3D SPH

PA#3

Term Project

2015410039 이현건

기본 구현 사항

1. 3D SPH

```
vec3    position3;
vec3    velocity3;
vec3    acceleration3;
vec3    fpressure3;
vec3    fviscosity3;
```

```
void integrate3(double dt, vec3 gravity)
{
    vec3 fgrav = gravity * mass;
    double boxsize = 10.0f;

    // Update velocity and position
    acceleration3 = (fpressure3 + fviscosity3) / density + fgrav;
    velocity3 = velocity3 + acceleration3 * dt;
    position3 = position3 + velocity3 * dt;

    // Boundary condition
    if (position3.x < -boxsize && velocity3.x < 0.0)
    {
        velocity3.x *= -restitution;
        position3.x = -boxsize + 0.1;
    }
    if (position3.x > boxsize && velocity3.x > 0.0)
    {
        velocity3.x *= -restitution;
        position3.x = boxsize - 0.1;
    }
    if (position3.y < -boxsize + 8.0f && velocity3.y < 0.0)
    {
        velocity3.y *= -restitution;
        position3.y = -boxsize + 8.0f + 0.1;
    }
    if (position3.y > boxsize + 8.0f && velocity3.y > 0.0)
    {
        velocity3.y *= -restitution;
        position3.y = boxsize + 8.0f - 0.1;
    }
    if (position3.z < -boxsize && velocity3.z < 0.0)
    {
        velocity3.z *= -restitution;
        position3.z = -boxsize + 0.1;
    }
    if (position3.z > boxsize && velocity3.z > 0.0)
    {
        velocity3.z *= -restitution;
        position3.z = boxsize - 0.1;
    }
}
```

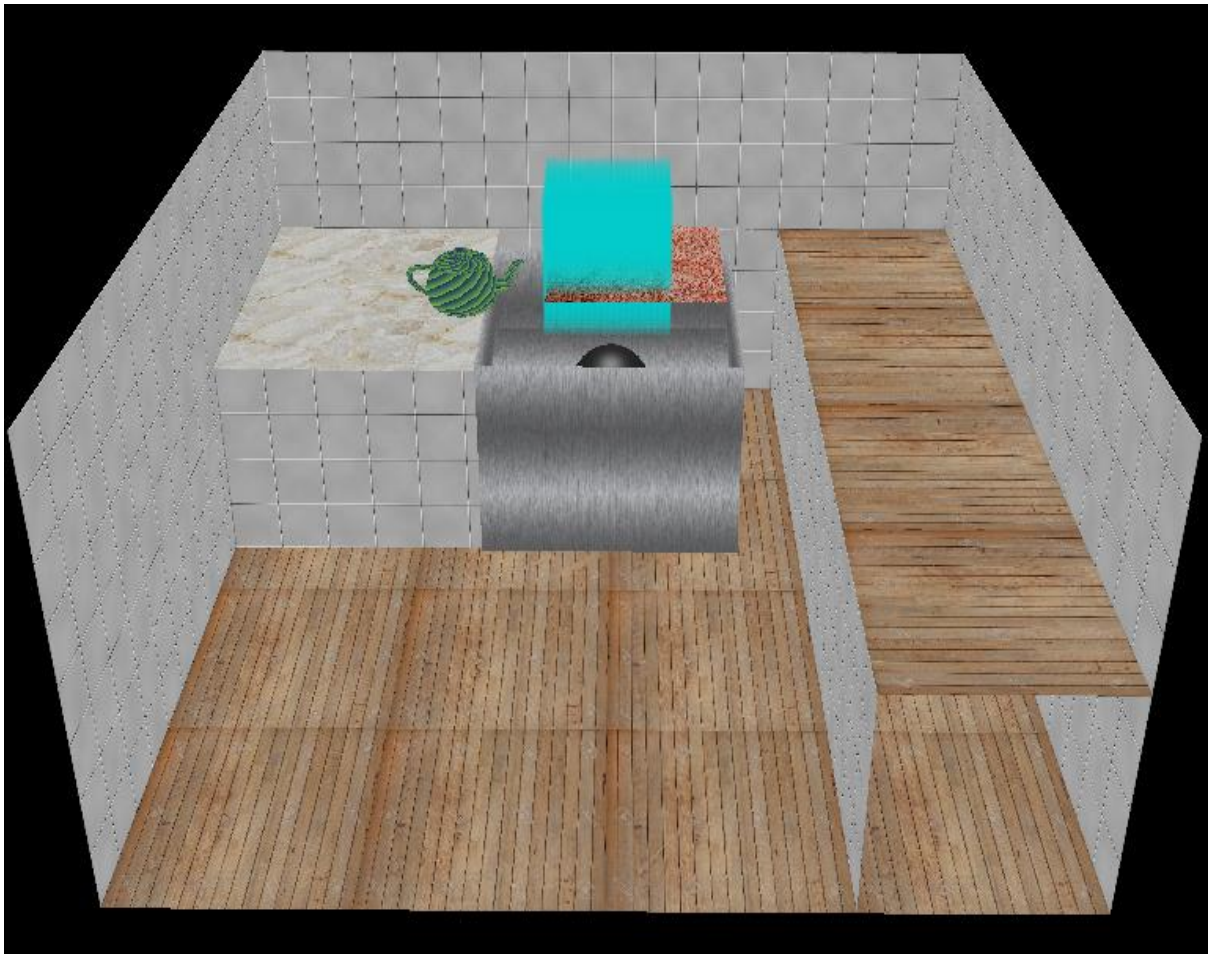
스켈레톤으로 주어진 2D SPH를 기반으로 vec2값들을 vec3로 변경하는 등 z축을 반영해 3D SPH로 변경하였다. 이에 맞춰 particle에 적용되는 함수들(integrate, hashtable, computedensity등) 또한 vec3를 사용하도록 수정하였다.

Kernal Derivation들 또한 2D에서 3D로 변환하려 하였는데, 기존 스켈레톤에 존재하던 Kernal이 3D 커널의 수식에서 vec2만 사용하는 식으로 동작하고 있었다. 때문에 커널의 계산식은 유지한 채 vec2만 vec3으로 사용하도록 수정하였다.

또한 2D공간이 3D공간으로 확장됨에 따라 연산이 제곱에서 세제곱으로 크게 증가해, 기존 40*40의 크기를 가졌던 world 공간과 hash table을 20*20*20 크기로 감소시켰다.

