CSED451 Assignment #3-2

3D DRAWING REPORT

Team: GPS

CSE 20202405 문민재 csmmj4594

CSE 20202728 김진수 fusion4268

**Overview of Program**

본 프로그램은 기존 assn3\_1에서 구현한 3D 추격 게임에서 OpenGL의 fixed pipeline이 아닌 GLSL을 이용하여 구현한 shader를 통해 동작하도록 수정하였다. floor는 이전과 동일하게 여러 선들의 조합으로 그려지며, 캐릭터와 벽은 자체적으로 구현한 원과 원기둥으로 그려진다. 원과 원기둥은 GL\_TRIANGLES로 구현하였다. 기존의 hierarchical model을 사용하기 위하여 matrxiStack을 구현하였다. OpenGL에서 제공하는 modelview matrix와 projection matrix를 glm에서 제공하는 함수와 자체적으로 구현한 matrixStack과 shader로 수정하였다.

원과 원기둥을 shader를 활용하여 직접 구현하는 방식을 선택하였다. vertex shader와 fragment shader를 구현하였으며, vertex shader는 position과 관련된 연산을 fragment shader는 color와 관련된 연산을 수행하는 역할을 한다. 실제로 객체를 그리는 draw 함수에서는 binding한 buffer에 data를 추가하고 buffer의 형식을 지정해줌으로써 glDrawElements 함수를 활용하여 그릴 수 있도록 구현하였다.

**Programming Environment**

Visual Studio 2019

OpenGL – 4.6

GLSL – 4.60

Freeglut-MSVC-3.0.0

Glew-2.1.0

Glm-0.9.9.7

**Design and Implementation**

1. 자료구조 설계
2. Object (object.h)

맵 상의 오브젝트를 정의하는 클래스이다.  
멤버 변수로 type, color, x, y, z를 가지고 있다. 이들은 각각 오브젝트 타입(0일 경우 wall, 1일 경우 character), 오브젝트의 색깔, 오브젝트를 그릴 때 필요한 x, y, z좌표에 해당한다.

1. Character: public object (character.h)

Object를 상속받아 맵 상의 캐릭터 오브젝트(플레이어, 도둑)를 정의하는 클래스이다.  
character 오브젝트들은 type이 1에 해당하며, x, y, z는 torso의 중심 좌표를 나타낸다. 그리고 원의 반지름을 나타내기 위한 변수인 rad와 벽과의 충돌 여부를 나타내는 변수인 isCollided가 있다. 그 외에 인간형 캐릭터 구현에 필요한 여러 멤버변수들을 가지고 있다.

1. treeNode (treeNode.h)

인간형 캐릭터의 hierarchy를 표현하기 위해 필요한 Left Child Right Sibling(LCRS) tree를 구현하는 클래스이다.

현재 modelView matrix의 오른쪽에 곱해질 기본적으로 matrix인 mtx,  
기본 mtx transform에 추가적으로 행해질 transform을 나타내는 additionalTransform,  
이 node에 해당하는 신체 부위를 그려주기 위한 함수 포인터인 draw,  
그리고 sibling과 child 정보를 가지고 있다.

1. pose (pose.h)

플레이어와 도둑의 포즈를 정의한 클래스이다.  
캐릭터의 포즈에 따른 변화는 상체에서만 이루어지므로, 왼팔과 오른팔 각각 2가지씩 총 4가지 각도와 포즈에 따른 색을 변수로 갖고있다.

1. camera(camera.h)

이번 과제에서는 3가지 시점을 구현했는데, 각 시점에 해당하는 카메라 정보(camera position, reference point, up vector))을 저장하는 클래스이다.

1. Floor(Floor.h)

바닥 역할을 하는 격자를 그리기 위한 클래스이다.  
floor vertices와 floor indices를 변수로 가지며, GL\_LINES를 이용하여 여러 선들로 격자를 형성한다. 격자의 색은 검정색으로 한다.

1. Cylinder(Cylinder.h)

Wall과 character의 limb, torso를 그리기 위해 원기둥을 정의한 클래스이다. 생성자는 baseRadius, topRadius, height, sectorCount, stackCount를 parameter로 전달받는다. baseRadius는 원기둥의 밑면의 반지름을 의미하며, topRadius는 원기둥의 윗면의 반지름을 의미한다. baseRadius와 topRadius가 다를 때, 윈기둥이 아닌 밑면과 윗면의 크기가 다른 기둥을 생성할 수 있다. heigh는 원기둥의 높이를 의미한다. sectorCount와 stackCount는 각각 원기둥을 그릴 때 사용되는 sector와 stack의 수를 의미한다. 여기서 sector는 경도를 기준으로 원기둥을 몇 부분으로 분할할지를 의미하며, stack은 위도를 기준으로 원기둥 몇 부분으로 분할할지를 의미한다. 원기둥의 vertices를 GL\_TRIANGLES를 이용해 그리기 위한 indices를 변수로 가지며, 5가지 색을 구현하기 위하여 색에 따른 vertices\_(color) 변수를 갖는다.

1. Sphere(Sphere.h)

character의 head와 joint를 그리기 위해 구를 정의한 클래스이다. 생성자는 radius, sectorCount, stackCount를 parameter로 전달받는다. radius는 구의 반지름을 의미하며, sectorCount와 stackCount는 각각 구를 그릴 때 사용되는 sector와 stack의 수를 의미한다. sector, stack 그리고 원의 색과 관련된 설명은 원기둥과 같다.

