2007

POV-IT: the POV-Ray Interactive Tutorial

Eleanor O'Rourke
Colby College

Follow this and additional works at: https://digitalcommons.colby.edu/honorstheses

Part of the Computer Sciences Commons

Colby College theses are protected by copyright. They may be viewed or downloaded from this site for the purposes of research and scholarship. Reproduction or distribution for commercial purposes is prohibited without written permission of the author.

Recommended Citation

This Honors Thesis (Open Access) is brought to you for free and open access by the Student Research at Digital Commons @ Colby. It has been accepted for inclusion in Honors Theses by an authorized administrator of Digital Commons @ Colby. For more information, please contact mfkelly@colby.edu.
POV-IT:
The POV-Ray Interactive Tutorial

Nell O’Rourke

Senior Honors Thesis

Advisor: Dale Skrien
Department of Computer Science

Spring, 2007
ACKNOWLEDGEMENTS

First and foremost I would like to thank my advisor Dale Skrien for his constant help and support this year. I had a number of difficulties pinning down the focus of my project, and when my original idea did not work out Dale was there to help me design a completely new project and convince me that I would still be able to finish the thesis. Additionally, his comments and suggestions have greatly helped shape the system I finally developed, and for that I am greatly thankful. I would also like to thank Jon Hallstrom for inspiring me to work with POV-Ray in the first place. Even though my system did not prove particularly useful to his work in the end, he has always displayed great interest in my project. Finally, I would like to thank Clare Bates Congdon for agreeing to be my second reader and for continually keeping me on track with my work.

I would also like to thank my family for their continuing support. My Dad has inspired me throughout the year through his encouragement and strong interest in my work. He also provided me with the original idea for the POV-Ray Interactive Tutorial, and helped me develop a design for the system. His experience and advice has been invaluable. I would also like to thank my Mom for her continuing encouragement. She has always believed that I could finish this project even when I was not so sure of it myself, and her moral support has been wonderful. Last but not least, I would like to thank Russell for his continuing interest in the project and his excitement at being able to finally try out the system. Your positive feedback has made all of my work feel worthwhile.

Finally, I would like to thank my friends. Mike, Thomas and Andrea have always have provided endless advice and suggestions, which as been immeasurably helpful. I would especially like to thank Mike for telling me about Ruby on Rails and letting me borrow his book on and off all semester. I would also like to thank my non-technological friends for their continuing interest, and most especially Naomi for listing to my talk late at night and user testing my system throughout the semester. Finally, I would like to thank Alex for keeping me sane and always being supportive. Your constructive criticism and intellectual excitement in POV-Ray has been an endless inspiration.
# TABLE OF CONTENTS

Chapter 1: Introduction to POV-IT 1

Chapter 2: Project Motivation 5

Chapter 3: Background 11
- 3.1: Introduction 11
- 3.2: Ray Tracing 11
- 3.3: The POV-Ray Package 14
- 3.4: Interactive Web Design 18
  - 3.4.1: Perl CGI 19
  - 3.4.2: Ruby on Rails and Ajax 20
  - 3.4.3: Cascading Style Sheets 21
- 3.5: Conclusion 22

Chapter 4: The Implementation of POV-IT 25
- 4.1: Introduction 25
- 4.2: POV-IT System Design 25
- 4.3: Writing a Perl CGI for an Apache Web Server 27
  - 4.3.1: Configuring Apache 28
  - 4.3.2: Writing a Perl CGI 31
  - 4.3.3: Creating a User Interface 34
- 4.4: Problems Encountered 36
- 4.5: Incorporating Dynamic Content Using Ajax and Ruby on Rails 37
  - 4.5.1: Developing Interactive Capabilities with Ruby on Rails 38
  - 4.5.2: Designing a CSS Layout 43
- 4.6: Conclusion 45

Chapter 5: The Formation of the POV-IT Lessons 47
- 5.1: Introduction 47
- 5.2: The Insufficiencies of Available POV-Ray Documentation 47
- 5.3: Writing the POV-IT Lessons 50
- 5.4: Conclusion 52

Chapter 6: The POV-Ray Interactive Tutorial 55

Chapter 7: Conclusions and Future Work 135
CHAPTER 1
Introduction

The Persistence of Vision Ray Tracer, commonly known as POV-Ray, is a computer graphics package that can be used to produce highly sophisticated and photorealistic images and animations. POV-Ray is widely used in the computer graphics community for a diverse variety of applications, including teaching college courses, creating visual music animations, and generating computer art. Although the popularity of POV-Ray is apparent, the teaching materials available for the package are confusing, disorganized, and inaccessible. In this project, I have designed an online system that provides a series of interactive tutorials that teach users how to create images using the POV-Ray package. My system is known as the POV-Ray Interactive Tutorial, or POV-IT for short.

The POV-Ray package is free to download, and the program is therefore accessible to a large variety of people. As a result, POV-Ray enjoys widespread popularity among a diverse community of artists and computer scientists. The artistic applications of the program are many, ranging from the Visual Music animations of Dennis Miller to the still images of Tor Olav Kristensen, Jaime Vives Piqueres and others who have submitted work to the POV-Ray hall of fame (1, 2, 3). Additionally, many computer science departments use POV-Ray to teach computer graphics including those at the California Institute of Technology, the University of Illinois-Springfield, and Smith College (4, 5). POV-Ray images are created using an object-oriented programming language, however this language is simple an intuitive enough to ensure that the program can be utilized by people without extensive programming experience.
While the POV-Ray package is designed to be accessible to a wide variety of users, it can be difficult for beginners to learn about the program. The majority of the official reference material available for the POV-Ray has been written by volunteers in their spare time, and as a result much of the documentation is disorganized and confusing. Additionally, the beginner tutorials provided are heavily text-based, catering specifically to experienced programmers and therefore excluding the large community of artists interested in learning to use POV-Ray. While a number of user-designed tutorials have been developed, these are often inaccurate and none attempt to create an interactive environment that would specifically aid users without much programming experience.

