Exploring the physics with computer animation

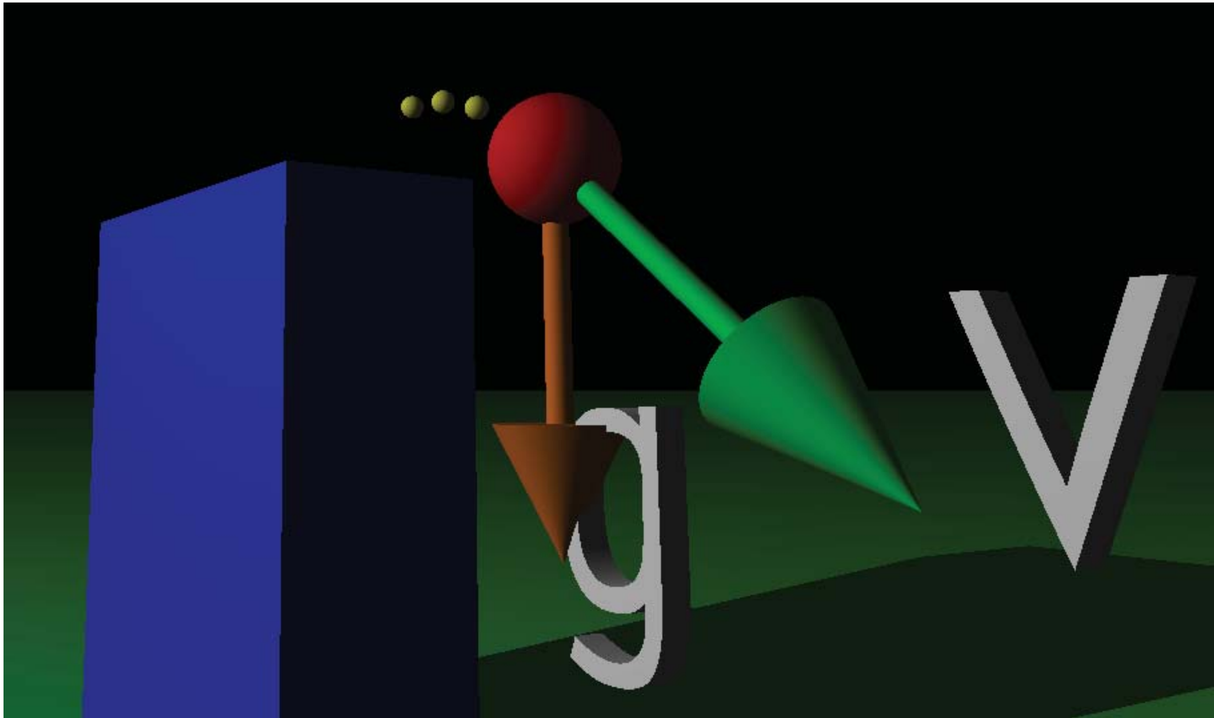

Figure 1: A virtual ball being launched off of a table. The gravity and velocity vectors are shown.

Dr. T.J. Bensky
Department of Physics
Cal Poly, San Luis Obispo
San Luis Obispo, CA 93407
tbensky@calpoly.edu
Winter 2012 edition
Updated: August 23, 2012

Contents

1 Using Computer Animation to Learn Physics **6**
1.1 What is computer animation? 6
1.2 Why computer animation with physics? 6
1.3 Why computer animation in a course on physics? 6
1.3.1 To learn physics 6
1.3.2 To test theories physics puts forth 7
1.3.3 See motion 8
1.3.4 Vector-centric visualization 9
1.3.5 Calculators are on the way out 11
1.3.6 Pencil and paper physics: Why? 11
1.3.7 Creativity 13
1.3.8 Projects 14
1.4 Like video games? 14

2 Installing Software to Study Physics **15**
2.1 Povray and Raytraced Computer Graphics 15
2.1.1 Povray 15
2.1.2 Seeing a scene 16
2.2 For a Windows 7, XP or Vista PC: Povray 16
2.2.1 Installation 17
2.2.2 Make a simple still scene 17
2.2.3 Make an animation 18
2.2.4 Make a playable movie 19
2.3 For Macintosh/OS X 22
2.3.1 Installation: MegaPov 22
2.3.2 Make a simple still scene 22
2.3.3 Make an animation 24
2.3.4 Make a playable movie 26

3 Preliminaries: Things you should know **27**
3.1 Mathematics 27
3.2 The idea of a function 27
3.3 Vectors: More than “magnitude and direction” 28

3.3.1 What's a vector? 28
3.3.2 Vector "lingo" 29
3.3.3 The Common Manipulations: the "magnitude and direction" 29
3.3.4 The magnitude of a vector 30

4 Drawing with Povray **31**
4.1 Introduction 31
4.2 Coordinate System 31
4.3 Drawing Examples 31
4.3.1 Spheres 32
4.3.2 Boxes 33
4.3.3 Cylinders 33
4.3.4 Planes 34
4.4 Colors 34
4.4.1 Stock Colors 34
4.4.2 Make your own color 35

5 Getting Started with Simple Programming **36**
5.1 Introduction 36
5.2 Skeleton Code 36
5.2.1 Making a movie 38
5.3 Basic Povray Statements 38
5.4 Structure of the Skeleton Code 40
5.5 The if statement 43
5.5.1 An if statement in Povray 43
5.5.2 Conditions 44
5.5.3 Testing *where* an object is 44
5.5.4 Testing *the direction* in which an object is moving 46
5.5.5 Careful with = ... 46

6 How do I... **47**
6.1 Use a vector component of a vector variable? 47
6.2 Find the angle a vector is making with respect to the $+x$-axis? 47
6.3 Draw a vector on an object? 47
6.4 Draw just the x or y component of a vector on an object? 47
6.5 See if two objects have collided? 48
6.6 See if an object has hit the ground? 48
6.7 Add trails behind an object? 48
6.8 Remove trails behind an object? 48
6.9 Draw an object as a sphere given that I know its position? 49
6.10 Draw an object as a box given that I know its position? 49
6.11 Draw something other than a box or sphere for my object? 49
6.12 Draw the ground or a big wall in a scene? 49
6.13 Find the magnitude of a vector? 50

6.14 Get a spring to look right as it pushes against a moving object? 50
6.15 How do I draw a spring? 51
6.16 Have something other than the black sky in my images? 51
6.17 Paint an object with an image file? 51

7 What to do if your movie won't render 53
7.1 My movie won't render 53
7.2 My movie doesn't work right 54

8 Drawing Physics 55
8.1 Introduction 55
8.2 Ideas on notation and usage 55
8.3 Useful functions in physics.inc 56

9 One dimensional motion 64
9.1 Introduction and Goals 64
9.2 The Physics 64
9.3 Projects 65
9.4 Wrap-up Questions 67

10 Two dimensional motion 69
10.1 Introduction and Goals 69
10.2 The Physics 69
10.3 Projects 70
10.4 Wrap-up Questions 73

11 Forces and Newton's Laws (Part I) 75
11.1 Introduction and Goals 75
11.2 The Physics 75
11.3 Projects 77
11.4 Wrap-up Questions 80

12 Forces and Newton's Laws (Part II) 82
12.1 Introduction and Goals 82
12.2 The Physics 82
12.3 Projects 83
12.4 Wrap-up Questions 89

13 Energy: Work, Kinetic, Potential, and Conservation 90
13.1 Introduction and Goals 90
13.2 The Physics 90
13.3 Projects 91
13.4 Wrap-up Questions 94

14 Momentum and Conservation of Momentum **96**
14.1 Introduction and Goals 96
14.2 The Physics 96
14.3 Projects 98
14.4 Wrap-up Questions 102

15 Rotational Motion **103**
15.1 Introduction and Goals 103
15.2 The Physics 103
15.3 Projects 104
15.4 Wrap-up Questions 106

16 Torque, Angular Acceleration and Momentum **107**
16.1 Introduction and Goals 107
16.2 The Physics 107
16.3 Projects 108

17 Final Project **113**

18 Future Plans **116**

Chapter 1

Using Computer Animation to Learn Physics

1.1 What is computer animation?

Computer animation is the process of drawing objects on a computer screen, that then appear to move around the screen. It can be anything from a simple bouncing ball, to something as complicated as characters in the movie "Toy Story." Either way, the objects are not real and they are not actually moving. The objects and their motion are "virtual" (they're just pixels) and computer programming techniques have been developed to make them *appear* to move. Rapid advances in computer technology have allowed for the production of computer animations with very lifelike realism, where one can easily forget the virtual nature of it all.

1.2 Why computer animation with physics?

Something has to drive the virtual motion of a computer animation. Even in the case of a simple red ball, how is the computer to know where the ball is supposed to be during each frame of the animation? Typically mathematical x, y, z coordinates can provide the position, but they must be calculated first, as the computer must know a *precise* position before it can start filling pixels on the screen. If the animation is supposed to mimic the way things move in real-life, then the laws of physics can be used compute these positions, and this is the theme of this book: *using laws of physics to produce computer animations.* It's a very natural fit for the branch of physics called "mechanics," as we'll see.

1.3 Why computer animation in a course on physics?

1.3.1 To learn physics

As far as *learning physics* goes, it turns out that students who learn physics concepts via static pictures (i.e. from the textbook alone), can be led to construct incomplete or incorrect men-

tal models that hamper their understanding of physical concepts (see "Open Source Physics," http://goo.gl/EuGml). Also, it is pretty well known that the typical physics student will solve problems by first trying to find an equation that seems to "fit" the problem. This is quicker and easier (and may lead to an answer) than trying to understand some underlying concept of physics. It is a extremely shallow way of "doing physics." We often see students "reading the textbook in reverse," meaning they start with the homework problem at the end of a chapter, then flip *backward* through the book until an example or equation seems to fit.

Computer animation in this setting does two things remarkably well. First, it brings physics concepts to "life" by, for example, showing how a force can change an object's velocity vector, how friction sucks energy out of a system, or how the spring force grows with compression length, all in realtime. The emphasis in this text will be to observe the wildly dynamic nature of velocity, force, and acceleration vectors, and how they interact with one another as an object moves under their influences. This alone is a much more compelling way of learning physics that viewing static pictures. Second, a computer animation will refuse to work (properly) if the physics concepts are not applied, or not applied correctly. It is not possible to produce a correct computer animation by, for example, reading the textbook in reverse.

1.3.2 To test theories physics puts forth

The first term physics class is typically about "mechanics" which has to do with the "nuts and bolts" of basic motion. So you'll be studying the physics of motion. The concepts of force, velocity, acceleration, momentum, and energy dominate the theory of motion. Computers are really good at crunching numbers and making animations, but they need instructions on how to do so. In this case, the instructions will be the equations that come from the physics theory on how motion works.

So using a computer to learn physics will be about you inputting physics equations into a computer, as instructions on how it should make an on-screen object move. If the theory is correct, then motion will illustrate the "motion theory" that you are studying. What of this theory? Where does it come from? It'll come from a variety of concepts, but will always driven by two primary equations, which are

$$x = x_0 + v_{0x}\Delta t + \frac{1}{2}a_x\Delta t^2 \tag{1.1}$$

and

$$v_x = v_{0x} + a_x\Delta t. \tag{1.2}$$

The "x's" iin these equations are used to denote position (x) and speed (v_x) along the x (or horizontal) axis. There are two other equations that look like these two for the y-coordinate, which are $y = y_0 + v_{0y}\Delta t + \frac{1}{2}a_y\Delta t^2$ and $v_y = v_{0y} + a_y\Delta t$. These are used to denote motion along the y (or vertical) axis.

These equations enable you to *predict* the motion of an object, given that you know its acceleration (a_x and a_y), current velocity (v_{0x} and v_{0y}), and Δt, which is how far in the future you'd like to look. "Predict" is a funny word; it implies knowing what's going to happen in the

future. This is not a safe business to be in, as no one can really predict the future, but it turns out that physics can do a wonderful job at predicting the motion of everyday objects, if you work carefully. Most of the hard work this in making computer animations is in figuring out what a (or acceleration) to put into these equations. It'll come from a variety of sources this quarter, which will drive our studies.

What do these have to do with motion? Well, take the x_0 and y_0, buried in the equations. They are an (x, y) coordinate that may be plotted on a coordinate system. If you plotted a point at (x_0, y_0), it would represent where some object is *right now*. The left hand side of the x and y equations give you a coordinate point (x, y), which may also be plotted on a coordinate system; *this is the position of the object some time* Δt *in the future relative to when the object was at* (x_0, y_0). See? The equations allow you to predict where an object will be in the future (x, y) relative to where it was in the past (x_0, y_0).

For computer graphics, this ability to predict, or *continually predict* positions is the key element in producing animations. If you start a ball at the left edge of the screen, and predict where it'll be say a Δt later, then you can plot the ball at this new position. Next, this new position becomes the "current" position, so we make still another prediction based on this new position. Then again, and again. Pretty soon, the ball has moved across the screen, according the the (predictive) laws of physics.

What about the v_{0x} and v_{0y}? They are the two components of the the object's velocity, v. The first, v_{0x} is how fast the object is moving horizontally along the x-axis. The other, v_{0y} is how fast the object is moving along the vertical, or y-axis.

What about the accelerations, a_x and a_y? They are more difficult to prescribe simply, as they can come from a variety of sources. To experiment, however, one may simply put in a numbers for these and see what happens.

1.3.3 See motion

So first term physics is about motion. You'll see a lot of equations and a lot of algebraic manipulations. You'll plug in a lot of numbers here and there. You'll read a lot of problems from the end of your book's chapter. You'll put boxes around a lot of answers, then flip to the back of the book and see if your answer is correct. But considering physics is a class about *motion*, you typically *don't actually see*every much *MOTION!* It seems odd then that all of the connection points you'd like to make about physics *must* be constrained to a paper and pencil mode.

As an example, find a picture of "projectile motion" in your physics text. Look at it for a bit. Are multiple objects drawn smeared across the same figure? Does this seem like motion to you? Is the only way to study motion to look at these "stroboscopic" images, while imagining (correctly or not) what the actual motion looks like? Or is there another way? What is a "stroboscopic" even mean?

The trouble with learning physics on paper is that you never get any exposure to the Δt portion of the equations, which is the passage of time. This simply cannot be represented realistically on a piece of paper or a chalkboard. Both of these media are spatial. They are so many inches wide and high, for instance. They do not include any element of time. Computers

however, with their screens, can do a wonderful job of illustrating the passage of time, by producing frame-by-frame movies that actually evolve a real clock ticks on the wall. That is, sequential frames on a computer screen can illustrate the passage of real time. This why using computer animation to learn physics is so powerful. It elicits the very important aspect of time, which a piece of paper simply cannot do.

1.3.4 Vector-centric visualization

Answer this question: "What is a vector?" Your answer is likely "a quantity with a magnitude and direction," and this is strictly true. Vectors, however, take on an entirely different "life" in the context of computer animation. In fact, they can be wonderfully dynamic quantities that stretch, shrink, and meander into predetermined directions as an animation unfolds. Go back to the stroboscopic view of projectile motion discussed in the above section. Can you draw the velocity vector on each ball? How about a block sliding along some frictionless ice that suddenly encounters a rough patch? What about a ball thrown vertically upward? Are you sure you got the sizes and directions right? Ok, now describe how the velocity vector would behave in these scenarios, as the motion unfolds. Most likely you'll start using your fingers, perhaps using your fist as the object.

In mechanics, the time evolution of vectors really tell the story of motion that you simply must see to appreciate. No drawing or or "hand acting' will ever suffice. Throughout the projects here, drawing vectors on all of the moving objects is going to be a big deal, watching them in completed movies will be an even bigger deal. You'll never be so delighted as to see the normal, velocity, and acceleration vectors on an object on a flat surface that is about to encounter and climb a hill that leads to another flat surface at a higher level.

A special vector: the v-vector

In addition to the idea of watching vectors evolve, computer animation highlights one vector in particular: the *velocity vector*. Why would this be? Take a look at the six balls in Figure 1.1.

Each object in the figure has a different vector sticking out it; the vectors are common throughout elementary physics: v for velocity, a for acceleration, F for a force, T for tension τ for torque, x for position. The question is then, from which object/vector combination can you say something about where the object will be a small time in the future? In other words, which figure allows you to *predict* the subsequent motion of the object?

As you'll see in your animation studies, only the v-vector allows you to make this prediction (the object will generally be up and to the right a small Δt in the future). In other words, the v-vector is the indicator of future motion, at least for a small time step into the future. None of the other vectors offer this information. (Later we'll see that the momentum vector, or $\vec{p}$ also allows us to predict motion, but p is just the product of mass and the v-vector.)

As you create computer animations based on physics, you'll always be asked to render the v-vector on the moving objects. Try to pay close attention to this vector in your work and see if it's indeed a predictor of an object's motion. We claim that

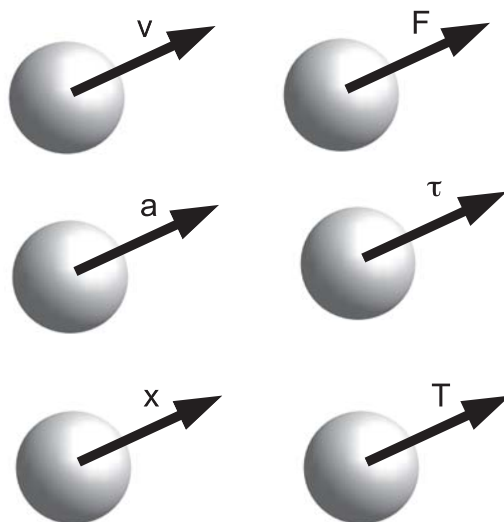

Figure 1.1: Six objects with different vectors sticking out of each.

> *If you know the v-vector, then you can make a very intelligent guess as to where the object will likely be a small Δt in the future.*

In other words, you can make a *prediction*, and the smaller Δt, the better your prediction. We stress that it isn't a good idea to get too greedy with predictions. Keep your demands for "how far in the future" small, and your predictions will be just fine.

To conclude then, take a look at Figure 1.2 that shows, an object with a velocity vector pointing from it. There is a twofold theme for this entire class that comes from this figure.

1. The v-vector tells us the direction the object is currently moving and about where it will be a small interval of time in the future.

2. This entire class is about different laws of physics that allow us to *make the v-vector of an object change, either in direction or length, or both.*

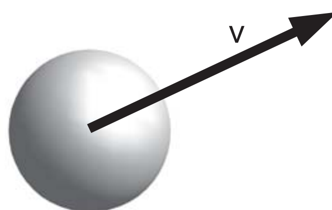

Figure 1.2: An object with its velocity vector.

Lastly, why is understanding how a v-vector is manipulated so important?

- Because basic physics is about understanding how objects move. A key element in this understanding is in being able to *predict* where an object will be at a given time in the future, and to be correct in your prediction. All of this is contained in an object's v-vector.

- Changing the *magnitude (or length)* of the v-vector changes the speed of an object. Making the length grow means the object is moving faster. Shrinking the length means the object is moving slower. Changing the *direction* of a v-vector changes the direction in which the object will move over a small time interval into the future. The v-vector dictates the impending motion of an object. Manipulating it, or understanding it is the key to controlling or understanding an object's motion, whether it be a car, spacecraft, or bicycle.

- To really understand each lesson this quarter ask yourself: "Do I understand how the physics in this lesson can change the v-vector of an object?"

1.3.5 Calculators are on the way out

You are probably somewhat trained in science via the use of a calculator like in Figure 1.3. You can use it to crunch through numbers to obtain a result. Well, here's the bad news: modern scientists rarely use calculators anymore. The screens are too small and hard to read. The keyboard are awkward. The overall form-factor is terrible, and they're way overpriced. Results are hard to check, publish, put on the web, or share with anyone. Supposing you have a result, it is also hard to make a small change and see what new result might pop out. Some will move on to a spreadsheet, which is OK, but most scientists will find some specialty computational software they like to use, like Mathematica, Maple, Matlab, Octave, Scilab, etc. Indeed, science today is done on *computers* not *calculators*, and this spans the gamut of the sciences, from biology to chemistry to physics and engineering. Figure 1.4 shows a physics problem being done in Mathematica, where results can be easily printed, shared, or changed to produce new results.

So why not start your own training on using computers in science as a way into using computers as scientific investigative tools. Why not start now?

1.3.6 Pencil and paper physics: Why?

The standard physics course experience in dominated by having students solve problems with pencil and paper. Exams are like this, as are regular homework assignments. You get frustrated when you can't figure out how to solve to these problems (meaning write out the step-by-step solutions). The typical student rarely pursues such solutions to completion, often scraping tips or full solutions off of the Internet and just "following along" (if even that). Professors get upset when they "teach and teach" and their students can't solve problems in this mode. So here we are, getting all frustrated and upset. As a student, you shouldn't feel bad. It turns out, that very few people *in the history of the human existence* are successful in this mode of study. As professors, we should feel bad, because we need to change our instructional tools.

An entire field of research called "physics education," has existed for 20 or so years now, and is fairly consumed at studying and enhancing the teaching and learning of physics using "pencil and paper." This field has given us things like "free body diagrams," color in textbooks, "content

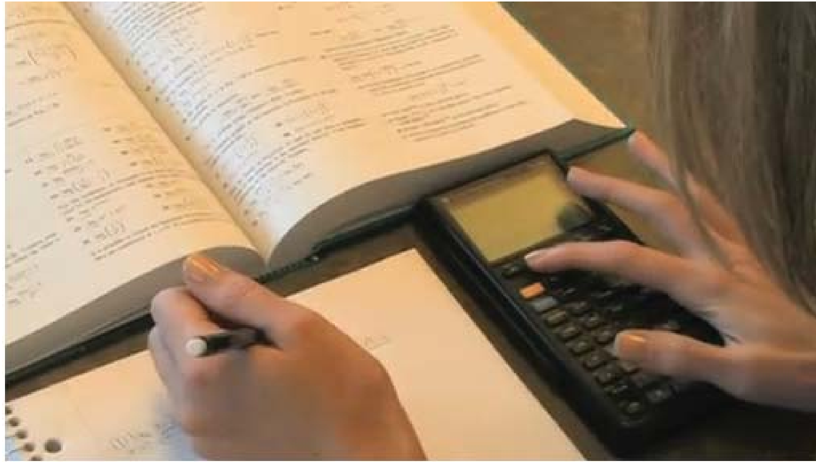

Figure 1.3: A student using a calculator to work on problems from a textbook. Scientists don't really use calculators anymore.

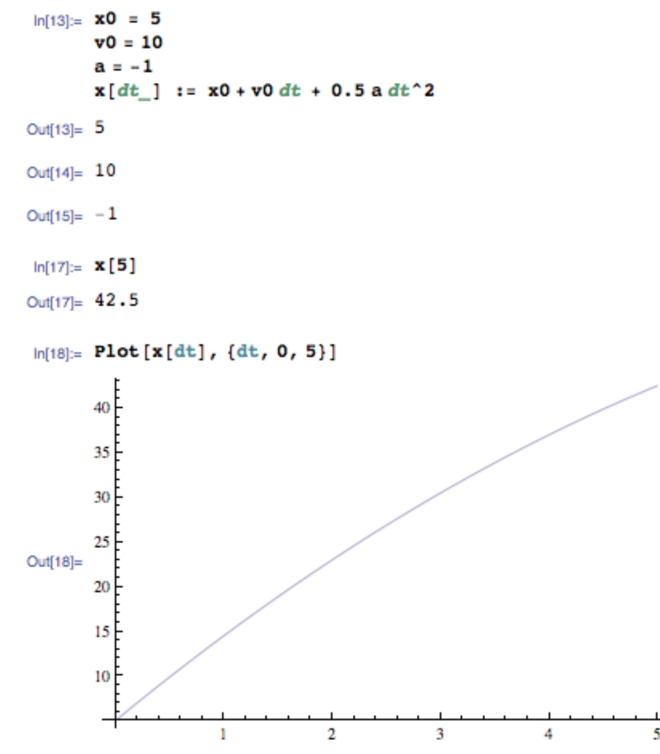

Figure 1.4: A physics problem being solved using Mathematica.

rich problems," "motion diagrams," the "meter per second," and thousands of problems that can result in an answer around which we can put a box and say "done." All of these (and many more) are carefully thought out ideas that help one learn physics using a pencil and paper. Go to the library and find a physics book from the 1960s. It is very cold, unfriendly, and hard to even imagine using.

This "pencil and paper" mode is formally known as "theoretical physics." It is a mode of study where one has exceptional linkage between thought and expression. It is the mode in which Albert Einstein worked. It is a mode of working on problems with little more than paper, a pencil, and your mind. Wikipeda defines it as "...a branch of physics which employs mathematical models and abstractions of physics to rationalize, explain and predict natural phenomena." The field of theoretical physics is known to be one of the most difficult of all human endeavors to penetrate (see Joao Magueijo, "Faster than the speed of light"). In fact, since the 1500s only 64 people are known to be theoretical physicists that have made a major contribution to the field of physics (Wikipedia). Now this fact isn't quite fair, because there are likely people around you (like your professor) who can do some pretty amazing analysis (relative to your own skill set) with just a pencil and paper, but the bigger picture is still that *pencil and paper physics is hard.*

The point here is that this mode of problem solving is difficult. Period. Most of us just aren't Einstein, but it's not that we can't be; it would just likely take a lifetime of "practice" before you'd be proficient enough to get anything done. And, since the 1500s, only 64 people have really pursued this to the points where their work had long lasting effects. You're just trying to learn something about basic physics. So why so much emphasis on this mode at such an early stage of your education? We don't really know, but it has a lot to do with efficiency, cost, practicality, and "teaching inertia" (i.e. it has always been done this way). How else can one professor "teach" a class of 50-ish students? How else can we possibly agree that you "know physics" unless you can solve a problem that starts with "A car moving at 10 m/s..." all by yourself?

We think this is perhaps where the computer can help. The computer is a much more interactive tool than a piece of paper. It is more visual and a more compelling medium for most of us for "playing around" with physics problems.

1.3.7 Creativity

Everyone, no matter what their level of science training, is creative. Tapping into your own creativity is a lifelong skill worth developing. Solving end-of-chapter homework problems do not require creativity. The problems themselves have value in the logic they might require to solve, but that is all. They are canned problems, with known solutions, having only one or two possible solutions paths. The pdf solutions book on the web is tempting to consult. The problems offer little to discover, explore or disseminate.

Because the computer is such a creative tool, laws of physics can be studied with styles, colors, and perspectives that you find appealing. You can tap into your own creative motivations as you create a representation of the laws of physics. You can certainly "solve for" how long it takes a ball to fall from a building, or you can render a building and ball and *watch* the ball fall

on your screen. Why not then change gravity and pretend the scene is on Jupiter? Or maybe add a crosswind? Or a pedestrian walking on the sidewalk? Do they get hit? What about a drag force? All of this is possible and limited only by your own creativity.

1.3.8 Projects

Paper sheets of homework problems and spiral notebooks from your physics class become unimportant, lost, or even thrown out as a school term ends. Years later, there is no evidence you even took a physics course, other than your vague memory and a grade on your transcript. By creating computer animations, you are producing electronic content perhaps for the first time. You are breaking out of a more typical role of *consuming* digital content. Your work can be posted on Youtube or submitted as part of a social-sharing site where its lifetime will be many times longer than your first paper homework assignment. You can show your "physics work" to friends, family, or even future employers.

1.4 Like video games?

Physics is actually "out there" more than you think. If you've ever played a video game, like a "first person shooter," flying, driving, climbing, etc. game, then likely all of the motion is done using a "physics engine," which is a large software tool that uses the laws of physics to handle the motion of animated objects like bullets, tanks, aliens, cars, and robots. In fact, the video-card maker NVIDIA maintains such an engine called "physx," to get developers to use their hardware and tools. "Havok" is another. Underneath all of the glitter are basic physics equations predicting where the car, bullet, alien, or rocket should go next. These objects must be placed realistically on the screen, or the animation will not appear lifelike, severely hampering the quality of the final product. This goes for video games as well as computer-generated movies; *they are all driven by physics.*

Chapter 2

Installing Software to Study Physics

2.1 Povray and Raytraced Computer Graphics

We'll need some computer software to create animations. The software must allow us to create physically-realistic motion using equations and theories that will come up along the way. To do this, the software must provide us with two core abilities. First it needs to be able to process the equations we give it that will come from our study of physics. Second, it must be able to draw objects, based on our equations, so we can visualize them and how are theories cause them to move. (It would be nice if the graphics produced looked *compelling and professional*.)

2.1.1 Povray

Software called Povray meets our two requirements, and creates stunning-looking graphics, fairly easily. Povray is a free, open source graphics package that renders drawings on the screen using a technique called "ray tracing." Unlike drawing on the screen using a mouse and a palette of circles and lines (etc.), ray tracing is a drawing technique that "renders" images by simulating how light rays would interact with objects placed in a virtual scene. The ray-tracing technique often produces graphics that are quite stunning as you can see here http://en.wikipedia.org/wiki/File:Glasses_800_edit.png. Ray traced images are lifelike in quality with a high degree of realism, including shadows, shading, light intensity effects, and perspective. They can be hard to distinguish from actual photographs. In essence, Povray provides us with a virtual 3D world in which we can produce beautiful images with minimal effort.

There are versions of Povray for both Windows and OSX computers. The Windows version is called "Povray." The OSX version is called "MegaPOV." Both are "open source" software packages, freely available on the internet. Don't visit these links right now (following the installation direction below), but for reference, the home of Povray can be found at http://www.povray.org, while that of MegaPOV can be found at http://megapov.inetart.net/. This chapter will lead you through the process of installing the software on the platform of your choice, and preparing it to visualize physics and create animations.

