GRAPHICS PIPELINE

참조: OpenGL Rendering Pipeline Overview



OpenGL 파이프라인은 일단 시작되면 다음 순서로 작동합니다. 세부적인 사항은 더 밑에 정리하였습니다.

1. Vertex processing

1. VAO( vertex array object)에 정의된 vertex array에서 찾아진 각 vertex는 vertex shader에 의해 작동됩니다. Stream의 각 vertex는 차례로 output vertex로 처리됩니다. (Queue형식)
2. Optional한 tessellation 단계
3. Optional한 Geometry shader 단계 : 출력은 a sequence of primitives 입니다.

2. Vertex Post Processing: 마지막 단계의 출력이 조정되거나 다른 위치로 옮겨집니다.

1. Transformation 처리는 여기서 합니다. (obj- world – camera – clip)
2. Primitive 도형을 Assemble 합니다. (버퍼에 저장되었던 삼각형 정보들을 그리는 것을 의미하는 것으로 보입니다.)
3. Clipping, perspective divide , viewport transform to window space를 여기서 진행합니다. 이 곳에서 하는 일들은 순서대로 안보이는 부분을 자른다거나, 원근법을 구현한다거나, 3D 세계를 우리가 보는 window로 옮겨줍니다.

3. Scan conversion and primitive parameter interpolation , generating Fragment. : 이 단계는 rasterizer 하드웨어가 하는 단계입니다. 우리가 pixel을 처리하기위한 fragment를 만드는 단계입니다.

4. fragment shader: 각각의 만들어 놓은 fragment를 처리하는 fragment shader 단계입니다. 각 fragment는 여러 출력을 생성합니다.

5. Per Sample Processing

1. [Scissor Test](https://www.khronos.org/opengl/wiki/Scissor_Test)
2. [Stencil Test](https://www.khronos.org/opengl/wiki/Stencil_Test)
3. [Depth Test](https://www.khronos.org/opengl/wiki/Depth_Test)
4. [Blending](https://www.khronos.org/opengl/wiki/Blending)
5. [Logical Operation](https://www.khronos.org/opengl/wiki/Logical_Operation)
6. [Write Mask](https://www.khronos.org/opengl/wiki/Write_Mask)

Vertex Specification

파이프라인에 보낼 vertex 정보를 순서에 맞게 buffer에 저장하는 단계입니다. 이러한 vertex들은 primitive(삼각형 같은 기본 선,점 모양)의 경계를 정의합니다. 이 vertex array가 도형으로 어떻게 해석되는지는 추후에 다뤄집니다.

이 단계에서는 VAO 라고 vertex array object와 vertex buffer object를 만들어서 많은 vertex관련 object들을 관리합니다. vertex array object에서는 각 vertex의 데이터를 정의하는 반면 vertex buffer object는 vertex 데이터 자체를 저장합니다.

Vertex’s data는 일련의 attribute라고 칭할 수 있습니다. 보통 position, normal , textcoord 등이 이에 속합니다. 보통 이후 나오는 shader의 inputs으로 사용됩니다. 하지만 이 또한 정해진 것은 없고 임의적입니다. 정확히 attribute가 가지는 유일한 의미는 다음 vertex processing 단계에서 처리된다는 점입니다.

Vertex rendering : vertex 데이터가 제대로 지정되면 draw call 명령을 통해 기본 데이터로 rendering 됩니다.

Vertex Processing

이전 정점 렌더링 단계로 가져온 vertex는 여기에서 처리를 시작합니다. Vertex 처리 단계는 programming 가능한 단계입니다. User code는 vertex가 처리되는 방식을 사용자가 정의할 수 있습니다. 이후 나올 단계들은 다른 종류의 셰이더 작업을 나타냅니다. 이 중 대부분은 선택 사항입니다.

Vertex Shader(essential) : 이 단계에서는 사진과 같이 사용자 지정 입력과 사용자 지정 아웃풋이 존재하는데 보통 렌더링을 위한 clip space에 있는 location이 아웃풋으로 정해질 것이라고 예상됩니다.

