1. 3D 그래픽 기초(3D Graphics Fundamentals)

(2) 변환(Transformation)에 대한 수학적 이해

① 변환 파이프라인(Transformation Pipeline)

다음 그림과 같은 변환 파이프라인은 3D로 표현된 모델 좌표계의 정점을 입력하면 컴퓨터 화면에 그릴 수 있도록 2D 화면(픽셀) 좌표로 변환하는 함수(Function)라고 간주할 수 있다.

변환 파이프라인은 순차적으로 실행되는 4개의 단계(월드 변환(World Transform), 카메라 변환(Camera Transform), 투영 변환(Projection Transform), 화면 변환(Screen Transform))로 구성된다. 4개의 단계 각각을 다음과 같은 함수로 표현할 수 있다.

```
CVertex WorldTransform(CVertex vtxModel, CObject *pObject);
CVertex CameraTransform(CVertex vtxWorld);
CVertex ProjectionTransform(CVertex vtxCamera);
CVertex ScreenTransform(CVertex vtxProject);
```

함수 WorldTransform()은 모델 좌표계의 정점과 게임 객체를 입력으로 받아 모델 좌표계의 정점 좌표를 월드 좌표계로 변환한다. 함수 CameraTransform()은 월드 좌표계의 점을 카메라 좌표계로 변환한다. 카메라 좌표계는 플레이어의 눈(카메라)으로 게임 세상을 바라볼 때 카메라의 위치와 방향을 기준으로 표현하는 좌표계이다. 현재는 이러한 카메라 변환을 위하여 가상적인 카메라 객체가 있다고 가정하자. 함수 ProjectionTransform()은 카메라 좌표계의 점을 투영 좌표계로 변환한다. 투영 좌표계는 카메라의 시야각(FOV: Field Of View)과 화면의 종횡비(가로와 세로 길이의 비율)에 따라 보정을 하기 위한 좌표계이다. 함수 ScreenTransform()은 투영 좌표계의 점을 화면 좌표계로 변환한다.

간단한 렌더링 과정을 함수로 표현하기 위하여 다음과 같은 자료구조를 사용한다. CVertex 클래스는 정점 또는 3차원 좌표를 표현하고, CPolygon 클래스는 하나의 다각형을 표현하고, CMesh 클래스는 메쉬(모델)를 표현하고, CObject 클래스는 게임 객체를 표현한다.

```
class CVertex
{
    float       x;
    float       y;
    float       z;
};

class CPolygon
{
    UINT        nVertices;
    CVertex     *pVertices;
```

```
};
class CMesh
{
    UINT         nFaces;
    CPolygon     *pFaces;
};

class CObject
{
    CMesh        *pMesh;
    ...
};
```

변환 파이프라인은 다음과 같은 함수 Transform()으로 표현할 수 있다. 이 함수는 렌더링을 할 게임 객체와 모델 좌표계의 한 정점을 입력으로 받아 화면 좌표계의 픽셀 좌표를 반환한다.

```
CVertex Transform(CVertex vtxModel, CObject *pObject)
{
   CVertex vtxWorld = WorldTransform(vtxModel, pObject);
   CVertex vtxCamera = CameraTransform(vtxWorld);
   CVertex vtxProject = ProjectionTransform(vtxCamera);
   CVertex vtxScreen = ScreenTransform(vtxProject);

   return(vtxScreen);
}
```

렌더링은 게임 세상의 모든 게임 객체들을 그리는 것이라고 할 수 있으므로 렌더링을 다음과 같은 함수 RenderObjects()로 표현할 수 있다. 함수 Draw()는 게임 객체의 메쉬를 구성하는 모든 다각형에 대하여 다각형의 정점들을 픽셀 좌표로 변환하고 와인딩 순서대로 선분으로 이어서 그리는 함수이다. Draw2DLine()는 화면의 두 픽셀을 하나의 선분으로 그리는 함수이며 윈도우 API에서 LineTo(...)와 MoveTo(...)를 사용하여 작성할 수 있다.

```
int gnObjects;          //게임 객체의 개수
CObject *gpObjects;     //게임 객체들의 배열
CCamera *gpCamera;      //카메라 객체

void RenderObjects()
{
   for (int i = 0; i < gnObjects; i++) Draw(&gpObjects[i]);
}

void Draw(CObject *pObject)
{
   CVertex vtxPrevious;
   for (int i = 0; i < pObject->pMesh->nFaces; i++)
   {
      CPolygon *pPolygon = &pObject->pMesh->pFaces[i];
      for (int j = 0; j < pPolygon->nVertices; j++)
      {
```

```
            CVertex v = Transform(pPolygon->pVertices[j], pObject);
            if (j != 0) Draw2DLine(vtxPrevious.x, vtxPrevious.y, v.x,
v.y);
            vtxPrevious = v;
        }
    }
}

void Draw2DLine(float x0, float y0, float x, float y)
{
    HDC hDC = GetDC(...);
    ::MoveToEx(hDC, (long)x0, (long)y0, NULL);
    ::LineTo(hDC, (long)x, (long)y);
    ::ReleaseDC(...);
}
```

② 평행이동(Translation) 변환

직교 좌표계에서 평행이동 변환은 점을 좌표축에 평행하게 이동하는 변환이다. 3차원 직교 좌표계의 점 (x, y, z)를 x축으로 a 만큼, y축으로 b 만큼, z축으로 c 만큼 평행이동한 점은 (x+a, y+b, z+c) 가 된다. 3차원 직교 좌표계의 평행이동 변환 T는 다음과 같이 표현할 수 있다.

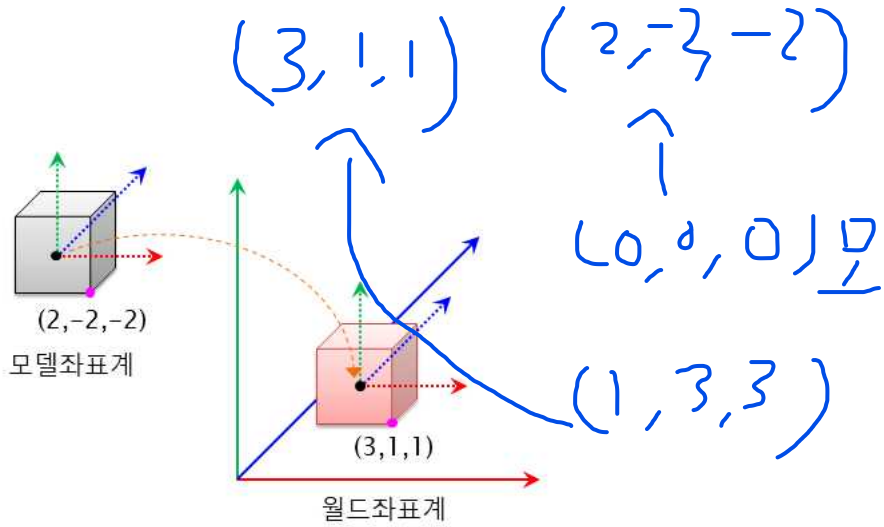
T: (x, y, z) → (x+a, y+b, z+c)

모델 좌표계의 원점과 월드 좌표계의 원점이 같고 두 좌표계의 좌표축의 방향이 같다고 가정하자(실제로 두 원점이 같지 않아도 상관없다). 월드 좌표계는 게임 객체의 위치와 방향을 표현하기 위하여 사용하는데 게임 객체의 회전이 없다고 가정하면, 월드 좌표는 게임 객체의 위치만을 표현하는 것이고, 모델 좌표계의 점 (x, y, z)는 월드 좌표계에서도 (x, y, z)가 된다. 게임 객체의 위치는 게임 객체(모델)의 중심이 월드 좌표계의 어떤 점에 대응되는 가를 표현하는 것이다. 특별한 이유가 없다면 모델의 중심 즉, 모델 좌표계의 원점은 (0, 0, 0)이라고 가정할 수 있다. 월드 좌표계의 원점도 (0, 0, 0)이라고 가정하자.

월드 좌표계에서 게임 객체의 위치가 (a, b, c)이라고 가정하자. 점 (a, b, c)는 월드 좌표계의 원점 (0, 0, 0)을 x축으로 a 만큼, y축으로 b 만큼, z축으로 c 만큼 평행이동한 결과와 같다. 그러면 모델 좌표계의 원점 (0, 0, 0)도 같은 평행이동을 하게 된다. 모델의 모든 정점들에 대하여 같은 평행이동을 적용하면 월드 좌표계로 표현된 정점들을 구할 수 있다.

정리하면, 게임 객체의 위치가 (a, b, c)일 때 모델의 모든 정점들에 대하여 (a, b, c) 만큼의 평행이동을 적용하여 월드 좌표계의 모델 좌표를 얻을 수 있다는 것이다. 즉, 월드 좌표계로 표현된 게임 객체의 위치가 모델 좌표계의 원점이 이동한 위치가 되며, 그리고 게임 객체의 위치가 모델의 각 정점들에 대한 평행이동의 양을 의미한다.

다음 그림은 게임 객체의 위치가 (1, 3, 3)일 때 평행이동 변환에 의해 모델 좌표계의 원점이 월드 좌표계 (1, 3, 3)이 되고, 모델 좌표계의 (2, -2, -2)가 월드 좌표계 (3, 1, 1)이 되는 것을 보이고 있다.



