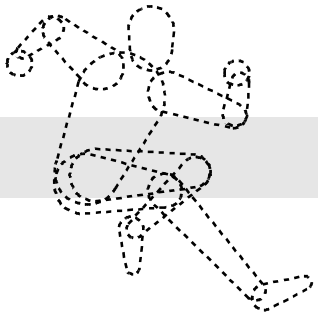

11

The Rigid-Body Physics Engine

Our physics engine is now capable of simulating rigid bodies in full 3D. The spring forces and other force generators will work with this approach, but the hard constraints we discussed in Chapter 7 will not. We will look at collision detection in Part IV and then return to full 3D constraints in Part V.

Even without hard constraints, there is still a lot we can do. This chapter will look at two applications of physics that don't rely on hard constraints for their effects: boats and aircraft. We'll build a flight simulator and a boat model. Adding the aerodynamics from the flight simulator allows us to build a sailing simulation.

11.1 Overview of the Engine

The rigid-body physics engine has two components:

1. The rigid bodies themselves keep track of their position and movement, and their mass characteristics. To set up a simulation, we need to work out what rigid bodies are needed, and set their initial position, orientation, and velocities (both linear and angular). We also need to set their inverse mass and inverse inertia tensor. The acceleration of an object due to gravity is also held in the rigid body (this could be removed and replaced by a force, if you so desire).
2. The force generators are used to keep track of forces that apply over several frames of the game.

DOI: 10.1016/B978-0-12-381976-5.00011-5

We have removed the contact resolution system from the mass aggregate system (it will be reintroduced in Parts IV and V).

We can use the system we introduced in Chapter 8 to manage the objects to be simulated. In this case, however, they are rigid bodies rather than particles.

The `World` structure is modified accordingly:

Excerpt from file include/cyclone/world.h

```
/**
 * The world represents an independent simulation of physics.  It
 * keeps track of a set of rigid bodies, and provides the means to
 * update them all.
 */
class World
{
public:
    typedef std::vector<RigidBody*> RigidBodies;

protected:
    /**
     * Holds the rigid bodies being simulated.
     */
    RigidBodies bodies;

};
```

As before, each frame of the `startFrame` method is first called, which sets up each object by zeroing its force and torque accumulators, and calculating its derived quantities as follows:

Excerpt from file include/cyclone/world.h

```
class World
{
    // ... other World data as before ...

    /**
     * Initializes the world for a simulation frame. This clears
     * the force and torque accumulators for bodies in the
     * world. After calling this, the bodies can have their forces
     * and torques for this frame added.
     */
    void startFrame();
};
```

Excerpt from file src/world.cpp

```
void World::startFrame()
{
    for (RigidBodies::iterator b = bodies.begin();
        b != bodies.end();
        b++)
    {
        b->clearAccumulators();
        b->calculateDerivedData();
    }
}
```

And again, additional forces can be applied after calling this method.

To execute the physics, the `runPhysics` method is called. This calls all the force generators to apply their forces and performs the integration of all objects:

Excerpt from file include/cyclone/world.h

```
class World
{
    // ... other World data as before ...

    /**
     * Processes all the physics for the world.
     */
    void runPhysics(real duration);
};
```

Excerpt from file src/world.cpp

```
void ParticleWorld::integrate(real duration)
{
    for (RigidBodies::iterator b = bodies.begin();
        b != bodies.end();
        b++)
    {
        // Integrate the body by the given duration.
        b->integrate(duration);
    }
}

void World::runPhysics(real duration)
{
    // First, apply the force generators.
```

```
    registry.updateForces(duration);

    // Then integrate the objects.
    integrate(duration);
}
```

It no longer calls the collision detection system. The calls to `startFrame` and `runPhysics` can occur in the same place in the game loop.

Note that I've made an additional call to the `updateTransform` method of each object. The object may have moved during the update (and in later sections during collision resolution), so its transform matrix needs updating before it is rendered. Each object is then rendered in turn using the rigid body's transform.