1. Wall: public object (Wall.h)

Object를 상속받아 맵 상의 벽을 정의하는 클래스이다.  
Wall 오브젝트들은 공통적으로 type이 0이며, x, y, z는 벽의 위치를 의미한다. 상황에 따라 총 5가지 종류의 벽을 그린다. 5가지 벽의 종류로는 1) 원기둥, 2) 윗면이 더 큰 원기둥, 3) 밑면이 더 큰 원기둥, 4) 원뿔, 그리고 뒤집어진 원뿔이 있다. 5가지 종류의 벽은 모두 cylinder의 baseRadius와 topRadius를 조절해 그릴 수 있다.

1. matrixStack.h

matrixStack은 OpenGL fixed pipeline이 가진 glPushMatrix()와 glPopMatrix()를 대체하기 위해 구현된 헤더이다. modelViewStack과 projectionStack으로 기존의 fixed pipeline과 같게 stack을 이용해 구현하였으며, matrix의 계산을 직접 수행할 수 있도록 mtxView와 mtxProj를 변수로 가진다. vertex shader에서 matrixStack.h의 projection matrix와 modelView matrix를 활용하여 실제로 그려질 위치인 gl\_position을 계산한다.

1. 기본 기능 구현(이전 ASSN 관련)
2. 벽의 생성 및 이동

Assn3\_1과 동일하게 구현되었다.

1. 충돌 판정

Assn3\_1과 동일하게 구현되었다.

1. 플레이어 캐릭터의 조작

Assn3\_1과 동일하게 구현되었다.

1. 치트 기능

Assn3\_1과 동일하게 구현되었다.

1. Pass/Fail 시의 action

Assn3\_1과 동일하게 구현되었다.

1. Jump 구현

Assn3\_1과 동일하게 구현되었다.

1. 3D 시점 구현

Assn3\_1과 동일하게 구현되었다.

1. 시점 변환 및 줌 인/아웃

Assn3\_1과 동일하게 구현되었다.

1. Shader 적용
2. Vertex Shader

이번 assn3\_2에서 사용한 vertex shader는 ‘vShader.glvs’에 정의되어 있다. Vertex shader는 input으로 좌표의 위치와 색상을 받아 fragment shader로 컬러를 넘겨주는 역할을 수행한다.

Input 변수로 aPos, aColor, output 변수로 ourColor이 있다. uniform 변수로는 modelView, projection이 있는데 이들은 각각 model-view matrix, projection matrix를 나타내는 mat4 타입의 uniform 변수이다.

Main 함수에서 input으로 들어온 aPos를 vec4로 바꾸어준 후 modelView와 projection matrix에 차례로 곱해준 값으로 gl\_Position을 설정한다. 컬러의 경우, 별도의 처리 없이 바로 output인 ourColor에 input인 aColor를 대입해주었다.

1. Fragment Shader

이번 assn3\_2에서 사용한 fragment shader는 ‘fShader.glfs’에 정의해 놓았다. 이번 과제에서 fragment shader는 픽셀의 컬러만 설정해주는 단순한 역할을 수행한다. input으로 vec3 타입의 ourColor를 받아, output으로 vec4 타입의 fragColor를 만들어내는데, 단순하게 ourColor에 vec4의 마지막 요소로 1.0을 넣어준 것뿐이다.

1. Applying shader in our program

기존 assn3\_1까지 programmable shader를 사용하지 않고, openGL에서 자체적으로 제공하는 그래픽스 파이프라인을 이용하여 프로그램을 구현하였었다. 이번 assn3\_2에서는 기존 openGL 자체 파이프라인을 벗어나 shader를 이용하기 위해 기존의 코드를 일부 수정하거나, 기존의 코드에 새로 추가해야 할 부분이 있었다.

* 1. Programmable shader 사용을 위한 준비 (main.cpp)

첫째로, 이번 assn3\_2에서 사용한 vertex/fragment shader는 파일 형태로 저장되어 있었기 때문에 셰이더를 사용하기 위해 assn3\_2의 최초 실행 시 두 파일을 읽어올 필요가 있었다. 그를 위해 readShaderSource 라는 함수를 정의하여 두 셰이더 파일을 읽어온 후 glCompileShader()를 이용하여 두 셰이더를 컴파일하였다.

둘째로, shader program을 정의하였다. glAttachShader 함수를 사용하여 우리가 사용할 shader program에 앞에서 가져온 vertex/fragment 셰이더를 붙이고, glLinkProgram 함수로 프로그램을 링크하고, 우리가 정의한 CheckProgram 함수를 통해 이 shader program이 정상적으로 링크되었는지 확인해주었다. 이와 같이 우리가 사용할 shader program을 정의한 후, 이번 assn3\_2에서는 프로그램 시작부터 종료까지 한 종류의 vertex/fragment shader만 사용하므로 쓸모를 다한 vertex shader와 fragment shader 오브젝트는 glDelteShader를 이용해 제거해주었다.

마지막으로, Vertex Array Object(VAO), Vertex Buffer Object(VBO), Element Buffer Object(EBO)를 정의하였다. Main.cpp에서는 init() 함수에서 glGenVertexArray 함수를 이용하여 VAO를, glGenBuffers 함수를 이용하여 VBO와 EBO를 정의하고, display3D() 함수에서의 binding만 이루어진다. 이들의 실제 vertex attribute linking은 object를 그릴 때 필요한 기본 도형을 그려주는 Cylinder.cpp와 Sphere.cpp에서 이루어진다.

