Project #3 – Physical Engine

Yoke Kai Wen, 2020280598

May 31, 2021

Contents

1 Introduction

In this project, I implemented six physical engine scenarios (collision, collision_stress, damping_test, newtons_cradle,rotation_test, spring_rotation) by adding code to the starter project code files. A preview of the six scenes can be seen in Figure 1.

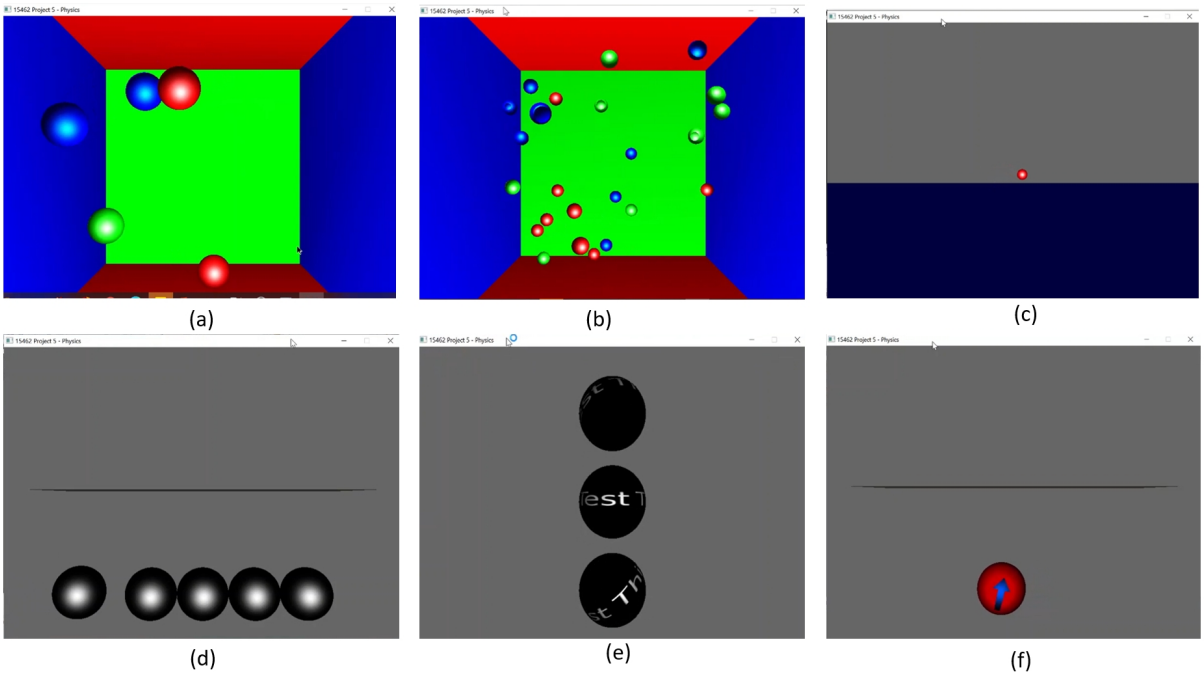

Figure 1: Preview of six scenes: (a) collision, (b) collision_stress, (c) damping_test, (d) newtons_cradle, (e) rotation_test, (f) spring_rotation

2 Files, directory structure and usage

I retain the same directory structure as the given starter project code directory, with a few added/-modified scripts. I ran the entire project on Windows with Visual Studio 2008, which was already given in the starter code under the `PhysicalEngine/p5out/msvc/physics.sln` file path, set up with the appropriate `Include` and `Lib` files. Note that on newer versions of Visual Studio, some of the dependencies become incompatible, so instead of checking through and updating each dependency, it was easier for me to just download VS 2008. The exact description of the starter code files and directory structure can be found in the given README under `PhysicalEngine/p5out/README`. Here, I emphasise the files that I added/modified.

1. `PhysicalEngine/p5out/src/physics/`: This directory contains the source scripts that are modified to complete the physical engine scenarios. The scripts that are edited are `physics.cpp, spherebody.cpp, collisions.cpp, spring.cpp`.

2. `PhysicalEngine/p5out/SUBMIT/`: This directory contains the report and screen recordings of the program running in the six scenes.

3. `PhysicalEngine/p5out/scenes/`: This directory contains the six scenes that should be set as the command line argument. Each scene defines the bodies' properties, such as damping constant, starting positions, material etc, which will determine how the graphics is rendered. In VS 2008, this can be done by setting `Project > Properties > Debugging > Command Arguments`.

3 Design

There are four main scripts involved in the design of the physics engine: `physics.cpp, sphere-body.cpp, collisions.cpp, spring.cpp`.

