COLUMBIA UNIVERSITY

MECE 4510 EVOLUTIONARY COMPUTATION AND DESIGN AUTOMATION

Parametric Robot

Chiqu Li& Yi Jinag

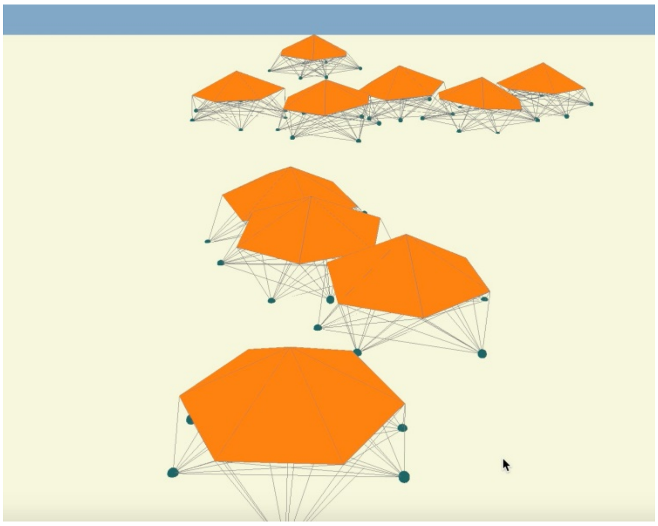
UNI: cl3895

Instructor: Dr. Hod Lipson

Grace Hours Used: 2
Grace Hours Remaining: 146h

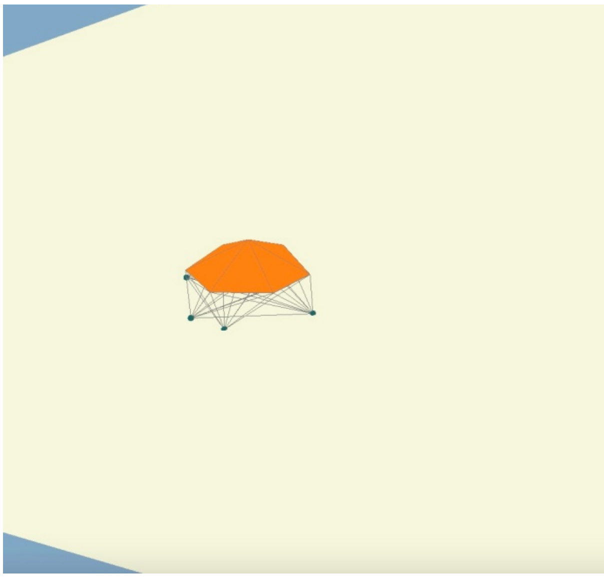
Nov 25, 2019

Result

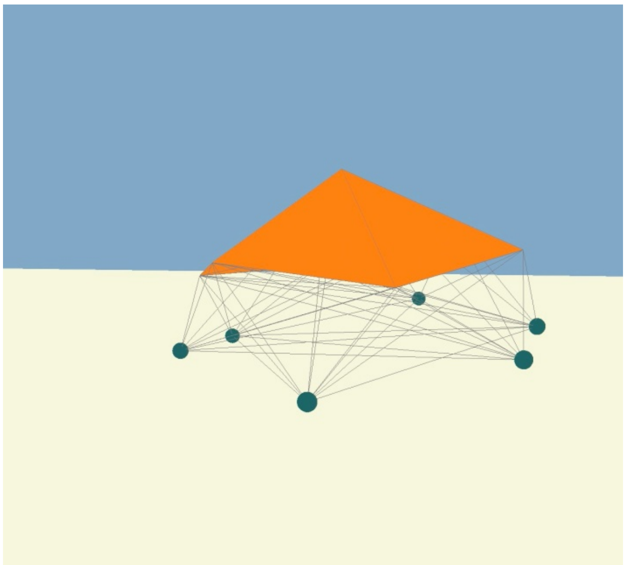


https://youtu.be/hAcLQ7_2fcw

the fast robot



https://youtu.be/GttI6kPbkUk



https://youtu.be/GttI6kPbkUk

Method

Description

In this phase, we should evolve a robot with a fixed morphology. Unlike last phase of assignment 3, we change the cube robot to connected tetrahedrons robot. This robot contains 13 masses and 72 springs, the picture as followed. In order to evolve it to move in a direction, we choose 12 springs which is connected to robot's feet to evolve. Also, to make it stable, the springs at the top of robot are set to be harder (K is bigger). The robot runs for 20 second for each generation and then we evolve it.

Evolution

Like assignment1 and assignment2, in the evolution process, we need population size, definition of fitness, selection, crossover and mutation. In this phase, a gene is the parameter b, c, k of the function $L_0=a+b*sin(wt+c)$ and we have 12 genes in a robot and 10 robots are a population. Also, the fitness is the distance of a robot moves in one generation. We select top 50% population to the next generation and choose two parents from whole population to crossover. The crossover is that exchanging two pieces of gene randomly. And then we let the new population mutate, randomly changing a gene of a spring.