* 1. Glm을 이용한 Matrix 연산

이번 assn3\_2에서는 기존 openGL의 fixed pipeline을 사용할 수 없기에, glBegin(), glEnd(), glPushMatrix(), glPopMatrix(), glTranslatef(), glRotatef() 등의 함수를 사용할 수 없었다. 따라서, Matrix 연산을 위해서 기존의 matrix 계산 체계를 glm을 사용한 체계로 고칠 필요가 있었다.

우선, 이전 assn3\_1까지 사용했었던 행렬 연산 함수들 중 glm으로 대체 가능한 모든 행렬 함수를 대체하고, matrix stack은 matrixStack.h에 기존 fixed pipeline 상에서의 matrix stack과 동일한 기능을 수행하는 pushMatrix(GLenum mode)와 popMatrix(GLenum mode) 함수를 구현하여 기존 matrix stack을 사용하는 컨셉을 유지하였다.

1. Vertex Array를 이용한 오브젝트 구현
   1. 캐릭터

캐릭터는 구와 원기둥의 조합으로 구현된다. 캐릭터의 head와 joint는 구 torso와 limb는 원기둥으로 구현하였다. class내의 private 변수로 head, joint, limb, 그리고 torso가 추가되었으며, draw 함수가 원과 원기둥은 parameter로 전달받으면서 그리고자 하는 model을 그릴 수 있도록 구현하였다.대부분이 기존 assignment3\_1과 동일하며, glut를 이용하여 cylinder와 sphere를 그리는 부분이 자체적으로 구현한 GLSL로 그릴 수 있도록 수정되었다. Hierarchical model은 기존의 glRotatef, glTranslatef 함수가 glm::rotate 함수와 glm::translate 함수로 대체되었다.

* 1. 벽

벽의 종류는 Default와 4가지 포즈에 따른 서로 다른 모양과 색의 4가지 벽, 총 5가지 종류의 벽을 구현하였다. Wall class는 cylinder, cylinder1, cylinder2, cone, 그리고 reverse cone을 클래스의 private 변수로 가진다. 벽이 가진 각각의 모양과 색을 일대일 대응 관계이다. 벽은 cylinder.h의 topRadius와 baseRadius 값을 변화시키며 구현하였으며, 모든 벽의 높이는 동일하다. 벽의 윗면과 밑면의 반지름, 그리고 색은 다음 표를 따른다.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Cylinder | Cylinder1 | Cylinder2 | Cone | ReverseCone |
| baseRadius | 27.5 / 2 | 27.5 / 4 | 27.5 / 2 | 27.5 / 2 | 0 |
| topRadius | 27.5 / 2 | 27.5 / 2 | 27.5 / 4 | 0 | 27.5 / 2 |
| Color | gray | yellow | blue | red | green |

**How to Run**

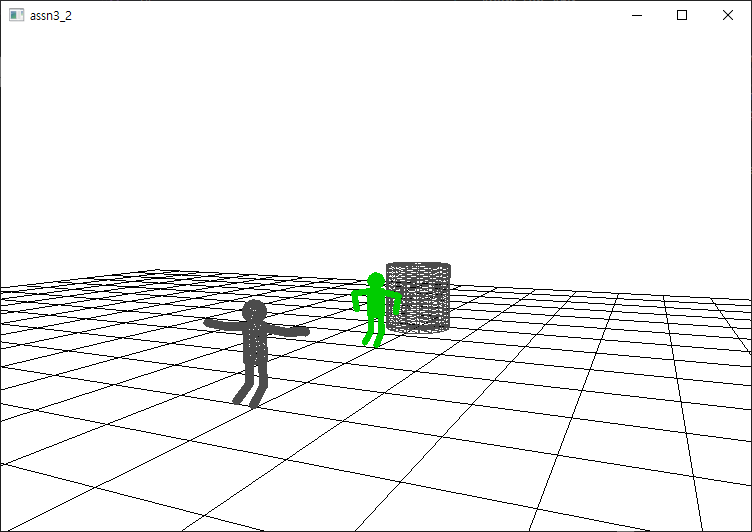
프로젝트 폴더의 \bin\x64 폴더로 이동하여 graphics\_assn3\_2.exe를 실행시킨다.

캐릭터의 포즈 변경은 방향키로 입력하며, 위, 아래, 왼쪽, 오른쪽 방향키는 각각 플레이어 캐릭터를 적색, 녹색, 청색, 황색으로 변경시키고, 플레이어 캐릭터는 각 색에 대응하는 포즈를 취한다. 스페이스바를 입력하여 캐릭터를 점프시킬 수 있다.

