# A Rigid-Body Simulator

Christopher D. Van Horn

July 2008

# 1 Overview

I developed a rigid-body simulator in the Java programming language. A rigid body is an object, in this case a sphere, that experiences no deformations and no changes in size. The simulator calculates the positions and orientations of a group of objects that may collide with each other, over a given time span. I chose to write the simulator in Java so that the simulator will be portable and to improve my knowledge of the language.

The graphical element to the simulation uses OpenGL for 3D rendering. Specifically, JOGL, Java's OpenGL API bindings, handles rendering. Simulation objects are textured so that the viewer can tell if the object is rotating. Otherwise, the graphical representation is rather rudimentary; its main goal is to convey to the viewer what is happening in the simulation.

The simulation is driven by a configuration file that determines the initial positions and orientations of all objects in the simulation, acceleration due to gravity, if the simulation should pause when objects collide, the number of renderers, and the camera location and orientation for each renderer. While the simulation is running, the user can pause, resume, or restart the simulation, with commands entered via the keyboard or via the graphical user interface.

The simulation models translational and rotational kinematics of a group of rigid bodies. Kinematics is a portion of mechanics that describes motion in terms of space and time, ignoring the agents that cause said motion [5, p. 23]. Since objects may collide with each other, it is necessary to take dynamics into consideration. Dynamics "describes the relationship between the motion of a body and the forces acting on that body [5, p. 104]."

The simulator is a multi-threaded application. There are three threads: one for rendering, one for physics computations, and one for input. The physics thread produces frames to be consumed by the rendering thread. If the buffer of frames becomes empty, the rendering thread sleeps until there is a frame in the buffer. Since the rendering thread may sleep, input is polled in a separate thread so the application remains responsive.

## 1.1 Rigid-Body Simulator Uses

There are a few uses for a rigid-body simulator. One is as an education aid and the other is for games. With a rigid-body simulator, students can explore concepts in ways that are not possible in a lab. For instance, students can simulate how objects react without gravity. Students can simulate objects that interact with no friction, which as far as I know is not possible on this planet. Students can examine interactions between any type of material without actually having to have the materials on hand.

Rigid-body simulators are also a major component in video games. The simulators allow games to be more realistic and dynamic than their strictly deterministic counterparts; a slight change in control input can cause different things to happen in the game. Physics simulators can also be used for the core of the game, for instance, a game where the user attempts to build a tower that withstands gravity and ground motion.

# 2 Rigid-Body Simulator Design

A rigid-body simulator has three duties: update object locations, detect collisions, and respond to collisions (Figure 1). To update the object locations, the simulator first takes into account all forces acting on the object and computes the current acceleration with Newton's second law, $F = ma$. Finally, the simulator determines the object's current location using the current acceleration.

To demonstrate how a rigid-body simulator updates object locations, let's look at a collision-free case. We assume that the objects do not collide or, if they do, they are allowed to pass through each other.

To compute the new location of the objects, we start by integrating $F = ma$, where $F$ is the total force on the object, $m$ is the mass of the object, and $a$ is the acceleration of the object. Acceleration is the change in velocity over the change in time, so let's substitute for acceleration, giving us $F = m\frac{dv}{dt}$. Next, divide each side by $m$ and multiply each side by $dt$, after which our equation is, $dv = \frac{F}{m}dt$. In this form we can compute the change in velocity given the total force on the object, its mass, and how much time has passed. This problem is an initial-value problem of the form $v_{t+\Delta t} = v_t + \frac{F}{m}\Delta t$, $v_{t=0} = v_0$, so we need to have the initial velocity of the objects in order to compute the new velocities.

With the object's new velocity we can calculate its new position. Velocity is the change in location over the change in time, $v = \frac{ds}{dt}$. If we multiply both sides by $dt$, we have an equation for the change in location, $ds = vdt$. This problem is another initial-value problem of the form $s_{t+\Delta t} = s_t + v_{t+\Delta t}\Delta t$, $s_{t=0} = s_0$, so given the current location of the object, its velocity at the new time step, and the change in time, we can compute the new location.

The same steps can be used to create equations for the change in angular velocity and the change in orientation. In those cases, we start with the equation $M = I\alpha$, where $M$ is the total moment on the object, $I$ is the moment of inertia about the axis of rotation, and $\alpha$ is the angular acceleration. Angular acceleration is the change in angular velocity over the change in time, so let's substitute for angular acceleration, giving us $M = I\frac{d\omega}{dt}$. Divide each side by $I$ and multiply each side by $dt$, so that our equation is now $d\omega = \frac{M}{I}dt$. Using this equation, we can compute the new angular velocity given the total moment on the object, its moment of inertia about the axis of rotation, and how much time has passed. This is another initial-value problem of the form $\omega_{t+\Delta t} = \omega_t + \frac{M}{I}\Delta t$, $\omega_{t=0} = \omega_0$; once again we need the initial angular velocity of the object in order to compute new angular velocities.

Using the new angular velocity, we can calculate the object's new orientation. Angular velocity is defined as the change in orientation over the change in time, $\omega = \frac{d\Omega}{dt}$. As before, if we multiply both sides of the equation by $dt$, we have $d\Omega = \omega dt$, which is another initial-value problem of the form $\Omega_{t+\Delta t} = \Omega_t + \omega_{t+\Delta t}\Delta t$, $\Omega_{t=0} = \Omega_0$.

Next, let's look at how a rigid-body simulator detects collisions. We assume there are multiple objects that are not allowed to overlap. Also, let's assume that the objects are spheres or finite planes.

To determine if a collision occurs, we can use the geometric properties of the objects. For example, if we are checking for a collision between two spheres, we calculate the vector between the centers of the spheres and then we calculate the length of the vector. By comparing the length of the vector to the sum of the radii, we can determine if the two spheres have collided.

There are two types of contact: resting contact and colliding contact. Resting contact is when the objects are in contact but are not moving towards each other. Colliding contact is when the objects are in contact and are moving towards each other.

Let's assume two objects are touching, the velocity of object $a$ is $v_a$, and the normal from object $b$ to object $a$ is $N$. Then there is colliding contact if $v_a \cdot N < 0$, resting contact if $v_a \cdot N = 0$, and the objects are separating if $v_a \cdot N > 0$.

Once we have determined that two objects have collided, we need to respond to the collision. When two objects collide, they both experience an impulse force. An impulse force is equal to the change in momentum an object experiences, $J = \Delta p = m_+v_+ - m_-v_-$. We use the form $J = m(v_+ - v_-)$, since the mass of the object does not change during collision.

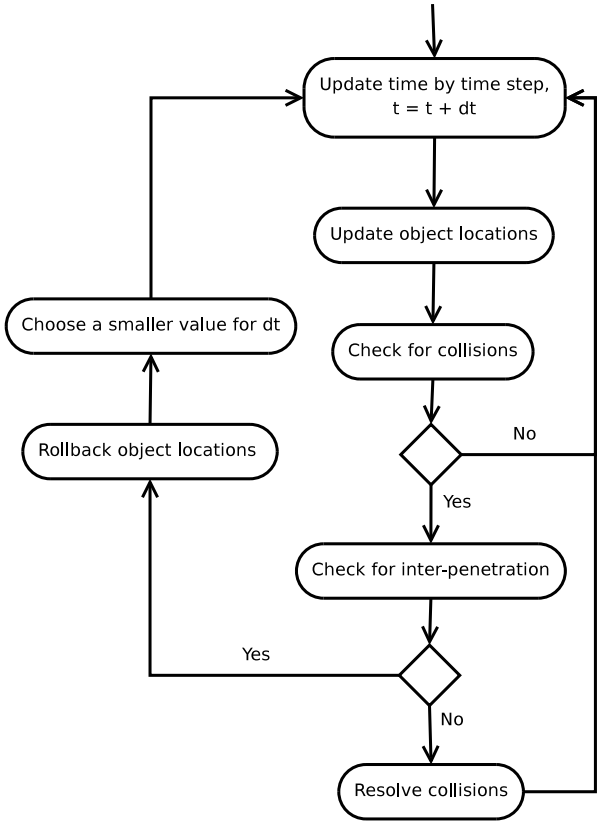



Figure 1: Rigid-body simulator flowchart

When objects collide, energy is lost in the form of sound, heat, and object deformation. Instead of trying to accurately compute the loss of energy, we use one coefficient to model the energy loss, called the coefficient of restitution. The coefficient of restitution can be between 0 and 1. If it is equal to 1, no energy is lost; if it is equal to 0, all energy is lost. The coefficient of restitution ($R$) relates the before and after velocities of the two objects involved in the collision:

