서 울 대 학 교
SEOUL NATIONAL UNIVERSITY

Programming Assignment Report #2

soheon yoo (2021-22823)
yumk0717@snu.ac.kr
Seoul National University - Advanced Graphics (Spring 2023)

Abstract

Position Based Dynamics("PBD") which is developed by Matthias Muller in 2006 is one kind of method in soft-body simulation[1]. This method simulate system only in level of position without explicitly computing forces on system. So it has advantage in speeds than other method. In this Assignment, Soft-body buddy was implemented using Position Based Dynamics via THREE.js graphics API. Collision constraint, Volume conservation constraint and Distance constraint was considered for bunny simulation.

1 Introduction

In the field of computational science, Most important thing in simulation is accuracy. But in Computer graphics, stability and speed of simulation is very important. Especially in Computer games, because real time rendering needs to be done, simulation speeds should be fast. In dynamic objects simulation, force based simulation is traditional. Force can be transformed into acceleration in objects and calculation can be done based on physics rule. Position based dynamics simulate based on position and velocity of vertices not on forces. So calculating speed is faster than force based simulation and the results is not much different from physical reality. Because PBD method is fast and similar to reality, it can be used in simulation of dynamic objects in computer games.

2 Methods

Following model was implemented using python code. Position and velocity of vertices of bunny was calculated each step and was rendered using THREE.js API. Rigid-body ball and Boundary were also implemented. Soft-body bunny and Rigid-body ball cannot cross Boundary.

3 Model explanation

The sudo-algorithm of particle-based dynamics is on (algotirhm 1). The algorithm of PBD can be divided to two part. One is integration part and the other is solving part. Integration part is normal and common in other simulation methods. Acceleration(or force $F = ma$) is integrated to velocities and velocities are integrated to positions. In solving part update position of vertices to satisfy constraints. In this way, vertices of dynamic objects try to make stability via constraint through simulation. From now on various constraints in this assignment will be explained.

for *all vertices* i **do**
  Initialize x_i, v_i ;
end
while *simulating* **do**
  for *N times* **do**
    for *all particles* i **do**
      $v\_i \leftarrow v\_i + hf\_i$
      $p\_i \leftarrow x\_i$
      $x\_i \leftarrow x\_i + hv\_i$
    end
    for *all Constraints* C **do**
      $solve(C, h)$
    end
    for *all particles* i **do**
      $v\_i \leftarrow (x\_i - p\_i)/h$
    end
  end
end
$solve(C, h)$:
for *all particles i of C* **do**
  compute δx_i
  $x\_i \leftarrow x\_i + \Delta x\_i$
end

Algorithm 1: PBD Algorithm

4 Constraint solve

The main problem in this method is setting constraint function for each mechanism. For Soft-body in this project, Collision constraint, edge distance constraint and tetrahedron volume constraint. In solving the constraint, constraint function at $x + \Delta x$ should be zero. When n particles are attached to constraint c, Δx has direction to gradient of constraint and inversely proportional to weighted sum of square of gradient.

$$\Delta x_i = -\frac{w_i c(x_i, ..., x_n)}{\Sigma_j w_j |\nabla_{x_j} c(x_1, ..., x_n)|^2} \nabla_{x_i} c(x_i, ..., x_n)$$

$$\Delta x_i = \lambda w_i \nabla_{x_i} c(x_i, ..., x_n)$$

Each constraint have different constraint function, So calculating largrange multifliers and gradient should be different.

4.1 Volume Conservation Constraint

For volume conservation constraint set constraint function as $6(V - V_0)$. V_0 is initial volume of tetrahedron.Therefore solving this constraint is trying to restore a distorted tetrahedron. For soft body object solve every tetrahedron in mesh every iterations.

Listing 1: Volume Conservation code

```
def ConstrainVolume(self):
    for i in range(self.NumOfTetra):
    #index in tetrahedron
        index_0= self.tetraID[i][0]
        index_1= self.tetraID[i][1]
        index_2= self.tetraID[i][2]
        index_3= self.tetraID[i][3]

    #positions in tetrahedron

        p0 = self.positions[index_0]
        p1 = self.positions[index_1]
        p2 = self.positions[index_2]
        p3 = self.positions[index_3]

        volume = CalcVolume(p0,p1,p2,p3)
    #Cons is constraint
        Cons = 6.0*(volume -self.tetraV0[i])

        grad0 = MYCross(p1-p2,p3-p2)
        grad1 = MYCross(p2-p0,p3-p0)
        grad2 = MYCross(p0-p1,p3-p1)
        grad3 = MYCross(p1-p0,p2-p0)

        weighted_sum_grad = MyNormSq(grad0)+MyNormSq(grad1)+MyNormSq(grad2)+MyNormSq(grad3)
        if(weighted_sum_grad<1e-7):
            continue

        lamda= (-Cons)/weighted_sum_grad
        self.positions[index_0]+=lamda*grad0
        self.positions[index_1]+=lamda*grad1
        self.positions[index_2]+=lamda*grad2
        self.positions[index_3]+=lamda*grad3
```

4.2 Edge distance constraint

Edge distance constraint is maintaining constraint for edge. This constraint is almost similar to volume conservation constraint. If two points for edge is $x1, x2$, the constraint $c(x_1, c_{x2})$ is $|x1 - x2| - l_0$. l_0 is initial rest length for edge.