Parameter

1. simulation parameters

timeStep = 0.0002;
Nground = 50000;
dampening = 1;
frictionCoefficient = 0.5;
generationT = 6s; // the time in one generation

2. robot parameters

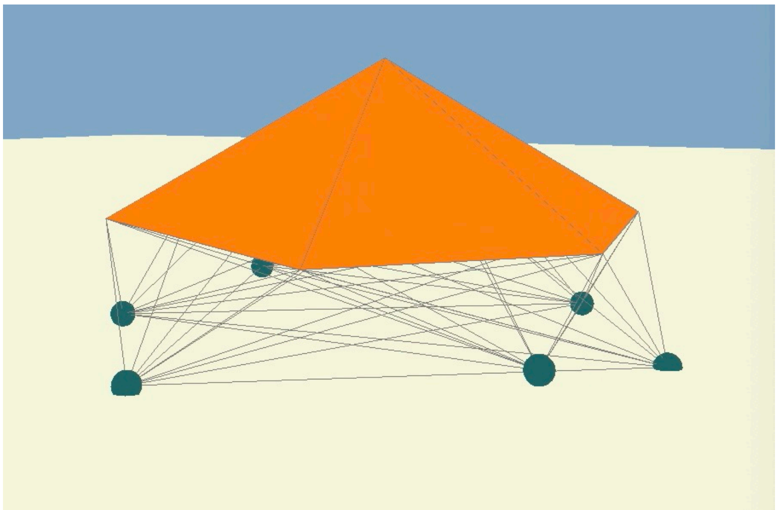
k = 9000; // spring constant to other spring that are not evolved
```
mass = 2; // mass of one joint of robot
length = 0.1;
gravity = 9.81;
```
num_sp = 72; //number of total spring
num_m = 13; //number of total mass

3. evolutionary parameters

k : 600-5000; the range of evolved spring
b: -1——1; the range of b
c: -2PI——2PI;the range of c
size = 30; //population size
generation = 2000; // evaluation
mutation probability = 0.5

