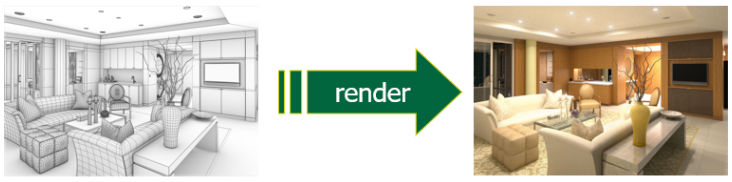
2. Basic Concepts

💡 **What is rendering?**
Graphics rendering pipeline
Basic rendering algorithms
Displays

Rendering

: **3D Scene → 2D Image** 로 변환하는 과정



Scene

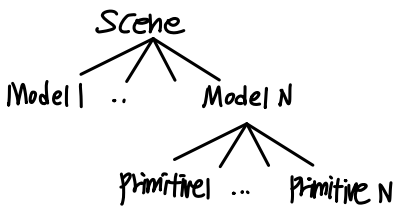
: **3D Scene** 에 존재하는 여러 **Model** 로 구성

Model

: 3D Scene 에서 객체, **Primitive** 라는 작은 도형들로 구성

Primitive

: 삼각형 같은 **기본 도형**으로, 3D Model의 가장 작은 구성 요소



Geometric Primitive

💡 Model은 여러 Geometric Primitive로 이루어져 있음

- **GPU로 바로 render 가능한 primitive**
  - 점
  - 선
  - 면 (주로, 삼각형)
- **GPU로 바로 render 불가능한 primitive**
  - (Explicit) Curves/surfaces
  - Implicit Surfaces: 방정식으로 정의된 표면들 (예: f(x, y, z)=0)
  - Voxels

💡 그럼 어떻게 render할까? **점, 선, 면**으로 근사하여 render

💡
- **Point = 점 = 0D**, 크기 없음 (위치만 가짐)
- **Pixel = Picture element = 2D**, 크기 있음
- **Voxel = Volume element = 3D**, 부피 있음

Graphics Rendering Pipeline

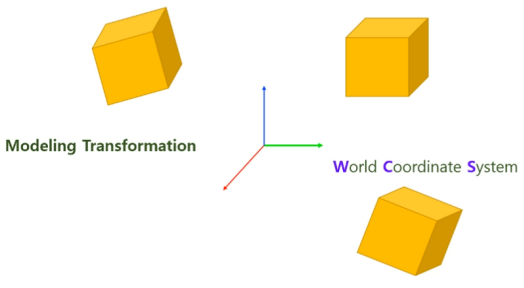
- **3D Scene → 2D Image** 일련의 과정
- **Model → Scene → Image**
  1. **Model → Scene**:
     - Model 의 primitives 를 3D 공간 내에 배치하여 Scene 을 구성
     - **Modeling Transformation**
  2. **Scene → Image**:
     - Scene 을 2D Image 로 변환하는 과정
     - **Viewing Transformation, Projection Transformation, ...**
     - 마지막으로 Shading 을 통해 빛과 재질을 계산하여 현실감 있게 표현
- **Transformation** 반복 + 마지막에 **Shading**

💡 이 pipeline 은 GPU 에 구현되어있음
pipeline 이 어떻게 작동하는지 몰라도 OpenGL 로 작동 가능함
그러나 우리는 작동 방법도 배움

💡 **Coordinate System**: 3D 공간에서 객체의 **위치, 크기, 방향**을 정의

1️⃣ Modeling Coordinate System

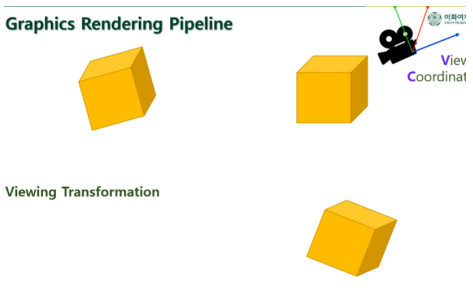
- 개별 Model 의 로컬 좌표계



Modeling Transformation

- 2️⃣ **World Coordinate System** 로 좌표계 변환
- 이를 통해 Model 이 전체 Scene 에서 적절히 배치됨

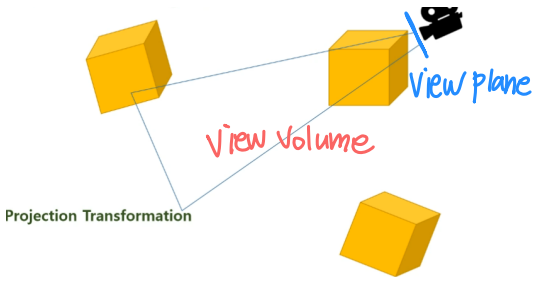
💡 비로소 Scene 이 결정됨
모든 Model 이 하나의 wcs 에 존재