11.2 Using the Physics Engine

Both our sample programs for this physics engine use aerodynamics. We will create a new force generator that can fake some important features of flight aerodynamics, or enough to produce a basic flight model suitable for use in a flight action game. We will use the same generator to drive a sail model for a sailing simulator.

11.2.1 A Flight Simulator

There is no need for contact physics in a flight simulator, except with the ground, of course. Many flight simulators assume that if you hit something in an airplane, then it's all over: a crash animation plays and the player starts again. This makes it a perfect exercise for our current engine.

The dynamics of an aircraft are generated by the way air flows over its surfaces (both the surfaces that don't move relative to the center of mass, like the fuselage, and control surfaces that can be made to move or change shape, like the wings and rudder). The flow of air causes forces to be generated. Some forces, like drag, act in the same direction as the aircraft is moving. The most important force, lift, acts at right angles to the flow of air. As the aircraft's surfaces move at different angles through the air, the proportion of each kind of force can change dramatically. If the wing is slicing through the air, it generates lift, but if it is moving vertically through the air, then it generates no lift. We'd like to be able to capture this kind of behavior in a force generator that can produce sensible aerodynamic forces.

The Aerodynamic Tensor

To model the aerodynamic forces properly is very complex. The behavior of a real aircraft depends on the fluid dynamics of air movement. This is a horrendously complex discipline involving mathematics well beyond the scope of this book. To create a truely realistic flight simulator involves some specialized physics that I don't want to venture into.

To make our lives easier, I will use a simplification: the "aerodynamic tensor." The aerodynamic tensor is a way of calculating the total force that a surface of the airplane is generating based only on the speed of the air that is moving over it.

The tensor is a 3×3 matrix, exactly as we used for the inertia tensor. We start with a wind speed vector, and transform it using the tensor to give a force vector:

$$\boldsymbol{f}_a = A\boldsymbol{v}_w$$

where f_a is the resulting force, A is the aerodynamic tensor, and $\boldsymbol{v}_w$ is the velocity of the air. Just like we saw for the inertia tensor, we have to be careful of coordinates here. The velocity of the air and the resulting force are both expressed in world coordinates, but the aerodynamic tensor is in object coordinates. Again we need to change the basis of the tensor in each frame before applying this function.

To fly the plane we can implement control surfaces in one of two ways. The first, and most accurate way, is to have two tensors representing the aerodynamic characteristics when the surface is at its two extremes. At each frame, the current position of the control surface is used to blend the two tensors to create a tensor for the current surface position.

In practice, three tensors are sometimes needed to represent the two extremes plus the normal position of the control surface (which often has a quite different, and not intermediate, behavior). For example, a wing with its aileron (the control surface on the back of each wing) in line with the wing produces lots of lift, and only a modest amount of drag. With the aileron out of this position, either up or down, the drag increases dramatically, but the lift can be boosted or cut (depending on whether it is up or down).

The second approach is to actually tilt the entire surface slightly. We can do this by storing an orientation for the aerodynamic surface, and allowing the player to directly control some of this orientation. To simulate the aileron on the wing, the player might be effectively tilting the entire wing. As the wing changes orientation, the air flow over it will change, and its single aerodynamic tensor will generate correspondingly different forces.

The Aerodynamic Surface

We can implement an aerodynamic force generator using this technique. The force generator is created with an aerodynamic tensor, and it is attached to the rigid body at a given point. This is the point at which all its force will be felt. We can attach as many surfaces as we need to one rigid body. The force generator looks like this:

