COLUMBIA UNIVERSITY

MECE 4510 EVOLUTIONARY COMPUTATION AND DESIGN AUTOMATION

# HW3 Bouncing Cube

Chiqu Li & Yi Jiang

UNI: cl3895

Instructor: Hod Lipson

Grace Hours Used: 0
Grace Hours Remaining: 148h

Nov 9, 2019

# 1. Result

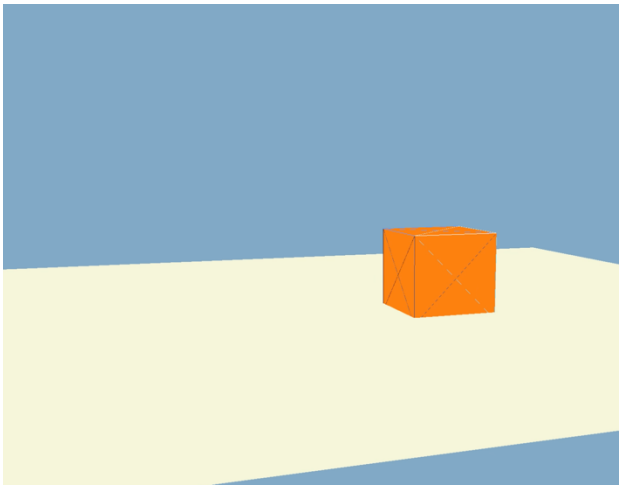

Figure 1. Bouncing Cube
URL: https://www.youtube.com/watch?v=LGUFiFmRGOA

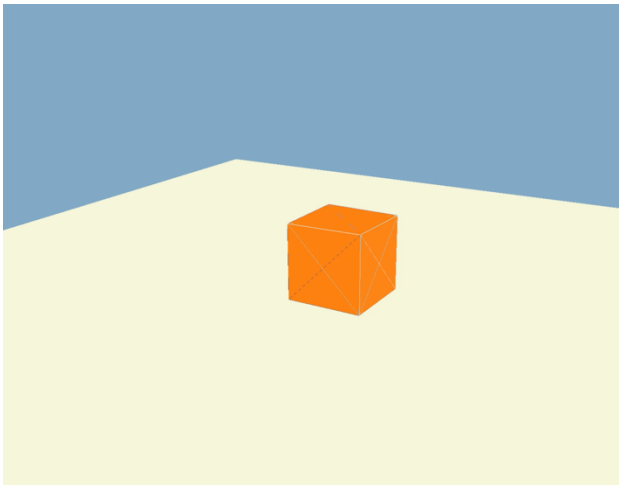

Figure 2. Breathing Cube
URL: https://youtu.be/PQIYZ_yDTgw

# 2. Method

**Description**

we designed a physics simulator which could create a bouncing and breathing cube. The key factor of this project is to have a clear vision of the relationship between mass and spring. Parameters are selected follow the instructions.

**Parameters**

double mass = 0.1;
double length = 0.1;
double gravity = 9.81;
double T = 0;

double timeStep = 0.0005;
double Nground = 10000;
double k = 8000;
double dampening = 0.9999999999;
double frictionCoefficient = 0.1;

**Breathing Parameters**

Longest 4 Springs breathe as the code below, with a frequency of 1/20.

```
if (T > 0) {
    spring[24].L_0 = 1.0 * length + 0.05 * length * sin(20 * T);
    spring[25].L_0 = 1.0 * length + 0.05 * length * sin(20 * T);
    spring[26].L_0 = 1.0 * length + 0.08 * length * sin(20 * T);
    spring[27].L_0 = 1.0 * length + 0.08 * length * sin(20 * T);
      }
```

**Analysis**

In this assignment, our group follows the instruction step by step. We visualize the all process by OpenGL with C++, which is much faster then python. Thanks to the OpenGL tutorial website and YouTube video, we can start a litter easier at the first stage. The spring constraint, k, and ground restoration constant is the points in this simulation. If the structure is too "wobbly", k is too small. If the structure "vibrates", k is too high. Also, if the ground restoration constant is too small, the force from ground will be large and our cube will be rebounded far away. So choosing reasonable parameter is important. For the breathing, we change the rest length of certain springs (diagonal) regularly.