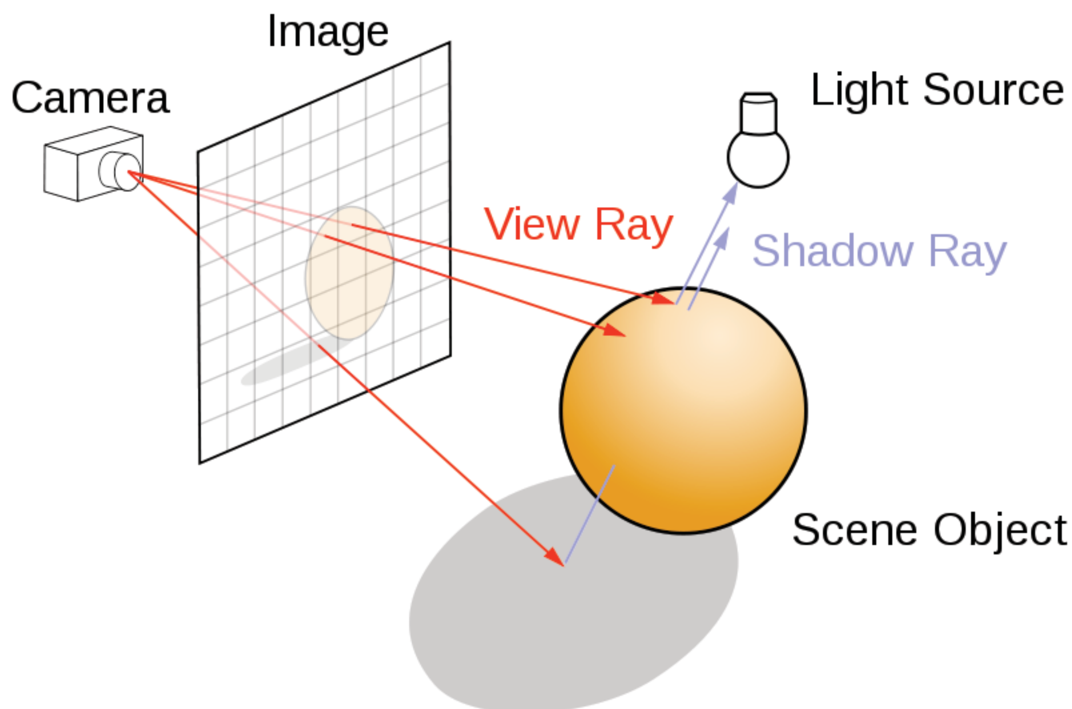

Figure 2.1: Three requirements for ray tracing: a camera, light source, and object (from http://en.wikipedia.org/wiki/File:Ray_trace_diagram.svg).

2.1.2 Seeing a scene

If you think about it, you are able to see a scene for three reasons. The first is that there is light, or light rays filling the area you wish to view. The second is that there are objects in the scene (boxes, balls, a floor, sky, etc.). The third is that there is a recording device, like your eye or a camera that takes a snapshot of the scene from some vantage point.

Ray tracing takes these three items and *calculates* what a scene should look like. You can read more about ray tracing on Wikipedia. The three requirements for ray tracing can be seen in Figure 2.1.

The useful feature in studying physics in specifying where *the objects* will reside in the scene. For us, these positions will come from the theory of motion we are trying to study. In other words, we will used physics theories and equations to instruct Povray where to draw objects. Using the realism of ray tracing, Povray will render our scene so we can then visualize the motion of our object.

2.2 For a Windows 7, XP or Vista PC: Povray

To install Povray on a Windows PC follow these two steps.

2.2.1 Installation

Step 1: Download and Install Povray

1. Install Povray on your computer by going to http://www.povray.org/download and scrolling down to the "Windows" tab. Click on the "Download 32-bit (12 MB) via the web." This will download a file called `povwin362-32bit.msi` to your computer.

2. **For Windows XP or Windows 7:** Double click on `povwin362-32bit.msi` this file to install Povray onto your computer. You can delete it when the installation has finished. Go to step #2.

3. **Windows Vista:**

   (a) Double click on `povwin362-32bit.msi`, but don't install POV-Ray in "Program Files" folder that the installer suggests. Install it somewhere else like "C:\POV-Ray3.6."

   (b) After installation, find the POV-Ray binary file in "C:\ POVRay3.6\ bin\ pvengine.exe" (or wherever you installed Povray).

   (c) Right click on the icon then select "Properties" .

   (d) Click the "Compatibility" tab

   (e) Click the button "Run this program in compatibility mode for:",

   (f) Select "Windows XP" (or "Windows 98/Windows Me" if that doesn't work) and select lower on this tab "Disable visual themes"

   (g) Click on the "Security" tab and make sure that "all users" have "full control" selecting the "permissions" box accordingly.

Step 2: Get the software "physics-ready"

1. Download a file called "physics.inc" from this folder `http://goo.gl/zTtuQ` .

2. When downloaded, this should produce a file on your computer called "physics.inc." Put it on your Desktop where you can find it.

3. Look in your "My Documents" folder. Click on "Povray," then "v3.6." You should now see a folder called "include." Copy `physics.inc` into the folder called "include."

2.2.2 Make a simple still scene

Run Povray by double clicking on the icon the installation program created. The first time you run Povray, a bunch of samples will auto-load. Do a "File→Close All" to get rid f these. Then follow these steps.

1. Pull down "File→New."

2. An text-edit window will come up called “Untitled.”

3. Immediately pull down “File→Save As...” and call it something like `sphere.pov`. It is very important that you have the extension of “.pov” at the end of your name.

4. To render a simple still scene in Povray:

5. Into the edit window, type in this Povray code:

   ```
   #include "physics.inc"

   camera { location <0,0,-20> look_at <0,0,0> }
   light_source { <0,0,-50> color White }

   sphere { <0,0,0>,1 pigment {Red} }

   draw_vector(<0,0,0>,<3,3,0>,Green,"hi")
   ```

   This will put a camera at (0,0,-20) and have it aim or “look” at the point (0,0,0). The scene is illuminated with a white light source at (0,0,-50). A red sphere will be the only object in the scene, centered at (0,0,0) with a radius of 1.

6. To render (or build) the scene, click the “Run” icon in the toolbar at the top of the screen.

7. You should see a new window pop up with the rendered sphere.

You should experiment with the sphere position, radius, color, the camera location and look_at point, etc. After making any changes to the code, click the “Run” icon again to re-render the scene. You can change the render size by pulling down “Render→Edit Settings/Render” and using the selection box to the right of the “Section:” label.

2.2.3 Make an animation

1. Start a new Povray window by pulling down “File→New”

2. Immediately do “File→Save As...” and choose a new file name like spheremovie.pov.

3. Type the following code into the spheremovie.pov window.

```
#include "physics.inc"

camera { location <0,0,-15> look_at <0,0,0> }
light_source { <-0,0,-20> color White }

#declare pos = <-5,0,0>;
#declare vel = <5,0,0>;

#declare dt = 0.1;
#declare xtime = 0.0;

#while(xtime <= clock)
        #declare a = <0,0,0>;
        #declare pos = pos + vel*dt + 0.5 * a * dt * dt;
        #declare vel = vel + a *dt;
        sphere { pos,0.1 pigment {Yellow} }
        #declare xtime = xtime + clock_delta;
#end

sphere { pos,1 pigment {Red} }
draw_vector(pos,vel,Green,"v")
```

4. Type `Final_Frame=10` into the white box that is directly below the "Queue, Rerun, and Show" icons, as shown in Figure 2.2.

   Tip: More frames provides for more detail and smoother animations. Feel free to change this as needed. Typing `Final_Frame=50` will make a 50-image run of your code, and will make Povray's internal "clock" variable run from 0 to 1 in steps of 0.02 (1/50). Typing `Final_Frame=10` will make a 10-image run of your code. It will be coarser with animation that is a bit jumpier.

5. Click "Render."

6. You will see your movie being calculated, slowly, scene by scene. Each scene is saved as a .bmp image file in the same folder as the source file, spheremovie.pov. The files are sequentially numbered, like spheremovie01.bmp, spheremovie02.bmp, spheremovie03.bmp, etc.

2.2.4 Make a playable movie

The individual files Povray produced now need to be stitched into a single movie file for viewing and/or uploading to Youtube. To stitch your movie together into a watchable movie, follow these steps.

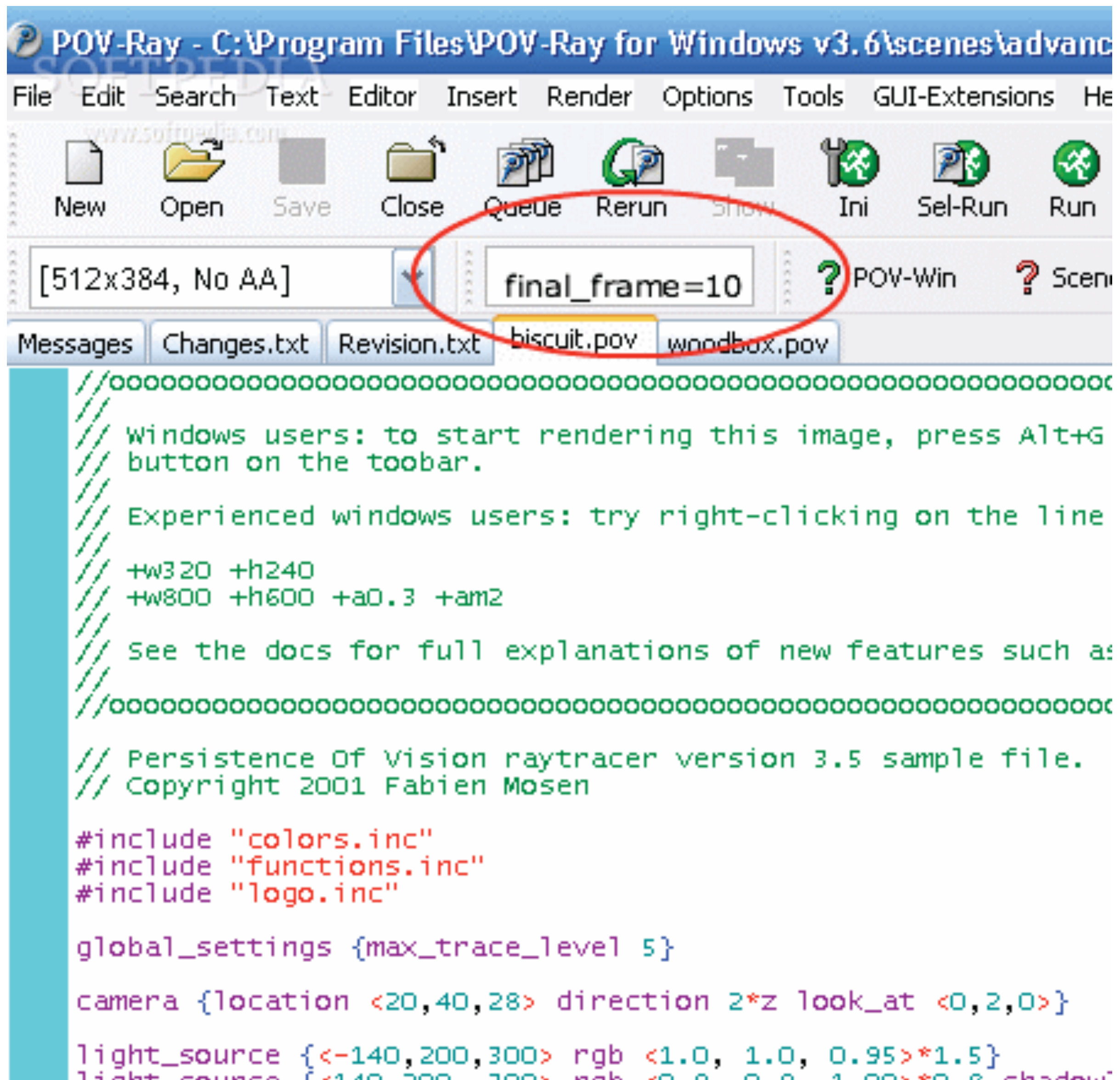

Figure 2.2: How to tell Povray to render your frame multiple times (in this case 10), causing time to evolve from render to render.

1. Download another software packaged `stitch_windows.zip` from this folder `http://goo.gl/zTtuQ` .

2. When the download is complete, double click on the icon of the folder that is created. This icon should have the image of a zipper running down the center of it.

!!STOP!! Stitch is not ready to use yet! Keep following these steps.

3. When you double click on the icon of the folder, a list of files will be displayed, which are the files in the stitch package.

4. Click on the "Extract All" button in the upper left corner of the window. Select a suitable spot to extract the stitch files. This will install the stitch program on your computer, in a folder called "stitch_win."

5. Inside of this folder is a program called "stitch." this is the one to double click on and run when needed. Always keep all of the stitch files together in a single folder. If you separate them, the program will not work.

6. The stitch software is now installed and ready for use. Delete the folder that has the zipper running through it.

To make the movie from your images, follow these steps.

1. Run the stitch program in the `stitch_win` folder.

2. Click on the top "Browse" button and select the very first file, in the sequence of images that Povray produced. In this case it should be just `spheremovie01`.

3. Click on the lower "Browse" button and tell "stitch" where to put your final movie file.

4. Choosing 10 frames per second is fine. If you want your movie to run faster, choose a higher number.

5. Click the "Stitch" button. It might take a few seconds to complete the stitching job.

6. To watch your movie, click the "Watch movie" button that will become available when the stitching is done.

7. If everything looks OK, then the same movie file created is what you can upload to YouTube for sharing or submitting.

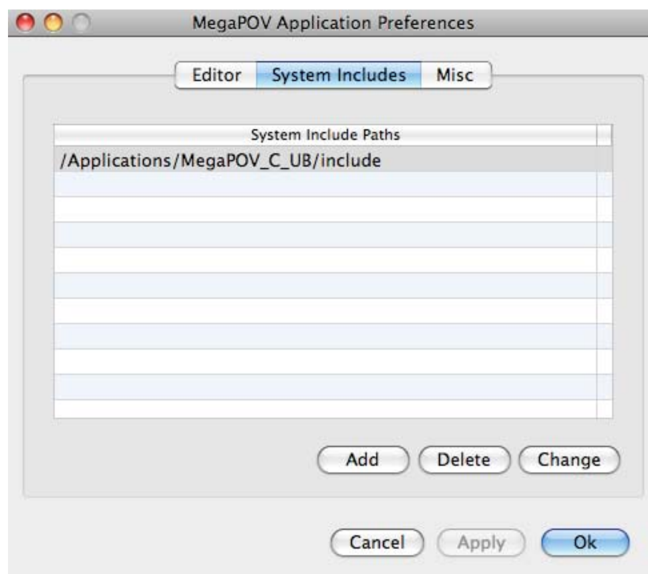

Figure 2.3: A configuration step for MegaPov.

2.3 For Macintosh/OS X

2.3.1 Installation: MegaPov

For OSX, we'll use ray tracing software called MegaPov. If you look in the folder at this link `http://goo.gl/zTtuQ` , download the file called `Megapov.dmg`. To install it, simply drag the Megapov folder out of the DMG file and onto your desktop (or Applications folder, or wherever you like to install software on your computer).

Next, there are two configuration steps you must do as you run Megapov for the first time. Run MegaPov by clicking on its icon in the folder you pulled from the DMG file above. Then do the following two steps.

1. Pull down the MegaPov→Preferences... menu. Click System includes, then the Add... button. A file selection box will pop up that look like Figure 2.3.

   Navigate to the same MegaPov folder that you dragged to your desktop above. Inside of this folder is another sub-folder called "include." Select it then click "Ok." Your window might look that shown in Figure 2.3.

2. Next, be sure to set the "Image type" to "png" as shown in Figure 2.4. It defaults to "Don't save image."

2.3.2 Make a simple still scene

1. Run Megapov and pull down the "File→New" window.

Figure 2.4: A configuration step for MegaPov.

2. A window will come up that you can type into called "untitled." Immediately pull down "File→Save As..." to call your work something like sphere.pov. It is very important that you have the extension of ".pov" at the end of your name.

3. Into the edit window, input this MegaPov code:

```
#include "physics.inc"

camera { location <0,0,-20> look_at <0,0,0> }
light_source { <0,0,-50> color White }

sphere { <0,0,0>,1 pigment {Red} }

draw_vector(<0,0,0>,<3,3,0>,Green,"hi")
```

4. This will put a camera at (0,0,-20) and have it aim or "look" at the point (0,0,0). The scene is illuminated with a white light source at (0,0,-50). A red sphere will be the only object in the scene, centered at (0,0,0) with a radius of 1.

5. To render (or build) the scene, press Command-R or pull down "Render→Render."

6. You should see a new window pop up with the rendered sphere.

You can experiment with the sphere position, radius, color, the camera location and look_at point, etc. After making any changes to the code, press Command-R to re-render the scene (you might have to save your changes first). You can change the render size or output type by using the MegaPov "Preferences" window that should be visible on your screen somewhere. To do so, click on the output tab and select an "Image Size" and/or "Save Image As" setting as needed. The "image size" is the size (width and height, in pixels) that you want your image to have. The "output type" is the type of graphics image you want your output image to be formatted as. These are the usual graphics formats like jpg, png, bmp, etc. You should also experiment with the sphere position and radius, the camera location and look_at point.

2.3.3 Make an animation

1. Start a new edit window by pulling down "File→New"
2. Immediately do "File→Save As..." and choose a new file name like spheremovie.pov. You are advised to save your work in a new and separate folder. In this section, MegaPov is potentially going to generate many files and you should keep them organized in a folder (instead of scattering them all over your desktop).
3. Type the following code into the spheremovie.pov window.

```
#include "physics.inc"

camera { location <0,0,-15> look_at <0,0,0> }
light_source { <-0,0,-20> color White }

#declare pos = <-5,0,0>;
#declare vel = <5,0,0>;

#declare dt = 0.1;
#declare xtime = 0.0;

#while(xtime <= clock)
        #declare a = <0,0,0>;
        #declare pos = pos + vel*dt + 0.5 * a * dt * dt;
        #declare vel = vel + a *dt;
        sphere { pos,0.1 pigment {Yellow} }
        #declare xtime = xtime + clock_delta;
#end

sphere { pos,1 pigment {Red} }
draw_vector(pos,vel,Green,"v")
```

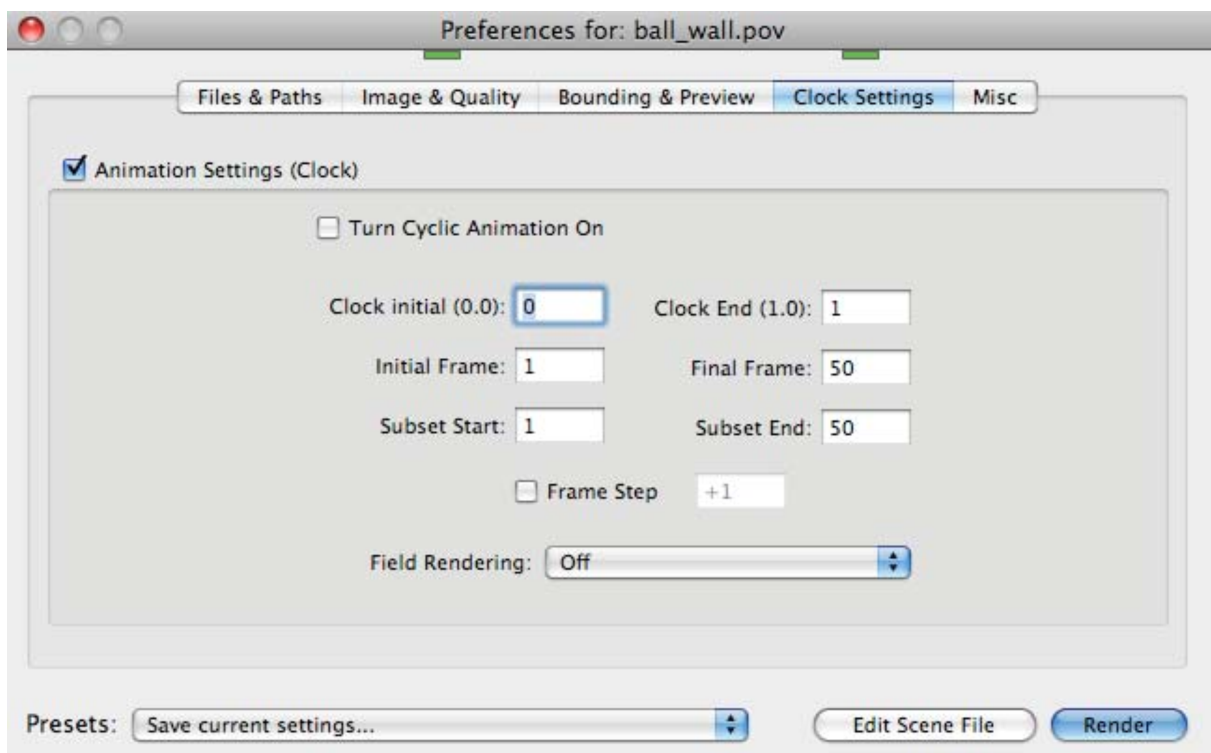

Figure 2.5: Tell MegaPOV to run through your code several times in sequence. In this case your code will be executed 50 times. The variable `clock_delta` is internal to Povray will be set to $1/50 = 0.02$, which controls the main `#while` loop to advance through 50 frames. Note the checkbox ("Animation Settings (Clock)") and the "Final Frame" numbers.

4. Click on the "Clock Settings" tab in the "Preferences" window. On this tab, check the "Animations Settings (Clock)" box see Figure 2.5).

5. Then set the "Final Frame" number equal to 10 (or so), "Clock Initial" to 0, and "Clock End" to 1.0. This will render a series of 10 images with the "clock" variable changing in your code from 0 to 1 in steps of 0.1 (1/10).

   Tip: More frames provides for more detail and smoother animations. Feel free to change the "Final Frame" to a larger number for a final render. I would think your final renders should contain at least 50 frames. This will make the "clock" variable run from 0 to 1 in steps of 0.02 (1/50).

   Note: Be sure the Subset numbers are the same as the Frame numbers. So Subset Start should always be the same as Initial Frame and Subset end should always be the same as Final Frame.

6. Render your movie by pressing Command-R. MegaPov will run your code N times, where N is the "Final Frame" number. Each run will generate a different image file, like spheremovie01.jpg, spheremovie02.jpg, etc. Each image is what your scene looks like at different

points in time.

2.3.4 Make a playable movie

The individual files MegaPov produced now need to be stitched into a single movie file for viewing and/or uploading to Youtube. Go to this folder `http://goo.gl/zTtuQ` and download the file called `stitch_for_mac.dmg`. Those with newer (Intel, Lion, etc.) Macintosh computers can download the `stich_osx_intel_new_macs.dmg` version.

After the download is complete, double click on the white stitch .dmg icon. Inside of it you should see a folder called “stitch_osx.”

> !!STOP!! Stitch is not ready to use yet! Keep following these steps.

Drag the “stitch_osx” folder from the white DMG icon, to some other place on your computer, like into the “Applications” folder or on your desktop. You can now remove the white .dmg file icon. This completes installation of this program. The stitch program as it exists in the “stitch_osx” folder is now ready for use. There are other files in the stitch package as well. Be sure to keep all of these files together.

To stitch your movie together into a watchable movie, run the stitch program and follow these steps.

1. Double click on the sewing machine in the “stitch_osx” folder.

2. Click the top “Browse” button to navigate to the folder where you saved you code, spheremovie.pov. Click on the very first image, called spheremove01.png (or .jpg or .bmp, etc.).

3. Click on the bottom “Browse” button to navigate to where you want your movie file to be stored. It is recommended that you have the movie stored in the same folder as your .pov file and your image sequence.

4. Choosing 10 frames per second is fine. If you want your movie to run faster, choose a higher number.

5. Click “Stitch.” All of your images will be stitched together into a self-contained mp4 movie (a recognized computer movie format).

6. You can watch the movie by clicking on the “Watch movie” button that will become available when the stitching job is done.

7. The saved .mp4 movie file is the last step in the animation job. This file can be played, emailed, and/or uploaded to Youtube.

Chapter 3

Preliminaries: Things you should know

Here are some preliminaries you should be comfortable with before studying physics or expecting to create computer animations. These are things that should be automatic to you. They don't really have anything to do with physics, and aren't necessarily something you'll learn in physics, but should already know from your preparation to begin learning a technical field.

3.1 Mathematics

Variables. Variables are letters that stand for numerical values. Something like x^2 means that is x is known, we should multiply it by itself. On paper, we could write $x = 4$, then know that x^2 will evaluate to 16. *Computers, calculators, and spreadsheets* behave in the same manner; textual variables can hold values for later use. So on a computer we could type `x=5` assigning the value of 5 to variable `x`. Variable names on computer are often longer, to make them more descriptive. So instead of `x` we might see `sphere_x`, meaning the x-coordinate of the sphere. In a spreadsheet (like Excel) the variable name might be something like `A9`, representing the cell at column `A`, row `9`.

Basic Trigonometry. Know what a right triangle is and how sin, cos, and tan work with that right triangle. Know how the pythagorean theorem works with a right triangle.

Basic Trigonometry. Know that $\sin 0 = 0^\circ$, $\cos 0^\circ = 1$, $\sin 90^\circ = 1$ and $\cos 90^\circ = 0$.

Basic Trigonometry. Know the difference between a radian and degree and how to interconvert between them.

Basic Calculus. Given a function $y(x)$, know how to find basic derivatives, such as dy/dx and d^2y/dx^2. As an alternative notation, given $y(x)$, know how to find $y'(x)$ and $y''(x)$.

3.2 The idea of a function

Functions in mathematics. In mathematics, you should be familiar with the use of a function. For example if you know that $f(x) = x^2$, then you are free to *use* the function. You

can differentiate it: $f'(x) = 2x$. You can evaluate it at $x = 5$, or $f(5)$ to get 25. Functions can also have different names, like g, and be functions of more than one variable, like $g(x, y, t) = x^2 + y^2 - t^2$ for example.

Functions on your graphing calculator or a computer. *With computer, calculators, or spreadsheets* there are also functions, but instead of just returning a number, like `sin(x)` a function on a computer can cause something to happen, like to color the screen or draw a vector. Function names on a computer are typically longer than just f or g, such as `draw_vector`. Computer functions often have parameters too, in the same format as their mathematical counterparts, as in *name* then *parenthesized list of parameters*. So instead of $f(x, y)$, we'd have *draw_vector(tip,tail,color,label)* where *tip* and *tail* are the tip and tail coordinates of a vector to draw, with a color of *color* and a label of *label*. Calling this function doesn't return a number; it draws a vector on the screen.

3.3 Vectors: More than "magnitude and direction"

3.3.1 What's a vector?

When asked, most students will say that a vector is a "quantity with a magnitude and direction." There is much more to vectors than this textbook meaning, and the sooner you "become friends" with vectors, the easier time you're going to have in your core math and science classes.

To start, think of a vector as a "container" for information about an object. To emphasize the container aspect, we'll write vectors enclosed in a $<$ and $>$, or the "ordered set" notation (see above). As an example, an object might be located at $x = 5$, $y = 3$, and $z = -1$. In vector form, this would be written as $\vec{r} =< 5, 3, -1 >$, or $\overrightarrow{pos} =< 5, 3, -1 >$. In both cases, the 5, 3, and -1 are called the components (or parts) of the vector. The arrow over the symbol means it's a vector (it has the three components). Notice how compact the vector is as a container. One look at $< 5, 3, -1 >$ and you can immediately see the x, y, and z position of an object.

A velocity vector can be stated in the same manner. Suppose an object has an x-velocity of 6 m/s and a y-velocity of 2 m/s. It's v-vector could be written as $\vec{v} =< 6, 2, 0 >$, or $\overrightarrow{speed} =< 6, 2, 0 >$. Acceleration vectors can be written similarly. For example, an object in free fall has $\vec{a} =< 0, -9.8, 0 >$.

The real convenience of vectors is in their algebraic operations. For example, a vector can be multiplied by a scalar, by simply multiplying all of its components by the scalar. So, $5\times < 6, 2, 0 >=< 30, 10, 0 >$. This is useful in the physics equations $v = v_0 + a\Delta t$ and $x = x_0 + v_0\Delta t + \frac{1}{2}a\Delta t^2$, Because Δt is always a scalar (a time interval), but in these equations it's multiplied by either a or v_0, which can be vectors. This means that an a-vector of $< 0, -9.8, 0 >$ times a Δt of 2 seconds would be $< 0, -9.8, 0 > \times 2$ or $< 0, -19.6, 0 >$. But since $v = a\Delta t$, the $< 0, -19.6, 0 >$ is the object's new v-vector.

For a complete example, suppose $\vec{v_0} =< 2, 1, 0 >$ and $\vec{a} =< 0, -9.8, 0 >$. If you wish to know the object's new velocity after 2 seconds has gone by, you can use $\vec{v} = \vec{v_0} + \vec{a}\Delta t$, or $\vec{v} =< 2, 1, 0 > + < 0, -9.8, 0 > \times 2$. Working this, we'll get $\vec{v} =< 2, 1, 0 > + < 0, -19.6, 0 >$

or $\vec{v} =< 2, -18.6, 0 >$. In other words, after 2 seconds, the object is moving 2 m/s along the x-axis, 18.6 m/s along the $-y$-axis, and it is not moving at all along the z-axis.

