Adventures with Optics at Home

Frank Wattenberg
Department of Mathematical Sciences
United States Military Academy (Emeritus)
Frank.Wattenberg@mac.com

November 30, 2020

Contents

Chapter 1

Introduction

Figure 1.1: Reflections in a Spherical Make-Up Mirror

Figure 1.1 shows typical experiments that students can do at home with everyday objects – in this case, a make-up mirror and a cell phone. Experiments like this often produce surprising results that can be understood and predicted using middle and high school mathematics. They are a great setting for project-based-learning. Working individually and in teams every student can produce satisfying work and, at the same time, these projects are open-ended with lots of opportunities for the most motivated and best prepared students. Our first examples look at reflections in mirrors like those seen in Figure 1.1. Our second examples look at refraction, like that seen in Figure 1.2.

Most importantly, students will have fun as they play with optics. They will have stories they can tell their friends and parents about the surprising discoveries they made. They will see their own work in puddles and places like Chicago's "Bean" (Figure 1.3).

Figure 1.2: Refraction in a Washbasin

Figure 1.3: Reflections in Chicago's "Bean"

Students will learn both science and mathematics. They will work and play like scientists and mathematicians as they go back-and-forth between experimentation and theory. They will check their answers not by looking in the back of the book but by doing experiments. They will discover and work with physical principles like Fermat's Principle and Snell's Law and they will use geometry, graphics, and their knowledge of linear functions to understand the often unexpected phenomena they see.

> Begin by looking inside and around your home and school for optical phenomena. Look in ponds and lakes as well as clear glasses that can be filled with water. Look for optical phenomena in seasonal objects like ornaments. Look for rainbows on your walls and floor as well as in the sky. See Figure 1.4. Take lots of pictures.

Figure 1.4: Rainbows on Floors and Walls

An Adventure Gallery

We are about to embark on an adventure in optics and before we start it is worthwhile to reflect on other adventures you've enjoyed – for example, maybe you and your family or friends have visited Washington DC. Figure 1.5 is an aerial view showing many of the tourist sites in Washington DC. This figure shows some of the well-known sites in the center of Washington. You can see the White House, the U.S. Capitol, the museums on the National Mall, Washington's Monument – enough sites just in this view to keep you busy for many days. But, you can't see some of my family's favorite places – for example, Mount Vernon and the new Air and Space Museum near Dulles Airport.

As you can imagine, each of my family members wants to visit different places. High on all our lists are the Washington Monument, the World War II Memorial (Figure 1.6) and the White House and the National Tree (See Figure 1.7). Some of us put Mount Vernon (Figure 1.8) near the top of our list and some of us put the new Air and Space Museum (Figure 1.9) at the very top. At that museum some of us can't wait to get to the Space Shuttle Discovery and others want to spend more time with all the historical airplanes.

As we and our students embark on adventures with optics, different people will be most interested in different aspects of these adventures. As we plan our students' adventures with optics we need to encourage them to follow and spend time with the parts of these adventures that they find most engaging. For example, some of our students may get

Figure 1.5: Aerial View of Some Well-Known Sites in Washington DC

Figure 1.6: The Washington Monument from the World War II Memorial

Figure 1.7: The White House from the State Trees near the National Tree

Figure 1.8: At Mount Vernon

Figure 1.9: The New Air and Space Museum

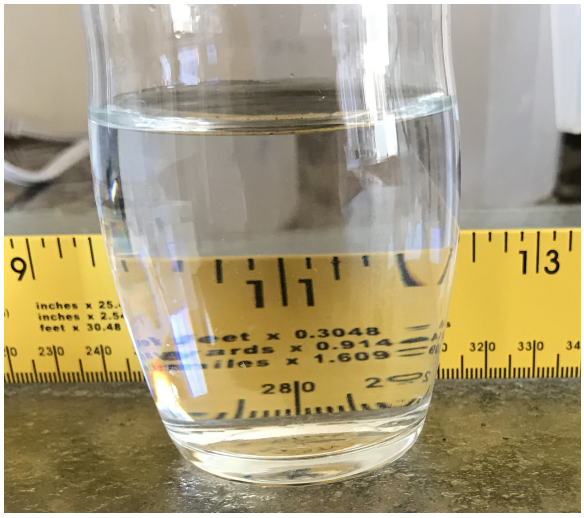

Figure 1.10: A Water Glass on a Kitchen Countertop

really interested in taking pictures of interesting optical phenomena around their house. See Figure 1.10. We should encourage them, have them show their work to the whole class and maybe even talk about how the ideas we're developing in this adventure can be used to analyze these phenomena.

Students do not come to class just to listen. They do not come to class just to talk. They come to class to have rich conversations, learning from their teachers and each other and perhaps, most importantly, learning how others react to their own words, pictures and ideas. Good projects give students a chance to present and explain their ideas – with words, pictures, numbers and even a bit of algebra. They give students the opportunity to develop their story telling skills by seeing how others react to their presentations and by reflecting on how they themselves react to others' presentations.

When students talk about the optical phenomena they've observed in and around their own homes, they will almost certainly need to take pictures that show what they see with their eyes and they will need to talk about what they "see" in the pictures. They will need to discuss the phenomena they see and the questions these phenomena raise in their minds. For examples of questions people ask just start typing "why are mirror images" into Google and see all the questions people search on. By following some of the links you can see some of the many answers people have given to questions like "Why do mirrors reverse left-to-right instead of upside-down." Ultimately, different people give different answers to questions like this – answers that play into the way they think about things. Learning is as much, maybe more, about asking questions as it is about finding answers. Our students

Figure 1.11: At the Train Show

will use all the tools of expression including words and pictures to frame questions and to build and convey their understanding. Our students will learn all these skills.

One of our holiday traditions is the Holiday Train Show (Figure 1.11) at the U. S. Botanic Garden, near the U.S. Capitol Building. This is a very popular Washington attraction and is usually very crowded. We usually get there early in the morning sometime between Christmas and New Year's Day and join a long line waiting for the opening. One year, we ran into another family doing the same thing. They invited us to play a game – "Heads-Up" – that they had on their cell phones. It was really cool and turned the usually long chilly wait into happy memories. We keep hoping we'll run into them again. This is an example of one of the most important aspects of life – "serendipity" – completely unplanned and unexpected seemingly random occurrences that often have a big impact. Collaborative, project-based learning, centered on everyday objects often provides surprising serendipitous opportunities. Be alert for them and as you plan explorations for your students try to set up conditions that encourage serendipity.

The best way to encourage serendipity is by making your students' work/play more social and more visible. The publications in which I was most proud as a student appeared on the refrigerator or in my schools' hallways and I learned a lot from chance encounters with other students, teachers, and parents gathering in front of my posters and in front of theirs. At West Point we often invite visitors, our students' coaches, and their families to our students' poster sessions and presentations.

Chapter 2

Overview

2.1 Collaboration

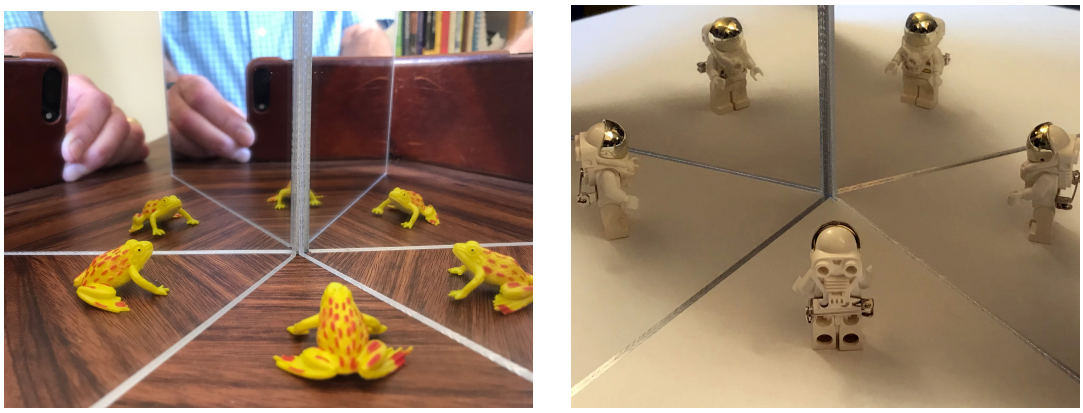

Figure 2.1: Desktop Kaleidoscopes

Figure 2.1 shows two of my favorite experiments – kaleidoscopic images created by a pair of mirrors taped together at one edge. I did these experiments using a pair of small mirrors and a toy frog (on the left) and a toy spaceman (on the right) but it is much more fun and more instructive to take the place of the frog or spaceman yourself and stand in the angle formed by a hinged pair of door-sized mirrors. Building this life-sized kaleidoscope would be a great project for a shop class or for parents and is just one example of how a community can collaborate to build exciting experiences for their students.

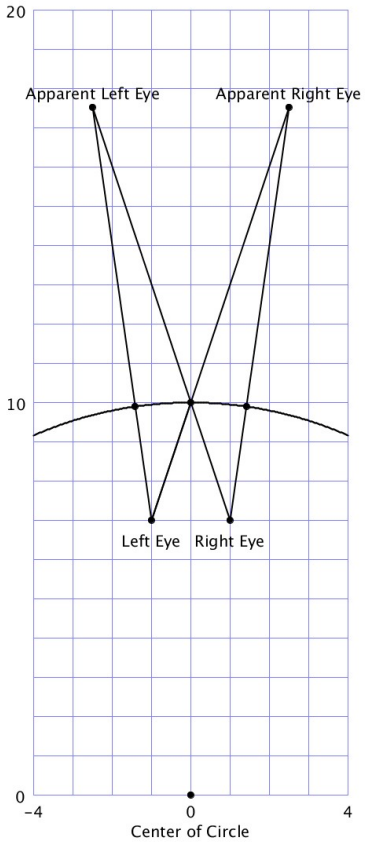

Figure 2.2: Multiple Steps

The optical phenomena we will study are complex and building understanding requires multiple steps. In some cases the steps are very repetitive. For example, one of the more surprising phenomena your students will discover is what a fish sees when he looks at a fisherman waist deep in water. Understanding this phenomenon requires finding the path followed by light rays traveling from different features (forehead, eyes, nose, mouth, ... feet) on the fisherman to the fish. For one person, this is boring and repetitive but a team of people can split up the work. This is exactly what computers with multiple processors, or cores, do. The technical term is "parallel processing."

In other cases we have to follow a sequence of steps, not at the same time but one after the other. In Figure 2.2, for example, we are looking at what a person sees in a make-up mirror. This is a complex problem with an unexpected result. It involves our two eyes in

two different ways. First, we use our eyes to look at our reflection. Second, our eyes are features that we look at in our reflection.

Most people, when asked what parts of our body we use for vision would say "our eyes" but our brain also plays a huge role. It puts together the information it receives from our two eyes to build a three-dimensional model of our world. In particular, it determines the "apparent position" of each of our two eyes.

The complexity of this problem and the number of steps required can be seen by observing the complexity of Figure 2.2. All of this requires a different kind of collaboration with different team members doing very different parts of the problem. Because optics is so rich and complex it can become a theme that recurs throughout a student's life and a setting in which high school students can help out middle school students at the same time they see the potential for further study in advanced placement or college classes and for future careers. For example, today's optical engineers are designing cool lenses like the fish eye lenses that produce images like those shown in Figure 2.3 and today's optical engineers working together with today's software engineers are designing interactive 360 degree spherical imagery like that seen at the links below. You can look all around you in these images by clicking and dragging in the images – even up and down. You can even zoom in and out. All this is based on the same ideas your students will study doing Optics at Home.

- The Lincoln Memorial (Washington DC):
  https://theta360.com/s/fTTJRHWrPkg4SpImvIvuJS9gW
- The World War II Memorial (Washington DC):
  https://theta360.com/s/svc900vJ4HYVOrvA9WybDuOwa
- The Bear Mountain Bridge (near the U. S. Military Academy (West Point) in NY):
  https://theta360.com/s/oUMoYdIEa0DpBz8wfd1LVtjOa?utm_source=app_theta_ios&utm_medium=referral

The power of mathematics and computer programming is inexpensive creative power. With their cell phones, math, and computing, your students can create their own museum quality pictures that tell stories and that move people. The same ideas are used to create immersive three-dimensional movies like the Star Wars movies. The acronym STEAM is a partnership between STEM (Science, Technology, Engineering and Mathematics) and the Arts. Optics at Home is a STEAM playground.

Figure 2.3: Two Photographs Made with Fisheye Lenses

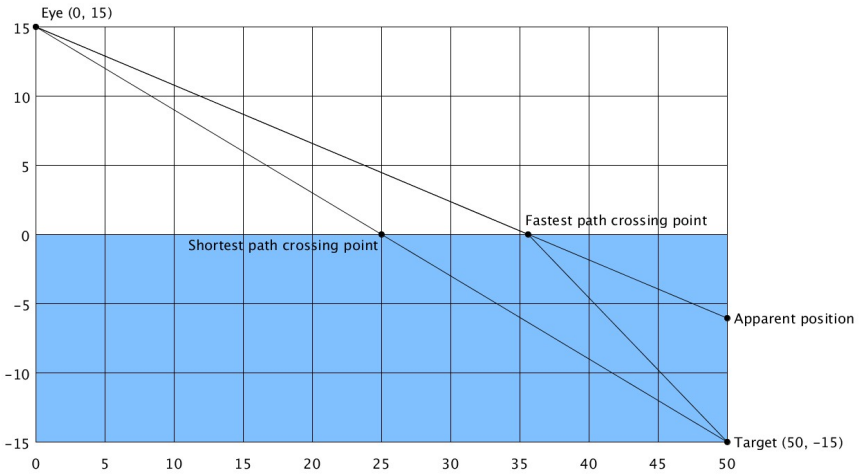

Figure 2.4: Fermat's Principle in an Aquarium

2.2 Scientific Principles

The overarching and most important scientific message is the "scientific method" – the interplay between experimentation/observation and theory. The first optical principle we will explore is "Fermat's Principle." Its easiest form – "light rays traveling between two points follow the fastest possible path" – is almost anthropomorphic, as if light rays are like people in a hurry. It "explains" why light rays traveling between two points in a single medium travel along straight lines – because the shortest path between two points is a straight line. It is supported by a wealth of observation. Every time you reach out and touch something, you add to the body of evidence supporting this principle.

Figure 2.4 shows a straightforward application of Fermat's Principle that can be used to predict the phenomena we see when we look at the drain in the bottom of a washbasin. See Figure 1.2 on Page 2. In Figure 2.4 an eye located 15 cm above the surface of the water in an aquarium is looking at a target 15 cm below the surface of the water. Using the coordinate system shown in the figure the location of the eye is $(0, 15)$ and the location of the target is $(50, -15)$. The x-axis runs along the surface of the water.

The shortest possible path from the target to the eye would be a straight line but this is not the fastest possible path for our anthropomorphic light because light travels faster (30.0 cm per nanosecond) in air than it does in water (22.5 centimeters per nanosecond). Both paths are shown in Figure 2.4. Notice that light travels a bit further in air than in water to take advantage of its higher speed in the air. We'll talk about finding the fastest

path in Section 2.3.

You can also see in Figure 2.4 why the target appears to be higher than its actual location. Your brain has learned from long experience that incoming light rays have traveled on straight lines. The "apparent path" – that is, the path that your brain assumes the incoming light ray took is the straight line that goes through the eye and the place where the fastest path crossed the water's surface. Your brain believes that the apparent position of the target is on the apparent path.

Of course, the photons that make up light[1] aren't really equipped with the same computational tools that we have and aren't able to plan and then follow the fastest possible paths. As you and your students explore optical phenomena around the house you will find that sometimes photons take the longest possible path and we'll have to revise Fermat's Principle. We'll also discover other laws like Snell's Law and "the angle of incidence equals the angle of coincidence" – that provide more understanding of how light travels.

2.3 Mathematics – Big Ideas and Tools

The mathematical tools we need are computational, visual, geometric and algebraic. Careful drawings like Figure 2.4 on Page 15 will play a huge role as will the ability to graph functions. We will need to be able to find minimums (and maximums and "critical points") of functions, and we will need the expressive power of algebra and computer programming. Some of our most important tools involve the use of linear functions. For example, we will need to be able to find the equations that describe a straight line between two points and we will need to be able to find the intersection of two straight lines. See Figure 2.2 on Page 12 to see how these tools will be used.

> *As a theme Optics at Home can tie together the math, science, computer programming and arts that students learn at different times in different courses. But in any single class they can focus on the material they are studying at the time. For example, middle school students will be able to discover and understand both reflection and refraction using just the material they are studying on graphs, linear functions, and geometry using whatever technology they are using whether it's graphics calculators, computer software, or a bit of coding. As students progress in their mathematical careers, they will see how trigonometry, geometry, calculus, multivariable calculus and linear algebra all contibute to understanding and applying optical phenomena.*

[1] Yes, we'll need to talk about waves, too, and the fact that light sometimes behaves like photonic billiard balls and sometimes like waves.

As an example, suppose you are teaching middle school mathematics and want to focus on the visual geometry of circles and straight lines and using graphing calculators to graph functions and find minimums. Your students could begin with a superficially simple experiment with surprising results – looking at themselves in a concave, "magnifying" make-up mirror. With a flat mirror it is easy to magnify an eye, for example, by just getting really close to the mirror but there is a catch. When you get really close to the mirror you cannot focus on your image. Try it!! If, for example, your eye is three inches from the mirror then it is six inches from its "mirror image" – too close for all but really near-sighted eyes.

Our students' ultimate goal is to produce Figure 2.2 on Page 12. This figure shows that at the same distance from a concave, "magnifying" make-up mirror[2] the "mirror" image is more than ten inches from your eye. Now you can focus on it. It is also considerably larger than it would be if the mirror was flat. This is exactly what your students see in their own make-up mirrors.

To cover this in the middle school math class described above, you could begin by giving students copies of the sample worksheet in Figure 2.5 or having them draw this figure with ruler and compass on graph paper. Notice that the mirror is described by the function

$$f(x) = \sqrt{100 - x^2}.$$

They should mark the location of a left eye at the point $(-1, 7)$ and a right eye at the point $(1, 7)$ on the worksheet.

Then they would go through the following steps:

- Find the path a light ray would follow from the target left eye to the observer left eye. They do this by first finding a function whose output is the total distance from the left eye (in its role as a feature on the observer's reflection) to a bounce point $(x, f(x))$ on the mirror and then back to the left eye (in its role as one of the observer's eyes looking at the observer's reflection). Then they graph that function and find its minimum – using, for example, a graphing calculator. Next they mark the point $(x, 0)$ where this path bounces off the mirror.

- Mark on the worksheet the apparent line that the observer left eye thinks comes from the target left eye. They do this with a ruler, drawing the line determined by the observer left eye and the bounce point.

- Do exactly the same thing for the target left eye and the observer right eye.

[2]This mirror is a spherical mirror whose radius is ten inches.

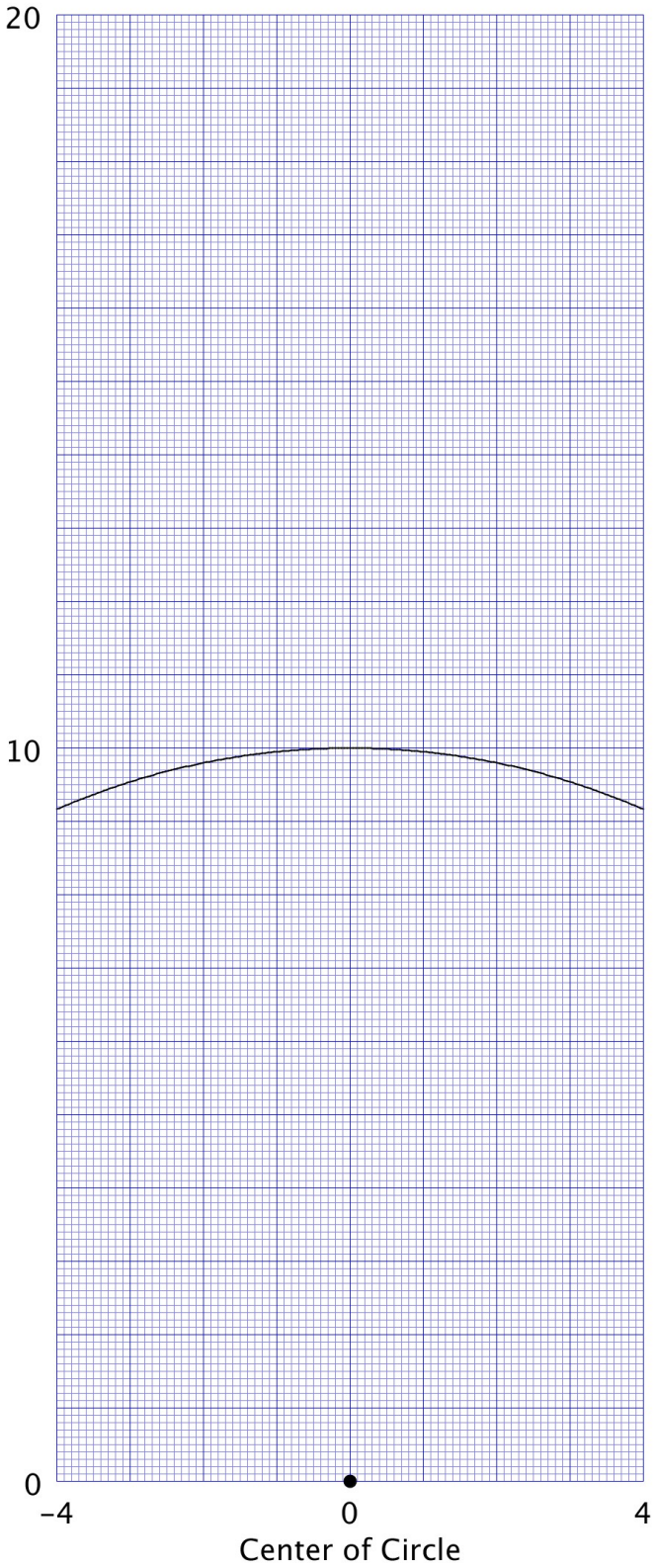

Figure 2.5: A Sample Worksheet

- Find on the worksheet the apparent location of the left eye – the point where the two apparent lines found above intersect.
- Do exactly the same thing for the right eye as a target.

