
컴퓨터그래픽스

과제 2 보고서



세종대학교

제출일	2025.11.10	전공	소프트웨어학과
과목	컴퓨터그래픽스	학번	21011746
담당교수	박상일 교수님	이름	양현석

1. A Simple Robot Arm

a) 문제 분석

로봇팔이 주어진 Target을 쫓아갈 수 있어야 한다. 로봇팔의 3개의 각도를 적절히 넣어 주어진 Point에 팔의 끝점이 도달할 수 있게 해야 하는데, 기본적으로 3중 For문을 통해 구할 수 있지만 시간 복잡도가 매우 높고 로봇팔의 움직임이 매우 부자연스럽다. 그리하여 이를 빠르게 찾는 것과, 그 당시의 각도의 주변에서 찾는 것을 기본적 아이디어로 갖는다. 앞선 아이디어는 Target이 빙글빙글 도는 상황에서는 적절할 수 있지만, 랜덤하게 등장하는 경우에는 Jittering 등의 문제가 발생할 수 있으므로, 이에 적절한 대처를 해야 한다.

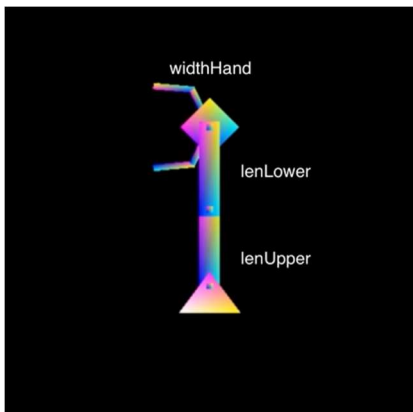
b) 문제 풀이

1) 위치 찾기

Target의 위치는 MyTarget 클래스의 메소드를 통해 얻어낼 수 있다. 그렇기에, 로봇팔의 끝점을 찾는 것이 첫 번째 목표이다. 그렇다면, 원점에서 시작하여 끝내 도달하는 지점까지의 행렬 곱을 통해 얻어내야 한다. 기본적으로 다음 식을 따른다.

```
Translate(0, -0.4, 0) * RotateZ(deg1) * Translate(0, lenUpper, 0) * RotateZ(deg2)
* Translate(0, lenLower, 0) * RotateZ(deg3) * Translate(-widthHand / 2, 0, 0) *
initPos;
```

여기서 initPos는 원점 (0, 0, 0, 1)을 뜻하고, 실제 그림 그리기에서 사용된 CTM의 핵심 부분만을 차용해서 생성된 행렬 곱의 형태이다. 또한 마지막 Translate() 함수를 보면 widthHand 부분에 -가 붙어 있는 것을 볼 수 있는데, 이는 타겟이 z축을 기준으로 Counter-clockwise 회전을 하기에 자연스러움을 위해 음수로 붙여주었다.



또한 lenUpper, lenLower는 실제 도형의 길이가 아닌, 회전축이 되는 지점부터 다음 회전축 지점까지의 거리를 표현한다. 현재 그림에서는 Hand 부분이 45도 회전되어 있는 형태이지만, 실제 가로 길이(폭)를 나타내기 위해 사용되었다. 즉, 실제로는 저 widthHand의 절반 만큼 더 진행하여, 45도로 회전된 육면체의 왼쪽 변의 중앙 즈음에 위에서 계산된 Hand의 끝 점이 위치하게 되는 것이다. 그리하여, 저 집게손 안에 Target이 들어오게 된다.