1. `physics.cpp` contains all physical objects and updates everything in a step function for the world, including the object velocities after collision, the forces and torques acting on the objects, object positions and orientations after collision and spring interactions etc. It calls all the functions I implemented in other files, as seen in code fragment below.

```
    void Physics::step( real_t dt )
{
    // TODO step the world forward by dt. Need to detect collisions, apply
    // forces, and integrate positions and orientations.
    //
    // Note: put RK4 here, not in any of the physics bodies
    //
    // Must use the functions that you implemented
    //
    // Note, when you change the position/orientation of a physics object,
    // change the position/orientation of the graphical object that represents
    // it

  for (int i = 0; i < num_spheres(); i++) {
        spheres[i]->step_position(dt, 0);
        spheres[i]->step_orientation(dt, 0);
    }

    for (int i = 0; i < num_spheres(); i++) {
        //detect collision between spheres
    for (int j = i + 1; j < num_spheres(); j++) {
            collides(*(spheres[i]), *(spheres[j]), collision_damping);
        }
        //detect collision with plane
        for (int j = 0; j < num_planes(); j++) {
            collides(*(spheres[i]), *(planes[j]), collision_damping);
        }
    //detect collision with triangle
        for (int j = 0; j < num_triangles(); j++) {
            collides(*spheres[i], *(triangles[j]), collision_damping);
        }
    }

  for (int i = 0; i < num_springs(); i++) {
        springs[i]->step(dt);

        springs[i]->body1->step_position(dt, 0);
        springs[i]->body1->step_orientation(dt, 0);

        springs[i]->body2->step_position(dt, 0);
```

```
        springs[i]->body2->step_orientation(dt, 0);
    }
}

```

2. `spherebody.cpp` contains the spherebody class, and includes functions to update the position and orientation of the sphere using RK4 integration, with the updated forces and torques as input. While the instructions stated that RK4 integration should be put in `physics.cpp`, I found that it made more sense to call RK4 in the step position and step orientation functions for each sphere in `spherebody.cpp`

3. `collisions.cpp` contains functions for detecting collisions and updating object velocities if collisions occurred.

4. `spring.cpp` contains the spring class, and calculates the spring force and applies it on the objects.

3.1 Add gravity

The gravity value is given as a member in the physics class, and is applied to every sphere body in `physics.cpp` before adding the sphere to the vector storing all spheres. This ensures that gravity is only applied once to each sphere, since gravity is constant throughout.

```
void Physics::add_sphere( SphereBody* b )
{
  //apply gravity to sphere ONCE
  b->apply_force(gravity, Vector3::Zero);
  spheres.push_back( b );
}
```

3.2 Implementing collisions between bodies

In `collisions.cpp`, I implemented three types of collisions: collisions between sphere to sphere, sphere to plane and sphere to triangle.

3.2.1 Collisions between two sphere bodies

The criteria for detecting a collision between two sphere bodies are:

1. The spheres are travelling towards each other.

   This is checked by calculating the dot product of the relative positions and relative velocities of the two spheres. If the dot product is negative (i.e. relative position and relative velocity in opposite directions), the two spheres are moving towards each other. For instance, assume sphere A is on the left of Sphere B. Sphere A is moving right, while Sphere B is still. Then the relative position between A and B is pointing left (posA - posB); the relative velocity is pointing right (velA - velB). Since relative position and relative velocity are in opposite directions, indeed the spheres are travelling towards each other.

2. The distance between the two spheres is smaller than the sum of their radius.

Collision detection can be seen in the code fragment below:

```
    // TODO detect collision. If there is one, update velocity

  Vector3 deltaPos = body1.position - body2.position;
  Vector3 deltaVel = body1.velocity - body2.velocity;

  //check if there is collision: (1) check relative pos and velocity; (2) check
    distance between objects
  bool collision = (dot(deltaPos, deltaVel) < 0) && ((length(deltaPos)) < body1.
    radius + body2.radius);
```

If a collision is detected, the object velocities are updated according to the formula for elastic collision. Damping is also accounted for.

```
  if (collision) {
    //calculate new velocities after elastic collision
    Vector3 d = deltaPos / length(deltaPos);
    Vector3 tmpv2 = 2 * d * body1.mass /(body1.mass + body2.mass) * dot(deltaVel,
    d);
    Vector3 u2 = body2.velocity + tmpv2;
    Vector3 u1 = 1 / body1.mass * (body1.mass * body1.velocity + body2.mass *
    body2.velocity - body2.mass * u2);

    //account for damping
    body1.velocity = u1 - collision_damping*u1;
    body2.velocity = u2 - collision_damping*u2;

    return true;
  }
  else {
    return false;
  }
```