Now your students have constructed Figure 2.2 on Page 12. This figure is necessarily a bit rough. They can find out more precisely what you see from this close to a make-up mirror by applying what they know about linear functions as follows:

- Find the linear function that describes the apparent line seen by the left eye looking at the left eye using the location of the left eye and the bounce point.
- Find the linear function that describes the apparent line seen by the right eye looking at the left eye using the location of the right eye and the bounce point.
- Find the apparent position of the left eye by finding the intersection of the two apparent paths found above.
- Do the same thing for the right eye.

Have students describe what this math tells us about what we see in a concave, magnifying make-up mirror. Have them compare their conclusions with what they actually do see. Some students may notice that the path followed from the left eye to the right eye bounces off the mirror at the top of the mirror and others may have noticed that the path followed from the left eye to the left eye is part of the straight line (or radius) from the center of the circle through the left eye. Ask them to talk about why? Try some other experiments – for example, find the apparent position of the nose at the point $(0, 7.75)$.

Although minute-by-minute students are learning and applying the skills required by the Core Standards, they are also learning the big ideas. As one example, they are learning how we build understanding – that is, how we build mental models that help us understand and navigate our world. Fermat's Principle is a good example of a descriptive model. We can use this model to predict what we will see when we look in a mirror or in the drain at the bottom of a washbasin filed with water but it does not explain why light rays follow the paths they do. Light rays are not mathematicians carrying miniature calculators they use to plan their trips. As we look at other principles and analogies we will build more explanatory models that explain why we see what we see.

2.4 Technology Tools

In this section we talk about the following:

- Calculators, mathematical software and coding.
- Digital cameras including cell phone cameras.
- Digital photography and imagery.

2.4.1 Calculators, Mathematical Software and Coding

Your students are almost certainly using graphing calculators, mathematical software like Excel or Desmos, or doing some coding. Whatever they are using gives them the power to draw graphs, make calculations, and find minimums (or maximums) of functions – exactly what they need as they do Optics at Home. At the same time that they are seeing the power of whatever technology you choose they will build their skills with that technology. You should begin by doing an inventory of the technology tools and skills your students have available and, if they are using more than one of these tools, choosing the tools with which you want them to build their skills.

2.4.2 Digital Cameras Including Cell Phone Cameras

Modern digital cameras are incredibly powerful tools for exploration. They are very common, small and portable. The pictures they take are close to free. We no longer buy film and pay for photographs to be developed and printed. Most people have truly remarkable cell phone cameras. You and your students should do an inventory – what cameras do you have and what can they do?

Paradoxically, older cameras may have some features that are better for some experiments than more modern cameras. For example, the apparent distance of our image in a mirror is extremely important. Our eyes focus automatically and we usually become aware of the apparent distance only if we need glasses or if we are just too close to focus. Cell phone cameras and the vast majority of modern cameras also focus automatically but if you have a camera that can be focused manually you can use it for some great experiments.

Cameras and, especially, lenses also are examples of the kinds of things optical engineers can do. When you and your students inventory your cameras in their new roles for Optics at Home look for older cameras as well as newer ones and look for lenses, tripods and other equipment that might be interesting and useful.

2.4.3 Digital Photography and Imagery

Figure 2.6 is just one example of the many things we can do digitally. In this case creating a kaleidoscopic image. We are all familiar from movies like the Star Wars movies with the creative power of digital imagery. We can also use digital imagery experimentally – for example, to record the changes of color in the spring and fall. At the same time that our students develop and apply this power they will learn more mathematics and programming.

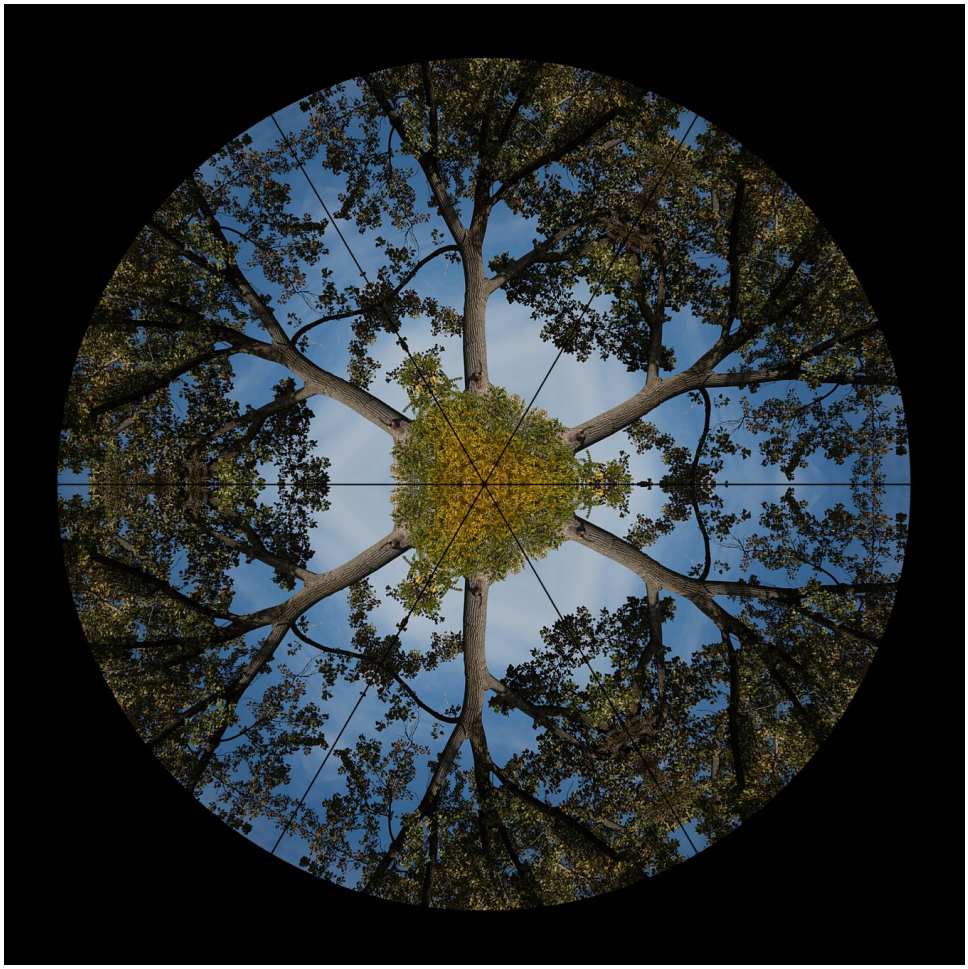

Figure 2.6: A Digitally Kaleidoscopic Image

Chapter 3

Reflections in a Flat Mirror

> *We and our students always begin with experimentation and observation. For this topic these are simple. Find a convenient flat mirror and look at your reflection. What do you see? Describe your "mirror image" as completely as you can.*

This can be a short simple section. The key observations are:

- Your mirror image appears to be flipped or reversed right-to-left.
- Your mirror image appears to be behind the mirror – the same distance behind the mirror as you are in front of the mirror.

Mathematically, (See Figure 3.1) if the mirror lies along the x-axis and a feature is at the point (x, y) then its reflection (mirror image or apparent position) is at the point $(x, -y)$.

This is all you and your students need to do before moving on but you can also take advantage of this opportunity to build some experimental expertise for more complicated mirrors and even make this a more social experience. If you want to and are able to spend more time on this experiment continue on. Otherwise skip ahead to Page 26.

Print a copy of Figure 3.2 on Page 25. Trim the margins from this sheet and tape it to a piece of cardboard. Then cut out the eyehole in the middle of the figure. Look through the eyehole at your reflection in a flat mirror. See Figure 3.3. I took this photograph holding the lens on my cell phone up to the eyehole. You and your students can do the same thing.

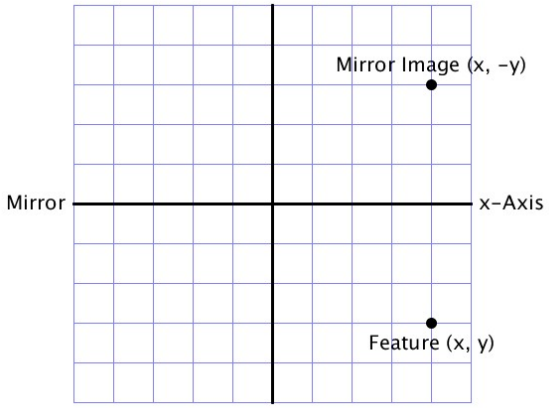

Figure 3.1: Reflection in a Flat Mirror

You might want to try looking at your reflection in other mirrors – for example, the convex mirrors used at intersection – through your eyehole. If you have trouble focusing your eyes on some reflections, you've got a preview of some of the experiments we'll do later and some of the optical phenomena we'll discover and . You may want to use Figure 3.5 instead of Figure 3.2 for some mirrors.

You can do more experimentation with the same apparatus and a lab partner. For example, you can print out Figure 3.4 on Page 27 and have your lab partner hold it up and walk back-and-forth toward you and away from you as you look through the eyehole until it looks like your reflection in the mirror. Then you can switch roles with your lab partner so he or she can join in your optical explorations. This is one way that your students can work/play with their homework/play with siblings and parents.

You can even set up experiments like this using larger posterboards and mirrors in your school yard or hallways – investigating optical phenomena can be a very social activity full of surprises.

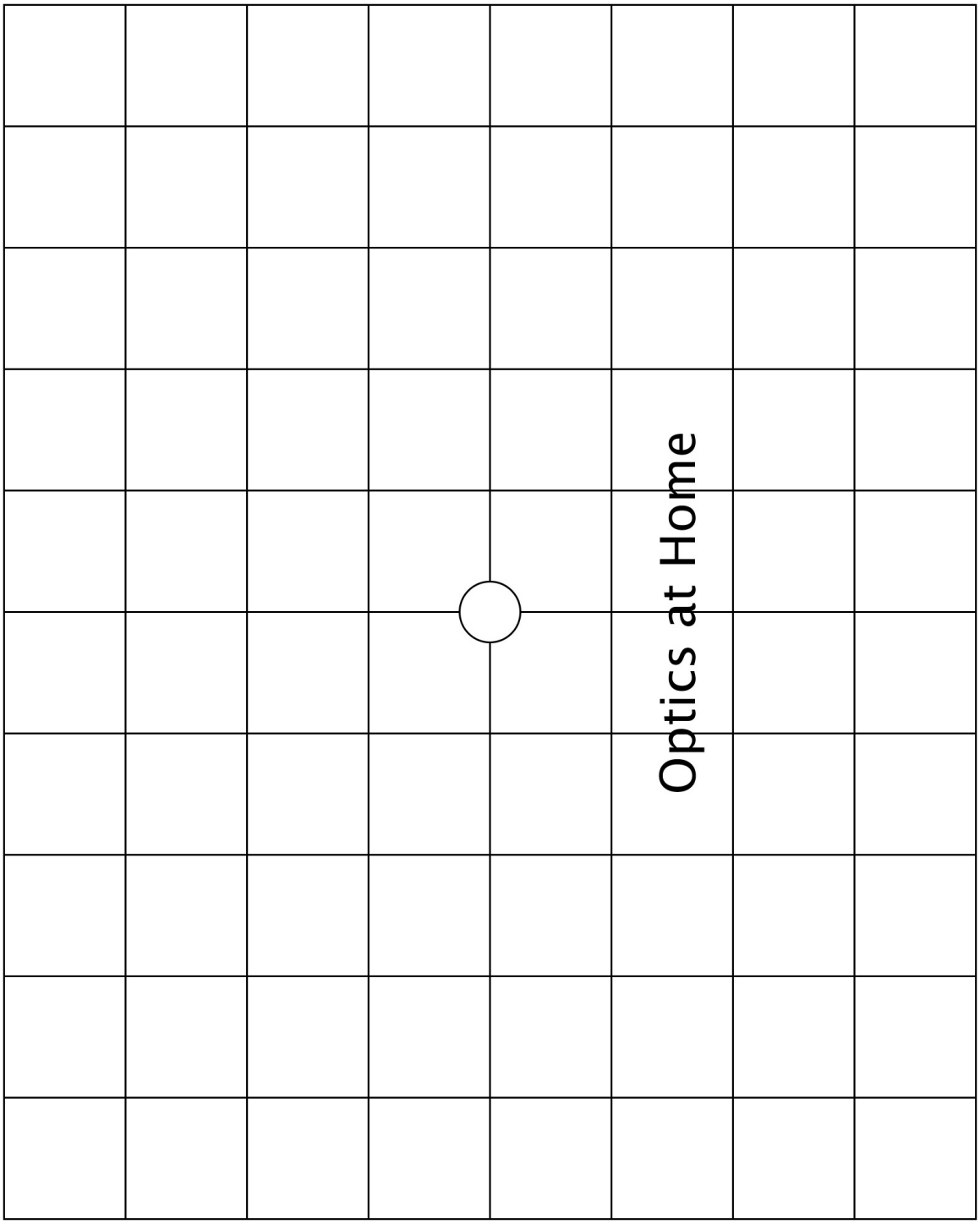

Figure 3.2: Flat Mirror Experiment, 1

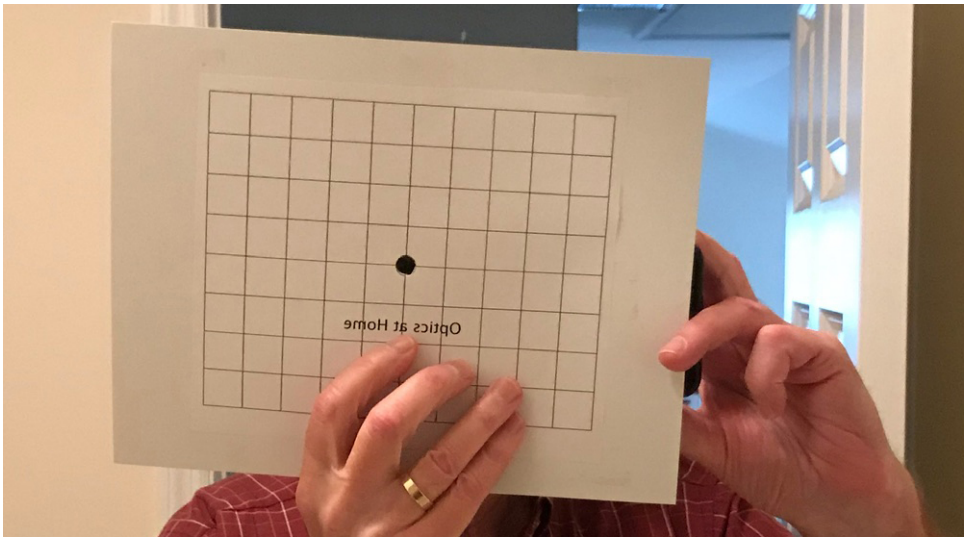

Figure 3.3: Looking Through the Eye Hole

Continue here – OK, now we are ready to move on. Recall your two observations about your reflections in flat mirrors:

- Your mirror image appears to be flipped or reversed right-to-left.
- Your mirror image appears to be behind the mirror – the same distance behind the mirror as you are in front of the mirror.

Let's the first observation first. You may wonder why the mirror image is reversed left-to-right instead of upside-down. If so, you're not alone. If you Google "Why do mirrors flip left ..." or "Why do mirrors reverse left ..." the autocomplete will complete your question and provide links to many web pages that address this question. We'll begin with a simple experiment that helps answer this question. Start with a thin sheet of paper – one that you can see through. Write the words "Army Math" on the paper and hold it on your chest as you look in the mirror. As you continue looking at those words in the mirror extend your arms straight in front of you so that you can see through the paper. Compare the words in the mirror with the words you see looking through the paper. Notice these words are reversed right-to-left because you are looking at them from the back.

Now let's see whether Fermat's Principle can help us understand reflection in a flat mirror. In its first and simplest form Fermat's Principle says that light rays traveling between two points follow the fastest path. When we study refraction, light rays will be traveling at

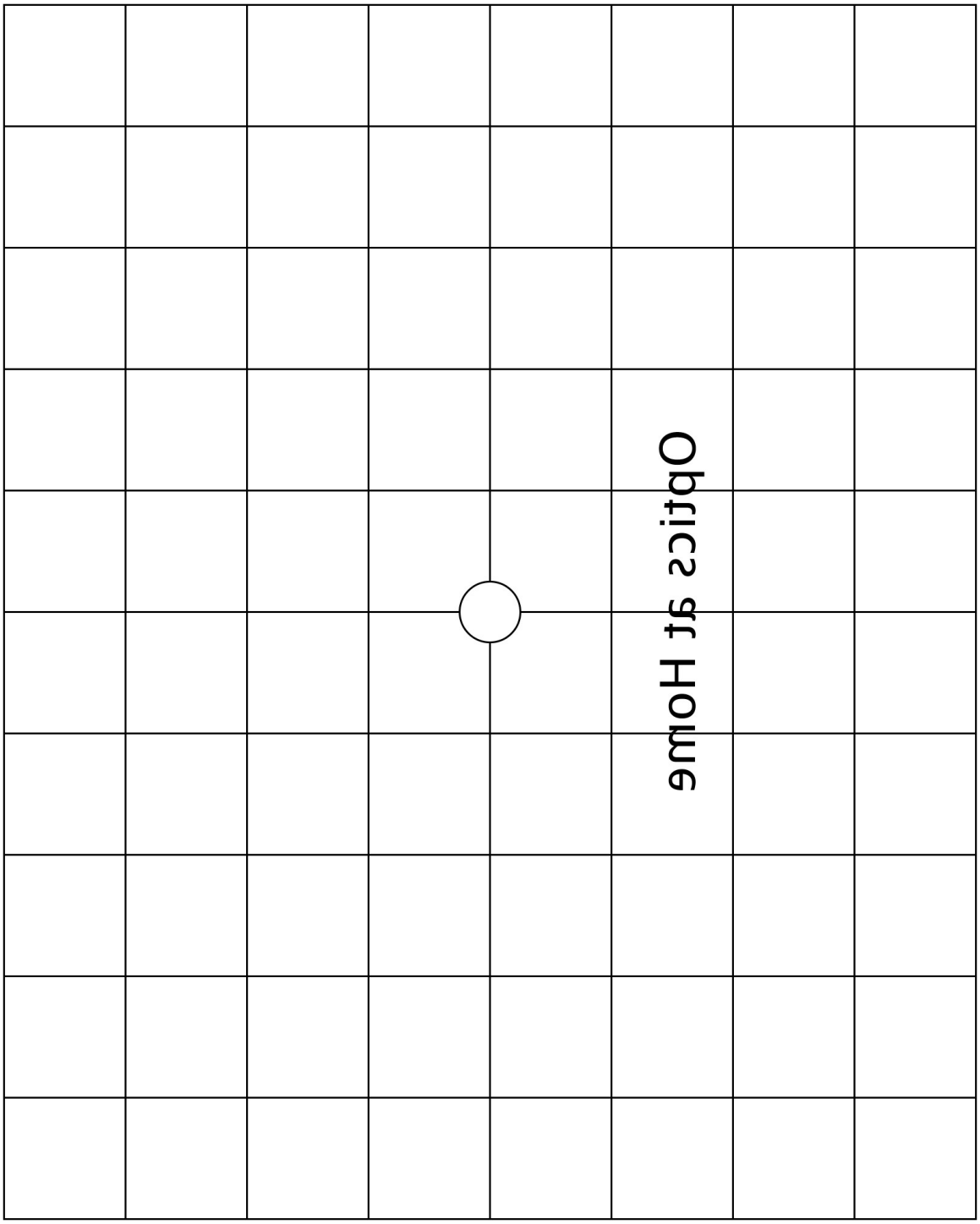

Figure 3.4: Flat Mirror Experiment, 2

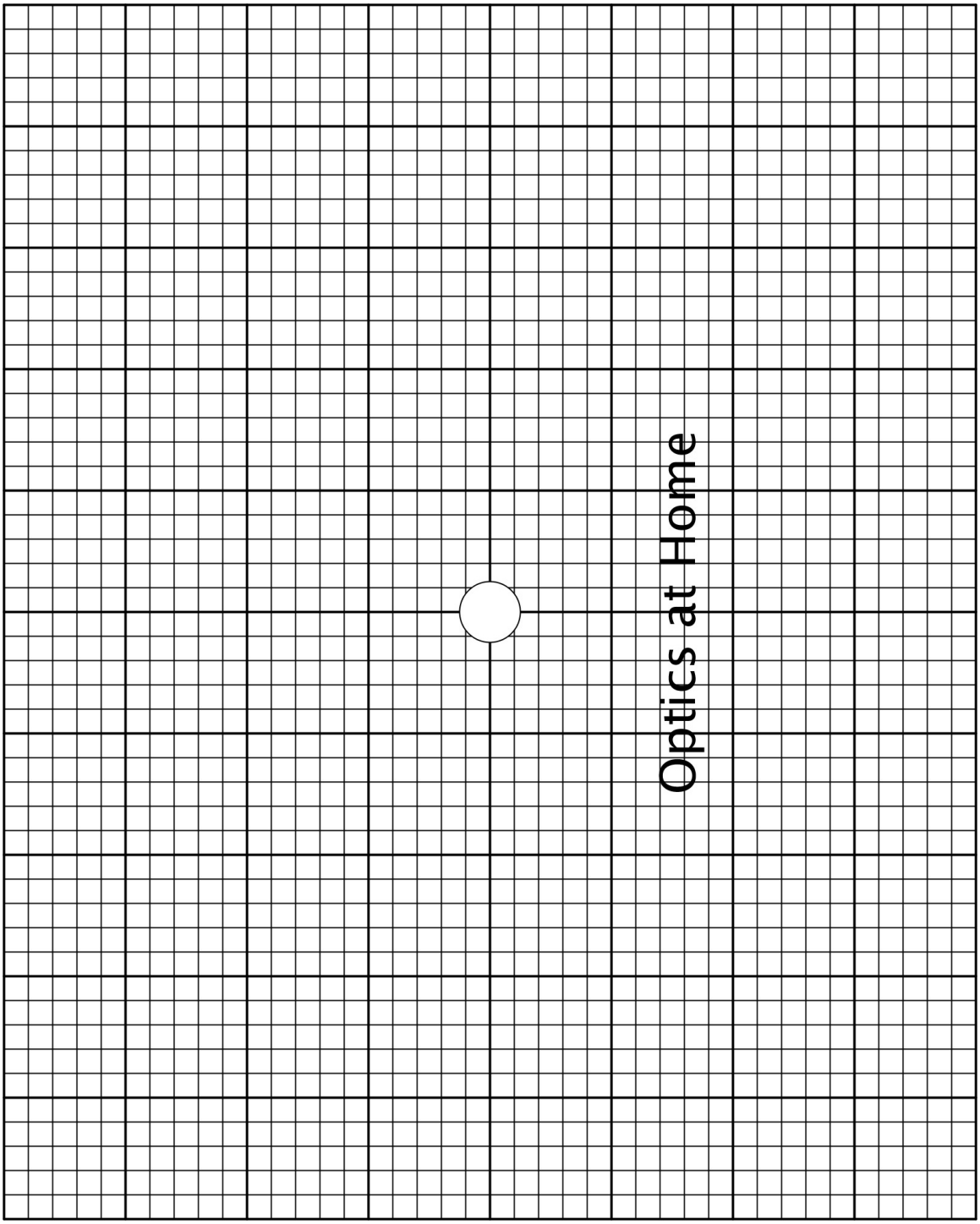

Figure 3.5: Flat Mirror Experiment, 3

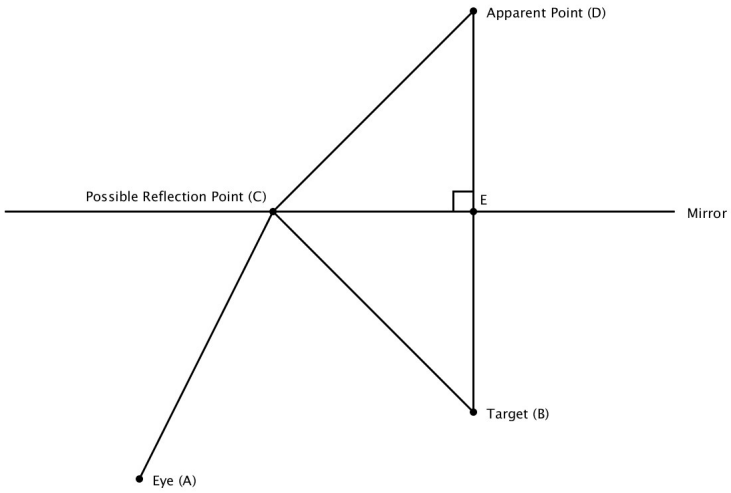

Figure 3.6: A Possible path

different speeds in different mediums, like air and water. But in our present situation is only one medium, air, and therefore the fastest path is just the shortest path.

