3D게임
프로그래밍2

기말 정리

SO

▶ 스트림 아웃풋 오브젝트 3가지
- PointStream: 출력 프리미티브가 점인 경우
- LineStream: 출력 프리미티브가 선분인 경우
- TriangleStream: 출력 프리미티브가 삼각형인 경우

▶ D3D12_SO_DECLARATION_ENTRY: 정점이 버퍼로 출력되는 방법을 정의
- 멤버: 스트림 인덱스, 시맨틱 이름, 시맨틱 인덱스, 출력 시작 요소, 출력 요소 개수, 스트림 출력 버퍼 슬롯
- 위치 벡터의 모든 요소를 0번 슬롯의 버퍼로 출력
- 법선 벡터의 요소를 0번 슬롯의 버퍼로 출력
- 텍스쳐 좌표의 요소를 0번 슬롯의 버퍼로 출력

▶ SO 단계에 연결되는 리소스는 리소스 스테이트가 SO 상태일 때만 가능

▶ SOSetTargets
- UINT StartSlot // 스트림 출력 버퍼를 연결할 시작 개수
- UINT NumViews // 연결할 스트림 출력 버퍼의 개수

▶ D3D12_STREAM_OUTPUT_BUFFER_VIEW
- BufferLocation: 스트림 출력 버퍼
- BufferFilledSizeLocation: 스트림 출력 버퍼의 출력 위치를 저장하는 버퍼, GPU가 쓰기 작업을 완료한 후, CPU가 버퍼에 얼마나 많은 데이터가 들어 있는지를 확인 가능

▶ 스트림 출력 카운트
- SO시 GPU는 출력하기 위한 버퍼의 현재 위치(BufferFilledSize)를 알아야 함
- 응용 프로그램은 이 위치를 저장하기 위한 버퍼(32비트)를 할당해야 함
- 스트림 출력 버퍼에 잇는 현재까지 출력한 바이트 수를 나타냄
- GPU가 스트림 출력을 할 때 이 값을 사용하여 새로운 정점을 추가할 위치를 결정, 카운트를 증가
- 카운트 버퍼는 CPU에서 출력 위치를 변경하기 위해 접근 가능

Tessellation

▶ 테셀레이션

- 헐 셰이더 단계: 입력 제어점으로 부터 출력 제어점과 패치 상수 생성, 리턴
- 테셀레이션 단계: 도메인을 작고 많은 프리미티브로 분할
- 도메인 셰이더 단계: 분할된 점의 정점 위치 계산
- 테셀레이션은 적은 다각형으로 구성된 표면을 아주 세밀한 프리미티브로 변경

▶ 헐 셰이더

① 헐 셰이더 입력 및 출력 제어점 정의
② 출력 패치 상수 데이터 정의
③ 패치 상수 함수 정의(HS_CONSTANT_OUTPUT 리턴)
④ 헐 셰이더 함수를 정의, 각 출력 제어점에 대해 한 번씩 호출

▶ 헐 셰이더 시멘틱

▷ SV_TessFactor

- 패치의 각 엣지에 대한 테셀레이션 양 정의
- 사각형 패치의 시작은 왼쪽 엣지부터, 삼각형 패치의 시작은 두 번째 엣지부터
- 모든 테셀레이션 인자가 0이면 그 패치는 파이프라인에서 처리 X

▷ SV_InsideTessFactor

- 패치 표면에 대한 테셀레이션 양 정의
- 테셀레이션 인자들은 배열로 선언되어야 함

▷ SV_OutputControlPointID: 헐 셰이더에서 현재 사용하는 출력 제어점의 인덱스 정의
▷ SV_DomainLocation: 현재 도메인 점의 헐에서의 위치 정의
▷ domain: 패치 유형
▷ partitioning: 분할 방법(테셀레이션 인자 사용 방법)
▷ outputtopology: 출력 프리미티브 유형
▷ outputcontrolpoints: 출력 제어점의 개수
▷ pathconstantfunc: 패치 상수 함수 이름
▷ maxtessfactor: 최대 테셀레이션 인자

▶ 도메인 셰이더

- 헐 셰이더에서 넘겨주는 패치 상수에 해당하는 정보
- 실제적인 정점들을 생성하고, 그 정점들에 대해 트랜스폼 하는 과정
- 테셀레이터가 생성한 정점의 위치는 출력 패치의 헐에 대한 중심 좌표계로 주어짐

▶ 테셀레이션 설정법

① 테셀레이션 컨트롤 포인트 밀도 정의(헐 셰이더)
② PutputPatch 형식으로 입력을 받고 테셀레이션 팩터를 설정
③ 헐 셰이더에서 생성한 데이터와 테셀레이션 좌표를 사용해 최종 정점 데이터 계산(도메인 셰이더)
④ 테셀레이션 후의 데잍터를 기반으로 픽셀 그림
⑤ 카메라 거리 기반으로 LOD를 조절하여 성능 최적화

