당 렌더큐에 대한 값을 임의로 지정해 두었다.

- Background(1000) : 다른것들보다 먼저 렌더링 하게 된다. 일반적으로 백그라운드에 있어야 하는 작업에 사용
- Geometry(Default-2000) : 대부분의 불투명 오브젝트에 해당. 기본값
- AlphaTest(2450) : 알파 테스트를 거친 Geometry는 이 큐값을 사용하게 된다. 모든 불투명 오브젝트를 그린후에 알파테스트된 오브젝트를 렌더링 하는것이 효율적이기 때문에 이를 별개의 큐로 구분해서 사용하게 된다.
- Transparent(3000) : Geometry 및 AlphaTest이후에 이 렌더큐 항목은 뒤에서 앞의 순서로 렌더링 하게 된다. 알파 블렌딩된 모든것(예를 들어 depth buffer에 쓰지 않은 셰이더)은 여기로 이동하게 된다(예 : 유리, 파티클 이펙트등)
- Overlay(4000) : 오버레이 효과를 위한 것으로 마지막으로 렌더링 된 모든 항목이 여기에 해당.(예 : 렌즈 플레어)

렌더큐값 2500까지는 불투명(opaque) 메시로 간주되며 이 값 이후부터는 불투명(transparent)으로 간주된다.(Geometry+500)

Render type
RenderType 태그는 셰이더를 미리 정의 된 여러 그룹으로 분류하게 된다. 불투명한 셰이더 또는 알파 테스트를 거친 셰이더 등이 여기에 해당되며, 이것은 셰이더 교체에서 사용되거나 경우에 따라 카메라의 depth texdture를  생성하는데 사용된다.

- Opaque : 대부분의 불투명 셰이더
- Transparent : 대부분의 반투명 셰이더.
- TransparentCutout : mask된 투명 셰이더(AlphaTest 된 셰이더가 여기에 해당된다)
- Background : skybox shader
- Overlay : Halo, Flare shader
- TreeOpaque : 터레인에서 나무와 줄기부분
- TreeTransparentCutout : 터레인의 나무 잎 부분
- TreeBillboard : 터레인의 빌보드된 나무
- Grass : 터레인의 grass
- GrassBillboard : 터레인의 빌보드된 잔디.

### Pass

Pass block은 shader에서 사용할 각 렌더 패스를 결정한다. 패스에서 LightMode 태그를 설정하지 않으면 URP는 해당 패스에 대해 SRPDefaultUnlit 태그 값을 사용하게 된다.

```
Pass { [Name and Tags] [RenderSetup] }
```

Basic 과정에서는 각 Pass를 다루지는 않는다. 개략적인 구조는 아래 simpleLit shader의 구조를 참고한다.

```
SubShader
  {
    Tags { "RenderType" = "Opaque"
           "RenderPipeline" = "UniversalPipeline"
           "UniversalMaterialType" = "SimpleLit"
           "IgnoreProjector" = "True"
            "ShaderModel"="4.5"}
     Pass
     {
       Name "ForwardLit"
       Tags { "LightMode" = "UniversalForward" }
       …
     }
```

[3] 이 tag항목은 유니티 내부 빌트인 태그 이외에도 Material.GetTag function을 사용해 쿼리할 수 있다.

```
        Pass
        {
            Name "ShadowCaster"
            Tags{"LightMode" = "ShadowCaster"}
          …
        }
        Pass
        {
            Name "GBuffer"
            Tags{"LightMode" = "UniversalGBuffer"}
        …
        }
        Pass
        {
            Name "DepthOnly"
            Tags{"LightMode" = "DepthOnly"}
        …
        }
```

### HLSL Snippet(HLSLPROGRAM ~ ENDHLSL)

Unity SRP는 HLSL로 주로 작성되며 HLSLPROGRAM 으로 시작해서 ENDHLSL로 끝나는 구조를 가지고 있다.

- HLSL(High Level Shader Language) : DX(DirectX) 기반의 Shader 언어
- GLSL(OpenGL Shader Language) : OpenGL 기반의 Shader 언어
- Gg : Nvidia에서 만들어진 Shader 언어

HLSL 프로그램 스니핏은 CGPROGRAM과 ENDCG 키워드 사이 또는 HLSPROGRAM과 ENDHLSL 사이에 작성됩니다. 후자는 자동으로 HLSLSupport와 UnityShaderVariables 빌트인 헤더 파일을 포함하지 않는다.

CG버젼은 #include “HLSLSupport.cginc”를 자동으로 Shader에 추가하며, 기본 shader library를 계속 사용하므로 파일을 셰이더에서 include 않는 것을 권장한다.

몇몇 전처리 구문이 동작하지 않는다.

CG shader를 사용할 경우에는 .cginc 파일을 include 하지만 HLSL에서는 .hlsl 파일을 include하게 된다.

### pragma

C언어에서 pragma는 컴파일러가 입력을 처리하는 방법을 지정하는 언어 문법이며 이는 셰이더가 어떻게 컴파일 될지를 지시하는 전처리 문법이다.

```
        #pragma prefer_hlslcc gles
```

~~Unity에는 다양한 셰이더 컴파일러와 크로스 컴파일러를 사용하는데, 안드로이드를 위해서는 SRP에서는 HLSLcc를 사용한다. OpenGL ES 그래픽 API(예 : Android)를 사용하는 플랫폼에서는 HLSLcc가 기본적으로 사용되지 않으므로 강제로 실행할 필요가 있다.~~

~~hlslcc는 DirectX 셰이더 바이트 코드 크로스 컴파일러로써 https://github.com/James-Jones/HLSLCrossCompiler 기반의 셰이더 컴파일러로 다음 언어로 변환 된다.~~

- ~~GLSL (OpenGL 3.2 이상)~~
- ~~GLSL ES (OpenGL ES 2.0 이상)~~
- ~~GLSL for Vulkan consumption (as input for Glslang to generate SPIR-V)~~
- ~~Metal Shading Language~~

~~이 Library는 Unity for OpenGL, OpenGL ES 3.0+, Metal 및 Vulcan의 모든 셰이더 언어를 생성하는데 사용되게 된다.~~

~~보다 구체적인 정보는 https://github.com/Unity-Technologies/HLSLcc 에서 확인할 수 있다.~~

```
#pragma exclude_renderers d3d11_9x
```

~~DirectX 9는 더 이상 지원되지 않으므로 제외하게 된다.~~

```
#pragma vertex vert
#pragma fragment frag
```

~~Vertex shader와 Pixel shader를 정의한다.~~

2025년 추가사항. **Unity HLSL은 더이상 hlslcc를 사용하지 않고 DXC를 컴파일러로 사용한다.**

Unity에서 HLSL을 처리할 때 사용하는 **셰이더 컴파일러**는 플랫폼 및 설정에 따라 여러 가지를 사용할 수 있고, Unity 버전이 올라갈수록 **DXC (DirectX Shader Compiler)** 중심으로 전환되고 있다.아래는 HLSL 컴파일러 구성을 간략히 정리한 내용이다.

- DirectX (Windows) FXC 또는 DXC DX11/12용. 최신 Unity는 DXC 기본
- Metal (macOS, iOS) Unity 자체 HLSL-to-MSL 변환기 HLSL → MSL 번역 후 Metal 셰이더 컴파일러 사용
- Vulkan DXC → SPIR-V DXC로 HLSL → SPIR-V 컴파일
- OpenGL/WebGL Unity GLSL Translator HLSL → GLSL로 번역 (자체 변환기)
- Consoles (PS5, Switch 등) 각 플랫폼 SDK 셰이더 컴파일러 HLSL → 해당 플랫폼 셰이더로 번역됨

**주요 Compiler**

FXC (기존)

- 오래된 DirectX HLSL 컴파일러 (d3dcompiler_47.dll 등)
- DX11 대상
- 단점: 느림, SPIR-V/Metal 대응 어려움

DXC (현행)

- 최신 DirectX Shader Compiler (LLVM 기반)
- HLSL 2021, Shader Model 6.x 지원
- SPIR-V (Vulkan)와 Metal도 직접 지원 가능
- Unity 2021.2 이후부터 점진적으로 DXC 전환 시작됨

```
// DXC 기반 Shader 컴파일 로그 예시
Compiling shader with DXC...
Target: SM 6.0
```

**Unity 설정에서 DXC 사용 확인 방법**(Unity 2023+ 기준)

- Project Settings → Graphics에서 DXC 사용 옵션 있음
- PlayerSettings.SetUseDXC(true) (API에서도 가능)
- Shader import 로그나 -logShaderCompilation CLI 옵션에서 확인 가능

**DXC 사용의 장점**

- SPIR-V 출력 Vulkan/Android에서 더 정밀한 결과 + 에러 검출 향상
- Shader Model 6.x Wave Intrinsics, Raytracing, bindless 지원
- Cross-platform 번역 DXC는 HLSL → MSL, SPIR-V, DXIL 모두 출력 가능
- Metal 대응 Unity 내부적으로 Metal로 번역할 때 정확도 향상

**hlslcc는 이제 거의 사용 되지 않음**

- hlslcc 사용 : Legacy 플랫폼(특히 GLES2, WebGL1)에서만 부분 사용. 구버전 플랫폼 하위 호환용으로 존재하지만 일반적 사용은 권장되지 않음
- Unity 최신 버전 (2021~2023 이상)에서는 대부분 DXC, 자체 변환기(GLSL, MSL)로 전환됨

- DXC가 HLSL → SPIR-V → Vulkan/Metal 번역 가능 (더 정확하고 robust함)
- Unity는 자체 GLSL 변환기(GLSLTranslator)를 보유 (hlslcc 대체)
- 오래된 GLSL 버전 대응, layout/binding 오류 발생 등. 또한, hlslcc 업데이트 중단 추세

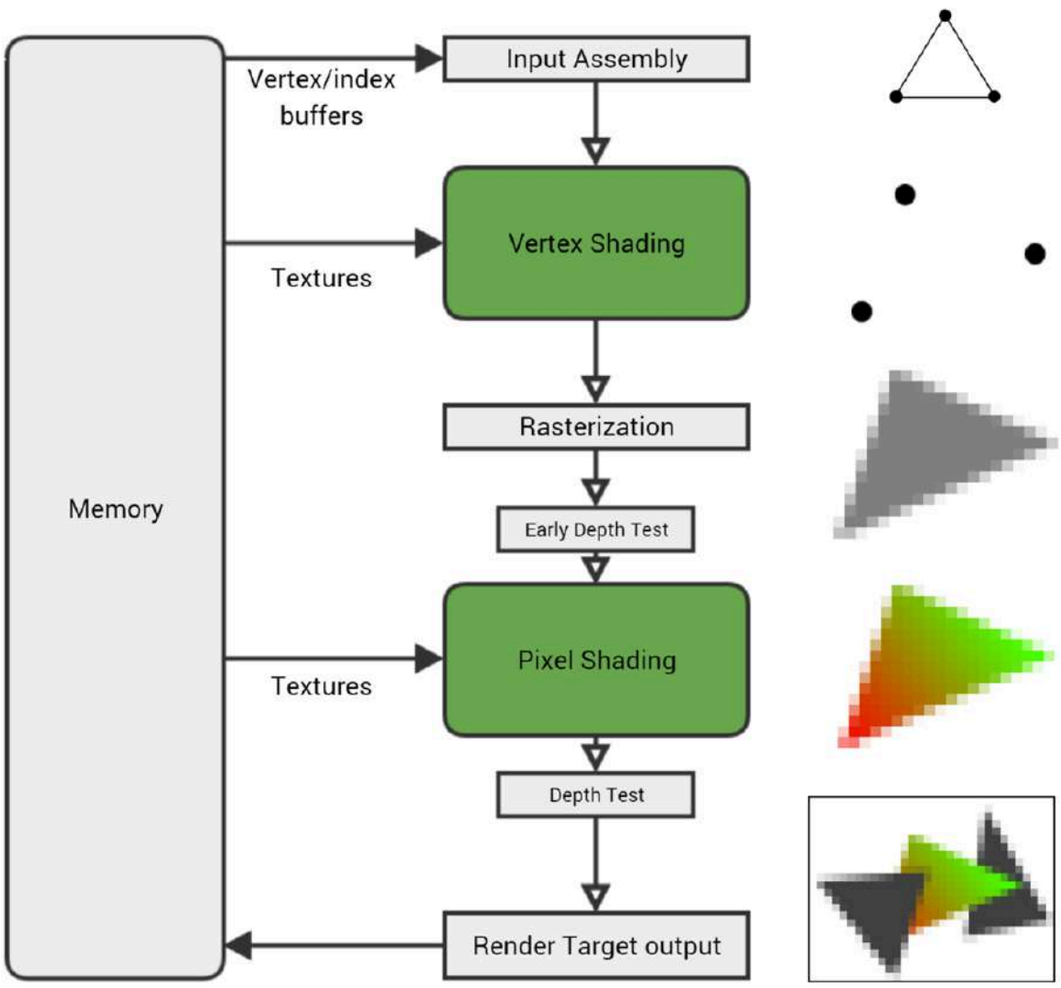
```
#pragma vertex vert
#pragma fragment frag
#pragma hlslcc_define FANCY_FEATURE 1
```

**플랫폼사용 중인 변환기 / 컴파일러**

| Platform | Compiler |
|---|---|
| Windows / DirectX | DXC (HLSL → DXIL) |
| Vulkan / Android | DXC (HLSL → SPIR-V) |
| Metal (macOS/iOS) | Unity 자체 Metal Translator |
| WebGL (ES 3.0) | Unity GLSL Translator 사용 |
| WebGL (ES 2.0, Legacy) | **일부 hlslcc fallback 가능성 있음** |

추가 내용은 해당 포스팅 참고 : https://illu.tistory.com/1488

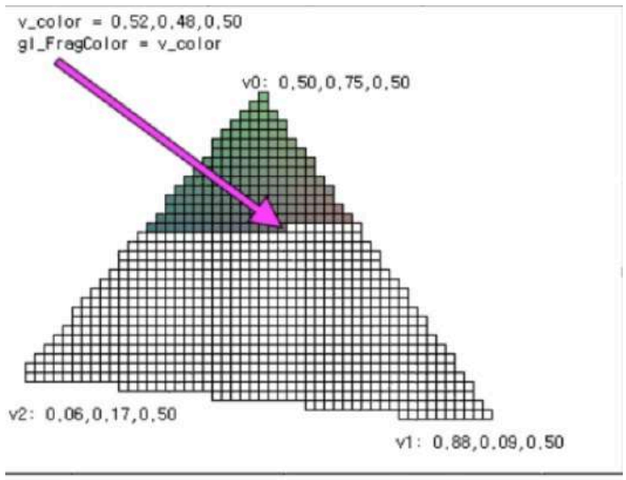
Rendering Pipeline




**Input Assembly :** GPU는 메모리에서 버텍스 및 인덱스 버퍼를 읽고, 버텍스가 Tri를 형성하는 방식을 결정하고 나머지는 파이프 라인에 전달.

**Vertex Shading :** 한번에 하나의 버텍스에서 실행되는 메쉬의 모든 버텍스에 대해 한번 실행된다. 여기에서 버텍스를 변환하고 위치를 받은다음 카메라 및 뷰포트 설정을 사용해 화면서 최종 위치를 계산

**Rasterization :** 버텍스 셰이더가 삼각형의 각 버텍스에 실행되고 GPU가 화면에 표시될 위치를 알고 있으면 삼각형이 레스터화 되어 개별 픽셀의 모음으로 변환된다. 버텍스별 정보(UV좌표, 버텍스 컬러, 노멀등)은 삼각형 픽셀을 통해 보간된다(레스터화 된 픽셀은 각 버텍스의 보간된 값을 가지게 된다)



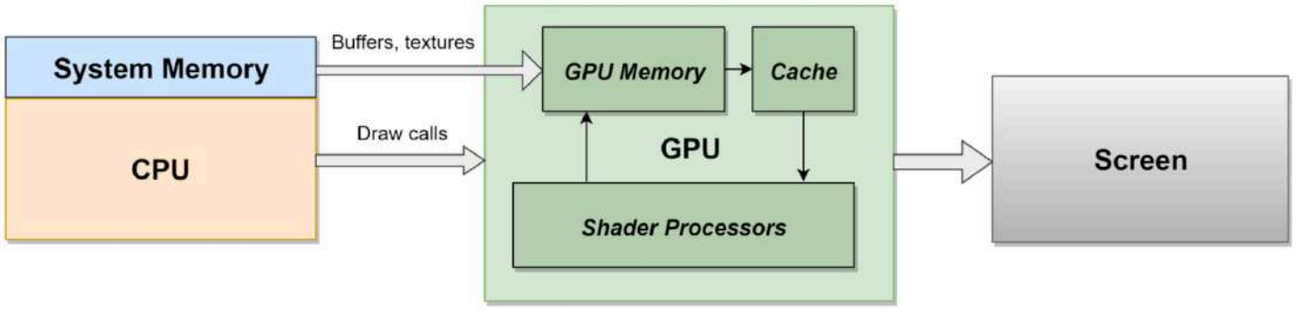

v_color is interpolated between v0, v1 and v2

**Pixel Shading :** 레스터화 된 각 픽셀은 픽셀 셰이더를 통해 실행(기술적으로는 아직 픽셀이 아닌 프래그먼트 셰이더라고 한다). 매터리얼 속성, 텍스쳐, 광원등 기타 매개 변수를 프로그래밍 된 방식으로 결합해 특정 색상을 얻음으로써 픽셀에 색상을 부여

**Render Target output :** 마지막으로 픽셀은 렌더링 대상에 쓰여지지만 일부 테스트를 거쳐야 유효하게 된다. 예를 들어 깊이 테스트는 이미 렌더링 대상보다 멀리 떨어진 픽셀을 생략할 수 있다. 이러한 모든 테스트(Depth, alpha, stencil등)를 통과하면 메모리의 렌더링 타겟에 쓰여지게 된다.

**Z sort 방식 :** 렌더링 하는 물체를 카메라 시점으로부터 거리로 소팅(정렬)해 두고 시점으로 부터 먼 것부터 순서대로 렌더링 하는 방식. 뒤에 있는 것을 앞에 있는 물체가 덮어 씌워간다.(메쉬가 겹칠경우 소팅이슈 발생)

**Z buffer 방식 :** 물체를 렌더링하면서 그 물체까지의 거리를 픽셀(텍셀) 단위로 깊이 버퍼에 저장해두고 다음에 런더링 할때 이미 렌더링 되어 있는 물체의 거리와 렌더링 하려는 물체의 거리를 비교하면서 앞에 있으면 렌더링(반투명의 경우 렌더링 어려움)




Application to Vertex Shader Structure(Attributes)

```
struct[4] VertexInput
        {
        float4 vertex : POSITION;
        };
```

| Type(s) | Name(변경가능) | Semantic | Notes |
|---|---|---|---|
| float4 | vertex | **POSITION** | 로컬 공간 (모델 공간)의 정점 위치 |
| float3 | normal | **NORMAL** | 버텍스의 노멀 |
| float4 | texcoord[n] | **TEXCOORD[n]** | 버텍스의 UV 좌표. |

[4] 구조체 형(Structure Types) : struct명령어는 구조체 형을 정의할때 사용하며, 구조체가 정의되면 ID를 통해서 접근할 수 있다. struct [id] { member_list } member_list는 하나 이상의 멤버 선언으로 구성되는데, 초기값을 가질수 없으며, 또한 static, extern, volatile, const등의 접근범위 명령어를 사용해서 개별적으로 선언될 수 없다.

| | | | |
|---|---|---|---|
| float4 | tangent | **TANGENT** | 메시에서 계산된 또는 import된 탄젠트 값 |
| float4 | color | **COLOR** | 버텍스의 컬러값. |

Input assembly 단계에서 버텍스 버퍼에서 필요한 정보를 가져오는 역할을 한다.[5]

*LitForwardPass.hlsl*

```
struct Attributes
{
    float4 positionOS   : POSITION;
    float3 normalOS     : NORMAL;
    float4 tangentOS    : TANGENT;
    float2 texcoord     : TEXCOORD0;
    float2 lightmapUV   : TEXCOORD1;
    UNITY_VERTEX_INPUT_INSTANCE_ID
};
```

*LitMetaPass.hlsl*

```
struct Attributes
{
    float4 positionOS   : POSITION;
    float3 normalOS     : NORMAL;
    float2 uv0          : TEXCOORD0;
    float2 uv1          : TEXCOORD1;
    float2 uv2          : TEXCOORD2;
#ifdef _TANGENT_TO_WORLD
    float4 tangentOS     : TANGENT;
#endif
};
```

URP Lit shader에서는  LitForwardPass.hlsl 이나 GbufferPass.hlsl, LitMetaPass.hlsl 등 각 패스별 hlsl파일에 선언되어있다.