# 3. Performance Plot

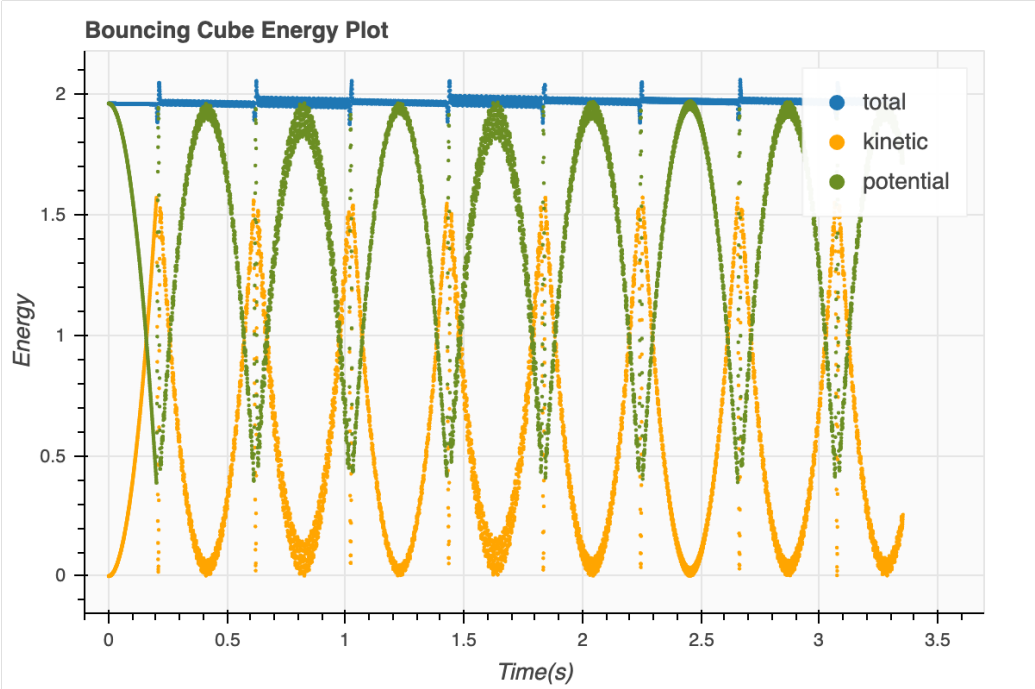


Figure 3. Breathing Cube Energy Plot

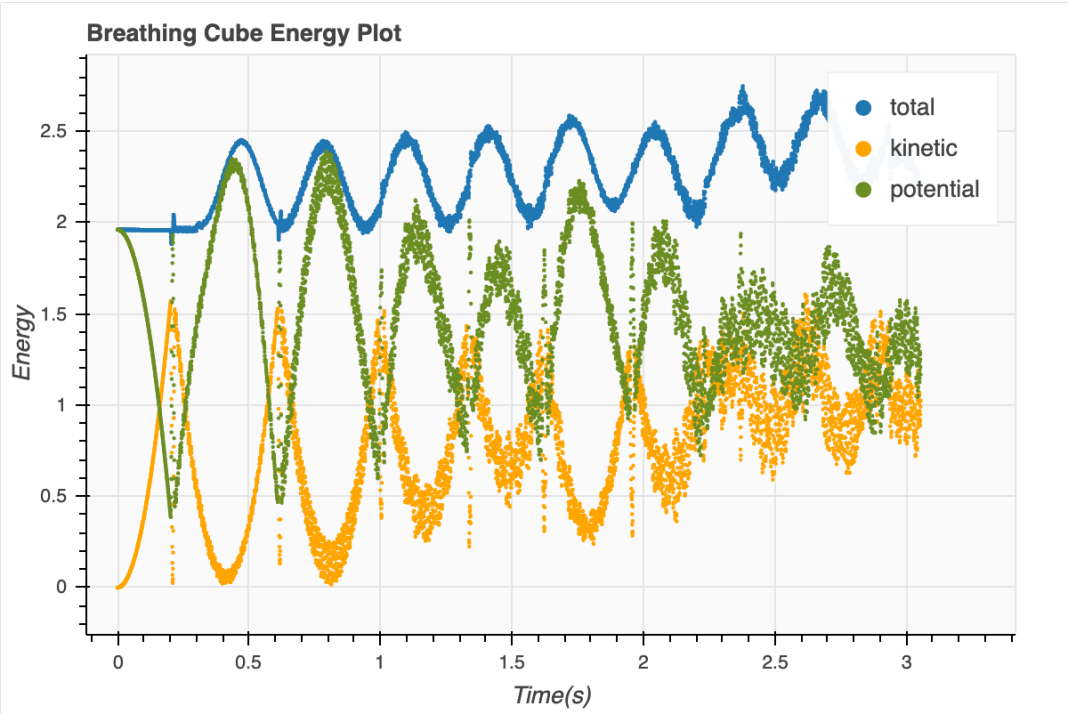

Figure 4. Breathing Cube Energy Plot

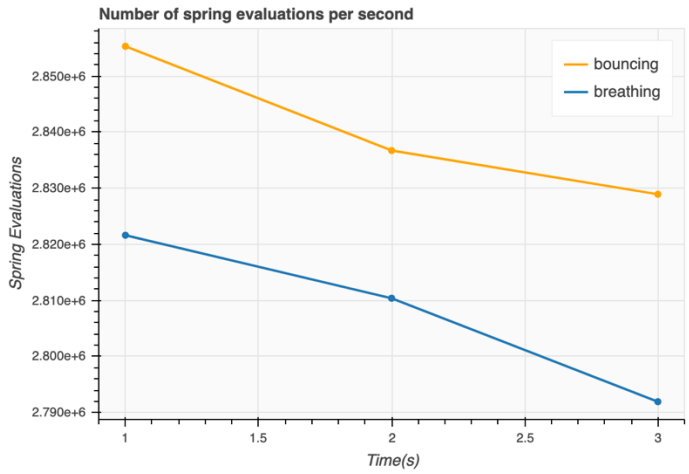


Figure 5. Number of spring evaluations per second

```
for (int i = 0; i < 8; i++) {
    if (cube[i].p[2] <= 0) {
        cForces[i][2] -= Nground * cube[i].p[2];
        groundenergy += Nground * pow(cube[i].p[2], 2) / 2;
        double Fh = sqrt(pow(cForces[i][0], 2) + pow(cForces[i][1], 2));
        double Fv = cForces[i][2];
        if (Fh < Fv * frictionCoefficient) {
            cForces[i][0] = 0;
            cForces[i][1] = 0;
            cube[i].v[0] = 0;
            cube[i].v[1] = 0;
        }
        else {
            double Fh_new = Fh - Fv * frictionCoefficient;
            cForces[i][0] = cForces[i][0] * Fh_new / Fh;
            cForces[i][1] = cForces[i][1] * Fh_new / Fh;
        }
    }
```