Excerpt from file include/cyclone/fgen.h

```
/**
 * A force generator that applies an aerodynamic force.
 */
class Aero : public ForceGenerator
{
```

```
protected:
    /**
     * Holds the aerodynamic tensor for the surface in body
     * space.
     */
    Matrix3 tensor;

    /**
     * Holds the relative position of the aerodynamic surface in
     * body coordinates.
     */
    Vector3 position;

    /**
     * Holds a pointer to a vector containing the wind speed of the
     * environment. This is easier than managing a separate
     * wind speed vector per generator and having to update it
     * manually as the wind changes.
     */
    const Vector3* windspeed;

public:
    /**
     * Creates a new aerodynamic force generator with the
     * given properties.
     */
    Aero(const Matrix3 &tensor, const Vector3 &position,
         const Vector3 *windspeed);

    /**
     * Applies the force to the given rigid body.
     */
    virtual void updateForce(RigidBody *body, real duration);

protected:
    /**
     * Uses an explicit tensor matrix to update the force on
     * the given rigid body. This is exactly the same as for updateForce,
     * except that it takes an explicit tensor.
     */
    void updateForceFromTensor(RigidBody *body, real duration,
                               const Matrix3 &tensor);
};
```

Excerpt from file src/fgen.cpp

```
void Aero::updateForce(RigidBody *body, real duration)
{
    Aero::updateForceFromTensor(body, duration, tensor);
}

void Aero::updateForceFromTensor(RigidBody *body, real duration,
                                 const Matrix3 &tensor)
{
    // Calculate total velocity (wind speed and body's velocity).
    Vector3 velocity = body->getVelocity();
    velocity += *windspeed;

    // Calculate the velocity in body coordinates.
    Vector3 bodyVel =
      body->getTransform().transformInverseDirection(velocity);

    // Calculate the force in body coordinates.
    Vector3 bodyForce = tensor.transform(bodyVel);
    Vector3 force = body->getTransform().transformDirection(bodyForce);

    // Apply the force.
    body->addForceAtBodyPoint(force, position);
}
```

The air velocity is calculated based on two values: the prevailing wind and the velocity of the rigid body. The prevailing wind is a vector, containing both the direction and magnitude of the wind. If the rigid body were not moving, it would still feel this wind. We could omit this value for a flight game that doesn't need to complicate the player's task by adding wind. It will become very useful when we come to model our sailing simulator in the next section, however.

This implementation uses a single tensor only. To implement control surfaces, we need to extend this in one of the ways we looked at above. I will choose the more accurate approach with three tensors to represent the characteristics of the surface at the extremes of its operation:

Excerpt from file include/cyclone/fgen.h

```
/**
* A force generator with a control aerodynamic surface. This
* requires three inertia tensors, for the two extremes and
* ''resting'' position of the control surface. The latter tensor is
* the one inherited from the base class, while the two extremes are
* defined in this class.
*/
```

```
class AeroControl : public Aero
{
protected:
    /**
     * The aerodynamic tensor for the surface when the control is at
     * its maximum value.
     */
    Matrix3 maxTensor;

    /**
     * The aerodynamic tensor for the surface when the control is at
     * its minimum value.
     */
    Matrix3 minTensor;

    /**
    * The current position of the control for this surface. This
    * should range between -1 (in which case the minTensor value
    * is used), through 0 (where the base-class tensor value is
    * used) to +1 (where the maxTensor value is used).
    */
    real controlSetting;

private:
    /**
     * Calculates the final aerodynamic tensor for the current
     * control setting.
     */
    Matrix3 getTensor();

public:
    /**
     * Creates a new aerodynamic control surface with the given
     * properties.
     */
    AeroControl(const Matrix3 &base,
                const Matrix3 &min, const Matrix3 &max,
                const Vector3 &position, const Vector3 *windspeed);

    /**
     * Sets the control position of this control. This * should
    range between -1 (in which case the minTensor value * is
    used), through 0 (where the base-class tensor value is used) *
    to +1 (where the maxTensor value is used). Values outside that
```

```
    * range give undefined results.
    */
    void setControl(real value);

    /**
     * Applies the force to the given rigid body.
     */
    virtual void updateForce(RigidBody *body, real duration);
};
```