$$R = \frac{-(v_{a+} - v_{b+})}{v_{a-} - v_{b-}} \tag{1}$$

Using the definition of impulse and the coefficient of restitution, we can compute the after-collision velocity of the objects involved [3, p. 96]. On collision, object $a$ receives a positive impulse and object $b$ receives a negative impulse

$$J = m_a(v_{a+} - v_{a-}) \tag{2}$$

$$-J = m_b(v_{b+} - v_{b-}) \tag{3}$$

We have three equations with three unknowns $v_{a+}$, $v_{b+}$, and $J$ that we solve:

$$J = \frac{-v_r(R+1)}{1/m_a + 1/m_b} \tag{4}$$

$$v_{a+} = v_{a-} + \frac{Jn}{m_a} \tag{5}$$

$$v_{b+} = v_{b-} + \frac{-Jn}{m_b} \tag{6}$$

where $n$ is the normal between the centers of mass of the two objects and $v_r$ is the relative velocity.

If the two objects are rotating, we have to take the velocity of the contact points into account. The velocity of the contact point is

$$v = v_{cg} + (\omega \times r)$$

Substituting the contact-point velocity into our impulse equations (5) and (6), we obtain

$$v_{acg+} + (\omega_{a+} \times r_a) = \frac{J}{m_a} + v_{acg-} + (\omega_{a-} \times r_a)$$

$$v_{bcg+} + (\omega_{b+} \times r_b) = \frac{-J}{m_b} + v_{bcg-} + (\omega_{b-} \times r_b)$$

There are two unknowns, the angular velocities after impact, so we need two more equations. So we use the definition of angular impulse

$$(r_a \times J) = I_a(\omega_{a+} - \omega_{a-})$$

$$(r_b \times -J) = I_b(\omega_{b+} - \omega_{b-})$$

Solving for $\omega_{a+}$ and $\omega_{b+}$ and substituting these equations into the coefficient of restitution we obtain an equation to compute the impulse.

$$J = \frac{-v_r(R+1)}{1/m_a + 1/m_b + (n \cdot (\frac{r_a \times n}{I_a} \times r_a)) + (n \cdot (\frac{r_b \times n}{I_b} \times r_b))}$$

Once we compute the impulse $J$, we can solve our velocity equations:

$$v_{a+} = v_{a-} + \frac{Jn}{m_1} \tag{7}$$

$$v_{b+} = v_{b-} + \frac{-Jn}{m_2} \tag{8}$$

$$\omega_{a+} = \omega_{a-} + (r_a \times Jn)/I_a \tag{9}$$

$$\omega_{b+} = \omega_{b-} + (r_b \times -Jn)/I_b \tag{10}$$

Finally, we take friction into account, yielding:

$$v_{a+} = v_{a-} + \frac{Jn + \mu Jt}{m_1} \tag{11}$$

$$v_{b+} = v_{b-} + \frac{-Jn + \mu Jt}{m_2} \tag{12}$$

$$\omega_{a+} = \omega_{a-} + (r_a \times (Jn + \mu Jt))/I_a \tag{13}$$

$$\omega_{b+} = \omega_{b-} + (r_b \times (-Jn + \mu Jt))/I_b \tag{14}$$

where t is the unit vector tangent to the collision,

$$t = (n \times v_r) \times n$$

and $\mu$ is the coefficient of friction, which is determined by the materials of the two objects.

When there is resting contact, we cancel any forces along the contact normal to prevent the objects from inter-penetrating. So, if a sphere is resting on a horizontal plane, we cancel gravity to prevent the sphere from entering the plane.

Now we put our three components, object updating, collision detection, and collision response into a loop at the heart of the rigid-body simulator. The simulator advances time by $\Delta t$, a time-step value. The simulator uses the change in time to calculate the new position and orientation of each object in the simulation.

Next, we run our collision-detection test on all objects. Each object is compared to every other object in the simulation, which can result in one of three cases: Objects are not touching, objects are touching, or objects are inter-penetrating. If the objects are not touching, we move on to the next pair. If the objects are touching, we use our collision-response equations to determine the new translational and rotational velocity of the objects involved. If the objects are inter-penetrating, we return to the beginning of this time step and calculate a smaller value for $\Delta t$, then start the collision detection/response checks over again. A time step is completed when all objects have been updated and no two objects inter-penetrate.

# 3 My Simulator Design

My simulator is based on the concept of a film projector. A film projector consists of three major components: a light source with a lens, a film strip, and various controls. In my simulator, each of three major components is represented by a thread. The simulation thread generates the filmstrip, the rendering thread displays frames like the light source and lens, and the input thread handles projector controls.

Let's start with the simulation thread (Figure 2), since it is the most important part of the simulator. The simulation thread takes the number of time steps, the length of each time step, the strength of gravity, whether gravity is expressed as a force or an acceleration, a file describing materials, and a file describing rigid bodies. The length of a time step, along with the number of time steps, determines the amount of time simulated. If the time step is too long and the objects are moving quickly, it is possible for objects to pass through each other.

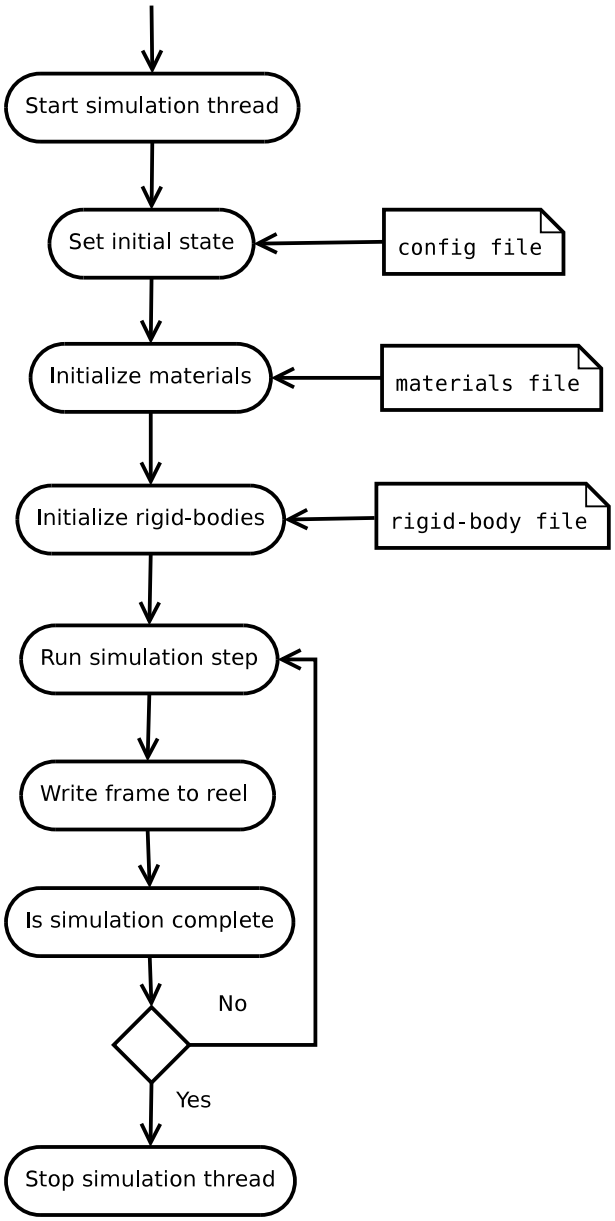



Figure 2: Flowchart of the simulation thread

Materials determine the friction when two objects collide. Each object is assigned a material; when they collide, the coefficient of friction is found in a table indexed by the materials of the two objects.