3.2.2 Collision between sphere and plane

To detect if there is a collision between sphere and plane, the criteria is largely similar to that of sphere-sphere collision. First, we check if the relative position and relative velocity are in opposite directions as before. Second, we have to project the centre of the sphere onto the plane, so that we can find the shortest perpendicular distance between the sphere and the plane. This requires computing of the plane's unit normal (pointing towards sphere). If the distance is smaller than that of the sphere's radius, then a collision is detected. See code fragment below.

```
    // TODO detect collision. If there is one, update velocity

  Vector3 deltaPosDiag = body1.position - body2.position; // not the shortest
    perpendicular distance from sphere to plane
  Vector3 planeNormUnit = normalize(body2.normal); // unit normal vector
  if (dot(deltaPosDiag, planeNormUnit) < 0){ //ensure normal is pointing towards
    sphere
    planeNormUnit *= -1;
  }

  Vector3 deltaVel = body1.velocity - body2.velocity;

  real_t d = dot(deltaPosDiag, planeNormUnit); //distance between sphere and plane
  Vector3 deltaPos = d * planeNormUnit; //shortest vector pointing from plane to
    sphere

```

```
  //check if there is collision: (1) check relative pos and velocity; (2) check
    distance between objects
  bool collision = (dot(deltaPos, deltaVel) < 0) && (std::abs(d) < body1.radius);
```

If the sphere collides with the plane, then the velocity of the sphere is updated according to the elastic collision formula, and also adjusted for collision damping.

```
    if (collision) {
    Vector3 u = body1.velocity - 2 * dot(body1.velocity, planeNormUnit) *
    planeNormUnit;
    body1.velocity = u - collision_damping*u;

    return true;
  }
  else {
    return false;
  }
```

3.2.3 Collision between sphere and triangle

Note that triangles here actually look like planes when rendered in the collision scenes, but triangles and spheres are actually two different object types. Hence collision between a sphere and a triangle looks like a ball bouncing off a wall.

To determine if there is a collision between sphere and triangle, there is one more condition to check as compared to the previous two types of collision: checking if the centre of the sphere projected onto the plane lies within the triangle. This is achieved using barycentric coordinates, as shown in the code fragment below.

```
    bool pointInTri(Vector3 A, Vector3 B, Vector3 C, Vector3 P)
{
  // check if point P lies in triangle with vertices A, B, C
  // ----- BARYCENTRIC TECHNIQUE START ---------
  Vector3 v0 = C - A;
  Vector3 v1 = B - A;
  Vector3 v2 = P - A;

  real_t dot00 = dot(v0, v0);
  real_t dot01 = dot(v0, v1);
  real_t dot02 = dot(v0, v2);
  real_t dot11 = dot(v1, v1);
  real_t dot12 = dot(v1, v2);

  // Compute barycentric coordinates
  real_t invDenom = 1 / (dot00 * dot11 - dot01 * dot01);
  real_t u = (dot11 * dot02 - dot01 * dot12) * invDenom;
  real_t v = (dot00 * dot12 - dot01 * dot02) * invDenom;

  // Check if point is in triangle
  bool inTri = (u >= 0) && (v >= 0) && (u + v < 1);
  // ------ BARYCENTRIC TECHNIQUE END -----------

  return inTri;
}
```

After checking the three conditions for sphere-triangle collision: (1) check relative pos and velocity; (2) check distance between objects (3) check if point projected by sphere on plane lies in

triangle, the process for updating velocities after sphere-triangle collision is similar to that for sphere-plane collision, as seen in code fragment below.

```
    bool collides( SphereBody& body1, TriangleBody& body2, real_t
    collision_damping )
{
    // TODO detect collision. If there is one, update velocity

  // get triangle vertices
  Vector3 A = body2.vertices[0];
  Vector3 B = body2.vertices[1];
  Vector3 C = body2.vertices[2];

  Vector3 normalUnit = normalize(cross(B - A, C - A));
  Vector3 deltaPosDiag = body1.position - A;
  if (dot(deltaPosDiag, normalUnit) < 0){ //ensure normal is pointing towards
    sphere
    normalUnit *= -1;
  }

  real_t d = dot(deltaPosDiag, normalUnit); //distance between sphere and plane
    containing triangle
  Vector3 deltaPos = d * normalUnit; //shortest vector pointing from plane to
    sphere
  Vector3 deltaVel = body1.velocity - body2.velocity;
  Vector3 projectedPoint = body1.position - deltaPos; //point projected by sphere
    on plane

  //check if there is collision: (1) check relative pos and velocity; (2) check
    distance between objects
  //(3) check if point projected by sphere on plane lies in triangle
  bool collision = (dot(deltaPos, deltaVel) < 0) && (std::abs(d) < body1.radius)
    && (pointInTri(A, B, C, projectedPoint));

  if (collision) {
    Vector3 u = body1.velocity - 2 * dot(body1.velocity, normalUnit) * normalUnit;
    body1.velocity = u - collision_damping*u;

    return true;
  }
  else {
    return false;
  }
}
```

