
컴퓨터그래픽스

과제 4 보고서



세종대학교

제출일	2025.12.01	전공	소프트웨어학과
과목	컴퓨터그래픽스	학번	21011746
담당교수	박상일 교수님	이름	양현석

1. Simple Obj Model Viewer

a) 문제 분석

.obj 파일을 읽고, 이를 화면에 그려내는 프로그램을 작성하는 과제이다. .obj 파일에는 v라인과 f라인이 존재한다. 각각 vertex의 좌표와 face(triangle)의 vertex를 가지고 있다. 이후 이를 바탕으로 Shader로 보내어 화면에 그려내면 된다. 또한 Perspective Projection을 통해, 원근감 있게 화면에 그려지게 해야 한다. 추가적으로 Phong, Flat Shading을 동시에 지원한다.

과제의 주 목적이라고 생각이 드는 것은 Specular, Shininess의 변화를 관찰하여 각 값이 물체를 그려내는 데에 어떤 효과를 주는지 확인하고자 함이다. 이를 관찰할 수 있는 추가적인 입력을 지원하도록 구성한다.

b) 문제 풀이 및 학습 내용

1) .obj 파일 읽고, MyObject 클래스 구성하기

.obj 파일은 텍스트로 구성되어 있기 때문에, C++의 FILE 구조체를 통해 일반 텍스트 파일과 같이 읽어올 수 있다. MyObject라는 클래스(별도의 헤더)를 만들어, .obj 입력, 그리기 등의 역할을 수행하도록 한다.

먼저 .obj 파일을 입력을 받을 수 있는 `bool MyObject::ReadFile(const std::string filePath)`이다. `main.cpp`를 통해 파일 경로를 입력 받고, 파일이 존재하여 읽을 수 있다거나 이미 파일을 읽어온 적이 있다면 `true` 반환, 그렇지 않다면 `false`를 반환한다. `false`를 반환 받으면 `main.cpp`에서 `while loop`를 통해 입력을 다시 받는다.

파일의 길이가 굉장히 길 수 있기에, 여러 번 읽어오게 되면 물체를 그리기까지의 시간이 오래 걸릴 수 있다. 따라서 여러 개의 `vector`를 이용하여 한 번의 `Read`를 통해 물체를 생성할 수 있도록 한다. v라인(Position List), f라인(Triangle List), Vertex Normal 누적용(Normal List), 최종 Vertex에 대한 정보(Vertex List), 총 네 개의 `vector`를 사용한다.

최종 Vertex는 구조체로써 구성한다. `MyObjectVertex`는 Vertex의 position, color, vertex normal, face normal을 저장한다. 이를 VBO에 넣어주는 것이다. Vertex normal은 본 Vertex가 구성하는 삼각형(Surface)들의 normal의 평균을 저장하기 때문에, f라인을 모두 읽어오고 결정할 수 있다. 따라서, 나머지 값들은 바로 결정할 수 있지만, vertex normal은 normal list의 해당하는 값에 계속 더해주고 실제 Vertex List의 Vertex Normal에 저장할 때 `normalize(정규화)`하여 갱신할 수 있도록 한다.

파일은 `char` 배열로 한 줄씩 읽어온다. 거기서 맨 앞 글자가 'v'이거나 'f'일 때만 처리할 수 있도록 구성했다.

v인 경우에는 Vertex의 Position 정보를 가지고 있기에, 이 정보를 Position List에 추가한다. 또한 각 Vertex에는 Normal 정보가 갯수가 동일하게 들어가야 하기에 Normal List에 vec3(0.f)를 넣어 추가함과 동시에 초기화한다.