저 계산된 끝점 위치를 쉽게 계산할 수 있도록 하기 위해 함수화 하여 사용한다.

```
vec4 getCurPos(const float deg1, const float deg2, const float deg3)
{
    // CTM 과 계산
    return Translate(0, -0.4, 0) * RotateZ(deg1) * Translate(0, lenUpper, 0) *
```

```
RotateZ(deg2) * Translate(0, lenLower, 0) * RotateZ(deg3) * Translate(-widthHand
/ 2, 0, 0) * initPos;
}
```

2) 3개의 각도를 결정 짓기

가장 기본적으로 3중 for문을 사용하여 계산할 수 있다. 그러나, 본 방법은 360도를 1도씩 진행한다고 한다면, $360 * 360 * 360$ 이라는 계산을 한 프레임마다 진행해야 한다. 물론 특정 상수 이하로 내려오면 반복문을 탈출하는 식으로 시간 복잡도를 개선할 수는 있지만, 여전히 계산해야 할 연산량이 많은 것은 물론 매 프레임마다 새로운 각도가 결정될 수 있다. 더하여 특정 상수를 결정 짓기가 매우 애매하다는 것도 있다. 따라서 경사 하강법(Gradient Descent)를 이용하여 연산량 감소 및 현재 유지하고 있는 각도에서 크게 달라지지 않는 범위 내에서 계산하도록 한다.

경사 하강법은 특정 값으로 유발되는 Error를 함수화하였을 때, 이를 최소가 되게끔 연산하는 것으로 즉 Error에 대한 함수 값이 최소가 되는 극값을 찾는 것이다. 현재의 에러에서 특정 변수를 조절하였을 때 Error 함수에서 값이 어떤 식으로 변화하는지 추적하여, 이를 반영하는 것이다. 기본적인 경사 하강법은 다음과 같은 식을 따른다.

$$x_0 = x_0 - \eta \frac{\partial f}{\partial x_0}$$

현재의 Error 값에서 변화율을 빼서 다시 현재의 Error 값에 넣어주는 형태이다.

그렇다면, 왜 변화율을 빼주는가에 대해서는 만일 어떤 지점에서의 순간 변화율이 양수라면 x_0 값은 더 작아지게 될 것이다. 반대로 음수라면 x_0 값은 더 커지게 되는데, 이는 x_0 가 결론적으로 함수의 그래프 상에서 경사를 타고 내려가는 형태가 된다. 즉, 최소 값에 점차 도달하게 되는 것이다. 결국에는 순간 변화율이 0이 되는 지점에 도달하고, 이를 최소를 나타내는 극값임을 결정할 수 있게 된다.

그러나 에러에 대한 함수는 기본적으로 3개의 각도를 갖는 다변수 함수이며, 그래프로 표현하여 시각화하기도 쉽지 않다. 더군다나, 이차 함수와 같이 단일 최솟값을 갖는 형태도 아닐 것이다. 즉, 여러 개의 극값을 가지게 된다면 그 중 진정한 최소값(Global Minima)을 특정 지을 수 없는 문제도 존재한다. 이를 Local Minima 문제라 하며, 가짜 최소값을 가져오는 문제를 유발한다. 그러나, 본 프로그램에서는 Error에 대한 함수가 복잡하지 않기에, Local Minima를 고려하지 않아도 거의 정확한 값들이 도출된다. 그러나 Gradient가 매우 작다는 문제가 존재한다. 이 문제를 완화하기 위해 η 를 사용한다. 이는 Learning Rate(기계 학습 분야에서 사용하는 용어) 혹은 Step Size라고도 한다. 기본적으로 순간 변화율에 가중치를 곱하여 이동하는 양을 조절하여 문제를 일부 해소할 수 있다. 위 식을 사용하기 위해 먼저 Error를 계산하는 식을 작성한다. Error(편차)를 절댓값화하기 위해 제곱으로 사용했다.

```
float getSquaredError(const vec4 & curPos)
{
    vec3 targetPos = target.GetPosition(g_time);
    float dx = targetPos.x - curPos.x;
```

```

float dy = targetPos.y - curPos.y - 0.4f; // 0.4 만큼 이동 - 원래 위치로 보존
return (dx * dx + dy * dy);
}

```

0.4만큼 이동한 이유는 주어진 초기 CTM 값에 -0.4만큼 y축 평행이동이 되어 있기 때문이다. 이 오차값을 보정하여야 정확한 값을 도출할 수 있다.