After setting up its initial state, the simulator loads the rigid bodies from the file. The file contains a complete description of every rigid body in the scene. Each rigid body is described by

its initial location in world coordinates, initial orientation, outward appearance (color, textures), object-specific dimensions (radius of a sphere), material, mass, center of mass in object coordinates, initial force on the object, what point the initial force is acting through, the magnitude of the initial force, and the duration of the initial force. Given this information about each object, we can determine its position, velocity, and spin after a time step.

The simulation thread generates a frame for each time step. It first updates the position and orientation of all objects, having saved their states in case of rollback.

Each rigid-body data structure (Figure 3) contains its current kinematic state as well as any forces that need to be applied to the object during the next update. Each force data structure contains its force normal, magnitude of the force, and the point on the rigid body through which the force is acting. The rigid-body data structure also contains its physical representation. The simulation thread updates a rigid body by applying forces. Each force causes an acceleration, which modifies the body's translational velocity and angular velocity, which finally modifies the body's location and orientation.

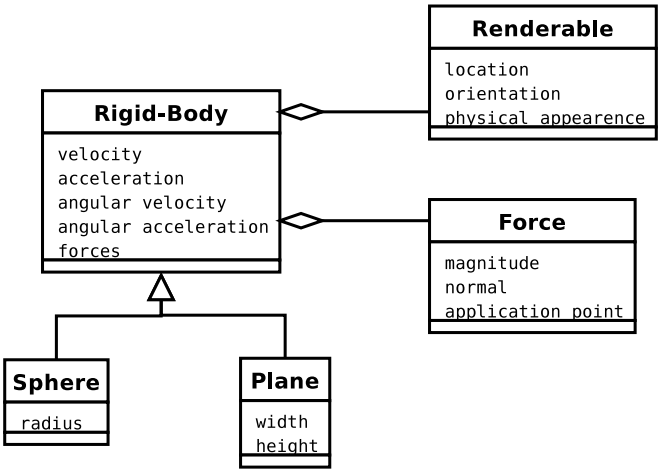



Figure 3: Simplified class diagram of a rigid-body object

After the simulation thread updates all of the objects, it makes sure that no object is penetrating another object. It compares each object to every other object. If it finds that any objects inter-penetrate, it rolls back to the beginning of this time step and takes a smaller time step. The simulation thread calculates the new time step to be half of its previous value. If objects are touching, it computes the collision response.

The simulation thread finally generates a frame so the current state can be rendered. The new frame contains the physical representation of each object. The simulation thread then places the frame on the reel.

After the simulation thread creates the frame, it increments the current time and starts work on the next frame.

### 3.1 Rendering Thread

The rendering thread (Figure 4) pulls frames from the reel and renders them on the screen. The thread takes the number of renderers, the viewing frustum for each renderer, the camera position for each renderer, and the type of each renderer. There can be a maximum of four renderers. Using the given information, each renderer initializes and enters the main renderer loop.

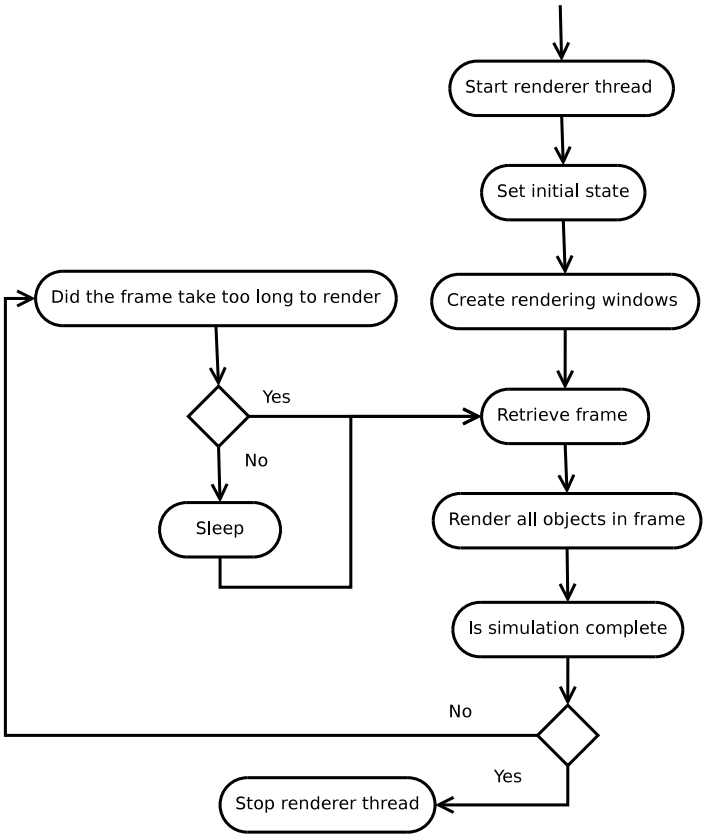



Figure 4: Flowchart of the rendering thread

The main renderer loop pulls the next available frame from the reel and renders each object in the frame. Renderable objects (Figure 5) know how to render themselves, so the thread tells each renderable object to render itself. The renderable object takes advantage of the decorator design pattern to specify how the object should look on the screen. Each renderable can be decorated with color or with textures. The on-screen appearance is determined by the objects file.

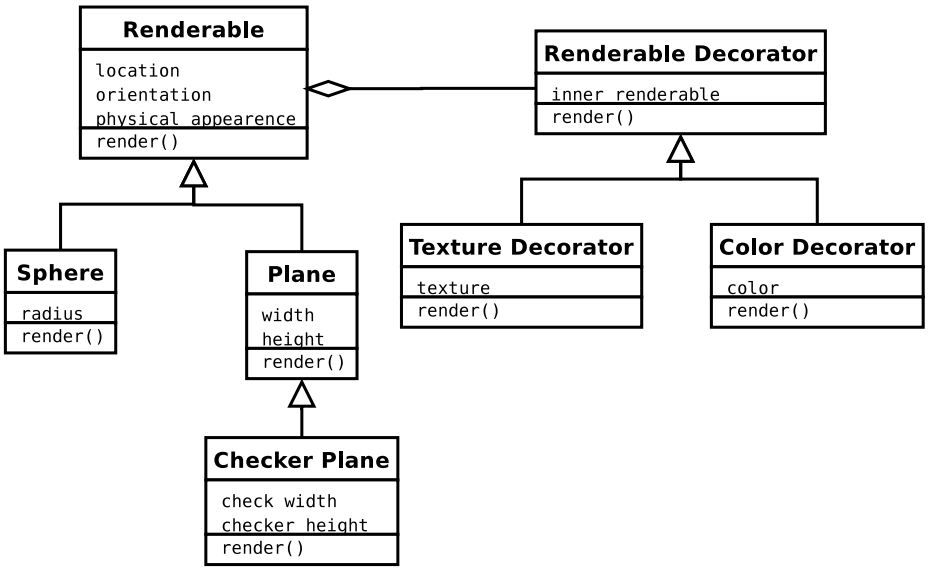



Figure 5: Simplified class diagram of a renderable object

What is rendered on the screen is determined by the list of vertices specified by the program along with the current state of OpenGL. All vertices go through a rendering pipeline (see Figure 6).

