<https://heinleinsgame.tistory.com/11>

[Learn OpenGL 번역] 2-8. 시작하기 - 좌표 시스템

하인라인 2018. 7. 15. 19:57

**좌표 시스템**

  마지막 강좌에서 변환 행렬을 사용하여 모든 vertex들을 변환함으로써 수학을 잘 활용하는 방법을 배웠습니다. OpenGL읜 각 vertext shader 실행 된 후 우리가 그리기 원하는 모든 vertex들이 정규화된 디바이스 좌표로 표시되기를 원합니다. 즉, 각 vertex의 x, y, z 좌표가 -1.0 ~ 1.0 범위 안에 있어야 합니다. 이 범위 밖의 좌표는 보이지 않게됩니다. 일반적으로 우리가 직접 구성한 범위의 좌표를 지정하고 vertex shader에서 이 좌표들을 NDC로 변환합니다. 그런 다음 이 NDC 좌표들은 rasterizer에게 보내지고 화면위의 2D 좌표/픽셀로 변홥됩니다.

  NDC로 좌표를 변환한 다음 스크린 좌표로 변환하는 것은 일반적으로 최종 스크린 좌표로 변환하기 전에 오브젝트의 vertex를 여러 좌표 시스템으로 변환하는 단계적인 방법으로 수행 가능합니다. 이들을 여러 *중간(intermediate)* 좌표 시스템으로 변환하면 일부 연산/계산들이 특정 좌표 시스템에서 수행하기 쉬워진다는 장점이 있습니다. 이는 곧 알게 될것입니다. 우리에게 중요한 좌표 시스템에는 총 5가지가 있습니다.

* Local space (또는 Object space)
* World space
* View space (또는 Eye space)
* Clip space
* Screen space

  이 것들은 vertex들이 변환되어 결국 fragment로 되기전에는 모두 다른 상태입니다.

  지금 여러분은 공간이나 좌표 시스템이 실제로 무엇을 의미하는지 혼란스러울 것입니다. 그래서 전체적인 그림과 각 공간들이 실제로 무엇을 수행하는지 보여줌으로써 좀 더 이해하기 쉬운 방법으로 설명해드리겠습니다.

**전체 그림**

  하나의 공간에서 다음 좌표 공간으로 좌표를 변환하기 위해 여러가지 변환 행렬을 사용하는데 이 중 가장 중요한 것들은 model, view, projection 행렬입니다. 버텍스 좌표는 먼저 local 좌표를 사용하여 local 공간에서 시작하고 좀 처리되어 world 좌표, view 좌표, clip 좌표 마지막으로 screen 좌표로 변환됩니다. 다음 이미지는 이 과정을 보여주고 각 변환이 무엇을 수행하는지 보여줍니다.

A screenshot of a computer

Description automatically generated

1. Local 좌표는 오브젝트의 원점을 기준으로한 좌표입니다. 오브젝트의 처음 좌표입니다.
2. 다음 단계는 local 좌표를 좀 더 큰 world에 대한 좌표인 world-space 좌표로 변환하는 것입니다. 이 좌표는 전체적인 world의 원점을 기준으로 하는 좌표이고 world의 원점을 기준으로 하여 많은 다른 오브젝트들을 배치할 수 있습니다.
3. 그 다음 world 좌표를 각 좌표들이 카메라나 보는 사람의 시점에서 보이는 view-spcae 좌표로 변환합니다.
4. 좌표들이 view space로 변환된 후 좌표들을 clip 좌표로 투영합니다. Clip 좌표는 -1.0 ~ 1.0 범위로 처리되고 vertex들이 화면위에 그려지게 될 것인지 결정합니다.
5. 그리고 마지막으로 viewport 변환이라고 불리는 작업(-1.0 ~ 1.0 범위의 좌표들을 glViewport 함수를 통해 정의된 좌표 범위로 변환)을 통해 clip 좌표를 screen 좌표로 변환합니다. 그런 다음 결과로 나온 좌표들은 rasterizer로 보내져 fragment로 변환됩니다.

  각각의 공간들이 무엇을 위해 쓰이는지 약간은 이해했을 것입니다. 우리가 vertex들을 이 모든 다른 공간들로 변환하는 이유는 일부 연산들이 특정 좌표 시스템에서 좀 더 사용하기 쉬워지기 때문입니다. 예를 들어 하나의 오브젝트를 수정할 때 local space에서 작업하는 것이 가장 직관적입니다. 반면에 다른 오브젝트의 위치에 대해서 하나의 오브젝트의 특정 연산을 수행하는 것은 world 좌표에서 가장 직관적입니다. 원한다면 local space에서 clip space로 한번에 가는 변환 행렬을 정의할 수 있습니다. 하지만 이는 유연성을 저하시킵니다.