Figure 6. Adding Friction

# 4. Additional Tasks

## 4.1 Slight spin

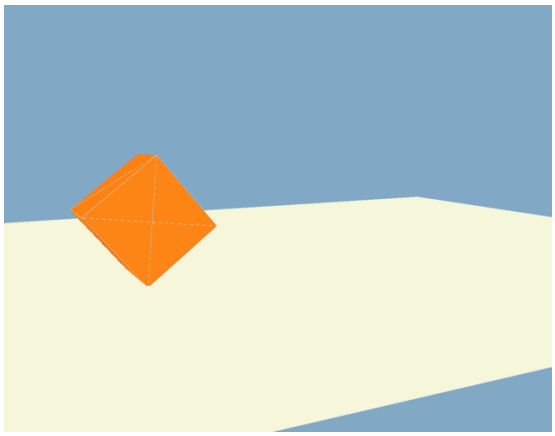

Figure 7. Cube with slight spin

URL: https://www.youtube.com/watch?v=b6_nLgMcqRU

## 4.2 Simple test

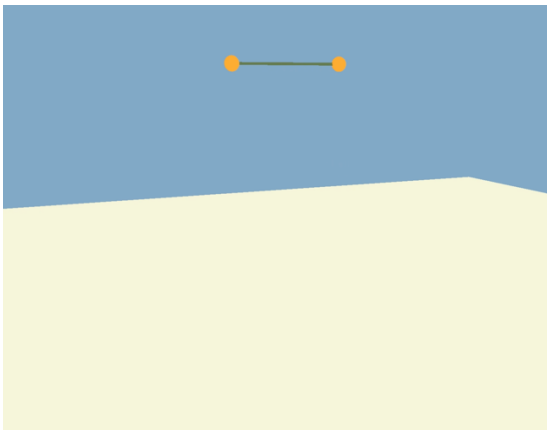

Figure 8. Simple test

URL: https://www.youtube.com/watch?v=IHxncEhd92s&feature=youtu.be

## 4.3 Multiple Cubes

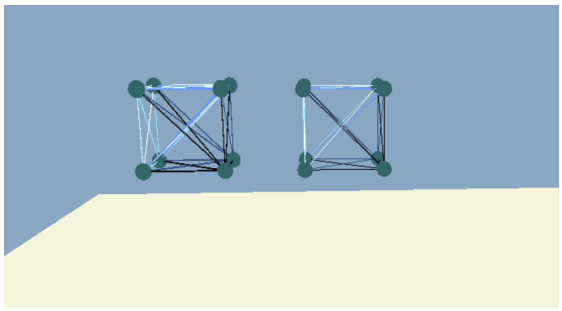

Figure 9. Multiple Cubes

# 5. Appendix

```
#include "HW3.h"
int ppp = 0;
GLfloat worldRotation[16] = { 1,0,0,0,0,0,1,0,0,1,0,0,0,0,0,1 };
std::ofstream myFile("breathing.txt");*/
ofstream outfile4("efficienty.txt");
struct MASS
{
    double m;       // mass
    double p[3];    // 3D position
    double v[3];    // 3D velocity
    double a[3];    // 3D acceleration
};

struct SPRING
{
    double k;       // spring constant
    double L_0;     // rest length
    int m1;         // first mass connected
    int m2;         // second mass connected
};

vector<MASS> cubemass(double mass, double length, double x, double y, double z)
{
    vector<MASS> cube(8);
    cube[0] = { mass, {x,y,z}, {0,0,0}, {0,0,0} };
    cube[1] = { mass, {x ,y+length,z}, {0,0,0}, {0,0,0} };
    cube[2] = { mass, {x ,y,z+length}, {0,0,0}, {0,0,0} };
    cube[3] = { mass, {x + length,y + length,z}, {0,0,0}, {0,0,0} };
    cube[4] = { mass, {x + length,y,z + length}, {0,0,0}, {0,0,0} };
    cube[5] = { mass, {x,y + length,z + length}, {0,0,0}, {0,0,0} };
    cube[6] = { mass, {x + length,y,z}, {0,0,0}, {0,0,0} };
    cube[7] = { mass, {x + length,y + length,z + length}, {0,0,0}, {0,0,0} };
    return cube;
}
vector<SPRING> cubespring(double length, double k)
{
    double short_diagonals = sqrt(2) * length;
```

```
    double long_diagonals = sqrt(3) * length;
    vector<SPRING> spring(28);
    spring[0] = { k,length,0,1 };
    spring[1] = { k,length,0,6 };
    spring[2] = { k,length,0,2 };
    spring[3] = { k,length,1,3 };
    spring[4] = { k,length,3,6 };
    spring[5] = { k,length,2,4 };
    spring[6] = { k,length,4,6 };
    spring[7] = { k,length,2,5};
    spring[8] = { k,length,1,5 };
    spring[9] = { k,length,5,7 };
    spring[10] = { k,length,4,7};
    spring[11] = { k,length,3,7 };

    spring[12] = { k,short_diagonals,3,4 };
    spring[13] = { k,short_diagonals,6,7 };
    spring[14] = { k,short_diagonals,1,7 };
    spring[15] = { k,short_diagonals,3,5 };
    spring[16] = { k,short_diagonals,1,2 };
    spring[17] = { k,short_diagonals,0,5 };
    spring[18] = { k,short_diagonals,2,6 };
    spring[19] = { k,short_diagonals,0,4 };
    spring[20] = { k,short_diagonals,2,7 };
    spring[21] = { k,short_diagonals,4,5 };
    spring[22] = { k,short_diagonals,0,3 };
    spring[23] = { k,short_diagonals,1,6 };

    spring[24] = { k,long_diagonals,0,7 };
    spring[25] = { k,long_diagonals,2,3 };
    spring[26] = { k,long_diagonals,1,4 };
    spring[27] = { k,long_diagonals,5,6 };
    return spring;
}
vector<MASS> cube = cubemass(mass, length, 0, 0, 0);
vector<vector<double>> cForces(8, vector<double>(3));
vector<SPRING> spring = cubespring(length, 1000);
GLuint tex;
GLUquadric* sphere;

void make_tex(void)
```

```
{
    unsigned char data[256][256][3];
    for (int y = 0; y < 255; y++) {
        for (int x = 0; x < 255; x++) {