### Vertex Shader to Fragment Shader Structure (Varying)

| Type(s) | Name(변경가능) | Semantic | Notes |
|---|---|---|---|
| float4 | pos | **SV_POSITION** | 투영공간으로 변환된 후의 버텍스 포지션 |
| float3 | normal | **NORMAL** | 뷰 공간으로 변형된 후 버텍스의 노멀값 |
| float4 | uv | **TEXCOORD[n]** | 첫번째 UV 채널 좌표값 |
| float4 | tangent | **TANGENT** | 탄젠트 값 |
| float4 | diff, spec | **COLOR** | 버텍스 컬러 혹은 그외의 임의의 컬러값 |
| Any | Any | **사용자 정의** | Vertex stage에서 계산한 값을 Pixel shader로 전달할 때 임의로 정의할 수 있다. |

interpolate(보간기)의 숫자는 shader model에 영향을 받는다. Advanced의 #pragma target 설정을 참고.

struct Varyings[6]

```
{
    float4 positionCS   : SV_POSITION;
    float2 uv           : TEXCOORD0;
};
```

### Vertex Function(Vertex shader stage)

```
VertexOutput vert(VertexInput v)
```

[5] MSDN Semantics https://docs.microsoft.com/en-us/windows/win32/direct3dhlsl/dx-graphics-hlsl-semantics
[6] 이부분은 사용자 정의이므로 임의로 선언할 수 있다.

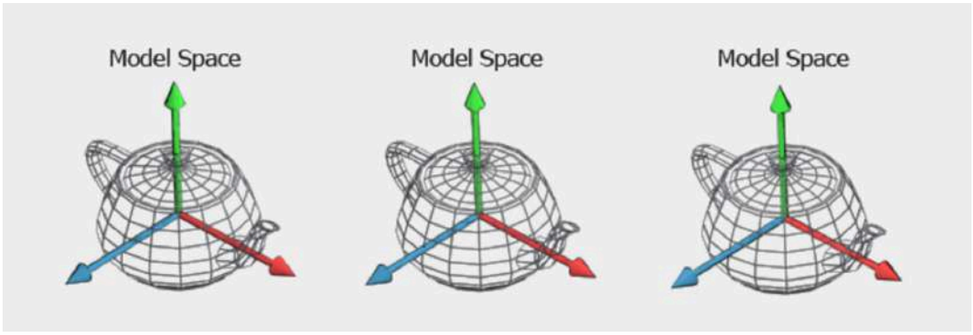
```
{
        VertexOutput o;
        o.vertex = TransformObjectToHClip(v.vertex.xyz);
        return o;
}
```

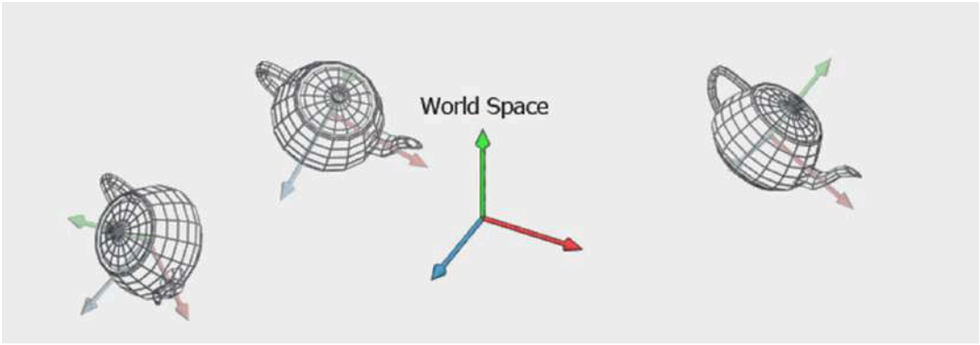
버텍스 셰이더의 주 목적은 3D의 정보를 2D로 변화하는데 있다. 이를 위해 필요한 정보를 vertex buffer에서 가져온 값을 사용하게 되며(VertexInput v), 계산된 결과를 VertexOutput o를 통해 보간기로 픽셀 셰이더에 전달하게 된다. 예제코드에서는 버텍스의 포지션 정보만을 계산해 화면에 그리는 아주 단순한 코드로 구성되어 있다. 이를 통해 셰이더에서 어떻게 이 과정이 처리되는지를 살펴본다.

### 공간 변환(transform)

3D 메시의 각 버텍스를 사용하는 툴의 3D공간으로 변환하는 과정을 거치게 된다. 이 변환 과정은 간략하게 [오브젝트 공간] >> [월드 공간] >> [카메라 공간] >> [클립 공간] 으로 거치게 된다. 초급과정에 아래 내용을 완전히 이해하기 힘들다면 MVP를 통해 화면에 그려지는 위치가 결정된다고 이해하면 된다.

- 월드 변환(Transform Object to World) : Local Space, Object Space, 물체공간 모두 다 같은 의미로 오브젝트 별로 가지고 있는 자기만의 공간을 월드 공간내의 좌표로 변환하는 과정을 이야기 한다.



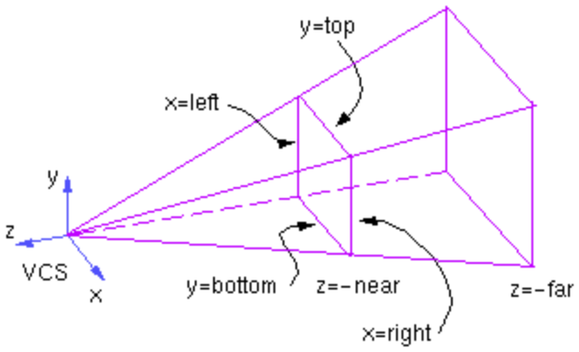

```
float4x4 GetObjectToWorldMatrix()[7]
{
    return UNITY_MATRIX_M;
}

...

float3 TransformObjectToWorld(float3 positionOS)
{
    return mul(GetObjectToWorldMatrix(), float4(positionOS, 1.0)).xyz;
}
```

```
float4x4 GetWorldToObjectMatrix()
{
    return UNITY_MATRIX_I_M;
}
```

- 카메라 변환(Transform World to View) : 월드 공간으로 변환된 물체를 카메라의 위치를 기준으로 다시 공간을 변환하는데 이를 카메라 변환이라고 한다.

  이때 카메라의 가시 볼륨 영역에서 near(근평면), far(원평면)에 의해 절단되어 View Frustum(절두체)의 영역으로 다시 정의된다.

[7] Core/ShaderLibrary/SpaceTransforms.hlsl 을 살펴보면 확인할 수 있다.

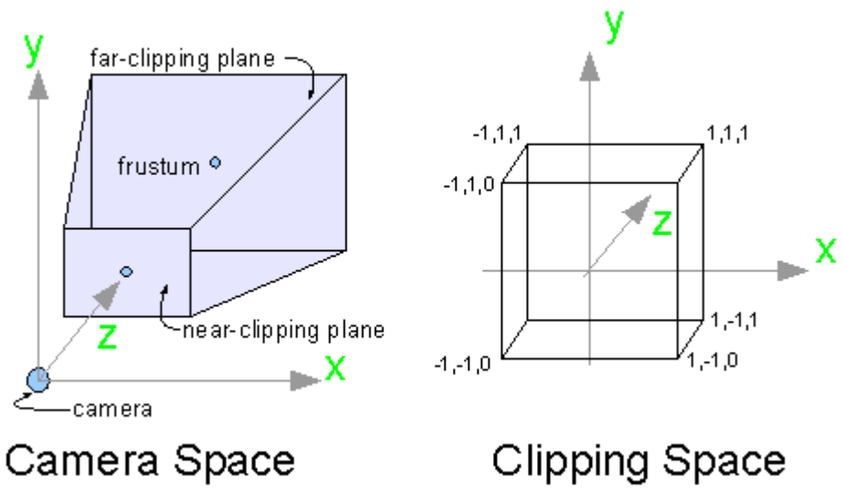



[view frustum]

```
float4x4 GetObjectToWorldMatrix()
{
    return UNITY_MATRIX_M;
}
…

float3 TransformWorldToView(float3 positionWS)
{
    return mul(GetWorldToViewMatrix(), float4(positionWS, 1.0)).xyz;
}
```

```
float4x4 GetWorldToObjectMatrix()
{
    return UNITY_MATRIX_I_M;[8]
}
```

- 투영 변환 : (Transform Projection) : 투영 변환은 원근법을 구현하기 위해 카메라 공간에서 정의된 절두체를 축에 나란한 직육면체 볼륨으로 변경하여 카메라 공간의 모든 물체를 3차원 클립 공간으로 변환하는 것을 의미한다.




```
 // Transform to homogenous clip space
float4x4 GetViewToHClipMatrix()
{
    return UNITY_MATRIX_P;
}

// Tranforms position from view space to homogenous space
float4 TransformWViewToHClip(float3 positionVS)
{
    return mul(GetViewToHClipMatrix(), float4(positionVS, 1.0));
}
```

원근 처리를 위해 3차원 공간을 직육면체 볼륨으로 물체들을 변환시키면서 카메라 공간 밖의 폴리곤들을 잘라내게 된다.(Clipping)

[8] 위 행렬의 역행렬(Inverse Matrix)

이를 간략하게 처리하면 아래와 같다.

```
// Transforms position from object space to homogenous space
float4 TransformObjectToHClip(float3 positionOS)
{

// More efficient than computing M*VP matrix product
return mul(GetWorldToHClipMatrix(), mul(GetObjectToWorldMatrix(), float4(positionOS, 1.0)));

}
```

위 내용을 풀어서 다시 버텍스 스테이지에서 계산된 내용을 살펴보면 아래와 같다.

```
VertexOutput vert(VertexInput v)
// VertexInput : vertex buffer에서 읽어온 정보는 v를 붙여 계산에 사용한다.
{
    VertexOutput o;
    //Vertex stage에서 계산한 결과는 o.를 붙여 보간기를 통해 픽셀 셰이더로 전달한다.
    o.vertex = TransformObjectToHClip(v.vertex.xyz);[9]
    // Vertex buffer에서 읽어온 값을 함수에 넣어 계산한다음 o.vertex를 통해 픽셀 셰이더로 전달
    return o;
}
```

### Fragment Function(Pixel shader stage)

픽셀로 배열된 오브젝트의 최종 컬러값을 계산하는 단계이다.

```
half4 frag(VertexOutput i) : SV_Target
{
return half4(_TintColor);
}
```

## 주석 처리

코드의 특정 라인을 컴파일시 포함하지 않는 줄을 지정한다. 해당라인은 셰이더 연산에서 제외되며 코드에 필요한 내용을 담거나 수정할때 지우지 않고 남겨둘 코드등을 위해 주로 사용된다.

// 를 문장의 가장 앞에 붙이면 해당 라인은 컴파일시 포함되지 않는다.

```
//#pragma multi_compile_fog
```

/* ~ */ 안에 포함되는 내용은 줄에 상관없이 주석 처리 된다.

```
 /*
[Enum(UnityEngine.Rendering.CullMode)] _Cull ("Cull", Float) = 2
[Enum(UnityEngine.Rendering.BlendMode)] _SrcBlend ("Src Blend", Float) = 1
[Enum(UnityEngine.Rendering.BlendMode)] _DstBlend ("Dst Blend", Float) = 0
*/
```

## Properties에 Range 항목을 추가하고 색 밝기/변경으로 슬라이드로 조절하기

Properties에 Range 변수및 변수 타입 선언

```
_Intensity("Range Sample", Range(0, 1)) = 0.5
// 색상의 세기를 최소 0에서 최대 1로 조절하는 변수 선언. 기본값은 0.5

_TintColor(“Tint Color”, color) = (1, 1, 1, 1)
// color 지정하는 변수를 선언. 변수 타입은 color 기본값은 RGBA (1,1,1,1)
```

Shader내에 변수 타입 선언

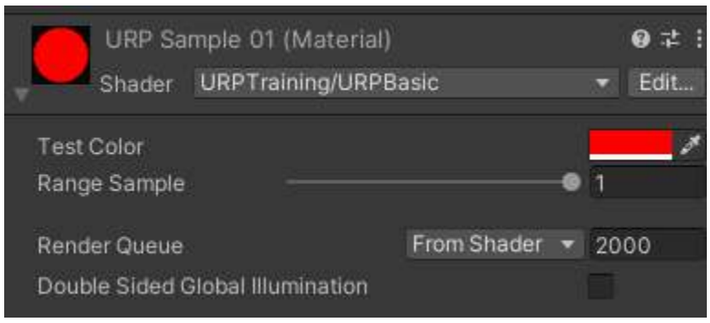
```
float _Intensity;

float4 _TintColor;
```

[9] URP Shader code를 살펴보면 positionOS라고 되어있는데 이는 Object Space를 줄여서 정의한것. positionWS는 World Space

Pixel Shader에서 계산식 추가

```
half4 frag(VertexOutput i) : SV_Target
{
        float4 color = _TintColor * _Intensity;

        return color;
}
```

최종 결과물




아래와 같이 작성해도 된다.

```
half4 frag(VertexOutput i) : SV_Target
{
        return float4(_TintColor * _Intensity);
}
```

## Texture Sampling

### Coupled textures and samplers

메모리에 올려진 텍스쳐를 해당 픽셀에 매핑 하는 과정.[10] 샘플러와 텍스쳐를 연결해서 쓰는경우에는 아래와 같이 사용하게 된다.

우선 Properties에 사용할 텍스쳐를 정의 한다.

```
_MainTex("Main Texture", 2D) = "white" {}
```

그리고 shader 내부에 texture sampling을 선언해 준다.

```
sampler2D _MainTex;
float4 _MainTex_ST;[11]
```

Vertex buffer에서 uv 정보를 읽어오는 semantic을 선언한다..

```
struct VertexInput
        {
                float4 vertex : POSITION;
                float2 uv       : TEXCOORD0;
        };
```

Pixel shader로 전달하기 위한 보간기를 선언한다.

```
struct VertexOutput
{
        float4 vertex    : SV_POSITION;
        float2 uv        : TEXCOORD0;
};
```

Vertex shader에서 이제 이를 계산한다.

```
VertexOutput vert(VertexInput v)
        {
                VertexOutput o;
                o.vertex = TransformObjectToHClip(v.vertex.xyz);
                o.uv = v.uv;
```

[10] 정확히는 Texture filtering 된 texel의 정보값을 해당 pixel에 매핑하는 과정이다.
[11] tile과 offset을 사용하기 위한 값.

```
            return o;
        }
```

Pixel shader에서 이제 이를 계산해준다.

```
    half4 frag(VertexOutput i) : SV_Target
        {
            float2 uv = i.uv.xy * _MainTex_ST.xy + _MainTex_ST.zw;[12]
            float4 color = tex2D(_MainTex, uv) * _TintColor * _Intensity;
            return color;
         }
```

위 방법은 DX9 스타일의 셰이더를 사용할때 기본 구문이며, 구형 그래픽스 API(OpenGL ES)에서는 이 옵션만 지원.

### Separate textures and samplers

많은 그래픽스 API에서는 texture보다 적은 sampler를 사용할 수 있으므로 연결된 texture+sampler구조보다 이를 분리해서 관리하는것이 더 많은 텍스쳐를 사용할 수 있다.

우선 shader 내부에 texture sampling 선언을 아래와 같이 해준다.

```
    float4 _MainTex_ST;[13]
    Texture2D _MainTex;
    SamplerState sampler_MainTex;  // "sampler" + "_MainTex"[14]
```

Pixel shader에서 이제 이를 계산해준다.

```
    half4 frag(VertexOutput i) : SV_Target
        {
            float2 uv = i.uv.xy * _MainTex_ST.xy + _MainTex_ST.zw;
            float4 color = _MainTex.Sample(sampler_MainTex, uv) * _TintColor * _Intensity;
            return color;
        }
```

이렇게 사용하면 하나 이상의 텍스쳐를 샘플링 하면서 샘플러를 다른 텍스쳐에 재사용하면서 셰이더를 작성할 수 있다.

```
    Texture2D _MainTex;
    Texture2D _SecondTex;
    Texture2D _ThirdTex;
    SamplerState sampler_MainTex; // "sampler" + "_MainTex"

    half4 color = _MainTex.Sample(sampler_MainTex, uv);
    color += _SecondTex.Sample(sampler_MainTex, uv);
    color += _ThirdTex.Sample(sampler_MainTex, uv);
```

그러나 이렇게 작성된 샘플러는 일부 구형 플랫폼(예: OpenGL ES 2.0)에서는 동작하지 않는다.
#pragma target 3.5를 지정해 구형 플랫폼 지원을 생략하고 셰이더를 작성할 수 있다.

유니티에서는 빌트인 매크로를 통해 분리된 샘플러 접근에 대한 여러 코드를 제공하고 있다.[15]

Shader 내부에 선언하는 texture와 sampler를 아래와 같이 선언할 수 있다.

```
    TEXTURE2D(_MainTex);
    SAMPLER(sampler_MainTex);
```

Pixel shader에서는 아래와 같이 계산된다.[16]

```
    float4 color = SAMPLE_TEXTURE2D(_MainTex, sampler_MainTex, i.uv)* _TintColor * _Intensity;
```

전체 셰이더 코드는 아래와 같다.

```
    Shader "URPTraining/URPBasic"
    {
```

---

[12] 이 항목은 사실 vertex stage에서 구해도 차이는 없다. (일반적인 경우에는 성능을 고려한다면 vertex stage 권장)
[13] 각 texture 속성에 대해 unity는 _ST로 끝나는 float4 uniform 값을 제공한다. xy는 Tile을, zw는 scale값으로 사용한다.
[14] LWRP 이전 버젼에서 tex2D가 아닌 SamplerState 를 사용한경우에 inspector에서 타일과 오프셋이 작동하지 않는 경우가 있다.
[15] https://docs.unity3d.com/kr/2020.2/Manual/SL-SamplerStates.html
[16] 이 경우에는 인스펙터의 tiles와 offset이 동작하지 않는다

```
    Properties {

        _TintColor("Test Color", color) = (1, 1, 1, 1)
        _Intensity("Range Sample", Range(0, 1)) = 0.5
        _MainTex("Main Texture", 2D) = "white" {}

    }

    SubShader
    {
    Tags
    {
       "RenderPipeline"="UniversalPipeline"
       "RenderType"="Opaque"
       "Queue"="Geometry"
    }
    Pass
    {
     Name "Universal Forward"
    Tags {"LightMode" = "UniversalForward"}

    HLSLPROGRAM
    #pragma prefer_hlslcc gles
    #pragma exclude_renderers d3d11_9x

    #pragma vertex vert
    #pragma fragment frag

    #include "Packages/com.unity.render-pipelines.universal/ShaderLibrary/Lighting.hlsl"

    half4 _TintColor;
    float _Intensity;

    float4 _MainTex_ST;
    Texture2D _MainTex;
    SamplerState sampler_MainTex;

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

    struct VertexOutput
     {
           float4 vertex    : SV_POSITION;
           float2 uv          : TEXCOORD0;
     };

    VertexOutput vert(VertexInput v)
     {
    VertexOutput o;

    o.vertex = TransformObjectToHClip(v.vertex.xyz);
    o.uv = v.uv.xy * _MainTex_ST.xy + _MainTex_ST.zw;

    return o;
     }

    half4 frag(VertexOutput i) : SV_Target
     {
    float4 color = _MainTex.Sample(sampler_MainTex, i.uv) * _TintColor * _Intensity;
    return color;
     }
    ENDHLSL
     }
  }

}
```

# Alphatest Shader 작성하기

## Alphatest(Cutout) 이란?

불투명(opaque) 오브젝트를 그린후 알파 테스트를 거쳐 픽셀값을 제거한 오브젝트가 그려지게 된다. 모든 불투명 오브젝트를 그린후에 알파테스트된 오브젝트를 렌더링 하는것이 효율적이기 때문에 이를 별개의 큐로 구분해서(TransparentCutout, 2450) 사용하게 된다.

