1 Introduction

Physically based modeling has been an extremely active area of research in computer graphics for over a decade. Fueled by parallel research in computational physics, the field is moving towards modeling physical phenomena in a more realistic way. Physically based modeling has developed as an alternative way of modeling some forms of motion that were previously quite tedious using traditional keyframe/interpolation animation.

The purpose of this assignment is to give you the chance to implement a physically-based modeling system based upon methods discussed in class and some new ones presented here. Some material may be familiar as a simplification of parts of the Baraff notes available off the home page (which describes a somewhat different design for a 3D simulation using quaternions). As before, some of you may use the skills you acquire in this assignment when the time comes to implement your final project.

In this assignment, you create a 2D room in which particles and boxes both fall and bounce off of the walls. To be a bit more precise, your executable should support the following objects and interactions (we’ll go through each in detail later in the handout):

- Falling particle masses
- Spring forces on objects
- Collision with the room for particles
- Free falling rigid bodies (a rectangle)
- Wall collision for rigid bodies

We will be giving you some support code for the 2D room, written in C++ and using the poorly documented but open source GUI library GTK. We will also be giving you code to implement all but the last two features. This means less code for you to write, but, as in the last assignment, more code for you to understand. Some of the extra credit options in this assignment may warrant your editing the support code and/or potentially throwing it out the window and doing your own thing completely (in the case of a full “3D with articulated figures” dynamic simulation).

Note: this handout, very similar to the one used in cs224 3 or 4 years ago (when this was a miniproject), just covers the theory. I don’t think it’s fair to not explain the support code at all, so
I’m writing a supplement (henceforth referred to as “supplementary handout”). Look for a link on the web page when it’s done.

2 Theory: Point Bodies

2.1 Point Bodies: Forces

As you know from high school physics, force equals mass times acceleration. Given the forces acting on a body and its initial position and velocity, this fact allows you to determine the body’s motion. Another fun fact about force is that it can be decomposed into its x and y components. In fact, given a force and an arbitrary direction, the force can be represented as an amount of force along the given direction and the amount of force perpendicular to it. This will become especially important when determining collisions with the boundaries of your room.

2.2 Point Bodies: Spring Forces

In addition to gravity, you will allow the user to exert spring forces on any object in your system. A spring force is exerted along the axis of the spring and depends on how far the spring has been stretched from its rest length. The formula for force generated by a spring with axis $d$ is as follows:

$$f = k(l - l_{rest})d$$

Where $f$ is a unit vector and $k$ is a spring constant which affects the force that the spring generates. Note that if a spring is compacted to a length shorter than its rest length, it exerts a force to expand itself.

In the support code, when the user clicks the mouse, the system creates a spring with a rest length of twenty pixels between the mouse and the closest particle (or object vertex). When the cursor or object moves, it changes the spring length. You can change those constants or the general method of interaction at your leisure.

2.3 Point Bodies: Particles in a “Room”

The “room” consists of four immobile walls. Each wall is be represented by a unit normal and a point $p$ on the wall. The normal points into the room. This makes it simple to check if a particle at point $q$ (whose position is described as vector $q$ from the origin of the world frame $f$) is inside of the room. Note that $n$ is a unit normal. See Figure (1).

If the dot product $(q - p) \cdot n$ is negative (e.g. the angle between the vectors is greater than $\pi$), then the particle is on the wrong side of the wall.

2.4 Point Bodies: Particles Colliding in a Room

In this simulation, with the exception of collisions, everything which affects the motion of the particles is time dependent. Forces affect a particle over a given time step by specifying a net
acceleration. Acceleration is integrated with respect to time to update velocity, which is again integrated to update position. You should integrate using the Runge-Kutta routines provided by the TAs. See the supplementary handout for the lowdown on the integration routines.

Collisions are computed by an entirely different process and do not involve integration at all. In your simulation, after you finish computing the state of your particles for a given time step, you should then check for collisions, compute collision responses if necessary, and redraw the scene.