            unsigned char* p = data[y][x];
            p[0] = p[1] = p[2] = (x ^ y) & 8 ? 255 : 0;
        }
    }
    glGenTextures(1, &tex);
    glBindTexture(GL_TEXTURE_2D, tex);
    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, 256, 256, 0, GL_RGB, GL_UNSIGNED_BYTE, (const
GLvoid*)data);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
}

void init(void)
{
    glEnable(GL_DEPTH_TEST);
    make_tex();
    sphere = gluNewQuadric();
    glEnable(GL_TEXTURE_2D);
}

void drawground()
{
    glPushMatrix();
    glColor3f(0.96078,0.96078,0.86274);
    glBindTexture(GL_TEXTURE_2D,grassTexture);
    glBegin(GL_QUADS);
    glNormal3f( 0, 1, 0);
    glTexCoord2f(0.0,0.0);  glVertex3f(-0.5,+0.0,-0.5);
    glTexCoord2f(0.0,1.0);  glVertex3f(+0.5,+0.0,-0.5);
    glTexCoord2f(1.0,1.0);  glVertex3f(+0.5,+0.0,+0.5);
    glTexCoord2f(1.0,0.0);  glVertex3f(-0.5,+0.0,+0.5);
    glEnd();
    glPopMatrix();
    glDisable(GL_TEXTURE_2D);
    for (int i=0; i < 10; i++) {
   for (int j=-9;  j<10; j++) {
        glColor3f(0, 1, 0);
```

```
        glPushMatrix();
        glMultMatrixf(worldRotation);
        glBegin(GL_LINES);
        glLineWidth(10);
        glVertex3f(-0.5*i/10,0.02 , 0.5*j/10);
        glEnd();
        glPopMatrix();
   }}
}
void drawcube()
{
    for (int i=0; i < 8; i++) {
        for (int j=i+1;  j<8; j++) {
            glColor3f(0.2*i, 0.3*i, 0.5*i);
            glPushMatrix();
            glMultMatrixf(worldRotation);
            glBegin(GL_LINES);
            glLineWidth(10);
            glVertex3f(cube[i].p[0], cube[i].p[1], cube[i].p[2]);
            glVertex3f(cube[j].p[0], cube[j].p[1], cube[j].p[2]);
            glEnd();
            glPopMatrix();

        }
    }

       glPushMatrix();
       glMultMatrixf(worldRotation);



       //  Front
       glColor3f(242.0/255, 138.0/255, 58.0/255);
       glBindTexture(GL_TEXTURE_2D,slimeTexture);
       glBegin(GL_QUADS);
       glNormal3f( 0, 0, 1);
       glTexCoord2f(0.0f,0.0f);    glVertex3f(cube[4].p[0],cube[4].p[1],cube[4].p[2]);
       glTexCoord2f(1.0f,0.0f);    glVertex3f(cube[7].p[0],cube[7].p[1],cube[7].p[2]);
       glTexCoord2f(1.0f,1.0f);    glVertex3f(cube[3].p[0],cube[3].p[1],cube[3].p[2]);
       glTexCoord2f(0.0f,1.0f);    glVertex3f(cube[6].p[0],cube[6].p[1],cube[6].p[2]);
       glEnd();
```

```
//  Back
   glColor3f(240.0/255, 136.0/255, 56.0/255);
   glBindTexture(GL_TEXTURE_2D,slimeTexture);
   glBegin(GL_QUADS);
   glNormal3f( 0, 0, 1);
   glTexCoord2f(0.0f,0.0f);    glVertex3f(cube[2].p[0],cube[2].p[1],cube[2].p[2]);
   glTexCoord2f(1.0f,0.0f);    glVertex3f(cube[5].p[0],cube[5].p[1],cube[5].p[2]);
   glTexCoord2f(1.0f,1.0f);    glVertex3f(cube[1].p[0],cube[1].p[1],cube[1].p[2]);
   glTexCoord2f(0.0f,1.0f);    glVertex3f(cube[0].p[0],cube[0].p[1],cube[0].p[2]);
   glEnd();

//  Right
   glColor3f(240.0/255, 136.0/255, 56.0/255);
   glBindTexture(GL_TEXTURE_2D,slimeTexture);
   glBegin(GL_QUADS);
   glNormal3f( 0, 0, 1);
   glTexCoord2f(0.0f,0.0f);    glVertex3f(cube[5].p[0],cube[5].p[1],cube[5].p[2]);
   glTexCoord2f(1.0f,0.0f);    glVertex3f(cube[7].p[0],cube[7].p[1],cube[7].p[2]);
   glTexCoord2f(1.0f,1.0f);    glVertex3f(cube[3].p[0],cube[3].p[1],cube[3].p[2]);
   glTexCoord2f(0.0f,1.0f);    glVertex3f(cube[1].p[0],cube[1].p[1],cube[1].p[2]);