  각 좌표 시스템에 대해 좀 더 상세히 설명드리겠습니다.

**Local space**

  Local space는 여러분의 오브젝트에 대한 좌표 공간입니다. 즉, 오브젝트가 시작하는 공간입니다. 모델링 소프트웨어(Blender 같은)에서 정육면체를 생성한 것을 상상해보세요. 최종 응용 프로그램에서 다른 위치로 바뀔것임에도 불구하고 이 정육면체의 원점은 아마 (0,0,0)일것입니다. 아마도 여러분이 생성한 모든 모델들이 그들의 초기 위치로 (0,0,0)을 가질것입니다. 그러므로 모델의 모든 vertex들은 *local* space에 있습니다. 이 것들은 오브젝트를 기준으로 합니다.

  우리가 사용했던 컨테이너의 vertex들은 0.0을 원점으로 -0.5와 0.5 사이의 좌표들로 지정했습니다. 이 것들이 local 좌표입니다.

**World space**

  만약 우리의 모든 오브젝트들을 응용 프로그램에 직접 넣는다면 오브젝트들은 아마 모두 원점이 (0,0,0)인 world를 중심으로 어딘가에 서로 쌓일 것입니다. 이 것은 우리가 원하는 것이 아닙니다. 우리는 각 오브젝트들을 더 큰 world에 배치하기 위해 각 오브젝트에 대한 위치를 정의하고 싶습니다. world space의 좌표는 정확히 이 것입니다. 모든 vertex들의 좌표는 (game) world를 기준으로 합니다. 이 것은 여러분이 원하는(오브젝트들이 여러 곳에 산재되어 배치되는 방법으로 변환)(가급적이면 현실적인 방법) 좌표 공간입니다. 여러분의 오브젝트의 좌표는 local space에서 world space로 변환되었습니다. 이는 model 행렬을 사용하여 수행됩니다.

  model 행렬은 오브젝트를 그들이 속한 위치/방향으로 world에 배치하기 위해 오브젝트를 이동, 스케일, 회전하는 변환 행렬입니다. 변환을 통해 축소된 집(local space에서는 약간 컷었습니다)을 생각해보세요. 이 집을 교외로 이동시키고 y 축을 중심으로 왼쪽으로 약간 회전시켜서 옆집과 깔끔하게 정렬됐다고 가정해봅시다. 컨테이너를 scene 전체에 위치시키기 위해 앞 강좌에서의 행렬을 일종의 model 행렬로 생각할 수 있습니다. 우리는 컨테이너의 local 좌표를 scene/world의 다른 곳으로 변환했습니다.

**View space**

  View space는 사람들이 일반적으로 OpenGL에서 카메라를 나타내는 것입니다(때때로 camera space나 eye space라고도 불립니다). view space는 world-space 좌표를 유저의 시점 앞에 있는 좌표로 변환했을 때의 결과입니다. 따라서 view space는 카메라의 관점에서 바라보는 공간입니다. 일반적으로 scene 이동/회전시키기 위해 이동, 회전의 조합을 사용하여 수행되어 특정 아이템이 카메라 앞으로 변환됩니다. 이 조합된 변환들은 일반적으로 view matrix에 저장되고 view 행렬은 world 좌표를 view 공간으로 변환합니다. 다음 강좌에서 카메라를 시뮬레이션하여 이러한 view 행렬을 좀 더 광범위하게 다루어 볼 것입니다.

**Clip space**

  각 vertex shader 실행의 마지막에 OpenGL은 지정된 범위의 좌표를 받아들이고 이 범위에서 벗어난 모든 좌표는 clipped(자르다)됩니다. clip된 좌표들은 폐기되고 남은 좌표들은 최종적으로 fragment가 되어 화면에 보이게 됩니다. clip space에서 이름을 가져오는 위치이기도 합니다.