우선 tag에서 Render type과 queue를 선언해준다.

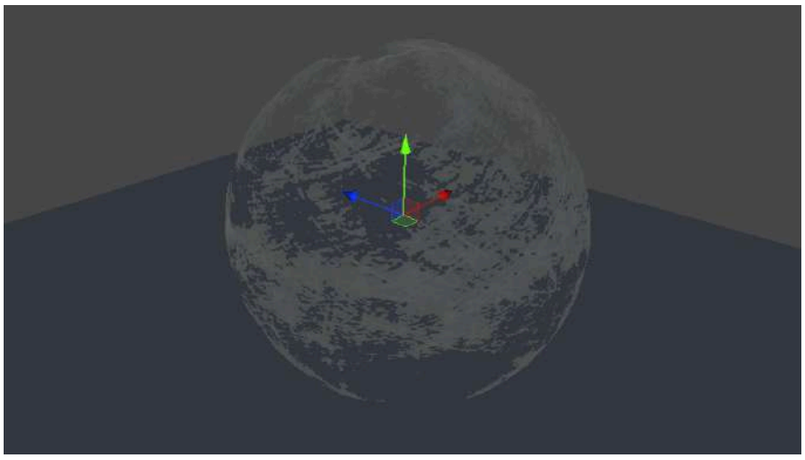
```
Tags {"RenderPipeline" = "UniversalPipeline"  "RenderType" = "TransparentCutout" "Queue" = "AlphaTest"}
```

Pixel shader에서 Clip함수로 해당 픽셀을 연산해준다.

clip(x) : x의 한 원소가 0보다 작으면 현재 픽셀을 버린다. 이 함수는 픽셀셰이더에서만 사용할 수 있다.

이제 픽셀셰이더에서 해당 픽셀을 그릴지 버릴지를 판정하는 코드를 삽입한다.[17]

```
half4 frag(VertexOutput i) : SV_Target
{
float4 color = _MainTex.Sample(sampler_MainTex, i.uv) * _TintColor * _Intensity;
clip(color.a - 0.5);
return color;
}
```



Albedo texture에 사용된 texture에 alpha channel이 있다면 0보다 작은 값이 있다면 픽셀을 그리지 않는 결과를 확인할 수 있다.

전체 셰이더 코드는 아래와 같다. Opaque에서 달라진 부분을 확인해본다.

```
Shader "URPTraining/URPBasic"
{

  Properties {

        _TintColor("Test Color", color) = (1, 1, 1, 1)
        _Intensity("Range Sample", Range(0, 1)) = 0.5
        _MainTex("Main Texture", 2D) = "white" {}

        _Alpha("AlphaCut", Range(0,1)) = 0.5

      }

      SubShader
      {
      Tags
      {
       "RenderPipeline"="UniversalPipeline"
      "RenderType"="TransparentCutout"
      "Queue"="Alphatest"
              }
       Pass
```

[17] 즉 0과 1 만을 가지는 알파로 투명이나 아니냐로만 그려지고 반투명으로 그려지는건 아니다.

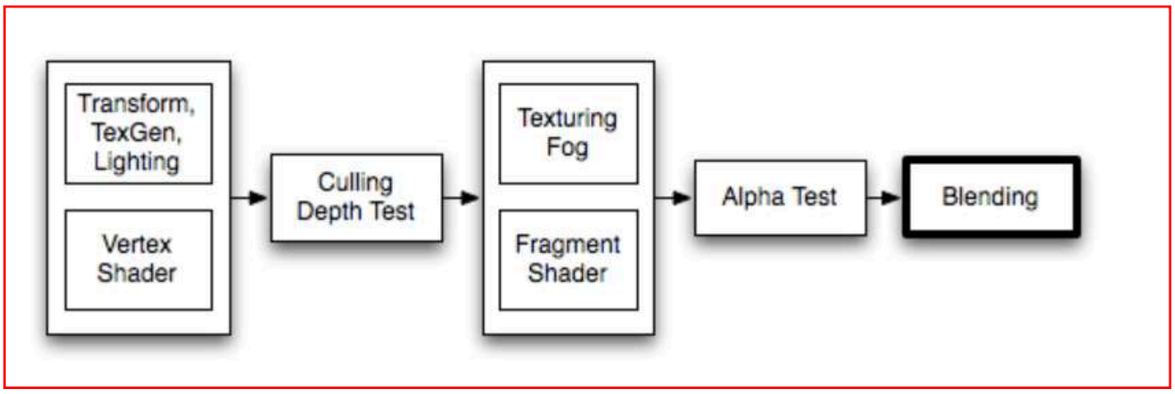
```
        {
         Name "Universal Forward"
        Tags {"LightMode" = "UniversalForward"}

        HLSLPROGRAM
        #pragma prefer_hlslcc gles
        #pragma exclude_renderers d3d11_9x

        #pragma vertex vert
        #pragma fragment frag


        #include "Packages/com.unity.render-pipelines.universal/ShaderLibrary/Lighting.hlsl"

        half4 _TintColor;
        float _Intensity;
        float _Alpha;

        float4 _MainTex_ST;
        Texture2D _MainTex;
        SamplerState sampler_MainTex;

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

        struct VertexOutput
         {
                float4 vertex     : SV_POSITION;
                float2 uv          : TEXCOORD0;
         };

        VertexOutput vert(VertexInput v)
         {
        VertexOutput o;

        o.vertex = TransformObjectToHClip(v.vertex.xyz);
        o.uv = v.uv.xy * _MainTex_ST.xy + _MainTex_ST.zw;

        return o;
         }

        half4 frag(VertexOutput i) : SV_Target
        {
        float4 color = _MainTex.Sample(sampler_MainTex, i.uv);
        color.rgb *= _TintColor * _Intensity;

        clip(color.a - _Alpha);

        return color;
         }
        ENDHLSL
         }
    }

}
```

## Transparent Shader 작성하기

### Transparent 란?

해당 픽셀의 값을 알파값에 따라 덧 그려 반투명으로 그려지게 된다. 자동으로 정렬되는 2500까지의 shader와 달리 여기서부터는 사용자가 렌더큐를 직접 관리해야 한다.

모든 셰이더가 실행되고, 모든 텍스쳐가 적용된 후 픽셀이 화면에 작성되게 되며, 이런 픽셀이 기존 픽셀에 결합되는 방법은 Blend command(operation)으로 제어하게 된다. 블렌딩 과정은 아래와 같이 알파테스트가 수행한 이후 진행되게 된다.




우선 tag에서 Render type과 queue를 선언해준다.

```
Tag {"RenderPipeline"= "UniversalPipeline"  "RenderType"= "Transparent" “Queue”=”Transparent”}
```

이제 Blend Operation을 선언해준다. Blend operation은 해당 픽셀을 그릴때 앞에 그려진 픽셀과 현재 픽셀을 어떻게 계산할 것인지를 선언하는 과정이다.

```
Pass
  {
  Blend SrcFactor DstFactor

  Name "Universal Forward"
  Tags {"LightMode" = "UniversalForward"}
```

Src(Source)는 계산된 컬러를 말하고 Dst(Destination)은 이미 화면에 표시된 컬러를 말한다. 아래는 일반적인 Blend operation 타입이다.[18]

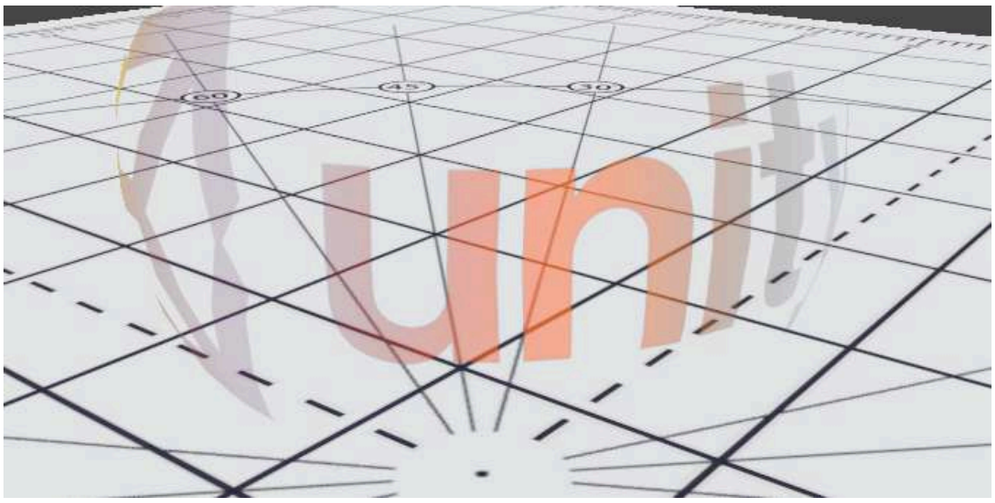
```
Blend SrcAlpha OneMinusSrcAlpha // Traditional transparency
Blend One OneMinusSrcAlpha // Premultiplied transparency
Blend One One // Additive
Blend OneMinusDstColor One // Soft Additive
Blend DstColor Zero // Multiplicative
Blend DstColor SrcColor // 2x Multiplicative
```

이제 픽셀 셰이더에서 아래와 같이 계산해준다.

```
half4 frag(VertexOutput i) : SV_Target
{
        float4 color = _MainTex.Sample(sampler_MainTex, i.uv);
        color.rgb *= _TintColor * _Intensity;
        color.a = color.a * _Alpha;
        return color;
}
```

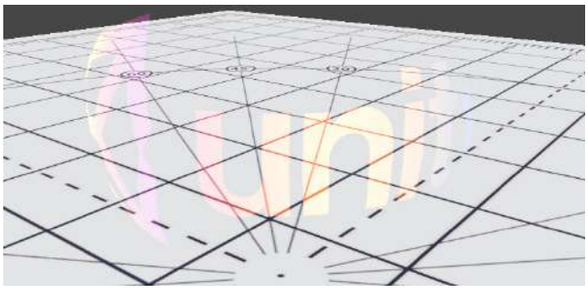
이제 아래와 같이 Blend된 결과를 확인 할 수 있다.

[18] https://docs.unity3d.com/kr/current/Manual/SL-Blend.html Blend operation 인자는 이보다 훨씬 다양하게 응용할 수 있다. 이는 Advanced 항목에서 다루게 된다.



Blend Operation 인자 계산은 아래와 같다.

| | |
|---|---|
| **One** | 1값입니다. 소스 또는 대상 컬러가 완전히 표시되도록 하려면 이 값을 사용합니다. |
| **Zero** | 0값입니다. 소스 또는 대상 값을 제거하려면 이 값을 사용합니다. |
| **SrcColor** | 스테이지 값을 소스 컬러 값으로 곱합니다. |
| **SrcAlpha** | 스테이지 값을 소스 알파 값으로 곱합니다. |
| **DstColor** | 스테이지 값을 프레임 버퍼 소스 컬러 값으로 곱합니다. |
| **DstAlpha** | 스테이지 값을 프레임 버퍼 소스 알파 값으로 곱합니다. |
| **OneMinusSrcColor** | 스테이지 값을 (1 - 소스 컬러)로 곱합니다. |
| **OneMinusSrcAlpha** | 스테이지 값을 (1 - 소스 알파)로 곱합니다. |
| **OneMinusDstColor** | 스테이지 값을 (1 - 대상 컬러)로 곱합니다. |
| **OneMinusDstAlpha** | 스테이지 값을 (1 - 대상 알파)로 곱합니다. |

| | |
|---|---|
|  |  |
| Blend One One<br>(Additive) | Blend DstColor SrcColor<br>(2x multiplicative) |

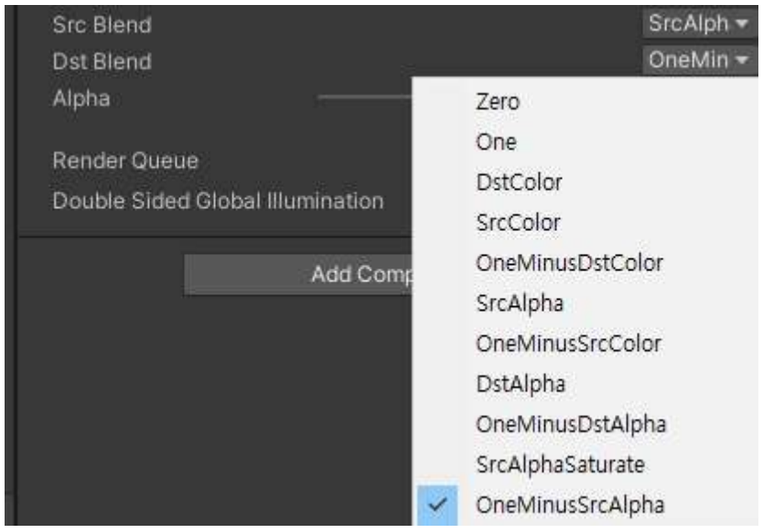
Shader에서 Enum으로 api를 불러와 제어할 수 있다. Properties에 아래와 같이 추가한다.

```
[Enum(UnityEngine.Rendering.BlendMode)] _SrcBlend ("Src Blend", Float) = 1
[Enum(UnityEngine.Rendering.BlendMode)] _DstBlend ("Dst Blend", Float) = 0
```

이제 Pass에 아래와 같이 추가해준다.

```
Pass
  {
   Blend [_SrcBlend] [_DstBlend]

Name "Universal Forward"
Tags {"LightMode" = "UniversalForward"}
```

이제 Material inspector에서 아래와 같이 확인 할 수 있다. 이를 토글로 필요한 방식을 결정할 수 있다.

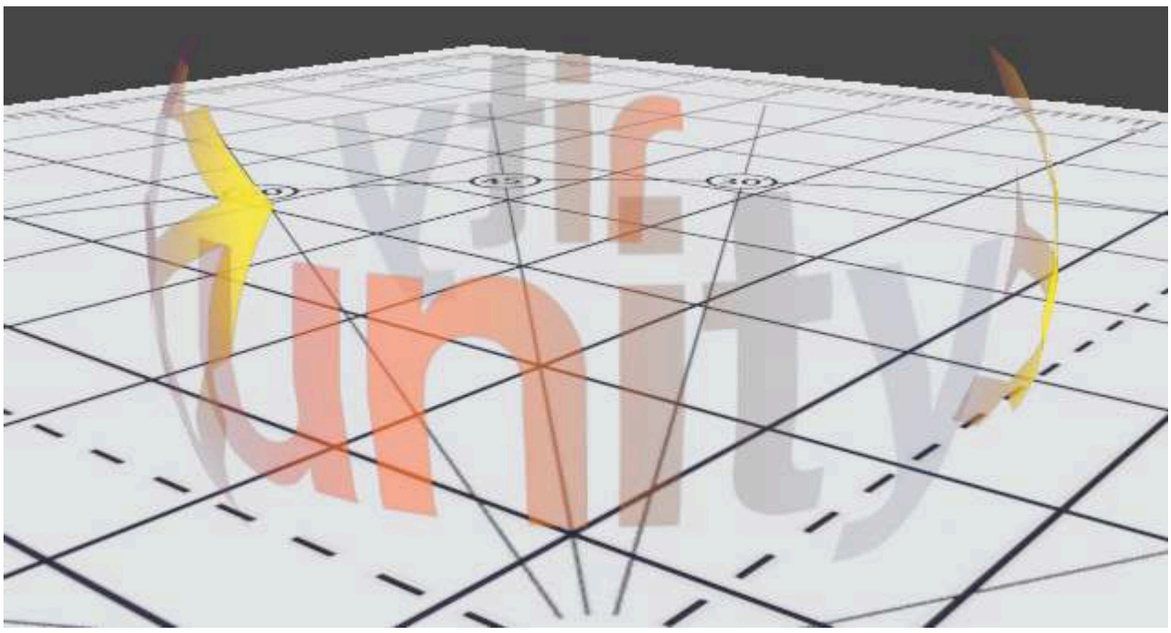



## Culling

컬링은 화면에 그려지는 삼각형의 앞,뒷면을 그릴지를 결정하는 방법이다. 기본값은 면이 보여지는 기본 방향을 그릴수 있으나 이를 사용자가 앞면, 뒷면 혹은 앞뒷면을 모두 다 그릴수 있다.

```
Pass
  {
   Blend SrcFactor DstFactor
Cull Back | Front | off

Name "Universal Forward"
Tags {"LightMode" = "UniversalForward"}
```

- Back : 보는 반대방향을 그리지 않는다(기본값)
- Front : 보는 방향을 그리지 않는다.(보통 오브젝트를 뒤집는데 사용)
- Off : 컬링을 비활성화 한다. 앞뒷면을 모두 그린다[19]



[20]

```
Pass
  {
   Blend SrcFactor DstFactor
Cull Back | Front | off

Name "Universal Forward"
```

[19] 보통 2-side라고 부르는 앞뒷면을 모두 그리는 방식이 이에 해당한다. 주의해야 할 점은 이렇게 그리게 되면 앞뒷면을 모두 그리기 때문에 Triangle counter도 2배로 늘어난다.

[20] https://drive.google.com/file/d/1DVns4E2WK4Q3eydSrgptspCUKcU-Y7Sj/view?usp=sharing 예제 텍스쳐는 여기서 받을수 있다.

```
Tags {"LightMode" = "UniversalForward"}
```

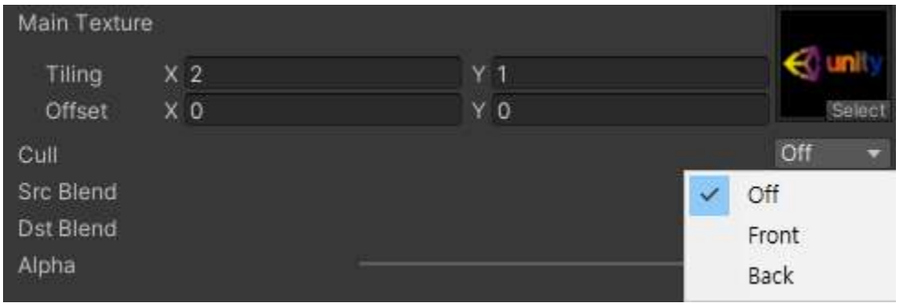
Shader에서 culling mode를 api를 통해 제어할 수 있다. Properties에 아래와 같이 추가한다.

```
[Enum(UnityEngine.Rendering.CullMode)] _Cull ("Cull Mode", Float) = 1
```

이제 Pass에 아래와 같이 추가해준다.

```
Pass
  {
   Blend SrcFactor DstFactor
   Cull [_Cull]

Name "Universal Forward"
Tags {"LightMode" = "UniversalForward"}
```

이제 material inspector에 아래와 같이 Toggle로 culling mode를 변경할 수 있게 된 것을 확인할 수 있다.