   glEnd();

//  Left
   glColor3f(240.0/255, 136.0/255, 56.0/255);
   glBindTexture(GL_TEXTURE_2D,slimeTexture);
   glBegin(GL_QUADS);
   glNormal3f( 0, 0, 1);
   glTexCoord2f(0.0f,0.0f);    glVertex3f(cube[2].p[0],cube[2].p[1],cube[2].p[2]);
   glTexCoord2f(1.0f,0.0f);    glVertex3f(cube[4].p[0],cube[4].p[1],cube[4].p[2]);
   glTexCoord2f(1.0f,1.0f);    glVertex3f(cube[6].p[0],cube[6].p[1],cube[6].p[2]);
   glTexCoord2f(0.0f,1.0f);    glVertex3f(cube[0].p[0],cube[0].p[1],cube[0].p[2]);
   glEnd();

//  Top
   glColor3f(240.0/255, 136.0/255, 56.0/255);
   glBindTexture(GL_TEXTURE_2D,slimeTexture);
   glBegin(GL_QUADS);
   glNormal3f( 0, 0, 1);
   glTexCoord2f(0.0f,0.0f);    glVertex3f(cube[5].p[0],cube[5].p[1],cube[5].p[2]);
```

```
      glTexCoord2f(1.0f,0.0f);    glVertex3f(cube[7].p[0],cube[7].p[1],cube[7].p[2]);
      glTexCoord2f(1.0f,1.0f);    glVertex3f(cube[4].p[0],cube[4].p[1],cube[4].p[2]);
      glTexCoord2f(0.0f,1.0f);    glVertex3f(cube[2].p[0],cube[2].p[1],cube[2].p[2]);
      glEnd();

   //  Bottom
      glColor3f(240.0/255, 136.0/255, 56.0/255);
      glBindTexture(GL_TEXTURE_2D,slimeTexture);
      glBegin(GL_QUADS);
      glNormal3f( 0, 0, 1);
      glTexCoord2f(0.0f,0.0f);    glVertex3f(cube[0].p[0],cube[0].p[1],cube[0].p[2]);
      glTexCoord2f(1.0f,0.0f);    glVertex3f(cube[1].p[0],cube[1].p[1],cube[1].p[2]);
      glTexCoord2f(1.0f,1.0f);    glVertex3f(cube[3].p[0],cube[3].p[1],cube[3].p[2]);
      glTexCoord2f(0.0f,1.0f);    glVertex3f(cube[6].p[0],cube[6].p[1],cube[6].p[2]);
      glEnd();
       glPopMatrix();
    glDisable(GL_TEXTURE_2D);
}

float L(MASS mass1, MASS mass2) {
   double length = sqrt(pow((mass1.p[0] - mass2.p[0]), 2) + pow((mass1.p[1] -
mass2.p[1]), 2) + pow((mass1.p[2] - mass2.p[2]), 2));

   return length;
}
void simulate() {

   for (int i = 0; i < 8; i++) {
   cForces[i][0] = 0.0;
   cForces[i][1] = 0.0;
   cForces[i][2] = -cube[i].m * gravity;}
//    if (oneforce==0){
//       cForces[1][0] = 0.1;
//       oneforce = 1;
//    }

   if (T == 0.005) {
    cForces[5][1] = 2.0;
   }

   for (int i = 0; i < 28; i++) {
```

```
        cout<<ppp<<endl;

        ppp++;

        if (T==0.001 or T == 1.0 or T == 2.0 or T == 3.0) {
         outfile4 << " " << T << " " << ppp << endl;
         ppp = 0;
        }
        if (T > 0) {
         spring[24].L_0 = 1.0 * length + 0.05 * length * sin(20 * T);
       spring[25].L_0 = 1.0 * length + 0.05 * length * sin(20 * T);
                 spring[26].L_0 = 1.0 * length + 0.08 * length * sin(20 * T);
                 spring[27].L_0 = 1.0 * length + 0.08 * length * sin(20 * T);
        }
        MASS mass1 = cube[spring[i].m1];
        MASS mass2 = cube[spring[i].m2];
        double pd[3] = { mass2.p[0] - mass1.p[0],mass2.p[1] - mass1.p[1],mass2.p[2] -
mass1.p[2] };
        double new_L = L( mass1, mass2);
        double L_0 = spring[i].L_0;
        double force = k * fabs(new_L - L_0);
        //cout <<i<<"---new_L---" <<new_L << endl;
        double norm_pd[3] = { pd[0] / new_L, pd[1] / new_L, pd[2] / new_L };
        //compression
        if (new_L < spring[i].L_0) {
            cForces[spring[i].m1][0] -= norm_pd[0] * force;
            cForces[spring[i].m1][1] -= norm_pd[1] * force;
