This is a simple double buffered program that demonstrates double buffering and animation. More importantly, it demonstrates collision detection and response for spheres (here, 2-D balls). This works fine assuming we don't have "too many" collisions at one time.

The following to aim the blue ball at (x_OFFSET, 0.0) rather than the origin.

#undef OFFSET
#define X_OFFSET 10.0

Use the following to make the blue ball be stationary at the (x_STATIONARY, y_STATIONARY). This option has priority over OFFSET.
#undef 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>

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;

Prototypes for basic vector operations. These should really be methods associated with some classes. In particular, it would be nice to be
81: * overloading operators so that we don't have so many nested function calls
82: * later.
83: ******************************************************************************
84: double distanceSquared(double x, double y);
85: Vector2 scalarProduct(double s, Vector2 v);
86: double vectorLength(Vector2 v);
87: double dotProduct(Vector2 a, Vector2 b);
88: Vector2 normalize(Vector2 v);
89: Vector2 negateVector(Vector2 v);
90: Vector2 vectorSum(Vector2 a, Vector2 b);
91: Vector2 vectorDifference(Vector2 a, Vector2 b);
92: 
93: /******************************************************************************
94: * Prototypes for collision detection and response, and for setting
95: * attributes of the simulation objects.
96: ******************************************************************************
97: int collision(Ball ball1, Ball ball2);
98: void collisionResponse(Ball* ball1, Ball* ball2);
99: void placeBalls(void);
100: void initBalls(void);
101: 
102: /******************************************************************************
103: * Prototypes for the basic OpenGL functions.
104: ******************************************************************************
105: void display(void);
106: void init(void);
107: void reshape(int w, int h);
108: void idle(void);
109: void keyboard(unsigned char key, int x, int y);
110: 
111: /******************************************************************************
112: * Data structure for holding the simulation objects. */
113: Ball balls[MAX_BALLS];
114: 
115: /******************************************************************************
116: * Definitions for basic vector operations.
117: ******************************************************************************
118: double distanceSquared(double x, double y)
119: { return x * x + y * y; }
120: 
121: Vector2 scalarProduct(double s, Vector2 v)
122: { v.x *= s;
123: v.y *= s;
124: return v;
125: }
126: 
127: double vectorLength(Vector2 v)
128: { return sqrt(distanceSquared(v.x, v.y)); }
129: 
130: double dotProduct(Vector2 a, Vector2 b)
131: { return a.x * b.x + a.y * b.y; }
132: 
133: Vector2 normalize(Vector2 v)
134: { double length = vectorLength(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;
}

Vector2 vectorDifference(Vector2 a, Vector2 b)
{
    a.x -= b.x;
    a.y -= b.y;
    return a;
}

int collision(Ball ball1, Ball ball2)
{
    double radiusSum = ball1.radius + ball2.radius;

    Vector2 collisionNormal = vectorDifference(ball1.position, ball2.position);

    return (distanceSquared(collisionNormal.x, collisionNormal.y) <= radiusSum * radiusSum) ? 1 : 0;
}

void collisionResponse(Ball* ball1, Ball* ball2)
{
    double radiusSum = ball1->radius + ball2->radius;

    Vector2 collisionNormal = vectorDifference(ball1->position, ball2->position);

    return (distanceSquared(collisionNormal.x, collisionNormal.y) <= radiusSum * radiusSum) ? 1 : 0;
}
double distance = sqrt(distanceSquared(collisionNormal.x, collisionNormal.y));
double penetration = radiusSum - distance;

Vector2 relativeVelocity = vectorDifference(ball2->velocity, ball1->velocity);
/* 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 *
* with radius 40.0. *
* 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;

  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);
    balls[1].velocity = scalarProduct(VELOCITY_SCALE, 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
}

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;
401:     balls[1].mass = 1.0;
402:     balls[1].color.r = 0.0;
403:     balls[1].color.g = 0.0;
404:     balls[1].color.b = 1.0;
405: 
406:     /* Create display lists for each ball. */
407: 
408:     for (i = 0; i < MAX_BALLS; ++i)
409:     {
410:         balls[i].handle = glGenLists(1);
411:         qobj = gluNewQuadric();
412:         glNewList(balls[i].handle, GL_COMPILE);
413:             gluDisk(qobj, 0.0, balls[i].radius, 72, 1);
414:         glEndList();
415:     }
416: }

419: /* Open GL functions.
420:  *-----------------------------------------------------------------
421:  *
422:  */
423:  *-----------------------------------------------------------------
424:  *
425:  * Recall, this will do our rendering for us. It is called following
426:  * each simulation step in order to update the window.
427:  *-----------------------------------------------------------------
428: void display(void)
429: {
430:     int i;
431:     
432:     glClear(GL_COLOR_BUFFER_BIT);
433:     
434:     /* Render balls. */
435:     for (i = 0; i < MAX_BALLS; ++i)
436:     {
437:         glColor3f(balls[i].color.r, balls[i].color.g, balls[i].color.b);
438:         glPushMatrix();
439:         glTranslatef(balls[i].position.x, balls[i].position.y, 0.0);
440:         glCallList(balls[i].handle);
441:         glPopMatrix();
442:     }
443:     
444:     /* Swap buffers, for smooth animation. This will also flush the
445:      * pipeline.
446:      */
447:     glutSwapBuffers();
448: }

455:     initBalls();
456: }

463:      *
464:      *-----------------------------------------------------------------
465:      *
466: void keyboard(unsigned char key, int x, int y)
467: {
468:     switch (key)
469:     {
470:         case ESC:
471:             exit(0);
472:             break;
473:     }
474: }

487:     initBalls();
488: }

531:     glBegin(GL_TRIANGLES);
532:         glVertex3f(balls[i].x, balls[i].y, 0.0);
533:         glVertex3f(balls[i].x + 100, balls[i].y, 0.0);
534:         glVertex3f(balls[i].x + 100, balls[i].y + 100, 0.0);
535:     glEnd();
536: }

552:     glTranslatef(0.0, 0.0, -i * 100.0);
553:     glCallList(balls[1].handle);
554:     
555:     glTranslatef(0.0, 0.0, (i - 1) * 100.0);
556:     glCallList(balls[1].handle);
557: 
560:     /***************************************************************
561:     * Set the basic world coordinates to screen coordinates mapping.
562:     ***************************************************************
563: void setViewPort(void)
564: {
565:     glMatrixMode(GL_MODELVIEW);
566:     glLoadIdentity();
567: 
568:     // Set up projection and viewport
569:     projection = gluPerspective(45.0, 1.0, 0.1, 1000.0);
570:     
571:     gluLookAt(50.0, 50.0, 50.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);
572: }
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();
}

/***************************************************************************/
/* This computes a simulation step. Updated ball positions are computed */
/* using each ball’s velocity. Then, we check to see if the balls have */
/* collided. If so, we compute the response. Finally, we see if either */
/* ball is leaving the clipping region. If so, we call placeBalls() to */
/* re-start the simulation. */
/***************************************************************************/

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();
}

/***************************************************************************/
/* Request double buffer display mode for smooth animation. */
/***************************************************************************/

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;
}