When a particle collides with a wall in this system, the effect is to change the particle’s velocity perpendicular to the wall; we assume that the wall is immobile and does not move as a result of a collision. Since there is no friction in the system, there is no change in the particle’s velocity parallel to the wall. The change in normal velocity depends on the elasticity of the particle. This varies from totally elastic collisions (like a super ball) to totally inelastic collisions (like mashed potatoes). The formula for change in velocity of a particle due to an impulse is as follows:

$$\Delta v = (-1 - e)(v \cdot n)n$$

Where $v$ is the current velocity vector, $n$ is the normal to the surface, and $e$ is a coefficient of elasticity ($0 < e < 1$).

2.5 Point Bodies: The General Algorithm as Implemented

- Calculate time varying forces and update positions and velocities of all the particles in the room using Runge-Kutta integration.

- Then, for each particle:
  - Check whether it is on the wrong side of any wall as outlined above.
  - If the particle lies on the wrong side of a wall, and is moving in the wrong direction, displace it along the wall’s normal so that it lies exactly on the wall (note: this is a cheap hack, but it works.)
– Update the velocity according to the formula for impulses (described in the auxiliary code-specific handout).
– Repeat until the particle lies entirely within the room.
• Draw the room.

3 Theory: Rigid Bodies

In the formulas given in this section, we use the following notation:

• \( P \) is a point to which an impulse is applied. In your program, \( P \) will either be a particle body or a vertex of a rigid body
• \( I_z \) is the moment of inertia
• \( f_P \) is an force applied to a point \( P \)
• \( \tau \) is the torque (a scalar since in 2D we only rotate around one axis, \( a_{zP} \))
• \( m \) is the mass
• \( r \) is the vector from the center of mass to a vertex
• \( n \) is a unit vector normal to a wall
• \( e \) is the coefficient of elasticity of the rigid body
• \( \mathbf{v}_P \) is the velocity of a vertex \( P \)
• \( \mathbf{v}_{\text{center}} \) is the velocity of the center of mass (and therefore equal to the linear component of the velocity)
• \( \mathbf{v}_{\text{vertex}} \) is the component of the velocity along the normal of the wall
• \( \theta \) is the angle between the \( a_{\text{xy}P} \) axis of the object and the \( a_{\text{xy}0} \) axis of the world (i.e. its \( x \) and \( y \) components are 0)
• \( \omega \) is the angular velocity
• \( \dot{\omega} \) is the angular acceleration
• \( J \) is the impulse applied at a point \( P \)
3.1 Rigid Bodies: Moment of Inertia

Although technically the world is made entirely of particles, they tend to stick together and form objects or rigid bodies. These differ from particles in that you can apply force to any one of the particles in a rigid body and it affects the motion of the others. Thus, in addition to translating, rigid bodies also rotate about their center of mass. Translation of rigid bodies is no different than translation of particles. You calculate acceleration from the accumulated forces and then integrate to get linear velocity and position. Rigid bodies also have a moment of inertia $I$, which essentially describes how the mass of the body is distributed throughout the body and affects how it rotates. In 3D, $I$ is a tensor. In 2D, we can treat it as a scalar, denoted by $I_z$. For a 2D rectangular rigid body we have:

$$I_z = \frac{m(x^2 + y^2)}{12}$$

where $x$ is the length and $y$ the width of the body.

You may have derived this result in a Physics class, or you might just recognize it as a 2D simplification of the derivation of the inertia of a block in section 5.1 of the Baraff papers.

3.2 Rigid Bodies: Angular Velocity, Angular Acceleration, Torque

In addition to translational motion, you need to keep track of rotation. More precisely, rotation is the angle between the vector defining the $x$ or $y$ axis of the object and the $x$ or $y$ axis (respectively) of the world. (We say “$x$ or $y$” because it doesn’t matter which one; the angle is measuring a rotation of the rigid body about its $z$ axis.) We’ll use $\theta$ to represent this angle and $\omega$ to represent its derivative, angular velocity.

So the next question is, given a force at a point $P$, how do you calculate $\omega$ and $\theta$? To do this, we make use of the vector, $\mathbf{r}$, between $P$ and the center of mass.