2. Rendering

(marching cube 적용 전)



2-1. Lighting

```
glShadeModel(GL_SMOOTH);

glEnable(GL_LIGHTING);
glEnable(GL_LIGHT0);
glEnable(GL_COLOR_MATERIAL);

float light_pos[] = { 0.0, 1500.0, 0.0, 1.0 };
float light_dir[] = { 0, -1, 0 , 0.0 };
float light_ambient[] = { 0.6, 0.6, 0.6, 1.0 };
float light_diffuse[] = { 0.8, 0.8, 0.8, 1.0 };
float light_specular[] = { 0.4, 0.4, 0.4, 1.0 };
float frontColor[] = { 0.8, 0.8, 0.8, 1 };

float matShininess = 20;
float noMat[] = { 0, 0, 0, 1 };

float matSpec[] = { 1.0, 1.0, 1.0, 1.0 };
glMaterialfv(GL_FRONT, GL_EMISSION, noMat);
glMaterialfv(GL_FRONT, GL_SPECULAR, matSpec);
glMaterialfv(GL_FRONT, GL_AMBIENT, frontColor);
glMaterialfv(GL_FRONT, GL_DIFFUSE, frontColor);
glMaterialf(GL_FRONT, GL_SHININESS, matShininess);

glLightfv(GL_LIGHT0, GL_AMBIENT, light_ambient);
glLightfv(GL_LIGHT0, GL_DIFFUSE, light_diffuse);
glLightf(GL_LIGHT0, GL_SPOT_CUTOFF, 80.0f);
glLightf(GL_LIGHT0, GL_SPOT_EXPONENT, 80.0f);
glLightfv(GL_LIGHT0, GL_POSITION, light_pos);
glLightfv(GL_LIGHT0, GL_SPOT_DIRECTION, light_dir);
glLightfv(GL_LIGHT0, GL_SPECULAR, light_specular);
```

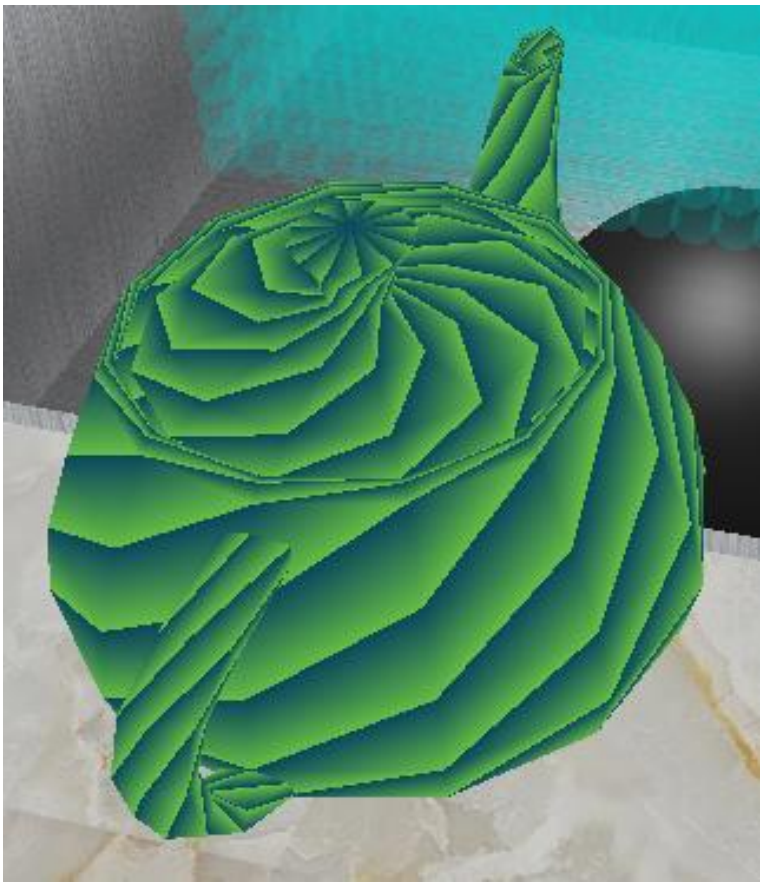
물 파티클들을 하나하나 그릴 경우에는, particle들의 색깔이 카메라의 위치에 따라 변하는 현상이 발생하였다. 원인 파악 결과 particle에 정반사된 빛이 카메라로 들어오는 경우, specular 색이 반영되게 되어 색이 변하게 된다. 일반적으로 물은 빛을 받았을 때 대부분이 반사하기보단 통과하므로, specular값을 기존 1.0에서 낮추어 조금 더 자연스럽게 보이게 하였다.

하지만 marching cube기법을 사용한 이후에는 위와 같은 현상이 사라졌기 때문에, 다시 1.0으로 설정하였다.

또한 SPH시뮬레이션 공간을 집 내부처럼 보이도록 설정했기 때문에, 빛 또한 집 내부의 빛처럼 보이도록 백색광 위주로 -Y축 정방향으로 비추도록 설정하였다.

2-2. Background Texture

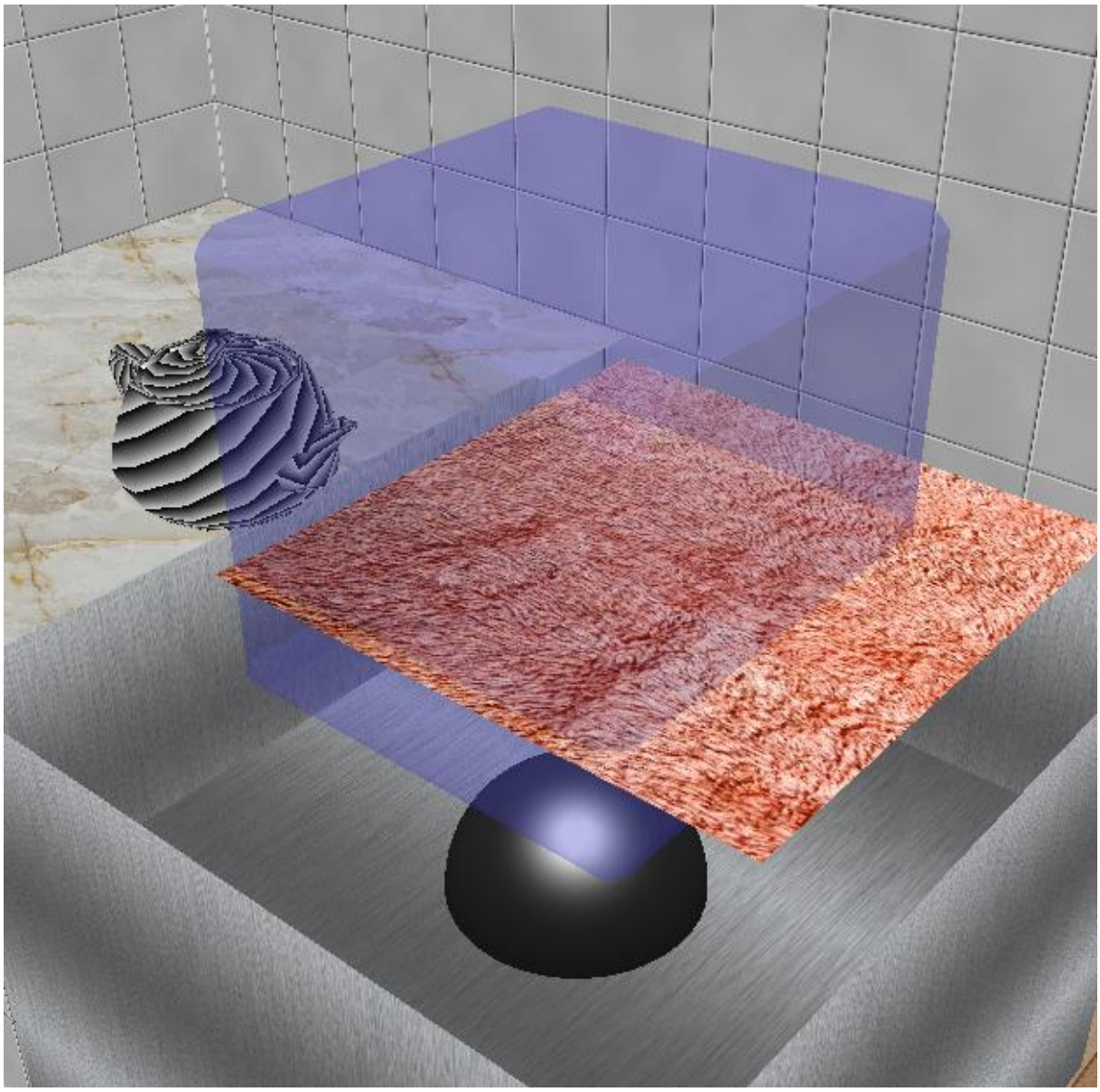
단순히 바닥, 벽에 텍스처만 넣으니 부족한 느낌이 들어서, 여러가지 텍스처를 사용해 부엌 같은 형태로 만들었다. 물을 담는 공간도 싱크대와 같은 느낌이 들도록 금속질의 텍스처를 사용하였다.