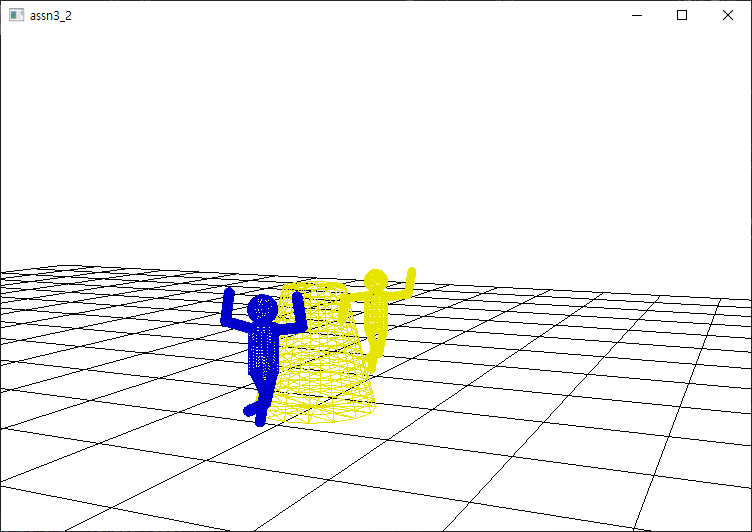
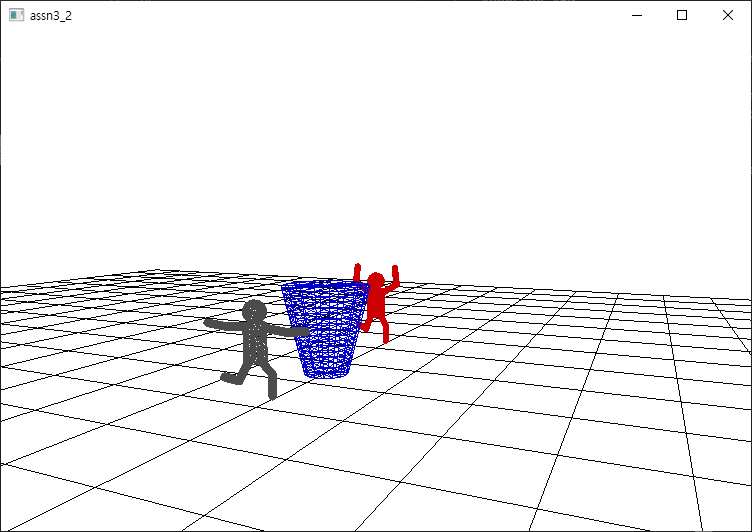
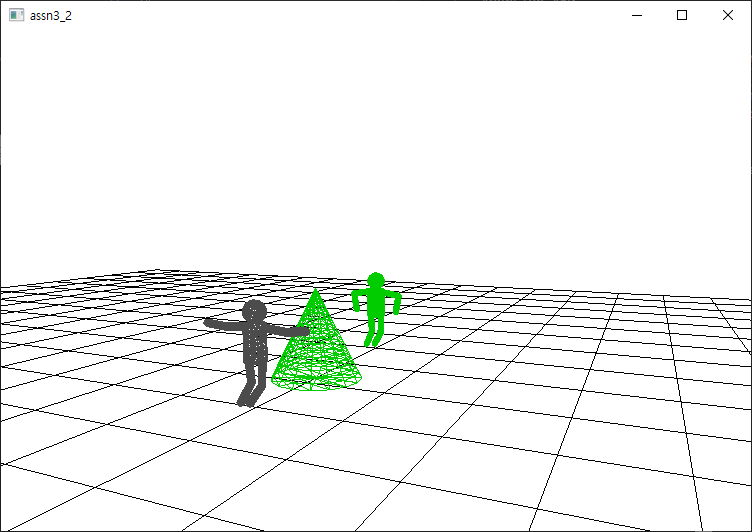
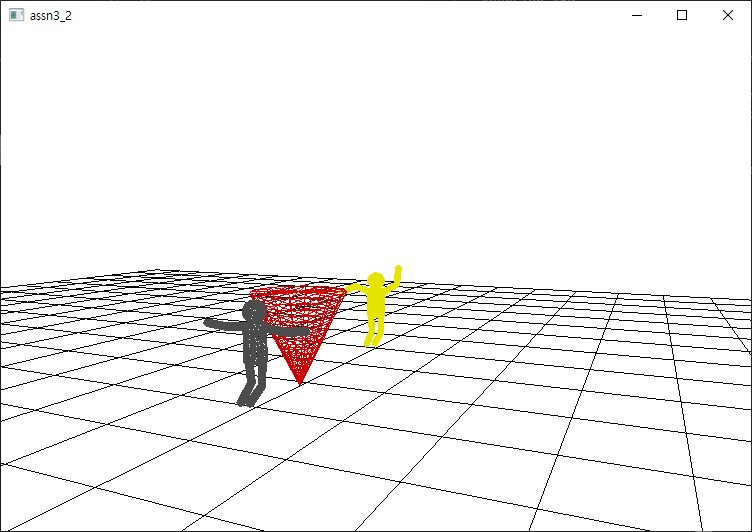
숫자키 1번을 눌러 1인칭 시점으로, 3을 눌러 3인칭 시점으로, 9를 눌러 XY 평면을 바라보는 시점으로 시점을 전환시킬 수 있다.