Excerpt from file src/fgen.cpp

```
Matrix3 AeroControl::getTensor()
{
    if (controlSetting <= -1.0f) return minTensor;
    else if (controlSetting >= 1.0f) return maxTensor;
    else if (controlSetting < 0)
    {
        return Matrix3::linearInterpolate(minTensor,
                                          tensor,
                                          controlSetting+1.0f);
    }
    else if (controlSetting > 0)
    {
        return Matrix3::linearInterpolate(tensor,
                                          maxTensor,
                                          controlSetting);
    }
    else return tensor;
}

void AeroControl::updateForce(RigidBody *body, real duration)
{
    Matrix3 tensor = getTensor();
    Aero::updateForceFromTensor(body, duration, tensor);
}
```

The `linearInterpolate` method is defined on the `Matrix3` class as follows:

Excerpt from file include/cyclone/core.h

```
class Matrix3
{
    // ... Other Matrix3 code as before ...
```

```
    /**
     * Interpolates a couple of matrices.
     */
    static Matrix3 linearInterpolate(const Matrix3& a,
                                     const Matrix3& b,
                                     real prop);
};
```

Excerpt from file src/core.cpp

```
Matrix3 Matrix3::linearInterpolate(const Matrix3& a,
                                   const Matrix3& b,
                                   real prop)
{
    Matrix3 result;
    real omp = 1.0 - prop;
    for (unsigned i = 0; i < 9; i++) {
        result.data[i] = a.data[i] * omp + b.data[i] * prop;
    }
    return result;
}
```

Each control surface has an input wired to the player's (or AI's) control. It ranges from -1 to $+1$, where 0 is considered the "normal" position. The three tensors match these three positions. Two of the three tensors are blended together to form a current aerodynamic tensor for the setting of the surface. This tensor is then converted into world coordinates, and used as before.

Putting It Together

In the sample code, the **flightsim** demo shows this force generator in operation. You control a model aircraft (seen from the ground, for a bit of added challenge). The only forces applied to the aircraft are gravity (represented as an acceleration value) and the aerodynamic forces from surface and control-surface force generators. Figure 11.1 shows the aircraft in action.

I have used four control surfaces: two wings, a tailplane, and a rudder. The tailplane is a regular surface force generator, with no control inputs (in a real plane the tailplane usually does have control surfaces, but we don't need them). It has the following aerodynamic tensor:

$$A = \begin{bmatrix} -0.1 & 0 & 0 \\ 1 & -0.5 & 0 \\ 0 & 0 & -0.1 \end{bmatrix}$$

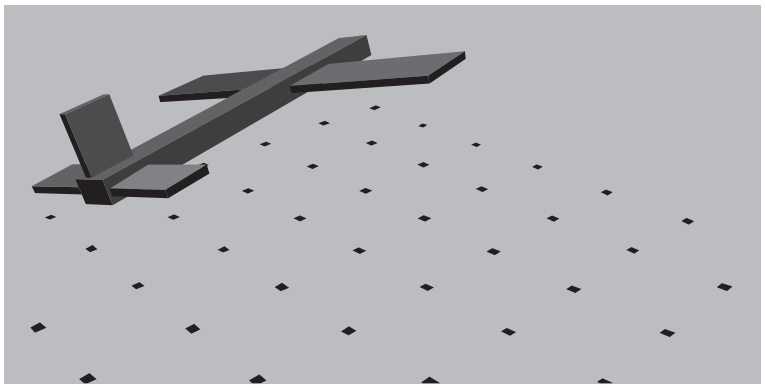

FIGURE 11.1 Screenshot of the **flightsim** demo.

