/* Uncomment the following to aim the blue ball at (X_OFFSET, 0.0) rather than the origin. */
#define OFFSET
#define X_OFFSET 10.0

/* Uncomment the following to make the blue ball be stationary at the (X_STATIONARY, Y_STATIONARY). This option has priority over OFFSET. */
/* #define STATIONARY */
#define X_STATIONARY 10.0
#define Y_STATIONARY 0.0

/* Some basic constants. MAX_BALLS is rather meaningless at this point. */
/* ESC is the ASCII value of the Esc key. ELASTICITY is used to define the elasticity of collisions. It may range between 1.0 (completely elastic) to 0.0 (completely inelastic). VELOCITY_SCALE is used to scale velocity to a reasonable value on fast machines. */
/* */
#define MAX_BALLS 2
#define PI 3.14159265
#define ESC 0x1b
#define ELASTICITY 1.0
#define VELOCITY_SCALE 0.33

#include <GL/glut.h>
#include <stdlib.h>
#include <math.h>
#include <sys/types.h>
#include <time.h>

/* Basic data structures for the simulation. We would be better off doing this in C++ and using proper classes. */
*/
typedef struct Color {
    GLdouble r, g, b;
} Color;

typedef struct Vector2 {
    GLdouble x, y;
} Vector2;
typedef struct Ball {
    Vector2 position;  /* Ok, so it’s not really a vector.  Sue me. */
    Vector2 velocity;
    GLdouble radius;
    GLdouble mass;
    Color color;
    GLuint handle;
} Ball;

double distanceSquared(double x, double y);
Vector2 scalarProduct(double s, Vector2 v);
double vectorLength(Vector2 v);
double dotProduct(Vector2 a, Vector2 b);
Vector2 normalize(Vector2 v);
Vector2 negateVector(Vector2 v);
Vector2 vectorSum(Vector2 a, Vector2 b);
Vector2 vectorDifference(Vector2 a, Vector2 b);

int collision(Ball ball1, Ball ball2);
void collisionResponse(Ball* ball1, Ball* ball2);
void placeBalls(void);
void initBalls(void);

void display(void);
void init(void);
void reshape(int w, int h);
void idle(void);
void keyboard(unsigned char key, int x, int y);

Ball balls[MAX_BALLS];

/* Definitions for basic vector operations. */
double distanceSquared(double x, double y)
{
    return x * x + y * y;
}

Vector2 scalarProduct(double s, Vector2 v)
{
    v.x *= s;
    v.y *= s;
    return v;
}

double vectorLength(Vector2 v)
{
    return sqrt(distanceSquared(v.x, v.y));
}

double dotProduct(Vector2 a, Vector2 b)
{
    return a.x * b.x + a.y * b.y;
}

Vector2 normalize(Vector2 v)
{
    double length = vectorLength(v);
    v.x /= length;
    v.y /= length;
    return v;
}

Vector2 negateVector(Vector2 v)
{
    v.x = -v.x;
    v.y = -v.y;
    return v;
}

Vector2 vectorSum(Vector2 a, Vector2 b)
{
    a.x += b.x;
    a.y += b.y;
    return a;
}
190: Vector2 vectorDifference(Vector2 a, Vector2 b) {
191:    a.x -= b.x;
192:    a.y -= b.y;
193:    return a;
194: }
195:
196: /***********************************************************************
197: * Collision detection and response functions.
198: ************************************************************************/
199: int collision(Ball ball1, Ball ball2) {
200:    double radiusSum = ball1.radius + ball2.radius;
201:    /* Vector from center of ball2 to center of ball1. This vector is */
202:    /* normal to the collision plane. */
203:    Vector2 collisionNormal = vectorDifference(ball1.position, ball2.position);
204:    /* Note that we're comparing square of distance, to avoid computing */
205:    /* square roots. We've had a collision if the distance between */
206:    /* the centers of the balls is <= to the sum of their radii. */
207:    return (distanceSquared(collisionNormal.x, collisionNormal.y) <= radiusSum * radiusSum) ? 1 : 0;
208: }
209:
210: /***********************************************************************
211: * We may have to make modifications to ball1 and ball2, so we need to
212: * pass in pointers to them. This function will determine the response
213: * (result) to the collision.
214: ***********************************************************************/
215: void collisionResponse(Ball* ball1, Ball* ball2) {
216:    double radiusSum = ball1->radius + ball2->radius;
217:    /* Vector from center of ball2 to center of ball1. This vector is */
218:    /* normal to the collision plane. */
219:    Vector2 collisionNormal = vectorDifference(ball1->position, ball2->position);
220:    /* Penetration distance is sum of radii less distance between centers */
221:    /* of the two balls. */
222:    double distance = sqrt(distanceSquared(collisionNormal.x, collisionNormal.y));
223:    double penetration = radiusSum - distance;
224:    Vector2 relativeVelocity = vectorDifference(ball2->velocity, ball1->velocity);
225:    /* Dot product of relative velocity and collision normal. If this
* is negative, the balls are already moving apart, and we need not
* compute a collision response.
*/