**Example**

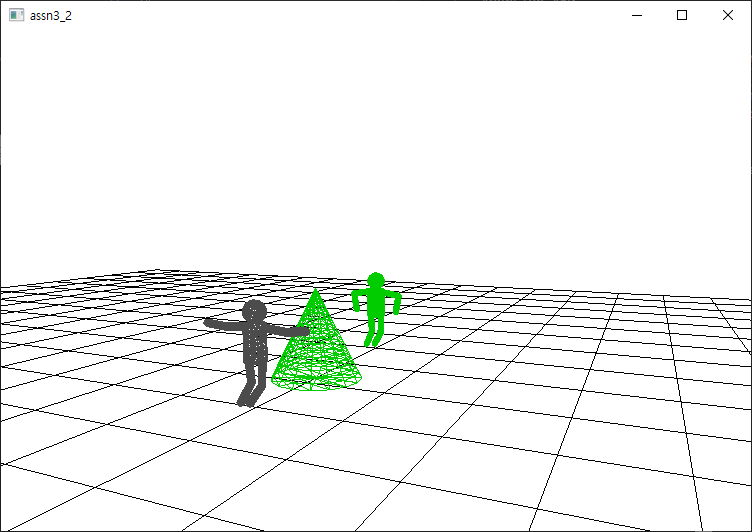
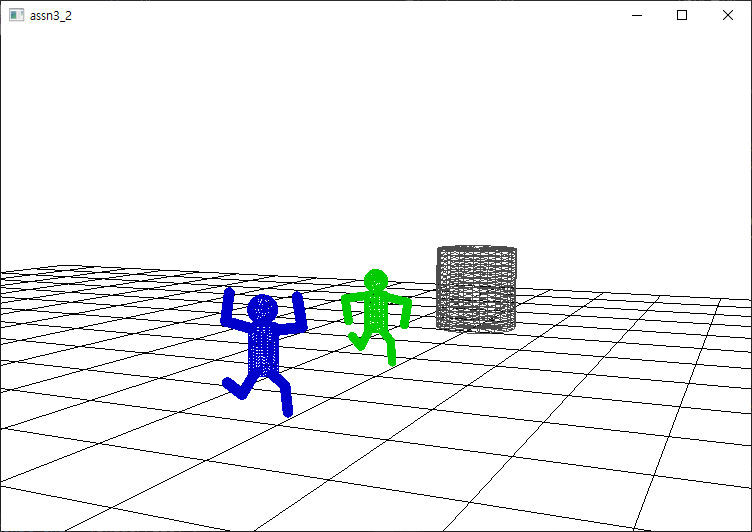
기존 Assn2와 중복되는 부분은 제외하였다.



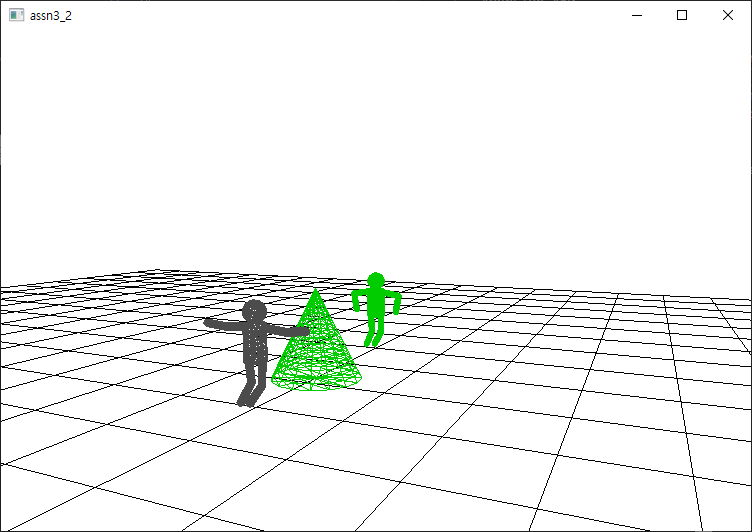
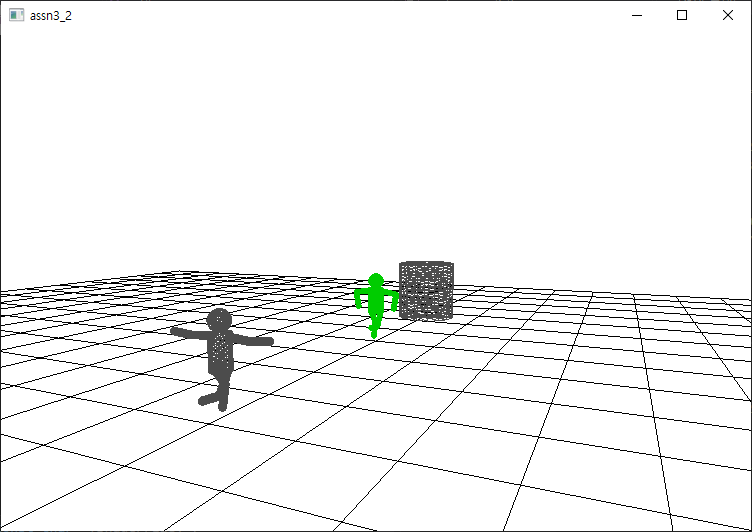
프로그램 실행 시 화면으로, 카메라가 플레이어의 오른쪽 뒤에 있는 3인칭 뷰로 시작한다. 벽은 기본적으로 회색 원기둥 모양이다.