Utah teapot에도 텍스처를 적용하고 싶었지만, 주전자의 좌표 각각에 tex_coord를 자연스럽게 매칭하는 방법을 찾지 못하여 적용하지 못했고, 대신 각 좌표들에 color값을 설정할 때 패턴을 줘 일정한 무늬를 이루도록 하였다.

추가 구현 사항

1. Advanced Rendering



Marching Cube 기법을 사용해 particle들을 렌더링하였다.

```
void SPH::marchingCube() {
    for (int x = 0; x < GRIDSIZE; x++) {
        for (int y = 0; y < GRIDSIZE; y++) {
            for (int z = 0; z < GRIDSIZE; z++) {
                int value = 0;

                vector<Particle*> ris;
                vector<Particle*> rjs = getNeighbor3(x, y, z, h, ris);

                for (int i = 0; i < ris.size(); i++)
                {
                    Particle *pi = ris[i];
                    if (pi->water == false) continue;

                    int rx = (int)round(pi->position3.x - (x-10));
                    int ry = (int)round(pi->position3.y - (y-2));
                    int rz = (int)round(pi->position3.z - (z-10));
                    int sum = rx * 100 + ry * 10 + rz;

                    switch (sum)
                    {
                    case(1):
                        value |= 1;
                        break;
                    case(101):
                        value |= 2;
                        break;
                    case(100):
                        value |= 4;
                        break;
                    case(0):
                        value |= 8;
                        break;
                    case(11):
                        value |= 16;
                        break;
                    case(111):
                        value |= 32;
                        break;
                    case(110):
                        value |= 64;
                        break;
                    case(010):
                        value |= 128;
                        break;
                    default:
                        break;
                    }
```

각 Cube의 범위는 hash grid를 나눈 범위와 동일하게 사용하였고, 각 cube에 대해 hash grid함수를 통해 주변 파티클 정보를 받아온 후, 파티클들의 position을 반올림해 나온 x,y,z 정수 좌표가 Cube의 정점의 좌표와 일치하면 해당 정점을 그려야 할 정점으로 판단하였다. Cube의 총 8개 정점에 대해 그릴지, 말지 여부는 value 변수에 8bit 2진수 형식으로 0부터 255사이의 값으로 저장하게 된다.

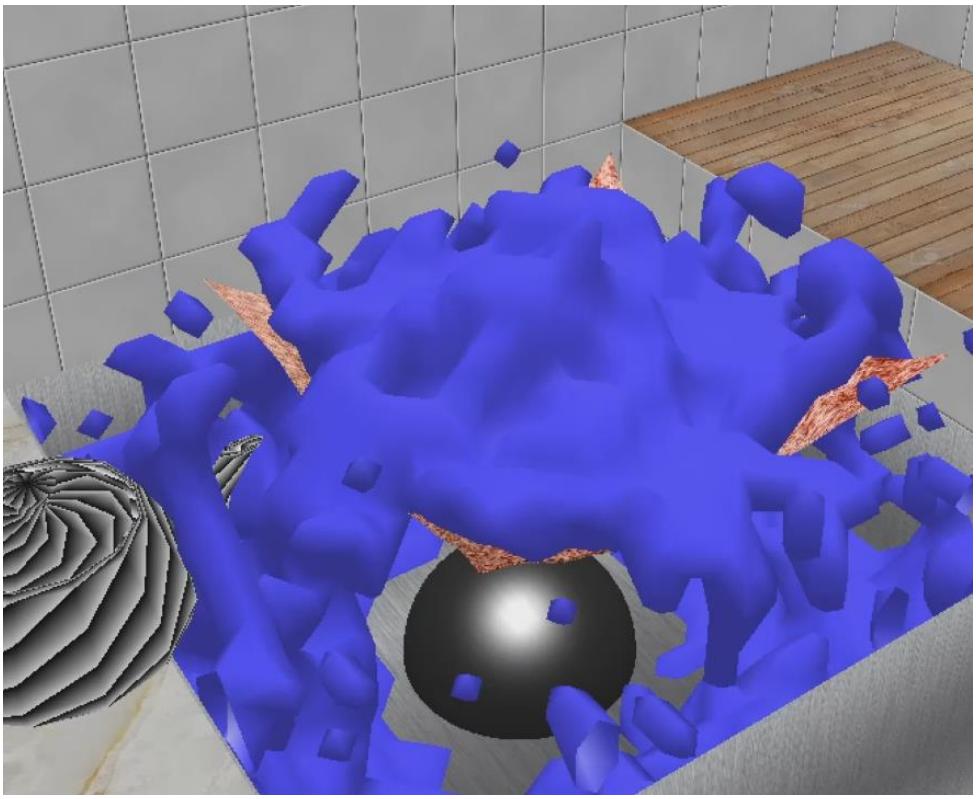
```
int triTable[256][16] =
{ {-1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
{0, 8, 3, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
{0, 1, 9, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
{1, 8, 3, 9, 8, 1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
{1, 2, 10, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
{0, 8, 3, 1, 2, 10, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
{9, 2, 10, 0, 2, 9, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
{2, 8, 3, 2, 10, 8, 10, 9, 8, -1, -1, -1, -1, -1, -1, -1},
{3, 11, 2, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
{0, 11, 2, 8, 11, 0, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
{1, 9, 0, 2, 3, 11, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
{1, 11, 2, 1, 9, 11, 9, 8, 11, -1, -1, -1, -1, -1, -1, -1},
{3, 10, 1, 11, 10, 3, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
{0, 10, 1, 0, 8, 10, 8, 11, 10, -1, -1, -1, -1, -1, -1, -1},
{3, 9, 0, 3, 11, 9, 11, 10, 9, -1, -1, -1, -1, -1, -1, -1},
{9, 8, 10, 10, 8, 11, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
{4, 7, 8, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
{4, 3, 0, 7, 3, 4, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
{0, 1, 9, 8, 4, 7, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
{4, 1, 9, 4, 7, 1, 7, 3, 1, -1, -1, -1, -1, -1, -1, -1},
{1, 2, 10, 8, 4, 7, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
{3, 4, 7, 3, 0, 4, 1, 2, 10, -1, -1, -1, -1, -1, -1, -1},
{9, 2, 10, 9, 0, 2, 8, 4, 7, -1, -1, -1, -1, -1, -1, -1},
{2, 10, 9, 2, 9, 7, 2, 7, 3, 7, 9, 4, -1, -1, -1, -1},
{8, 4, 7, 3, 11, 2, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
{11, 4, 7, 11, 2, 4, 2, 0, 4, -1, -1, -1, -1, -1, -1, -1},
{9, 0, 1, 8, 4, 7, 2, 3, 11, -1, -1, -1, -1, -1, -1, -1},
{4, 7, 11, 9, 4, 11, 9, 11, 2, 9, 2, 1, -1, -1, -1, -1},
{3, 10, 1, 3, 11, 10, 7, 8, 4, -1, -1, -1, -1, -1, -1, -1},
{1, 11, 10, 1, 4, 11, 1, 0, 4, 7, 11, 4, -1, -1, -1, -1},
{4, 7, 8, 9, 0, 11, 9, 11, 10, 11, 0, 3, -1, -1, -1, -1},
{4, 7, 11, 4, 11, 9, 9, 11, 10, -1, -1, -1, -1, -1, -1, -1},
{9, 5, 4, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
{9, 5, 4, 0, 8, 3, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
{0, 5, 4, 1, 5, 0, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
{8, 5, 4, 8, 3, 5, 3, 1, 5, -1, -1, -1, -1, -1, -1, -1},
{1, 2, 10, 9, 5, 4, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
{3, 0, 8, 1, 2, 10, 4, 9, 5, -1, -1, -1, -1, -1, -1, -1},
```

