1. 서론

1.1 소개

본 프로젝트는 언리얼 엔진 5(UE5)의 디퍼드 랜더링(Deferred Rendering) 파이프라인을 활용하여 새로운 캐릭터 셰이딩 모델을 개발하는 것을 목표로 한다. 이를 위해, UE5의 엔진 소스코드에 Custom Shading Model을 추가하고, 디퍼드 랜더링 파이프라인의 G-buffer를 활용하여 캐릭터 카툰 렌더링에 특화된 맞춤형 셰이딩 효과를 구현한다.

이를 통해, 언리얼 엔진의 렌더링 시스템을 보다 깊이 있는 이해를 바탕으로, Custom Shading Model을 활용하여 기존 셰이딩 모델에서 구현이 어려운 다양한 아트 스타일을 표현할 수 있을 것으로 기대된다.

1.2 주요 구현

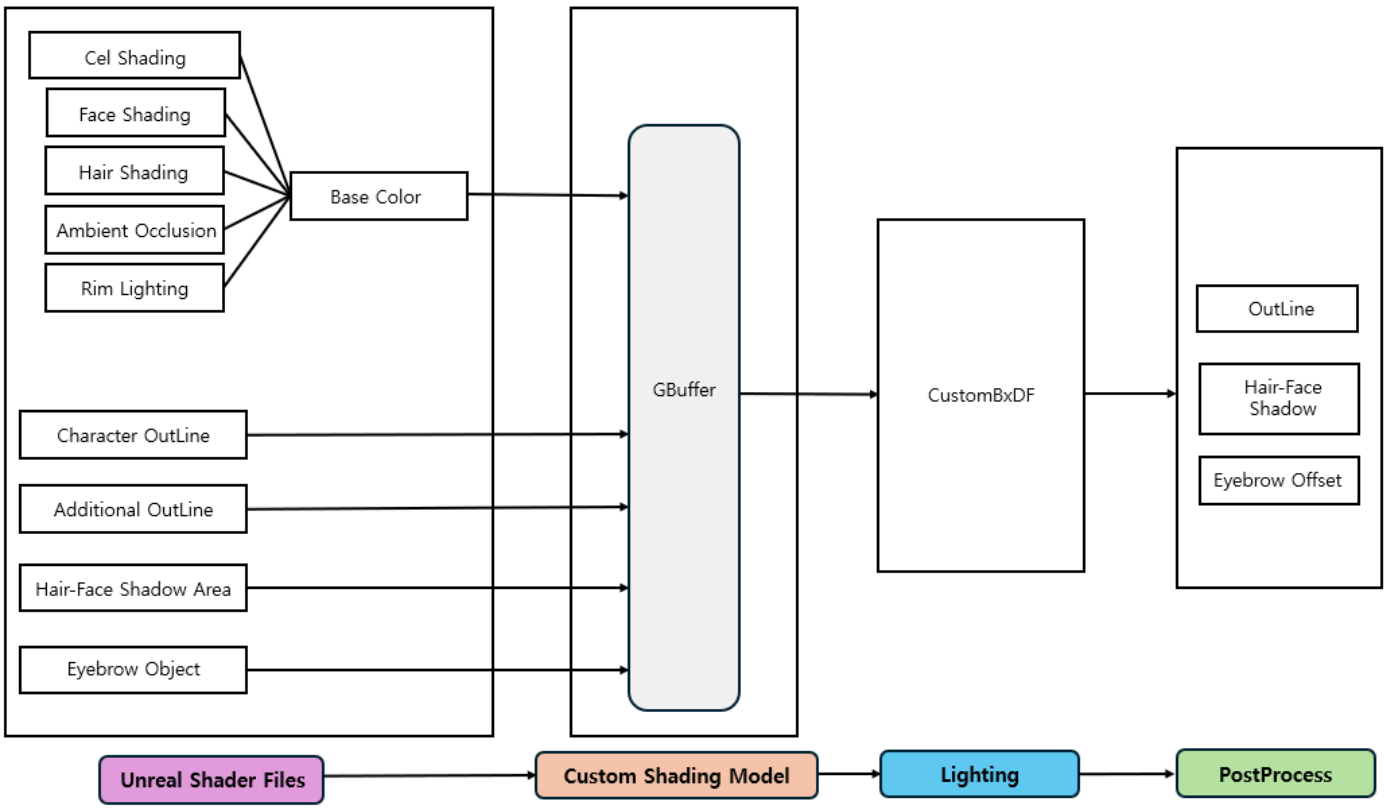
- 단계별 음영 : 캐릭터의 표면에 단계별 음영이 표현되도록 한다.
- 캐릭터 얼굴 셰이딩 : 캐릭터의 얼굴에 자연스러운 구형의 셰이딩이 표현되도록 한다.
- 캐릭터 머리카락 셰이딩 : 캐릭터의 머리카락에 자연스러운 셰이딩이 표현되도록 한다.
- 림라이트 : 캐릭터의 외곽에 림 라이트가 표현되도록 한다.
- 눈썹 Depth 커스텀 : 머리카락과 눈썹의 Depth를 커스텀하여 카툰 스타일의 눈썹을 렌더링한다.
- 캐릭터 외곽선 : 포스트 프로세싱 단계에서 캐릭터의 외곽 및 내부에 외곽선 효과가 표현되도록 한다.
- 머리카락 세부 셰도잉 : 포스트 프로세싱 단계에서 머리카락에 더 자세한 그림자를 만들어낸다.