Due to the insufficiencies of the available POV-Ray documentation, I decided to design a new tutorial that would bridge the gap between the accessibility of the program and the confusing nature of its documentation. I wanted to create an online environment that would enable users to learn the POV-Ray programming language both by reading textual lessons and by interactively designing their own images. This type of tutorial would cater to a larger variety of people by emphasizing individual practice through exercises specifically designed to help users learn how to visualize scenes in three dimensions. Additionally, I wanted to design a system that would allow users to practice using POV-Ray without needed to have the package installed on their computer. This would allow potential users to learn about the types of images that can be generated with POV-Ray without needing to go through the hassle of downloading and installing the application. POV-IT was designed to accomplish all of these goals, providing an intuitive online interactive environment where beginning users can learn how to create images using the POV-Ray application.
REFERENCES

1. Miller, Dennis H. dennismiller.neu.edu ©2005

2. Kristensen, Tor Olav. home.no/t-o-k/povray ©2004

3. POV-Ray Hall of Fame. hof.povray.org ©2007


CHAPTER 2
Project Motivation

The POV-Ray Interactive Tutorial, also known as POV-IT, was designed and implemented to provide a concise yet complete introduction to the ray tracing program POV-Ray. It improves on the tutorials and documentation currently available through its unique interactive environment designed to teach POV-Ray enthusiasts of all levels, even those without extensive programming experience. The inspiration for the project was derived from my experience working with professor Jonathan Hallstrom in the music department at Colby, who is interested in using POV-Ray to create visual music animations. He experienced some frustration finding the time to tackle the massive POV-Ray documentation and learn the scene description language, even though he enjoys working with computers and has experience programming. This situation reveals that although the POV-Ray language is user friendly and accessible to artists interested in computer science, the complexity of the documentation makes it difficult for new users to become involved with POV-Ray. The goal of this project was therefore to bridge the gap between the accessible POV-Ray syntax and the complex POV-Ray documentation by providing an interactive tutorial with a focus on concise descriptions and user participation as a means to teach the scene description language.

POV-Ray, which stands for the Persistence of Vision Ray Tracer, is an open source ray tracing program that can be used to create highly sophisticated three-dimensional graphics. A ray tracer is a program that generates a graphical image from the model of a three-dimensional scene by simulating the physical movement of light to
create realistic shadows and reflections. The POV-Ray modeling language, known as the “scene description language,” is used to describe the three-dimensional objects that make up the scene. The process of creating a photo-realistic image with POV-Ray involves first defining the geometry of the scene using the scene description language. After the scene has been designed, the user can “render” this code, in other words generate an image from this scene description, using the POV-Ray ray tracer. The POV-Ray scene description language is intuitive and boasts relatively simple syntax, making it an accessible to a wide variety of users. The rendering process removes much of the work from the user because it accurately models light automatically, one of the many advantages of using POV-Ray.

POV-Ray is an open source program, in other words a program that is available for users to download and use without charge. Like many open source programs, POV-Ray has a large online user community providing programming support and suggestions. The POV-Ray package is maintained by a group of people know as the POV-Team, volunteers who add to the functionality and documentation of the POV-Ray scene description language in their spare time. Due to the highly communal nature of POV-Ray, much of the extensive documentation available is disorganized and difficult to parse. The table of contents alone consumes nine pages, making the task of learning the scene description language appear daunting (1). While the vastness of the POV-Ray community and online documentation is one of the main benefits of using the program, the hassle involved with learning the language may deter some potential users.

The online documentation provided at povray.org contains two different means of obtaining information about the scene description language. One section, entitled
“Introduction to POV-Ray,” provides tutorial-like lessons for beginning users with the goal of presenting an overview of the language’s syntax. A second section entitled “POV-Ray Reference” contains in-depth documentation designed for frequent users. While both sections provide useful examples and explanations, it can be difficult to gain a clear understanding of the functionality of the scene description language without parsing both sections. Useful tips provided in the tutorial are excluded from the reference section, yet the tutorial fails to elaborate on some concepts making it difficult to gain a comprehensive understanding of the language without also reading the reference section. The scattered nature of the POV-Ray documentation makes it unnecessarily difficult for beginning users to learn the scene description language quickly.

Many POV-Ray users have noted the deficiency in the online POV-Ray documentation and have developed beginner tutorials of their own (2, 3, 4, 5). These tutorials often provide clear and intuitive descriptions of some concepts, significantly improving on the documentation provided by the POV-Team, but rarely manage to cover all beginning topics accurately. Additionally, all of the tutorials encountered thus far have been entirely text based; none attempt to create an interactive learning environment placing emphasis on user practice. Due to the highly visual nature of POV-Ray, learning to create images using the scene description language is highly dependent on understanding the visual effects that can be created with different objects, keywords, and techniques. This indicates the need for tutorials and documentation that stress the importance of learning to visualize a three-dimensional scene through individual practice.

