```
/**********************************************************************
 *  Tom Kelliher, Goucher College
 *  Feb. 15, 2013
 *  collision.cpp
 *
 *  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 for double buffering.  If single buffering,
 * VELOCITY_SCALE will be need to be significantly reduced (start by
 * shrinking it by a factor of 100).
 */

#define DOUBLE_BUFFER


/* Uncomment the following to handle window reshape events */

#define RESHAPE


/* Uncomment the following to use a viewport to correct the aspect ratio
 * during reshape events.  Otherwise, the clipping region will be
 * reshaped to match the window's aspect ratio.
 */

//#define VIEWPORT


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

#define OFFSET
const float 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
const float X_STATIONARY = 10.0;
const float 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.  SLICES is the number
 * of vertices to generate for rendering a ball, which is rendered as a
 * circle.
 */

const int MAX_BALLS = 2;
const int ESC = 0x1b;
const float ELASTICITY = 1.0;
const float VELOCITY_SCALE = 1.0;
```

```
const int SLICES = 72;


#include <time.h>
#include "Angel.h"


/* Identifiers for the shader programs and the uniform projection and model
 * view matrices in the vertex shader (see vshader41.glsl).
 */

GLuint program;
GLuint projection;
GLuint model_view;


/* Basic data structures for the simulation. */

typedef vec3 Color;


/* Most of these are self-explanatory.  vao is the indentifier for the vertex
 * array object holding the vertex attributes for this ball.  numVertices
 * is the number of vertices represented within the VAO.  geometry is the
 * geometry (GL_LINES, GL_TRIANGLES, etc.) to use when drawing the VAO.
 */

typedef struct Ball
{
   vec2 position;
   vec2 velocity;
   GLdouble radius;
   GLdouble mass;
   Color color;
   GLuint vao;
   GLuint numVertices;
   GLuint geometry;
} Ball;


/* Initial window width and height */

GLuint windowWidth = 500;
GLuint windowHeight = 500;

/* We use a 1:1 aspect ratio.  For convenience in setting the projection
 * matrix, WORLD_HALF is half the clipping region’s width/height.  Refer
 * to display().
 */

const GLfloat WORLD_HALF = 50.0;
GLfloat worldRight = WORLD_HALF;
GLfloat worldTop = WORLD_HALF;


/*********************************************************************
 * Prototypes for basic vector operations not already defined in vec.h.
 *********************************************************************/

GLfloat distanceSquared(vec2 v);


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

```
 * 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);
GLuint createCircle(Ball ball);

/***********************************************************************
 * 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];


/***********************************************************************
 * Definitions for basic vector operations.
 ***********************************************************************/


/***********************************************************************
 * We use distanceSquared() wherever we can to avoid computing a square
 * root (expensive).
 ***********************************************************************/

GLfloat distanceSquared(vec2 v)
{
   return dot(v, v);
}


/***********************************************************************
 * Collision detection and response functions.
 ***********************************************************************/

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

   /* Vector from center of ball2 to center of ball1.  This vector is
    * normal to the collision plane.
    */

   vec2 collisionNormal = ball1.position - ball2.position;

   /* Note that we’re comparing square of distance, to avoid computing
    * square roots.  We’ve had a collision if the distance between
    * the centers of the balls is <= to the sum of their radii.
    */

   return (distanceSquared(collisionNormal) <= radiusSum * radiusSum)
      ? 1 : 0;
```

```cpp
}


/***********************************************************************
 * We may have to make modifications to ball1 and ball2, so we need to
 * pass in references to them.  This function will determine the
 * response to the collision and modify each ball’s position and
 * velocity vector to account for the collison response.
 ***********************************************************************/

void collisionResponse(Ball& ball1, Ball& ball2)
{
   double radiusSum = ball1.radius + ball2.radius;

   /* Vector from center of ball2 to center of ball1.  This vector is
    * normal to the collision plane.
    */

   vec2 collisionNormal = ball1.position - ball2.position;

   /* Penetration distance is sum of radii less distance between centers
    * of the two balls.
    */

   double distance = length(collisionNormal);
   double penetration = radiusSum - distance;

   vec2 relativeVelocity = 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 = ball1.position + 0.5 * penetration * collisionNormal;

   ball2.position = ball2.position - 0.5 * penetration * collisionNormal;

   vDOTn = dot(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 = ball2.velocity + impulse / ball2.mass * collisionNormal;

   ball1.velocity = ball1.velocity - 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.  To simplify matters,
 * we reset when either ball leaves the circle of radius 50.0 centered at
 * the origin.  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 * M_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 = VELOCITY_SCALE * balls[0].velocity;
   balls[0].position = 40.0 * balls[0].position;

   /* Compute position and velocity of second ball. */

   angle += M_PI / 4.0 + 1.5 * M_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 = 40.0 * balls[1].position;
#else
   /* Aim second ball so that it passes through (X_OFFSET, 0.0). */
   balls[1].position = 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

   balls[1].velocity = VELOCITY_SCALE * balls[1].velocity;

#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.
 ***********************************************************************/

void initBalls(void)
{
   /* Compute starting positions and velocities for the balls. */

   placeBalls();

   /* Assign physical attributes to the balls. */

   balls[0].radius = 7.0;
   balls[0].mass = 2.0;
   balls[0].color = vec3(1.0, 0.0, 0.0);

   balls[1].radius = 5.0;
   balls[1].mass = 1.0;
   balls[1].color = vec3(0.0, 0.0, 1.0);