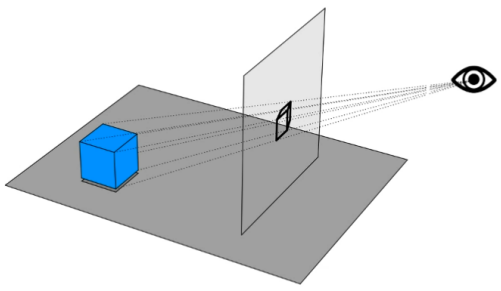


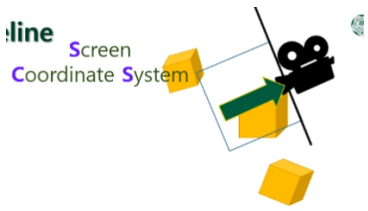
Viewing Transformation

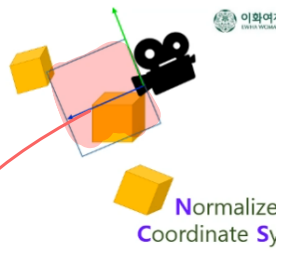
- 3️⃣ **Viewing Coordinate System** 로 좌표계 변환
- 카메라의 위치와 방향에 맞춰 Scene 을 보는 관점을 설정

💡 카메라 위치 결정
카메라 위치 따라 Model 좌표 변환됨









Projection Transformation

- 3️⃣ **Viewing Coordinate System** 에서 Projection
- 4️⃣ **Normalized Coordinate System** 로 좌표계 변환

cf) **Perspective Projection**

- 원근감 (가까운 건 크게, 멀리 있는 건 작게)

💡 다른 종류의 projection 도 있음 나중에 배움

Viewport Transformation

- 5️⃣ **Screen Coordinate System** 로 좌표계 변환
- 화면의 픽셀 좌표에 맞게 크기와 위치를 조정하여 최종 Image 생성

💡
1. x, y, z 중에 z를 날려서 3D 를 2D 로
2. 최종 Display screen에 맞게 scaling
   (각 Display 마다 해상도가 다르기 때문에)

💡 2D 로 만들기 위한 **준비**

3D 의 물체를 카메라 앞에 놓여있는 **View Plane** 쪽으로 Projection 하기 위한 Transformation

View Volume (VV) 가 2x2×2 정육면체가 됨 ⇒ 그래서 'Normalized' (정규화) 된 좌표계라고 부름

💡 해당 Coordinate System 에 맞게 각 Transformation으로 좌표체계 바꿔주는 것
- 각 Transformation 은 4x4 행렬 (나중에 배움)
- 아직 Rendering pipeline 안 끝남. Shading 등의 작업 남음

Coordinate System	Transformation	정리	설명
Modeling Coordinate System			각 Model 마다 다름, 여러 개 존재
World Coordinate System	**Modeling Transformation**	모델 위치 결정	하나의 Scene안에 있는 모든 Model이 **하나의 wcs**에 위치함
Viewing Coordinate System	**Viewing Transformation**	카메라 위치 결정	카메라 위치가 vcs의 원점 & xyz축 결정
Normalized Coordinate System	**Projection Transformation**	3D → 2D 준비	1) View volume 결정 2) Projection type 결정
Screen Coordinate System	**Viewport Transformation**	3D → 2D	1) z 좌표값 날리기 2) Display screen 사이즈에 맞게 scaling

💡 **(1) View volume 결정**

: clipping (vv 밖을 날림) ⇒ vv가 2x2×2 정육면체가 됨

(2) Projection type 결정

: z 좌표 날리기 전 x, y 값 조정을 어떻게 할지 결정하는 것

a. parallel → orthographic, obligue

b. perspective

Basic Rendering Algorithms

위의 표 말고 Rendering pipeline 에는 어떤 것들이 더 있는가!

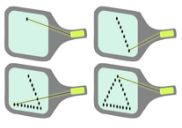
- **Transformation**
  - 위의 표에서 봤던 행렬 연산들
- **Clipping/Visible Surface determination**
  - Clipping = 화면에 보이지 않는 부분(viewing volume 밖)을 잘라내어 처리 효율을 높임
  - Visible Surface determination = vv 안에 있더라도, 여러 객체가 겹칠 때 카메라에서 보이는 표면만을 결정하여 렌더링
- **Rasterization**
  - Viewport Transformation 된 화면 공간의 Primitive 를 **Pixel 로 변환**. 그 과정을 Rasterization 알고리즘을 통해 수행
  - 각 pixel 에 해당하는 **색상과 깊이** 값을 계산
    - **Shading & Illumination :** 각 pixel 의 color 결정
      - Shading = 음영 처리 = 빛을 향하는 부분은 밝고 아닌 부분은 어둡게
      - 반사, 굴절, color bleeding 등도 고려

Displays

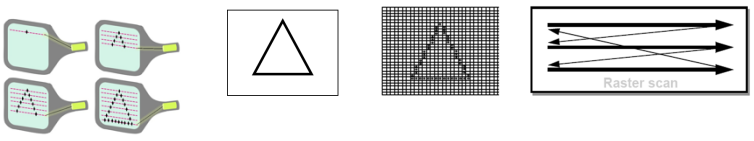
💡 화면에 보여질 최종 이미지를 생성

Display 장비에 따라서 Display 하는 방식이 다름!

