**기초 컴퓨터 그래픽스**

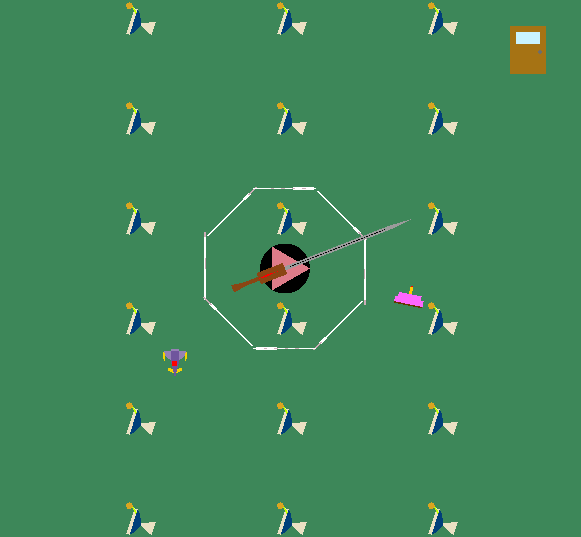
**HW2 README**

20211584 장준영

**1. 환경 명세**

Windows 10 64bit, AMD Ryzen 5 5600X, RTX 3070Ti, Visual Studio 2022 Release x64

**2. 미리보기**



OpenGL API 함수를 통해 공 튀기기 게임을 구현하였다. 마우스로 공을 클릭해 드래그 하면 그 정도만큼의 속력으로 공이 회전하며 움직이고 벽을 만나면 튕긴다. 하얀색 선은 속력의 최대치를 시각화한 것으로 저 이상 당길 수는 없다. 배경의 비행기와 공이 접촉하면 공이 커짐과 동시에 무거워지고, 케이크와 접촉하면 공이 작아지고 가벼워진다. 공의 무게에 따라 마찰력이 정해져 무거울수록 빨리, 가벼울수록 느리게 정지한다. 우측 상단의 문을 클릭하여 비행기와 케이크를 만들 수 있으며, 문을 더블 클릭하면 프로그램이 종료된다.

(※동작 예시 영상을 압축 파일에 함께 첨부합니다.)

**3. 요구사항**

(a) 다음 왼쪽 그림의 9개의 물체 중 5개를 선택한 후, 각 물체에 대하여 서로 다른 창의적인 동작 효과를 생성할 것.

[1] 칼

칼은 공을 좌클릭하면 공의 중앙을 기준으로 마우스를 따라 늘어나는 물체이다. 공이 운동할 방향을 미리 볼 수 있으며, 칼의 길이가 길어질수록 공이 빠르게 운동한다. 마우스 드래그가 얼마나 되었는지 체크하기 위해, mouse와 motion callback 함수를 통해 공의 중앙 기준 드래그가 얼마나 되었는지를 공의 속도로 저장하였다(balls[0].vx/balls[0].vy). 칼의 해당 움직임을 구현하기 위해선 translate, scale, rotate을 활용하여야 한다.

1) translate : 칼을 공의 중앙으로 옮겨야 한다.

>> ModelMatrix = glm::translate(glm::mat4(1.0f), glm::vec3(balls[0].cx, balls[0].cy, 0.0f));

2) scale : 마우스가 드래그 된 만큼 칼의 y scale을 증가시켜야 한다.

>> length = sqrt(pow((dxo - balls[0].cx), 2) + pow((dyo - balls[0].cy), 2));

>> ModelMatrix = glm::scale(ModelMatrix, glm::vec3(1.0f, -length/25, 1.0f));

3) rotate : 마우스를 따라 회전해야 하므로, 공의 중심과 마우스의 각도만큼 회전해야 한다.

>> angle = atan2(dxo - balls[0].cx, dyo - balls[0].cy);

>> ModelMatrix = glm::rotate(ModelMatrix, -angle, glm::vec3(0.0f, 0.0f, 1.0f));

[2] 셔츠

셔츠는 최대 속력을 표현하는 원으로 사용되었다. 칼의 갈색 손잡이 부분이 이 원을 벗어나면 자동으로 최대 속력으로 설정된다. 공을 두르는 동그라미 모양으로 만들기 위하여, 원점에서 x 기준 축소한 선분을 동그라미 모양으로 만들고 공의 중심 위치로 이동하였다. 이를 위해 translate, rotate, scale을 모두 사용해야 한다.