   /* Create geometry information and vaos for each ball. */

   for (int i = 0; i < MAX_BALLS; ++i)
   {
      balls[i].vao = createCircle(balls[i]);
      balls[i].geometry = GL_TRIANGLE_FAN;
      balls[i].numVertices = SLICES;
   }
}


/***********************************************************************
 * Create and setup a complete vao, buffer, and set of shader programs
 * to render the given ball as a circle.
 ***********************************************************************/
```

```

GLuint createCircle(Ball ball)
{
   GLuint vao, buffer;

   vec2 points[SLICES];
   vec3 colors[SLICES];
   GLfloat angle = 0.0;
   GLfloat sliceAngle = 2.0 * M_PI / (GLfloat) SLICES;

   for (int i = 0; i < SLICES; i++)
   {
      points[i] = ball.radius * vec2(cos(angle), sin(angle));
      colors[i] = ball.color;
      angle += sliceAngle;
   }

   glGenVertexArrays(1, &vao);
   glBindVertexArray(vao);

   glGenBuffers(1, &buffer);
   glBindBuffer(GL_ARRAY_BUFFER, buffer);

   glBufferData(GL_ARRAY_BUFFER, sizeof(points) + sizeof(colors),
                NULL, GL_STATIC_DRAW);

   glBufferSubData(GL_ARRAY_BUFFER, 0, sizeof(points), points);
   glBufferSubData(GL_ARRAY_BUFFER, sizeof(points), sizeof(colors), colors);

   glUseProgram(program);

   GLuint vPosition = glGetAttribLocation(program, "vPosition");
   glEnableVertexAttribArray(vPosition);
   glVertexAttribPointer(vPosition, 2, GL_FLOAT, GL_FALSE, 0,
                         BUFFER_OFFSET(0));

   GLuint vColor = glGetAttribLocation(program, "vColor");
   glEnableVertexAttribArray(vColor);
   glVertexAttribPointer(vColor, 3, GL_FLOAT, GL_FALSE, 0,
                         BUFFER_OFFSET(sizeof(points)));

   return vao;
}


/***********************************************************************
 * OpenGL functions.
 ***********************************************************************/

/***********************************************************************
 * Recall, this will do our rendering for us.  It is called following
 * each simulation step in order to update the window.
 ***********************************************************************/

void display(void)
{
   mat4 mv;   /* Model view matrix */
   mat4 p;    /* Projection matrix */

   glClear(GL_COLOR_BUFFER_BIT);

   /* Define the projection matrix and make it available to the vertex
    * shader.
```

```
    */

   p = Ortho(-worldRight, worldRight, -worldTop, worldTop, -1.0, 1.0);
   glUniformMatrix4fv(projection, 1, GL_TRUE, p);

   /* Render balls. */

   for (int i = 0; i < MAX_BALLS; ++i)
   {
      glBindVertexArray(balls[i].vao);

      /* Define the object-appropriate model view matrix and make it
       *  available to the vertex shader.
       */

      mv = Translate(balls[i].position.x, balls[i].position.y, 0.0);
      glUniformMatrix4fv(model_view, 1, GL_TRUE, mv);

      glDrawArrays(balls[i].geometry, 0, balls[i].numVertices);
   }

   /* 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. */

   /* Load shaders and use the resulting shader program. */

   program = InitShader("vshader41.glsl", "fshader41.glsl");

   /* Get the locations of the uniform matrices in the vertex shader. */

   model_view = glGetUniformLocation( program, "model_view" );
   projection = glGetUniformLocation( program, "projection" );

   initBalls();
}


/***********************************************************************
 * Hitting the Esc key will exit the program.
 ***********************************************************************/

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


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

```
 * Handler for reshape events.  This is done by either selecting the
 * largest viewport of the correct aspect ratio (in this case, 1:1) or
 * by reshaping the clipping rectangle so that its aspect ratio matches
 * that of the window.
 ***********************************************************************/

void reshape(int w, int h)
{
   windowWidth = w;
   windowHeight = h;

#ifdef VIEWPORT
   if (w < h)
      glViewport(0, 0, (GLsizei) w, (GLsizei) w);
   else
      glViewport(0, 0, (GLsizei) h, (GLsizei) h);
#else
   if (w < h)
   {
      worldRight = WORLD_HALF;
      worldTop = (float) h / (float) w * WORLD_HALF;
   }
   else
   {
      worldRight = (float) w / (float) h * WORLD_HALF;
      worldTop = WORLD_HALF;
   }

   /* Because we’re adjusting the aspect ratio of the clipping region
    * to match that of the window, use the entire window.
    */

   glViewport(0, 0, (GLsizei) w, (GLsizei) h);
#endif
}


/***********************************************************************
 * 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 += balls[i].velocity;

   /* 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 centered
    * at the origin.
    */
```

```

   if (distanceSquared(balls[0].position) > 2500.0
       || distanceSquared(balls[1].position) > 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);
#ifdef DOUBLE_BUFFER
   glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB);
#else
   glutInitDisplayMode(GLUT_RGB);
#endif
   glutInitWindowSize(windowWidth, windowHeight);
   glutInitWindowPosition(100, 100);
   glutInitContextVersion(3, 2);
   glutInitContextProfile(GLUT_CORE_PROFILE);
   glutCreateWindow("Colliding balls");

   glewExperimental = GL_TRUE;
   glewInit();

   init();

   glutDisplayFunc(display);
#ifdef RESHAPE
   glutReshapeFunc(reshape);
#endif
   glutKeyboardFunc(keyboard);
   glutIdleFunc(idle);

   glutMainLoop();

   return 0;
}
```