3.3.2 Vector “lingo”

Vectors or Arrows. In physics we often draw arrows on objects to indicate that something is happening to it. The arrow is also called a “vector” and it’ll be labeled with some quantity, like F for force or v for velocity, etc. For example if you see a ball with an arrow pointing up and to the right, and the arrow is labeled v, you might be able to conclude that the ball is moving in that direction.

Discussing Vectors. Vectors often need to be described in words. Know what it means for a vector to point at “30° with the $+x$-axis,” “16° north of east,” or “south east.”

Operation on Vectors (1). If you have a vector, no matter what direction it is pointing, you should be able to find its x and y components. This is most easily done by drawing a small xy-coordinate system at the tail of the arrow. Next, treat the vector itself like the hypotenuse of a right triangle and draw in the legs, one along the x axis and the other along the y axis of your little coordinate system. Label in some angle and use sine and cosine as needed to find the lengths of the x and y components (the legs).

Operation on Vectors(2). If you have the x and y components of a vector, you should be able to draw the vector itself and determine the angle the vector makes with respect to the x-axis. This is all done with basic trigonometry. Invariably this will involve using $\tan^{-1}$ somewhere. You should also know how to find the magnitude of a vector, which comes from the Pythagorean Theorm; if you know the “legs” of a right triangle (the components), you should be able to find the hypotenuse (or the magnitude of the vector).

Handling Vectors. There are many ways of representing vectors; here are the most common.

- Magnitude and angle. Specify the magnitude (strength, length, etc.) and the angle. Like 10 m/s at 45° up from the $+x$-axis.
- $\hat{\imath}, \hat{\jmath}, \hat{k}$-notation (or engineering notation). Specify the components of the vector directly. $\hat{\imath}$ stands for x, $\hat{\jmath}$ stands for y and $\hat{k}$ stands for z. In this class the z-component will always be zero. So a vector written like $5\hat{\imath}+2\hat{\jmath}$ means a vector that has a strength of 5 units in the x-direction and 2 units in the y-direction. You should be able to find the magnitude and angle of this vector (if needed) directly from the 5 and the 2.
- Ordered set notation. $< x, y, z >$ where x, y, and z are the components of the vector, so $< 5, 2, 0 >$ would be the same as the vector above.

3.3.3 The Common Manipulations: the “magnitude and direction”

The notation $< x, y, z >$ is a very succinct vector notation, as it explicitly shows the three components of a given vector. But regardless, this is what a vector is: a quantity that has

three components, one for x, one for y and one for z. All of the common vector results can be found by simply using these three bits of information. Two of the most important results are the magnitude of a vector and angle the vector makes in the xy-plane, relative to the z-axis.

3.3.4 The magnitude of a vector

The magnitude of a vector is how long its arrow is. At the risk of confusing physical terms, think of the magnitude of a vector as the "power" or "strength" of a vector. It is always found my squaring each component, then adding the squares together, and taking the square root of the final sum.

For example, suppose $\vec{v} =< 3, 2, -1 >$. The magnitude of v can be found by $|v| = \sqrt{3^2 + 2^2 + (-1)^2}$, or $|v| = 3.7$.

The angle or direction of a vector

The direction (most commonly needed in the xy-plane) is best classified as an angle with respect to some axis, perhaps the $+x$-axis. If you know the x and y components of a vector, say v_x and v_y of vector $\vec{v}$, you can always find the angle the vector is making (with the $+x$-axis) using

$$\tan^{-1} \frac{v_y}{v_x}. \tag{3.1}$$

For example, suppose $\vec{v} =< 3, 2, -1 >$. The angle the v vector is oriented at in the xy-plane is found from $\theta = \tan^{-1}(2/3)$ or 33.6° with respect to the $+x$-axis.

Chapter 4

Drawing with Povray

4.1 Introduction

Povray is a drawing tool. It is software that *calculates* and *renders* a scene for you, based on the mathematical relationship between the camera, light source and objects. Thus, once a camera and light source are placed, you have a virtual, three-dimensional world in which you can draw. This chapter will brief you on the coordinate system, examples of how to draw into Povray, and a list of Povray-defined colors you may choose from.

4.2 Coordinate System

The coordinate system is Povray is similar to the "cartesian coordinate system" that you are used to from your many years of math in school. It looks like that shown in Figure 4.2. The $+x$-axis runs right, $-x$ is left, $+y$ is up and $-y$ is down. The only difference now is that the $+z$-axis is into the screen and $-z$-axis is out of the screen, into the direction of the viewer. Technically this is a "left-handed" coordinate system, instead of a "right-handed" coordinate system. The right handed system is the type you use in your math classes; the left handed system is useful for optimizing computer graphics, so it is pretty standard in this realm.

Drawing in Povray is then simply a matter of visualizing the coordinate system, selecting an object, and telling Povray to draw into the coordinate system.

4.3 Drawing Examples

Povray is able to draw many "primitive" shapes, like spheres, boxes, cylinders, and cones. A whole list can be found here http://povray.org/documentation/view/3.6.1/273/. In our study of physics, we can get a lot done by using only a few. Let's start with the sphere.

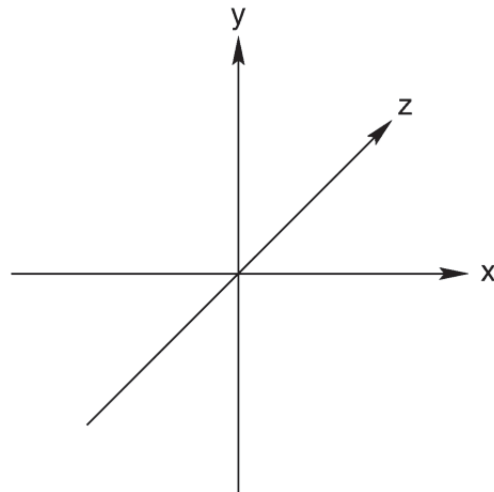

Figure 4.1: The Povray coordinate system: $+x$ runs right, $-x$ is left, $+y$ is up and $-y$ is down, $+z$ is into the screen and $-z$ is out of the screen, into the direction of the viewer (from http://www.povray.org/documentation/view/3.6.0/15/).

4.3.1 Spheres

Spheres are defined by a position and radius. This is also true in Povray, in addition to the color with which is should appear. The syntax for a sphere is:

- `sphere {<center-x,center-y,center-z>,radius pigment {color}}`

Note word `sphere` and all of the curly braces are required, as they're part of the Povray syntax. Note the vector in ordered-set form specifying the center. Note the comma then the radius parameter, and the word pigment which specifies the color which is to be put the the curly brackets. Here are some examples:

sphere {$<0,0,0>$,1 pigment {Blue}} Draws a blue sphere of radius 1 at the origin.

sphere {$<0,0,-5>$,1 pigment {Blue}} Draws a blue sphere of radius 1 at at $x=0$, $y=0$, and $z=-5$.

Of course drawing objects at fixed positions is nice for scenery, but not so for studying the physics of motion. But remember, we communicate with Povray via these simple, text-based commands, so the vector position of the sphere can be replaced by a variable (that itself is a vector). Consider this code:

```
#declare pos=<5,1,0>;
sphere {pos,1 pigment {Blue}}
```

The `sphere` statement is complete, but the position vector doesn't appear explicitly. It is instead contained in the variable called `pos`. Now granted this code draws a blue sphere at $< 5, 1, 0 >$, which is a fixed location. But, the advantage of doing this is that `pos` may be the result of a calculation, as in `pos=pos+vel*dt+0.5*a*dt*dt`, or in other words, a physics equation! This is the real power of using Povray for producing animations:

> Because Povray *calculates* images, based on the mathematical position of objects, we can use *physics* and its results to tell Povray where objects are to be drawn. This will give us a physically-realistic scene.

4.3.2 Boxes

Boxes can be useful for representing objects too. Drawing a box looks like this

- `box` $\{< x1, y1, z1 >, < x2, y2, z2 >$ `pigment` $\{$`color`$\}\}$

where the two vector points $< x1, y1, z1 >$ and $< x2, y2, z2 >$ are any two opposite corners of the box. That is, if you tell Povray two corners of the box, it'll fill in the rest with smooth sides having the color of `color`. Here is an example:

box {$< -1, -1, -1 >, < 1, 1, 1 >$ **pigment {Green}}**

What about basing the position of a box on a variable, as in using `pos` in the sphere section above? It's most convenient to use some position variable as the center of the box and calculate each corner relative to this position. Supposing again that `pos` is our position variable, one could do this

```
#declare pos=<5,1,0>;
box {pos-<1,1,1>,pos+<1,1,1> pigment {Blue}}
```

where a vector displacement of $< 1, 1, 1 >$ is explicitly subtracted and added to a base position of `pos` to defined the two corners. The box would have a side length of 2 in this case.

Povray also has a vector shorthand notation where 1 means $< 1, 1, 1 >$ (and 2 means $< 2, 2, 2 >$, etc.), so this would work as well

```
#declare pos=<5,1,0>;
box {pos-1,pos+1 pigment {Blue}}
```

4.3.3 Cylinders

Cylinders are useful for a variety of drawing. Lines can be drawn using very thin cylinders, for example. Drawing a cylinder looks like this

- `cylinder` $\{< x1, y1, z1 >, < x2, y2, z2 >$, `radius pigment` $\{$`color`$\}\}$

where the two vector points $< x1, y1, z1 >$ and $< x2, y2, z2 >$ are the ends of the cylinder and `radius` is the radius of the cylinder, and of course the pigment construct is used to specific the color of the cylinder. The same rules apply for substituting the explicit vector ends with variables.

4.3.4 Planes

Planes are useful for representing the ground in a physics problem. A plane in Povray has this syntax

- `plane` { $< n_x, n_y, n_z >$, `level pigment` {`color`}}

where $< n_x, n_y, n_z >$ is a vector that you wish to be *normal* (or perpendicular) to your plane. This vector essentially sets the orientation of the plane. The `level` parameter is a number of how far the plane should be displaced along the normal vector. The pigment sets the color of the plane as in the above examples. The plane that is drawn to be very thin along the normal axis, while extending to infinity in all other directions.

plane {$< 0, 1, 0 >, 0$ **pigment { Blue }}** Draws a blue plane whose normal is along the y-axis, with the x and z axes in the plane.

plane {$< 0, 1, 0 >, -5$ **pigment { Yellow }}** Draws a yellow plane whose normal is along the y-axis, with the plane being lowered 5 units below the x and z axes.

plane {$< 1, 1, 0 >, 0$ **pigment { Green }}** A crooked green plane.

plane {$< 0, 1, 0 >, 0$ **pigment {checker color Red, color Blue }}** A red and blue checkered (or tiled) plane, which is somewhat of a "standard" in computer graphics.

You have to think a bit about the positions of your camera and light source if your plane isn't visible.

4.4 Colors

Colors are often linked to objects using the `pigment` {`color`} construct as in the examples above. For the `color` part here, you may substitute any of the following. Note that color names are *case sensitive!* So you can use `Black`, but `black` or `BLACK` won't work.

4.4.1 Stock Colors

`Aquamarine BakersChoc Black Blue BlueViolet Brass BrightGold Bronze Bronze2 Brown CadetBlue Clear CoolCopper Copper Coral CornflowerBlue Cyan DarkBrown DarkGreen DarkOliveGreen DarkOrchid DarkPurple DarkSlateBlue DarkSlateGray DarkSlateGrey DarkTan DarkTurquoise DarkWood DimGray DimGrey DkGreenCopper DustyRose Feldspar Firebrick Flesh ForestGreen Gold Goldenrod Gray Gray05 Gray10 Gray15 Gray20 Gray25 Gray30 Gray35 Gray40 Gray45 Gray50 Gray55 Gray60 Gray65 Gray70 Gray75 Gray80 Gray85 Gray90 Gray95 GreenCopper GreenYellow Grey HuntersGreen IndianRed Khaki Light_Purple LightBlue LightGray LightGrey LightSteelBlue LightWood LimeGreen Magenta MandarinOrange Maroon Med_Purple MediumAquamarine MediumBlue MediumForestGreen MediumGoldenrod MediumOrchid MediumSeaGreen MediumSlateBlue MediumSpringGreen MediumTurquoise MediumVioletRed`

`MediumWood Mica MidnightBlue Navy NavyBlue NeonBlue NeonPink NewMidnightBlue NewTan OldGold Orange OrangeRed Orchid PaleGreen Pink Plum Quartz RichBlue Salmon Scarlet SeaGreen SemiSweetChoc Sienna Silver SkyBlue SlateBlue SpicyPink SpringGreen SteelBlue SummerSky Tan Thistle Turquoise VLightGray VLightGrey Very_Light_Purple VeryDarkBrown Violet VioletRed Wheat White Yellow YellowGreen`

4.4.2 Make your own color

If you don't find a color you like from the above table, you can make your own color by mixing various amount of red, green, and blue. The fraction of each color is a number between 0 and 1. Colors can be made by a construct like `color rgb` $< r, g, b >$, where r, g, and b are numbers between 0 and 1, or the fraction of red, green, or blue respectively. Instead of a color name, as in the above, the `color rgb` $< r, g, b >$ construct would go inside of the curly braces of the pigment. Thus a pigment construct might be of the form `pigment {color rgb` $< 0.5, 0.5, 0.5 >$`}` would be a shade of gray. `pigment {color rgb` $< 1, 0, 0 >$`}` would be pure red, etc.

Chapter 5

Getting Started with Simple Programming

5.1 Introduction

In a word, "yes," this approach to studying physics will require you to write simple computer programs. Most students do not like programming and do not know how to already. But as a budding scientist, knowing something about programming is a good idea, since computer software essentially runs the world (see "Software eating the world" here `http://goo.gl/zTtuQ`). Also, as you grow to need computers to do more and more sophisticated tasks (perhaps as part of a future research project), "point and click" software won't always accomplish what you need.

Luckily, Povray makes the programming work about as simple as it can possibly get. For us, it will really be a matter of putting together simple sequences of text lines that will enable you to study physics and make all kinds of interesting animations.

5.2 Skeleton Code

All movies you will create can be started with the following code, called the "skeleton code" because it is pretty bare. If you want a copy of it, can you can find it in this folder `http://goo.gl/zTtuQ` .

```
#include "physics.inc"
camera { location <0,0,-15> look_at <0,0,0> }
light_source { <-0,0,-20> color White }
#declare pos = <0,0,0>;
#declare vel = <0,0,0>;
#declare dt = 0.1;
#declare xtime = 0.0;
#while(xtime <= clock)
        #declare a = <0,0,0>;
```

```
        #declare pos = pos + vel*dt + 0.5 * a * dt * dt;
        #declare vel = vel + a *dt;
        sphere { pos,0.1 pigment {Yellow} }
        #declare xtime = xtime + clock_delta;
#end
sphere { pos,1 pigment {Red} }
draw_vector(pos,vel,Green,"v")
draw_vector(pos,a,Yellow,"a")
```

This program is 17 lines long. If you type it into the Povray or MegaPov text window, then prepare the software for animation (see Section 2.2.3 for Windows, or Section 2.3.3 for OSX), you'll see a red sphere drawn on the screen that will just sit there. Pretty boring, but you've created a working program for Povray/MegaPov that actually creates a scene.

Look over the lines. Computers execute the lines one at a time from top to bottom. Here are some symbols you'll find:

- Notice the ordered set vector notation throughout, as in $< 0, 0, 0 >$. Constructs like this identify x, y and z components of a vector.
- "pos" is a variable name short for "position," or the position of the object.
- "vel" is a variable name short for "velocity," or the speed control of the object.
- "a" is the acceleration.
- "dt" is the time step, or Δt.
- Notice the two physics equations.
- Notice two lines that draw spheres on the screen. Sphere need a center point (pos), a radius (1 or 0.1 in this case), and a pigment or a color to be drawn with.
- Notice placement of a camera and a light source, as required by raytracing.

When rendered, why doesn't the sphere move? Well, read the program. It looks like the variable `vel` (short for velocity) has all three components of it set equal to zero. Also, the variable `a` (acceleration) has all three components zero as well. Think physics now: if an object is at rest $\vec{v} =< 0, 0, 0 >$ and has a zero acceleration $\vec{a} =< 0, 0, 0 >$, will it ever move? No. This is, in fact, of of Newton's Laws ("an object at rest, stays at rest..."). To get this sphere to move, change the `#declare vel=`$< 0, 0, 0 >$`;` line to `#declare vel=`$< 5, 0, 0 >$`;`. Re-render the program. You should see the sphere moving right across the screen. See? You're already testing physics theories using computer animation.

If the red sphere is too big, you can make it smaller by changing its radius from 1 to something smaller in the line `sphere { pos,1 pigment {Red} }` . If the vector scales are too small (or large) for you, you can alter this in the line `set_vector_scale(0.5)`. Label sizes and vector thicknesses can be changed in these lines.

```
set_vector_label_scale(0.5)
set_vector_thickness(0.5)
```

So you really have full control over all aspects of your movie by making small edits and tweaks to the skeleton code. Here are some things to try:

- To make the sphere go faster, change the `#declare` $a =< 0, 0, 0 >$; line to `#declare` $a =< 5, 0, 0 >$;
- To make it slow down, stop and turn around change the line to `#declare` $a =< -2, 0, 0 >$;
- To change where the sphere starts on the screen, you can change the line `#declare pos` $=< 0, 0, 0 >$; to `#declare pos` $=< -3, 0, 0 >$; The Povray coordinate system is what you'd think: $+x$ runs right and $-x$ runs left; $+y$ is up and $-y$ is down. $-z$ is straight at you as the viewer, where $+z$ is into the screen.
- You can also make the sphere travel along a different axis, since the ordered set notation contains information about the x, y and z axes. So try setting the velocity to `#declare vel=`$< 0, 5, 0 >$ and `#declare a` $=< 0, -3, 0 >$;

5.2.1 Making a movie

Changing the code and re-rendering the scene is where most of your work will be. Today's computers (and Povray) are just fast enough to allow you to see a "flickery" version of your animation in the render window. This allows for a convenient edit→render→edit cycle. Remember that as Povray works, it creates one image file on your computer per scene you see rendered. Each image file represents your scene at a slightly later time. To make the final movie, you have to stitch these images together into one single movie file. Then you'll see a beautiful, smooth animation that'll look quite nice. See the software installation chapter on how to stitch these images together, but in sum, download stitching software, either for Windows or Macintosh here `http://goo.gl/zTtuQ` .

5.3 Basic Povray Statements

To create physics movies, you will not need to know or learn an endless list of programming statements, and this is not a programming class. But invariably as you use a computer as a tool in science, "point and click" software won't always do just what you'd like. You'll *have* to program some kind of macro or line-by-line list of instructions for the computer. The same applies here. With that, here are the few instructions you'll have to be familiar with:

#declare This statement is what allows you to set a variable equal to some value. In a math class, you'd write $x = 5$ to set the variable x to 5. In Povray, you need to start the line with "#declare" then a variable name, then an equal sign, then what you want to set it to, followed by a semi-colon. Here are some examples:

- #declare a=5;
- #declare Radius=10;
- #declare velocity=$< 5, 1, 0 >$;

Notice in these examples that Povray recognizes both scalar and vector numbers. The vector notation uses "ordered set notation" exclusively.

#declare (again) ****CAUTION**** The #declare statement is the only Povray line that must be ended with a semi-colon. This is needed so Povray knows where the right hand side of a #declare action ends. It can be very frustrating to leave off such a semicolon. It will result in your movie scene not rendering. Be careful!

sphere Most objects discussed in this class can be represented by round spheres. A sphere is characterized by its center position in 3D space, its radius, and a color. The sphere statement looks like this

```
sphere {<center-x,center-y,center-z>,radius pigment {color}}
```

where the first vector is the 3D center of the sphere. It can be a vector variable previously declared, or a literal like $< 5, 2, -1 >$. The `radius` is just a number and the `color` is what color you'd like to see the sphere appear with. Basic colors like `Red`, `Blue`, `Green`, `White`, and `Yellow` are available. A complete list can be found in this folder: `http://goo.gl/zTtuQ` .

box Boxes can be useful for representing objects too. Drawing a box looks like this

```
box {< x1, y1, z1 >, < x2, y2, z2 > pigment {color}}
```

where the two vector points $< x1, y1, z1 >$ and $< x2, y2, z2 >$ are any two opposite corners of the box. That is, if you tell Povray two corners of the box, it'll fill in the rest with smooth sides having the color of `color`.

cylinder Cylinders are useful for a variety of drawing, including objects that move or otherwise. Drawing a cylinder looks like this

```
cylinder {< x1, y1, z1 >, < x2, y2, z2 >, radius pigment {color}}
```

where the two vector points $< x1, y1, z1 >$ and $< x2, y2, z2 >$ are the ends of the cylinder and `radius` is the radius of the cylinder, and of course the pigment construct is used to specific the color of the cylinder.

Other shapes Povray knows how to draw a variety of shapes. You can find documentation on them here http://www.povray.org/documentation/view/3.6.0/273/.

Vector components Vector components of a variable can be accessed by adding a ".x," ".y," or ".z" to the end of a variable. So for instance, if you need just the x-component of position, you can use `pos.x` (given that `pos` is the name of your position vector). If `vel` is your velocity variable, then `vel.y` would be the y-component of the velocity.

Doing math. Povray (and most programming language) do not recognize "implied multiplication" like $5x$. You have to explicitly tell it to multiply 5 and x using the * symbol, which means multiply. So, $5x$ would be programmed as $5 * x$ wherever it's needed. Signs like $+, -$ do what you'd think. Divide is the forward slash or /. There is no convenient exponent, so for squaring or cubing quantities, just multiply them by themselves two or three times as needed.

More math Other math-related items are `pi` (all lower case), and of course `sin`, `cos`, and `tan`, which take radians (not degrees) as their arguments. Povray has a lot of built in functions which you can find here http://www.povray.org/documentation/view/3.6.1/228/.

More math Square roots are done with `sqrt`, which is useful for finding the magnitude of a vector as in `sqrt(vel.x*vel.x+vel.y+vel.y)`. As an example, suppose you needed the magnitude of a v-vector. We'll call the result `vmag` (assuming your velocity vector is in a variable called `vel`), you could write

```
#declare vmag = sqrt(vel.x*vel.x+vel.y+vel.y);
```

More math The arctangent is normally used to find the angle a vector is making with respect to the $+x$-axis. Povray has a special version of arctangent called `atan2` which works like this. Suppose you want to take the arctangent of Q_y/Q_x or $atan(Q_y/Q_x)$, where Q_x and Q_y are the components of some vector. In Povray you'd do `atan2`(Q_y, Q_x). So suppose you wanted the angle a velocity vector is making (with respect to the $+x$-axis), assuming your velocity vector is in a variable called `vel`. To assign this angle into a variable called `angle` You could write

```
#declare angle = atan2(vel.y,vel.x);
```

5.4 Structure of the Skeleton Code

If you're not much of a programmer, then you're in luck. The skeleton code (that will form the core of all work here) is a highly structured roadmap, guiding you through your studies. It is almost "form like," in that you need only fill in the relevant physics in order to create an animation. This structure is shown in Figure 5.4, and remember that the computer will process it line-by-line, from top to bottom.

Most of the time, you will start a movie by thinking carefully about what physics you need to illustrate. Often in mechanics, this comes down to recognizing three things:

1. Where should the object start?
2. What velocity should it have when it starts?
3. What acceleration should it have?

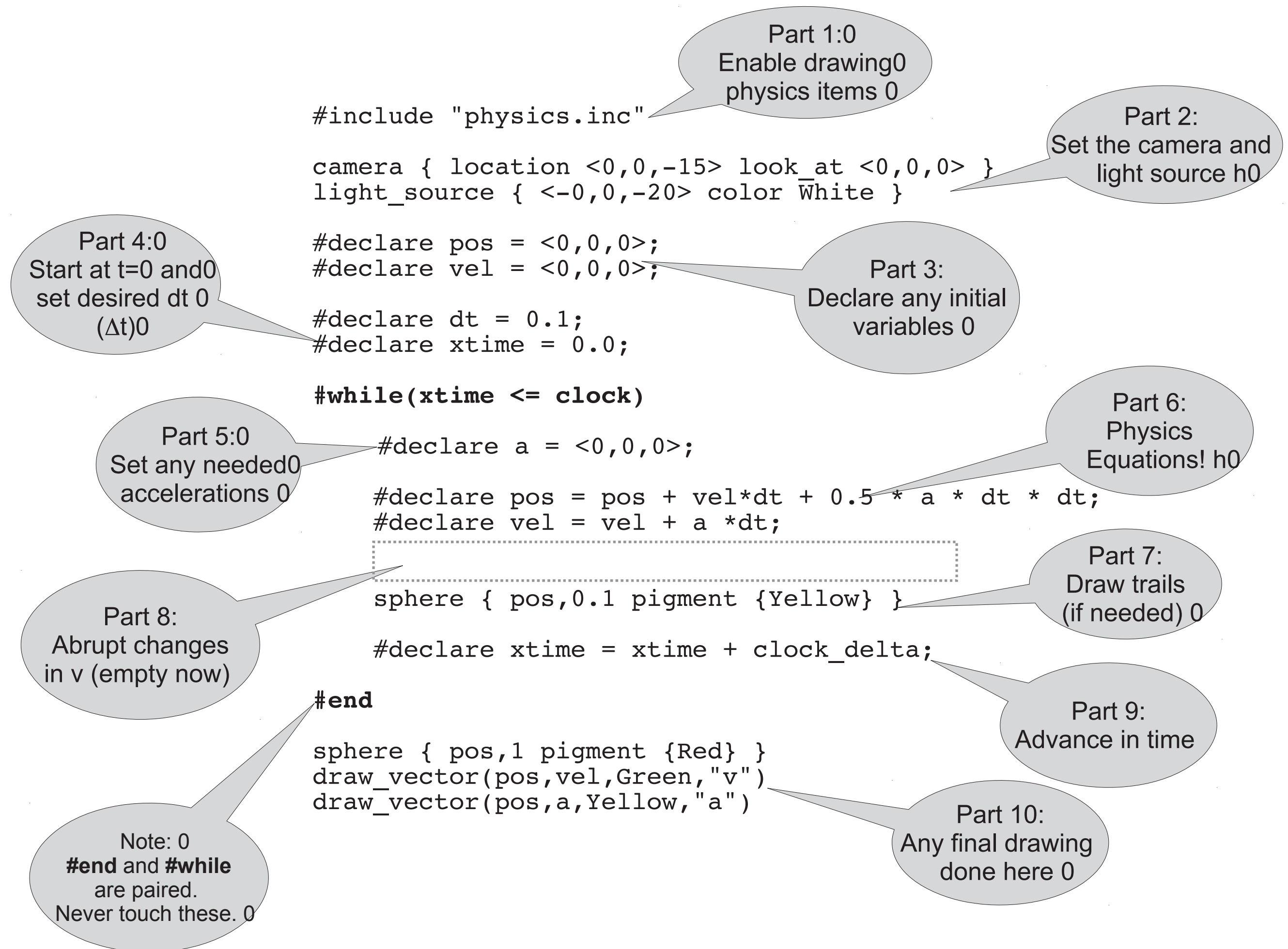

Figure 5.1: The highly structured skeleton code, which serves as the core for all work here.

4. What do I want to draw for my object?

If you can answer these four questions, then you can fill in the skeleton code and produce an associated movie. Part 3, for example is where the initial position and velocity of your object should be set, in full vector form (of course). Part 5 is where you'd put in the needed acceleration, again in full vector form. As far as drawing your object, that would all go in Part 10. If you can do this, without being tempted to change other parts of the code, then you should be able to quickly produce a working movie.

Here are a few more notes about the skeleton code structure:

- Part 1 always has to be there. Never change it. This makes drawable physics items available to you, like vectors, ropes, etc. What's in this file is discussed in Section 8.3.
- Part 2 can largely be left as-is throughout. However, if you want to change where the camera is located or at what position it is aimed (look_at), feel free to experiment. The

same goes with the position of the light source. The most popular change to Part 2 is in the z coordinate of the camera position. This allows you to zoom in or out on your scene (if things are too small or too large).

- Two things happen in Part 4. First, the frame-to-frame time-step of your movie, or Δt (`dt`), is set. A $dt = 0.1$ works well most of the time. But if you want an ultra-smooth, finely timed movie, you'll want to decrease dt maybe to 0.01. Second, the variable `xtime` is set equal to zero. This variable is the global "clock" in your movie, always holding the amount of time that has elapsed.
- Part 6 never needs to be changed as they are immutable laws of physics. But if you look carefully, you have to satisfy what they need to work by the time this part of the code is reached. In particular, you have to have the acceleration needed defined and ready to go, so the equations can do their work. Luckily though, Part 5 is where any accelerations must be defined, so they may feed into the physics equations of Part 6.
- If you want trails to be seen as your object moves, leave the `sphere` statement in Part 7. If you do not want trails, take it out. Feel free to alter the size or color of the trail spheres as well.
- Part 10 is special because the code is about to end and what is certain is that the latest, up-to-date values of all physics variables are known in this part. In other words, `pos` contains the best known and latest position of the object, and `vel` contains the best known, and latest velocity. That's why all final scene drawing happens in this part of the code, because you know where the object is, and what it's velocity is.
- Lastly, **never change or alter any these lines**:

  ```
  #include "physics.inc"

  #declare xtime = 0.0;

  #while(xtime <= clock)

  #declare xtime = xtime + clock_delta;
  ```

  or the

  ```
  #end
  ```

  line. They are needed to complete the logic that drives an animation to run. If you're curious though, the variable `xtime` holds the simulated time over which the movie has been running; it's like the internal "clock" for the movie. The `#while-#end` pair form a "while loop" (see http://en.wikipedia.org/wiki/While_loop) that ensures each run through your code movie occurs at a later and later time.

