# 요구사항

본 수업에서 배포한 예제 프로그램을 기반으로 하여 **재미있고 복잡도가 있는 2차원 기하 변환**을 설계하고, 이를 OpenGL API 함수 (Core Profile)를 사용하여 구현하라.

## (a) [요구 조건 1]

다음 왼쪽 그림의 9개의 물체 중 5개를 선택한 후, 각 물체에 대하여 서로 다른 **창의적인 정적/동적 효과**를 생성할 것. (각 물체 사용 당 최대 15점)

### 1. Airplane

- Airplane은 게임의 중심 객체(플레이어)로, update() 루틴 내에서 WASD 키 입력을 기반으로 player\_pos를 업데이트한다. 예를 들어, if (key\_pressed['w'] || key\_pressed['W']) player\_pos.y += player\_speed; 와 같은 방식으로 w키를 누르면 플레이어가 위로 이동한다.

- 화면 경계 바깥으로 이동하지 않도록 glm::clamp() 또는 수동 범위 제한을 통해 위치 제한이 적용된다.

- 현재 모드(sword/gun)에 따라붙는 무기(Sword 또는 Hat)의 위치도 Airplane의 위치를 기준으로 업데이트된다.

- 플레이어는 마우스 위치에 따라 바라보는 방향이 달라지므로, 총 쏘는 방향 벡터 glm::normalize(mousePos - playerPos)가 계산되고, 이 방향을 따라 Cake 총알이 생성된다.

### 2. Cake

- Cake는 gun 모드에서 마우스 좌 클릭 시 생성된다.

- 한 번에 4개의 총알이 생성되며, 그 중 3개는 중심 총알을 기준으로 반지름 20.0f 정도의 원형 궤도로 회전한다.

- 회전은 float angle = bullet\_tick + i \* (360.0f / num\_orbit); 식으로 업데이트된다.

- 즉, 중심 총알을 중심으로 계층적 위치 계산이 수행되며, 중심 총알이 앞으로 이동하면서 회전 총알도 같이 이동하게 된다.

- 충돌 판정은 총알의 현재 위치와 좀비의 위치 간의 거리를 계산하여 glm::distance(...) < radius 조건으로 처리된다.

### 3. Hat

- Hat은 초기엔 맵 어딘가에 고정되어 있으며, if (distance(player, hat) < pickupRadius) 조건을 만족하면 장착된다.

- 장착된 상태에서 X키를 누르면 GUN MODE가 활성화되고, 이때만 화면에 렌더링된다.

- 마우스 방향은 atan2(mouse.y - player.y, mouse.x - player.x)를 통해 얻으며, 해당 방향을 따라 Hat의 위치 이동이 이루어진다.

- Hat은 총알 발사 시 Cake의 시작 위치와 방향을 설정하는 기준점 역할도 한다.

### 4. Sword

- 초기엔 맵에 배치되어 있고, Hat과 마찬가지로 근접 시 장착된다.

- 초기 MODE는 SWORD MODE이므로 장착 즉시 발동된다. GUN MODE 상태에서 Z 키를 누르면 다시 sword 모드가 활성화되고, 이때만 화면에 렌더링된다.

- sword는 장착 개수에 따라 동일한 간격으로 회전하는 애니메이션이 진행된다.

- 회전은 float angle\_deg = sword\_angle + (360.0f / num\_swords) \* i; 식으로 누적되며, 최대 3개까지 중첩될 수 있다. 즉, 1개 장착 시 1개만 회전, 2개 장착 시 180도 간격으로 회전, 3개 장착 시 120도 간격으로 회전한다.

- 렌더링 시 modelMatrix = translate(playerPos) \* rotate(attackAngle)로 회전된 위치에 그려진다.

### 5. Car2

- Car2가 좀비에 해당한다. 총 30개의 좀비 객체가 존재하며, 초기화 시 for (int i = 0; i < 30; ++i) 반복문에서 각 좀비에 대해 패턴이 할당된다.

- 0~14번 좀비는 플레이어 추적형이며, dir = normalize(playerPos - zombiePos)를 따라 매 프레임 이동한다.

- 나머지 15마리는 다음과 같은 패턴 중 하나로 무작위 지정된다.

**패턴 1: 좌우 왕복 운동** → z.pos.x = z.origin.x + 50.0f \* sin(z.tick);

(사인파 sin(z.tick)를 사용해 X축을 중심으로 좌우 반복 운동)

**패턴 2: 상하 왕복 운동** → z.pos.y = z.origin.y + 50.0f \* sin(z.tick);

(Case 1과 동일한 사인파 진동을 Y축으로 적용)

**패턴 3: 원형 회전 운동** → z.pos.x = z.origin.x + 50.0f \* cos(z.tick); z.pos.y = z.origin.y + 50.0f \* sin(z.tick);

(삼각함수 (cos, sin)을 동시에 적용해 원을 따라 회전)

**패턴 4: 대각선 이동 + 화면 벗어나면 재등장** → z.pos += z.velocity;

if (z.pos.x > half\_w + 50.0f || z.pos.y < -half\_h - 50.0f)

z.pos = glm::vec2(-half\_w - 50.0f, half\_h + 50.0f);