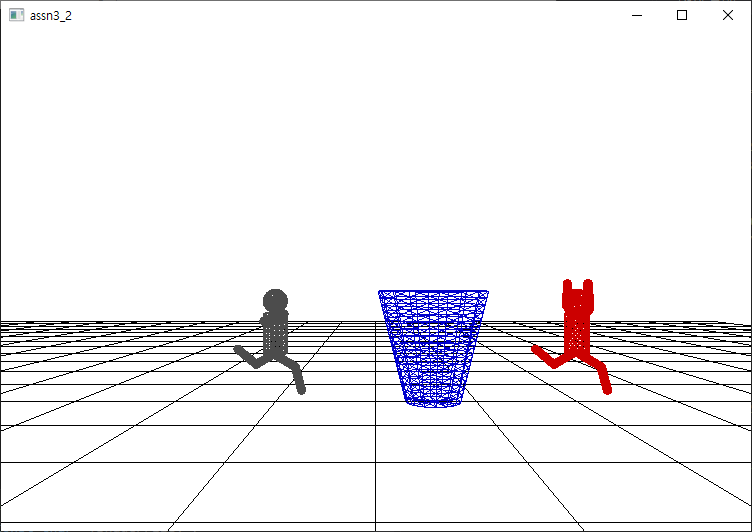
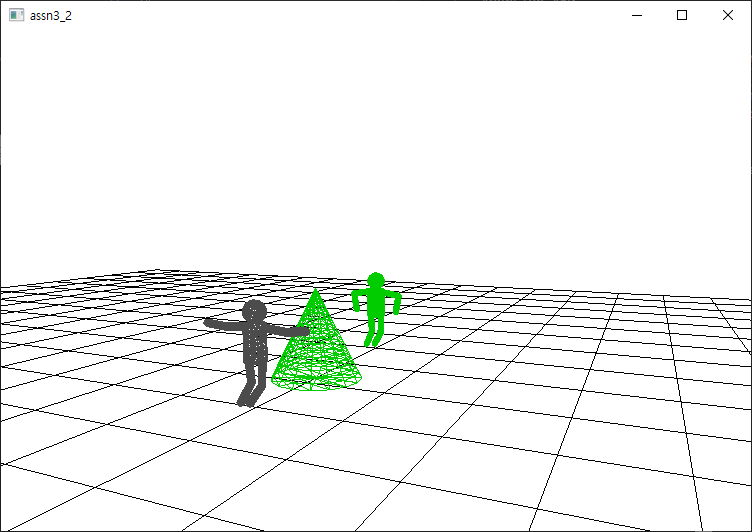
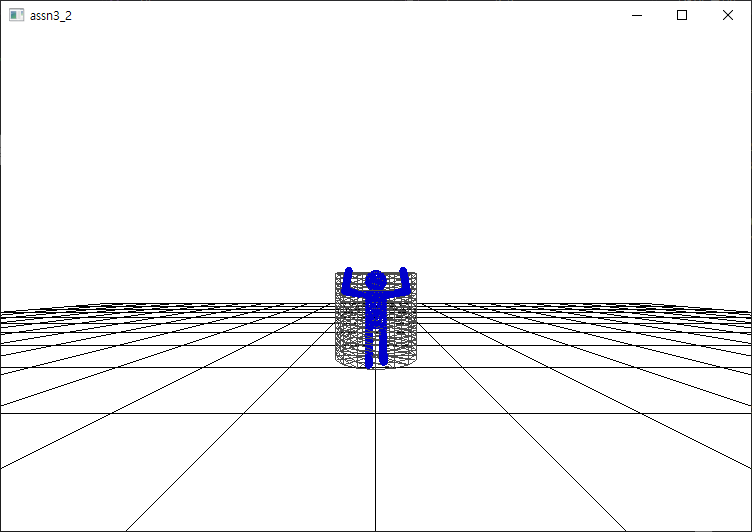
도둑의 색과 자세는 일정시간마다 랜덤하게 변경되며, 도둑 색의 변경 간격은 콘솔에 Pass가 출력되는 횟수가 많아질수록 줄어든다.  
도둑과 벽이 충돌할 경우 벽의 색이 도둑의 색과 그 색에 해당하는 모양으로 변경되어 플레이어에게 다가온다. Red – Reversed cone, Green – Cone, Blue – Reversed frustum of cone, Yellow – frustum of cone(원뿔대) 모양이다.

 🡪 

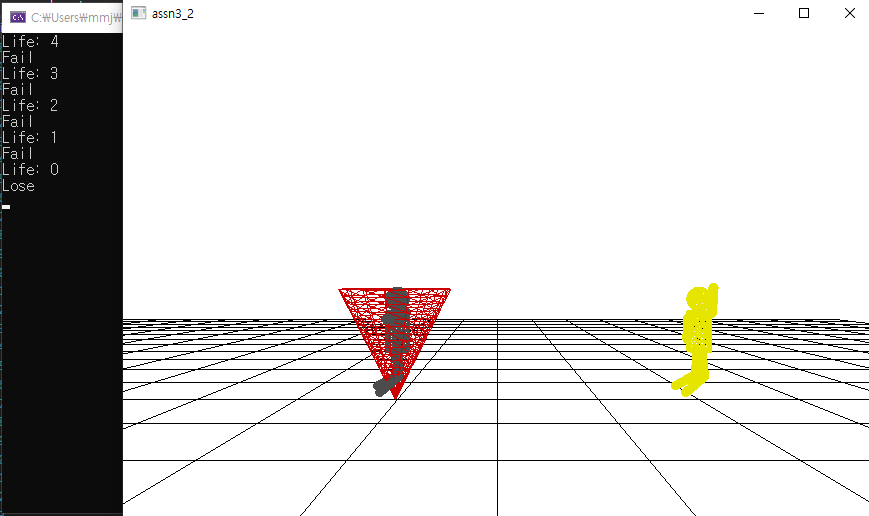
플레이어가 벽을 점프하지 않고 성공적으로 통과할 경우, 플레이어가 도둑에게 조금 더 다가가고, 시야각이 좁아져 조금 더 좁은 화면을 비추게 된다. 1인칭 뷰, 3인칭 뷰 모두 해당된다.