5.5 The if statement

If statements allow your program to *make decisions* as they run. The decision determines if a given section of your code will get visited by the computer or not. Here is an example.

Suppose you were creating a movie showing a block is sliding horizontally along a table, toward the table's edge. While on the table, the acceleration of the block along the y-axis is zero, since the block is sliding horizontally. Now suppose the table ends and the block ends up in free fall. The acceleration of the block now is now -9.8 m/s^2. So you really have two acceleration zones here. The acceleration along the y-axis, or $a_y = 0$ if the block is on the table, and $a_y = -9.8$ if the block is over the edge of the table. Read this last sentence again; the word "if" is used twice! Hence the "if" statement; logic dictates that a condition must be evaluated before a conclusion can be reached. In this case, the position of the block must be assessed (the condition) before a conclusion (on what a_y should be) can be reached.

5.5.1 An if statement in Povray

In Povray, here's how you would assign the acceleration to the block in Part 5 of the code, supposing that `pos` is the position vector of the block. Suppose also that the table extends from $x = -5$ to $x = 0$:

```
#if (pos.x < 0)
     #declare a=<0,0,0>;
#end

#if (pos.x >= 0)
     #declare a=<0,-9.8,0>;
#end
```

Note that we can't just say "if" in Povray, but instead `#if`; that is we need the # symbol in front of the word "if." This is just how Povray was made.

Reading the code top to bottom, you see the first `#if` statement: it reads "if the x-component of pos is less than zero." If this condition is true then the `#if` statement will execute all programming lines between the `#if` statement and the `#end` tag. If the condition is false, the computer will skip to the first line after the `#end` tag. Read the second `#if` statement: it reads "if the x-component of pos is greater than or equal to zero." The same execution branch holds depending on if this condition is true or false. For both `#if` statements, you can see the associated assignment of an acceleration vector, (i.e. zero if it's on the table, or -9.8 along the y-axis if it's in free fall).

In Povray, here is the general form of an `#if` statement, that you may play anywhere in your code:

```
#if (condition)
   These lines will be executed
```

```
   if condition is true.
#end
```

Again, if `condition` evaluates to true, the lines between the `#if` and `#end` will be executed. If `condition` is false, the code execution will skip to the first available line after the `end` statement.

There's even a more general form that has explicit groups of lines to be executed if condition is true or false. It looks like this

```
#if (condition)
   These lines will be executed
   if condition is true.
#else
   These lines will be executed
   if the condition is false.
#end
```

and is called an "if-else" structure.

5.5.2 Conditions

Now what about the "condition" mentioned above. How are these formed? Conditions that you've seen before are most like expanded searches you might do in the library. You might try to find a book about "computers," so you could run a search for this term. But you might like a book about "computers" written by an author named "Doe." Your search would then resemble "subject=computers and author=doe." This is an example of a condition that might be used in programming.

Generally speaking a condition must evaluate to either true or false. You can use symbols from math to test the values of variables, such as $<$, $>$, or $=$ to test for less than, greater than, or equal to. Povray has more such as $>=$ and $<=$. Use $! =$ for "not equal." These simple conditions may be tied together using & for "and" and | for "or."

Table 5.1 shows some example conditions that might be helpful when animating physics. It assume the position of an object is being held in the vector variable `pos` and its velocity in a variable `vel`.

5.5.3 Testing *where* an object is

As you might guess, testing *where* an object is located is best done by setting up conditions using the x, y, and/or z components of a variable that holds the object's vector position. So if `pos` holds your object's position, `pos.x`, `pos.y`, and `pos.z` are the components that you may use to form some conditions to check where the object is located.

To check if the box above is on the table, for example, we might write `pos.x >= -5 & pos.x <= 0`. This condition would evaluate to true if the x component of the object's position is greater than or equal to -5 or less than or equal to 0 (which are the horizontal bounds of the

Condition	**Description**	**Possible Use**
`(pos.x > 0)`	True if the x-component of the position is positive	Check if an object has passed a certain point in space.
`(vel.x < 0 & pos.x < -3)`	True if v_x is less than zero (the object is moving left) and the object has moved to the left of $x = -3$	Check for an imminent collision with an object moving leftward into an object whose edge is at $x = -3$.
`(vel.y >= -2 | pos.x = 1)`	True if the v_y is greater than or equal to -2, and and the x-component of `pos` is equal to one	See if the object has exceeded 2 units of speed in the downward direction, and if the x coordinate of the object is at 1.
`(vel.y < 0 & pos.y <= 0)`	True if the object is moving down ($v_y < 0$) and the y-position is zero or lower.	See if the object is moving down and has encountered the ground at $y = 0$. Good for now reversing the sign of v_y, to make the object bounce off of the ground.
`(dist < R1+R2 & vel1.x> 0 & vel2.x< 0)`	With `dist` set to the center-to-center distance between two spheres (of radius R1 and R2), see if the distance is less than the sum of their radii (i.e. closest approach), and if object 1 is moving right ($v_x > 0$) and object 2 moving left ($v_x < 0$).	Collision detection between two spheres.

Table 5.1: Sample conditions, what they test, and where they might be used, assuming `pos` and `vel` are the position and velocity vectors of the object.

table). The condition itself would be written inside of the parentheses that is part of the `#if` statement construct described above.

5.5.4 Testing *the direction* in which an object is moving

The *direction* an object is moving in may be tested by examining the x, y, and/or z components of a variable that holds the object's vector velocity. So if `vel` holds your object's velocity, `vel.x`, `vel.y`, and `vel.z` are the components that you may use to form some conditions. For example, an object is moving toward the right if `vel.x > 0`. It is moving up if `vel.y > 0` and so on. The condition itself would be written inside of the parentheses that is part of the `#if` statement construct described above.

5.5.5 Careful with =

When crunching numbers for physics movies, be careful with expecting the "=" (equal) to be a robust condition. In the example of the box on the table, the edge of the table is at $x = 0$. So suppose we were testing if the object reaches the edge by writing `if (pos.x = 0)`. Would this work? Sometimes yes and sometimes no. Why?

It's because of how numbers work on the computer. As the computer crunches away in computing the position of your block using $x = x_0 + v_0\Delta t + 1/2a\Delta t^2$, numbers might be non-integers. Perhaps the box starts at -5 and moves toward the right. The next position might be -4.8 then -4.63, then -3.225. As it approaches the edge, the position might be -0.005, -0.001, then 0.002. So it crossed the edge of the table between -0.001 and 0.002, but it never hit zero *exactly* so the condition of `pos.x =0` would *never* become true.

What does one do? Make the condition more accommodating. Never expect a bunch of computed decimal numbers to ever be anything exact. Instead check for a range, that will typically involve $<$ and $>$ than compares. Here are some most robust possibilities for checking on the edge of the table.

pos.x $>=$ **0** Will be true if the box is off of the table.

pos.x $<=$ **0** Will be true if the box is on the table.

pos.x $>=$ **-0.1 & pos.x** $<$ **0.1** Will be true if the box is within $|0.1|$ units of the table's edge. You can hone in the 0.1 as needed, but this is about as good as condition as you'll create for checking if the box is at the table's edge.

abs(pos.x) $<$ **0.01** Checks if the absolute value of the box's x component of position is less than 0.01. This will be true for $-0.01 \leq pos.x \leq 0.01$. A pretty nice edge condition!

pos.x $= 0$ Not a robust condition. Hard to say if the computer's predicted position will ever *exactly* be zero.

Chapter 6

How do I...

6.1 Use a vector component of a vector variable?

Use `.x`, `.y`, or `.z` after the variable name. So if your vector variable is called `pos`, the x-component can be accessed via `pos.x`. The y-component via `pos.y`.

6.2 Find the angle a vector is making with respect to the $+x$-axis?

You need to take the arctangent of the y-component divided by the x-component, or $\theta = \tan^{-1}(x/y)$ of the vector. Use the Povray `atan2` function. So if the vector is called `vel`, you would do `#declare angle=atan2(vel.y,vel.x);`. This will put the angle of the vector `vel` into the variable `angle`.

6.3 Draw a vector on an object?

Use `draw_vector(tail,vector,color,"label")` as outlined in Section 8.3. This will draw the vector in vector-variable `vector` with its tail at vector position `tail`. It'll have the color `color` and be given a text label of `label`, which is the text you put in double quotes as the last parameter. See the `set_vector...` functions in Section 8.3 to tweak the size of the vector as needed.

6.4 Draw just the x or y component of a vector on an object?

Review the question above first. Now, for just a component, the "vector" part of the parameter list for `draw_vector` is where you construct a vector that represents just the x or y component. If you want to draw v_x given that your object's velocity variable is `vel`, the vector parameter

would be $< vel.x, 0, 0 >$, since the x component of a vector has a zero y and z component. Just drawing v_y would be $< 0, vel.y, 0 >$

6.5 See if two objects have collided?

Suppose you have two spheres of radius `R1` and `R2`, where initially sphere 1 is to the left of sphere 2. Suppose sphere 1 is moving right and sphere 2 is moving left, and the positions of each are in the vector variables `pos1` and `pos2`. First you need to compute the distance between the two objects using the distance formula like this

```
#declare d=sqrt((pos1.x-pos2.x)*(pos1.x-pos2.x)+(pos1.y-pos2.y)*(pos1.y-pos2.y));
```

or

```
#declare d=vlength(pos1-pos2);
```

An if statement to see if they collided would check three things. 1) if `d` is less than the sum of the two radii (which is their closest possible approach), 2) if sphere 1 is moving toward the right 3) if sphere two is moving toward the left, like this:

```
#if (vel1.x>0 & vel2.x<0 & d<= R1+R2)
   do this if they collided
#end
```

6.6 See if an object has hit the ground?

If the ground is at $y = 0$, see if the object is moving down and if its position is at or below $y = 0$, like this

```
#if (vel.y<0 & pos.y <= 0)
   do this if the object hits the ground
#end
```

6.7 Add trails behind an object?

Check the skeleton code, Figure 5.4, Part 7. Put in the sphere drawing statement to draw trails. If you have more than one object moving, you'll need a sphere statement in Part 7 for each one.

6.8 Remove trails behind an object?

Check the skeleton code, Figure 5.4, Part 7. Remove the sphere drawing statement to draw trails.

6.9 Draw an object as a sphere given that I know its position?

If its position is in a position vector called `pos`, you can draw a sphere at its position with a line like

```
sphere {pos,1 pigment {Red}}
```

See Figure 5.4, and be sure you put this line in Part 10.

6.10 Draw an object as a box given that I know its position?

The easiest way is to use the `draw_box` statement as outlined in Chapter 6, as in `draw_box(pos,2,Red)` which will draw a red box at the position given by the vector variable `pos`, with a side length of 2.

If you need a more general approach, then if a position is in a vector called `pos`, you can draw a box centered at `pos` with a line like

```
box {pos-<1,1,1>,pos+<1,1,1> pigment {Red}}
```

Where the $\pm < 1,1,1 >$ are references to opposite corners of a box, which is what Povray needs to render a box. See Figure 5.4, and be sure you put this line in Part 10.

6.11 Draw something other than a box or sphere for my object?

See `draw_car` and `draw_rocket` in Chapter 8.

6.12 Draw the ground or a big wall in a scene?

If you want the ground at $y = 0$, do a

```
plane{<0,1,0>,0 pigment {Green}}
```

If you want the ground to be a checkerboard pattern, do this

```
 plane { <0, 1, 0>, 0 pigment {checker color Red, color Blue } }
```

Note that the vector is the vector normal to the plane and the scalar number if how far to translate the plane up or down along the normal vector.

6.13 Find the magnitude of a vector?

If a vector is called $\vec{A}$, its magnitude is $A = \sqrt{A_x^2 + A_y^2}$. In Povray this can be done two ways

1. `#declare Amag=sqrt(A.x*A.x+A.y*A.y);`
2. `#declare Amag=vlength(A);`

either statement will put the magnitude of $\vec{A}$ into the variable called `Amag`.

6.14 Get a spring to look right as it pushes against a moving object?

Here are some pointers about springs:

- Springs have a fixed and and a free end. Do you know the position of the fixed end?
- Springs have an equilibrium position. Do you know where the equilibrium position is to be? You should put this in a variable called `s0`.
- Springs have a spring constant. Be sure to declare this in Part 3 of the skeleton code.
- The position of the free end of the spring should be a variable because it will change. Let's call it `s`. It'll be the equilibrium position when your object is not in contact with the spring. When your object is in contact with the spring, it'll be at the same position as the object.
- Initially, supposing your object is not in contact with the spring, the free end will be at the spring's equilibrium position. This means that in Part 3 of the skeleton code, you should declare a value for `s0`, the equilibrium position, then in a second declare statement, set `s` equal to `s0`. This sets the equilibrium position in `s0` and puts the free end of the spring there too (`s`).
- The interaction between the spring and your object can be tricky, but can be handled with one or two `#if` statements. The important task is to get the spring accelerations assigned for Part 5 of the skeleton code. Think about it all like this. Suppose your object starts to the right of `s0` and is moving toward it. We'll assume its position is held in a variable called `pos`. Assume everything is aligned along the x-axis. As long as `pos.x`>`s0`, the object is not in contact with the spring. The free end of the spring should be at `s0` or somewhere you should declare `s=s0`. If `pos.x <= s0` the object is in contact with the spring. The free end of the spring to now always be equal to the position of the object, and somewhere you should declare `s=pos`. The spring force can always be found using $s - s0$ as the displacement of the spring.

6.15 How do I draw a spring?

Use `draw_hspring` for a horizontal spring or `draw_vspring` for a vertical spring. See Chapter 8 for a full description of these.

6.16 Have something other than the black sky in my images?

Use the `sky_sphere` statement. It's best to Google something like "povray sky_sphere" for some examples. There are many examples out there of nice blue skies with scattered white clouds, etc. It takes more time to render, but is a nice effect. Adding a line like this (to Part 10) will make the black sky look light blue.

```
sky_sphere {pigment {LightBlue}}
```

6.17 Paint an object with an image file?

You can paint objects (spheres, boxes, etc.) with an image you may have (like a picture file from your camera). Here's how it's done. Put the image file (jpg, png, etc.) in the same folder as your Povray code (your .pov file). For this example, we'll assume your image is called "Nebula.jpg." This code will paint a box with the picture of the nebula on it.

```
box {<0,0,0>,<1,1,1> pigment { image_map {jpeg "Nebula.jpg" map_type 0} }}
```

Note that the moral here is to draw your object at the origin, unit dimensions. For a box, the `map_type` option is zero. If you need the box to be at a position other than the origin, you have to put it into an object statement, then translate it, like this,

```
object
{
  box {<0,0,0>,<1,1,1> pigment {image_map {jpeg "Nebula.jpg" map_type 0}}}
  translate <-5,-2,2>
}
```

which will move the box to $< -5, -2, 2 >$. Note that you still need to draw the box at the origin for the image map painting to work. Here's an example that paints a sphere,

```
sphere {<0,0,0>,1 pigment { image_map {jpeg "earth_land.jpg" map_type 1} } }
```

where you'll note that the sphere is also drawn at at the origin with unit size (i.e. the radius is 1), and the `map_type` has been changed to 1 for the spherical painting. You can move the sphere by again putting it into an `object` structure and translating it, after drawing it at $< 0, 0, 0 >$.

If you use an image type other than a jpg, then you have to change the file description from `jpeg` to `png`, etc.

Chapter 7

What to do if your movie won't render

Two things can be frustrating. The first is when you code won't render at all. The second is when it renders but doesn't work properly. Here are some hints.

7.1 My movie won't render

If your movie won't render, then you won't even get any graphics to come up. It is likely that you have a typo or error with some usage of a Povray line. Like your password to your favorite website, everything must be exactly right in order to work (computers are funny that way). Check the following, for the most common problems:

- Do all of your `#declare` statements end in a semi-colon?
- Do all of your open { and close } balance in a given line? That is, there must be as many { as } in a given line.
- Do all of your open (and close) balance in a given line? That is, there must be as many (as) in a given line.
- Does each `#if` statement have an `#end` statement that goes with it? You must have the `#end` statement no matter what (even if your `#if` block has only a single line in it).
- Look at the bare skeleton code in Section 5.2. The core components in it must also be in your own work. Look carefully that you didn't delete anything by accident during your editing.
- If you start a render and the process never seems to end, you probably altered the while-loop at the core of the skeleton code. Look at it in Section **??** .The `#while` statement must have a matching `#end` statement with it. Is this still there? Did you accidentally delete it? What about the `#declare xtime=xtime+clock_delta;` line?
- Do you have the # in front of any "if's" Remember it's `#if` not just `if`.

- Do you have the # in front of any "declare" lines? Remember it's `#declare` not just `declare`.
- Do you have `#declare` at all for variable assignments? Remember you have to say `#declare m=5;` not just `m=5`.
- Look at your formulas. Did you spell all variable names correctly?
- Look at your formulas. Is each variable used in a given formula defined *before* it is used in a given line?

A code scanner has been posted to help you along (if it can). You can access it here

`http://ocean.physics.calpoly.edu/povrayscan/`

7.2 My movie doesn't work right

Good luck with this one. There any any number of reasons why your movie doesn't work right, and likely it's something you did wrong (as tempting as it is to blame it on the "physics" or "the computer.") If implemented correctly, the physics equations will work beautifully. Here are a few things to consider:

- Check your physics. Are your equations right? Logic correct? What about the signs of your vector components?
- All final drawing is done in Part 10 *after* that final `#end` statement that goes with the `#while` line (neither of which you are supposed to touch).
- There should be no drawing statements between the `#while` and `#end` statements, with the exception of a `sphere` statement that is used to plot the trails behind an object.
- If vectors are too small or large, see the `set_vector...` functions in Section 8.3.
- Check that your accelerations are all set up in Part 5, *before* the physics equations.
- Check your code versus the bare skeleton code in Section 5.2. Other than Part 10, this code is your core structure. Be sure your work has all of the elements in the skeleton code at minimum. Sometimes lines get accidentally deleted or moved as you work.

Chapter 8

Drawing Physics

8.1 Introduction

Although Povray (and MegaPov) are very powerful drawing tools, we need them to do a bit more for our goal of *visualizing physics*. In particular, we'd like to easily draw vectors, energy bars, ropes, springs, and a few other salient items that will allow us to fully visualize the physics that will go into driving our animations. This all might sound very general, so let's take an example.

Suppose you wanted to draw a vector on a moving object. A vector is an arrow that has a tail, head, and length that is proportional to the quantity it is supposed to represent. Unfortunately, Povray does not have a native "arrow" drawing function. However, with Povray, you can make a vector from a thin cylinder, and use a cone as an arrow head. Since we'd like to focus on physics, definitions for vectors, springs, etc. have been created *for you*.

For this reason, as part of adapting Povray and MegaPov installations discussed in the previous chapter, a small file called "physics.inc" ("inc" for include) has been created. This file contains several instructions for allowing you to easily add physics visuals to your movies. You'll note that at the beginning of your Povray code is always reference to including a file called "physics.inc." This line does just what you think: includes some physics definitions from the file called `physics.inc` for you to use. What is available is described here.

8.2 Ideas on notation and usage

Drawing physics objects like ropes, vectors and spring will come from predefined functions in the file called `physics.inc`. You've heard the word "function" before. Think from your math classes what functions do, something like $f(x)$. When you write $f(x)$ you are implying that if you put in an x, and out will come another value or expression that depends on x. In this case, f is the name of the function and x is the single parameter of the function; the parenthesis punctuate it all. So if $f(x)=x^2$, then $f(5)=25$, just as $f(x)=a^2$. You put in a 5 and got out a 25. You put in a and got a^2. You can also have functions that involve more than one variable, like $f(x,y,z)$, but it means the same thing.

With computers, functions are represented in the same way; that is, a name, some parenthesis, then some parameters, separated by comma, but can given more meaningful names, like "draw_vector" or "draw_rope" (instead of "f" or "g"), since you have a keyboard, more storage, and a large screen, etc.

With computers, functions are often used not just to return some new value, but to actually *do* something. So while $f(5)$ returns 25 in your last math class, `draw_vector(`$< 1,1,1 >, < 2,2,2 >$`,Red,"x")` will draw a vector from $x = 1, y = 1, z = 1$ to $x = 2, y = 2, z = 2$ in a red color and label it "x."

Read Wikipedia on "Vector Notation." Vectors can be represented in many ways. The three most popular are 1) magnitude angle form, like "50 N at 30 degrees with the +x-axis." 2) "i,j,k" or "Engineering notation" like $5\hat{\imath} + 7\hat{\jmath}$ or 3) "Ordered Set Notation" as in $< 5, 7, 0 >$. Povray uses "ordered set" notation to represent vectors, as you'll see below.

Below are the functions available for you to use, as defined in `physics.inc`. These become available to your Povray code by virtue of the line in your code `#include "physics.inc''`. You don't need to understand how they work how they do what they do. Just use them, get your work done, visualize physics, and move on.

8.3 Useful functions in physics.inc

draw_vector : Drawing a vector on an object

Usage: `draw_vector(<xt,yt,zt>,<Ax,Ay,Az>,color,"label")`

Description: Draws a vector whose tail is at the x,y,z coordinate of xt,yt,zt. The vector drawn has components Ax,Ay,Az. The vector will have a color of color, and the textual label label will be drawn near the vector's head.

Examples:

```
draw_vector(<0,0,0>,<5,5,0>,Red,"v")
draw_vector(<0,0,0>,<F.x,0,0>,Blue,"Fx")
draw_vector(<sx,sy,0>,<ax,ay,0>,Red,"a")
```

In another setting you may have a vector variables, called `pos` that is the position vector of your object. Suppose also that your object has a velocity vector `vel` and an acceleration vector `a`. You can draw these vectors on the object in this way:

- Draw the v-vector: `draw_vector(pos,vel,Red,"v")`
- Draw the a-vector: `draw_vector(pos,a,Yellow,"a")`
- Draw just vx: `draw_vector(pos,<vel.x,0,0>,White,"vx")`
- Draw just vy: `draw_vector(pos,<0,vel.y,0>,SpicyPink,"vy")`

set_vector_scale : Zoom (or unzoom) all vectors drawn in a scene

Usage: set_vector_scale(n)

Description: Zooms all vector lengths by the factor n. At times, vectors drawn will be too long or too short and it would be nice (visually), if they could be rescaled. By making a call to this function, and passing it a real number, all vectors will be rescaled by the number. Passing a 1 for example will have no effect on the vector scaling. Passing a 0.5 will reduce all vector lengths by 1/2. Passing a 2 will double the length of all vectors.

Examples:
set_vector_scale(0.5)
set_vector_scale(2)

set_vector_thickness : Zoom (or unzoom) the thickness of all vectors

Usage: set_vector_thickness(n)

Description: Zooms the thickness of all vectors by the factor n. Sometimes vectors drawn will be thick or too thin, and it would be nice (visually), if their thickness could be controlled. By making a call to this function, and passing it a real number, the thickness of all vectors be rescaled by the number. Passing a 1 for example will have no effect on the vectors' thickness. Passing a 0.5 will reduce all vector thicknesses by 1/2. Passing a 2 will double the thickness of all vectors.

Examples:
set_vector_thickness(0.5)
set_vector_thickness(2)

set_vector_label_scale : Zoom (or unzoom) the size of the vector labels

Usage: set_vector_label_scale(n)

Description: Zooms the size of the vector labels by the factor n. Sometimes the textual labels drawn by vector heads will be too large or too small, and it would be nice (visually), if their size could be controlled. By making a call to this function, and passing it a real number, the size of all vector labels will be rescaled by the number. Passing a 1 for example will have no effect on the labels' sizes. Passing a 0.5 will reduce all vector labels by 1/2. Passing a 2 will double the labels' sizes.

Examples:
set_vector_label_scale(0.5)
set_vector_label_scale(2)

set_vector_label_color : Sets the color of the label drawn on a vector

Usage: set_vector_label_color(color)

Description: Sets the drawing color of the vector labels to color. Use this to set what color you'd like the textual label of a vector to be. Common colors are Red, Blue, Green, Yellow, etc. You can find more colors in Section 4.4.

Examples:
set_vector_label_color(Red)

set_vector_label_color(Yellow)

draw_text : Draw a text (i.e. words of your choice) on the screen

Usage: draw_text(the_location“the_text”,text_color,text_scale)

Description: Draws the text the_text at the vector location the_location in color text_color, with the size of the text being scaled by text_scale. Use this to render any text you wish at some location on the screen.

Examples:
draw_text(<5,1,3>,“hi there”,Blue,2)
Draws the text ”hi there” at location x=5, y=1, z=3 in blue, and double height.

draw_text(<sx,sy,0>,”the object”,Yellow,0.25)
Draws the text ”the object” at the location $(sx, sy, 0)$ in yellow, and 1/4 height.

draw_variable : Draws the numerical value of a variable on the screen

Usage: draw_variable(the_location,the_variable,“the_units”,text_color,text_scale)

Description: Takes the internal numerical value of variable the_variable and draws it on the screen at the vector location the_location in color text_color, with the size of the text being scaled by text_scale. The number will be labeled with the units the_units. Use this to render the value of any variable you wish at some location on the screen.

Examples:

- draw_variable(<5,1,3>,r,“meters”,Blue,2)
  Draws the value of the variable r at location x=5, y=1, z=3 in blue, and double height with units of meters.
- draw_variable(<-5,-5,0>,xtime,“seconds”,Yellow,0.25)
  Draws the value of the variable xtime at the location (-5,-5,0) in yellow, and 1/4 height.

dump_variable : Dumps the value of a variable into the big text window that Povray uses while rendering

Usage: dump_variable(“the_name”,the_variable)

Description: Takes the internal numerical value of variable the_variable and dumps it to the big text window you see while Povray/MegaPov are rendering. The value will be prefixed by the text contained in the variable the_name. This function is useful for debugging your code and checking on the value of a variable during the rendering process.

Examples:
dump_variable(“r=”,r)
Dumps the text “r=” followed by the numerical value of the variable r into the Povray text window.

draw_real_rope : Easy version for Drawing a real-looking rope between two points on the screen (courtesy of student T.W.W. Fall 2009).

Usage: draw_real_rope(<x1,y1,z1>,<x2,y2,z2>,thick)

Description: Draws a real-looking rope between the points <x1,y1,z1> and <x2,y2,z2> with a thickness of thick. A thickness of 1 is recommended; change from there as needed visually.

Examples:

- draw_real_rope($< 0, 0, 0 >, < 2, 2, 2 >$,1)
  Draws a rope of thickness=1 between the points (0,0,0) and (2,2,2).

- draw_real_rope(pos,pos+$< 2, 0, 0 >, 0.5$)
  Draws a rope of thickness=0.5 between the points at `pos` and `pos` with 2 added to the x-component.

real_rope : More user controlled function for drawing a real-looking rope between two points on the screen

Usage: real_rope(<x1,y1,z1>,<x2,y2,z2>,thick,color1,color2,amb,detail)

Description: Draws a real-looking rope between the points <x1,y1,z1> and <x2,y2,z2> with a thickness of thick. A thickness of 1 is recommended; change from there as needed visually. color1 and color2 are the two intertwined colors of the rope. Parameters amb and detail are numbers you can pass to modify the overall look of the rope. It is recommended you start with amb=0.3 and detail=5 and adjust from there.

Examples: `real_rope(<0,0,0>,<2,2,2>,1,Red,White,0.3,5)`
Draws a red and white rope of thickness=1 between the points (0,0,0) and (2,2,2). Visual affects of 0.3 and 5 are used.

draw_vspring : Draws a vertical spring

Usage: draw_vspring(y1,y2,x1,rad,thick)

Description: Draw a vertical spring between the y-coordinates y1 and y2. It will be positioned horizontally at location x1. The spring will have a coil radius of rad and the "wire" used to make the coil will have a thickness of thick. A thickness of 0.2 and a radius of 1 are recommended to start; adjust from there.

Examples:

- draw_vspring(10,15,0,1,0.2)
  Draws a metallic vertical spring between the y-coordinates 10 and 15, at an x-coordinate of 0, which a coil thickness of 1 and a wire thickness of 0.2
- draw_vspring(by,0,0,1,0.2)
  Draws a metallic vertical spring between the y-coordinates by and 0, at an x-coordinate of 0, which a coil thickness of 1 and a wire thickness of 0.2

draw_hspring : Draws a horizontal spring

Usage: draw_hspring(x1,x2,y1,rad,thick)

Description: Draw a horizontal spring between the y-coordinates x1 and x2. It will be positioned vertically at location y1. The spring will have a coil radius of rad and the "wire" used to make the coil will have a thickness of thick. A thickness of 0.2 and a radius of 1 are recommended to start; adjust from there.

Examples:

- draw_hspring(0,5,0,1,0.2)
  Draws a metallic vertical spring between the x-coordinates 0 and 5, at a y-coordinate of 0, which a coil thickness of 1 and a wire thickness of 0.2
- draw_hspring(bx,0,3,1,0.2)
  Draws a metallic vertical spring between the x-coordinates bx and 0, at a y-coordinate of 3, which a coil thickness of 1 and a wire thickness of 0.2

draw_bar : Draws a bar (as in an energy or momentum bar)

Usage: draw_bar($< xb, yb, zb >$,height,the_color,"the_label")

Description: Draw a vertical bar with a base at the position $< xb, yb, zb >$. The bar will have a height of height, as drawn up from the base coordinate. It will have a color of the_color and just below the base will have the textual label of the_label.