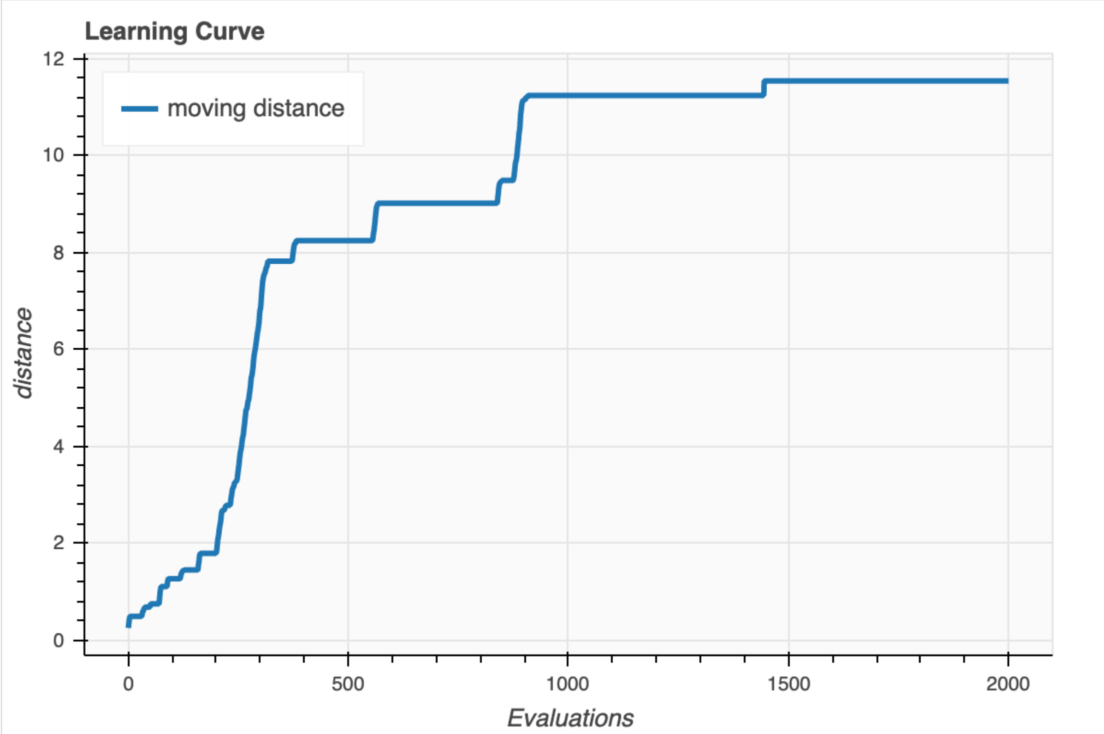
Analysis

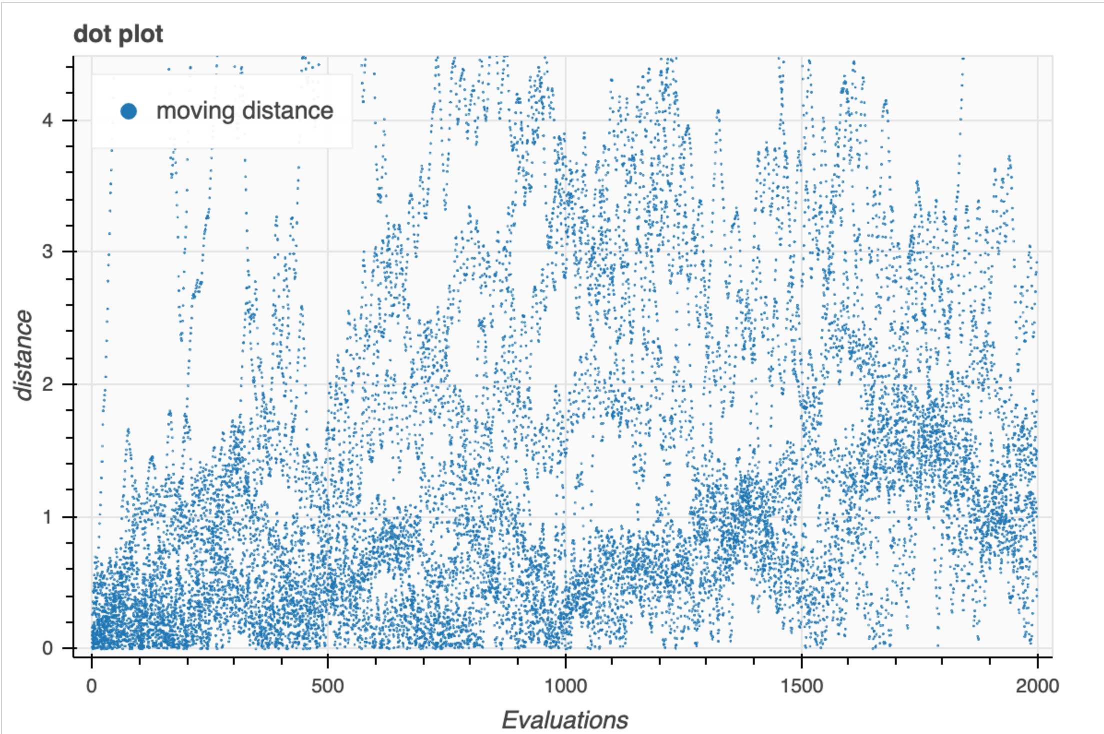
After lots of tempts to get a reasonable range of parameters, we concluded that all of the parameters plays a significant role in the robot performance. Among three parameters(k, b, c), it turns out that the k is the key of the robot. If the k is too large or too small, robots would be very unstable. So we choose a range of 600-5000.

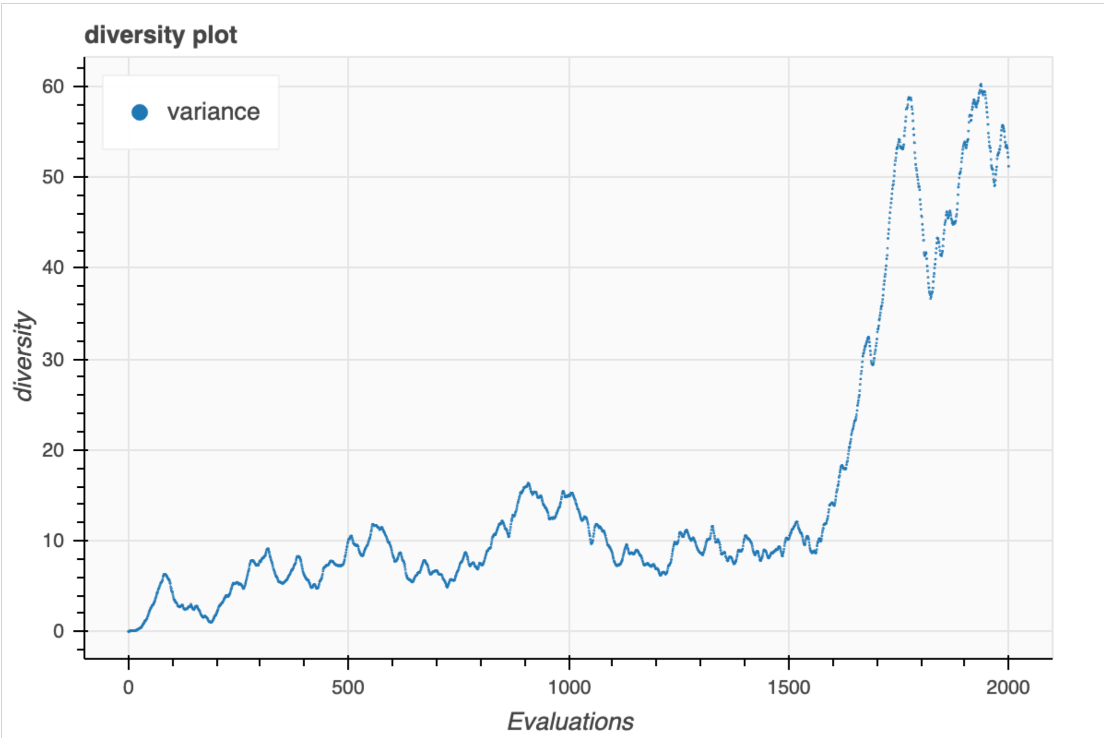


robot

3. Performance






diversity plot
variance
diversity
Evaluations
0
10
20
30
40
50
60
0
500
1000
1500
2000