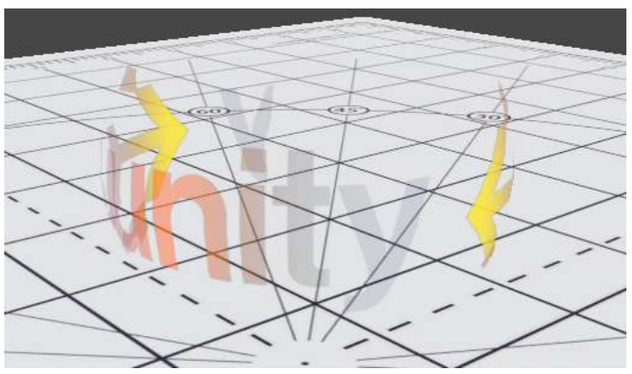
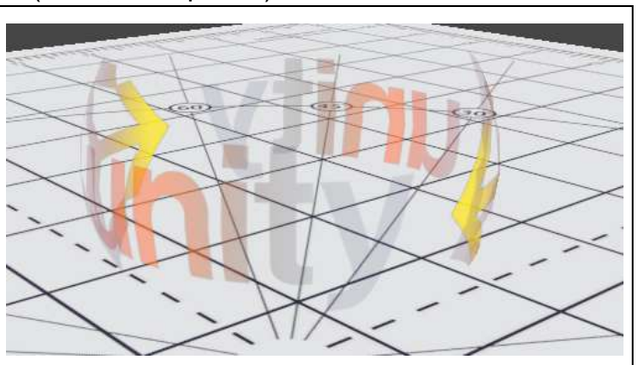



## Zwrite

오브젝트의 픽셀이 depth buffer에 작성되는지 여부를 제어(기본값은 On). 불투명한 오브젝트를 그릴 경우 On 상태로 유지하면 되지만 반투명한 효과를 그릴경우 ZWrite Off로 전환한다.[21]

```
Pass
{
Blend [_SrcBlend] [_DstBlend]
Cull [_Cull]
Zwrite on | off

Name "Universal Forward"
Tags {"LightMode" = "UniversalForward"}
```

아래 이미지는  2side로 Zwrite를 on/off한 결과 비교한 것이다.(2side, Transparent)

|  |  |
|---|---|
| Zwrite On | Zwrite off |

Properties에 Enum을 사용해 토글로 제어가 가능하게도 할 수 있다.

```
[Enum(Off, 0, On, 1)] _ZWrite ("ZWrite", Float) = 0
```

[21] URP에서 Depth texture의 활용은 링크 내용을 참고한다. https://illu.tistory.com/1444

```
Pass
{
Blend [_SrcBlend] [_DstBlend]
Cull [_Cull]
Zwrite [_ZWrite]

Name "Universal Forward"
Tags {"LightMode" = "UniversalForward"}
```


## ZTest

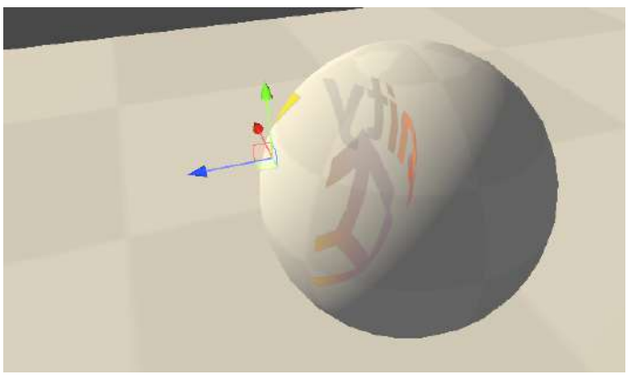
뎁스 테스트(depth Test)를 실행할 방법을 선택한다. 이 항목은 이전 픽셀과 현재 픽셀을 비교해서 어떻게 그릴지를 수행하는 단계.

```
Pass
{
Blend [_SrcBlend] [_DstBlend]
Cull [_Cull]
ZTest Less | Greater | LEqual | GEqual | Equal | NotEqual | Always

Name "Universal Forward"
Tags {"LightMode" = "UniversalForward"}
```

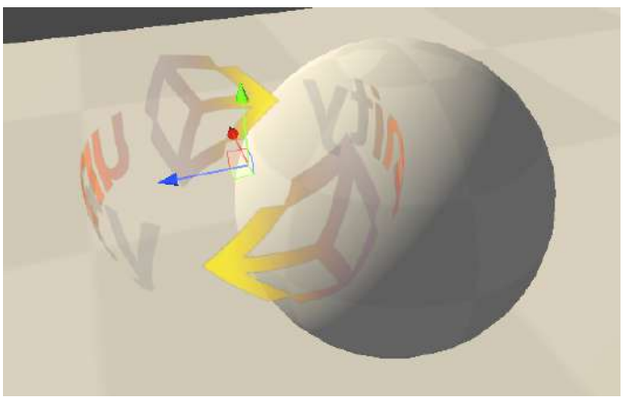
아래는 오브젝트가 겹쳤을때 해당 픽셀을 어떻게 처리하는지를 나타내는 표이다. 기본값은 LEqual(LessEqual), 기존 오브젝트와 같은 거리에 있거나 그 앞에 있는 오브젝트를 드로우하고 그뒤의 오브젝트는 숨김[22].

| Enum | Test | Enum | Test |
|---|---|---|---|
| Never | Always fails. | Always | Always passes |
| Less | < | LessEqual | ≤ |
| Greater | > | GreaterEqual | ≥ |
| Equal | = | NotEqual | ≠ |

아래 이미지는 두가지 옵션을 비교한 결과이다.



Greater



Always

Properties에 Enum을 사용해 토글로 제어가 가능하게도 할 수 있다.

```
[Enum(UnityEngine.Rendering.CompareFunction)] _ZTest("ZTest", Float) = 0
```

[22] 특별하게 겹쳤을때 앞으로 나오는 경우를 제외하고는 보통 건드릴 일이 없는 항목이다. 객체가 겹쳤을때 효과는 보통 stencil buffer를 활용하고 URP에서는 custom pass에서 이를 설정한다

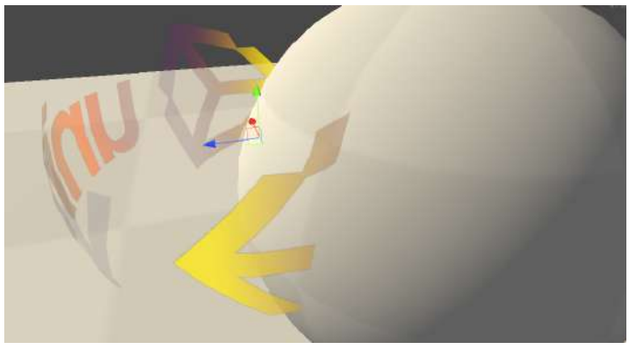
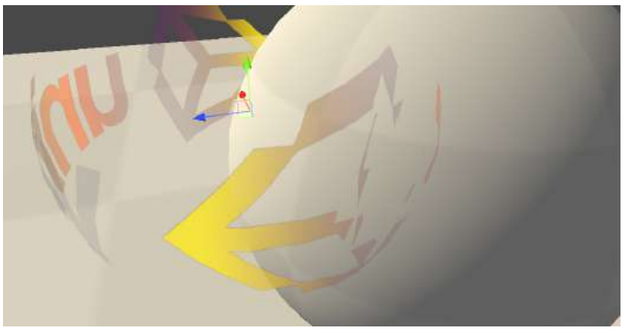
```
Pass
{
 Blend [_SrcBlend] [_DstBlend]
 Cull [_Cull]
 ZWrite [_ZWrite]
 ZTest [_ZTest]

 Name "Universal Forward"
Tags {"LightMode" = "UniversalForward"}
```

## Offset

*Factor* 및 *units* 파라미터 2개를 사용하여 뎁스 오프셋을 지정. *Factor* 는 폴리곤의 X 또는 Y를 기준으로 최대 Z 기울기를 스케일하고 *units* 는 최소 분석 가능 뎁스 버퍼 값을 스케일 하게된다.이를 통해 두개의 오브젝트가 겹칠경우 특정 오브젝트를 앞에 그리게 조절할 수 있다. 예를 들어, Offset 0, -1은 폴리곤의 기울기를 무시하고 폴리곤을 카메라로 더 가까이 당기게 된다.

```
Pass
{
Blend [_SrcBlend] [_DstBlend]
Cull [_Cull]
ZWrite [_ZWrite]
ZTest [_ZTest]
Offset 0, -1

Name "Universal Forward"
Tags {"LightMode" = "UniversalForward"}
```

아래 이미지는 두가지 옵션을 비교한 결과이다.

|  |  |
|---|---|
| Offset 0, 0 | Offset -100, 0 |

Properties에 Enum을 사용해 토글로 제어가 가능하게도 할 수 있다.

```
_Factor("Factor", int) = 0
_Units("Units", int) = 0
```

```
Pass
{
Blend[_SrcBlend][_DstBlend]
Cull[_Cull]
ZWrite[_ZWrite]
ZTest[_ZTest]

Offset [_Factor],[_Units]

Name "Universal Forward"
Tags {"LightMode" = "UniversalForward"}
```

## Alpha To Coverage

포워드 렌더링을 사용하는 멀티샘플 안티앨리어싱(MSAA, QualitySettings 참조)을 사용하는 경우 알파 투 커버리지 기능을 사용해 알파채널 텍스쳐의 AA를 적용할 수 있다. MSAA 가 메시 외곽에만 AA를 적용하고 알파처럼 텍스쳐 안의 이미지에 대한 AA를 적용할수 없기 때문에 이와 같은 방식으로 AA를 적용한다.

```
Pass
{
Blend [_SrcBlend] [_DstBlend]
Cull [_Cull]
ZWrite [_ZWrite]
ZTest [_ZTest]
AlphToMask On

Name "Universal Forward"
Tags {"LightMode" = "UniversalForward"}
```

MSDN에서는 이를 Alpha-to-Coverage로 정의하고 있으며, 아래와 같이 설명하고 있다.[23]

> Alpha-to-coverage는 표면 내의 가장자리를 정의하기 위해 알파투명도를 사용하는 겹치는 다각형이 여러 개있는 빽빽한 잎과 같은 상황에 가장 유용한 멀티 샘플링 기술입니다.
>
> D3D11_BLEND_DESC1 또는 D3D11_BLEND_DESC의 AlphaToCoverageEnable 멤버를 사용하여 런타임이 출력 레지스터 SV_Target0의 .a 구성 요소 (알파)를 픽셀 셰이더에서 n-step 커버리지 마스크 (n-sample  RenderTarget 제공)로 변환할지 여부를 전환 할 수 있습니다. 런타임은 모든 활성 RenderTarget에서 업데이트 할 샘플을 결정하기 위해 프리미티브의 픽셀에 대한 일반적인 샘플 커버리지(샘플 마스크 외에)로 이 마스크의 AND 연산을 수행하게 됩니다.
>
> 픽셀 셰이더가 SV_Coverage를 출력하는 경우 런타임은 alpha-to-coverage를 비활성화합니다.

아래 이미지는 두가지 옵션을 비교한 결과이다.

|  |  |
|---|---|
| AlphaToMask Off | AlphaToMask On |

Properties에 Enum을 사용해 토글로 제어가 가능하게도 할 수 있다.

```
[Enum(Off, 0, On, 1)] _Mask ("Alpha to Coverage", Float) = 0
```

```
Pass
{
Blend [_SrcBlend] [_DstBlend]
Cull [_Cull]
```

[23] https://docs.microsoft.com/en-us/windows/win32/direct3d11/d3d10-graphics-programming-guide-blend-state

```
        ZWrite [_ZWrite]

        Offset [_Factor], [_Unit]

        AlphaToMask [_Mask]

       Name "Universal Forward"
      Tags {"LightMode" = "UniversalForward"}
```

여기까지 진행한 전체코드는 아래와 같다.

```
        Shader "UnityTraining/ShaderTest"
        {
        Properties{

        _TintColor("Test Color", color) = (1, 1, 1, 1)
        _Intensity("Range Sample", Range(0, 1)) = 0.5
        _MainTex("Main Texture", 2D) = "white" {}
        [Enum(UnityEngine.Rendering.BlendMode)] _SrcBlend("Src Blend", Float) = 1
        [Enum(UnityEngine.Rendering.BlendMode)] _DstBlend("Dst Blend", Float) = 0
        [Enum(UnityEngine.Rendering.CullMode)] _Cull("Cull Mode", Float) = 1

        [Enum(Off, 0, On, 1)] _ZWrite("ZWrite", Float) = 0

        [Enum(UnityEngine.Rendering.CompareFunction)] _ZTest("ZTest", Float) = 0

        _Factor("Factor", int) = 0
        _Units("Units", int) = 0

        [Enum(Off, 0, On, 1)] _Mask ("Alpha to Coverage", Float) = 0

        _Alpha("Alpha", Range(0,1)) = 0.5
        }

        SubShader
        {
        Tags
        {
         "RenderPipeline" = "UniversalPipeline" "RenderType" = "Transparent"  "Queue" = "Transparent"}
                    Pass
                    {
                     Blend[_SrcBlend][_DstBlend]
                     Cull[_Cull]
                     ZWrite[_ZWrite]
                     ZTest[_ZTest]

                     Offset [_Factor],[_Units]

                    AlphaToMask [_Mask]

                     Name "Universal Forward"
                    Tags {"LightMode" = "UniversalForward"}

                    HLSLPROGRAM
                    #pragma prefer_hlslcc gles
                    #pragma exclude_renderers d3d11_9x
                    #pragma vertex vert
                    #pragma fragment frag

                    #include "Packages/com.unity.render-pipelines.universal/ShaderLibrary/Lighting.hlsl"
```

```
                half4 _TintColor;
                float _Intensity;
                float _Alpha;

                float4 _MainTex_ST;
                Texture2D _MainTex;
                SamplerState sampler_MainTex;

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

                struct VertexOutput
                 {
                  float4 vertex            : SV_POSITION;
                  float2 uv       : TEXCOORD0;
                 };

                VertexOutput vert(VertexInput v)
                {
                        VertexOutput o;
                        o.vertex = TransformObjectToHClip(v.vertex.xyz);
                        o.uv = v.uv.xy * _MainTex_ST.xy + _MainTex_ST.zw;
                        return o;
                }

                half4 frag(VertexOutput i) : SV_Target
                {
                        float4 color = _MainTex.Sample(sampler_MainTex, i.uv);
                        color.rgb *= _TintColor * _Intensity;
                        color.a = color.a * _Alpha;
                        return color;
                 }
               ENDHLSL
              }
        }
}
```

## UV 정보를 픽셀 셰이더에 출력

기본 UV 값을 픽셀 셰이더에 출력해본다. 앞서 설명한 geometry shader를 기본으로 다시 작성한다.
우선 texture sampling을 추가한다. Properties 창에 아래와 같이 추가한다.

```
Shader "URPTraining/URPBasic"
{
  Properties {
   }

        SubShader
        {
        Tags
        {
      "RenderPipeline"="UniversalPipeline"
      "RenderType"="Opaque"
      "Queue"="Geometry"
        }
        Pass
        {
         Name "Universal Forward"
       Tags {"LightMode" = "UniversalForward"}
```

```
        HLSLPROGRAM
        #pragma prefer_hlslcc gles
        #pragma exclude_renderers d3d11_9x
        #pragma vertex vert
        #pragma fragment frag

        #include "Packages/com.unity.render-pipelines.universal/ShaderLibrary/Lighting.hlsl"

      struct VertexInput
       {
        float4 vertex : POSITION;
       };

      struct VertexOutput
       {
        float4 vertex   : SV_POSITION;
        };

        VertexOutput vert(VertexInput v)
        {
                VertexOutput o;
                o.vertex = TransformObjectToHClip(v.vertex.xyz);
                return o;
        }

        half4 frag(VertexOutput i) : SV_Target
        {
           return float4(1,1,1,1);
        }
         ENDHLSL
         }
    }
}
```

이제 메시의 버텍스 버퍼에서 uv정보를 읽어온다.

```
struct VertexInput
        {
         float4 vertex      : POSITION;
         float2 uv          : TEXCOORD0;
        };
```

이 정보를 보간기를 통해 픽셀 셰이더에 전달한다.

```
 struct VertexOutput
      {
         float4 vertex    : SV_POSITION;
         float2 uv        : TEXCOORD0;
      };
```

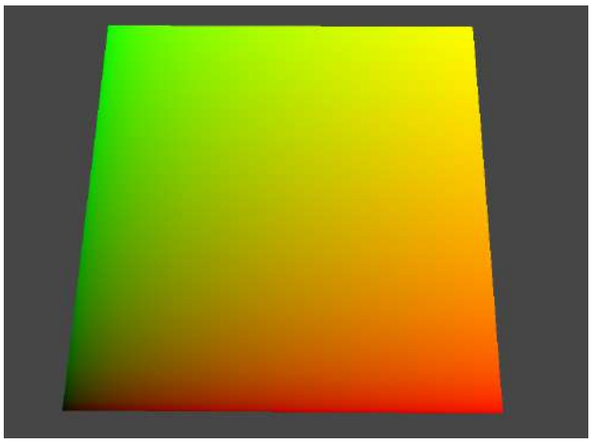
버텍스 셰이더에서 아래와 같이 계산해주고

```
  VertexOutput vert(VertexInput v)
        {
        VertexOutput o;
        o.vertex = TransformObjectToHClip(v.vertex.xyz);
        o.uv = v.uv.xy ;
        return o;
        }
```

픽셀셰이더에서 아래와 같이 계산해 화면에 출력해 준다.

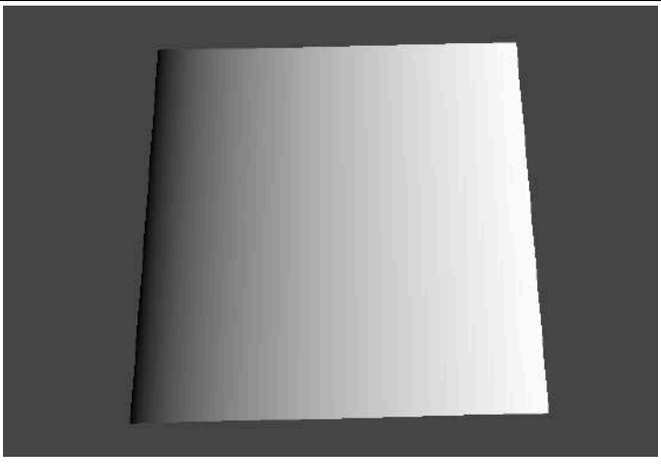
```
  half4 frag(VertexOutput i) : SV_Target
        {
```

```
        float4 color = float4(i.uv.x, i.uv.y, 0, 1);
        return color;
        }
```



이제 이를 U값만을 메시에 출력해본다.

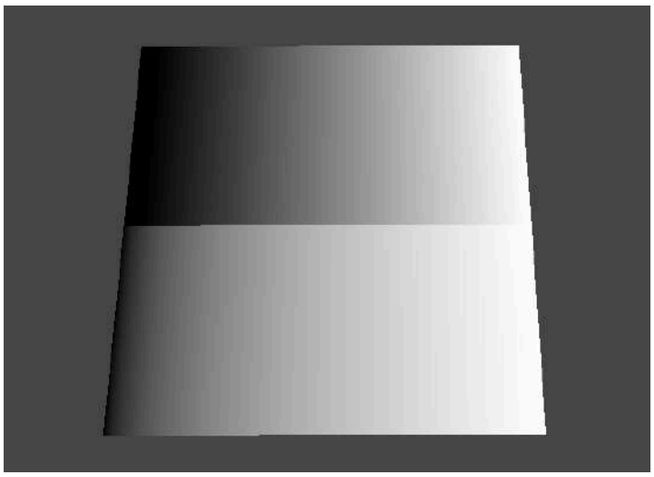
```
 half4 frag(VertexOutput i) : SV_Target
        {
        float4 color = i.uv.x;
        return color;
        }
```



이를 V값을 기준으로 위와 아래를 gamma correction[24]이 적용된 그라데이션으로 출력해본다.

```
 half4 frag(VertexOutput i) : SV_Target
        {
        float4 color = color;
                if(i.uv.y > 0.5)
                {
                color = pow(i.uv.x, 2.2) ;
                }
                else[25]
                {
                color = i.uv.x;
                }
        return color;
        }
```

[24] 자세한 내용은 감마가 어디감마 참조 https://youtu.be/XwIm5V-bnBc
[25] 추가 조건을 사용하려면 else if(  )로 조건을 추가하면 된다.