The POV-IT project was conceived as a means of providing succinct online documentation for the POV-Ray scene description language that would serve as an
alternative way to inspire users to learn about the ray tracer. The tutorial was designed to create an intuitive interactive learning environment that would appeal to a wide variety of users, even those without extensive programming experience. The central goals of the project revolved around creating an atmosphere that would make the process of learning to write POV-Ray code simple and enjoyable, coupled with precise textual descriptions designed to inspire user participation. In order to create a successful interactive web tutorial, I strived to design an intuitive user interface. The goal was to allow users to practice writing and rendering POV-Ray code online without being distracted by uniqueness of this type of interactive website. I therefore tried to create a web application that would function much like the desktop applications users are comfortable interacting with.

One of the goals of the POV-IT project involved creating a clear flow between lessons so that a user who completed the tutorial from start to finish would never encounter a keyword or technique that had not been previously explained. Additionally, each lesson intended to cover a complete topic, concisely describing concepts but not excluding any information that could prove useful. Due to the interactive nature of the tutorial, I wanted to create textual descriptions of concepts that would provide the information necessary to get the user started, but which would also depend on the participation of the user to some degree. Instead of describing every nuance of a particular keyword, the text suggests that the user experiment with that keyword individually to gain a better understanding of its functionality.

While I encountered a number of obstacles in the process of designing and implementing POV-IT, the resulting web application achieved the majority of my
original goals. POV-IT incorporates a fully functional interactive environment that allows users to write, edit, and render POV-Ray code as part of the tutorial. The textual descriptions provide complete documentation of the topics discussed, and create a comfortable balance between useful descriptions and interactive examples. The nature of the project lends itself to future improvement, however, allowing for the incorporation of additional lessons. The following chapters discuss the process taken to complete the POV-IT project, providing background information, my research techniques, and the textual body of the tutorial.
REFERENCES

1. **POV-Ray User Documentation.**
   http://vergil.chemistry.gatech.edu/resources/povmol/povuser.pdf ©1999

2. **The Online POV-Ray Tutorial.** library.thinkquest.org/3285/ © 1996


4. Shene, C. K.  **An Introduction to POV-Ray.**

5. Lohmueller, Friedrich A.  **Descriptions and Samples for the Ray Tracer POV-Ray.**
   http://www.f-lohmueller.de/pov_tut/pov__eng.htm ©2007
CHAPTER 3
Background

3.1 Introduction

The ray-tracing program POV-Ray is a popular graphics programming environment used by computer scientists and artists alike. As an open source program, POV-Ray is highly dependent on the online documentation and tutorials developed by volunteers (5). The haphazard nature of much of the information available to beginning POV-Ray users motivated the POV-IT project. Before delving into the efforts taken to create this interactive tutorial, however, some background knowledge about ray-tracing, the POV-Ray environment, and the development tools available for interactive websites is necessary. The following chapter will discuss these three topics in some depth, providing the knowledge necessary to understand the work completed for this project.

3.2 Ray-Tracing

Ray-tracing is a rendering technique that has been developed to generate realistic three dimensional images by taking advantage of the modern understanding of the physics of light. Rendering can be thought of as a means of converting a model that describes objects in three dimensions into a two dimensional image depicting these objects. A program such as POV-Ray that can be used to render a three-dimensional model using the ray-tracing technique is known as a ray tracer.

Most ray tracers have a similar structure in that they take a three-dimensional model known as a “scene” as input (1). A scene contains a camera as well as a collection of
light sources and objects, all of which are arranged in a three-dimensional world according to a particular coordinate system. The objects are the physical items depicted in the scene, for example a table, a chair, or a glass of water. The lights in a scene are especially important because they emit the light rays that will be traced by the ray-tracer. They are placed according to the coordinate system just as objects, but are usually invisible. Finally, every scene has one camera that defines the location and orientation from which the scene will be viewed.

The camera in a three-dimensional scene is simply a point in space associated with a vector that determines its facing direction. In front of the camera there is a screen through which the scene will be viewed. This screen represents the final two-dimensional image, and is divided into a grid where each square in the grid represents one pixel in the image (2). See Diagram 1 for further clarification. The first step in the ray-tracing process is also known as ray casting. It is a technique that is used to determine which objects are visible from the perspective of the camera. One ray is sent out from the camera through each pixel in the screen to determine whether the ray intersects with any of the objects in the scene. When an object is intersected, its distance from the screen is saved so that if one ray intersects two objects, the ray-tracer knows which object is in front and therefore which object should be displayed. This is a simple and clean method of determining which objects are visible from the camera’s viewpoint.
Figure 1: Typical camera setup for Ray Casting

The ray-casting method consists solely of shooting rays into a scene to determine which objects are visible from the camera’s location. Ray-tracing incorporates this technique, but expands upon it to not only determine the visible surfaces but also model the movement of light through the scene to calculate the color at each pixel in the final image. When a ray cast through a particular pixel intersects with an object in the scene, the color at that point on the object’s surface is determined by casting a number of secondary rays (3). Shadow rays are cast from the object’s surface to each light source in the scene to discern whether the point is lit or in shadow. If the surface is reflective, additional rays are cast to determine which other objects might contribute to the color of the reflection. Finally, if the surface is transparent, rays are refracted through the object’s surface to discover whether the objects behind this one contribute to the point’s color.
After all of these rays have been sent, the final color of the pixel can be determined. This process is repeated for each pixel in the final image, creating an extremely realistic result.