Appendix

```
GLfloat worldRotation[16] = { 1,0,0,0,0,0,1,0,0,1,0,0,0,0,0,1 };

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
};

struct GENE {
    double k;
    double b;
    double c;
};

//ofstream outFile1("distance.txt");
//ofstream outFile2("bestgene.txt");
//ofstream outFile3("dot.txt");

//struct Cube {
//    struct MASS cubemass[8];
//    struct SPRING cubespring[28];
//};

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;
}


//vector<MASS> joint = jointmass(mass, length, 0, 0, 0.01);
//
```

```
//
//vector<SPRING> spring = cubespring(length, k);
vector<vector<double>> cForces(13, vector<double>(3));

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);
}



class Robot {
private:
    vector<GENE> walkgene;
public:
    vector<MASS> joint;
    vector<SPRING> spring;

    vector<double> original;
    double initl[3] = { 0,0,0 };
    Robot(double x, double y, double z, vector<GENE> newGene)
    {
```

```
        initl[0] = x;
        initl[1] = y;
        initl[2] = z;
        walkgene = newGene;
        joint = jointmass(x, y, z);
        cubespring(walkgene);

    }

    vector<MASS> jointmass(double x, double y, double z)
    {
        vector<MASS> joint(13);
        double subtle = length / 2;
        joint[0] = { mass, {x + 1.5 * length,y + length * sqrt(3) / 2, z + 1 * length},
{0,0,0}, {0,0,0} };
        joint[1] = { mass, {x + length   ,y + sqrt(3) * length,z + 0.5 * length}, {0,0,0},
{0,0,0} };
        joint[2] = { mass, {x + 2 * length  ,y + sqrt(3) * length ,z + 0.5 * length},
{0,0,0}, {0,0,0} };
        joint[3] = { mass, {x + 0.5*length ,y + length / 2 * sqrt(3),z + 0.5 * length},
{0,0,0}, {0,0,0} };
        joint[4] = { mass, {x + 2.5* length ,y + length / 2 * sqrt(3),z + 0.5 * length},
{0,0,0}, {0,0,0} };
        joint[5] = { mass, {x + length,y,z + 0.5 * length}, {0,0,0}, {0,0,0} };
        joint[6] = { mass, {x + 2 * length,y ,z + 0.5 * length}, {0,0,0}, {0,0,0} };
        joint[7] = { mass, {x + 1.5 * length ,y + length * 3 * sqrt(3) / 2 - subtle,z},
{0,0,0}, {0,0,0} };
        joint[8] = { mass, {x + subtle ,y + sqrt(3) * length - subtle ,z}, {0,0,0},
{0,0,0} };
        joint[9] = { mass, {x + 3 * length - subtle,y + sqrt(3) * length - subtle,z },
{0,0,0}, {0,0,0} };
        joint[10] = { mass, {x + subtle ,y + subtle ,z }, {0,0,0}, {0,0,0} };
        joint[11] = { mass, {x + 3 * length - subtle, y + subtle ,z}, {0,0,0}, {0,0,0} };
        joint[12] = { mass, {x + 1.5 * length,y - length * sqrt(3) / 2 + subtle, z},
{0,0,0}, {0,0,0} };
        return joint;
    }
    void cubespring(vector<GENE> walkgene)
    {

        int pointer = 0;
        for (int i = 0; i < 13 - 1; i++) {
            for (int j = i + 1; j < 13; j++) {
```

```
        original.push_back(L(joint[i],joint[j]));
        spring.push_back( { walkgene[pointer].k,L(joint[i],joint[j]),i,j });
        cout<<spring[i*j].k<<endl;
        pointer++;
        if (i == 7 and j == 8) {
            spring.pop_back();
            pointer--;
            cout<<"enterrrrrr"<<endl;
        }
        else if (i == 7 and j == 9) {
            spring.pop_back();
            pointer--;
        }
        else if (i == 8 and j == 10) {
            spring.pop_back();
            pointer--;
        }
        else if (i == 10 and j == 12) {
            spring.pop_back();
            pointer--;
        }
        else if (i == 11 and j == 12) {
            spring.pop_back();
            pointer--;
        }
        else if (i == 9 and j == 11) {
            spring.pop_back();
            pointer--;
        }

    }
}
//    spring[0] = { walkgene[0].k,L(joint[0],joint[3]),0,3 };
//    spring[1] = { walkgene[1].k,L(joint[0],joint[4]),0,4 };
//    spring[2] = { walkgene[2].k,L(joint[0],joint[5]),0,5 };
//
//    spring[3] = { walkgene[3].k,L(joint[1],joint[3]),1,3 };
//    spring[4] = { walkgene[4].k,L(joint[1],joint[5]),1,5 };
//    spring[5] = { walkgene[5].k,L(joint[1],joint[4]),1,4 };
//
//    spring[6] = { walkgene[6].k,L(joint[2],joint[4]),2,4 };
//    spring[7] = { walkgene[7].k,L(joint[2],joint[5]),2,5 };
//    spring[8] = { walkgene[8].k,L(joint[2],joint[3]),2,3 };
//
```

```
    //    spring[9] = { walkgene[9].k,L(joint[4],joint[5]),4,5 };
    //    spring[10] = { walkgene[10].k,L(joint[4],joint[3]),4,3 };
    //    spring[11] = { walkgene[11].k,L(joint[5],joint[3]),3,5 };
    //
    //    spring[12] = { walkgene[12].k,L(joint[3],joint[6]),3,6 };
    //    spring[13] = { walkgene[13].k,L(joint[4],joint[6]),4,6 };
