```
/*************************************************************************
 *  Tom Kelliher, Goucher College
 *  Mar. 2, 2011
 *  collision.c
 *
 *  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.
 *************************************************************************/


/* 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 <stdio.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;


/* This should really be extended to three dimensions */

typedef struct Vector2
{
   GLdouble x, y;
```

```
} Vector2;


/* Most of these are self-explanatory.  handle is the display list necessary
 * for rendering a ball.
 */

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
 * overloading operators so that we don't have so many nested function calls
 * later.
 ************************************************************************/

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);


/*************************************************************************
 * Prototypes for collision detection and response, and for setting
 * attributes of the simulation objects.
 ************************************************************************/

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


/*************************************************************************
 * Prototypes for the basic OpenGL functions.
 ************************************************************************/

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


/* Data structure for holding the simulation objects. */

Ball balls[MAX_BALLS];


/*************************************************************************
```

```
 127:  * Definitions for basic vector operations.
 128:  ************************************************************************/
 129:
 130: /************************************************************************
 131:  * We use distanceSquared() wherever we can to avoid computing a square
 132:  * root (expensive).
 133:  ************************************************************************/
 134:
 135: double distanceSquared(double x, double y)
 136: {
 137:    return x * x + y * y;
 138: }
 139:
 140:
 141: Vector2 scalarProduct(double s, Vector2 v)
 142: {
 143:    v.x *= s;
 144:    v.y *= s;
 145:
 146:    return v;
 147: }
 148:
 149:
 150: double vectorLength(Vector2 v)
 151: {
 152:    return sqrt(distanceSquared(v.x, v.y));
 153: }
 154:
 155:
 156: double dotProduct(Vector2 a, Vector2 b)
 157: {
 158:    return a.x * b.x + a.y * b.y;
 159: }
 160:
 161:
 162: Vector2 normalize(Vector2 v)
 163: {
 164:    double length = vectorLength(v);
 165:
 166:    v.x /= length;
 167:    v.y /= length;
 168:
 169:    return v;
 170: }
 171:
 172:
 173: Vector2 negateVector(Vector2 v)
 174: {
 175:    v.x = -v.x;
 176:    v.y = -v.y;
 177:
 178:    return v;
 179: }
 180:
 181:
 182: Vector2 vectorSum(Vector2 a, Vector2 b)
 183: {
 184:    a.x += b.x;
 185:    a.y += b.y;
 186:
 187:    return a;
 188: }
 189:
```

```
  190:
  191: Vector2 vectorDifference(Vector2 a, Vector2 b)
  192: {
  193:    a.x -= b.x;
  194:    a.y -= b.y;
  195:
  196:    return a;
  197: }
  198:
  199:
  200: /**************************************************************************
  201:  * Collision detection and response functions.
  202:  **************************************************************************/
  203:
  204: int collision(Ball ball1, Ball ball2)
  205: {
  206:    double radiusSum = ball1.radius + ball2.radius;
  207:
  208:    /* Vector from center of ball2 to center of ball1.  This vector is
  209:     * normal to the collision plane.
  210:     */
  211:
  212:    Vector2 collisionNormal = vectorDifference(ball1.position,
  213:                                               ball2.position);
  214:
  215:    /* Note that we're comparing square of distance, to avoid computing
  216:     * square roots.  We've had a collision if the distance between
  217:     * the centers of the balls is <= to the sum of their radii.
  218:     */
  219:
  220:    return (distanceSquared(collisionNormal.x, collisionNormal.y)
  221:            <= radiusSum * radiusSum)
  222:      ? 1 : 0;
  223: }
  224:
  225:
  226: /**************************************************************************
  227:  * We may have to make modifications to ball1 and ball2, so we need to
  228:  * pass in pointers to them.  This function will determine the response
  229:  * (result) to the collision.
  230:  **************************************************************************/
  231:
  232: void collisionResponse(Ball* ball1, Ball* ball2)
  233: {
  234:    double radiusSum = ball1->radius + ball2->radius;
  235:
  236:    /* Vector from center of ball2 to center of ball1.  This vector is
  237:     * normal to the collision plane.