  눈에 보이는 모든 좌표들이 -1.0와 1.0 범위 안으로 지정하는 것은 직관적이지 않기 때문에 우리만의 좌표를 지정하고 작업을 수행한 후 그들을 다시 OpenGL이 원하는 NDC로 변환합니다.

  vertex 좌표를 view에서 clip-space로 변환하기 위해 우리는 좌표의 범위를 지정(예를들어 각 축에 대해 -1000에서 1000까지)하는 project matrix라고 불리는 것을 정의합니다. 그런 다음 projection 행렬은 이 지정된 범위에 잇는 좌표들을 NDC(-1.0, 1.0)로 변환합니다. 지정된 범위 밖에 있는 좌표들은 -1.0 ~ 1.0에 매핑되지 않으므로 clip됩니다. projection 행렬에서 지정한 이 범위를 사용한다면 좌표 (1250, 500, 750)은 보이지 않게됩니다. x 좌표가 범위밖에 있어 NDC에서 1.0보다 높아지므로 clip되기 때문입니다.

  삼각형과 같은 primitive의 일부만 clipping volume에서 벗어났다면 OpenGL은 clipping 범위에 맞도록 하나 이상의 삼각형을 재구성합니다.

  projection 행렬이 생성하는 *viewing box*는 절두체(frustum)라고 불리고 이 절두체 내부에 있는 각 좌표들은 유저의 화면에 나타나게됩니다. 지정된 범위에서 NDC(2D view-space 좌표로 쉽게 매핑가능함)로 변환하는 전체적인 과정은 projection(투영)이라고도 불립니다. projection 행렬이 3D 좌표를 2D에 매핑하기 쉬운 NDC로 투영하기 때문이죠.

  모든 vertex들이 clip space로 변환되었다면 perspective division이라고 불리는 마지막 작업이 수행됩니다. 여기에서 위치 벡터의 x, y, z 요소들을 벡터의 w 요소로 나눕니다. perspective division은 4D clip space 좌표를 3D NDC로 변환하는 것입니다. 이 단계는 vertex shader의 실행 마지막에 자동으로 수행됩니다.

이 단계 이후에는 결과 좌표들을 screen 좌표에 매핑(glViewport 함수의 설정을 사용하여)하고 fragment로 변환하는 것입니다.

  projection 행렬은 view 좌표를 clip 좌표로 변환하기 위해 두개의 다른 형식을 받습니다. 각 형식은 자신만의 고유한 절도체를 가집니다. 우리는 정사영(orthographic) projection 행렬과 원근(perspective) projection 행렬 중에서 하나를 생성할 수 있습니다.

**Orthographic projection**

  orthographic projection 행렬은 정육면체와 같은 절도체 상자를 정의합니다. 이 절도체 상자는 이 상자 밖에 있는 vertex들을 clip하는 clipping 공간을 정의합니다. orthographic projection 행렬을 생성할 때 눈에 보이는 절도체의 너비, 높이, 길이를 정의합니다. orthographic projection 행렬로 변환이 완료된 후에 이 절도체 안에 있는 모든 좌표들은 clip되지 않습니다. 이 절도체는 컨테이너와 비슷하게 생겼습니다.

A close up of text on a white background

Description automatically generated

  이 절도체는 보이게 될 좌표들을 결정하고 너비, 높이, 가까운(near) 평면, 먼(far) 평면으로 이루어집니다. 가까운 평면의 앞에있는 모든 좌표들은 clip되고 먼 평면의 뒤에 있는 좌표들도 마찬가지입니다. orthographic 절도체는 절도체 내부에 있는 모든 좌표들을 NDC로 직접 매핑합니다. 각 벡터의 w 요소를 건드리지 않기때문입니다. 만약 w 요소가 1.0이라면 perspective division은 좌표를 수정하지 않습니다.

orthographic projection 행렬을 생성하기 위해 GLM의 glm::ortho 함수를 사용합니다.

glm::ortho(0.0f, 800.0f, 0.0f, 600.0f, 0.1f, 100.0f)

  처음 두 개의 파라미터는 절두체의 왼쪽 오른쪽 좌표를 지정합니다. 그리고 세, 네 번째 파라미터는 절두체의 맨 밑과 맨 위의 좌표를 지정합니다. 이 4개의 지점과 함께 가까운 평면과 먼 평면의 크기를 다섯, 여섯 번째 파라미터로 정의할 수 있습니다. 그런 다음 가까운 평면과 먼 평면 사이의 거리를 정의합니다. 지정된 projection 행렬은 x, y, z 범위 값을 가진 모든 좌표들을 NDC로 변환합니다.

  orthographic projection 행렬은 좌표들을 화면의 2D 평면에 똑바로 매핑하지만 실제로 똑바로 투영하는 것은 비현실적인 결과를 생성합니다. 원근감을 고려하지 않았기 때문입니다. 이는 perspective project 행렬이 해결해줄 것입니다.