특정 정점들을 그려야 하는 큐브가 어떻게 그려야 하는지에 대한 정보는 위 triTable 배열에서 가지고 있다. triTable 배열은 256*16의 크기를 가지며, cube가 그려질 수 있는 총 256가지의 경우에 대해 어떻게 그려야 하는지 정보를 가지고 있다. 내부의 숫자값들은 0부터 11까지 Cube의 12개의 변들을 의미하고, 3개씩 묶어서 하나의 triangle을 그리게 된다. 12개 변의 중심 좌표는 쉽게 구할 수 있으므로 value값을 triTable에 제공해주면 해당 Cube를 어떻게 그려야 하는지 알 수 있게 된다.

triTable 배열은 http://paulbourke.net/geometry/polygonise/ 페이지에서 찾아 사용하였다.

```
switch (triTable[marchingData[(int)x][(int)y][(int)z]][t + i])
{
case(0):
    marchingNormal[(int)x][(int)y][(int)z][0] += finalnormal;
    if (z+1<GRIDSIZE)marchingNormal[(int)x][(int)y][(int)z +1][2] += finalnormal;
    if (y > 0 && z + 1 < GRIDSIZE)marchingNormal[(int)x][(int)y-1][(int)z +1][6] += finalnormal;
    if (y > 0)marchingNormal[(int)x][(int)y-1][(int)z][4] += finalnormal;
    break;
case(1):
    marchingNormal[(int)x][(int)y][(int)z][1] += finalnormal;
    if (x + 1 < GRIDSIZE)marchingNormal[(int)x+1][(int)y][(int)z][3] += finalnormal;
    if (y > 0 && x + 1 < GRIDSIZE)marchingNormal[(int)x+1][(int)y - 1][(int)z][7] += finalnormal;
    if (y > 0)marchingNormal[(int)x][(int)y - 1][(int)z][5] += finalnormal;
    break;
case(2):
    marchingNormal[(int)x][(int)y][(int)z][2] += finalnormal;
    if (z > 0)marchingNormal[(int)x][(int)y][(int)z - 1][0] += finalnormal;
    if (y > 0 && z > 0)marchingNormal[(int)x][(int)y - 1][(int)z-1][4] += finalnormal;
    if (y > 0)marchingNormal[(int)x][(int)y - 1][(int)z][6] += finalnormal;
    break;
case(3):
    marchingNormal[(int)x][(int)y][(int)z][3] += finalnormal;
    if (x>0)marchingNormal[(int)x - 1][(int)y][(int)z][1] += finalnormal;
    if (y > 0 && x > 0)marchingNormal[(int)x - 1][(int)y - 1][(int)z][5] += finalnormal;
    if (y > 0)marchingNormal[(int)x][(int)y - 1][(int)z][7] += finalnormal;
    break;
```

또한, normal을 계산할 때 face를 그릴 때 같이 하게되면, 각 face들 사이에 normal이 크게 변화하여 각진 도형 느낌으로 그려지게 된다. 이를 완화하기 위해 face를 그리기 전에 미리 face의 좌표를 계산해서 normal을 구하고, 이를 face를 이루는 변에 저장하는 작업을 모든 cube에 대해 진행하여 각자의 normal값들이 서로 보간되도록 하였다.



다음과 같이 각 cube들 사이에서 급격히 그 모양이 변하는 일 없이 연속적으로 그려지는 것을 확인할 수 있다.

```
for (int t = 0; t < 16; t+=3) {
    if (triTable[marchingData[(int)x][(int)y][(int)z]][t] == -1)continue;

    vec3 trianglepos1 = vec3(x,y,z) + marchingPosition[triTable[marchingData[(int)x][(int)y][(int)z]][t]];
    vec3 trianglepos2 = vec3(x, y, z) + marchingPosition[triTable[marchingData[(int)x][(int)y][(int)z]][t+1]];
    vec3 trianglepos3 = vec3(x, y, z) + marchingPosition[triTable[marchingData[(int)x][(int)y][(int)z]][t+2]];

    vec3 normal1 = trianglepos1 - trianglepos2;
    vec3 normal2 = trianglepos1 - trianglepos3;
    vec3 finalnormal = normal1.Cross(normal2);
    finalnormal.Normalize();
    //glNormal3f(finalnormal.x, finalnormal.y, finalnormal.z);
    glNormal3f(marchingNormal[(int)x][(int)y][(int)z][triTable[marchingData[(int)x][(int)y][(int)z]][t]].x, marchi
    glVertex3f(trianglepos1.x, trianglepos1.y, trianglepos1.z);
    glNormal3f(marchingNormal[(int)x][(int)y][(int)z][triTable[marchingData[(int)x][(int)y][(int)z]][t+1]].x, marc
    glVertex3f(trianglepos2.x, trianglepos2.y, trianglepos2.z);
    glNormal3f(marchingNormal[(int)x][(int)y][(int)z][triTable[marchingData[(int)x][(int)y][(int)z]][t+2]].x, marc
    glVertex3f(trianglepos3.x, trianglepos3.y, trianglepos3.z);


}
glEnd();
```