  238:     */
  239:
  240:    Vector2 collisionNormal = vectorDifference(ball1->position,
  241:                                               ball2->position);
  242:
  243:    /* Penetration distance is sum of radii less distance between centers
  244:     * of the two balls.
  245:     */
  246:
  247:    double distance = sqrt(distanceSquared(collisionNormal.x,
  248:                                           collisionNormal.y));
  249:    double penetration = radiusSum - distance;
  250:
  251:    Vector2 relativeVelocity = vectorDifference(ball2->velocity,
  252:                                                ball1->velocity);
```

```
   253:    /* Dot product of relative velocity and collision normal.  If this
   254:     * is negative, the balls are already moving apart, and we need not
   255:     * compute a collision response.
   256:     */
   257:
   258:    double vDOTn;
   259:
   260:    /* The following are used to compute the collision impulse.  This is
   261:     * energy added to each ball to draw them apart following the collision.
   262:     * The total energy in the system remains the same, or is less than
   263:     * before the collision if the collision is inelastic.
   264:     */
   265:
   266:    double numerator;
   267:    double denominator;
   268:    double impulse;
   269:
   270:    collisionNormal = normalize(collisionNormal);
   271:
   272:    /* Readjust ball position by translating each ball by 1/2 the
   273:     * penetration distance along the collision normal.
   274:     */
   275:
   276:    ball1->position
   277:       = vectorSum(ball1->position,
   278:                   scalarProduct(0.5 * penetration,
   279:                                 collisionNormal));
   280:
   281:    ball2->position
   282:       = vectorDifference(ball2->position,
   283:                          scalarProduct(0.5 * penetration,
   284:                                        collisionNormal));
   285:
   286:    vDOTn = dotProduct(relativeVelocity, collisionNormal);
   287:
   288:    if (vDOTn < 0.0)
   289:       return;
   290:
   291:    /* Compute impulse energy. */
   292:
   293:    numerator = -(1.0 + ELASTICITY) * vDOTn;
   294:    denominator = (1.0 / ball2->mass + 1.0 / ball1->mass);
   295:    impulse = numerator / denominator;
   296:
   297:    /* Apply the impulse to each ball. */
   298:
   299:    ball2->velocity = vectorSum(ball2->velocity,
   300:                                scalarProduct(impulse / ball2->mass,
   301:                                              collisionNormal));
   302:
   303:    ball1->velocity = vectorDifference(ball1->velocity,
   304:                                       scalarProduct(impulse / ball1->mass,
   305:                                                     collisionNormal));
   306: }
   307:
   308:
   309: /*************************************************************************
   310:  * Assign initial positions for the balls.  This is done as follows.
   311:  * For the first ball, assign it a random position about the unit
   312:  * circle.  Convert this to Cartesian coordinates (x, y).  This position,
   313:  * when looked at as a vector, has length 1.0.  The vector (-x, -y) then
   314:  * can be used as a normalized velocity vector pointing toward the origin,
   315:  * our default collision point.  Then, scaling the position vector by 40.0
```

```
316:  * translates the ball out to the corresponding point along the circle
317:  * with radius 40.0.
318:  *
319:  * A similar algorithm is used to place the second ball, with one slight
320:  * difference: We don't want the balls to overlap when we start out.  To
321:  * avoid this, we compute the second ball's position as an offset to the
322:  * first ball's position.  The range of this offset is (PI/4.0) to
323:  * (7.0*PI/4.0).  Thus, the two balls are at least (PI/4.0) radians away
324:  * from each other.
325:  *
326:  * This code was factored out of initBalls() so that we could call it each
327:  * time the two balls leave the clipping rectangle.  initBalls() need only
328:  * be called once, at the beginning of the simulation.
329:  * ********************************************************************/
330:
331: void placeBalls(void)
332: {
333:    double angle;
334:
335:    /* Compute position and velocity of first ball.