Fermat's Principle did not arise from thin air but from observation – most importantly the multiple observations we make everyday when we reach out and touch something. Our current investigation of reflections in flat mirrors provides more evidence leading to postulating this principle.

Figure 3.6 shows a mirror, an eye (at the point A) and a target (at the point B). Of course the shortest path from the target to the eye is just a straight line but we are interested in more complicated paths – paths that go from the target to the mirror, hitting the mirror at a point C, and then traveling to the eye. These paths are made up of two straight line segments, the line segment BC and the line segment AC. The light rays that travel on this kind of path will pick the reflection point C to minimize the total length of the two segments BC and AC. We must figure out where on the mirror this best reflection point is.

We make use of our observation that the mirror image of a target is behind the mirror at the same distance behind the mirror as the target is in front of the mirror. We mark that point on Figure 3.6 and label it "Apparent Point (D)". The line BD is perpendicular to the mirror. Notice that the two triangles ΔCED and ΔCEB are congruent. Therefore the line segments CD and CB have the same length. Therefore the path – from B to C and then from C to A has the same total length as the path from D to C and then from C to A. If we choose the point C to be on the line from A to D as shown in Figure 3.7

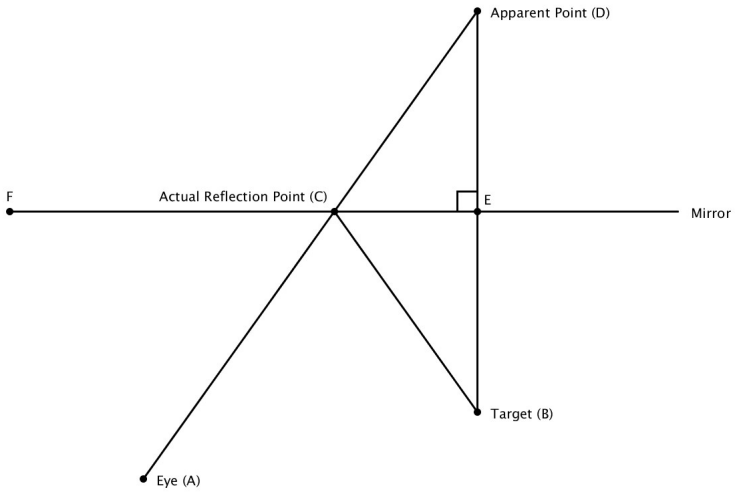

Figure 3.7: The fastest (shortest) path

then the path from A to C and then to D is the shortest possible path and, thus, so is the path from A to C and then to B.

Notice the angle $\angle FCA$ is the same as the angle $\angle DCE$. The angle $\angle FCA$ is the angle at which the light ray bounces off the mirror. Notice the angle $\angle BCE$ is the same as the angle $\angle DCE$ because the two triangles ΔCED and ΔCEB are congruent are congruent. The angle $\angle BCE$ is the angle at which the light ray from the target to the mirror hits the mirror. The observation that $\angle FCA = \angle DCE$ accords with our observations about how balls bounce off walls. This analogy helps us build a mental model of how light rays bounce off mirrors.

We sense depth – that is, we build three-dimensional models of our world – using binocular vision. In Figure 3.8 we add a second eye and do the same analysis we did for the first eye. Our brain takes the input from the two eyes and assumes that the incoming light rays are straight lines without any changes of direction. These two straight lines are called **apparent paths**. So our brain assumes that the target is at the point where the two apparent paths meet – that is, at the apparent point D. Thus, the apparent position of the target is behind the mirror directly opposite its actual position and the same distance behind the mirror as the actual target is in front of the mirror. Mirror images are reversed left-to-right because they are directly behind the mirror directly opposite actual images. In effect, we are looking at the actual image from behind.

Notice what we have done. We started with observations and we showed that Fermat's

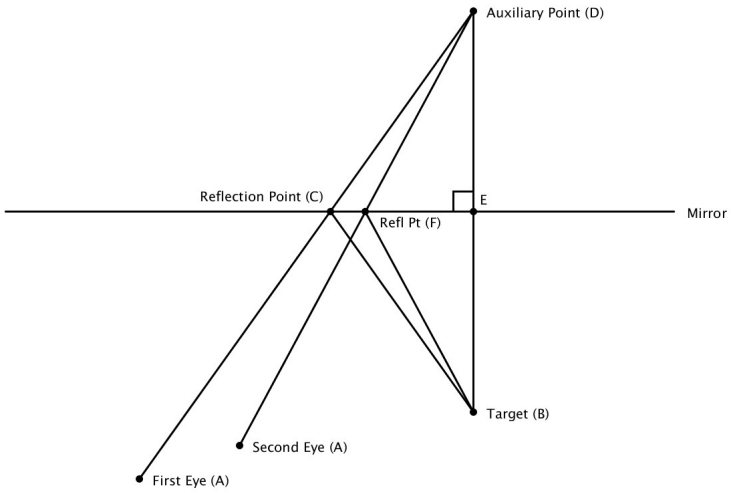

Figure 3.8: Two eyes and binocular vision

Principle is suggested by these observations. We also observed that light rays seem to bounce off mirrors the same way that pool or billiard balls bounce off the rails on pool or billiard tables. We will get more evidence for these ideas in the next set of experiments – with curved mirrors.

Chapter 4

Reflections in Make-Up and Other Curved Mirrors

Figure 4.1: Reflections in a Spherical Make-Up Mirror

If you have a make-up mirror like the one shown in Figure 4.1 you can do some interesting experiments. If you look at your reflection from fairly far away you will see a reflection like the one shown on the left in Figure 4.1 but when you use the mirror as a make-up mirror you get very close to it and see a reflection more like the one on the right (without the cell phone camera).

For your first experiment place your make-up mirror on a shelf or maybe have someone hold it at eye level when you're standing up. Start fairly far away from it, maybe six or eight feet. Then walk slowly towards it looking straight at the center of the mirror. As

you get closer, close one eye and keep the reflection of your open eye in the center of the mirror. As you get closer, your image will get larger until at one point the center of your eye will fill the entire mirror. We'll see why soon. When you get even closer your image will flip and be very large.

Mirrors like this are often advertised as "magnifying make-up mirrors." You can magnify anything you're looking at by getting closer to it and the easy answer to why your reflection looks so big is – because you're so close to the mirror. But, it's a bit more interesting than that. Take a piece of paper with some writing on it and while you're very close to the make-up mirror looking at your very large eye, slide the paper right next to the mirror. Unless you're very near-sighted you won't be able to read the writing because it is out-of-focus. You can focus on your mirror image because it is farther away than the mirror is. We learned a bit about this when we studied flat mirrors and the mirror image was behind the mirror. We'll learn more about this as we continue.

Understanding curved mirrors, like concave make-up mirrors, and the convex mirrors often used on vehicles and at intersections to avoid blind spots is complicated. The best way to build understanding is step-by-step – doing experiments and using geometry, careful drawings, and math to help explain what we see. This chapter is intended to help you and your students discover what curved mirrors do and understand how and why they do what they do. The end of this chapter is an appendix with our analysis but this appendix is intended only as a last resort if you get stuck or to help summarize what you have discovered. We most emphatically do not use "back-of-the-book" answers to check our work. As scientists we check our work by experimentation.

Make-up mirrors are spherical and we can learn a lot about them by studying circles. Figure 4.2 shows a circle of radius $R = 10$ inches with its center at the origin. Points (x, y) on this circle satisfy the equation:

$$x^2 + y^2 = R^2.$$

We have marked two points on the circle in Figure 4.2 with black dots and we have drawn the line from the center of the circle to each point as well as the tangent to the circle at each point. These two lines meet at a right angle – that is, the tangent at a point on the circle is perpendicular to the line (sometimes called the "radius") from the center of the circle to the point. Tangents are important because when a light ray bounces off a mirror it bounces as if it hit the tangent.

If you're looking at a a spherical mirror from its center every light ray going toward the mirror in any direction hits the mirror and bounces right back to your eye. This is the reason why your eye appears to fill the entire mirror. This provides an experimental method

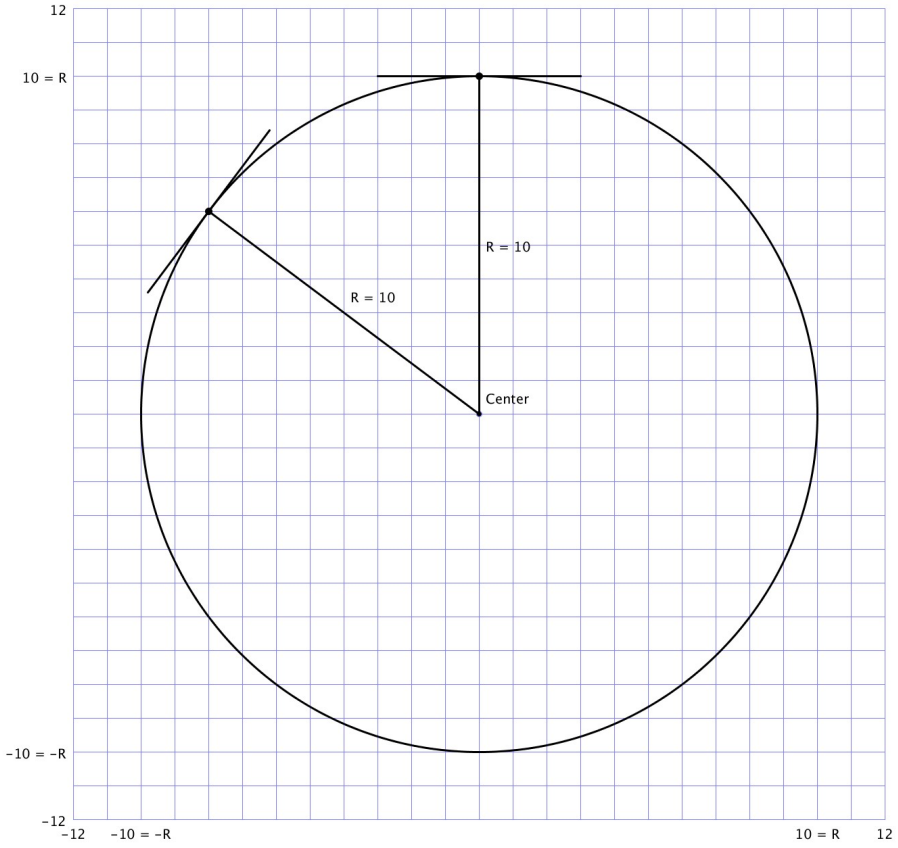

Figure 4.2: The Circle $x^2 + y^2 = R^2$

for determining the radius of a spherical mirror – simply measure the distance from your eye to the mirror when your eye fills the entire mirror. We'll continue with an example using a spherical make-up mirror whose radius is ten inches.

We begin with the example shown in Figure 4.3. We are looking at our reflection from three inches in front of the make-up mirror. Our eyes are located at the points $(-1, 7)$ and $(1, 7)$. We use these two eyes for two purposes – to look at our reflection and also as features on our face that will be reflected in the mirror. Notice that our eyes are two inches apart. Our analysis has several parts:

- We find the point at which a light ray traveling from the left eye to the right eye bounces off the mirror using Fermat's Principle, which says that this light ray follows the shortest total path from the left eye to a point $(x, \sqrt{100 - x^2})$ on the mirror and then to the right eye. Finding this point requires the following three steps:

  - Defining a function $f(x)$ whose input (independent variable) x is the x-coordinate of the bounce point and whose output (dependent variable) is the total distance traveled by a light ray traveling from the left eye to the bounce point $(x, \sqrt{100 - x^2})$ on the mirror and then to the right eye.
  - Graphing the function $f(x)$. You can do this in various ways – with a graphing calculator, with computer software, or writing your own program.
  - Finding its minimum. Again, you can do this in various ways – with a graphing calculator, with computer software, or writing your own program. This is also a good Calculus problem when you are studying optimization. You will most likely get a numerical estimate rather than an exact answer.

  For this particular problem you can probably make an educated guess what the answer will be. The answer you get following the steps above should agree with your intuition. This is a good check on your work. Notice that because are are using calculations with some natural round-off error the answer is usually a bit off. That is OK. Our answers will be close enough to the exact answers. If you run into problems you can as a last resort see the graph and its minimum in Figure 4.7 in the Appendix on page 43.

  Mark the point you found on Figure 4.3 and draw the path followed by this light ray.

- Next we do exactly the same thing for a light ray traveling from the left eye to the mirror and then back to the same eye. If necessary, see Figure 4.8 in the Appendix on page 43. Mark the point you found on Figure 4.3 and draw the path followed by this light ray.

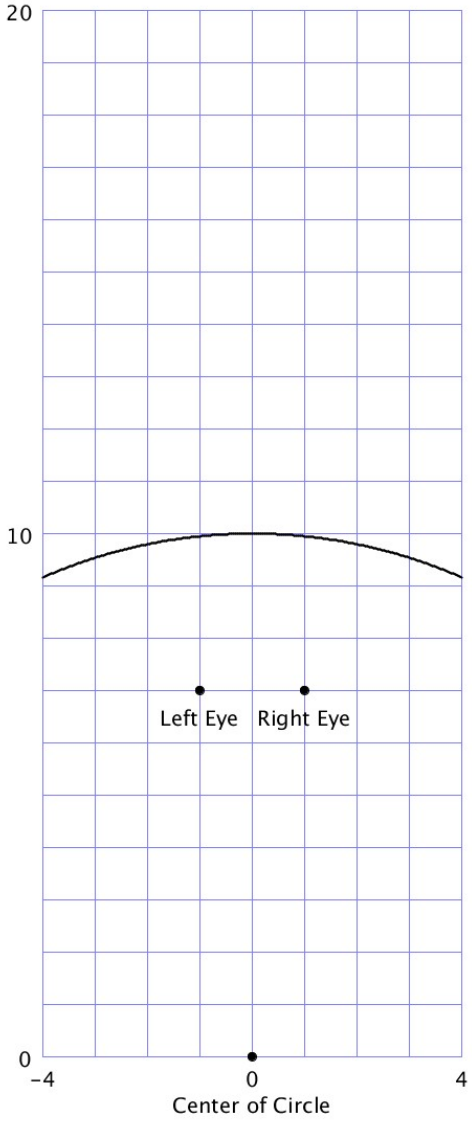

Figure 4.3: Example

- Our brain determines the location of the reflected left eye by combining input from the left eye and the right eye. For the left eye the brain assumes that the reflected left eye lies on a straight line that goes through the left eye and the bounce point and continues on behind the mirror – as if the mirror was not there. This line is called the "apparent path." Add this apparent path to Figure 4.3. This is the information received by the brain from the left eye.

- Do the same thing for the view that the right eye has looking at the left eye in the mirror – that is, add the apparent path for the right eye looking at the reflection of the left eye. This is the information received by the brain from the right eye.

- The apparent position of the left eye's reflection is the intersection of the two apparent paths. Mark it on Figure 4.3.

- Repeat the steps above to determine the apparent position of the right eye. What does all this tell you about the effects of a make-up mirror? Do an experiment to compare your result with what you actually see in your make-up mirror.

Although careful hand-drawn figures like the one you've been working on can tell us a lot, we can do much better with more mathematics, computers and calculators. In addition, more mathematics, computers and calculators make it much easier to explore other reflections in concave and convex mirrors.

You and your students need the following mathematical skills and this exploration is a good place to exercise these skills and apply them to obtain interesting results:

- The ability to use a linear function to describe a line that goes through two points – like the line that goes through the left eye and the point where a light ray traveling from the left eye to the left eye bounces off the mirror. We will use this skill to find the apparent paths.

- The ability to find the intersection of two lines. We will use this skill to find the apparent points. This ability is based on solving two linear equations in two unknowns.

Example 1 *The apparent path from the right eye at the point* $(1, 7)$ *looking at the mirror image of the left eye hits the mirror at the point* $(0, 10)$*. The linear function describing this line is*

$$y = -3x + 10.$$

Example 2 *The apparent path from the left eye at the point* $(-1, 7)$ *to its own mirror image hits the mirror at about the point* $(-1.4162, 9.8992)$. *See Figure 4.8 on page 43. The slope of this line is*

$$m \approx \frac{9.8992 - 7}{-1.4162 + 1} \approx -6.966$$

and its y-intercept is approximately 0.034. So its equation is

$$y = -6.966x + 0.034.$$

Example 3 *The intersection of the two lines*

$$y = -3x + 10 \quad \text{and} \quad y = -6.966x + 0.034$$

is approximately $(-2.51, 17.53)$.

Use these ideas to do a more precise analysis of this example. Check your answer experimentally and by comparing your answers with your rough drawings.

Back to Experimentation and Analysis:

Next use the same ideas to analyze what you see in a convex mirror. A good example uses a mirror of radius 10 inches whose center is at the point $(0, 20)$ and two eyes, one at the point $(-1, 0)$ and the other at the point $(1, 0)$. See Figure 4.4. Check your answer experimentally if you have a convex mirror.

Next we want to use the same ideas to study what you see in a concave mirror if you are looking at your reflection from a point beyond the mirror's center. See Figure 4.5. Your analysis will run into an unexpected problem. We've been using a natural, almost anthropomorphic version of Fermat's Principle – light rays follow the fastest path – almost as if they are in a hurry. However, when they bounce off mirrors they choose bounce points that are "critical points" of the distance function – points where the graph of the distance function has a horizontal tangent – like maximums or minimums. When you look at the distance functions that arise for this example, you will discover that they don't have minimums. The bounce points will correspond to maximums. See Figure 4.6.

You and your students should experiment with other questions – for example, does your nose appear longer or shorter in a make-up mirror?

The following appendix has worked examples with discussions. It is full of spoilers.