1.3 개발 환경

- 개발 환경 Unreal Engine 5.1, Window10 OS
- 사용 언어 : C++ , hlsl
- 사용 도구 : Visual Studio 2022, Git

2. 시스템 설계

(시스템 구성도)



1) Custom Shading Model 생성

Custom Shading Model을 생성하기 위해 언리얼 엔진의 엔진 소스코드를 수정한다.

파일명	**EngineTypes.h**
용도	- Custom Shading Model의 모델 이름을 추가한다.

파일명	**MaterialShader.cpp**
용도	- Custom Shading Model의 상세 이름 추가 및 컴파일된 셰이더를 스텟에 업데이트한다.

파일명	**HLSLMaterialTranslator.cpp**
용도	- HLSL 컴파일러 설정, 셰이더코드에서 #if MATERIAL_SHADINGMODEL_CUSTOM_SHADING 으로 구분할 수 있게 한다.

파일명	**Material.cpp**
용도	- SubsurfaceColor, CustomData0 속성이 Custom Shading Model에서 활성화 될 수 있도록 한다.

파일명	**PixelInspectorResult.h**
용도	- Custom Shading Model을 PixelInspector tool로 확인 할 수 있도록 ID를 부여한다.

파일명	**PixelInspectorResult.cpp**
용도	- Custom Shading Model을 PixelInspector tool에서 확인 할 수 있도록 한다.

파일명	**ShadingCommon.ush**
용도	- 셰이더 파일에 Custom Shading Model의 ID를 부여하고, 언리얼 프로젝트 내에서 확인 할 수 있도록 색상을 지정한다.

파일명	**ShadingModel.ush**
용도	- Custom Shading Model의 라이팅 모델을 설정한다.

파일명	**DefferedShadingCommon.ush**
용도	- SubsurfaceColor와 CustomData를 사용한다는 것을 등록한다.

파일명	**ShadingModelsMaterial.ush**
용도	- SubsurfaceColor 데이터를 어떻게 GBuffer에 넣을 것인지 결정한다.

파일명	**ShaderGenerationUtil.cpp**
용도	- 셰이더 컴파일러 환경을 설정한다.

파일명	**MaterialShared.h**
용도	- Custom Shading Model을 SubsurfaceColor를 가지고 있는지 확인하는 함수에 추가한다.

