Graphics Study

Real-Time Rendering 4th edition

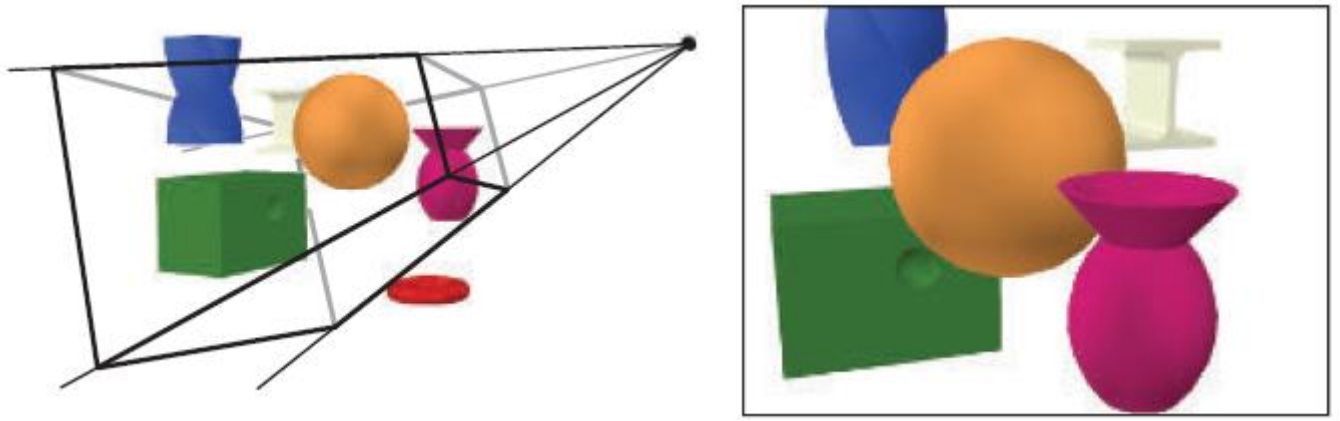
Ch 2, 3

개요

- 그래픽스 렌더링 파이프라인을 구성하는 각 단계를 순서대로 따라가며 개념과 구현을 살펴봅니다.

그래픽스 렌더링 파이프라인(The Graphics Rendering Pipeline)

- 입력된 정보(일반적으로 3차원)를 가공하여 2차원 래스터 이미지로 변환하는 공정 과정



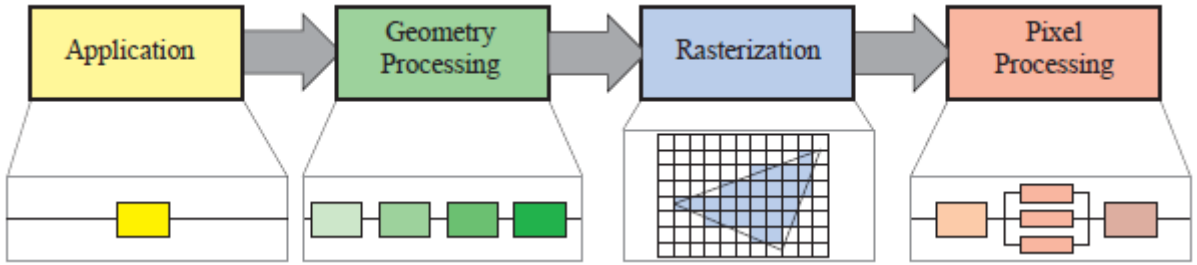
래스터 시스템(Raster System)

- 작은 화소를 격자 형태로 배치하여 이미지를 출력하는 장치
  - 화소: 화면을 전기적으로 분해한 최소 단위 면적
- 이 시스템의 출력 형식에 맞게 작성된 이미지가 래스터 이미지
- 렌더링 파이프라인을 구성하는 단계 중 하나인 래스터화(Rasterization)의 래스터(Raster)와 동일

아키텍처(Architecture)