Vertex shader는 한 가지 제한 사항이 있습니다. 각각의 input vertex는 반드시 특정 output에 mapping(사상) 되어야합니다. 왜냐하면 Vertex shader의 호출이 attribute들 간의 상태를 공유할 수 없기 때문에 1:1 대응이여야 합니다.

Tessellation(optional) : 이 단계는 optional이며 사전에서 찾아본 바로는 도형을 빈틈없이 겹치는 것없이 채우는 역할을 한다고 합니다. openGL에서 tesellation은 patch라고 부르는 고차 surface를 점, 선, 삼각형 등으로 분할하는 일을 수행합니다. 각 patch는 여러 제어점(control point)으로 만듭니다. patch 당 제어점 개수는변경이 가능합니다. 자세한 사항은 밑 링크를 참조하면 됩니다.

<https://niklasjang.tistory.com/44>

Geometry Shader(optional): Geometry shader는 입력으로 예를 들어 점이나 삼각형같은 하나의 기본 도형을 이루는 vertex들의 모음을 받습니다. 이 geometry shader는 이 vertex들을 다음 shader 단계에 이들을 보내기 전에 적절한 형태로 변환시킬 수 있습니다. 하지만 geometry shader가 만드는 흥미로운 점은 vertex들을 원래 주어진 vertex들보다 더 많은 vertex들을 생성하는 완전히 다른 기본 타입 도형으로 변환시킬 수 있다는 점입니다. 자세한 사항은 밑 링크를 참조하면 됩니다.

<https://heinleinsgame.tistory.com/34>

Vertex Post-Processing

**Transform Feedback** : primitive assembly 나 geometry shader의 output은 일련의 buffer object에 쓰여집니다. 이거는 transform feedback mode라고 불리웁니다. 이를 통해 사용자는 데이터를 변환한 다음 나중에 사용하기위해 buffer에 저장해 보유할 수 있습니다. 버퍼로 출력되는 데이터는 이 단계에서 내보낸 각 primitive의 데이터입니다.

**Primitive Assembly :** 실제 기본 요소인 선,점 또는 삼각형들의 순서화된 시퀀스를 출력으로 합니다. 실제 primitive sequence로 구성하는 단계입니다. 예를들어 입력이 12개의 꼭짖점을 포함하는 삼각형 primitive인 경우 이 process의 출력은 10개의 삼각형이 됩니다.

**Clipping :** 우리가 transform을 통해 clip space에 도달했다면 이제 어떤 물체는 카메라에 잘려서 보이게 될 것입니다. 그렇기에 잘라주는 작업이 필요합니다. 이 작업은 카메라 시야 경계선에 걸리는 object의 부분을 새로운 vertex로 정의함으로써 해결할 수 있습니다.

**Face culling :** 우리에게는 물체의 뒷면이 안보이기에 필요 없는 부분 일지도 모릅니다. Face culling은 렌더링 없이 삼각형의 방향에 따라 버려질 수 있습니다. 뷰어에서 반대쪽을 향하는 삼각형이 렌더링 되는 것을 피함으로 물체의 안보이는 부분을 처리할 수 있습니다. Face culling은 이러한 primitive rendering을 피하는 방법입니다.

Vertex position은 clip space에서 window space로 Perspective division 과 Viewport Transform을 통해 옮겨집니다.

Rasterization : 이 stage 까지 도착한 primitive는 rasterize 됩니다. 이 동작이 진행된 상태 집합을 fragment라고 부릅니다. Fragment는 출력 프레임 버퍼에서 픽셀에 대한 최종 데이터를 계산하는 데 사용됩니다. Fragment의 상태에는 화면공간에서의 위치, 다중 샘플링이 활성화된 경우에는 샘플 적용 범위, 이전 vertex 또는 geometry shader에서 출력된 임의 데이터 목록이 포함됩니다.

