컴퓨터그래픽스

과제 1 보고서

세종대학교

제출일	2025.10.01	전공	소프트웨어학과
과목	컴퓨터그래픽스	학번	21011746
담당교수	박상일 교수님	이름	양현석

1. A Waving Plain

a) 문제 분석

xy 평면에 사각형을 그린다. 이를 회전 행렬을 이용하여 비스듬히 보이도록 하고, 회전은 z축을 기준으로 진행한다. 그 후 삼각함수를 이용하여 z축의 높이를 조절해서 파동을 생성한다. 마지막으로, 평면을 구성하는 정사각형의 개수를 '1', '2' key, loop는 'space bar'를 이용하여 조절할 수 있다. 이러한 방식으로 웨이브를 형성하는 평면을 제작한다.

b) 문제 풀이

1) MyPlain Class

가장 먼저 MyPlain 클래스를 생성한다. 평면은 정사각형의 집합으로 이루어져 있다. 직관적으로 그리기 위해 xy평면을 기준으로(회전하지 않은 상태에서는 정사각형으로 보임) 먼저 그린다. MyPlain 클래스는 다음과 같은 멤버 변수를 갖는다.

멤버 명	역할
int divNum	가로, 세로 당 나누는 개수
int numVertices	총 필요한 vertex 숫자
float len	평면의 한 변의 길이
float divLen	평면을 이루는 한 정사각형의 한 변의 길이
float offset	평면의 한 변과 스크린 사이즈와의 거리
GLuint vao	Vertex array object에 대한 Handler 저장
GLuint vbo	Vertex buffer object에 대한 Handler 저장
GLuint prog	사용하는 shader 프로그램의 Handler 저장
vec4 * positions	Vertex의 위치를 저장하는 배열
vec4 * colors	Vertex의 색상을 저장하는 배열
vec4 plainColor[2]	짙은 회색과 옅은 회색 저장

먼저 각 변수를 생성자에서 초기화한다. 그러나, 실제적으로는 MyPlain을 전역 변수로 선언하고 있기 때문에, init이라는 함수를 만들어 vao, vbo 등의 생성 및 bind를 해준다. init 함수에서는 divNum을 가지고, vertex의 개수를 정하고 평면을 이루는 각 정사각형의 한 변의 길이를 정한다. 추가로 numVertices를 사용해 동적할당을 한다. 다음은 그 코드이다.

```
this->divNum = divNum;
this->numVertices = divNum * divNum * 3 * 2;
divLen = len / divNum;

if (positions != nullptr) delete[] positions;
if (colors != nullptr) delete[] colors;

positions = new vec4[numVertices];
colors = new vec4[numVertices];
```

또한 크기를 조정하는 기능을 구성해야 하기에, 이미 positions와 colors에 메모리가 할당되어 있다면, 해제 후 진행한다. 메모리를 구성한 후, 평면을 구성하는 각 정사각형의 꼭짓점의 좌표를 구하고 색상을 지정한다. 하나의 정사각형은 2개의 삼각형, 총 6개의 vertex로 이루어져 있고, 삼각형 단위로 그려내기 위해 6개의 vertex를 4개의 꼭짓점 좌표를 이용하여 정한다.

```
int curIdx = 0;

for (int i = 0; i < divNum; i++) {
   for (int j = 0; j < divNum; j++) {
      //  여기서 각각의 사각형 처리

      //  0 : DarkGray, 1 : LightGray
      int colorIdx = (j + (i)*divNum) % 2;
      if (divNum % 2 == 0 && i % 2 == 0) colorIdx = (colorIdx + 1) % 2;

      float x1 = offset + divLen * j - 1.f;
      float y1 = offset + divLen * i - 1.f;
      float x2 = offset + divLen * (j + 1) - 1.f;
      float y2 = offset + divLen * (i + 1) - 1.f;

      //  First Define the Square
      vec4 v1 = vec4(x1, y1, 0.0f, 1.f);
      vec4 v2 = vec4(x2, y1, 0.0f, 1.f);
      vec4 v3 = vec4(x1, y2, 0.0f, 1.f);
      vec4 v4 = vec4(x2, y2, 0.0f, 1.f);

      positions[curIdx] = v3; colors[curIdx] = plainColor[colorIdx]; curIdx++;
      positions[curIdx] = v1; colors[curIdx] = plainColor[colorIdx]; curIdx++;
      positions[curIdx] = v2; colors[curIdx] = plainColor[colorIdx]; curIdx++;

      positions[curIdx] = v3; colors[curIdx] = plainColor[colorIdx]; curIdx++;
      positions[curIdx] = v4; colors[curIdx] = plainColor[colorIdx]; curIdx++;
      positions[curIdx] = v2; colors[curIdx] = plainColor[colorIdx]; curIdx++;
   }
}
```

