Texture

▶ 밉맵

해상도가 연속적으로 절반인 순서화된 일련의 텍스쳐들
축소하기 위한 필터링 알고리즘
▷ 4개의 픽셀의 평균을 내서 하나의 작은 픽셀을 생성
▷ 4개의 픽셀을 정렬해서 가운데 색을 픽셀로 생성
레벨0: 밉맵의 오리지널 이미지

▶ 밉맵의 장점

카메라가 멀어진 경우 축소 계산 필요 없음
프레임 레이트 감소
그러나 모델과 텍스쳐의 개수는 항상 같을 수 없기에 샘플링이 필요하긴 함

▶ LOD(레벨 오브 디떼일)

레벨: 카메라와 모델의 거리
카메라까지 거리에 따라 자동적으로 밉맵 레벨 선택
레벨에 따라 폴리곤 개수 결정

▶ 샘플링과 필터링

프리미티브를 구성하는 각 픽셀에 대한 과정
① 각 텍스쳐 좌표에 대한 보간된 텍스쳐 좌표 계산
② 텍스쳐 이미젱 대한 픽셀 좌표 계산
③ 텍스쳐 이미지의 픽셀을 변환(어떤 텍셀을 선택하고 어떻게 색상을 결정?: 필터링)

▶ 블럭 압축

오리지널 텍스쳐를 손실하는 손실 압축 방법
메모리 효율 8배 이득
압축 해제 CPU말고 GPU에서 함(색이 필요할 때)
텍스쳐를 4x4 텍셀 블럭으로 분할, 각 텍셀 2비트로 표현(0~3 인덱스)
00: 최소색, 01: 최대색, 10과 11: 2/3과 1/3
각 블럭은 2개의 색상을 가짐
선형 보간으로 계산
오리지널 이미지를 저장하기 위한 메모리: 256바이트
각 색을 표현하기 위해서는 4바이트 필요
1개의 블럭: 4 + 16 * 2 = 36바이트

▶ DDS WIC 차이점

DDS는 밉맵이 존재
DDS는 배열이 필요한 경우에만 사용 가능

▶ 서브 리소스

텍스쳐 리소스의 부분 집합(배열)
구분하기 위해 인덱스를 사용
서브리소스 인덱스 = 밉슬라이스 + (배열슬라이스 * 밉레벨)

▶ 텍스쳐를 여러 개 사용하기 위해선 배열의 개념 사용
CTEXTURE라는 클래스 생성
CMaterial 안에 쉐이더와 텍스쳐를 넣고
게임 오브젝트는 그 머터리얼을 갖고 있는 형태
텍스쳐는 서술자 테이블로만 넘겨야 함
오브젝트의 텍스쳐가 바뀔 일이 있으면 원래 있는 SRV에 새로운 텍스쳐를 입히는 방식이 아닌,
SRV를 두 개 만들어서 Set으로 바인딩만 바꿔주면 됨

▶ 텍스쳐 필터링
하나 이상의 텍셀을 읽고 결합하여 하나의 색상을 생성하는 과정
▷ 점 필터링
장: 구현 빠름, 실시간 렌더링 용이
단: 계단 현상, 픽셀화 현상
▷ 선형 필터링
장: 부드러움, 계단 현상 완화
단: 계산 비용, 세부 정보 손실 가능
▷ 밉맵 필터링
장: 계단 현상 크게 감소, 좋은 품질
단: 추가 메모리 사용, 밉맵 레벨 사이의 전환 지점이 너무 잘보임
▷ 비등방 필터링
장: 경사진 표면 높은 품질, 원근감
단: 개많은 계산 비용, 추가 메모리 사용, 하드웨어 지원 필요(GPU 있어야됨)

▶ 텍스쳐 어드레스 모드
WRAP: 텍스쳐가 범위를 벗어나면 텍스쳐가 반복됨
CLAMP: 텍스쳐의 가장자리 픽셀이 확장되어 나머지 영역을 채움
BORDER: 텍스쳐의 범위 밖의 영역을 지정된 테두리 색상으로 채움
MIRROR: 텍스쳐가 범위를 벗어나면 텍스쳐가 거울처럼 반사되이 반복

▶ 텍스쳐 배열
배열 형식으로 텍스쳐를 사용할 수 있음
Texture2D gTreeTextures[4] : register(t4); // 여러 개의 디스크립터 사용(4~7)
-> UV 좌표를 사용
Texture2DArray gTreeTextureArray : register(t4); // 한 개의 디스크립터 사용(4번만)
-> UVW좌표 사용(W: 배열의 인덱스)
텍스쳐의 개수가 정해지지 않은 경우 gTxtruers4[] : registerr(t20) // UnBounded Size
-> D3D12_DESCRIPTOR_RANGGE.NumDescriptors = -1 // 크기가 정해지지 않으면 -1로 씀