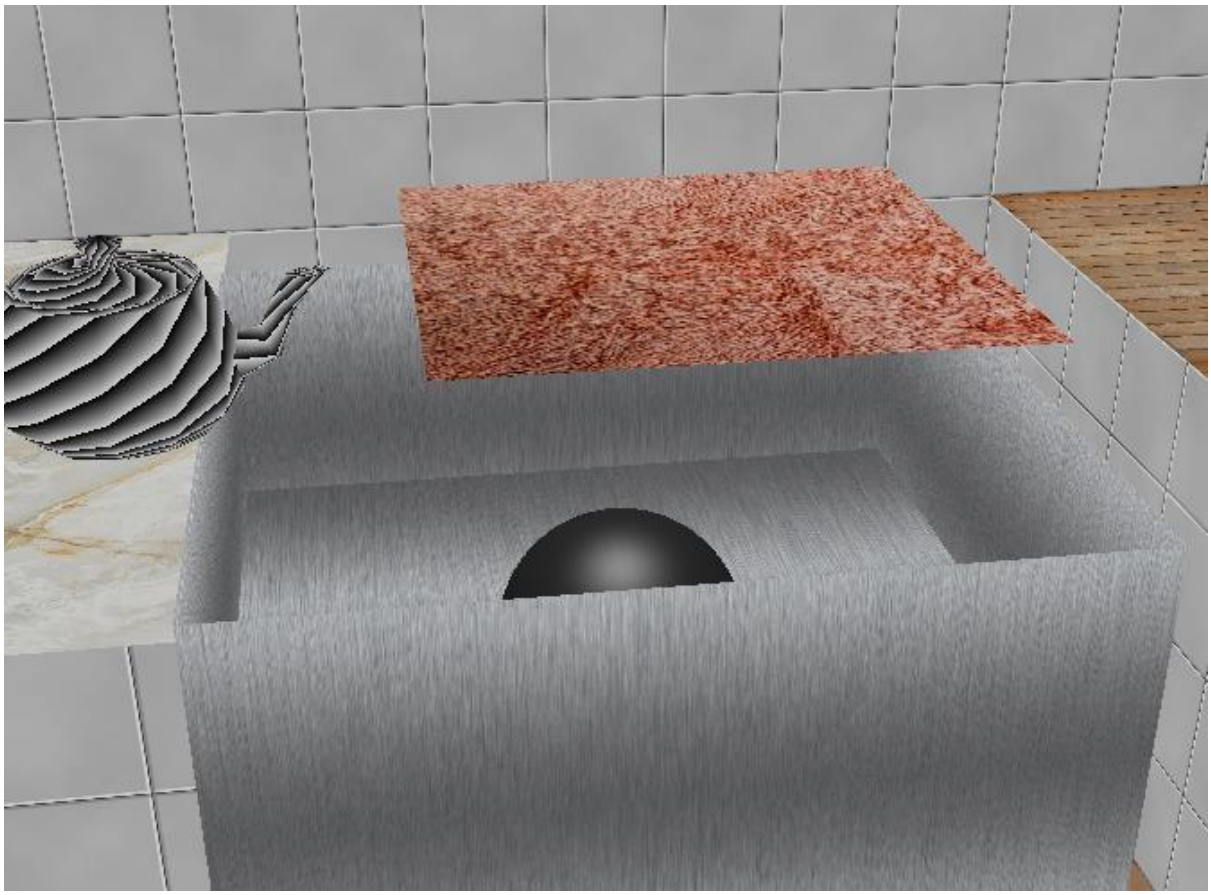
위와 같이 marchingData의 정점 데이터를 triTable에 제공해 삼각형의 좌표를 구하고, 해당 좌표들을 Vertex정보로 가공해 위에서 구한 normal값과 함께 opengl로 그려주면 정상적으로 Marching Cube에 의해 particle들을 렌더링 할 수 있다.

2. Advanced Data Structure

```
void SPH::makeHashTable3()
{
    for (int p = 0; p < GRIDSIZE; p++)
    {
        for (int q = 0; q < GRIDSIZE; q++)
        {
            for (int r = 0; r < GRIDSIZE; r++) {
                hashGrid3[p][q][r].clear();
                hashGrid3_cloth[p][q][r].clear();
            }
        }
    }

    for (int i = 0; i < particles.size(); i++)
    {
        Particle *p = particles[i];

        double x = (p->position3.x + GRIDSIZE / 2);
        double y = (p->position3.y + GRIDSIZE / 2);
        double z = (p->position3.z + GRIDSIZE / 2);
        int gridx = (int)(x);
        int gridy = (int)(y-8);
        int gridz = (int)(z);

        if (gridx < 0) gridx = 0;
        if (gridx > GRIDSIZE - 1) gridx = GRIDSIZE - 1;
        if (gridy < 0) gridy = 0;
        if (gridy > GRIDSIZE - 1) gridy = GRIDSIZE - 1;
        if (gridz < 0) gridz = 0;
        if (gridz > GRIDSIZE - 1) gridz = GRIDSIZE - 1;

        hashGrid3[gridx][gridy][gridz].push_back(p);


    }
```

스켈레톤 코드에서 제공된 2차원의 hash grid를 3차원으로 바꾼 후, GRIDSIZE를 줄여 사용하였다.

