객체지향 프로그래밍 프로젝트 보고서

201511033 김민수
201511034 김민재
201511058 김태연

서론

이번 객체지향 프로그래밍을 통해서 교수님 제안 주제인 N-BODY 시뮬레이션으로 프로젝트를 진행하게 되었다. 처음에는 N-BODY라는 주제가 어려워 보이고, 그렇게 재미있어 보이지도 않았지만, 이 프로젝트의 의의를 생각해보고 이 주제로 해야 겠다고 결심하게 되었다. N-BODY 시뮬레이션을 구현함으로써 기초학부동안 배웠던 기초학문을 이용하여 다른 분야를 공부 할 수 있다는 점이 우리 학교의 융복합교육과 상당히 잘 맞는다고 생각이 되었고, 나중에도 각자의 분야에서 프로그래밍을 해야 되는 경우가 생겼을 때 이렇게 다른 분야의 개념을 이용해서 해야 되는 경우가 많을 것이라고 판단하여 N-BODY의 주제로 하면 좋은 경험이 될 것이라고 생각하였다. 또한, N-BODY 시뮬레이션을 구현하면서 여러 data structure 사이의 관계를 구조적으로 파악하고 이어주는 작업을 연습하는 것에도 많은 도움이 될 것이라고 생각했다. 조금 더 구체적으로 말하면, 이번 객체지향 프로그래밍 과목에서 class들의 상속관계 둥에 대해서 공부를 하고, 그 예시를 수업시간을 통해서 몇가지를 확인을 했는데, 그 예시들은 비교적 class 사이의 관계가 확실한 경우들이 많았다. 하지만, 이번 N-BODY 프로젝트에서는 일단 class의 관계가 눈에 바로 들어오지 않기 때문에, class간의 관계를 여러 방향으로 고민해 볼 수 있어 좋은 연습이 될 것이라고 생각했다.

N-BODY 시뮬레이션에 대해서 간단히 설명을 하면, 여러 가지의 입자가 서로 상호작용을 하는 것을 시뮬레이션 하는 것으로, 기본적으로는 각 입자들 사이에 만유인력이 작용한다는 것을 이용해서 각 입자들의 운동을 예측해주는 것이다. 이번 프로젝트에서는 time-tick을 정해서 그 시간 간격으로 duration의 시간동안 측정을 하게 된다. 또한, 이러한 N-BODY 시뮬레이션의 대표적인 예시가 우주에서의 운동이 될 수 있을 텐데, 이 경우에는 각 입자사이의 만유인력 뿐만 아니라 외력으로 전자기력, 팽창하는 우주 모델이라면 이로 인한 외력 등 여러가지의 다른 힘이 부분별로 작용할 수 있다. 그래서 이러한 부분도 구현해주기 위해 set이라는 입자의 집합을 만들어주어 set단위로 외력을 넣어줄 수 있도록 하였다. 또한, 추가적으로 중력이 사라졌을 때의 운동도 확인할 수 있게 하기 위해서 중력을 꺼줄 수 있는 기능과, 태양계와 같은 계를 시뮬레이션 할 때 태양과 같은 입자를 고정시키기 위한 입자의 고정 기능까지 구현을 해주었다.

본론

일단 프로그램이 받는 입력과 프로그램의 기능은 N-BODY의 instruction과 동일하게 구현해주었다.