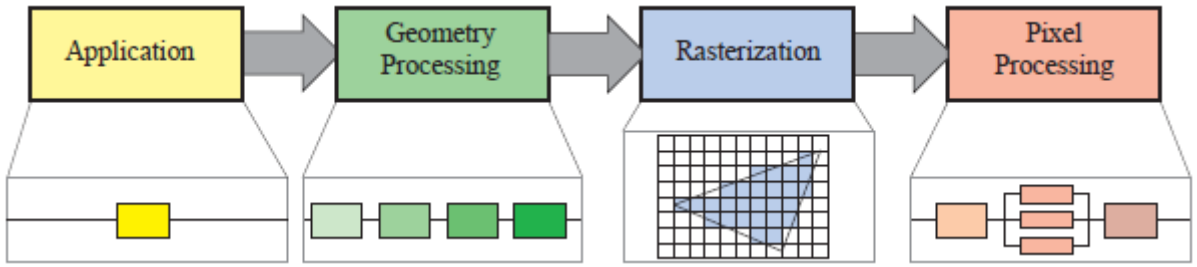
- 렌더링 파이프라인의 구조도

- 그 자체가 또 하나의 파이프라인으로 구성되는 경우도 있고, 부분적으로 병렬 처리가 수행되는 경우도 있음



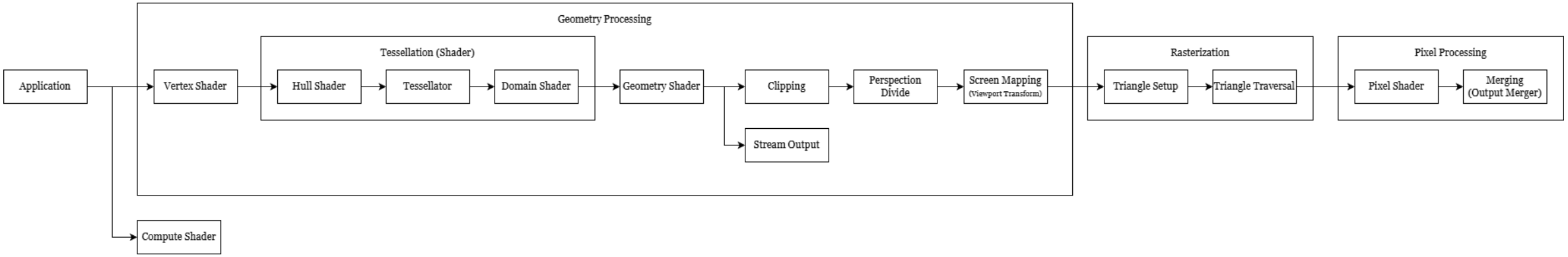
아키텍처(Architecture)

- 파이프라인 구조가 가져오는 이점
  - n배의 성능 향상
    - 처음 동작하여 결과물이 나타날 때까지는 논파이프라인 시스템과 차이가 없음
    - 그러나 입력이 계속 진행되어 파이프라인이 꽉 차게 된다면
    - 파이프라인 구조의 작업 완료 주기가 더 짧아짐 (단계 별 수행 시간이 동일하다는 균일하다는 가정 하에 n배의 성능 향상)



아키텍처(Architecture)

- 렌더링 파이프라인의 구조도
- 투영(Projection)이 실행되는 지점은 지오메트리 프로세싱의 마지막 단계 (VS, DS, GS 중 택일)



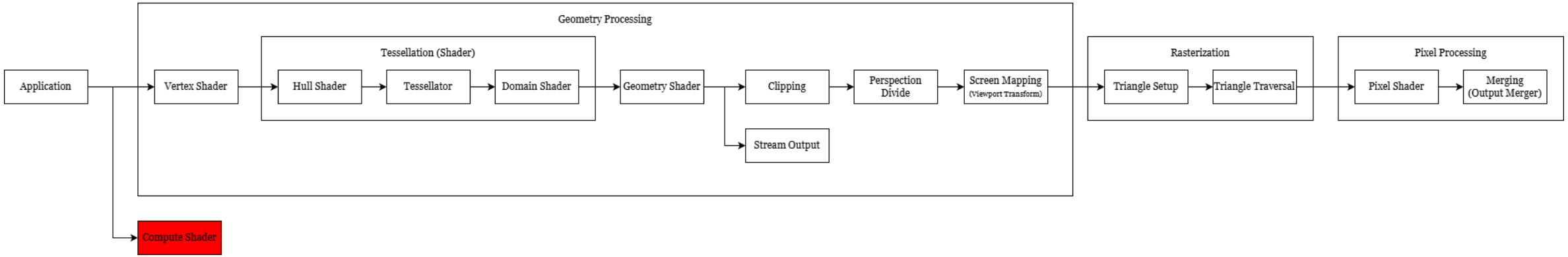
애플리케이션 단계(The Application Stage)

- CPU(Central Processing Unit)에 의해 실행되는 단계
  - 쉽게 말해, 저희가 평소 코딩하는 공간입니다.