    //    spring[14] = { walkgene[14].k,L(joint[5],joint[6]),5,6 };
    //
    //    spring[15] = { walkgene[15].k,L(joint[0],joint[6]),0,6 };
    //    spring[16] = { walkgene[16].k,L(joint[1],joint[6]),1,6 };
    //    spring[17] = { walkgene[17].k,L(joint[2],joint[6]),2,6 };
    //
    //    spring[18] = { walkgene[18].k,L(joint[3],joint[7]),3,7 };
    //    spring[19] = { walkgene[19].k,L(joint[4],joint[7]),4,7 };
    //    spring[20] = { walkgene[20].k,L(joint[5],joint[7]),5,7 };
    //       spring[21] = { walkgene[21].k,L(joint[6],joint[7]),6,7 };
    //
    //   spring[22] = { walkgene[22].k,L(joint[3],joint[8]),3,8 };
    //   spring[23] = { walkgene[23].k,L(joint[4],joint[8]),4,8 };
    //   spring[24] = { walkgene[24].k,L(joint[5],joint[8]),5,8 };
    //   spring[25] = { walkgene[25].k,L(joint[6],joint[8]),6,8 };
    cout<< spring[0].k <<endl;
    cout<<"creatspring"<<endl;

}

void drawcube()
{

    glPushMatrix();
    glMultMatrixf(worldRotation);

    glBegin(GL_TRIANGLES);
    glColor3f(240.0/255, 136.0/255, 56.0/255);
    glVertex3f(GLfloat(joint[0].p[0]),                      GLfloat(joint[0].p[1]),
GLfloat(joint[0].p[2]));
    glVertex3f(GLfloat(joint[1].p[0]),                      GLfloat(joint[1].p[1]),
GLfloat(joint[1].p[2]));
    glVertex3f(GLfloat(joint[2].p[0]),                      GLfloat(joint[2].p[1]),
GLfloat(joint[2].p[2]));
    glEnd();

    glBegin(GL_TRIANGLES);
    glColor3f(240.0/255, 136.0/255, 56.0/255);
```

```
	glVertex3f(GLfloat(joint[0].p[0]),			GLfloat(joint[0].p[1]),
GLfloat(joint[0].p[2]));
	glVertex3f(GLfloat(joint[1].p[0]),			GLfloat(joint[1].p[1]),
GLfloat(joint[1].p[2]));
	glVertex3f(GLfloat(joint[3].p[0]),			GLfloat(joint[3].p[1]),
GLfloat(joint[3].p[2]));
	glEnd();

	glBegin(GL_TRIANGLES);
	glColor3f(240.0/255, 136.0/255, 56.0/255);
	glVertex3f(GLfloat(joint[0].p[0]),			GLfloat(joint[0].p[1]),
GLfloat(joint[0].p[2]));
	glVertex3f(GLfloat(joint[5].p[0]),			GLfloat(joint[5].p[1]),
GLfloat(joint[5].p[2]));
	glVertex3f(GLfloat(joint[3].p[0]),			GLfloat(joint[3].p[1]),
GLfloat(joint[3].p[2]));
	glEnd();

	glBegin(GL_TRIANGLES);
	glColor3f(240.0/255, 136.0/255, 56.0/255);
	glVertex3f(GLfloat(joint[0].p[0]),			GLfloat(joint[0].p[1]),
GLfloat(joint[0].p[2]));
	glVertex3f(GLfloat(joint[5].p[0]),			GLfloat(joint[5].p[1]),
GLfloat(joint[5].p[2]));
	glVertex3f(GLfloat(joint[6].p[0]),			GLfloat(joint[6].p[1]),
GLfloat(joint[6].p[2]));
	glEnd();

	glBegin(GL_TRIANGLES);
	glColor3f(240.0/255, 136.0/255, 56.0/255);
	glVertex3f(GLfloat(joint[0].p[0]),			GLfloat(joint[0].p[1]),
GLfloat(joint[0].p[2]));
	glVertex3f(GLfloat(joint[4].p[0]),			GLfloat(joint[4].p[1]),
GLfloat(joint[4].p[2]));
	glVertex3f(GLfloat(joint[6].p[0]),			GLfloat(joint[6].p[1]),
GLfloat(joint[6].p[2]));
	glEnd();

	glBegin(GL_TRIANGLES);
	glColor3f(240.0/255, 136.0/255, 56.0/255);
	glVertex3f(GLfloat(joint[0].p[0]),			GLfloat(joint[0].p[1]),
GLfloat(joint[0].p[2]));
	glVertex3f(GLfloat(joint[4].p[0]),			GLfloat(joint[4].p[1]),
GLfloat(joint[4].p[2]));
```

```
	glVertex3f(GLfloat(joint[2].p[0]),			GLfloat(joint[2].p[1]),
GLfloat(joint[2].p[2]));
	glEnd();
	//foot
//	glBegin(GL_TRIANGLES);
//	glColor3f(0.2, 0.1, 0.3);
//			glVertex3f(GLfloat(joint[1].p[0]),	GLfloat(joint[1].p[1]),
GLfloat(joint[1].p[2]));
//			glVertex3f(GLfloat(joint[7].p[0]),	GLfloat(joint[7].p[1]),
GLfloat(joint[7].p[2]));
//			glVertex3f(GLfloat(joint[2].p[0]),	GLfloat(joint[2].p[1]),
GLfloat(joint[2].p[2]));
//	glEnd();
//
//	glBegin(GL_TRIANGLES);
//	glColor3f(0.2, 0.1, 0.3);
//			glVertex3f(GLfloat(joint[8].p[0]),	GLfloat(joint[8].p[1]),
GLfloat(joint[8].p[2]));
//			glVertex3f(GLfloat(joint[1].p[0]),	GLfloat(joint[1].p[1]),
GLfloat(joint[1].p[2]));
//			glVertex3f(GLfloat(joint[3].p[0]),	GLfloat(joint[3].p[1]),
GLfloat(joint[3].p[2]));
//	glEnd();
//
//	glBegin(GL_TRIANGLES);
//	glColor3f(0.2, 0.1, 0.3);
//			glVertex3f(GLfloat(joint[9].p[0]),	GLfloat(joint[9].p[1]),
GLfloat(joint[9].p[2]));
//			glVertex3f(GLfloat(joint[2].p[0]),	GLfloat(joint[2].p[1]),
GLfloat(joint[2].p[2]));
//			glVertex3f(GLfloat(joint[4].p[0]),	GLfloat(joint[4].p[1]),
GLfloat(joint[4].p[2]));
//	glEnd();
//
//	glBegin(GL_TRIANGLES);
//	glColor3f(0.2, 0.1, 0.3);
//			glVertex3f(GLfloat(joint[11].p[0]),	GLfloat(joint[11].p[1]),
GLfloat(joint[11].p[2]));
//			glVertex3f(GLfloat(joint[6].p[0]),	GLfloat(joint[6].p[1]),
GLfloat(joint[6].p[2]));
//			glVertex3f(GLfloat(joint[4].p[0]),	GLfloat(joint[4].p[1]),
GLfloat(joint[4].p[2]));
//	glEnd();
//