1) translate : 공 주변 위, 아래, 대각선 방향(8가지)에 따라 위치를 지정한다.

>> ModelMatrix = glm::translate(glm::mat4(1.0f), glm::vec3(balls[0].cx - 55, balls[0].cy + 55, 0.0f));

2) scale : 셔츠를 선분 모양으로 만들기 위해 x 기준 축소, y 기준 확장하였다.

>> ModelMatrix = glm::scale(ModelMatrix, glm::vec3(0.1f, 3.5f, 1.0f));

3) rotate : 방향에 따라 알맞은 회전을 진행한다.

>> ModelMatrix = glm::rotate(ModelMatrix, -45.0f \* TO\_RADIAN, glm::vec3(0.0f, 0.0f, 1.0f));

[3] 칵테일

칵테일은 배경을 장식하는데 사용되었다. 배경을 가득 채우기 위해 tan 그래프를 y=x 기준으로 대칭시킨 그래프를 따라 움직이고, 그때의 tan 함수값만큼 확장되며, tan 함수값의 역수(cos)만큼 회전한다.

>> int cocktail\_clk = (timestamp % 1442) / 2 - 360;

float rotation\_angle\_cocktail = atanf(180.0f \* TO\_RADIAN \* cosf((cocktail\_clk)\*TO\_RADIAN));

float scaling\_factor\_cocktail = 5.0f \* (1.0f - fabs(sinf(cocktail\_clk \* TO\_RADIAN)));

>>(60개의 칵테일 중 하나의 아핀 변환. 반복문을 통해 타일처럼 배경을 채웠다.)

ModelMatrix = glm::translate(glm::mat4(1.0f), glm::vec3(centerx + win\_width / 4, centery - 100 \* i, 0.0f));

ModelMatrix = glm::translate(ModelMatrix, glm::vec3(100.0f \* tanf(cocktail\_clk \* TO\_RADIAN), (float)cocktail\_clk, 0.0f));

ModelMatrix = glm::scale(ModelMatrix, glm::vec3(0.5f / scaling\_factor\_cocktail + 1.0f, 0.5f / scaling\_factor\_cocktail + 1.0f, 1.0f));

ModelMatrix = glm::rotate(ModelMatrix, rotation\_angle\_cocktail, glm::vec3(0.0f, 0.0f, 1.0f));

ModelViewProjectionMatrix = ViewProjectionMatrix \* ModelMatrix;

glUniformMatrix4fv(loc\_ModelViewProjectionMatrix, 1, GL\_FALSE, &ModelViewProjectionMatrix[0][0]);

draw\_cocktail();

[4] 비행기

비행기는 우측 상단의 문을 좌클릭하면 최대 다섯 개 윈도우 위 무작위의 위치에 생성된다. (※문을 더블 클릭하면 프로세스가 종료된다. 문을 한 번 누르고 다른 곳을 누르면 문이 다시 닫히므로, 닫은 후에 눌러야 여러 개의 비행기를 생성할 수 있다.) 생성된 비행기는 공과 닿으면 사라지며 공의 크기가 커진다. 이를 구현하기 위해 구조체와 배열을 사용하였다.

>> typedef struct {

float cx, cy;

int exist;

} Big; Big big[5];

>> for (i = 0; big[i].exist == 1; i++);

if (i != 5) {

big[i].cx = (float)((rand() % (win\_width / 2)) - win\_width / 4);

big[i].cy = (float)((rand() % (win\_height / 2)) - win\_height / 4);

big[i].exist = 1;

}

Big 구조체는 비행기의 x, y 좌표와 존재하는지를 표시하는 exist로 이루어져 있다. 문이 좌클릭되면 배열을 돌며 exist가 0인 칸을 찾아, 무작위의 좌표를 부여하고 exist를 1로 만든다.

