Exploring physics with computer animation and PhysGL

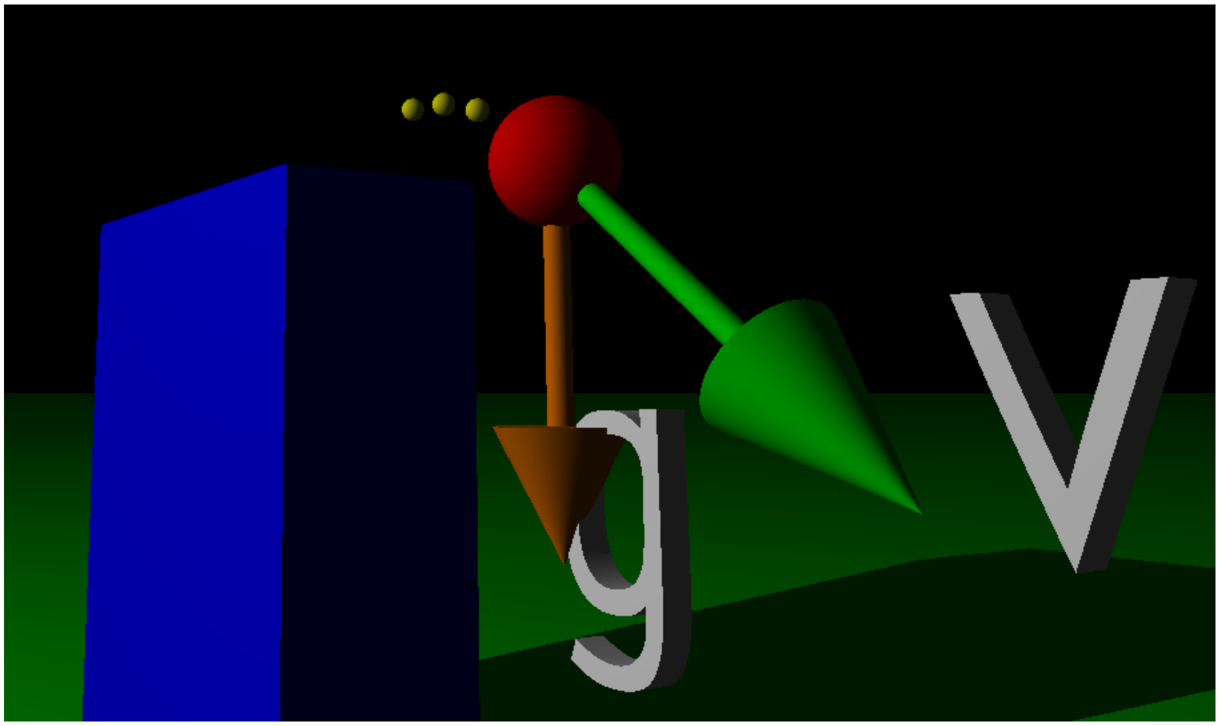


Figure 1: A virtual ball being launched off of a table. The gravity and velocity vectors are shown.

Dr. T.J. Bensky
Department of Physics
Cal Poly, San Luis Obispo
San Luis Obispo, CA 93407
tbensky@calpoly.edu

Spring 2014 edition
Updated: June 14, 2014

Contents

Acknowledgments

The author wishes to thank J. Christiansen and M. Moelter for their many helpful suggestions about this text during the Spring of 2014.

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 which 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 on the screen) 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 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 the laws of physics to produce computer animations.* It's a 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 mental 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. But it is a 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, all in real time. The emphasis in this text will be to observe the wildly dynamic nature of velocity, force, and acceleration vectors, and how they relate to one another as an object moves under their influences. This alone is a more compelling way of learning physics than viewing static pictures. Second, a computer animation will refuse to work (properly) if the physics concepts are not applied or applied incorrectly. 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 physical theory of motion.

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 is this theory? Where does it come from? It'll come from a variety of concepts, but will always be 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" in these equations are used to denote the position (x) and speed (v_x) of an object along the x (or horizontal) axis. There are two other equations that look similar to 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 into 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 in making computer animations is figuring out what acceleration ($\vec{a}$) 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 (x, y) coordinates 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 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 (predictive) laws of physics.

What about the v_{0x} and v_{0y}? They are the two components of the the object's velocity, $\vec{v}$. 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 do 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 this physics class is about *motion*, you typically *don't actually see* very much *MOTION!* It seems odd then that all of the connection points you'd like to make about physics *must* be constrained to the 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 does "stroboscopic" even mean?

The trouble with learning physics via just paper and pencil 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 as a real clock ticks on the wall. That is, sequential frames on a computer screen can illustrate the passage of real time. This is 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 in 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 tells the story of motion that you simply must see to appreciate. No drawing or "hand waving" 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 $\vec{v}$-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: $\vec{v}$ for velocity, $\vec{a}$ for acceleration, $\vec{F}$ for a force, $\vec{T}$ for tension, $\vec{\tau}$ for torque, $\vec{x}$ (or $\vec{r}$) for position. The question is, 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 $\vec{v}$-vector allows you to make this prediction. (For the object in the figure it will generally be up and to the right a small Δt in the future). In other words, the $\vec{v}$-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 $\vec{p}$ is just the product of mass and the $\vec{v}$-vector.)

As you create computer animations based on physics, you'll always be asked to render the $\vec{v}$-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

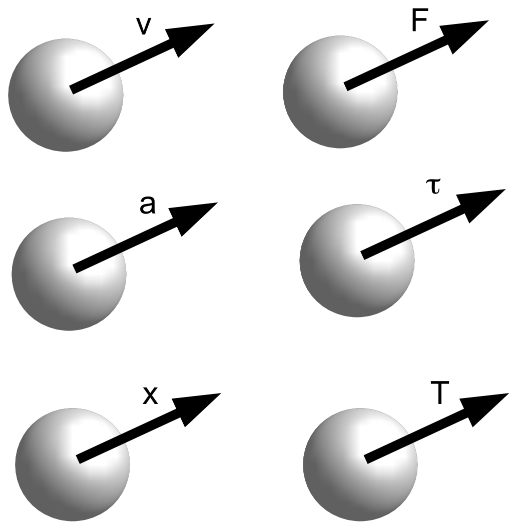


Figure 1.1: Six objects with different vectors sticking out of each.

> *If you know the $\vec{v}$-vector, then you can make an good estimate 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 which 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 $\vec{v}$-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 $\vec{v}$-vector of an object change, either in direction or length, or both.*

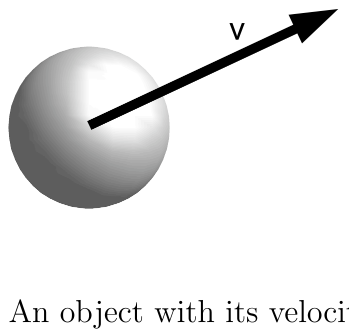


Figure 1.2: An object with its velocity vector.

Lastly, why is understanding how a $\vec{v}$-vector is manipulated so important?

- Because basic physics is about understanding how objects move. A key element in this understanding is 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 $\vec{v}$-vector .
- Changing the *magnitude (or length)* of the $\vec{v}$-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 $\vec{v}$-vector changes the direction in which the object will move over a small time interval into the future. The $\vec{v}$-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 $\vec{v}$-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 as 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 is 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 students will move on to a spreadsheet, which is OK, but most scientists will find some specialized 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 is 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 gathering tips or full solutions off of the Internet and just "following along" (if 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 also feel a bit bad, because we need to change our instructional tools.

An entire field of research called "physics education," has existed for 30 or so years now, and is fairly consumed at studying and enhancing the teaching and learning of physics using "pencil

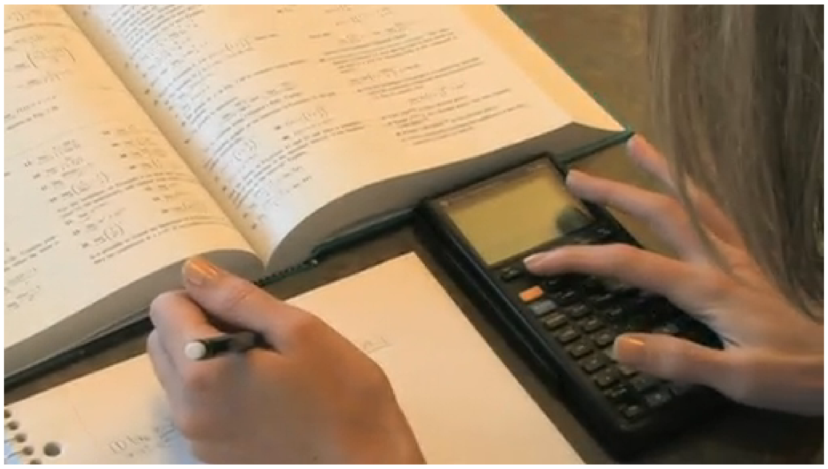
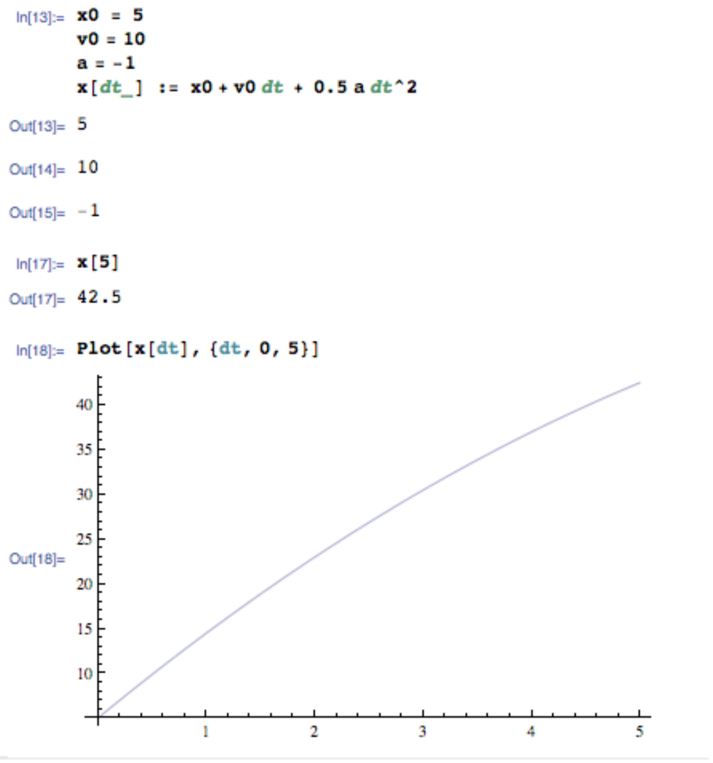


Figure 1.3: A student using a calculator to work on problems from a textbook. Scientists don't really use calculators anymore.



```
In[13]:= x0 = 5
         v0 = 10
         a = -1
         x[dt_] := x0 + v0 dt + 0.5 a dt^2

Out[13]= 5

Out[14]= 10

Out[15]= -1

In[17]:= x[5]

Out[17]= 42.5

In[18]:= Plot[x[dt], {dt, 0, 5}]

Out[18]=
```

Figure 1.4: A physics problem being solved using Mathematica.

and paper." This field has given us things like "free body diagrams," color in textbooks, "context 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 *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 point 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 about 40 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 when "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 does 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 solutions book found on the web is tempting to consult. The problems offer little to discover, explore or disseminate.

Because the computer is such a creative tool, the 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 that adheres to 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 generated 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.*

bng

Chapter 2

A taste of using computer for physics

This chapter is meant to give you a feeling for how computers can be useful in studying a subject like physics, which tends to require a lot of calculations. It turns out that computers are *really good* at doing two types of calculations, which are "what-if" calculations and calculations that tend to be rather repetitive. You'll also be introduced to a free, web-based commutation environment called "PhysGL," that we'll be using to do our work.

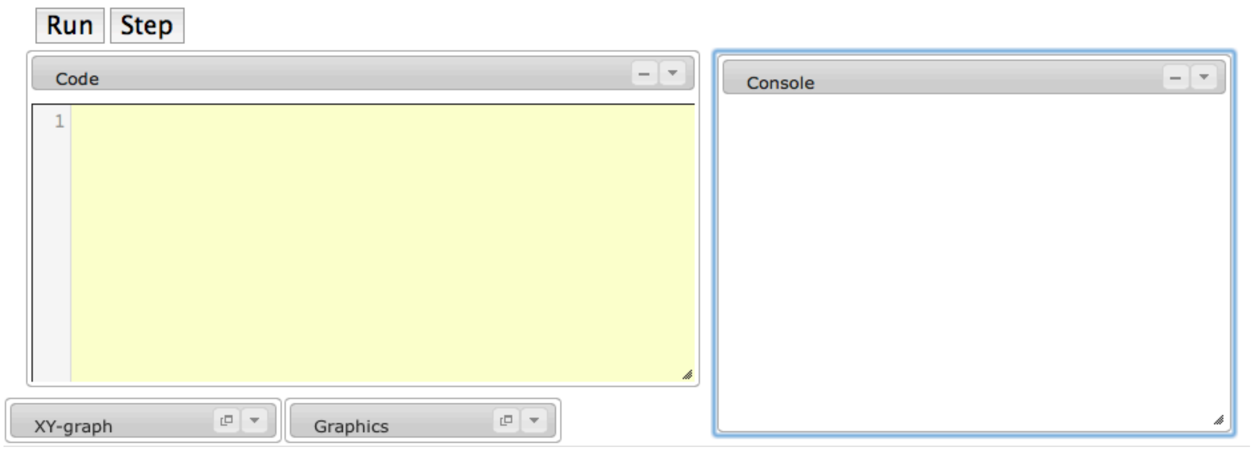
2.1 PhysGL: Easy computer graphics in the cloud

For this work, we'll be using a free service called `PhysGL.` The name is a mix of "physics" and "OpenGL," one of the leading graphics frameworks.

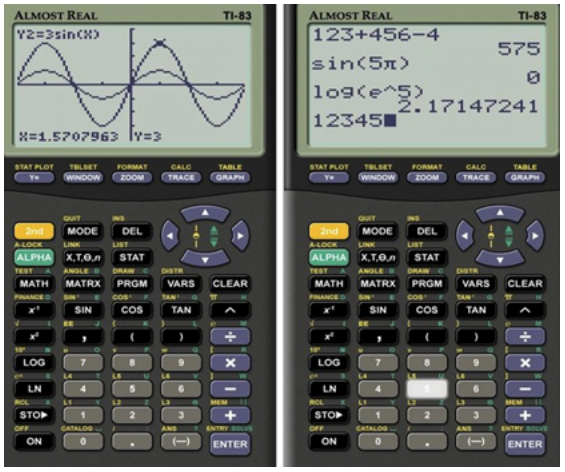
2.1.1 PhysGL

To get started, go to `physgl.org` and make an account for yourself. A backup server is also in use at `ocean.physics.calpoly.edu/physgl`. When using PhysGL, we recommend that you download and use the Chrome web-browser for best results.

When you are ready, click the `New project` link. This will open up the main work page. Give your project a name, and arrange the windows as needed. You probably want the "Code" window to be nice and large and near the upper left. Here is a sample PhysGL workspace:



In general, there are four windows available to you, the Code, XY-Graph, Graphics, and Console windows. One reason for using this software is to expose you to a more enriching way of using "a machine" for various mathematical/visualization work. You are probably used to using a calculator, like the popular TI-83, as shown here.



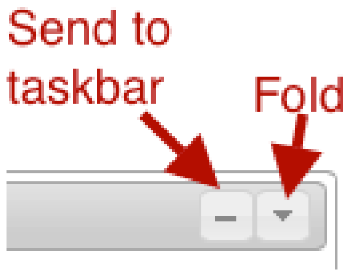
These show all output (graphics, numbers, etc.) on the same screen. PhysGL organizes output a bit better, by putting each into its own screen, so you can see them all simultaneous (if needed). Computers tend to be easier to use than calculators, because of their keyboard, bigger screen, and ability to easier share and present your results.

Just as on your calculator, you type expressions or commands into the text-screen, with PhysGL, your instructions to the computer will always be typed into the "Code" window. While text output appears on the same screen with your calculator, it will always appear in the "Console" window with PhysGL. Graphs will appear in the "XY-graph" window, and the animations (which are hard for your calculator to do) appear in the "Graphics" window.

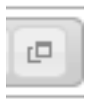
To save a project, put a name in the text box and click the "Save" button near the top of the Code window. If your code is running, you will have to click `Stop` before the "Save" button will appear again.

2.1.2 Controlling the Windows

The windows you see when using PhysGL behave similarly to windows you are used to using when you use Microsoft Windows or Apple's OSX. The upper right corner of each window has two controls as shown here:



The horizontal line will send a window down to the task bar at the bottom of the screen. The small arrow will fold up the window. If folded or in the task bar, the control that looks like this



will unfold the window again, or move it from the task bar, and back into the main work area. The four windows can be resized by grabbing (clicking and grabbing) the lower right corner, or either the right or bottom edge of the window. So although PhysGL runs in your browser (for convenience), you have the "usual" controls over the layout of the four windows.

2.2 What to turn in

Start a Word document, and carefully answer and assess all things asked by the ♦ symbols you'll see in the text below.

2.3 What-if Calculations

What-if calculations are those where you might be using some kind of formula to investigate a situation. We do this in everyday situations, for example involving money. If we have $10 on us, we might go to a restaurant and figure out just what we can buy for this amount of money. In other words, we might get more food if we buy a burrito and side-salad, as opposed to a baked-potato, chicken-salad and large soda. Here we are asking "what-if" scenarios using the mental formula Total $= \Sigma c_i$, where c_i is the cost of the i^{th} item.

Computers are also very good at doing repetitive calculations, since they don't get bored. We could easily have a computer display the squares of the numbers from 1 to 1000 very easily, where we might decide to quit after 20 or so.

In this chapter, we'll look at these two types of calculations, as a way of introducing you to the software you'll be using throughout this book, and to convince you of the utility of computers in this regard.

The distance an objects travels, d, as a function of time, t, is given by the expression

$$x = 1.5 + At^2 + B\sqrt{4t} + C\cos(\pi t), \tag{2.1}$$

where A, B and C are specific constants that define properties of its motion. Let's do some "what-if" calculations on this object's motion using PhysGL.

To begin, point your browser to `Physgl.org`, and arrange the windows so that the `Code` and `Console` windows are right next to each other. Type the following code into the `Code` box:

```
A=1
B=0.5
C=7.3
t=5
d=1.5 + A*t^2 + B*sqrt(4*t) + C*cos(Pi*t)
writeln("The distance traveled is ",d)
```

When done, click the `Run` button. You should see the number `21.43` appear in the console window. This is the position of the object at time `t=5`. The computer knows how to use the variables you defined and the lengthy expression for distance, to compute the distance.

> ⧫ **Turn-In Ch. 2#2.1:** Arrange your Code and Console windows to be next to one another. Take a screenshot of them, and paste it into a Word document. You can snip a portion of the screen into the clipboard on the Mac by 1) pressing [Shift]-[Control]-[Command]-[4] simultaneously, 2) seeing your cursor change to a cross-hair 3) dragging a rectangle over the portion of the screen you wish to snip. When done, you'll hear a camera sound. The screen portion will be in the clipboard, ready to be pasted into a Word document using `Edit`$\rightarrow$`Paste`.

Suppose we wanted to know how long it would travel in 17.43 seconds, or "What if the object traveled for 17.43 seconds, instead of 5 seconds. To get our answer, simply change the `t=5` to `t=17.43` and click the `Run` button again.