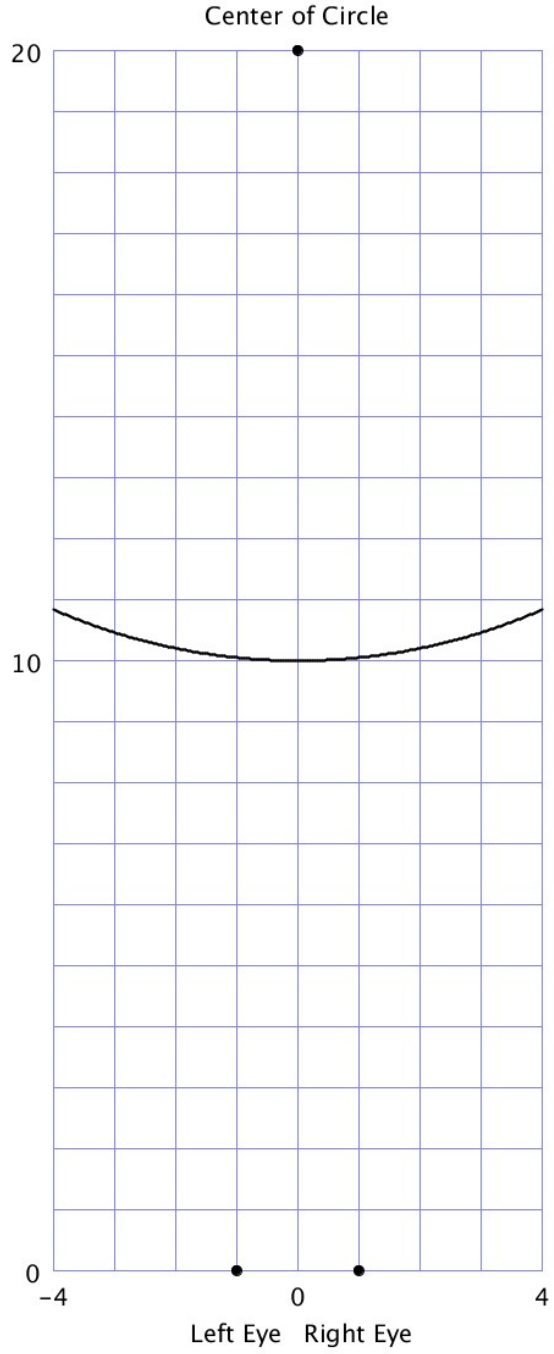

Figure 4.4: Reflection in a Convex Mirror

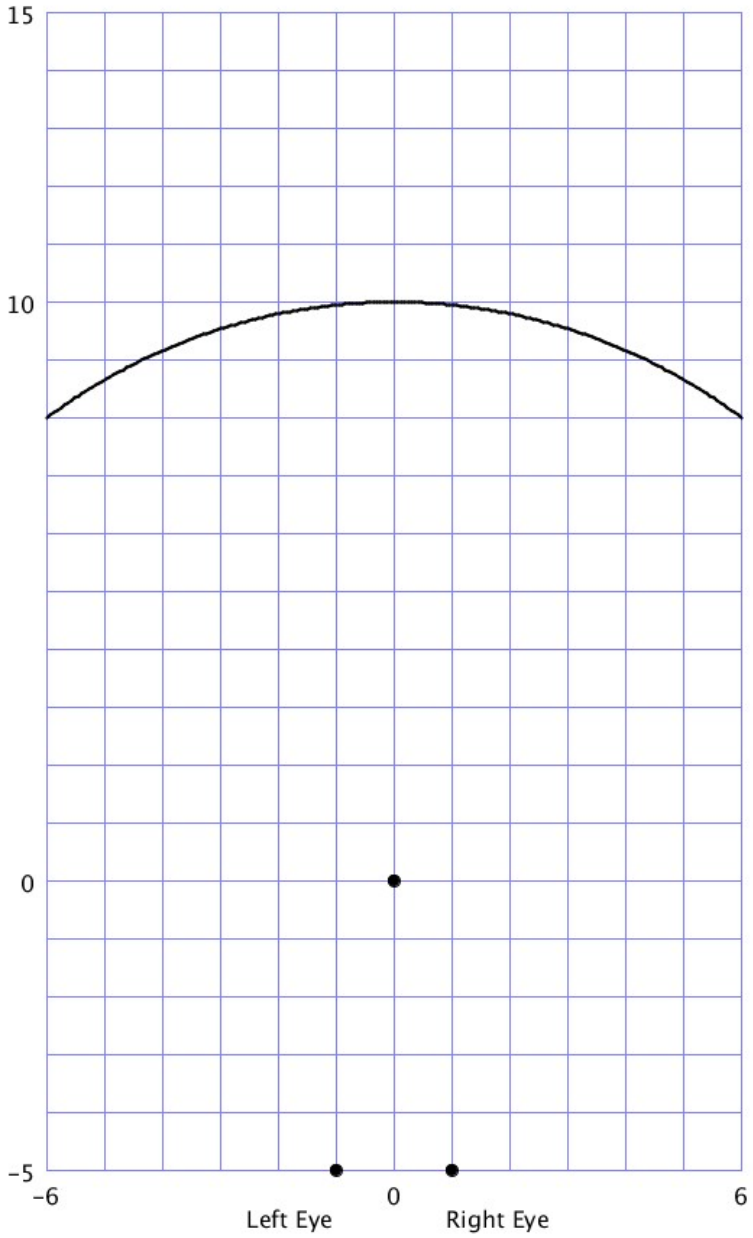

Figure 4.5: From the Other Side of the Mirror's Center

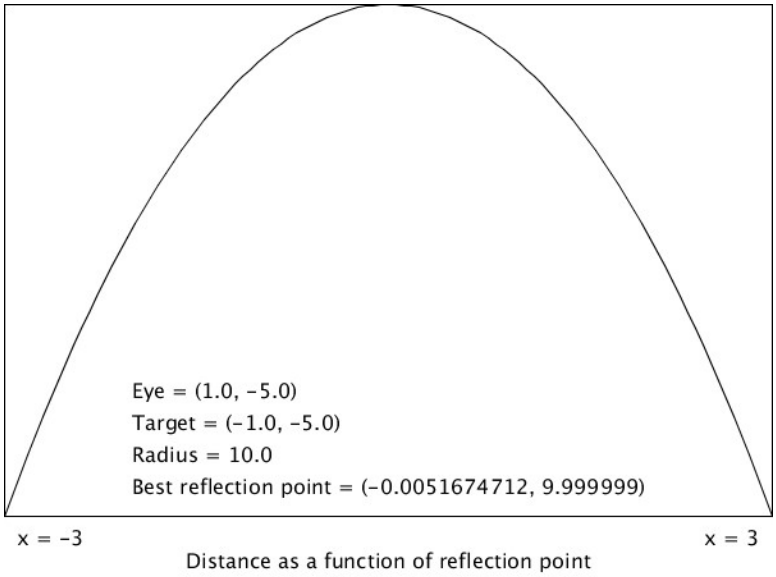

Figure 4.6: From the Other Side of the Mirror's Center – Finding a Maximum

Chapter Appendix – Spoilers

Spoiler Alert!!

The figures in this appendix are spoilers. Like all spoilers, these spoilers can ruin the fun – the joy of discovery. Students should build similar figures for themselves. You should also avoid looking ahead. We include this appendix because if you get stuck it can show you where we're going and motivate our plans for getting there. This appendix is not intended as a back-of-the-book set of answers used for checking work. The real check is comparing your mathematical analysis with what you see in real mirrors. In addition to creating figures like these we need to know how to interpret them and what they tell us about reflections in concave and convex mirrors. The units used in this figure are inches.

Figures 4.7, 4.8 and 4.9 show the first example. Figure 4.7 shows the graph and minimization we do to determine the path followed by a light ray from the left eye to the right eye.

Figure 4.8 shows the graph and minimization we do to determine the path followed by a light ray from the left eye to the left eye.

Figure 4.9 shows the completed drawing we use to find the apparent positions of the two eyes. Most of the work in Figure 4.9 is shown in black. For each eye you see the path

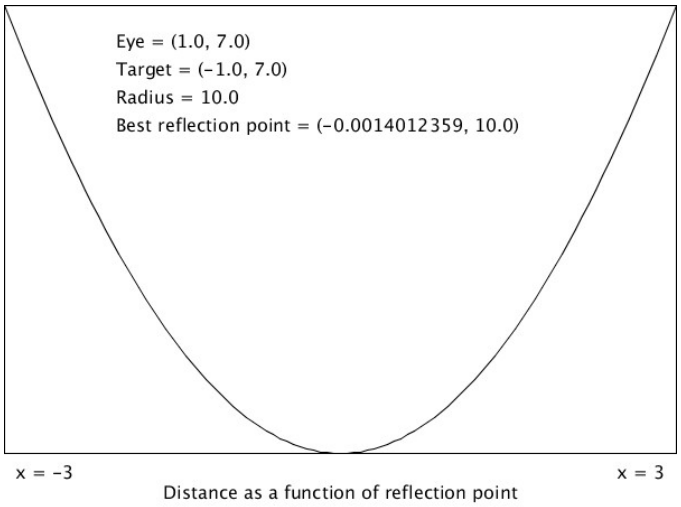

Figure 4.7: Finding the Minimum Distance for Opposite Eyes

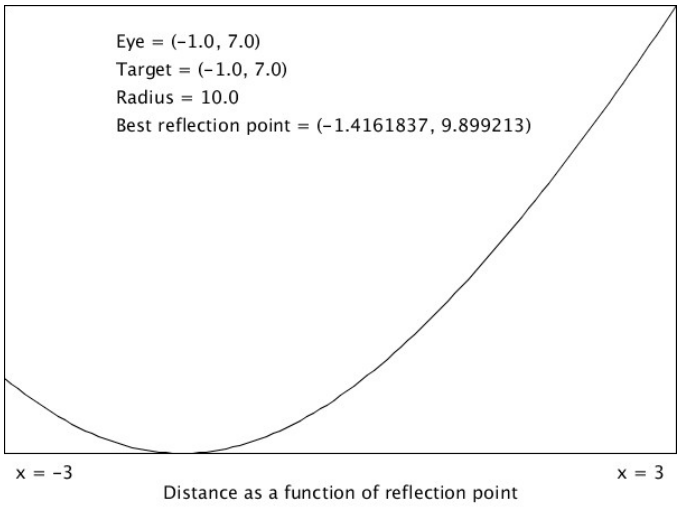

Figure 4.8: Finding the Minimum Distance for the Same Eye

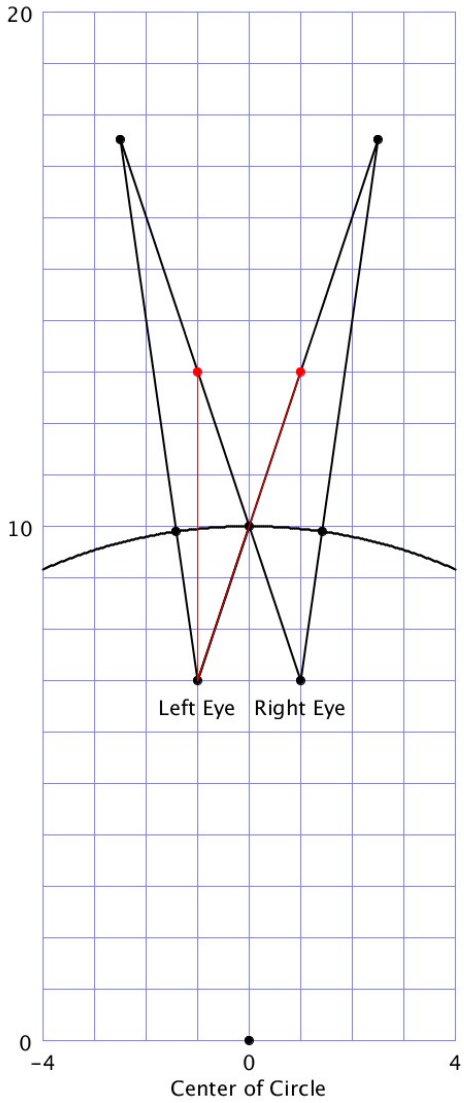

Figure 4.9: The Goal – Binocular Vision, Depth Perception and Magnification

followed from that eye to the mirrors and back to itself – this path determines what that eye "sees" as it looks at itself. Your brain does not actually see the mirror. It assumes that the incoming light ray is a straight line extending beyond the mirror. This extended line is called the "apparent path" and your brain assumes that the source of the light ray is someplace on the apparent path.

The figure also shows the actual paths from each eye to the other eye and the corresponding apparent paths. When your left and right eyes look at the reflection of the right eye your brain assumes that your right eye is at the intersection of the two apparent paths for the two eyes looking at the right eye. That point is marked by a black dot and is called the "apparent position" of your right eye. The apparent position of your left eye is determined and marked in the same way.

Figure 4.9 also shows the reflections of your two eyes in flat mirrors lying on the line $y = 10$ – tangent to the curved mirrors at the point $(0, 10)$. These points are marked with red dots. Understanding the effects of concave and convex mirrors requires comparing the black apparent positions for the curved mirrors with the red apparent positions for the flat mirrors.

Focus first on what your left eye sees as it looks at both eyes. The apparent positions of the two eyes are spread out more that they would have been if the mirror was flat. This is a relatively modest magnifying effect. Notice that if the mirrors was flat the apparent position of the two eyes would only be six inches from your eyes – too close for any but really near-sighted eyes to focus on. In contrast their apparent positions are more than ten inches away and you can probably focus on them. This is the real power of concave mirrors. In effect, they allow you to get really close to your reflection without it becoming blurred.

Figure 4.10 looks at our second example – reflection in a convex mirror. Notice that your reflection appears smaller than it would be if the mirror was flat. It also appears closer than it would if the mirror was flat. This might be a bit unexpected. The most common convex mirrors in our lives are passenger side rear-view mirrors and they often bear the warning – "objects are closer than they appear." Our brains use several cues to judge distance. One of those cues is size and because we are familiar with the size of other vehicles the fact that approaching vehicles seem smaller fools our brains into thinking they are further away.

Figure 4.11 looks at your reflection when you are on the other side of the concave mirror's center. This is more than a bit surprising. Notice that the left and right eyes are reversed. In addition, it looks as if your reflection is floating in front of the mirror!! You should check this in your make-up mirror.

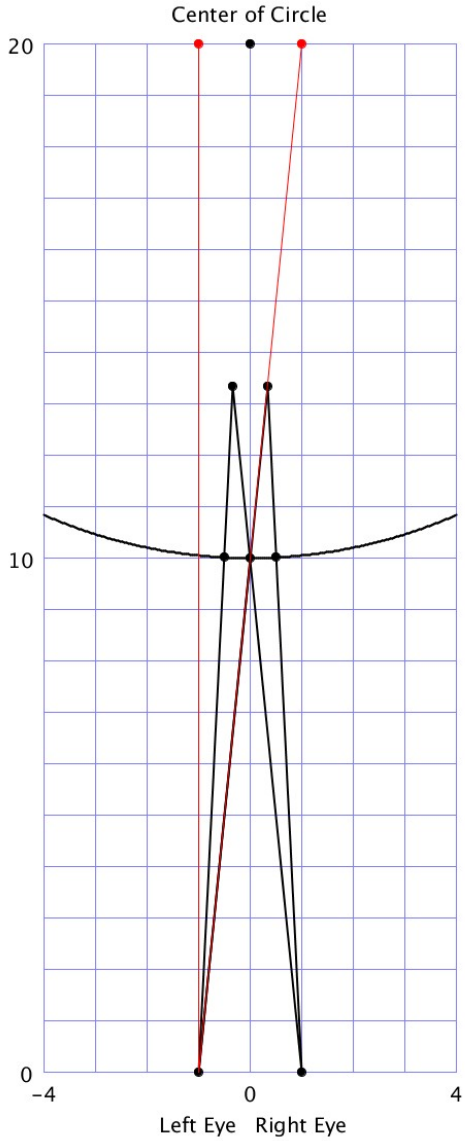

Figure 4.10: A Convex Mirror

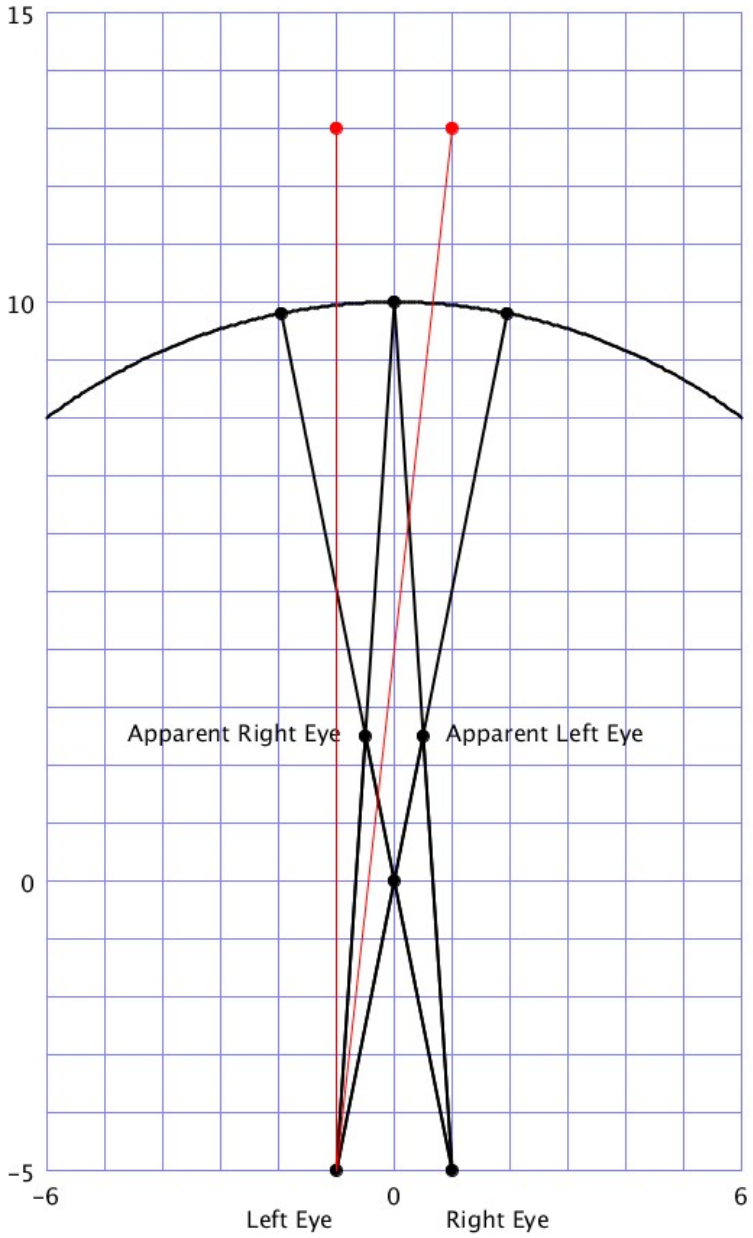

Figure 4.11: From the Other Side of the Mirror's Center

Chapter 5

Refraction in Washbasins and Ponds

Figure 5.1: Refraction in a Washbasin

> *As usual we begin with an experiment. Start with an empty washbasin as shown on the left in Figure 5.1 and notice the drain at the bottom. You might even take a quick picture. Then fill the washbasin with water and wait until any bubbles disappear and the water is still. See the right side of Figure 1.2. Notice the drain appears to be higher than it was when the washbasin was empty. You might even take a second picture from the same vantage point and compare the two pictures side-by-side.*

Whenever light rays cross from one medium, like water, to another medium, like air, their

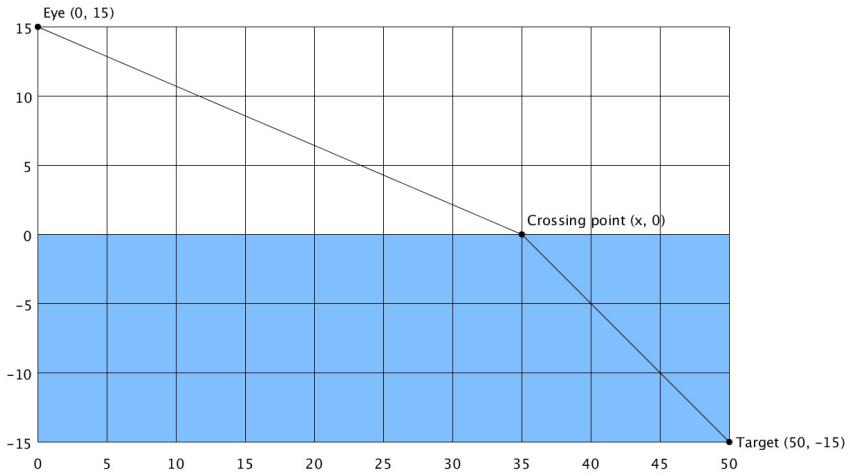

Figure 5.2: A Possible Path

path is bent. This behavior is called "refraction" and will provide additional evidence supporting the almost anthropomorphic first rough statement of Fermat's Principle – light rays follow the fastest possible path between two points.

Figure 5.2 shows a straightforward application of Fermat's Principle that can be used to understand the phenomena we see when we look at the drain in the bottom of a washbasin. In this figure an eye located 15 cm above the surface of the water in an aquarium is looking at a target 15 cm below the surface of the water at the other end of the 50 cm long aquarium. Using the coordinate system shown in the figure the location of the eye is $(0, 15)$ and the location of the target is $(50, -15)$. The x-axis runs along the surface of the water.

The shortest possible path from the target to the eye would be a straight line but this is not the fastest possible path for our anthropomorphic light because light travels faster (30.0 cm per nanosecond) in air than it does in water (22.5 centimeters per nanosecond). In Figure 5.2 we see one possible path light might follow. We use the letter x to denote the x-coordinate of the crossing point because we don't know where the best possible crossing point is. Because light travels faster in air than in water our anthropomorphic light rays will travel further in the air than in the water. The function

$$f(x) = \frac{\sqrt{x^2 + 15^2}}{30.0} + \frac{\sqrt{(50 - x)^2 + 15^2}}{22.5},$$

computes the total travel time for a light ray following the path shown in Figure 5.2. To

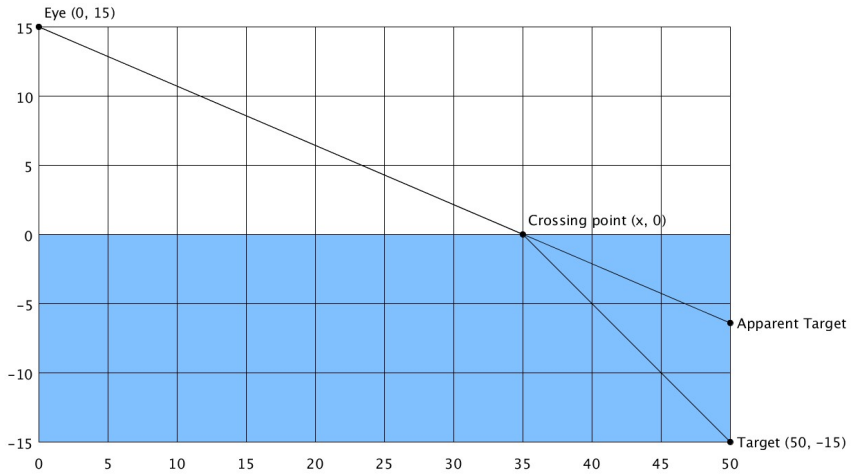

Figure 5.3: Apparent Path from One Eye

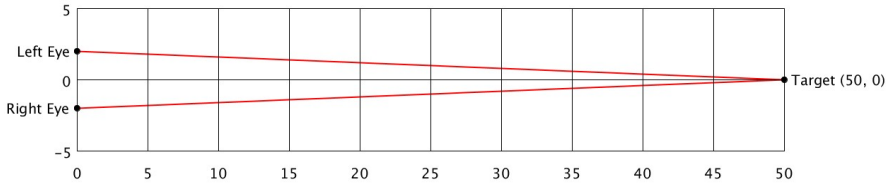

Figure 5.4: Two Apparent Paths from Above

determine the actual path followed by our light rays we must find the value of x that miniimizes the function $f(x)$. You can do this in various ways, for example, using a graphing calculator or computer software. When you plan activities for your students, the choice will depend on what they have available and what technology skills you want them to develop.