```

```
//       glBegin(GL_TRIANGLES);
//       glColor3f(0.2, 0.1, 0.3);
//                       glVertex3f(GLfloat(joint[12].p[0]),   GLfloat(joint[12].p[1]),
GLfloat(joint[12].p[2]));
//                        glVertex3f(GLfloat(joint[6].p[0]),   GLfloat(joint[6].p[1]),
GLfloat(joint[6].p[2]));
//                        glVertex3f(GLfloat(joint[5].p[0]),   GLfloat(joint[5].p[1]),
GLfloat(joint[5].p[2]));
//       glEnd();
//
//       glBegin(GL_TRIANGLES);
//       glColor3f(0.2, 0.1, 0.3);
//                       glVertex3f(GLfloat(joint[10].p[0]),   GLfloat(joint[10].p[1]),
GLfloat(joint[10].p[2]));
//                        glVertex3f(GLfloat(joint[3].p[0]),   GLfloat(joint[3].p[1]),
GLfloat(joint[3].p[2]));
//                        glVertex3f(GLfloat(joint[5].p[0]),   GLfloat(joint[5].p[1]),
GLfloat(joint[5].p[2]));
//       glEnd();
//
        glPopMatrix();



//
        GLUquadric* quad[13];
        for (int i = 7; i < 13; i++) {

            glColor3f(0.2, 0.4, 0.4);
            if (i == 6) { glColor3f(1, 0, 0); }
            quad[i] = gluNewQuadric();
            glPushMatrix();
            glMultMatrixf(worldRotation);
            glTranslated(joint[i].p[0], joint[i].p[1], joint[i].p[2]);

            gluSphere(quad[i], 1.0 / 200, 10, 10);
//            if (i == 6) { gluSphere(quad[i], 1.0 / 100, 10, 10); }
            glPopMatrix();

        }

        for (int i = 0; i < num_sp; i++) {
            int a = spring[i].m1;
            int b = spring[i].m2;
```

```
            glColor3f(0.5, 0.5, 0.5);
            glPushMatrix();
            glMultMatrixf(worldRotation);
            glBegin(GL_LINES);
            glLineWidth(20);
            glVertex3f(joint[a].p[0], joint[a].p[1], joint[a].p[2]);
            glVertex3f(joint[b].p[0], joint[b].p[1], joint[b].p[2]);
            glEnd();
            glPopMatrix();

        }
    }

    void drawground()
    {
        glPushMatrix();
        glColor3f(0.96078, 0.96078, 0.86274);
        glBegin(GL_QUADS);
        glNormal3f(0, 1, 0);
        glTexCoord2f(0.0, 0.0);  glVertex3f(-1.5, +0.0, -1.5);
        glTexCoord2f(0.0, 1);  glVertex3f(+1.5, +0.0, -1.5);
        glTexCoord2f(1, 1);  glVertex3f(+1.5, +0.0, +1.5);
        glTexCoord2f(1, 0.0);  glVertex3f(-1.5, +0.0, +1.5);
        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);
                //
glV/Users/chiqu/class2019fall/EA/assignment3/HW3/HW3.cppertex3f(0.5*i/10,     0.5*j/10,
0.01);
                glEnd();
                glPopMatrix();
            }
        }
    }

    void simulate() {
```

```
        for (int i = 0; i < 13; i++) {
            cForces[i][0] = 0.0;
            cForces[i][1] = 0.0;
            if (T == 1 or T == 10) {
                cForces[i][0] = 10.0;
            }

            cForces[i][2] = -joint[i].m * gravity;
        }

        for (int i = 0; i < num_sp; i++) {
            MASS mass1 = joint[spring[i].m1];
            MASS mass2 = joint[spring[i].m2];
            //if (T < 1) {
            if (i == 18 or i == 37 or i == 17 or i == 27 or i == 29 or i == 39 or i == 46
or i == 48 or i == 54 or i == 56 or i == 62 or i == 64) {

            spring[i].L_0 = original[i] * (1 + walkgene[i].b * sin(500 * T +
walkgene[i].c));
            if (i > num_sp / 2) {
                spring[i].L_0 = original[i] * (1 + walkgene[i].b * cos(500 * T +
walkgene[i].c));
            }}
            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 = spring[i].k * fabs(new_L - L_0);
            springenergy += k * pow((new_L - L_0), 2) / 2;
            //cout <<i<<"---new_L---" <<new_L << endl;
            double norm_pd[3] = { pd[0] / new_L, pd[1] / new_L, pd[2] / new_L };
            //cout << i << "---force---" << force << endl;
            //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 << T << "tan " << springenergy << endl;
        for (int i = 0; i < 13; i++) {
            if (joint[i].p[2] <= 0) {
                cForces[i][2] -= Nground * joint[i].p[2];
                //cForces[i][2] -= joint[i].m*0.981;
                groundenergy += Nground * pow(joint[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;
                    joint[i].v[0] = 0;
                    joint[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;
                }
            }
            //cout << T << " " << groundenergy << endl;
            for (int j = 0; j < 3; j++) {
                joint[i].a[j] = cForces[i][j] / joint[i].m;
                joint[i].v[j] += joint[i].a[j] * timeStep;