Suppose now, the dynamics of the motion changes so that $A = 4.32$ instead of 1. Modify your `Physgl` code to determine how far the object would travel in 17.43 seconds now.

> ⧫ **Turn-In Ch. 2#2.2:** Report these two answers. Verify both using your calculator. Have a partner in your group time how long it takes you to do it on your calculator.

Reflect for a minute now on what you've been doing. How long would it take you to do these (simple) calculations by hand? What about on your calculator?

Lastly, to jazz things up a bit, let's make a graphical output of our result. Minimize the `Console` window and bring the `Graphics` window next to your code window. Key in this code

```
A=1
B=0.5
C=7.3
t=5
d=1.5 + A*t^2 + B*sqrt(4*t) + C*cos(Pi*t)
draw_sphere(<0,0,0>,d,"yellow")
```

♦ **Turn-In Ch. 2#2.3:** Include a screenshot of the code and graphics window in your Word document.

This will draw a yellow sphere whose radius is the distance traveled.

♦ **Turn-In Ch. 2#2.4:** Answer these questions:

1. How is the $\sqrt{\ }$ operation done on the computer?
2. What symbol is used to raise a quantity to some power, as in t^2?
3. How does the `draw_sphere` function work? What does the `<0,0,0>` mean?
4. Why do we need to type `4*t` and not just `4t`? What does the `*` do in the mathematical expression?
5. When using a trigonometric function like `sin` or `cos`, are the arguments to be in radians or degrees? Is this different, or the same as, your calculator?
6. How do you represent π on your calculator? How do you represent it using PhysGL?

As you'll see in the next section, `Physgl` is tailored to make producing computer graphics very easy.

2.4 A motion diagram

When studying motion, most physics books introduce you to the idea of a *motion diagram* in order to illustrate an object's motion. Such a diagram might resemble the following, showing three gray balls, each moving horizontally, from left to right.

Note the direction of the acceleration in each case, and that the arrow sticking out of each ball is its velocity. The top ball has an acceleration of zero. The middle has an acceleration toward the right and the bottom an acceleration toward the left. Each ball is drawn several times to represent its position and velocity at different times.

Figures like this show the position of an object, as viewed at equally spaced time intervals. Features to recognize from such a motion diagram are as follows. In the top, where $a = 0$, the ball is evenly spaced and its velocity vector is always a constant length. In the middle, the ball gets farther away from its previous position, with an ever-increasing velocity vector. In the bottom, the ball is always closer to its previous position, and its velocity vector is always shrinking (and even disappears as v becomes zero).

Let's begin our study of physics-on-the-computer by creating such motion diagrams ourselves.

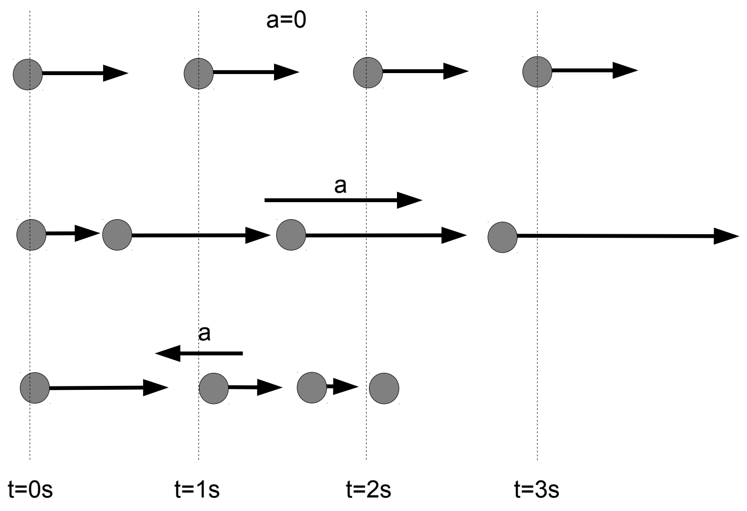


Figure 2.1: Motion diagram for a ball. Top: Initial speed to the right with no acceleration. Middle: Initial speed to the right with a rightward acceleration. Bottom: Initial speed to the right with a leftward acceleration.

2.5 Our own motion diagram

Let's begin by creating a motion diagram using computer graphics with PhysGL as follows. Suppose a ball is at $x_0 = 0$, has an acceleration $a = 0$, and an initial speed of $v = 4$ m/s toward the right. Using $x = x_0 + v_0\Delta t + \frac{1}{2}a\Delta t^2$, let's compute it's position starting at $t = 0$, then at $\Delta t = 5$ second intervals. At $t = 0$, the ball is at $x = 0$. At $t = 5$ s, the ball is at $x = 0 + 4$ m/s(5 s) $= 20$ m. At $t = 10$ the ball is at $x = 20 + 4$ m/s(5 s) $= 40$ m. Continuing the calculation, we'll get 60 m, then 80 m. So we have a ball that is at the vector positions $< x, y, z > = < 0, 0, 0 >$, $< 20, 0, 0 >$, $< 40, 0, 0 >$, $< 60, 0, 0 >$, then $< 80, 0, 0 >$.

To plot these positions, we can use the `draw_sphere` function in PhysGL, which will draw a sphere on the screen, given the center position for the sphere, its radius, and color. Here's PhysGL code that will draw the spheres at the positions we computed above.

```
draw_sphere(<0,0,0>,5,"red")
draw_sphere(<20,0,0>,5,"red")
draw_sphere(<40,0,0>,5,"red")
draw_sphere(<60,0,0>,5,"red")
draw_sphere(<80,0,0>,5,"red")
```

♦ **Turn-In Ch. 2#6.5:** Put a screenshot of your code and graphics windows in your Word document.

♦ **Turn-In Ch. 2#6.6:** Fully explain what the `<80,0,0>` stands for. What does it mean as a unit? And what do the each of the three numbers means by themselves?

Notice the spheres (balls) are all red with a radius of 5 pixels. What about the velocity vectors? Those can be drawn using the `draw_vector` function. To draw a vector you need two bits of information. The first is where the tail of the vector should be planted, and the other is the length of the vector itself. To draw the $\vec{v}$-vector on the first sphere, we'd add the line `draw_vector(<0,0,0>,<4,0,0>,"yellow")` to the code above. This will draw a vector of length 4 (that's our speed) in the x-direction, $< 4, 0, 0 >$ (the $\vec{v}$-vector) with its tail planted at the position $< 0, 0, 0 >$. To draw the $\vec{v}$-vector on the next sphere, we'd use the line `draw_vector(<20,0,0>,<4,0,0>,"yellow")`. Notice the tail position is the first vector given to `draw_vector`, so this will change from position to position.

♦ **Turn-In Ch. 2#6.7:** Explain how the `draw_vector` function works.

Finish the code up now, so a $\vec{v}$-vector will appear on all of your balls. When done, then congratulations! You've just finished your first computer graphics motion diagram!

♦ **Turn-In Ch. 2#6.8:** Include a screenshot of your motion diagram.

2.6 More from the computer

The work involved above was a bit tedious. Calculating all positions ourselves, then writing lines for each position and $\vec{v}$-vector was a lot of work. Further, suppose that the velocity of the ball changes. We'd have to change the entire program to reflect this, as all positions and vector-lengths will be wrong. You may know that computers are supposed to be good at doing (repetitive) calculations, and they are. Using something called a `while`-loop, let's see if we can automate our motion diagram a bit more, by having the computer do more of the work.

2.6.1 The while-loop

A `while`-loop is a construct in programming that is used for counting. For example, if you run this code in PhysGL,

```
i = 0
while i < 100 do
     writeln(i)
     i=i+1
end
```

you'll see the numbers from 1 to 100 appear in the text (or console) window. You can count by fives too, like this

```
i = 0
while i < 100 do
     writeln(i)
     i=i+5
end
```

and print the square of the number right next to it, like this

```
i = 0
while i < 100 do
     writeln(i,",",i^2)
     i=i+1
end
```

Notice the counting nature of the `while`-loop, and that lines affected by this counting are all contained between the `do` and `end` words in the `while`-loop structure.

> ⧫ **Turn-In Ch. 2#6.9:** Include a screenshot of your console window of some variation of the while-loops discussed above, that you invented yourself.

2.6.2 A while-loop driven motion diagram

Now back to our physics, and the motion diagram. Recall that the core quantities here are the x-positions of the ball, which are at $0, 20, 40, 60$, and 80 m. So, starting at 0, and counting to 80 in steps of 20 will deliver the proper positions. Try typing this code into PhysGL:

```
x=0
while x < 80 do
     draw_sphere(<x,0,0>,5,"red")
     x=x+20
end
```

If you do, you'll see the balls on the screen, using only a fraction of the lines required before (and tedium). Suppose the velocity changed from 4 to 5, what would change? We'd need positions that started at 0 and increased in increments of 25 (if $\Delta t = 5$). Thus the code could be changed to

```
x=0
while x < 80 do
     draw_sphere(<x,0,0>,5,"red")
     x=x+25
end
```

What about the $\vec{v}$-vectors? Returning to the case where $v_x = 4$, what if we placed a `draw_vector` function into the `for`-loop as well, right after the `draw_sphere` function, like this?

```
x=0
while x < 80 do
     draw_sphere(<x,0,0>,5,"red")
     draw_vector(<x,0,0>,<4,0,0>,"yellow")
     x=x+20
end
```

Notice here the tail of the vector is always planted at the center position of the sphere, and the $\vec{v}$-vector itself, $< 4, 0, 0 >$ is the same as in the previous section.

♦ **Turn-In Ch. 2#6.10:** Include a screenshot of this last motion diagram.

2.7 Even more from the computer

Let's generalize the creation of the motion diagram so we can have it generated for any position, velocity, and acceleration we like. Consider the following code.

```
r = <0,0,0>
v = <4,0,0>
a = <0,0,0>
dt = 5
t=0
while t < 80 do
        draw_sphere(r,5,"red")
        draw_vector(r,v,"yellow")
        r = r + v*dt + 0.5*a*dt^2
        v = v + a*dt
        t=t+dt
end
```

Here we see our choices for $\vec{r}$, $\vec{v}$ and $\vec{a}$ in the first 3 lines of the code. Our choice for Δt (`dt`) follows. Next we see that the `for`-loop has changed to one that counts from $t = 0$ to $t = 80$ in steps of the Δt that we chose.

♦ **Turn-In Ch. 2#6.11:** Run this code several times, generating a few different motion diagrams. Include Code/Graphics screen-shot pairs in your document of the following:

- Try starting the ball at $r_x = -50$, so you can fill your screen with the motion diagram.
- Try increasing v_x.
- Try increasing a_x.
- Try making v_x and a_x have opposite signs.
- Trying making v_x and a_x both negative.
- Trying making Δt smaller and see what effects it has on your motion diagram.

2.8 What about an animation?

The motion diagram is an artifact from a printed textbook. It's an attempt to illustrate motion using a medium (print) that is not able to show an animation (or movie) of the ball actually moving. This lack of animation support may limit what you can learn about motion. So, now let's make our ball actually *move*, using computer animation.

PhysGL makes animation easy. We can start using the code from the last program, and then make just a few changes to it. In particular:

- After the `dt=5`, add a `t=0` line.
- Replace the `for t=0,80,dt do` line with `while t<80 animate`
- Make `dt` a bit smaller, perhaps 0.1.
- Add a `t = t + dt` to the line just above the `end` line as shown.
- Add the line `draw_vector(r,a,"green")` after the `draw_vector(r,v,"yellow")` line to draw a green acceleration vector on your ball.

Your finished code should look like this:

```
r = <0,0,0>
v = <10,0,0>
a = <-1,0,0>
dt = .1
set_vector_scale(3)
t=0

while t<80 animate
        draw_sphere(r,5,"red")
        draw_vector(r,v,"yellow")
        draw_vector(r,a,"green")
        r = r + v*dt + 0.5*a*dt^2
        v = v + a*dt
        t = t + dt
end
```

When run, you will see a red ball moving rightward across the screen, with its yellow velocity vector pointing toward the right. You are now in a powerful position to experiment with the motion of your red ball, using the laws of physics. You may endow it with any initial position, velocity and acceleration that you wish, and watch what its resulting motion will be.

If you want to make more of an impact with your vectors, try inserting a line like `set_vector_scale(2)` somewhere above your `while t< 80 animate` line. This will scale up the length of all vectors by 2.

Things to try

- Try increasing and decreasing v_x.
- Try increasing and decreasing a_x.
- Try making v_x and a_x have opposite signs.
- Try making v_x and a_x both negative.
- Try making v_x and a_x both positive.

⧫ **Turn-In Ch. 2#6.12:** Describe how v_x behaved as the animation unfolded, for the last 3 bullet points above.

If you run through these exercises, you'll find that the physics illustrated will cover essentially the entire chapter in your book on one-dimensional kinematics!

Other things to try include:

- Making Δt smaller and see what effects it has on your motion diagram.
- Giving $\vec{v}$ and $\vec{a}$ components other than just in the x-direction.

  ⧫ **Turn-In Ch. 2#6.13:** Describe some arrangement of $\vec{v}$ and $\vec{a}$ and what motion you saw in this last bullet point.

- Making the sphere have a larger (or smaller) radius.
- Adding a "trail" behind your sphere by adding the line `draw_sphere(r,1,"green",true)` after the `draw_vector` line. This will draw a small green sphere at the position of your main sphere. The `true` you see in this line means "leave the green sphere on the screen." You might have to increase `dt` (try 0.5) to see a space between the trailing spheres.

  ⧫ **Turn-In Ch. 2#6.14:** Include a screenshot of this moving just as the main object is about to leave the screen.

- Experimenting with the run-time of your animation. What happens if the `while t < 80 animate` line was changed to `while t < 5 animate` or `while t < 50 animate`?

  ⧫ **Turn-In Ch. 2#6.15:** Describe what the number in this `while` loop seems to do. Translate the `while` statement into an english-sentence.

2.9 Motion Diagram

⧫ **Turn-In Ch. 2#6.16:** Starting with the code in Section 2.7, re-create the motion diagram shown in Problem #52 on Page 32 of your textbook. Be sure each dot has the velocity and acceleration vectors drawn on it. Hint: You'll have to add a second set of r's, v's, a's, physics equations, and sphere drawing lines–one for each object represented in the motion diagram.

2.10 Different accelerations on an object

If you flip through Chapter 2 in your textbook, you'll see many x vs. t and v vs. t graphs. In your studies, you should be working to fully understand these. Take a look at the a vs. t graph in Fig. 2.37 on page 57. It shows a being zero for a time, abruptly becoming negative for a time, before it returns to zero again. How would this be put into the computer? Turns out using "piecewise functions" you learned about in calculus are very useful here. In Fig. 2.37, suppose the acceleration first went to -1 at $t = 5$ seconds, then went to zero again at $t = 10$ seconds. The acceleration function, $a(t)$ would then be

$$a(t) = \begin{cases} 0 & t < 5 \text{ or } t > 10 \\ -1 & 5 \leq t \leq 10, \end{cases}$$

which is the form of piece-wise functions you learned about in your calculus course. Such functions can be put into the computer using an `if-then` statement, as in

```
if t < 5 or t > 10 then
 a=<0,0,0>
end

if t >= 5 and t <= 10 then
 a=<-1,0,0>
end
```

An `if-then` statement is a way of telling the computer to run lines of your code only if some condition is true. In this case, the conditions all come from the piece-wise function shown above. Here a will be $\langle 0, 0, 0\rangle$ only if t is less than 5 or greater than 10. It will be $\langle -1, 0, 0\rangle$ only if t is greater than or equal to 5 and less than or equal to 10.

Starting with the code in Section 2.8, create an animation of a sphere that has this piecewise-defined acceleration. Be sure to show all v and a vectors on the sphere at all times. In addition to the graphics screen, bring up the XY-graph window as well. Just above the `while` statement, add the line `new_graph("t","vx")` and somewhere between the `while` and `end` lines, add the line `go_graph(t,v.x)`. The `new_graph` line will start a new graph whose horizontal axis is labeled "t" and the vertical-axis is labeled "vx." The `go_graph` line will add (t, v_x) points to the graph as your sphere moves.

⧫ **Turn-In Ch. 2#6.17:** Turn in a snapshot of the v_x vs. t graph that results. How does it compare with the one in Fig. 2.37 in your textbook?

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 for you. They don't really have anything to do with physics, and aren't necessarily something you'll learn in physics. But to begin learning a technical field you should already know them from your prior preparation .

3.1 Mathematics

Variables. Variables are letters that stand for numerical values. Something like x^2 means that if 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 is used with a right triangle.

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

Basic Trigonometry. Know the difference between a radian and degree and how to go back and forth 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 can 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 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 (or using) 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 begin, 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 $\vec{v}$-vector could be written as $\vec{v} =< 6, 2, 0 >$, or $\overrightarrow{velocity} =< 6, 2, 0 >$. 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 $\vec{a}$-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 $\vec{v} = \vec{a}\Delta t$, $< 0, -19.6, 0 >$ 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's 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 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.
- $\hat{i},\hat{j},\hat{k}$-notation (or engineering notation). Specify the components of the vector directly. $\hat{i}$ corresponds to x, $\hat{j}$ to y and $\hat{k}$ to z. In this class the z-component will always be zero. So a vector written like $5\hat{i}+2\hat{j}$ 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 x-axis.

3.3.4 The magnitude of a vector

The magnitude of a vector is how long the 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 by 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 $\vec{v}$ can be found by $|\vec{v}| = \sqrt{v_x^2 + v_y^2 + v_z^2}$ which in this case is $\sqrt{3^2 + 2^2 + (-1)^2}$, or $|\vec{v}| = 3.7$.

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 vector $\vec{v}$, say v_x and v_y, you can always find the angle the vector is making with the $+x$-axis, using

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

For example, suppose $\vec{v} =< 3, 2, 0 >$. The angle the v vector is oriented in the xy-plane is found from $\theta = \tan^{-1}(2/3)$ or 33.6° with respect to the $+x$-axis.

Chapter 4

Doing Math with PhysGL

4.1 Introduction

It has been said that mathematics is the "language of physics," and studying physics using PhysGL will involve having the computer do mathematical calculations. You are likely used to doing such things on your calculator, like evaluating expressions or using `sin` and `cos`. This chapter summarizes the mathematical capabilities contained in PhysGL.

4.2 Simple computation

Just like your calculator or a spreadsheet, the basic operations are recognized in PhysGL. These are `+`, `-`, `/`, and `*` for addition, subtraction, division and multiplication. Note that PhysGL does not do algebra, so if you want $2x$, you have to put in `2*x`. PhysGL follows the order of operations too.

Other operations are `^` for exponent, and you may use the grouping symbols `(` and `)` as needed.

As an example, the physics equation $x = x_0 + v_0 t + (1/2)at^2$ may be put into PhysGL as `x=x0+v0*t+0.5*a*t^2`.

4.3 Trigonometry

PhysGL recognizes the following trigonometric functions. NOTE: The arguments to these functions *must* be in radians, and the result returned from an inverse function will also be in radians.

cos(x) - returns $\cos(x)$ where x is in radians.

sin(x) - returns $\sin(x)$ where x is in radians.

acos(x) - finds the angle whose cos is x. Returns $\cos^{-1}(x)$ (the arc-cosine).

asin(x) - find the angle whose sin is x. Returns $\sin^{-1}(x)$ (the arc-sine).

atan(x) - find the angle whose tan is x. Returns $\tan^{-1}$(x) (the arc-tangent).

atan2(y,x) - finds the tan whose angle equals y/x (and gracefully returns $\pi/2$ in the case where $x = 0$).

tan(x) - returns $\tan(x)$ where x is in radians.

csc(x) - returns $\csc(x)$ (cosecant) where x is in radians.

sec(x) - returns $\sec(x)$ (secant) where x is in radians.

cot(x) - returns $\cot(x)$ where x is in radians.

sinh(x) - returns $\sinh(x)$ (hyperbolic sine) where x is in radians.

cosh(x) - returns $\cosh(x)$ (hyperbolic cosine) where x is in radians.

tanh(x) - returns $\tanh(x)$ (hyperbolic tangent) where x is in radians.