Question 1

Find the point $(x, 0)$ *where light traveling from the target at the point* $(50, -15)$ *to the eye at the point* $(0, 15)$.

When you look at an underwater target, you use two eyes and your brain determines the apparent location of the target using the information received from both eyes. For each eye alone we look at Figure 5.3 but we also need Figure 5.4, which shows a view looking

down on the aquarium from directly overhead. Think of your nose pointing at the target with one eye on each side of your nose looking at the target. The two red lines show the two apparent paths from above. Since we are looking directly down from above we don't see the bends in the apparent paths. The two apparent paths determined by the two eyes cross directly above the target. Thus, your brain thinks the x-coordinate of the target is unchanged from its actual x-coordinate.

Question 2

Find the apparent position of the target in the aquarium shown in Figures 5.2, 5.3 and 5.4. How deep does it appear to be?

Question 3

Suppose that a fish that is 5 cm long and 2 cm tall is 15 cm below the surface of the water at the end of our 50 cm aquarium. How long and how tall does it appear to be to our usual observer?

If you or any of your students are bow fishermen you or they won't be surprised by this analysis. Bow fishermen know that they must aim below a fish's apparent position. Our next work/play question will produce some real surprises for all with the possible exception of fly fishermen and scuba divers.

Question 4

Suppose a six foot tall fisherman is standing waist deep in a pool that is three feet deep and suppose that a fish at the bottom of the pool 20 feet away is looking at the fisherman. What does the fish see?

If you or your students have an underwater camera, you can use it to verify your surprising answer. You will also see a phenomenon that can't be understood by refraction alone. We will discuss this in the spoiler appendix.

The spoiler appendix below answers the last question. You should not look at it until after you have answered it for yourself.

Chapter Appendix

SPOILER ALERT!! Do not read this appendix until after you've answered the last question above yourself.

The rest of this page is purposely blank.

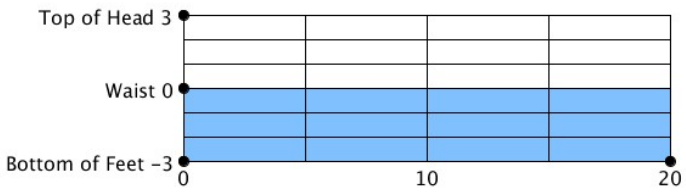

Figure 5.5: The Fish Looking at the Fisherman

Figure 5.5 shows the situation for Question 4. We have marked three features on the fisherman. The first step is to determine the apparent position of the top of the fisherman's head. As usual we begin by finding the point $(x, 0)$ where the light ray from the top of the fisherman's head at $(0, 3)$ crosses the surface of the water (the x-axis) on its way to the fish. We do this by finding the minimum of the function.

$$f(x) = \frac{\sqrt{x^2 + 3^2}}{30.0} + \frac{\sqrt{(20 - x)^2 + 3^2}}{22.5}.$$

The answer is $x \approx 16.71$.

Then we use this result to find the best path from the top of the fisherman's head to the fish and the apparent path the fish's eye reports to the fish's brain. This gives us the apparent position, $(0, 15.24)$, of the top of the fisherman's head as seen by the fish. See Figure 5.6. This is our first surprise. The top of the fisherman's head appears (to the fish) to be floating very high above its actual location!!

Next we want to find the apparent location of the fisherman's waist, whose actual location is right at the surface of the water. We actually have two choices:

- Let the location of the fisherman's waist be very, very slightly above the water – perhaps, $(0, 0.00001)$. In this case we go through exactly the same analysis as for the top of the Fisherman's head. The result is shown in Figure 5.7.
- Let the location of the fisherman's waist be very slightly below the water – perhaps, $(0, -0.00001)$. In this case light traveling from the fisherman's waist to the fish's eye travels entirely through the water and, therefore, follows a straight line. The apparent location of the fisherman's waist is its actual position.

The fish sees the fisherman cut in half at the waist with his torso floating high above the

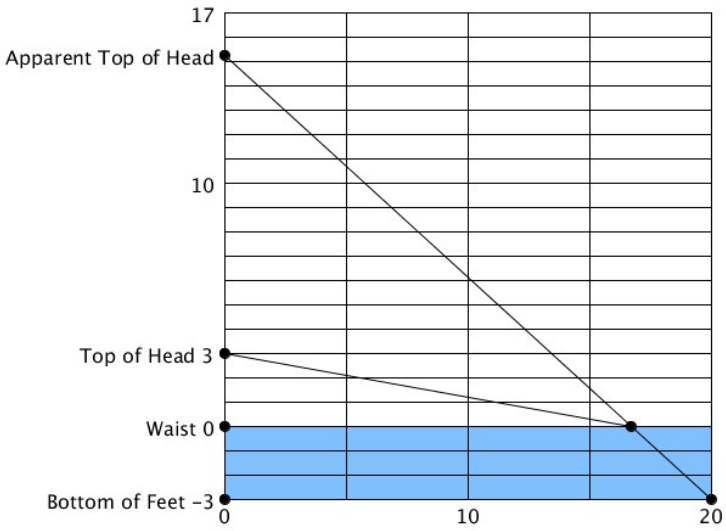

Figure 5.6: The Fish Looking at the Top of the Fisherman's Head

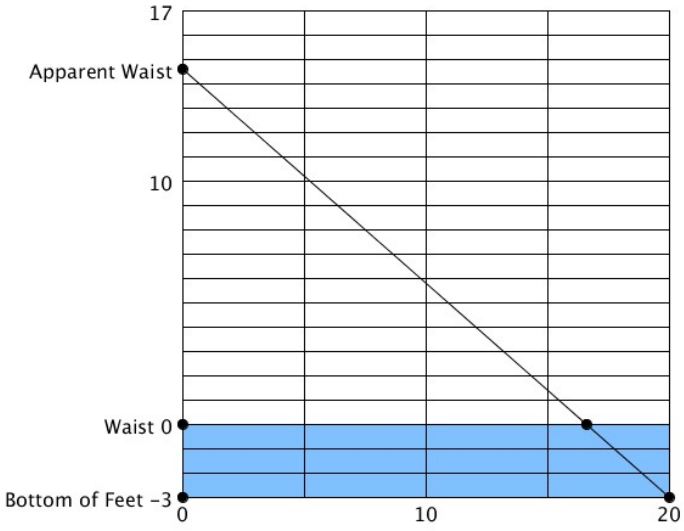

Figure 5.7: The Fish Looking at the Fisherman's Waist

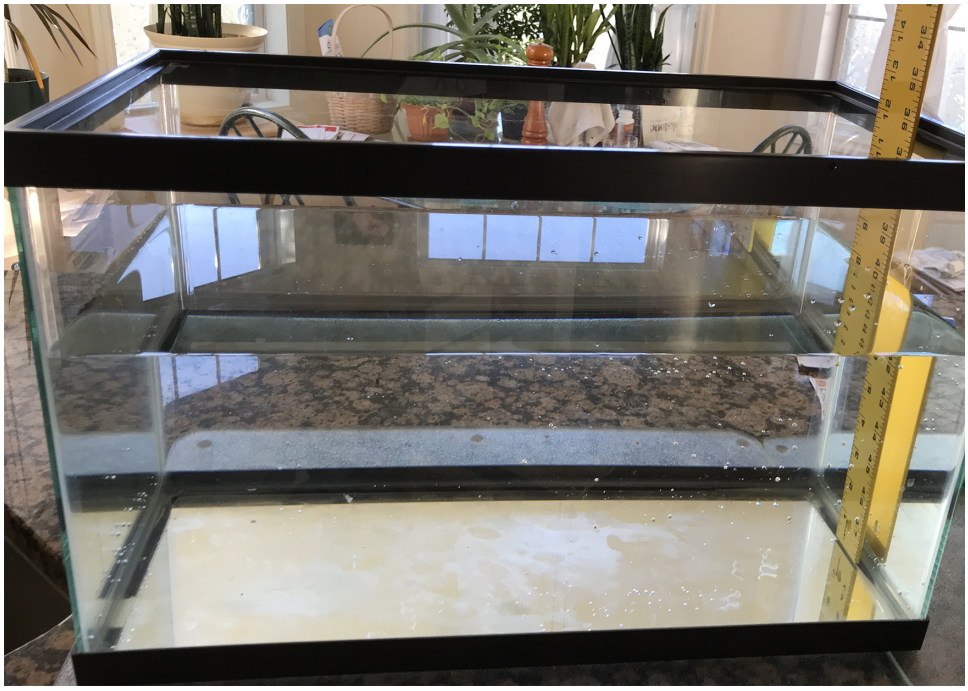

Figure 5.8: Kitchen Counter Experiment

rest of his body!! This cries out for some experimental verification. Figure 5.8 shows an experiment I did on my kitchen counter and Figure 5.9 which shows the result of this experiment. The photograph on the left side of Figure 5.9 was taken from above the water at the end opposite the yardstick. The photograph on the right side of Figure 5.9 was taken using an underwater camera at the bottom of the aquarium at the end opposite the yardstick on the right side. Notice that as predicted by our analysis the yardstick appears to the fish to be cut in half at the water's surface with the upper part of the yardstick floating high above the water.

There is another surprise. The region between the underwater portion of the yardstick and the above water part of the yardstick apparently floating high above the water is filled by a reflection of the underwater part of the scene.

This raises another set of experiments and efforts to understand their results. We already know from experience that we see two things when we look at the surface of water from above:

- We see reflections. See, for example Figure 5.10.

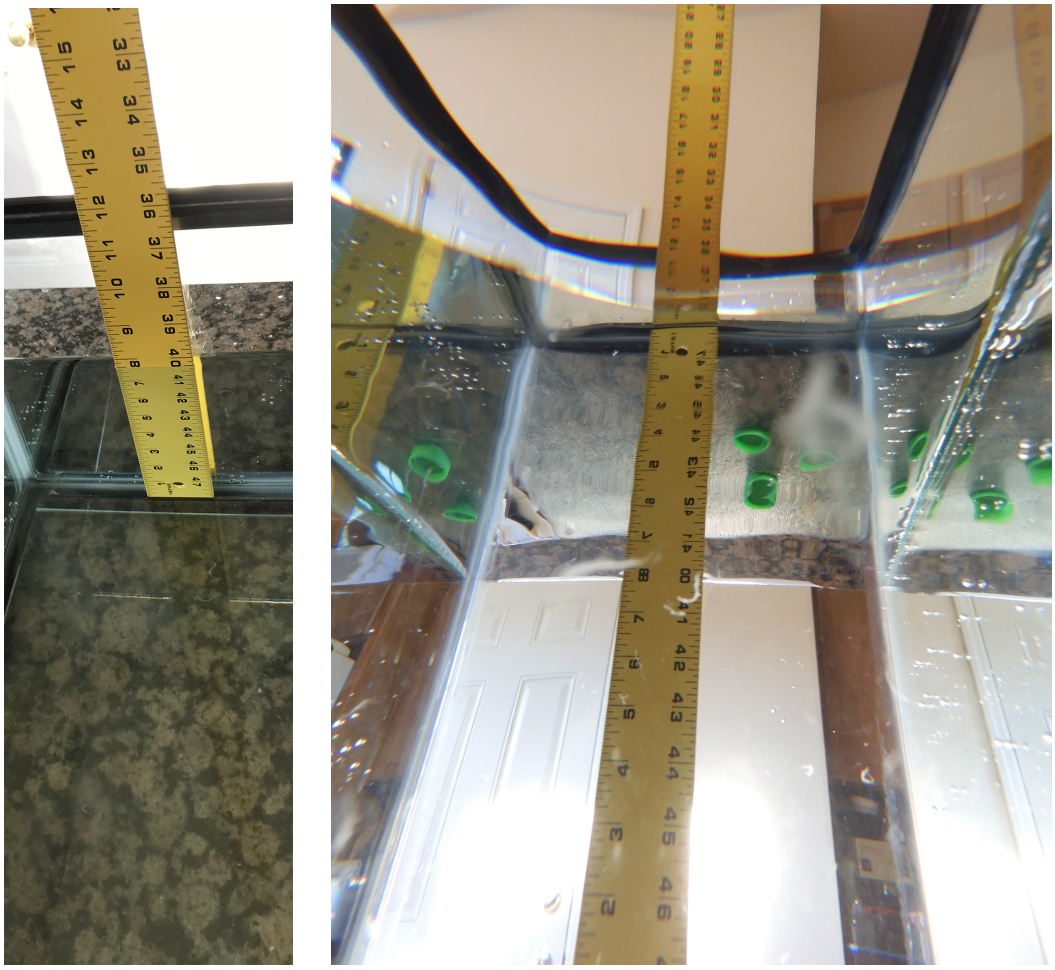

Figure 5.9: Verifying the Surprise View Seen by a Fish

Figure 5.10: Reflections in Hessian Lake at Bear Mountain State Park

- We see through the surface to whatever is underwater. See the left side of Figure 5.8.

We did the Figure 5.9 experiment using an aquarium on a kitchen table but this is another opportunity for back yard or school yard experiments using a toy swimming pool or even experiments in a school swimming pool. Experiments in swimming pools are difficult because you need the water to be still but they are possible. Scuba divers can also see these phenomena when the water is very still if they look up.

Chapter 6

Your Very Own Museum – Digital Photography

Figure 6.1: An Art Gallery in My Very Own Museum

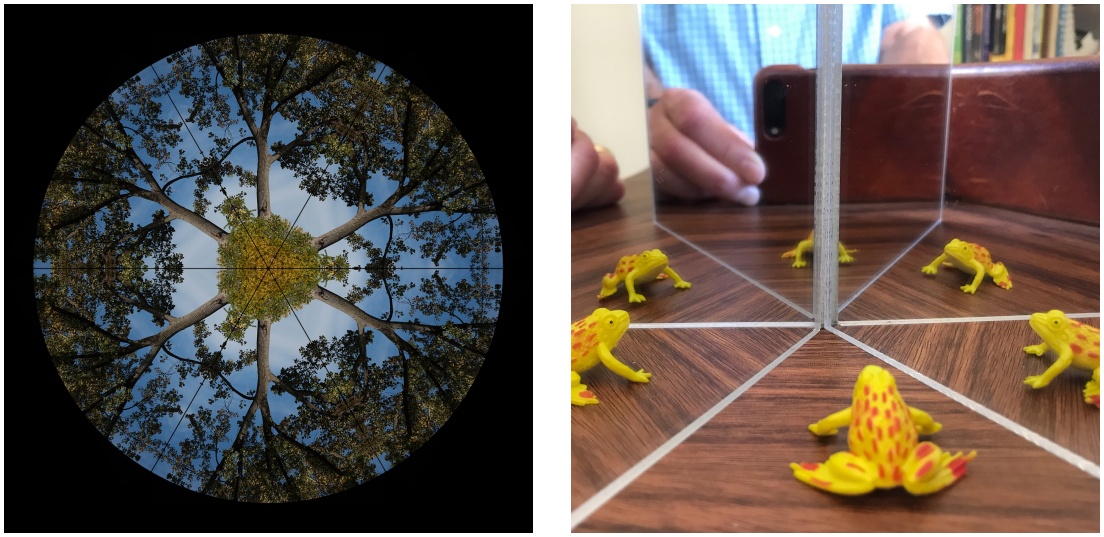

Figure 6.2: A Kaleidoscopic Image Inspired by a Real Kaleidoscope

6.1 Introduction

You can have your very own museum, an architectural marvel, ideally situated with its own cybercopter landing pad, ready to welcome visitors from all over the world or just across the street. A museum with large welcoming rooms where you can meet people and exchange ideas. ... and you can have your very own cybercopter, ready to carry you to other museums where you can talk with their artists and with other visitors.

This chapter is about creating and mounting exhibits for the galleries in your own museum. You can make drawings or take photographs using the digital camera built into your cell phone or other cameras. We will use the free computer language Processing to work with photographic images and even to create new images limited only by your imagination. For example, the left side of Figure 6.2 shows an image inspired by the real kaleidoscope on the right side of the same figure. I made this new image with a picture from my album and the digital image wizardry we will develop in this chapter. Digital creativity can be cheap – almost free – if you are able to code. The companion web site https://justaddmath.org/optics-at-home/ for these notes has a number of example program files and other auxiliary files for your use with this chapter and beyond. We chose Processing for two reasons but you may want to use a different language.

- Processing was developed by artists rather than computer scientists and is designed for the kinds of creative work we want to do.

Figure 6.3: A Sample Screen Image

- Processing has excellent online tutorials and documentation.

If you are not already familiar with Processing, go to its website https://processing.org download and install the necessary software and work/play through the introductory tutorial. We will assume that as you go through these notes you will also learn Processing using the resources on the Processing web site. These notes by themselves are not a programming course but they do contain many example programs that can help you learn programming.

6.2 Basics – Our First Program – FirstPointer

Images on a computer screen are made up of a rectangular array of tiny colored dots or pixels. For example, the image in Figure 6.3 is made up of 533 rows, each of which has 800 pixels. Our first Processing example program, **firstPointer**, illustrates the coordinate system used to locate pixels in an image like the image above. The pixels are numbered $0, 1, 2, \ldots 799$ from left to right and $0, 1, 2, \ldots 532$ from top to bottom. Each pixel is addressed by a pair, (x, y), of numbers, or coordinates. The first coordinate, x, is the location from left to right and the second coordinate, y, is its location from top to bottom. Thus, the address, or the coordinates, of the pixel at the top left is $(0, 0)$ and the address of the pixel at the bottom right is $(799, 532)$.

The program is listed below but you should download it from our web site, open it in Processing and work and play with it in Processing. Like all Processing programs, this one

is a folder that contains the program itself and any necessary auxiliary files. When you download the program from our web site it will be as a .zip file. After the .zip file has been unzipped, open it and double click the firstPointer.pde file to open it in Processing.

```
 1 String fileName = "sample.jpg";     // The name of the image file.
 2
 3 int width, height;                  // The width and height of the image.
 4 PImage photo;                       // PImage is an object that contains the information for an image.
 5
 6 void settings()                     // This routine runs once, before the setup routine.
 7 {
 8   photo = loadImage(fileName);      // This loads the file as an image.
 9   width = photo.width;              // The width of the image.
10   height = photo.height;            // The height of the image.
11   size(width, height);              // This line sets the width and height of the display area.
12 }
13
14 void setup()                        // The setup routine is run once after the settings routine.
15 {
16   image(photo, 0, 0);               // This displays the iamge in the graphics display area.
17 }
18
19 void draw()                         // The draw routine is run once for each frame of an animation.
20 {                                   // This one does nothing but the routine should not be omitted.
21 }
22
23 void mouseClicked()                 // The mouseClicked routine is run each time the user clicks the mouse.
24 {                                   // This routine is only needed to respond to mouse clicks.
25   print("You clicked at x = ");     // These lines print to the console below.
26   print(mouseX);                    // mouseX and mouseY are the mouse coordinates.
27   print(" y = ");                   // The print function prints to the console without a carriage return.
28   println(mouseY);                  // The println function prints to the console with a carriage return.
29 }
}
```

You should run the program and click anyplace in the image. You should see the coordinates of the pixel on which you clicked in the console below the program as seen in the screenshot in Figure 6.4. You may need to rearrange the windows on your desktop so that you can see the console.

Since this is our first example program we want to establish some useful habits, the first of which is the use of comments to document how the program works. Recall that any typing after a pair `//` are comments. Comments do not affect what the program does.

The heart of this program is four routines, or functions, that are executed at different times.

- `settings` This routine is executed at the very beginning when the program is first loaded. You may not have seen Processing programs that use this routine but it is extremely useful and should not be omitted. We use it to read an image file and open a display area whose width and height match that of the image file.

- `setup` This routine is executed immediately after the settings routine and is executed only once. In this program this routine places the image in the display area.

Figure 6.4: Running Our First Program

- `draw` This routine is executed over and over again. Many Processing programs produce interactive animations or movies and this routine is executed once for each frame. This routine should never be omitted. In this case it doesn't do anything but we need it to keep the program running, ready to respond to user action.

- `mouseClicked` This routine is one of many routines that respond to user action. This routine responds when the user clicks on a point in the display area. In this case it prints the $x-$ and $y-$coordinates of the pixel on which the user clicked in the console below the program.

The four routines are preceded by "global variables," variables that are shared by one or more of the routines. You can find more specific documentation for Processing commands, like `size`, by highlighting their names and choosing `Find in Reference` from the `Help` menu. You may also find the other Processing documentation, especially the tutorials, to be particularly useful.