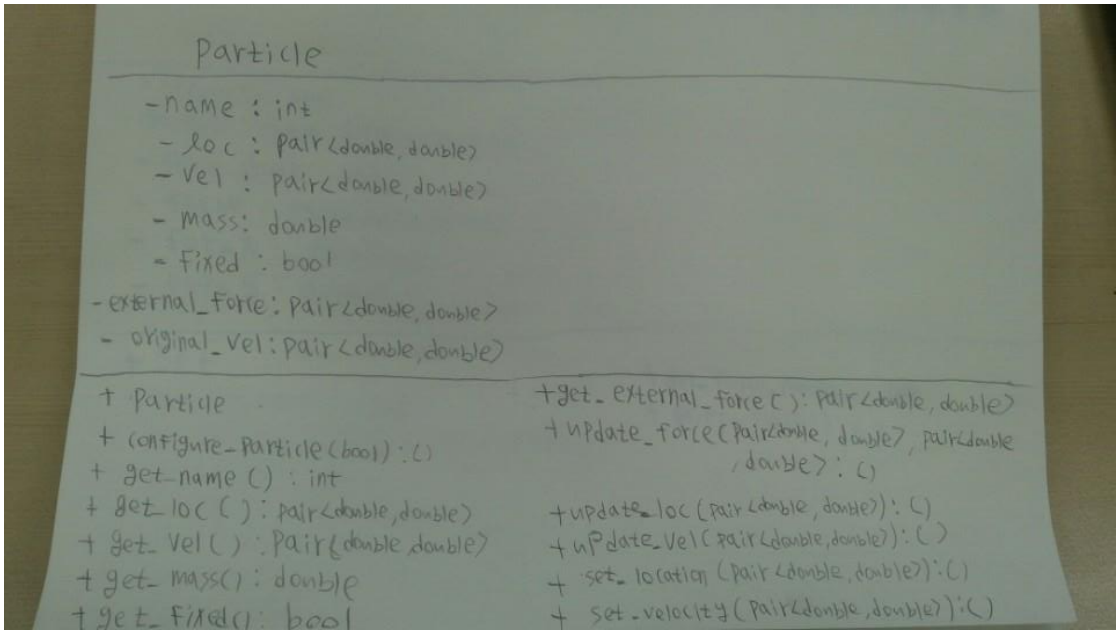
다음으로, 우리의 프로젝트에서 정의해준 class들에 대해서 설명을 하도록 하겠다.

1. Particle

첫번째 class는 Particle이라는 class이다.

이 class에서는 이름에서도 알 수 있듯이 입자 하나하나의 정보를 저장하며, 이와 관련된 함수들이 구현 되어있다. Member variable로는 int형으로 입자의 이름과 double의 pair로 입자의 위치와 속도, double형의 입자의 질량, 고정 여부를 저장해주는 bool, double의 pair로 외력을 가지고 있게 된다. 다른 member variable들은 입자의 기본 정보로, 함수 구현에서도 그 정보를 출력하고, 갱신해주는 함수가 구현되어 있다. 외력에 대한 정보의 경우는 입자가 생성될 때는 default로 x방향의 힘과 y방향의 힘이 0으로 생성이 되며, set 단위로 force가 추가될 때 그 set에 있는 입자들에 대해서 그 만큼의 외력이 입자의 member variable에서 더해진다. 또한, 외력을 제거해줄 때 역시 입자 안에서 그 크기만큼의 힘을 빼 주는 형식으로 구현이 된다. 또한 위치와 속도를 업데이트하는 함수를 구현하여 시뮬레이션을 돌릴 때 각 입자의 위치와 속도를 갱신할 수 있도록 했다.

Class diagram을 그린 것을 보면 다음과 같다.

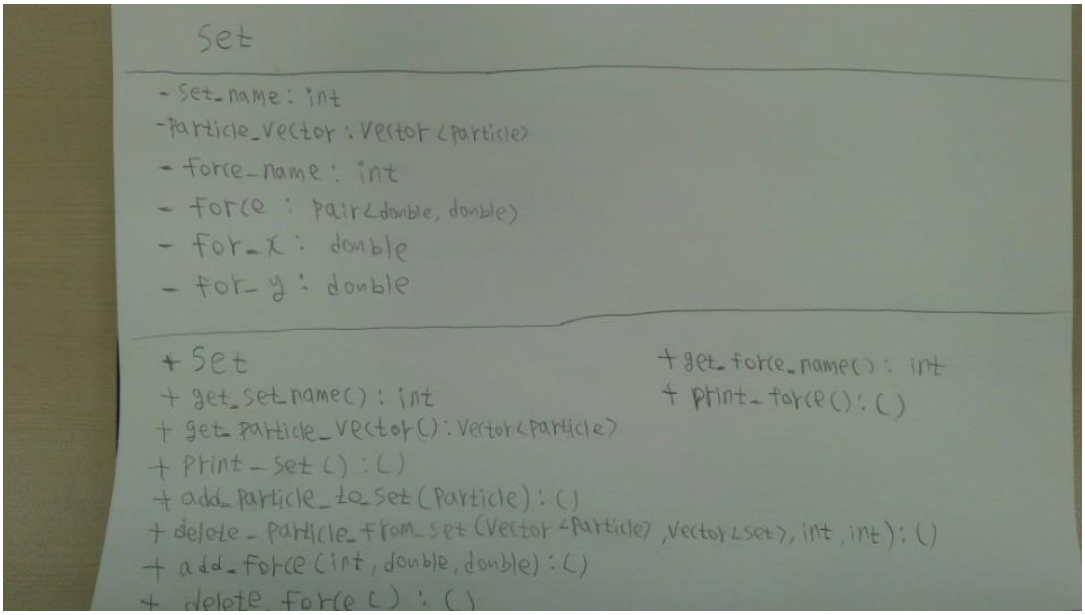


2. Set

다음으로는 입자의 외력을 넣어 주기 위한 집합 개념인 Set이라는 class이다.