The pipeline takes vertices as input and produces pixels on the screen. The first stage the vertices go through is the per-vertex operations stage, which assembles lists of vertices into primitives. If texturing is enabled, OpenGL generates texture coordinates during this stage. Next, OpenGL clips the primitives against the viewing frustum, and, if culling is enabled, OpenGL removes non-visible polygons.

The rasterization stage of the pipeline takes the completed primitives. OpenGL converts primitive data into fragments. Each fragment represents a pixel on the screen. At this point OpenGL applies color to the fragments.

Before OpenGL writes the fragments to the framebuffer, it applies textures and performs depth-buffer tests. The depth-buffer test verifies that the tested fragment is not covered by another fragment. As long as the depth-buffer test passes, OpenGL writes the fragment to the framebuffer.

It is important for the rendering thread to maintain its timing. Each frame contains the amount of time the frame should be displayed for, the target render time. The rendering thread times how long it takes to render each frame and compares the actual render time to the target render time. If the actual render time is less than the target render time, the rendering thread sleeps the difference between the two times. If the actual render time is greater than or equal to the target render time, the rendering thread immediately moves on to the next frame.

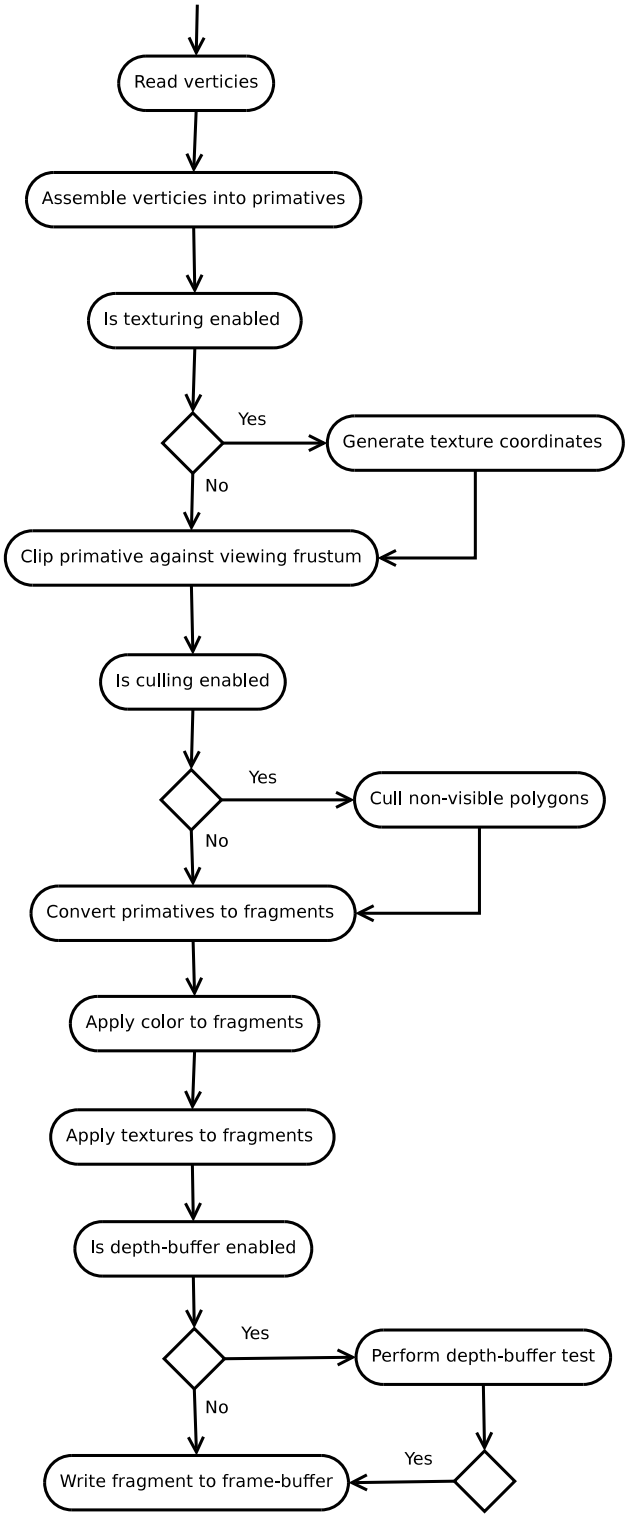



Figure 6: Flowchart of the basic OpenGL rendering pipeline

### 3.2 Input Thread

The input thread (Figure 7) is in charge of taking user input and notifying the other threads when important events happen. The input thread sets up the renderer and simulation threads first. If graphical playback controls are used, the thread launches a thread to service the controls.

The command-line interface and the graphical user interface call the same functions for the same commands. Classes in the simulator implement a playback interface; if a user presses the GUI play button or types "play" into the command-line, the same function is called.

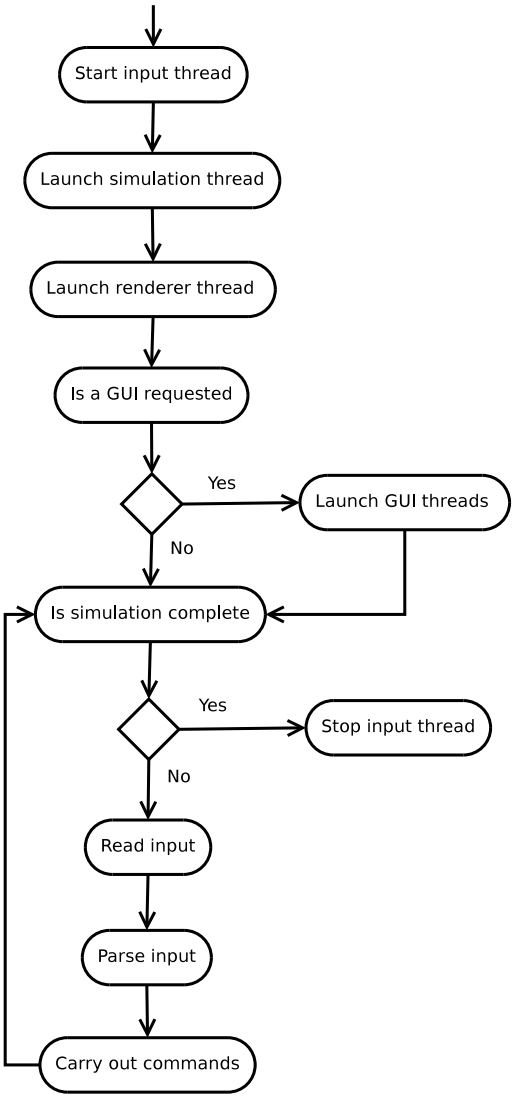



Figure 7: Flowchart of the input thread

The input loop is straightforward. The command-line thread and the GUI thread listen for input. When they receive input, they parse the input to identify the command that needs to be called. Once the proper command has been located, the input thread calls the function that maps to the selected command.

The user also interacts with the simulator via the configuration files. The `config` file describes the renderers, gravitational constants, the user interface, and the number of time steps and total

runtime. The `config` file is organized in name-value pairs. The `rigid-body` file describes the objects in the simulation, and the `materials` file describes friction properties of materials.

# 4 Experience

## 4.1 Difficulties

As in any software project, problems arose during development. One of the major problems that I ran into was designing the `RigidBody` class. During initial development, I modified an object's velocity by directly manipulating a field in the `RigidBody` class. This method causes problems when an object rests on a plane and another object falls upon it from above. The object resting on the plane experiences two forces from two separate contacts, and depending on what order the contacts are processed, the resting object could react differently. In this situation the resting object's final velocity should be away from the plane. The falling object pushes down on the resting object; since the resting object is resting on the plane, all forces downward into the plane are reflected back into the resting object. If the resting contact is processed first, not all of the energy returns into the resting sphere, which therefore remains in contact with the plane.