파일명	**ShaderMaterialDerivedHelpers.cpp**
용도	- 커스텀 데이터 사용을 위해 GBuffer 쓰기를 허용한다.

파일명	**BasePassPixelShader.usf**
용도	- 핀에서 받아온 데이터를 적절히 처리할 수 있도록 한다.

2) 머터리얼

머터리얼 에디터에서 커스텀 노드를 사용하여 HLSL 코드를 작성하고, 각 기능들을 usf파일로 만들어서 관리하여 불러온다. 이를 통해 표준 노드 그래프에서 제공되지 않는 셰이딩 효과를 효율적으로 구현한다. usf 파일을 통해 코드의 재사용성을 높이고, 복잡한 셰이더 효과를 더 체계적으로 관리한다.

파일명	**CelShading.usf**
세부내용	목표 : 명암 처리를 통해 단계적 음영을 표현한다. RampTexture를 통해 음영의 정도를 조절할 수 있다. 입력 : ShadowMask 텍스쳐, Ramp 텍스쳐, BaseColor 텍스쳐 단계별 구현 : 1. Half-Lambert 공식을 이용해서 조명강도를 구한다. 조명 방향은 SkyAtmosphereLightDirection을 사용한다. *HalfLambert = NdotL * 0.5 + 0.5* 2. ShadowMask 텍스쳐를 샘플링하여 HalfLambert값과 보간해 Shadow 값을 연산한다. *ShadowValue = lerp(HalfLambert, ShadowMask, ShadowMaskAlpha)* 3. Ramp 텍스쳐를 이용해 단계별 음영을 표현한다. 보간된 ShadowValue를 기반으로 Ramp 텍스처에서 샘플링한다. *RampValue = TextureSample(RampTexture, ShadowValue)* 4. BaseColor 텍스쳐와 결합하여 최종 색상을 표현한다. *FinalColor = BaseColor * RampColor*

파일명	**Ambient Occlusion.usf**
세부내용	목표 : 캐릭터의 음영을 더 깊이감 있게 표현하는 효과를 구현한다. AO 강도를 조절할 수 있다. 입력 : AO 텍스쳐, AOIntensity 단계별 구현 : 1. AOTexture를 샘플링하여 AOValue를 가져온다. *AOValue = TextureSample(AOtexture, UV)* 2. AOIntensity를 AOValue를 보간하는데 사용하여 AO 강도를 조절할 수 있게한다. *AOAdjusted = lerp(1.0, AOValue, AOIntensity)* 3. Half-Lambert 값에 곱하여 AO를 적용한다. *LightWithAO = HalfLambert * AOAdjusted*

파일명	**FaceShading.usf**
세부내용	목표 : 캐릭터 얼굴의 음영을 자연스럽게 표현한다. 빛의 방향에 따라 캐릭터 코의 음영을 입체감 있게 처리한다. 입력 : SDF 텍스쳐(ShadowMask), FrontVector, RightVector 단계별 구현 : 1. SDF 텍스쳐를 ShadowMask로 사용한다. 이 텍스쳐는 그림자 영역을 정의하는 데 사용된다. 2. 캐릭터의 FrontVector와 SkyAtmosphereLightDirection을 내적하여 그림자 적용에 사용할 threshold 값을 구한다. *threshold = dot(FrontVector, SkyAtmosphereLightDirection)* 3. 캐릭터의 RightVector와 SkyAtmosphereLightDirection을 내적하여 그림자 방향을 결정하는 sdfDir을 구한다. *sdfDir = dot(RightVector, SkyAtmosphereLightDirection)* 4. 그림자의 방향 변화를 부드럽게 하기 위해, sdfDirection 을 threshold에 따라 보간한다. *smoothedThreshold = smoothstep(threshold - smoothRange, threshold + smoothRange, sdfDirection)* 5. smoothedThreshold 값을 ShadowMask와 곱하여 최종 sdfRange 값을 계산한다. *sdfRange = ShdowMask * smoothedThreshold* 6. sdfRange와 ShadowMask를 보간하여 최종 그림자 값을 구한다. *ShadowValue = lerp(sdfRange, ShadowMask, ShadowMaskAlpha)*