3. Coupling with 3D Objects



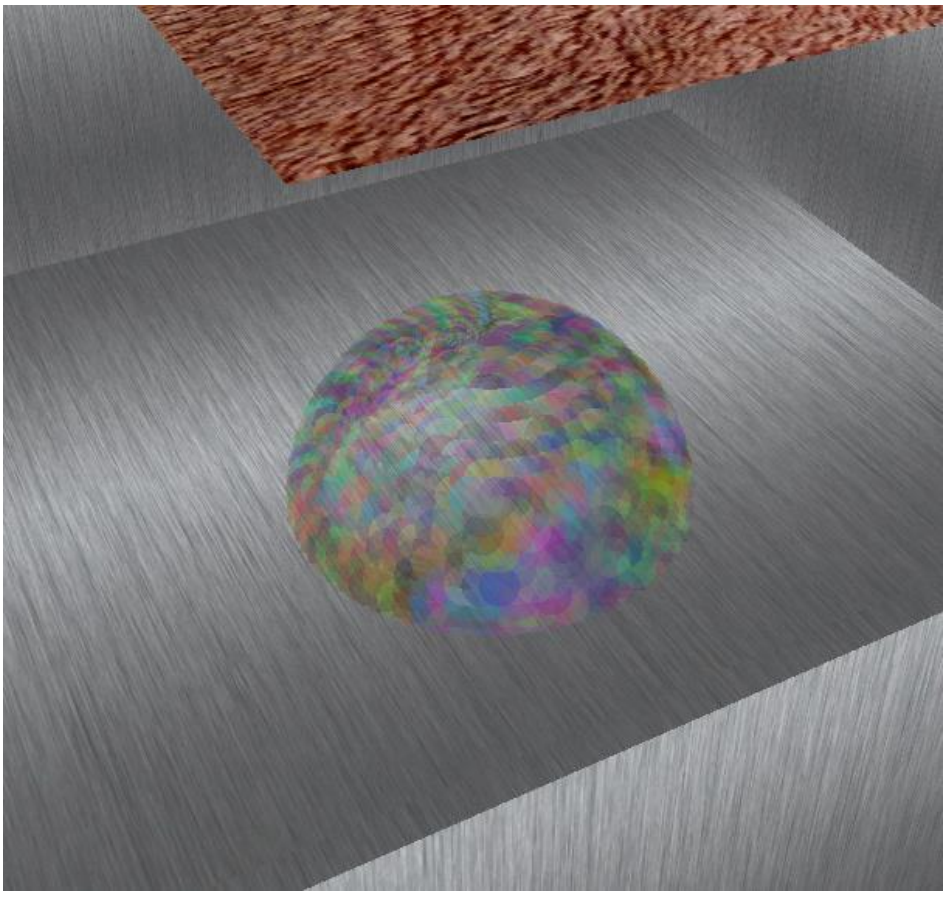
Mass-spring, Sphere, Teapot 총 3가지의 object와 particle의 상호작용을 구현하였다.

3가지의 object들 모두 공통으로 파티클의 water 변수가 false로 설정되어 density계산을 하지 않고 고정 density값을 가지며, force 계산 또한 물 파티클들에 대해서만 계산하도록 되어있다.

Sphere와 Teapot은 파티클에서 계산된 force값이 object에 적용되지 않지만, Mass-spring의 경우 파티클에서 계산된 force가 cloth의 integrate 단계에서 적용되게 된다.

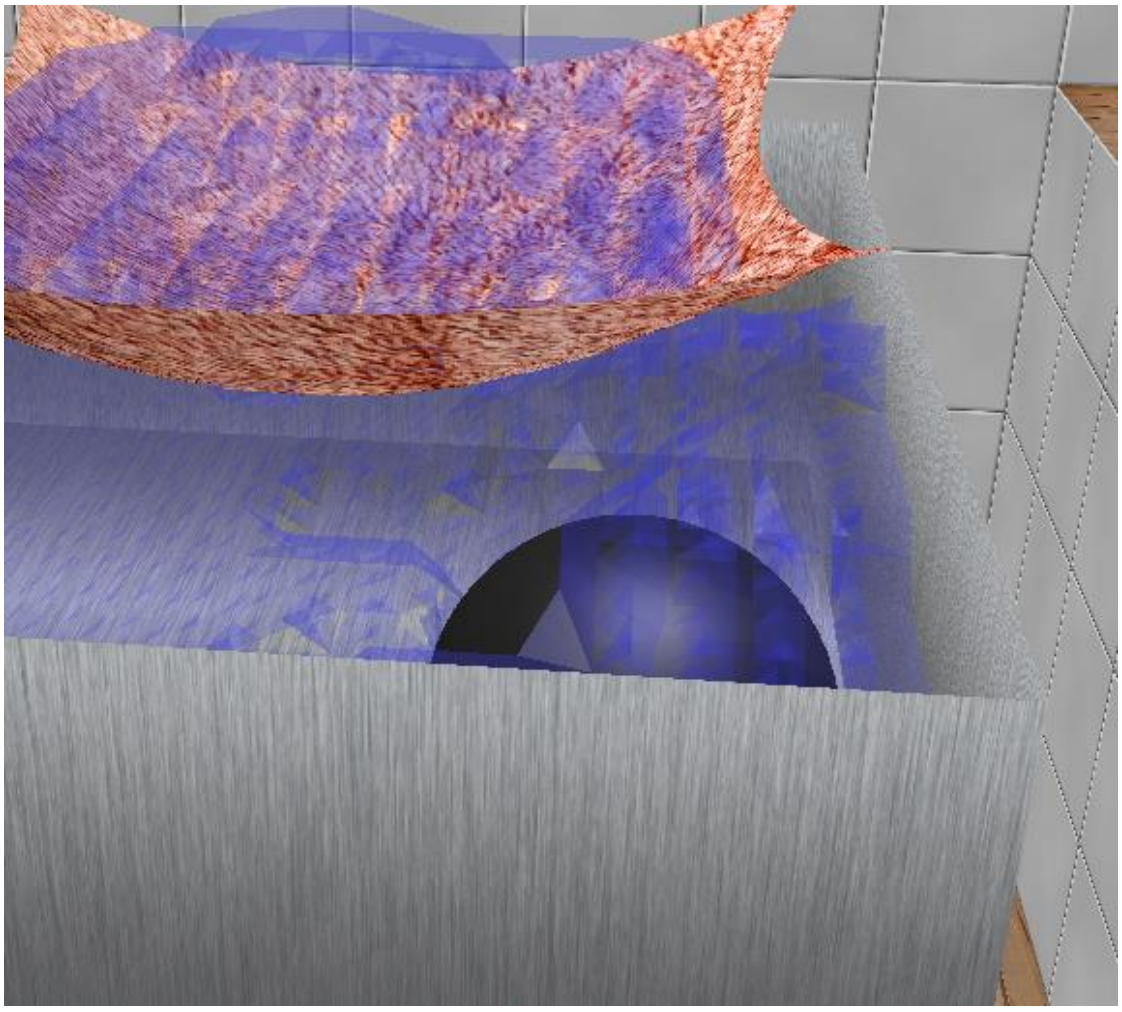
```cpp
void Simulator::SphereParticle() {
    spherePos = vec3(0.0f, -2.0f, 0.0f);
    sphereRad = 3.0f;
    int i = 0;
    for (float x = 0.0; x < 90; x += 5.0) {
        float tempRad = sphereRad * glm::cos(glm::radians(x));
        float tempY = sphereRad * glm::sin(glm::radians(x));

        for (float y = 0.0; y <= 360; y += 5.0) {

            glm::vec3 rot = glm::vec3(tempRad * glm::cos(glm::radians(y)), 0.0f, tempRad * glm::sin(glm::radians(y)));

            //glVertex3f(rot.x, rot.y, rot.z);
            //Particle* p = new Particle(x,y,z, i++);
            rot.x += spherePos.x;
            rot.z += spherePos.z;
            rot.y += (tempY + spherePos.y);

            Particle* p = new Particle(rot.x, rot.y, rot.z, i++);
            p->water = false;
            p->density = 15.0;
            mySPH->sphere_particles.push_back(p);
            //mySPH->particles.push_back(p);
        }
    }
```

먼저 Sphere의 경우, 구의 중심 좌표, 반지름을 바탕으로 조금씩 회전시켜가면서 표면에 파티클을 생성하였다. 또한 표면에만 파티클을 생성했을 경우 물 파티클이 빠르게 이동할 경우 반발력이 충분히 작용하지 못해 Sphere내부로 들어가버리는 경우가 있어, 구의 반지름보다 조금 작게 파티클을 한번 더 생성해서 위의 경우를 방지하였다.