- 이 단계에서 적절한 알고리즘과 설정을 통해 렌더링할 삼각형의 수를 줄일 수도 있음
  - CPU 기반 컬링 등 (이후의 챕터에서 다루게 될 예정)

애플리케이션 단계(The Application Stage)

- 컴퓨트 셰이더(Compute Shader)를 사용하여 일부 작업은 GPU에서 처리할 수 있음
  - 컴퓨트 셰이더: 렌더링을 위한 특수 기능들을 무시하고 고도로 병렬화된 다목적 프로세서(general-purpose parallel processor)처럼 활용하는 모드
  - 계산 결과는 VRAM에 저장됨
    - 때문에 CPU 메모리로 읽어올(readback) 때에 실행 성능에서 오버헤드가 존재
- GPGPU(General-Purpose computing on Graphics Processing Units)의 부분집합



애플리케이션 단계(The Application Stage)

- 실행 성능의 향상을 위한 솔루션은 CPU에도 존재

- 슈퍼스칼라(Superscalar) 구조
  - 한 번의 클럭(Clock)에서 여러 개의 명령어를 동시에 처리하는 구조
    - 분기 예측(Branch Prediction)
    - 명령어 재정렬(Instruction Reordering)
    - 레지스터 리네이밍(Register Renaming)

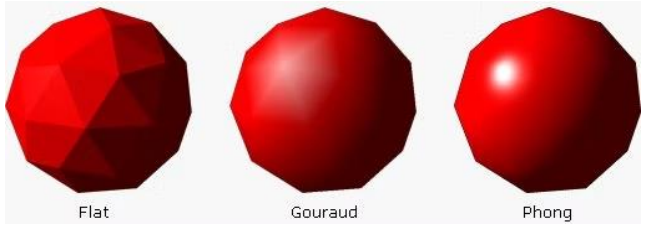
- 멀티스레딩(Multithreading)과 멀티프로세싱(Multiprocessing)

입력 어셈블러(Input Assembler)

- 애플리케이션 단계에서 사용자가 입력한 데이터를 조합하여 그래픽스 파이프라인으로 전달
  - 버퍼 스트림과 인덱스 버퍼(Index Buffer)를 조합하여 지오메트리 프로세싱(Geometry Processing)으로 전달
  - 인스턴싱(Instancing) 기법 또한 입력 어셈블러가 담당

버텍스 셰이딩(Vertex Shading)

- 현대의 셰이더 프로그래밍 패턴에서 버텍스 셰이딩의 주요 작업
  - 입력 받은 정점의 위치를 적절한 공간 상의 위치로 변환
  - 법선, 텍스처 좌표 등 정점의 속성을 함께 전달



- 근데 왜 버텍스 '셰이딩'?
  - 과거의 GPU에서는 조명과 색상의 계산을 버텍스 셰이더에서 조명과 색상 계산까지 수행
    - 과거의 픽셀 셰이더는 거의 프레임버퍼(Framebuffer)로 값을 전달만 하는 수준
  - 계산된 색상은 래스터라이저(Rasterizer)에 의해 보간이 이루어짐
    - 이러한 방식을 고러드 셰이딩(Gouraud Shading)이라 부름
  - GPU의 발전과 함께 픽셀 셰이더에서 처리하는 형태로 굳어짐
    - 이제는 사용자가 원한다면 셰이딩 방정식을 아예 처리하지 않을 수도 있게 되었음
    - 더욱 범용적인 처리 공간으로 발전

- 프로그래머블 버텍스 프로세싱 유닛(Programmable Vertex Processing Unit) = 버텍스 셰이더(Vertex Shader)
  - 프로그래머블 = 프로그래밍 가능한 = 사용자 정의 프로그램(셰이더)을 실행할 수 있는

버텍스 셰이딩(Vertex Shading)

- 이 단계에서 정점이 변환을 거치며 위치하는 공간들
  - 모델 스페이스(Model Space)
  - 월드 스페이스(World Space)
  - 뷰 스페이스(View Space)
    - 위 세 행렬은 행렬 곱을 통해 하나의 행렬로 결부되어 사용되기도 함
  - 클립 스페이스(Clip Space)
    - 투영 행렬(Projection Matrix)의 경우, 환경에 따라 이 곳에서 처리되지 않을 수도 있음

버텍스 셰이딩(Vertex Shading)