6.3 Colors and the Processing Program – colorPointer

Superficially our second Processing program, `colorPointer`, is very similar to `firstPointer`. The only change is the `mouseClicked` routine, which now prints information about the

color of the pixel at which the user clicked. The new routine is shown below. You should download `colorPointer` and work/play with it in Processing.

```
24 void mouseClicked()              // The mouseClicked routine is run each time the user clicks the mouse
25 {                               // This routine is only needed to respond to mouse clicks.
26   color pixel;                  // Used for pixel information.
27   print("You clicked at x = "); // These lines print to the console below.
28   print(mouseX);                // mouseX and mouseY are the mouse coordinates.
29   print(" y = ");               // The print function prints to the console without a carriage return.
30   print(mouseY);                // The println function prints to the console with a carriage return.
31   pixel = photo.pixels[mouseY * width + mouseX];   // Get pixel color at point.
32   print("  red = ");            // Print the RGB values for this pixel.
33   print(red(pixel));
34   print("   green = ");
35   print(green(pixel));
36   print("   blue = ");
37   println(blue(pixel));
38 }
```

Even though, superficially, this program looks very similar to our first program, `firstPointer`, it actually introduces three new ideas:

- `firstPointer` used only global variables, which are shared by one or more routines. Complex programs with many different routines use many different variables, some of which are shared or "global" variables and others of which are "local" variables that are used only by one routine. This new `mouseClicked` routine uses the local variable `pixel`. This variable is of type `color`.

- The global variable `photo` of type `PImage` contains color information about each of the pixels in the image. This information is in the form of an array with one element of type `color` for each pixel. The array `photo.pixels` is this array. The value of the color variable for the pixel whose coordinates are (x, y) is the $(y * width + x)^{th}$ element of this array. The line

  ```
  31   pixel = photo.pixels[mouseY * width + mouseX];   // Get pixel color at point.
  ```

  puts this information for the pixel at which the user clicked in the variable `pixel`, which we recall is of type `color`.

- The three most immediate qualities of a color are its red, green, and blue components. You can think[1] of each pixel in the display area as having three tiny LEDs, a red LED, a green LED, and a blue LED. Each LED shines at a different intensity. The number 0 represents the lowest level, off, of intensity and the number 255 represents

[1] This is a useful simplifcation. In the same way you can think of each pixel on the sensor of your digital camera as having being made up of a red sensor, a green sensor, and a blue sensor. Both simplifcations are useful but wrong.

Figure 6.5: Color and Black-and-White Images

the brightest possible intensity. The color you see is produced by mixtures of red, green and blue. The code fragments `red(pixel)`, `green(pixel)` and `blue(pixel)` extract these three intensities.

Play with the program `colorPointer` to see the RGB (red, green, blue) components of the various colors in the image.

6.4 Black-and-White Photography and the Program – makeBW

You have probably noticed that many striking photographs are black-and-white rather than color. Black-and-white photographs and color photographs often set different moods. Black-and-white photographs often emphasize patterns and forms. Black-and-white photographs can also create a historical feel because historical photography is black-and-white. It wasn't until 1898 that color photography became generally available.

The Processing example program, **makeBW**, is designed to convert a color image, like the one on the left in Figure 6.5, to a black-and-white image, like the one on the right in the same figure. The program is listed below but you should download it from the usual web site and work/play with it in Processing.

```
1 String fileName = "sample.jpg";     // The name of the image file.
2
3 int width, height;                  // The width and height of the image.
4 PImage photo;                       // PImage is an object that contains the information for an image.
5
6 void settings()                     // This routine runs once, before the setup routine.
7 {
8   photo = loadImage(fileName);      // This loads the file as an image.
9   width = photo.width;              // The width of the image.
```

```
10   height = photo.height;           // The height of the image.
11   size(width, height);             // This line sets the width and height of the display area.
12 }
13
14 void setup()                       // The setup routine is run once after the settings routine.
15 {
16   image(photo, 0, 0);              // This displays the iamge in the graphics display area.
17 }
18
19 void draw()                         // The draw routine is run once for each frame of an animation.
20 {                                   // This one does nothing but the routine should not be omitted
21 }
22
23 void mouseClicked()                 // The mouseClicked routine is run each time the user clicks the mouse
24 {                                   // This routine is only needed to respond to mouse clicks.
25   color pixel;                      // Used for pixel information.
26   float red, green, blue, gray;     // Used for the RGB components of the pixel color
27   for(int i = 0; i < width * height; i = i + 1) // Do the following code for each pixel.
28   {
29     red = red(photo.pixels[i]);     // Get the RGB color values
30     green = green(photo.pixels[i]);
31     blue = blue(photo.pixels[i]);
32     gray = red/3 + green/3 + blue/3; // Use the average for the gray value
33     photo.pixels[i] = color(gray, gray, gray);   // Change the pixel to gray
34   }
35   photo.updatePixels();             // Update the image information with new pixel values
36   image(photo, 0, 0);               // Show the black-and-white image
37   save("sampleBW.jpg");             // Save the black-and-white image as "sampleBW.jpg"
38 }
```

The key line (line 32) computes a gray scale intensity by averaging the red, green and blue intensities. Then the red, green, and blue components of the color for the pixel are all set to the same gray scale intensity value.

The three lines (lines 35-37) after the loop update the pixels for the photo image, put the updated photo image in the display area and then save the image in the display area as a jpg file.

Programming Exercises

1. Make a copy of this program in a new folder and put the sample jpg file in the same folder. Rename the folder and program `makeRed`. Then modify `makeRed` to produce an image with just the red component of the color in each pixel. See Figure 6.6.

2. Create an image with just the green component.

3. Create an image with just the blue component.

4. Black-and-white photographers often use filters to create special effects. For example, yellow filters remove the blue component from a photograph. This has the effect of darkening the sky and making clouds stand out. The three images in Figure 6.7 show the effect of a yellow filter. The original color image is on the left, the unfiltered black-and-white image is in the center and the yellow-filtered black-and-white is on the right. Modify our program to digitally try different filters with a color image of your own.

Figure 6.6: Just the Red Component

Figure 6.7: Using Filters

Figure 6.8: Original Photograph

6.5 Spotlights and the Program – spotLight

Figure 6.8 is one of many group pictures I have in my photographic album. You probably have lots of group pictures like this too. With group pictures we often want to spotlight one particular person. See, for example, Figure 6.9. When you're shooting a movie or staging a play you can control the houselights to dim the background and shine a spotllght on one person. This requires both advance planning and equipment. We can accomplish the same thing with no special equipment and a program like `spotLight` and we can do it after the fact with no advanced planning. As usual you should download this program from the usual web site and work/play with it in Processing. The program is listed below.

Figure 6.9: Added Spotlight

```
 1 String fileName = "original.jpg";   // The name of the image file.
 2
 3 int width, height;                  // The width and height of the image.
 4 int centerX = 588;                  // The x-coordinate of the center of the spot.
 5 int centerY = 415;                  // The y-coordinate of the center of the spot.
 6 int radius  = 75;                   // The rediius of the spot.
 7 float darken = 0.60;                // Factor for darkening the background.
 8 float lighten = 1.50;               // Factor for lightening the spot.
 9 PImage photo;                       // PImage is an object that contains the information for an image.
10
11 void settings()                     // This routine runs once, before the setup routine.
12 {
13   photo = loadImage(fileName);      // This loads the file as an image.
14   width = photo.width;              // The width of the image.
15   height = photo.height;            // The height of the image.
16   size(width, height);              // This line sets the width and height of the display area.
17 }
18
19 void setup()                        // The setup routine is run once after the settings routine.
20 {
21   image(photo, 0, 0);               // This displays the iamge in the graphics display area.
22   loadPixels();
23 }
24
25 void draw()                         // The draw routine is run once for each frame of an animation.
26 {                                   // This one does nothing but the routine should not be omitted.
27 }
28
29 void mouseClicked()                 // The mouseClicked routine is run when the user clicks the mouse.
30 {
31    float dist;
32    float radiusSquared = pow(radius, 2);;
33    float red, green, blue;
34    color pixel;
35    int i;
36    for(int x = 0; x < width; x = x + 1)
```

```
37    {
38        for(int y = 0; y < height; y = y + 1)
39        {
40           i = y * width + x;
41           pixel = pixels[i];
42           if(pow(x - centerX, 2) + pow(y - centerY, 2) > radiusSquared)    // Darken pixels outside spot.
43           {
44              red = darken * red(pixel);
45              green = darken * green(pixel);
46              blue = darken * blue(pixel);
47           }
48           else
49           {
50              red = min(255, lighten * red(pixel));
51              green = min(255, lighten * green(pixel));
52              blue = min(255, lighten * blue(pixel));
53           }
54           pixels[i] = color(red, green, blue);
55        }
56    }
57    updatePixels();                  // Update display screen with modified array of pixels.
58    save("spotlit.jpg");             // Save image with spot.
59 }
```

Lines 4, 5 and six describe the area we want to be in the spotlight. In this example, the center of this area is at pixel coordinates $(588, 415)$ and it is a circle of radius 75 pixels. Line 45 specifies how much we want to darken the "houselights" – that is, the general illumination. In this example, we turn the general illumination down to 60% of its original value. Line 8 specifies how strong we want the spotlight to be. In this example, we multiply the original illumination by 1.50. For your photos you will experiment with these values to get the effect that you want.

The key lines are lines 36-56. We look at each pixel and determine whether it is inside the spotlit area or not (line 42). If it is outside the spotlit area we darken it (lines 44-46) and if it is inside the spotlit area we brighten it (lines 50-52). Notice that the maximum possible brightness of each color for each pixel is 255. Line 50, for example, "clips" the red brightness value at 255. Clipping is an unavoidable feature of photography. When you take a picture the film or sensor is only able to handle a limited amount of light and when you display an image the monitor is only able to produce a limited amount of light.

6.6 Transitions and the Program – wipe

This example program is our first animation. See Figure 6.10. Suppose we want a smooth transition from a scene that takes place in the White House (upper left) to one that takes place in the U.S. Capitol Building (upper right). At the end of the first scene we display the image of the White House. At the beginning of the next scene we display the image of the U.S. Capitol Building. The program `wipe` produces a transition called a "wipe." Starting at the left edge, each frame of the transition replaces a vertical line from the first scene with the corresponding vertical line from the next scene. The left bottom image in Figure 6.10 shows the frame one-third of the way through the transition and the right

Figure 6.10: Transitioning Between Two Scenes

bottom image shows the frame two-thirds of the way through the transition. The program `wipe` is shown below but as usual you should work with the program from the web site.

```
String firstImage  = "first.jpg";      // File name for the first image.
String secondImage = "second.jpg";     // File name for the second image.

int frame = -1;                        // Keeps track of frame number
PImage first, second;                  // PImage is an object that contains the information for an image.
int width, height;

void settings()                        // This routine runs once, before the setup routine.
{
  first = loadImage("first.jpg");      // Load the first image.
  second = loadImage("second.jpg");    // Load the second image.
  width = first.width;
  height = first.height;
  if((second.width != width) || (second.height != height))   // Check that the two images are the same size.
  {
    println("The two images are not the same size");
  }
  size(width, height);
}
void setup()                           // The setup routine is run once when the program (sketch )starts.
{
  image(first, 0, 0);                  // This displays the first image in the graphics display area.
  loadPixels();                        // Get array of pixel information for the first image (in the display area).
  second.loadPixels();                 // Get array of pixel information for the second image.
}

void draw()                            // The draw routine is run once for each frame of an animation.
{
  int i;
  if((frame == -1) || (frame >= width)) return;       // The wipe is in progress if 0 < frame < width
  for(int y = 0; y < height; y = y + 1)               // For each pixel in the frame column,
  {
    i = y * width + frame;
    pixels[i] = second.pixels[i];                     // Substitute the pixel from the second image
  }
  frame = frame + 1;                                  // Next we will substitute the next column.
  updatePixels();                                     // Update the pixel information for the display area.
}

void mouseClicked()                    // The mouseClicked routine is run each time the user clicks the mouse
{                                      // Start the wipe.
  frame = 0;                           // Start the substitution at the left edge.
  image(first, 0, 0);                  // This displays the first image in the graphics display area.
  loadPixels();                        // Get array of pixel information for the first image (in the display area).
}
```

The new element of this program is the `draw()` routine that is executed once for each frame of the animation after the user clicks the mouse button.

Programming Exercises

1. Create a transition in which the new scene comes in from the edges. See Figure 6.11.

6.7 Building a Digital Kaleidoscope

Now we want to the digital kaleidoscope shown on the left side of Figure 6.12. This is a complex undertaking and this section has several subsections and example programs.

Figure 6.11: Another Transition Between Two Scenes

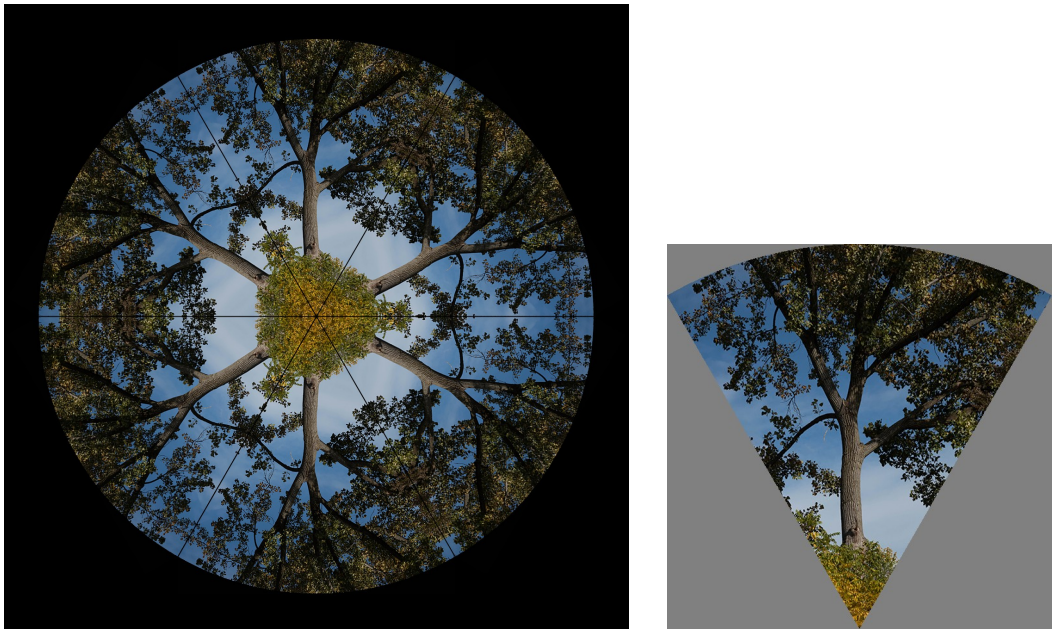

Figure 6.12: Our Goal

Figure 6.13: Our Goal

Notice that the digital kaleidoscope is made up of six copies of the wedge-shaped image shown on the right side of Figure 6.12 and the right side of Figure 6.13. This is an example of a **sprite**. Sprites are key building blocks for computer-generated graphics and animations.

6.7.1 Making Sprites and the Program – sprite

You already know how to put an image into the display area using a line like

```
image(photo, 0, 0)
```

but, although this works fine for rectangular images, we have to work a bit harder for sprites, which typically are not rectangular. We start with a rectangular image like the one on the left side of Figure 6.13. The sprite we want is shown on the right side of the same figure. The difference between the left and right side is that some of the pixels on the right side are transparent, so when you add it to a background, the background shows through. Mathematically, the opaque (nontransparent) part of the sprite is easily described using circles and lines. One technique that is often used to make fake images is to add sprites with more complicated boundaries. For example, you might want to create a fake image showing yourself on Mars. You would start with a photograph of yourself as you'd like to appear on Mars and make all the pixels that were not part of you transparent. You

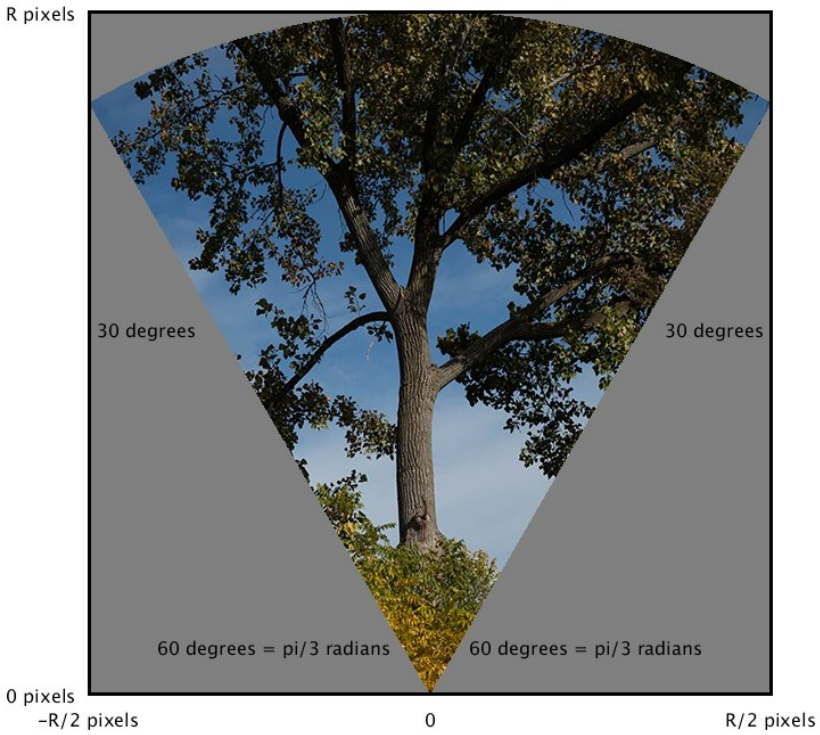

Figure 6.14: Describing the Sprite Geometrically

may have heard the terms "green screen" or "blue screen," that are used to describe one common way of doing this. For this section we'll just use simpler sprites whose boundaries can be described geometrically. For our kaleidoscopes we will always start with square images whose length and width are the same and are even numbers.

Figure 6.14 is the key to describing our sprite geometrically. We are using the most natural (mathematical) cooordinates for this figure and our unit of length is the "pixel" – that is the width or height of a pixel in an image. The kaleidoscope is made up of six wedge-shaped copies of the sprite. Since a whole circle is 360 degrees or 2π radians, the angle at the vertex of each wedge is 60 degrees or $\pi/3$ radians. Notice the two triangles, one on each side of the sprite. These are 30-60-90 degree triangles. The hypotenuse of each is a radius of the circle and, thus, has length R. The base of each has length $R/2$ and by the Pythagorean Theorem the height of each is $R\sqrt{3}/2$. The slope of the hypotenuse of the right triangle is $\sqrt{3}$ and the slope of the hypotenuse of the left triangle is $-\sqrt{3}$.

We want to make each point, with coordinates (x, y), transparent if it is outside the circle of radius R – that is, if

$$x^2 + y^2 > R^2$$

or if it is in one of the two 30-60-90 triangles, that is, if

$$y < |x|\sqrt{3}.$$

The listing of our first program, `sprite`, that makes and displays this sprite is shown below. If you look at lines 40 - 47 you will see the code that decides which pixels to make transparent. There are a few complications but those eight lines are the heart of the program.

```
 1 String fileName = "tree.jpg";                      // The name of the original image file.
 2 PImage sector;                                     // This PImage is used for the sprite
 3 int radius;                                        // Radius of the sector.
 4 int screenWidth;                                   // Width of the display area.
 5
 6 void settings()
 7 {
 8    sector = loadImage(fileName);                    // Get the image that will be cropped for the sector
 9    radius = sector.width;
10    if(sector.height != radius)                      // Check that the original image is square.
11    {
12        println("Error - image not square");
13    }
14    size(radius, radius);                            // Set up the display area.
15 }
16
17 void setup()
18 {
19     sector.loadPixels();                            // Load its pixels;
20     background(128);                                // Make the display background gray.
21     image(sector, 0, 0);                            // Place the sector in the display area.
22 }
23
24 void draw()                                         // Although this routine is not used,
25 {                                                   // it cannot be omitted.
26 }
27
28 void makeSector()                                   // This routine does the cropping
29 {
30    color transparent = color(255, 255, 255, 1);     // Transparent pixel color
31    int k;
32    float x, y;                                      // Natural (mathematical) coordinates
33    x = -radius/2 + 0.5;                             // x runs from -radius/2 + 0.5 to radius/2 - 0.5
34    for(int j = 0; j < radius; j = j + 1)            // For each column (x-coordinate)
35    {
36       y = 0.5;                                      // y runs from 0.5 to radius - 0.5
37       for(int i = 0; i < radius; i = i + 1)
38       {
39          k = (radius - 1 - i) * radius + j;         // Index for this pixel based on unnatural coordinates.
40          if(x * x + y * y > radius * radius)        // Make pixels outside circle transparent.
41          {
42             sector.pixels[k] = transparent;
43          }
44          if(y <= sqrt(3) * abs(x))
45          {
46             sector.pixels[k] = transparent;         // Make pixels outside wedge transparent.
47          }
48          y = y + 1;                                 // Next column (x-coordinate).
```

```
49      }
50      x = x + 1;                                  // Next row (y-coordinate).
51    }
52    sector.updatePixels();                        // Update the pixels.
53 }
54
55 void mouseClicked()                              // Show cropped sector when mouse is clicked.
56 {
57    makeSector();                                 // This routine does the cropping.
58    background(128);                              // Make the display background gray.
59    image(sector, 0, 0);                          // Place the sector in the display area.
60    save("result.jpg");                           // Save the result as a .jpg
61 }
```

The first complication is that an image is not made up of an infinite number of infinitely small points. Rather, it is made up of a finite number of tiny but not infinitely small pixels. The color of each pixel is determined by the color of the point in its center.

The program uses two nested `for` loops, lines 34 and 37, to go through all the pixels. Computer calculations are haunted by round off error. For example, $3 \times 1/3 = 1$ but $3 \times 0.3333333333 = 0.999999999 \neq 1$. For this reason it is particularly dangerous to use `float` variables in `for` loops.

Our two `for` loops use `int` variables. The outer loop, line 34, uses the `int` var `j` that represents the column of each pixel. Notice that before this loop starts, in line 33, a `float` variable `x` starts at $-R/2 + 0.5$. This is the x-coordinate of the center of the pixels in the first column. As this loop repeats, the value of `x` increases by 1, line 50, so that this variable specifies the x-coordinate of the center of the pixels in the column under consideration.

Similarly, the inner `for` loop, line 37, uses the `int` var `i` to go through the rows from bottom to top and the `float` var `y` to specify the y-coordinate of the center of the pixels in each row, lines 36 and 48.