 🡪 

도둑이 점프하지 않았는데 플레이어만 점프하여 벽을 통과할 경우, 점프로 인해 플레이어가 도둑으로부터 멀어지게 되므로, 시야각을 넓혀 조금 더 넓은 화면을 비추게 된다. 1인칭 뷰, 3인칭 뷰 모두 해당된다.



숫자 키 ‘1’을 눌러 1인칭 뷰로 변경할 수 있고, ‘3’을 눌러 3인칭 뷰로 변경할 수 있다.  
추가적으로 ‘9’를 눌러 XY평면을 보는 뷰로 전환할 수도 있다.



이번 assn3\_2에서는 기존 assn3\_1까지 존재하던 화면 life 표시 기능이 빠졌다. 대신, 플레이어가 벽 통과에 실패할 경우 콘솔 상에 남은 life가 출력된다.

**Discussion**

1. Shader 적용
2. Vertex/Fragment Shader

이번 assn3\_2에서 사용될 shader를 구현할 때, vertex의 위치와 색깔 정보를 어떻게 표현할지에 대해 고민하였었다. 이번에 사용한 shader들은 programmable shader이기 때문에, 프로그래머의 의지에 따라 같은 vertex라도 shader를 어떻게 정의하느냐에 따라 그래픽스 pipeline 상에서 shader들을 통과하며 다르게 나타날 수 있기 때문이다.

Vertex position의 경우, vertex shader에서 처리하였는데, 이전 그래픽스 수업의 3Dviewing 챕터에서 배운 model-view matrix와 projection matrix를 적용시키고자 하였다. 이를 위해 vertex shader 내에 model-view matrix와 projection matrix를 나타내는 uniform 변수를 선언하였고, assn3\_2 프로그램 내에서 이 두 종류의 matrix 값을 변경하면서 오브젝트 그리기 및 시점 변환을 표현하였다.

Vertex color의 경우, vertex shader를 거쳐 최종적으로 fragment shader에서 처리되는데, 색깔에서 복잡한 작업을 처리하기보다는, input으로 들어온 색 정보를 그대로 출력하게 하여 가장 기본적인 기능의 fragment shader를 구현하였다.

1. Vertex Array를 이용한 object 구현
2. SectorCount와 StackCount

SectorCount와 StackCount는 구와 원기둥을 GL\_TRIANGLES를 이용하여 그릴 때 곡면을 얼마나 여러 개의 삼각형으로 나타낼 것인지 결정한다. Sector와 stack을 많은 개수로 그릴수록 도형의 곡면을 더욱 실제 곡면과 가깝게 나타낼 수 있다. 그러나, 너무 큰 값으로 설정할 경우 계산량이 지나치게 많아져 불필요한 계산을 진행하게 된다. Assignment3 요구사항에 따르면 머리는 적어도 일부분은 부드러운 표면을 가지고 있어야 하며, 정점의 개수는 1000개를 넘지 않아야 한다. 정점의 개수가 1000개를 넘지 않는 최대한 구와 가까운 도형을 그리기 위하여 sector와 stack의 개수를 모두 16개로 설정하였다. 모두 16개로 설정하였을 때, vertex의 개수는 (3+4\*14+3)\*16 = 992개로 1000보다 작은 stack과 sector가 모두 10이상인 최대값을 갖는다.

1. 캐릭터

기존의 glut를 이용하여 원기둥과 구를 그리는 방식과 자체적으로 구현한 Cylinder.h와 Sphere.h를 이용하여 그리는 방식이 달라 기존에 구현된 부분에서 수정이 필요했다. Debugging할 때 많은 시간이 소요된 부분은 glRotatef와 glm::rotate에서의 차이에서 발생했다. glm::rotate는 glRotatef와 달리 각도 parameter를 radian으로 전달 받으며 degree로 전달할 경우 원하지 않는 각도로 회전하는 현상이 발생하였다. 이는 character를 그리는 부분 뿐만 아닌 camera의 fovy에서도 비슷한 현상을 보였으며, OpenGL 프로그래밍 시 유의해야할 점이라는 것을 알 수 있었다.

1. 벽

다양한 형태의 벽을 구현하기 위해 여러 개의 원기둥이나 원을 붙여 형성하는 방향도 고려되었지만, 비효율적이라고 생각되었다. 기존의 cylinder.h와 sphere.h로 다양한 형태의 벽을 생성할 수 있는 방법을 고려하였고, cylinder의 topRadius와 baseRadius 값을 이용해 여러 형태의 벽을 그리는 방법을 선택하였다.

**Conclusion**

1. Shader 적용

이번 assn3\_2에서는 openGL에서 제공하는 fixed pipeline만을 이용해 작성한 프로그램을 programmable vertex/fragment shader를 적용하여 user-defined pipeline을 적용하여 만들어보았다. 이를 통해, programmagle shader의 적용법을 알게 되었고, programmable shader를 사용하면 프로그램 작성자가 원하는 대로 그래픽스 파이프라인을 설정할 수 있음을 알게 되었다. 또한 assn2에서 처음 사용해 본 glm을 모든 행렬 연산에 사용함으로써 기존 fixed pipeline에서 사용하던 행렬 연산을 glm으로 대부분 대체할 수 있음을 알게 되었고, glm 자체의 사용법 또한 더 자세히 알게 되었다.