Each wing has an identical control surface force generator. I have used two so that their control surfaces can be operated independently. They use the following aerodynamic tensors for each extreme of the control input:

$$A_{-1} = \begin{bmatrix} -0.2 & 0 & 0 \\ -0.2 & -0.5 & 0 \\ 0 & 0 & -0.1 \end{bmatrix}$$

$$A_0 = \begin{bmatrix} -0.1 & 0 & 0 \\ 1 & -0.5 & 0 \\ 0 & 0 & -0.1 \end{bmatrix}$$

$$A_1 = \begin{bmatrix} -0.2 & 0 & 0 \\ 1.4 & -0.5 & 0 \\ 0 & 0 & -0.1 \end{bmatrix}$$

When the player banks the aircraft, both wing controls work in the same direction. When the player rolls, the controls work in opposition.

Finally, I have added a rudder, a vertical control surface to regulate the yaw of the aircraft. It has the following tensors:

$$A_{-1} = \begin{bmatrix} -0.1 & 0 & -0.4 \\ 0 & -0.1 & 0 \\ 0 & 0 & -0.5 \end{bmatrix}$$

$$A_0 = \begin{bmatrix} -0.1 & 0 & 0 \\ 0 & -0.1 & 0 \\ 0 & 0 & -0.5 \end{bmatrix}$$

$$A_1 = \begin{bmatrix} -0.1 & 0 & 0.4 \\ 0 & -0.1 & 0 \\ 0 & 0 & -0.5 \end{bmatrix}$$

The surfaces are added to the aircraft in a simple setup function and the game loop is exactly as we've seen it before. The user input is passed to the game as it is received by the software (this is a function of the OpenGL system we are using to run the demos; in some engines you may have to call a function to explicitly ask for input). The input directly controls the current values for each control surface.

The full code for the demo can be found in the accompanying source.

11.2.2 A Sailing Simulator

Boat racing is another genre that doesn't require hard constraints, at least in its simplest form: if we want close racing with bumping boats, then we may need to add more complex collision support. For our purposes, we'll implement a simple sailing simulator for a single player.

The aerodynamics of the sail is very similar to the aerodynamics we used for flight simulation. We'll come back to the sail-specific setup in a moment, after we look at the floating behavior of the boat.

Buoyancy

What needs revisiting at this point is our buoyancy model. In Section 6.2.4, we created a buoyancy force generator to act on a particle. We need to extend this to cope with rigid bodies.

Recall that a submerged shape has a buoyancy that depends on the mass of the water it displaces. If that mass of water is greater than the mass of the object, then the net force will be upward and the object will float. The buoyancy force depends only on the volume of the object that is submerged. We approximated this by treating buoyancy like a spring: as the object is gradually more submerged, the force increases until it is considered to be completely underwater, whereupon the force is at its maximum. It doesn't increase with further depth. This is an approximation because it doesn't take into account the shape of the object being submerged.

In our original buoyancy generator, force directly acted on the particle. This is fine for representing balls or other regular objects. On a real boat, however, the buoyancy does two jobs: it keeps the boat afloat and upright. In other words, if the boat begins to lean over (say a gust of wind catches it), more of one side of the boat will be submerged and the added buoyancy on this side of the boat will act to right it.

This tendency to stay upright is caused by the torque component of the buoyancy force. Its linear component keeps the boat afloat, and its torque keeps it vertical. It does this because, unlike in our particle force generator, the buoyancy force doesn't act at the center of gravity.

A submerged part of the boat will have a center of buoyancy, as shown in Figure 11.2. The center of buoyancy is the point at which the buoyancy force can be thought to be acting. Like the buoyancy force itself, the center of buoyancy is related to the displaced water. The center of mass of the displaced water is the same as the center of buoyancy that it generates.

So just as the volume of water displaced depends on the shape of the submerged object, so does the center of buoyancy. The further the center of buoyancy is from the center of mass, the more torque will be generated and the better the boat will be at righting itself. If the center of mass is above the center of buoyancy, then the torque will apply in the opposite direction, and the buoyancy will act to topple the boat.