**Perspective projection**

*실제 세상*을 그래픽으로 구현하고 싶다면 멀리있는 오브젝트는 작아져야 한다는 것을 말씀드려야 할 것 같습니다. 이 이상한 효과는 perspective(원근감)이라고 불립니다. Perspective 무한의 고속도로나 철도를 바라보고 있을 때 알 수 있을 것입니다.

A screenshot of a cell phone

Description automatically generated

  보시다시피 원근법때문에 선이 멀어질 수록 서로 만나고 있습니다. 이 것이 정확히 perspective projection을 했을 때 나타나는 효과이고 perspective projection 행렬을 사용하여 수행할 수 있습니다. projection 행렬은 주어진 절도체를 clip된 공간에 매핑할 뿐만 아니라 각 vertex 좌표의 w 값을 조작합니다. 시점으로부터 vertex 좌표가 멀어질수록 이 w 요소가 증가합니다. 좌표들이 clip space로 변환되고 나면 그들은 -w ~ w 범위 안에 있게 됩니다(이 범위 밖에 잇는 모든 것들은 clip됩닌다). OpenGL은 vertex shader의 최종 출력으로 -1.0 ~ 1.0 범위 안에있는 좌표들만 보이도록 합니다. 따라서 좌표들이 clip space에 있으면 perspective division은 clip space 좌표에 적용됩니다.  
A picture containing object, clock, meter

Description automatically generated

  vertex 좌표의 각 요소들은 w 요소로 나누어집니다. 시점에서 멀리 떨어진 vertex에게 작은 vertex 좌표를 주어줍니다. 이 것이 w 요소가 중요한 또다른 이유입니다. 그것은 우리가 persprective projection을 할 수 있도록 돕기 때문입니다. 그런 다음 결과 좌표들은 NDC에 있게 됩니다. orthographic projection과 perspective projection 행렬이 실제로 어떻게 계산되는가에 대해서 흥미가 있다면 Songho의 [this excellent article](http://www.songho.ca/opengl/gl_projectionmatrix.html" \t "_blank)을 추천드립니다.

perspective projection 행렬은 GLM에서 다음과 같이 생성할 수 있습니다.

glm::mat4 proj = glm::perspective(glm::radians(45.0f), (float)width/(float)height, 0.1f, 100.0f);

  glm::perspective 함수가 하는 일은 눈에 보이는 공간을 정의하는 큰 *절도체(frustum)*를 생성하는 것입니다. 절도체 밖에 있는 모든 것은 clip spade 내부에 있지 않으므로 결국엔 clip됩니다. perspective 절도체는 균일하지 않은 상자모양으로 시각화될 수 있습니다. 이 상자 내부에 있는 각 좌표들은 clip space에 매핑될 수 있습니다. perspective 절도체의 이미지는 다음과 같습니다.

A picture containing text, map

Description automatically generated

  첫 번째 파라미터는 fov 값을 지정합니다. fov는 field of view의 줄임말로서 view space가 얼마나 큰지를 설정합니다. 현실적인 시점을 위해서 일반적으로 45도로 설정되지만 둠-스타일 결과를 원한다면 좀 더 높은 값으로 설정할 수 있습니다. 두 번째 파라미터는 viewport의 너비를 높이로 나눔으로써 계산되는 화면 비율을 설정합니다. 세 번째, 네 번째 파라미터는 *가까운(near)* 평면과 *먼(far)* 평면의 거리를 설정합니다. 우리는 일반적으로 가까운 평면의 거리는 0.1f로 먼 평면의 거리는 100.0f로 설정합니다. 가까운 평면과 먼 평면 사이에 있고 절도체 내부에 있는 모든 vertex들은 렌더링 될 것입니다.

  perspective 행렬의 *가까운(near)* 평면의 값이 약간 큰 값(10.0f같은)으로 설정될때마다 OpenGL은 카메라와 가까운 모든 좌표들(0.0f ~ 10.0f)d을 모두 clip합니다. 이는 비디오게임에서 볼 수 있는 친숙한 비주얼을 나타냅니다. 그래서 만약 오브젝트에 가까이 다가간다면 오브젝트를 뚫고 볼 수 있게 됩니다.