다음은 이를 이용해 경사 하강법 식을 작성하였다.

```

void computeAngle()
{
    // 분자 (delta)
    const float delta = 0.1f;

    // Gradient 가 작은 문제를 해결하기 위한 값 : 각기 다른 비율을 위해 3 개 사용
    const float learningRate1 = 2200.f;
    const float learningRate2 = 2000.f;
    const float learningRate3 = 2000.f;

    vec4 curPos = getCurPos(ang1, ang2, ang3);
    float curErr = getSquaredError(curPos);

    std::cout << "Cur Error : " << curErr << std::endl;

    vec4 ang1Pos = getCurPos(ang1 + delta, ang2, ang3);
    float ang1Err = getSquaredError(ang1Pos);
    float slope1 = (ang1Err - curErr) / delta;

    vec4 ang2Pos = getCurPos(ang1, ang2 + delta, ang3);
    float ang2Err = getSquaredError(ang2Pos);
    float slope2 = (ang2Err - curErr) / delta;

    vec4 ang3Pos = getCurPos(ang1, ang2, ang3 + delta);
    float ang3Err = getSquaredError(ang3Pos);
    float slope3 = (ang3Err - curErr) / delta;

    // Gradient Descent (원래 값 - 변화율)
    ang1 = ang1 - learningRate1 * slope1;
    ang2 = ang2 - learningRate2 * slope2;
    ang3 = ang3 - learningRate3 * slope3;
}

```

각 각도의 learning rate를 다르게 한 이유는 이동에 대한 가중치가 달라야 하기 때문이다. ang1의 경우 ang2, ang3에 비해 이동하는 거리가 더 많은 것이 정상적인 작동을 할 수 있게 했다. 이는 민감도와 관련이 있는 문제로, 만일 ang2나 ang3가 ang1보다 더 빠르다면, ang1이 target을 쫓아가기도 전에 ang2, ang3가 먼저 target을 쫓아가려 애를 쓸 것이다. 이는 로봇팔의 부자연스러운 운동을 유발한다. 또한 Gradient(Slope)값이 아주 작아서 Learning Rate가 충분히 크지 않으면 거의 이동을 하지 않는다. 그리하여 여러 번의 값을 대입해본 결과 대략 2000 이상의 값을 유지하도록 했다. Slope 값은 순간 변화율까지는 아니지만, 작은 값(delta)에 대비하여 발생한 변화율을 사용할 수 있게 한다. 또한 반복문을 사용하지 않아도 된다. 애초에 computeAngle() 함수가 매 프레임마다 호출된다는 점을 생각했을 때, 3중 반복문을 사용했던 상황에서 무반복문 상황이 되면 시간 복잡도를 극한까지 줄일 수 있다.

다변수 함수의 변화율을 사용하기 위해 자코비안 행렬(Jacobian matrix)를 사용하는 방법도 있지만, 그 대신 수치적 미분을 사용하여 Slope를 계산했다.

또한, 찾아본 자료들에 따르면 Jittering 문제는 learning rate에 추가적으로 Error(Dist)를 곱하여 해결하는 경우도 있었다. 즉 거리에 비례하여 step size를 조절하는 것인데, 가까울수록 더 작은 값이 곱해져 조금만 이동하게 되고 이는 Jittering 문제를 해결할 수 있다. 그러나, 현재 Jittering이 발생하지 않고 적용 후에 오히려 너무 이동이 느려지는 현상이 발생하여 삭제하였다.

3) 로봇팔 그리기

주어진 모양을 조금 변형하여 만들기 위해 집게발을 단 로봇팔 모형으로 만들어보기로 하였다. 먼저 초기 값을 저장해야 한다. 이는 Target에서 CTM을 사용하기 때문이다. 기존에는 temp라는 변수에 저장되었지만, 조금 더 명확하게 initM으로 변경하여 저장하고, 그림이 다 그려진 후 처음에 저장된 CTM을 다시 반환해주었다. 초기 CTM은 y로 -0.4 평행이동된 행렬을 담고 있다.

다음은 위치를 보존하기 위해 중간중간 CTM을 저장하는 temp를 사용하였다. temp는 묶음으로 묶이는 도형 집합 내에서 그리기 시작하는 원점에 대해 보존하는 역할을 수행한다.