Examples: draw_bar($< -3, -5, -2 >$,KE,Red,"Kinetic Energy")

Draws a bar whose base is at the coordinate (vector position) $< -3, -5, -2 >$ whose height will be the number contained in the variable KE. The bar will be red and will have the label "Kinetic Energy" drawn just below the base point.

set_bar_scale : Scales the overall length of all bars (as in an energy or momentum bar)

Usage: set_bar_scale(n)

Description: Sometimes energy or momentum bars can be too long, extending off of the screen, or too small, not showing much action. This function allows you to zoom the length of all bars by a factor n. If $n > 1$ bars will be magnified. If $n < 1$, bars will be reduced in size.

Examples:

- set_bar_scale(2)
  All bars will be drawn and zoomed longer by a factor of two.
- set_bar_scale(0.1)
  All bars will be drawn at one-tenth of their original size.

Note: Place such a line before any draw_bar usage and only use once in a given program.

set_bar_zoom : Zooms the overall width

Usage: set_bar_zoom(n)

Description: Magnifies an entire bar. If $n > 1$ the bar will appear fatter and closer to the camera. If $n < 1$ it will appear skinnier and farther away. Used only for aesthetic reasons, if you think your bars do not look good in your movie.

Examples:

- set_bar_zoom(2)
  All bars will be drawn twice and thick.
- set_bar_zoom(0.1)
  All bars will be drawn at one-tenth as thick..

Note: Place such a line before any draw_bar usage and only use once in a given program.

set_bar_label_zoom : Zooms just the textual label of a bar.

Usage: set_bar_label_zoom(n)

Description: Magnifies just the label that appears near a bar. If $n > 1$ the text will appear larger. If $n < 1$ the text will appear smaller.

Examples:

- set_bar_label_zoom(2)
  All bar text labels will be twice as large.
- set_bar_label_zoom(0.1)
  All bars text labels will be one-tenth as large.

Note: Place such a line before any draw_bar usage.

plot_curve : Plots a curve of some function supplied by the user.

Usage: (two steps)

- Define the function to plot using a `#macro` statement. The function name must be called `curve` as shown:

  ```
  #macro curve(xp) xp*xp #end
  ```

- Make a call to `plot_curve(x0,x1,dx,color,scale)`

Description: Plots a curve assuming the x-axis contains the independent variable. Plots `curve(x)` for `x0` $\leq x \leq$ `x1` with a step size of `dx`. The curve will be in a color `color` and its thickness can be scaled by the multiplicative factor `scale`

Examples:

- These two lines will plot the function $curve(x) = x$ from $-5 \leq x \leq 5$ in Red, with a $\Delta x = 0.01$ at the default scaling.

  ```
  #macro curve(xp) xp #end
  plot_curve(-5,5,.01,Red,1)
  ```

- These two lines will plot the function $curve(x) = 1 + tanh(x)$ from $-10 \leq x \leq 10$ in Green, with a $\Delta x = 0.1$, with 3 times the thickness.

  ```
  #macro curve(xp) 1+tanh(xp) #end
  plot_curve(-10,10,.1,Green,3)
  ```

draw_car : Draws a simple car.

Usage: `draw_car(position,scale,angle)`

Description: Draws a car in the xy-plane at vector position given by `position`. The car will be scaled in size by the number given by `scale` and rotated about the z-axis by the angle `angle` (in radians).

Examples:

- draw_car($<0,0,0>$,1,0)
  Draws a car at at the origin ($x = 0, y = 0, z = 0$) with a scale of 1 and with no rotation about the z axis.
- draw_car($<0,1,0>$,3,pi/2)
  Draws a car at ($x = 0, y = 1, z = 0$) with a scale of 3 and with a rotation of 45° about the z axis.
- draw_car(pos,2,atan2(vel.y,vel.x))
  Draws a car at the position contained in the vector variable `pos` with a scale of 2. The car will be rotated as per the instantaneous angle of the velocity vector contained in `vel` using the `atan2` statement, which is the Povray/MegaPov version of $\tan^{-1}$.

draw_rocket : Draws a simple rocket.

Usage: `draw_rocket(position,scale,angle)`

Description: Identical in usage to `draw_car` (above), except that it draws a rocket instead.

draw_box : Draws a box.

Usage: `draw_box(position,length,color)`

Description: Draws a box at the vector position given by `position`. The length of each side of the box is given by `length` and the box will be drawn with a color given by `color` (such as a color given in Section 4.4).

Examples:

- draw_box($< 0, 0, 0 >$,1,Red)
  Draws a red box at the origin ($x = 0, y = 0, z = 0$) with a side length of 1.
- draw_box(pos,2,Blue))
  Draws a blue box at the position contained in the vector variable `pos` with a side length of 2.

Chapter 9

One dimensional motion

9.1 Introduction and Goals

Your goal for these project is to demonstrate that you understand how an object moves in one dimension.

- You'll change an object's v-vector by applying an arbitrary acceleration to an object either parallel or antiparallel to the object's v-vector.
- Demonstrate that you understand the interplay between x, v, and a, in how an object moves while constrained to a single axis of motion, in this case either the x or y axis.
- Show you understand what effect a has on v, and ultimately x, particularly when a and v have the same, then opposite signs.
- Demonstrate understanding of one-dimensional vectors.
- Demonstrate that seeing velocity and acceleration vectors on an object gives clues to the object's subsequent motion.

9.2 The Physics

The v-vector of an object will be changed by: Applying an acceleration either parallel or anti-parallel to v. The goal of this week is to understand how acceleration, a, can be used to primarily change an object's v-vector, and secondarily force changes in an object's position (x), all in just one-dimension. An "object" means anything that can move, like a ball, car, truck, or person. One-dimensional means the object will only move along a straight line, typically along the x-axis if it's moving left or right or y-axis if it's moving up or down. There are two equations you need for this, $x = x_0 + v_0\Delta t + \frac{1}{2}a\Delta t^2$, and the second is $v = v_0 + a\Delta t$. If the object is at x_0 with speed v_0, then x and v will be the object's new position and speed at some time interval Δt later. These two equations allow you to compute the new position and speed of an object (x and v), based on its old position and speed (x_0 and v_0), given some

acceleration a that is acting on the object, over a time interval Δt. Δt is sometimes called the "time step" and is a small interval of time that separates when the object has x_0 and v_0, and when it will have x and v. We said that a drives changes in v and x. Notice in these equations if $a = 0$, then $v = v_0$, meaning that v doesn't change between time steps; v is constant if $a = 0$. In order for v to change, a must be nonzero. In other words, an object's speed can change only if it has an acceleration. For the x-axis (left-right motion), we have that $x = x_0 + v_{0x}\Delta t + \frac{1}{2}a_x\Delta t^2$ and $v_x = v_{0x} + a_x\Delta t$. For the y-axis (up-down motion), we have that $y = y_0 + v_{0y}\Delta t + \frac{1}{2}a_y\Delta t^2$ and $v_y = v_{0y} + a_y\Delta t$. These equations are the same, just the notation is different, being very specific as to the axis which axis it pertains. Suppose you have a sphere at $x = 5$ m with speed $v = 1$ m/s and an acceleration of $a = 0.5$ m/s^2. When the next frame comes up, say $\Delta t = 0.1$ s later, where will the sphere be and what will its speed be? Use the equations to get that $x = 5\text{m} + (1\text{m/s})(0.1\text{s}) + (0.5)(0.5\text{m/s}^2)(0.1\text{s})^2$ or $x = 5.1025$ m and $v = 1\text{m/s} + (0.5\text{m/s}^2)(0.1\text{s})$ or $v = 1.05$ m/s. You can iteratively use this new x and v as a new x_0 and v_0 (i.e. $x \to x_0$ and $v \to v_0$) for computing still another x and v another Δt in the future. Can you find x and v after another Δt has gone by (ans: $x = 5.208$ m and $v = 1.06$ m)? Be very aware of signs. Think of a cartesian coordinate system with $+x$ to the right, $-x$ to the left, $+y$ up and $-y$ down (assume Δt is always positive). Positive values of position mean the object is to the right (x) (or up, y) relative to the origin. Negative means the object is left (x) (or down, y) relative to the origin. Positive values of speed mean the object is moving toward the right (v_x) or up (v_y), negative means to the left (v_x) or down (v_y). The sign of a alone doesn't immediately help to characterize the object's motion. If, however, a and v have the same sign, $v = v_0 + a\Delta t$ will predict an increase in v (that is if v and a have the same sign, an object will speed up). Likewise, an object will slow down if v and a have opposite signs. A case where opposite signs of v and a persist means v will get smaller and smaller, until eventually $v = 0$ at which case the object will stop. If a still persists, then v will begin to increase in the same direction as a; now the object is speeding up, but in the opposite direction to its original motion. All told the object slowed down, stopped, then started speeding up in the opposite direction. All combinations of signs between v and a are possible. $v > 0$ and $a < 0$ is a slow-down and potential turn-around case, as is $v < 0$ and $a > 0$. $v > 0$ and $a > 0$ or $v < 0$ and $a < 0$ are speed up cases, but in opposite directions. Lastly, you should be able to draw arrows on an object, representing its v and a and that instant. An arrow should point in the direction of the parameter it represents, and its length should be proportional to its amount (or strength). For example, if on an object the arrow for v and the arrow for a were opposite, you'd know the object was slowing down. An object going 2 m/s would have a v arrow half as long as one going 4 m/s. **Book reading: 1.3-1.6, 2.4, 2.5**.

9.3 Projects

Learning outcomes: To understand how an object's v-vector will be affected by an acceleration either parallel and anti-parallel direction v.

Movie 1D.a: This movie can be made almost directly from the skeleton code which you can download here `http://goo.gl/zTtuQ` . Create a movie that shows a sphere moving from

the right edge of the screen to the left edge of the screen. The motion of the object should be for $v_0 \neq 0$, and $a = 0$. Using the `draw_vector` macro, render the velocity vector on the sphere at all times. Be sure the object leaves a "trail" behind it, indicating past positions of the object. See Figure 5.4, Part 7 for more on trails. See Chapter 5.4 for tips on drawing vectors. Why do this? To test your understanding of how an object with a constant velocity moves.

Movie 1D.b: From the code used for the last movie, repeat the right-to-left motion for the object speeding up ($v_0 \neq 0$ and $a \neq 0$). Using the `draw_vector` macro, render both the velocity and acceleration vector on the sphere at all times. Be sure the object leaves a "trail" behind it, indicating past positions of the object. Why do this? To test your understanding of how an object moves when it has an acceleration acting on it in the parallel to its velocity.

Movie 1D.c: Adapt the code again, but for an object that starts at the right and proceeds toward the left. It should have an acceleration on it that is opposite to the velocity; in other words v and a should have opposite signs. It should slow down, stop, then turn back around and go into the direction from which it came originally. Be sure your movie clearly shows the object obtaining good speed in the opposite direction. Using the `draw_vector` macro, render both the velocity and acceleration vector on the sphere at all times. Be sure the object leaves a "trail" behind it, indicating past positions of the object. Why do this? To test your understanding of how an object moves when it has an acceleration that is opposite to its velocity; to see what makes an object stop; and to see what makes it turn around.

Movie 1D.d: Show an object being launched upward, along $+y$ at some speed v_{0y}, under the downward ($-y$) acceleration of gravity, g. The object should move up, slow, stop, then turn around and fall back down again. Using the `draw_vector` macro, render both the velocity and acceleration vector on the sphere at all times. Be sure the object leaves a "trail" behind it, indicating past positions of the object. Why do this? To test your understanding that 1D vertical motion is really the same as 1D horizontal motion, only all of the "action" is along the y-axis now.

Movie 1D.e: Download the code called `ball_wall.pov` here `http://goo.gl/zTtuQ` . If you render it straight out, it will show a ball being sent directly into a brick wall with $v_{0x} = 10$ (or $\vec{v} = 10\hat{x}$). Without altering this initial $\vec{v}$, change only the x-component of $\vec{a}$ in Part 7 of the skeleton code (Figure 5.4) so that the ball is just able to turn around before slamming into the wall. No calculations are necessary here. Do this by trial and error with values for a_x in Part 7.

Movie 1D.f: Download the code called `balltower.pov` here `http://goo.gl/zTtuQ` . If you render it straight out, it will show on the top of a very tall wooden tower, being sent directly toward the edge with $v_{0x} = 5$ (or $\vec{v} = 10\hat{x}$). Without altering this initial $\vec{v}$, change only the x-component of $\vec{a}$ in Part 7 of the skeleton code (Figure 5.4) so that the ball is

just able to turn around before falling over the edge. Do this by trial and error with values for a_x in Part 7.

Note: The object in all movies must have $\vec{v}$ and $\vec{a}$ vectors clearly labeled on it, using the `draw_vector` function discussed in Section 8.3.

9.4 Wrap-up Questions

1. Draw a simple car. Indicate that it is moving at constant speed to the left, by drawing appropriate v and a vectors on it.
2. Draw a simple car. Indicate that it is speeding up to the left, by drawing appropriate v and a vectors on it.
3. Draw a simple car. Indicate that it is moving toward the right but slowing down, by drawing appropriate v and a vectors on it.
4. Draw 5 circles evenly spaced apart. Each circle is different instant of time for a circle that is speeding up as it moves toward the right. The leftmost one is moving the slowest, and the rightmost one is moving the fastest. Draw v and a vectors on each sphere that reflect the observed motion.
5. Draw the "trail" or dots an object would leave that was thrown straight up, to the point where it stops at the top of its flight.
6. Draw the "trail" or dots an object would leave that is accelerating toward the right.
7. Draw the "trail" or dots an object would leave that is moving at constant speed to the right.
8. Describe how the v-vector behaves for an object that is slowing down, then stops briefly, then begins accelerating back in the direction from which it came.
9. A sphere with a v-vector sticking out of it. Discuss what you can infer about its future motion.
10. A sphere has a v-vector sticking out of it. A second sphere has its own v-vector sticking out of it, pointing in the same direction as the first, but twice as long. Discuss what you can infer about the future motion of both spheres.
11. A sphere has a v-vector sticking out it and pointing to the right. It also has an a-vector sticking out of it, and pointing up and to the left. Where will the sphere be a very short time in the future? Sketch its trajectory from this initial condition for many seconds into the future.
12. What acceleration value did you need in Movie 1D.6, that allowed the ball to turn around?

13. In Movie 1D.e or 1D.f, what would happen if you accidentally put your guess for a_x into the 2nd position of the `a=`$< 0, 0, 0 >$ definition?

Chapter 10

Two dimensional motion

10.1 Introduction and Goals

The goal of this project is to demonstrate that you understand how an object moves in two dimensions (projectile motion).

- In this lesson, you'll change an object's v-vector by applying an acceleration to an object in such a way that the a- and v-vectors do not point along the same axis (the way they did last week).
- Demonstrate that you understand the interplay between x and v_x, y and v_y as they simultaneously evolve in time for an object.
- Show you understand that acceleration directed along an axis has no effect on the motion along an orthogonal axis. In terms of symbols, you must show that a_x does not affect y or v_y and a_y does not affect x or v_x.
- Demonstrate understanding of two-dimensional vectors.
- Demonstrate understanding of the acceleration vector for a projectile in flight.
- Demonstrate understanding of the velocity vector of a projectile in flight, as well v_x and v_y and all instants during flight.
- Demonstrate understanding of how a drag force will affect the motion of a projectile.

10.2 The Physics

The v-vector of an object will be changed by: Applying a downward, vertical acceleration to the object. The goal of this week is to understand how objects move in fully two dimensions. Last week you concentrated on motion strictly along the x or y axis. Two dimensional motion is where an object undergoes motion along the x and y axes *at the same time*. The position of an object in two-dimensional space can be plotted by its (x, y) coordinate. These

coordinates are found by the equations $x = x_0 + v_{0x}\Delta t + \frac{1}{2}a_x\Delta t^2$ and $y = y_0 + v_{0y}\Delta t + \frac{1}{2}a_y\Delta t^2$. Note that also evolving as an object moves are its speeds along two axes as well, $v_x = v_{0x} + a_x\Delta t$ and $v_y = v_{0y} + a_y\Delta t$. Remember that the x and y coordinates are perpendicular to each other, that is the x and y axes are orthogonal. This is a special relationship in math and physics, and means that processes along one axis do not affect processes along the other axis. Therefore, whatever happens along the x axis does not affect what happens along the y axis, and vice-versa. This is a key concept to understand this week. Two-dimensional motion is sometimes called "projectile motion" which encompasses objects flying through space under the influence of gravity. Baseballs, cannon balls, basketballs moving through space are all examples of projectile motion. The movies this week will show projectiles in flight, restricted to motion where $a_x = 0$ and $a_y = -g = -9.8$ m/s^2. You can immediately find forms of the $x =$ and $y =$ equations above, given these restrictions. An any given time, your object will have four quantities describing its motion: x, y, v_x, and v_y. Since position and speed now each have two components (or parts), position and speed will be "vectors," called $\vec{r}$ and $\vec{v}$ respectively. $\vec{r}$ will consist of two components, the x and y coordinates of the object. Similarly, $\vec{v}$ will consist of the components v_x and v_y. As you will now see, the two components of both $\vec{r}$ and $\vec{v}$ gives them both a magnitude (strength, length, etc.) and direction, which you must know how to handle. There are two ways of dealing with vectors, and you should be proficient with both. The first way is in "magnitude-angle form," where you report the magnitude of the vector and the angle at which it is pointing. For the position, the magnitude (or total distance from the origin) is $r = \sqrt{x^2 + y^2}$. The angle this vector will make relative to the $+x$-axis is given by θ where $\theta = \tan^{-1}\frac{|y|}{|x|}$. The absolute value signs are important to remove any negative values that might pop up and ensure the angle is with respect to the $+x$-axis. The velocity vector is tracked similarly, namely $v = \sqrt{v_x^2 + v_y^2}$ with $\alpha = \tan^{-1}\frac{|v_y|}{|v_x|}$, where α is the angle the velocity vector makes with respect to the $+x$-axis, and is essentially the direction the object is moving in at that instant of time. Be sure you understand why a vector has a magnitude and an angle, and be sure you can always compute both from a given vector's components. The other way of handling a vector is in "component form." In this form, you list each component directly, next to a unit vector specifying what axis the component goes to. So if an object is 5 meters along the x-axis and 2 m along the y-axis then $\vec{r} = 5\hat{i} + 2\hat{j}$, where $\hat{i}$ and $\hat{j}$ are unit vectors meaning x-axis and y-axis, respectively. The other type of two-dimensional motion that is important is circular motion, which describes how an object moves in a circle. In this type of motion, the object is always has an acceleration that points toward the center of the circle around which it traveling. If you choose a circle of radius r and want the object to move around the circle with a speed v, then the strength of the acceleration, called the "centripetal acceleration" must be $a = \frac{v^2}{r}$, and it must always point toward the center of the circle. **Book reading: 4.2-4.3,4.5**.

10.3 Projects

Learning outcomes: To understand how an object's v-vector will be affected by an acceleration applied on an axis different than that of v's.

twod.a: Start with the code `2D.1_movie.pov` which you can find online. This movie should

show a sphere starting near the right edge of the screen. After being launched with some speed v_0 at some angle θ relative to the horizontal, it should fly across the screen to the left, much like a basketball, baseball or cannon ball would. Your Povray code should clearly show how the object is given a particular initial velocity (v_0 and θ). That is your initial v_0 should resemble $v0 =< v_0 \cos\theta, v_0 \sin\theta, 0 >$, after θ and the magnitude v_0 have been declared. All told, these are the "launch conditions" for your particle. The projectile should also be given an initial position $< x_0, y_0, 0 >$. The movie should end right about the time your object hits the ground again, and it would be nice if your launch and landing positions were not at the same vertical level. Be sure trails are left, showing your object's overall trajectory. Attach the following 4 vectors on your object: a, v, v_x, and v_y. All told, your object should have four vectors sticking out of it as it flies. Put a "ground" in your movie using Povray's `plane` statement as in `plane {<0,1,0>,0 pigment {Green}}`. Here the `<0,1,0>` is the vector normal to the plane and the lone 0 is how far up or down along the normal to move the plane. Since there's a "1" in the y-position of the normal, the plane is normal to the y-axis. You can use the Movie1.1.pov code found online to start this project.

twod.b: Start with code online called `basketball.pov`. If you render it you'll see a basketball hoop and a ball. In Part 2 of the code, set v_0 and θ so that the ball goes into the hoop with "nothing but net." Feel free to adjust the camera position if the ball goes off of the screen, or to make the view more appealing.

twodd.c: Start with your completed Movie `twodim.a` code. Suppose that when your movie is 60% done, a strong horizontal wind develops that gives the ball an a_x to the left of 15 m/s^2 and up of 3 m/s^2. You can put this into the movie by tapping into the `xtime` variable, which is the simulated time in your movie. In Part 5 of your skeleton code you can put

```
#if (xtime > 0.6)
#declare a=..the acceleration with the wind included...;
#end
```

Note that the variable `xtime` is the simulated time of the movie. It always runs from 0 to 1, so 0.6 is the 60% mark. The `#if` statement, covered in Section 5.5 is used to trigger the wind. Don't forget about g throughout!

twod.d: Imagine a ball sliding across a table with some horizontal speed (parallel to the table) $< v_x, 0, 0 >$. As it slides, $a =< 0, 0, 0 >$. When it comes to the edge of the table, it gets launched off of the edge with the same $< v_x, 0, 0 >$, but now $a =< 0, -9.8, 0 >$. Using the code `movie2D.3.pov` at `http://goo.gl/zTtuQ` to get you started, you'll see that the ball starts on the table in the region where $x > 0$. It moves left toward $x = 0$, where the edge of the table exists. So if $x > 0$, $a =< 0, 0, 0 >$ and if $x \leq 0$ then $a =< 0, -9.8, 0 >$. Using the `#if` statement (see Section 5.5) in Part 5 of your code, create a movie that shows the ball across the table, then leaving the edge of the table and flying through the air in projectile motion. Adjust the initial speed of your ball so that it flies through the hoop shown. Be sure a_y, v_x, v_y, and v vectors are rendered on the ball throughout.

twod.e: Here is another form of two-dimensional motion. Choose a value for the radius of a circle, R. Place a ball at the starting position $< R, 0, 0 >$ and choose an initial velocity for it, $\vec{v}$. Initially, place the entire magnitude of $\vec{v}$ along the y-axis, as in $\vec{v}_0 =< 0, |v|, 0 >$. This essentially starts an object at $< R, 0, 0 >$ with an upward velocity. Now place an acceleration on the object that has a magnitude of $a = v^2/R$, so that a is always *perpendicular to* v. How? At a particular instant in your movie (at the start of Part 5 of the skeleton code), find the angle the v-vector is making with respect to the x-axis. You can do this using the arctangent idea described in Section 3.3.3, Section 3.3.4, and Chapter 6. Suppose this angle is called A. An angle that is always perpendicular to A is would be found by adding $90°$ or $\pi/2$ to A as in $A + \pi/2$, or `A+pi/2` in Povray. Now, you can find a_x and a_y using $a_x = a\cos(A + pi/2)$ and $a_y = a\sin(A + pi/2)$. Your ball should being moving in a perfect circle, and this is how circular motion works: *when* $\vec{a}$ *is always perpendicular to* $\vec{v}$. Be sure $\vec{v}$ and $\vec{a}$ are drawn on the ball at all times.

twod.f: If you added a plane to Movie 2D.1, did you wonder why the ball didn't bounce off of the ground? This is an interesting aspect of making computer movies: limits of motion. To the computer, the ball and the ground are nothing but pixels, that it is happy to draw. There is no reason to expect that the computer will automatically handle something like a bounce. You have to program in such interactions yourself. One way is as follows.

Would it seem reasonable that when an object collides with the ground, reversing the sign of v_y should cause it to bounce. In other words, an object with a negative v_y (downward moving ball) that encounters the ground at $y = 0$, should immediately be given a $+v_y$, to send it back in the direction from which it came. This should cause it to bounce. This argument is an abrupt change in the ball's velocity, and so should go into Part 8 of the skeleton code (see Figure 5.4). Reversing v_y can be accomplished with a redefinition of the velocity variables as in

```
#declare vel=<vel.x,-vel.y,0>;
```

to reverse v_y. You can add coefficients of restitution (see Wikipedia) to simulate imperfect bounces by adding a decimal in front of the component being reversed, like this:

```
#declare vel=<vel.x,-0.7*vel.y,0>;
```

Lastly, the `#if` statement (in Part of the skeleton code) that would check for a collision must see if the object is moving down ($v_y < 0$) and if the object is in contact with, or even embedded into the ground `pos.y <= 0`, as in)

```
#if (vel.y < 0 & pos.y <= 0)
...reverse vy
#end
```

See Section 5.5.2 for more on handling such `#if` statements and nuances about using a strict equality in such conditions.

twod.g: Start with the basic skeleton code. Set the camera at $< -1, 5, -25 >$ and the light source at $< 0, 10, -50 >$. Start an object at $< 0, 0, 0 >$. Next launch it toward the camera, in projectile motion, so that it brushes right by the camera on the right, almost hitting the camera itself. The v-vector should appear to "poke you in the eye" as it passes. Pedagogical goal: motion that includes the z-axis.

twod.h: Find code called `balltower.pov` at `http://goo.gl/zTtuQ` . You might have used it in a project from the previous chapter. If you render this code, it'll show a ball going over the edge of a tower. The edge of the tower is at $x = 0$. Assign the proper free fall acceleration so that the ball actually falls when it's over the edge of the tower. Adjust your camera/view so that we see a nice, long, dramatic fall "into nowhere."

twod.i: Take your code from `twod.a`, but instead of drawing a red sphere, draw a rocket using the statement `draw_rocket(pos,scale,theta)` where `pos` is the position to draw the rocket, `scale` is a zoom factor (try 1), and `theta` is the orientation angle, in radians at which to draw the rocket. Your movie should show the rocket flying in projectile motion, with its nose always pointing in the direction that it is moving. WHY? THIS MOVIE IS TO TEST YOUR UNDERSTANDING OF HOW TO FIND THE ANGLE A VECTOR MAKES WITH THE HORIZONTAL AT ANY GIVEN MOMENT.

10.4 Wrap-up Questions

1. A ball is launched in the vacuum of outer space with no planets or stars nearby at all, with some v_0 at some θ_0 relative to some axis you call the horizontal. List the acceleration or acceleration(s) acting on the ball as it flies and describe its trajectory.

2. A ball is launched through the air, back here on earth with some v_0 at some θ_0 relative to the horizontal. List the acceleration or acceleration(s) acting on the ball as it flies.

3. Draw the parabolic path a ball would take in a vacuum given that it was launched with some v_0 at some θ_0 relative to the horizontal. On the path (and all on the same figure), draw the ball and its a and v vectors when the ball is at:

   (a) It's peak (highest) point.

   (b) At a position midway between its launch point and the peak point.

   (c) At a position midway between the peak point and it's landing point.

4. A ball is launched straight up with $v = 10$ m/s from $y = 0$ (the ground). Compute how long it takes the ball to reach the ground again. Another ball is launched with $v_x = 10$ m/s and $v_y = 10$ m/s from $y = 0$. How long does it take this ball to reach the ground? Compare/contrasts/discuss the two answers.

5. What are your thoughts on how air resistance works?

6. Watch your movie from last week about the object that is launched straight upward. Pay attention to the v_y vector. Now watch your movie 2.1 again, paying attention to the v_y vector. Compare and contrast the two v_y vectors. Discuss.

7. Describe the behavior of v_y as the projectile flies through the air.

8. Describe the behavior of v_x as the projectile flies through the air.

9. Discuss: An object's v-vector will continually change its orientation until it points along the same direction as the a-vector.

10. Discuss the last question as it might pertain to *circular motion* where $a \perp v$.

11. In project `twodim.b`, the ball started 25 horizontal meters away from a hoop 8 meters high. What v_0 and θ did you find in your animation made the ball go through the hoop? Rework with problem with pencil and paper (like a textbook problem) and see if you get the same result.

Chapter 11

Forces and Newton's Laws (Part I)

11.1 Introduction and Goals

The goal of this project is to demonstrate that you understand how the superposition of forces on an object directs its ultimate motion.

- In this lesson, we'll change an object's v-vector by applying one or more *forces* to an object. That is, we're going to use *forces* to change v-vectors.
- Demonstrate that you understand what $\Sigma\vec{F}$ means.
- Demonstrate that you understand the meaning of F_{net} or "net force" on an object.
- Demonstrate that in order to use $\Sigma\vec{F}$ you must actually use ΣF_x and ΣF_y.
- Demonstrate that you know how to find an object's $\vec{a}$ from its $\Sigma\vec{F}$, and that $a_x \leftrightarrow F_x$ and $a_y \leftrightarrow F_y$.
- Demonstrate that you understand how to sum two-dimensional forces applied to an object.
- Demonstrate that you know how the net force on an object can be used to find the object's acceleration.
- Demonstrate that you know how forces can be used to change an object's v-vector.