                joint[i].v[j] *= dampening;
                joint[i].p[j] += joint[i].v[j] * timeStep;
            }
            //cout << cube[i].p[j] << endl;
        //gravityenergy
            gravityenergy += joint[i].m * gravity * joint[i].p[2];
            //kinetic
            double  norm_v  =  sqrt(pow(joint[i].v[0],  2)  +  pow(joint[i].v[1],  2)  +
pow(joint[i].v[2], 2));
            kineticenergy += joint[i].m * pow(norm_v, 2) / 2;
```

```
            //          cout << i << joint[i].p[2] << endl;
            //          cout << cForces[1][2] << endl;
        }
        //cout << "hhhhh"<< endl;

        //cout << "wtfh" << endl;

        drawground();

        T = T + timeStep;

    }

};

vector<vector<GENE>> populationGene;
vector<Robot> robot;
vector<double> totald2;
vector<vector<GENE>> newgenepop;
vector<vector<GENE>> nextpop;
vector<GENE> bestGene;
int populationsize = size;



vector<double> bestdistance;

//double maxdistance;
vector<vector<GENE>> bestgene;



    vector<GENE> crossover(vector<GENE> p1, vector<GENE> p2) {
        int crossposition = rand() % static_cast<int>(num_sp);
        vector<GENE> offspring = p2;
        //cout << "crossover" << endl;
        for (int i = 0; i < crossposition; i++) {
            offspring[i] = p1[i];
        }

        return offspring;
    }
    vector<vector<GENE>> genereateGene() {
        vector<vector<GENE>> populationGene1;
        for (int i = 0; i < populationsize; i++) {
```

```
            vector<GENE> temp;
            GENE temp_2;
            double k_1 = 9000;
            double b_1 = 0;
            double c_1 = 0;
            for (int j = 0; j < num_sp; j++) {
                double k_1 = 100 + static_cast <float> (rand()) / (static_cast <float>
(RAND_MAX / (1000 - 100)));
                if (j == 18 or j == 37 or j == 17 or j == 27 or j == 29 or j == 39 or j ==
46 or j == 48 or j == 54 or j == 56 or j == 62 or j == 64) {
                    k_1  =  600  +  static_cast  <float>  (rand())  /  (static_cast  <float>
(RAND_MAX / (5000 - 600)));
                    b_1 = static_cast <float> (rand()) / static_cast <float> (RAND_MAX /
1);
                    c_1 = -2 * M_PI + static_cast <float> (rand()) / (static_cast <float>
(RAND_MAX / (4 * M_PI)));
                    }
                else{
                    k_1 = 9000;
                    b_1 = 0;
                    c_1 = 0;
                }
//

                temp_2 = { k_1, b_1, c_1 };
                temp.push_back(temp_2);
                //              tempVec.push_back(k1);
                //              tempVec.push_back(b1);
                //              tempVec.push_back(c1);

            }
            populationGene1.push_back(temp);



        }
        return populationGene1;
    }
    vector<vector<GENE>>  selection(vector<vector<GENE>>  populationGene,vector<double>
totald) {
        vector<vector<GENE>> newgenepop;
        vector<int> index;
        for (int i = 0; i < populationsize; i++) {
            double temp_min = totald[0];
            int pointer = 0;
```

```
        for (int j = i; j < populationsize; j++) {
            if (totald[j] > temp_min) {
                temp_min = totald[j];
                pointer = j;
            }

        }
        index.push_back(pointer);
    }

    newgenepop.clear();
    newgenepop.shrink_to_fit();
    for (int i = 0; i < index.size() / 2; i++) {
        newgenepop.push_back(populationGene[index[i]]);
    }
    bestgene.push_back(populationGene[index[0]]);
    bestdistance.push_back(totald[index[0]]);
    cout<<"best distance"<< totald[index[0]]<<endl;
    evaluation++;
//
//      outFile1<<evaluation<<" "<< totald[index[0]]<<endl;

    cout<<"evaluation: "<<evaluation<<endl;
//      cout<<"worst"<<totald[index[-1]]<<endl;
    //      for (int i=0;i<newgenepop.size(); i++){
    //              bestGene << newgenepop[i].k << " " <<newgenepop[i].b<< " "
<<newgenepop[i].c << " ";
    //      }
    //      bestGene << "\n"
    return newgenepop;
  }
vector<GENE> mutation(vector<GENE> p1) {
  vector<int> list={18,37, 17,27,29,39,46,48,54,56,62,64};
  int i = rand() % static_cast<int>(12);
  i = list[i];
  vector<GENE> offspring = p1;
  GENE temp;
  temp.k = 600 + static_cast <float> (rand()) / (static_cast <float> (RAND_MAX / (1000
- 100)));
  temp.b = static_cast <float> (rand()) / static_cast <float> (RAND_MAX / 1);
  temp.c = -2 * M_PI + static_cast <float> (rand()) / (static_cast <float> (RAND_MAX