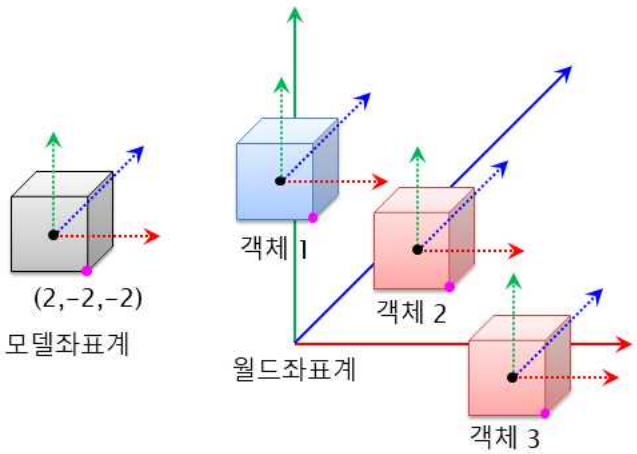
③ 인스턴싱(Instancing)

여러 게임 객체들이 같은 외관(겉모양)을 가지면 각 게임 객체가 모델을 따로 가질 필요가 없이 하나의 모델을 서로 공유하면 된다. 모델의 정보는 렌더링 과정에서 바뀌지 않음에 유의하라. 이렇게 같은 외관을 갖는 게임 객체들이 하나의 모델을 공유하게 표현하고 렌더링하는 것을 인스턴싱이라고 한다.

평행이동 변환만을 고려할 때 인스턴싱을 위해 게임 객체를 C++ 프로그래밍 언어로 다음과 같이 표현할 수 있다. 렌더링을 위하여 게임 객체가 필수적으로 가져야 하는 정보는 위치(월드 좌표계)와 메쉬(모델)에 대한 포인터이다. xPosition, yPosition, zPosition는 게임 객체의 위치가 원점에서 x-축, y-축, z-축 방향으로의 평행이동할 양을 나타내고, pMesh는 게임 객체의 외관을 나타내는 메쉬에 대한 포인터이다.

```
class CObject
{
public:
   CMesh         *pMesh;

   float         xPosition;
   float         yPosition;
   float         zPosition;
};
```



평행이동 변환만을 고려할 때 변환 파이프라인의 함수 WorldTransform()은 다음과 같이 표현할 수 있다.

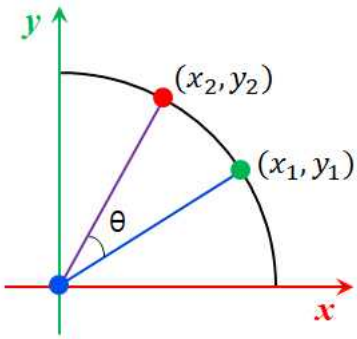
```
CVertex WorldTransform(CVertex vtxModel, CObject *pObject)
{
   CVertex vtxWorld;
   vtxWorld.x = vtxModel.x + pObject->xPosition;
   vtxWorld.y = vtxModel.y + pObject->yPosition;
   vtxWorld.z = vtxModel.z + pObject->zPosition;

   return(vtxWorld);
}
```

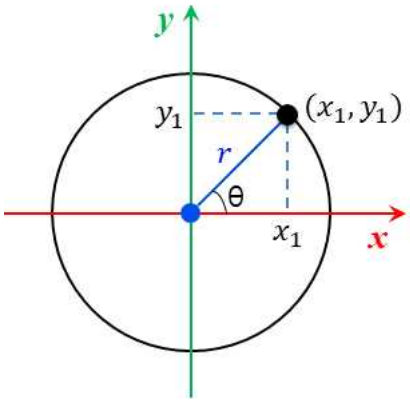
게임 객체의 위치

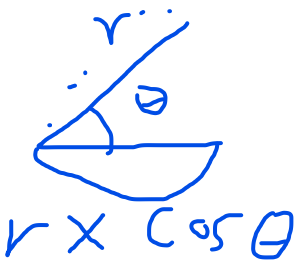
④ 회전(Rotation) 변환

다음 그림은 2차원 평면에서 원점을 중심으로 점 (x_1,y_1)을 θ만큼 반시계방향으로 회전한 결과가 점 (x_2,y_2)임을 나타내고 있다. 점 (x_2,y_2)를 (x_1,y_1)과 θ로 표현해보자.

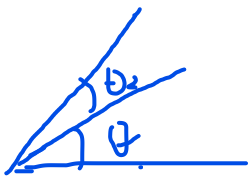


2차원 평면에서 반지름이 r인 원주 위의 점 (x_1,y_1)는 다음과 같이 표현할 수 있다.





$$x_1 = r^*\cos(\theta)$$
$$y_1 = r^*\sin(\theta)$$

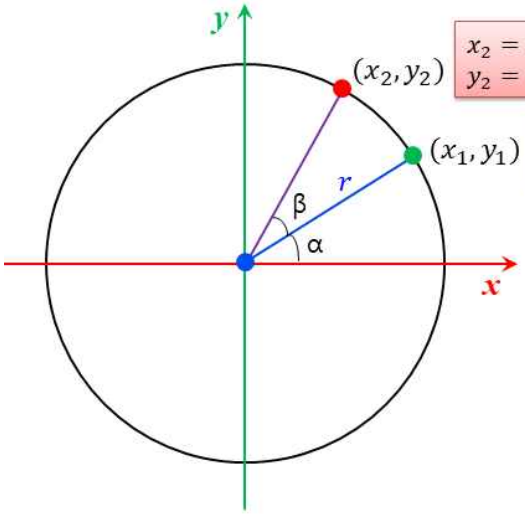


점 (x_2,y_2)를 (x_1,y_1)과 θ로 표현하면 다음과 같다.

$$x_2 = x_1^*\cos(\theta) - y_1^*\sin(\theta)$$
$$y_2 = x_1^*\sin(\theta) + y_1^*\cos(\theta)$$

이것에 대한 증명은 다음 그림을 참고하라. 증명 과정을 이해할 필요는 없다. 또한 그 결과를 외울 필요도 없다. 단지 "2차원 좌표계에서 어떤 점을 원점을 중심으로 회전을 할 때 그 회전의 결과를 수학적으로 구할 수 있다"라는 정도만 기억하도록 하자.

$$x_2 = x_1 * cos(\beta) - y_1 * sin(\beta)$$
$$y_2 = x_1 * sin(\beta) + y_1 * cos(\beta)$$



$$x_2 = r * cos(\alpha + \beta)$$
$$y_2 = r * sin(\alpha + \beta)$$

$$x_1 = r * cos(\alpha)$$
$$y_1 = r * sin(\alpha)$$

$$x_2 = r * cos(\alpha + \beta)$$
$$= r * cos(\alpha) * cos(\beta) - r * sin(\alpha) * sin(\beta)$$
$$= x_1 * cos(\beta) - y_1 * sin(\beta)$$
$$y_2 = r * sin(\alpha + \beta)$$
$$= r * sin(\alpha) * cos(\beta) + r * sin(\beta) * cos(\alpha)$$
$$= x_1 * sin(\beta) + y_1 * cos(\beta)$$

$$cos(\alpha + \beta) = cos(\alpha) * cos(\beta) - sin(\alpha) * sin(\beta)$$
$$sin(\alpha + \beta) = sin(\alpha) * cos(\beta) + sin(\beta) * cos(\alpha)$$

이제 3차원 좌표계(왼손 좌표계)에서 z-축을 중심으로 회전하는 경우를 생각해보자. 위의 그림에서 파란색 원은 z-축에 해당한다. 그리고 2차원 좌표계의 점 (x_1, y_1)과 (x_2, y_2)는 3차원 좌표계의 점 $(x_1, y_1, 0)$과 $(x_2, y_2, 0)$이라고 생각할 수 있다. 3차원 좌표계(왼손 좌표계)에서 z-축을 중심으로 회전하는 것은 2차원 좌표계의 원점을 중심으로 회전하는 것과 같다. 그리고 3차원 좌표계에서 임의의 점 (x_1, y_1, z)를 z-축을 중심으로 회전하면 (x_2, y_2, z)가 된다. 왜냐하면 z-축을 중심으로 회전하면 z-좌표는 변하지 않기 때문이다.

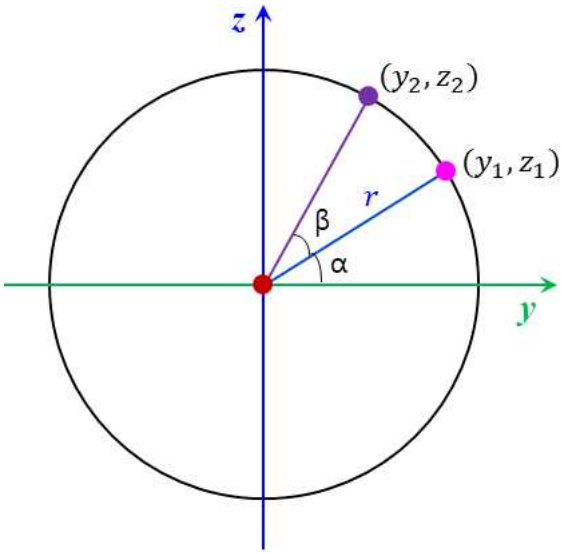
3차원 좌표계(왼손 좌표계)에서 점 (x_1, y_1, z)를 z-축을 중심으로 θ만큼 회전한 결과 (x_2, y_2, z)는 다음과 같이 표현할 수 있다.

$$x_2 = x_1 * \cos(\theta) - y_1 * \sin(\theta)$$
$$y_2 = x_1 * \sin(\theta) + y_1 * \cos(\theta)$$

3차원 좌표계에서 좌표축을 중심으로 회전할 때 회전의 방향은 회전축의 +방향의 점에서 원점을 바라볼 때를 기준으로 한다. 시계방향의 회전을 양(+)의 회전, 그리고 반시계방향의 회전을 음(-)의 회전 방향으로 정한다. 그러면 2차원 좌표계에서 원점을 중심으로 반시계방향으로 회전하는 것을 양(+)의 회전으로 표현하는 것과 회전의 방향이 일치하게 된다.