11.2 The Physics

The v-vector of an object will be changed by: Applying a force or a net-force on an object. The goal of this week to use Newton's Laws to see that accelerations actually come from forces applied to an object. In the past two weeks, a was simple a "given" quantity. It simply existed in the equations of motion and you were allowed to give it any value. This week we will see that accelerations come from *forces*. Forces are pushes or pulls on objects that you witness everyday (push a door to open it, pull on your book to lift it, push a cell phone button

to click it). With the exception of gravity, forces are always "contact forces" meaning a force must actually touch an object to exert its influence on it. Forces also require an agent, meaning that you should always be able to identify what (the agent) is producing the force. Forces are also vectors, meaning their strength (push or pull) can be in any direction. You probably know that Newton's Law says $F = ma$, but this is a horrible equation to ever try and use in a physics course. $a = \frac{F}{m}$ is better, but still isn't quite right. It is more correct to say that $a = \frac{\Sigma F}{m}$, which still isn't fully correct. The best version is $\vec{a} = \frac{\Sigma \vec{F}}{m}$, stressing the vector property of forces. Be sure you fully understand what this last version means and how to use it. $\vec{a}$ is the acceleration, m is the object's mass and $\Sigma \vec{F}$ is the sum of all forces acting on the object. Mass is the amount of "stuff" an object is made from and never changes unless portions of the object are somehow broken off. We will only be concerned with five forces: weight, tension, normal, friction, and drag. Weight is $w = mg$, where m is an object's mass and g is the earth's acceleration of gravity of 9.8 m/s^2. Do not confuse mass and weight; they are not the same thing and be sure you know the difference between them. Mass is also known as object's inertia, or resistance to want to change its current state of motion. It would hurt if you placed a bowling ball on the floor in front of you and kicked it as hard as you can. Would it also hurt to kick the bowling ball in the middle of outer space where $g = 0$? Tension is the tug an object feels when pulled by a rope attached to it. Normal is the force an object feels when it is sitting on a surface, is always perpendicular to the surface, and is not always equal to mg. Friction is a force that always opposes all motion; it always acts in a direction exactly opposite to that in which an object is moving (or trying to move), and typically comes when the object rubs or drags against another object as it moves. It is defined as $\vec{f} = \mu \vec{N}$ where μ is the coefficient of friction (p. 150 in your book), and $\vec{N}$ is the normal force acting on the object. Drag is like friction in that it always acts in a direction opposite to that in which an object is moving, but comes from air or water, through which an object might be moving. Drag, $D = Cv^2$, where C depends on the shape and size of the object, and v is the object's speed. Air resistance is a type of drag force. The crux of this entire week is the part, $\Sigma \vec{F}$, because it requires three hard steps. The first, which most students have great difficult with, is to identify all forces acting on an object. The second is to correctly draw these forces, each pointing in the proper direction, as they act on the object (even more difficult for most students). The third is to realize that $\Sigma \vec{F}$ which is only useable when you break it up into component form, or ΣF_x and ΣF_y. Your working equations for this week are then $a_x = \Sigma F_x/m$ and $a_y = \Sigma F_y/m$. The connection points with weeks 1 and 2 then are that these accelerations, which come from forces, are the same a's that go into the equations of motion for x and y. Thus, $x = x_0 + v_0 \Delta t + \Sigma F_x \Delta t^2/(2m)$ and $v_x = v_{0x} + \Sigma F_x \Delta t/m$. Be sure these make sense to you and do not causally read over the ΣF_x and ΣF_y. Know what they mean: Using Newton's law really means that all forces acting on an object need to be broken up into their x and y components, properly signed (+ or $-$), then added together along a given axis. **Book reading: 5.1, 5.2, 5.3, 5.7, 6.2, 6.3, 6.6**.

11.3 Projects

(Note: NL in the movie names stands for "Newton's Laws." The "I" stand for "one" as in "Part 1." Next week will be part 2.)

NLI.a: Have a box start near the left edge of your screen. Give it an initial v_x that is > 0 (that sends it rightward). Next, apply two x-forces to the box so that the forces cancel each other. The movie should show the subsequent motion. Be sure to draw a trail behind the box, both force vectors, the net force vector, and the v-vector. Part 5 of your code should show explicit definitions of forces, `F1` and `F2`. Part 5 should end with a line like `#declare a=(F1+F2)/m`.

NLI.b: Same as MovieNLI.a, but make the leftward force larger than the rightward force.

NLI.c: Same as MovieNLI.b, but make the rightward force larger than the leftward force.

NLI.d: Have a box start near the top edge of your screen. Give it an initial v_y that is < 0 (that sends it downward). Next, apply two y-forces to the box so that the forces cancel each other. The movie should show the subsequent motion. Be sure to draw a trail behind the box, both force vectors, the net force vector, and the v-vector. Part 5 of your code should show explicit definitions of forces, `F1` and `F2`. Part 5 should end with a line like `#declare a=(F1+F2)/m`. Why do this? To test your understanding of how a net force of zero changes an object's velocity.

NLI.e: Same as MovieNLI.d, but make the upward force larger than the downward force. Why do this? To test your understanding of how a non-zero net force changes an object's velocity.

NLI.f: Same as MovieNLI.d, but make the downward force larger than the upward force. Why do this? To test your understanding of how a non-zero net force changes an object's velocity.

NLI.g: The Force-Maze. Look online and get the code called `forcemaze.pov`. Set your final frames equal to some big number like 200 or so. If you render this movie outright, you will see a red sphere at the entrance to a maze. For this movie, start by sending the sphere up into the maze. You can change the mass to whatever you want (in the `#declare m=...` statement. The golden bars are a maze. Your job here is to apply forces to the sphere that will steer it through the maze, so that comes out through the exit opening to the right of where it started. The sphere **may not touch the golden bars making the maze**. You will do this by applying a succession of forces in Part 5 of the code, that are based on the system time variables `xtime`, as discussed in class. The `#if` statements you supply must only define a force vector, in the form `#declare F=<Fx,Fy,0>;`. Part 5 of your code should end with $a = F/m$. Draw the force and velocity vectors on the sphere at all times. You will likely complete this work by rendering your move over and over again, keeping watch on the time variable rendered in the lower left corner of the screen, grabbing this number and using it in successive `#if` statement in Part 5. Good luck.

NLI.h: The Force-Well. Look online at `http://goo.gl/zTtuQ` and get the code called `forcewell.pov`. Set your final frames equal to some big number like 200 or so. If you render this movie outright, you will see a red sphere moving into the entrance to a path that leads down into a well. A green bar is at the bottom of the well. You can change the mass to whatever you want (in the `#declare m=...` statement in Part 2 of your code. The golden bars are walls. Your job here is to apply forces to the sphere that will steer it through the walls, down to the bottom of the well. The sphere must just barely touch the green bar, then return to leave, back out of the path again. We must see the red sphere pass leftward through the yellow star as the movie ends. The sphere **may not touch the golden bars making the maze, and must just barely touch the green bar at the bottom of the well before during around again**. You will do this by applying a succession of forces in Part 5 of the code, that are based on the system time variables `xtime`, as discussed in class. The `#if` statements you supply must only define a force vector, in the form `#declare F=<Fx,Fy,0>;`. Part 5 of your code should end with $a = F/m$. Draw the force and velocity vectors on the sphere at all times. You will likely complete this work by rendering your move over and over again, keeping watch on the time variable rendered in the upper left corner of the screen, grabbing this number and using it in successive `#if` statement in Part 5. Good luck. WHY DO THIS? TO TEST YOUR UNDERSTANDING OF HOW A 2D FORCE CAN CHANGE A 2D VELOCITY; THIS IS A FORCE$\rightarrow \Delta v$ STUDY.

NLI.i: Lunar Lander. Look online at `http://goo.gl/zTtuQ` and get the code called `lander.pov`. Set your final frames equal to some big number like 200 or so. If you render this movie outright, you will see lunar lander (rocket) falling toward the surface of a strange planet. A green landing pad is at the bottom of a valley. You have to apply thrusts (i.e. forces) on your rocket to guide it to a soft ($|v| < 1$ m/s) landing on the landing pad, without the lander touching the mountains at all. You can change the mass to whatever you want (in the `#declare m=...` statement in Part 2 of your code. You can apply a succession of thrusts in Part 5 of the code, that are based on the system time variables `xtime`, as discussed in class. The `#if` statements you supply must only define a force vector, in the form `#declare F=<Fx,Fy,0>;`. Hint: for quick rendering while you are working on your thrusts, comment out the mountains by putting // (two slashes) in front of the `cone` statements found in Part 10 of the code. Draw the force and velocity vectors on the sphere at all times. Be sure the rocket is always oriented in the direction of the F-vector (use `atan2` etc. for this), so it looks like the rocket is going in the direction of the thrust (but be sure F is never complete zero or `atan2` will fail). It must land upright as well (do not send it into a nose-dive into the landing pad). You will likely complete this work by rendering your move over and over again, keeping watch on the time variable rendered in the upper left corner of the screen, grabbing this number and using it in successive `#if` statement in Part 5. Happy landing! WHY DO THIS? TO TEST YOUR UNDERSTANDING OF HOW A SUPERPOSITION OF FORCES IN 2D CAN CHANGE A VELOCITY, ALSO IN 2D. THE SUPERPOSITION HERE IS THE ADDITION OF THE (PERSISTENT) WEIGHT (DOWNWARD) AND THE THRUST OF YOUR CHOOSING.

NLI.j: The rocket. Get the code called `rocket.pov` at `http://goo.gl/zTtuQ` . If you render

it outright, a small rocket will be at the bottom and just sort of sit there. Go into the code now, and in Part 2, give it a mass of 10 kg. Next, in Part 5 of the code, give it a net downward force that is equal to its weight. The last line of Part 5 should be a declaration that $a = F/m$. Render the movie now and the rocket should fall downward under its own weight. Next, go back to Part 5 and under the line where you assigned it a force of its weight, start an #if statement that checks if the y-coordinate of the rocket (`pos.y`) is less than -20. (This line will mean if the rocket is between it starting position of -25 and -20, then give it an upward thrust. If so, in addition to its weight, give the rocket an upward thrust of 600N. Set your final frames to some big number, like 200 and render the movie. Draw F and v vectors on the rocket at all times.

NLI.k: Inertia. Start an object with $m = 1$ at an initial position of $< 0, -10, 0 >$ with an initial velocity of $< 0, 10, 0 >$. If the the y-coordinate of the object (i.e. pos.y) becomes greater than 0, apply a force of $\vec{F} =< 10, 10, 0 >$. Part 2 of your code should end with $a = F/m$. Render the movie with F and v-vectors on the object at all times. WHY DO THIS? TO TEST THE STATEMENT: "INERTIA IS A MEASURE OF THE AMOUNT OF TIME THAT MUST PASS BEFORE AN OBJECT EXHIBITS SOME DESIRED RESPONSE TO A FORCE."

NLI.L: Inertia. Repeat NLI.k only with $m = 10$.

NLI.m: Start an object at position $< 0, 0, 0 >$ and moving toward the right. Apply the following force on it at the beginning of Part 5 of your code.

$$F(x) = \begin{cases} -5 & \text{if } x \geq 5 \\ 5 & \text{if } x \leq -5 \end{cases} \tag{11.1}$$

Be sure Part 5 ends with a statement like `#declare a=F/m`. Render many frames of this movie so you can fully understand what the force does to the speed of the object.

NLI.n: Drag. Launch a projectile above the ground at some angle θ with some velocity $\vec{v}$ as you did in the last chapter. Next, apply a drag force to the object of the form $D = cv^2$, where $c = 0.1$, and v is the magnitude of the velocity at any instant. The drag force should always be exactly anti-parallel with respect to $\vec{v}$, or at 180° with respect to v. So in Part 5 of the skeleton code, these steps should help in applying the drag force:

1. Compute the magnitude of $\vec{v}$. Finding the magnitude of a vector is covered in Chapter 6 and Section 3.3.4.
2. Compute the magnitude of the drag force using $D = cv^2$ or $D = 0.1v^2$, where v^2 is the square of the magnitude of the velocity.
3. Compute the angle the v vector makes with respect to the xy-axis. You can do this using the arctangent idea described in Section 3.3.3, Section 3.3.4, and Chapter 6.
4. Find the components of the drag force as follows. Suppose this angle the v-vector makes is called A. An angle that is always antiparallel to A is would be found by adding 180° or π to A as in $A + \pi$, or `A+pi` in Povray. Now, you can find D_x and D_y using $D_x = D\cos(A + pi)$ and $D_y = D\sin(A + pi)$.

5. Use your newly found drag force to define the net force on the object as Part 5 of your skeleton code comes to a close.

NLI.o Start an object in the lower left corner of your screen. Put a speed of $v_x = 5$ on it. After some time has gone by, in Part 5 of the code, apply a force on it in any direction you like. Here is an example.

```
#if (xtime > 0.2)
#declare F=<1,1,1>;
#end
```

After some more time has gone by, apply a different force (by adding more `#if` statements like the one above, after the one above). Keep doing this until you apply a total of 5 difference forces on the object, all at different times. Apply the forces so that you keep the object on the screen for as long as possible.

NLI.p: Start an object at position $< 0, 0, 0 >$ and moving upward. Apply the following force on it at the beginning of Part 5 of your code.

$$\vec{F} = \left\{ \begin{array}{ll} < 0, -5, 0 > & \text{if } y \geq 5 \\ < 0, 5, 0 > & \text{if } y \leq -5 \\ < 0, 0, 0 > & \text{otherwise} \end{array} \right\} \tag{11.2}$$

Be sure Part 5 ends with a statement like `#declare a=F/m`. Render many frames of this movie so you can fully understand what the force does to the speed of the object. Be sure F and v-vectors appear on the object at all times.

11.4 Wrap-up Questions

1. Suppose an object (at rest) has an F_{net} vector pointing up and to the left. Sketch the object's subsequent motion as time ticks onward.

2. Suppose an object has an F_{net} vector pointing up and to the left, and it has an initial velocity which is directly rightward. Sketch the object's subsequent motion as time ticks onward.

3. Discuss: As time ticks forward, the v vector always orients to align with the F_{net} vector.

4. Suppose v is aligned with the F_{net} vector. What does the v vector do now?

5. Same question as above, but for F_{net} antiparallel to v.

6. Suppose an object has a v vector on it pointing to the right. The F_{net} vector on it is zero. What does the v vector do now?

7. An object has a v-vector on it pointing toward the right. There are 113 forces on it, and after a bunch of work you notice that $\Sigma F_x = 0$ and $\Sigma F_y = 0$. What does the v vector do now?

8. Discuss: accelerations come from forces.

9. In NLI.a (or d), how did the v-vector change when two forces are applied to an object that cancel each other?

10. In NLI.j, describe how the v-vector changes when the rocket is subjected to its weight, and then its weight + the thrust.

11. In NLI.p, the rocket should oscillate up and down. Why does it do this?

12. Compare and contrast the motion you see in movies NLI.k and NLI.L in the context of a concept called "inertia."

13. One of Newton's Laws says "An object at rest stays at rest unless acted on by an external force." Discuss this statement in light of NLI.a or NLI.d.

14. What would happen in NLI.g (or NLI.h) if you doubled the mass of your object? Why?

15. Summarize how a force can change a velocity vector.

Chapter 12

Forces and Newton's Laws (Part II)

12.1 Introduction and Goals

The goal of this project is to demonstrate that you understand how forces, tied to an identifiable agent, direct an object's motion.

- This week, we'll change an object's v-vector by allowing forces with a clearly identifiable agent to act on an object. These *forces* will change v-vectors.
- Demonstrate you understand the force a spring imparts to an object including the "equilibrium position."
- Demonstrate that you understand how a pulley and rope can be used to link the motion of two otherwise independent objects.
- Demonstrate that you can simulate the motion of simple mechanical systems (i.e. machines) driven by Newton's Laws.
- Demonstrate that you understand the forces experienced by an object on a sloped surface.

12.2 The Physics

The v-vector of an object will be changed by: Identifying the net-force on an object, that is likely linked to another object. Objects connected by ropes and pulleys. Ropes in freshman physics are always massless and these rules apply. 1) Ropes always pull away from their points of contact. That is, tensions in ropes are always drawn pointing away from the point where an the rope connects to an object, along the rope itself. 2) You cannot push on a rope. Ropes may only be pulled on. 3) Objects connected by ropes will always have the same *magnitude* of v (velocity) and a (acceleration), although the algebraic signs of v and a might be different for each object. 4) The tension in a rope connecting two objects is the same throughout the rope. 5) Tensions on opposite ends of a rope must have opposite algebraic signs for use in any equations. That is, the tensions on either ends of a rope that are pulling on their

respective objects, always point toward each other, along the rope. This is the only way in which both ends can *pull* on the objects to which they are connected. 6) If a rope passes over a massless pulley, the magnitude of its tension will not change. A massless pulley just changes the direction of the rope, hence tension. That is tension on one side of a pulley will be the same on the other side, save for the opposite algebraic signs required. 7) If a rope passes over a real pulley (with a defined mass and radius), then the tension on one side of the pulley *does not* in general equal to the tension on the other side of the pulley, and you should not assume it is. The opposite sign rule applies to the different tensions. **Objects on sloped surfaces.** If an object is on a surface sloped at an angle θ, there will be a downward "sliding force" on the object of magnitude $mg\sin\theta$, in a direction pointing down and parallel to the slope. This sliding force is what causes objects on sloped surface to want to slide or roll downhill. Gravity g is the originator of this force. **Objects that interact with springs.** Suppose a spring has a free end and a fixed end. The free end can "spring along" the direction of the spring itself, and the fixed end cannot move at all. Suppose also when nothing is touching the spring (when the spring is in equilibrium), the free end is physically located at position s_0. When the free end of the spring is displaced to a position s, the force the spring exerts on an object connected to its free end is $F = -k(s - s_0)$. This is Hooke's Law. The minus sign indicates that the spring force always opposes the direction of s relative to s_0, and k is the "spring constant" or the stiffness of the spring (the larger k the stiffer the spring). **Weight.** When an object that has a mass m is in a gravitational field, it will have a force on it called weight, which is $W = mg$. This force always points straight down, no matter what other orientation or situation the with which the object might be involved. **Normal force.** When an object with weight W is placed on a surface, it disrupts the equilibrium position of the molecular structure forming the surface on which the object is placed. The desire of these molecules to want to return to equilibrium causes them to push back on the object with a force called the normal force. The normal force is *always* perpendicular to the surface on which the object is sitting. It is tempting to call the magnitude of the normal force mg, or the weight of the object, but this only true when the object is sitting on a flat surface. In other situations, the magnitude of the normal force can only be found by summing forces perpendicular to the surface and setting the sum equal to zero, then solving for N. **Friction.** Kinetic friction is a force that is always oriented exactly opposite to an object's v and has a magnitude of $f = \mu N$, where μ is the coefficient of kinetic friction and N is the normal force on the object, due to the surface on which it sits. **Centripetal force.** If a situation arises where a force F is locked at 90° with respect to an object's v-vector, then the force is called a centripetal force. The force will cause the object to move in a circle of radius R if the force magnitude is mv^2/R, where m is the mass of the object. **Book reading: 7.4, 7.5**.

12.3 Projects

NLII.a. Friction: Show how a block is slowed by friction.] Start with the skeleton code online and put these lines into Part 10 of your code (after the last #end statement).

```
box { <10,-1,-10>,<30,-2,10> pigment {Green}}
box { <10,-1,-10>,<-10,-2,10> texture{Brown_Agate}}
```

```
box { <-10,-1,10>,<-30,-2,-10> pigment {Green}}
```

This will draw two frictionless green surfaces around a central rough friction surface. Next, give a box, starting at $x_0 =< 30, 0, 0 >$, some initial $v_0 =< -v_x, 0, 0 >$ toward the right. Start this movie with the skeleton code found online. You can find help drawing boxes in Section 4.3.

Your block must move from right to left in this movie. The ground from $10 \leq x \leq 30$ is frictionless. From $-10 \leq x \leq 10$ it has friction with coefficient μ. In Part 2 of your code, declare values for m, g, and μ. Your block should slide appropriately across the frictionless surface. When it hits the friction, compute and apply the proper frictional force to the block. You can complete this work by declaring needed variables in Part 2, placing proper `#if-#end` statements in Part 5 to apply the proper forces based on `pos.x`, then drawing the object at `pos` in Part 10. Render both v and the given F on the block at all times.

NLII.b Friction: Start with the code `frict01.pov` at `http://goo.gl/zTtuQ` . If you render the code, a block will be shown moving from left to right. There middle section is rough (with friction) and the outer regions are smooth with no friction. You job is to apply friction on the block when it passes over the rough patch. Choose your block's v_x and μ so that the block goes over the right edge of the surface, enters free fall, and goes through the hoop. See Figure 12.1 for the spatial outlay of the objects in this scene. You'll need to use an `#if` statement to assign the friction force and to test for freefall. Draw F and v vectors on the block at all times.

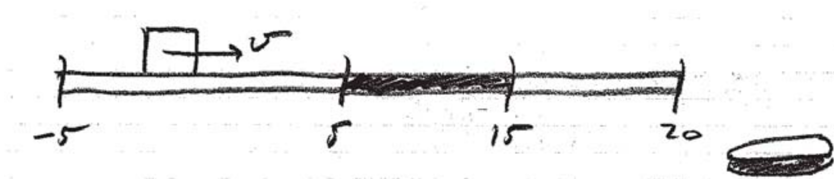

Figure 12.1: Spatial outlay of objectsfor NLII.b

NLII.c Friction: Start with the code `frict_spring.pov` at `http://goo.gl/zTtuQ` . If you render it, a block is heading left toward a brick wall. There is a patch of friction in the scene too. Insert a spring (with proper spring physics) between the block and the brick wall, so that the spring will send the block back toward the right, over the friction, then over the edge of the platform. Adjust μ (of friction), k (of the spring), and v_{0x} of the block so that the block eventually goes through the hoop, after it falls over the right edge of the platform. The spring should have its fixed end attached to the brick wall. The spatial outlay is the same as that shown in Figure 12.1. The brick wall starts at $x = -5$ and extends toward the left. Draw F and v vectors on the block at all times.

NLII.d Friction: Start with the code `frict_hang.pov` at `http://goo.gl/zTtuQ` . If you render it, it'll show two blocks, a table, some friction, and a pulley. Your job is to connect

the blocks with a rope, and make the hanging block more massive than the block on the table. When run, we should see the hanging block move down, and pull the block on the table over the patch of friction. The spatial outlay of the objects in the scene is shown in Figure 12.2. Draw F and v vectors on the block at all times.

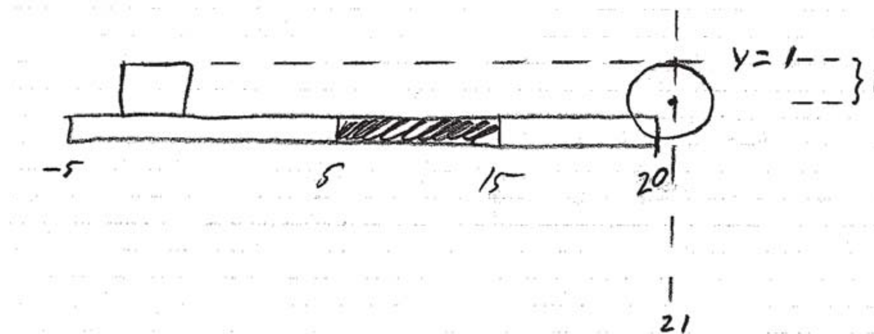

Figure 12.2: Spatial outlay of objectsfor NLII.d

NLII.e. Friction: This movie is to have you show how a block is slowed by friction. Start with the skeleton code (online) called `friction.pov` which has these lines into Part 10 of it (after the last #end statement).

```
box { <15,-1,-5>,<30,-2,5> pigment {Green}}
box { <15,-1,-5>,<5,-2,5> texture{Chrome_Metal}}
box { <5,-1,5>,<-5,-2,-5> pigment {Green}}
box { <-5,-1,-5>,<-10,-2,5> texture{Cherry_Wood}}
box { <-10,-1,5>,<-15,-2,-5> pigment {Green}}
cylinder { <-22,-5,0>,<-22,-5.3,0>,3 open pigment {Yellow}}
```

These lines will draw frictionless green surfaces with a section of metal between $5 \leq x \leq 15$ and a section of wood between $-10 \leq x \leq -5$. The leftmost edge is at $x = -15$. Start a box at $x_0 =< 25, 0, 0 >$, some initial $v_0 =< -v_x, 0, 0 >$ toward the right. Color the box with a pigment of `Pine_Wood` to make the box look wooden. You can find help drawing boxes in Section 4.3.

Your block must move from right to left in this movie. The metal has a coefficient of friction μ_m and the wood μ_w. Since the block is wood, find values for the two $\mu's$ and declare them in Part 2 of your code. Also in Part 2, declare values for m and g. Your block should slide across the entire surface from right to left. When it hits either friction patch, compute and apply the proper frictional forces to the block. You can complete this work by declaring needed variables in Part 2, placing proper `#if-#end` statements in Part 5 to apply the proper forces based on `pos.x`, then drawing the object at `pos` in Part 10. Render both v and the given F on the block at all times. Choose all values so the box falls through the yellow hoop just off of the left edge.

NLII.f. Spring launcher: Start with the code online called `spring_launch.pov`. Render it and you'll see a box sliding toward a spring on a table. Program in both the spring force and the free fall force so that the box will hit the spring, recoil off of it slide off the edge of the table. `#if-#end` statements in Part 2 will handle the whole thing. Draw F and v vectors on the box at all times.

Your code should explicitly define k for the spring, the equilibrium position of spring, and the position of the "active" edge of the spring (the edge of the spring that will touch the box). For the spring to work properly on the computer, you must use a $dt = 0.01$. If your step size is larger, the speed at which the block leaves the spring will be larger than the speed at which it initially impacted the spring, which is not correct here. With this small dt, you will need to render several hundred frames to see the full motion. Be sure to stitch this movie together at no less than 30 frames per second. *As you watch your finished movie, try to understand how a spring, and its Hooke's Law force, changes an object's v-vector.*

You can find help with springs in Chapter 6. Drawing your spring (in Part 10) should resemble `draw_hspring(-20,x_s,0,1,0.3)`, where -20 is the fixed end of the spring, way off to the left. If you so desire you can draw a nice "spring plunger" by also drawing a thin horizontal block on the free end of the spring.

NLII.g. Vertical spring: Draw a vertical spring between $y = -10$ and $y = 0$, at $x = 0$. See `draw_vspring` in Chapter 8 for help drawing springs. From a position of $< 0, 10, 0 >$ allow a red ball to fall onto the spring. Program just the weight on the ball if $y > 0$ and the weight + the spring force if $y <= 0$. The ball should fall into the spring, compress the spring to some stopping point, then be relaunched upward by the spring. Work carefully and you'll see that as complicated as this sounds, it's really just a matter of adding two `#if` statements setting up your forces in Part 5. Draw F and v vectors on the box at all times.

Your code should explicitly define k for the spring, the equilibrium position of spring, and the position of the "active" edge of the spring (the edge of the spring that will touch the box). For the spring to work properly on the computer, you must use a $dt = 0.01$. If your step size is larger, the speed at which the block leaves the spring will be larger than the speed at which it initially impacted the spring, which is not correct here. With this small dt, you will need to render several hundred frames to see the full motion. Be sure to stitch this movie together at no less than 30 frames per second. *As you watch your finished movie, try to understand how a spring, and its Hooke's Law force, changes an object's v-vector.*

You can find help with springs in Chapter **??**. Drawing your spring (in Part 10) should resemble `draw_vspring(0,-10,y_s,0,1,0.3)`, where -10 is the fixed end of the spring and y_s is the free end of the spring. If you so desire you can draw a nice "spring plunger" by also drawing a thin horizontal block on the free end of the spring.

NLII.h. The Atwood Machine: Start with the skeleton code. An Atwood Machine can be drawn as shown in Figure 12.3.

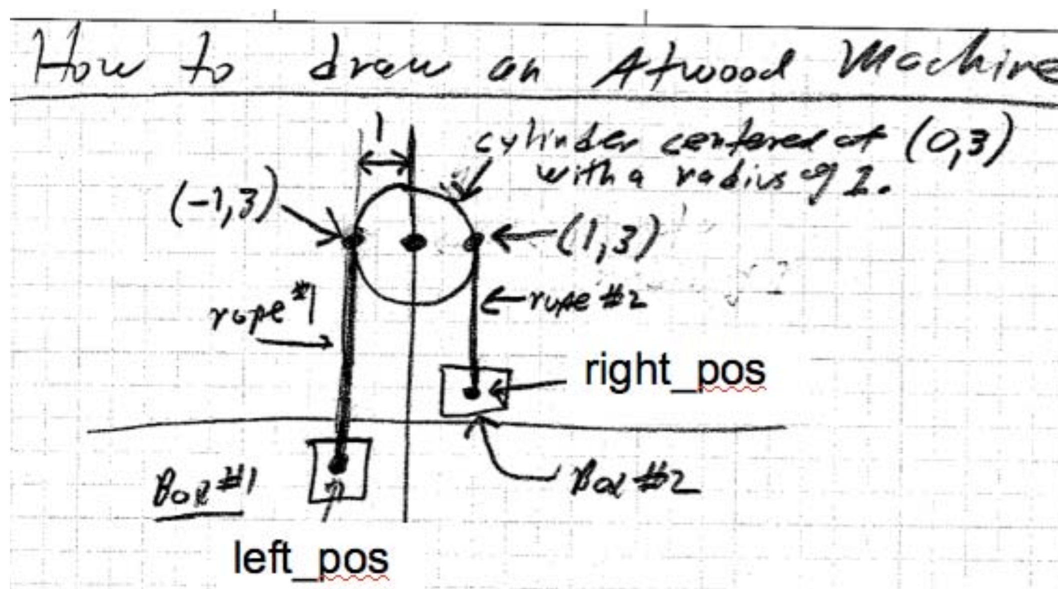

Figure 12.3: A guide in helping you to draw at Atwood machine using Povray.

It consists of a cylinder as a pulley and two ropes extending from the edges of the pulley down to the active position of each object. The pulley and ropes are important parts of this movie. In Part 10, draw the Atwood using a cylinder as the pulley as in $cylinder\{<0,3,1>,<0,3,-1>,1\ pigment\{Gray\}\}$. You'll have to declare an m_1 and m_2 in Part 2. This movie is also a bit different because two objects will be moving around, instead of the usual single object. So instead of just a `pos` and `vel`, in Part 2, you'll need a `left_pos`, `left_vel`, `right_pos` and `right_vel` to set and track the positions and speeds of both blocks. You'll need to fix Part 6 to have the two physics equations for both objects. Ropes can be drawn as described in Chapter 8 with the `draw_real_rope` statement.