4.3 Collision constraint

Collision also can be handled by position based dynamics. In this project, Soft body can collide with bounding box or sphere. Setting $C(p) = (p - q_s) \cdot n_s$ is solution for static collision. q_s is near point of p in surface and n_s is normal of surface at q_s. This solution is almost same as clamping outside from static material. Finding q_s, and n_s for floor and sphere is different. Solving for collision only happen if the constraint is negative(collision happened).

Listing 2: Collision constraint

```
def WallCheck(self):
    for i in range(len(self.positions)):
        if(self.positions[i][2]>-boundRange):
            ns=np.array([0,0,1])
            qs=np.array([self.positions[i][0],self.positions[i][1],-boundRange)
            diff=self.positions[i]-qs
            Con= MyDot(diff,ns)
            self.positions[i]+=(-Con*ns)
def BallCheck(self):
    for i in range(len(self.positions)):
        for j in range(BallNum):
            p=self.positions[i]
            ball=BallArray[j]
            radius= ball.radius
            ball_position= ball.positions

            ns = self.positions[i]-ball_position
            ns= ns/np.linalg.norm(ns)
            qs = (radius * ns) +ball_position
            diff = self.positions[i]-qs
            Con = MyDot(diff,ns)
            if(Con<0):
                self.positions[i]+=(-Con*ns)
```

5 Discussion

TimeStepSize and Number of step is important in soft-body simulation . Because constraint is stiffer in smaller time step. In other words, if time step is too long, solving constraint is not enough to maintaining constraint. Long time step will make position of vertices far from constraint, so it may not satisfactory in visual.

In this assignment, mouse picking is implemented. If you click object, objecte will be selected and color will change to blue. Selected object will follow mouse point in world and stick to mouse. For mouse world point, I used function in THREE.js API. And for mouse picking, i updated velocity to be $vec = (mouseworldposition - positions)$. If i implement vec to be force, objects does not stick to mouse. Making constraint between mouse point and objects with some length was one option. But object behave like spring, not stall near mouse point. Below is code for finding world position of mouse pointer.

Listing 3: Function for mouse world position

```
def GetMouseWorld():
    vec= THREE.Vector3.new(mouse.x,mouse.y,0.5)
    vec.unproject(camera)
    dir = vec.sub(camera.position).normalize()
    dist=-camera.position.z/dir.z
    pos= camera.position.clone().add(dir.multiplyScalar(dist))
    return np.array([pos.x,pos.y,pos.z])

```

In the case of rigid body(rigid sphere), only collision handling need to be considered in constraint solving part. If collision happend, position of object was clamped and force was introduced for elastic collision. If sphere collide with boundary or another sphere the velocity will be changed. The change is like reflecting on collision surface, but in this implementation collision is not perfectly elastic collision.

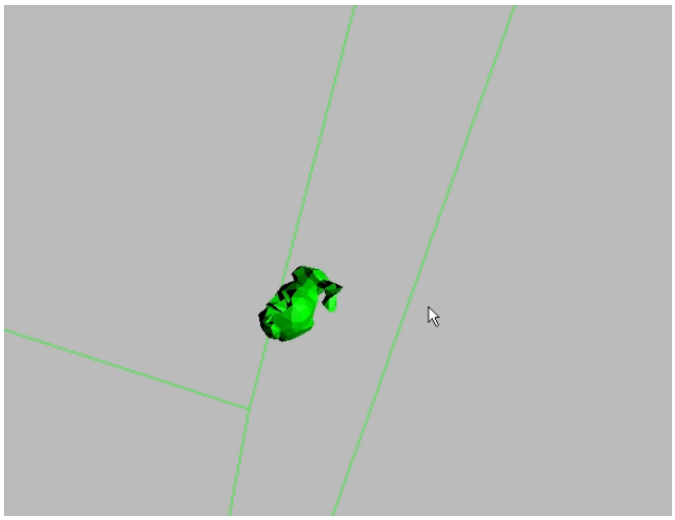

Figure 1: **Soft-body bunny object**

References

[1] M. Müller, B. Heidelberger, M. Hennix, and J. Ratcliff, "Position based dynamics," *Journal of Visual Communication and Image Representation*, vol. 18, no. 2, pp. 109–118, 2007.