(일정한 속도 벡터 z.velocity를 따라 ↘ 방향으로 이동)

**패턴 5: 직선 이동 + wrap-around (화면 가로지르기)** → z.pos.x += 3.0f;

if (z.pos.x > win\_width / 2.0f)

z.pos.x = -win\_width / 2.0f;

(일정 속도로 오른쪽 직선 이동. 화면을 넘으면 왼쪽으로 이동)

**패턴 6: 크기 변화 운동 (팽창/수축)** → float scale = 1.0f + 0.5f \* sin(z.tick \* 2.0f); z.angle = scale;

(사인파를 이용해 scale 값을 진동시켜 크기를 주기적으로 변화)

**패턴 7: 지그재그 운동** → float amplitude = 40.0f; float frequency = 5.0f;

z.pos.x = z.origin.x + amplitude \* sin(z.tick \* frequency);

(고주파 사인파를 X축에 적용하여 지그재그 형태로 진동)

**패턴 8: 랜덤 워크** **(무작위 방향 이동)** → float dx = (rand() % 200 - 100) / 100.0f;

float dy = (rand() % 200 - 100) / 100.0f;

glm::vec2 random\_dir = glm::normalize(glm::vec2(dx, dy));

z.pos += random\_dir \* 2.0f;

(매 프레임마다 새로운 무작위 방향을 선택해 조금씩 이동)

## (b) [요구 조건 2]

최소 한 개의 물체는 삼각 함수의 궤적을 따라 자연스럽게 회전하면서 움직이어야 한다. (최대 15점)

텍스트, 폰트, 소프트웨어, 스크린샷이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

- Sword는 위 코드 내용을 바탕으로 삼각 함수의 궤적을 따라 플레이어(Airplane)를 기준으로 회전하면서 위치가 종속되도록 구현하였다.

- glm::translate → glm::rotate → glm::translate의 행렬 곱 연산을 통해 플레이어 중심을 기준으로회전하는 궤적을 구현하였다.

- 삼각 함수 기반 궤적은 cos, sin 함수가 내포된 회전 행렬을 이용해 구현되었으며, 이는 시간에 따른 부드러운 원형 회전을 가능하게 한다.

- 이 구조 덕분에 sword는 항상 airplane에 종속된 위치를 가지며, 회전도 중심축 기준으로 정확하게 이루어진다.

## (c) [요구 조건 3]

최소 두 개의 물체는 움직이면서 서로 다른 방식으로 크기가 계속하여 변해야 한다. (각 최대 15점)

### 1. Car2

텍스트, 폰트, 스크린샷이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.텍스트, 폰트, 스크린샷이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

- 좀비(Car2) 패턴 5번은 화면을 수평으로 관통하여 이동하며, sin 함수를 이용해 크기가 커졌다 작아졌다를 반복한다.

- zombie\_scale = 1.0f + 1.3f \* sin(z.tick); 와 같은 실시간 sin 변화를 적용하여 프레임 단위로 scale 변화가 일어난다.

### 2. Airplane

텍스트, 스크린샷, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

- 플레이어(Airplane)의 크기는 게임 시간에 따라 선형적으로 변화하며, 위 코드와 같이 구현되었다.

- 게임 시간 60초 중 30초는 점점 커지고, 이후 점점 작아져 초기 크기로 되돌아온다.

- 위 코드에서 player.scale 은 모델링 시 실제 그래픽 객체의 크기에 영향을 미치는 scale 값이다.

## (d) [요구 조건 4]

최소 한 번의 계층적 모델링 (hierarchical modeling)을 적용한 움직임이 있어야 한다. (최대 20점)

텍스트, 스크린샷, 폰트이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

- 회전 총알은 중심 총알(bullet\_pos)을 기준으로 계층적으로 종속된 위치에 존재한다.

- 총알(hat)은 glm::translate → glm::rotate → glm::translate 순서로 행렬을 조합하여 원형 궤도에 위치하며, 중심 총알이 움직이면 회전 총알도 함께 이동한다.

- 총알의 회전 각도는 매 프레임 증가하는 bullet\_tick 값에 따라 실시간으로 업데이트되며, 이는 자연스러운 원형 회전을 표현한다.

- 이 구조는 부모-자식 관계에 의한 좌표 종속성을 정확히 구현한 예로, 계층적 모델링의 요구 조건을 충족한다.

## (e) [요구 조건 5]

최소 한 개의 물체는 마우스로 조작할 수 있어야 한다. (최대 15점)

텍스트, 폰트, 스크린샷, 라인이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.텍스트, 소프트웨어, 폰트, 멀티미디어 소프트웨어이(가) 표시된 사진

AI가 생성한 콘텐츠는 부정확할 수 있습니다.

- 총(hat)은 hat.pos = glm::vec2(mouseX, mouseY); 와 같이 마우스 입력 이벤트를 통해 직접 제어된다. hat.pos = glm::vec2(mouseX, mouseY); 에서 mouseX, mouseY는 마우스 좌표이며, 이를 실시간으로 총(hat)의 위치에 반영한다.