파일명	**HairShading.usf**
세부내용	목표 : 빛과 카메라 각도에 따라 머리카락이 자연스럽게 표현되게 한다. Mask 맵을 사용하여 머리카락의 하이라이트가 결을 따라 보여지게 한다. 입력 : BaseHairColor, Mask 맵 텍스처, SpecularIntensity, Shininess 단계별 구현 : 1. Blinn-Phong 셰이딩 모델을 사용하여 처리한다. 조명 방향은 SkyAtmosphereLightDirection을 사용한다. 2. Normal 벡터와 Light Direction의 내적을 이용해 확산광을 계산한다. *NdotL = max(0, dot(Normal, LightDirection))* 3. 머리카락의 기본 색상에 확산광을 곱해 diffuseColor를 구한다. *diffuseColor = BaseHairColor * NdotL* 4. LightDirection과 View 벡터의 중간 벡터인 Halfway 벡터를 구하고, Normal 벡터와 내적하여 반사광을 계산한다. *H = normalize(LightDirection + ViewDirection)* *NdotH = max(0, dot(Normal, H))* 5. SpecularIntensity와 shiness값으로 반사광의 강도와 넓이를 결정한다. *specularColor = SpecularIntensity * pow(NdotH, Shininess)* 6. Mask 맵을 샘플링한 maskValue를 specularColor에 곱하여 하이라이트가 자연스럽게 나타나도록 한다. *maskedSpecular = specularColor * maskValue* 7. 하이라이트가 좀 더 선명하게 띠 모양으로 나타나도록 smoothstep 함수를 사용하여 하이라이트의 경계를 세밀하게 제어한다. 8. diffuseColor와 finalSpecular를 더하여 최종 머리카락 셰이딩 색상을 계산한다.

파일명	**Rimlight.usf**
세부내용	목표 : 윤곽선 부근에 빛이 감싸듯이 적용되는 림 라이트를 구현한다. 림의 색상과 림 라이트의 강도를 조절할 수 있다. 입력 : rimAmount, RimLightColor 단계별 구현 : 1. Fresnel 공식을 이용해서 구현한다. 물체의 가장자리에 가까워질수록 내적 값이 작아지게 노말 벡터와 뷰 벡터를 내적한다. *NdotV = dot(Normal, CameraVector)* 2. 림 라이트는 가장자리에서 나타나므로, 1 - 내적 값을 이용한다. *rimAmount = 1.0 - saturate(NdotV)* 3. RimExponent를 사용해 림 라이트의 강도를 제어한다. *rimAmount = pow(rimAmount, RimExponent)* 4. rimAmount가 임계값보다 낮으면 림 라이트를 제거한다. *if(rimAmount < threshold) rimAmount = 0* 5. rim light에 적용할 색상과 rimAmount를 곱해 최종 림 색상을 계산한다. *rimColor = RimLightColor * rimAmount*

3) GBuffer

언리얼 엔진 디퍼드 렌더링에 사용되는 기본 Buffer 중 BaseColor, SubsurfaceColor, Normal, Roughness, Metallic, CustomDepth, SceneDepth에 해당하는 버퍼를 사용하여 계산된 색상값을 넘겨준다.

Buffer	**BaseColor (GBufferA.rgb)**
용도	- Cell Shading, Face Shading, Rim Light, Hair Shading 에서 계산된 값을 받아 Lighting 과정의 BaseColor로 사용한다.

Buffer	**Roughness (GBufferB.r)**
용도	- 캐릭터 모델링의 최외곽 Outline을 받아 포스트 프로세싱 Outline 처리 과정의 Outline Input으로 사용한다.