Computer scientists made some very unnatural decisions when they implemented graphics on computer screens. The most obvious was the unnatural choice putting the origin at the upper left corner and having the y-axis run from top to bottom rather than the more natural bottom to top. The more insidious and serious bad decision was using what is called a left hand coordinate system for three dimensions rather than the right hand system in common use. Those decisions have caused huge problems. For us, the problem is easily solved. Because the computer science y-axis runs from top to bottom, the location `k` of the pixel in row `i` and column `j` is computed, line 39, by

```
k = (radius - 1 - i) * radius + j;
```

As usual, you should run this program to see what it does and look at it line-by-line to see how it does it. It is designed to make the sprite when the user clicks the mouse button. Then you should try to make some sprites with different shapes before going on.

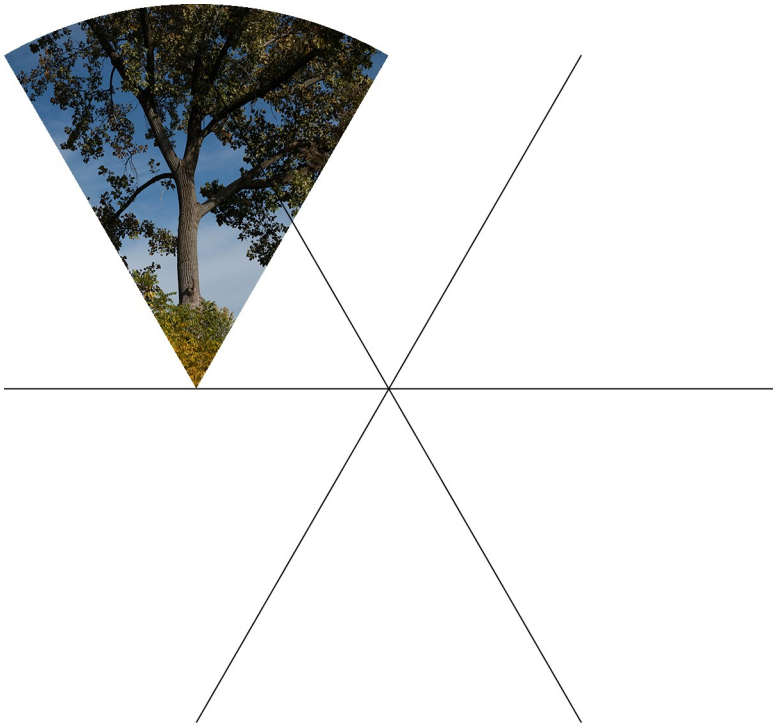

Figure 6.15: Placing Sprites in the Display Area

6.7.2 Placing Sprites on the Screen and the Program – spritePlay

Now we turn to placing sprites in the display area. We begin with a relatively short program `spritePlay` that will enable us to experiment with the Processing code for placing sprites in the display screen. The program `spritePlay` is listed below but as usual you should download it from our web site and work/play with it in Processing. Run this program. You should see the output shown in Figure 6.15 with a copy of our sprite in the upper left corner and lines setting up the boundaries of the six wedges or sectors that make up the kaleidoscopic image. The last lines, lines 33-69, make the sprite. Now we look at lines 20-28, the heart, of this program.

```
1 String fileName = "tree.jpg";                    // The name of the original image file.
2 int radius;                                      // Radius of the sector.
3 PImage sector;
4
5 void settings()
6 {
7    sector = loadImage(fileName);                  // Get the image that will be cropped for the sector
8    radius = sector.width;
```

```
 9    if(sector.height != radius)                      // Check that the original image is square.
10    {
11        println("Error - image not square");
12    }
13    size(2 * radius, 2 * radius);                          // Set up the display area.
14 }
15
16 void setup()
17 {
18    sector.loadPixels();
19    makeSector();
20    background(255, 255, 255);                        // Make the background white
21
22    image(sector, 0, 0);                              // Put the sprite in the display area.
23
24    stroke(0);                                        // Draw the lines between the six sectors.
25    strokeWeight(2);
26    line(xOf(-1), yOf(0), xOf(1), yOf(0));
27    line(xOf(-0.5), yOf(-sqrt(3)/2), xOf(0.5), yOf(sqrt(3)/2));
28    line(xOf(-0.5), yOf(sqrt(3)/2), xOf(0.5), yOf(-sqrt(3)/2));
29
30    save("firstFigure.jpg");
31 }
32
33 void makeSector()                                   // This routine does the cropping
34 {
35    color transparent = color(255, 255, 255, 1);     // Transparent pixel color
36    int k;
37    float x, y;                                       // Natural (mathematical) coordinates
38    x = -radius/2 + 0.5;                              // x runs from -radius/2 + 0.5 to radius/2 - 0.5
39    for(int j = 0; j < radius; j = j + 1)             // For each column (x-coordinate)
40    {
41       y = 0.5;                                       // y runs from 0.5 to radius - 0.5
42       for(int i = 0; i < radius; i = i + 1)
43       {
44          k = (radius - 1 - i) * radius + j;          // Index for this pixel based on unnatural coordinates.
45          if(x * x + y * y > radius * radius)         // Make pixels outside circle transparent.
46          {
47             sector.pixels[k] = transparent;
48          }
49          if(y <= sqrt(3) * abs(x))
50          {
51             sector.pixels[k] = transparent;         // Make pixels outside wedge transparent.
52          }
53          y = y + 1;                                 // Next column (x-coordinate).
54       }
55       x = x + 1;                                    // Next row (y-coordinate).
56    }
57    sector.updatePixels();                           // Update the pixels.
58 }
59
60
61 int xOf(float x)
62 {
63    return round(radius * x + radius);
64 }
65
66 int yOf(float y)
67 {
68    return round(radius - radius * y);
69 }
```

- Line 20 makes the background white.
- Line 22 puts the sprite in the display area with its upper left corner at the upper left corner of the display area. Notice that the white background shows through the transparent pixels of the sprite.
- Lines 24-28 draw the lines between the six wedges, or sectors. Figure 6.16 shows the most natural coordinate system that most people would use to draw the lines between

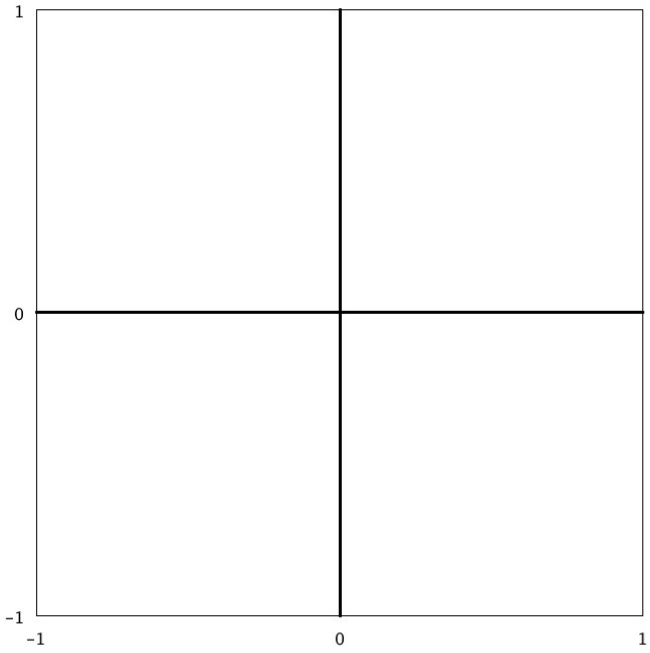

Figure 6.16: Natural Coordinates for Drawing Sector Lines

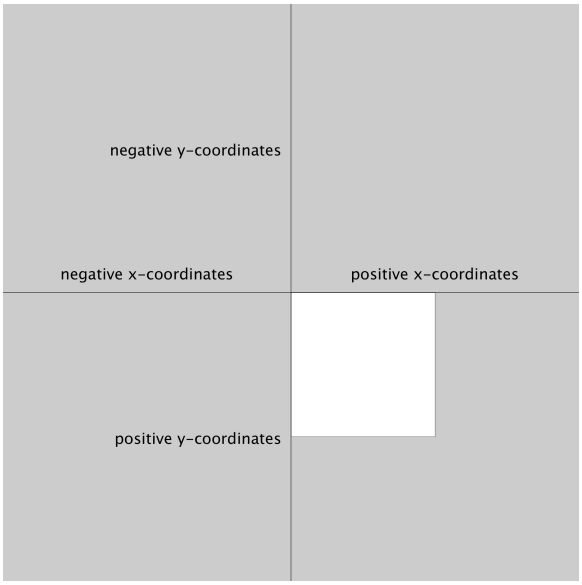

Figure 6.17: Potential and Actual Display Area

the wedges. Both the x- and y-axes run from -1 to 1; the x-axis runs from left to right and the y-axis runs from bottom to top. In this natural coordinate system the three lines that separate the six wedges go from $(-1, 0)$ to $(1, 0)$; from $(-1/2, -\sqrt{3}/2)$ to $(1/2, \sqrt{3}/2)$; and from $(-1/2, \sqrt{3}/2)$ to $(1/2, -\sqrt{3}/2)$. Before drawing these lines in the display area on the screen we need to convert from this natural coordinate system to the coordinate system used for the display area. This is the job of the functions `xOf` and `yOf`, lines 61-69.

The digital kaleidoscope is made up of six different copies of the same sprite placed in six different places and rotated or reflected in different ways. The key idea behind placing multiple sprites in different ways in different places in the display area is different sets of coordinates.

We begin with a virtual potential display area and the actual display area that appears on the screen. See Figure 6.17. The x-axis of the virtual potential display area runs from left to right with points represented by negative values of the x-coordinate to the left of the origin and positive coordinates to the right of the origin. The y-axis of the virtual potential display area runs from top to bottom with points represented by negative values of the y-coordinate above the origin and points represented by positive values of the y-coordinate below the origin.

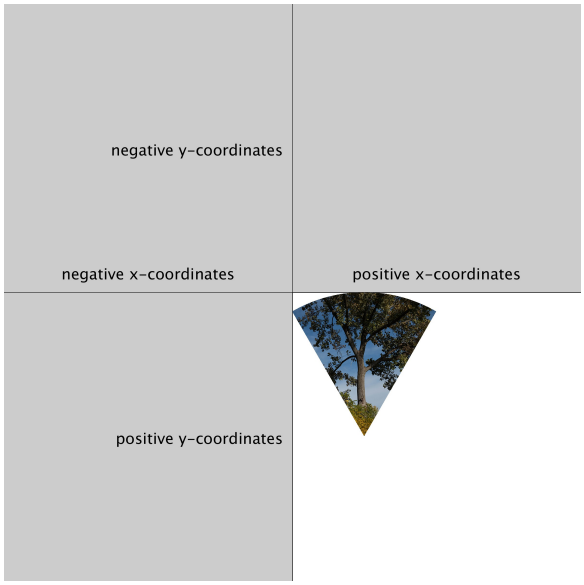

Figure 6.18: Working in the Potential and Actual Display Areas

The actual display area shown on the screen has its upper left corner at the origin of the potential display area and its width in in pixels is given by the first parameter of the `size` command and its height is given by the second parameter. Notice the actual display area (the white area) is below and to the right of the origin. The potential display area is infinite but the actual display area is limited by the size of the display area on the screen as specified by the `size` command.

Figure 6.18 shows how we work with the potential and actual display areas. We will eventually place the kaleidoscope in the actual display area. Line 22 of `spritePlay` places the sprite on the potential display area with its upper left corner at the origin so that it appears in the actual display area.

Our kaleidoscope will have six copies of the same sprite placed in six different places in the actual display area. For different copies we might want to rotate the sprite around its vertex and we might want to reflect it around the vertical line that runs through its vertex. Languages like Processing were designed to do exactly these kinds of things, not just in two dimensions, but also in three dimensions and not just with simple sprites like our wedges but with complex sprites that are made up of many moving parts. For example, a human sprite would have a neck, head, arms, and legs . . . and hands, fingers, and toes. A computer-generated gymnast would be rotating about her center of gravity as her center-of-gravity moved. At the same time her arms might be rotating around her shoulder joints

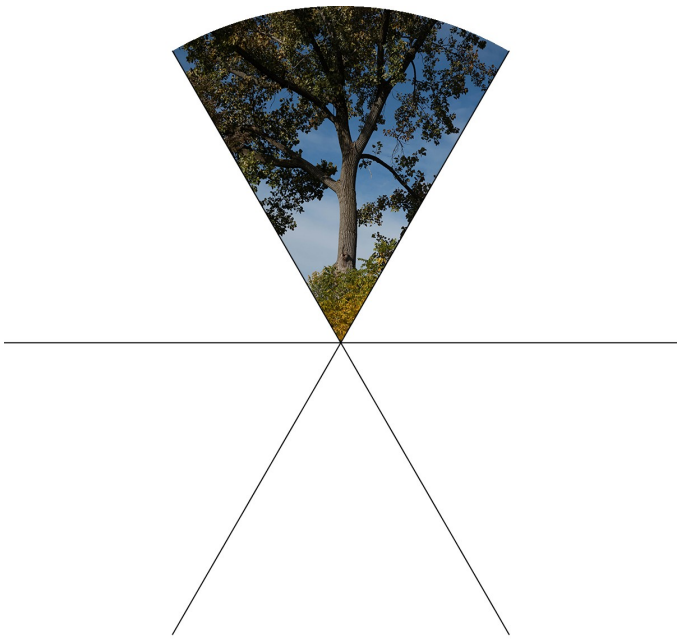

Figure 6.19: The First Copy of the Sprite

and ....

Our first copy of the sprite is shown in Figure 6.19. Notice that it has been reflected in the y-axis and translated or shifted.

We will use three of Processing's computer graphics commands: `translate`, `scale` and `rotate`. These commands affect the entire infinite potential display area.

- The command `translate(x, y)` slides the contents of the potential display area `x` pixels left or right. If the value of `x` is positive the contents slide to the right and if the value of `x` is negative they slide to the left. If the value of `y` is positive they slide down and if the value of `y` is negative they slide up. Compare the position of the sprite in Figure 6.18 with its position in Figure 6.20. Figure 6.20 shows the result of applying `translate(-radius/2, -radius)` to the sprite in Figure 6.18. This is line 24 in the modified version of `spritePlay` shown below.
- The command `scale(x, y)` scales the sprite by a factor of `x` in the x-direction and a factor of `y` in the y-direction. The origin is fixed. If the absolute value of `x` is less than one then the sprite shrinks in the x-direction. If the absolute value of `x` is bigger

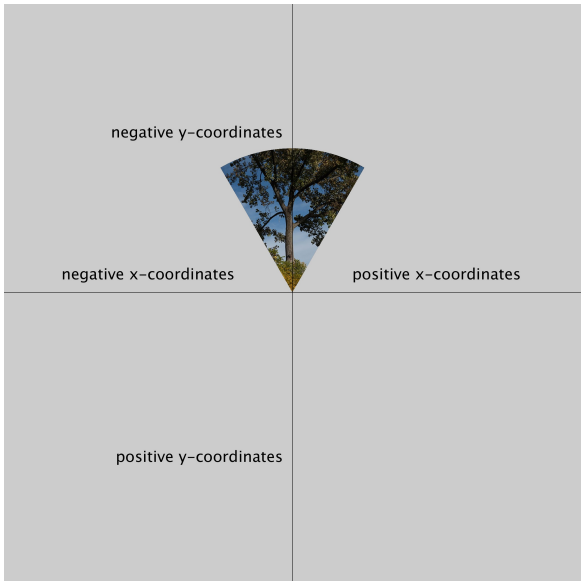

Figure 6.20: First Step

than 1 than the sprite is magnified in the x-direction. If the value of `x` is negative then the sprite is also flipped, or reflected, in the x-direction as if there were a mirror along the y-axis. Similar comments apply to `y`. We use the command `scale(-1, 1)` to flip our sprite around the y-axis. This takes us from Figure 6.20 to Figure 6.21. This command is line 23 in the listing above. The order of these lines seems, at first, to be unnatural – the reverse of what you might expect.

- Finally, we use the command `translate(radius, radius)` (line 22) to move the sprite from its position in Figure 6.21 to its position in Figure 6.22.

```
1 String fileName = "tree.jpg";                    // The name of the original image file.
2 int radius;                                      // Radius of the sector.
3 PImage sector;
4
5 void settings()
6 {
7    sector = loadImage(fileName);                  // Get the image that will be cropped for the sector
8    radius = sector.width;
9    if(sector.height != radius)                    // Check that the original image is square.
10   {
11       println("Error - image not square");
12   }
13   size(2 * radius, 2 * radius);                          // Set up the display area.
14 }
15
16 void setup()
17 {
18   sector.loadPixels();
19   makeSector();
```

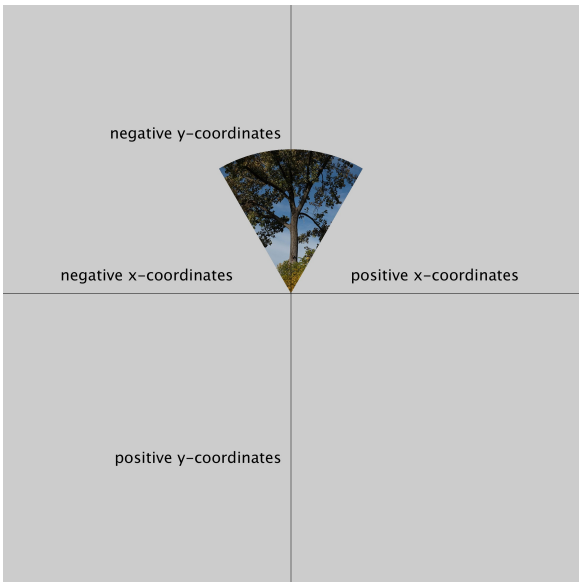

Figure 6.21: Second Step – Mirror Image

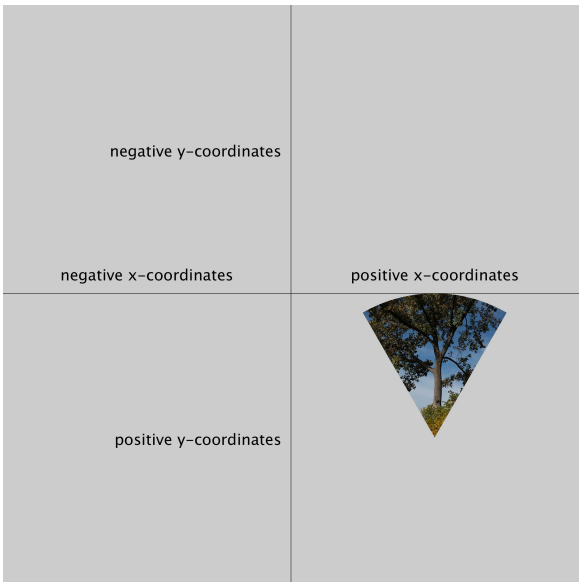

Figure 6.22: Third Step – Translate

```
20    background(255, 255, 255);                          // Make the background white
21
22    translate(radius, radius);                          // Finally move the sprite down into the actual display area.
23    scale(-1, 1);                                       // Then reflecrt the sprite in the y-axis
24    translate(-radius/2, -radius);                      // First put the vertex of the sprite at the origin
25    image(sector, 0, 0);                                // Put the sprite in the display area.
26
27    resetMatrix();                                      // Reset the transformation matrix to its original state.
28    stroke(0);                                          // Draw the lines between the six sectors.
29    strokeWeight(2);
30    line(xOf(-1), yOf(0), xOf(1), yOf(0));
31    line(xOf(-0.5), yOf(-sqrt(3)/2), xOf(0.5), yOf(sqrt(3)/2));
32    line(xOf(-0.5), yOf(sqrt(3)/2), xOf(0.5), yOf(-sqrt(3)/2));
33
34    save("firstFigure.jpg");
35 }
36
37 void makeSector()                                      // This routine does the cropping
38 {
39    color transparent = color(255, 255, 255, 1);       // Transparent pixel color
40    int k;
41    float x, y;                                         // Natural (mathematical) coordinates
42    x = -radius/2 + 0.5;                                // x runs from -radius/2 + 0.5 to radius/2 - 0.5
43    for(int j = 0; j < radius; j = j + 1)               // For each column (x-coordinate)
44    {
45       y = 0.5;                                         // y runs from 0.5 to radius - 0.5
46       for(int i = 0; i < radius; i = i + 1)
47       {
48          k = (radius - 1 - i) * radius + j;            // Index for this pixel based on unnatural coordinates.
49          if(x * x + y * y > radius * radius)           // Make pixels outside circle transparent.
50          {
51             sector.pixels[k] = transparent;
52          }
53          if(y <= sqrt(3) * abs(x))
54          {
55             sector.pixels[k] = transparent;          // Make pixels outside wedge transparent.
56          }
57          y = y + 1;                                   // Next column (x-coordinate).
58       }
59       x = x + 1;                                      // Next row (y-coordinate).
60    }
61    sector.updatePixels();                             // Update the pixels.
62 }
63
64
65 int xOf(float x)
66 {
67    return round(radius * x + radius);
68 }
69
70 int yOf(float y)
71 {
72    return round(radius - radius * y);
73 }
```

The lines separating the six wedges or sectors that will eventually make up our kaleidoscope are drawn by lines 27-32. Notice line 27, `resetMatrix()`, we will use this operation repeatedly. Each part of our image will be drawn using different transformations. Lines like lines 22-24 set up the transformations that will be used by line 25 to place one copy of the sprite in the display area. The operation `resetMatrix()` erases these transformations so they are not applied to later drawing operations, like those in lines 30-32.

It is important to keep the following items in mind as we build our kaleidoscope.

- For each item we place on the screen we first specify any transformations that should be applied. The transformations are specified in reverse order. See lines 22-24.