            cForces[spring[i].m1][2] -= norm_pd[2] * force;
            cForces[spring[i].m2][0] += norm_pd[0] * force;
            cForces[spring[i].m2][1] += norm_pd[1] * force;
            cForces[spring[i].m2][2] += norm_pd[2] * force;
        }

        //tension
        else{
            cForces[spring[i].m1][0] += norm_pd[0] * force;
            cForces[spring[i].m1][1] += norm_pd[1] * force;
            cForces[spring[i].m1][2] += norm_pd[2] * force;
            cForces[spring[i].m2][0] -= norm_pd[0] * force;
            cForces[spring[i].m2][1] -= norm_pd[1] * force;
            cForces[spring[i].m2][2] -= norm_pd[2] * force;
```

```
        }
        //cout <<i<<"---52---" <<cForces[5][2] << endl;
    }
    for (int i = 0; i < 8; i++) {
        //cout << cForces[i][2] << endl;
        //cout << 's' << endl;
        if (cube[i].p[2] < 0) {
         cForces[i][2] -= Nground * cube[i].p[2];
         double Fh = sqrt(pow(cForces[i][0], 2) + pow(cForces[i][1], 2));
         double Fv = cForces[i][2];
         if (Fh < Fv * frictionCoefficient) {
          cForces[i][0] = 0;
          cForces[i][1] = 0;
          cube[i].v[0] = 0;
          cube[i].v[1] = 0;
         }
         else {
          double Fh_new = Fh -  Fv * frictionCoefficient;
          cForces[i][0] = cForces[i][0] * Fh_new / Fh;
          cForces[i][1] = cForces[i][1] * Fh_new / Fh;
         }
        }
//       if (cube[i].p[2] < 0) {
//           cForces[i][2] -= Nground * cube[i].p[2];
//       }
        for (int j = 0; j < 3; j++) {
           cube[i].a[j] = cForces[i][j] / cube[i].m;
           cube[i].v[j] += cube[i].a[j] * timeStep;
           cube[i].p[j] += cube[i].v[j] * timeStep;
           //cout << cube[i].p[j] << endl;

        }
    //cout <<i <<cube[i].p[2] << endl;
    //cout << cForces[1][2] << endl;
    }
    drawcube();
    drawground();
    T = T + timeStep;
}
```

```
void Print(const char* format, ...)
{
    char    buf[LEN];
    char* ch = buf;
    va_list args;
    //  Turn the parameters into a character string
    va_start(args, format);
    vsnprintf(buf, LEN, format, args);
    va_end(args);
    //  Display the characters one at a time at the current raster position
    while (*ch)
       glutBitmapCharacter(GLUT_BITMAP_HELVETICA_18, *ch++);
}

void display()
{
    const double len = 0.2;  //  Length of axes
    //  Erase the window and the depth buffer
    glClearColor(0.5372549, 0.6549019, 0.760784, 1.0);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    //  Enable Z-buffering in OpenGL
    glEnable(GL_DEPTH_TEST);
    //  Undo previous transformations
    glLoadIdentity();
    //  Eye position
    double Ex = -1 * dim * Sin(th) * Cos(ph);
    double Ey = +1 * dim * Sin(ph);
    double Ez = +1 * dim * Cos(th) * Cos(ph);
    gluLookAt(Ex, Ey, Ez, 0, 0, 0, 0, Cos(ph), 0);

    simulate();
    //drawcube();


    //  Draw axes
    glColor3f(1, 0, 0);
//
    glFlush();
    //  Make the rendered scene visible
    glutSwapBuffers();
```

```
}

/*
 *  GLUT calls this routine when an arrow key is pressed
 */
void special(int key, int x, int y)
{
   //  Right arrow key - increase angle by 5 degrees
   if (key == GLUT_KEY_RIGHT)
      th += 5;
   //  Left arrow key - decrease angle by 5 degrees
   else if (key == GLUT_KEY_LEFT)