/ (4 * M_PI)));

  offspring[i] = temp;
```

```
    return offspring;
}
    vector<vector<GENE>> nextgeneration(vector<vector<GENE>> newgenepop) {
        vector<vector<GENE>> nextpop = newgenepop;
        for (int a = 0; a<int(newgenepop.size()); a++) {
            int i = rand() % static_cast<int>(newgenepop.size());
            int j = rand() % static_cast<int>(newgenepop.size());

            vector<GENE> p1 = newgenepop[i];
            vector<GENE> p2 = newgenepop[j];
            vector<GENE> off = crossover(p1, p2);

            nextpop.push_back(off);

        }

        double r = static_cast <float> (rand()) / (static_cast <float> (RAND_MAX));
        if (r < 0.5) {
            vector<GENE> temp_off;
            int ra_ro = 1 + rand() % static_cast<int>(nextpop.size()-1);
            temp_off = mutation(nextpop[ra_ro]);
            nextpop[ra_ro] = temp_off;
            cout<<"mutationing"<<endl;



        }
//        for(int i =0; i<10; i++){
//                          outFile2<<evaluation<<"  "<<i  <<"  "<<  nextpop[0][i].k<<"
"<<nextpop[0][i].b<<" "<<nextpop[0][i].c<<endl;
//        }
        return nextpop;
    }



    vector<Robot> getrobot(vector<Robot> robots, vector<vector<GENE>> nextpop) {
        for (int i = 0; i < populationsize; i++) {

            //double x = -2 + static_cast <float> (rand()) / (static_cast <float> (RAND_MAX
/ 4));
            double y = -1 + static_cast <float> (rand()) / (static_cast <float> (RAND_MAX
/ 2));
            //robots.push_back(ROBOT(0.0,         3.0*(i-populationSize/2),         0.0,
populationGene[i]));
```

```
            Robot robot1(0, y, 0.2, nextpop[i]);
            robots.push_back(robot1);

        }
        return robots;
    }
//    void move() {
//        for (int i = 0; i < populationsize; i++) {
//            robots[i].simulate();
//            //robots[i].drawplain();
//        }
//
//    }
//    void draw() {
//        for (int i = 0; i < populationsize; i++) {
//            robots[i].drawcube();
//
//        }
//
//    }
    double fittness(Robot r) {
        double fit = 0;
        double startpoint;
        double sumdis = 0;
        for (int i = 0; i < 13; i++) {
            fit += (r.joint[i].p[0] - r.initl[0]);
            sumdis += r.joint[i].p[0] - r.initl[0];
        }
        startpoint = sumdis/13;
        r.initl[0] = startpoint;
//        outFile3
//
//
//        << evaluation << " " << fit/13 << endl;
        //cout<<"startpoint"<<startpoint<<endl;
        return fit / 13;
    }


    vector<double> totaldistance(vector<Robot> robots) {
        vector<double> totald;
        for (int i = 0; i < populationsize; i++) {
            double f = fittness(robots[i]);
            totald.push_back(f);
```

```
        }

        return totald;
    }

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++);
}

/*
 *  OpenGL (GLUT) calls this routine to display the scene
 */


void display()
{
    //drawground();
    double len = 0.2;  //  Length of axes
    //  Erase the window and the de5th 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();
    //loop
    //for (int i = 0; i < generation; i++) {
       if (T<6) {

          for (int j = 0; j<populationsize; j++){

          robot[j].simulate();
          if (j == 5){
             robot[j].drawcube();}
          }

        }
       else {
          totald2 = totaldistance(robot);
          newgenepop = selection(populationGene,totald2);
          nextpop = nextgeneration(newgenepop);
          populationGene = nextpop;
          //cout<<"in loop evolve"<<endl;
          T = 0;
       }


    //Draw axes
    //glColor3f(1, 0, 0);
//   if (axes)
//   {
//      glBegin(GL_LINES);
//      glLineWidth(2);
//      glVertex3d(0.0, 0.0, 0.0);
//      glVertex3d(len, 0.0, 0.0);
//      glVertex3d(0.0, 0.0, 0.0);
//      glVertex3d(0.0, len, 0.0);
//      glVertex3d(0.0, 0.0, 0.0);
//      glVertex3d(0.0, 0.0, len);
//      glEnd();
//      //  Label axes
//      glRasterPos3d(len, 0.0, 0.0);
//      Print("X");
//      glRasterPos3d(0.0, len, 0.0);
```

```
//       Print("Y");
//       glRasterPos3d(0.0, 0.0, len);
//       Print("Z");
//    }
   //  Render the scene
   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[])
{

    populationGene =  genereateGene();
    robot = getrobot(robot, populationGene);

    // 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("evolve");
    //  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;



};
```