The rendering time of a ray-traced image can be significantly long, especially as the complexity of the scene increases. The technique is widely used, however, because it can model realistic effects such as shadow, reflection, and transparency that can be difficult or impossible to imitate using other methods (1). As the speed of computers increases and as new methods are developed to reduce the cost of ray-tracing, the technique is being applied to much larger projects. In 2004, for example Pixar incorporated ray-tracing into their RenderMan software package for use in the production of the movie Cars (4). As the popularity of ray-tracing as a rendering technique increases, the need for reliable and usable ray-tracers becomes more apparent.

3.3 The POV-Ray Package

POV-Ray stands for “The Persistence of Vision Ray Tracer.” It is an open source ray-tracing program that began as an extension of DKBTrace, a ray-tracer developed by David K. Buck and Aaron A. Collins in the 1980s (6). In 1989 with the release of DKBTrace 2.12, Buck and Collins were feeling the pressure of their increasingly popular ray-tracer as their users expressed desires for more and more new features. To remedy the increasing demand for a fully capable ray-tracing program, Buck offered the DKBTrace system as the base for the POV-Ray project (6). A group of volunteers known as the POV-Team, mostly previous users of the DKBTrace system including both Buck and Collins, combined forces to add the desired features to the program and update its documentation. While both Buck and Collins left the project shortly after its creation
to pursue other interests, the POV-Team continues to expand and modify POV-Ray’s capabilities to this day (6).

The Persistence of Vision Ray-Tracer reads in a text file written in a language known as the POV-Ray scene description language (7). The scene description language provides all of the functionality necessary to describe a three-dimensional scene including objects, lights, and a camera. Additionally, the POV-Ray package provides a number of include files supplying predefined colors, textures, pigments, shapes, and functions. As POV-Ray’s capabilities has grown since the early 90s, many advanced features such as lathe objects, isosurfaces, and radiosity have also been incorporated into its syntax. POV-Ray therefore provides both advanced modeling and lighting capabilities, making it an extremely useful program.

While there are a number of other software packages available that provide ray-tracing capabilities, POV-Ray was chosen as the focus of this project due to its widespread popularity and its user-friendly syntax. Competitor programs such as Radiance and Indigo, both open-source ray-tracers that are free to download, do not boast the massive system of online forums and documentation enjoyed by POV-Ray. More importantly, neither provides a user interface to facilitate scene creation, relying instead on command-line options than can be confusing to some users (13, 14). Indigo differs from POV-Ray in that it serves solely as a ray-tracer, rendering three-dimensional models created using other programs without providing any of its own modeling capabilities (14). Radiance is similar to POV-Ray in that it provides both modeling and rendering options, but its low-level syntax makes it difficult to learn and use. Consider the following example, which displays the code necessary to describe a red box object in Radiance and POV-Ray:
According to the Radiance syntax, a user must define the color of an object in a separate materials statement, which is tedious when compared to the simplicity and readability of POV-Ray’s predefined colors. A box is defined in Radiance by calling the system command `genbox`, signified by the exclamation point, which only allows the user to specify one of the box’s corners (5,8,3) and requires that the other lie at the origin. In the POV-Ray scene description language, the user provides two arbitrary points inside of a `box` block to create a box object. Even in this simple case, it is clear that the POV-Ray syntax is much clearer than the Radiance syntax. For this reason, and due to its large online community, POV-Ray is more accessible to artists and those without extensive programming experience than its ray-tracing competitors.

On the other hand, POV-Ray provides lower-level control than most of the GUI-based modeling programs. Blender, for example, is an extensive open-source modeling program that may be a more attractive option for many artists (15). Its complex GUI environment provides many options to users, but it could have just as large of a learning curve as POV-Ray’s simple scene description language while limiting some aspects of the user’s control over her work. Blender focuses on mesh modeling, which POV-Ray avoids in favor of geometric modeling due to the need to calculate ray-surface
intersections so frequently. While mesh modeling can be used to create complex models, Blender favors an interactive interface that can sometimes be difficult to control. Since POV-Ray allows users to directly specify the coordinate locations of their objects, they can more precisely dictate their appearance. For this reason, POV-Ray serves as an ideal middle ground between computer science and art, providing both a simple syntax and a sizeable amount of control.

The popularity of POV-Ray is evidenced by the large number of registered users. The POV-Ray community map available at povray.org/community/map/ provides listings of all registered users in different areas of the world, including 1,054 users within the United States (16). The actual number of POV-Ray users is most likely much larger, since registration is only necessary for posting questions and comments on the community forums. The artistic applications of POV-Ray are widespread, ranging from the Visual Music animations of Dennis Miller to the still images of Tor Olav, Jaime Vives Piqueres and others who have submitted work to the POV-Ray hall of fame (17, 18, 19). Many computer science departments use POV-Ray to teach computer graphics, including those at the California Institute of Technology, University of Illinois-Springfield, and Smith College (20, 21). While POV-Ray is widely used, the documentation available for the application is confusingly disorganized. The official website povray.org provides extensive resources, but because they have been produced gradually over a number of years by volunteers, the resulting documents can difficult to parse for useful information. For this reason, the development of a clear and concise tutorial providing interactive options will be a useful addition to the currently available documentation.
3.4 Interactive Web Design

Since the explosion of the Internet during the mid-90s, the quality of web development has changed dramatically. While websites used to serve as static means of displaying information, in recent years large complicated web applications have become the standard and interactivity is increasingly desirable. Despite these facts, the interactivity of websites has always lagged far behind that available through desktop software (8). With the introduction of new interactive web development tools such as Ajax, however, this gap has begun to close. Google first popularized Ajax with its real-time interactive sites such as Google Suggest, which offers website suggestions as users type letters into a search bar, and Google Maps, which allows users to scroll through maps without reloading the page (8). The new tools and technologies used to provide this interactivity in websites have spread like wildfire, significantly complicating the job of web developers but also increasing the design possibilities and creating a more comfortable environment for general users.