이제 생성한 vertex를 가지고, Vertex Array Object와 Vertex Buffer Object를 각각 생성하고 bind한 후 BufferData와 BufferSubData를 만들어준다. 이 과정에서 Shader를 연결하고, 준비한 데이터들을 Shader로 보낸다. Shader로 데이터를 보내는 작업은 connectDataToShader() 함수를 활용한다.

```
if (!isInitialzed)
   glGenVertexArrays(1, &vao);
glBindVertexArray(vao);

if (!isInitialzed)
   glGenBuffers(1, &vbo);
glBindBuffer(GL_ARRAY_BUFFER, vbo);

glBufferData(GL_ARRAY_BUFFER,
          sizeof(vec4) * numVertices * 2, nullptr, GL_STATIC_DRAW);
```

```
glBufferSubData(GL_ARRAY_BUFFER, 0, sizeof(vec4) * numVertices,
positions);
glBufferSubData(GL_ARRAY_BUFFER, sizeof(vec4) * numVertices, sizeof(vec4)
* numVertices, colors);

if (!isInitialzed)
    prog = InitShader("vshader.glsl", "fshader.glsl");

connectDataToShader();

isInitialzed = true;
```

이때, boolean 변수인 isInitialized를 사용하여, 기존에 각 오브젝트가 생성된 적이 있는지 확인하여, 생성된 적이 없는 경우에만 생성한다.

connectDataToShader() 함수에서는 prog를 사용하도록 등록하고, Shader내의 변수를 가져와 Data를 읽어들이도록 한다.

```
void connectDataToShader() {
    glUseProgram(prog);

    GLuint vPosition = glGetAttribLocation(prog, "vPosition");
    glEnableVertexAttribArray(vPosition);
    glVertexAttribPointer(vPosition, 4, GL_FLOAT, GL_FALSE, 0, BUFFER_OFFSET(0));

    GLuint vColor = glGetAttribLocation(prog, "vColor");
    glEnableVertexAttribArray(vColor);
    glVertexAttribPointer(vColor, 4, GL_FLOAT, GL_FALSE, 0, BUFFER_OFFSET(sizeof(vec4) * numVertices));
}
```

이제 도형에 대한 정보를 Shader로 넘겼으니, draw()라는 함수를 구성하여 그릴 수 있도록 세팅한다. 움직임을 고려해야 하기에, 인자로 시간을 받는다. 또한 파동을 그려내야 하기에 Boolean 변수인 isWave를 받아, 파동을 그릴지 판단한다. 그렇기에, 변수 vao를 다시 Bind 해주고, prog를 재등록해준 후 glDrawArray() 함수를 호출하여 그림을 그린다. 이 때, 시간인 curTime과 boolean인 isWave를 Vertex Shader에서 사용하기 위해 uniform 변수로 등록하는 것도 함께 수행한다. 다음은 그 코드이다.

```
void draw(float curTime, bool isWave) {
    glBindVertexArray(vao);
    glUseProgram(prog);
    connectDataToShader();

    GLuint uTime = glGetUniformLocation(prog, "uTime");
    glUniform1f(uTime, curTime);

    GLuint uWave = glGetUniformLocation(prog, "uWave");
    glUniform1f(uWave, isWave);

    glDrawArrays(GL_TRIANGLES, 0, numVertices);
}
```

마지막으로, 평면을 이루는 정사각형의 개수 즉, 분할 수를 변경하는 기능을 추가한다. 각각 decrease(), increase() 함수를 구성하여 init() 함수의 인자에 변경된 divNum을 주어주는 것으로 간단하게 구현했다.

2) Vertex Shader

이번에는 그려질 점 위치를 결정하는 Vertex Shader에 대한 코드이다. 먼저 Vertex의 지점과 색상을 받아줄 vPosition, vColor를 입력받는다. 더하여 uTime, uWave로 draw에서 지정한 uniform 변수를 받아주고, 최종적으로 Fragment Shader에 color, position 벡터를 넘겨준다.