  orthographic project을 사용할 때 vetex의 각 요소들은 똑바로 clip space에 매핑됩니다. 복잡한 perspective division(perspective division을 하긴하지만 w 요소가 조작되지 않습니다(1로 유지됩니다). 따라서 아무런 효과가 없습니다) 없이 말이죠. orthographic projection이 perspective projection을 사용하지 않기 때문에 멀리 떨어진 오브젝트들이 작게 보이지 않습니다. 이는 이상한 비주얼을 출력하게 됩니다. 이러한 이유때문에 orthographic projection은 주로 원근법에 의해 왜곡된 vertex들을 사용할 필요 없는 2D 렌더링과 일부 구조적이거나 공학적인 응용 프로그램에서 사용됩니다. *Blender*와 같은 3D 모델링을 위한 응용 프로그램은 때때로 모델링을 위해 orthographic projection을 사용합니다. 오브젝트를 각축에 대해 정밀하게 그릴 수 있기 때문입니다. 다음은 Blender에서 두 가지의 projection 방법을 비교한 것입니다.

A picture containing computer

Description automatically generated

  perspective projection에서는 멀리 떨어진 vertex들이 작아진 것을 볼 수 있고 orthographic projection 에서는 각 vertex들이 사용자와 같은 거리에 있음을 볼 수 있습니다.

**모든 것을 한 곳에 넣기**

  앞서 언급한 각 단계에 필요한 변환 행렬들을 생성합니다. model, view, projection 행렬이 있습니다. 그런 다음 vertex 좌표는 다음과 같이 clip 좌표로 변환됩니다.



  행렬 곱셈의 순서는 반대인 것을 생각하세요(행렬 곱은 오른쪽에서 왼쪽으로 읽어야한다는 것을 기억하세요). 그런 다음 결과 vertex는 vertex shader의 gl\_Position에 할당 되어야 합니다. 그런 다음 OpenGL은 perspective division과 clipping을 자동으로 수행할 것입니다.

**그리고 다음엔?**  
  vertex shader의 출력은 좌표들이 방금 변환행렬로 수행하여 만들어진 clip-space 내부에 있기를 요구합니다. OpenGL은 *normalized-device coordinates*로 변환하기 위해 *clip-space 좌표*에서 *perspective division*을 수행합니다. 그런 후에 OpenGL은 glViewPort 함수의 파라미터를 사용하여 NDC 좌표를 *screen 좌표*에 매핑합니다. screen 좌표에서는 각 좌표가 해당 화면(우리의 경우 800x600화면)의 지점에 매핑합니다. 이 과정을 *viewport 변환*이라고 합니다.

  이 주제는 이해하기 어려운 주제입니다. 각 공간이 무엇을 위해 사용되는지 아직 정확하게 이해하지 못했어도 걱정하지 마세요. 밑에서 우리가 실제로 이 좌표 공간들을 잘 사용하기 위한 충분한 예제를 볼 수 있을 것입니다.

**Going 3D**

  이제 우리는 3D 좌표를 2D 좌표로 변환한느 방법을 알게 되었습니다. 이제 오브젝트를 2D 평면이 아니라 현실 3D 오브젝트처럼 렌더링할 수 있습니다.

3D로 그리기 위해서 먼저 model 행렬을 생성해야 합니다. model 행렬은 vertex들을 world space로 변환하기 위한 이동, 스케일, 혹은 회전과 같은 *변환*들로 이루어져 있습니다. 평면을 x 축을 중심으로 조금 회전시켜 변환해봅시다. 그러면 바닥에 누워있는 것 처럼 보일 것입니다. 이 model 행렬은 다음과 같습니다.

glm::mat4 model;

model = glm::rotate(model, glm::radians(-55.0f), glm::vec3(1.0f, 0.0f, 0.0f));

  vertex 좌표를 이 model 행렬과 곱하면 vertex 좌표를 world 좌표로 변환할 수 있습니다. 우리 평면인 이제 바닥에 약간 누워있습니다. 따라서 평면은 글로벌 world에서 나타나진 것입니다.

  다음에 view 행렬을 생성해야 합니다. 우리는 좀 뒤로 가서 오브젝트가 보이도록 하고 싶습니다(world space에 있을 때는 우리는 원점(0,0,0)에 위치합니다). scene 주위를 이동하기 위해 다음을 생각해봅시다.

* 카메라를 뒤로 이동하는 것은 모든 scene을 앞으로 이동하는 것과 같습니다.