3.3 Applying force and torque to bodies

In `spherebody.cpp`, I implemented the `apply_force` function to update the force and torque acting on each sphere. The new applied force is simply added to the force vector, while the change in torque is calculated using the cross product of offset and force, and then added to the torque. Note that since multiple forces are applied to an object (e.g. spring, gravity), the forces have to be incremented instead of being directly replaced by the new force applied, as seen in code fragment below.

```
    void SphereBody::apply_force( const Vector3& f, const Vector3& offset )
{
    // TODO apply force/torque to sphere
```

```
4
5   // increment force and torque so that multiple forces (e.g. spring + gravity)
      can be addeded
6   force += f;
7     torque += cross(offset, f);
8 }
```

3.4 Implementing RK4 to integrate bodies position and orientation

In `spherebody.cpp`, I implemented RK4 which is then called in the `step_position` and `step_orientation` functions to update the position and orientation of each sphere body.

3.4.1 RK4 integration

To implement RK4, I first define a `State` and a `Derivative` struct to store both position and velocity quantities and their derivatives, so both can be updated simultaneously conveniently.

```
1     struct State
2 {
3     Vector3 x;      // position
4     Vector3 v;      // velocity
5 };
6
7 struct Derivative
8 {
9     Vector3 dx;      // dx/dt = velocity
10    Vector3 dv;      // dv/dt = acceleration
11 };
```

Then, I implemented an `evaluate` function to advance the physical `State` (position and velocity) from t to $t + dt$ using one set of `Derivative`, and then calculate the output `Derivative` at this new `State`. The derivative of position is velocity, and the derivative of velocity is acceleration, which is calculated in another function `calc_acceleration`. See code fragment below.

```
1 Derivative SphereBody::evaluate( const State &initial, real_t t, real_t dt, const
    Derivative &d, int angular )
2 {
3     State state;
4     state.x = initial.x + d.dx*dt;
5     state.v = initial.v + d.dv*dt;
6
7     Derivative output;
8     output.dx = state.v;
9     output.dv = calc_acceleration( angular );
10    return output;
11 }
```

`calc_acceleration` calculates the positional and the angular acceleration according to the physics formula, as shown in the code fragment below. The acceleration is the key to driving the RK4 integration.

```
1 Vector3 SphereBody::calc_acceleration(int angular)
2 {
3   Vector3 acceleration;
4   if (angular == 0) {
5     acceleration = force / mass;
6   }
7   else {
```

```
    real_t I = 2.0/5.0 * mass * radius * radius;
    acceleration = torque / I;
  }
  return acceleration;
}
```

Finally, I implement the actual integration function `integrate` by calling `evaluate` with the appropriate `Derivative` and timesteps as input. As each `Derivative` is computed, it is fed as input to calculate the next `Derivative`. This feedback of the current `Derivative` into the calculation of the next is what gives the RK4 integrator its accuracy. The integration function finally updates the current `State` and returns the change in position `deltaPos`.

```
Vector3 SphereBody::integrate( State &state, real_t t, real_t dt, int angular )
{
    Derivative a,b,c,d;

    a = evaluate( state, t, 0.0f, Derivative(), angular);
    b = evaluate( state, t, dt*0.5f, a, angular );
    c = evaluate( state, t, dt*0.5f, b, angular );
    d = evaluate( state, t, dt, c, angular );

    Vector3 dxdt = 1.0f / 6.0f * ( a.dx + 2.0f * ( b.dx + c.dx ) + d.dx );

    Vector3 dvdt = 1.0f / 6.0f * ( a.dv + 2.0f * ( b.dv + c.dv ) + d.dv );

    state.x = state.x + dxdt * dt;
    state.v = state.v + dvdt * dt;

  return dxdt * dt;
}
```