1. **Vector displays**: (초기 디스플레이) 선의 끝점을 기반으로 전자빔을 무작위 경로로 이동시키는 방식, 이 과정을 Vector Scan
   - Primitive 가 너무 많은데 그걸 다 따라가기가 힘들어서 요새는 사용 안 함



2. **Raster displays**: (현대 디스플레이) 전자빔이 화면을 규칙적인 패턴으로 스캔하여 이미지를 생성, Raster Scan
   - 화면을 수평선으로 나누고, 각 라인을 위에서 아래로 스캔하여 픽셀을 채우는 방식
   - **Rasterization 이 선행되고 디스플레이**
     - Rasterization 된 Pixel 이 저장 되어있는 곳 = **Frame Buffer**



Frame Buffers

- **2D array** of pixels (화면에 표시될 최종 이미지 구성)
  - 각 행은 **Scan line**, **Raster line** 이라 부름 (화면의 한 줄을 나타냄)
  - 주로 **GPU**와 같은 그래픽 하드웨어 내부에 존재

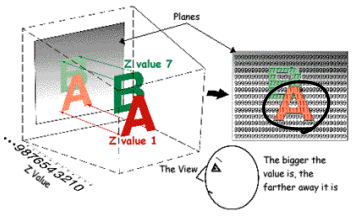
- 각 픽셀의 **색상 저장** - 각 픽셀은 빨강(R), 초록(G), 파랑(B) 으로 구성
  - (예전) RGB 각각 **8 bits (1 byte)**
    - 각 색상 채널당 **8 bits (0~255 사이 integer)**
    - 총 2^8 x 2^8 × 2^8 = 2^24 개의 색 표현
  - (현대 GPU의 지원): RGB 각각 **32 bits (4 byte)**
    - 각 색상 채널당 **32 bits** (**0~1 사이 floating point**)

- 각 픽셀의 **Alpha Channel**, **A, 투명도**
  - 0은 투명, 1은 불투명
  - 32 bits (0~1 사이 floating point)
  - 각 픽셀에 대한 추가 채널, **RGBA**

$$Out_{RGB} = Current_{RGB} * Current_{A} + Existing_{RGB} * (1 - Current_{A})$$

새로 그려는 것의 RGB　　이미 그려든 것의 RGB

- 각 픽셀의 **Z-buffer, depth buffer**, **Z, D**
  - 각 픽셀에 대한 **깊이** 값을 저장하는 버퍼
  - ~~0은 카메라에 가장 가까움, 1은 가장 멂~~ 숫자가 작을수록 가까움, 클수록 멀다.
  - 32 bits (실제 깊이 범위는 구현 방식에 따라 달라짐)
  - **Visible Surface Determination**



7보다 1이 앞에 있으니까
기존에 있는 Z-buffer 값보다 작을 때에만 렌더

- **Double buffering**
  - 깜빡임(flicking) 없는 **실시간 애니메이션**을 위해 두 개의 **전체 프레임 버퍼** 사용 (그리는 과정은 눈에 안 보이게 하기 위함)
    - **front buffer**: 사용자에게 보이는 프레임 버퍼
    - **back buffer**: 보이지 않는 프레임 버퍼로, 이곳에서 먼저 **그린 다음 swapping**

> 💡 정리하자면...
> **Rasterization** 을 마친 후, 각 픽셀마다 [**RGBA D**] 를 **Frame buffer** 에 저장 → Raster line 따라 display 하는 방법을 **Raster display**
> 똑같은 Frame buffer 두 개를 이용해 (front buffer, back buffer) 그리는 과정은 눈에 안 보이게 하는 것을 **Double buffering**

문제 1

Order the transformation steps correctly:

하나를 선택하세요.

- a. Modeling -> Viewing -> Viewport -> Projection
- b. Modeling -> Viewing -> Projection -> Viewport ✔
- c. Viewing -> Modeling -> Viewport -> Projection
- d. Viewing -> Viewport -> Modeling -> Projection

정답 : Modeling -> Viewing -> Projection -> Viewport

문제 2

What do you call the process of "converting a projected screen primitive to a set of pixels"?

하나를 선택하세요.

- a. Shading
- b. Clipping
- c. Rasterization ✔
- d. Transformation

정답 : Rasterization

문제 3

Find which of the following primitives can not be directly rendered by GPU.

하나를 선택하세요.

- a. Voxels ✔
- b. Line segments
- c. Points
- d. Polygons

정답 : Voxels