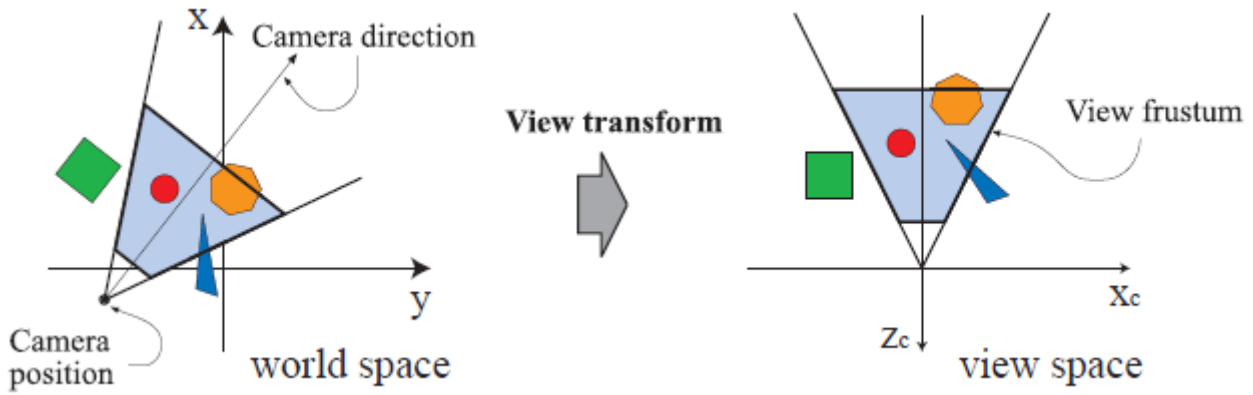
- 모델 스페이스
  - 모델 고유의 위치 오프셋
  - 회전
  - 스케일

버텍스 셰이딩(Vertex Shading)

- 월드 스페이스
  - 다른 모델들과 함께 존재하는 공간
  - 논리적으로 유일성을 가짐

버텍스 셰이딩(Vertex Shading)

- 뷰 스페이스
  - 카메라가 원점에 위치하고, 음의 z축을 바라보는 공간
    - 바라보는 방향은 그래픽스 API에 따라 다를 수 있음
    - 예를 들어, DirectX는 카메라가 양의 z축을 바라봄
  - 투영과 클리핑(Clipping) 계산을 간소화하기 위해 존재하는 단계
  - 뷰 프러스텀(View Frustum)
    - 절두체(꼭짓점이 절삭된 피라미드 형상)라고도 함
    - 원근(Perspective) 투영, 직교(Orthographic) 투영 등 현재 적용되는 투영에 따라 형상에 차이가 있음
    - 논리적으로 존재하는 개념



버텍스 셰이딩(Vertex Shading)

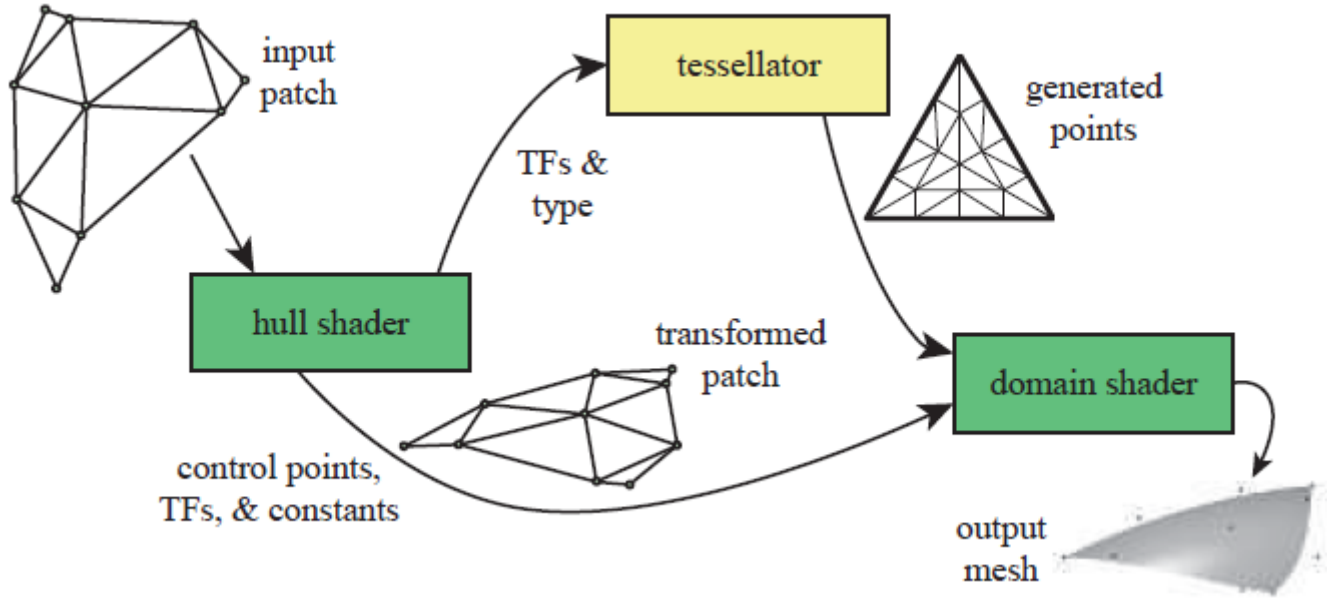
- 클립 스페이스
  - 클립 좌표계(Clip coordinates), 동차 좌표계(homogeneous coordinates), 동차 클립 공간 등 다양하게 불림
  - 마치 프러스텀의 두 밑면이 정사각형으로 변환된 공간
    - 원근 투영 중이었다면 가까이 있는 정점은 확대, 멀리 있는 정점은 축소