To fix this problem I added a Force class, and each `RigidBody` class now contains a list of forces that need to be processed each frame. Reactionary forces are processed last. A reactionary force is one whose magnitude is determined by all other forces on the object. Resting contacts introduce reactionary forces, so they always react properly. Now that the only way to change the kinematic properties of an object is to apply a force to it, collision response is greatly simplified. It's easier to calculate the collision impulse force than to directly calculate the new velocity or acceleration. Making the simulation impulse-based also moves all velocity force-dependent calculations to one place.

Another difficulty I ran into was the design of input. Initially, a class in the input thread implemented all possible input commands. This layout made it difficult to implement some commands, because the commands need access to some classes that they should not have access to. So I implemented a design (Figure 8) in which the most obvious class is responsible for implementing the commands. Now the graphical interface and the command-line interface can easily call the same functions.

## 4.2 New Technologies Learned

To develop this project, I had to learn some new technologies. To present the output of the simulation required a 3D API. I chose OpenGL, since OpenGL allows the simulator to be run on most platforms.

In order to use OpenGL with Java, the program requires a binding layer called JOGL[2]. The Game Technology Group at Sun Microsystems is developing JOGL. JOGL allows access to the OpenGL C libraries via the Java Native Interface (JNI). JNI allows code executing in the JVM to make calls into native libraries and applications.

As a single example, we describe how to pass vertices to the pipeline by looking at how to render a triangle. First, we need to let OpenGL know what type of primitive we will be rendering by calling `glBegin`. Since we are rendering a triangle, we pass `GL_TRIANGLES` as an argument to `glBegin`. We then make three calls to `glVertex` that specify the three vertices of the triangle. If we want to render more than one triangle, we make three calls to `glVertex` for each triangle. After we have specified the vertices for the triangle, we call `glEnd` to tell OpenGL we are done rendering triangles.

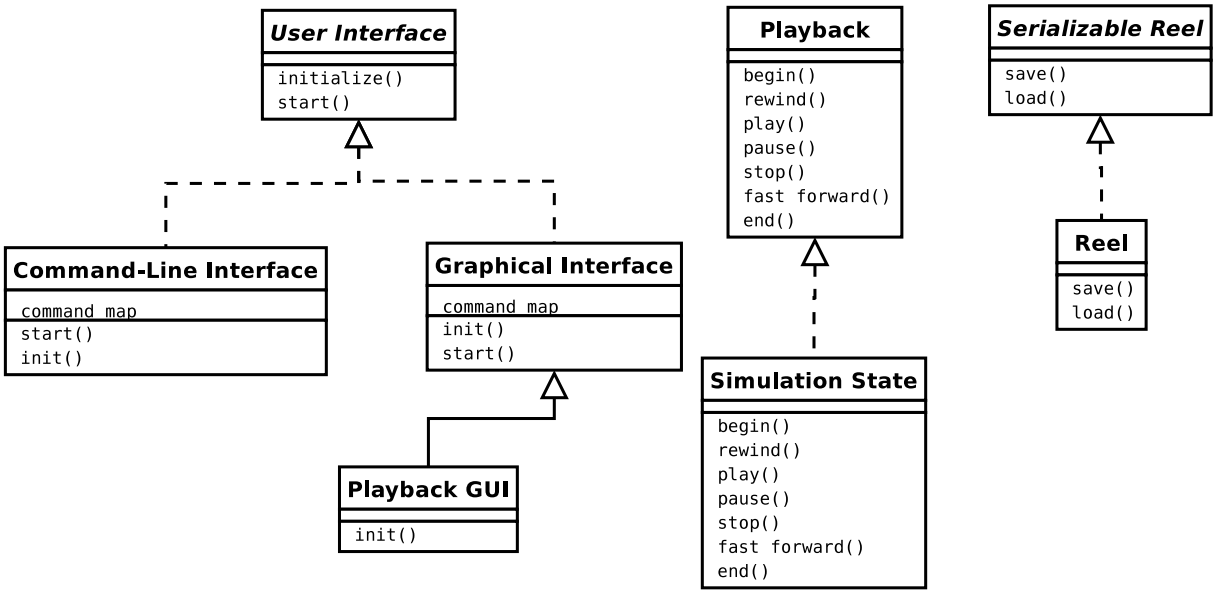



Figure 8: Simplified class diagram of the user interface.

To render a sphere, we could generate a list of triangles or we could use quadrics. Quadrics generate spheres, cylinders, and disks using a quadric equation. First we create a quadric object using `gluNewQuadric`. Next, we specify how the object should be rendered, including texturing. We can render the object using either `gluSphere`, `gluCylinder`, `gluDisk`, or `gluPartialDisk`.

Texture mapping allows us to attach an image to an object by setting texture coordinates to each vertex of a polygon. In addition to making calls to `glVertex`, we also make calls to `glTexCoord` to render a textured polygon.

Without lighting, the objects rendered by OpenGL look flat. For lighting to work, we specify a normal vector for each vertex, create and position a light source, and define the material properties for each object. To specify the normal of a vertex, we call `glNormal`, which works like `glVertex` and `glTexCoord`. We create light sources using `glLight`, by which we create spotlights and ambient lights. Finally, the material determines how the object reflects light. The interaction between materials and lights is difficult to master, so it takes a lot of tweaking to get objects looking right.

### 4.2.1 Ant

Ant is a build tool written in Java to take advantage of Java's platform independence. It was initially written for the Tomcat project, but people saw the advantages of a truly cross-platform build tool, so the Tomcat project released it for use by the general public [1].

There are a few reasons for using Ant. One is that it is cross-platform, so we don't have to deal with getting `make` to install on all platforms. Also, it is easily extensible, allowing any Java developer to quickly add new tasks.

Ant takes an XML file that contains a series of targets. Each target consists of a list of tasks. A typical task would be to call the Java compiler. The following is a sample Ant build file.

```
<?xml version="1.0"?>
<project default="default" name="Rigid Body Simulator">
    <description>
        Build file for my rigid body simulator.
```

```
    </description>

    <property name="srcDir" location="src"/>
    <property name="libDir" location="lib"/>

    <target name="default" depends="rbs"/>

    <target name="rbs_wall">
        <mkdir dir="${libDir}"/>
        <javac srcDir="${srcDir}" classpath="${libDir}" destDir="${libDir}">
            <compilerarg value="-Xlint"/>
        </javac>
    </target>

    <target name="rbs">
        <mkdir dir="${libDir}"/>
        <javac srcDir="${srcDir}" classpath="${libDir}" destDir="${libDir}"/>
    </target>
</project>
```

This Ant build file has three targets: `default`, `rbs_wall`, and `rbs`. By default, Ant builds the `default` target, which depends on the `rbs` target, so it builds the `rbs` target by default. Each target consists of a list of tasks that have to be completed. Both targets do the same tasks, except `rbs_wall` passes an extra argument to the compiler so that all recommended warnings are displayed during compilation.

### 4.2.2 Java Design Patterns

Design patterns are general solutions to common software problems. I used three different design patterns in the design of the rigid-body simulator: the singleton pattern, the decorator pattern, and the factory pattern.

The singleton design pattern insures that only one instance of a given class can be created during the life of the program. The singleton pattern makes the constructor for the class private and adds a method (usually called `getInstance`) that creates a new instance or returns the existing instance. The class through which all threads communicate is a singleton object.

The decorator pattern allows the program to dynamically modify the behavior of an object. The decorator pattern modifies how renderable objects render themselves. One decorator renders objects with a specified color; the other decorator renders objects with a specified texture. The decorator class wraps the original class and overloads methods to cause the class to behave differently (Figure 9). The decorators override the render method, causing textures or colors to be applied to the objects being rendered.