먼저, Base를 그려본다. 총 3개의 오브젝트가 존재하는데, 앞, 뒤 그리고 나사의 형태인 Joint이다. 사실 CTM을 사용하지 않아도 되었지만, 가독성과 코드 통일성을 위해 Translate을 CTM에서 진행했다.

```
// Back-Base
CTM *= Translate(0, 0, 0.1);
M = Scale(0.3, 0.2, 0.1);
glUniformMatrix4fv(uMat, 1, true, CTM * M);
pyramid.Draw(program);

CTM = temp;

// Front-Base
CTM *= Translate(0, 0, -0.1);
M = Scale(0.3, 0.2, 0.1);
glUniformMatrix4fv(uMat, 1, true, CTM * M);
pyramid.Draw(program);

CTM = temp;

// Base-Joint
CTM *= Translate(0, 0.03, 0);
M = Scale(0.03, 0.03, 0.35);
glUniformMatrix4fv(uMat, 1, true, CTM * M);
cube.Draw(program);

CTM = temp;
```

앞 뒤로 Base를 담당하는 Pyramid가 그려지게 하고, Upper Arm과 체결될 Joint Cube를 두었다. 연결될 수 있도록 Joint Cube는 y로 0.03만큼 올려 배치한다.

다음은 Upper Arm이다. 먼저 회전축을 기준으로 ang1만큼 회전하기에 이를 CTM에서 반영해야 한다. 또한 길이의 절반 + 겹쳐질 수 있게 0.03 만큼 위로 이동해준다.

```
// Upper Arm
CTM *= RotateZ(ang1);
M = Translate(0, 0.2 + 0.03, 0) * Scale(0.1, lenUpper, 0.1);
```

```
glUniformMatrix4fv(uMat, 1, true, CTM * M);
cube.Draw(program);
```

Lower Arm은 Base와 마찬가지로, 세 개의 오브젝트로 이루어져 있는 형태이다. 먼저 lenUpper만큼 위로 상승시킴을 CTM, temp에 저장해 모든 오브젝트에 적용될 수 있게 한다. Joint는 현재 CTM의 위치에서 그려지게 하고, 앞 및 뒤 오브젝트는 z축으로 차이를 두어 구분한다. 또한 앞, 뒤 오브젝트는 높이의 절반 만큼 올려서 그려야 한다. 이 때, Joint와 겹쳐질 수 있도록 y로 0.06만큼 스케일을 더 키워서 그린다. 이렇게 하면 위, 아래 Joint와 겹쳐지는 모양이 나올 수 있다.

```
// Lower Arm
{
    // 먼저 y 축으로만 이동
    CTM *= Translate(0, lenUpper, 0);

    // 공통 부분 저장
    temp = CTM;

    // Lower-Joint
    CTM *= Translate(0, 0, 0) * RotateZ(ang2);
    M = Scale(0.03, 0.03, 0.25);
    glUniformMatrix4fv(uMat, 1, true, CTM * M);
    cube.Draw(program);

    CTM = temp;

    // Back
    CTM *= Translate(0, 0, 0.075) * RotateZ(ang2);
    M = Translate(0, 0.2, 0) * Scale(0.1, lenLower + 0.06, 0.05);
    glUniformMatrix4fv(uMat, 1, true, CTM * M);
    cube.Draw(program);

    CTM = temp;

    // Front
    CTM *= Translate(0, 0, -0.075) * RotateZ(ang2);
    M = Translate(0, 0.2, 0) * Scale(0.1, lenLower + 0.06, 0.05);
    glUniformMatrix4fv(uMat, 1, true, CTM * M);
    cube.Draw(program);

    // Angle2를 따라가도록 CTM 설정
    CTM = temp * RotateZ(ang2);
}
```

마지막에 보면 CTM이 ang2를 따라갈 수 있도록 저장하고 있다. 이는 그리는 순서 때문에 따로 적용한 모습을 볼 수 있다.