선택적 버텍스 프로세싱(Optional Vertex Processing)

- 테셀레이션(Tessllation)
  - 헐 셰이더(Hull Shader)
    - 처리의 전반부
    - OpenGL에서는 테셀레이션 제어 셰이더(Tessellation Control Shader)라 칭함
  - 테셀레이터(Tessellator)
  - 도메인 셰이더(Domain Shader)
    - 처리의 후반부
    - OpenGL에서는 테셀레이션 평가 셰이더(Tessellation Evaluation Shader)라 칭함
- 지오메트리 셰이더(Geometry Shader)
- 스트림 출력(Stream Output)
- 사용 중인 그래픽스 API와 하드웨어 환경에 따라 지원하지 않을 수도 있음

테셀레이션(Tessellation)

- 묘사의 정밀도를 올리기 위해 사용하는 선택적 단계
  - 가까이에 있는 물체는 묘사를 세밀하게, 멀리 있는 물체는 간단하게
    - 픽셀 한 두 칸 정도를 차지하는 아주 멀리 있는 공은 사각형이 삼각형 정도로 렌더링해도 문제 없을 것임
  - 특히 곡면 표현에 유용하게 사용 가능
  - 테셀레이터는 삼각형 뿐 아니라 점(Points), 사각형(Quads), 아이솔라인(Isoline)도 지원



테셀레이션 – 헐 셰이더(Hull Shader)

- 테셀레이터로 분할에 사용할 프리미티브, 컨트롤 포인트, 테셀레이션 팩터 정보를 전달
  - 프리미티브에는 삼각형(Triangle) 뿐 아니라 사각형(Quadrilateral), 아이솔라인(Isoline)도 존재
  - 테셀레이션 팩터로 0 이하의 값 또는 NaN의 논리적으로 적절하지 않은 값을 보내면 테셀레이터를 건너뛰는 것 또한 가능

테셀레이션 – 테셀레이터(Tessellator)

- 고정 기능 유닛으로 실제 분할을 진행 (셰이더 밖에서 이루어지는 기능)
  - 세분된 프리미티브를 구성하는 모든 정점에 대해 질량 중심 좌표(Barycentric Coordinates) 및 도메인 셰이더로 전달할 데이터를 출력
    - 질량 중심 좌표계는 삼각형 내부의 한 점을 세 꼭짓점의 선형 조합으로 표현하는 좌표계

테셀레이션 – 도메인 셰이더(Domain Shader)

- 테셀레이터가 전달한 정보를 바탕으로 버텍스 셰이더처럼 정점의 최종 위치를 결정

지오메트리 셰이더(Geometry Shader)

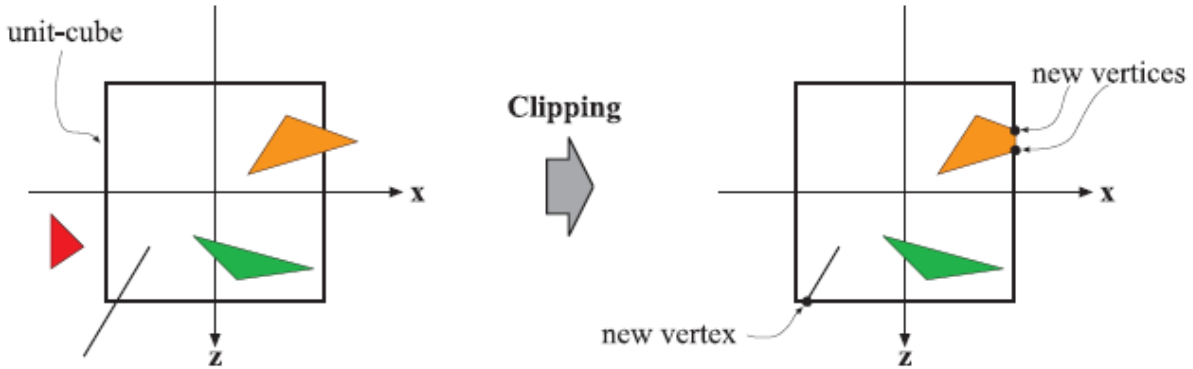
- 프리미티브를 다른 프리미티브로 변환하는 작업을 수행
  - 삼각형 각각의 변을 선분으로 만들어 와이어프레임 뷰로 변환하거나
  - 선분 대신 빌보드(Billboard, 면이 카메라를 향하는) 사각형으로 표현하여 더 두꺼운 선처럼 보이게 만드는 것도 가능
- 따라서 단일 프리미티브 및 그에 속한 정점들이 입력으로 들어옴
- 하드웨어의 사양에 민감한 단계로, 일부 하드웨어가 지원하지 않는 모바일 디바이스에서는 소프트웨어로 구현되기 때문에 실행 성능이 많이 떨어질 수 있음