위 분기문은 아래와 같이 삼항 연산자로 간력화 할 수 있다.

```
half4 frag(VertexOutput i) : SV_Target
        {
         float4 col = i.uv.y > 0.5 ? pow(i.uv.x, 2.2) : i.uv.x;
        //        조건문 ?  참(true)일때 실행값 :  거짓(false)일때 실행값 ;
        return color;
        }
```

참고) C언어 비교 연산자

| | | | | | |
|---|---|---|---|---|---|
| **==** | 같음 | **>** | 큼 | **>=** | 크거나 같음 |
| **!=** | 같지 않음(다름) | **<** | 작음 | **<=** | 작거나 같음 |

참고) C언어 논리 연산자

| | |
|---|---|
| **&&** | AND(논리곱), 양쪽 모두 참일때 참 |
| **\|\|** | OR(논리합), 양쪽 중 한쪽만 참이라도 참 |
| **!** | NOT(논리 부정), 참과 거짓을 뒤집음 |

## Texture Splatting

두장의 텍스쳐를 섞는 방법을 말한다. Texture blending은 여러가지 방법이 있을수 있는데 여기에서 개략적인 방법들을 다루어 본다.

### Linear interpolation

lerp 함수를 사용해서 두장의 텍스쳐를 섞는 방식이다.

lerp(a, b, x) = a + x(b - a) 를 리턴한다. a, b, x는 모두 동일한 타입으로 지정.
이 함수는 x가 0인 경우 a, 1인 경우 b를 돌려주도록 a와 b의 사이를 선형보간 하게 된다. 수식을 풀어서 쓴다면 아래와 같다.

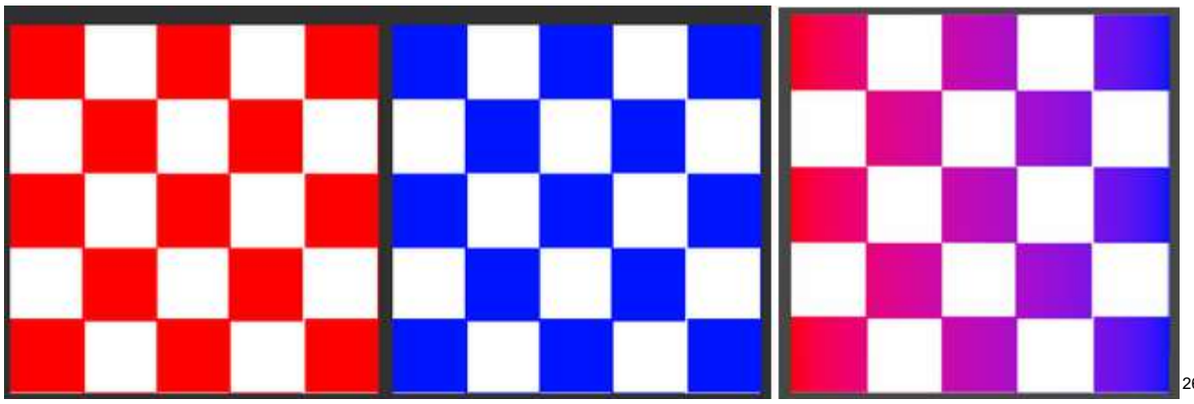
```
float lerp(float a, float b, float x)
{
    return a + x(b - a) ;
}
```

아래는 전체 코드이다.

```
Shader "UnityShader02"
{
   Properties
 {
   _MainTex("RGB 01", 2D) = "white" {}
   _MainTex02("RGB 02", 2D) = "white" {}
 }
```

```
SubShader
        {
        Tags
        {
        "RenderPipeline"="UniversalPipeline"
        "RenderType"="Opaque"
        "Queue"="Geometry"
         }
         Pass
         {
        Name "Universal Forward"
        Tags {"LightMode" = "UniversalForward"}

                 HLSLPROGRAM
                 #pragma prefer_hlslcc gles
                 #pragma exclude_renderers d3d11_9x
                 #pragma vertex vert
                 #pragma fragment frag

                  #include "Packages/com.unity.render-pipelines.universal/ShaderLibrary/Lighting.hlsl"

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

                struct VertexOutput
                {
                float4 vertex     : SV_POSITION;
                 float2 uv       : TEXCOORD0;
                 float2 uv2      : TEXCOORD1;
                };

                 float4 _MainTex_ST;
                 Texture2D _MainTex;
                 SamplerState sampler_MainTex;

                 float4 _MainTex02_ST;
                 Texture2D _MainTex02;

        VertexOutput vert(VertexInput v)
                 {
                 VertexOutput o;
                 o.vertex = TransformObjectToHClip(v.vertex.xyz);
                 o.uv = v.uv.xy * _MainTex_ST.xy + _MainTex_ST.zw;
                 o.uv2 = v.uv.xy * _MainTex02_ST.xy + _MainTex02_ST.zw;
                 return o;
                 }

          half4 frag(VertexOutput i) : SV_Target
                 {
                 float4 tex01 = _MainTex.Sample(sampler_MainTex, i.uv);
                 float4 tex02 = _MainTex02.Sample(sampler_MainTex, i.uv2);
                 float4 color = lerp(tex01, tex02, i.uv.x);
                 return color;
                 }
        ENDHLSL
                 }
        }
}
```

두개의 텍스쳐를 UV x좌표값에 따라 Blending처리한 결과는 아래와 같다.

 [26]

### 마스크 텍스쳐를 사용한 텍스쳐 블렌딩

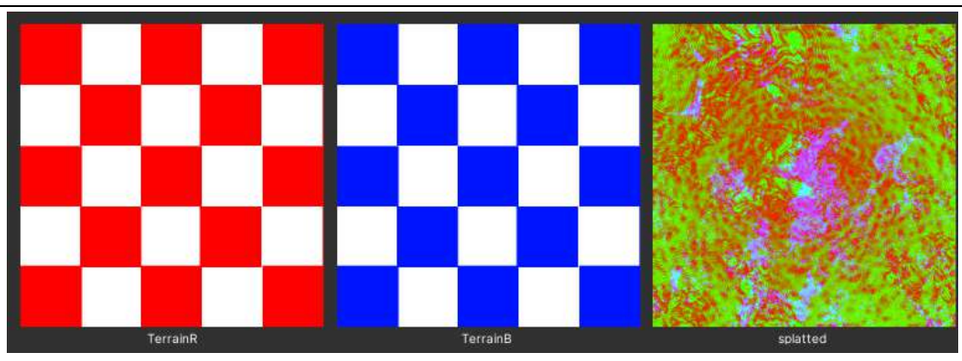
두장의 텍스쳐를 마스크 텍스쳐를 통해 add로 더하는 방식이다. 이 경우에는 RGBA채널을 사용하는 마스크 텍스쳐에서 각 색영역에 대한 부분이 중복되는 부분이 없어야 깔끔하게 나온다.

Properties에 Mask texture 를 추가한다.

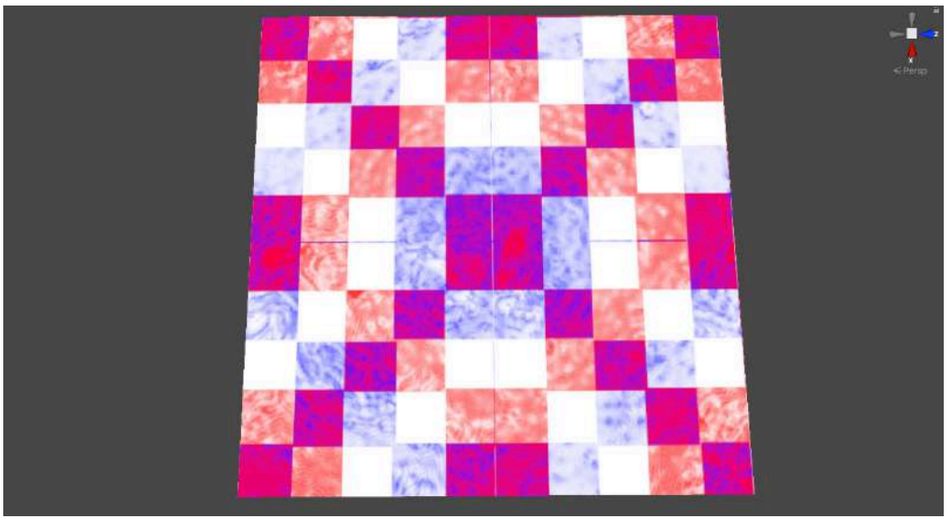
```
Properties
 {
  _MainTex("RGB 01", 2D)     = "white" {}
  _MainTex02("RGB 02", 2D) = "white" {}

  _MaskTex("Mask Texure", 2D) = "white" {}
 }
```

픽셀 셰이더에서 아래와 같이 계산해준다.

```
        half4 frag(VertexOutput i) : SV_Target
            {
                float2 uv = i.uv.xy * _MainTex_ST.xy + _MainTex_ST.zw;
                float2 uv2 = i.uv.xy * _MainTex02_ST.xy + _MainTex02_ST.zw;

                float4 tex01 = _MainTex.Sample(sampler_MainTex, uv);
                float4 tex02 = _MainTex02.Sample(sampler_MainTex, uv2);

                float4 mask = _MaskTex.Sample(sampler_MainTex, uv);

                float4 color = lerp(tex01, tex02, mask.r);
               // float4 color = tex01.mask.r + tex02.mask.g;

                return color;
            }
```




[26] 사용된 리소스는 다음 링크에서 다운로드 가능하다.
https://drive.google.com/drive/folders/1oMfi-dgnPnOZz7rfo6DE7dDXuLLqyWFd?usp=sharing



## UV Scroll

기본 UV 값을 콘트롤 해서 다양한 효과를 셰이더에서 구현해본다.

아래는 기본 Geometry shader이다.

```
Shader "URPTraining/URPBasic"
{

  Properties {
        _TintColor("Test Color", color) = (1, 1, 1, 1)
        _Intensity("Range Sample", Range(0, 1)) = 0.5
        _MainTex("Main Texture", 2D) = "white" {}
   }

        SubShader
        {
        Tags
        {
           "RenderPipeline"="UniversalPipeline"
     "RenderType"="Opaque"
     "Queue"="Geometry"
                }
        Pass
        {
                 Name "Universal Forward"
      Tags {"LightMode" = "UniversalForward"}

        HLSLPROGRAM
        #pragma prefer_hlslcc gles
        #pragma exclude_renderers d3d11_9x

        #pragma vertex vert
        #pragma fragment frag

        #include "Packages/com.unity.render-pipelines.universal/ShaderLibrary/Lighting.hlsl"

                half4 _TintColor;
                float _Intensity;

                float4 _MainTex_ST;
                Texture2D _MainTex;
                SamplerState sampler_MainTex;

        struct VertexInput
         {
                float4 vertex : POSITION;
                float2 uv           : TEXCOORD0;
         };
```

```
        struct VertexOutput
         {
                float4 vertex    : SV_POSITION;
                float2 uv        : TEXCOORD0;
         };

        VertexOutput vert(VertexInput v)
         {
        VertexOutput o;

        o.vertex = TransformObjectToHClip(v.vertex.xyz);
        o.uv = v.uv.xy * _MainTex_ST.xy + _MainTex_ST.zw;
        o.uv.x += _Time.x;
        return o;
         }

        half4 frag(VertexOutput i) : SV_Target
        {
         float4 color = _MainTex.Sample(sampler_MainTex, i.uv);
         color.rgb *= _TintColor * _Intensity;

        return color;
        }
        ENDHLSL
         }
    }

}
```

Mesh uv 방향으로 scroll 되는 shader 제작

Unity는 내부 Built-in value를 가진다 그중 Time함수를 사용해 시간에 따라 UV를 움직여 보는걸 만들어 본다.

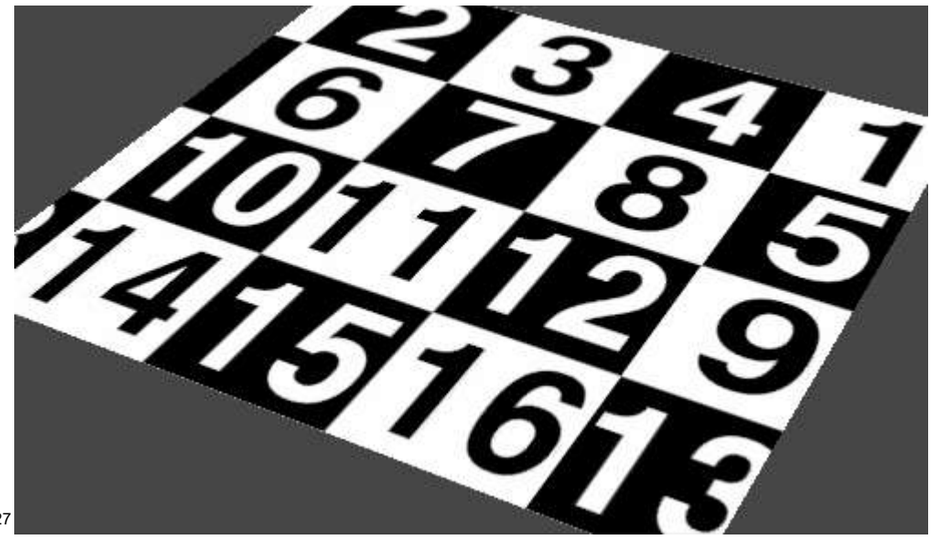
| Name | Type | Value |
|---|---|---|
| _Time | float4 | Time since level load (t/20, t, t*2, t*3), use to animate things inside the shaders. |
| _SinTime | float4 | Sine of time: (t/8, t/4, t/2, t). |
| _CosTime | float4 | Cosine of time: (t/8, t/4, t/2, t). |
| unity_DeltaTime | float4 | Delta time: (dt, 1/dt, smoothDt, 1/smoothDt). |

이제 버텍스 셰이더 계산에서 _Time 함수를 사용해 Offset을 이동시켜 본다.

```
VertexOutput vert(VertexInput v)
        {
        VertexOutput o;
        o.vertex = TransformObjectToHClip(v.vertex.xyz);
        o.uv = v.uv.xy * _MainTex_ST.xy + _MainTex_ST.zw;
        o.uv.x += _Time.x;
        return o;
        }
```

픽셀 셰이더에서도 응용이 가능하다.

```
half4 frag(VertexOutput i) : SV_Target
        {
        i.uv.x += _Time.x;
        float4 color = _MainTex.Sample(sampler_MainTex, i.uv);
        color.rgb *= _TintColor * _Intensity;
        return color;
        }
```



[27]

## Flow map을 활용한 uv animation shader 제작

다른 텍스쳐를 사용해서 UV 움직임에 임의의 값을 추가해서 변화를 주는 방식을 사용해 다양한 움직임을 제어해보자.

우선 texture sampling을 추가한다. Properties 창에 아래와 같이 추가한다.

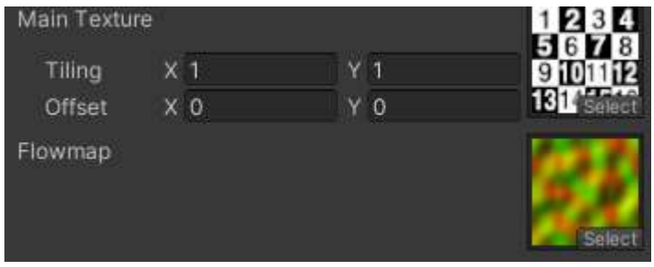
```
Properties {
        _TintColor("Test Color", color) = (1, 1, 1, 1)
        _Intensity("Range Sample", Range(0, 1)) = 0.5
        _MainTex("Main Texture", 2D) = "white" {}
        [NoScaleOffset] _Flowmap("Flowmap", 2D) = "white" {}
    }
```

[NoScaleOffset]은  Properties창에서 tile과 offset 창을 숨기는 명령으로, 앞의 텍스쳐에 사용되는 샘플러값을 같이 사용하기 때문에 인스펙터 창에 노출될 필요가 없어 숨기기 위해 사용한다. 자세한 Properties 확장 명령어는 mid term에서 다루게 된다. 자 이제 아래와 같이 Shader 코드 내부에서 아래와 같이 선언한다.

```
        float4 _MainTex_ST;
        Texture2D _MainTex, _Flowmap;;
        SamplerState sampler_MainTex;
```

픽셀 셰이더에서 이제 추가 텍스처 계산을 넣어준다.

```
        half4 frag(VertexOutput i) : SV_Target
        {
        float4 flow = _Flowmap.Sample(sampler_MainTex, i.uv);
        float4 color = _MainTex.Sample(sampler_MainTex, i.uv);
        color.rgb *= _TintColor * _Intensity;
        return color;
        }
```



[28]

여기까지 진행하면 인스펙터 창에 위 이미지와 같이 텍스처 항목이 추가된다.

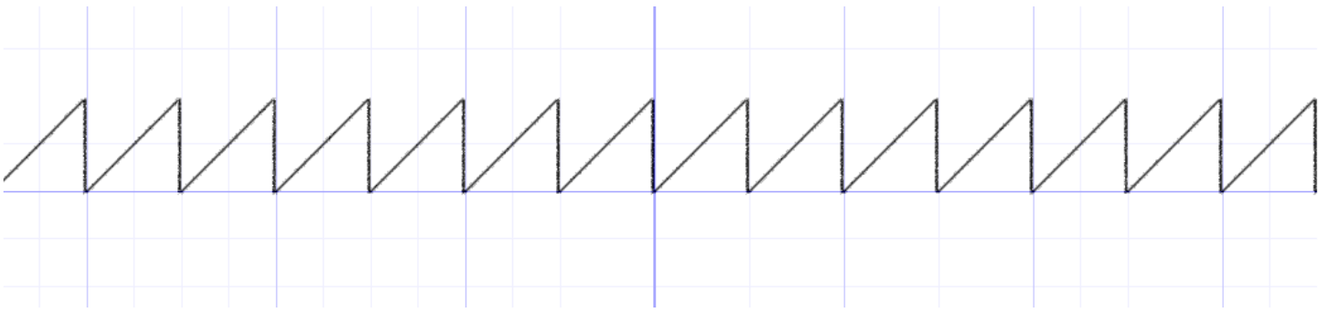
우선 스크롤 되는 타임값은 시간이 지날수록 계속 커지게 된다. 이를 잘라서 사용하기 위해 frac함수를 쓴다.
픽셀 셰이더에서 이제 추가 텍스처 계산을 넣어준다.

```
        half4 frag(VertexOutput i) : SV_Target
        {
        float4 flow = _Flowmap.Sample(sampler_MainTex, i.uv);
        i.uv += frac(_Time.x);
        float4 color = _MainTex.Sample(sampler_MainTex, i.uv);
```

[27] https://drive.google.com/file/d/1si3MTmqUgPXbOIsSLuVmFxnThm1i9UTo/view?usp=sharing
[28] https://drive.google.com/file/d/1l2tmFr7zu_D5_FSGy4DKtZotPBAHyq3c/view?usp=sharing

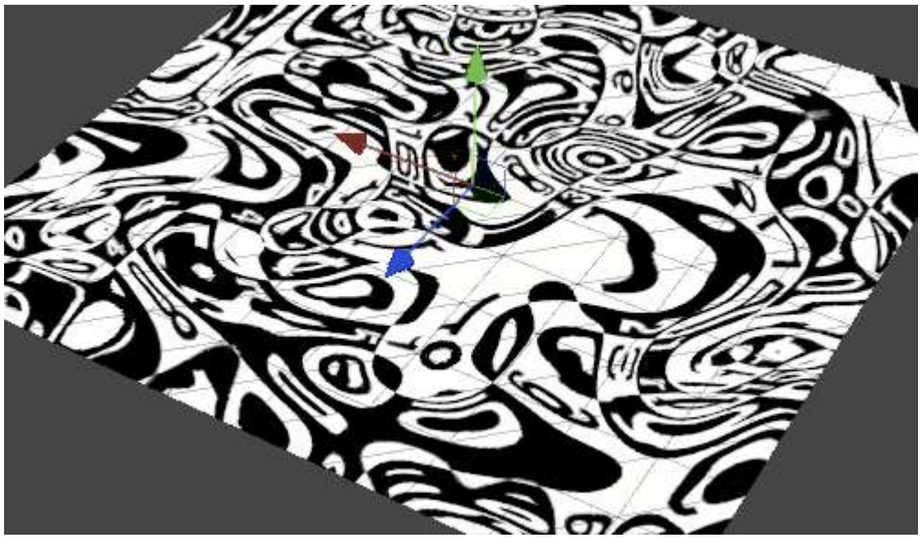
```
color.rgb *= _TintColor * _Intensity;
return color;
}
```

frac(x) : x의 소수점 이하 부분을 리턴한다.