![Figure 2: Calculating the torque $\tau$](image)

To calculate the angular velocity $\omega$ (for 2D only):

- Calculate the torque: $\tau = |\mathbf{r} \times \mathbf{f}|$
- Calculate the angular acceleration: $\dot{\omega} = \frac{\tau}{I_z}$ (see the translational analogue: $\dot{v} = \frac{f}{m}$)
- Integrate the angular accleration $\dot{\omega}$ to find the angular velocity $\omega$.
- Integrate the angular velocity $\omega$ to find the angle $\theta$. 
3.3 Rigid Bodies: Impulses

When a rigid body collides with the wall of the room, the body changes its velocity instantly. This is caused by an impulse $J$ applied at the point of collision (see Section 8 of the Baraff papers). Impulses are applied at a given point on the body, negating the component of the velocity of that point along a given direction. In our case, the direction will be normal to the wall with which the body collides.

![Figure 3: An impulse $J$]

3.3.1 Rigid Bodies: Finding the Change in Linear Velocity

The velocity $v$ of a given point has a linear and an angular component:

$$v = v_{\text{center of mass}} + \omega \times r$$

Note that the linear component of the velocity of a vertex $P$ is the same as the velocity of the body’s center of mass, and the angular component of the velocity in 3D equals $\omega \times r$, which has a magnitude of $\omega r$.

The impulse $J$ is defined:

$$J = m(\Delta v_{\text{center}})$$

So to find the change in linear velocity:

$$\Delta v_{\text{center}} = \frac{J}{m}$$

3.4 Rigid Bodies: Finding the Change in Angular Velocity

From the formulas for torque $\tau = |r \times f|$ and angular acceleration $\ddot{\omega} = \frac{\tau}{I}$, we get
\[ \dot{\omega} = \frac{r \times f}{I_z} \]

From this we can derive a formula for the change in angular velocity of a point \( P \):

\[ \frac{\Delta \omega}{\Delta t} = \frac{r \times \frac{J}{I_z}}{I_z} \]
\[ \Delta \omega = \frac{r \times J}{I_z} \]

### 3.5 Rigid Bodies: Finding the Impulse

To find \( J \):

\[ J = Jn \]

The component of velocity along the vector \( n \) is

\[ v_{\text{vertex}} = v \cdot n \]

where, as before,

\[ v = v_{\text{center of mass}} + \omega \times r \]

We can use the previous formula for the change in velocity of a particle due to an impulse and eventually get the following:

\[ J = \frac{-(1 + e)v_{\text{vertex}} \cdot n}{m + n \cdot \frac{1}{I_z}(r \times n) \times r} \]

The derivation for this is worth understanding. I am going to attempt to translate Nancy’s notes on said derivation and add it to the supplementary handout.

### 3.6 Implementation Notes

Implement your RigidBody class as a subclass of your particle class. See the notes in the support code for more details.

### 3.7 Extra Credit

You may notice that the demo for this assignment looks kind of dumb in several respects. For one thing, all collisions are frictionless, which detracts from the realism. For another, the rigid bodies (and particles) pass through each other – collisions occur only between bodies and walls. Lastly, when a rigid body stops bouncing around and comes to rest on the floor, it frequently continues
to jerk and jump as though it has little ants crawling under it. That’s because only collision impulses are computed by the demo, not contact forces which arise when a body is resting quietly against a wall. So corners of the resting body keep "colliding" with the floor, causing the corner to jerk back, which causes another corner to collide, and so on ad infinitum. Ideally, we should distinguish between the case when the body is colliding with the wall and when it is simply resting against the wall. Computing contact forces is in general a difficult problem (see “Fast Contact Force Computation for Nonpenetrating Rigid Bodies” by Baraff in Siggraph ‘94). The problem is not as difficult if you deal just with the case when a body is resting on the floor. (The demo makes a quick pass at dealing with this case, with little success). You are invited to address any of these issues for extra credit.

Additionally, you are encouraged to attempt to add articulated objects to the system. Nancy’s forward dynamics handout describes all you need to know to do this in 2 dimensions. 3 dimensions would be even cooler, and if you could hurl around objects with joints in 3D you would get more extra credit than you could ever want plus the adoration of your peers. I’d love to see a few people tackle these extensions.

3.8 Handing in

tar up your code and handin with:

```
/cs/bin/handin -c cs229 dynamics [filename]
```

A README and vaguely commented code would be again excellent.