

# 인터랙티브 mini project 보고서

2016320123 이동현

---

## 개발 환경

---

## 개발 툴

- visual studio 2019 community

## 라이브러리 환경

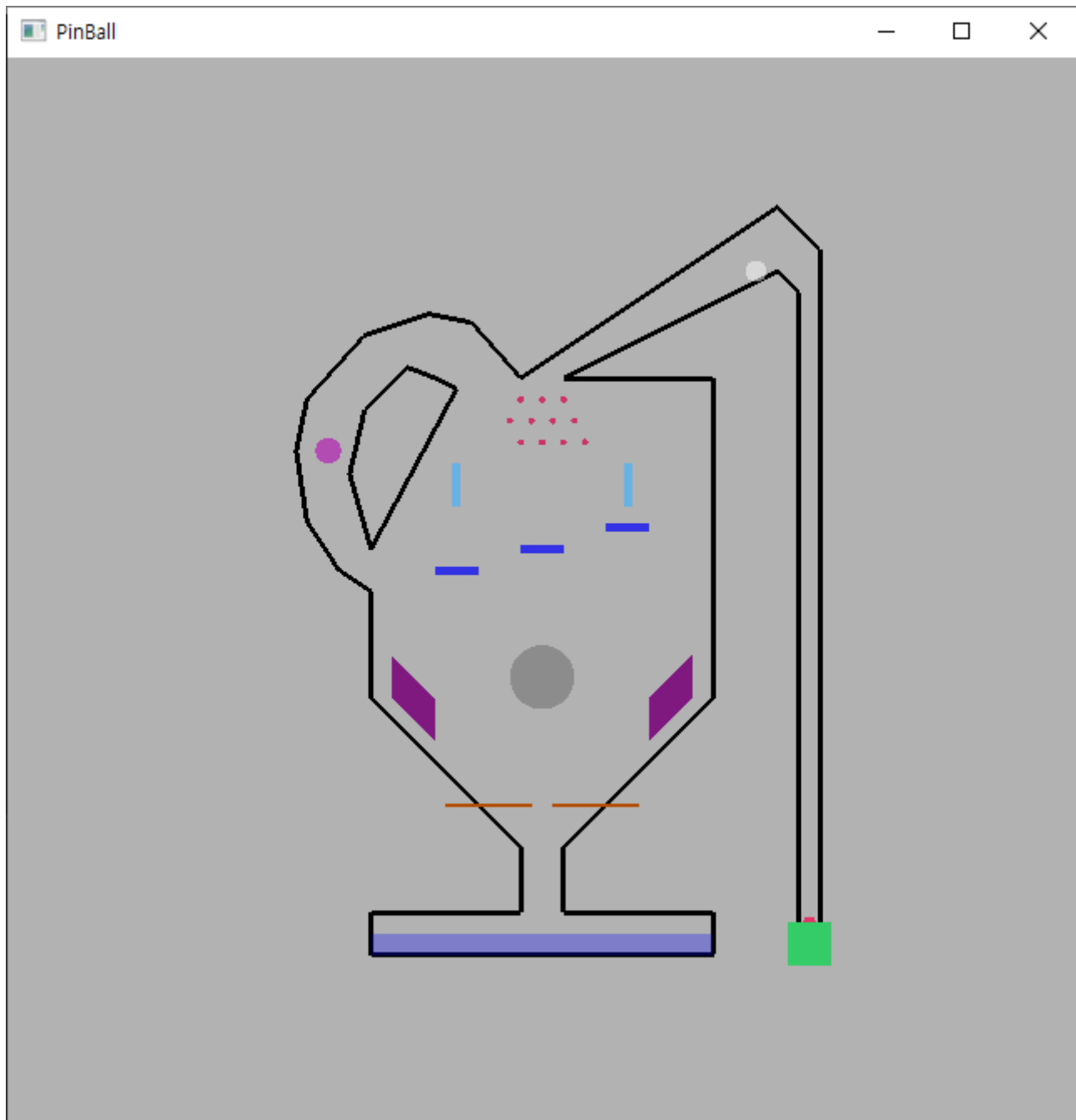
- `freeglut` 라이브러리 사용
- `Box2D` 라이브러리 사용