f인 경우에는 삼각형(Surface)를 이루는 3개의 Vertex Index를 가지고 있다. 이들을 읽어와 makeTriangle() 함수로 넘겨준다. makeTriangle()에서는 각 Index를 통해 Position List에 저장되어 있는 Vertex 정보를 가져온다. 이들을 이용하여 Triangle의 Normal을 계산(a를 기준으로 수행)한 후에 각 Index에 맞는 Normal List에 더한다. 여기서 평균을 계산하지 않고 더하는 이유는 앞서 말했듯이 이후에 Normalize를 진행하기 때문이다. 크기가 모두 더해서 크기가 1인 벡터를 만들어 주는 것을 평균 내기라 표현했다. 이후 삼각형 정보(3개의 Index, Surface Normal)을 Triangle List에 저장하고, Vertex List에도 세 개의 Vertex를 추가한다.

```
void MyObject::makeTriangle(const int idx1, const int idx2, const int idx3)
{
    vec3 a = PositionList[idx1 - 1];
    vec3 b = PositionList[idx2 - 1];
    vec3 c = PositionList[idx3 - 1];

    vec3 n = normalize(cross(b - a, c - a));

    normalList[idx1 - 1] += n;
    normalList[idx2 - 1] += n;
    normalList[idx3 - 1] += n;

    triangleList.push_back({ idx1 - 1, idx2 - 1, idx3 - 1 });

    vertexList.push_back({ vec4(a, 1), color, vec3(0), n });
    vertexList.push_back({ vec4(b, 1), color, vec3(0), n });
    vertexList.push_back({ vec4(c, 1), color, vec3(0), n });
}
```

현재는 MyObjectVertex에 Vertex Normal(normal) 정보를 제외하고 나머지 정보들만 들어가 있는 상태이다. 파일을 모두 읽은 후에, Normal List에 저장되어 있는 각 Vertex를 포함하고 있는 삼각형들의 Normal의 합을 Normalize해준다. 이후 이들을 MyObjectVertex에 넣어주어야 한다. 이때 Triangle List를 통해 각 f라인을 다시 참조하여 각 Vertex Index의 Normal List에 있는 값을 Vertex List의 각각의 normal 값에 대입해준다.

```
/*
 * 읽어들이는 순서대로 삼각형을 순회(f 기반 인덱싱)하면서 VertexList의 normal에
 * normalList에 들어있는 값으로 채워준다. (이는 v 기반 인덱싱)
 */
int idx = 0;
for (auto& t : triangleList)
{
    vertexList[idx].normal = normalize(normalList[t.idx1]);
    vertexList[idx + 1].normal = normalize(normalList[t.idx2]);
    vertexList[idx + 2].normal = normalize(normalList[t.idx3]);
    idx += 3;
}
```

이제 Init(), SetPositionAndOtherAttributes(), Draw()함수를 구성한다. Init() 함수는 VAO, VBO 생성 및 Bind를 수행한다. SetPositionAndOtherAttributes() 함수는 Shader의 in 변수들을 vector와 연결한다. Draw()는 그리기 자체를 수행한다.

이렇게 그려본다면, 각각의 오브젝트가 서로 각기 다른 크기로 그려짐을 확인할 수 있다. 따라서 각 오브젝트의 경계 박스를 만든다. 이후 Width, Height를 모두 고려할 수 있도록 대각선 크기를 재고 Scale Transformation을 통해 사이즈를 조절할 수 있도록 한다. 경계 박스 정보는 일전의 v라

인을 읽어올 때 함께 수행하며 leftBottom, rightTop으로 최대 최소 값을 저장하도록 한다. leftBottom은 vec3(1000)으로 rightTop은 vec3(-1000)으로 초기화하여 단순한 비교를 통해 갱신한다. 이 둘 사이의 거리가 오브젝트의 대각선 거리이며, 또한 이 둘의 평균 점이 오브젝트의 중심일 것이다. 이 대각선 거리와 중심 좌표를 이용해 main.cpp에서 각 도형의 스케일과 그려지는 위치를 조정한다.

2) Display

도형을 원근감 있게 그리기 위해서 Perspective Projection을 만들어야 한다. 또 동시에 카메라 위치를 잡기 위해 Look-At Function을 구성한다. 이들은 수업 시간에 사용한 코드를 바탕으로 구성하였다.

이제 마우스 클릭을 통한 회전 전환을 구현해야 한다. 좌클릭, 중클릭, 우클릭을 통해 각각 x, y, z 축 기준으로 회전을 실시한다. Enum을 통한 상태를 관리하도록 한다. 클릭시 curState가 바뀌고 idle() 호출 때마다 각 상태에 맞는 각도가 돌아갈 수 있도록 하는 것이다. 따라서 축별 각도 값을 따로 전역 변수로 저장하고, idle에서 알맞은 상태에 해당하는 축 각도에 Delta Angle을 더해준다. 추후에 축별 각도를 RotateX(Y, Z)로 Model Matrix에 적용하면 된다.