The factory design pattern allows the program to create an object without knowing its exact class. The factory contains static methods that create and return objects based on a string passed to the method. The factory returns objects that adhere to an interface. The simulator uses the factory pattern extensively to create objects, including renderable objects, 3D simulation objects, and different types of frame reels.

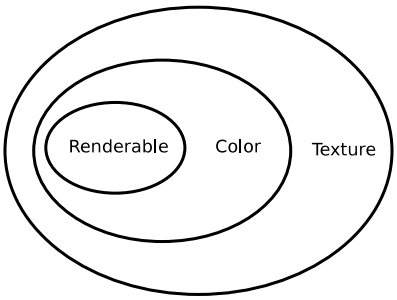



Figure 9: Conceptual representation of decorator pattern.

#### 4.2.3 Java Generics

Java generics constitute an extension to the Java programming language introduced in JDK version 1.5. Generics specify the type of objects that containers should contain. Generics allow Java to perform checks during compilation, ensuring compile-time type safety.

The simulator uses generics during configuration-file parsing. The configuration file object contains a list of settings that map setting names to a generic value. To retrieve a setting, the simulator parses the configuration file and retrieves values by name. The advantage of using generics is that if the setting value does not satisfy its defined type, the simulator throws an error right away.

### 4.3 Future Improvements

There is always room for improvement. One thing that needs to be improved is the object file that specifies the initial state of all of the objects. Currently, it contains a list of strings, integers, and floating-point numbers. The problem is that there is no real structure to the file, and when we try to make a change, we have to look at a reference file to determine what each field means. The object file could be changed to either an XML file or to a properties file. Both types of file are easy to parse, but a properties file is easier to edit by hand, so if an XML file is used, a object configuration utility should be written to create object files. An XML file has one advantage over a properties file: It can use a document type definition (DTD), which verifies the file to be syntactically correct.

To improve the simulator, I would implement selective rollback when object inter-penetration is found. Instead of rolling back all objects to their previous state, we could rollback only objects that inter-penetrate. Then we shrink the time step to resolve the collisions between the inter-penetrating objects. Once the collision is resolved, we could update the inter-penetrating objects to the same time as the others.

Extending the simulator to handle other convex rigid bodies besides planes and spheres would be a major improvement. Collision detection would have to be improved, since computing collisions between arbitrarily shaped rigid bodies is much more difficult than collision detection between two spheres. One method that could be used is the Method of Separating Axes[4, p. 284]. It would also be necessary to extend the `RigidBody` class and the `Renderable` class for each new type of rigid body.

A useful improvement to the simulator would be to add the ability to mount the camera to one of the objects involved in the simulation. This enhancement would involve adding a field to determine what object the camera is mounted to and translating the camera to the proper location for each frame rendered. Another improvement along these lines is allowing the user to move the camera while the simulation is running. This enhancement would involve adding camera controls

to the user interface or tying camera location to the location of the mouse.

A further way the simulator could be extended is to fully simulate gravity instead of assuming everything takes place on earth. If gravity were fully simulated, we could run solar-system simulations. This enhancement would require calculating the gravitational force on each object (see Appendix A) before each frame.

**APPENDIX**

# A Physics

Before building a rigid-body simulator you need to understand the basics of physics. First of all, a "rigid body is characterized by the region that its mass lives in [4, p. 14]." The most basic rigid body is one particle of mass $m$ at a given location. Having a one-particle mass is kind of boring, so most rigid bodies consist of a finite number of particles $p$, with each particle having its own mass. So to find the mass of a rigid body take the sum of all the particles' mass.

$$m_{rigid-body} = \sum_{i=1}^{n} m_i$$

An object behaves as if its mass is concentrated at a single point, called the center of mass [4, p. 44]. To calculate the center of mass of a three dimensional object, the following equations can be used.

$$\bar{x} = \frac{\sum_{i=1}^{n} m_i x_i}{\sum_{i=1}^{n} m_i}$$

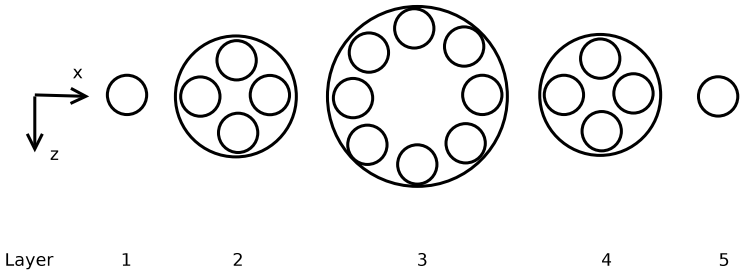
$$\bar{y} = \frac{\sum_{i=1}^{n} m_i y_i}{\sum_{i=1}^{n} m_i}$$

$$\bar{z} = \frac{\sum_{i=1}^{n} m_i z_i}{\sum_{i=1}^{n} m_i}$$

where $n$ is the number of particles in the object, $x_i$, $y_i$, and $z_i$ is the location, and $m_i$ the mass of the $i$th particle.

### A.0.1 Example

1. Calculate the center of mass of the sphere in the following diagram,




To show all of the particles that make up the sphere it has been split into layers. To compose the sphere, layer 5 is stacked on layer 4, which is stacked on layer 3, and so forth. All particles have a mass of $1kg$ except for the particle on layer 1, which has a mass of $2kg$. The center of the sphere is at the origin. The radius of the sphere is $2.5cm$, so each layer is $1cm$ high.

Solution

$$\sum_{i=1}^{18} m_i x_i = 2(-1.77*1kg)+2(1.77*1kg)+(-2.5*1kg)+2(-1.77*1kg)+2(1.77*1kg)+(2.5*1kg) = 0.0cm*kg$$

$$\bar{x} = \frac{0.0cm * kg}{19kg} = 0.0cm$$

$$\sum_{i=1}^{18} m_i y_i = (-2.5 * 2kg) + 4(1.77 * 1kg) + 4(-1.77 * 1kg) + (2.5 * 1kg) = -2.5cm * kg$$

$$\bar{y} = \frac{-2.5cm * kg}{19kg} = -0.13cm$$

$$\sum_{i=1}^{18} m_i z_i = 2(-1.77*1kg)+2(1.77*1kg)+2(1.77*1kg)+2(-1.77*1kg)+(-2.5*1kg)+(2.5*1kg) = 0.0cm*kg$$

$$\bar{z} = \frac{0.0cm * kg}{19kg} = 0.0cm$$

## A.1 Rigid-Body Kinematics

Kinematics describes the translational and rotational motion of objects. Translational motion is tracked by the center of mass, and the orientation is determined by rotation about the center of mass. Let the location of a rigid body be $s$, its velocity be $v = \frac{ds}{dt}$, and its acceleration be $a = \frac{dv}{dt}$. We can use the following equations to track a rigid body under constant acceleration:

$$v_+ = v_- + at$$

$$v_+^2 = 2a(s_+ - s_-) + v_-^2$$

$$s_+ = s_- + tv_- + \frac{1}{2}at^2$$

Let the rotation of a rigid body about its center of mass be $\Omega$, its angular velocity be $\omega = \frac{d\Omega}{dt}$, and its angular acceleration be $\alpha = \frac{d\omega}{dt}$. We can use the following equations to track a rigid body under constant angular acceleration:

$$\omega_+ = \omega_- + \alpha t$$

$$\omega_+^2 = 2\alpha(\Omega_+ - \Omega_-) + \omega_-^2$$

$$\Omega_+ = \Omega_- + t\omega_- + \frac{1}{2}\alpha t^2$$

To compute the velocity of any point of the rigid body,

$$v_{total} = v + (\omega \times r)$$

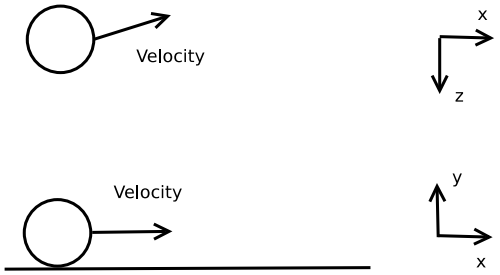
where $r$ is the distance of the point from the center of mass. To compute the acceleration of any point of the rigid body,

$$a_{total} = a + (\omega \times (\omega \times r)) + (\alpha \times r)$$

which is the sum of center of mass acceleration, tangential acceleration, and centripetal acceleration.