Make the left block more massive than the right block and initially have the right block moving down and the left block moving up. Run your movie so that we can see the blocks move, stop and reverse direction. Be sure the size of the hanging objects are indicative of their relative masses. That is, if the left object has more mass than the right block, make the left appear larger. The acceleration of this system will be derived in class. Render the v and F vectors on both blocks at all times. See the sketch under the help section for this project (online), which suggests how your Atwood might be oriented. For this movie, the blocks will have constant x-coordinates throughout their motion. Their y-coordinates will be calculated using the physics equations. *With this movie, try to understand how the blocks, coupled by the rope, change each others' v-vectors.*

NLII.i: The "arctan" hill. This project is meant to "test your belief" that accelerations are what drive motion. This movie will allow you to observe the nature of the normal force exerted by a surface by showing you an object moving along a sloped surface. Start with the code online called "arctan.pov." A good sloped surface is the function $y(x) = 1 - \arctan(x)$,

as shown in class, which makes a nice flat ground, a gentle upward slope, followed by a flat plateau. In this movie, start a sphere on the right (flat) portion of the function and send it toward the right with some $v =< -v_x, 0, 0 >$. Animate the subsequent motion of the sphere as it moves. Choose the initial v_x to the left so that your object doesn't quite make it all the way up the sloped portion, and that it'll stop on the slope and slide back down. Render the v, a and N vectors on the sphere at all times. The components of the Normal force (needed to draw the N-vector) can be found from $N = m < a_x, a_y + g, 0 >$. Notes:

- For $y(x) = 1 - \arctan(x)$, find y' and y'' (this will be your answer to question #1 below).
- Start with some v_x that you choose. From this, you can find a_x and a_y from

$$a_x = \frac{-y'(y''v_x^2 + g)}{1 + y'^2}, \tag{12.1}$$

  and

$$a_y = y''v_x^2 + y'a_x. \tag{12.2}$$

- With your a_x and a_y, you can now compute your net acceleration vector at the end of Part 5, for feeding into Part 6 from $a =< a_x, a_y, 0 >$.
- You'll probably have to render a lot of frames for this, something like 200 or so. Stitch your movie together at a nice fast frame rate, like 25 frames per second. We don't want to watch and grade a movie that takes 2 minutes to run!
- ORGANIZE YOUR VARIABLES. Declare two variables called `yp` and `ypp` to set what y' and y'' are at that instant, that you can later use in subsequent calculations.
- When your movie is done, just watch it. I hope you can appreciate two things. First, watch how the a vector "struggles" to keep v moving along the track. Second, watch where N gets big and small; can you think of why it behaves like it does? There is much more to N that "is is a force perpendicular to the surface, etc."

  With this movie, try to see how the normal force exerted by a surface is an ultra dynamic process resulting in variable length N-vectors that are always perpendicular to the surface. Also try to see how the slope of a surface changes an object's v-vector.

NLII.j: Do you believe that "a" drives motion? This project is meant to "test your belief" that accelerations are what drive motion. This movie will allow you to observe the nature of the normal force exerted by a surface by showing you an object moving along a sloped surface. Start with the code online called `poly_start.pov`. This will draw a sphere at the center bottom of a parabolic track. In this movie, start the sphere going left or right some $v =< v_{x0}, 0, 0 >$. Animate the subsequent motion of the sphere as it moves. The dt in your movie must be small, like around 0.01. Don't make v_{x0} much bigger than 5 or so. Render the v, a and N vectors on the sphere at all times. The components of the Normal

force (needed to draw the N-vector) can be found from $N = m < a_x, a_y + g, 0 >$. Notes: For $y(x) = Ax^2$, start by finding y' and y''. Read the bullet notes under the previous problem for more hints.

12.4 Wrap-up Questions

1. For the $y(x) = 1 - \arctan(x)$ hill, or $y = Ax^2$ (whichever you did) find $y'(x)$ and $y''(x)$.
2. What happens to a of an Atwood machine if one of the masses is cut off?
3. Fully describe your observation of the spring force while the block is in contact with the spring, from first contact to final release.
4. How does friction change an object's v-vector?
5. How does a spring change on object's v-vector?
6. How does "the other mass" change a given mass's v-vector in the Atwood machine?
7. How does the slope of a surface change on object's v-vector?
8. Draw a wildly curved surface. Draw several points along the surface and draw the normal force that would be exerted by the surface on an object at that point. Careful with the magnitude of your normal vectors noting that $|N| \sim \cos\theta$, where θ is the inclination angle of the surface.
9. Sketch the $1 - \arctan(x)$ hill or the $y = Ax^2$ track (whichever you did). Where is N the largest and under what circumstances? Smallest? If this were a roller coaster, what portions of the track would you need to build to be very strong?
10. Fill out the study grid found in the "out of class work" document for the 7 days extending from 1/25-1/31. How many hours did you put in for this course, and how does it compare with what the 25/35 program recommends?

Chapter 13

Energy: Work, Kinetic, Potential, and Conservation

13.1 Introduction and Goals

The goal of this project is to demonstrate that you understand kinetic energy, potential energy, work and the conservation of energy.

- You'll change an object's v-vector by allowing *kinetic energy* to flow into or out of an object.
- Demonstrate that you understand how to compute and use KE and PE using the kinematic variables x, y and v.
- Demonstrate that you understand that $KE + PE =$ a constant.
- Show how "energy bar charts" can be used to illustrate the instantaneous energy distribution of an object.
- To see how the instantaneous energy distribution of an object depends on its speed and position.

13.2 The Physics

The v-vector of an object will be changed by: Adding or removing kinetic energy from an object. No one know what energy is, but it can be compared to money and time (you can lose, gain, save, waste, or spend them, etc.), and we all "know" what energy, money, and time are until someone asks us to tell them what they are! We'll focus on two types of energy, Kinetic energy (KE) and Potential energy (PE), a way of "processing" energy, called "work," (W) and a guiding principle, called "conservation of energy." The units of energy (KE, PE, and W) are in Joules, or J. KE is energy something (of mass m) has because it it moving with some speed v, or $KE = \frac{1}{2}mv^2$. If an object is moving it has KE; if it is at rest, it doesn't. PE is stored

energy that has not been released yet to do something. Our society is usually concerned with chemical or nuclear PE (oil/gasoline, natural gas, nuclear power plants), but in this class we'll only concern ourselves with mechanical PE, and further, only three types of it. Gravitational PE, or $U_g = mgh$, spring PE, or $U_{sp} = \frac{1}{2}k(x - x_n)^2$, and pendulum PE, $U_{pend} = mg(1 - \cos\theta)$. U_g is PE an object has because it is not trapped at some lowest possible position to which it may fall. This lowest PE position can be tricky to identify and is not always at ground level. You must examine an object's position and ask yourself: "If the object was carefully placed at rest, at this position, would be able to fall down any farther if given a small nudge?" In U_g, mg is the weight of the object and h is its vertical distance above the lowest possible position. U_{sp} is energy a compressed or expanded spring may store, where k is the spring constant, x is how far the end of the spring has been expanded or compressed past its "natural" position at x_n. If $x = x_n$, then the spring is neither expanded or compressed, and $U_{sp} = 0$. If $x > x_n$ or $x < x_n$ then $U_{sp} = 0$. Pendulums are any mass that can swing from a very light rope attached to some higher point and are great examples of objects whose PE is zero when the mass is not on the ground. In the equation for U_{pend} above, θ is the angle the pendulum makes with respect to the vertical (where the mass is directly below the upper attachment point of the rope, when $\theta = 0$). Work is a way of using a force to inject or remove energy to or from an object. For us, $W = F \cdot \Delta r \cos\theta$, where F is a force applied to the object, Δr is how far the object moved while the force was applied, and θ is the angle between the force and the direction of Δr. if $W > 0$ then energy will be added to the object, if $W < 0$ then energy will be taken from the object. For us, F and Δr will always positive; the sign of $\cos\theta$ will determine the sign of W. Friction always results in energy loss or $W < 0$. Lastly, we have the law of "conservation of energy" (CE). CE states that the sum of KE and PE is always a constant. This means if PE goes up, KE must go down and if PE goes down, KE must go up, both in such a way to keep $KE + PE =$ a constant. The "constant" is the total energy of the system. The most useful form of this law is that in realizing that if the sum of KE and PE are constant, then the sum of KE and PE, say at some point A in the object's motion will be the same as the sum of KE and PE at some point B in the object's motion, or $KE_A + PE_A = KE_B + PE_B$. Also, since KE is always $\frac{1}{2}mv^2$, then $\frac{1}{2}mv_A^2 + PE_A = \frac{1}{2}mv_B^2 + PE_B$, which, if you just fill in your associated PE function begins to form a useful "physics equation" that can be used to solve problems. Work ties into this all by showing where energy is injected or is lost by the object. Here's a useful form that includes work: $\frac{1}{2}mv_A^2 + PE_A + W = \frac{1}{2}mv_B^2 + PE_B$, which shows how if $W > 0$ (energy into the object) will lead to a greater total energy represented on the left side of the equation. $W < 0$ would lead to a smaller total energy on the left hand side. **Book reading: p. 270 starting at "Kinetic Energy," 10.5, 11.2, 11.3, 11.8**.

13.3 Projects

In these projects, you'll be having the computer calculate energies, like KE, PE and E. These can be handled in Part 10 of your code with lines like

```
#declare vmag=vlength(vel);
#declare KE=0.5*m*vmag*vmag;
```

```
#declare PE=m*g*pos.y;
#declare E=KE+PE;
```

Energy.a: Using your projectile (2D) motion movie (twodim.a, in Chapter 11), add "energy bars" to it that show the total energy (E), kinetic energy (KE), and potential energy (PE) of the object as it flies. Use the `draw_bar` functionas described in Section 8.3, to draw your energy bars. Also, find a place on your screen to draw the numerical values of E, KE and PE, using the `draw_variable` function (see Section 8.3) . As an example `draw_variable(<-5,1,0>,E,"J",Yellow,1)` will draw the current numerical value of the variable E at $< -5, 1, 0 >$ with the label of "J" (for Joules) in a yellow color with an overall zoom factor of 1. Draw v on the object at all times. All energy computations and bar drawing should occur in Part 10. Use `pos0.y` for the instantaneous height of the object, and use `vmag=vlength(vel)` to compute the magnitude of the object's velocity, needed for KE calculations. Bars that are too long or too short can be scaled with the `set_bar_scale` and `set_bar_zoom` functions (see Section 8.3).

Energy.b: Using your spring movie from last week (NLII.d), add total energy E, KE, spring potential energy (PE_{spring}), and gravitational PE bars to the movie as the block moves. Draw v and F vectors on the project at all times. Use `draw_variable` to draw numerical values of E, KE, and PE as well.

Energy.c: Using your arctan movie from last week (NLII.f.), add total energy E, KE, and gravitational potential energy ($PE_{gravity}$) bars to the movie as the object movies up and back down the hill. Draw v and F vectors on the project at all times. Use `draw_variable` to draw numerical values of E, KE, and PE as well.

Energy.d: Using your friction movie (NLII.b), add E, KE, and PE bars to the block as it slides. The effects of the friction patch must be very noticeable in your movie. This means we should see the total E bar drop with each pass over the friction patches.

Energy.e Start with your code for `Energy.d`, but remove part 5 entirely, and replace it with an `a=<0,0,0>` line. Let's use the work done by friction explicitly to slow the block down as it rubs over the friction surface, not accelerations. For this, we'll put code into Part 8 of the skeleton code. See Section 5.4 to review this part, which involves abrupt changes in *velocity*.

To start, put a line like this `#declare pos0=pos;` right at the end of Part 5. A line like this will save the current position of the block in the variable `pos0`, just before the physics equations advance it to the next position. We need this because work involves a Δr, which the net displacement of the object.

Now, in Part 8, start an `#if` statement to check if you are on a patch of friction. If true, follow these steps to suck a bit of energy from the block, as per the "work done by friction."

$W_f \leftarrow -\mu mg|x - x0|$ (compute work due to friction; this is $W_f = -f\Delta x$.)
$v \leftarrow \sqrt{v_x^2 + v_y^2}$ (compute the magnitude of velocity)

$KE \leftarrow \frac{1}{2}mv^2$ (compute the KE)
$KE_1 \leftarrow KE + W_f$ (compute the new KE after work due to friction)
$\vec{v} =< \pm\sqrt{\frac{2KE_1}{m}}, v_y, 0 >$ (compute the new velocity-vector from the lower KE)

Next to the energy bars, draw a "work bar," which shows the totality of work done by friction in slowing down the block (i.e. this bar should grow and grow as the block slides over friction). Computing the magnitude of the velocity is discussed in Chapter 6 under "finding the magnitude of a vector." You'll have to think about on which root to choose ($\pm$) when computing the new velocity-vector.

Energy.f: A skateboarder dude is riding up and back in a "half-pipe." The pipe is described by the function $y(x) = 10 - \sqrt{100 - x^2}$. Use the code `skateboarder_start.pov` online to get started. Place the dude (in the form of a lame-looking sphere) at the bottom of the half-pipe $< 0, 0, 0 >$ with some initial $v_{x0} = 0$ that will send him toward the right. The movie should show him ride up and down the half pipe a few times, going up the "arms," where he'll slow, stop, then slide back down again, only to start up the opposite "arm." Draw KE, PE, and E bars at all times as well as the v and F vectors. Physically, you are sending an object to ride on a surface $y(x) = 10 - \sqrt{100 - x^2}$, like last week with the arctan hill. So, you'll need the acceleration equations from last week's work on the arctan movie to get your object moving properly. Recompute y' and y'' and use your results in your code. Putting y'' or `ypp` into your code is kind of a pain in the neck, so here's a line that should do it

```
#declare ypp=pos0.x*pos0.x/pow(100-pos0.x*pos0.x,1.5)+1/sqrt(100-pos0.x*pos0.x);
```

In your final render, you'll have to make dt very small, like 0.01, and render about 500 frames to see the energy bars work out properly. *Your total energy bar must remain nearly constant throughout the motion!* **Important!** With so many frames, you final movie will run very slowly if stitched together at 10 frames per second. Be sure to stitch it at 30 frame per second minimum! Points will be deducted for movies that run too slowly.

Energy.g: Same as Energy.e, but add a patch of friction for $-1 \leq x \leq 1$, so the skateboarder dude drags over it both on his way up the and back down. Use the code `skateboarder_friction_start.pov` online to get started. Friction should do negative work on the skateboarder with each pass. We should see the KE and total energy bars decrease with each pass over the friction. Draw the v and a vectors on the object at all times. Use the same $dt = 0.01$ considerations as in the last movie. Removing energy due to work involves direct speed changes of the object. This logic should be in Part 7 (first time this quarter) of your code and should help you to handle friction from the energy standpoint, given that the block's current position is x, last known position is x_0 and it has components of speed of v_x and v_y:

if ($f_{start} \leq x \leq f_{end}$) THEN

$W_f \leftarrow -\mu mg|x - x_0|$ (compute work due to friction)
$v \leftarrow \sqrt{v_x^2 + v_y^2}$ (compute the full velocity)
$KE \leftarrow \frac{1}{2}mv^2$ (compute the KE)
$KE_1 \leftarrow KE + W_f$ (compute the new KE after work due to friction)
if ($v_x < 0$) THEN (get signs right on new post-friction KE)
  $v_x \leftarrow -\sqrt{\frac{2KE_1}{m}}$ (moving to the left)
ELSE
  $v_x \leftarrow \sqrt{\frac{2KE_1}{m}}$ (moving to the right)
END
END

Note: The effects of the friction patch must be very noticeable in your movie. This means we should see the total E bar drop with each pass over the friction and we should see the skateboarder's ride get lower and lower and lower. In a nutshell: please run your movie run long enough for the effects of the frictional patch to become apparent.

Energy.h: Take your basketball movie (`twod.b`) from a few weeks back and add KE, PE, E bars to it. Use `draw_variable` to show the numerical values of these energies as well.

Energy.i: Take your `NLII.j` movie and add KE, PE, and E bars to it. Use `draw_variable` to show the numerical values of these energies as well.

Energy.j: Get the code `spring_energy.pov` from online. If you render it, a block will run into a spring and be pushed back in the other direction. Add KE, PEs, PEg, and E bars to it. Use `draw_variable` to show the numerical values of these energies as well.

Energy.k: Take your friction movie `NLII.b`. Add KE, PE, E, and Wf bars. Use `draw_variable` to show the numerical values of these energies as well.

13.4 Wrap-up Questions

1. Discuss the similarities between time, energy, and money.
2. Draw a parabolic path of a projectile in flight. Label point B on the path at the peak. Label A on the upward slant and B on the downward slant. Draw E, PE, and KE bars for points A, B, and C.
3. Discuss the exchange of energy between KE and PE for a skateboarder on a half-pipe.
4. Discuss the exchange of energy between KE and PE for a block sliding on a flat (frictionless) surface that runs into a spring.

5. Look up the mass of the car that you drive. Compute how many Joules it takes you to go from 0 mph to 55 mph. There are about $21,000$ Joules of energy in a gram of chocolate chip cookies. Careful with units. Miles and hours do not mix with Joules. How many grams of cookies are you using to get to this speed?

6. There are about $27,000$ Joules of energy is a gram of coal. Think of a single charcoal briquet for your BBQ as almost pure coal. Find out how much mass a single briquet has and compute how many charcoal brickets you burn up each time you accelerate from 0 to 55 mph in your car. Imagine throwing this many briquets out of the window each time you accelerate like this. What would the roadside look like?

7. Draw a sketch illustrating how work due to friction slows an object by sucking energy out of it as it rubs across the rough surface.

Chapter 14

Momentum and Conservation of Momentum

14.1 Introduction and Goals

The goal of this project is to demonstrate that you understand what happens when two objects collide.

- This week, we'll change an object's v-vector by changing it directly with a Δv found by considering the momentum of a system.
- Demonstrate that you understand how to compute and use momentum.
- Demonstrate that you understand that $\Sigma\vec{p} =$ a constant.
- Show how "momentum bars" can be used to illustrate the instantaneous momentum distribution of a system.
- Demonstrate how the momentum bars show that total momentum is constant for a closed system.
- Demonstrate you understand how Newton's 3rd law and "equal of opposite" reaction forces.

14.2 The Physics

The v-vector of an object will be changed by: Causing an object to interact (or collide) with another object. In all of the previous weeks, we concerned ourselves only with isolated objects. This week, we'll see what happens when two (or more) objects interact which each other, in the form of contact between the two bodies (as in a collision, or in the sudden motion of two or more objects due to a need for them to separate due to an explosion). When two bodies come in contact with each other, each exerts a pushing force on the other (think of

the last time you bumped into someone in a crowded place: you felt a push, and so did they). **Further, the force that one exerts on the other is always exactly the same.** This is Newton's third law of "equal and opposite reaction forces." For example, if two cars collide, during the collision, car A will exert a force on car B (F_{AB}), and car B will exert the exact same force on car A (F_{BA}). The two forces will have the same strength, but be in exactly opposite directions to one another. In other words, $F_{AB} = -F_{BA}$. It doesn't matter is one car is heavier (more massive) than the other. The push force from one car will equal the push force from the other. What if one car is a small Honda and the other car a huge SUV? If so, when in contact, the force the Honda exerts on the SUV will be equal to the force the SUV exerts on the Honda, only in the opposite direction. What about a bug hitting a car windshield? The force of the bug on the windshield is equal to the force of the windshield on the bug, only in the opposite direction. Why then does the bug get crushed and the SUV doesn't even feel the collision? Because the resulting motion *after the collision* is driven by the acceleration the body takes from the collision, while in contact with the other object. Suppose the equal and opposite force of the bug-windshield collision is 0.1 N. The bug has a mass of 0.001 gram, or 1×10^{-6} kg. It's resulting acceleration will be $a = F/m = 0.1$ N$/1 \times 10^{-6}$ kg $= 100,000$ m/s^2. The SUV, with a mass of about $4,000$ kg gets an acceleration of $a = 0.1$ N/4000 kg $= 0.000025$ m/s^2. Collisions are typically very brief, say 1 ms, or 0.001 s. During this time, a parameter called "impulse" exists, defined as $J = \Delta p$, which is the change in an object's momentum. How far does each move in this time? The bug will move 5 cm, the SUV will "move" about the diameter of of an atom making up the windshield. The bug get crushed because its internal structure cannot sustain an acceleration of about $10,000$g. This equal and opposite force idea leads to momentum, which is defined as $p = mv$ or more correctly, $\vec{p} = m\vec{v}$. Notice $\vec{p}$ involves velocity directly. We also have "conservation of momentum" that says that $\vec{p}_{before} = \vec{p}_{after}$. The "before" and "after" refer to before and after a collision. This law itself allows us to ignore the physics *of the collision* and instead focus on the physics *just before and just after the collision.* More correctly, the law is $\Sigma\vec{p}_{before} = \Sigma\vec{p}_{after}$, indicating that the law means add all objects carrying momentum before a collision and set equal to the sum of all momenta carrying objects after the collision. Since p is a vector, so you must sum the momenta of all objects in the x direction, then in the y direction, both before and after the collision in order the the conservation law to be helpful. Lastly, there are two types of collisions, elastic and inelastic. In elastic collisions, the colliding objects bounce off of each other, while in inelastic, they all stick together creating a new "conglomerate mass" which is the sum of the individual masses that stuck together. In applying the conservation law for an inelastic collision, you typically have something like $m_1\vec{v}_{1before} + m_2\vec{v}_{2before} + m_3\vec{v}_{3before} + ... = (m_1 + m_2 + m_3 + ...)\vec{v}_{after}$. Notice that there's only one velocity after the collision ($\vec{v}_{after}$) because only the "big blob" is moving after they all collided and stuck together. For an elastic collisions, where two objects (1 and 2) collide along a single axis, we'll have $v_{1after} = \frac{m_1 - m_2}{m_1 + m_2}v_{1before} + \frac{2m_2}{m_1 + m_2}v_{2before}$ and $v_{2after} = \frac{2m_1}{m_1 + m_2}v_{1before} + \frac{m_2 - m_1}{m_1 + m_2}v_{2before}$.
Book reading: 9.1, 9.2, 9.3., 9.4, 10.6.

14.3 Projects

- These movies involve objects colliding just once. To ensure the collision response only occurs once, be sure to direct the sign of the component of velocity involved in the collision.
- You'll have two objects colliding, which means you will have two objects moving in each movie. This means two sets of initial variables (position and velocity) for both objects, and physics equations for each. Review what you did in the Atwood movie. Some might even require separate accelerations for each.
- Objects of differing masses are required in these movies. You must make more massive objects appear larger than less massive objects. Points will be deducted if you do not do this, as this really degrades the physics-based visuals of your movie, in how collisions work.

(Note: in these problems, "mom" stands for momentum.)

mom.a: Start with the projectile code `twodim.a` from week #2, which launches a projectile across the screen. Change the launch angle to something steep like 65°. When the hits the ground, make it bounce, by reversing the sign v_y. You logic condition for detecting a downward collision with the ground should be something like "if $v_y < 0$ and $y <= 0$ then reverse the sign of v_y." It is important in the collision detection to check both the position and direction of travel of the object. Add a coefficient of restitution of 0.85 with each bounce. See hints in problem `momentum.c` for help reversing signs of velocity components. Your animation should show the complete life of the ball, from launch to becoming more-or-less motionless on the ground after bouncing a few times.

mom.b Starting with the code `ballwell.pov` which you can find online, cause the ball coming out of the tube to fall into the red well. Make it bounce off of both the horizontal walls and vertical floor. Put a coefficient of restitution on bounces from the floor. Render the movie until the ball appears to stop bouncing. See hints in problem `momentum.c` for help reversing signs of velocity components. Draw v, v_x and v_y on the ball at all times.

mom.c: Starting with your projectile code from week #2, add the following lines to Part 10

```
box { <30,0,30>,<32,30,-30> pigment { brick pigment{White}, pigment{Red} }}
box { <-30,0,30>,<-32,30,-30> pigment { brick pigment{White}, pigment{Red} }}
plane { <0,1,0>,0 pigment {Green}}
```

which will add two large brick walls at $x = \pm 30$. Let your projectile fly as usual, but have it bounce off of the floor and brick walls by reversing the component of velocity that is along the axis where the collision occurs. Such reversals are to be put into Part 8 of your code and can be done with

```
#declare vel=<-vel.x,vel.y,0>;
```

to reverse v_x or

```
#declare vel=<vel.x,-vel.y,0>;
```

to reverse v_y, as needed. Remember to refer to variables **pos** and **vel** in Part 8. You can add coefficients of restitution (see Wikipedia) to simulate imperfect bounces by adding a decimal in front of the component being reversed, like this:

```
#declare vel=<vel.x,-0.7*vel.y,0>;
```

Draw the v-vector, v_x and v_y on the ball at all times. Draw KE ad p $(= mv)$ bars somewhere in the scene too. Your movie must show several bounces off of the floor and both walls. Make your collisions look realistic by not letting the ball "bury" itself into the walls or ground. More realism will include the coefficient of restitution in the collision response.

mom.d: Start two balls (A and B) moving toward each other. One starts near the top of the screen and moves down. The other starts near the bottom of the screen and moves up. Keep all motion along the y-axis. This is not a free fall movie, just the top view of two balls colliding. Each sphere should be assigned different masses that are reflected in their size on the screen. When they collide, compute the impulse, J that moderates the collision from $\vec{J} = (1+e)m_a m_b(\vec{v}_{Ai} - \vec{v}_{Bi})/(m_a + m_b)$, where e is the coefficient of restitution, which is a number *between* 0 and 1 (say like $e = 0.7$). Also choose a $\Delta t_{collision}$, then compute $\vec{F}_{collision} = \vec{J}/\Delta t_{collision}$. Your animation should show the entire collision, from approach, to collision, to motion after the collision. Your code must explicitly compute set $\Delta t_{collision}$ and e in part 2, followed by the calculation of J from the above formula. Make your dt smaller if the balls appear to collide before they touch.

Important requirements:

1. *Draw the force vectors on both objects during the collision. This is very important to see and understand!*
2. Draw momentum vectors on the objects at all times.
3. Draw v vectors on the balls at all times.
4. Put your momentum bar charts in each frame. There should three total momentum bars. One for each sphere and one for the total momentum of the system, which is the sum of the individual momenta from each sphere.

The discussion above allows you to handle the *collision response*. You'll need to place an `#if` statement in Part 8 of your code to handle the *collision detection*. This will be discussed in class. Draw the p vector $(= mv)$ on both balls at all times.

mom.e: Start two balls (A and B) moving toward each other. One from the left and the other from the right. Keep all motion along the x-axis. Each sphere should be assigned different masses that are reflected in their size on the screen. When they collide, compute the impulse, J that moderates the collision from $\vec{J} = (1+e)m_a m_b(\vec{v}_{Ai} - \vec{v}_{Bi})/(m_a + m_b)$, where e is the coefficient of restitution, which is a number *between* 0 and 1 (say like $e = 0.7$). Also choose a $\Delta t_{collision}$, then compute $\vec{F}_{collision} = \vec{J}/\Delta t_{collision}$. Your animation should show the entire collision, from approach, to collision, to motion after the collision. Your code must explicitly compute set $\Delta t_{collision}$ and e in part 2, followed by the calculation of J from the above formula.

Important requirements:

1. *Draw the force vectors on both objects during the collision. This is very important to see and understand!*
2. Draw momentum vectors on the objects at all times.
3. Draw v vectors on the balls at all times.
4. Put your momentum bar charts in each frame. There should three total momentum bars. One for each sphere and one for the total momentum of the system, which is the sum of the individual momenta from each sphere.

The discussion above allows you to handle the *collision response*. You'll need to place an `#if` statement in Part 8 of your code to handle the *collision detection*. This will be discussed in class. Draw the p vector ($= mv$) on both balls at all times.

mom.f: Download the code `elastic_coll.pov` from `http://goo.gl/zTtuQ` . If you render this, you'll see two boxes on a surface moving toward each other and surrounded by bricks walls on both sides (at $x = \pm 10$). Call them the blocks 1 and 2. For this movie make block 1 (the left block) more massive than block 1 (the right block). See the mass statements in Part 2 of the code. Using collision detection in Part 8, program in an **elastic collision response**, making both blocks bounce off of each other. The elastic equation response equations can be found in the physics discussion above. When each block reaches the edge a brick wall, have it make a "hard collision" and bounce off of it (i.e. reverse v_x). Your movie should end just after each block has its collision with a wall, and moves noticeable away from it. Show total energy and total momentum bars in the movie at all times. **Note: Do not use any of the techniques in this movie that you may have used in mom.d or mom.e The point of this movie is to learn about and make use of the elastic collision equations. Remove all references to J (impulse), etc. in this work. The only collision response equations you can use are the ones in the very last sentence of Section 14.2**

mom.g: Rerender `mom.f` but make $m_2 > m_1$.

mom.h: Rerender `mom.f`, but make $m_1 = m_2$.

mom.i: Repeat Movie `mom.f`, but make the collision *inelastic*. Inelastic means the collision response will assign the same velocity to both blocks after the collision. Make the blob of boxes bounce off any brick wall it hits. End the movie just as the blob of boxes bounces off a wall, and moves noticeable away from it.

mom.j: Draw a large table (a box) in the middle of your screen and place two blocks of differing mass on the table's surface. Call them the "left" and "right" blocks. For this movie make the left block more massive than the right block and make the size of each block indicative of its mass. Make the leftmost block initially move toward the right and the rightward block move toward the left. Using collision detection in Part 8, program in an **elastic collision response**, making both blocks bounce off of each other. The elastic equation response equations can be found in the physics discussion above. When a block reaches the edge of the table, have it enter free fall. Draw total KE and total p bars on the screen at all times.

