인터렉티브 Final project 보고서

2016320123 이동현

개발 환경

개발 툴

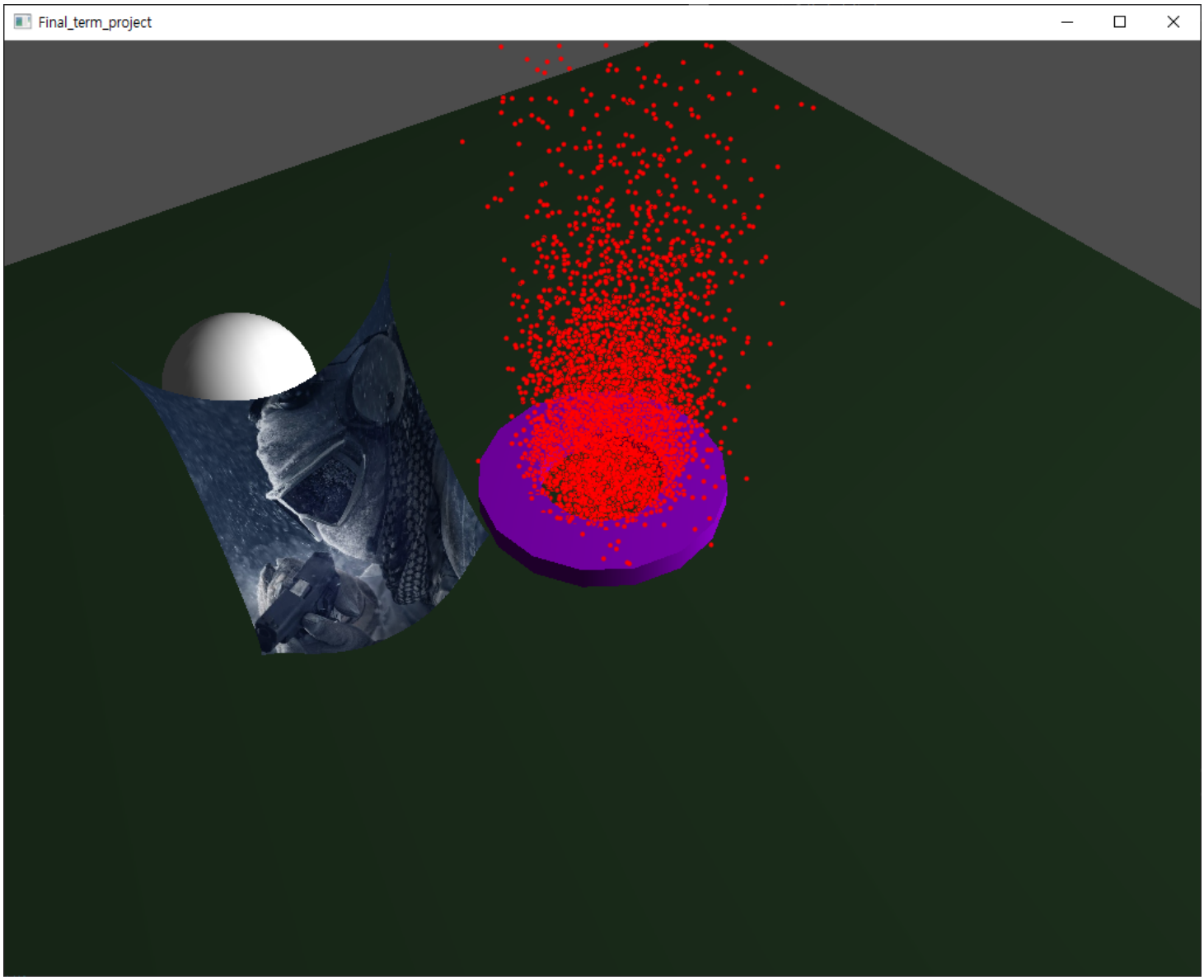
- visual studio 2019 community

라이브러리 환경

- `freeglut` 라이브러리 사용
- `glm` 라이브러리 사용

물리 시뮬레이션 프로젝트 설명

구현한 맵에 대한 설명



- 맵은 전체적으로 초록색 평면 위에 존재
- 구성요소
  - 맵 중앙에 위치한 굴뚝
    - particle system을 시뮬레이션하는 역할
  - 맵 왼쪽에 위치한 `cloth`
  - `cloth` 뒤쪽에 위치한 구
- 이 맵에서는 particle system + rigidbody simulation + deformable body simulation 세 가지를 구현하였다.

조작법에 대한 설명

키보드

- `p` : 시뮬레이션 일시정지(토글)
- `q` : 시뮬레이션 종료

- `f` : cloth simulation에 외부 힘을 주기(토글 - 다시 누르면 외부 힘을 없앰)
- `g` : cloth 뒤에 위치한 구에 힘을 주기(토글 - 다시 누르면 외부 힘을 없앰)
- `r` : 시뮬레이션을 reset한다.