double vDOTn;

/* The following are used to compute the collision impulse. This is
* energy added to each ball to draw them apart following the collision.
* The total energy in the system remains the same, or is less than
* before the collision if the collision is inelastic.
*/
double numerator;
double denominator;
double impulse;
collisionNormal = normalize(collisionNormal);

/* Readjust ball position by translating each ball by 1/2 the
* penetration distance along the collision normal.
*/

ball1->position = vectorSum(ball1->position,
   scalarProduct(0.5 * penetration,
                     collisionNormal));

ball2->position = vectorDifference(ball2->position,
   scalarProduct(0.5 * penetration,
                    collisionNormal));
vDOTn = dotProduct(relativeVelocity, collisionNormal);

if (vDOTn < 0.0)
   return;

/* Compute impulse energy. */

numerator = -(1.0 + ELASTICITY) * vDOTn;
denominator = (1.0 / ball2->mass + 1.0 / ball1->mass);
impulse = numerator / denominator;

/* Apply the impulse to each ball. */

ball2->velocity = vectorSum(ball2->velocity,
   scalarProduct(impulse / ball2->mass,
                 collisionNormal));

ball1->velocity = vectorDifference(ball1->velocity,
   scalarProduct(impulse / ball1->mass,
                 collisionNormal));

/**********************************************************************
 * Assign initial positions for the balls. This is done as follows.
 * For the first ball, assign it a random position about the unit
 * circle. Convert this to Cartesian coordinates (x, y). This position,
 * when looked at as a vector, has length 1.0. The vector (-x, -y) then
 * can be used as a normalized velocity vector pointing toward the origin,
 * our default collision point. Then, scaling the position vector by 40.0
 * translates the ball out to the corresponding point along the circle
 **********************************************************************/
A similar algorithm is used to place the second ball, with one slight difference: We don’t want the balls to overlap when we start out. To avoid this, we compute the second ball's position as an offset to the first ball’s position. The range of this offset is (PI/4.0) to (7.0*PI/4.0). Thus, the two balls are at least (PI/4.0) radians away from each other.

This code was factored out of initBalls() so that we could call it each time the two balls leave the clipping rectangle. initBalls() need only be called once, at the beginning of the simulation.

void placeBalls(void)
{
    double angle;

    /* Compute position and velocity of first ball. */
    /* (double) rand() / (double) RAND_MAX will give us a random double value on the closed interval [0.0, 1.0]. */
    angle = 2.0 * PI * (double) rand() / (double) RAND_MAX;
    balls[0].position.x = cos(angle);
    balls[0].position.y = sin(angle);
    balls[0].velocity.x = -balls[0].position.x;
    balls[0].velocity.y = -balls[0].position.y;
    balls[0].velocity = scalarProduct(VELOCITY_SCALE, balls[0].velocity);
    balls[0].position = scalarProduct(40.0, balls[0].position);

    /* Compute position and velocity of second ball. */
    angle += PI / 4.0 + 1.5 * PI * (double) rand() / (double) RAND_MAX;
    balls[1].position.x = cos(angle);
    balls[1].position.y = sin(angle);

    #ifndef OFFSET
    /* Aim second ball so that it passes through (0.0, 0.0). */
    balls[1].velocity.x = -balls[1].position.x;
    balls[1].velocity.y = -balls[1].position.y;
    balls[1].position = scalarProduct(40.0, balls[1].position);
    #else
    /* Aim second ball so that it passes through (X_OFFSET, 0.0). */
    balls[1].position = scalarProduct(40.0, balls[1].position);
    balls[1].velocity.x = X_OFFSET - balls[1].position.x;
    balls[1].velocity.y = -balls[1].position.y;
    /* The velocity vector, as computed, isn’t normalized. Let’s normalize it. */
    balls[1].velocity = normalize(balls[1].velocity);
    #endif