4.4 Other functions

sqrt(x) - returns the square root of x.

exp(x) - returns e^x.

abs(x) - returns the absolute value of x or $|x|$.

4.5 Vectors

magnitude(A) - returns the magnitude of vector $\vec{A}$.

distance(A,B) - returns the distance between the points specified by vectors $\vec{A}$ and $\vec{B}$.

A * B - returns the dot product between vectors $\vec{A}$ and $\vec{B}$.

Aˆ2 - returns the square magnitude of $\vec{A}$.

A # B - returns the cross product of vectors $\vec{A}$ and $\vec{B}$ (not implemented yet).

Chapter 5

Drawing with PhysGL

5.1 Introduction

PhysGL, located at `http://www.physgl.org` is a browser-based drawing tool. It uses the popular OpenGL graphics library and is able to *render* a scene for you, in full 3D space, 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 PhysGL, and a list of PhysGL-defined colors you may choose from.

5.2 Coordinate System

The coordinate system is PhysGL is similar to the "cartesian coordinate system" that you are familiar with from your math courses. It looks like that shown in Figure 5.2. The $+x$-axis runs right, $-x$ is left, $+y$ is up and $-y$ is down. For full 3D space, we need a third axis, which is the z-axis. For this axis, $+z$-axis is out of the screen (toward the viewer) and the $-z$-axis is into the screen (away from the viewer). This is a right-handed coordinate system, just like the kind you use in your math classes.

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

5.3 Drawing Examples

PhysGL is able to draw many "primitive" shapes, like spheres, boxes, cylinders, and cones. In our study of physics, we can get a lot done by using just a few. Let's start with the sphere.

5.3.1 Spheres

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

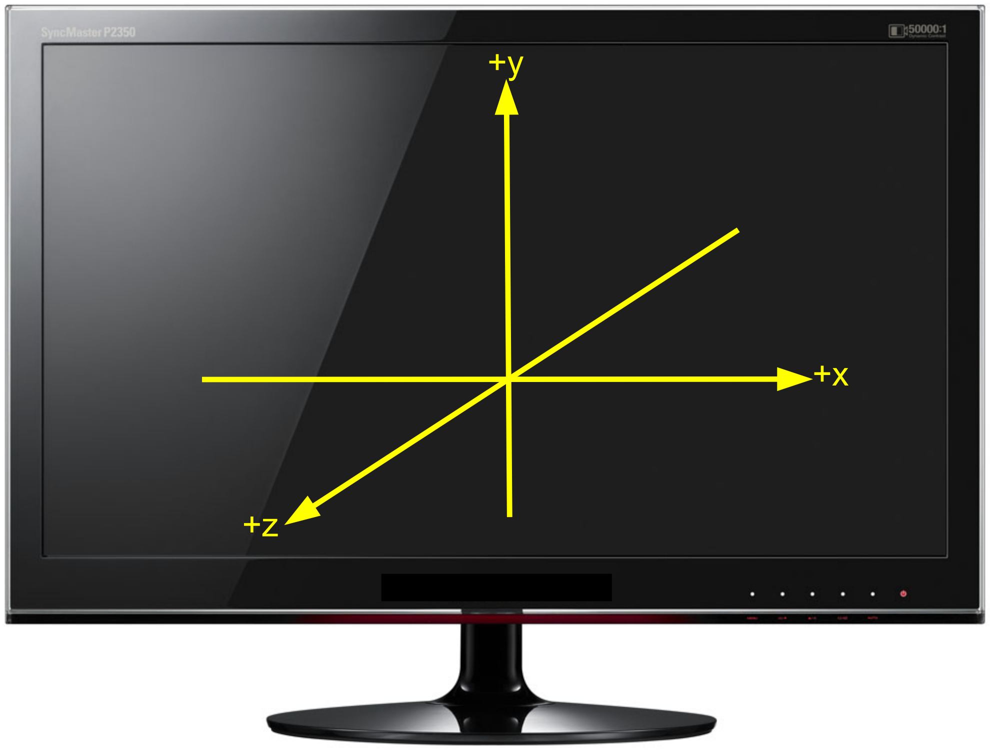


Figure 5.1: The PhysGL coordinate system: $+x$ runs right, $-x$ is left, $+y$ is up and $-y$ is down, $+z$ is out of the screen (toward the viewer) and $-z$ is into the screen (away from the viewer).

```
draw_sphere(<center-x,center-y,center-z>,radius,"color")
```

Note the word `draw_sphere` and the parenthesized arguments that follow. They're part of the PhysGL syntax. There is a vector in ordered-set form specifying the location of the center of the sphere. There is a comma then the radius parameter, and a word (in double quotes) which is the color of the sphere. Here are some examples:

`draw_sphere(<0,0,0>,5,"blue")` draws a blue sphere of radius 5 at the origin.

`draw_sphere(<0,0,-5>,15,"green")` draws a green sphere of radius 15 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 PhysGL 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:

```
pos=<5,1,0>
draw_sphere(pos,5,"blue")
```

The `draw_sphere` statement is complete, but the position vector doesn't appear explicitly. The position 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 PhysGL for producing animations:

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

5.3.2 Cubes

Cubes can be useful for representing objects too. Drawing a cube looks like this

```
draw_cube(<x,y,z>,side-length,"color")
```

where the vector point $< x, y, z >$ will be the center of the cube and "side-length" will be the length of each side of the cube. The "color" parameter is used here as well. Here is an example:

```
draw_cube(<5,1,0>,20,"blue")
```

This will draw a blue cube, centered about $x = 5$, $y = 1$, $z = 0$, with a side length of 20 pixels. What about basing the position of the cube on a variable, as we did by using `pos` in the sphere section above? Supposing again that `pos` is our position variable, one could do this

```
pos=<5,1,0>
draw_cube(pos,20,"blue")
```

If you want a box-like object that is not a cube, you can use the `draw_box` function. Here, you specify the location of two diagonal corners of the box (as vectors), and PhysGL will do the rest, as in

```
draw_box(<-35,-25,-50>,<-20,-20,50>,"green")
```

5.3.3 Lines

Lines are useful for a variety of general drawing tasks. Drawing a line is done like this

```
draw_line(<x1,y1,z1>,<x2,y2,z2>,"color",thickness)
```

where the two vector points $< x1, y1, z1 >$ and $< x2, y2, z2 >$ are the two ends of the line in 3D-space. The `"color"` is the color and the thickness of the line is given by the `thickness` parameter. The thickness is on a relative scale. Start with a thickness of 1.0 and go from there. The same rules apply for substituting the explicit vector ends with variables. Here is an example

```
draw_line(<20,10,10>,<-40,-50,-50>,"red",1)
```

which will draw a red line, with a thickness of 1, from $x = 20$, $y = 10$, $z = 10$ to $x = -40$, $y = -50$, $z = -50$.

5.3.4 Planes

Planes are useful for representing the ground, a table-top, or a wall in a physics problem. A plane in PhysGL has this syntax

```
draw_plane(<nx,ny,nz>,offset,"color",side-length)
```

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 `offset` parameter is a number representing how far the plane should be displaced along the normal vector, and the `"color"` parameter is what color the plane should be. The `side-length` sets how long each side of the plane should be. The plane is drawn to be very thin along the normal axis, while extending to infinity in all other directions. Here are some examples

- This draws a plane perpendicular to the y-axis, in blue, displaced downward along the y-axis by 10 pixels, with a side length of 50 pixels.

  ```
  draw_plane(< 0, 1, 0 >, -10,"blue",50)
  ```

- This draws a plane oriented at 45° between the x and y axes in yellow, pulled down and the left by 5 pixels, with a side-length of 100.

  ```
  draw_plane(<1,1,0>,-5,"yellow",100)
  ```

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

5.4 Colors

Colors are often linked to objects using the `"color"` parameters shown above. For the `"color"` part, you may substitute any of the following (don't forget the double quotes—color names are strings of text characters). Note that color names are *case sensitive!* So you can use `black`, but `Black` or `BLACK` won't work.

5.4.1 Stock Colors

```
aliceblue antiquewhite aqua aquamarine azure beige bisque black blanchedalmond
blue blueviolet brown burlywood cadetblue chartreuse chocolate coral
cornflowerblue cornsilk crimson cyan darkblue darkcyan darkgoldenrod darkgray
darkgreen darkgrey darkkhaki darkmagent darkolivegreen darkorange darkorchid
darkred darksalmon darkseagreen darkslateblue darkslategray darkslategrey
darkturquoise darkviolet deeppink deepskyblue dimgray dimgrey dodgerblue
firebrick floralwhite forestgreen fuchsia gainsborod ghostwhite
gold goldenrod gray green greenyellow grey honeydew
```

```
hotpink indianred indigo ivory khaki lavender lavenderblush
lawngreen lemonchiffon lightbluea lightcoral
lightcyan lightgoldenrodyellow lightgray lightgreen lightgrey
lightpink lightsalmon lightseagreen lightskyblue lightslategray
lightslategrey lightsteelblue lightyellow lime
limegreen linen magenta maroon mediumaquamarine mediumblue mediumorchid
mediumpurple mediumseagreen mediumslateblue mediumspringgreen mediumturquoise
mediumvioletred midnightblue mintcream mistyrose moccasin
navajowhite navy oldlace olive olivedrab
orange oranger orchid palegoldenrod palegreen paleturquoise
palevioletred papayawhip peachpuff peru pink plum
powderblue purple red rosybrown royalblue saddlebrown salmon
sandybrown seagreen seashell sienna silver skyblue slateblue
slategray slategrey snow springgreen steelblue tan
teal thistled tomato turquoise violet wheat
white whitesmok yellow yellowgreen
```

5.4.2 Make your own color

If you don't find a color you like from the above list, you can make your own color by mixing various amount of red, green, and blue. The fraction of each color is a number between `00` and `ff` (hexadecimal). Colors can be made by a construct like `rrggbb`, where `rr`, `gg`, and `bb` are numbers between `00` and `ff` (hexadecimal) which are numbers between 0 and 255 in decimal. These numbers indicate the fraction of red, green, or blue respectively, that should go into a color, sort of like mixing differing amounts of red, green and blue light to make a color. Instead of a color name, as in the above, you would use the number construct `rrgbb`, which would go in the double quotes where a color name above would normally go. Thus a color might be `"808080"`, which would be a shade of gray (because 80 hex = 128 decimal), while `"ff0000"` would be pure red. Do a google-search for "HTML colors" for more on how to construct your own colors.

5.5 Textures

The textures called `brick`, `metal`, `rope`, `crate`, `water`, and `grass` have been added. Use these names in the place of any colors to get a textured look.

Chapter 6

Getting Started with Simple Programming

6.1 Introduction

Will this approach to learning physics require you to write simple computer programs? In a word, "yes." As a budding scientist, knowing something about programming is a good idea, since computer software essentially runs our world these days (see "Software eating the world" here `http://goo.gl/ndMdOs`). Almost no science or engineering is done with pencil and paper anymore. In addition to the classic uses of *theory* and *experiment*, *the computer* is now the an integral part of scientific progress.

As a practical matter, as you grow in your studies, you'll 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.

There is also a point to be made about programming. When you program, your work must be correct with logic and implementation, or it won't work properly. The computer demands this. This correctness demands a level of clarity of your thinking about physics, that simply does not exist if you are doing end-of-chapter homework problems (with the answer guide nearby). Often, if your program doesn't work here, you're not thinking about the necessary *physics* properly. As Steve Jobs said (in the "Lost Interivew"):

> *Everybody in this country should learn how to program a computer because it teaches you how to think.*

Luckily, PhysGL 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.

6.2 Skeleton Code

All the animations ("movies") you will create can (and should) be started with the following lines, called "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/ndMdOs` .

```
pos = <??,??,??>
vel = <??,??,??>
t = 0
dt = ??
while t < ?? animate
        a = <??,??,??>
        draw_sphere(pos,5,"red")
        pos=pos+vel*dt+0.5*a*dt*dt
        vel=vel+a*dt
        t=t+dt
end
```

This code represents the *physics logic* required to make an object move. The details of the motion (where the object starts, how fast it moves, what's its acceleration) are to be implemented by replacing the `??` in the code with the necessary details.

This program is 11 lines long. We hope this short length will allow you to think of the programming going on here, not as *programming*, but as a means of allowing you to express your logic and understanding of the physics of motion. Here's the logic the code implements:

- 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.
- `t` is the time variable
- There are the two physics equations evolving the position and velocity in time.
- Next is the line that draws the red sphere on the screen. Spheres need a center point (`pos`), a radius (5 in this case), and the color to be drawn.
- Notice the `while t < ?? animate` construct, and the `end` statement it is paired with. In words this says "while time is less than ??, animate..." You'll have to fill in some condition for the `??` that is specific to your needs. What is animated is the code between the `while` and `end` lines, until the `??` condition fails. This is the main PhysGL animation method.

Suppose we wanted to adapt the code to make a sphere move from left to right across the screen. The `??` mentioned above is the condition `t < 15` in the code below. So the skeleton code above might become the following.

```
pos = <-50,10,0>
vel = <0,0,0>
t = 0
dt = 0.1
while t < 15 animate
        a = <0,0,0>
        draw_sphere(pos,5,"red")
        pos=pos+vel*dt+0.5*a*dt*dt
        vel=vel+a*dt
        t=t+dt
end
```

If you enter this into PhysGL and hit the "Run" button, you'll see a red sphere drawn on the screen. Pretty boring, but according to the laws of physics, an object "at rest" can "remain at rest" unless acted on by a force with the corresponding acceleration.

Why doesn't the sphere move? Well, read the program. It looks like the variable `vel` (short for velocity) has all three components set 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, one of Newton's Laws ("an object at rest, stays at rest..."). To get this sphere to move, change the `vel=<0,0,0>` line to `vel=<5,0,0>`. Re-render the program. You should see the sphere moving to the 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 5 to something smaller in the line `draw_sphere(pos,5,"red")`. 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 `a=<0,0,0>` line to `a=<5,0,0>`
- To make it slow down, stop and turn around change it to `a=<-2,0,0>`
- To change where the sphere starts on the screen, you can change the line `pos=<0,0,0>` to `pos=<-3,0,0>` The PhysGL 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 (away from the viewer).
- 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. Try changing the velocity to `vel=<0,5,0>` and `a=<0,-3,0>`.

6.3 Basic PhysGL Programming

To create physics animations, 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 need to be familiar with:

Assigning values to variables. In a math class, you'd write $x = 5$ to set the variable x to 5. In PhysGL, you do the same thing: a variable name, then an equal sign, then what you want to set it to. Here are some examples:

- `a=5`
- `Radius=10`
- `velocity=<5,1,0>`
- `g=9.8`

Notice in these examples that PhysGL recognizes both scalar and vector quantities. The vector specification uses "ordered set notation" exclusively.

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. You can also set a vector component to a given value in the same way. So if you have `a=<0,0,0>`, later letting `a.y=9.8` will set a_y to the value 9.8.

Doing math. PhysGL (and most programming languages) 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 /. For the exponents, use the caret, which is a ^ ([Shift]-[6]), so you can write `x=5.25^2` or `dt^2`.

More math Other math-related items are `Pi`, and of course `sin`, `cos`, and `tan`, which take radians (not degrees) as their arguments. PhysGL has all of the math functions that you're used to built in. You can find a list of them in the PhysGL documentation, which is linked to from the site itself.

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 $\vec{v}$-vector. We'll call the result `vmag` (assuming your velocity vector is a variable called `vel`), you could write

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

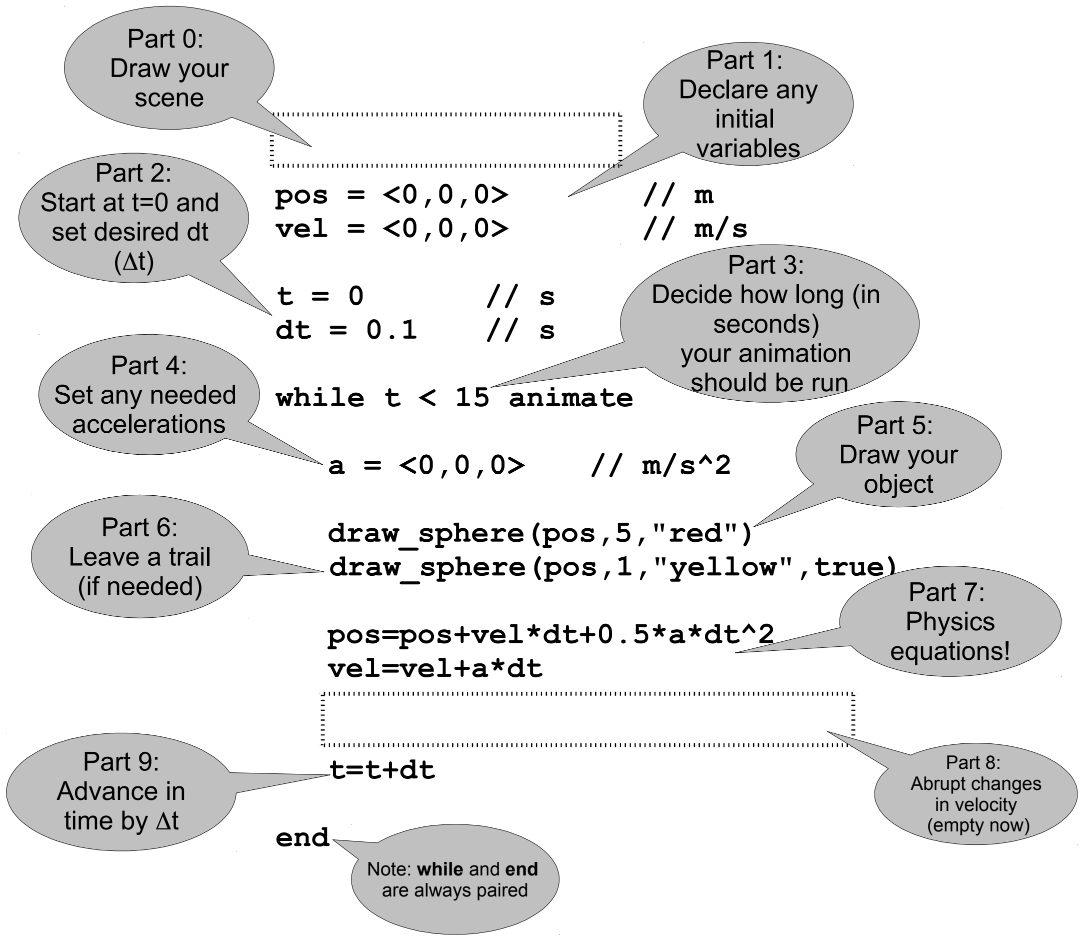
More math The arctangent is normally used to find the angle a vector is making with respect to the $+x$-axis. PhysGL has a special version of arctangent called `atan2` which works like this. Suppose you want to take the arctangent of Q_y/Q_x or $\text{atan}(Q_y/Q_x)$, where Q_x and

Q_y are the components of some vector. In PhysGL you'd do `atan2(Qy,Qx)`. So suppose you wanted the angle a velocity vector is making (with respect to the $+x$-axis), assuming your velocity vector is the variable called `vel`. To assign this angle into a variable called `angle` you could write

```
angle = atan2(vel.y,vel.x)
```

6.4 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 6.1, and remember that the computer will process it line-by-line, from top to bottom.



```
pos = <0,0,0>        // m
vel = <0,0,0>        // m/s

t = 0       // s
dt = 0.1    // s

while t < 15 animate

   a = <0,0,0>    // m/s^2

   draw_sphere(pos,5,"red")
   draw_sphere(pos,1,"yellow",true)

   pos=pos+vel*dt+0.5*a*dt^2
   vel=vel+a*dt

   t=t+dt

end
```

Part 0: Draw your scene
Part 1: Declare any initial variables
Part 2: Start at t=0 and set desired dt (Δt)
Part 3: Decide how long (in seconds) your animation should be run
Part 4: Set any needed accelerations
Part 5: Draw your object
Part 6: Leave a trail (if needed)
Part 7: Physics equations!
Part 8: Abrupt changes in velocity (empty now)
Part 9: Advance in time by Δt
Note: while and end are always paired

Figure 6.1: The highly structured skeleton code which serves as the core for all work here.