Buffer	**Normal (GBufferC)**
용도	- Base Model의 normal vector를 받아 Lighting 과정의 Normal로 사용한다.

Buffer	**SubsurfaceColor(GBufferE)**
용도	- Additional Outline에서 계산된 캐릭터에서 강조될 Outline을 받아 포스트 프로세싱 과정 Outline 처리 과정의 Additional Outline Input으로 사용한다.

Buffer	**Custom Depth (StencilBuffer)**
용도	- Eyebrow Object에서 계산된 값을 받아 포스트 프로세싱 눈썹 처리 과정의 CustomDepth Input으로 사용한다.

Buffer	**Scene Depth (DepthBuffer)**
용도	- Cell Shading, Face Shading, Rim Light, Hair Shading 에서 계산된 값을 받아 포스트 프로세싱 과정 눈썹 처리 과정의 Scene Depth Input으로 사용한다.

4) 라이팅

파일명	**ShadingModels.ush**
용도	- 언리얼 엔진의 DefaultLitBxDF() 라이팅 연산을 기반으로 CustomGBuffer에 대응되는 CustomCartoonBxDF() 함수를 만들어 라이팅을 처리하도록 한다.

5) 포스트 프로세싱

파일명	**Eyebrow.ush**
세부내용	목표 : 머리카락에 가려지는 눈썹과 눈썹의 윤곽을 표현한다. 윤곽의 영역, 색을 조절할 수 있다. 입력 : CusomDepth 버퍼, SceneDepth 버퍼, threshold, outlineColor 단계별 구현 : 1. 눈썹 메시에 대해 Render CustomDepth Pass 옵션을 활성화 시켜 눈썹이 따로 Custom Depth 버퍼에 기록되게 한다. 2. 포스트 프로세스 머터리얼에서 Custom Depth 버퍼의 데이터를 가져와 눈썹 오브젝트를 따로 처리할 수 있게한다. 3. 눈썹의 Custom Depth 값과 기존 오브젝트의 Depth 값을 비교하여, 가려진 눈썹 영역을 구한다. *depthDifference = abs(SceneDepth - CustomDepth)* 4. threshold를 기준으로 가려진 눈썹들의 픽셀들을 결정한다. *edgeDetection = step(threshold, depthDifference)* 5. 해당 픽셀들에 눈썹 색상을 적용하고, 아래 Outline.ush 부분에 적힌 외곽선 검출 알고리즘을 적용하여 눈썹의 외곽선을 그려낸다.

파일명	**Outline.ush**
세부내용	목표 : 캐릭터의 외곽선을 그린다. 3종류의 Gbuffer를 이용하여 원하는 부위에 외곽선이 그려지게 한다. 외곽선의 색과 두께를 설정할 수 있다. 입력 : Roughness 버퍼, WorldNormal 버퍼, Subsurface 버퍼, OutlineColor, Thickness 단계별 구현 : 1. 아웃라인 렌더링 : 캐릭터의 최외각 아웃라인을 그린다 - 캐릭터의 외곽정보를 가지고 있는 Roughness 버퍼를 이용한다. - 캐릭터의 외곽선을 감지하기 위해, UV 좌표를 각각 +1, -1로 offset하여 인접한 픽셀들의 Roughness 값을 샘플링한다. 이때 Thickness를 통해 선의 굵기를 조절한다. *OffsetUp = TextureSample(RoughnessTexture, TexCoord + Thickness * (0,1))* *OffsetDown = TextureSample(RoughnessTexture, TexCoord + Thickness * (0,-1))* *OffsetRight = TextureSample(RoughnessTexture, TexCoord + Thickness * (1,0))* *OffsetLeft = TextureSample(RoughnessTexture, TexCoord + Thickness * (-1,0))* - 인접한 픽셀의 Roughness 값과 현재 픽셀의 Roughness 값을 비교하여 차이를 계산한다. *outlineValueVertical = abs(currentValue - OffsetUp) + abs(currentValue - OffsetDown)* *outlineValueHorizontal = abs(currentValue - OffsetRight) + abs(currentValue - OffsetLeft)* - 세로 방향과 가로 방향에서 얻은 차이 값을 합산하여 최종적으로 아웃라인의 범위를 결정한다. *outlineValue = outlineValueVertical + outlineValueHorizontal;* - 감지된 아웃라인 범위에 아웃라인 색상을 적용하여 외곽선을 그린다. *if(outlineValue > threshold) FinalColor = OutlineColor;* 2. 노멀라인 렌더링 : 캐릭터의 내부 외곽선을 그린다 - WorldNormal 버퍼를 이용한다. - 캐릭터의 외곽선을 감지하기 위해, UV 좌표를 각각 +1, -1로 offset하여 인접한 픽셀들의 Normal 값을 샘플링한다. 이때 offset에 Thickness를 곱해 선의 굵기를 조절한다. - 인접한 픽셀의 Normal 값과 현재 픽셀의 Normal값을 비교하여 차이를 계산한다. - 세로 방향과 가로 방향에서 얻은 차이 값을 합산하여 최종적으로 아웃라인의 범위를 결정한다. - 감지된 아웃라인 범위에 아웃라인 색상을 적용하여 외곽선을 그린다.