Set은 Particle로부터 상속을 받지는 않지만 member variable로 Particle이라는 class의 벡터를 받게 된다. 그리고 그 외에도 set이름과 외력의 이름과 force의 double형 pair를 받게 된다. Set에서의 기능을 위해 가장 중요한 기능은 외력을 추가하고, 제거해주는 부분이다. 처음 Set이 생성될 때는 외력에 대한 정보가 없는 경우가 많을 것이다. 그래서 생성될 때는 다른 조건이 없으면 외력의 크기는 (0,0)으로, 외력의 이름은 -1의 값이 default로 만들어진다. 그리고 af명령어로 force가 추가될 때 add_force라는 함수가 실행되며 이 함수는 member variable인 외력의 이름과 크기를 갱신해주고, 이 외력의 크기를 member variable particle vector에 있는 모든 입자들에 대해서 외력부분을 갱신해준다. 그리고 df명령어로 force가 없어질 때는 delete_force라는 함수가 실행되며 이 함수는 기존에 있던 force의 음의 값 만큼 member variable로 있는 입자들에 대해서 갱신을 해주고, force의 크기는 (0,0)으로, 이름은 -1으로 다시 default값으로 바꾸어 준다.

Class diagram을 그린 것을 보면 다음과 같다.



다음으로는 class외에 구현이 되어있는 주요한 함수에 대해 설명을 하도록 하겠다.

1. calculate 함수

   시뮬레이션의 가장 중요한 입자의 위치와 속도를 계산해주는 부분이다.

   기본적으로 이 함수가 한 번 불려지면 인수로 전체 입자의 벡터와 timetick과 gravity를 받아서 한 번의 timetick 이후의 모든 입자들의 위치 및 속도를 구해주게 된다. 크게 중력이 있을 때와 없을 때를 나눠서 계산이 이루어지며, 중력이 있는 경우에는 각각 두 입자들에 대해서 힘을 구한 후에 더해주며 이 때 벡터합을 해줘야 하기 때문에 각각 입자들에 대해서 힘의 크기와 방향을 크기가 1인 단위벡터로 만든 후에 x방향과 y방향으로 각각 더해준다. 그리고 그 합력에 외력을 추가해주기 위해 입자의 member variable로 있는 힘들을 방향별로 더해주어 최종 합력을 구해주며, 외력이 없는 경우는 외력이 (0,0)으로 구현되어 있기 때문에 문제 없이 구현이 된다. 이렇게 각 입자별로 timetick에 대해서 받는 힘과 가속도를 구해주며, 각각의 timtick 안에서는 등가속도운동을 한다고 가정을 한 후 timtick 이후의 입자의 위치와 속도를 구해준 후, 그 입자의 정보를 갱신해준다. 그러면 이 함수가 한 번 실행이 되고 난 후에는 모든 입자들의 정보가 하나의 timetick이후의 정보로 갱신이 되어 있게 된다.

   하지만 이 함수 구현에서는 몇가지 한계가 있다. 일단, 두 입자의 충돌은 고려를 하지 못한다. 두 입자의 충돌을 고려하기 위해서는 두 입자가 충돌 할 때 충돌각에 따른 충돌계수를 알고, 이를 통해서 계산을 해야 하고, 어느 경우에 두 입자가 하나가 되어 운동을 하고, 어느 경우는 두 입자가 충돌한 후 다시 나누어지는지도 알아야 하는데, 이에 대한 정보를 정확히 구해주기가 힘들어서 이 부분의 구현은 생략하였다. 또한 N-BODY 시뮬레이션에서는 기본적으로 입자들을 하나의 질점으로 본다. 따라서 충돌이 일어나기 위해서는 두 입자의 좌표가 정확히 일치하게 되어야 만 하며, 일반적인 경우에서 이러한 상황이 일어날 확률은 매우 낮아서 고려를 하지 않아도 된다고 판단하였다. 두번째 한계점은 두 입자가 매우 가까