In the following sections, I will discuss the web development tools that facilitated the creation of POV-IT’s interactive functionality, providing the background necessary to understand how each tool functions in relation to this project. First, I will provide an overview of the dynamic functionality that can be created using Perl CGIs, which were utilized in the initial implementation of POV-IT. Then I will discuss the tools that were used for the final implementation of the system, including the popular web development framework Ruby on Rails, which enabled the creation of the system’s dynamic content, and Cascading Style Sheets (CSS), which were used to design the tutorial’s layout. All of these tools will be discussed in further detail in later chapters, and therefore these sections will simply provide the background necessary to understand later discussions.
3.4.1 Perl CGI

CGI is an acronym that stands for “Common Gateway Interface.” Websites that incorporate CGI programs utilize an important relationship between the client computer where a user views a website and the server computer that runs the website and processes requests. A CGI program runs on the server-side, meaning that it does not need to depend on the software and hardware of the client’s computer to function. It receives information passed to it from the client, processes this data, and returns a response back to the client. For example, in many online shopping websites users are asked to enter personal information into a form. When the user clicks the “submit” button, a request is sent to the server-side CGI program asking that it verify the entered information. If the information was entered correctly, the server will return a positive reply, signifying to the client website that it can proceed with the sale.

A CGI program can be written in a number of different programming languages, but Perl is the most popular language used for the task (22). Perl contains a CGI package that provides the functionality necessary to implement CGI programs in the language, including commands that signify the return type of the message that will be sent to the client computer. A Perl CGI can only return data in one particular format, which significantly limits the level of interactivity that can be created using this method. In most cases, a CGI sends HTML code back to the client website after processing a request, signifying that the page should reload and display the code returned by CGI script. This allows appearance of a website to change dynamically based on the data sent to the CGI by the client, but it also requires that the page reload every time a request is sent. Therefore, while Perl CGIs can be utilized to create an interactive web
environment, certain aspects of this technology limit the dynamic effects that can be created.

### 3.4.2 Ruby on Rails and Ajax

The Ruby on Rails framework is central to providing POV-IT with its interactive capabilities because it is specifically designed to facilitate the creation of dynamic web content. Ruby is an object-oriented programming language with clean syntax that is efficient and easy to use (9). Rails is an open source Ruby framework that can be used to design database-driven web applications. While many similar frameworks already exist, Rails has become immensely popular due to its ability to significantly speed up web development. In their popular article *Rolling with Ruby on Rails Revisited*, Bill Walton and Curt Hibbs state that “you can develop a web application at least ten times faster with Rails than you can with a typical Java framework” (9). Rails was designed as a means to significantly reduce amount of code needed to write an application. A few lines of configuration are all that is needed to set up a database-driven web application, since Rails creates most of the low-level database files automatically (11). While this may be the draw for many web developers, the appeal of Ruby on Rails for this project is the fact that it provides tools to include Ajax capabilities in an interactive website with comparatively little effort.

Ajax, which stands for Asynchronous JavaScript and XML, is a new combination of existing technologies. It combines asynchronous data retrieval using the XMLHttpRequest request object with XML data management, which are tied together using underlying JavaScript to provide more robust interactivity in web applications (8). The main problem with web applications that require server-side activity, such as form submissions or database lookups, is that in order to retrieve data from the server the user must wait for the request to complete and the page to reload before continuing to interact with the website (10).
the other hand, allows web applications to make requests to the server through small JavaScript calls, permitting the user to continue interacting with the website while awaiting a server response and allowing the web application to refresh data dynamically without reloading the page (10).

The capabilities provided by Ajax are highly desirable because they allow for a much higher level of interactivity in web applications. Programming an Ajax-enabled website from scratch, however, can be difficult and time consuming. The advantage of using Ruby on Rails is that allows users to incorporate Ajax in their applications without the need to write complicated client-side JavaScript. David Heinemeier Hansson, who designed and released the Ruby on Rails framework in 2004, stated that “instead of trying to soften the blow of doing client-side JavaScript libraries as many others were doing, we’ve gone ahead and more or less removed the need for hand-written client-side JavaScript entirely” (11). This allows users to incorporate complex Ajax interactivity into their web applications quickly without needing to write much code, making it an attractive option.

3.4.3 Cascading Style Sheets

Styles were added to HTML 4.0 by the World Wide Web Consortium to solve a problem that had developed as Netscape and Internet Explorer, the two major web browsers at the time, began adding many new tags and attributes to the original HTML specification (12). Originally, HTML was designed to describe the content of a web document, leaving its layout and formatting up to the web browser (12). However, as Netscape and Internet Explorer added new tags such as “<font>” and attributes such as “color” to give web developers more control over the appearance of their webpages, it became increasingly difficult to separate the content of a document from its presentation. For this reason, styles were added to HTML 4.0 to give web developers the layout control they desired while maintaining this separation (12).
Cascading Style Sheets are particularly advantageous because they allow web developers to specify the layout and style of multiple web pages simultaneously. Styles are usually specified in an external CSS file that can be uploaded and applied to as many web pages as desired. Developers are therefore able to modify the layout of an entire web by updating a single file. In order to make styles more flexible, however, there are a number of ways to define the style of a particular element. Styles can be specified in an external style sheet, an internal style sheet included in a particular HTML file, or an inline style defined within a particular HTML tag (12). This myriad of styles cascade into a new virtual style according to the following hierarchy, listed from lowest to highest priority: browser default; external style sheet; internal style sheet; inline style (12). The use of CSS significantly improves the web developer’s control over the layout and appearance of her web application, making it an essential tool for modern sites.