마우스

- 왼쪽 클릭하여 드래그 : 해당 방향으로 시점 `rotation` 을 할 수 있다.
- 휠을 클릭하여 드래그 : 해당 방향으로 시점 `translation` 을 할 수 있다.
- 왼쪽 클릭하여 드래그 : 시점 `zoom` 을 할 수 있다.

particle system simulation에 대한 설명

- 보라색 굴뚝 안쪽에서는 빨간색을 띈 파티클이 계속 뿜어져 나온다.
- 빨간색 불에 가깝게 형상화

구현상의 detail

해당 부분이 구현된 주요 클래스는 다음과 같다.

- `Particle_System.cpp`
- `Particle_System.h`
- `Particle.cpp`
- `Particle.h`

`Particle_System.xx` 는 전체 파티클 시스템을 관리하며 `Particle.xx` 는 개별 파티클을 관리한다.
처음에는 약 10000개의 파티클으로 시작하며(`NUMBER_OF_PARTICLE` 변수), update 시마다 100개의 파티클을 추가로 생성한다. 각 파티클의 초기 위치와 속도와 lifetime은 랜덤하게 정하였으며, 굴뚝 안의 원을 기준으로 그 안에서 랜덤하게 정해져 나온다. 하지만 불을 형상화하기 위해 굴뚝 중심의 y값에 파티클의 y값이 가까울수록 velocity가 높게 나올 확률이 있게끔 하였다.

```
float x = (rand_zero_center_float() * (MAX_INIT_VELOCITY - MIN_INIT_VELOCITY) + MIN_INIT_VELOCIT
        float y = (rand_zero_center_float() * (MAX_INIT_VELOCITY - MIN_INIT_VELOCITY + diff) + M
        float z = (rand_zero_center_float() * (MAX_INIT_VELOCITY - MIN_INIT_VELOCITY) + MIN_INIT
```

또한, 렌더링 부분에서는 약간 투명하게 불 뒷편이 보일 수 있게 `GL_BLEND` 를 사용하였다. 노말 벡터는 임의로 정하였다.

```
void ParticleSystem::draw() {
	glPushMatrix();

	// TODO : particle draw
	glEnable(GL_POINT_SMOOTH);

	glEnable(GL_BLEND);		// Enable Blending
	glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);	// Type Of Blending To Perform

	for (int i = 0; i < particles.size(); i++) {
		vec3 pos = particles[i].position;
		// color : red
		vec3 color = vec3(1, 0, 0);
		glColor3f(color.x, color.y, color.z);
		glPointSize(1.0f);

		glBegin(GL_POINTS);
		vec3 light_pos(150.0, 151.0, 150.0);
		vec3 normal = normalize(light_pos - pos);
		glVertex3f(pos.x, pos.y, pos.z);
		glNormal3f(normal.x, normal.y, normal.z);
		glEnd();
	}

	glDisable(GL_BLEND);

	glPopMatrix();
}
```

deformable body(cloth) simulation에 대한 설명

cloth는 `50*50*1` 개의 노드로 구성되었으며, 시뮬레이션 초반에는 y축과 수직하게 위치했다가 시뮬레이션을 시작하면 점차 내려오게 된다. 두 개의 node의 위치가 고정되어 빨래집게에 매달린 것처럼 시뮬레이션이 되는 것을 확인할 수 있다. `f` 키로 바람을 형상화하여 외부 힘을 줄 수 있다. 또한 모두 내려오게 되면 cloth 끝이 바닥에 닿아 충돌 처리가 되어 조금 접히게 되고, 내려오면서 주름이 생기는 것을 볼 수 있다. 또한 라이팅을 위한 노멀 벡터가 정점마다 계산되고 텍스쳐 좌표도 계산되어 라이팅과 텍스쳐링이 완성된 것을 볼 수 있다.

구현상의 detail

해당 부분이 구현된 주요 클래스는 다음과 같다.

- `Deformable_System.cpp`
- `Deformable_System.h`
- `node.cpp`

- `Mass_spring.cpp`

`node.cpp` 는 각 node를 나타내는 class이고 `Mass_spring.cpp` 는 각 node를 잇는 각 mass spring을 나타내는 class이다. `Deformable_System.xx` 는 전체의 큰 하나의 deformable system을 관리한다.