3.4.2 Updating position and orientation of sphere

The `step_position` and `step_orientation` functions call the RK4 integration routine to update sphere position and orientation for each timestep. Note that the velocity is first updated assuming that acceleration was constant in that timestep, and then the updated velocity is input into the RK4 integration routine to calculate the new object position/orientation. Also note that the update of orientation requires the use of quaternions and is not as trivially computed as position.

```
Vector3 SphereBody::step_position( real_t dt, real_t motion_damping )
{
    // Note: This function is here as a hint for an approach to take towards
    // programming RK4, you should add more functions to help you or change the
    // scheme
    // TODO return the delta in position dt in the future

    Vector3 deltaV = dt * calc_acceleration(0);
    velocity += deltaV;

    if (length(velocity) < EPSILON_VEL) {
        velocity = Vector3::Zero;
        return Vector3::Zero;
    }

  State curr_state;
  curr_state.x = position;
  curr_state.v = velocity;
```

```
19
20    real_t t = 0; //arbitrary
21
22    //update position and velocity with RK4
23    Vector3 deltaX = integrate(curr_state, t, dt, 0);
24
25    sphere->position = curr_state.x; //change pos of graphical object
26    position = curr_state.x;
27
28    return deltaX;
29  }
30
31  Vector3 SphereBody::step_orientation( real_t dt, real_t motion_damping )
32  {
33      // Note: This function is here as a hint for an approach to take towards
34      // programming RK4, you should add more functions to help you or change the
35      // scheme
36      // TODO return the delta in orientation dt in the future
37      // vec.x = rotation along x axis
38      // vec.y = rotation along y axis
39      // vec.z = rotation along z axis
40      Vector3 delta_angvel = dt * calc_acceleration(1);
41      angular_velocity += delta_angvel;
42
43      if (length(angular_velocity) < EPSILON_ANGVEL) {
44          angular_velocity = Vector3::Zero;
45          return Vector3::Zero;
46      }
47
48    Vector3 deltaTheta;
49
50    // dummy orient vector for rk4 processing
51    Vector3 orient_vec = Vector3::Zero;
52
53    State now_state;
54    now_state.x = orient_vec;
55    now_state.v = angular_velocity;
56
57    real_t t = 0; //arbitrary
58
59    //update orientation and angular velocity with RK4
60    deltaTheta = integrate(now_state, t, dt, 1);
61
62    Quaternion r = Quaternion(deltaTheta, length(deltaTheta));
63      orientation = r * orientation;
64      sphere->orientation = orientation; //change orient of graphical object
65
66      return deltaTheta;
67
68  }
```

3.5 Implementing spring interactions between bodies

In `spring.cpp`, I implemented spring interactions between bodies according to the Hooke's Law equations and damping forces. However first and foremmost, we have to be careful that the spring is attached to a fixed point on the sphere, so we have to rotate the specified offset with the sphere, in order to calculate accurately the displacement between the two bodies. The displacement is

then used to calculate spring force using Hooke's Law and subsequently damping is accounted for. Another thing to note, is that the previous force has to be subtracted before adding the new force, since the `apply_force` function adds the input force to the current force instead of resetting the force.

```
void Spring::step( real_t dt )
{
    // TODO apply forces to attached bodies

  //offset needs to be rotated along with the sphere
  Matrix3 rot_mat = Matrix3::Identity;
  body1->orientation.to_matrix(&rot_mat);
  Vector3 body1_offset_r = rot_mat * body1_offset;
  body2->orientation.to_matrix(&rot_mat);
  Vector3 body2_offset_r = rot_mat * body2_offset;

  // calculating spring force
  Vector3 displ = body1->position + body1_offset_r - body2->position -
    body2_offset_r;
  // Hookes Law
  Vector3 f = - constant * (length(displ) - equilibrium) * normalize(displ);
  // Damping
  f -= damping * (dot(body1->velocity - body2->velocity, normalize(displ))) *
    normalize(displ);

  //subtract previous force and torque, then add new force and torque
  //not same as resetting force and torque to zero, because other forces might be
    acting on body (e.g. gravity)
  body1->apply_force(-old_f, old_body1_offset);
  body2->apply_force(old_f, old_body2_offset);

  body1->apply_force(f, body1_offset_r);
  body2->apply_force(-f, body2_offset_r);

  old_body1_offset = body1_offset_r;
  old_body2_offset = body2_offset_r;
  old_f = f;
}
```

3.6 Implementing collision damping and spring damping

Collision damping is implemented in all the `collide` functions of `collisions.cpp`, and spring damping is implemented in the `step` function of `spring.cpp`.

4 Limitations

For the `collision_stress` scene, the spheres seem to be rotating randomly, not quite sure why.

5 References

1. Past year reference CAD project on physics engine: https://github.com/ArayCHN/Computer-Aided-Design-tree/master/physicsEngine
2. RK4 integration routine reference: https://gafferongames.com/post/integration_basics/

3. Barycentric coordinates reference: https://blackpawn.com/texts/pointinpoly/default.html