    #ifdef STATIONARY
    /* Place the second ball at a stationary position defined by (X_STATIONARY, Y_STATIONARY). */
    balls[1].position.x = X_STATIONARY;
    balls[1].position.y = Y_STATIONARY;
    balls[1].velocity.x = balls[1].velocity.y = 0.0;
    #endif
}
# Assign initial attributes to the two balls. This data should really be read from a file.

```c
void initBalls(void)
{
    int i;
    GLUquadricObj *qobj; /* Need this to generate the spheres (disks). */

    /* Compute starting positions and velocities for the balls. */
    placeBalls();

    balls[0].radius = 7.0;
    balls[0].mass = 2.0;
    balls[0].color.r = 1.0;
    balls[0].color.g = 0.0;
    balls[0].color.b = 0.0;

    balls[1].radius = 5.0;
    balls[1].mass = 1.0;
    balls[1].color.r = 0.0;
    balls[1].color.g = 0.0;
    balls[1].color.b = 1.0;

    /* Create display lists for each ball. */
    for (i = 0; i < MAX_BALLS; ++i)
    {
        balls[i].handle = glGenLists(1);
        qobj = gluNewQuadric();
        glNewList(balls[i].handle, GL_COMPILE);
        gluDisk(qobj, 0.0, balls[i].radius, 72, 1);
        glEndList();
    }
}
```

# OpenGL functions.

```c
void display(void)
{
    int i;
    glClear(GL_COLOR_BUFFER_BIT);

    /* Render balls. */
    for (i = 0; i < MAX_BALLS; ++i)
    {
        glColor3f(balls[i].color.r, balls[i].color.g, balls[i].color.b);
        ```
glPushMatrix();
glTranslatef(balls[i].position.x, balls[i].position.y, 0.0);
glCallList(balls[i].handle);
glPopMatrix();

/* Swap buffers, for smooth animation. This will also flush the */
* pipeline. */
glutSwapBuffers();
}

void init (void)
{
    glClearColor (0.0, 0.0, 0.0, 0.0);
    glShadeModel (GL_FLAT);   /* Probably unnecessary. */
    initBalls();
}

void keyboard(unsigned char key, int x, int y)
{
    switch (key)
    {
      case ESC:
          exit(0);
          break;
    }
}

void reshape(int w, int h)
{
    /* Probably needs to be fixed. */
    glViewport (0, 0, (GLsizei) w, (GLsizei) h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    glOrtho(-50.0, 50.0, -50.0, 50.0, -1.0, 1.0);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
}

/***********************************************************************/
/* Hitting the Esc key will exit the program. */
/***********************************************************************/
void keyboard(unsigned char key, int x, int y)
void idle(void)
{
    int i;

    /* Update positions. */
    for (i = 0; i < MAX_BALLS; ++i)
    {
        balls[i].position.x += balls[i].velocity.x;
        balls[i].position.y += balls[i].velocity.y;
    }

    /* Check for collisions and act. */
    if (collision(balls[0], balls[1]))
        collisionResponse(&balls[0], &balls[1]);

    /* For efficiency, do not compute square roots. This is checking to
     * see if either ball is outside the circle of radius 50.0.
     */
    if (distanceSquared(balls[0].position.x, balls[0].position.y) > 2500.0
        || distanceSquared(balls[1].position.x, balls[1].position.y) > 2500.0)
        placeBalls();

    /* Re-render the scene. */
    glutPostRedisplay();
}

int main(int argc, char** argv)
{
    srand((unsigned int) time(NULL));
glutInit (&argc, argv);
glutInitDisplayMode (GLUT_DOUBLE | GLUT_RGB);
glutInitWindowSize (500, 500);
glutInitWindowPosition (100, 100);
glutCreateWindow ("Colliding balls");
init();
glutDisplayFunc(display);
glutReshapeFunc(reshape);
glutKeyboardFunc(keyboard);
glutIdleFunc(idle);
glutMainLoop();
return 0;"