336:    /* (double) rand() / (double) RAND_MAX will give us a random double
337:     * value on the closed interval [0.0, 1.0].
338:     */
339:
340:    angle = 2.0 * PI * (double) rand() / (double) RAND_MAX;
341:    balls[0].position.x = cos(angle);
342:    balls[0].position.y = sin(angle);
343:    balls[0].velocity.x = -balls[0].position.x;
344:    balls[0].velocity.y = -balls[0].position.y;
345:    balls[0].velocity = scalarProduct(VELOCITY_SCALE, balls[0].velocity);
346:    balls[0].position = scalarProduct(40.0, balls[0].position);
347:
348:    /* Compute position and velocity of second ball. */
349:
350:    angle += PI / 4.0 + 1.5 * PI * (double) rand() / (double) RAND_MAX;
351:    balls[1].position.x = cos(angle);
352:    balls[1].position.y = sin(angle);
353:
354: #ifndef OFFSET
355:    /* Aim second ball so that it passes through (0.0, 0.0). */
356:    balls[1].velocity.x = -balls[1].position.x;
357:    balls[1].velocity.y = -balls[1].position.y;
358:    balls[1].position = scalarProduct(40.0, balls[1].position);
359: #else
360:    /* Aim second ball so that it passes through (X_OFFSET, 0.0). */
361:    balls[1].position = scalarProduct(40.0, balls[1].position);
362:    balls[1].velocity.x = X_OFFSET - balls[1].position.x;
363:    balls[1].velocity.y = -balls[1].position.y;
364:    /* The velocity vector, as computed, isn't normalized.  Let's normalize
365:     * it.
366:     */
367:    balls[1].velocity = normalize(balls[1].velocity);
368: #endif
369:
370:    balls[1].velocity = scalarProduct(VELOCITY_SCALE, balls[1].velocity);
371:
372: #ifdef STATIONARY
373:    /* Place the second ball at a stationary position defined by
374:     * (X_STATIONARY, Y_STATIONARY).
375:     */
376:    balls[1].position.x = X_STATIONARY;
377:    balls[1].position.y = Y_STATIONARY;
378:    balls[1].velocity.x = balls[1].velocity.y = 0.0;
```

```
379: #endif
380:
381: }
382:
383:
384: /**********************************************************************
385:  * Assign initial attributes to the two balls.  This data should really
386:  * be read from a file.
387:  **********************************************************************/
388:
389: void initBalls(void)
390: {
391:    int i;
392:    GLUquadricObj *qobj;  /* Need this to generate the spheres (disks). */
393:
394:    /* Compute starting positions and velocities for the balls. */
395:
396:    placeBalls();
397:
398:    balls[0].radius = 7.0;
399:    balls[0].mass = 2.0;
400:    balls[0].color.r = 1.0;
401:    balls[0].color.g = 0.0;
402:    balls[0].color.b = 0.0;
403:
404:    balls[1].radius = 5.0;
405:    balls[1].mass = 1.0;
406:    balls[1].color.r = 0.0;
407:    balls[1].color.g = 0.0;
408:    balls[1].color.b = 1.0;
409:
410:    /* Create display lists for each ball. */
411:
412:    for (i = 0; i < MAX_BALLS; ++i)
413:    {
414:       balls[i].handle = glGenLists(1);
415:       qobj = gluNewQuadric();
416:       glNewList(balls[i].handle, GL_COMPILE);
417:          gluDisk(qobj, 0.0, balls[i].radius, 72, 1);
418:       glEndList();
419:    }
420: }
421:
422:
423: /**********************************************************************
424:  * OpenGL functions.
425:  **********************************************************************/
426:
427: /**********************************************************************
428:  * Recall, this will do our rendering for us.  It is called following
429:  * each simulation step in order to update the window.