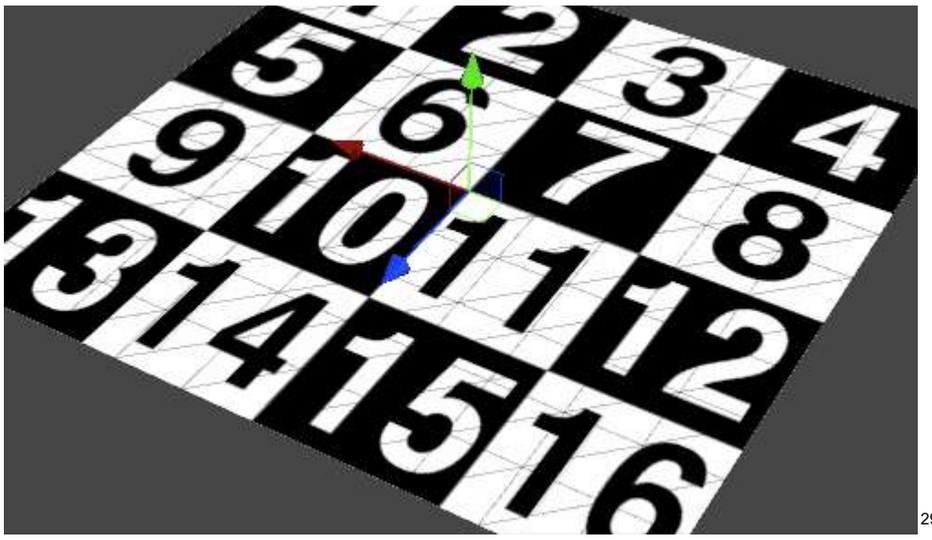
이제 스크롤 되는 uv에 flowmap을 적용해 uv값을 변형해본다.
셀 셰이더에서 이제 추가 텍스처 계산을 넣어준다.

```
half4 frag(VertexOutput i) : SV_Target
{
float4 flow = _Flowmap.Sample(sampler_MainTex, i.uv);
i.uv += frac(_Time.x) + flow.rg;

float4 color = _MainTex.Sample(sampler_MainTex, i.uv);
color.rgb *= _TintColor * _Intensity;
return color;
}
```



Properties 메뉴에 Flow intensity와 Time을 조절하는 기능 추가하기

```
half4 frag(VertexOutput i) : SV_Target
{
        float4 flow = _Flowmap.Sample(sampler_MainTex, i.uv);
        i.uv += frac(_Time.x * _FlowTime) + flow.rg * _FlowIntensity;

        float4 color = _MainTex.Sample(sampler_MainTex, i.uv);

        color.rgb *= _TintColor * _Intensity;
        return color;
}
```



[29]

전체 코드는 아래와 같다.

```
Shader "URPTraining/URPBasic"
{
   Properties {
            _TintColor("Test Color", color) = (1, 1, 1, 1)
            _Intensity("Range Sample", Range(0, 1)) = 0.5
            _MainTex("Main Texture", 2D) = "white" {}
            [NoScaleOffset] _Flowmap("Flowmap", 2D) = "white" {}
            _FlowIntensity("flow Intensity", Range(0,1)) = 1
            _FlowTime("flow time", Range(0,10)) = 1
    }

          SubShader
          {
          Tags
       {
       "RenderPipeline"="UniversalPipeline"
       "RenderType"="Opaque"
       "Queue"="Geometry"
       }
          Pass
          {
          Name "Universal Forward"
          Tags {"LightMode" = "UniversalForward"}

          HLSLPROGRAM
          #pragma prefer_hlslcc gles
          #pragma exclude_renderers d3d11_9x
          #pragma vertex vert
          #pragma fragment frag

         #include "Packages/com.unity.render-pipelines.universal/ShaderLibrary/Lighting.hlsl"

          half4 _TintColor;
          float _Intensity;
          float _FlowIntensity, _FlowTime;

          float4 _MainTex_ST;
          Texture2D _MainTex, _Flowmap;
          SamplerState sampler_MainTex;

       struct VertexInput
        {
          float4 vertex : POSITION;
          float2 uv           : TEXCOORD0;
        };
```

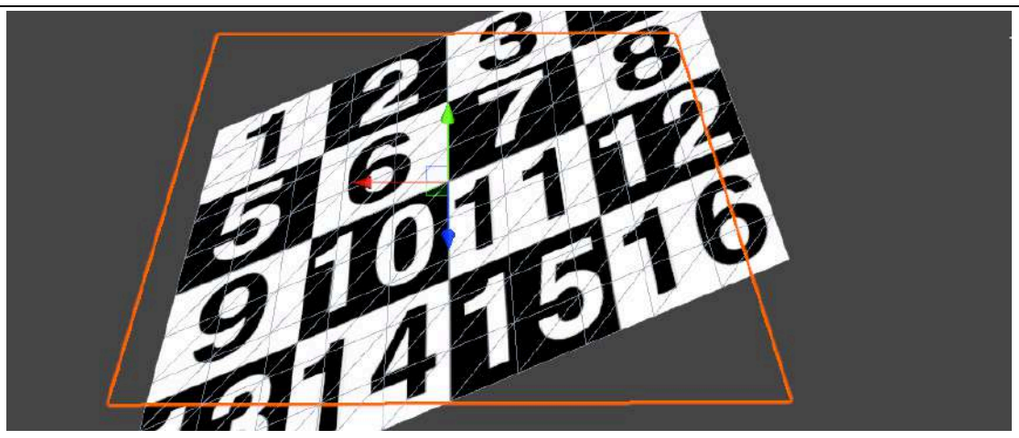
[29] 이 기능을 기반으로 연기, 물등 다양한 이펙트 효과를 셰이더로 표현할 수 있다.

```
        struct VertexOutput
         {
           float4 vertex   : SV_POSITION;
           float2 uv       : TEXCOORD0;
            };

           VertexOutput vert(VertexInput v)
           {
                    VertexOutput o;
                    o.vertex = TransformObjectToHClip(v.vertex.xyz);
                    o.uv = v.uv.xy * _MainTex_ST.xy + _MainTex_ST.zw;
                    return o;
           }

           half4 frag(VertexOutput i) : SV_Target
           {
                    float4 flow = _Flowmap.Sample(sampler_MainTex, i.uv);
                    i.uv += frac(_Time.x * _FlowTime) + flow.rg * _FlowIntensity;

                    float4 color = _MainTex.Sample(sampler_MainTex, i.uv);

                    color.rgb *= _TintColor * _Intensity;
                    return color;
           }
            ENDHLSL
            }
    }
}
```

## Vertex shader를 활용한 mesh animation 구현

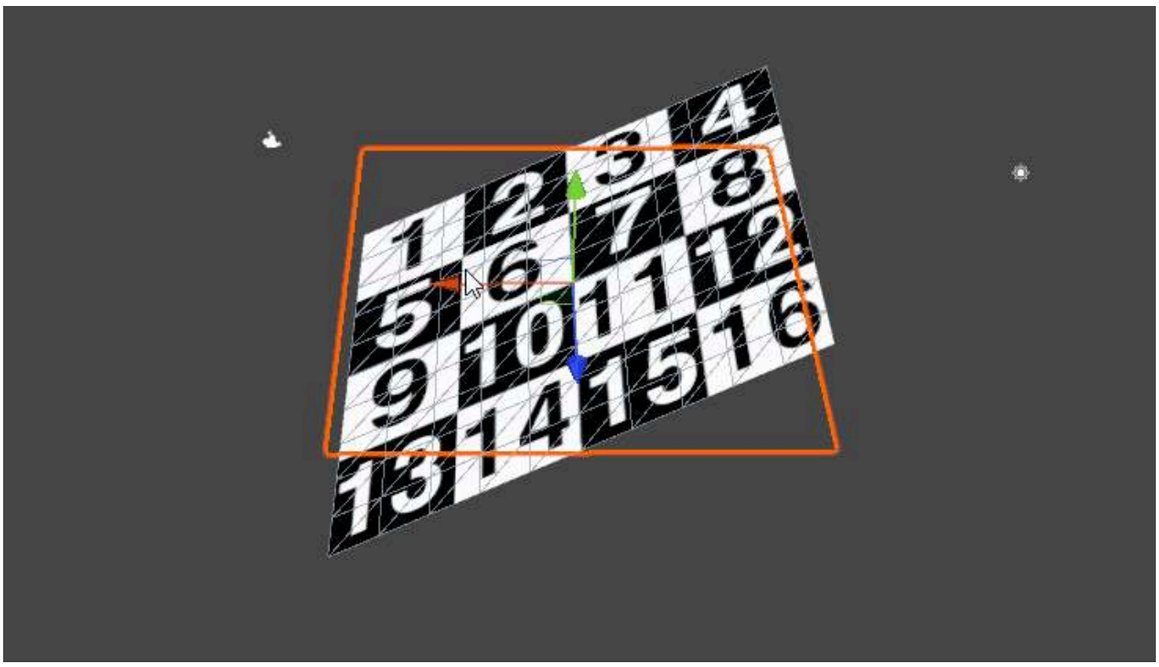
Plane에서 높이값(Y)를 임의의 순서로 움직이게 하는것이 포인트이므로  vertex shader에서 x좌표 값에 따라 vertex shader가 이동하도록 셰이더를 작성한다.
버텍스 셰이더에서 아래와 같이 계산식을 추가해준다.

```
           VertexOutput vert(VertexInput v)
           {
                    VertexOutput o;
                    o.vertex = TransformObjectToHClip(v.vertex.xyz);
                    o.vertex.y += v.vertex.x;
                    o.uv = v.uv.xy * _MainTex_ST.xy + _MainTex_ST.zw;
                    return o;
           }
```

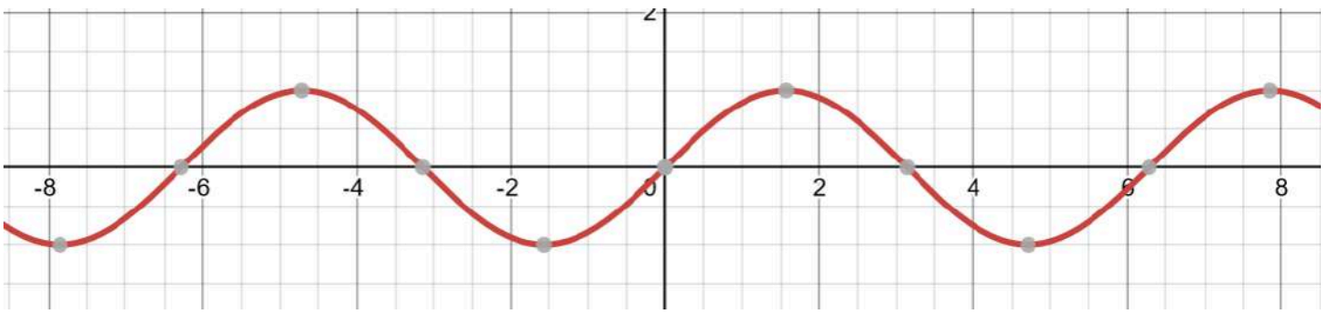



실제 그려지는 메시는 오렌지색 실루엣에 그려져야 하지만 vertex shader에서 mesh의 중심값을 기준으로 x값 만큼 버텍스 좌표를 y만큼 이동했기 때문에 왼쪽은 음수인 아래로 오른쪽은 양수인 위로 그려지고 있다. 이를 월드 공간 기준으로 바꿔서 그려보면 아래와 같이 사용할 수 있다.(공간 변환에서 다룬 내용이다)

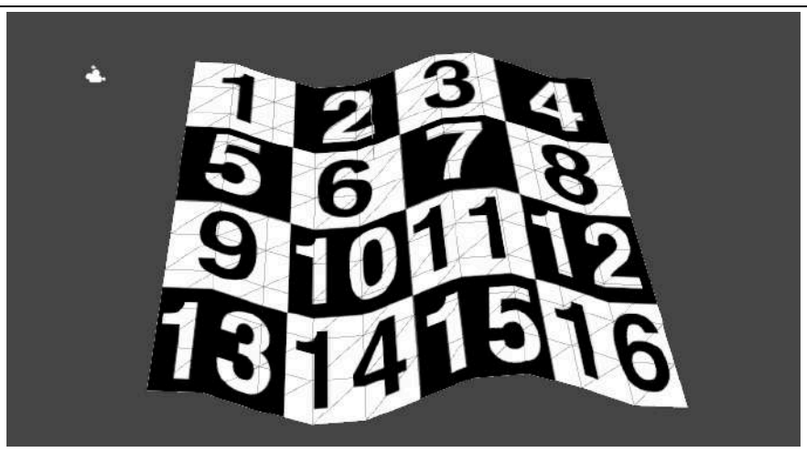
```
VertexOutput vert(VertexInput v)
{
	VertexOutput o;
	o.vertex = TransformObjectToHClip(v.vertex.xyz);
	half3 positionWS = TransformObjectToWorld(v.vertex.xyz);
	o.vertex.y += positionWS;
	o.uv = v.uv.xy * _MainTex_ST.xy + _MainTex_ST.zw;
	return o;
}
```



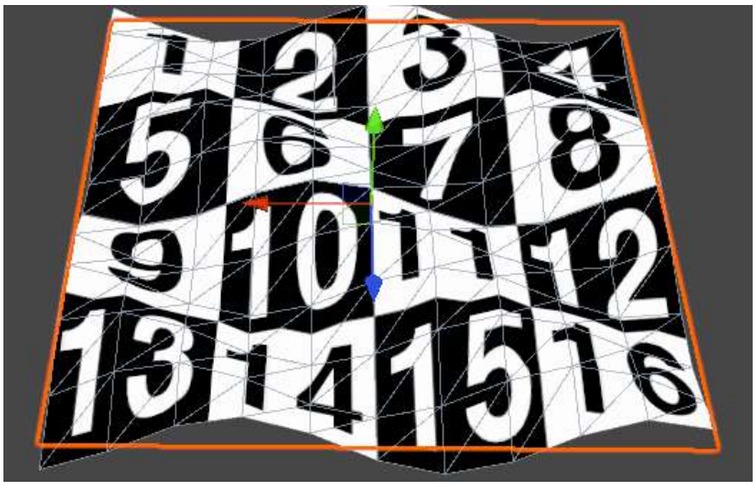
이걸 sin 함수를 이용해 파형 형태로 바꾸면 아래와 같다. Sin 함수는  -1~1사이의 값을 리턴하는 함수이고 함수 그래프는 아래와 같이 그려진다.




```
VertexOutput vert(VertexInput v)
{
	VertexOutput o;
	o.vertex = TransformObjectToHClip(v.vertex.xyz);
	o.vertex.y += sin(v.vertex.x);
	o.uv = v.uv.xy * _MainTex_ST.xy + _MainTex_ST.zw;
	return o;
}
```



여기에 v.vertex.z값도 추가하면 xz축 값에 따라 파형이 변하게는 모양을 확인 할 수 있다. 여기에 위에서 적용한 Time 함수를 더하면 파형이 움직이는 메시를 만들수 있다.



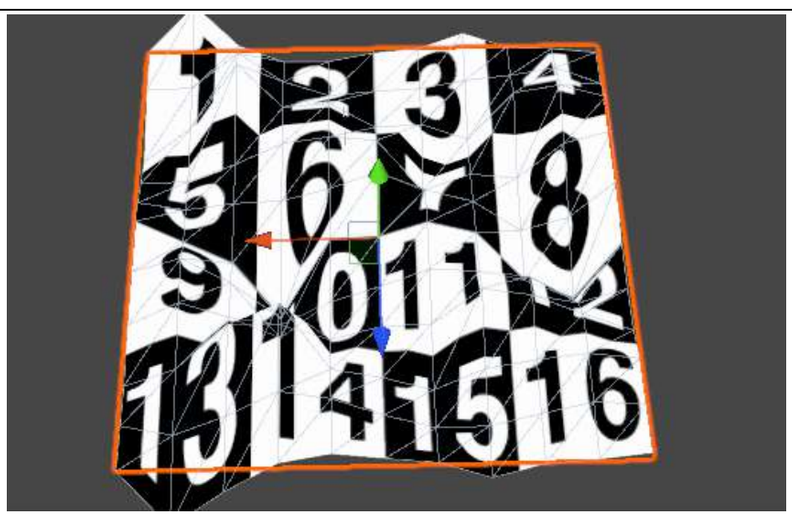
```
VertexOutput vert(VertexInput v)
{
	VertexOutput o;
	o.vertex = TransformObjectToHClip(v.vertex.xyz);
	o.vertex.y += sin(v.vertex.x + v.vertex.z + _Time.y);
	o.uv = v.uv.xy * _MainTex_ST.xy + _MainTex_ST.zw;
	return o;
}
```

파형을 수식으로 구할수도 있지만 noise texture를 사용해 버텍스 스테이지에서 반영하는 방법도 있다. 이경우 tex2Dlod함수를 사용해야 하므로 샘플러를 따로 정의 해줘야 한다. Properties에 정의한 샘플러를 내부에서 아래와 같이 선언한다.

```
float4 _MainTex_ST;
Texture2D _MainTex; //, _Flowmap;
sampler2D _Flowmap;
SamplerState sampler_MainTex;
```

이제 버텍스 셰이더에서 아래와같이 계산해 버텍스 애니메이션에 반영해준다.

```
VertexOutput vert(VertexInput v)
{
VertexOutput o;
o.vertex = TransformObjectToHClip(v.vertex.xyz);
o.uv = v.uv.xy * _MainTex_ST.xy + _MainTex_ST.zw;
half4 noise = tex2Dlod(_Flowmap, float4(o.uv, 0, 0));[30]
o.vertex.y += sin(_Time.y + v.vertex.x + v.vertex.z) * noise.r * _FlowIntensity ;
return o;
}
```



[30] 밉맵으로 2D 텍스처를 샘플링. 밉맵 LOD는 t.w로 지정. Shader model 3.0부터 지원.
https://docs.microsoft.com/en-us/windows/win32/direct3dhlsl/dx-graphics-hlsl-tex2dlod

그외 수식으로도 다양한 방법을 적용할 수 있다.[31]

## 보간기를 정의하고 world position 값을 사용해 오브젝트 컬러를 표현

앞서 정의한 vertex position의 월드 공간에서 구해본다. 우선 vertex shader에서 구한 값을 전달할 변수를 보간기에 선언한다.