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 four things.

1. Where should the object start?
2. What velocity should it have when it starts?
3. What acceleration should it have?
4. What do I want to draw for my object?

If you do not have firm answers to all four of these questions, you've likely not assembled the full physics picture for your system and it will be difficult to translate this into a working animation.

If you *can* answer these four questions, then you can fill in the skeleton code and likely have a working animation in short order. Part 1, for example is where the initial position and velocity of your object should be set, in full vector form (of course). Part 4 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 5. If you want to leave a trail behind your object as it travels, see Part 6 (where you just draw a small yellow sphere at the same position as your object; the "true" at the end means the trail will stay fixed on the screen). 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 is all about determining the position where your object will start, and what velocity it should have there. Both are specified in full vector form (of course).
- Two things happen in Part 2. First, the frame-to-frame time-step of your animation, 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 `t` is set equal to zero. This variable is the global "clock" in your movie, always holding the amount of time that has elapsed.
- Part 3 tells PhysGL to start an animation sequence. The condition between the `while` and `animate` words tells PhysGL when to stop the animation. Here we've specified `t < 15` meaning, to "keep animating as long as $t < 15$." If you wanted the animation to run forever you would use `while true animate`.
- Part 4 is about the core physics of your system. What acceleration should your object have?
- Part 5 is where you actually draw your object. Here the main object will be a red sphere with a radius of 5 pixels.
- In Part 7, you'll see the same physics equations (kinematic) as in your book. They sit in there and drive the animation throughout the whole animation sequence.

- Part 8 is empty for now, but deals with physics that involve abrupt changes to the object's velocity. Filling in this section will be important when you study collisions of objects.
- Part 9 advances the time t of the animation, or $t \to t + \Delta t$.

6.5 The if statement

We use `if` statements to allow your program to *make decisions* as it runs. 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 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 is now in free fall. The acceleration of the block now is now $a_y - 9.8$ m/s^2. So you really have two acceleration zones here. For the acceleration along the y-axis, you have $a_y = 0$ 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.

6.5.1 An if-then statement in PhysGL

In PhysGL, 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. When you think about it, the acceleration is a piece-wise function, like you encountered in your calculus class. Suppose also that the table extends from $x = -5$ to $x = 0$. In this case, the acceleration vector would be:

$$\vec{a}(x) = \begin{cases} \langle 0,0,0\rangle & -5 \le x \le 0 \\ \langle 0,-9.8,0\rangle & x > 5 \end{cases}, \tag{6.1}$$

which says the block's acceleration if $\vec{0}$ when it's on the table, and $a_y = -9.8$ when it's over the edge of the table.

The `if` statement very naturally implements such piece-wise functions, as follows.

```
if pos.x >= -5 and pos.x <= 0 then
      a=<0,0,0>
end

if pos.x > 0 then
      a=<0,-9.8,0>
end
```

Reading the code top to bottom, you see the first `if` statement: it reads "if the x-component of pos is greater than or equal to -5 and less than or equal to 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 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 PhysGL, here is the general form of an `if` statement, that you may place anywhere in your code:

```
if condition then
   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 execution will skip to the first available line after the `end` statement.

6.5.2 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 (note the TWO equal signs for testing for equality–this is very common in the world of computer code). PhysGL has more comparisons such as `>=` and `<=`. Use `<>` (or `!=`) for "not equal." These simple conditions may be tied together using `and` and `or` to construct complex conditions.

Table 6.1 shows some example conditions that might be helpful when animating physics. It assumes the position of an object is being held in the vector variable `pos` and its velocity in a variable `vel`.

6.5.3 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 is 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 our box is on the table 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

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 and pos.x<-3`	True if v_x is less than zero (the object is moving left) and the object is located 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 and pos.x==1`	True if the v_y is greater than or equal to -2, 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 and 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 and vel1.x>0 and 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 6.1: Sample conditions, what they test, and where they might be used, assuming `pos` and `vel` are the position and velocity vectors of the object.

equal to -5 or less than or equal to 0 (which are the horizontal bounds of the table). The condition itself would be written inside of the parentheses that is part of the `if` statement construct described above.

6.5.4 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 the variable that is 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.

6.5.5 Careful with = and ==

Most important, remember that a single `=` sign is for assigning values to variables, like `x=2`, which puts the value of `2` into variable `x`. The double equal sign, or `==` is for making comparisons, as in `x==2` which checks if `x` is `2` or not, and results in a true or false.

When crunching numbers for physics movies, be careful with expecting the "==" (comparision-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 $x = x_0 + v_0\Delta t + (1/2)a\Delta t^2$, 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 be anything in particular. Instead check for a range, that will typically involve < and > to compare. 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 and pos.x < 0.1 Will be true if the box is within $|0.1|$ units of the table's edge. You can adjust 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<pos.x<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 7

Drawing Physics

7.1 Introduction

Although PhysGL is a convenient drawing tool, we need it 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 relevant items that will allow us to fully visualize the physics that will be 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, PhysGL does not have a native "arrow" drawing function. However, with PhysGL, 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, many useful functions for drawing physics exist, and are described below.

7.2 Ideas on notation and usage

The above sections referred to "functions for drawing physics." Let's discuss this further. 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 $f(a) = a^2$. You put in 5 and got out 25. You put in a and got out a^2. You can also have functions that involve more than one variable, like $f(x, y, z)$, but the meaning is similar.

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 "the force is 50 N at 30 degrees above the +x-axis." 2) "i,j,k" or "Engineering notation" like $5\hat{\imath} + 7\hat{\jmath}$. 3) "Ordered Set Notation" as in $< 5, 7, 0 >$. PhysGL uses "ordered set" notation to represent vectors, as you'll see below.

7.3 Useful functions for visualizing physics

draw_label_vector : Draw a vector with a label

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 text `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>,"green","a")
```

In another setting you may have a vector variable, 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 $\vec{v}$-vector: `draw_label_vector(pos,vel,"red","v")`
- Draw the $\vec{a}$-vector: `draw_label_vector(pos,a,"yellow","a")`
- Draw just v_x: `draw_label_vector(pos,<vel.x,0,0>,"white","vx")`
- Draw just v_y: `draw_label_vector(pos,<0,vel.y,0>,"hotpink","vy")`

draw_vector : Draw a vector (with no label)

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

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."

Examples:

```
draw_vector(<0,0,0>,<5,5,0>,"red")
draw_vector(<0,0,0>,<F.x,0,0>,"blue")
draw_vector(<sx,sy,0>,<ax,ay,0>,"red")
```

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 $\vec{v}$-vector: `draw_vector(pos,vel,"red","v")`
- Draw the $\vec{a}$-vector: `draw_vector(pos,a,"yellow","a")`
- Draw just v_x: `draw_vector(pos,<vel.x,0,0>,"white","vx")`
- Draw just v_y: `draw_vector(pos,<0,vel.y,0>,"hotpink","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 too 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 ??.

Examples:

```
set_vector_label_color(Red)
set_vector_label_color(Yellow)
```

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

Usage: `printxyz(location,"the-text",size,"color")`

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

Examples:
`printxyz(<5,1,3>,"hello",20,"green")`
Draws the text "hello" at location x=5, y=1, z=3 in green, with a point size of 20.

`printxyz(<sx,sy,0>,"the object","yellow",10)`
Draws the text "the object" at the location $(sx, sy, 0)$ in yellow, with a point size of 10.

write : Writes textual information to the console window.

Usage: `write(arg1,arg2,arg3...)`

Description: Takes the comma delimited list of constants or variables `arg1, arg2, arg3...` and writes their values to the text-console window. Helpful for debugging or retrieving values of your programs variables.

Examples:

- `write(‘‘this value of x is",x)`

  Given that variable `x` is defined (suppose it contains a 4), this will display the string `The value of x is 4` to the console window.

- `write(vel,‘‘is the velocity vector ’’,’’at time’’,t)`
  If `vel` is a vector containing the value $< 6, 2, 1 >$ an t contains 3, will display `6,2,1 is the velocity vector at time 3` in the console window.

writeln : Same as `write` above, except a newline is added after each line.

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 effects 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 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 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` $\le x \le$ `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 $-5 \le x \le 5$ 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 $-10 \le x \le 10$ 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 PhysGL/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 **??**).

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 8

One-dimensional motion

8.1 Introduction and Goals

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

- You'll change an object's $\vec{v}$-vector by applying an arbitrary acceleration to an object either parallel or antiparallel to the object's $\vec{v}$-vector.
- Demonstrate that you understand the interplay between x, v, and a, and 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.

8.2 The Physics

The $\vec{v}$-vector of an object will be changed by: An acceleration either parallel or anti-parallel to $\vec{v}$. The goal of this section is to understand how the acceleration, $\vec{a}$, can be used to primarily change an object's $\vec{v}$-vector, and secondarily cause 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 the 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 velocity v_0, then x and v will be the object's new position and velocity at some time interval Δt later. These two equations allow you to compute the new position and velocity of an object (x and v), based on its old position and velocity (x_0

and v_0), given some acceleration a that is acting on the object, over a time interval Δt. The quantity 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 correspondingly in 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 $y = y_0 + v_{0y}\Delta t + \frac{1}{2}a_y\Delta t^2$ and $v_y = v_{0y} + a_y\Delta t$. These equations have the same physical meaning, but the notation is modified for the axis of interest.

Example: 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 the 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)?

SIGNS: You must 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 for y) relative to the origin. Negative means the object is to the left (x) (or down for y) relative to the origin.

Positive values of velocity mean the object is moving toward the right ($v_x > 0$) or up ($v_y > 0$), negative means to the left ($v_x < 0$) or down ($v_y < 0$). The sign of a alone does not immediately help to characterize an 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 point the object will stop. If a continues to persist, then v will begin to increase in the same direction as a; now the object is speeding up, but in the opposite direction from its original motion. All told the object slowed down, stopped (momentarily), 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 $\vec{v}$ and $\vec{a}$ at that instant. An arrow should point in the direction of the parameter it represents, and its length should be proportional to its magnitude (or strength). For example, if for 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 $\vec{v}$ arrow half as long as one going 4 m/s. **Book reading: 1.3-1.6, 2.4, 2.5**.

8.3 Projects

Learning outcomes: To understand how an object's $\vec{v}$-vector will be affected by an acceleration either parallel or anti-parallel to the direction of $\vec{v}$.

Start a document again and address and “Turn-in” items in the document. Note that when a “share link” is being asked for, you can generate this from any of your code by clicking the “Share” button, which is next to the “File Manager” button in PhysGL. When you put this link into your document, be sure that is it a “live link” (appears blue and underlined) so your professor can click it, in order to view your code and animation.

1. This animation can be made almost directly from the skeleton code, found on Page 44 in Section 6.1.

   - Create an animation that shows a sphere moving from the left edge of the screen to the right edge of the screen. The motion of the object should be for $v_0 > 0$, and $a = 0$. Draw *and label* the velocity vector on the sphere at all times. Review the usage of `draw_label_vector` on Page 52 if needed. Be sure the object leaves a “trail” behind it, indicating past positions of the object. See the Part 6 of the skeleton code for more on leaving trails. Get this part working before moving on.
   - Now, we want to look at graphs of x vs t, v_x vs t, and a_x vs t. To do this, somewhere above the `while` loop, add the line `new_multi_graph("t","x","v","a")` which will start a new series of graphs in your “Graph” window. Somewhere *in between* your `while` and `end` statements, add the line `go_multi_graph(t,pos.x,vel.x,a.x)` to add the t, x, v_x and a_x data points to the graphs. This assumes your time variable is called `t`, your position vector is called `pos`, your velocity vector is called `vel` and your acceleration `a`. Run the code and generate the graphs. Be sure to open/position your “Graph” window. ⧫ **Turn-In Ch. 8#1.1:** Include a screenshot of your graphs in your document.
   - ⧫ **Turn-In Ch. 8#1.2:** Discuss the trends and relationships you see between the graphs. Be thorough with your discussion. It is covered in your book. Treat it at the level of your textbook.
   - ⧫ **Turn-In Ch. 8#1.3:** Hover over the data points in your v_x vs. t graph. Compute the slope of this graph from the numbers you can retrieve. How does this number compare to the acceleration of your sphere?
   - ⧫ **Turn-In Ch. 8#1.4:** Put a clickable link in your document to your PhysGL program. The link can be generated using the “Share” button which is to the right of the “File Manager” button near the top of your PhysGL screen. Be sure this is a live link in your document (blue and underlined) for easy access by those grading your work.

2. From the code used for the last animation, repeat the left-to-right motion, but make the object speed up by giving v_x and a_x the same sign. Draw both the velocity and acceleration vectors on the sphere at all times. Be sure the object leaves a “trail” behind it, indicating past positions of the object. ⧫ **Turn-In Ch. 8#2.1:** Generate the graphs as above. ⧫ **Turn-In Ch. 8#2.2:** Include a screenshot of your graphs in your document. Again discuss the graphs as in the problem above. ⧫ **Turn-In Ch. 8#2.3:** Include the share link to your code, so we can grade it.

3. Adapt the code again, but for a sphere that has an acceleration that is opposite to the velocity; in other words $\vec{v}$ and $\vec{a}$ should have opposite signs. The object should slow down, stop, then turn back around and go into the direction from which it came originally. Be sure your animation clearly shows this happening, filling the screen with its overall motion. Draw both the velocity and acceleration vectors on the sphere at all times. Be sure the object leaves a "trail" behind it, indicating past positions of the object. ♦ **Turn-In Ch. 8#3.1:** Construct another discussion of the graphs as in the problem above, and include a screenshot. ♦ **Turn-In Ch. 8#3.2:** On the v vs. t graph, indicate where the object has turned around and discuss why you made this decision. ♦ **Turn-In Ch. 8#3.3:** Include the share link to your code, so we can grade it.

4. Show a sphere 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 (momentarily), then turn around and fall back down again. Draw 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. ♦ **Turn-In Ch. 8#4.1:** Include a share link to your code so we can grade it. ♦ **Turn-In Ch. 8#4.2:** Include a screenshot of the v_y vs. t graph, and indicate on the graph where the object turned around and discuss why you chose this location on the graph.

5. (Start this work with your code from true last problem.) An rocket (the object) starts from rest on the ground (the bottom of the screen) moving upward vertically with an acceleration of 6.0 m/s^2. It maintains this acceleration for 4 seconds. It then enters a stage of free-fall for 4 seconds. Then it enters another stage of upward acceleration with 3 m/s^2 for 12 seconds. It has now used all of its fuel and stays in free-fall. Using `draw_label_vector`, draw and label both $\vec{v}$ and $\vec{a}$ on the object at all times.

   Note: You can adjust the view of your scene by putting a `camera` statement somewhere above the `while` loop, as in `camera(pos,look-at)`, where `pos` is the position vector of the camera and `look-at` is the point the camera should stare at. A line like this `camera(<0,0,200>,<0,0,0>)` would pull the camera back to $z = 200$ giving you a wider field of view. Vector thicknesses can be adjusted with a `set_vector_thickness(n)` line. Their overall scale can be set with a `set_vector_scale(n)` line. (Here `n` is a zoom-factor; 1 has no effect, 2 would double the thickness or scale, etc.)

   Put a `new_multi_graph("t","y","vy","ay")` somewhere *above* your `while` statement, and log the object's y, v_y and a_y parameters using a `go_multi_graph(...)` statement that is placed somewhere *within* the `while-end` lines. ♦ **Turn-In Ch. 8#5.1:** Put a snapshot of this completed graph in your document. ♦ **Turn-In Ch. 8#5.2:** Put a live share-link to your working code in your document. ♦ **Turn-In Ch. 8#5.3:** Use information in this graph to answer these questions.

   (a) Write up a discussion about the relationship you see between y, v_y, and a_y from the graph. Use some of the ideas in Fig. 2.24 on page 48 of your book.

   (b) At what time does the object reach it highest point? Discuss how you used the graph to determine this.

(c) Discuss how you could use the v_y vs. t graph to determine the maximum height the object reaches.

(d) How long does it take to reach the ground again? Discuss how you used the graph to determine this.

♦ **Turn-In Ch. 8#5.4:** Compute the answers to the above questions by hand (pencil and paper). Include a snapshot of your work, with clearly boxed in answers in your document. Compare your hand vs. PhysGL answers.

6. In this code, we're going to send a block toward a brick wall with some $v_{0x} \neq 0$. Here is code you can put *before* your `while` statement that will set this up:

```
camera(<-100,50,250>,<0,0,0>)
light(<-20,50,10>)
pos=<-120,10,0>
vel=<15,0,0>
draw_plane(<0,1,0>,0,"green",500,true)
draw_box(<50,0,-150>,<55,50,150>,"brick",true)
```

Note the camera placement in the scene which takes two vectors, the camera's position and the point in space it should "look at." Finish the rest of the code so that the block actually moves toward the wall (use past work if needed), with both $\vec{v}$ and $\vec{a}$ vectors drawn and labeled (use `draw_label_vector`) at all times. The cube can be drawn with `draw_cube(pos,20,"red")`.

(a) By trial and error figure how what $\vec{a}$ you'll need so that the block will remain on the green ground, stop, then turn around, just before it would crash into the wall. ♦ **Turn-In Ch. 8#6.1:** Report this acceleration. ♦ **Turn-In Ch. 8#6.2:** Include a live link to your PhysGL code so we can grade it.

(b) Now, use physics equations to compute (with pencil and paper) what this acceleration would be. ♦ **Turn-In Ch. 8#6.3:** Include your work in your document that clearly shows your resultant acceleration. How does it compare with your trial and error PhysGL-derived acceleration above?

7. Download the code called `balltower.txt` here `http://goo.gl/ndMdOs` in the `1Dmotion` folder. If you run it as given, it will show a ball on the top of a tall wooden tower, being sent directly toward the edge with $v_{0x} = 15$ (or $\vec{v} = 15\hat{x}$). Without altering this initial $\vec{v}$, change only the x-component of $\vec{a}$ in Part 4 of the skeleton code (Figure 6.1) so that the ball is just able to turn around before falling over the edge. This means only change the `a=<0,0,0>` line. Do this by trial and error with values for a_x in Part 4 until the box stops just as it reaches the edge of the tower.