## 핀볼 프로젝트 설명

---

## 구현한 맵에 대한 설명

---



- 빨간색의 **pinball** object가 **map** 오른쪽 하단에 존재
- 초록색의 네모 **pinball** starter도 존재
  - 이 starter가 위로 이동하여 **pinball**을 이동시킨다.
- **map**에는 총 6가지의 장애물이 구현되어, 기존 5개의 장애물에 추가 구현된 장애물이 존재한다.
  - 1개의 원형 장애물(**보라색**)
  - 12개의 못 장애물(**빨간색**)
  - 2개의 회전하는 막대형 장애물(**하늘색**)
    - 물체의 중심은 고정되어 있고, 막대가 그 좌표를 중심으로 회전하는 형태이다.
  - 3개의 수평으로 이동하는 막대형 장애물(**파란색**)
    - **map**에 닿으면 이동하는 방향을 반대로 바꾸어 이동하게 된다.
  - 블랙홀(**회색**)과 화이트홀(**하얀색**)
    - 공이 블랙홀 주변에 가게 된다면 블랙홀 중앙으로 점차 이동하여, 결국 **map** 상단에 존재하는 화이트홀으로 순간이동하게 된다.
  - 2개의 평행사변형 장애물(**보라색**)
- 또한 왼쪽 상단의 곡선형 구조를 **map**에 추가하였다.

- `map` 아래에는 두 개의 `flipper`와, 최하단에는 물(`파란색`)이 존재한다.

## pinball simulation에 대한 설명

---

- `pinball`은 프로그램 실행 시 구현된 `map`의 오른쪽 아래 발사대 위에 위치한다.
- 프로그램 실행 후 `p`키를 누르지 않는다면 정지된 상태를 유지한다.
- `p`키를 눌러 `simulation`을 활성화시킨 후 `s`키를 눌러 초록색 네모 발사대를 위로 상승시켜 `pinball`을 출발시킬 수 있다.
  - 누른 시간만큼 발사대가 위로 올라가며, 그만큼 `pinball`은 받는 힘이 증가한다.
- 출발한 `pinball`은 `┐`자형으로 되어 있는 통로를 지나 아래로 떨어지게 된다.
- 장애물에 부딪히거나 블랙홀에 접근하면서, `pinball`이 향하는 방향과 위치는 달라지고 `flipper`로 `pinball`을 쳐서 올리거나 최하단에 위치한 물에 빠뜨리게 할 수 있다.
  - `flipper`는 왼쪽 것과 오른쪽 것 각각 `a`, `d`키로 조작할 수 있다.
  - 누르고 있는 동안은 `moter`가 동작하여, `joint`의 회전 상태가 위를 향해 있게 된다.

## 구현상의 detail

---

총 두 파일로 구성되어 있다.

- `main.cpp`
  - 전반적인 처리(`simulation pipeline`)를 담당하는 파일이다.
- `contactListener.cpp`
  - 부력 처리와, 수평으로 이동하는 막대 장애물의 방향 전환을 위해 `fixture`와 `fixture`가 충돌하는 것을 관리하는 클래스인 `MyContactListener`가 포함되어 있다.

## Box2D

- `pinball` 발사대
  - `kinematic body`로 설정하여 `static body`와는 충돌이 없게 하면서 `pinball`과는 충돌이 있게끔 구성하였다.
- `pinball`
  - `b2ChainShape`를 사용하여 `m_radius` property로 원의 반지름 정보를 유지하면서 사용하였다.
  - 구현된 기타 원형 장애물도 이와 동일하다.
- `ground (map)`
  - 검은색 선의 `map`은 `b2ChainShape`를 사용하여 구현하였다.
- 블랙홀 & 화이트홀
  - 블랙홀 중심과 공 중심 사이 거리에 비례하여 수치를 잡아, 사잇거리가 그 수치 안에 들어오면 공의 원래 속도와 중력에 영향을 받는 정도를 거리에 비례하여 줄이고, 블랙홀 중심으로 향하는 힘을 사잇거리에 반비례하여 공에 가하여 구현하였다.
  - 블랙홀의 중심에 상당히 가깝게 공이 접근하면, 받는 중력을 0으로 설정하고 속도도 0으로 설정하여 제자리에 있게 설정한 다음, 다음 `timestep`에서 화이트홀 중심으로 공을 이동시켰다.
  - 위 내용은 `main.cpp`의 `Update()`에 구현되어 있다.

```

b2Vec2 distvec = bhpoint - ballpoint;
// 2-norm?
float dist = distvec.Length();
if (!blackhole_captured) {
    if (dist <= bh_radius * 3.0) {
        float num = abs(distvec.x) + abs(distvec.y);
        b2Vec2 rev_distvec = distvec;
        distvec *= ((3.0f / num) * bh_radius / dist);

        bool check = false; // 블랙홀 중앙에 들어갔을 때 힘을 안 줘야 하나?

        if (dist <= bh_radius * 1.3) {
            b2Vec2 cur_velocity = ball->GetLinearVelocity();
            float cur_angle_velocity = ball->GetAngularVelocity();
            ball->SetLinearVelocity((dist / (bh_radius * 1.3)) *
cur_velocity);
            ball->SetAngularVelocity((dist / (bh_radius * 1.3)) *
cur_angle_velocity);

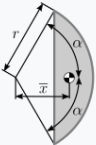
            if (dist / bh_radius * (3.0) <= 0.1f) {
                ball->SetLinearVelocity({ 0,0 });
                ball->SetAngularVelocity(0);
                ball->SetGravityScale(0);
                blackhole_captured = true;
                check = true;
            }
            // rev_distvec *= ((num / 1.0f) * dist / bh_radius);
            // ball->ApplyForce(rev_distvec, ball->GetWorldCenter(),
false);
        }

        if (!check) {
            ball->SetGravityScale(dist / (bh_radius * 3.0));
            ball->ApplyForce(distvec, ball->GetWorldCenter(), false);
        }
    }
    else
        ball->SetGravityScale(1.0f);
}
else {
    // 순간이동
    ball->SetTransform(whitehole_pos, 0.0f);
    blackhole_captured = false;
    ball->SetGravityScale(1.0f);
}
}