이제 3차원 좌표계(왼손 좌표계)에서 x-축을 중심으로 회전하는 경우를 생각해보자. 위의 그림에서 파란색 원을 x-축이라고 가정하면 빨간색 x는 y-축에 해당하고, 초록색 y는

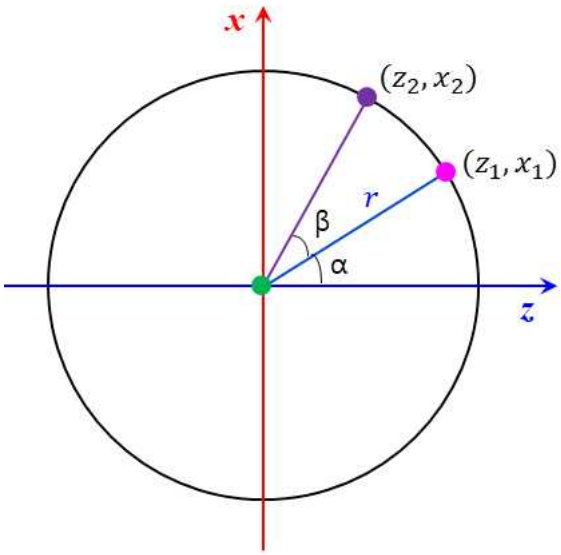
z-축에 해당한다(다음 그림 참조). 3차원 좌표계의 점 (x, y_1, z_1)과 (x, y_2, z_2)는 2차원 좌표계의 점 (y_1, z_1)과 (y_2, z_2)에 대응된다.



3차원 좌표계(왼손 좌표계)에서 점 (x, y_1, z_1)를 x-축을 중심으로 θ만큼 회전하면 x-좌표는 변하지 않기 때문에 회전의 결과 (x, y_2, z_2)는 다음과 같이 표현할 수 있다.

$$y_2 = y_1 * \cos(\theta) - z_1 * \sin(\theta)$$
$$z_2 = y_1 * \sin(\theta) + z_1 * \cos(\theta)$$

같은 방법으로 3차원 좌표계(왼손 좌표계)에서 점 (x_1, y, z_1)를 y-축을 중심으로 θ만큼 회전하면 y-좌표는 변하지 않기 때문에 회전의 결과 (x_2, y, z_2)는 다음과 같이 표현할 수 있다.



$$z_2 = z_1 * \cos(\theta) - x_1 * \sin(\theta)$$

$$x_2 = z_1 * \sin(\theta) + x_1 * \cos(\theta)$$

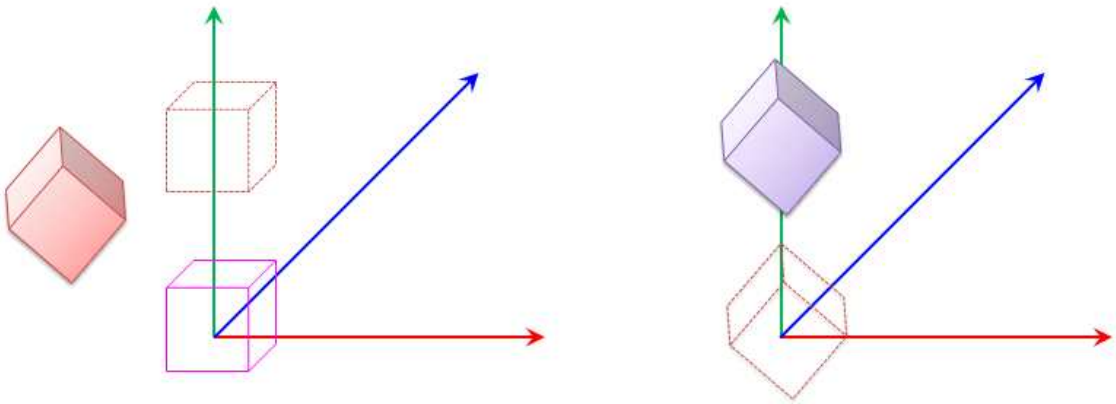
정리하면, 3차원 좌표계(왼손 좌표계)에서 점 (x_1, y_1, z_1)를 어떤 좌표축을 중심으로 θ만큼 회전하면 좌표축에 해당하는 좌표는 바뀌지 않으며, 나머지 좌표를 2차원 좌표계에서 원점을 중심으로 회전하는 방법으로 회전의 결과를 구하는 것과 같다는 것이다. 즉, 3차원 좌표계(왼손 좌표계)에서 좌표축을 중심으로 회전하는 것은 2차원 좌표계에서 원점을 중심으로 회전하는 것과 같다.

⑤ 평행이동 변환과 회전 변환의 적용 순서

회전 변환은 기본적으로 좌표계의 원점 또는 원점을 지나는 좌표축을 기준으로 하는 것으로 가정한다. 평행이동 변환과 회전 변환이 같이 적용될 때 변환을 하는 순서가 중요하다. 평행이동 변환을 먼저 하고 회전 변환을 나중에 하는 결과와 회전 변환을 먼저 하고 평행이동 변환을 나중에 하는 결과는 다르다.

다음 그림은 직육면체를 y-축 방향으로 2만큼 평행이동하는 변환과 z-축을 중심으로 45° 회전하는 변환을 같이 적용할 때 변환의 적용 순서에 따라 변환의 최종 결과가 다르다는 것을 보여준다. 왼쪽 그림은 직육면체를 y-축 방향으로 2만큼 평행이동을 먼저하고 그 결과를 z-축을 중심으로 45° 회전하는 것을 나타낸다. 오른쪽 그림은 직육면체를 먼저 z-축을 중심으로 45° 회전하고 그 결과를 y-축 방향으로 2만큼 평행이동하는 것을 나타낸다. 즉, 평행이동 변환과 회전 변환이 같이 적용될 때 변환을 하는 순서가 중요함을 보이고 있다. 왼쪽 그림에서 직육면체를 z-축을 중심으로 45° 회전할 때 회전의 중심이 객체(메쉬)의 중심이 아니라 좌표계(월드 좌표계)의 원점을 지나는 z-축이다. 오른쪽 그림에서 직육면체를 z-축을 중심으로 45° 회전할 때 회전의 중심이 객체(메쉬)의 중심이 좌표계(월드 좌표계)의 원점을 지나는 z-축과 일치한다.

왼쪽 그림에서와 같이 평행이동을 먼저하고 회전을 나중에 하면 회전은 좌표계의 원점을 기준으로 회전(공전)하게 된다. 오른쪽 그림에서와 같이 회전을 먼저하고 평행이동을 나중에 하면 회전은 객체(메쉬)의 원점을 기준으로 회전(자전)하게 된다.



일반적으로 특별한 상황이 아니면 모든 객체의 회전은 자전(객체의 중심을 기준으로 회전)하는 것으로 가정한다.

태양계에서 태양, 지구, 달은 자전을 하면서 공전을 한다. 달이 자전을 하면서 지구 주위를 공전을 하고, 지구는 자전을 하면서 태양 주위를 공전을 한다. 태양, 지구, 달의 움직임을 회전과 평행이동으로 표현할 수 있는가?

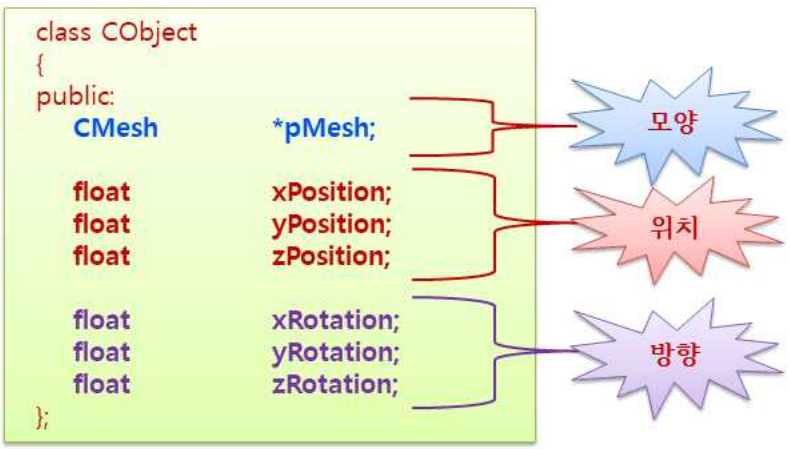
⑥ 회전 변환과 평행이동이 있는 게임 객체의 표현

일반적으로 게임 객체는 평행이동과 회전을 모두 할 수 있다. 평행이동을 하면 게임 객체의 위치가 변하고, 회전을 하면 게임 객체의 방향이 변한다. 게임 객체는 월드 좌표계에서의 위치와 방향을 표현할 수 있어야 한다. 회전 변환과 평행이동 변환을 모두 할 수 있는 게임 객체를 C++ 프로그래밍 언어로 다음과 같이 표현할 수 있다.

```
class CObject
{
public:
   CMesh        *pMesh;

   float        xPosition;
   float        yPosition;
   float        zPosition;

   float        xRotation;
   float        yRotation;
   float        zRotation;
};
```



xRotation, yRotation, zRotation는 x-축, y-축, z-축을 중심으로 회전하는 양을 나타낸다. x-축, y-축, z-축을 중심으로 회전하는 양을 피치(Pitch), 요(Yaw), 롤(Roll)이라고 한다.