Research Problems

8. A sphere starts at the top of the screen, and is in "free fall." Add an upward drag acceleration on the ball of the form Cv or Cv^2, where C is a constant (the drag coefficient) and v is the instantaneous speed of the ball. Use Wikipedia to find a suitable value for C. Use `new_graph` and `go_new_graph` to produce a plot v_y vs. t in a manner that will allow you to see the ball reaching a terminal velocity. ⧫ **Turn-In Ch. 8#8.1:** When you produce this graph, put a screenshot of it in your document. Next, vary C over several different values. Make an x-y plot (not in PhysGL) of the terminal velocity (vertical axis) vs. C (horizontal axis). ⧫ **Turn-In Ch. 8#8.2:** Draw some conclusions from this plot and include it in your document.

9. A rocket takes off under constant acceleration. Four seconds after liftoff, a bolt comes loose from the rocket and hits the ground 6 seconds later. Find the acceleration of the rocket by creating an animation in PhysGL. The animation should show the rocket moving upward, and the appearance and subsequent free-fall of a bolt as per the problem. Both the rocket and bolt can be represented by spheres. Be sure both $\vec{v}$ and $\vec{a}$ vectors appears on both objects at all times. By trial and error, vary the acceleration of the rocket, until the bolt hits the ground 6 seconds after coming off of the rocket. Use `writeln` to access any needed parameters in the "Console" window or `new_graph`/`go_graph` pairs to track any needed parameter graphically. Work the problem using pencil and paper and be sure the results agree. ⧫ **Turn-In Ch. 8#9.1:** Turn in some kind of report chronicling the use of PhysGL to solve the problem. Include your hand calculations after that.

10. Show a sphere 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 (momentarily), then turn around and fall back down again. When the object hits the ground, it should bounce with a coefficient of restitution C_R. Choose a suitable vertical position for the ground, and draw the ground using `draw_plane`. Draw both the velocity and acceleration vector on the spheres at all times. ⧫ **Turn-In Ch. 8#10.1:** Include a share link to your code so we can grade it. ⧫ **Turn-In Ch. 8#10.2:** Include a screenshot of the v_y vs. t graph, and indicate on the graph where the object turns around and discuss why you chose this location on the graph.

8.4 Wrap-up Questions

Answer the following questions on paper. Do not use the computer or PhysGL, etc.

1. Draw a simple car. Indicate that it is moving at constant speed to the left, by drawing appropriate $\vec{v}$ and $\vec{a}$ vectors on it.

2. Draw a simple car. Indicate that it is speeding up to the left, by drawing appropriate $\vec{v}$ and $\vec{a}$ vectors on it.

3. Draw a simple car. Indicate that it is moving toward the right but slowing down, by drawing appropriate $\vec{v}$ and $\vec{a}$ vectors on it.
4. Draw 5 circles evenly spaced apart. Each circle corresponds to a 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 $\vec{v}$ and $\vec{a}$ vectors on each circle 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 $\vec{v}$-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 $\vec{v}$-vector sticking out of it. Discuss what you can infer about its future motion.
10. A sphere has a $\vec{v}$-vector sticking out of it. A second sphere has its own $\vec{v}$-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 $\vec{v}$-vector sticking out it and pointing to the right. It also has an $\vec{a}$-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 Animation 6 that allowed the ball to turn around?
13. In Animations 6-7 above, what would happen if you accidentally put your guess for a_x into the 2nd position of the `a=`$< 0, 0, 0 >$ definition?

Chapter 9

Two-dimensional motion

9.1 Introduction and Goals

The goal of these projects is to demonstrate that you understand how an object moves in two dimensions (e.g. projectile motion).

- In this lesson, you'll change an object's $\vec{v}$-vector with an acceleration on an object such that the $\vec{a}$- and $\vec{v}$-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.
- 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 during flight.
- Demonstrate understanding of how a drag force will affect the motion of a projectile.

9.2 The Physics

The $\vec{v}$-vector of an object will be changed by: Applying a downward, vertical acceleration to the object. The goal of this section is to understand how objects move in 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 specified by its (x, y) coordinates. These coordinates are found from 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 as an object moves its velocity components along two axes are evolving 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 in many cases 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 in this chapter.

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 animations this week will show projectiles in flight, restricted to motions where $a_x = 0$ and $a_y = -g = -9.8$ m/s^2. You can immediately find appropriate forms of the $x(t) =$ and $y(t) =$ 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 velocity 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 see, the two components of both $\vec{r}$ and $\vec{v}$ give 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 at that instant of time. Be sure you understand why a vector has a magnitude and an angle, and be sure you can 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 $\vec{r} = (5\hat{\imath} + 2\hat{\jmath})$m, where $\hat{\imath}$ and $\hat{\jmath}$ are unit vectors corresponding to the 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 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 magnitude 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**.

9.3 Projects

1. $\vec{A} = \langle 5, 2, 0\rangle$, $\vec{B} = \langle 2, 7, 0\rangle$, and $\vec{C} = \vec{A} + \vec{B}$. Write a 3-line PhysGL program (using three `draw_vector` statements), that uses graphics to demonstrate how the vectors $\vec{C} = \vec{A} + \vec{B}$ are added graphically. ⧫ **Turn-In Ch. 9#1.1:** Include a side by side screenshot of both your code and graphics output, showing the vectors. Note! Start your code with the line `set_vector_scale(0.3)` to scale the vectors properly. To make you final image

bigger, trying moving the camera close with a line like `camera(<0,0,50>,<0,0,0>)`, which will move the camera's position to $x = 0$, $y = 0$, and $z = 50$, and pointing it to look at the point $\langle 0, 0, 0\rangle$.

2. Let's launch a ball into 2D motion. To do so, we need an initial position and velocity as always, but also an launch angle. Here is a framework for some code that will do this for you:

```
camera(<0,10,200>,<0,0,0>)
draw_plane(<0,1,0>,0,"green",250,true)

pos=
theta=
v0=
vel=

dt=.1
t=0

while t < 20 animate
...
end
```

The `camera` and `draw_plane` lines give your screen some perspective, with a green ground at $y = 0$. Now focus on the physics:

(a) Where to you want your ball to begin (pos)?
(b) What launch angle should it have (theta)?
(c) With what muzzle (launch) speed should it be launched (v_0)?
(d) From the last two, what full vector launch velocity should the ball have (vel)?
(e) What acceleration should the projectile have (it's in free fall)?

Inside of your while loop, place an appropriate line defining the ball's acceleration.

After being launched with some speed v_0 at some angle θ relative to the horizontal, it should fly across the screen. Your PhysGL 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 $v_0 =< v_0 \cos\theta, v_0 \sin\theta, 0 >$, 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 $< x_0, y_0, 0 >$. Be sure trails are left, showing your object's overall trajectory. Attach the following 4 vectors on your object: $\vec{a}$, $\vec{v}$, $\vec{v}_x$, and $\vec{v}_y$. All told, your object should have four vectors sticking out of it as it flies. ⧫ **Turn-In Ch. 9#2.2:** Paste a live share-link to your code in your document so it can be graded. ⧫ **Turn-In Ch. 9#2.3:** Put a screenshot of your ball's flight into your document.

Let's study the motion:

(a) Put the line `new_multi_graph("t","vx","vy")` somewhere above your while loop. In between the `while-end` statements, use the line `go_multi_graph(...)` to insert the instantaneous values of t, v_x, and v_y into the graph. ♦ **Turn-In Ch. 9#2.4:** Discuss the trend you see in v_x vs v_y as the ball flies, and why these quantities behave as they do. ♦ **Turn-In Ch. 9#2.5:** Put in a screenshot of your graph.

(b) Hovering over your graph or graphs to extract data points, compute what a_y must be acting on the ball. ♦ **Turn-In Ch. 9#2.6:** Report your value of a_y and show how you computed this number.

(c) When the ball encounters the green ground, it goes right through it. Can you think of how you might make the the ball bounce? (Hint: What changes about the ball's velocity when it hits the ground?) You'll need an `if` statement somewhere inside of your `while-loop`, just under your physics equations to check for the ball's encounter with the ground. Something like:

```
if $y < 0$ and $v_y < 0$ then
...
end
```

will work. ♦ **Turn-In Ch. 9#2.7:** Paste a live share link in your code, so it can be graded. ♦ **Turn-In Ch. 9#2.8:** Describe what the condition $y < 0$ and $v_y < 0$ addresses physically. Be very specific in your answer and don't just translate this condition into words.

(d) Change the launch conditions so that the ball bounces off of the ground 2-3 times. ♦ **Turn-In Ch. 9#2.9:** Include a screenshot of the trail left by this motion.

(e) Figure out what time t represents the time at about 2/3 of the way through the animation. At this time, use an `if` statement where you set the acceleration, to impose a horizontal wind (a horizontal acceleration) on your ball that points in a direction that is generally opposite to that of the ball's motion. Be sure to keep the downward gravity on the ball at all times (the wind doesn't cancel gravity, etc.). ♦ **Turn-In Ch. 9#2.10:** Include a screenshot of the trail left by this motion. ♦ **Turn-In Ch. 9#2.11:** Include a live share link to your finished PhysGL code.

3. Starting with a basic skeleton code, put the following lines at the very top

```
camera(<0,100,300>,<0,0,0>)
draw_box(<-200,0,100>,<100,-10,-100>,"red",true)
draw_torus(<90,50,0>,<0,1,0>,23,2,"yellow",true)
draw_box(<109,90,-60>,<111,40,60>,"gray",true)
draw_cylinder(<109,0,0>,<109,53,0>,7,"black",false,true)
```

This will draw a basketball hoop. Start an orange sphere at the position $\langle -100, 10, 0\rangle$. By trial and error, find v_0 and θ so that the ball (draw it as an orange sphere of radius 15) goes cleanly into the hoop with "nothing but net." Feel free to adjust the camera position if the ball goes off the screen, or to make the view more appealing. Leave trails behind the ball. ♦ **Turn-In Ch. 9#3.1:** Put a screenshot of your code and completed trajectory of your ball going through the hoop in your document. ♦ **Turn-In Ch. 9#3.2:** Interpret the code to extract any needed parameters (the ball starts at $\langle -100, 10, 0\rangle$ and the center of the hoop is at $\langle 90, 50, 0\rangle$), and now analytically (pencil and paper) solve for v_0 that would cause the shot to work, given the angle you chose in your code. Compare v_0 from the code and your hand-work. ♦ **Turn-In Ch. 9#3.3:** Put a snapshot of your hand work into your document. Clearly compare the answers at the end of your work.

4. Run the code found at `balltable.txt` which you can find at `http://goo.gl/ndMdOs` in the `2Dmotion` folder. It'll show 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, which is at a position $x = 100$, it should fall, right? But it doesn't in this animation. Why? (Hint: What is different about the acceleration of the ball on the table, vs. over the edge of the table)? Using an `if` statement in Part 4 of the code (see Section 6.5), modify this code so it will show the ball *falling* when it passed beyond the edge of the table. Be sure a_y, v_x, v_y, and $\vec{v}$ vectors are rendered on the ball throughout. Also choose an appropriate v_x so that the cube flies through the yellow hoop. The acceleration in this problem is a multi-valued function, like you've seen in calculus, that looks like this:

$$F(x) = \begin{cases} 0 & \text{if } x \leq 100 \\ -9.8 & \text{if } x \geq 100 \end{cases} \tag{9.1}$$

Such functions are naturally modeled in computers using an `if` statement. ♦ **Turn-In Ch. 9#4.1:** Include a share link to your code in your document. ♦ **Turn-In Ch. 9#4.2:** Include a screenshot of the object's motion with a good trail left behind it.

5. Take your code from 2, 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 animation should show the rocket flying in projectile motion, with its nose always pointing in the direction that it is moving. Think carefully about the direction the rocket should point. What dictates this direction? What vector is an indicator of an object's near-term motion? ♦ **Turn-In Ch. 9#5.1:** Using the "Step" button in PhysGL, grab 4 snapshots of the rocket throughout its flight and include them in your document. ♦ **Turn-In Ch. 9#5.2:** Include a screenshot of your code.

6. Start with your completed animation from problem 2 above. Suppose that when your animation is 60% done, a strong horizontal wind develops that gives the ball an a_x to the left 15 m/s^2 and upwards 3 m/s^2. You can put this into the animation by tapping into the `t` variable, which is the simulated time in your animation. In Part 4 of your skeleton code you can put

```
if t > ??? then
     a=..the acceleration with the wind included...
end
```

Note that the variable `t` is the simulated time of the animation. In the spot labeled `???`, you are to fill in the time representing about 60% through the duration of your animation. Don't forget about g throughout. ♦ **Turn-In Ch. 9#6.3:** Include a screenshot of your object's motion with a trail left behind it. ♦ **Turn-In Ch. 9#6.4:** Include a share link to your PhysGL code.

7. 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 animation (at the start of Part 5 of the skeleton code), find the angle the $\vec{v}$-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 17. 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 PhysGL. 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.

8. If you added a plane to Problem 2, did you wonder why the ball didn't bounce off of the ground? This is an interesting aspect of making computer animations: 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 possibility 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 is an abrupt change in the ball's velocity, and so should go into Part 8 of the skeleton code (see Figure 6.1). Reversing v_y can be accomplished with a redefinition of the velocity variables as in

```
vel=<vel.x,-vel.y,0>
```

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

```
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 and pos.y <= 0 then
...reverse vy
end
```

See Section 6.5.2 for more on handling such `if` statements and nuances about using a strict equality in such conditions.

9. Start with the the skeleton code found here `http://goo.gl/ndMdOs` . 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 $\vec{v}$-vector should appear to "poke you in the eye" as it passes. Pedagogical goal: motion that includes the z-axis.

10. Find code called `balltower.txt` which you can find at `http://goo.gl/ndMdOs` in the `2Dmotion` folder. 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 beyond the edge of the tower. Adjust your camera/view so that we see a nice, long, dramatic fall "into nowhere." The acceleration in this problem is a multi-valued function, like you've seen in calculus, that looks like this:

$$F(x) = \begin{cases} 0 & \text{if } x \leq 0 \\ -9.8 & \text{if } x \geq 0 \end{cases} \tag{9.2}$$

Such functions are naturally modeled on a computer using an `if` statement.

9.4 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(s) acting on the ball as it flies and describe its trajectory.

2. Back here on earth, a ball is launched through the air with 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 $\vec{a}$ and $\vec{v}$ vectors when the ball is at:

   (a) Its 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 its landing point.

4. A ball is launched straight up with $v_y = 10$ 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/contrast/discuss the two answers.

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

6. Watch your animation from last week about the object that is launched straight upward. Pay attention to the v_y vector. Now watch your animation 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 a projectile flies through the air.

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

9. Discuss: A projectile's $\vec{v}$-vector will continually change its orientation until it points more and more along the direction of the $\vec{a}$-vector.

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

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 10

Forces and Newton's Laws (Part I)

10.1 Introduction and Goals

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

- In this lesson, we'll change an object's $\vec{v}$-vector by applying one or more *forces* to an object. That is, we'll see that *forces* change $\vec{v}$-vectors.
- Demonstrate that you understand what $\Sigma\vec{F}$ means.
- Demonstrate that you understand the meaning of $\vec{F}_{\rm net}$ 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 $\Sigma\vec{F}$, 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 are related to changes in an object's $\vec{v}$-vector.

10.2 The Physics

The $\vec{v}$-vector of an object will be changed by: Applying a single force or a net-force on an object.

The goal of this chapter is to use Newton's Laws to see that accelerations actually come from forces applied to an object. Previously, $\vec{a}$ was simple a "given" quantity. It simply existed in the equations of motion and you were allowed to give it any value. Now 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, the forces we'll deal with are "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 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. The form $a = \frac{F}{m}$ is better, but 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 nature of forces. Be sure you fully understand what this last version means and how to use it. In this law $\vec{a}$ 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 The gravitational force is the weight and has magnitude $W = mg$, where m is the object's mass and g is the local acceleration due to gravity of 9.8 m/s^2. Sometimes written $\vec{F}_G$. 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 it's resistance to a change in 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 could. Would it also hurt to kick the bowling ball in the middle of outer space where $g = 0$?

Tension Tension is the tug an object feels when pulled by a rope attached to it.

Normal The normal force is the force an object feels due to touching a surface, it is always perpendicular to the surface. In general, it is *not* equal to mg.

Friction The friction force always opposes the tendency for 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. The magnitude is defined as $f = \mu N$ where μ is the coefficient of friction (see table in your book), and N is the magnitude of the normal force acting on the object.

Drag Drag is like friction in that it always acts in a direction opposite to that in which an object is moving, but comes from the fluid (air or water) through which an object might be moving. Drag, has magnitude $D = Cv^2$, 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 section is the numerator, $\Sigma \vec{F}$, because it requires three hard steps. The first, which can be difficult, is to identify all the 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). The third is to realize that $\Sigma \vec{F}$ is only useable when you break it into component form, or ΣF_x and ΣF_y. Your working equations for this section are then $a_x = \Sigma F_x/m$ and $a_y = \Sigma F_y/m$. The connection with previous material is that these accelerations, which come from forces, are the same a_x and a_y that go into the equations of motion for x and y. Thus, $x = x_0 + v_0\Delta t + (1/2)(\Sigma F_x/m)\Delta t^2$ and $v_x = v_{0x} + (\Sigma F_x/m)\Delta t$ (and similar expressions for y). 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.

10.3 Projects

1. (Theme: $\vec{a} = \Sigma\vec{F}/m$.) Start with the skeleton code on page 41. This will set a sphere to start at a position of $\langle 0, 0, 0\rangle$ with zero velocity. For this project, choose 5 random forces (in the xy-plane) to impose onto the sphere. Configure these forces in the lines above your `while` loop, in a manner perhaps like this

```
F1=<5,2,0>
F2=...
F3=...
F4=...
F5=...
```

Give the sphere some mass too in an `m=` line just below the `F5=...` line. Now, adapt your `while` loop as needed to animate the motion of the sphere given these 5 forces. You are to very explicitly compute the net force ($\Sigma\vec{F}$) then find the acceleration from that. Your `while` loop should look something like this

```
while t < 30
  Fnet=F1+F2+F3+F4+F5
  a=Fnet/m
  ...
end
```

Draw all 5 force vectors, the net force vector, and the velocity vector on the sphere at all times. Your code must contain an explicit assignment of mass as in `m=...` and the line `a=F/m` just before your physics equations. ⧫ **Turn-In Ch. 10#1.1:** Put a single screen snapshot of your sphere with all of the arrow sticking out of it, when the v-vector is nice and large. Be sure your snapshot includes your code window. ⧫ **Turn-In Ch. 10#1.2:** Include a live share-link to your finished PhysGL code. ⧫ **Turn-In Ch. 10#1.3:** Discuss the relationship you see between the net force and the evolution velocity vector.

2. (Theme: How $\vec{a} = \Sigma\vec{F}/m$ affects $\vec{v}$.)

(a) Starting with the skeleton code on page 41, configure a sphere to star near the left edge of your screen. Give it an initial v_x that is > 0 (this sends it rightward). Next, apply two x-forces to the box so that the forces cancel each other. The animation should show the subsequent motion. Be sure to draw a trail behind the box, both force vectors, the net force vector, and the $\vec{v}$-vector. When defining your acceleration, show explicit definitions of forces, `F1` and `F2`, mass (`m=...`), and a clear definition of $\vec{a}$ as in `a=(F1+F2)/m` just before your physics equations. ♦ **Turn-In Ch. 10#2.1:** Describe the behavior of the v-vector and why it behaves as it does. ♦ **Turn-In Ch. 10#2.2:** Include a live share-link to your finished PhysGL code. ♦ **Turn-In Ch. 10#2.3:** Include a screenshot that shows the your sphere with the force vectors on it and be sure yourcode window is in your snapshot as well.