▶ 후처리
렌더링된 이미지에 대해 추가적으로 시각적 효과 적용
블러링: 이미지를 부드럽게 / 텍스쳐 렌더링: 이미지에 텍스쳐 입히는 과정 / 이미지 필터링: 다양한 필터 효과 적용
샘플러 스테이트, 텍스쳐2D를 사용해서 텍스쳐 처리
픽셀 단위로 색상값 계산
UV 좌표를 이용한 텍스쳐 매핑

Deferred Shading(지연 조명)

▶ 스텐실 컬링

① 색상 출력을 비활성화하여 조명 볼륨을 렌더링

② 조명 계산 쉐이더를 사용하여 출력

▶ 지연 쉐이딩

씬과 조명을 분리

화면-좌표계의 쉐이딩 기법(2Pass)

① 쉐이딩을 먼저 수행하지 않고 보이는 픽셀에 대한 조명 처리 데이터를 수집,
각 표면을 위한 위치 벡터/법선 벡터/재질 등 일련이ㅡ 텍스쳐로 렌더링,
이런 일련의 텍스쳐를 기하 버퍼라고 함

② 각 픽셀에 대하여 화면 좌표계에서 조명 효과 계산

다중 렌더 타겟: 단일 패스로 G버퍼 속성 출력

씬을 텍스쳐로 렌더링(조명 계산 X)

장: 오브젝트 수에 비례하지 않는 조명 계산 가능

단: G버퍼에 여러 정보를 저장해야 하므로 메모리가 많지 않으면 속도가 느림

▶ 지연 렌더링

① G버퍼를 생성

② 지연 쉐이딩(조명 계산)

③ 추가 이펙트 처리

OM Blending Stencil

▶ 출력-병합 단계

최종적으로 픽셀의 색상을 생성하여 렌더 타겟으로 출려가는 단계

▶ 블렌딩

다중-패스 렌더링

여러 개의 픽셀 값을 결합하여 하나의 최종 픽셀 색상 결정

렌더 타겟이 색상과 픽셀 쉐이더의 출력 색상을 결합하여 최종 색상 결정

▶ 텍스쳐 블렌딩

렌더 타겟에는 알파값이 없음(모니터에 알파값이 없듯이)

소스 칼라: 픽셀 쉐이더의 리턴 색상

타겟 칼라: 렌더 타겟에 이미 그려져 있는 색상

이 둘을 섞는 것이 블렌딩

다중 텍스쳐링: 하나의 정점이 여러 개의 텍스쳐 좌표를 가진 경우 여러 개의 텍스쳐를 매핑 가능

▶ 알파 블렌딩

RS 단계에서 색상/알파 값 보간

투명 효과

정점의 색상, 재질의 색상, 텍스쳐 색상

▶ OM 블렌딩
블렌드 상태를 설정
D3D12_BLEND_DESC 구조체를 사용해서 설정(렌더 타겟 8개에 대해 설정)
BlendEnable
SrcBlend: 픽셀 색상에 곱하는 값
DestBlend: 렌더 타겟 색상에 곱하는 값
BlendOp: RGB 색상 블렌드 연산자
상태 설정 함수: OMSetBlendFactor

▶ 다중 샘플링(Multi-Sampling)
포함 여부(Coverage): 각 서브 픽셀의 중심이 다각형 내부에 존재하는가? 테스트 할 필요 있음

▶ 알파-커버리지 다중 샘플링
이미 투명한 텍스쳐는 알파 값으로 잘 되어있으므로 이 알파값을 사용하여 계산
AlphaToCoverateEnable: 내부적으로 Coverage 테스트를 안하겠음

▶ 텍스쳐 스플래팅
타일 형태의 텍스쳐 레이어 사용해서 렌더링
텍스쳐 여러 개 사용해서 이쁘게 매핑 가능

▶ 블렌딩 고려
빌보드가 아닌 경우 은면제거 X
Color = SrcColor * 1 + DestColor * SrcAlpha

▶ 렌더링 순서
투명한 다각형들은 카메라까지의 거리에 따라 정렬
카메라에서 거리가 먼 다각형 먼저 렌더링
블렌딩 연산이 더하기, 빼기, 곱하기 중 하나가 아닌 경우 기본적으로 정렬이 필요함

▶ 안개
결과 = 안개 인자 * 안개 적용 전 색상 + (1 - 안개 인자) * 안개 색상
정점 안개는 카메라 좌표계의 Z값을 사용하면 카메라 회전 시 문제 발생 가능

▶ 스텐실 검사
각 픽셀에 대해 스텐실 버퍼에 미리 저장된 값과 참조 값(reference value)을 비교하여 테스트를 수행
렌더 타겟 일부 영역을 렌더링 하지 않게 설정
거울 반사 표현 예
깊이 검사와 스텐실 검사 모두 통과해야 렌더링 됨(True, True)
하나만 성공했으면, 성공한 쪽의 갱신은 일어날 수 있음
검사 On 안해두면 성공한 것으로 취급