게임 객체가 회전 변환과 평행이동 변환을 모두 하는 경우 WorldTransform() 함수는 다음과 같이 표현할 수 있다. 회전 변환을 먼저하고 평행이동 변환을 나중에 한다는 것에 유의하라. x-축, y-축, z-축을 중심으로 회전(보라색 부분)은 앞에서 구한 회전 공식을 사용한다.

```
CVertex WorldTransform(CVertex vtxModel, CObject *pObject)
{
   float fPitch = pObject->xRotation;        회전 양
   float fYaw = pObject->yRotation;
   float fRoll = pObject->zRotation;
   CVertex vtxWorld = vtxModel, vtxRotated;

   if (fPitch) {
      vtxRotated.y = vtxWorld.y * cos(fPitch) - vtxWorld.z *
sin(fPitch);
      vtxRotated.z = vtxWorld.y * sin(fPitch) + vtxWorld.z *
cos(fPitch);
      vtxWorld = vtxRotated;
   }
   if (fYaw) {
      vtxRotated.x = vtxWorld.x * cos(fYaw) + vtxWorld.z *
sin(fYaw);
      vtxRotated.z = -vtxWorld.x * sin(fYaw) + vtxWorld.z *
cos(fYaw);
      vtxWorld = vtxRotated;
   }
   if (fRoll) {
      vtxRotated.x = vtxWorld.x * cos(fRoll) - vtxWorld.y *
sin(fRoll);
      vtxRotated.y = vtxWorld.x * sin(fRoll) + vtxWorld.y *
cos(fRoll);
      vtxWorld = vtxRotated;
   }

   vtxWorld.x += pObject->xPosition;        회전변환을 먼저 한 후 이동변환
   vtxWorld.y += pObject->yPosition;
   vtxWorld.z += pObject->zPosition;

   return(vtxWorld);
}
```

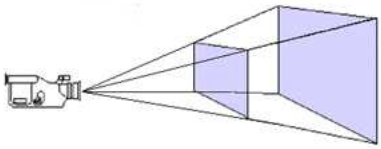
⑦ 카메라 변환(Camera Transformation, Viewing Transformation)

- 가상 카메라(Virtual Camera)

  게임 세계를 렌더링하기 위해서는 게임 세계를 보기 위한 가상의 카메라(객체)가 필요하다. 실세계에서 사람이 카메라 또는 눈을 통해 세상의 일부를 볼 수 있는 것처럼, 게임 플레이어는 이 가상 카메라를 통해 게임 세계의 일부를 볼 수 있다. 3차원 공간의 게임 객체들은 가상 카메라의 2차원 평면으로 투영되고 이 투영된 2차원 평면의 장면이 화면에 그려져야 한다. 그러므로 게임 플레이어가 현재 보고 있는 장면은 가상 카메라에 나타나는 장면이다. 일반적으로 이 가상 카메라는 플레이어 캐릭터에 부착되어 있다. 1인칭 게임의 경우 플레이어 캐릭터의 눈이 카메라에 해당한다. 카메라에 보이는 게임 객체들만 화면에 최종적으로 그려지게 된다(앞으로 가상 카메라를 카메라로 표기함).

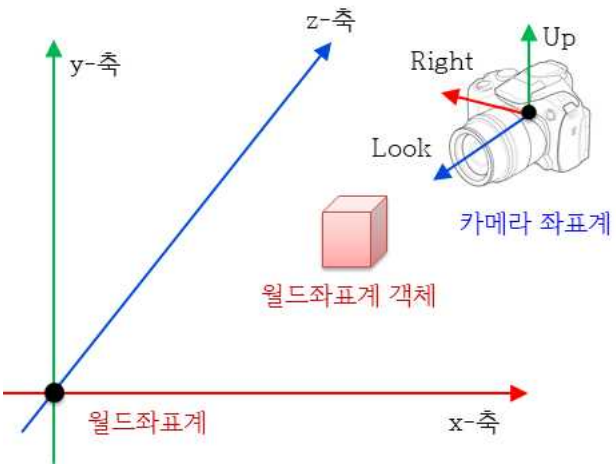
카메라가 가져야 하는 일반적인 정보는 다음과 같다.

- 카메라의 위치(Position)
  월드 좌표계에서 카메라가 어디에 있는 가를 표현
- 카메라의 방향(Viewing Direction)
  월드 좌표계에서 카메라가 바라보는 방향(카메라의 회전)을 표현
- 카메라의 화각(FOV(Field Of View))
  카메라가 바라보는 시야 각도(범위)를 표현



- 카메라 좌표계(View Space, Camera Space)

  3차원 게임에서 카메라가 이동을 하면 위치가 변하게 된다. 게임 세계에서 카메라가 현재 어느 위치에 있는 가는 월드 좌표계로 표현되어야 한다. 카메라가 회전을 하면 카메라가 바라보는 방향이 달라진다. 카메라가 바라보는 방향은 왼손 좌표계에서 z-축 방향이다. 일반적으로 카메라의 회전은 카메라의 중심을 기준으로 이루어져야 한다(자전). 카메라 좌표계는 카메라의 중심 위치(월드 좌표계)를 원점으로 하며 회전에 따라 달라진 방향을 좌표축으로 표현되는 좌표계이다. 게임 세계를 카메라를 중심으로 한 상대적인 좌표계로 표현하는 변환이 카메라 변환이다.



다음은 간단한 카메라를 표현하고 있다. (xPosition, yPosition, zPosition)는 월드 좌표계에서 가상 카메라의 위치, 즉, 가상 카메라가 게임 세계의 어디에 위치하는 가를 나타낸다. (xRotation, yRotation, zRotation)는 가상 카메라의 방향을 나타낸다.

```
class CCamera
{
public:
   float        xPosition; //카메라의 위치(월드 좌표계)
   float        yPosition;
   float        zPosition;

   float        xRotation; //카메라의 방향(회전 각도)
```

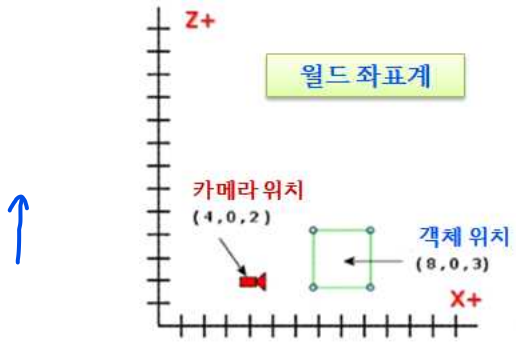
```
    float       yRotation;
    float       zRotation;

    float       fovAngle; //카메라의 화각
};
```

⑧ 카메라 변환(Camera Transformation)

카메라를 이동하고 회전할 때 주의할 점은 카메라의 움직임과 화면상의 게임 객체의 움직임은 방향이 서로 반대라는 것이다. 예를 들어, 카메라를 앞으로 움직이면, 화면상에서 게임 세계가 전체적으로 카메라의 이동 방향과 반대로(카메라에 가까워지는 방향으로) 이동하는 것처럼 보이게 된다. 이것은 카메라를 움직이지 않고 게임 세계의 게임 객체들을 카메라 이동 방향과 반대 방향으로 이동하는 것과 화면상의 결과와 같다. 또한, 카메라를 왼쪽으로 이동하면 화면상에서 게임 세계는 오른쪽으로 이동하게 된다. 카메라를 회전하는 경우도 화면상에서 게임 세계는 반대 방향으로 회전을 한다.

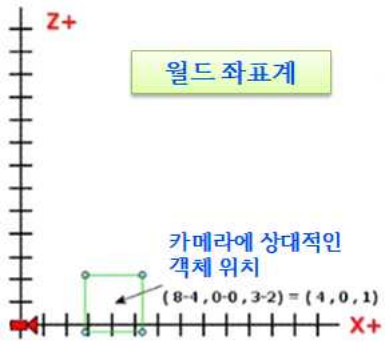
카메라 변환은 월드 좌표계로 표현된 점을 카메라 좌표계로 변환하는 것이다. 카메라를 회전하고 이동하기 전에, 카메라가 월드 좌표계의 원점에 위치하고, 카메라(객체) 지역 좌표계의 x-축, y-축, z-축의 방향이 월드 좌표계의 x-축, y-축, z-축과 각각 일치한다고 가정하자. 카메라를 y-축을 중심으로 시계 방향으로 90° 회전(자전)하였고, 평행이동에 의해 카메라의 위치가 (4, 0, 2)가 되었다. 그리고 (8, 0, 3)의 위치에 게임 객체(직육면체)가 있다. 다음 그림은 이 상황을 나타내고 있다. 이때 카메라는 직육면체의 오른쪽 부분을 보게 된다.



위의 그림과 같은 위치와 방향을 가진 카메라에 대한 카메라 변환 과정을 생각해보자. 카메라 변환은 월드 좌표계로 표현된 점들의 좌표를 카메라의 중심과 방향을 사용하여 상대적으로 표현하는 과정이다. 다르게 말하면 게임 세계가 월드 좌표계의 기준(월드 좌표계의 원점과 축 방향)이 아니라 카메라를 기준으로 표현되어야 한다는 것이다(카메라가 게임 세계를 표현하는 중심이다 또는 카메라가 세상의 중심이다). 월드 좌표계의 점을 카메라의 위치와 방향에 따라 직접 변환하지 않고, 카메라가 원래 세상의 기준(월드 좌표계의 원점과 축 방향)이 되도록 변환하도록 하자. “게임 객체 또는 카메라가 이동하고 회전을 하더라도 월드 좌표계의 기준은 불변이다”라는 것에 주의하라.

다음 그림은 카메라를 월드 좌표계의 원점으로 이동하는 변환을 월드 좌표계로 표현된

정점에 대해서도 같게 적용한 것을 나타낸다. 카메라의 위치가 (4, 0, 2)이므로 카메라를 월드 좌표계의 원점으로 이동하려면 x-축으로 -4, z-축으로 -2 만큼 평행이동을 한다. 게임 객체(직육면체)의 위치 (8, 0, 3)에 같은 평행이동을 적용하면 위치가 (4, 0, 1)이 된다. (4, 0, 1)은 카메라의 위치를 기준으로 직육면체가 상대적으로 떨어져 있는 위치가 된다. 이제 카메라는 변하지 않는 월드 좌표계의 기준(원점)에 있게 되었다.