1. Vertex Array를 이용한 object 구현

원과 원기둥을 그릴 때 사용되는 vertices와 indices를 직접 할당하여 그림으로써 vertex shader와 fragment shader의 내부 작동방식을 이해할 수 있었다. 비교적 쉽게 그릴 수 있는 평면으로만 이루어진 도형과는 달리 부드러운 면이 있는 도형을 그릴 때 stack과 sector를 활용하여 분할하여 그린다는 것을 알게 되었으며 그 작동 원리를 이해할 수 있었다.  
또한 vertex array의 작동 과정을 직접 구현해볼 수 있었으며, 서로 다른 크기의 vertex와 index를 사용하더라도 그 attribute를 설정함으로써 원하는 동작을 수행할 수 있도록 구현하는 방법을 알게 되었다.  
position을 결정하는 vertexArray 뿐만 아닌 fragment shader의 input으로 사용되는 color vertices를 구현하면서 기존의 OpenGL fixed pipeline을 사용할 때 보다 더 다양한 색을 사용하는 방법을 알 수 있었다.

**Direction of Improvement**

1. 벽의 원근감 부족

현재 구현된 프로그램에서는 벽이 다가올 때의 원근감이 부족하다. 이러한 단점은 특히 1인칭 시점에서 두드러지는데, 벽의 모양만 잡혀있고, 벽의 테두리를 알려주는 외곽선이 없고, 또한 벽의 색이 캐릭터(도둑, 플레이어)의 색과 정확히 일치하기 때문에 이에 파생하여 캐릭터와 벽이 겹치면 잘 보이지 않는 등의 문제가 발생한다.

하지만, 광원을 추가하여 벽과 광원 사이의 관계에 따라 빛이 달라져 색이 조금씩 달라지거나, 벽의 색을 캐릭터의 색과 조금 다른 색으로 설정하거나, 벽에 외곽선을 추가해준다면 벽의 원근감을 조금 더 살릴 수 있을 것으로 보인다.

1. 벽의 갑작스럽고 어색한 모양 변화

현재 구현된 프로그램에서는 도둑과 벽이 충돌하여 벽의 모양이 급작스럽게 바뀌어 부자연스럽다. 특히, 현재 assn3\_2에서는 벽에 캐릭터 모양의 구멍을 뚫는 것이 아닌, 벽의 모양을 임의의 모양으로 바꾸는 것이기 때문에 벽의 모양이 갑작스럽게 변하는 것이 특히 어색하게 느껴진다.

이 문제는 벽의 모양 변화에 애니메이션을 넣어 시간을 두고 조금 더 자연스럽게 벽의 모양이 바뀌게 하거나, 도둑과 벽이 충돌할 때 도둑의 포즈 모양으로 벽에 구멍을 뚫어주는 방식으로 구현한다면 해결될 수 있을 것으로 보인다.

1. Shading

현재 구현된 프로그램과 실제 그래픽 모델의 가장 큰 차이는 shading의 유무로 고려된다. texture와 shading이 없는 wireframe으로 구현된 모델이다 보니 model의 볼록한 면과 오목한 면을 구분하기 힘들다. 또한 shading이 구현되지 않아 빛이 없는 상황이라고 광원이 정해지지 않은 모든 부분이 지나치게 밝다. shading을 구현한다면, character의 애니메이션 생동감을 향상시킬 수 있을 것으로 보여진다.

1. Color

현재 구현된 캐릭터와 벽은 모두 단색으로 구성되어 있다. 심미적인 부분을 보완하기 위해서는 단색이 아닌 여러 색의 조합으로 객체를 구성할 수 있다. fragment shader의 input으로 사용되는 color vertices를 여러 색의 조합으로 구현한다면 프로그램의 심미성을 보다 높일 수 있을 것이다.

**참고문헌**

<https://alleysark.tistory.com/260>: 이번 과제의 구현에서, 파일로 된 vertex/fragment shader를 string으로 읽어오는 코드(readShaderSource 함수)와,  
program에 vertex/fragment shader를 attach 완료한 후 program을 점검하여 문제가 있을 경우 로그를 띄워주는 CheckProgram 함수를 참고하였다.

<https://heinleinsgame.tistory.com/7>: GLFW에서 shader를 적용시켜 삼각형 하나를 출력하는 기본적인 코드로, 이번 과제에서 shader를 사용하기 위한 setup(VAO, VBO, program setting 등)을 참고하였다.

<https://github.com/g-truc/glm/blob/master/manual.md#section5>: glm으로 기존 openGL의 fixed pipeline 상에서의 matrix 계산을 대체하는 방법을 참고하였다.

<http://www.songho.ca/opengl/gl_cylinder.html> - Cylinder

<http://www.songho.ca/opengl/gl_sphere.html> - Sphere

Cylinder와 Sphere를 구현하는 부분의 위의 링크를 참고하여 구현하였다. 위의 링크에서는 현재 프로그램에서 구현되어 있는 vertices와 indices 뿐만 아닌 normal vectices와 texture까지 구현한다. 과제 프로그램은 이를 간소화하여 vertices와 indices를 구하는 부분만 프로그램에 추가하였다.