How do we simulate this in a game? We don't want to get into the messy details of the shape of the water being displaced, and finding its center of mass. Instead, we can simply fix the center of buoyancy to the rigid body. In a real boat, the center of buoyancy will move around as the boat pitches and rolls and a different volume of water is displaced. Most boats are designed so that this variation is minimized, however. Fixing the center of buoyancy doesn't look odd for most games; it shows itself mostly with big waves, but can be easily remedied, as we'll see below.

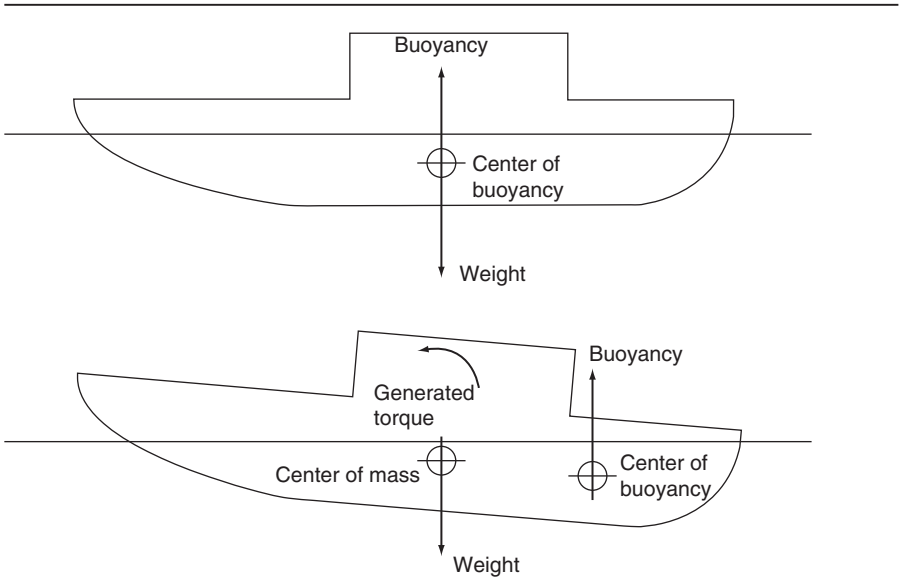

FIGURE 11.2 Different centers of buoyancy.

Our buoyancy force generator can be updated to take an attachment point; otherwise, it is as before:

Excerpt from file include/cyclone/fgen.h

```
/**
 * A force generator to apply a buoyant force to a rigid body.
 */
class Buoyancy : public ForceGenerator
{
    /**
     * The maximum submersion depth of the object before
     * it generates its maximum buoyancy force.
     */
    real maxDepth;

    /**
     * The volume of the object.
     */
    real volume;

    /**
     * The height of the water plane above y=0. The plane will be
     * parallel to the XZ plane.
     */
    real waterHeight;

    /**
     * The density of the liquid. Pure water has a density of
     * 1000 kg per cubic meter.
     */
    real liquidDensity;

    /**
     * The center of buoyancy of the rigid body, in body coordinates.
     */
    Vector3 centreOfBuoyancy;

public:

    /** Creates a new buoyancy force with the given parameters. */
    Buoyancy(const Vector3 &cOfB,
        real maxDepth, real volume, real waterHeight,
        real liquidDensity = 1000.0f);
```

```
    /**
     * Applies the force to the given rigid body.
     */
    virtual void updateForce(RigidBody *body, real duration);
};
```

There is nothing to stop us from attaching multiple buoyancy force generators to a boat to represent different parts of the hull. This allows us to simulate some of the shift in the center of buoyancy. If we have two buoyancy force generators, one at the front (fore) and one at the rear (aft) of a boat, then as it pitches forward and back (through waves, for example) the fore and aft generators will be at different depths in the water and will therefore generate different forces. The highly submerged front of the boat will pitch up rapidly and believably. Without multiple attachments, this wouldn't be anywhere near as believable, and may be obviously inaccurate.