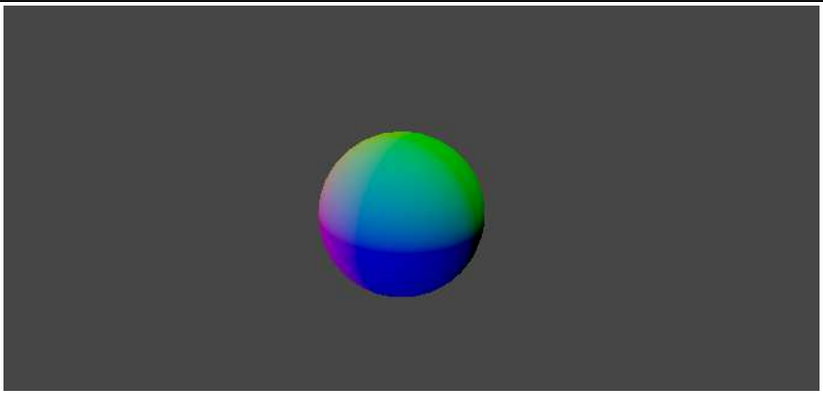
```
struct VertexOutput
{
float4 vertex   : SV_POSITION;
float3 color     : COLOR;
};
```

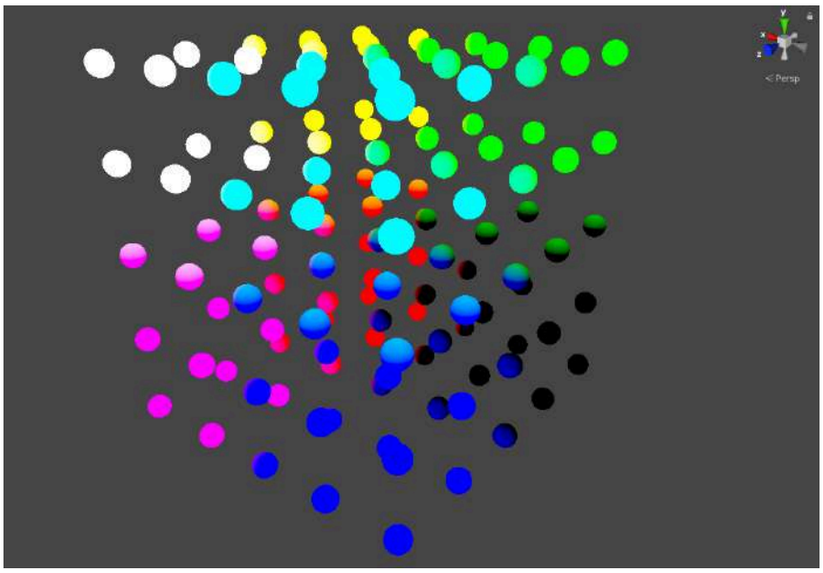
이제 버텍스 셰이더에서 월드 공간으로 변환된 버텍스를 구해준다.

```
VertexOutput vert(VertexInput v)
{
        VertexOutput o;
        o.vertex = TransformObjectToHClip(v.vertex.xyz);
        o.color = TransformObjectToWorld(v.vertex.xyz);
        return o;
}
```

이를 픽셀 셰이더에서 계산해준다.

```
half4 frag(VertexOutput i) : SV_Target
{
        float4 color = float(1, 1, 1, 1);
        color.rgb *= _TintColor * _Intensity * i.color ;
        return color;
}
```

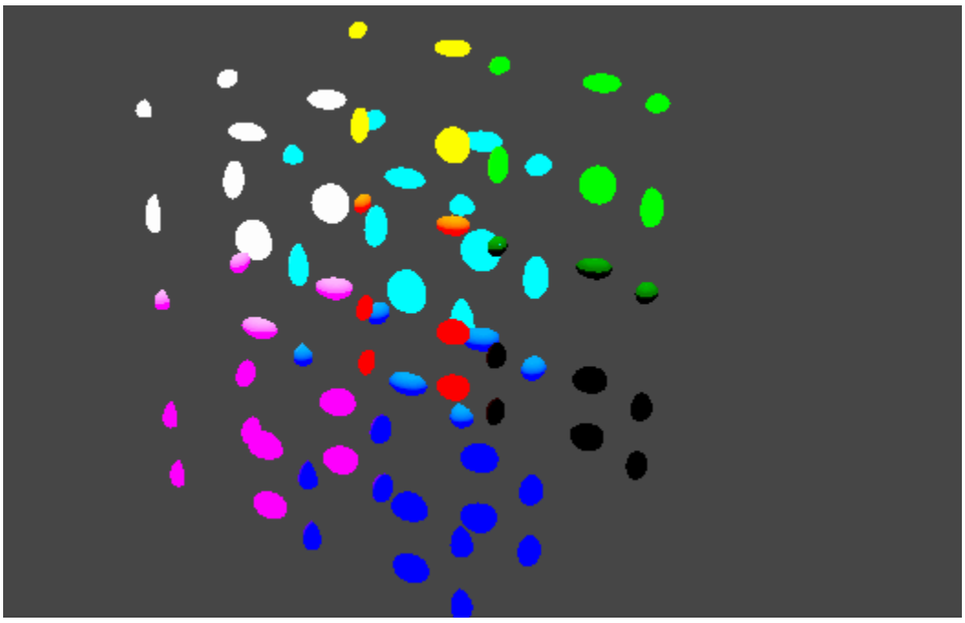




버텍스 셰이더에서 이를 응용해서 변형할 수도 있다.

[31] https://catlikecoding.com/unity/tutorials/flow/waves/

```
VertexOutput vert(VertexInput v)
{
	VertexOutput o;

	float4 positionWS = TransformObjectToHClip(v.vertex.xyz);
	float3 color = TransformObjectToWorld(v.vertex.xyz);

	o.vertex = positionWS + float4(sin(color +_Time.y), 1);
	o.color = color;
	return o;
}
```



## Light Vector를 활용한 라이팅 구현

Lighting.hlsl에는 아래와 같이 라이트 정보가 정의되어 있다. 이를 활용해 간단한 라이팅 연산을 구현해본다.

```
Light GetMainLight()
{
   Light light;
   light.direction = _MainLightPosition.xyz;
   light.distanceAttenuation = unity_LightData.z;
   light.shadowAttenuation = 1.0;
   light.color = _MainLightColor.rgb;
   return light;
}
```

우선 면의 방향을 계산하기 위해 normal값이 필요하다. 버텍스 버퍼에서 노멀값을 읽어온다.

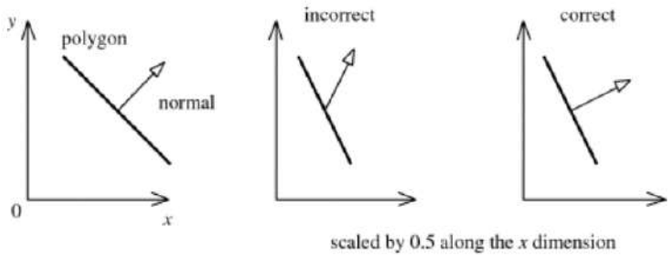
```
struct VertexInput
	{
	 float4 vertex  : POSITION;
	 float3 normal : NORMAL;
	};
```

그리고 픽셀셰이더 전달할 보간기를 선언한다.
우선 면의 방향을 계산하기 위해 normal값이 필요하다. 버텍스 버퍼에서 노멀값을 읽어온다.

```
 struct VertexOutput
    {
	 float4 vertex    : SV_POSITION;
	 float3 normal      : NORMAL;
    };
```

이제 버텍스 셰이더에서 버텍스 위치를 local에서 world로 공간변환을 해주었듯 노멀도 local에서 world로 변환해준다. 다만, 버텍스 위치의 변환과 달리 normal의 변환은 균등 스케일(Uniform scale)의 경우와 아닌 경우를 구분해서 사용하게 된다.[32] 우선 SpaceTransforms.hlsl 의 노멀 변환과 관련된 함수는 아래와 같다

```
float3 TransformObjectToWorldNormal(float3 normalOS, bool doNormalize = true)
{
    #ifdef UNITY_ASSUME_UNIFORM_SCALING
    return TransformObjectToWorldDir(normalOS, doNormalize);

    #else
      // Normal need to be multiply by inverse transpose
      float3 normalWS = mul(normalOS, (float3x3)GetWorldToObjectMatrix());
      if (doNormalize)
      return SafeNormalize(normalWS);

      return normalWS;
    #endif
}
```



scaled by 0.5 along the *x* dimension

이제 버텍스 셰이더에서 아래와 같이 계산해준다.

```
VertexOutput vert(VertexInput v)
    {
        VertexOutput o;
        o.vertex = TransformObjectToHClip(v.vertex.xyz);
        o.normal = TransformObjectToWorldNormal(v.normal);
        o.uv = v.uv.xy * _MainTex_ST.xy + _MainTex_ST.zw;
        return o;
    }
```

이제 픽셀셰이더에서 라이트와 메시의 노멀이 이루는 각도에 따라 라이트가 반영되는 정도를 구한다. 우선은 lighting.hlsl에 정의된 변수를 사용하지 않고 유니티에서 light 벡터와 light 컬러를 직접 불러온다.

```
half4 frag(VertexOutput i) : SV_Target
    {
    //_MainLightPosition은 라이트 벡터를 읽어옵니다.
    float3 light = _MainLightPosition.xyz;

    // 메시의 기본색상은 화이트로 선언합니다.(1,1,1)
    float4 color = float4(1,1,1,1);
    //얻어진 노멀과 라이트를 내적해서 면과 라이트가 같은 방향이면 1, 90도로 어긋나면 0을
    적용합니다.
    color.rgb *= saturate(dot(i.normal, light)) * _MainLightColor.rgb;[33]

    return color;
    }
```

saturate 함수

```
saturate(x) = x가 0 이하이면 0을 1이상이면 1을 리턴한다.(0~1사이의 값으로 clamp)
```

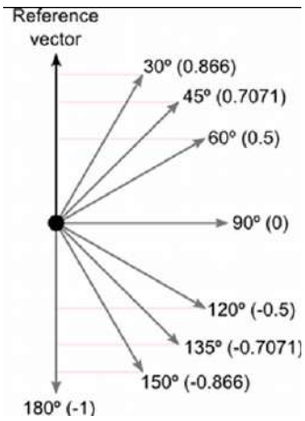
내적(Dot Product) : 두벡터가 얼마나 비슷한지에 대한 값을 구한다.

---

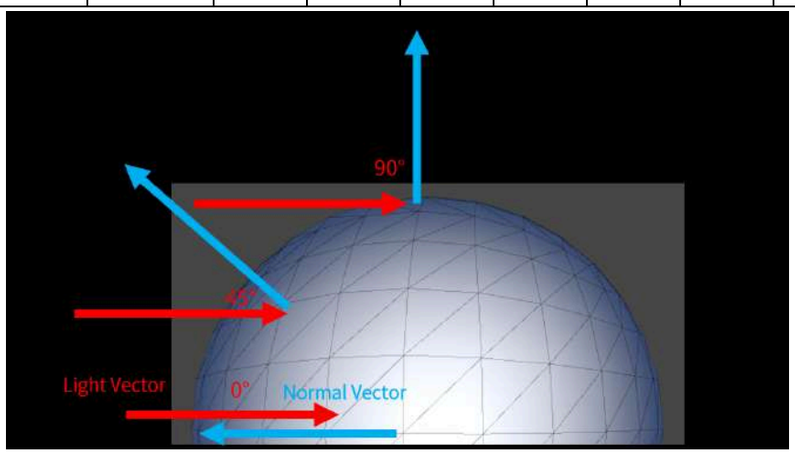
[32] 이런 현상이 발생하는 이유는 스케일이 적용된 경우에 노멀의 방향이 바뀌기 때문. 이와 관련한 내용은 Basic 과정에서는 자세하게 다루지 않고 아래 그림의 0.5 스케일이 적용된 경우 노멀의 방향이 바뀌는 이미지를 참고한다.
o.normal = normalize(mul(v.normal, (float3x3)UNITY_MATRIX_I_M));

[33] 이 방식이 lambert light의 기본이 됩니다.

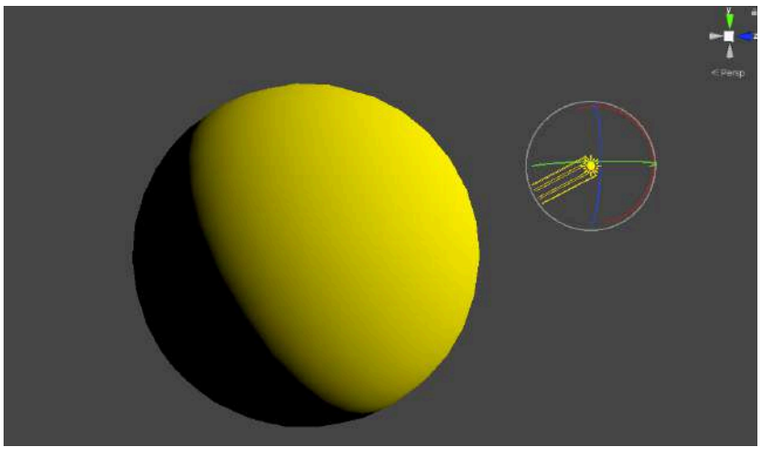
dot(a, b) = $\vec{a} \cdot \vec{b} = |\vec{a}||\vec{b}|\cos\theta$




| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| **cos** | 0 | 45 | 90 | 135 | 180 | 225 | 270 | 315 | 360 |
| **value** | 1 | 0.7 | 0 | -0.7 | -1 | -0.7 | 0 | 0.7 | 1 |




이렇게 계산하면 아래와 같은 결과를 확인할 수 있다.



Lighting.hlsl에 사용된 구조체를 사용하면 아래와 같이 사용하게 된다.[34]

```
half4 frag(VertexOutput i) : SV_Target
        {
        Light lit = GetMainLight();
        float4 color = float4(1,1,1,1);
        color.rgb *= saturate(dot(i.normal, lit.direction)) * lit.color;
        return color;
        }
```

또한 lighting.hlsl에는 lambert lighting이 아래와 같이 선언되어 있다.

```
        half3 LightingLambert(half3 lightColor, half3 lightDir, half3 normal)
        {
                half NdotL = saturate(dot(normal, lightDir));
                return lightColor * NdotL;
        }
```

[34] Self shadow와 shadow casting에 대한건 render pass를 다루는 mid term에서 다루게 된다

```
                half4 frag(VertexOutput i) : SV_Target
                {
                float4 color = float4(1,1,1,1);
                color.rgb *= LightingLambert(_MainLightColor.rgb, _MainLightPosition.xyz, i.normal);
                return color;
                }
```

전체 셰이더 코드는 아래와 같다.

```
Shader "URPTraining/URPBasic"
{
        Properties
         {

         }

         SubShader
              {
             Tags
              {
              "RenderPipeline"="UniversalPipeline"
              "RenderType"="Opaque"
              "Queue"="Geometry"
                          }
                Pass
                {
                 Name "Universal Forward"
                Tags {"LightMode" = "UniversalForward"}

                HLSLPROGRAM
                #pragma prefer_hlslcc gles
                #pragma exclude_renderers d3d11_9x
                #pragma vertex vert
                #pragma fragment frag

                 #include "Packages/com.unity.render-pipelines.universal/ShaderLibrary/Lighting.hlsl"

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

             struct VertexOutput
                {
                 float4 vertex    : SV_POSITION;
                 float3 normal      : NORMAL;
                  };
           VertexOutput vert(VertexInput v)
                 {
                 VertexOutput o;
                 o.vertex = TransformObjectToHClip(v.vertex.xyz);
                 o.normal = TransformObjectToWorldNormal(v.normal);

                 return o;
                 }

            half4 frag(VertexOutput i) : SV_Target
                 {
                 float3 light = _MainLightPosition.xyz;
                 float4 color = float4(1,1,1,1);
                 color.rgb *= saturate(dot(i.normal, light)) * _MainLightColor.rgb;
                 return color;
```

```
            }
        ENDHLSL
            }
        }
    }
```

이를 이제 버텍스 셰이더에서 구해서 적용해본다. 우선 보간기에 아래와 같이 선언해준다.

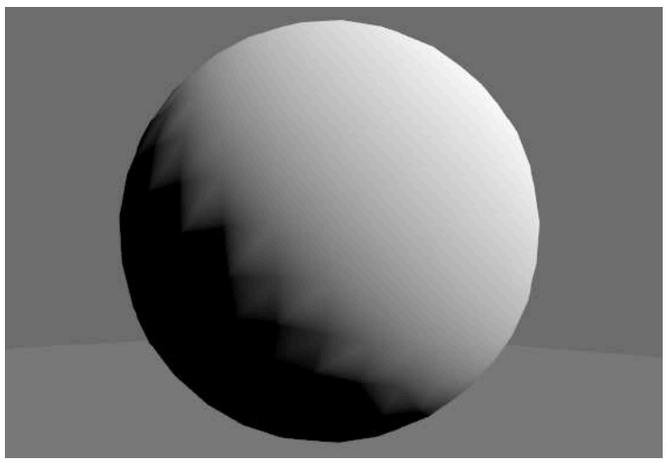
```
        struct VertexOutput
        {
            float4 vertex   : SV_POSITION;
            float3 normal     : NORMAL;
            float3 light : COLOR;
        };
```

이제 픽셀 셰이더에서 계산한것을 버텍스 셰이더로 옮겨서 계산해본다.

```
        VertexOutput vert(VertexInput v)
            {
            VertexOutput o;
            o.vertex = TransformObjectToHClip(v.vertex.xyz);
            o.normal = TransformObjectToWorldNormal(v.normal);

            float3 light = _MainLightPosition.xyz;
            o.light = saturate(dot(o.normal, light)) * _MainLightColor.rgb;

            return o;
            }
```

이를 보간기를 통해 픽셀셰이더에서 계산해준다.

```
        half4 frag(VertexOutput i) : SV_Target
            {
            float4 color = float4(1,1,1,1);
            color.rgb *= i.light;

            return color;
            }
```



35

## Toon lighting 기초

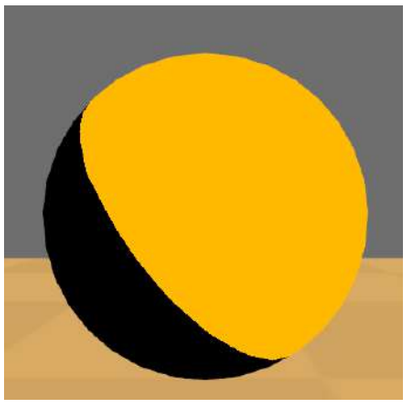
### 삼항연산자를 이용한 기본 라이팅을 툰 스타일로 변형

라이트 벡터를 활용해 카툰 스타일의 라이팅을 만들수 있다. 이에 대한 기초적인 라이팅을 구현해본다.
앞서 배운 삼항 연산자를 이용해서 노멀과 라이트의 내적값(NdotL)이 1 이상일때 라이트 컬러를 적용하는 걸 적용한다.