>> for (i = 0; i < 5; i++) {

if (big[i].exist == 1) {

if ((balls[0].cx - balls[0].rad <= big[i].cx) && (big[i].cx <= balls[0].cx + balls[0].rad) && (big[i].cy <= balls[0].cy + balls[0].rad) && (balls[0].cy - balls[0].rad <= big[i].cy)) {

ig[i].exist = 0;

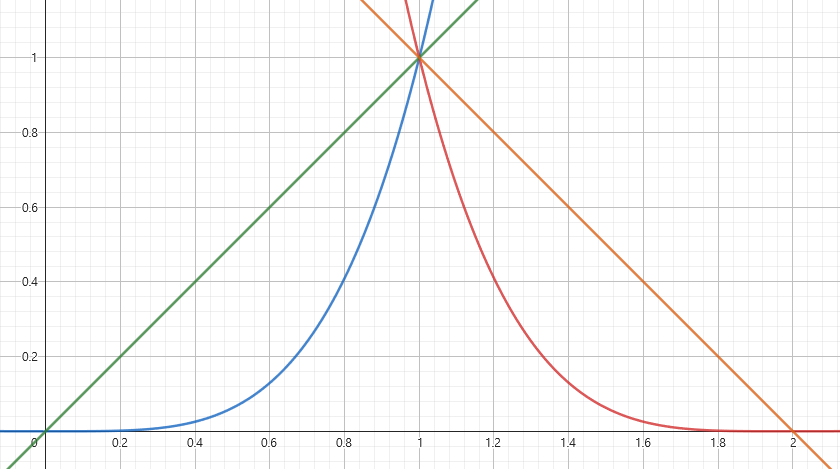
balls[0].rad \*= 1.25f;

}

}

이후 공이 비행기와 닿았는지를 display의 맨 처음부터 점검하여 닿았다면 그 big의 exist를 0으로 되돌린다. 공의 반지름인 balls[0].rad는 1.25배가 되어 커진다.

생성된 비행기는 아이템 같은 느낌을 주기 위해 회전과 확장을 반복한다. 모션그래픽은 움직임이 선형적일 때보다 곡선적일 때 더 생동감 있기 때문에, 회전과 확장을 y=x^4 그래프로 표현하였다.



회전 변환은 다음과 같은 개형이다. 위의 직선 그래프와 달리 기울기의 변화가 있는 아래의 그래프를 선택하였다.

>> int bigrotate = timestamp % 100;

if (bigrotate < 50) { //0~49

bigrotate = 360 \* (pow(bigrotate, 4)) / 5764801;

}

else {

bigrotate = -360 \* (pow(bigrotate - 100, 4)) / 5764801;

}

ModelMatrix = glm::rotate(ModelMatrix, (float)bigrotate \* TO\_RADIAN, glm::vec3(0.0f, 0.0f, 1.0f));

타임스탬프의 값을 구간에 따라 위와 같이 조절하여 구현했다. scale 역시 같은 방식이다.

>> int bigscale = timestamp % 100;

ModelMatrix = glm::scale(ModelMatrix, glm::vec3(pow((0.75 + (float)bigscale / 400), 4), pow((0.75 + (float)bigscale / 400), 4), 1.0f));

[5] 케이크

케이크는 우측 상단의 문을 우클릭하면 윈도우 위 무작위의 위치에 생성된다. [4]와 구현 방식은 완전히 동일하다.

>> typedef struct {

float cx, cy;

int exist;

} Small; Small small[5];

>> for (i = 0; small[i].exist == 1; i++);

if (i != 5) {

small[i].cx = (float)((rand() % (win\_width / 2)) - win\_width / 4);

small[i].cy = (float)((rand() % (win\_height / 2)) - win\_height / 4);

small[i].exist = 1;

}

>> for (i = 0; i < 5; i++) {

if (small[i].exist == 1) {

if ((balls[0].cx - balls[0].rad <= small[i].cx) && (small[i].cx <= balls[0].cx + balls[0].rad) && (small[i].cy <= balls[0].cy + balls[0].rad) && (balls[0].cy - balls[0].rad <= small[i].cy)) {

small[i].exist = 0;

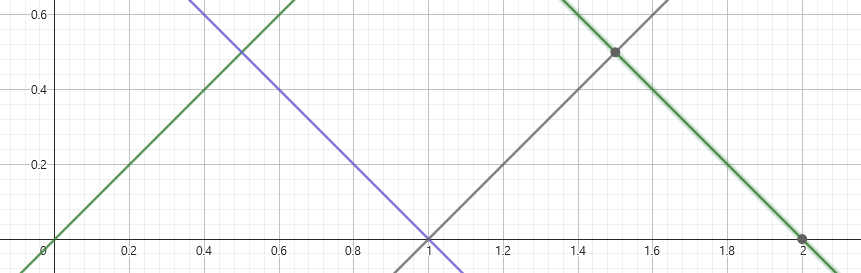
balls[0].rad \*= 0.75f;

}

}

}

케이크도 아이템 같은 느낌을 주기 위해 떨리는 듯한 동작을 한다. 따라서, 구간에 따라 그래프를 다음과 같이 설정하였다.