▶ 테셀리이션과 기하셰이더의 차이

▷ 테셀레이션

- 정점을 세밀하게 분할하여 높은 해상도 제공
- 헐, 도메인 셰이더에서 작동
- 패치 데이터를 받아 분할된 새로운 정점 및 위치 계산

▷ 기하셰이더

- 정점을 기반으로 새로운 기하학적 구조 생성
- 버텍스 셰이더에서 작동
- 정점/선/삼각형을 받아 새로운 기하학 구조 객체 생성

Compute Shader

▶ 컴퓨트 셰이더 사용 이후

- 비디오 메모리 내용을 읽는 연산은 매우 느리기 때문에 사용

▶ 컴퓨트 셰이더

▷ RWTexture2D<float4> gtxRWOutput : register(u3)

- 2중 for문 돌려야 할거 GPU를 통해 빠르게 가능
- Read Write 모두 가능
- GPGPU 프로그래밍 제공
- 스레드는 계산 셰이더의 기본 처리 단위
- Dispatch: 지금부터 병렬 처리 작업 시~작

▶ 컴퓨트 셰이더 시멘틱

- SV_GroupThreadID: 스레드 그룹에서 계산셰이더가 실행되고 있는 스레드의 인덱스
- SV_GroupID: 계산 셰이더가 실행하고 있는 스레드 그룹의 인덱스
- SV_DispatchThreadID: 계산 셰이더가 실행하고 있는 스레드와 스레드 그룹이 결합된 인덱스
- SV_GroupIndex: 하나의 스레드 그룹에서 스레드의 인덱스
- [numthreads(n, m, l)]: 하나의 스레드 그룹의 스레드 개수(최대 1024개),
  각 스레드 그룹은 n * m * l개의 스레드로 구성
- Dispatch(x, y, z): 해당 방향으로 디스패치될 그룹의 개수, 스레드 그룹의 크기를 정의, 65535보다 작아야 함

▶ GPU 메모리에서 시스템 메모리로 리소스 복사

① CPU에서 읽을 수 있는 버퍼 생성 D3D_HEAP_TYPE_READBACK

② ID3D12GraphicsCommandList::CopyResource()로 리소스를 버퍼로 복사

③ 버퍼를 Map하여 읽음

▶ 공유 메모리 동기화

- 스레드 그룹의 모든 그룹 공유 접근이 끝날 때까지 그리고 그룹의 모든 스레드가 이 함수를 호출할 때까지 각 스레드의 실행을 멈추고 기다림
- GroupMemoryBarrierWithGroupSync: 스레드 그룹의 모든 공유 접근이 끝날 때까지, 그룹의 모든 스레드가
  이 함수를 호출할 때까지 각 스레드의 실행을 멈추고 기다림
- GroupMemoryBarrier: 모든 그룹 공유 접근이 끝날 때까지 하나의 스레드 그룹의
  모든 스레드의 실행을 멈추고 기다림

Blurring

▶ 컨벌루션

- 하나의 함수가 다른 함수 위에서 움직이면서 겹치는 양을 표현하는 적분
- 스무딩: 모든 가중치를 같게
- 블러 마스크: 특정 픽셀의 가중치를 모두 같게
- 모션 블러 마스크: 대각선 픽셀의 가중치를 모두 같게
- 가우시안 블러: 중앙 픽셀에 가중치를 더 높게

▶ 가우시안 필터

- 가우시안 분포(평균을 중심으로 좌우 대칭의 종 모양을 갖는 확률 분포) 함수를 근사하여 생성한 필터 마스트를 사용하는 필터링 기법(정규 분포 함수)

1 2 1
2 4 2
1 2 1

▶ 기하셰이더에서의 블러링

- HorzBlurCS: x축, 수평으로 인접한 픽셀들 가중합 계산
- VertBlurCS: y축, 수직으로 인접한 픽셀들 가중합 계산
- 블러링 작업을 한 번에 2D 전체에 적용하면 두 번의 1D 블러링(수평 + 수직)으로 분리하여 계산량이 줄어듬

▶ 백열광 효과

- 씬을 블러링한 결과와 씬을 결합하여 렌더링

① 씬을 텍스쳐로 렌더링
② 화면 크기의 사각형을 렌더링, 렌더 타겟을 리소스로 결정, 렌더 타겟에서 밝은 부분을 추출하여 렌더링
③ 화면 크기의 사각형을 렌더링, 렌더 타겟을 리소스로 설정
④ 블러링 결과와 씬의 렌더링 결과 합성

Shadow

▶ 원형 그림자

- D3D12_BLEND_ZERO
- D3D12_BLEND_INV_SRC_COLOR
- D3D12_BLEND_OP_ADD

▶ 평면 투영 그림자

- 조명에서 정점을 지나는 광선이 평면과 교차하는 점을 구해 그림자 모델 생성
- 방향성 광원의 평면 투영

▶ 그림자 텍스쳐 동적 생성

- 객체의 모양과 같은 그림자 텍스쳐 생성

① 카메라를 조명 위치로 이동, 카메라 변환 행렬을 사용하여 설정
② 텍스쳐를 생성하여 렌더 타겟으로 설정, 렌더 타겟을 검정색으로 지움
③ 객체를 흰색으로 렌더링
④ 후면버퍼를 렌더 타겟으로 설정
⑤ 그림자 텍스쳐를 렌더링