(b) Repeat the above, but make the leftward force larger than the rightward force. ♦ **Turn-In Ch. 10#2.4:** Describe the behavior of the motion of the sphere and the v-vector and why they behaves as they do, in terms of the forces acting on the sphere.

(c) Repeat the above, but make the rightward force larger than the leftward force. ♦ **Turn-In Ch. 10#2.5:** Describe the behavior of the motion of the sphere and the v-vector and why they behaves as they do, in terms of the forces acting on the sphere.

3. (Theme: The role of mass.)

(a) Start with the skeleton code on page 41. Change it to give a sphere an initial position of $< -100, 0, 0 >$ and an initial velocity of $< 10, 0, 0 >$. If the the x-coordinate of the object (i.e. pos.x) becomes greater than 0, apply a force of $\vec{F} =< -10, 10, 0 >$. The line `a=F/m` must appear just above your physics equations. Somewhere above the `while` loop, give your object some mass by inserting an `m=...` line. Run the animation with $\vec{F}$ and $\vec{v}$-vectors drawn on the object at all times. Be sure a trail is left for your object. ♦ **Turn-In Ch. 10#3.1:** Include a screenshot of the completed trajectory of your object, and be sure your code window appears in your snapshot as well.

(b) Repeat the above but with $m = 10$. ♦ **Turn-In Ch. 10#3.2:** Include a screenshot of the completed trajectory of your object.

(c) ♦ **Turn-In Ch. 10#3.3:** Based on your observations of these two animations, noting that the *same force* was applied in both cases ($\vec{F} = \langle -10, 10, 0 \rangle$), describe how an object's mass affects its motion for a given applied force.

4. Start a sphere on your screen on your screen at $\langle 0, 0, 0 \rangle$. Give it an initial velocity of $\langle 0, 20, 0 \rangle$. Next, apply a series of no less than 5 sequential forces, one at a time, to keep the sphere on the screen for 30 seconds. You must apply *at least* 5 forces to the sphere. To apply forces sequentially as a function of time, you can use successive `if-then-end` statements in your `while` loop like this

```
while t < 30
 if t < 3 then
```

```
  F=<....>
 end

if t ... then
 F=<...>
end

 a=F/m
 pos = pos + ...
 vel = vel + ...
 t=t+dt
end
```

Use as many `if-end` blocks as necessary, defining forces to keep your ball on the screen. Leave a trail behind the object as it moves and draw the $\vec{F}$ and $\vec{v}$ vectors on the sphere at all times as it moves. ⧫ **Turn-In Ch. 10#4.1:** Include a live share-link to your finished PhysGL code. ⧫ **Turn-In Ch. 10#4.2:** Include a screen-shot of your graphics window after 30 seconds has elapsed. Be sure your code window appears in your snapshot as well. Your code must contain a clear definition of the sphere's mass with a `m=...` line above the `while` statement, and the line `a=F/m` just above your physics equations.

5. **The Force Maze**. Look in `http://goo.gl/ndMdOs` in the `Forces` folder and get the code called `forcemaze.txt`. If you run this script as given, you will see a green sphere moving up a left column of a maze. You can change the mass to whatever you want (in the `m=...` statement). The red bars are a maze. Your job here is to apply forces to the sphere that will steer it through the maze, so that it gently stops (or slows to a crawl) over the word "end." Your blue sphere **may not touch the red bars making the maze**. You will do this by applying a succession of forces in Part 4 of the code, that are based on the time variable `t`. The `if` statements you supply must only define a force vector, in the form `F=<Fx,Fy,0>`. Part 4 of your code should end with $a = F/m$. Draw the $\vec{F}$ and $\vec{v}$ vectors on the sphere at all times. You will likely complete this work by running your move over and over again, keeping watch on the time variable shown in the lower left corner of the screen, grabbing this number and using it in successive `if` statements in Part 4. Good luck. ⧫ **Turn-In Ch. 10#5.1:** A live share-link to your completed code so we can watch it. ⧫ **Turn-In Ch. 10#5.2:** Include a screenshot that shows the trail your object left through the maze, with your blue sphere over the word "end." Be sure your code appears in the screenshot as well.

6. **Lunar Lander**. Look online at `http://goo.gl/ndMdOs` and get the code called `lander.txt`. If you run this animation as given, 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 `m=...` statement) in Part 1 of your

code. You can apply a succession of thrusts in Part 4 of the code, that are based on the system time variable `t` with `if` statements. The `if` statements you supply must only define a force vector, in the form `F=<Fx,Fy,0>`. Somewhere above your `while` statement, insert the line `new_multi_graph("t","y","v")`, then somewhere inside of your `while` loop, insert the statement `go_multi_graph(t,pos.y,magnitude(vel))`, which will add the vertical position of your rocket and the magnitude of its v-vector into the graph. Be sure when you land, this graph will clearly show that $|v| < 1$ m/s.

It is very important in this project that you are working with forces in the spirit of $\Sigma\vec{F}$. In this case, it's important that you always have the rocket's weight acting on it, and thrusts are always *added* to this.

Draw the weight, thrust, velocity, and net-force vectors on the rocket at all times. Be sure the rocket is always oriented in the direction *opposite to* the thrust-vector (use `atan2` for this), so it looks like the rocket is going in the direction of the thrust. 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 4. Happy landing! ♦ **Turn-In Ch. 10#6.1:** A live share-link to your completed code so we can watch it. Be sure your rocket is oriented in the direction of the thrust at all times! ♦ **Turn-In Ch. 10#6.2:** Include a screenshot that shows your rocket on the landing pad. Be sure the screenshot clearly shows the graph, with the final magnitude of $\vec{v}$ upon landing, and your code window.

7. **Friction**. In this project, you'll 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).

```
draw_box(<10,-1,-10>,<30,-2,10>,"green",true)
draw_box(<10,-1,-10>,<-10,-2,10>,"brown",true)
draw_box(<-10,-1,10>,<-30,-2,-10>,"green",true)
```

This will draw two green surfaces (smooth) around a central brown surface (rough) which has friction. Next, give a box, starting off on the left green surface an initial rightward velocity. You can draw a box using

`draw_cube(vector-of-center-of-box,side-length,color)`.

Be sure you tweak the position and/or side-length of the box so that it looks like it's bottom edge is sliding along the surface.

Your block must move from left to right in this movie. The ground from $10 \le x \le 30$ and $-30 \le x \le -10$ is frictionless. From $-10 \le x \le 10$ it has friction with a coefficient μ. In Part 2 of your code, declare values for m, g, and μ. Your block should slide appropriately across the frictionless surface, then across the rough surface, then across the rightmost smooth surface. Render both $\vec{v}$ and $\vec{F}$ vectors on the block at all times and have your block leave a trail. As the block slides over the right edge of the surface, let it go into free fall and disappear forever. ♦ **Turn-In Ch. 10#7.1:** A screenshot of your completed animation showing the trail left by your object, just before it disappears from view. Be sure your code window appears in the screenshot as well. ♦ **Turn-In Ch. 10#7.2:** A live share-link to your completed code so we can watch it.

8. **Vertical spring**. Start with a new skeleton code and somewhere above the `while` statement, draw a vertical spring between $\langle 0,-40,0\rangle$ and $\langle 0,0,0\rangle$ like this:

   `draw_spring(<0,-40,0>,<?,?,?>,5,1,"yellow")`

   (Here 5 is the diameter of the spring and 1 is its thickness. The `<?,?,?>` is the moving end of the spring (which is a 3D vector in space) which you'll have to think carefully about for this work. From a position of $\langle 0,50,0\rangle$ 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 onto the spring, compress the spring to some stopping point, then be relaunched upward by the spring. Work carefully and you'll see that as not complicated as this sounds, it's really just a matter of adding two `if` statements, and setting up your forces just before your `a=Fnet/m` line inside of your `while` loop. Draw $\vec{v}$ and $\vec{F}$ 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 incorrect. ♦ **Turn-In Ch. 10#8.1:** Include a live link to your code so we can watch your animation. ♦ **Turn-In Ch. 10#8.2:** Include a screenshot of your work that includes your code. ♦ **Turn-In Ch. 10#8.3:** Include a `new_multi_graph(..)` and `go_multi_graph(..)` that makes two simultaneous plots of $F_{y,\text{net}}$ vs t and v_y vs t. Discuss trends you see in the graph and how $F_{y,\text{net}}$ and v_y seem to be related before and after the ball impacts the spring. (Do not just describe obvious features of the graphs. Focus on the physics they represent.) ♦ **Turn-In Ch. 10#8.4:** Change the k to a larger number (double it?), rerun the animation, and describe what happens. ♦ **Turn-In Ch. 10#8.5:** Make k smaller, rerun the animation, and describe what happens.

9. Start an object at position $<0,0,0>$ and moving toward the right. Apply the following force on it at the beginning of Part 4 of your code.

$$F(x)=\begin{cases}-5 & \text{if } x\geq 5\\ 5 & \text{if } x\leq -5\end{cases} \tag{10.1}$$

   Be sure Part 4 ends with a statement like `a=F/m`. Run this animation so you can fully understand how the net force affects the velocity of the object.

10. **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 to $\vec{v}$, or at 180° with respect to $\vec{v}$. So in Part 4 of the skeleton code, these steps should help in applying the drag force:

(a) Compute the magnitude of $\vec{v}$. Finding the magnitude of a vector is covered in Chapter 17 and Section 3.3.4.

(b) 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.

(c) 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 17.

(d) Find the components of the drag force as follows. Suppose this angle the $\vec{v}$-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 PhysGL. Now, you can find D_x and D_y using $D_x = D\cos(A + \pi)$ and $D_y = D\sin(A + \pi)$.

(e) Use your newly found drag force to define the net force on the object as Part 4 of your skeleton code comes to a close.

11. Start an object in the lower left corner of your screen. Start it with a speed of $v_x = 5$. After some time has gone by, in Part 4 of the code, apply a force on it in any direction you like. Here is an example that will apply a force after 3 seconds has gone by:

```
if t > 3 then
     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 different 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.

12. Start an object at position $<0,0,0>$ moving upward. Apply the following force on it at the beginning of Part 4 of your code.

$$\vec{F} = \begin{cases} <0,-5,0> & \text{if } y \geq 5 \\ <0,5,0> & \text{if } y \leq -5 \\ <0,0,0> & \text{otherwise} \end{cases} \tag{10.2}$$

Be sure Part 4 ends with a statement like `a=F/m`. Run this animation so you can fully understand what the force does to the speed of the object. Be sure $\vec{F}$ and $\vec{v}$-vectors appear on the object at all times.

13. This project is meant to "test your belief" that accelerations are what drive motion. This animation 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 animation, 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 $\vec{v}$, $\vec{a}$ and $\vec{N}$ vectors on the sphere at all times. The components of the Normal force (needed to draw the $\vec{N}$-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

    $$a_x = \frac{-y'(y''v_x^2 + g)}{1 + y'^2}, \tag{10.3}$$

    and

    $$a_y = y''v_x^2 + y'a_x. \tag{10.4}$$

    - 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 using $a =< a_x, a_y, 0 >$.
    - You'll probably have run your animation for quite a few seconds, so change the number in the `while` loop to something like `25` or so.
    - 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 animation is done, just watch it. We hope you can appreciate two things. First, watch how the $\vec{a}$ vector "struggles" to keep $\vec{v}$ moving along the track. Second, watch where $\vec{N}$ gets big and small; can you think of why it behaves like it does? There is much more to $\vec{N}$ that "is is a force perpendicular to the surface, etc."

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

14. This project is meant to "test your belief" that accelerations are what drive motion. This animation 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. This will draw a sphere at the center bottom of a parabolic track. In this animation, 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 animation must be small, say 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.

10.4 Wrap-up Questions

1. Suppose an object (at rest) has an $\vec{F}_{\text{net}}$ vector pointing up and to the left. Sketch the object's subsequent motion as time ticks onward.
2. Suppose an object has an $\vec{F}_{\text{net}}$ 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 $\vec{v}$-vector always changes orientation to try to align with the $\vec{F}_{\text{net}}$ vector.
4. Suppose $\vec{v}$ is aligned with (parallel to) the $\vec{F}_{\text{net}}$ vector. What does the $\vec{v}$-vector do now?
5. Same question as above, but for $\vec{F}_{\text{net}}$ antiparallel to $\vec{v}$.
6. Suppose an object has a $\vec{v}$-vector on it pointing to the right. The $\vec{F}_{\text{net}}$ vector on it is zero. What does the $\vec{v}$-vector do now?
7. An object has a $\vec{v}$-vector 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 $\vec{v}$-vector do now?
8. Discuss: Acceleration comes from forces.
9. In NLI.a (or d), how did the $\vec{v}$-vector change when two forces are applied to an object that cancel each other?
10. In NLI.j, describe how the $\vec{v}$-vector changes when the rocket is subjected to its weight, and then subject to (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 animations 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 11

Forces and Newton's Laws (Part II)

11.1 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 $\vec{v}$-vector by allowing forces with a clearly identifiable agent to act on an object. These *forces* will change $\vec{v}$-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.

11.2 The Physics

The $\vec{v}$-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 the following 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 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 $|\vec{v}|$ (velocity) and $|\vec{a}|$ (acceleration), although the signs of $\vec{v}$ and $\vec{a}$ 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. The tension on one side of a pulley will be the same on the other side, save for possibly opposite signs. 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 placed on sloped surface 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 $F_{\text{sp}} = -k(s - s_0)$. 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).

Gravitational force (or weight). When an object that has a mass m is in a gravitational field, it will have a force on it called weight, with magnitude $F_G = mg = W$. This force always points straight down, no matter what other orientation or situation in 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. In general the magnitude of the normal force is not equal to mg, or the weight of the object. (Sometimes it will have the same magnitude as the weight when the object is sitting on a horizontal surface and no other vertical forces are involved.) 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.

Kinetic friction. Kinetic friction is a force that is always oriented exactly opposite to an object's v and has a magnitude of $f_k = \mu_k N$, 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 (or component) F is locked at 90° with respect to an object's $\vec{v}$-vector, then this force (or component) is called a centripetal force. The force will cause the object to move in a circle of radius R if the force has a magnitude of mv^2/R, where m is the mass of the object. **Book reading: 7.4, 7.5.**

11.3 Projects

1. **Realistic incline.** You have been studying about objects that can move on an inclined surface. This project serves two purposes

   (a) Allows you to study how an object moves on an arbitrarily sloped surface.

   (b) Test your belief in Newton's Second Law, that states if you know an object's acceleration, you can predict its motion.

   In addition, this animation 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.

   To begin, you'll need to choose a mathematical function that you would like to use for your sloped surface. Choose $f(x) = 1.5 + \tan^{-1}(0.3x)$, or $f(x) = 0.05x^2$, or $f(x) = 1/(1+e^{-0.5x})$. Plot these using Wolfram Alpha and pick the one you like the most. All of them look like sloped surfaces.

   Start with a fresh copy of the skeleton code. The first thing you should do is get your function/slope displayed on the screen. To do this:

   (a) Somewhere at the very end of your code (under the final `end` statement), put your $f(x)$ into the computer like this:

   ```
   function f(x)
    return(  put your function body here )
   end
   ```

   That is, sandwich the expression (just the right hand side) of your function between the parenthesis of the `return` statement.

   (b) Next, near the top of your code, before your `while` loop starts, plot the function by inserting the line:

   ```
   plotxy(f,-15,15,0.5,0,"yellow",0.2,true)
   ```

   This will plot your function called `f` from $x = -15..15$ in steps of 0.5, with a thickness of 0.5, in a yellow color. The `true` says to keep it on the screen. Feel free to change any of these parameters.

   Next, start a sphere on the leftmost portion of your curve, $\langle x_0, f(x_0), 0\rangle$. Start your sphere with a velocity of $\vec{v} =< v_{x0}, 0, 0 >$, where v_{x0} is something small, like 3 or 5.

   To successfully negotiate your curve, you'll need to form $\vec{a}$ of your sphere. The x and y accelerations are:

$$a_x = \frac{-f'(f''v_x^2 + g)}{1 + f'^2}, \tag{11.1}$$

and

$$a_y = f'' v_x^2 + f' a_x, \tag{11.2}$$

where the primes mean derivatives of your function with respect to x, and $g = 9.8$ m/s^2. If you believe in Newton's Laws, then these equations for a_x and a_y should be all you need to complete this project. That is, these two components of acceleration should be all you need to show your sphere properly negotiating the curve you have chosen. Be sure to draw $\vec{v}$, $\vec{a}$, and $\vec{N}$ on your sphere at all times. Note that $\vec{N} = \langle ma_x, mg + ma_y, 0\rangle$. Ultimately, choose v_{x0} so that your sphere does not quite make it up "over the hump" of your curve, will stop somewhere on it, and fall back down again.

♦ **Turn-In Ch. 11#1.1:** A live link to your animation so we can watch it. It must show the sphere going up your curve, stopping, then sliding back down again. ♦ **Turn-In Ch. 11#1.2:** A screenshot of your entire PhysGL work area, including your graphics screen and code. ♦ **Turn-In Ch. 11#1.3:** Use `new_multi_graph(...)` and `go_multi_graph(...)` to produce a stack of vertical graphs of v vs t, a vs t, and $|N|$ vs t. ♦ **Turn-In Ch. 11#1.4:** From your graph, discuss as a roller coaster designer, where you would 1) need to fortify the track to support large normal forces and why? 2) If someone was riding on the sphere, where would they feel "butterflies" in their stomachs? 3) Where would they feel heavy?

2. A sphere starts at the top of the screen, and is in "free fall." Add an upward drag force on the ball of the form Cv^2, where C is a constant (the drag coefficient) and v is the instantaneous speed of the ball. Use Wikipedia to find a suitable value for C. Use `new_multi_graph(...)` and `go_multi_graph(...)` to produce plots of F_{net} vs. t, y vs t, and v_y vs. t. Choose C and tweak other parameters in a manner that will allow you to see the ball clearly reaching a terminal velocity. ♦ **Turn-In Ch. 11#2.1:** Include a live link so we can watch your animation. ♦ **Turn-In Ch. 11#2.2:** A screenshot of your whole PhysGL work area (code, graphics screen,and graphs). ♦ **Turn-In Ch. 11#2.3:** Do a piecewise interpretation of your y and v plots, making a careful interpretation of the slopes and trends you see in each. Be certain to carefully describe the relation between F_{net} and v as your object falls.

3. **Friction**. In this project, you'll 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).