  이게 정확히 view 행렬이 하는 일입니다. scene 전체를 우리가 카메라를 이동시키기 원하는 방향과 반대로 움직입니다.  
  우리는 뒤로 움직이길 원하고 OpenGL은 오른손좌표계이기 때문에 z 축의 양의 방향으로 이동하여야 합니다. 우리는 이것을 scene 전체를 z 축의 음의 방향으로 이동시켜 수행할 수 있습니다. 이는 우리가 뒤로 움직이고 있다는 인상을 줍니다.

**오른손좌표계**

  관례상, OpenGL은 오른손좌표계를 사용합니다. 이는 기본적으로 각 축에 대한 양의 방향이 x 축에서는 오른쪽, y 축에서는 위쪽, z 축에서는 뒤쪽을 향하는 것을 말합니다. 여러문의 화면의 중앙에 3개의 축이 있고 z 축의 양의 방향이 화면을 통해 여러분의 앞으로 향한다고 생각해보세요. 이 축들은 다음과 같습니다.

A picture containing clock

Description automatically generated

  이 것이 왜 오른손좌표계로 불리는 지 이해하기 위해 다음을 읽어보세요.

* 여러분의 오른팔을 y 축의 양의 방향에 따라 위로 뻗어보세요.
* 여러분의 엄지를 오른쪽으로 향하게 해보세요.
* 손가락이 위로 향하게 해보세요.
* 이제 가운데 손가락을 90도로 접어보세요.

  여러분이 이를 정확히 했다면 여러분의 엄지는 x 축의 양의 방향을 가르키고 손가락은 y 축의 양의 방향을 가리키며 가운데 손가락은 z 축의 양의 방향을 가리킬 것입니다. 이 것을 여러분의 왼쪽 팔로 해본다면 z 축이 반대로 향하는 것을 볼 수 있습니다. 이것이 왼손좌표계이고 흔히 DirectX에서 사용됩니다. normalized device coordinates에서 OpenGL은 실제로 왼손좌표계를 사용한다는 것을 알아두세요(projection 행렬이 이를 변경해줍니다).

다음 강좌에서 scene에서 이동하는 방법을 조금 더 상세히 다룰 것입니다. 지금의 view 행렬은 다음과 같습니다.

glm::mat4 view;

// 우리가 움직이고 싶은 방향과 반대의 방향으로 scene을 이동시키고 있다는 것을 알아두세요.

view = glm::translate(view, glm::vec3(0.0f, 0.0f, -3.0f));

우리가 정의해야할 마지막 하나는 projection 행렬입니다. 우리는 perspective projection을 사용할 것이므로 다음과 같이 선언합니다.

glm::mat4 projection;

projection = glm::perspective(glm::radians(45.0f), screenWidth / screenHeight, 0.1f, 100.0f);

이제 우리는 shader에 전달할 변환 행렬들을 생성하였습니다. 먼저 vertex shader에 변환 행렬을 uniform으로 선언하고 그들을 vertex 좌표에 곱해줍니다.

#version 330 core

layout (location = 0) in vec3 aPos;

...

uniform mat4 model;

uniform mat4 view;

uniform mat4 projection;

void main()

{

// 곱셈을 오른쪽에서 왼쪽으로 읽는다는 것을 알아두세요.

gl\_Position = projection \* view \* model \* vec4(aPos, 1.0);

...

}

또한 우리는 shader에 행렬을 전달해야 합니다(일반적으로 각 렌더링 루프가 돌때마다 수행됩니다. 변환 행렬은 자주 변하기 때문입니다).

int modelLoc = glGetUniformLocation(ourShader.ID, "model");

glUniformMatrix4fv(modelLoc, 1, GL\_FALSE, glm::value\_ptr(model));

... // same for View Matrix and Projection Matrix

  이제 vertex 좌표들은 model, view, projection 행렬에 의해 변환되어졌습니다. 최종 오브젝트는 다음과 같습니다.

* 바닥을 향해 뒤쪽으로 기울어져 있습니다.
* 우리에게서 약간 떨어져 있습니다.
* 원근법을 사용하여 출력되었습니다(vertex가 멀리 있을수록 작아집니다).

  결과가 이 조건들을 모두 만족하는지 확인해봅시다.

A screenshot of a computer

Description automatically generated

  실제로 상상의 바닥에서 쉬고있는 3D 평면처럼보입니다. 이와 같은 결과를 보지 못한다면 [소스 코드](https://learnopengl.com/code_viewer_gh.php?code=src/1.getting_started/6.1.coordinate_systems/coordinate_systems.cpp" \t "_blank)를 확인해보세요.