충돌 처리는 땅과 노드 사이의 충돌만 구현하고 node-face, node-node 간 충돌은 구현하지 않았다.
땅과 노드 사이의 충돌은 particle-boundary 간 충돌 처리로 모델링하였다.
수업 식을 대체로 따라가나 일정 계수는 시뮬레이션이 잘 되지 않아 바뀐 계수가 존재한다.
구현상의 특징은 face까지 만들어 face당 노멀 벡터를 계산하여 각 정점당 노멀 벡터를 계산한 부분과, 텍스쳐 좌표를 계산하여 텍스쳐링을 한 부분이 된다. 특히 face를 렌더링할 때, 어떻게 할지 감이 오지 않아 face를 triangle로 보고 `GL_TRIANGLES` 를 이용하여 삼각형들로 렌더링을 했더니 잘 되는 것이 신기하였다.

```
void mass_cloth::RenderFace(int width, int height) {
        // render face
        // texture도 매핑
        glEnable(GL_TEXTURE_2D);

        glBegin(GL_TRIANGLES);
        for (int i = 0; i < faces.size(); i+=3) {
                for (int j = 0; j < 3; j++) {
                        Node* np = faces[i + j];
                        glTexCoord2f(np->tex_coord.x, np->tex_coord.y);
                        glNormal3f(np->normal.x, np->normal.y, np->normal.z);
                        glVertex3f(np->position.x, np->position.y, np->position.z);
                }
        }
        glEnd();

        glDisable(GL_TEXTURE_2D);
}
```

euler에 관해서는, 일반 오일러 식이 아니라 `sympletric euler` 방식을 사용하였다. time step에 관해 훨씬 더 안정적이라고 한다.

```
void mass_cloth::integrate(double dt) {
        // 보다 나은 기법
        // Symplectic Euler
        int n_size = nodes.size();

        for (int i = 0; i < nodes.size(); i++) {
                if (!nodes[i]->isfixed) {

                        // integration
                        nodes[i]->velocity *= nodes[i]->damping_force;
                        vec3 temp = nodes[i]->position;

                        nodes[i]->velocity += (nodes[i]->force / nodes[i]->mass) * (float)(dt);
                        nodes[i]->position += nodes[i]->velocity * (float)(dt);
                        nodes[i]->old_position = temp;

                }
                nodes[i]->force = vec3(0, 0, 0);
        }

}
```

rigidbody simulation에 대한 설명

cloth 뒤에는 하얀색 3D 구가 존재한다. 이 구의 목적은 cloth에 충돌시켜 충돌 시뮬레이션을 하는 것과, ground와 구 간의 충돌 시뮬레이션을 하는 것에 의의가 있다. `G` 키를 누르면 공을 cloth가 있는 방향으로 힘을 주어 이동시킬 수 있다. 또한 공이 구르거나 가만히 있는 경우 설정된 ground와 충돌이 되는 것을 볼 수 있다.

구현상의 detail

해당 부분이 구현된 주요 클래스는 다음과 같다.

- `Sphere.cpp`

구현상에서 눈여겨 볼 것은 공과 cloth 간 충돌 처리, 공과 땅 간의 충돌 처리이다.
공과 cloth 간의 충돌 처리는 particle-particle 간의 충돌 처리로 모델링하였고, 공과 땅 간의 충돌 처리는 particle-boundary 간의 충돌 처리로 모델링하였다.

```
void collision_response(mass_cloth* cloth, vec3 ground) {
                // 구와 땅 충돌 + 구와 옷감 충돌

                // ground collision
                vec3 ground_normal(0, 1, 0);
                if (this->position.y - ground.y <= this->radius && dot(ground_normal, this->velo
                        // collision detected
                        // cout << "collision_response" << '\n';

                        vec3 v_n = dot(ground_normal, this->velocity) * ground_normal;
                        vec3 v_t = this->velocity - v_n;
                        this->velocity = v_t - (float)0.1*v_n;
                }

                // 구와 옷감 충돌
                // 파티클-파티클 간 충돌으로 모델링
                for (int i = 0; i < cloth->nodes.size(); i++) {
                        vec3 pos_ij = cloth->nodes[i]->position - this->position;
                        vec3 pos_ji = this->position - cloth->nodes[i]->position;
                        float distance = length(pos_ij);
                        float collidedist = this->radius + cloth->nodes[i]->radius;
                        // 공과 각 노드마다 수행
                        if (distance <= collidedist) {
                                vec3 normal1 = (pos_ij) / (float)distance;
                                vec3 normal2 = (pos_ji) / (float)distance;
                                vec3 vn1 = normal1 * (dot(normal1 , this->velocity));
                                vec3 vn2 = normal2 * (dot(normal2, cloth->nodes[i]->velocity));

                                vec3 vt1 = this->velocity - vn1;
                                vec3 vt2 = cloth->nodes[i]->velocity - vn2;

                                vec3 relvel = cloth->nodes[i]->velocity - this->velocity;
                                float mass_sum = this->mass + cloth->nodes[i]->mass;

                                if (dot(relvel, pos_ij) < 0) {
                                        // collision detected
                                        this->velocity = (2.0f * vn2 * cloth->nodes[i]->mass + v

                                        cloth->nodes[i]->velocity = (2.0f * vn1 * (float)this->m

                                }

                        }

                }

        }
```

어려웠던 점

처음 쓰는 API가 많았고 수식 계산이 들어가다 보니 계산 실수라든지 코딩 실수가 들어가서 시간이 많이 지연되었다.
디버깅하는 시간도 많이 들어갔다. 아쉽게 생각한다.