▶ 스텐실 표
뎁스는 뎁스와 스텐실 모두 true면 갱신
스텐실은 스텐실만 활성화 되도 갱신

▶ 깊이-스텐실 상태
▷ 구조체 D3D12_DEPTH_STENCIL_DESC
BOOL StencilEnable: 스텐실 검사 활성화
UINT8 StencilReadMask: 스텐실 버퍼 읽기 매스크
UINT8 StencilWriteMask: 스텐실 버퍼 쓰기 매스크
D3D12_DEPTH_STENCILOP_DESC FrontFace: 전면 다각형 스텐실 버퍼 갱신
D3D12_DEPTH_STENCILOP_DESC BackFace: 은면 다각형 스텐실 버퍼 갱신

▷ 구조체 D3D12 DEPTH_STENCILOP_DESC
D3D12_COMPARISON_FUNC StencilFunc: 스텐실 테스트시 비교 연산 정의

▷ enum D3D12_STENCIL_OP
KEEP, ZERO, REPLACE, INCR_SAT, DECR_SAT, INVERT, INCR, DECR

▶ 스텐실 참조 값
OMSetStencilRef(UINT StencilRef); // 참조값

▶ 뎁스-스텐실 뷰 생성 과정
① 리소스 설명자 설정
② 힙 속성 설정
③ 초기화 값 설정
④ 리소스 생성 및 뷰 생성

▶ 렌더 타겟과 뎁스-스텐실 버퍼 파이프라인에 연결
OMSetRenderTargets()

거울 세상의 경우에는 좌/우 앞면/뒷면이 바뀜(조명 구조체에서 위치와 방향이 바뀜)
순서
① 거울을 제외한 모든 객체 렌더링, 거울을 렌더링 하기 위해 파이프라인 상태가 4개 필요함
②-① 스텐실 버퍼 초기화(0)
②-② 거울을 스텐실 버퍼에 렌더링(렌더 타겟에는 출력X)
③-① 반사된 객체들 거울에 렌더링
③-② 거울에 반사된 조명 위치와 방향 반영
③-③ 반사된 객체 렌더링(반사 조명 사용)
④ 거울을 렌더링(블렌딩)

Geometry Shader
▶ 기하 쉐이더
하나의 완전한 프리미티브를 구성하는 정점
각 프리미티브에 대해 한번씩 호출
여러 번 그려야 하는 오브젝트를 GPU가 쉐이더로 여러 개 그려줌

▶ 기하 쉐이더 단계
SV_GSInstanceID: 실행되는 기하 쉐이더의 인스턴스 ID
[maxvertexcount(n)]: 각 프리미티브에 대하여 실행되는 기하 쉐이더의 인스턴수 개수 n개(최대 32개)
-> 삼각형의 정점이 18개라고 해서 삼각형이 6개가 아님: 알 수 없음(최대 16개)

▶ [instancce(n)]: 각 프리미티브에 대하여 실행되는 기하 쉐이더 인스턴스 n개

▶ PrimitiveType
▷ point: 점들의 리스트(1)
▷ line: 선분들의 리스트 또는 스트립(2)
▷ triangle: 삼각형들의 리스트 또는 스트립(3)
▷ lineadj: 인접성 선분들의 리스트 또는 스트립(4)
▷ triangleadj: 인접성 삼각형들의 리스트 또는 스트립(6)

▶ 함수
Append(StreamDataType): 출력 데이터를 스트림에 추가
RestartStrip(): 현재의 프리미티브 스트립을 끝내고 새로운 프리미티브 스트립 시작

▶ 쉐이더에서는 랜덤 제공X
무작위를 사용하려면?: 포지션을 가지고 난수를 생성

Stream-Output Stage
▶ 스트림 출력 단계
기하 쉐이더 또는 정점 쉐이더에서 출력되는 정점 데이터를 연속적으로 메모리 버퍼로 GPU를 사용하여 스트림 출력
스트림 출력 단계에 동시에 4개의 버퍼 연결 가능(출력 슬롯이 4개)
루트 시그니쳐가 D3D12_ROOT_SIGNATURE_FLAG_ALLOW_STREAM_OUTPUT을 사용해야 스트림 출력 단계 허용

▶ 스트림 출력 단계 설정
D3D12_RESOURCE_STATE_STREAM_OUT
▷ 스트림 출력 선언의 개수
▷ 배열
▷ 배열 원소 개수
▷ 래스터라이저 단계로 출력될 스트림 인덱스

▶ D3D12_SO_DELARATION_ENTRY
스트림 출력 버퍼의 정점 원소 표현
정점이 버퍼로 출력되는 방법을 정의
기하 쉐이더는 4개까지 출력 스트림 사용 가능
▷ 스트림 인덱스
▷ 시맨틱 이름
▷ 시맨틱 인덱스
▷ 출력 시작 요소
▷ 출력 요소 개수(1~4)　　▷ 스트림 출력 버퍼 슬롯(0~3)