>> int smallrotate = timestamp % 180;

if (smallrotate < 45);

else if (smallrotate < 90) {

smallrotate = -smallrotate + 90;

}

else if (smallrotate < 135) {

smallrotate = smallrotate - 90;

}

else {

smallrotate = -smallrotate + 180;

}

>> ModelMatrix = glm::translate(glm::mat4(1.0f), glm::vec3(small[i].cx, small[i].cy, 0.0f));

>> ModelMatrix = glm::rotate(ModelMatrix, (float)(smallrotate - 22.5f) \* TO\_RADIAN, glm::vec3(0.0f, 0.0f, 1.0f));

>> ModelMatrix = glm::scale(ModelMatrix, glm::vec3(1.0f + ((float)smallrotate / 45 + 0.2f), 1.0f + ((float)smallrotate / 45 + 0.2f), 1.0f));

(b) 충분히 복잡도가 있는 2차원 기하 물체를 자신이 한 개 모델링 한 후, 위의 물체들처럼 창의적인 동적 효과를 생성하라.

[1] 공

점을 이은 삼각형을 통해 유닛을 만들기 때문에, 부드러운 원을 만들기 위해선 많은 점을 찍어야 한다.

>> GLfloat ball1[360][2];

GLfloat tri[3][2];

void prepare\_ball1() {

for (i = 0; i < 360; i++) {

ball1[i][0] = balls[0].cx + balls[0].rad \* cos(i \* TO\_RADIAN);

ball1[i][1] = balls[0].cy + balls[0].rad \* sin(i \* TO\_RADIAN);

}

for (i = 0; i < 3; i = i++) {

tri[i][0] = balls[0].cx + balls[0].rad \* cos(120 \* i \* TO\_RADIAN);

tri[i][1] = balls[0].cy + balls[0].rad \* sin(120 \* i \* TO\_RADIAN);

}

…

}

반지름이 r인 원은 (rcosk, rsink) (0<=k<=2pi)인 점의 집합이므로, 반복문으로 360개 점의 좌표를 표현하였다. 또, 공의 회전을 구별하기 위해 120도마다 삼각형이 될 좌표도 설정하였다.

공은 좌표, 속도, 반지름(크기)에 따라 다른 개체에 많은 영향을 미치기 때문에 구조체를 통해 정보를 저장하였다. 이를 통해 공 자체의 움직임도 자연스럽게 설정할 수 있다.

>> typedef struct {

float cx, cy, rad, vx, vy;

int big, small;

} Ball;

- 드래그의 크기에 비례한 속도로 이동

[1]에서 저장한 공의 속도 vx, vy를 통해 다음 프레임에 공이 있어야 할 위치를 구할 수 있다. 거리=속력\*시간이고 일정한 단위 시간마다 프레임이 변하므로, 속력에 상수를 곱한 꼴로 공의 좌표인 cx, cy를 구하면 된다.

>> balls[0].cx += balls[0].vx;

balls[0].cy += balls[0].vy;

>> ModelMatrix = glm::translate(glm::mat4(1.0f), glm::vec3(balls[0].cx, balls[0].cy, 0.0f));

- 크기에 따른 마찰력 변화

공의 반지름이 커질수록 무게가 무거워진다고 가정하였다. 마찰력은 물체의 무게와 비례하는 힘이므로, 반지름과 마찰력을 비례 관계로 설정하였다.

>> balls[0].vx = balls[0].vx / (1+(balls[0].rad/2000)\*1.0000025);

balls[0].vy = balls[0].vy / (1+(balls[0].rad/2000)\*1.0000025);

이는 다음 프레임의 vx, vy를 구하는 식이다. 마찰력은 운동 방향의 반대로 가해지는 힘이므로 속력에 나눗셈을 하면 자연스러운 마찰력을 연출해낼 수 있다. 마찰력의 역할을 하는 반지름이 적당히 비례관계를 가질 수 있도록 상수식을 설정하였다.