```

- 평행사변형 장애물

- `static body`로 설정하였으며, `map`과 평행사변형 장애물 사이로 공이 지나갈 수 있게끔 구현하였다.
- 회전 막대 장애물
  - `b2RevoluteJoint`를 사용하였다.
  - `Setup()`에서 모터의 속도를 특정한 수치로 설정하여 계속 회전하게 하였다.
- 수평 이동 막대 장애물
  - `b2PrismaticJoint`를 사용하였다.
  - `map`과 장애물을 위 joint로 연결하여 1차원 이동(수평 방향)만 가능하게 하였다.
  - 벽과 충돌하였을 때 막대의 이동 방향을 반대로 바꾸기 위해, `ContactListener`를 사용하여 두 `fixture`가 충돌하는 상황을 검사하였다.
- flipper
  - `b2RevoluteJoint`를 사용하였다.
  - 최대각과 최소각을 설정하여 그 각도 범위 내에서만 움직일 수 있도록 하였다.
  - 사용자가 a키나 d키를 누를 때마다, `SetMotorSpeed()`로 각 키에 해당하는 flipper의 `motor speed`를 설정하였다.
    - `main.cpp`의 `Update()` 함수에 구현되어 있다.
- 부력
  - 수업 ppt의 내용에 기반을 두고 구현되었다.
  - 수업 ppt에서는 `intersection point`를 찾고 `centroid`와 두 물체 간 겹치는 넓이를 찾았지만, `pinball`이 `circle shape`이기 때문에, 공이 물에 잠긴 상황을 총 세 가지로 나누고 그에 따라 겹치는 넓이와 무게중심을 찾아 구현하였다.
    - 공의 절반 이하가 물에 잠긴 경우, 공의 절반 이상이 물에 잠긴 경우, 공이 물에 완전히 잠긴 경우로 나누었다.
    - 무게중심을 구한 식은 [wikipedia](#)를 참고하였다.

Circular segment		$\frac{4r \sin^3(\alpha)}{3(2\alpha - \sin(2\alpha))}$	0	$\frac{r^2}{2}(2\alpha - \sin(2\alpha))$
------------------	---	--	---	--

- 나머지 구현(ex : `applybuoyancy()`와 `applydrag()` 함수 등)은 수업 ppt와 동일하다.
- `main.cpp`의 `Update()` 함수에 구현되어 있다.(684번째 줄~)