워지면 만유인력을 구하는 식에 따라서 그 힘의 크기가 매우 커지게 되는데, 힘이 double 형 변수의 범위보다 커지면 음의 값으로 표현이 될 수 있다는 것이다. 이 부분에 대해서는 일반적인 경우에는 double 인수의 범위가 매우 크기 때문에 일어나지는 않을 가능성이 높지만, 혹시나 overflow가 발생하여 음의 값이 되었을 때는 두 입자 사이의 힘의 크기는 double형 인수의 최대값으로 표현하도록 조건문을 추가해주었다. 이 함수의 구현을 코드와 함께 보면 다음과 같다.

```
13  void calculate (vector<Particle> &default_Particles, int timetick, bool gravity)  //Set의 변수명은 'S' + Set_name으로 가정, 외력은 없으면 (0,0)으로 입력
14  {
15      //vector<Particle> default_Particles = default_default_Particles;
16
17      int size = default_Particles.size();
18      vector<int> names;
19      vector<double> x_loc;
20      vector<double> y_loc;
21      vector<double> x_vel;
22      vector<double> y_vel;
23      vector<double> mass;
24      //vector<double> distance;
25      vector<double> force;
26      vector<double> vec_x;
27      vector<double> vec_y;
28      for (int i = 0; i < size; ++i)  //계산하기 위해서 set의 모든 정보들을 뽑아오는 과정
29      {
30          int name = default_Particles[i].get_name();
31          pair<double, double> loc = default_Particles[i].get_loc();
32          pair<double, double> vel = default_Particles[i].get_vel();
33          double m = default_Particles[i].get_mass();
34          double x_loc1 = loc.first;
35          double y_loc1 = loc.second;
36          double x_vel1 = vel.first;
37          double y_vel1 = vel.second;
38          names.push_back(name);
39          x_loc.push_back(x_loc1);
40          y_loc.push_back(y_loc1);
41          x_vel.push_back(x_vel1);
42          y_vel.push_back(y_vel1);
43          mass.push_back(m);
44      }
```

일단, 이 함수는 모든 입자들을 저장하고 있는 default_Particles라는 벡터의 주소값과, timetick과 gravity의 여부를 인수로 받는다. 처음에는 계산의 중간 결과를 저장해주기 위한 여러 가지의 vector들을 초기화해주며, default_Particles들에 대해서 각 입자들의 질량, 위치, 속도에 대한 정보를 받아와서 저장을 해준다.

```
    if (gravity == 1)
    {
        for (int i = 0; i < size; ++i)
        {
            if (default_Particles[i].get_fixed() == 0) //fix 안 되어 있을 때만 계산
            {
                for (int j = 0; j < size; ++j)
                {
                    if (i != j)
                    {
                        //거리 구하기
                        double r_sq = pow(x_loc[i] - x_loc[j], 2) + pow(y_loc[i] - y_loc[j], 2);
                        //힘 구하기
                        double F = G*mass[i] * mass[j] / r_sq;
                        if ( F < 0)
                        {
                            F = MAX_DOUBLE;
                        }
                        //단위 벡터 구하기
                        double vec_x1 = (x_loc[j] - x_loc[i]) / sqrt(r_sq); // pow(r_sq, 0.5);
                        double vec_y1 = (y_loc[j] - y_loc[i]) / sqrt(r_sq); // pow(r_sq, 0.5);
                        //distance.push_back(r);
                        force.push_back(F);
                        vec_x.push_back(vec_x1);
                        vec_y.push_back(vec_y1);
                    }
                }
                double f_x = 0.;
                double f_y = 0.;
                //double unit_vec_x = 0.;
                //double unit_vec_y = 0.;
                int forcesize = force.size();
```