회전을 고려하기 전, 파동을 우선적으로 구성했다. 앞서 xy평면에 plain을 제작했기에, 파동은 z축을 조정하여 구현해야 한다. 주어진 예시를 확인하면 중앙에서 멀어질수록 그 파동은 점점 작아진다. (더하여, 보이지 않는 경우도 있다.) 파동을 가장 쉽게 구현하는 방법은 삼각함수를 활용하는 것인데, 중앙에서 멀어질수록 작아지고 또 이를 부드럽게 구현하기 위해 지수함수와 삼각함수의 곱꼴을 사용했다.

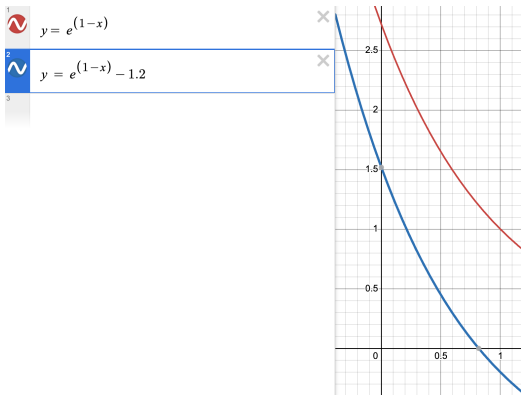
거리 계산은 vPosition의 x, y 값을 이용하여 중앙으로부터 떨어진 거리를 구했다. 이는 dist 변수에 저장한다.

Sin 함수에 넣어줄 각은 시간과 거리를 함께 고려하여 구성해야 한다. 함숫값이 거리에 따라 달라야 하고, 또 모션을 주려면 시간을 확인해야 하기 때문이다. 또 모션의 빠르기와 각을 radian으로 변환하는 것과 주기를 조정하기 위해 다음과 같이 rad 변수를 계산했다.

```
//  dist의 부호에 의해 방향이 결정
float rad = ((uTime / 4.0f - dist) / 180.0f * 3.141592f) * 500.0f * 3.141592f;
```

Z축은 화면을 기준으로 안쪽으로 들어갈수록 +이기에 dist에 -를 곱해 앞쪽으로 나오는 파동이 형성되도록 만들었다. (xy평면에 대칭이동)

sin앞에 곱해질 지수 함수는 exp() 함수를 이용하여 제작했다. 이 때, 중앙에서 거리가 멀어질수록 파동이 작아지기에 (최대 거리) – (점의 거리)를 인자로 사용해야 한다. 그러나 그저 (1 – dist)를 사용하면 빨간색 그래프와 같이 나오게 된다. 그러나 이는 너무 극단적인 값이 나올 수 있기에, 파란색 그래프와 같이 평행이동을 시킨다. 이때, 음수가 되는 값은 0으로 처리하여, 결과물이 예시와 유사하게 만든다.




```
//  rad는 음수로 계산해야 함 (z축이 반대)
pos.z = coef * sin(-rad) * 0.2f;
```

최종적으로, z값을 다음과 같이 결정한다. 0.2는 진폭을 조정하기 위해 사용한 임의의 값이다.

Fragment Shader에서 이 z 값을 그대로 사용할 수 있도록 회전 처리를 하기 전에 따로 저장한다. 이제, 회전 처리를 진행한다. X축에 대한 회전은 고정된 값을 사용하고(비스듬히 보이도록), 움직이는 축은 Z축을 사용한다. 각도는 uTime에 의해 결정되며, 회전 행렬을 활용하여 최종 위치를 결정한다.

```
const float angleX = 45.0f;
const float radX = angleX / 180.0f * 3.141592f;

//  움직이는 축은 z 축
float angleZ = uTime * 15.0f;
float radZ = angleZ / 180.0f * 3.141592f;

mat4 rotX = mat4(1);
rotX[1][1] = cos(radX);    rotX[2][1] = -sin(radX);
rotX[1][2] = sin(radX);    rotX[2][2] = cos(radX);

mat4 rotZ = mat4(1);
rotZ[0][0] = cos(radZ);    rotZ[1][0] = -sin(radZ);
rotZ[0][1] = sin(radZ);    rotZ[1][1] = cos(radZ);

pos = rotX * rotZ * pos;

gl_Position = pos;
```

3) Fragment Shader

Fragment Shader에서 입력은 color, Vertex Shader에서 회전되기 전 position을 받는다. 높이에 따라 색상을 결정해야 하기에, buff라는 변수를 이용해 보간을 진행한다. Buff의 최댓값은 position.z로부터 결정되는데, 이때 가질 수 있는 최대는 Vertex Shader의 coef * 0.2f에서 0.3정도로 알 수 있다. 따라서 buff의 최대를 1로 맞추기 위하여 다음과 같이 position.z 값을 조정한다.

```
float buff = position.z * 7;

const vec4 orange = vec4(1.0, 0.607, 0.223,1.0);
const vec4 skyblue = vec4(0.238, 0.502, 0.786, 1.0);
//  Interpolation;

vec4 dst;

//  z 축 반대로
if(buff > 0){
    dst = (1.0f - buff) * color + buff * skyblue;
}
else if(buff <= 0){
    buff *= -1;
    dst = (1.0f - buff) * color + (buff) * orange;
}
```