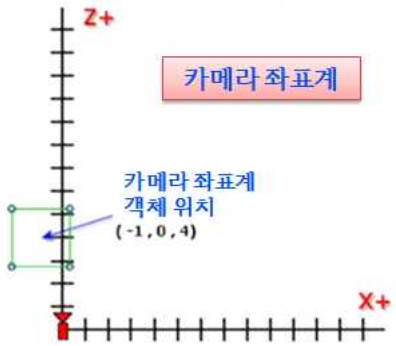
카메라를 y-축을 중심으로 회전을 했기 때문에 위의 그림에서 카메라가 바라보는 방향(z-축)은 월드 좌표계의 x-축이다. 카메라의 방향(z-축)을 변하지 않는 월드 좌표계의 z-축과 일치시키도록 하자.

다음 그림은 카메라를 y-축을 중심으로 반시계 방향으로 90° 회전하고, 이 회전 변환을 게임 객체(직육면체)의 위치 (4, 0, 1)에도 적용한 것을 나타내고 있다. 3차원 좌표계(왼손 좌표계)에서 점 (4, 0, 1)을 y-축을 중심으로 -90° 회전한 결과는 다음과 같이 표현할 수 있다(sin(-90) = -1, cos(-90) = 0).

$$z_2 = z_1 * \cos(\theta) - x_1 * \sin(\theta) \qquad z = 4 = 1 * 0 - 4 * (-1)$$

$$x_2 = z_1 * \sin(\theta) + x_1 * \cos(\theta) \qquad x = -1 = 1 * (-1) + 4 * 0$$

원래 카메라는z축을 바라보아야 한다.



이제 월드 좌표계의 원점과 카메라 좌표계의 원점이 일치하고 좌표축의 방향도 서로 같아졌다. 월드 좌표계에서 중심 좌표가 (8, 0, 3)인 게임 객체(직육면체)가 카메라 좌표계로 변환되었고 그 중심 좌표는 (-1, 0, 4)가 된다는 것을 보여주고 있다. 이때 카메라는 직육면체의 오른쪽 부분을 보게 되며, 카메라가 보는 이미지는 카메라 변환을 수행하기 전의 카메라 위치에서 보는 이미지와 완벽히 같게 된다. 그러나 월드 좌표계로 표현된 점이 카

메라 좌표계로 표현되었다.

카메라 변환은 다음과 같이 정리할 수 있다.

❶ 카메라를 월드 좌표계의 원점으로 옮기는 평행이동 변환을 월드 좌표계로 표현된 점들에 적용한다. 카메라를 월드 좌표계의 원점으로 옮기는 평행이동 변환은 카메라를 카메라의 위치로 이동한 평행이동 변환의 반대 방향으로 이동하는 것이다(월드 좌표계의 점에서 카메라의 위치를 빼면 된다).

❷ 카메라 좌표계의 축이 월드 좌표계의 축과 일치하도록 카메라를 회전하는 변환을 월드 좌표계로 표현된 점들에 적용한다. 카메라 좌표계의 축이 월드 좌표계의 축과 일치하도록 회전하는 변환은 카메라를 회전한 방향과 반대 방향으로 회전하는 것이다(회전 방향이 반대인 회전은 회전 각도의 부호를 반대로 회전하는 것이다).

다음은 카메라 변환을 하는 함수 CameraTransform()를 구현한 예이다.

```
CVertex CameraTransform(CVertex vtxWorld)
{
   CVertex vtxCamera, vtxRotated;
   vtxCamera.x = vtxWorld.x - gpCamera->xPosition;
   vtxCamera.y = vtxWorld.y - gpCamera->yPosition;
   vtxCamera.z = vtxWorld.z - gpCamera->zPosition;

   float fPitch = DegreeToRadian(-gpCamera->xRotation);
   float fYaw = DegreeToRadian(-gpCamera->yRotation);
   float fRoll = DegreeToRadian(-gpCamera->zRotation);

   if (fPitch) {
      vtxRotated.y = vtxCamera.y * cos(fPitch) - vtxCamera.z *
sin(fPitch);
      vtxRotated.z = vtxCamera.y * sin(fPitch) + vtxCamera.z *
cos(fPitch);
      vtxCamera = vtxRotated;
   }
   if (fYaw) {
      vtxRotated.x = vtxCamera.x * cos(fYaw) + vtxCamera.z *
sin(fYaw);
      vtxRotated.z = -vtxCamera.x * sin(fYaw) + vtxCamera.z *
cos(fYaw);
      vtxCamera = vtxRotated;
   }
   if (fRoll) {
      vtxRotated.x = vtxCamera.x * cos(fRoll) - vtxCamera.y *
sin(fRoll);
      vtxRotated.y = vtxCamera.x * sin(fRoll) + vtxCamera.y *
cos(fRoll);
      vtxCamera = vtxRotated;
   }

   return(vtxCamera);
```

```
}
```

⑨ 원근 투영 변환(Perspective Projection Transformation)

모델 좌표계의 정점(3차원)이 카메라 변환까지의 과정을 거치면 여전히 3차원 카메라 좌표가 된다. 이러한 3차원 점을 화면(2차원)에 그리려면 3차원 점(좌표)을 2차원 점(좌표)으로 변환해야 한다. 이러한 변환의 가장 쉬운 예는 그림자이다. 다음 그림에서 벽면은 2차원으로 가정할 수 있다. 3차원 물체(객체)들에 조명(빛)을 비추면 벽에 물체들의 그림자가 그려진다. 이 그림자들은 원래 3차원인 물체들이 2차원으로 변환된 것이다. 이렇게 3차원 점(좌표)을 2차원 점(좌표)으로 변환하는 과정을 투영(Projection)이라고 한다. 벽에 그려진 그림자들을 보면 물체의 형태와 크기를 잘 반영하고 있다. 또한 동일한 물체가 조명에 가까이 있을 때와 멀리 있을 때의 그림자 크기는 달라진다.

카메라를 통하여 3차원 게임 객체들을 2차원 화면에 렌더링하는 것은 현실 세계에서 휴대폰 카메라로 사진을 찍는 것과 유사하다. 현실 세계에서 사진을 찍는 과정도 투영의 과정이 포함되어 있다. 실세계 물체들의 표면에서 반사된 빛이 카메라 렌즈를 통과하면 그 빛을 2차원 필름 또는 센서에 기록하는 것이 사진을 찍는 것이다. 게임 세계의 카메라도 현실 세계의 카메라와 유사하다. 게임 세계의 3차원 게임 객체들을 카메라의 화면에 투영하는 것은 그림자가 그려지는 것과 유사하지만 그림자를 컬러(색상)로 그려야 하고 3차원 세상을 느낄 수 있도록 그려야한다는 것이 다른 점이라 할 수 있다.

결국 3차원 게임 객체들을 2차원 화면에 렌더링하였을 때, 화면에 그려진 그림(사진)을 보고 입체감 또는 거리감을 느낄 수 있도록 그려야 한다. 화가는 현실 세계를 보고 그림을 그릴 때 원근법을 사용하여 원근감(Perspective)을 표현한다. 원근감이란 카메라에서 멀리 떨어져 있는 객체는 작게 그려지고, 카메라에 가까운 객체는 크게 그려지는 것을 의미한다. 또한 아주 멀리 떨어져 있는 객체는 카메라 중심선 근처로 몰려서 위치하게 된다(소멸점: Vanishing point).

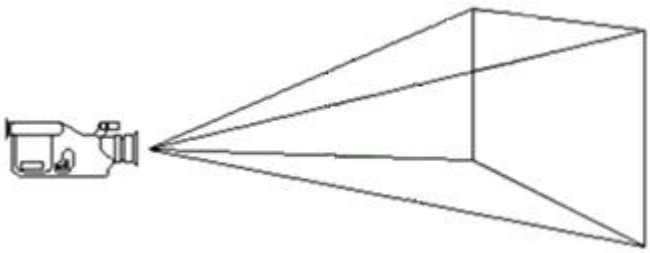
⑩ 원근 투영 나누기(Perspective Projection Division)

모델 좌표계의 임의의 한 정점이 카메라 변환을 거쳐서 3차원 좌표계의 점 (x, y, z)가 되었다고 가정하자. 점 (x, y, z)는 카메라 좌표계로 표현된 것이다. 점 (x, y, z)의 z 값으로 x, y, z를 모두 나누어 보자. 그러면 점 (x, y, z)는 (x/z, y/z, 1)이 된다. 이렇게 카메라 좌표계로 표현된 점의 x, y, z 좌표를 z 값으로 나누는 것을 원근 투영 나누기라고 한다. 원근 투영 나누기를 하면 모든 점의 z 좌표가 1이 된다. 모든 점의 z 좌표가 1이 되면 좌표의 표현에서 1을 빼도 될 것이다. 그러면 3차원 좌표계의 점 (x, y, z)가 2차원 좌표계의 점 (x/z, y/z)으로 변환된다. 원근 투영 나누기를 하는 것은 3차원 좌표계의 점을 2차원 좌표계의 점으로 변환하는 것이다.