Figure 3.13. Some uses of the geometry shader (GS). On the left, metaball isosurface tessellation is performed on the fly using the GS. In the middle, fractal subdivision of line segments is done using the GS and stream out, and billboards are generated by the GS for display of the lightning. On the right, cloth simulation is performed by using the vertex and geometry shader with stream out. *(Images from NVIDIA SDK 10 [1300] samples, courtesy of NVIDIA Corporation.)*

클리핑(Clipping)

- 뷰 볼륨에서 완전히 벗어난 프리미티브를 걸러내고, 부분적으로 포함되는 프리미티브를 볼륨에 맞춰 절삭하는 과정

- 투영 행렬을 적용함으로써 단위 큐브(unit cube) 기반의 절삭 연산을 수행할 수 있게 함
  - 단위 큐브의 범위는 그래픽스 API에 따라 다르게 정의 ([-1, 1] 또는 [0, 1])
  - 고정 기능 유닛에는 클리핑 제약식만 하드코딩되어 있고, 어떤 그래픽스 API가 실행되냐에 따라 다른 값이 적용되는 형태



원근 분할(Perspective Divide)

- 동차 좌표계 (Homogeneous Coordinates)
  - 동차
    - 동일한 형태를 유지한 채 배율을 바꿀 수 있음
    - 모든 항이 동일한 차수를 가짐
      - 투영기하학(projective geometry)에서 표현되는 점(투시에 따라 투영되는 점으로, 실제로는 벡터)의 방정식: ax + by + cz + dw = 0

- 4D 동차 공간인 클립 스페이스를 w 평면으로 투영하는 지점
  - x, y, z 성분을 w 성분으로 나눔으로써 구현됨

- 이 작업이 완료되면 정점이 NDC(Normalized Device Coordinates) 공간에 진입

스크린 맵핑(Screen Mapping)

- 클리핑 단계를 거쳐 뷰 볼륨 내에 존재하는 프리미티브들만 본 단계에 진입

- 각 프리미티브의 x, y 성분은 스크린 좌표(Screen Coordinates)로 변환됨
  - (스크린 좌표 x 성분, 스크린 좌표 y 성분, z 성분) = 윈도우 좌표(Windows Coordinates)
  - 변환 과정은 행렬곱으로도 표현할 수 있지만, 실제 GPU 파이프라인에서는 전용 스테이지가 따로 존재하여 하드코딩된 수식을 처리하는 구조로 알려짐

- 변환된 좌표는 래스터라이저에 전달

래스터라이제이션(Rasterization)

- 앞의 공정을 거친 정점 및 대응되는 셰이딩 데이터를 바탕으로, 프리미티브 안에 포함되는 모든 픽셀을 찾아내는 역할

- 삼각형 설정(Triangle Setup, Primitive Assembly)과 삼각형 순회(Triangle Traversal)의 두 단계로 구성됨

- 스캔 컨버전(Scan Conversion)이라고도 불리는데, 과거 스캔라인(Scan Line) 방식의 그래픽스 하드웨어에서 유래한 용어
  - 스캔라인: 가로로 한 줄씩 처리해야 할 픽셀을 평가해나가는 방식

래스터라이제이션 – 삼각형 설정(Triangle Setup)

- 삼각형에 대한 미분(differentials), 변 방정식(edge equations) 및 기타 필요 데이터를 계산
  - 미분의 경우 벡터 형태의 셰이딩 데이터에 각 성분마다 계산을 수행하는 낭비를 줄이기 위해 도함수를 계산해두는 것
  - 이게 없다면 각 픽셀마다 바리센트릭 좌표계 계산을 수행해야 함 (곱과 합 모두 필요)
  - 그러나 도함수 값을 사용하면 덧셈만으로 처리가 가능

- 계산된 데이터는 다음 단계인 삼각형 순회와 셰이딩 데이터 보간에 사용

- 하드웨어 고정 기능으로 처리함

래스터라이제이션 – 삼각형 순회(Triangle Traversal)

- 픽셀의 중심 혹은 샘플이 삼각형에 덮여 있는지 판정 (5.4 절에서 설명)
  - 안티앨리어싱이 이 지점에서 수행됨
- 덮여 있는 픽셀의 일부에 대해 보간과 함께 프래그먼트(Fragment)를 생성
  - 프래그먼트: 픽셀 셰이더의 입력 값이 담긴 데이터 묶음, 픽셀의 후보 같은 존재
- 생성된 정보를 다음 단계로 전달
  - 전달 전에 Early Z라는 간이 Z 테스팅을 수행할 수 있음
  - 이는 하드웨어 종속적인 기능