### 3.5 Conclusion

These modern web-development tools have served to make POV-IT significantly more interactive and user friendly. While CSS has been used mainly to increase the beauty and usability of the site, Ruby on Rails has provided a level of interactivity that otherwise would not have been available. One of the central goals of the POV-IT project has been to provide users with an interactive area where they can write and render POV-Ray code as they learn it. In order to achieve this goal, I have used Ruby on Rails to incorporate Ajax into POV-IT so that users can render code and view the newly created POV-Ray image without a page reload. This allows for intuitive interaction and easy navigation from one page to another, which would have been difficult or impossible without Ajax. For these reasons, CSS and Ruby on Rails have been crucial in the development of POV-IT.
REFERENCES


5. POVRay 3.6 Documentation: 1.5.3.1 Contacting the Authors http://www.povray.org/documentation/view/3.6.1/205/ © 2007


12. Introduction to CSS. http://www.w3schools.com/css/css_intro.asp


19. POVRay Hall of Fame. hof.povray.org/.
20. **CSC240 Syllabus.** maven.smith.edu/~orourke/240/Welcome.html


CHAPTER 4
The Implementation of POV-IT

4.1 Introduction

The implementation of the POV-IT system underwent a number of changes over the course of the semester as I learned more about the technologies available for creating interactive web applications. While the conceptual system design remained constant throughout the project, I developed two independent implementations of POV-IT, the first relying on a Perl CGI to provide the server-side capabilities and the second instead utilizing the Ruby on Rails framework for this purpose. The process that led me to redesign the implementation of the tutorial displays recent advances in development tools available that facilitate interactive web development, through which the awkward functionality of CGIs has been replaced by the sleek dynamic capabilities of techniques such as Ajax. In this chapter, I will discuss how I developed and implemented the POV-IT system, focusing on providing an overview of the process that led to the final result.

4.2 POV-IT System Design

Before beginning to implement POV-IT, I developed a design for the layout of the tutorial to help me visualize how its interactivity would function from the user’s perspective. I designed an interface that divided the main window of each lesson into two halves, a scrollable left-hand section that would provide a textual description of concepts and syntax, and a static right-hand section that would contain a simple GUI through which users could write and render POV-Ray code interactively. The interactive
section would display a POV-Ray image with an editable textbox beneath it containing
the code used to define that image. The user would be able to modify the code in the
textbox to practice new concepts and syntax, and the click a “Render” button to view the
resulting image. Each time the user rendered her code, the previously displayed POV-
Ray image would be replaced dynamically with the new image, creating a clean and
intuitive interactive flow.

After developing a conceptual understanding of how a user would interact with
POV-IT, I created a high-level system design describing the relationship between the
client and the server. I decided that the client-side website would serve solely as an
interface providing the tutorial’s interactive environment, while the server would be
responsible for all of the work involved in rendering the user-supplied code and
generating the appropriate POV-Ray image. This system design is appealing because it
does not require that the user have POV-Ray installed on his computer, instead asking
that the server handle the work of rendering POV-Ray code. Since one of the goals of
POV-IT is to inspire potential users to learn POV-Ray, providing an environment where
these users can experiment with the program without needing to install it on their
computers is beneficial.

I therefore set out to design a system that would allow users to practice writing
POV-Ray code using the interactive interface provided by the tutorial, and would then
submit this code to a server-side script that would render the code and return the resulting
image. The following system diagram displays the flow of information that would be
implemented for POV-IT.
This task of implementing this system required knowledge of server administration and dynamic web development, both of which I strived to learn over the course of the semester. Although I ended up developing two independent implementations of the system, my high-level understanding of the system remained relatively constant. The central goal as I implemented POV-IT was to create a simple and intuitive interactive environment that would be supported by a dynamic user interface and a server-side script.

4.3 Writing a Perl CGI for an Apache Web Server

When I first developed the concept of the POV-Ray Interactive Tutorial, I was not certain that it would be possible to implement the interactive environment I had imagined. My initial goal, before beginning to write lessons, was therefore to determine whether I would be able to create a server-side program that would render client POV-Ray code and return the resulting image. As I began to investigate the implementation of
interactive websites, I learned that Perl CGIs are one of the main technologies used to
generate interactivity online. I decided to try using a Perl CGI to create an online POV-
Ray interpreter that would enable users to write code and view the rendered result within
the confines of the tutorial’s main window.

The following sections detail the process of developing a CGI-enabled web
application. This includes a discussion of the steps needed to configure an Apache server
to run CGI programs, as well as an explanation of the Perl code used to create the desired
dynamic functionality for POV-IT. These descriptions provide a significant level of
detail, however this detail is important because it displays the complexity of the process
of developing interactive content in web pages. Finally, I discuss the user interface that I
developed for this initial implementation of POV-IT and the ways in which it was limited
by the nature of CGI applications.