이 데이터들은 vertex 사이의 interpolation을 통해 만들어집니다. Interpolation style은 해당 값을 출력한 셰이더에 의해 정의됩니다.

Fragment Shader : Rasterize된 Fragment들은 Fragment shader에서 처리됩니다. 각 색상이 결정되며 depth value, stencil value 등이 결정됩니다. Fragment shader는 fragment 에 대한 스텐실 데이터를 설정할 수 없지만 색상 및 깊이 값을 제어할 수는 있습니다.

프래그먼트 셰이더는 선택 사항입니다. 프래그먼트 셰이더 없이 렌더링하면 프래그먼트의 깊이(및 스텐실) 값이 일반적인 값을 얻습니다. 그러나 조각이 가질 수 있는 모든 색상의 값은 정의되지 않습니다.

Lighting은 Per vertex operation 단계에서 하나 Per fragment operation 단계에서 하나 관계는 없으나 보통 fragment shader 단계에서 색을 정하는 과정에서 같이 진행됩니다. 이 때 사용하는 빛을 측량하는 방식은 Phong model을 일반적으로 사용합니다.

Per-Sample Operations : 프래그먼트 프로세서에서 출력된 프래그먼트 데이터는 일련의 단계를 통해 전달됩니다.

첫 번째 단계는 컬링 테스트의 시퀀스입니다. 테스트가 활성이고 프래그먼트가 테스트에 실패하면 기본 픽셀/샘플이 업데이트되지 않습니다(일반적으로). 이러한 테스트 중 대부분은 사용자가 활성화한 경우에만 활성화됩니다. 테스트는 다음과 같습니다.

* Pixel ownership test: 조각의 픽셀이 OpenGL에 의해 "소유"되지 않은 경우(다른 창이 GL 창과 겹치는 경우) 실패합니다. [Framebuffer Object](https://www.khronos.org/opengl/wiki/Framebuffer_Object) 를 사용할 때 항상 통과합니다 . 실패는 픽셀에 정의되지 않은 값이 포함되어 있음을 의미합니다.
* Scissor test : 활성화되면 조각의 픽셀이 화면의 지정된 사각형 밖에 있으면 테스트가 실패합니다.
* [stencil](https://www.khronos.org/opengl/wiki/Stencil_Test) test : Stencil buffer는 (일반적으로) 8 비트의 stencil value를 가지고 있고 이 값은 pixel/fragment마다 256개의 값으로 나타내어집니다. 우리는 이 stencil 값을 설정하여 특정한 stencil 값을 가지고 있는 특정 fragment를 폐기할지 유지할지를 정할 수 있습니다. 활성화되면 테스트에서 제공한 스텐실 값이 스텐실 버퍼에 있는 기본 샘플의 스텐실 값과 사용자가 지정한 대로 비교되지 않으면 테스트가 실패합니다. 프레임 버퍼의 스텐실 값은 스텐실 테스트가 실패하더라도(심도 테스트가 실패하더라도) 계속 수정할 수 있습니다.
* [깊이 테스트](https://www.khronos.org/opengl/wiki/Depth_Test) : 활성화되면 사용자가 깊이 버퍼에 있는 기본 샘플의 깊이 값에 대해 지정한 대로 조각의 깊이가 비교되지 않으면 테스트가 실패합니다.

**참고: 이러한 작업은**[조각 셰이더](https://www.khronos.org/opengl/wiki/Fragment_Shader) 이후에 발생하도록 지정되었지만 특정 조건 [에서는 조각 셰이더 이전](https://www.khronos.org/opengl/wiki/Early_Fragment_Test) 에 발생하도록 만들 수 있습니다 . FS 이전에 발생하면 프래그먼트 컬링으로 인해 프래그먼트 셰이더가 실행되지 않아 성능이 절약됩니다.

마지막으로 프래그먼트 데이터가 프레임 버퍼에 기록됨으로써 pipeline은 끝이 나게 됩니다.