데이터 병렬 아키텍처(Data-Parallel Architectures)

- GPU는 SIMD(Single Instruction, Multiple Data) 기반의 하드웨어 설계
  - 고정된 수의 셰이더 프로그램을 락스텝으로 실행

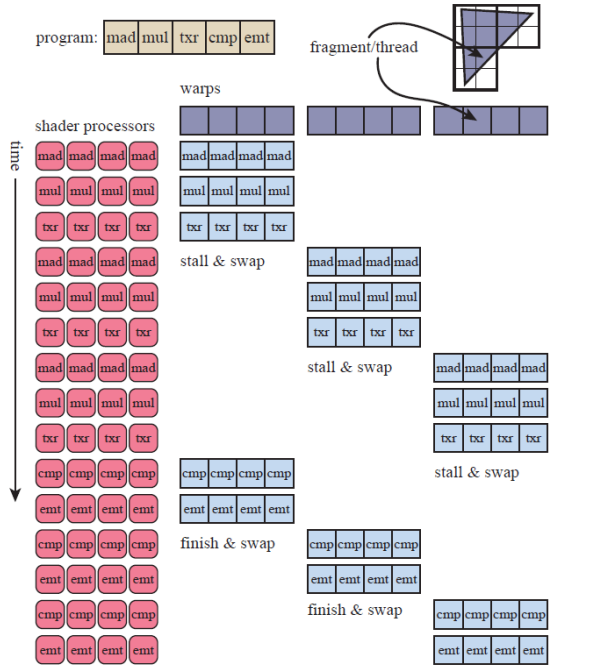


Figure 3.1. Simplified shader execution example. A triangle's fragments, called threads, are gathered into warps. Each warp is shown as four threads but have 32 threads in reality. The shader program to be executed is five instructions long. The set of four GPU shader processors executes these instructions for the first warp until a stall condition is detected on the "txr" command, which needs time to fetch its data. The second warp is swapped in and the shader program's first three instructions are applied to it, until a stall is again detected. After the third warp is swapped in and stalls, execution continues by swapping in the first warp and continuing execution. If its "txr" command's data are not yet returned at this point, execution truly stalls until these data are available. Each warp finishes in turn.

데이터 병렬 아키텍처(Data-Parallel Architectures)

- 스톨(Stall): 실행 지연

- CPU와 GPU는 하드웨어 구현의 차이 때문에 서로 다른 스톨 회피 기법을 가짐

- CPU의 스톨 회피 기법
  - 분기 예측(Branch Prediction)
  - 명령어 재정렬(Instruction Reordering)
  - 레지스터 리네이밍(Register Renaming)
  - 캐시 프리페칭(Cache Prefetching)

- GPU의 스톨 시나리오와 회피 방식
  - 대표적인 스톨 발생 원인으로 텍스처 페칭(Texture Fetching)이 있음
  - 수천 개의 스레드에 대해 스톨이 발생할 때 다른 명령으로 전환하며 실행

셰이더 가상 머신(Shader Virtual Machine)

- 작성된 셰이더 언어를 중간 단계 언어로 변환하는 가상 머신
  - GPU마다 실제 구현에는 차이가 있지만, 셰이더 컴파일러는 공통된 가상 머신 모델을 기준으로 코드를 변환함
- 데이터 로드 및 적재 성능을 올리기 위해 레지스터를 사용
  - 범용 입력 레지스터(Varying Input Registers): 이전 단계에서 넘어오는 데이터를 보관하기 위한 장소
  - 임시 레지스터(Temporary Registers): 셰이더의 실행에 필요한 지역 변수, 중간 계산 결과를 저장하는 레지스터
  - 출력 레지스터(Output Registers): 셰이더의 결과가 적재되는 장소
  - 상수 레지스터(Constant Registers): 외부에서 지정한 유니폼(Uniform) 프로퍼티를 보관하는 레지스터
  - 텍스처(Textures): 텍스처 샘플링 성능을 높이기 위해 특수화한 상수 레지스터

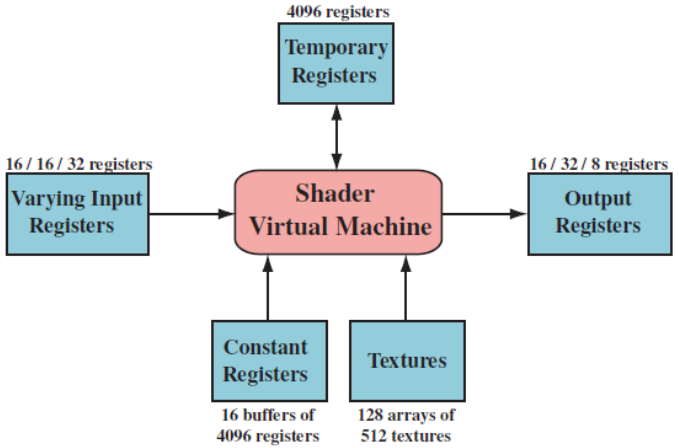


Figure 3.3. Unified virtual machine architecture and register layout, under Shader Model 4.0. The maximum available number is indicated next to each resource. Three numbers separated by slashes refer to the limits for vertex, geometry, and pixel shaders (from left to right).