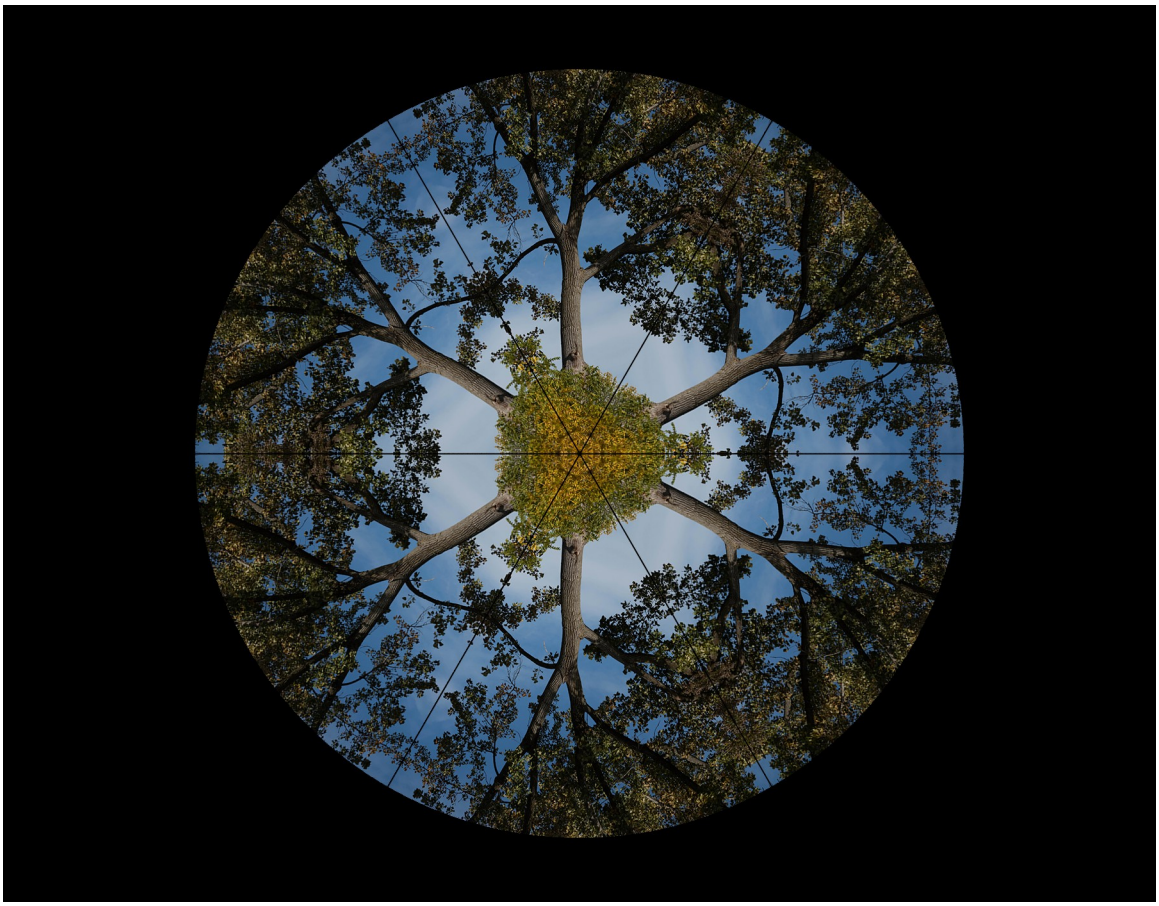

Figure 6.23: The Completed Kaleidoscope

- Then we specify the drawing operations, like line 25.
- We use the operation `resetMatrix()` to erase the transformations previously set before we work on another display element.

6.7.3 Putting it All Together – the Program kaleidoscope

Figure 6.23 shows the completed kaleidoscope built from the original square image shown in Figure 6.24. The program `kaleidoscope` is listed below but as usual you should download it from the usual web site and work/play with it in Processing.

Figure 6.24: The Original Square Image

```
String fileName = "tree.jpg";                       // The name of the original image file.
int topMargin = 100;                                // Border at top and b\ottom.
int leftMargin = 300;                               // Border at left and right.
int radius;                                         // Radius of the sector.  Must be even.
PImage sector;

void settings()
{
   sector = loadImage(fileName);                     // Get the image that will be cropped for the sector
   radius = sector.width;
   if(sector.height != radius)                       // Check that the original image is square.
   {
       println("Error - image not square");
   }
   size(2 * leftMargin + 2 * radius, 2 * topMargin + 2 * radius);        // Set up the display area.
}

void setup()
{
   sector.loadPixels();
   makeSector();
   background(0, 0, 0);                              // Make the background black

   translate(leftMargin + radius/2, topMargin);
   image(sector, 0, 0);

   resetMatrix();
   translate(leftMargin + radius, topMargin + radius);
   rotate(2 * PI/3);
   translate(-radius/2, -radius);
   image(sector, 0, 0);

   resetMatrix();
   translate(leftMargin + radius, topMargin + radius);
   rotate(4 * PI/3);
   translate(-radius/2, -radius);
   image(sector, 0, 0);

   resetMatrix();
```

```
40    translate(leftMargin + radius, topMargin + radius);
41    rotate(PI/3);
42    scale(-1, 1);
43    translate(-radius/2, -radius);
44    image(sector, 0, 0);
45
46    resetMatrix();
47    translate(leftMargin + radius, topMargin + radius);
48    rotate(PI);
49    scale(-1, 1);
50    translate(-radius/2, -radius);
51    image(sector, 0, 0);
52
53    resetMatrix();
54    translate(leftMargin + radius, topMargin + radius);
55    rotate(-PI/3);
56    scale(-1, 1);
57    translate(-radius/2, -radius);
58    image(sector, 0, 0);
59
60    resetMatrix();
61    stroke(0);                                        // Draw the lines between the six sectors.
62    strokeWeight(2);
63    line(xOf(-1), yOf(0), xOf(1), yOf(0));
64    line(xOf(-0.5), yOf(-sqrt(3)/2), xOf(0.5), yOf(sqrt(3)/2));
65    line(xOf(-0.5), yOf(sqrt(3)/2), xOf(0.5), yOf(-sqrt(3)/2));
66
67    save("kaleidoscope.jpg");
68 }
69
70 void makeSector()                                     // This routine does the cropping
71 {
72    color transparent = color(0, 0, 0, 1);       // Transparent pixel color
73    int k;
74    float x, y;                                         // Natural (mathematical) coordinates
75    x = -radius/2 + 0.5;                                // x runs from -radius/2 + 0.5 to radius/2 - 0.5
76    for(int j = 0; j < radius; j = j + 1)               // For each column (x-coordinate)
77    {
78       y = 0.5;                                         // y runs from 0.5 to radius - 0.5
79       for(int i = 0; i < radius; i = i + 1)
80       {
81          k = (radius - 1 - i) * radius + j;            // Index for this pixel based on unnatural coordinates.
82          if(x * x + y * y > radius * radius)           // Make pixels outside circle transparent.
83          {
84             sector.pixels[k] = transparent;
85          }
86          if(y <= sqrt(3) * abs(x))
87          {
88             sector.pixels[k] = transparent;          // Make pixels outside wedge transparent.
89          }
90          y = y + 1;                                   // Next column (x-coordinate).
91       }
92       x = x + 1;                                      // Next row (y-coordinate).
93    }
94    sector.updatePixels();                             // Update the pixels.
95 }
96
97 int xOf(float x)
98 {
99    return round(leftMargin + radius * x + radius);
100 }
101
102 int yOf(float y)
103 {
104    return round(topMargin + radius - radius * y);
105 }
```

We have discussed the key elements of this program in the preceding subsections but this new program has one new feature. The final image has black borders or margins for display purposes. The width of these borders is specified by the global variables `leftMargin` and `topMargin`. Remember that the original image used for the sprite must be a square image, whose width and height are the same even number. You should use this program or your

own version of it to produce your own digital kaleidoscopes. They make excellent birthday or holiday presents.

Chapter 7

Waves – Sound, Water and Light

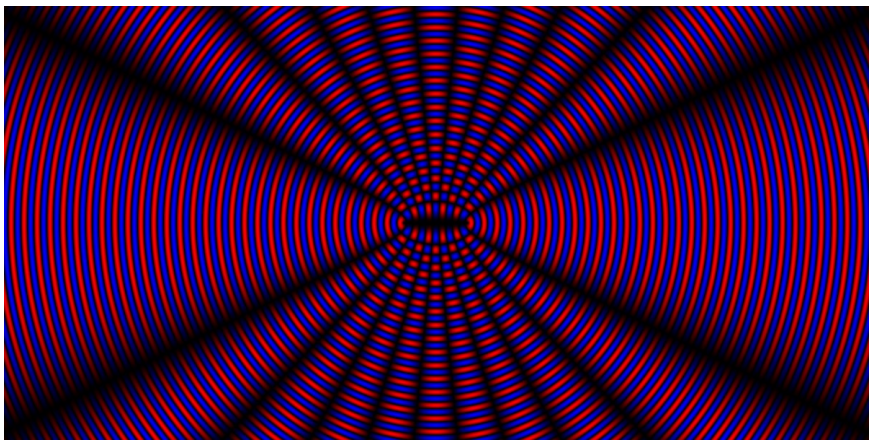

Figure 7.1: A Computer Simulation of Interference

7.1 Introduction

Waves are cool!! More than any other chapter this chapter will appeal to different students in different ways. People who love sound and music will be interested in sound waves. People who have heard about light waves, who love strange pictures like Figures 7.1 and 7.2 or who love programming will be interested in light waves and we all have experience with water waves in puddles and ponds. Give your students the opportunity to pursue their own interests.

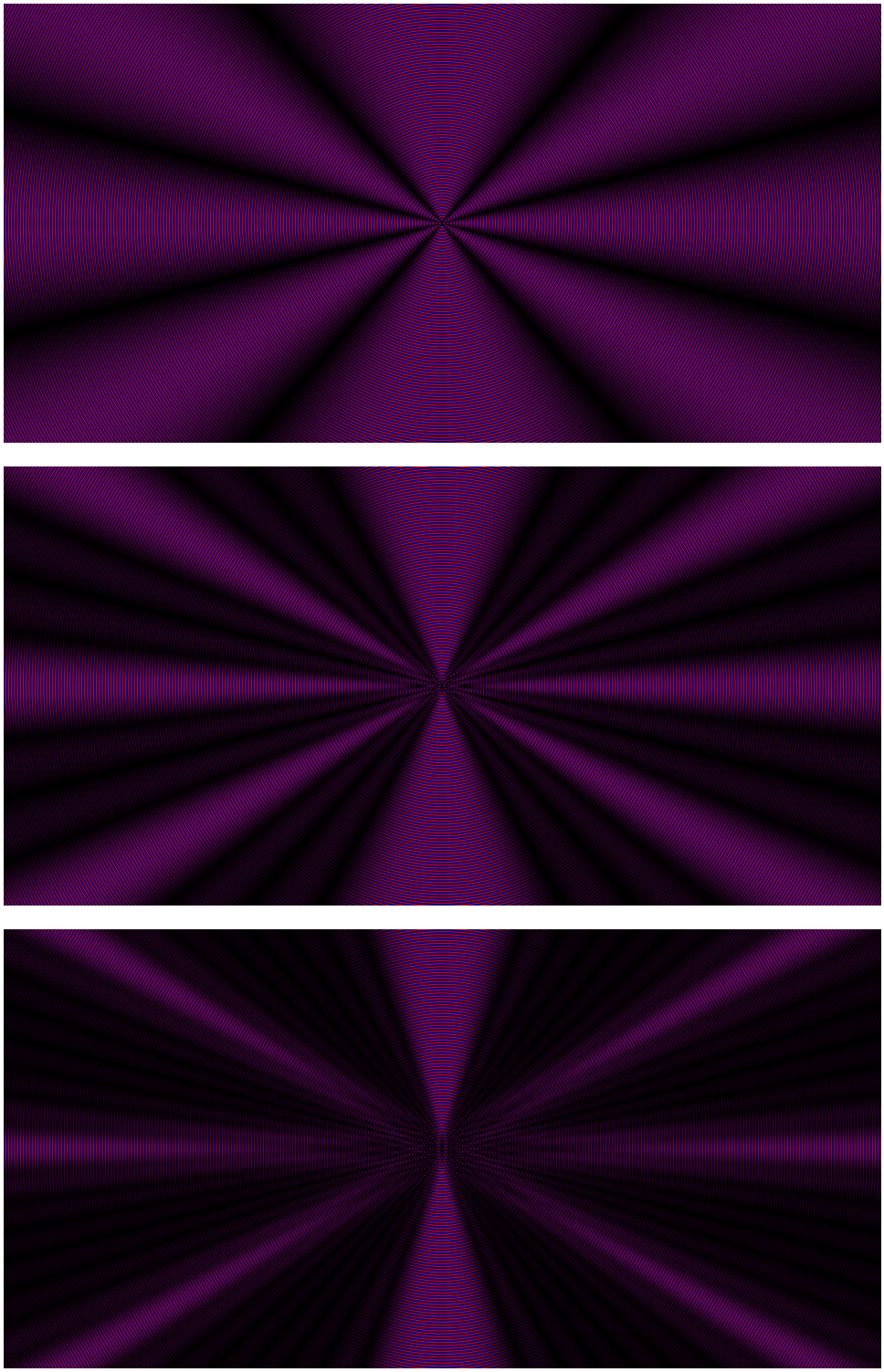

Figure 7.2: A Computer Simulation of Multiple Slit Diffraction (2, 4 and 8 slits)

Figure 7.3: Equipment for Diffraction Experiments

As usual we begin with some simple experiments that show why geometric optics, by itself, can't explain many optical phenomena. We will use three sets of readily-available equipment:

- Many cellphones have "flashlight apps" that are particularly well-suited for experiments that show the power of geometric optics. They use very small lights that cast sharp shadows, the same lights that are used for flash photography.

- Figures 7.1 and 7.2 show computer simulations of a phenomenon called "interference" that cannot be explained by geometry alone. This phenomenon is closely related to another phenomenon called "diffraction" that is easily seen with cheap lasers and diffraction glasses. See Figure 7.3. Lasers are hazardous. You must never look directly at a laser, even a cheap "low-powered" laser. You can buy lasers as laser pointers combined with remote presentation controls for computers for just over $10.00 and you can buy a set of three lasers of different colors as cat toys for under $20.00. You will also need a diffraction grating. They are available as "diffraction glasses" for $1.00 or $2.00. Both are available from Amazon.com or rainbowsymphony.com. Diffraction glasses are often sold at convenience stores for seeing rainbow patterns. See Figure 7.4.

- If you have a printer or copy machine and transparency paper you can print two copies of Figure 7.5, one on plain paper and one on transparency film. Cut both in half so you have two sets, each with one pattern of circles on plain paper and one on transparency film. Figure 7.6 shows one example of quick experiments you can do placing the transparency film copy on top of the plain paper copy. Try sliding the transparency copy around to see what happens.

Before diving into waves we explore the power and limitations of geometric optics by itself.

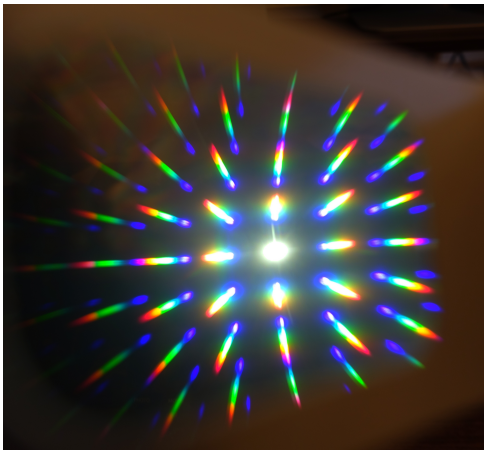

Figure 7.4: Looking Through Diffraction Glasses

7.2 The Power and Limitations of Geometric Optics

Our work in this section is based on a simple experiment. Use your cell phone flashlight app to cast a shadow of your hand on a wall. Your hand and the wall should be parallel to each other. Keeping your hand and the wall still move the light source back-and-forth away from and toward your hand. Describe what you see. Explain what you see using a figure like Figure 7.7 and a bit of geometry. This is the power of geometric optics.

Now do the same experiment using a laser as the light source and the diffraction glasses instead of your hand. Point the laser pointer through the center of one of the two lenses of the diffraction glasses. Describe what you see. Can you explain what you see with geometric optics?

7.3 The Sine Function and the Mathematics of Waves

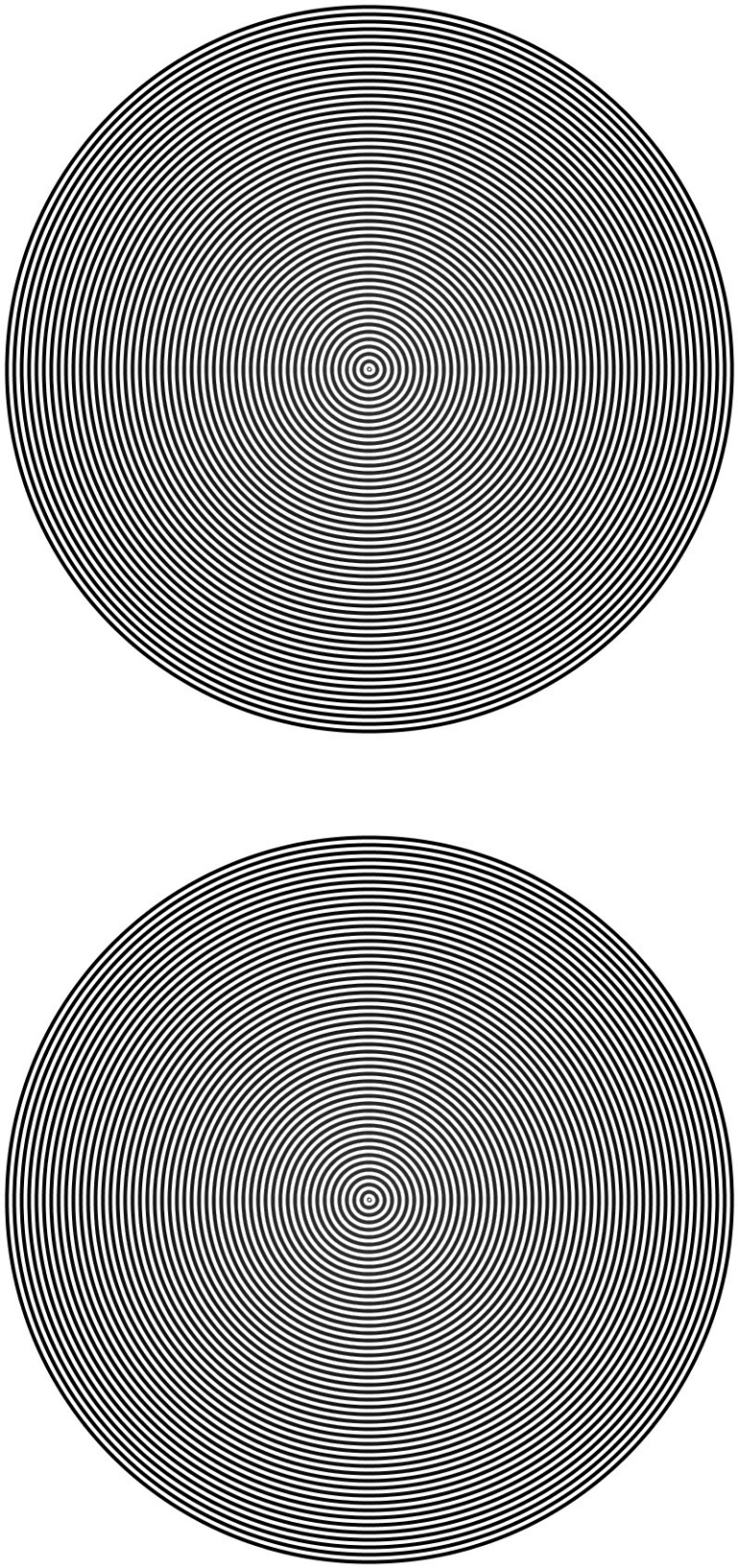

Figure 7.5: Print Your Own Interference Pattern Experiment

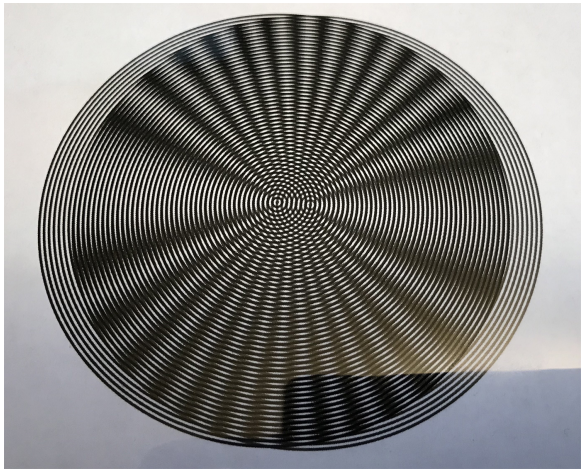

Figure 7.6: Quick Experiment

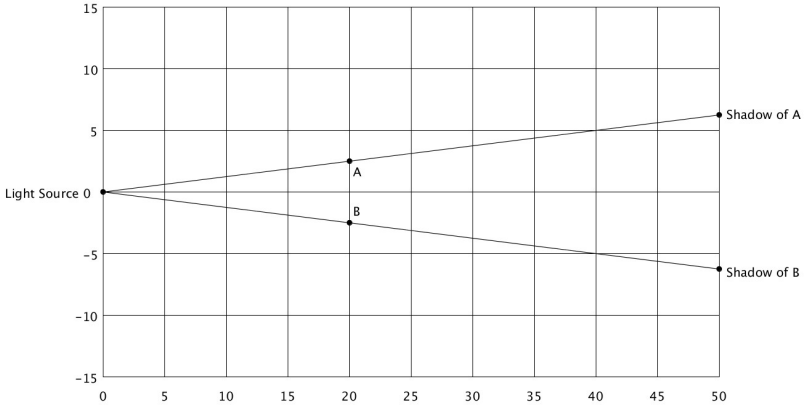

Figure 7.7: A Typical Diagram for Shadows on a Wall

Chapter 8

Three Dimensional Vision and Photography

Chapter 9

Beyond Narrative to Quests and Adventures