끝으로, Hand 부분이다. 크게 세 오브젝트로 나누어져 있다. Joint, Hand 부분과 Pincer(집게 발) 부분이다. Joint는 앞선 것들과 마찬가지로 아래의 Lower Arm과 겹쳐지는 모양이 나올 수 있도록 배치했다. 다음은 Hand 부분인데, 이는 조금 더 이쁜 모양이 나올 수 있도록 앞, 뒷면이 정사각형의 형태로 그려지도록 widthHand를 0.2로 줄여 그렸고, 더하여 Z축으로 45도 회전하게끔 그렸다.

집게발 부분은 기본적으로 25도, 60도로 두 개의 큐브를 붙여 생성하였다. 두 큐브가 이어지고 Rigid Object로 동작할 수 있게 CTM의 순서를 뒤바꿔 회전 -> 이동 -> 이동 -> 회전의 형태로 구성했다.

```

// Translate
CTM *= Translate(0, lenLower, 0);

// 공통 부분 저장
temp = CTM;

// Hand-Joint
CTM *= Translate(0, 0, 0) * RotateZ(ang3);
M = Scale(0.03, 0.03, 0.25);
glUniformMatrix4fv(uMat, 1, true, CTM * M);
cube.Draw(program);

CTM = temp;

// Hand
CTM *= RotateZ(ang3);
M = Translate(0, 0, 0) * Scale(widthHand - 0.2, 0.2, 0.1) * RotateZ(45);
glUniformMatrix4fv(uMat, 1, true, CTM * M);
cube.Draw(program);

temp = CTM;

pincerAngleInterpolation();

// Pincer
CTM *= RotateZ(curAngle) * Translate(0, 0.15, 0);
M = Scale(0.03, 0.1, 0.1);
glUniformMatrix4fv(uMat, 1, true, CTM * M);
cube.Draw(program);

CTM *= Translate(0, 0.05, 0) * RotateZ(60);
M = Translate(0, 0.1, 0) * Scale(0.03, 0.2, 0.1);
glUniformMatrix4fv(uMat, 1, true, CTM * M);
cube.Draw(program);

CTM = temp;

```

코드에는 위 부분의 집게발만 존재한다. 아랫 부분은 각도에 -를 곱하여 만들었다. 그런데, 위의 식을 보면 pincerAngleInterpolation이라는 함수와 Hand에 붙어있는 cube가 curAngle이라는 변수를 가지고 있는 것을 볼 수 있다. 이는 애니메이션을 적용하고자 한 것이다.

4) 애니메이션

집게발을 더 이쁘게 꾸미기 위해 특정 상수보다 Error 값이 작아지게 되면 집게발이 오그라들고, 그렇지 않다면 펼쳐지는 형태의 애니메이션을 구성하였다. 기본적으로 이름은 Interpolation이라 되어있지만, 반복문을 활용한 Linear 변형이다. 현재의 Error를 위의 경사 하강법에서 사용한 함수를 재활용하여 0.02 이하일 경우 집게발 안에 들어왔다고 판정하여, 오그라들게 그렇지 않다면 펼쳐지게 curAngle 값을 조정한다. startAngle과 endAngle이라는 하, 상한선을 두고 그 안에서 curAngle이 변하며 집게발 애니메이션이 동작한다. 코드는 다음과 같다.

```

// Pincer Angle
const float startAngle = 25.f;
const float endAngle = 50.f;

void pincerAngleInterpolation() {
    if (!bChasingTarget) return;