**좀 더 3D답게**

  지금 까지 우리는 3D 공간임에도 불구하고 2D 평면을 사용했습니다. 그래서 이제 2D 평면을 3D 정육면체로 확장해봅시다. 정육면체를 렌더링하기 위해서는 총 36개(6개의 면 \* 2개의 삼각형 \* 3개의 vertex)의 vertex가 필요합니다. 36개의 vertex를 요약하기에는 너무 많으므로 [여기](https://heinleinsgame.tistory.com/code_viewer.php?code=getting-started/cube_vertices" \t "_blank)에서 볼 수 있습니다.

재미를 위해 시간이 지남에 따라 정육면체를 회전시켜볼 것입니다.

model = glm::rotate(model, (float)glfwGetTime() \* glm::radians(50.0f), glm::vec3(0.5f, 1.0f, 0.0f));

그런 다음 glDrawArrays 함수를 사용해 정육면체를 그릴 것이고 이번에는 36개의 vertex를 사용할 것입니다.

glDrawArrays(GL\_TRIANGLES, 0, 36);

  다음과 같은 결과를 얻을 수 있을 것입니다.

  정육면체를 꽤 그럴듯하게 조립을 하였지만 뭔가 이상합니다. 정육면체의 일부 면들이 정육면체의 다른 면들 위로 그려집니다. OpenGL이 삼각형 단위로 그리기 때문에 발생하는 것입니다. 다른 픽셀이 그 위치에 이미 그려져 있음에도 불구하고 그위에 픽셀을 그리는 것입니다. 이 때문에 일부 삼각형들은 겹치지 않았음에도 다른 삼각형들 위에 그려져 겹치게 됩니다.

  운좋게도 OpenGL은 깊이 정보를 z-buffer라고 불리는 버퍼에 저장합니다. 이 버퍼는 OpenGL이 픽셀위에 그릴 것인지 안 그릴것인지 결정하게 해줍니다. z-버퍼를 사용하여 OpenGL이 depth-testing을 할 수 있도록 구성할 수 있습니다.

**Z-버퍼**

  OpenGL은 z-버퍼에 모든 깊이 정보들을 저장합니다. 이 버퍼는 깊이 버퍼(depth buffer)라고도 부릅니다. GLFW는 여러분을 위해 이와 같은 버퍼를 자동으로 생성합니다(출력 이미지의 컬러를 저장하는 컬러 버퍼처럼). 깊이는 각 fragment(fragment의 z 값으로)를 저장하고 fragment가 출력되길 원할때마다 OpenGL은 해당 깊이 값과 z-버퍼를 비교하고 만약 현재 fragment가 다른 fragment의 뒤에 있다면 폐기하며 앞에 있다면 덮어 씌웁니다. 이 과정을 depth testing이라고 부르고 OpenGL에 의해 자동적으로 수행됩니다.

하지만 OpenGL 실제로 depth testing을 수행하도록 하려면 우리는 먼저 OpenGL에게 depth testing을 사용할 것이라는 것을 알려주어야 합니다. 이 것은 기본적으로 사용하지 않도록 되어있습니다. 우리는 glEnable 함수를 사용하여 depth testing을 가능하게 할 수 있습니다. glEnable 함수와 glDisable 함수는 OpenGL에서 특정 기능에 대해서 사용가능/사용불가를 설정할 수 있습니다. 그러면 그 기능은 다른 사용가능/사용불가 명령을 내리기 전까지 사용가능/사용불가 상태가 됩니다. 지금 당장은 GL\_DEPTH\_TEST를 사용가능 상태로 만들어 depth testing이 수행되도록 하겠습니다.

glEnable(GL\_DEPTH\_TEST);

깊이 버퍼를 사용하고 있기 때문에 루프가 돌때마다 깊이 버퍼를 비워주어야합니다(그러지 않으면 이전 프레임의 깊이 정보가 그대로 버퍼에 남아있습니다). 컬러 버퍼를 지우는 것과 마찬가지로 glClear 함수에 DEPTH\_BUFFER\_BIT 비트를 지정함으로써 깊이 버퍼를 비울 수 있습니다.

glClear(GL\_COLOR\_BUFFER\_BIT | GL\_DEPTH\_BUFFER\_BIT);

  우리 프로그램을 다시 실행해보고 OpenGL이 이제 depth testing을 수행하는지 확인하세요.