430:  **********************************************************************/
431:
432: void display(void)
433: {
434:    int i;
435:
436:    glClear(GL_COLOR_BUFFER_BIT);
437:
438:    /* Render balls. */
439:
440:    for (i = 0; i < MAX_BALLS; ++i)
441:    {
```

```
   442:       glColor3f(balls[i].color.r, balls[i].color.g, balls[i].color.b);
   443:       glPushMatrix();
   444:       glTranslatef(balls[i].position.x, balls[i].position.y, 0.0);
   445:       glCallList(balls[i].handle);
   446:       glPopMatrix();
   447:    }
   448:
   449:    /* Swap buffers, for smooth animation.  This will also flush the
   450:     * pipeline.
   451:     */
   452:
   453:    glutSwapBuffers();
   454: }
   455:
   456:
   457: void init(void)
   458: {
   459:    glClearColor (0.0, 0.0, 0.0, 0.0);
   460:    glShadeModel (GL_FLAT);   /* Probably unnecessary. */
   461:
   462:    initBalls();
   463: }
   464:
   465:
   466: /*************************************************************************
   467:  * Hitting the Esc key will exit the program.
   468:  *************************************************************************/
   469:
   470: void keyboard(unsigned char key, int x, int y)
   471: {
   472:    switch (key)
   473:    {
   474:       case ESC:
   475:          exit(0);
   476:          break;
   477:    }
   478: }
   479:
   480:
   481: /*************************************************************************
   482:  * Set the basic world coordinates to screen coordinates mapping.
   483:  *************************************************************************/
   484:
   485: void reshape(int w, int h)
   486: {
   487:    /* Probably needs to be fixed. */
   488:
   489:    glViewport (0, 0, (GLsizei) w, (GLsizei) h);
   490:    glMatrixMode(GL_PROJECTION);
   491:    glLoadIdentity();
   492:    glOrtho(-50.0, 50.0, -50.0, 50.0, -1.0, 1.0);
   493:    glMatrixMode(GL_MODELVIEW);
   494:    glLoadIdentity();
   495: }
   496:
   497:
   498: /*************************************************************************
   499:  * This computes a simulation step.  Updated ball positions are computed
   500:  * using each ball's velocity.  Then, we check to see if the balls have
   501:  * collided.  If so, we compute the response.  Finally, we see if either
   502:  * ball is leaving the clipping region.  If so, we call placeBalls() to
   503:  * re-start the simulation.
   504:  *************************************************************************/
```

```
505:
506: void idle(void)
507: {
508:    int i;
509:
510:    /* Update positions. */
511:
512:    for (i = 0; i < MAX_BALLS; ++i)
513:    {
514:       balls[i].position.x += balls[i].velocity.x;
515:       balls[i].position.y += balls[i].velocity.y;
516:    }
517:
518:    /* Check for collisions and act. */
519:
520:    if (collision(balls[0], balls[1]))
521:       collisionResponse(&balls[0], &balls[1]);
522:
523:    /* For efficiency, do not compute square roots.  This is checking to
524:     * see if either ball is outside the circle of radius 50.0.
525:     */
526:
527:    if (distanceSquared(balls[0].position.x, balls[0].position.y) > 2500.0
528:        || distanceSquared(balls[1].position.x, balls[1].position.y) > 2500.0)
529:       placeBalls();
530:
531:    /* Re-render the scene. */
532:
533:    glutPostRedisplay();
534: }
535:
536:
537: /*****************************************************************
538:  *  Request double buffer display mode for smooth animation.
539:  *****************************************************************/
540:
541: int main(int argc, char** argv)
542: {
543:    setvbuf(stdout, (char *)NULL, _IONBF, 0);
544:    setvbuf(stderr, (char *)NULL, _IONBF, 0);
545:
546:    srand((unsigned int) time(NULL));
547:    glutInit(&argc, argv);
548:    glutInitDisplayMode (GLUT_DOUBLE | GLUT_RGB);
549:    glutInitWindowSize (500, 500);
550:    glutInitWindowPosition (100, 100);
551:    glutCreateWindow ("Colliding balls");
552:    init();
553:    glutDisplayFunc(display);
554:    glutReshapeFunc(reshape);
555:    glutKeyboardFunc(keyboard);
556:    glutIdleFunc(idle);
557:    glutMainLoop();
558:
559:    return 0;
560: }
```