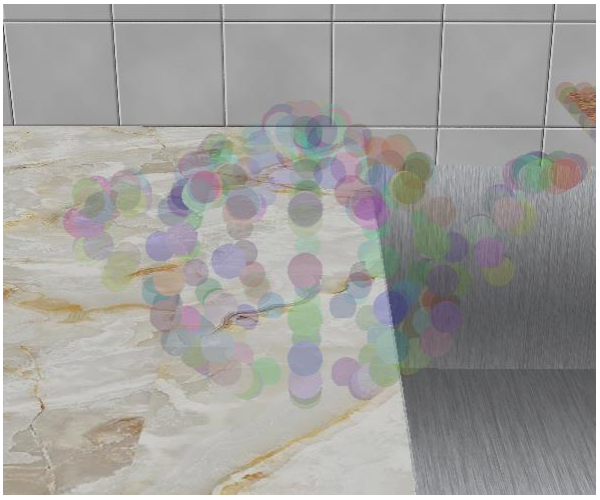


실제로 파티클이 구 모양으로 생성된 것을 확인할 수 있으며, 구의 중심 좌표에 glutSolidSphere함수로 구를 렌더링하였다.

또한 wasd 방향키를 이용해서 구의 이동을 구현하였다. wasd입력에 따라 구 중심 좌표의 x,z값이 업데이트되며, 매 프레임 변화하는 중심 좌표에 따라 구를 구성하는 파티클들 또한 위치가 업데이트되게 된다.



위 사진에서 구가 오른쪽으로 움직임에 따라 오른쪽의 물 파티클이 밀려 올라간 것을 볼 수 있다.



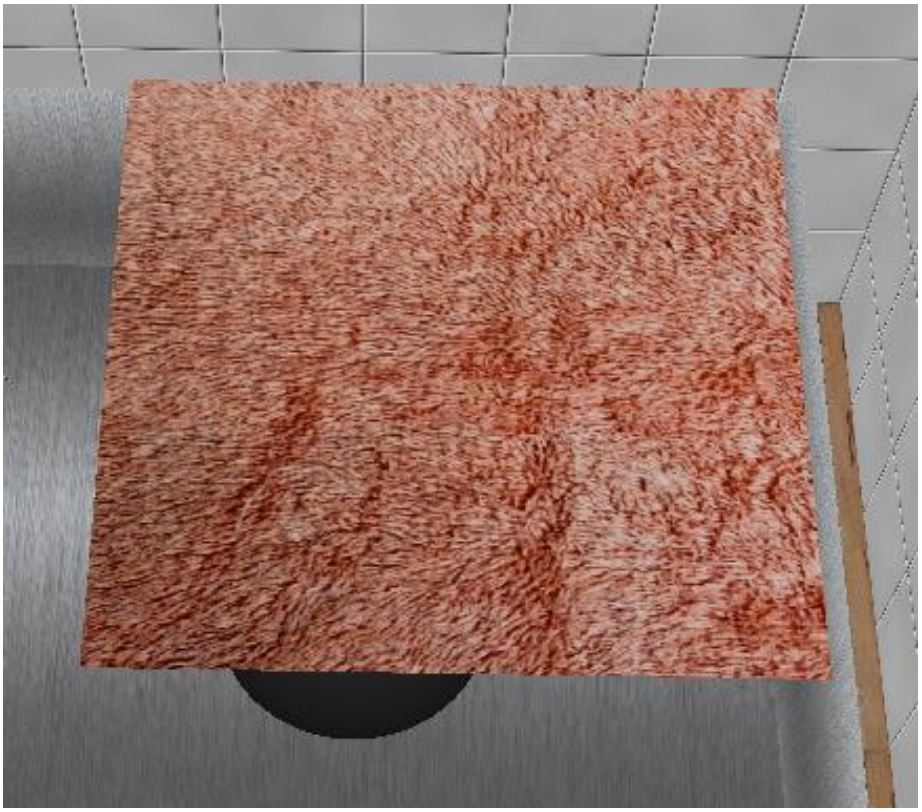
두번째론 teapot을 구현하였다.

```
#define teapot_count 5184

const float teapot[teapot_count] = {
  0.700000f, -1.200000f, 0.000000f,
  0.605600f, -1.200000f, -0.355700f,
  0.598800f, -1.243700f, -0.351700f,
  0.598800f, -1.243700f, -0.351700f,
  0.692100f, -1.243700f, 0.000000f,
  0.700000f, -1.200000f, 0.000000f,
  0.692100f, -1.243700f, 0.000000f,
  0.598800f, -1.243700f, -0.351700f,
  0.619600f, -1.243700f, -0.363900f,
  0.619600f, -1.243700f, -0.363900f,
  0.716200f, -1.243700f, 0.000000f,
  0.692100f, -1.243700f, 0.000000f,
  0.716200f, -1.243700f, 0.000000f,
  0.619600f, -1.243700f, -0.363900f,
  0.648900f, -1.200000f, -0.381100f,
  0.648900f, -1.200000f, -0.381100f,
  0.750000f, -1.200000f, 0.000000f,
  0.716200f, -1.243700f, 0.000000f,
  0.605600f, -1.200000f, -0.355700f,
```

```
glBegin(GL_TRIANGLES);
vec3 initPos = vec3(-12.0f, 4.8f, 0.0f);
for (int i = 0; i < teapot_count; i += 3) {
    if (i % 2 ==0) {
        glColor4f(1.0f, 1.0f, 1.0f, 1.0f);
    }
    else {
        glColor4f(0.0f, 0.0f, 0.0f, 1.0f);
    }

    glm::vec3 rot = glm::rotateX(glm::vec3(teapot[i], teapot[i + 1], teapot[i + 2]), glm::radians(180.0f));
    glVertex3f(3 * rot.x + initPos.x, 3 * rot.y + initPos.y, 3 * rot.z + initPos.z);
}
glEnd();
```

Teapot을 구성하는 좌표 정보를 github등에서 찾아 사용했으며, 해당 좌표를 바탕으로 렌더링하고 파티클을 생성하였다.