  해냈습니다! 적절한 depth testing이 적용된 완전히 텍스처가 입혀진 정육면체가 시간이 지남에 따라 회전하고 있습니다. [여기](https://learnopengl.com/code_viewer_gh.php?code=src/1.getting_started/6.2.coordinate_systems_depth/coordinate_systems_depth.cpp" \t "_blank)에서 소스 코드를 확인해 보세요.

**더 많은 정육면체들!**

  만약 화면에 10개의 정육면체를 출력하기를 원한다고 해봅시다. 각 정육면체는 똑같이 생겼겠지만 world에서 위치와 회전도 다를 것입니다. 정육면체의 그래픽 레이아웃은 이미 정의되어 있으므로 더 많은 오브젝트를 렌더링할 때 버퍼나 attribute 배열들을 수정할 필요가 없습니다. 각 오브젝트에 대해 우리가 수정해야할 단 한가지는 정육면체를 world로 변환할 그들의 model 행렬입니다.

먼저 각 정육면체에 대한 이동 벡터를 정의하여 world space에서 그들의 위치를 지정합시다. glm::vec3 배열에 10개의 정육면체 위치를 정의할 것입니다.

glm::vec3 cubePositions[] = {

glm::vec3( 0.0f, 0.0f, 0.0f),

glm::vec3( 2.0f, 5.0f, -15.0f),

glm::vec3(-1.5f, -2.2f, -2.5f),

glm::vec3(-3.8f, -2.0f, -12.3f),

glm::vec3( 2.4f, -0.4f, -3.5f),

glm::vec3(-1.7f, 3.0f, -7.5f),

glm::vec3( 1.3f, -2.0f, -2.5f),

glm::vec3( 1.5f, 2.0f, -2.5f),

glm::vec3( 1.5f, 0.2f, -1.5f),

glm::vec3(-1.3f, 1.0f, -1.5f)

};

  이제 게임 루프 안에서 glDrawArrays 함수를 10번 호출해야 합니다. 그리고 이번에는 렌더링할 때마다 다른 model 행렬을 vertex shader에게 보낼 것입니다. 게임 루프 안에 우리의 오브젝트를 다른 model 행렬을 사용하여 10번 렌더링할 작은 루프를 만들 것입니다. 또한 각 컨테이너에 약간의 회전도 추가할 것입니다.

glBindVertexArray(VAO);

for(unsigned int i = 0; i < 10; i++)

{

glm::mat4 model;

model = glm::translate(model, cubePositions[i]);

float angle = 20.0f \* i;

model = glm::rotate(model, glm::radians(angle), glm::vec3(1.0f, 0.3f, 0.5f));

ourShader.setMat4("model", model);

glDrawArrays(GL\_TRIANGLES, 0, 36);

}

  이 짧은 코드는 각 새로운 정육면체가 렌더링될 때마다 model 행렬을 수정하고 이 것을 총 10번 반복합니다. 이제 다르게 회전하는 10개의 정육면체가 채워진 world를 볼 수 있을 것입니다.

A picture containing room, green, many, table

Description automatically generated

  완벽합니다! 우리 컨테이너가 비슷한 친구들을 찾은 것 같습니다. 만약 문제가 생겼다면 여러분의 코드와 [소스 코드](https://learnopengl.com/code_viewer_gh.php?code=src/1.getting_started/6.3.coordinate_systems_multiple/coordinate_systems_multiple.cpp" \t "_blank)를 비교해보세요.

**연습**

* GLM에 있는 projection 함수의 FoV와 aspect-ratio 파라미터 수정하여 실험해보세요. perspective 절도체에 어떠한 영향을 주는지 살펴보세요.
* 여러가지 방향으로 view 행렬을 이용하여 이동하면서 놀아보세요 그리고 scene이 어떻게 바뀌는지 살펴보세요. view 행렬을 카메라 오브젝트라고 생각해보세요.
* model 행렬만을 사용하여 다른 컨테이너는 움직이지 않고 세 번째 컨테이너마다(첫번째를 포함하여 네 번째, 일곱 번째)만 시간이 지남에 따라 회전하게 만들어보세요: [해답](https://learnopengl.com/code_viewer.php?code=getting-started/coordinate_systems-exercise3)

출처 : [https://learnopengl.com](https://learnopengl.com/), [Joey de Vries](https://joeydevries.com/#home)