### A.1.1 Examples

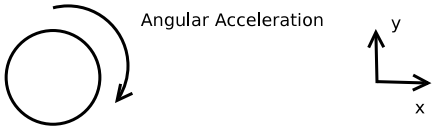
1. Calculate the final position of a sphere that starts out at point $(-5.0, 2.5, 0.0)m$, has a velocity of $(3.0, 0.0, -1.0)m/s$, and slides for $2s$. Assume there is no friction.




Solution

$$s_{x+} = -5.0m + 2s * 3.0m/s$$
$$s_{y+} = 2.5m + 2s * 0.0m/s$$
$$s_{z+} = 0.0m + 2s * -1.0m/s$$
$$s_+ = (s_{x+}, s_{y+}, s_{z+}) = (1.0, 2.5, -2.0)m$$

2. Calculate the final orientation of a sphere with no initial angular velocity, a constant angular acceleration of $(0.0, 0.0, 3.0)rad/s^2$, which spins for $2s$. Assume there is no friction.




Solution

$$\Omega_{x+} = 0.0rad + 2s * 0.0rad/s + 1/2 * 0.0rad/s^2 * 4s^2$$
$$\Omega_{y+} = 0.0rad + 2s * 0.0rad/s + 1/2 * 0.0rad/s^2 * 4s^2$$
$$\Omega_{z+} = 0.0rad + 2s * 0.0rad/s + 1/2 * 3.0rad/s^2 * 4s^2$$
$$\Omega_+ = (\Omega_{x+}, \Omega_{y+}, \Omega_{z+}) = (0.0, 0.0, 6.0)rad$$

3. Calculate the final velocity and the final acceleration of any point on the surface of the rotating sphere in the previous example. Assume the sphere has a radius of $2.5cm$ and there is no friction.

Solution

$$\omega_+ = (0.0, 0.0, 0.0)rad/s + (0.0, 0.0, 3.0)rad/s^2 * 2s$$
$$\omega_+ = (0.0, 0.0, 6.0)rad/s$$
$$v_{total} = (0.0, 0.0, 0.0)m/s + ((0.0, 0.0, 6.0)rad/s \times (0.0, 0.025, 0.0)m)$$
$$v_{total} = (-0.15, 0.0, 0.0)m/s$$
$$a_{total} = 0.0m/s^2 + ((0.0, 0.0, 6.0)rad/s \times ((-0.15, 0.0, 0.0)m/s) + ((0.0, 0.0, 3.0)rad/s^2 \times (0.0, 0.025, 0.0)m)$$
$$a_{total} = 0.0m/s^2 + (0.0, -0.9, 0.0)m/s^2 + (-0.075, 0.0, 0.0)m/s^2$$
$$a_{total} = (-0.075, -0.9, 0.0)m/s^2$$

## A.2 Forces

A force is a push or pull that results in motion of the the affected body. This section examines some common forces.

### A.2.1 Newton's Three Laws

Newton's three laws relate forces acting on an object with the bodies resulting motion.

1. An object at rest remains at rest, and an object in motion will continue in motion with a constant velocity unless it experiences a net external force [5, p. 109].

2. The acceleration of an object is directly proportional to the net force acting on it and inversely proportional to its mass [5, p. 111]. ($F = ma$)

3. If two bodies interact, the force exerted on body 1 by body 2 is equal to and opposite the force exerted on body 2 by body 1:
$$F_{12} = -F21$$
[5, p. 114].

### A.2.2 Gravitational Forces

A gravitational force is an attractive force that occurs between two objects, determined by

$$F_{gravity} = \frac{Gm_1m_2}{r^2}$$

where $G$ is a constant with value $6.67x10^{-11}N \cdot m^2$ and $r$ is the distance between the objects. For an object on earth, acceleration due to gravity is $9.81\frac{m}{s^2}$, which is approximately constant.

### A.2.3 Frictional Force

A frictional force opposes the sliding of one object over the surface of an adjacent object. The force acts tangent to the surfaces in contact and opposite to the direction of motion. The magnitude of the frictional force is determined by

$$F_{friction} = \mu N$$

where $\mu$ is the dynamic coefficient of friction, a surface-dependent constant, and $N$ is the normal force between the objects.

### A.2.4 Torque

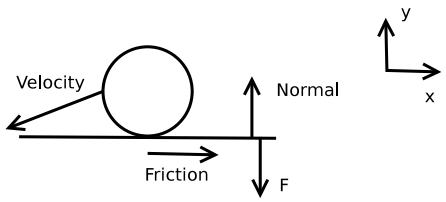
Torque is the force exerted on a object pivoted about an axis. For example, when one removes lug nuts from a car wheel, if the wheel is not on the ground, the torque causes the wheel to spin. The direction of torque is determined by the right-hand rule, and its magnitude is given by

$$\tau = r \times F$$

where $r$ is the distance from the object and $F$ is the force applied.

### A.2.5 Examples

1. Calculate the frictional force on the sphere from the first example when it strikes a plane with a velocity of $(-1.0, -2.0, 0.0)m/s$, given that the coefficient of friction between the sphere and the plane is 0.25.




Solution

$$N_1 = 19kg * -9.81m/s^2 = -186.39N$$
$$19kg * -2.0m/s = -19kg * v_+$$
$$v_+ = 2m/s$$
$$a = 4m/s^2$$
$$N_2 = 19kg * 4m/s^2 = 76N$$
$$N = -1.0 * (N_1 + N_2) = 110.39N$$
$$F_{friction} = 0.25 * 110.39N = 27.6N$$
$$F_{normal} = (1.0, 0.0, 0.0)$$

2. Calculate the torque on the sphere caused by the frictional force from the previous example. Assume the we are using the sphere from the first example, except that all the particles that compose the sphere have the same mass, $1kg$.

Solution

$$\tau = (0.0, -0.025, 0.0)m \times (27.6, 0.0, 0.0)N$$
$$\tau = (0.0, 0.0, 0.69)N \cdot m$$

## A.3 Momentum

Momentum is a measure of the power within a moving object.

### A.3.1 Linear Momentum

Application of an external force on an object causes the object to move, which is a change in momentum. Linear momentum is also the tendency of an object to move in a straight line unless acted upon by an external force. Its value is determined by the following equation.

$$p = mv$$

where $m$ is the mass and $v$ is the velocity of the object.

Linear momentum is conserved if the net external force on a system of objects is zero. This fact can be used to solve for the resultant velocities of two objects upon collision.

Linear momentum is not conserved when friction is present, because friction causes the net external force on an object to be nonzero.

### A.3.2 Angular Momentum

Angular momentum is the tendency of an object to remain spinning unless acted upon by an external force. Its value is determined by the following equations.

$$L = r \times mv$$

$$L = I\omega$$

Angular momentum $L$ is a vector perpendicular to the plane formed by the radius $r$ and the velocity $v$. Angular momentum, like linear momentum, is conserved if the net external force on a system of objects is zero. Also, like linear momentum, angular momentum is not conserved when friction is present.