▶ 투영 텍스쳐 매핑

① 조명(프로젝터)을 카메라로 생각할 수 있음
② 점(Px, Py, Pz)을 투영 좌표계로 변환(투영)
③ 텍스쳐 좌표계로 변환
④ 점(Px, Py, Pz)의 색상을 텍스쳐로 변환

▶ 그림자 매핑 - 조명을 카메라로 가정하고 렌더링 과정으로 이해

- 조명이 객체에 도달하면 여부를 계산하여 텍스쳐(깊이 맵)에 저장

① 카메라를 조명 위치에서 생성, 카메라 변환 행렬 V를 생성하여 설정, 투영 변환 행렬 P를 생성하여 설정
② 텍스쳐를 생성하여 깊이스텐실 버퍼로 설정, 렌더 타겟은 필요 없음
③ 그림자 맵 변환 행렬을 위한 상수 버퍼를 생성, 월드 좌표계의 점을 조명 좌표계로 변환하기 위한 변환 행렬
④ 객체들을 렌더링, 투명한 픽셀은 깊이 값을 출력할 필요 없음

▶ 그림자 렌더링

① 렌더 타겟과 깊이 버퍼 설정
② 그림자 맵 텍스쳐를 셰이더 리소스로 연결
③ 씬을 렌더링하면서 각 픽셀이 그림자에 해당하는 가를 결정
④ 정점 셰이더에서 렌더링할 메쉬의 각 정점의 월드 좌표를 조명 좌표계로 변환,
조명 좌표계를 텍스쳐 좌표계로 변환
⑤ 픽셀 셰이더에서 각 픽셀에 대한 조명 좌표계의 거리를 구함
⑥ 그림자 맵 텍스쳐의 깊이값과 픽셀의 깊이값을 비교하여 그림자 여부를 판단,
그림자에 해당하는 픽셀의 조명 효과를 줄이거나 제거(검은색으로 그림)

▶ 투영 텍스쳐링
- 점 P에 대한 그림자 맵(텍스쳐) 샘플링 UV 좌표의 계산
① 점 P를 조명의 투영 평면으로 투영 그림자 맵을 생성할 때의 변환 행렬이 필요
② 투영된 좌표를 텍스쳐 좌표로 변환, 투영된 좌표는 -1 <= x, y <= 1
XMMaterixPerspectiveFovLH(): 원근 투영 변환 행렬을 생성하여 반환(카메라 좌표계)
XMMatrixOrthographicLH(): 직교 투영 변환 행렬을 생성
XMMatrixOrthographicOffCenterLH(): 직교 투영 변환 행렬을 생성

▶ 그림자 맵 문제: 깊이 값 계산의 정밀도 오차 때문에 발생
- 그림자 여드름, 물결 패턴
- 원근/투영으로 인한 계단 현상
- 물결치는 엣지

▶ 바이어스
- 그림자 맵의 거리에 바이어스를 더해서 그림자 여드름 해결
- 바이어스를 동적으로 계산해주는 것이 GPU에 있음: RASTERIZER_DESC
▷ RASTERIZER_DESC 구조체
- DepthDias: 각 픽셀에 더해질 깊이 바이어스
- DepthBiasClamp: 픽셀의 깊이 바이어스의 최대값
- SlopeScaledDepthBias: 픽셀의 기울기에 따라 바이어스를 조절

▶ 그림자 맵
- 조명 투영 공간을 렌더링 카메라 투영 공간(절두체)에 가급적 밀접하게 만들어야 함
- 그림자 맵의 커버리지를 증가시킴
① 렌더링 카메라 절두체의 8개 꼭지점을 조명좌표계로 변환
② 변환된 x좌표와 y좌표들의 최소값과 최대값을 직교 투영 공간으로 사용

▶ PCF(Percentage Closer Filtering) - 뭔가 시험에 나올법함
- 주변의 텍셀을 무작위로 샘플링, 포인트 필터를 반드시 써야 함
① 샘플러 상태 객체 필터링 좌표에 대한 깊이 값 검사(D3D12_FILTER_MIN_MAG_MIP_POINT)
② 텍스쳐 좌표에 대한 샘플링
③ 깊이 값 d에 대한 그림자 맵 검사
④ 검사 결과에 대한 선형 보간
- SampleCmpLevelZero: 현재 UV 좌표에 해당하는 픽셀의 그림자맵 비교 검사를 GPU에서 해줌

▶ 방향성 조명 그림자
- 전체 씬에 대한 그림자 계산에 사용

▶ 원근 투영 에일리어싱
- 카메라에 가까운 그림자 맵의 픽셀들이 확대되어 그림자에 계단 현상이 발생
- 그림자 맵을 생성할 때 원근 투영에 의해 깊이 값이 저장되는 텍셀 수가 달라짐

▶ 직렬 그림자 맵
- 방향성 조명은 아주 넓은 영역을 비춤
- 카메라 절두체를 여러 영역으로 나누고 각 영역을 같은 해상도의 그림자 맵으로 생성