The table might be drawn with a statement like this in Part 10:

```
box { <-10,-10,-10>,<10,10,10> pigment {Red}}
```

As for drawing your boxes, see Section 4.3. The top of this box is at $y = 10$ and it extends between ± 10 along the x-axis. You'll need one box statement for each of the two boxes in this project. **Note: Do not use any of the techniques in this movie that you may have used in mom.d or mom.e The point of this movie is to learn about and make use of the elastic collision equations. Remove all references to J (impulse), etc. in this work. The only collision response equations you can use are the ones in the very last sentence of Section 14.2**

mom.k: Rerender `mom.h` but make the mass of the left block less than the mass of the right block (remember: make the size of each box indicative of its mass).

mom.L: Rerender `mom.h`, but make the masses of both blocks the same (remember: make the size of each box indicative of its mass).

mom.m: Repeat `mom.h`, but make the collision **inelastic**, where the two stick together after the collision. Make the left block more massive than the right. Making the blocks stick together is done by simply causing them to each have the same v (and a) after the collision. Draw total KE and total p bars on the screen at all times. Note that getting the KE and p right for this in Part 10 is a bit hard, so here are some hints:

- Declare a variable in Part 2 called `has_collided` and set it equal to `false`. This variable means "before the collision happens an indicator called "has collided" is false.
- In Part 8, be sure that this variable gets set to `true` when your collision detection `#if` statement fires.

- In Part 10, use an `#if` statement to properly handle calculating KE and p. Before the collision when `has_collided=false`, KE is the sum of the individual KEs of the blocks and p is the sum of individual momenta of the blocks. After the collision when `has_collided=true`, KE is the KE of the blob, and p is the momentum of the blob.

Note: Do not use any of the techniques in this movie that you may have used in mom.d or mom.e. The point of this movie is to learn about and make use of the inelastic collision response, where there is a "blob" of mass after the collision with a single speed. Remove all references to J (impulse), etc. in this work.

mom.n: Repeat `mom.i`, but make the left block less massive than the right.

14.4 Wrap-up Questions

1. It is critical in `mom.e` or `mom.d` that you understand the following concept, so discuss it here: No matter if the colliding objects are as different as a car and mosquito or as similar as a car and car, the collision force one exerts on the other *is always the same*, while they are in contact. This leads to an equal and opposite impulse, J experienced by both objects. **This is Newton's Third Law.** The resulting motion after the collision results from the acceleration acquired by a given body because of the collision force. And, as you know by now, $a = F/m$, so the smaller the mass, the larger the acceleration.

2. Discuss Newton's Third law.

3. Discuss "who feels what" when a tennis ball moving toward the right collides and bounces off of an SUV moving toward the left. Work with the fact that the force imparted on the SUV by the tennis is the same as the force imparted by the SUV on the tennis ball. Why does the SUV barely feel the impact, but the tennis ball goes flying off in the opposite direction? Discuss all of this. A simple numerical example would be nice.

4. Discuss the outcome of an elastic collision between mass m_1 and m_2 when $m_1 = m_2$, $m_1 > m_2$ and $m_1 < m_2$.

5. Fill out a study grid outlining the time and topics you studied for this class between Wed 2/15 and Wed 2/22. What are you thoughts/feelings on the 25/35 idea on campus?

Chapter 15

Rotational Motion

15.1 Introduction and Goals

- The goal of this project is to have you experience why and how objects can be made to rotate.
- Demonstrate that you understand the rotational variables θ, ω, and α.
- Demonstrate that you see the analogies between the kinematic variables x and θ, v and ω, and a and α.
- Demonstrate that you understand the kinematic equations for rotations.
- Demonstrate that you understand what moment of inertia.
- Demonstrate that you understand toque.
- Demonstrate that you understand the "a=F/m" for rotations, which is $\alpha = \tau/I$.

15.2 The Physics

The v-vector will now become an ω vector. ω will be changed by: Applying an angular acceleration either parallel or anti-parallel to ω. Thus far, we've discussed objects moving in straight lines, or "linear motion." Now we'll discuss "rotational motion," or how an object rotates or spins. Think of a merry-go-round, rolling wheel, or a pulley that actually turns as it guides a rope. For the rotating object, you should always be able to identify the axis about which it rotates, called the "axis of rotation," which might be through its center, but not always. Given what you know by now about straight line motion (x, y, F, etc.), much of the core concepts here can be taught by analogy. Linear motion, has three working variables: x (or y), v, and a, with units of m, m/s, and m/s^2. In rotational motion we aren't concerned with how many meters an object has moved, but how many degrees (or radians) it has rotated through, so for angular position we'll have θ, angular speed ω, and angular acceleration, α. By analogy, $x \leftrightarrow \theta$, $v \leftrightarrow \omega$, and $a \leftrightarrow \alpha$, so instead of $x = x_0 + v_0\Delta t + \frac{1}{2}a\Delta t^2$, we'll have

$\theta = \theta_0 + \omega_0\Delta t + \frac{1}{2}\alpha\Delta t^2$, and instead of $v = v_0 + a\Delta t$, we'll have $\omega = \omega_0 + \alpha\Delta t$. These are your core time-stepping equations for rotational motion. Here's an example (same numbers from week #1): A wheel has spun through $\theta = 5$ rad with an angular speed $\omega = 1$ rad/s and an acceleration of $\alpha = 0.5$ rad/s^2. How far will the wheel have spun $\Delta t = 0.1$ s later? Use the equations above to get that $\theta = 5\text{rad} + (1\text{rad/s})(0.1\text{s}) + (0.5)(0.5\text{rad/s}^2)(0.1\text{s})^2$ or $\theta = 5.1025$ rad and $\omega = 1\text{rad/s} + (0.5\text{rad/s}^2)(0.1\text{s})$ or $\omega = 1.05$ m/s. Like week #1's equations, you can compute a new θ and ω over the time step Δt, and can iteratively put $\theta \to \theta_0$ and $\omega \to \omega_0$ and use the equations again to compute the next θ and ω of the spinning object another Δt in the future. Also like week #1, be very aware of signs. θ can be positive or negative. Arbitrarily, we'll interpret a positive θ as a clockwise rotation and a negative θ as a counterclockwise rotation. With this sign convention, you can also place signs on ω and α. A positive ω means the object is rotating clockwise; a negative ω counterclockwise. If ω and α have the same sign, the object is spinning faster and faster. If ω and α have different signs, the object is spinning, but slower and slower. It may reach $\omega = 0$ in which case ω will start building up again in the same direction as α and acquire the same sign as α. The object will start rotating, faster and faster in the same direction as the original α. There is one last confusing point about the rotational world of θ, ω, α and the linear world of x, v, a. Think of the wheel on a car. A carbon atom (in the rubber) near the outer edge of the tire and one very close to the axle have the same ω, since they both rotate by the same amount in a given Δt. If they didn't the tire would warp and break apart. But the atom near the outer edge must have a linear speed (v) which is larger than the inner atom since it has a larger circle ($2\pi r$) to travel through on its way around. So although the atoms in the rubber have the same rotational speed ω, their linear speeds (v) are different. In fact, the v's scale with the distance from the axis of rotation, or, $v = r\omega$, where r is the distance from the axis of rotation. That is, if an atom is 5 cm from the axle, and the other is 10 cm from the axle, the latter has a v that is twice as large as the former. Similarly, $x = r\theta$ and $a = r\alpha$; linear distance and acceleration scale with r too. So you can describe a rotating object using the linear parameters x, v, a, but they aren't the most convenient, so we use θ, ω, and α. But each is related to the other via linear-variable $= r$(angular-variable), so they're really one in the same. To close, θ, ω, and α are the parameters that allow you to *observe* an object rotating. You'll see it rotate so far (θ) at some speed (ω). If the speed seems to be changing (speeding up or slowing down), then you can conclude that the object must have some α. **Book reading: 12.1, 12.7, 12.9**.

15.3 Projects

Project Descriptions

(Note: In these projects, "RM" stand for "rotational motion.")

RM.a: Use `rotate_skeleton.pov` at `http://goo.gl/zTtuQ` to get started. This will render a wheel on the center of the screen. Make a movie that shows this wheel spinning at some constant ω clockwise. Hint: the `rotate` line in the wheel object might look like this `rotate 180/pi*Theta` where `Theta` is your computed angular position of the wheel. Draw

the ω and α vectors on the wheel at all times using the "right hand rule" sign convention discussed in class. The tail of each vector should be at the center of the wheel, and the vectors should extend along the z axis (given that the wheel face is in the xy plane).

RM.b: Same as RM.1, but counterclockwise.

RM.c: Same as RM.2, but with $\omega > 0$ and $\alpha > 0$.

RM.d: Same as RM.2 but with $\omega > 0$ and $\alpha < 0$. Be sure we can see the wheel stop and start turing in the opposite direction from how it started.

RM.e: Same as RM.4, but make it spin along the $y-$axis, instead of the z-axis.

RM.f: Get out the code you wrote for the arctan hill, or the skateboarder on the parabola (choose one, it's your choice). Get rid of all energy bars and associated calculations. Now, change the object and code so that we see a wheel *rolling* along the surface, not just sliding on it. Show as much motion as possible, for example, up the incline, back down the incline, and onto the flat surface again. It is imperative that we see the details of your rolling wheel. You should already have the instantaneous v available in your code from previous weeks (or you can find it from `vmag=vlength(vel)`). All you have to do is recognize that $v = \omega r$ or $\omega = v/r$, and it is with this ω that you use to compute $\theta = \omega\Delta t$ to make your wheel rotate as it moves. Draw the ω and α vectors on the wheel at all times. The tail of each vector should be at the center of the wheel, and the vectors should extend along the z axis (given that the wheel face is in the xy plane). You should know where the ω vector comes from. What about α? Well, just like $\omega = v/R$, $\alpha = a/R$, so you can use:

```
#declare amag=vlength(a);
#declare Alpha=<0,0,amag/R>;
```

to compute `Alpha` which may be used to draw the α vector. This code in Part 10 can be used, instead of your sphere, to draw the wheel (assuming `Theta` and `pos` are the needed rotation amount and position of the wheel and R is the wheel's radius:

```
object
{
union
{
cylinder {<0,0,-1>,<0,0,1>,R pigment { checker Red, Blue scale <.5,1,.05> }}
cylinder {<R/2,R/2,0>,<R/2,R/2,-1.2>,R/4 pigment {Yellow}}
cylinder {<-R/2,-R/2,0>,<-R/2,-R/2,-1.2>,R/4 pigment {Yellow}}
cylinder {<0,0,2>,<0,0,-2>,0.1 pigment {Green}}
}
rotate Theta*180/pi
translate pos
}
```

Optional note: Povray always rotates objects about the point (0,0,0); this is the way its rotate function works. So to make an object look like it is rotating about a central axis,

you have to draw the object so that its axis is at (0,0,0). Once done, rotate it, then translate it out to where it needs to actually be placed. Thus the steps in the object statement above; draw, rotate, then translate.

RM.g: Go the `http://goo.gl/zTtuQ` and download the code called `kickoff.pov`. If you render this, it'll show a football and some goal posts. Choose a launch $\vec{v}$ and θ_0 that'll cause the football to go through the goal posts. Additionally, just like in field goal kicks in real football, make the football rotate in the opposite direction to which it is moving. To do this, set up a `Theta` and `Omega` variable for the football. The code renders the football at the rotation state given by the variable `Theta`.

RM.h: Get our your code from momentum work of the two blocks making an elastic collision on the high table from last week. Instead of blocks, make your code into rolling cylinders. The cylinders should roll both into and out of the collision. Draw the ω vector on both cylinders at all times.

15.4 Wrap-up Questions

1. Discuss the analogies you see between x and θ, v and ω and a and α.
2. Discuss what it means for θ to be a vector.
3. Discuss what it means for ω to be a vector.
4. Discuss the possible directions that ω and α can have. Discuss sign conventions.
5. Draw a wheel with ω and α vectors that would indicate the wheel is slowing down.
6. Draw a wheel with ω and α vectors that would indicate the wheel is speeding up.

Chapter 16

Torque, Angular Acceleration and Momentum

16.1 Introduction and Goals

The goal of this project is to have you experience how a collision between an object with linear momentum can be transferred into a rotatable object, giving it *angular momentum* (causing it to rotate).

- Demonstrate that you understand that $L = rp$, where L is angular momentum.
- Demonstrate that you understand that angular momentum is conserved.
- Demonstrate that you understand how linear momentum can be recast at angular momentum.
- Demonstrate how linear momentum can be transferred into angular momentum.

16.2 The Physics

The v-vector is now an ω vector. ω will be changed by: Applying a torque or net-torque to an object. α drives rotations, since if you have $\alpha = 0$, over successive Δt's, the α can lead to changes in ω, which together can lead to changes in θ. Here we address where α comes from. Just like $a = F/m$, here we'll have that $\alpha = \tau/I$, where τ is the torque on on object and I is the moment of inertia of the object. m is the mass of an object, or a measure of its resistance to want to change its state of motion, I is the resistance of an object to change its state of rotation (if it's not rotating, it wants to stay *not rotating*, etc.). Where mass is usually given for an object (so many kg's), I comes from simple formulas that resemble $I = cmR^2$, where m is the mass of the rotating object, R is the maximum extent of an object away from its axis of rotation, and c is a number like 1/2, 2/5, etc. Don't think of R as "radius;" an object doesn't have to be round in order to rotate. Look in your book for a chart of I's for objects rotating in various ways. Torque (τ) is like a "rotational force." From the discussion above,

α drives rotational motion, because with α, a ω will develop, which will develop a θ. Since $\alpha = \tau/I$, you must have a τ in order to get an α. So where do torques come from? Forces. You must ultimately apply a force to an object to get it to rotate, but it matters 1) where you apply the force 2) at what angle you apply the force. This is all seen in the equation for torque, $\tau = rF\sin\phi$, where F is the force you apply to the object you wish to rotate, r is the length of a line that directly connects the axis of rotation and the spot where the force is applied. ϕ is the angle between the direction the force is applied and the axis-force connector line. This torque equation can be wholly understood by thinking of how one opens a door. If you push near the hinges the door won't open since $r \approx 0$, meaning $\tau \approx 0$ meaing $\alpha \approx 0$. If $\alpha \approx 0$ and the door is not already rotating then $\omega_0 = 0$ and $\omega = \omega_0 + \alpha\Delta t$ will never give any appreciable ω, no matter how long you wait (Δt), since $\alpha \approx 0$. Lastly, if $\omega \approx 0$ and $\alpha = 0$, then θ will always equal to θ_0, meaning the door will remain in the same rotational position. In other words "the door won't open." You can also push on the edge of the door, farthest from the hinges, maximizing r, but if you push directly on the narrow edge of the door (toward the hinges), $\phi = 180°$, once again, giving $\tau = 0$ (since $\sin 180° = 0$). This also gives $\alpha = 0$, like above, meaning that getting the door to swing (or allowing it to acquire some ω or $\theta = \theta_0$) will simply never happen. The best place to push on a door is farthest from the hinges, maximizing r, and perpendicular to the door, making $\phi = 90°$. This will give some non-zero value of τ, which will give a non-zero value for $\alpha(= \tau/I)$. With a non-zero α, an ω of the door will start to develop as θ will begin to become different that θ_0: the door will rotate. So certainly θ, ω and α track the observable rotation of an object, driven by α. But α must come from somewhere, and it comes from a torque, which ultimately comes from a force applied to an object (at some distance at some angle). I factors in to how hard it is to get the door to swing. A heavy, solid wooden door (front door of your house) is harder to open than a similarly sized light hollow door (on your bathroom) because m is larger and $I \sim m$. Thus for a given τ (your hand), α would be smaller since $\alpha = \tau/I$. Now say you had two doors that had the same mass, but one was two times wider than the other. Since $I \sim R^2$, where R is the maximum extent of the door, the door that is twice as wide would be four times harder to swing for a given torque applied, for the same reason. $\alpha = \tau/I$. The wider door, with the larger I, gives a smaller α. So α's, which drive all rotations, come from torques, just like a's, which drive all motion, come from forces. **Book reading: 12.5, 12.6, 12.11, and problem 12.93**.

16.3 Projects

torque.a: A rod (like a yardstick) of length L is attached at its top by a frictionless nail at its topmost edge. Make an animation that shows the subsequent (rotational) motion of the object if it were let go from $\omega_0 = 0$. Here is the code that will draw the rod in part 10 (a red rod with a yellow nail).

```
object {
union
{
cylinder {<0,0,2>,<0,0,-2>, 0.1 pigment {Red}}
cylinder {<0,0,0>,<L,0,0>,0.25 pigment {Yellow }}
}
```

```
rotate 180/pi*Theta
}
}
```

Your Δt should be small for this work, at 0.01. You should plan on rendering about 1000 frames, so we can see several swings of the rod. Draw the ω and α vectors through the axis of rotation at all times. Stitch this movie together at no less than 60 frames per second.

You should start this work with `rotate_skeleton.pov`, found online. In part 2 of your code you should define the rod's mass, length, and moment of inertia (from a suitable chart). In part 5 of the code, you should define the instantaneous τ and $\alpha = \tau/I$ on the rod, so the physics equations in part 6 can adjust θ and ω of the rod appropriately. The instantaneous τ will be a function of θ, the rod's instantaneous orientation. The θ computed by the physics equations is the θ by which the rod should be rotated, as per the `Theta` variable in the part 10 drawing code given above.

torque.b: Repeat `torque.a` but with a non-zero ω_0 that sends the rod in such a direction that opposes that demanded by gravity. In other words, we want to see the rod rotate a bit against gravity, stop, then turn back around again, and start falling as per gravity.

torque.c: Render the Atwood machine again, but this time with a real pulley that has a momentum of inertia, I. We should see the pulley rotating, in unison with the rising and falling masses.

torque.d: Place two children (spheres) on a seesaw and show the subsequent motion. Center the the seesaw over the pivot, so the seesaw's weight does not produce a torque. The net torque should only come from the childrens' weight and their positions relative to the pivot point. Explicitly show in your code how the torque leads to angular acceleration. Be sure you run this movie long enough so we can really take in the full motion, including any turn-arounds or rocking. Your seesaw/pivot/children object might look like this:

```
object{ union
{
//the main rod
cylinder {<-15,0,0>,<15,0,0>,0.1 pigment {Red}}
//thet axis of rotation
cylinder {<0,0,-1>,<0,0,1>,0.2 pigment {Yellow}}
//the two weights on the rod (the children)
sphere {<s1r,0,0>,radius1 pigment {Blue}}
sphere {<s2r,0,0>,radius2 pigment {Green}}
//rotate the value of Theta given (in radians)
rotate <0,0,Theta*180/pi>
}
}
```

torque.e: A block of mass m is moves with speed v toward a vertical rod that is hinged at the very top. The rod has a length of D between its end and the hinged point, and a mass M. The block collides with the end of the rod and sticks to it, causing the rod to begin rotating. Your movie should show the block moving toward the rod, then the motion after the collision, when the block/rod combination begins swinging. Note that because

of the hinge, p is not conserved by the rod+block system, but L (angular momentum) is conserved. Here's how it works.

The block, while moving, has a momentum $p = mv$. When it sticks to the rod, it brings an angular momentum to the rod of (using $L = rmv$)

$$L_{block} = Dp = Dmv, \tag{16.1}$$

which is a direct use of $L = rp$, where r is the length of the rod and mv is the momentum of the block. That is, the block, due to it linear momentum (p) also has *angular momentum* with respect to the hinge-point of the rod.

Since L is conserved, it means that $L_{block} \rightarrow L_{rod}$ during the collision. Since we know $L = I\omega$, then for the rod,

$$L_{rod} = I\omega_{rod} \tag{16.2}$$

where I is the moment of inertia of the rod+block combination and ω is the angular speed of the rod+block combination after the collision. Thus

$$I_{rod+block}\omega = Dmv \tag{16.3}$$

or

$$\omega = \frac{Dmv}{I_{rod+block}} \tag{16.4}$$

giving you the initial angular speed of the rod+block after the collision.

From Wikipedia (http://en.wikipedia.org/wiki/List_of_moments_of_inertia), use $I_{rod} = \frac{1}{3}MD^2$. The contribution to I from the block is mD^2, so $I_{rod+block} = \frac{1}{3}MD^2 + mD^2$. Knowing the initial ω of the rod+block combination or ω_0 should allow you to launch the rotational motion of the rod+block combination.

To animate the system after the collision, you'll also need α, the angular acceleration of the rod+block combination. If the rod+block is at some angle θ (where $\theta = 0$ is when the rod is vertical) then the net torque on it is

$$\tau = \frac{D}{2}Mg\sin\theta + Dmg\sin\theta \tag{16.5}$$

where the first term is torque due to the rod's weight and the second is due to the block's weight, as it sticks on the rod. The angular acceleration of the rod is $\tau/I_{rod+block}$ or

$$\alpha = \frac{\frac{D}{2}Mg\sin\theta + Dmg\sin\theta}{I_{rod+block}}. \tag{16.6}$$

So you have ω_0 and α of the rod+block system, which is all you need to complete the movie.

What about drawing the block and rod? Assuming the block is at coordinates `bx,by` with a side length of 1 and a rod of length D, this will handle drawing them separately before the collision:

```
cylinder {<0,0,0>,<0,D,0>,0.25 pigment {Gold} }
box { <bx-0.5,by-0.5,-0.5>,<bx+0.5,by+0.5,0.5> pigment {Red} }
```

and stuck together, rotated by an angle θ_0 (`Theta0`) after the collision.

```
object {
     union {
          cylinder {<0,0,0>,<0,-D,0>,0.25 pigment {Gold} }
          box { <-0.5,-D-0.5,-0.5>,<0.5,-D+0.5,0.5> pigment {Red}}
               }
       rotate <0,0,Theta0*180/pi>
       translate <0,D,0>
       }
```

Povray note: do you see in the after collision drawing how the rod+block combination is drawn so that the hinge point of the rod is at $(0,0)$ as needed by the Povray rotate logic?

You can use some kind of `#if` statement to handle and track the "before" and "after" collision scenarios. Hint: Think of the `has_collided` logic used in the collision movies.

torque.f: The Indiana Jones Door. Let's animate the rotating tomb door from the Indiana Jones clip shown in class. Make Indy (a block of mass m) move from end to end across a big cylindrical door. For this, getting a block to move along the x axis will be fine. Do the normal stuff with `pos0` and `vel0` in Part 2. At the instantaneous position of the mass, your movie should compute the torque on a big door. From this torque will come α, then ω, then θ. Of course θ is the amount of rotate the door. The moment of inertia of the door is $I_{door} = mr^2/4$, where m and r are the mass and radius of the door. If Indy is at position `pos` and the door is to be rotated through an angle `Theta`, then this group in Part 10 will draw the door/Indy graphics

```
 object{ union
{
cylinder {<0,-0.5,0>,<0,0.5,0>,L/2 pigment {Red}}
box {pos-1,pos+1  pigment { Orange} }
rotate <0,0,Theta*180/pi>
}
}
```

assuming L is the diameter of the door, and it all needs to be rotated by an angle `Theta`. Draw the ω and α vectors on the door at all times. Draw the Indy's weight at all times. Your animation must show the door reversing it's ω due to Indy's position. This might require some if statements and adjustments of Indy's acceleration in Part 5 to cause him to tip the door accordingly. You have to keep dt small (0.01 maximum) and use several hundred frames to get this to work out.

Chapter 17

Final Project

Project Theme

A "Rube Goldberg Machine" is a machine that does a simple task in the most complicated possible manner. Search Youtube for "rube goldberg" and watch a few. In this final project, your assignment is to use what you know about creating physics-driven animations to create an animated Rube Goldberg machine. The simple task your animation must complete is to turn on a lightbulb.

Note that this project has *three deadlines* and is worth 50 points to the "computer work" portion of your grade.

Project Introduction

At this point, you know quite a bit about basic physics and computer animation. In particular, you know, via the laws of physics, how to animate:

1. 1D motion
2. 2D motion (projectile motion)
3. Basic forces and Newton's Laws
4. Newton's Law machines (ropes, pulleys, springs, curved paths, etc.)
5. Energy and work due to friction
6. Collisions (elastic and inelastic)
7. Rotations (kinematics and torque)

With your new set of skills, this sheet announces a final animation project. Your job is create an animation of a **Rube Goldberg machine** (see Wikipedia) that uses a series of physics interactions (above) to do something as simple as (for this project) turning on a lightbulb. Here is a sample:

From left to right on your screen: a ball is compressed against a spring. It is released and travels toward the right on a level surface. It collides elastically with a second ball. This second ball rolls up an inclined surface, going over its edge, subsequently executing projectile motion. It takes a couple of bounces until it collides (inelastically) with one mass of an Atwood Machine. The ball causes the Atwood Machine to become unbalanced, sending the mass/ball blob down toward the ground, where it lands on a push-button switch and turns on a lightbulb.

Hints, guidelines, and requirements

1. Your movie must use *all* of the physics interactions listed above.
2. Your movie must run at least 20 seconds in duration.
3. Your movie must show the continuous "life" of some object(s) on your screen concluding with a lightbulb switch being pressed and a lightbulb turning on.
4. The v and a vectors must be drawn on your object at all times.
5. Your movie must end with a lightbulb being turned on after the series of physics interactions. Turning on a lightbulb doesn't need to be anything fancy. Making a gray sphere suddenly turn bright yellow will be sufficient.
6. You may work in groups of up to 3 people.
7. You will show your movie in front of the class and discuss it and take questions about it during the last week of the quarter.
8. It is easiest to divide your screen up into several sections and apply accelerations or speed changes to the object(s) as needed with `#if` statements that check the x and/or y coordinates of the object(s).

Due Dates - three of them

Deadline #1: Monday Oct 31st (at the beginning of class) (5 points): Two section summary of your project, as follows:

- Section I: One paragraph written summary of your movie's story line. Tell me what your object is, and what is going to happen to it. A sketch (by hand) is required.
- Section II: List the names of all people in your group (up to 3 people). I need these names so I can schedule presentation times during the last week of the quarter.

Deadline #2: Monday Nov 7th (at the beginning of class) (5 points): A single page, two section progress report of your project, as follows:

- A Youtube link to a rough animation draft of your movie. This will not be your final movie and your movie will likely be incomplete, but it must show significant progress in implementing the story line you have proposed.
- The names of the people in your group, so I can give everyone credit.

Deadline #3: Week of Nov 28th ($\approx$ 80 points): In class (or lab), a 6 minute presentation of your work, consisting of:

1. Show us your movie.
2. Point out the different areas of physics used in the movie.
3. Discuss any results or technical hurdles that were difficult for you, and how you overcame it.
4. Upload your final movie to your Youtube account.
5. Turn in a CD-ROM (due at the time of your presentation) with your final, playable movie and complete Povray code burned onto it.

Chapter 18

Future Plans

We believe the software used in this work was unfortunately the *wrong approach*. For a classroom-based programming endeavor, where the programming is *not meant to be the sole course focus*, there must be minimal fussing with software installs, versions, downloads, and configuration. Despite popular belief, the typical freshman college-student is not very computer savvy. Organizing work with folders, following software-installation directions, using logic to write a few lines of code, stitching together image files and maintaining a Youtube account are all rather difficult for most students, and well beyond what they were expecting in a freshman physics course. Although we still remain convinced of the graphics approach to freshman physics (we cannot ignore the FMCE learning gains we consistently see when using this approach), such "computer issues" quickly lead to a negative outlook for the course.

But this isn't all of their fault. The Povray-based system used in this work was troublesome, but we feel as if it was the best choice, given what's available. We wanted easy programming access to beautiful looking graphics. Povray generates some of the best looking graphics one can find, through its relatively simple scripting language, but it has not been maintained since the early 1990s. The scripting language, despite being "Turing complete" is is very contrived and awkward. The OSX version, called MegaPOV has a few strange bugs on both older PPC machines and on the latest Intel-based machines. We now are convinced that for such a classroom endeavor, software needs to be 100% web-based.

A web-based system would be an ideal substitute as a "lighter" programming environment. It would require only that the student point their browser to a URL in order to do their work. As for animation, the HTML5 canvas and WebGL are of great promise. With performance improvements in JavaScript (i.e. Google's V8 engine), such a web-platform is becoming a viable alternative. JavaScript, however, is a strange language, and we don't think students will like using it any more than Povray's scripting language.

We these considerations, we have begun work on such a system that includes the following features:

1. Integrated editor and output-screen all in a single browser window, with a single "run" button.

2. Zero-framework, meaning there are no include files or libraries to import. Typing a single

line into the editor such as

```
sphere(<0,0,0>,5,"red")
```

will draw a red sphere with a radius of 5 pixels at $x = 0, y = 0, z = 0$.

3. A JavaScript preprocessor, that rids the "apparent language" of seeming unnecessary parentheses, semi-colons and curly braces, in favor of much less punctuation and `do-end` pairs for grouping code. We find the Lua (www.lua.org) language to be the ideal language, so are writing a preprocessor to morph JavaScript into Lua.

4. WebGL animation for a compelling look to the graphics.

5. Support for a native vector data type (as part of the preprocessor), so a line like `r=<5,1,3>` is valid.

A early prototype of this work is available as an open-source package at:

`https://github.com/tbensky/physgl`