4.3.1 Configuring Apache

Before I could begin writing a Perl CGI that would accomplish the desired task, I
needed to set up a web server on my personal computer that would run CGIs. I
developed POV-IT on a Mac laptop, and was therefore able to utilize the built-in Apache
web server already installed on my computer. To enable web, I simply needed to check
Personal Web Sharing under the Sharing menu of System Preferences. While this
modification started my server correctly, I soon discovered that the pre-installed Apache
server is initialized such that CGI programs are disabled.
After some investigation, I learned that the Apache configuration file must be modified in order to enable CGI programs. Following the detailed instructions provided by in the book *CGI Programming 101* by Jacqueline Hamilton, I was able to modify the immense httpd.conf file to enable CGI programming (1). This process involved two steps. First, I scrolled through the httpd.conf file until I found the following segment of commented out code:

```plaintext
#<Directory /home/*/Sites>
#  AllowOverride FileInfo AuthConfig Limit
#  Options MultiViews Indexes SymLinksIfOwnerMatch IncludesNoExec
#  <Limit GET POST OPTIONS PROPFIND>
#     Order allow,deny
#     Allow from all
#  </Limit>
#  <LimitExcept GET POST OPTIONS PROPFIND>
#     Order deny,allow
#     Deny from all
#  </LimitExcept>
#</Directory>
```

I uncommented this entire section by removing the pound symbols from the beginning of each line. Following Hamilton’s instructions, I then changed the `Options` line to the following:

```
Options MultiViews Indexes SymLinksIfOwnerMatch Includes ExecCGI
```

The `Options` line specifies the options that are available to the `/home/*/Sites` directory. Specifying the option `Indexes` permits server-side includes, and specifying the `ExecCGI`
option enables CGI programs to run in this directory (2). After the options line, I added the following to the Directory block:

\[
\text{DirectoryIndex index.html index.cgi}
\]

This completed the first set of modifications I made to the httpd.conf file. Next, I scrolled down to the AddHandler section and uncommented the following line by removing the pound symbol:

\[
# \text{AddHandler cgi-script .cgi}
\]

This statement indicates that all files with the extension .cgi should be processed as CGI programs (2). Finally, I uncommented the following line:

\[
# \text{AddHandler server-parsed .shtml}
\]

I also changed the extension from .shtml to .html, so that the resulting line looked like this:

\[
\text{AddHandler server-parsed .html}
\]

This statement signifies that all html files should be searched for server-side include tags (2).

After completing these modifications, I saved the httpd.conf file and restarted Apache by first disabling Personal Web Sharing under Sharing in System Preferences,
and then enabling it again. My website appeared as expected on the server at http://localhost/, displaying the index.html file stored in /Library/WebServer/Documents.

4.3.2 Writing a Perl CGI

After setting up my Apache web server to run CGIs, I began learning how to write a CGI program in Perl that would run in conjunction with my website. Hamilton’s book CGI Programming 101 contains a broad introduction to the Perl programming language as well as the functions that can be used to add dynamic content to web applications, so I began by reading her introductory chapters. I ran into my first challenge after creating the simple “Hello World!” CGI provided as an example in the book. I placed my CGI code into the folder /Library/WebServer/CGI-Executables as instructed and then attempted to view the result by accessing /localhost/cgi-bin/helloworld.cgi. Instead of seeing the expected “Hello World!” message, I received an error stating “You don’t have permission to access /cgi-bin/helloworld.cgi on this server.” I was surprised at this result since it appeared that I had followed Hamilton’s instructions precisely, but after some investigation I learned that the permissions of a CGI program must be set explicitly using the chmod Unix command so that the file is “executable” (3). To accomplish this, I opened the folder /Library/WebServer/CGI-Executables in the terminal and typed the command chmod 755 helloworld.cgi to give the file the correct permissions. After this modification, the CGI presented the correct “Hello World!” message when accessed through /localhost/cgi-bin/helloworld.cgi.

Once I had successfully run a simple CGI program on my server, I set out to learn how to create files, call system commands, and upload image files using Perl so that I
could obtain the desired functionality from my server-side script. As mentioned above, I wanted to implement the interactive section of the tutorial so that the user could enter
POV-Ray code into a textbox and then click a “Render” button to see the resulting image. I decided that the simplest way to obtain this functionality would be to create an HTML form that would submit the user’s code to the server-side Perl script when the “Render” button was clicked. Perl CGIs are often used to create dynamic HTML forms, therefore descriptions of the code needed to link an HTML form to a Perl CGI were readily available online.

I began by creating a new script `getCode.cgi`. Initially, I simply copied the code for the “Hello World” script I had already tested into this CGI. This way, I would be able to focus on linking my HTML form with the server-side script without worrying about potential errors in the script. I designed a simple HTML homepage that contained a form with textarea and a submit button. I set the form’s action parameter to reference the `getCode.cgi` script using the POST method so that the code in the textarea would be passed to the CGI when the user clicked the button. The resulting HTML code was as follows:

```html
<form action="cgi-bin/getCode.cgi" method="POST">
   <p>Edit the code below and click:
   <input type="submit" value="Render"></p>
   <textarea  name="code" wrap="logical" rows="30" cols="45">
   </textarea>
</form>
```

I tested my simple website by clicking the “Render” button. This action submitted a request to the `getCode.cgi` script, which returned the HTML code needed to display the
message “Hello World.” As I expected, the page reloaded and displayed the correct message.