```
draw_box(<10,-1,-10>,<30,-2,10>,"green",true)
draw_box(<10,-1,-10>,<-10,-2,10>,"brown",true)
draw_box(<-10,-1,10>,<-30,-2,-10>,"green",true)
```

This will draw two green surfaces (smooth) around a central brown surface (rough) which has friction. Next, give a box, starting off on the left green surface an initial rightward velocity. You can draw a box using `draw_cube(vector-of-center-of-box,side-length,color)`.

Be sure you tweak the position and/or side-length of the box so that it looks like it's bottom edge is sliding along the surface.

Your block must move from left to right in this movie. The ground from $10 \le x \le 30$ and $-30 \le x \le -10$ is frictionless. From $-10 \le x \le 10$ it has friction with a coefficient μ. In Part 2 of your code, declare values for m, g, and μ. Your block should slide appropriately across the frictionless surface, then across the rough surface, then across the rightmost smooth surface. Render both $\vec{v}$ and $\vec{F}$ vectors on the block at all times and have your block leave a trail. As the block slides over the right edge of the surface, let it go into free fall and disappear forever. ♦ **Turn-In Ch. 11#3.1:** A screenshot of your completed animation showing the trail left by your object, just before it disappears from view. Be sure your code window appears in the screenshot as well. ♦ **Turn-In Ch. 11#3.2:** A live share-link to your completed code so we can watch it.

4. **Friction**: Start with the code `frict_spring.pov` at `http://goo.gl/ndMdOs` . 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 ??. The brick wall starts at $x = -5$ and extends toward the left. $\vec{v}$ and $\vec{F}$ vectors on the block at all times.

5. **Friction**: Start with the code `frict_hang.pov` at `http://goo.gl/ndMdOs` . 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 layout of the objects in the scene is shown in Figure 11.1. Draw $\vec{v}$ and $\vec{F}$ vectors on the block at all times.

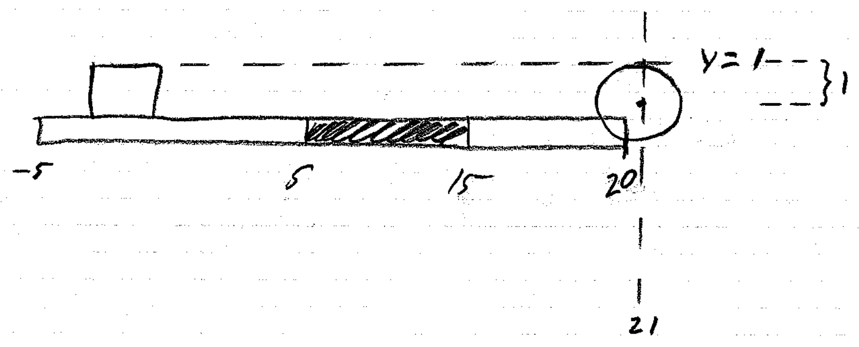


Figure 11.1: Spatial outlay of objects for NLII.d

6. **Friction**: In this animation you show how a block is slowed by friction. Start with the skeleton code (online) called `friction.pov` which has these lines in Part 10 (after the last `end` statement).

   ```
   draw_box(<-15,-2,-5>,<-10,0,5>,"green")
   draw_box(<-10,-2,-5>,<-5,0,5>,"tan")
   draw_box(<-5,-2,-5>,<5,0,5>,"green")
   draw_box(<5,-2,-5>,<15,0,5>,"darkgrey")
   draw_box(<15,-2,-5>,<25,0,5>,"green")
   draw_box(<23,0,-2>,<25,2,2>,"peru")
   draw_cylinder(<-20,-5,0>,<-20,-6,0>,3,"yellow")
   ```

   These instructions draw smooth green surfaces with a section of gray metal between $5 \leq x \leq 15$ and a section of tan wood between $-10 \leq x \leq -5$. The leftmost edge is at $x = -15$. Start a box at $x_0 =< 24, 0, 0 >$, with some initial $v_0 =< -v_x, 0, 0 >$ toward the left. The box is also colored a shade of brown to suggest wood. You can find help drawing boxes in Section 5.3.

   Your block must move from right to left in this animation. The metal has a coefficient of friction $\mu_{\rm m}$ and the wood $\mu_{\rm w}$. 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 patch of 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 $\vec{F}$ and $\vec{v}$ vectors on the block at all times. Choose appropriate values so the box falls through the center of the yellow disk just off the left edge.

7. **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 $\vec{v}$ and $\vec{F}$ 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, via the Hooke's law force, changes an object's $\vec{v}$-vector.*

   You can find help with springs in Chapter 17. 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.

8. **Vertical spring**: Start with a new skeleton code. In it somewhere above the `while` statement, draw a vertical spring between $\langle 0,-40,0\rangle$ and $\langle 0,0,0\rangle$ like this:

   `draw_spring(<0,-40,0>,???,5,1,"yellow")`

   (Here 5 is the diameter of the spring and 1 is its thickness. The `???` is one end of the spring (which is a 3D vector in space) which you'll have to think carefully about for this work. From a position of $\langle 0,50,0\rangle$ 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 not complicated as this sounds, it's really just a matter of adding two `if` statements setting up your forces, just before your `a=Fnet/m` line inside of your `while` loop. Draw $\vec{v}$ and $\vec{F}$ 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. ♦ **Turn-In Ch. 11#8.1:** Include a live link to your code so we can watch your animation. ♦ **Turn-In Ch. 11#8.2:** Include a screenshot of your work that includes your code. ♦ **Turn-In Ch. 11#8.3:** Include a `new_multi_graph(..)` and `go_multi_graph(..)` that makes two simultaneous plots of F_{net} vs t and v_y vs t. Discuss trends you see in the graph and how F_{net} and v seem to be related before and after the ball impacts the spring. (Do not just describe obvious features of the graphs. Focus on the physics they represent.) ♦ **Turn-In Ch. 11#8.4:** Change the k to a larger number (double it?), rerun the animation, and describe what happens. ♦ **Turn-In Ch. 11#8.5:** Make k smaller, rerun the animation, and describe what happens.

9. **Atwood Machine**: Start with the skeleton code. An Atwood Machine can be drawn as shown in Figure 11.2.

   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 7 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

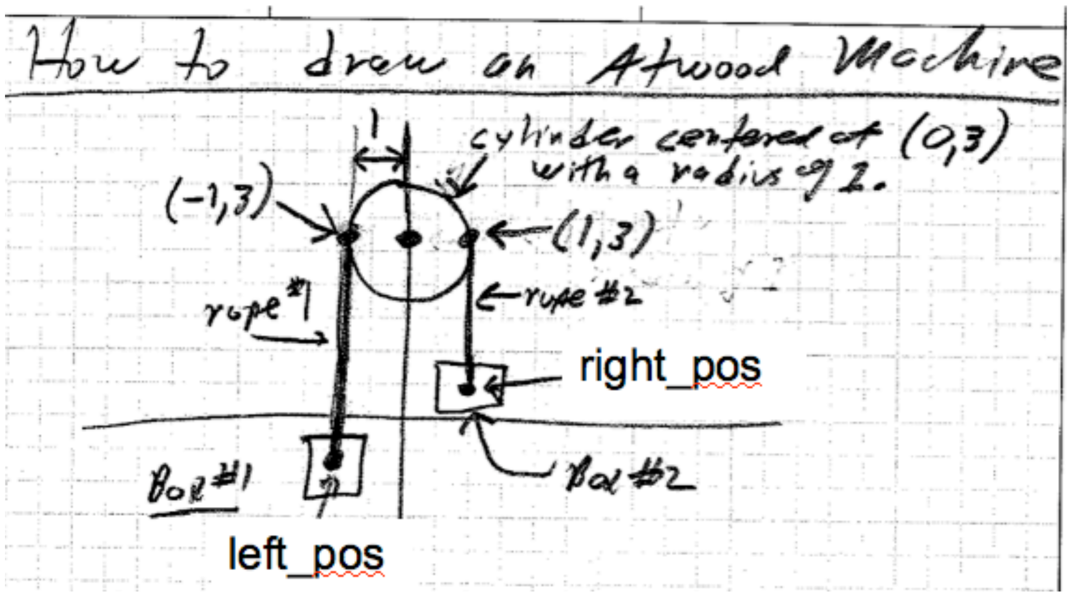

How to draw an Atwood Machine
1
cylinder centered at (0,3) with a radius of 1.
(-1,3)
(1,3)
rope #1
rope #2
right_pos
Box #1
Box #2
left_pos

Figure 11.2: A guide in helping you to draw an Atwood machine using PhysGL.

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 object appear larger. The acceleration of this system will be derived in class. Render the $\vec{v}$ and $\vec{F}$ 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 animation, try to understand how the blocks, coupled by the rope, change each others' $\vec{v}$-vectors.*

10. 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.

11.4 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 $\vec{v}$-vector?
5. How does a spring change on object's $\vec{v}$-vector?
6. How does "the other mass" change a given mass's $\vec{v}$-vector in the Atwood machine?
7. How does the slope of a surface change on object's $\vec{v}$-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 12

Energy: Kinetic, Potential, Conservation, and Work

12.1 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 $\vec{v}$-vector by allowing *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 in the absence of friction, $KE + PE =$ 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.

12.2 The Physics

The $\vec{v}$-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), along with a guiding principle, called "conservation of energy." The units of energy (KE, PE, and W) are Joules, or J.

KE is the energy something (of mass m) has because it is moving with some speed v, or $K = \frac{1}{2}mv^2$. If an object is moving it has KE; if it is at rest, it doesn't. PE is stored energy that has not yet been released to do something.

Our society is usually concerned with chemical or nuclear PE (oil/gasoline, natural gas, nuclear materials), but in this class we'll only concern ourselves with mechanical PE, and further, only three types of it. Gravitational PE, or $U_g = mgy$, spring PE, or $U_{\rm sp} = \frac{1}{2}k(x - x_n)^2$, and pendulum PE, $U_{\rm pend} = mg(1 - \cos\theta)$.

U_g is the 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 it be able to fall down any farther if given a small nudge?" In U_g, mg is the weight of the object and y is its vertical distance above the lowest possible position.

$U_{\rm sp}$ is the energy a compressed or extended 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 $U_{\rm sp} = 0$. If $x > x_n$ or $x < x_n$ then $U_{\rm sp} \neq 0$.

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 $U_{\rm pend}$ 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, $W = F\Delta r\cos\theta$, where F is the magnitude of the 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 in the absence of friction. This means if PE goes up, KE must go down and if PE goes down, KE must go up, both in such a way as to keep $KE + PE =$ a constant. The "constant" is the total energy of the system. The most useful form of this law is 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 fill in the appropriate PE function(s) forms 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**.

12.3 Projects

In these projects, you'll be having the computer calculate energies, like KE, PE and E. These can be handled inside of your `while` loop with lines like (assuming your velocity is kept in variable `vel` and your position is kept in variable `pos`.

```
KE=0.5*m*vel^2
PE=m*g*pos.y
E=KE+PE
```

Note here the first line computes the magnitude of the velocity vector. The lines below that should be self explanatory.

1. Using your projectile (2D) motion movie from a few weeks ago, let's add an energy analysis to it. Get that code out and be sure it still works. Change the name of it, so a new share link will be created (preserving the old one). Next, somewhere in the while loop, put in lines to compute kinetic energy, gravitational potential energy, and total energy. Use the variables E, KE, and PE for this. Somewhere above the `while` loop add the line `new_graph("t","E","KE","PE")` to start a new graph with time (t) on the x-axis, and E, KE, and PE on the y-axis. Somewhere in your `while` loop, add the line `go_graph(t,E,KE,PE)`. This will make a plot of your energies as a function of time.

   (a) ⧫ **Turn-In Ch. 12#1.1:** Turn in a live link to your work so we can watch it. ⧫ **Turn-In Ch. 12#1.2:** Turn in a screenshot of your work area, including your code and the graph.

   (b) What is it from the graph that leads to conclude that total energy is conserved? ⧫ **Turn-In Ch. 12#1.3:** Again from the graph, describe the interaction between KE and PE that leads to E=constant. In your answer, link the KE/PE interaction to the motion of the object that you see.

   (c) Change the name of your project (then click Save) to generate a new share link (and to preserve the old one you used in the Turn-in above). Next, remove the `new_graph` and `go_graph` lines. In the `while` loop (under your energy calculations), add the line

   ```
   bar_graph("Quantity","Energy","Total",E,"Kinetic",KE,"Potential",PE)
   ```

   This will create a bar-graph whose x-axis is labeled "Quantity" and y-axis is labeled "Energy." It'll add bars to the graph, with labels `Total`, `Kinetic`, and `Potential` and feed the quantities E, KE and PE to the graph to form the bar heights. ⧫ **Turn-In Ch. 12#1.4:** Describe the behavior of the bars, as it relates to the motion you see in the animation. ⧫ **Turn-In Ch. 12#1.5:** What do you see in the bar graph that would indicate that "energy is conserved?"

2. Get your vertical spring movie from a couple of weeks ago. Add total energy E, KE, spring potential energy PE_s, and gravitational PE_g calculations inside of the `while` loop (use the variables `E`, `KE`, `PEs` and `PEg`). Somewhere above the `while` loop, add the line `new_graph("t","E","KE","PEs","PEg")` to start a new graph with time (t) on the x-axis, and E, KE, PE_s and PE_g on the y-axis. Somewhere in your `while` loop (after your energy calculations), add the line `go_graph(t,E,KE,PEs,PEg)`. This will make a plot of your energies as a function of time. ⧫ **Turn-In Ch. 12#2.1:** Turn in a live link to your work so we can watch it. ⧫ **Turn-In Ch. 12#2.2:** Turn in a screenshot of your work area, including your code and the graph.

(a) ⧫ **Turn-In Ch. 12#2.3:** What is it from the graph that leads to conclude that total energy is conserved? ⧫ **Turn-In Ch. 12#2.4:** Again from the graph, describe the interaction between KE, PE_g and PE_s that leads to E=constant. ⧫ **Turn-In Ch. 12#2.5:** What happens to the spring potential energy as KE drops? ⧫ **Turn-In Ch. 12#2.6:** What happens to spring potential energy as KE rises?

(b) Change the name of your project (the click Save) to generate a new share link (and to preserve the old one you used in the Turn-In above). Next, remove the `new_graph` and `go_graph` lines. In the `while` loop (under your energy calculations), add the line

```
bar_graph("Quantity","Energy","Total",E,","Kinetic",KE,"PEg",PEg,"PEs",PEs)
```

This will create a bar-graph whose x-axis is labeled "Quantity" and y-axis is labeled "Energy." It'll add bars to the graph, with labels `Total`, `Kinetic`, `PEg`, and `PEs` and feed the quantities E, KE and PE_g and PE_s to the graph to form the bar heights. ⧫ **Turn-In Ch. 12#2.7:** Describe the behavior of the bars, as it relates to the motion you see in the animation. ⧫ **Turn-In Ch. 12#2.8:** What do you see in the bar graph that would indicate that "energy is conserved?" ⧫ **Turn-In Ch. 12#2.9:** How, in particular, do the kinetic and spring PE bars relate?

3. Find your code from a couple of weeks ago that involved a block sliding across a patch of ground with friction. Make sure the code still runs. Add the computation of KE, ΔE_{th}, and E_{Tot} inside the loop. Notice that ΔE_{th} is zero until the friction starts to do work. Within the `if` statement related to the friction, you will need to add a calculation of the work done by friction, $W_f = \vec{f_k} \cdot \Delta\vec{x} = -\Delta E_{th}$. The friction should already be in your code, but you will have to be careful with the dot product because it points in the direction opposite the path, meaning the dot product has a factor of $\cos 180°$. Also, the magnitude of the path needs to be calculated as the distance traveled across the rough patch, Δx = position of the box - start of the rough patch. Plot the energy variables using `new_graph()` and `go_graph(t,KE,Eth,Etot)`. ⧫ **Turn-In Ch. 12#3.1:** A screenshot of your code and your plot and explain how energy is conserved for the block. Explain where all the energy starts and where it ends up.

4. Find your code from a couple of weeks ago that involved a block sliding across friction. Get rid of the frictional force part in the `if` statement (keep the explicit force on the block to be `F=<0,0,0>` throughout. 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 *below* your physics equations that abruptly changes the *velocity*.

   Right off, make these additions to you code:

   - Under your physics equations, compute the kinetic energy of the block and put it in a variable called `KE`.
   - Under your calculation of KE, set a variable called `Wf` (or work due to friction) equal to zero.

- Somewhere above your while loop, set a variable called `Eth` (thermal energy) equal to zero.
- Put a line like this `pos0=pos` just before your physics equations. (This will save the current position of the block into 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, start an `if-then` 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." Assuming your kinetic energy is in a variables called `KE`, this pseudo-code will demonstrate energy lost due to friction.

```
$W_f \leftarrow -\mu mg|x-x0|$ (compute work due to friction; this is $W_f=-f\Delta x$.)
$E_{th} \leftarrow E_{th}+W_f$ (run a tally of the energy that continually goes into heating the block.)
$KE_1 \leftarrow KE+W_f$ (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)
```

(a) Let's graph the energies of the block as it slides. Set up a graph using `new_graph("x","Etot","KE","Eth")` (above the `while` loop) and `go_graph(pos.x,Etot,KE,Eth)` (inside of the `while` loop) to watch the total, kinetic energy and thermal energy of the block. Be sure your block makes it across the frictional patch, and stops just as it hits the far edge of the platform. ♦ **Turn-In Ch. 12#4.1:** Turn in a live link to your code, so we can watch it. ♦ **Turn-In Ch. 12#4.2:** Turn in a clear screenshot of your graph. ♦ **Turn-In Ch. 12#4.3:** By looking at your graph, tell us the energy-conservation situation that seems to be happening. ♦ **Turn-In Ch. 12#4.4:** What does friction do to a moving object from an energy perspective? ♦ **Turn-In Ch. 12#4.5:** What does it mean physically, that you set $E_{th}=0$ above your while loop (before the block started moving)? ♦ **Turn-In Ch. 12#4.6:** What is the `pos0` variable needed for (answer in terms of the physics we are trying to use).

(b) Your book works with energy "bar charts." Let's use them here too. Change the name of your code (which will generate a new share link). Get rid of the `go_graph(..)` and `new_graph(..)` lines and add the line

`bar_graph("Quantity","Energy","KE",KE,"Wf",Wf,"Eth",Eth)`

somewhere in the `while` loop. ♦ **Turn-In Ch. 12#4.7:** Watch the bar-graph as the block slides. Describe the nature of the "work" and E_{th} bars, and what happens to the kinetic energy when the block is over the friction. ♦ **Turn-In Ch. 12#4.8:** Does energy seem to be conserved even though the block is losing energy due to friction? How can this be? ♦ **Turn-In Ch. 12#4.9:** Describe both when the block is not over the friction. ♦ **Turn-In Ch. 12#4.10:** Turn in a live link to your code, so we can watch your bar chart.

5. Using your arctan movie, 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.

6. Using your friction movie, 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.

7. 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 $v_{x0} \neq 0$ 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

   ```
   ypp=pos.x*pos.x/pow(100-pos.x*pos.x,1.5)+1/sqrt(100-pos.x*pos.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.

8. 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.po` 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.

9. Take your basketball movie from a few weeks back and add KE, PE, E bars to it.

12.4 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 13

Momentum and Conservation of Momentum

13.1 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 $\vec{v}$-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.

13.2 The Physics

The $\vec{v}$-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 forces that objects exert on each other are always the same magnitude.** 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 magnitude 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, $\vec{F}_{AB} = -\vec{F}_{BA}$.

It doesn't matter if 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 magnitude of the force the Honda exerts on the SUV will be equal to the magnitude of 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 experiences due to 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 $a = F/m = 0.1\text{ N}/1 \times 10^{-6}\text{ kg} = 100,000\text{ m/s}^2$. The SUV, with a mass of about $4,000$ kg gets an acceleration of $a = 0.1\text{ N}/4000\text{ kg} = 0.000025\text{ m/s}^2$.

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 in the absence of external forces that $\vec{p}_{\text{before}} = \vec{p}_{\text{after}}$. 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 $\Sigma\vec{p}_{\text{before}} = \Sigma\vec{p}_{\text{after}}$, 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 $m_1\vec{v}_{1\text{before}} + m_2\vec{v}_{2\text{before}} + m_3\vec{v}_{3\text{before}} + ... = (m_1 + m_2 + m_3 + ...)\vec{v}_{\text{after}}$. Notice that there's only one velocity after the collision ($\vec{v}_{\text{after}}$) because only the "big blob" is moving after they all collided and stuck together. For an elastic collision, where two objects (1 and 2) collide along a single axis, we'll have $v_{1\text{after}} = \frac{m_1-m_2}{m_1+m_2}v_{1\text{before}} + \frac{2m_2}{m_1+m_2}v_{2\text{before}}$ and $v_{2\text{after}} = \frac{2m_1}{m_1+m_2}v_{1\text{before}} + \frac{m_2-m_1}{m_1+m_2}v_{2\text{before}}$.
Book reading: 9.1, 9.2, 9.3., 9.4, 10.6.