For our sailing simulator, we will use a catamaran with two hulls and four buoyancy force generators, including one fore and one aft on each hull.

The Sail, Rudder, and Hydrofoils

We will use aerodynamics to provide both the sail and the rudder for our boat. The rudder is like the rudder on the aircraft: it acts to keep the boat going straight (or to turn under the command of the player). On many sailboats, there is both a rudder and a dagger board. The dagger board is a large vertical fin that keeps the boat moving in a straight line and keeps it from easily tipping over when the wind catches the sail. The rudder is a smaller vertical fin that can be tilted for turning. For our needs we can combine the two into one. In fact, in many high-performance sailing boats the two are combined in a single structure.

The sail is the main driving force of the boat, as it converts wind into forward motion. It acts very much like an aircraft wing, turning air flow into lift. In the case of a sailing boat, the lift is used to propel the boat forward. There is a misconception that the sail simply catches the air and the wind drags the boat forward. This can be achieved, certainly, and downwind an extra sail (the spinnaker) is often deployed to increase the aerodynamic drag of the boat, and cause it to be pulled along relative to the water. In most cases, however, the sail acts more like a wing than a parachute. In fact, the fastest boats can achieve incredible lift from their sails, and travel considerably faster than the wind speed.

Both the rudder and the sail are control surfaces; they can be adjusted to get the best performance. They are both rotated, rather than having pop-up control surfaces to modify their behavior (although the sail can have its tension adjusted on some boats). We will therefore implement a force generator for control surfaces using the second possible adjustment approach from Section 11.2.1: rotating the control surface. The force generator looks like this:

Excerpt from file include/cyclone/fgen.h

```
/**
 * A force generator with an aerodynamic surface that can be
 * reoriented relative to its rigid body.
 */
class AngledAero : public Aero {
    /**
     * Holds the orientation of the aerodynamic surface relative
     * to the rigid body to which it is attached.
     */
    Quaternion orientation;

public:
    /**
     * Creates a new aerodynamic surface with the given properties.
     */
    AngledAero(const Matrix3 &tensor, const Vector3 &position,
         const Vector3 *windspeed);

    /**
     * Sets the relative orientation of the aerodynamic surface
     * relative to the rigid body that it is attached to. Note that
     * this doesn't affect the point of connection of the surface
     * to the body.
     */
    void setOrientation(const Quaternion &quat);

    /**
     * Applies the force to the given rigid body.
     */
    virtual void updateForce(RigidBody *body, real duration);
};
```

Note that the force generator keeps an orientation for the surface, and uses this, in combination with the orientation of the rigid body, to create a final transformation for the aerodynamic surface. There is only one tensor, but the matrix by which it is transformed is now the combination of the rigid body's orientaton and the adjustable orientation of the control surface.

Although I won't add them in our example, we could also add wings to the boat, that is, hydrofoils to lift it out of the water. These act just like wings on an aircraft, producing vertical lift. Typically, on a hydrofoil boat, they are positioned lower than any part of the hull. The lift raises the boat out of the water (whereupon there is no buoyancy force, of course, but no drag from the hull either) and only the hydrofoils

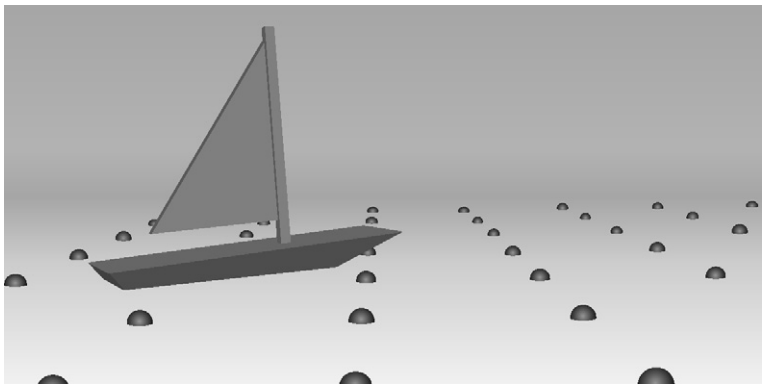

FIGURE 11.3 Screenshot of the **sailboat** demo.