### A.3.3 Moment of Inertia

The moment of inertia is the measure of rotational inertia of a body about an axis. It is like the rotational mass of an object. It can be thought of as an object's resistance to rotation. Given an object composed of $n$ particles, each having mass $m_i$ the following equations can be used to calculate the moment of inertia.

$$I_{xx} = \sum_{i=1}^{n} m_i(y_i^2 + z_i^2)$$

$$I_{yy} = \sum_{i=1}^{n} m_i(x_i^2 + z_i^2)$$

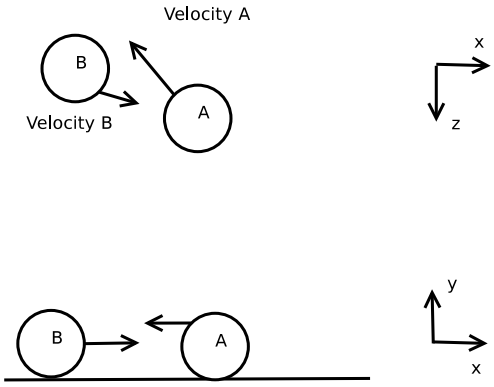
$$I_{zz} = \sum_{i=1}^{n} m_i(x_i^2 + y_i^2)$$

Standard-shaped objects have standard formulas for the moment of inertia. For example, the moment of inertia of a uniformly dense sphere is

$$I_{xx} = I_{yy} = I_{zz} = (2/5)mr^2$$

### A.3.4 Examples

1. Calculate the new velocities of sphere $A$ and sphere $B$ after they collide. The radius of both spheres is $2.5cm$, and the center of mass of both spheres is in the geometrical center. Sphere $B$ has a mass of $18kg$, a velocity of $(2.0, 0.0, 0.0)m/s$, and is located at $(0.0, 0.0, 0.0)$ just before the collision. Sphere $A$ has a mass of $2kg$, a velocity of $(-1.0, 0.0, 0.0)m/s$, and is located at $(5.0, 0.0, 0.0)$ just before the collision. Assume there is no friction between the spheres.




Solution

$$pa_- + pb_- = pa_+ + pb_+$$
$$34 = 18kg * v_{b+} + 2kg * v_{a+}$$

Use coefficient of restitution

$$v_{a+} = 3m/s - v_{b+}$$
$$34 = 18v_{b+} + 6 - 2v_{b+}$$
$$v_{b+} = 1.75m/s$$
$$v_{a+} = 1.25m/s$$

2. Calculate the angular momentum of the sphere in Example section A.1.1 problem 2 after $2s$ of spinning. Assuming the sphere has a radius of $2.5cm$.

Solution

$$L = r \times mv$$
$$v_{total} = (-0.15, 0.0, 0.0)m/s$$
$$L = (0.0, 0.025, 0.0)m \times 19kg * (-0.15, 0.0, 0.0)m/s$$
$$L = (0.0, 0.025, 0.0)m \times (-2.85, 0.0, 0.0)m/s$$
$$L = (0.0, 0.0, 0.07)N \cdot m \cdot s$$

3. Calculate the moment of inertia of the sphere in the first example using the general equations. Then calculate the moment of inertia of a sphere of uniform density with mass $19kg$ and a radius of $2.5m$.

Solution

$$I_{xx} = 5(2.5^2) + 4(1.77^2 + 177^2) + 8(1.77^2)$$
$$I_{xx} = 31.25 + 25.04 + 25.04 = 81.33kg \cdot m^2$$
$$I_{yy} = 8(1.77^2) + 4(2.5^2) + 4(1.77^2 + 1.77^2)$$
$$I_{yy} = 25.04 + 25 + 25.04 = 75.08kg \cdot m^2$$
$$I_{zz} = 5(2.5^2) + 8(1.77^2) + 4(1.77^2 + 1.77^2)$$
$$I_{zz} = 81.33kg \cdot m^2$$

$$I_{xx} = I_{yy} = I_{zz} = (2/5)mr^2$$
$$I_{xx} = I_{yy} = I_{zz} = (2/5)19 * 2.5^2$$
$$I_{xx} = I_{yy} = I_{zz} = 47.5kg \cdot m^2$$

## A.4 Energy

Energy is the ability to do work.

### A.4.1 Kinetic Energy

Kinetic energy is energy associated with a moving particle. It is the sum of translational kinetic energy and rotational kinetic energy. Kinetic energy can be computed by

$$KE = (1/2)mv^2 + (1/2)I\omega^2$$

where $(1/2)mv^2$ is the translational kinetic energy and $(1/2)I\omega^2$ is the rotational kinetic energy.

### A.4.2 Potential Energy

Potential energy is the energy associated with the position of a particle. The most common form of potential energy is gravitational potential energy, which is computed with the following equation.

$$PE = mgh$$

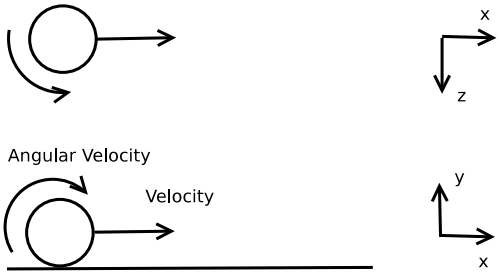
where $g$ is the acceleration due to gravity and $h$ is the height above the surface of the earth.

### A.4.3 Conservation of Energy

The total mechanical energy of a system is the sum of all kinetic energy and all potential energy. Mechanical energy of an object is not conserved when friction is present, since energy is lost in the form of heat, sound, and deformation. When friction is present, the amount of energy lost to friction can be computed by subtracting the total mechanical energy after from the total mechanical energy before.

### A.4.4 Examples

1. Calculate the total kinetic energy of a uniformly dense sphere with a mass of $18kg$ and a radius of $2.5cm$. The sphere has a translational velocity of $(5.0, 0.0, 0.0)m/s$ and a angular velocity of $(0.0, 3.0, 1.0)rad/s$.

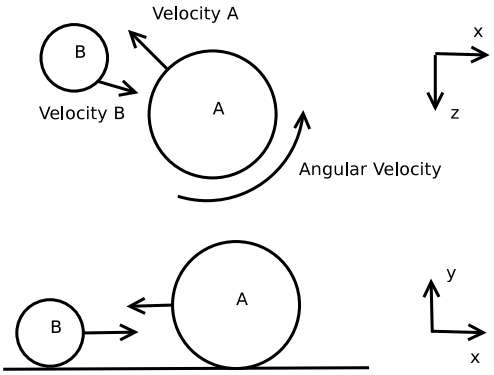



Solution

$$KE = (1/2)18 * 5^2 + (1/2)0.004 * (0.0, 3.0, 1.0)^2$$

$$KE = (225, 0.0, 0.0) + (0.0, 0.018, 0.002) = (225, 0.018, 0.002)joules$$

2. Calculate the total mechanical energy just before and just after the collision between sphere $A$ and sphere $B$. Carry out the calculations once assuming there is no friction and once taking friction into account. Sphere $A$ has a mass of $18kg$, a radius of $5m$, a velocity of $(-2.0, 0.0, -1.0)m/s$, and is located at $(0.0, 5.0, 0.0)$ just before impact. Sphere $B$ has a mass of $3kg$, a radius of $2.5m$, a velocity of $(1.0, 0.0, 3.0)m/s$, and is located at $(7.07, 2.5, 0.0)$ just before impact. The coefficient of friction between the two spheres is 0.3.



## References


[1] Apache Ant Freqently Asked Questions, 2008. `http://ant.apache.org/faq.html`.

[2] The JOGL API Project, 2008. `https://jogl.dev.java.net`.

[3] David M. Bourg. *Physics for Game Developers*. O'Reilly Media Inc., 1005 Gravenstein Highway North, Sebasopol, CA 95472, 2002.

[4] David H. Eberly. *Game Physics*. Morgan Kaufmann Publishers, San Francisco, CA 94111, 2004.

[5] Raymond A. Serway. *Physics for Scientists and Engineers, Fourth Edition*. Saunders College Publishing, Philadelphia, PA, 1996.