13.3 Projects

1. Start two spheres (1 and 2), of mass m_1 and m_2 moving toward each other. Draw the radii of each sphere to be proportional to its mass. One starts near the left of the screen and moves rightward. The other starts near the right of the screen and moves leftward. Both should be moving at constant speed. Keep all motion along the x-axis. When they collide, compute the impulse, $\vec{J}$ that moderates the collision. Note that $\vec{J}$ is a full vector, which is $\vec{J} = \Delta\vec{p}$, or $\vec{J} = \vec{p}_{1f} - \vec{p}_{2f}$. In this case, we'll compute $\vec{J}$ from $\vec{J} = (1+e)m_1m_2(\vec{v}_{1i} - \vec{v}_{2i})/(m_1+m_2)$, where e is the coefficient of restitution, which is a number *between* 0 and 1 (say like $e = 0.7$). Your animation should show the entire collision, from approach, to collision, to motion after the collision. Your code must explicitly apply $\vec{J}$ to each ball to moderate the collision. Make your dt smaller if the balls appear to collide before they touch. Draw the momentum vector on both objects at the same time. The collision can be detected by seeing if the centers of the two balls are within the run of their two radii and are moving toward each other.

   In this work, we want to study the momentum as it unfolds throughout the animation. Somewhere above your `while` loop, place the line

   ```
   new_graph("t","ptotal","p1x","p2x")
   ```

   to start a new graph with time on the x-axis, and the total momentum (ptotal), p_{1x} and p_{2x} on the y-axis. Somewhere inside of your `while` loop, compute the momentum of each ball and call them `pa` and `pb`. Also compute the total momentum `ptotal`. Put the line

   ```
   go_graph(t,magnitude(ptotal),p1.x,p2.x)
   ```

   somewhere inside of your `while` loop. ♦ **Turn-In Ch. 13#1.11:** A live link to your animation so we can watch it. ♦ **Turn-In Ch. 13#1.12:** A screenshot, that clearly shows your code and graph window.

   (a) ♦ **Turn-In Ch. 13#1.13:** Run your animation for m_a larger than m_b (sphere A should appear larger than B). ♦ **Turn-In Ch. 13#1.14:** Include a screenshot of the graph produced. ♦ **Turn-In Ch. 13#1.15:** From the graph, comment if you think momentum is conserved.

   (b) ♦ **Turn-In Ch. 13#1.16:** Run your animation for m_a less than m_b (sphere A should appear smaller than B). ♦ **Turn-In Ch. 13#1.17:** Include a screenshot of the graph produced. ♦ **Turn-In Ch. 13#1.18:** From the graph, comment if you think momentum is conserved.

   (c) Set $e = 1$. This is a perfectly elastic collision, or a collision where energy is conserved. Run your code to replicate the "three cases" described on page 266 (bottom) of your book. ♦ **Turn-In Ch. 13#1.19:** Clearly and completely discuss what you see both in the animation and in the graph for each case. Be sure to discuss why each collision proceeds as it does for each case. Next, put a `new_graph("t","Etot","KE1","KE2")` somewhere above your `while` loop. Somewhere inside of your `while` loop, put the line `go_graph(t,Etot,KE1,KE2)`, where

`KE1` and `KE2` are the kinetic energies of each sphere, and `Etot` is the total energy of the two (`Etot=KE1+KE2`). Run your code and watch the graph. ♦ **Turn-In Ch. 13#1.20:** Include a screenshot of your work area that clearly shows your graph. ♦ **Turn-In Ch. 13#1.21:** Describe what you see on the graph before and after the collision that would indicate energy is conserved. Be sure to describe the changes in the kinetic energies of the sphere, and the trends there in that seems to keep the total energy constant.

2. Start with a fresh skeleton code. 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 (scale a sphere's radius with its mass). When they collide, compute the the outgoing velocities using the results from elastic collision theory (see the equations at the end of Section 13.2 in this text). You are not applying forces or accelerations to your spheres in this work. You are abruptly changing the velocities of the spheres in Part 8 of the skeleton code, as shown in on page 44 in Figure 6.1. Draw the momentum vectors on the balls at all times. Somewhere in your `while` loop, add the line (assuming you call the momentum of sphere 1 `p1` and that of sphere 2, `p2`):

```
bar_graph("Quantity","Momentum","Total",p1.x+p2.x,"Sphere 1",p1.x,"Sphere 2",p2.x)
```

This will create a bar-graph whose x-axis is labeled "Quantity" and y-axis is labeled "Momentum" It'll add bars to the graph, with labels `Total`, `Sphere 1`, and `Sphere 2` and feed the quantities $p_{1x}+p_{2x}$, p_{1x} and p_{2x} to the graph to form the bar heights. Your animation should show the two balls approach each other, collide, and react to the collision based on the results for an elastic collision. The bar graph should look different before and after the collision. ♦ **Turn-In Ch. 13#2.1:** A live link to your code so we can watch it. ♦ **Turn-In Ch. 13#2.2:** A screenshot of your work area including the code, graphics area and the bar-graph. ♦ **Turn-In Ch. 13#2.3:** Watch the bar graph before and after the collision. Describe the total momentum before and after the collision. Is momentum conserved? ♦ **Turn-In Ch. 13#2.4:** Again watching the bar graph, describe the momentum of each ball before and after the collision. What happens to them? ♦ **Turn-In Ch. 13#2.5:** How does what happens to the bars for each sphere relate to what you see in the animation? ♦ **Turn-In Ch. 13#2.6:** What does `p1.x+p2.x` mean in the `bar_graph` line above? ♦ **Turn-In Ch. 13#2.7:** What relationship always seems to hold between the 3 bars in the bar graph? ♦ **Turn-In Ch. 13#2.8:** Why? ♦ **Turn-In Ch. 13#2.9:** Run your code to replicate the "three cases" described on page 266 (bottom) of your book. Clearly discuss what you see both in the animation and in the bar graphs for each case. ♦ **Turn-In Ch. 13#2.10:** Get rid of the `bar_graph` statement, and put a `new_multi_graph("t","v1","v2")` line somewhere above your while loop. Next, put a `go_multi_graph(t,vel1.x,vel2.x)` somewhere inside of your while loop. Run your animation and study the velocity vs. time graphs that come up for both objects. Describe how the graphs relate to what you see before, during and after the collision.

3. Repeat the last problem, but use an *inelastic collision*, where the two balls stick together. Sticking together can be accomplished simply by assigning the balls the same velocity (as predicted by the inelastic theory) after the collision. ♦ **Turn-In Ch. 13#3.11:** A live link to your code so we can watch it. ♦ **Turn-In Ch. 13#3.12:** A screenshot of your work area including the code, graphics area and the bar-graph. ♦ **Turn-In Ch. 13#3.13:** Watch the bar graph before and after the collision. Describe the total momentum before and after the collision. Is momentum conserved? ♦ **Turn-In Ch. 13#3.14:** Again watching the bar graph, describe the momentum of each ball before and after the collision. What happens to them? ♦ **Turn-In Ch. 13#3.15:** How does what happens to the bars for each sphere relate to what you see in the animation? ♦ **Turn-In Ch. 13#3.16:** What does `p1.x+p2.x` mean in the `bar_graph` line above? ♦ **Turn-In Ch. 13#3.17:** What relationship always seems to hold between the 3 bars in the bar graph? ♦ **Turn-In Ch. 13#3.18:** Why? ♦ **Turn-In Ch. 13#3.19:** Get rid of the `bar_graph` statement, and put a `new_multi_graph("t","v1","v2")` line somewhere above your while loop. Next, put a `go_multi_graph(t,vel1.x,vel2.x)` somewhere inside of your while loop. Run your animation and study the velocity vs. time graphs that come up for both objects. Describe how the graphs relate to what you see before, during and after the collision.

4. Download the code `elastic_coll.pov` from `http://goo.gl/ndMdOs` . 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 13.2**

5. 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.

6. 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.

7. 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.

8. 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 5.3. 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 13.2**

9. 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.

13.4 Wrap-up Questions

1. It is critical in collision movies 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 $m_1 < m_2$.

Chapter 14

Rotational Motion

14.1 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 the moment of inertia.
- Demonstrate that you understand torque.
- Demonstrate that you understand the corresponding "a=F/m" for rotations, which is $\alpha = \tau/I$.

14.2 The Physics

The $\vec{v}$-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 now know about straight line motion (x, y, F, etc.), many 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 $\theta \rightarrow \theta_0$ and $\omega \rightarrow \omega_0$ 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 in 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 and the same. To close, θ, ω, and α are the parameters that allow you to *observe* an object rotating. You'll see it rotate an angle (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**.

14.3 Projects

Project Descriptions

1. Let's make a wheel rotate on the screen and study the interaction between θ, ω, and α. To start, load code that looks like this into your PhysGL code window:

```
theta=0
omega=0
alpha=0

dt=0.1
t=0
while t<20 animate

  draw_cylinder(<0,0,5>,<0,0,-5>,35,"checker01",false)

  //theta=....
  //omega=...
  t=t+dt
end
```

Note the syntax of `draw_cylinder` is to supply two vectors giving the two desired *end-points* for the cylinder. After the end points comes the radius, then a color (we use a checkered texture so we can see it rotating), followed by a `true` or `false` as to whether or not we want the cylinder to have caps on it or not. Run the code and look at the cylinder.

To make the cylinder rotate, you'll need to uncomment and fill in the `theta=...` and `omega=...` lines so that the rotation steps forward with the kinematic equations. Add the `rotate(theta)` function just prior to the `draw_cylinder` call so that the cylinder rotates. ♦ **Turn-In Ch. 14#1.1:** Discuss how this code is similar to the skeleton code you've been using all quarter.

(a) In this part, try to set ω to control the rotation. Is the wheel rotating CCW or CW? Make the wheel rotate clockwise and then make it rotate counter clockwise.

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 rotation axis, which in this case is along the z axis (given that the wheel face is in the xy plane). Use `draw_label_vector` and clearly label the ω and α vectors. Add `new_multi_graph(...)` and `go_multi_graph(...)` lines to your code (as needed), to make simultaneous plots of θ, ω, and α as a function of time. ♦ **Turn-In Ch. 14#1.2:** Turn in a live link so we can watch your wheel too. ♦ **Turn-In Ch. 14#1.3:** Include a screenshot of your work, including the code, graphics and your graph. ♦ **Turn-In Ch. 14#1.4:** Discuss the trends you see in each graph (discuss in terms of physics; do not write words that simply describe the shape of the graphs).

(b) Repeat the above, but make the wheel go counterclockwise for ω =constant. Explain how you control the direction of the rotation. ♦ **Turn-In Ch. 14#1.5:** Include a screenshot of your graph (as described above).

(c) Repeat the above, but with $\omega > 0$ and $\alpha > 0$. ⧫ **Turn-In Ch. 14#1.6:** Include a screenshot of your graph (as described above).

(d) Rename your project to generate a new share link. Same as above, 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. ⧫ **Turn-In Ch. 14#1.7:** Include a screenshot of your graph (as described above). ⧫ **Turn-In Ch. 14#1.8:** Include a live share link to this animation.

(e) Same as above, but make it spin along the $y-$axis, instead of the z-axis.

2. Let's examine now to make an object roll without slipping. Start with a fresh copy of the skeleton code that makes a cylinder with a radius of $R = 10$ pixels move across the screen like this (note the usage of the `draw_cylinder` function in the problem above):

```
camera(<50,20,250>,<0,0,0>)
pos=<-70,0,0>
vel=<5,0,0>
dt=.5
t=0
R=10
draw_plane(<0,1,0>,0,"green",150,true)
draw_box(<75,0,-75>,<70,50,75>,"brick",true)
while t<100 animate
        a=<0,0,0>
        draw_cylinder(<pos.x,R,5>,<pos.x,R,-5>,R,"checker01")
        pos=pos+vel*dt+0.5*a*dt^2
        vel=vel+a*dt
        t=t+dt
```

Choose an acceleration for the cylinder that will just make it stop and turn around before hitting the brick wall, whose close edge is at $x = 70$.

(a) If an object is to roll without slipping, it's rotational speed, ω must be equal to $|\vec{v}_t|/R$. Use this fact and an appropriate `rotate(...)` line just above your `draw_cylinder(...)` statement that will cause the cylinder to rotate as it moves (i.e. rotate). It should look like a real rolling object as it moves. Draw the ω vector on the object at all times (tail at the center of the cylinder, `pos`, and extending out along the z-axis). ⧫ **Turn-In Ch. 14#2.1:** Include a live link to your code so we can watch it. ⧫ **Turn-In Ch. 14#2.2:** Include a screenshot of your code and graphics window showing the moving cylinder. ⧫ **Turn-In Ch. 14#2.3:** Make the object's speed (i.e. `vel`) twice as fast. What seems to happen to the rotational speed as it moves? ⧫ **Turn-In Ch. 14#2.4:** Make the object's speed twice as slow. What seems to happen to the rotational speed as it moves? ⧫ **Turn-In Ch. 14#2.5:** For a given `vel`, double the radius of the cylinder. What seems to happen to its rotational speed? ⧫ **Turn-In Ch. 14#2.6:** For a given `vel`, halve the radius of the cylinder. What seems to happen to its rotational speed?

(b) When your cylinder turns around, its rotational motion won't look right. Think carefully about your logic, to be sure the rotation direction of the object reverses direction too. This is a lesson in the signs involved with ω. ⧫ **Turn-In Ch. 14#2.7:** Include a live share link to your code so we can watch it. ⧫ **Turn-In Ch. 14#2.8:** What did you do to make the object rotate in the proper direction after it turned around?

3. Go the `http://goo.gl/ndMdOs` 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`.

4. 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 $\vec{\omega}$-vector on both cylinders at all times.

14.4 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 15

Torque, Angular Acceleration and Momentum

15.1 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.

15.2 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 $\alpha \neq 0$, 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 $\theta \neq \theta_0$) 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**.

15.3 Projects

1. 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.

2. 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.

3. 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.

4. 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>
}
}
```

5. 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 $L = rmv$)

$$L_{block} = Dp = Dmv, \tag{15.1}$$

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,

$$L_{rod} = I\omega_{rod} \tag{15.2}$$

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

$$I_{rod+block}\omega = Dmv \tag{15.3}$$

or

$$\omega = \frac{Dmv}{I_{rod+block}} \tag{15.4}$$

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

$$\tau = \frac{D}{2}Mg\sin\theta + Dmg\sin\theta \tag{15.5}$$

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

$$\alpha = \frac{\frac{D}{2}Mg\sin\theta + Dmg\sin\theta}{I_{rod+block}}. \tag{15.6}$$

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.

6. 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 16

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 17

How do I...

17.1 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`.

17.2 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 `angle=atan2(vel.y,vel.x)`. This will put the angle of the vector `vel` into the variable `angle`.

17.3 Draw a vector on an object?

Use `draw_vector(tail,vector,color,"label")` as outlined in Section 7.3. 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 7.3 to tweak the size of the vector as needed.

17.4 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`$, 0 >$

17.5 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. This can be done in two ways. The first uses the distance formula (that you know). If `pos1` has components x_1, y_1, and z_1, and `pos2` has components x_2, y_2, and z_2, then the distance formula says that $d = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2 + (z_1 - z_2)^2}$. The second method uses a built in PhysGL function called `distance`.

- `d=sqrt(pos1.x-pos2.x)^2+(pos1.y-pos2.y)^2+(pos1.z-pos2.z)^2`
- `d=distance(pos1,pos)`

(Note in the distance formula, if you're object is in the xy-plane ($z_1 = z_2 = 0$), then you don't need the `pos1.z-pos2.z` part since this will be zero.)

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 and vel2.x < 0 and d <= R1+R2 then
   do this if they collided
end
```

17.6 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 and pos.y <= 0 then
   do this if the object hits the ground
end
```

17.7 Add trails behind an object?

A "trail" behind an object is nothing more than another rendering of your main object, with two differences. First, it likely looks smaller and with a different color and second, it persists on the screen for the duration of your animation. To do this, after you draw your main object, add another drawing line below it, to draw the object again, but perhaps make it smaller and with a different color. Lastly, to make the trail object stay on the screen, set the very last argument in the drawing statement to "true." This last argument is known as the "persist" condition. When `true`, the drawn object will persist on the screen (or, it won't be erased).

Formally, this will go in Part 6 of the skeleton code (recall all object drawing is done in Part 5). Here is an example. To draw a sphere at the position contained in the variable `pos`, you might do this

```
draw_sphere(pos,5,"red")
```

in Part 5 of the skeleton code. To draw a trail wherever this red sphere exists, follow this drawing statement with one like this

```
draw_sphere(pos,1,"yellow",true)
```

where we simply redraw a sphere at the same position, but make it smaller (radius of 1), with a different color (yellow instead of red), and with important "persist" parameter set to `true`.

17.8 Remove trails behind an object?

Look for a function in the code that looks like it draws an object at some calculated position, where the last parameter in the drawing statement is "true." This is likely the one that leaves a trail behind the main object. When you find this line, remove it from the code (or comment it out, by preceding the line with a `//` sequence.

17.9 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

```
draw_sphere(pos,1,"red")
```

See Figure 6.1, and be sure you put this line in Part 5.

17.10 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 17, 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 you can draw a box between two corners in 3D space with a line like

```
draw_box(<corner1>,<corner2>,"color")
```

where `<corner>` is a vector locating the diagonal corners of the box.

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

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

```
draw_plane(<0,1,0>,0,"green")
```

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

17.12 Find the magnitude of a vector?

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

1. `Amag=sqrt(A.x*A.x+A.y*A.y)`
2. `Amag=magnitude(A)`

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

17.13 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`. Put this in Part 1 of the skeleton code.
- Springs have a spring constant. Be sure to declare this in Part 1 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 1 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 4 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.

17.14 How do I draw a spring?

Use `draw_hspring` for a horizontal spring or `draw_vspring` for a vertical spring. See Chapter 7 for a full description of these.

Chapter 18

What to do if your animation won't run

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.

18.1 My animation doesn't appear at all

If you click "Run" and you don't see an animation at all, then it is likely that you have a typo or error with some usage of a PhysGL 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:

- Are all of your needed variable declared?
- Are variables that must be vectors declared as vectors? If for example `a` is to be your acceleration, be sure you are doing `a=<1,0,0>` and not just `a=1` for an x-acceleration of 1.
- 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.
- Does each `if` statement have a `then` word after the condition?
- Look at the bare skeleton code in Section 6.2. 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.
- The `while-condition-animate` statement must have a matching `end` statement with it. Is this still there? Did you accidentally delete it?
- Look at your formulas. Did you spell all variable names correctly?
- Look at your formulas. Is each variable you use in a given formula defined *before* it is used in a given line?

18.2 My animation runs, but 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 might be to blame it on "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?
- If vectors are too small or large, see the `set_vector...` functions in Section 7.3.
- Check that your accelerations are all set up in Part 4 of the your code, *before* the physics equations.
- Check your code versus the bare skeleton code in Section 6.2. 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.
- If all else fails, consider starting over with a clean skeleton code and clearly think through the physics logic needed to drive your animations: 1) initial position and velocity variables, 2) set up accelerations, 3) draw your object, 4) apply physics equations.