Object를 그리기 위해서, Shader에서 필요한 추가적인 Uniform 변수를 넘겨준다. Model Matrix, Projection Matrix, Specular, Shininess 그리고 Vertex, Surface Normal 모드를 전환하기 위한 NormalMode 변수도 넘겨주도록 한다. Object를 그리기 위해 사용하는 프로그램은 vPhong.glsl, fPhong.glsl이다.

vPhong.glsl은 MyObject를 통해 in 변수들과 main.cpp에서 계산된 uniform 변수를 넘겨받는다. 여기서는 각 Vertex의 위치를 Transformation Matrix들을 이용해서 계산하여 gl_Position을 결정한다. uNormalMode으로, vNormal(Vertex Normal)과 vFaceNormal(Surface Normal) 중 사용할 Normal을 결정한다. 결정된 Normal은 uModelMatrix로 Transformation하여 이를 fPhong.glsl으로 보낸다. 또 Projection을 고려하지 않고 Model Matrix만을 고려한 Position과 입력 받은 vColor를 추가로 fPhong에 보낸다.

fPhong.glsl에서는 vShader를 통해 입력 받은 color, pos, normal과 uniform 변수인 uSpc, uShininess를 통해 Phong 혹은 Flat Shading을 수행한다. 빛의 위치와 색상(세기), ka는 임의로 지정하고, kd는 넘겨 받은 color 값을 사용하도록 한다.

$$I = k_a I_a + k_d I_l (N \cdot L) + k_s I_l (R \cdot V)^n$$

위 공식에서 I_a , I_l 값은 빛의 색상을 사용하고 계산한다. 또한 k_s 와 n 은 외부에서 값을 변경할 수 있으므로 기존에 shader 내부에서 지정해주던 방식이 아닌 uniform으로 받은 것을 알 수 있다.

Ambient Light는 설정한 k_a 와 LColor를 곱하여 구한다. Diffuse Color는 다음과 같이 구한다. 현재

위치에서 빛으로 향하는 벡터를 L3(vec3)에 저장하고, Normal 값을 Normalize하여 N3(vec3)에 저장한다. 이후 이들을 내적하는데 음수가 되면 물체 안쪽을 향한 빛을 구하는 것이 되어버리므로 max()를 이용한 Clamp를 적용하여 NL 값을 만든다. 이후 color 값인 kd와 LColor, NL 값을 곱해 결정한다. 마지막으로 Specular Light를 구한다. 먼저 법선 벡터인 Normal과 빛 방향인 L3를 내적하고 법선벡터를 곱하여 법선 벡터에 대한 Projection을 수행한다. 이후 이 벡터의 2 스칼라 배에서 빛 방향인 L3를 빼준다면 빛의 반사 방향인 R3를 구할 수 있다. 현재 위치에서 바라보는 위치까지의 벡터인 V3를 구하여 마찬가지로 내적을 하여 cosine값을 구한다. 그 다음 이 값을 uShininess만큼 거듭제곱을 해주면 RV를 구할 수 있다. uShininess 값에 의해 빛나는 정도가 결정된다. 이제 uSpc, LColor와 위에서 구한 RV를 곱해 Specular를 결정한다.

위에서 구한 Ambient, Diffuse, Specular를 모두 더해 fColor로 넣어주면 fPhong.glsl에서의 색상 결정이 모두 끝나게 된다.

이제 Model Matrix를 구하는 부분이다. 위에서 설정된 축별 각도와 Object의 중심 및 대각선 길이와 View Matrix를 고려하여 다음과 같이 작성한다. Shader 내부에서는 편의를 위하여 Model Matrix값을 사실상 View * Model로 생각하고 사용한다.

```
// View, Model, Projection Matrix
mat4 ViewMat = myLookAt(vec3(0, 0, 2), vec3(0, 0, 0), vec3(0, 1, 0));

mat4 ModelMat = ViewMat * RotateZ(g_ZAngle) * RotateY(g_YAngle) * RotateX(g_XAngle)
    * Scale(sqrt(2) / object.dist, sqrt(2) / object.dist, sqrt(2) / object.dist)
    * Translate(-object.mid);

mat4 ProjMat = myPerspective(45, g_aspect, 0.01, 10.0f);
```