다음으로는 중력이 있을 경우에 계산을 해주는 과정이다. 일단 입자 하나씩 차례대로 계산을 해주며, 하나의 입자에 대해서 나머지 입자들과의 힘을 계산해준다. 이 힘은 벡터 값이기 때문에 이 방향을 저장해주기 위해서 각각의 힘에 대해서 방향을 나타내기 위한 단위벡터를 계산해준 뒤 이 값들을 각각 미리 생성해둔 벡터에 순서대로 저장해준다.

```
                }
                double f_x = 0.;
                double f_y = 0.;
                //double unit_vec_x = 0.;
                //double unit_vec_y = 0.;
                int forcesize = force.size();
                for (int j = 0; j < forcesize; ++j)
                {
                    //합력 구하기
                    f_x = f_x + force[j] * vec_x[j];
                    f_y = f_y + force[j] * vec_y[j];
                    //방향 구하기
                    //unit_vec_x = unit_vec_x + vec_x[j];
                    //unit_vec_y = unit_vec_y + vec_y[j];
                }
                pair<double, double> external_force = default_Particles[i].get_external_force();  //파티클에 return값이 pair가 되도록 구현해야 함
                //외력 더해주기
                f_x = f_x + external_force.first;
                f_x = f_x + external_force.second;
                //가속도 구하기
                double a_x = f_x / mass[i];
                double a_y = f_y / mass[i];
                //나중속도 구하기
                double vel_x_later = x_vel[i] + a_x * timetick;
                double vel_y_later = y_vel[i] + a_y * timetick;
                //이동거리 구하기
                double s_x = (x_vel[i] + vel_x_later)*timetick / 2;
                double s_y = (y_vel[i] + vel_y_later)*timetick / 2;
                // 이동 후 위치 구하기
                double x_loc_later = x_loc[i] + s_x;
                double y_loc_later = y_loc[i] + s_y;
                pair<double, double> location = make_pair(x_loc_later, y_loc_later);
                pair<double, double> velocity = make_pair(vel_x_later, vel_y_later);
```

다음으로는 힘의 합력을 구해준다. 각각 힘들에 대해 단위벡터들을 구해놨으니, 이를 힘과 곱해주어 각각 x축과 y축 방향의 힘의 합을 f_x와 f_y로 구해준 뒤, f = ma를 이용해서 x와 y축으로 각각 가속도를 구해주고, timetick 안에서는 등가속도 운동이라고 가정하였기 때문에 등가속도 운동의 공식에 맞춰서 나중속도와 이동거리를 구해준다. 이동 거리를 구한 후에는 이동 후 입자의 위치를 구해주며, 이동 후 입자의 위치와 나중속도는 pair로 만들어

준다.

```
            pair<double, double> location = make_pair(x_loc_later, y_loc_later);
            pair<double, double> velocity = make_pair(vel_x_later, vel_y_later);
            default_Particles[i].update_loc(location);
            default_Particles[i].update_vel(velocity);
            force.clear();
            vec_x.clear();
            vec_y.clear();
        }

    }
}
else
```

그 후 그 만든 pair들을 각각의 particle의 member variable의 위치와 속도 pair로 업데이트를 시켜주고, 써준 벡터들을 모두 지워 준다. 그리고 이 과정을 for문을 이용해서 모든 입자들에 대해서 반복해준다.

```
113     else
114     {
115         for (int i = 0; i < size; ++i)
116         {
117             if (default_Particles[i].get_fixed() == 0) //fix 안 되어 있을 때만 계산
118             {
119                 pair<double, double> external_force = default_Particles[i].get_external_force();
120                 if (external_force.first == 0. && external_force.second == 0.)
121                 {
122                     double x_loc_later = x_loc[i] + x_vel[i] * timetick;
123                     double y_loc_later = y_loc[i] + y_vel[i] * timetick;
124                     pair<double, double> location = make_pair(x_loc_later, y_loc_later);
125                     default_Particles[i].update_loc(location);
126                 }
127                 else
128                 {
129
130                     double f_x = external_force.first;
131                     double f_y = external_force.second;
132                     //가속도 구하기
133                     double a_x = f_x / mass[i];
134                     double a_y = f_y / mass[i];
135                     //나중속도 구하기
136                     double vel_x_later = x_vel[i] + a_x * timetick;
137                     double vel_y_later = y_vel[i] + a_y * timetick;
138                     //이동거리 구하기
139                     double s_x = (x_vel[i] + vel_x_later)*timetick / 2;
140                     double s_y = (y_vel[i] + vel_y_later)*timetick / 2;
141                     // 이동 후 위치 구하기
142                     double x_loc_later = x_loc[i] + s_x;
143                     double y_loc_later = y_loc[i] + s_y;
144                     //timetick이후의 정보 업데이트 해주기  particle에 추가 함수 필요
145                     pair<double, double> location = make_pair(x_loc_later, y_loc_later);
146                     pair<double, double> velocity = make_pair(vel_x_later, vel_y_later);
147                     default_Particles[i].update_loc(location);
148                     default_Particles[i].update_vel(velocity);
149                 }
150             }
151         }
```

다음으로는 중력이 꺼져 있을 경우인데, fix 안 되어 있을 때만 외력이 없다면 이전의 속도로 등속운동을 한다고 가정을 하여 구해준다. 외력이 있을 경우에는 그 외력이 합력이 되므로, 이를 이용해서 중력이 있을 때와 마찬가지로 이동거리와 나중 속도를 구한 후 그 정보들을 업데이트해준다.