	3. 컬러라인 렌더링 : 추가적으로 외곽선을 그린다 - 추가적으로 외곽선을 그릴 영역의 정보를 가지고 있는 Subsurface 버퍼를 이용한다. - 캐릭터의 외곽선을 감지하기 위해, UV 좌표를 각각 +1, -1로 offset하여 인접한 픽셀들의 Subsurface 값을 샘플링한다. 이때 offset에 Thickness를 곱해 선의 굵기를 조절한다. - 인접한 픽셀의 Subsurface 값과 현재 픽셀의 Subsurface 값을 비교하여 차이를 계산한다. - 세로 방향과 가로 방향에서 얻은 차이 값을 합산하여 최종적으로 아웃라인의 범위를 결정한다. - 감지된 아웃라인 범위에 아웃라인 색상을 적용하여 외곽선을 그린다.

파일명	**HairFaceShadow.usf**
세부내용	목표 : 캐릭터의 얼굴에 머리카락 그림자가 뚜렷하게 그려지게 한다. 머리카락 그림자가 캐릭터 얼굴에만 그려져야 한다. 그림자 색과 그림자 위치를 조정할 수 있다. 머리카락 모양과 동일한 그림자를 만들어 offset을 주어 적용되게 한다. 입력 : Metalic 버퍼, ShadowOffset, ShadowColor 단계별 구현 : 1. 머리카락 부분을 나타내기 위해 Metalic 버퍼에 값을 넣어 구현한다. 머리카락 값을 1.0으로 설정, 그 외 부분을 0.0으로 설정한다. 2. Metalic buffer 값을 샘플링하여 MetalicValue를 구한다. *MetallicValue = TextureSample(MetalicBuffer, TexCoord)* 3. ShdowOffset을 이용해 머리카락 부분에 offset을 준다. *offsetUV = TexCoord + ShadowOffset* 4. ShadowColor값을 offsetUV에 해당하는 부분에 적용한다. 이 때 머리카락이 있는 영역에 그림자가 드리워지지 않도록, Metalic 값이 1.0인 부분에서는 그림자가 적용되지 않게한다.

3. 일정 관리 계획

일자	6W	7W	8W	9W	10W	11W	12W	13W	14W	15W	16w
설계서 작성	O										
개발환경 구축		O	O								
Custom Shading Model 생성			O	O							
CelShading, Face Shading				O	O						
Hair Shading, Hightligting					O	O					
Lighting 구현						O	O				
Outline						O	O	O			
Hair-Face Shading								O	O		
Eyebrow Offset									O	O	
테스트 및 최종수정										O	O
발표 준비 및 발표										O	O