```

```

const float offset = 1.f;

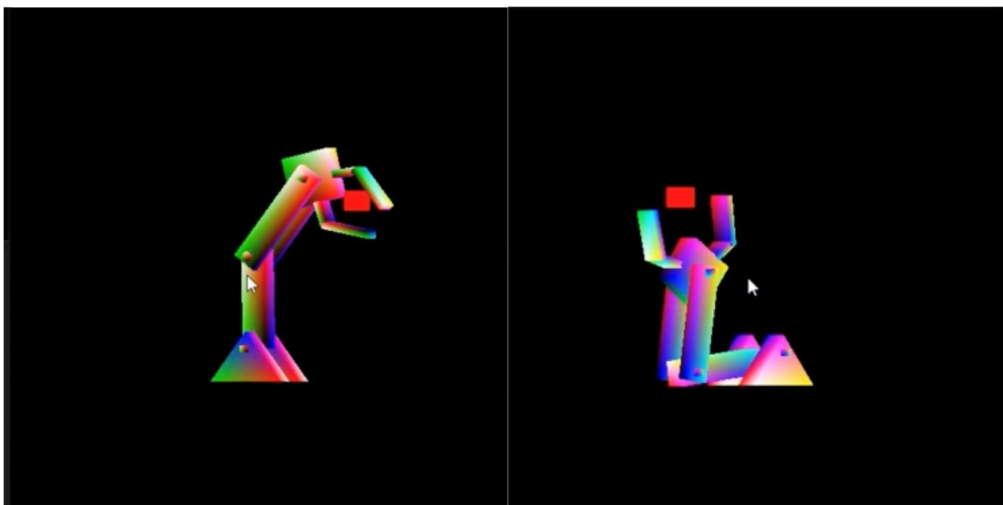
vec4 curPos = getCurPos(ang1, ang2, ang3);
float curErr = getSquaredError(curPos);

if (curErr < 0.02f) {
    if (curAngle >= endAngle) return;
    curAngle = (curAngle + offset);
    return;
}

if (curAngle <= startAngle) return;
curAngle = (curAngle - offset);
return;
}

```

이 함수를 통해 집게발이 그려지기 전에 curAngle이 조정되는 형태가 된다. 아래는 각각 잡은 모습과 잡지 못한 모습을 보여준다.



c) 시행착오 및 배운점

시행 착오가 아주 많았기에, 대표적인 것들로만 추려보았습니다.

1) 끝점 찾기

CTM으로 끝점을 찾을 수 있다는 것을 처음에는 알지 못했습니다. 각도 3개를 이용해 적절히 x, y, z 값을 조정하여 적용하고자 하였지만 복잡한 문제들이 존재했습니다. 예를 들어 RotateZ() 함수는 y -> x으로 회전한다는 것과 실제 회전 행렬을 사용하는 것은 x -> y로 진행한다는 것이 있었습니다. 아래의 코드를 보면, 매우 복잡하고 또한 평행 이동 등을 정확하게 고려하지 않아 제대로 된 위치가 나오지 않았습니다.

```

const float RAD = 180.f / 3.141592f;
//const float RAD = 1;

// 2. Iterate to Target
for (float t1 = 0; t1 < 360.f; t1 += 5.f) {
    for (float t2 = 0; t2 < 360.f; t2 += 5.f) {
        for (float t3 = 0; t3 < 360.f; t3 += 5.f) {

```

```

// x 축 기반을 y 축 기반으로 변경하기 위한 -> RotateZ 는 Y -> X 임
float x = -lenUpper * sin(t1 / RAD) - lenLower * sin((t1 + t2) / RAD) +
(widthHand / 2) * cos((t1 + t2 + t3) / RAD) - 0.1f * sin((t1 + t2 + t3) / RAD);
// CTM 초기값을 고려
float y = lenUpper * cos(t1 / RAD) + lenLower * cos((t1 + t2) / RAD) +
(widthHand/2) * sin((t1 + t2 + t3) / RAD) + 0.1f * cos((t1 + t2 + t3) / RAD) - 0.4f;

if (calDistFromTarget(x, y, 0) < minDist) {
    minDist = calDistFromTarget(x, y, 0);
    min1 = t1;
    min2 = t2;
    min3 = t3;
}
}
}
}
}

```

이후 CTM을 통해 값을 조정할 때도, 초기 CTM에서의 원점 이동과 Target의 offset을 고려하다 보니 제대로 된 위치를 잡지 못하는 문제들이 발생했었습니다. 이 문제들은 그림을 그리기 위해 사용한 CTM을 그저 그대로 사용하고, offset도 고려하지 않고 오직 World 좌표계만을 이용하여 찾아보니 정확한 위치를 계산할 수 있게 되었습니다.

2) 고려해야 할 것들이 너무 많았습니다.