픽셀 셰이더(Pixel Shader)

- 래스터라이저에 의해 전달 받은 데이터를 기반으로 목표 버퍼에 출력할 최종 데이터를 계산하는 지점

- 초기에는 유일한 출력 버퍼로만 값을 출력할 수 있었으나, 픽셀 셰이더가 단위 시간에 처리할 수 있는 능력이 오르고 메모리 또한 증대됨에 따라 출력 대상을 여러 개로 지정할 수 있게됨
  - 이를 MRT(Multiple Render Targets)라 부름
  - MRT의 지원으로 디퍼드 셰이딩(Deferred Shading)이라는 새로운 파이프라인이 탄생

픽셀 셰이더(Pixel Shader)

- 프래그먼트를 폐기하는 기능도 존재하는데, 이를 사용하여 사용자 정의 클리핑을 구현할 수 있음

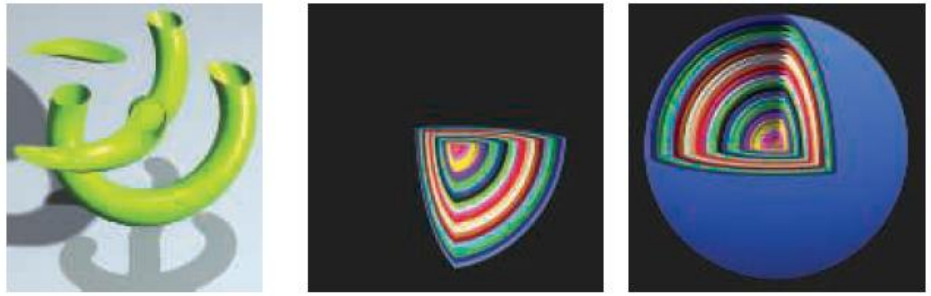


Figure 3.14. User-defined clipping planes. On the left, a single horizontal clipping plane slices the object. In the middle, the nested spheres are clipped by three planes. On the right, the spheres' surfaces are clipped only if they are outside all three clip planes. *(From the three.js examples webgl_clipping and webgl_clipping_intersection [218].)*

픽셀 셰이더(Pixel Shader)

- 픽셀 셰이더는 일반적으로 이웃 픽셀에 접근할 수 없음

- 예외적으로 그래디언트 함수를 사용하여 이웃에 접근할 수 있음
  - 락스텝(lockstep)으로 진행되는 GPU의 동작을 응용한 것
    - 락스텝: 같은 명령어 스트림을 실행하기 때문에 같은 시점에 같은 연산을 수행해야 하는 처리 구조

픽셀 셰이더(Pixel Shader)

- UAV(Unordered Access View)/SSBO(Shader Storage Buffer Object)
  - UAV는 DirectX 11에서 도입된 기능
  - SSBO는 OpenGL에서 UAV와 동일한 개념을 제공하는 기능
  - 셰이더에서 전역 위치에 계산 결과를 읽고 쓰기 위해 지원되기 시작
- 그러나 동시 접근이 가능해지면서 데이터 해저드(Data Hazard)가 나타남
- 이를 방지하기 위한 해결책으로 원자적 연산, ROV(Rasterizer-Ordered Views) 등이 지원됨
  - ROV: UAV의 발전 형태로, 래스터라이저에서 픽셀 셰이더 호출 순서를 조정함으로써 UAV 쓰기 순서를 보장하는 기능
  - 그러나 이들 역시 저마다의 성능 오버헤드가 존재

병합(Merging)

- 픽셀 셰이더에서 처리한 각 프래그먼트의 깊이 및 색상이 프레임버퍼와 결합되는 단계
  - DirectX: 출력 병합기(Output Merger)
  - OpenGL: 샘플 별 연산(Per-Sample Operations)
- 스텐실 테스트, Z 테스트도 이 지점에서 수행됨

컴퓨트 셰이더(Compute Shader)

- 비그래픽 작업에 사용하기 위해 별도로 존재하는 실행 모드
- 워프(Warp)와 스레드(Thread)의 개념을 더 직접적으로 접할 수 있음