3) Axis 그리기

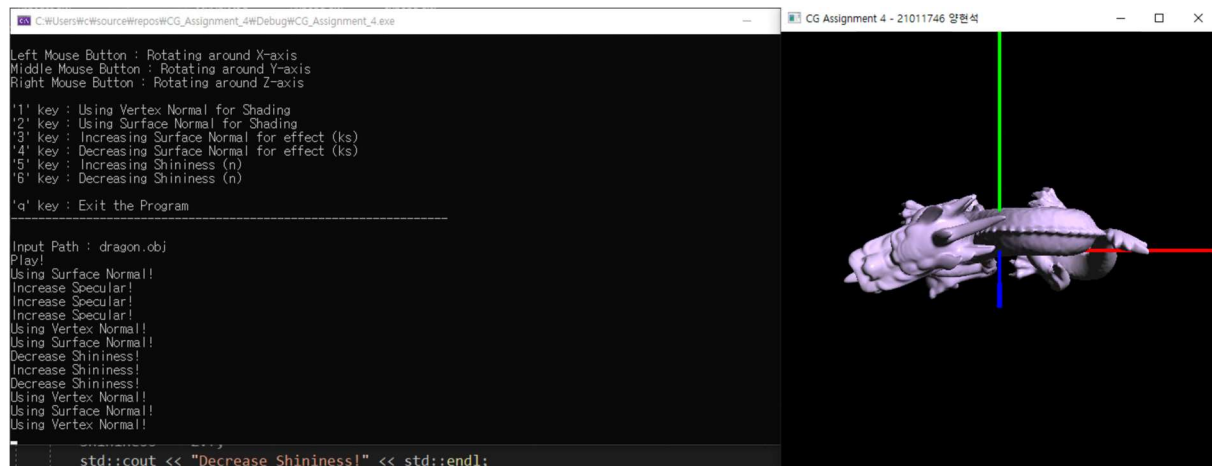
각 축을 그리기 위해 기존 수업시간에 사용하던 MyCube.h와 또 다른 Shader인 fShader.glsl, vShader.glsl를 사용한다. 색상과 Model Matrix를 uniform 변수로 Shader에 보내어 그리고, Object의 회전을 따라가게 하기 위해 축별 각도 회전을 동일하게 Model Matrix에 적용시켜준다. 또한 Perspective Projection과 View가 동일하게 적용될 수 있도록 함께 곱하여 보내준다. 아래는 X축에 대한 예시이다.

```
mat4 x_a = RotateZ(g_ZAngle) * RotateY(g_YAngle) * RotateX(g_XAngle) * Translate(0.5f, 0, 0)
    * Scale(1, 0.01, 0.01);
glUniformMatrix4fv(uMat, 1, GL_TRUE, ProjMat * ViewMat * x_a);
glUniform4f(uAxisColor, 1, 0, 0, 1);
cube.Draw(axisProgram);
```

본 방법으로 세 개의 축을 모두 그려준다.

4) Interaction

본 과제의 명세와 같이 키보드를 통한 상호작용을 할 수 있도록 함수를 구성한다. Space를 누르면 정지, 시작이 토글된다. 1, 2번을 누르면 각각 NormalMode가 true, false가 된다. NormalMode가 true이면, Vertex Shading을 이용하고 false는 Flat Shading을 사용한다. 3, 4번은 각각 Specular 값을 변동시키는데 Delta 값은 $\text{vec4}(0.1, 0.1, 0.1, 0)$ 로 설정하였고, 추가적으로 각 원소가 음수가 되지 않게 하기 위해 spc의 x값이 0이면 return하도록 했다. 5, 6번은 각각 Shininess 값을 2씩 더하고, 뺀다. Spc와 유사하게 Shininess는 0이 되면 cosine 값이 무조건 1이 되기에 0에 가깝게 되지 않게 3 이하로 떨어지지 않게 방지해두었다.



c) 시행착오 및 추가적인 학습 사항

본 과제를 수행하면서는 자연 시간과 관련한 문제가 발생했습니다. 파일을 열고 오브젝트를 그려내기까지의 시간이 체감될 정도로 느렸기에 이를 해결하기 위한 과정에 대하여 기술해보겠습니다. 또 추가적으로 Shader에 대하여 알게 된 점 몇 가지도 함께 기술해보겠습니다.