현재 기본적으로 전체 도형이 y축을 기준으로 돌아가고 있습니다. 이 때문에 z축과 관련하여 고려해야 하는 것들이 너무 많았고 결국 어떻게 코드를 작성해야 할 지 고민이 많았습니다. 그러나 결국에는 y축을 공통적으로 회전하기 때문에 이 회전을 고정한 상태에서 코드를 작성해도 동일한 결과가 나온다는 것을 알게 되었습니다. 따라서 공통된 어떤 변형에 대해서는 그 변형을 고려하지 않고, 즉 변수를 통제하고 이에 대해 생각하면 더 빠르고 명확한 결과를 도출할 수 있다는 것을 알게 되었습니다.

```

// CTM Initialization
CTM = Translate(0, -0.4, 0); // *RotateY(g_time * 30);
drawRobotArm(ang1, ang2, ang3);

```

3) Stack의 사용

처음에 로봇팔을 그릴 때, 수업 시간에 배운 스택 사용을 막연하게 사용하려 했었습니다. 그렇지만, 이는 좋은 선택이 아니었습니다. mat4 변수 하나만 사용하면 되는 것을 오히려 코드가 훨씬 복잡해지게 된 것입니다. 이 복잡성이 처음에는 stack을 사용하기 위해 다음과 같은 방식이 발생하여 코드가 복잡해진다고 생각했습니다. 따라서 함수를 만들어 사용했지만, 여전히 좋지 못한 코드였습니다.

```

// Stack 사용 편의를 위한 함수
const mat4 popBack(std::list<mat4>& MStack) {
    mat4 temp = MStack.back();
    MStack.pop_back();

    return temp;
}

```

Stack과 관련된 사용에 대한 오해가 있었던 저의 불찰이었습니다. 따라서 Stack을 통해 CTM을 저장하는 방식을 폐기하고 하나의 임시 변수를 두어 CTM을 저장하고 통제했습니다.

4) 시간복잡도 및 Jittering 관련 문제

위의 문제 풀이에서 보시다시피, `computeAngle` 함수에서는 반복문을 사용하고 있지 않습니다. 하지만, 처음에는 3중 반복문에서 코드를 시작하였기 때문에, 반드시 반복문을 사용해야 한다는 생각이 지배적으로 있었습니다. 그러나 몇 번 반복문을 돌지에 대해서도 명확한 판단이 서지 않았고, 100회 단일 반복문을 사용했었습니다. 이 때 Jittering 문제가 발생하고 더하여 끊기는 문제가 생겼습니다. 코드를 전체적으로 보던 중 애초에 `computeAngle` 함수는 매 프레임마다 호출되고, 현재 `ang`들을 기준으로 값을 도출하기 때문에 반복문을 돌 필요가 없다는 것을 깨달았습니다. 반복문을 삭제하니 Jittering 현상과 끊김 현상이 모두 해결되었습니다.

5) 경사 하강법 관련

경사 하강법은 처음 공부하는 입장에서 어려운 개념이었습니다. 원래의 값에서 변화율을 빼면 최소 값에 도달한다는 개념은 직관적으로 이해가 가지 않았기 때문입니다. 그래서 여러 개의 극값을 갖는 그래프를 임의로 그려 Step Size를 조절하며 값을 직접 내보았고, 마침내 경사 하강법의 원리에 대해서 알 수 있었습니다. 정확한 값은 아니더라도 이에 근사하여 최적의 값을 찾는다는 문제는 제게 매력적이었고, 또한 게임 개발을 희망하고 있는 학생으로써 여러 가지 최적화 문제에 활용할 수 있겠다고 생각했습니다. 또 전에는 말로만 들어보았고 제대로 이해를 하지 못했던 IK 문제를 직접 구현해봄으로써 어떤 식으로 IK 문제들을 해결해야 할 지에 대한 갈피를 잡을 수 있던 좋은 과제였던 것 같습니다.

d) 참고 자료

- [신나는 경사하강법, 코딩애플 Youtube](#)
- [경사하강법, two-jun's tech blog](#)
- [자코비안 행렬, 박재형의 블로그](#)
- [그레디언트, 헤시안, 자코비안, 이지 데이터사이언스](#)

이상입니다.