2. commander 함수

command를 받아서 main함수로 명령어를 실행해주기 위한 함수이다.

command로 들어온 line을 모두 string으로 받아주며, 띄어쓰기 단위로 나눠준다. 그리고 나눠준 첫번째 원소를 보고 해당되는 명령어를 실행하여 준다. 이때 command 함수에서 오류처리도 해준다. 기본적으로는 정해진 input format이 아닌 형태로 들어왔을 때와, 파라미터들의 정보가 정해진 format이 아닐 때, 그리고 "₩n"으로 입력할 하거나 white space의 입력

이 들어왔을 때 오류 안내 ouptput을 처리해주도록 하였다.

결론

시뮬레이션을 만든 결과, 일단 github에 md파일로 있는 예시 코드는 모두 문제 없이 정상적으로 작동하였으며, 예시로 올라와 있는 지구의 태양 주위를 공전하는 운동도 어느 정도 잘 작동하는 것을 확인 할 수 있었다. 주어진 예시와 출력 값은 약간 달랐지만, 지구의 초기 위치는 (1.4960e+11, 0)이었고, 시뮬레이션 결과 (1.50209e+11, -4.35858e+09)에 위치하는 것을 확인 할 수 있었다. 약 1년을 공전한 결과이기 때문에 거의 처음의 위치와 같은 위치로 갈 것으로 예측하였지만, x좌표의 위치는 어느 정도 상당히 비슷하지만 y축의 위치는 꽤 많이 차이가 난 것을 확인 할 수 있었다. 이에 대한 이유를 생각해보면, 시뮬레이션을 해준 duration이 약 1년이긴 하지만, 정확히 한 바퀴를 도는데 걸리는 시간이 아니었기 때문에, 결과만을 보면 원래 위치로 돌아오고 어느 정도 더 진행한 상태라고 할 수 있을 것이다.

다음으로는 추가적인 시뮬레이션을 진행해보았다. 태양 정도의 질량의 두개의 물체가 같은 축상에서 서로를 향해 운동을 한다면 어떻게 될지 시뮬레이션을 해봤다. 이 시뮬레이션을 통해서는 계산부분에서 지적했던 두 가지의 한계에 대해서 어떻게 구현되는지 확인 할 수 있을 것으로 예측된다. 이에 대해서 어떻게 시뮬레이션을 돌렸는지 설명하면, ap 0 1.9890e+30 0. 0. 2.9800e+04 0. 과 ap 1 1.9890e+30 1.0000e+10 0. -3.0000e+06 0. 의 두 명령어로 태양 질량의 입자가 서로를 향해 x축 위에서 운동을 하도록 해주었고, timetick은 3600초, rv 300000 명령어로 시뮬레이션을 돌려주었다. 그 결과 입자 두개가 x축 위에 -10^(11)정도의 위치에서 붙어서 거의 일정한 위치에서의 운동을 하는 것을 확인 할 수 있었다. 이를 통해서 확인 할 수 있는 점은, 일단 충돌의 경우는 두 입자가 충돌은 하지 못하므로, 초기 속도가 더 빨랐던 1입자의 운동방향으로 같이 이동해서 평행점을 찾은 것을 알 수 있고, 두번째로 두 입자가 매우 가까워지는 시기가 분명 있고, 이때 double 형 인수의 overflow가 일어날 수도 있는 상황이지만, 두 입자사이의 힘의 크기를 구할 때 이에 대한 예외처리를 해주었기 때문에 결과적으로 중력이 매우 큰 두 입자가 어느 정도 가까운 거리에서 마치 거의 하나의 입자가 된 것처럼 행동을 하는 것을 확인 할 수 있었다. ~~우리는 이 프로젝트를 12월 16일 새벽부터 시작하여 시간내에 끝냈습니다. 이게 다른 조보다 가장 훌륭한 점이라고 생각됩니다.~~