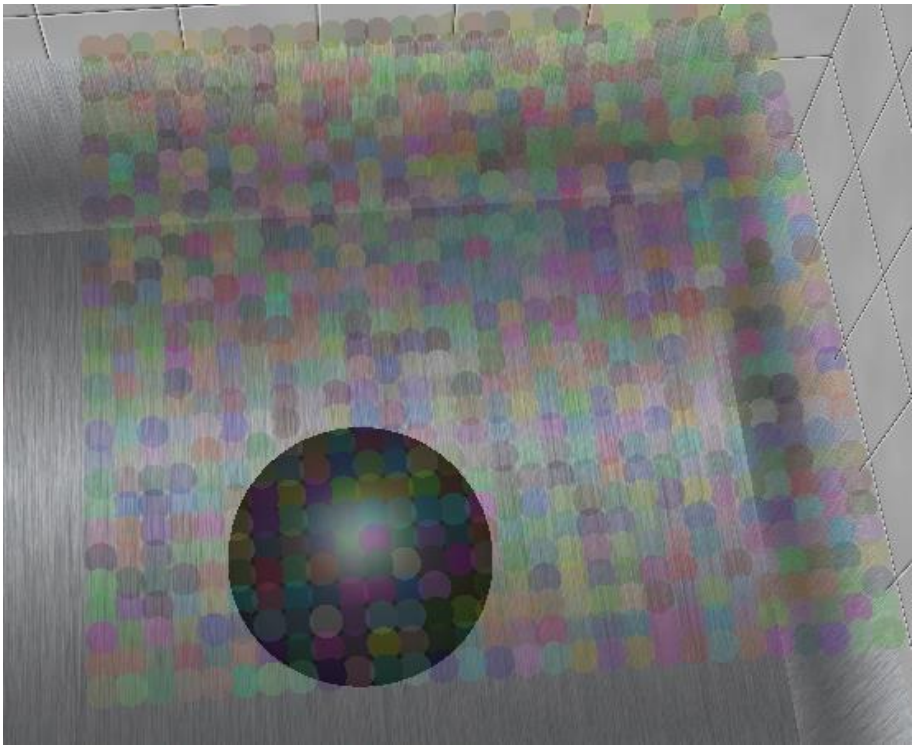
Sphere와는 달리 이동하지 않기 때문에 좌표는 고정되어 있으며, teapot을 구성하는 vertex가 굉장히 많아 모든 vertex마다 particle을 생성하는 것이 아니라 vertex 3개마다 하나꼴로 파티클을 생성하였다.



마지막으로 PA#2에서 만들었던 mass-spring 모델을 가져와서 물 파티클과 상호작용하도록 구현하였다.

PA#2와 동일하게 F를 누른 후 마우스 이동을 통해 천에 힘을 가할 수 있으며, 이를 통해 천에 담겨있던 물을 위로 튕겨 올리는 등의 동작 또한 가능하다.

단 위의 Sphere와는 다르게 파티클이 한 층으로 이루어져 있어 물이나 천의 파티클이 빠르게 이동하게 되면 서로 통과하게 된다.



각 node들마다 파티클을 생성하면 사이사이의 빈 공간이 생겨서 물 파티클들이 다 빠져나가게 된다. 때문에 structural spring마다 각 하나씩 파티클을 생성하고, shear spring 2개당 하나꼴로 파티클을 생성해서 파티클이 촘촘하게 배치되어 상호작용하도록 하였다.

또한 cloth의 이동에 따라 매 프레임 파티클들의 위치 또한 업데이트하도록 하였다. Node의 파티클들은 node의 좌표를 따르고, spring의 파티클들은 spring의 두 노드 p1, p2좌표의 중간값으로 업데이트된다.

```
void Simulator::compute_cloth_particle() {
    //particle on each node
    for (int i = 0; i < cloth->nodes.size(); i++) {
        cloth->nodes[i]->particle_accel +=
        ((mySPH->cloth_particles[i]->fpressure3 + mySPH->cloth_particles[i]->fviscosity3) / mySPH->cloth_particles[i]->density);
    }
    //particle on structual spring(1/2 to each node)
    int str_spring_num = cloth->size_x * (cloth->size_y - 1) + cloth->size_y * (cloth->size_x - 1);
    for (int i = cloth->nodes.size(); i < cloth->nodes.size() + str_spring_num; i++) {
        int i1 = i - cloth->nodes.size();
        cloth->spring[i1]->p1->particle_accel +=
        ((mySPH->cloth_particles[i]->fpressure3 + mySPH->cloth_particles[i]->fviscosity3) / (2.0f * mySPH->cloth_particles[i]->density));
        cloth->spring[i1]->p2->particle_accel +=
        ((mySPH->cloth_particles[i]->fpressure3 + mySPH->cloth_particles[i]->fviscosity3) / (2.0f * mySPH->cloth_particles[i]->density));

    }

    for (int i = 0; i < (cloth->size_x - 1) * (cloth->size_y - 1); i++) {
        int i1 = cloth->nodes.size() + str_spring_num + i;
        int i2 = str_spring_num + i * 2;
        cloth->spring[i2]->p1->particle_accel +=
        ((mySPH->cloth_particles[i1]->fpressure3 + mySPH->cloth_particles[i1]->fviscosity3) / (4.0f*mySPH->cloth_particles[i]->density));
        cloth->spring[i2]->p2->particle_accel +=
        ((mySPH->cloth_particles[i1]->fpressure3 + mySPH->cloth_particles[i1]->fviscosity3) / (4.0f*mySPH->cloth_particles[i]->density));
        cloth->spring[i2+1]->p1->particle_accel +=
        ((mySPH->cloth_particles[i1]->fpressure3 + mySPH->cloth_particles[i1]->fviscosity3) / (4.0f*mySPH->cloth_particles[i]->density));
        cloth->spring[i2+1]->p2->particle_accel +=
        ((mySPH->cloth_particles[i1]->fpressure3 + mySPH->cloth_particles[i1]->fviscosity3) / (4.0f*mySPH->cloth_particles[i]->density));
    }
}
```

```
acceleration = force / mass; // at
acceleration += particle_accel / 4;
```

그리고 천의 파티클들이 물 파티클들과 상호작용하여 받은 force를 바탕으로 accelation을 계산할 수 있다. 이 accelation을 node의 파티클들은 해당 node에, structural spring의 파티클은 인접 두 node에 절반씩, shear spring의 파티클은 인접 네 node에 1/4씩 전달해 integrate단계에서 반영될 수 있도록 하였다. 이때 각 노드는 node파티클 하나에서 1, structural spring 파티클 4개에서 1/2, shear spring파티클 4개에서 1/4씩 가속도를 전달받게 되서 총 4배의 힘을 받게되고, 이를 해결하기 위해 더해진 값을 4로 나누어 사용하였다.