- 총이 마우스를 따라 움직이고, 마우스 왼쪽 클릭 시, 총의 현재 위치를 시작점으로 하여 마우스를 향하는 방향으로 총알을 발사한다.

- 발사된 총알의 방향은 위 코드와 같이 벡터 차이 + 정규화로 구한 마우스 방향 벡터를 기반으로 설정되었다.

- 이로써 사용자는 마우스 방향만으로도 즉각적인 시각 피드백과 함께 정확한 제어가 가능하게 된다.

## (f) [요구 조건6]

충분히 복잡도가 있는 2차원 기하 물체를 자신이 최대 2개 모델링 한 후, 위의 물체들처럼 **창의적인 동적 효과**를 생성하라.

### 1. Star

[복잡한 기하 구조]

- 별은 총 11개의 정점으로 구성된 다각형 형태로, 일반적인 삼각형, 사각형이 아닌 복잡한 형태의 기하 모델이다.

- GL\_TRIANGLE\_FAN 방식으로 그려지며, 정중앙에서 뻗어나가는 다각형 형태는 정적인 별이 아닌 방사형 표현을 가진 복합 도형이다.

[동적 효과: 나선형 회전 + 느린 회전 속도]

s.angle\_offset = 360.0f \* i / NUM\_STARS;

s.distance = base\_distance + i \* 20.0f;

- init\_stars() 함수에서 각 별은 다음과 같은 속성으로 초기화된다:

* angle\_offset: 360도를 고르게 분산시켜 나선형 구조 생성
* distance: 별이 중심에서 얼마나 떨어질지를 설정, 팔이 점점 길어지며 나선이 형성됨
* rotation\_speed: 각 별의 회전 속도 (현재는 전체 공통 속도 사용)

star\_global\_angle += 0.1f;

...

float rad = TO\_RADIAN \* (star\_global\_angle + s.angle\_offset);

glm::vec2 pos = s.center + s.distance \* glm::vec2(cos(rad), sin(rad));

draw\_star(pos);

- 매 프레임마다 전체 각도가 업데이트되며, 이를 기반으로 각 별의 위치를 계산한다.

- 결과적으로 각 별은 중심점을 기준으로 일정 간격을 두고 회전하면서 나선형 경로를 구성하며, 이는 시각적으로 창의적이고 동적인 배경을 형성한다.

### 2. Butterfly

[복잡한 기하 구조]

- 나비는 다음과 같은 세 부분으로 구성된 복합 도형이다:

* 왼쪽 날개 (left\_wing): 4개의 점으로 삼각형 팬 구성
* 오른쪽 날개 (right\_wing): 4개의 점
* 몸통 (butterfly\_body): 3개의 점

- 세 부분이 다른 색으로 구분되어 있고, 개별적으로 회전하거나 이동하기 때문에 계층적이고 복합적인 시각 표현을 가진다.

[동적 효과 1: 타원 궤도 회전 + 위상차 진동]

b.pos.x = b.center.x + b.radius\_x \* cos(rad) + 10.0f \* sin(time \* 2.0f + i);

b.pos.y = b.center.y + b.radius\_y \* sin(rad) + 10.0f \* cos(time \* 3.0f + i);

- update\_butterflies()에서 각 나비는 중심점 주변을 타원 궤도로 회전하면서, sin/cos 기반의 고주파 진동을 추가로 겹친다:

- 위상 차를 index i에 따라 부여해, 나비들이 서로 다른 타이밍으로 떨리는 듯한 복잡한 움직임을 만든다.

[동적 효과 2: 날개 펄럭임]

b.wing\_flap = sin(time\_ms \* 0.01f + rad) \* 20.0f;

- 날개의 각도는 sin 함수를 기반으로 시간에 따라 부드럽게 변화한다.

glm::mat4 ML = glm::rotate(base, TO\_RADIAN \* b.wing\_flap, glm::vec3(0, 0, 1));

glm::mat4 MR = glm::rotate(base, TO\_RADIAN \* -b.wing\_flap, glm::vec3(0, 0, 1));

- 왼쪽 날개는 +wing\_flap, 오른쪽 날개는 -wing\_flap으로 각각 회전하여 양쪽 날개가 번갈아 접히고 펼쳐지며 펄럭이는 효과를 준다. 이 효과는 단순한 이동/크기 변환이 아닌, 자연스러운 날갯짓을 표현하는 복잡한 동적 효과라고 볼 수 있다.

# 채점 시 필요한 내용

- 60초동안 좀비(Car2)들을 모두 죽이면 게임을 클리어할 수 있다. 중간에 좀비에게 닿거나, 60초가 지나면 게임이 종료된다.

- 10초가 지날 때마다 플레이어를 추격하는 좀비들의 위치가 초기화된다.

- 칼(sword)로 적을 죽이면 2점, 총(hat)으로 죽이면 1점이다.

- 매초마다 시간과 현재 점수를 출력한다.

- 게임이 종료되면 r버튼을 통해 게임을 재시작할 수 있다.

(배경 색상을 glClearColor(100 / 255.0f, 90 / 255.0f, 40 / 255.0f, 1.0f); 로 변경하였다.)