- 속도에 따른 회전

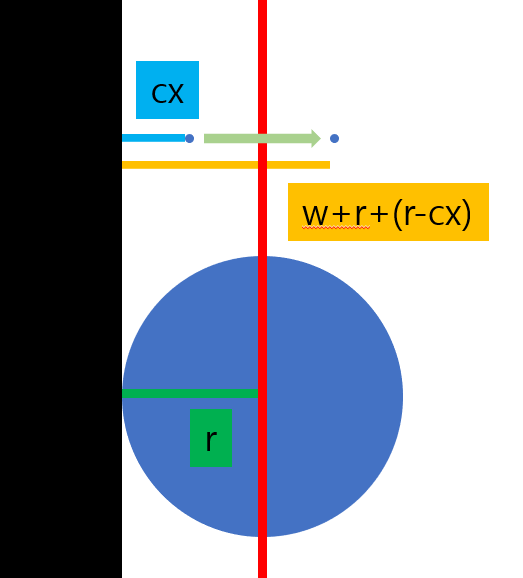
속도가 빠를수록 빠르게, 느릴수록 느리게 회전하도록 설정하였다.

>>ModelMatrix = glm::rotate(ModelMatrix, balls[0].vx\* balls[0].vy \*10\*TO\_RADIAN, glm::vec3(0.0f, 0.0f, 1.0f));

회전 정도를 구하는 식은 vx\*vy\*10\*TO\_RADIAN이다. 빠르게 운동할수록 회전 각도도 커진다.

- 벽과 충돌할 때 운동량 변화

해당 운동은 마찰력은 있지만 운동량 손실은 없다고 설정하였다. 따라서, 움직이지 않는 강체인 벽(윈도우의 모서리)에 공이 부딪치면 반대 방향으로 동일한 속력을 갖도록 충돌이 발생한다.



이를 구현하기 위해 벽에 닿았을 때 다음과 같은 대칭 이동을 하고 속도의 부호를 반전했다. 네 모서리 모두에 대해서는 다음과 같다.

>>if (balls[0].cx - balls[0].rad <= -win\_width/2) {

balls[0].cx = -win\_width / 2 + balls[0].rad + (-win\_width / 2 + balls[0].rad - balls[0].cx);

balls[0].vx = -balls[0].vx;

}

if (balls[0].cx + balls[0].rad >= win\_width/2) {

balls[0].cx = (win\_width / 2 - balls[0].rad) - (balls[0].cx - win\_width / 2 + balls[0].rad);

balls[0].vx = -balls[0].vx;

}

if (balls[0].cy - balls[0].rad <= -win\_height/2) {

balls[0].cy = -win\_height / 2 + balls[0].rad + (-win\_height / 2 + balls[0].rad - balls[0].cy);

balls[0].vy = -balls[0].vy;

}

if (balls[0].cy + balls[0].rad >= win\_height/2) {

balls[0].cy = (win\_height / 2 - balls[0].rad) - (balls[0].cy - win\_height / 2 + balls[0].rad);

balls[0].vy = -balls[0].vy;

}

- 반지름에 따른 scaling

처음 설정된 반지름은 25.0f이므로 반지름이 커지면 다음과 같은 scale 연산을 해주어야 한다.

>> ModelMatrix = glm::scale(ModelMatrix, glm::vec3(balls[0].rad / 25.0f, balls[0].rad / 25.0f, 1.0f));

[2] 문

사각형 네 개로 문, 창문, 손잡이를 만들어 닫힌 문의 앞 면을 표현하였다.

>> GLfloat door[4][2] = { {-15.0,0.0}, {-15.0,40.0}, {15.0,40.0}, {15.0,0.0} };

GLfloat door\_window[4][2] = { {-10.0,25.0}, {-10.0,35.0}, {10.0,35.0}, {10.0,25.0} };

GLfloat door\_handle[4][2] = { {10.0,20.0}, {12.0,18.0}, {10.0,16.0}, {8.0,18.0} };

문은 비행기와 케이크를 생성하는 버튼과 프로세스를 퇴장하는 버튼 역할을 모두 수행한다.

[3] 열린 문

[2]의 문을 좌클릭하면 나타나는 열린 문이다. 문을 두 번 클릭했을 때만 프로세스가 종료되도록 하기 위해 확인 차 만들었다. 문, 창문, 공간 네 개의 사각형으로 모델링하였다.

>> GLfloat od[4][2] = { {-15.0,0.0}, {-15.0,40.0}, {-40.0,50.0}, {-40.0,10.0} };

GLfloat od\_window[4][2] = { {-20.0,35.0}, {-20.0,25.0}, {-35.0,31.5}, {-35.0,41.5} };

GLfloat od\_back[4][2] = { {-15.0,0.0}, {-15.0,40.0}, {15.0,40.0}, {15.0,0.0} };