      th -= 5;
   //  Up arrow key - increase elevation by 5 degrees
   else if (key == GLUT_KEY_UP)
   {
      if (ph + 5 < 90)
      {
         ph += 5;
      }
   }
   //  Down arrow key - decrease elevation by 5 degrees
   else if (key == GLUT_KEY_DOWN)
   {
      if (ph - 5 > 0)
      {
         ph -= 5;
      }
   }
   //  Keep angles to +/-360 degrees
   th %= 360;
   ph %= 360;
   //  Tell GLUT it is necessary to redisplay the scene
   glutPostRedisplay();
}

/*
 *  Set projection
 */
void Project(double fov, double asp, double dim)
{
```

```
   //  Tell OpenGL we want to manipulate the projection matrix
   glMatrixMode(GL_PROJECTION);
   //  Undo previous transformations
   glLoadIdentity();
   //  Perspective transformation
   if (fov)
      gluPerspective(fov, asp, dim / 16, 16 * dim);
   //  Orthogonal transformation
   else
      glOrtho(-asp * dim, asp * dim, -dim, +dim, -dim, +dim);
   //  Switch to manipulating the model matrix
   glMatrixMode(GL_MODELVIEW);
   //  Undo previous transformations
   glLoadIdentity();
}

/*
 *  GLUT calls this routine when a key is pressed
 */
void key(unsigned char ch, int x, int y)
{
   //  Exit on ESC
   if (ch == 27)
      exit(0);
   //  Reset view angle
   else if (ch == '0')
      th = ph = 0;
   //  Toggle axes
   else if (ch == 'a' || ch == 'A')
      axes = 1 - axes;
   //  Change field of view angle
   else if (ch == '-' && ch > 1)
      fov++;
   else if (ch == '=' && ch < 179)
      fov--;
   //  PageUp key - increase dim
   else if (ch == GLUT_KEY_PAGE_DOWN) {
      dim += 0.1;
   }
   //  PageDown key - decrease dim
   else if (ch == GLUT_KEY_PAGE_UP && dim > 1) {
```

```
        dim -= 0.1;
    }
    //  Keep angles to +/-360 degrees
    th %= 360;
    ph %= 360;
    //  Reproject
    Project(fov, asp, dim);
    //  Tell GLUT it is necessary to redisplay the scene
    glutPostRedisplay();
}

/*
 *  GLUT calls this routine when the window is resized
 */
void reshape(int width, int height)
{
    //  Ratio of the width to the height of the window
    asp = (height > 0) ? (double)width / height : 1;
    //  Set the viewport to the entire window
    glViewport(0, 0, width, height);
    //  Set projection
    Project(fov, asp, dim);
}

/*
 *  GLUT calls this toutine when there is nothing else to do
 */
void idle()
{
    glutPostRedisplay();
}

int main(int argc, char* argv[])
{
    // Initialize GLUT and process user parameters
    glutInit(&argc, argv);
    // double buffered, true color 600*600
    glutInitWindowSize(1000, 800);
    glutInitDisplayMode(GLUT_RGB | GLUT_DEPTH | GLUT_DOUBLE);
    // create the window
    glutCreateWindow("Slight Spin_yj2563_cl3895");
```

```
    //  Tell GLUT to call "idle" when there is nothing else to do
    glutIdleFunc(idle);
    //  Tell GLUT to call "display" when the scene should be drawn
    glutDisplayFunc(display);
    //  Tell GLUT to call "reshape" when the window is resized
    glutReshapeFunc(reshape);
    //  Tell GLUT to call "special" when an arrow key is pressed
    glutSpecialFunc(special);
    //  Tell GLUT to call "key" when a key is pressed
    glutKeyboardFunc(key);
    init();
    //  Pass control to GLUT so it can interact with the user
    glutMainLoop();
    return 0;
};
```