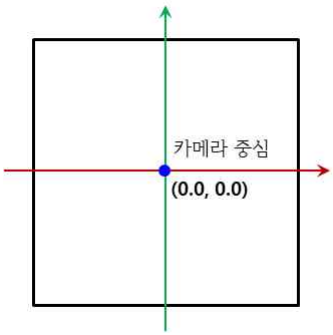
카메라 좌표계에서 임의의 점 (x, y, z)에 대하여 생각을 해보자. 이 점이 카메라에 가깝다면 z 좌표가 작고 카메라에서 멀다면 z 좌표가 클 것이다. 이 점에 대한 원근 투영의 결과 (x/z, y/z)에서 z값이 크다면 x/z 또는 y/z는 작아지고, z값이 작다면 x/z 또는 y/z는 커질 것이다. x 좌표와 y 좌표가 같은 두개의 점 (x_0, y_0, z_0)와 (x_0, y_0, z_1)에 대한 원근 투영의 결과는 $(x_0/z_0, y_0/z_0)$와 $(x_0/z_1, y_0/z_1)$이다. 점 (x_0, y_0, z_0)가 (x_0, y_0, z_1) 보다 카메라에 가까이 있다면 $(z_0 \leq z_1)$이다. $(z_0 \leq z_1)$이면 $(x_0/z_0 \geq x_0/z_1)$이고 $(y_0/z_0 \geq y_0/z_1)$이다. 이것은 카메라에서 멀리 있는 객체가 가까이 있는 객체보다 더 작게 투영이 된다는 것을 나타낸다. 3차원 좌표계의 점을 2차원 좌표계의 점으로 변환할 때 원근 투영 나누기를 하면 원근법을 적용하는 것이다.

⑪ 원근 투영 좌표계(Perspective Projection Space, Clip-Space)

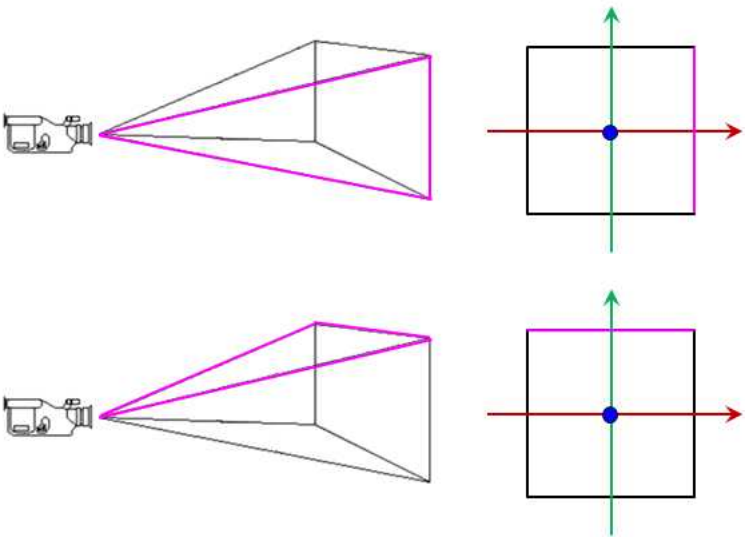
카메라의 시야각(FOV)은 카메라 좌표계의 z-축을 기준으로 왼쪽(-x 축 방향), 오른쪽(+x 축 방향), 위쪽(+y 축 방향), 아래쪽(-y 축 방향)으로 볼 수 있는 각도이다. 카메라가 볼 수 있는 영역은 다음의 그림과 같이 사각뿔 형태이다. 게임 세계의 게임 객체들이 이 사각뿔 영역에 완전히 포함되거나 일부라도 포함되면 카메라에 보일 것이다. 카메라의 뒤쪽에 있는 게임 객체들은 당연히 카메라에 보이지 않는다. 이때 카메라 좌표계에서 이 게임 객체들의 z-좌표가 음수일 것이다.



카메라에 보이는 게임 세계의 게임 객체들은 카메라의 2차원 사각형 영역으로 원근 투영될 것이다(실세계의 사진기 또는 디지털 카메라를 생각해보라). 이 사각형의 중심은 카메라의 중심(카메라 좌표계의 원점)이 되고, 사각형의 오른쪽은 카메라 좌표계의 +(x-축)이 되고, 사각형의 위쪽은 카메라 좌표계의 +(y-축)이 된다. 이 사각형은 투영 사각형(Projection rectangle)이라고 한다.

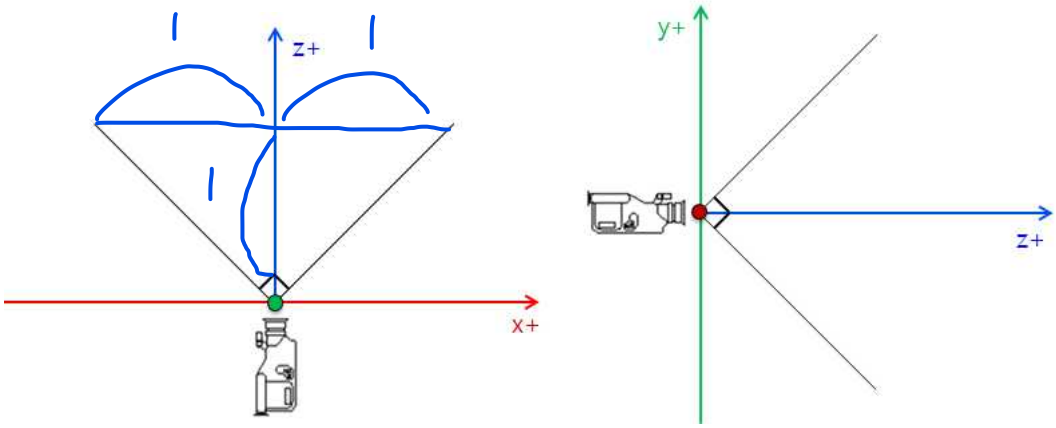


다음 그림은 카메라의 투영 사각형으로 원근 투영되는 공간(사각뿔)의 오른쪽 삼각형에 있는 점들은 투영 사각형의 오른쪽 선분으로 투영됨을 보여준다. 사각뿔의 위쪽 삼각형에 있는 점들은 투영 사각형의 위쪽 선분으로 투영됨을 보여준다.



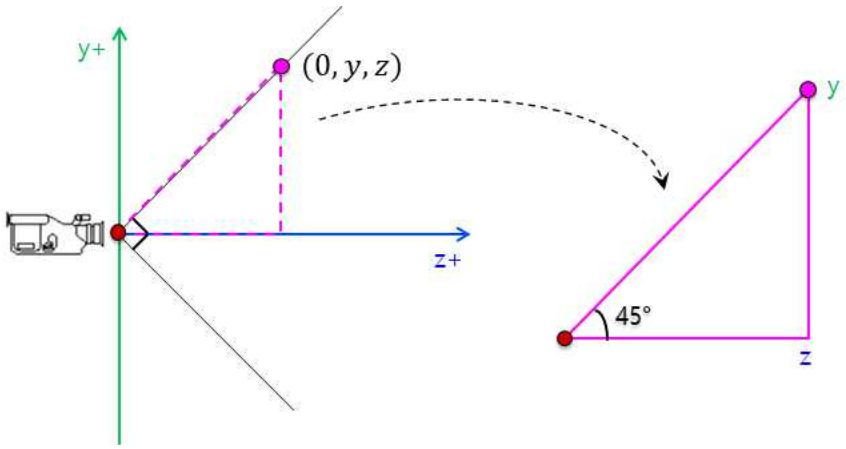
카메라의 시야각이 90°라고 가정해 보자. 시야각이 90°이면, 다음 왼쪽 그림과 같이 카메라 좌표계의 z-축을 기준으로 왼쪽으로 45°, 그리고 오른쪽으로 45°의 영역을 볼 수 있다.

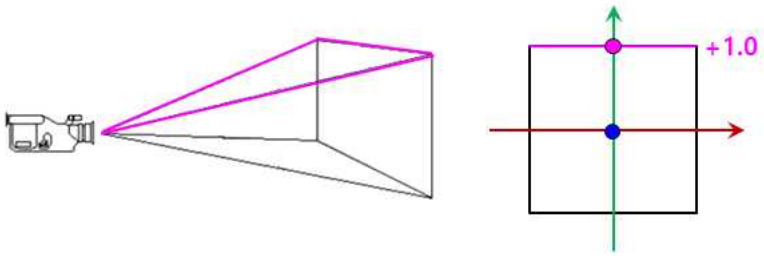
이 그림은 카메라가 볼 수 있는 영역을 +y 축에서 내려다보는 것을 나타낸다.



시야각이 90°이면, 위의 오른쪽 그림과 같이 위쪽과 아래쪽으로 45°씩의 영역을 볼 수 있다. 이 그림은 카메라가 볼 수 있는 영역을 +x 축에서 보는 것을 나타낸다.

다음의 그림은 시야각이 90°일 때, +x 축에서 카메라가 볼 수 있는 영역의 일부를 나타낸 것이다. 카메라의 투영 사각형으로 투영되는 사각뿔의 위쪽 삼각형에 있는 점 (0, y, z)와 원점 (0, 0, 0)으로 구성되는 직각삼각형을 생각해 보자. 이 직각삼각형의 한 예각이 45°이므로 이 직각삼각형은 직각이등변삼각형이고, 빗변이 아닌 두 변의 길이가 같다. 점 (0, y, z)의 y와 z가 같다(y = z). 사각뿔의 위쪽 삼각형에 있는 점 (0, y, z)를 투영 사각형으로 투영하면(원근 투영 나누기를 하면) 2차원 좌표 (0, 1)이 된다. 사각뿔의 위쪽 삼각형에 있는 점 (x, y, z)를 투영 사각형으로 투영하면 2차원 좌표 (x/z, 1)이 된다. 즉, 사각뿔의 위쪽 삼각형에 있는 점을 투영 사각형으로 투영하면 2차원 좌표의 y 좌표는 +1이 된다. 사각뿔의 아래쪽 삼각형에 있는 점을 투영 사각형으로 투영하면 2차원 좌표의 y 좌표는 -1이 된다. 사각뿔의 오른쪽 삼각형에 있는 점을 투영 사각형으로 투영하면 2차원 좌표의 x 좌표는 +1이 되고, 사각뿔의 왼쪽 삼각형에 있는 점을 투영 사각형으로 투영하면 2차원 좌표의 x 좌표는 -1이 된다.