---

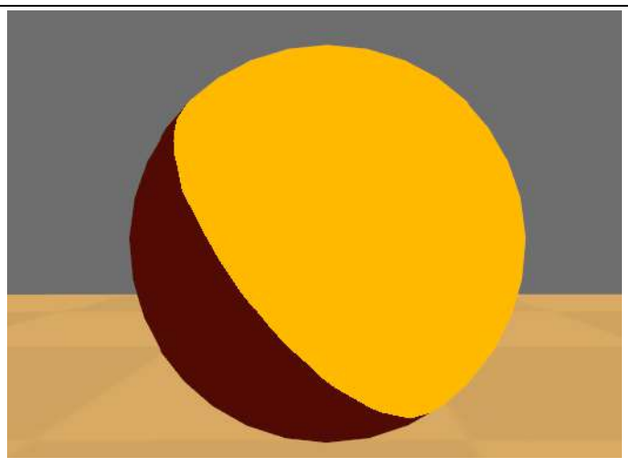
[35] 이를 vertex lighting이라고 한다. 픽셀 수가 늘어나는 요즘 플랫폼에서는 상대적으로 vertex lighting이 연산에 좀 더 가벼운 편이나 버텍스 갯수에 따라 라이팅의 퀄리티가 좌우되므로 주의해서 사용해야 한다.

```
half4 frag(VertexOutput i) : SV_Target
{
	float3 LightColor = _MainLightColor.rgb;
	float3 Light = _MainLightPosition.xyz;

	float NdotL = saturate(dot(Light, i.normal));
	float4 color = float4(1,1,1,1);
	float3 toonlight = NdotL > 0 ? _MainLightColor.rgb : 0;
	color.rgb *= toonlight;
	return color;
}
```



이제 ambient 영역도 원하는 색상을 적용해본다.[36]

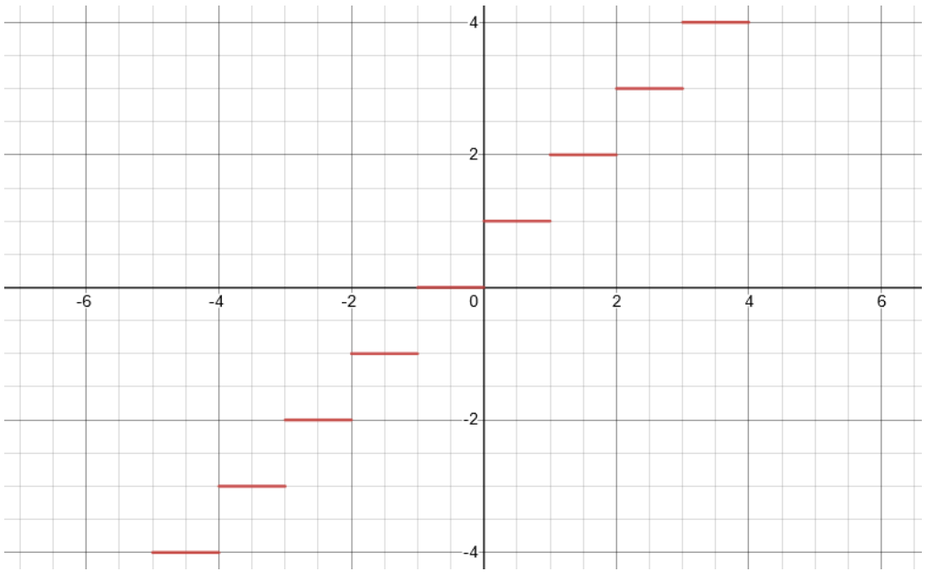
```
half4 frag(VertexOutput i) : SV_Target
{
	float3 LightColor = _MainLightColor.rgb;
	float3 Light = _MainLightPosition.xyz;

	float3 NdotL = saturate(dot(Light, i.normal));
	float4 color = float4(1,1,1,1);
	float3 toonlight = NdotL > 0 ? _MainLightColor.rgb : _Ambientcolor.rgb;
	color.rgb *= toonlight + ambient;
	return color;
}
```



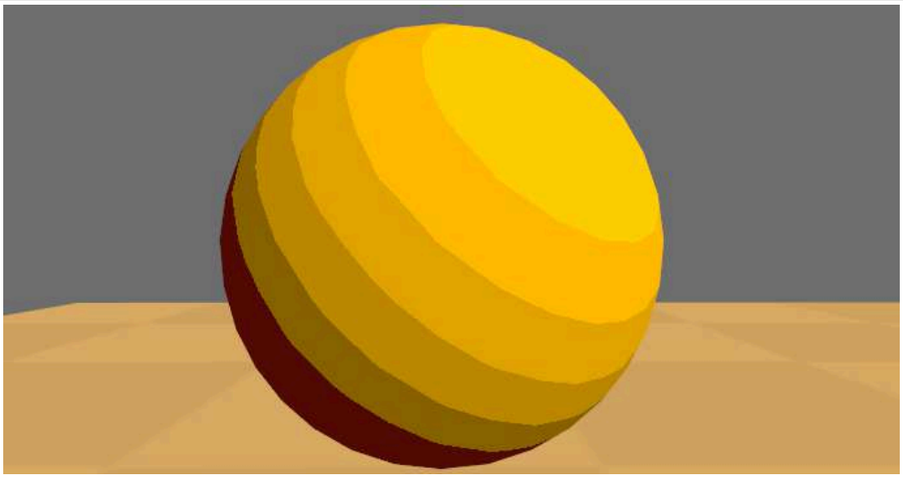
이제 ceil 함수를 사용해서 라이트를 받는 영역을 계단화 해서 표현해본다.

```
ceil(x) = x의 올림한 정수만을 리턴한다.[37]
```

[36] Ambient color를 적용하는 SH함수의 사용은 다음 단락에서 다루게 된다.
[37] ceil(x)은 올림 floor(x)는 내림. round(x)는 반올림이다,




```
half4 frag(VertexOutput i) : SV_Target
{
	float3 LightColor = _MainLightColor.rgb;
	float3 Light = _MainLightPosition.xyz;

	float NdotL = saturate(dot(Light, i.normal));
	float4 color = float4(1,1,1,1);

	float3 toonlight = ceil((NdotL) * _lightwidth) / _lightStep * LightColor;
	float3 ambient = NdotL > 0 ? 0 : _Ambientcolor.rgb;

	color.rgb *= toonlight + ambient ;

	return color;
}
```



ToonRamp Texture를 활용한 Toon shading

앞에서 계산한 라이팅과 노멀의 내적값을 uv 좌표로 활용해 Ramp texture로 사용해본다. 우선 Properties와 내부 변수 선언에 Ramp texture를 선언해주고(**_RampTex**) 픽셀 셰이더에서 아래와 같이 계산해준다

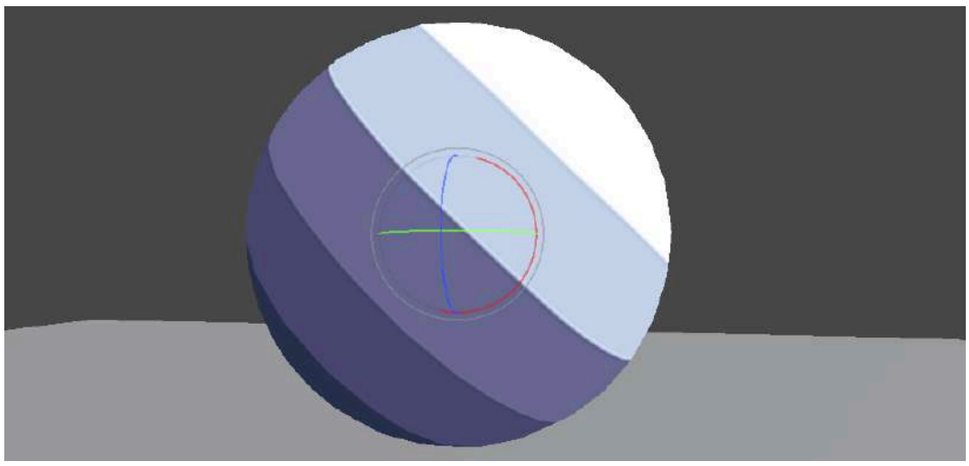
```
half4 frag(VertexOutput i) : SV_Target
{
	float3 Light = _MainLightPosition.xyz;

	float NdotL = dot(Light, i.normal);
	float halfNdotL = NdotL * 0.5 + 0.5;

	float4 color = _MainTex.Sample(sampler_MainTex, i.uv);
	float3 ambient = SampleSH(i.normal);
	float3 ramp = _RampTex.Sample(sampler_MainTex, float2(halfNdotL, 0));
```

```
            color.rgb = color.rgb * ramp + ambient;[38]

            return color;
        }
```

[39]

toonRamp texture(512x32)

라이트벡터와 노멀벡터를 내적한 값은 -1 ~ 1을 가지게 되는데 이를 -0.5 ~ 0.5로 반으로 줄인다음 0.5를 더해 0~1 값으로 변환한다. 이를 Ramp texture의 UV좌표의 U값으로 사용하게 된다.



이방식은 Ramp texture에 따라 다양한 toon lighting을 표현할 수 있다는 장점이 있으나 라이트의 방향만을 사용하기 때문에 scene에 따라 라이트의 느낌을 바꾸기 힘들다.

## Ambient color 및 light probe color의 적용

Ambient color 및 light probe 적용은 구면조화함수를 사용해 적용하게 된다(spherical harmonics)[40] 구면조화 함수의 적용은 주석의 블로그를 참고하고 본 과정에서는 어떻게 사용하지에 대해서만 살펴보도록 한다.

URP의  구면조화 함수는 Lighting.hlsl에 아래와 같이 정의되어있다.

```
        // Samples SH L0, L1 and L2 terms
        half3 SampleSH(half3 normalWS)
        {
            // LPPV is not supported in Ligthweight Pipeline
            real4 SHCoefficients[7];
            SHCoefficients[0] = unity_SHAr;
            SHCoefficients[1] = unity_SHAg;
            SHCoefficients[2] = unity_SHAb;
            SHCoefficients[3] = unity_SHBr;
            SHCoefficients[4] = unity_SHBg;
            SHCoefficients[5] = unity_SHBb;
            SHCoefficients[6] = unity_SHC;

            return max(half3(0, 0, 0), SampleSH9(SHCoefficients, normalWS));
        }

        // SH Vertex Evaluation. Depending on target SH sampling might be done completely per vertex or
        mixed with L2 term per vertex and L0, L1 per pixel. See SampleSHPixel.
        // SH 버텍스 결과값. 대상에 따라 SH 샘플링은 버텍스별로 완전히 수행되거나 버텍스별로 L2 term 및
        픽셀별로 L0, L1과 혼합 될 수 있습니다. SampleSHPixel을 참조하십시오.
        half3 SampleSHVertex(half3 normalWS)
        {
        #if defined(EVALUATE_SH_VERTEX)
            return SampleSH(normalWS);
        #elif defined(EVALUATE_SH_MIXED)
```

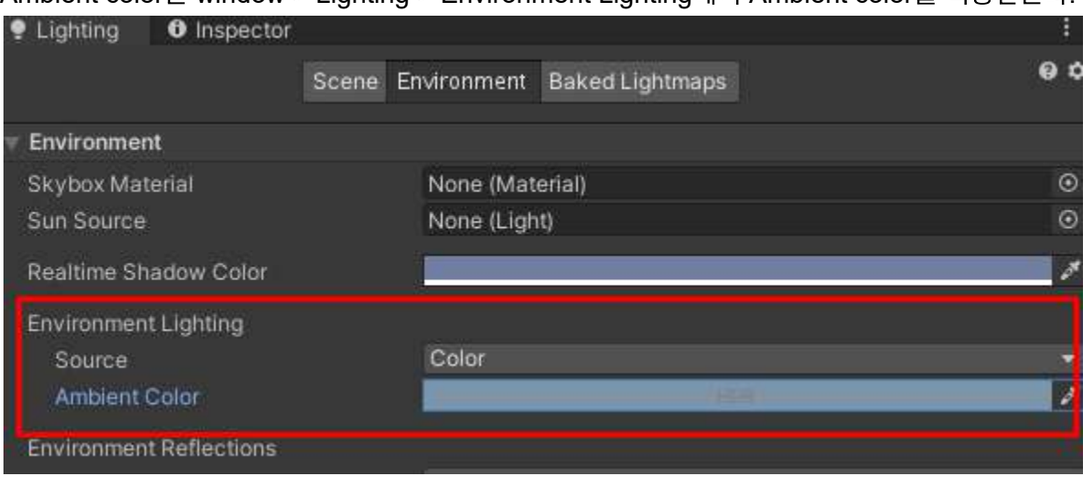
[38] SampleSH에 대한 설명은 다음 chapter에서 하기로 한다
[39] Ramp texture는 https://drive.google.com/drive/folders/1oMfi-dgnPnOZz7rfo6DE7dDXuLLqyWFd?usp=sharing 에서 받을수 있습니다.
[40] 수학없는 SH 설명 https://blog.naver.com/tigerjk0409/221447554201 - Unity Spotlight team lead Jungsuk Ko.

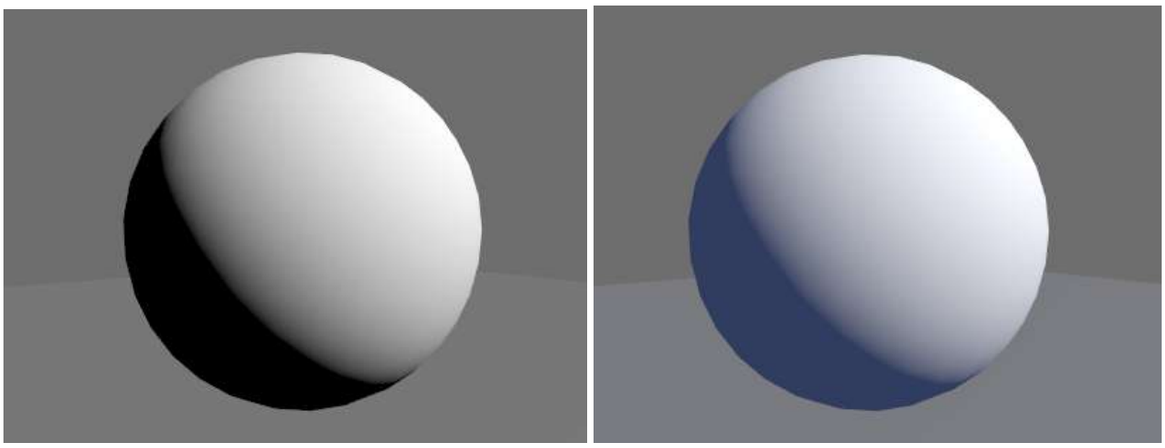
```
    // no max since this is only L2 contribution(L2에만 적용되므로 최대치가 없음)
    return SHEvalLinearL2(normalWS, unity_SHBr, unity_SHBg, unity_SHBb, unity_SHC);
#endif

// Fully per-pixel. Nothing to compute.
    return half3(0.0, 0.0, 0.0);
}

// SH Pixel Evaluation. Depending on target SH sampling might be done mixed or fully in pixel. See SampleSHVertex.
SH 픽셀 평가. 타겟 SH 샘플링에 따라 혼합 또는 전체 픽셀로 수행 될 수 있습니다. SampleSHVertex를 참조하십시오.
half3 SampleSHPixel(half3 L2Term, half3 normalWS)
{
#if defined(EVALUATE_SH_VERTEX)
    return L2Term;
#elif defined(EVALUATE_SH_MIXED)
    half3 L0L1Term = SHEvalLinearL0L1(normalWS, unity_SHAr, unity_SHAg, unity_SHAb);
    half3 res = L2Term + L0L1Term;
#ifdef UNITY_COLORSPACE_GAMMA
    res = LinearToSRGB(res);
#endif
    return max(half3(0, 0, 0), res);
#endif

// Default: Evaluate SH fully per-pixel
    return SampleSH(normalWS);
}
```

기본 라이팅 모델에다 SH를 적용한 결과를 구해본다. 픽셀셰이더에서 아래와 같이 사용한다.

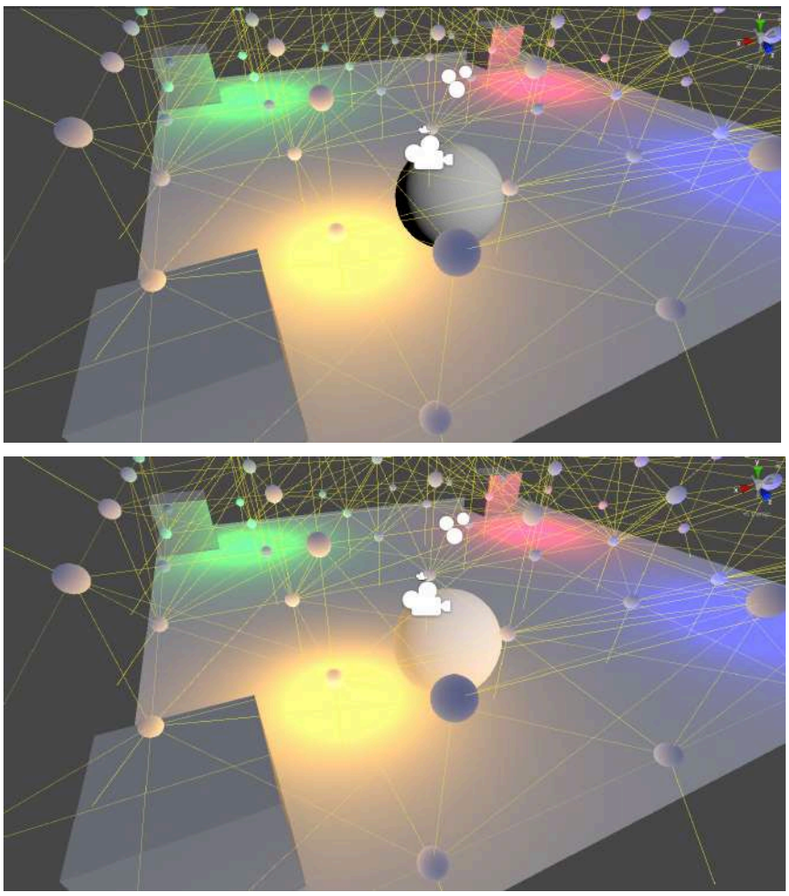
```
half4 frag(VertexOutput i) : SV_Target
        {
        float3 light = _MainLightPosition.xyz;
        float4 color = float4(1,1,1,1);

        half3 ambient = SampleSH(i.normal);

        color.rgb *= saturate(dot(i.normal, light)) * _MainLightColor.rgb + ambient;

        return color;
        }
```

Ambient color는 window > Lighting > Environment Lighting에서 Ambient color를 적용받는다.





Realtime lighting에서 ambient color가 적용된 상태



Light Probe에서 영향을 받는지를 확인해 볼 수 있다.[41]

## 카메라 벡터를 활용한 rim light의 적용

```
Shader "UnityShader02"
        {
         Properties
         {
         _RimPower("Rim Power", Range(0.01, 0.1)) = 0.1
         _RimInten("Rim Intensity", Range(0.01, 100)) = 1
         [HDR]_RimColor("Rim Color", color) = (1,1,1,1)
          }

          SubShader
            {
              Tags
               {
               "RenderPipeline"="UniversalPipeline"
               "RenderType"="Opaque"
               "Queue"="Geometry"
               }
```

[41] URP에서는 아직 LPPV(Light Probe Proxy volume)이 아직 적용되지 않는다.

```
        Pass
        {
         Name "Universal Forward"
        Tags {"LightMode" = "UniversalForward"}

        HLSLPROGRAM
        #pragma prefer_hlslcc gles
        #pragma exclude_renderers d3d11_9x
        #pragma vertex vert
        #pragma fragment frag

         #include "Packages/com.unity.render-pipelines.universal/ShaderLibrary/Lighting.hlsl"

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

        struct VertexOutput
       {
        float4 vertex    : SV_POSITION;
        float3 normal   : NORMAL;
        float3 WorldSpaceViewDirection : TEXCOORD0;
          };

        float _RimPower, _RimInten;
        half4 _RimColor;

        VertexOutput vert(VertexInput v)
        {

        VertexOutput o;
        o.vertex = TransformObjectToHClip(v.vertex.xyz);
        o.normal = TransformObjectToWorldNormal(v.normal);
        // 지정된 객체 공간 정점 위치에서 카메라 방향으로 월드 공간 방향을 계산하고 정규화 함 월드공간 카메라 좌표 - 월드공간버텍스 좌표
        o.WorldSpaceViewDirection = normalize(_WorldSpaceCameraPos.xyz - TransformObjectToWorld(v.vertex.xyz));
        // o.WorldSpaceViewDirection = normalize(_WorldSpaceCameraPos.xyz - mul(GetObjectToWorldMatrix(), float4(v.vertex.xyz, 1.0)).xyz);

        return o;
        }

         half4 frag(VertexOutput i) : SV_Target
        {
        float3 light = _MainLightPosition.xyz;
        float4 color = float4(0.5, 0.5, 0.5, 1);

        half3 ambient = SampleSH(i.normal);
        //월드 카메라 벡터와 노멀을 내적해 방향에 대한 값을 구합니다. 바라보는 방향이 같은 1(밝음) 90도면 0(어두움)이 됩니다.
        half face = saturate(dot(i.WorldSpaceViewDirection, i.normal));

        half3 rim = 1.0 - (pow(face, _RimPower));

       //emissive term
        color.rgb *= saturate(dot(i.normal, light)) * _MainLightColor.rgb + ambient ;

        //emissive term
        color.rgb += rim * _RimInten * _RimColor;
```

```
            return color;
            }
            ENDHLSL
        }
    }
}
```