## OpenGL

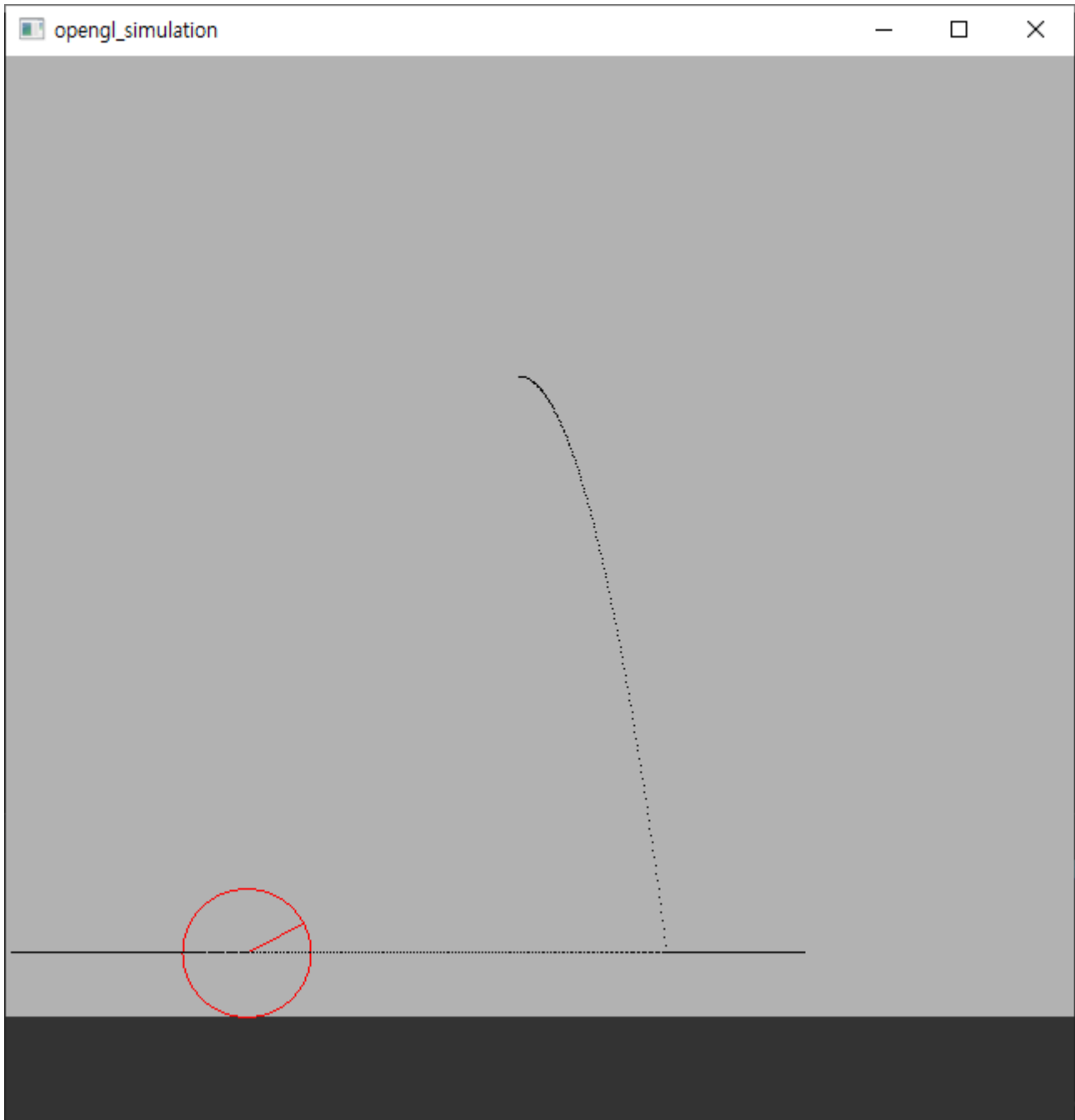
- `DoReleasekey()`와 `Dokeyboard()` 함수를 각각 `glutKeyboardUpFunc()`와 `glutKeyboardFunc()`의 콜백 함수로 두어, 사용자의 키보드 입력을 감지하였다.
- 원형 장애물의 rendering
  - 중심점을 기준으로 1도마다 원을 구성하는 점을 그려 `polygon` 형태로 rendering을 수행하였다.

```
// Draw pinball!
glBegin(GL_POLYGON);
float radius = ballshape.m_radius;
for (double angle = 0.0f; angle <= 360.0f; angle += 1.0f) {
    double degtorad = angle * b2_pi / 180.0;
    glVertex2f(cos(degtorad) * radius, sin(degtorad) * radius);
}
glEnd();
```

- `GL_PROJECTION` 모드로 카메라 시점을 약간 변경하였다.
- 블랙홀, 화이트홀 등의 색깔을 나타낼 때 홀 안의 공이 들어갔을 때 공이 보이도록 `GL_BLEND`를 사용하여 투명함을 구현하였다.

## OpenGL 물리 시뮬레이션에 대한 설명

---



- `box2d`를 사용하지 않고 `opengl`으로 물리 시뮬레이션을 구현하였다.
- `pinball project`와 동일하게 `p`키를 눌러 시뮬레이션을 시작해야 한다.
- 화면 중앙 위쪽에서 공의 위치가 시작되며, 포물선을 그리며 땅으로 떨어진 다음 키보드의 `a`, `d`키로 공을 각각 왼쪽, 오른쪽으로 굴릴 수 있다.
- 땅과 공 사이의 마찰 계수(`friction` 변수)와 공기 저항(`air_power` 변수)이 구현되었다.
  - 공을 굴리다가 아무 입력도 넣지 않고 가만히 있으면, 공이 구르다가 점차 멈추는 것을 확인할 수 있다.

### 구현 detail에 관한 설명

- `Update()` 함수에서 시뮬레이션 경우를 크게 두 가지로 나누었다.
  - 공이 땅바닥에 닿지 않은 경우와 닿은 경우
  - 전자는 `y`축 방향으로의 중력에 해당하는 힘을 가하여 가속도, 속도, 위치를 조절하였고 후자는 키보드 입력에 따라 `x`축 방향으로의 힘을 가하여 가속도, 속도, 위치를 조절하였다.

- 키보드 입력은 `pinball project`와 동일하게 `glutKeyboardFunc()`와 `glutKeyboardUpFunc()` 함수를 사용하였다.