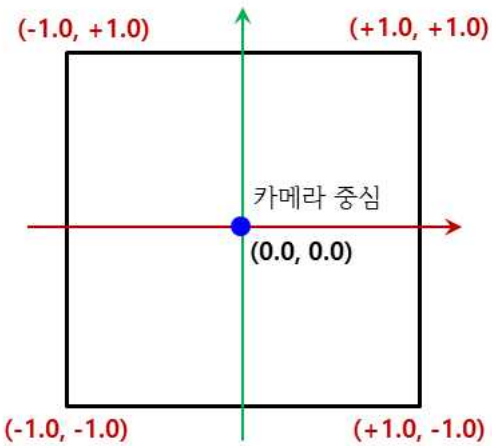




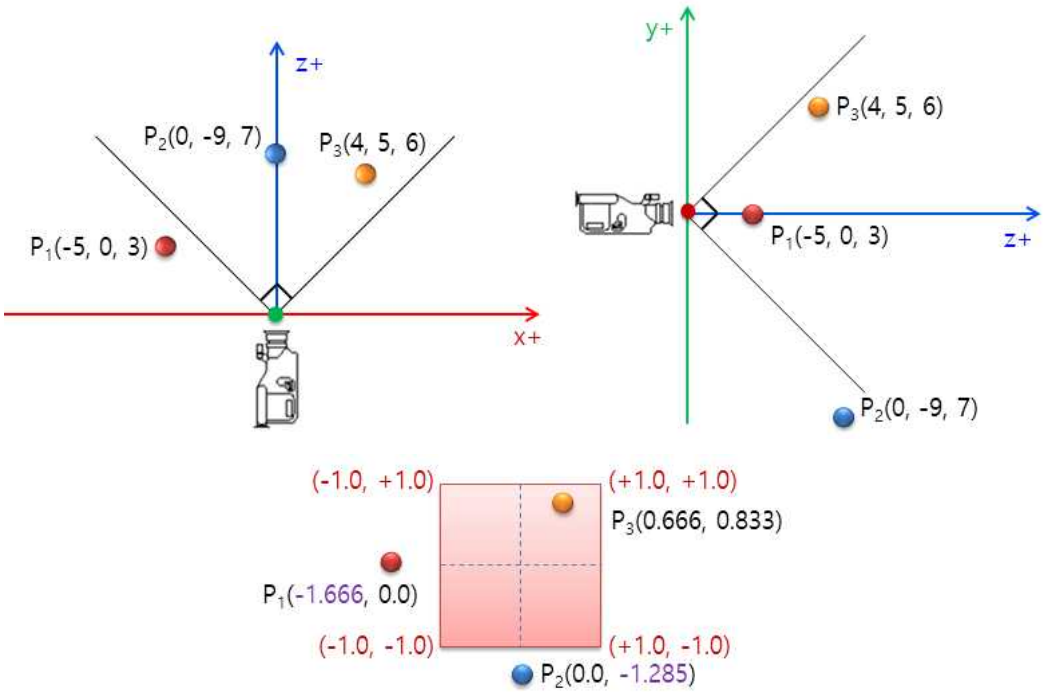
다음의 그림은 시야각이 90°인 카메라가 볼 수 있는 영역이 투영 사각형으로 투영될 때 투영 사각형의 좌표를 표현한 것이다. 시야각이 90°인 카메라에 보일 수 있는 모든 점 (x, y, z)를 원근 투영을 하면 2차원 점 (x/z, y/z)이 되며 다음을 만족하며, 투영 사각형은 다음 그림과 같이 정사각형(정규화된 투영 사각형)이 된다.

이래야지 보인다. $-1 \le \frac{x}{z} \le 1 \quad -1 \le \frac{y}{z} \le 1$

카메라의 시야각이 90°일 때, 카메라에 보이는 3차원 점을 원근 투영을 하면 투영된 점의 x 좌표와 y 좌표는 모두 -1 보다 크거나 같고 +1 보다 작거나 같은 값을 갖는다. 투영된 점의 x 좌표와 y 좌표는 -1 보다 작거나 또는 +1 보다 크면, 이 점으로 투영된 3차원 점은 카메라에 보이지 않는 점이다. 다음 그림의 투영 사각형을 NDC(Normalized Device Coordinates) 또는 클립 공간(Clip space)이라고 부르기도 한다.

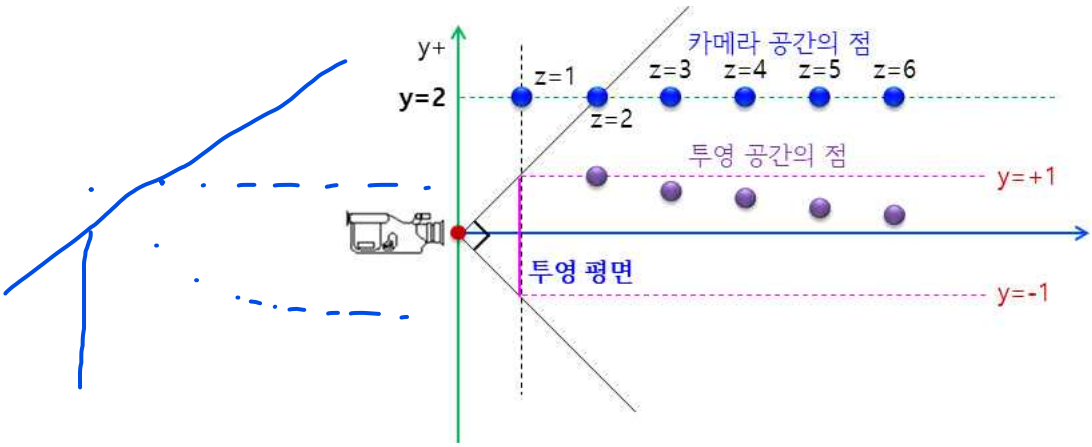


다음의 그림은 시야각이 90°인 카메라에 대하여 세 점(P1, P2, P3)이 원근 투영되는 상황을 나타낸다. P1(-5, 0, 3), P2(0, -9, 7), P3(4, 5, 6)에 대하여 원근 투영 나누기를 하면 P1(-1.666, 0.0), P2(0.0, -1.285), P3(0.666, 0.833)가 된다. P3(0.666, 0.833)의 x 좌표와 y 좌표는 모두 -1 보다 크거나 같고 +1 보다 작거나 같은 값을 가지므로 카메라에 보이는 점이다. P1(-1.666, 0.0)과 P2(0.0, -1.285)는 그렇지 않으므로 카메라에 보이지 않는 점이다.



⑫ 투영 평면(Projection)

시야각이 90˚인 카메라에서 점 (x, y, z)를 원근 투영 나누기를 하면 (x/z, y/z, 1)이 되므로 투영 사각형과 카메라 사이의 거리는 1이다. 게임 세계에 있는 모든 점들은 카메라에서 거리가 1인이고 z-축에 수직인 평면으로 투영된다. 이 평면을 투영 평면이라고 한다. 투영 사각형은 이 평면에 위치한다.



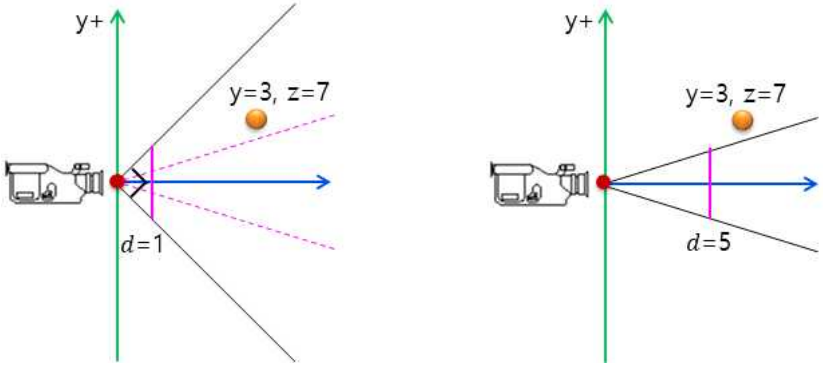
위의 그림은 시야각이 90˚인 카메라에서 y 좌표가 2이고 z 좌표가 1, 2, 3, 4, ...인 점들의 원근 투영의 결과를 보이고 있다. 투영 평면은 (z = 1)에 있다. z 좌표가 클수록 원근 투영의 결과는 z-축에 점점 접근함을 보이고 있다(Vanishing point).

시야각이 90˚인 카메라는 원근 투영의 결과로 투영 사각형이 다음과 같이 정규화가 되는 장점이 있다.

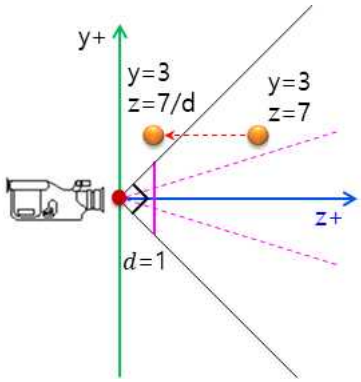
$$-1 \le \frac{x}{z} \le 1 \qquad -1 \le \frac{y}{z} \le 1$$

이것은 원근 투영 나누기 연산을 한 후 정규화가 된 투영 사각형을 벗어나는 점은 카메라에 보이지 않는 점이므로 그리지 않아도 된다고 빠르게 판단을 할 수 있다.

투영 평면과 카메라 사이의 거리를 d라고 하면, 시야각이 90˚인 카메라에서 d는 1이다. 카메라의 시야각이 90˚가 아닌 경우도 점 (x, y, z)를 원근 투영 나누기를 하면 (x/z, y/z, 1)이 되므로 투영 사각형(투영 평면)과 카메라 사이의 거리 d는 1이다. 그러나 원근 투영 나누기의 결과는 시야각이 90˚인 카메라와 같이 정규화가 되지 않는다. 다음의 왼쪽 그림에서 보라색 점선은 시야각이 90˚보다 작은 경우의 카메라 공간이며 이 카메라에 노란색 점은 보이지 않는다.