1) 파일 입출력

처음에는 C++의 파일 입출력 라이브러리인 fstream를 사용하려 했습니다. fstream을 통해 파일의 한 줄을 가져오고, 이를 변수에 쉽게 담기 위해 String Stream을 이용해 문자열을 스트림으로 써 받아서 가져오게 구성했습니다. 그러나, 이 방식은 C의 파일 입출력보다 훨씬 느린 성능을 보여주었고 dragon.obj나 buddha.obj와 같은 파일은 열리는 데에만 4-5초 이상이 걸렸습니다. 따라서 C의 FILE 구조체를 이용하여 읽어오는 방식으로 변형하였고, 현재는 파일을 거의 바로 읽어올 수 있는 수준까지 자연 시간을 줄였습니다. 아래는 수정 전 코드의 일부입니다.

```
std::ifstream file(filePath);
std::string lineReader;

if (!file.is_open()) return false;

// Start of f
//int pos = -1;
```

```

leftBottom = vec3(1000.f, 1000.f, 1000.f);
rightTop = vec3(-1000.f, -1000.f, -1000.f);

while (std::getline(file, lineReader))
{
    // 비어 있는 라인의 경우
    if (lineReader.size() == 0) continue;

    if (lineReader[0] == 'v')
    {
        char head;
        float x, y, z;
        std::stringstream ss(lineReader);

        ss >> head >> x >> y >> z;

        PositionList.push_back( vec3(x,y,z) );
        normalList.push_back(vec3(0.f));
    }
}

```

2) 추가적인 지연 시간

항상 수업 때 Buffer Data로 동적 할당된 배열만 넣어왔기에 C++의 유용한 자료구조인 vector를 사용할 수 있다는 사실을 몰랐습니다. 동적 할당 배열을 넣으려면 사이즈를 결정해야 하는데 이는 파일을 두 번 읽어와야 하는 많은 지연 시간을 발생시켰습니다. 자료를 찾아본 결과 vector::data()라는 함수를 통해 동적 할당 배열과 동일한 효과를 가질 수 있다는 것을 알아내었고, 따라서 파일을 읽는 횟수를 1회로 줄일 수 있었습니다.

3) Shader 관련

Shader를 만들고, 물체를 그려낼 때 문제가 하나 발생했습니다. 물체가 반짝반짝 빛나며, 의도했던 것과는 전혀 다른 방향으로 그려지는 것이었습니다. 게다가 곳곳에 그려지지 않은 삼각형도 보이며 많은 문제가 발생했습니다. 이것을 해결하기 위해서 cube.obj를 가지고 디버깅을 시작했습니다. 그러나, 단순히 어디서 문제가 발생한 것인지 알 방법이 없었습니다. 처음에는 Shader 부분에 문제가 있는지 계속 확인했지만, 아무리 생각해도 틀린 부분을 찾을 수 없었습니다. 결국에는 Color 값을 uniform을 변수를 넘겨 MyObject에 문제가 있는지 확인했습니다. 이때, Uniform 변수로 넘겨 받은 Color는 제대로 칠해졌고 추가적으로 Normal에 문제가 있을 것이라는 생각이 들었습니다. 그러나, 이 문제는 Vertex Attribute Pointer를 설정할 때 Location 값의 오타로 인해 발생한 문제였습니다. Shader의 in 변수를 찾지 못하니 데이터 한 덩어리가 그대로 사라진 것이었습니다. 이 오타를 수정하고 난 후 정상 작동함을 확인할 수 있었습니다.

기존에는 Shader에 복잡한 연산이 들어가지 않는 과제들을 수행해 왔기에, 어디서 문제가 생기는지 확인하기가 수월했습니다. 그러나 이제 Shader에도 문제가 생길 수 있기에 더욱 디버깅이 어려워졌습니다. 그러나, Uniform 변수를 활용하는 등의 방식으로 변수를 줄여나가면서 어디가 문제가 되는지 찾는 법을 알게 되었습니다. 이를 통해 복잡한 구현을 할 때도 어떻게 디버깅에 접근해야 하는지를 알 수 있었습니다.

d) 참고 자료

- [실전 C프로그래밍 \(나중채 외 9인, 21세기사\)](#)

이상입니다.