원래는 3정도를 곱하는게 맞겠지만, 조금 더 예시와 유사한 결과가 나오게 하기 위해 7를 곱했다. Buff가 음수일 때, 즉 파형이 안쪽으로 들어간 경우에는 -1을 곱해 정상적으로 보간이 수행되도록 했다.

4) Source code

위에서 만든 MyPlain을 전역으로 생성하고, display, myIdle, myKeyboard Callback 함수를 구성한다. 초기화를 수행하기 위해 myInit() 함수에서 plain.init()이 호출될 수 있게 하고 이를 glut Initialization이 끝난 후 호출한다.

display() 함수는 glClear()를 이용하여 지우고, Double Buffering을 사용할 수 있도록 설정했다. 그 후 plain.draw() 함수를 호출해 그릴 수 있도록 했다.

myIdle() 함수는 Sleep() 함수를 이용해 16ms 마다 호출되며, boolean 변수인 g_bPlay(재생 여부), g_bWave(파동 생성 여부)를 확인하여 다음과 같이 작동할 수 있게 했다.

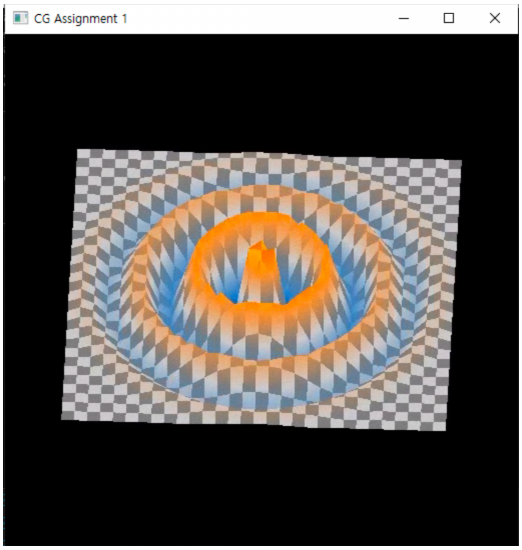
```
void myIdle() {
    Sleep(16);

    if (g_bPlay) {
        //printf("curTime : %f\n", g_Time);
        g_Time += 0.016f;

        glutPostRedisplay();
    }

    if (g_bWave) {
        glutPostRedisplay();
    }
}
```

myKeyboard() 함수에서는 space, '1', '2', 'w', 'q'의 입력을 확인하고 예시에서 주어진 각 기능을 수행할 수 있도록 구성한다.



c) 시행착오

1) 파형의 모양

처음에 파형을 만들 때 sin함수를 이용하여 만드려고 생각은 했었지만, 거리에 따른 높이 변화를 어떻게 줄 지 고민이었다. 또한 변수 인자가 중심에서 떨어진 거리와 시간 총 두 개로 구성되어 구현에 헷갈리는 부분도 있었다.

먼저 거리에 따른 높이 변화를 주기 위해 처음에는 1차 함수 (1.0 – 떨어진 거리)를 사용해서 했지만, 예시와는 괴리감이 있는 결과물이 나왔기에 거리가 멀어질수록 극단적으로 함숫값이 작아질 수 있는 지수함수를 사용했다.

변수 인자가 중심에서 떨어진 거리와 시간이라는 두 개의 변수로 구성이 되어있기에 sin과 진폭 함수에 어떤 변수를 넣어야할 지에 대한 고민이 있었다. 이 부분은 실제로 그래프를 그려보면서 확인했고, 이를 통해 주기와 진폭에 대해 더 세밀한 조정을 진행할 수 있었다.

2) 구조체의 사용

조금 더 효율적이고 가독성이 높은 코드를 작성하기 위해 구조체를 사용하고자 했지만, 이렇게 하면 어떻게 Vertet Array Object나 Buffer Object를 넘겨줄 지에 대해 문제가 생겼다. 결론적으로는 해결하지 못했지만, 지난 시간 수업을 들으며 Stride 값을 변경함으로 해결할 수 있음을 깨달았다.

이상입니다.