시야각이 90˚보다 작은 경우 위의 오른쪽 그림과 같이 투영 사각형이 정규화가 되도록 하려면 투영 사각형(투영 평면)을 z-축의 + 방향으로 이동하면 된다. 이때 투영 평면과 카메라 사이의 거리 d는 1 보다 크다. 시야각이 90˚보다 큰 경우에는 투영 사각형이 정규화가 되도록 하려면 투영 사각형(투영 평면)을 z-축의 - 방향으로 이동하면 된다. 이때 투영 평면과 카메라 사이의 거리 d는 1 보다 작다. 실세계 카메라에서 렌즈를 교체하는 것이 가상 카메라에서 투영 사각형을 이동하는 하는 것과 유사하다. 가상 카메라는 렌즈를 교체할 수 없으므로(렌즈의 교체를 수학적으로 처리하기 어려우므로) 우리가 사용하는 가상 카메라의 시야각이 항상 90˚라고 가정하자. 즉, 투영 평면과 카메라 사이의 거리 d는 항상 1이다. 시야각이 90˚가 아닌 카메라를 시야각이 90˚인 카메라라고 가정한다.



위의 그림에서 노란색 점은 시야각이 90˚보다 작은 카메라에는 보이지 않지만 시야각이 90˚인 카메라에는 보인다. 우리는 게임 세계를 시야각이 90˚인 카메라를 사용하여 렌더링

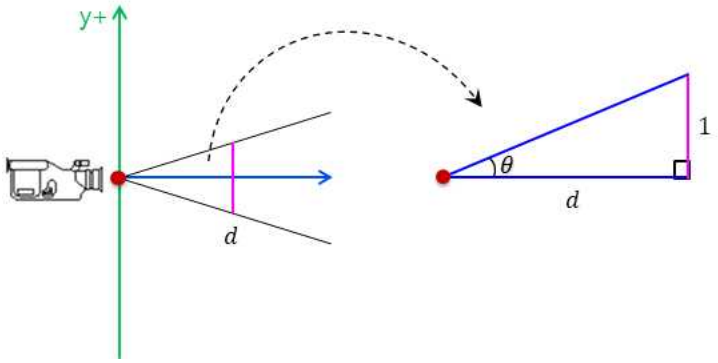
시야각>90 -> z↑

<90 -> z↓

할 것이므로 이 노란색 점이 시야각이 90°인 카메라에 보이지 않도록 처리하면 된다. 노란색 점을 시야각이 90°인 카메라에서 보이지 않게 만들려면 이 점을 z-축의 - 방향으로 이동시키면 된다. 즉, z 좌표를 작게 하면 된다. 시야각이 90°보다 큰 카메라의 경우에는 z 좌표를 크게 하면 된다. 투영 평면과 카메라 사이의 거리 d는 시야각이 90°보다 크면 1보다 작고, 시야각이 90°보다 작으면 1보다 크다.

시야각이 90°가 아닌 카메라를 시야각이 90°인 카메라처럼 동작하게 하려면 원근 투영 나누기를 하기 전에 z 좌표를 투영 평면과 카메라 사이의 거리 d로 나누면 된다. 즉, 카메라 좌표계의 점 (x, y, z)를 (x, y, z/d)로 변환하고 원근 투영 나누기를 한다.

카메라의 시야각의 절반이 θ라고 하면 투영 사각형이 세로의 길이가 2인 정사각형이므로 투영 사각형과 카메라 사이의 거리 d는 다음 그림에서 계산할 수 있다.



위의 그림의 직각삼각형에서 tan(θ) = 1/d이므로 d = 1/tan(θ)이다.

원근 투영 변환을 하는 함수 ProjectionTransform()은 다음과 같이 표현할 수 있다.

```
CVertex ProjectionTransform(CVertex vtxCamera)
{
   CVertex vtxProject;
   float d = 1 / tan(DegreeToRadian(gpCamera->fovAngle * 0.5f));
   vtxProject.x = vtxCamera.x / (vtxCamera.z / d);
   vtxProject.y = vtxCamera.y / (vtxCamera.z / d);
   vtxProject.z = vtxCamera.z;

   return(vtxProject);
}
```

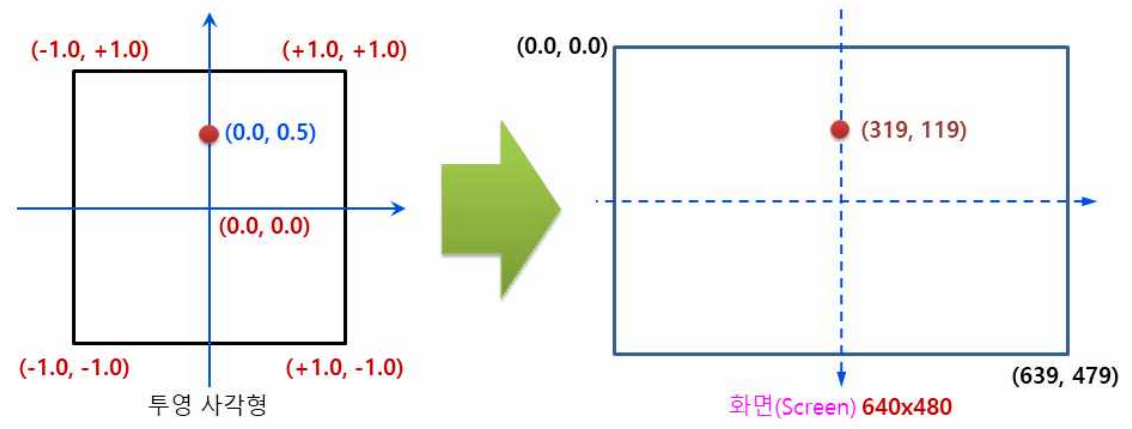
⑬ 화면 좌표 변환(Screen Space Mapping)

화면 좌표 변환은 투영 좌표 공간(투영 사각형)을 화면(스크린)으로 매핑하기 위한 변환이다. 화면 좌표 변환에서 주의할 점은 투영 좌표계의 y-축의 방향이 화면 좌표계의 y-축의 방향이 반대라는 것이다. 투영 사각형을 화면으로 매핑하기 위한 화면의 영역(사각형)을 뷰포트(Viewport)라고 한다. 뷰포트는 화면의 사각형 영역은 화면 좌표계(픽셀 좌표계)로 표현되며, 다음과 같이 사각형의 좌측 상단의 좌표와 사각형의 가로 길이와 세로 길이로 표현할 수 있다.

```
struct Viewport
```

```
{
   float left;
   float top;
   float width;
   float height;
};
```

다음 그림은 투영 사각형을 화면 전체로 매핑하는 것을 보이고 있다. 화면의 크기(해상도)가 640×480 픽셀이라면 뷰포트의 좌측 상단은 (0, 0), 뷰포트의 가로 길이는 640, 세로 길이는 480으로 설정하면 된다.



화면 좌표 변환을 하기 위하여 카메라의 투영 사각형이 매핑될 화면 영역을 표현하는 뷰포트를 카메라가 포함해야 한다. 앞에서 정의한 카메라 클래스에 뷰포트 구조체를 멤버 변수로 다음과 같이 추가한다.

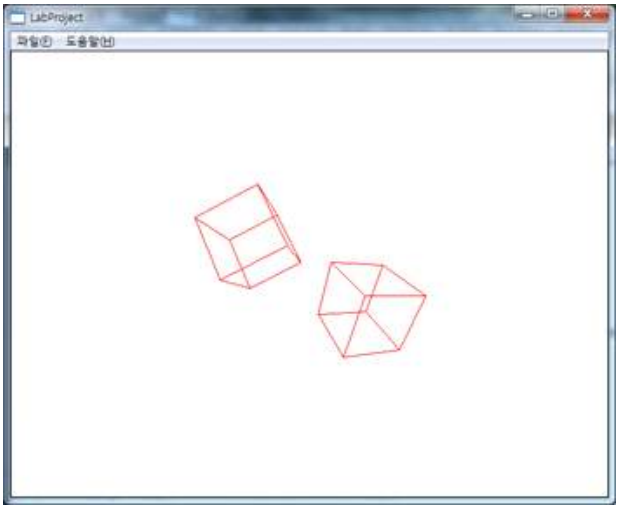
```
class CCamera
{
public:
   float        xPosition; //카메라의 위치(월드 좌표계)
   float        yPosition;
   float        xPosition;

   float        xRotation; //카메라의 방향(회전 각도)
   float        yRotation;
   float        zRotation;

   float        fovAngle;

   Viewport     viewport;
};
```

다음 그림은 투영 사각형을 어떤 윈도우의 클라이언트 영역 전체(화면)로 매핑하는 예이다. 이렇게 매핑하기 위하여 윈도우 프로그램에서 윈도우 핸들이 hWnd일 때 다음과 같이 뷰포트를 설정할 수 있다.



```
RECT rc;
::GetClientRect(hWnd, &rc);
gpCamera->viewport.left = rc.left;
gpCamera->viewport.top = rc.top;
gpCamera->viewport.width = rc.right - rc.left;
gpCamera->viewport.height = rc.bottom - rc.top;
```

화면 좌표 변환을 하기 위한 함수 ScreenTransform()은 다음과 같이 표현할 수 있다.

```
CVertex ScreenTransform(CVertex vtxProject)
{
   CVertex vtxScreen;
   float left = gpCamera.viewport.left;
   float top = gpCamera.viewport.top;
   float halfWidth = gpCamera.viewport.width * 0.5f;
   float halfHeight = gpCamera.viewport.height * 0.5f;
   vtxScreen.x = vtxProject.x * halfWidth + left + halfWidth;
   vtxScreen.y = -vtxProject.y * halfHeight + top + halfHeight;
}
```