remain submerged. The hyrdofoils can be easily implemented as modified surface force generators. The modification needs to make sure that the boat doesn't start flying: it generates no lift once the foil has left the water. In practice, a hydrofoil is often designed so that it produces less lift the higher the boat is out of the water, so the boat rapidly reaches its optimum cruising height. This behavior also wouldn't be difficult to implement by scaling back the tensor-generated force based on how near the wing is to the surface of the water. These modifications form part of one of the exercises for this chapter.

The Sailing Example

The **sailboat** demo in the accompanying source code puts all these bits together. You can control a catamaran on a calm ocean. The orientation of the sail and rudder are the only adjustments you can make. The prevailing wind direction and strength are indicated, as you can see from the screenshot in Figure 11.3.

The boat is set up with four buoyancy force generators, a sail, and a rudder. The wind direction changes slowly but randomly over time. It is updated in each frame with a simple recency weighted random function.

The update of the boat is exactly the same as for the aircraft demo, and user input is also handled as before. See the accompanying source code for a complete listing.

11.3 Summary

In this chapter, we've met a set of real-game examples where our physics engine combines with real-world physics knowledge to produce a believable simulation. In the case of both sailing and flight, we use a simplification of fluid dynamics to quickly and simply generate believable behavior.

The aerodynamic tensor isn't sufficient for games that intend to simulate flight or sailing accurately. We'd need to do a lot more work for that. But as they stand they are perfectly sufficient for games that are not intended to be realistic.

The situations I chose for this chapter were selected carefully, however, not to embarrass the physics engine. As it stands, our engine is less capable than the mass aggregate engine we built in Part II of the book. To make it truly useful, we need to add collisions back in. Unfortunately, with rotations in place this becomes a significantly more complex process than we saw in Chapter 7. It is worth taking the time to get it right. In that spirit, before we consider the physics of collisions again, we'll build the code to detect and report collisions in our game. Part IV of the book does that.

11.4 Projects

Mini-Project 11.1

(**a**) Add a new force generator to the **flightsim** demo to represent the propulsion force from the aircraft's engines (this is currently hardcoded in the `update` method).

(**b**) Allow the player to change the power going to the engines.

Mini-Project 11.2

Create a force generator, as in the previous project, to represent the propulsion of the aircraft. Allow the player to control the power of the propulsion and allow it to swivel between a forward force and a downward force. This will allow your aircraft to perform vertical take-off maneuvers.

Mini-Project 11.3

Change the **sailing** demo to represent a speedboat rather than a sailing boat. Make sure that the boat can't get any propulsion power when it has bounced out of the water.

Project 11.1

Create an airplane racing game. Using a simple aerodynamic model, allow the player to control an aircraft, including the power sent to its engines. The game level should consist of a series of hoops that need to be navigated in order in the minimum amount of time. You need to write code to detect whether a plane has passed through a hoop, but you shouldn't need to detect collisions with the hoop itself. The game should display the best lap time and the current lap time. You can extend the game with different airplanes having different performance and aerodynamic features.

Project 11.2

Create a rigid-body remake of the classic game Thrust. The player controls a ship that is under the influence of (very weak) gravity, but can be rotated and propelled.

If the ship hits a wall, it is destroyed (this neatly sidesteps our lack of contact physics). Use torque generators controlled by the player for rotation, and a force generator for propulsion.[1] Start with a simple shape for the level, such as a plane. This allows you to write the collision detect routine simply. This project can be extended with the ideas in the next part of the book to implement more complex level geometry.

1. The original Thrust game, by Jeremy Smith, did not use physics to control the rotation of the ship. The ship rotated at a constant velocity when the appropriate key was pressed.