Once I had successfully linked my HTML form to this simple CGI, I began to develop new content for the getCode.cgi script. First, I used the Perl open() function to create an empty POV-Ray file. In order to write the user-supplied code to this new file, I used the param() method to grab the text from the form’s textarea and then used the print command to place the code in the file. After successfully creating the POV-Ray file, I rendered its contents using the Perl system() function to call the povray system command, which generating the desired POV-Ray image.

Implementing this part of the Perl CGI was not particularly difficult due to the extensive function documentation available at perldoc.perl.org and the clear explanations of file creation available in Hamilton’s CGI book (4). I ran into some difficulty, however, when I attempted to format the script’s output so that it would display the newly rendered image. As mentioned in Chapter 2, Perl CGIs must return a response to the client website in one particular format. Up to this point, I had been returning the HTML code that would be displayed when the page reloaded after each render request. This approach would no longer function, however, because the povray system command called by the CGI saved the output POV-Ray image in the CGI-Executables folder, which could not be accessed by the webpage. In order to display the POV-Ray image that had been rendered, the CGI would need to return an image file to the client as well as the HTML code referencing it.

I had a difficult time finding an example of a web application that uses a Perl CGI to load an image file from the server to a client-side website. After some investigation, I realized that for the most part CGIs are used to respond to the textual content entered into
HTML forms instead of for creating graphically dynamic content in web applications. Eventually, however, I encountered an example describing how to display binary image data from a Perl CGI (5). Using the code supplied by this article, I was able to write a separate CGI entitled displayImage.cgi that returned an image file as its output instead of HTML code. Since I could only return one type of output to the client following a render request, I had to reference this second CGI from within the HTML code returned by the original getCode.cgi program. The HTML code describing the new POV-Ray image to be displayed appeared as follows:

```html
<img src="imageDisplay.cgi" alt="POV-Ray Image">
```

This statement allowed me to successfully display the newly rendered POV-Ray image in the HTML page returned to the client website by inciting a call to a new CGI in charge of displaying the image correctly. This discovery completed my original implementation of the POV-IT system.

4.3.3 Creating a User Interface

After I had developed a functional implementation of the interactive section of the tutorial, I began implementing the user interface I had envisioned. This proved challenging because every time the Perl CGI received a render request, the interactive portion of the tutorial needed to reload to display the resulting POV-Ray image. Additionally, I wanted to divide the main window of each lesson into two sections, with the textual explanations on the left and the interactive interface on the right. After some
investigation, I learned that HTML frames can be used to divide the content of a webpage into separate sections. The idea of implementing the tutorial layout using frames was immediately appealing because they separate the content of the main into different HTML pages. Since content of each child frame within the main window is defined by a reference to an HTML file, the tutorial would be able to reload the interactive frame after each render request without needing to affect the textual frame. Consider the following HTML code used to define the body of the main page for the first tutorial lesson:

```html
<frameset cols="55%, 45%">
    <frame src="text1.html"/>
    <frame src="interactivel.html"/>
</frameset>
```

This code defines a frameset, indicating that a frame layout will be used. The cols option determines that the layout will contain two columns, a left-hand one that will consume 55% of the page’s total width, and a right-hand one that will consume 45% of its width. The frame tags within the frameset block indicate the HTML files that should be loaded in each column. Since the content displayed in each frame functions as an independent webpage, the Perl CGI could reload the interactive column without affecting the rest of the page, which was a highly beneficial feature of this implementation.

While this layout successfully arranged the tutorial pages as I had imagined and provided the functionality needed for the interactive sections, there were a number of problems with the design that eventually led me to change both the interface and the interactive implementation of POV-IT.
4.4 Problems Encountered

While pleased that I was able to successfully implement the interactive tutorial I had envisioned, I encountered number of problems with both the CGI implementation and the tutorial layout as I continued to develop the POV-IT system. Once I had written a handful of lessons, I realized that it would be nearly impossible to link them together due to the tutorial’s frame-based HTML layout. If I included a link to the next lesson at the bottom of a lesson’s textual description, only the left half of the window would load the new page. For the same reason, I could not include links to other tutorials or reference pages that might offer useful information to the readers. This was a frustrating side effect of my implementation signifying that perhaps I had not developed a particularly functional layout.

Additionally, while the Perl CGI implementation operated correctly and therefore achieved my initial goal, it was not as elegant a solution as I would have liked. I found it inefficient for my Perl script to return essentially unchanged HTML code to the client to be reloaded. The only aspect of the interactive section that needed to change dynamically was the POV-Ray image, and it therefore seemed odd to reload the HTML code unnecessarily. Ideally, I would have liked to come up with a design that would not require the entire interactive page to reload, instead simply substituting the newly rendered POV-Ray image for the old one.

Furthermore, while the HTML frame layout I had designed adequately separated the lesson pages into two sections, it was very ugly. Each frame was equipped with its own scrollbar, making the page look disorganized and overly crowded. It was also impossible to create a header or footer that would span the entire page, which gave the tutorial lessons a very disjointed feeling. As I continued to read about frames, I learned
that they are an deprecated option maintained in HTML for backwards compatibility, but that for the most part modern web applications use CSS layouts to arrange the content of their pages instead of the ugly and unmanageable HTML frames.

For all of these reasons, I began looking for alternative tools that I could use to more effectively implement the POV-IT system. My initial goal was to find a way to dynamically update the POV-Ray image displayed in the interactive section of the tutorial without needing to reload the entire page, yet my investigation eventually led to the reimplementation of the entire POV-IT system.
4.5 Incorporating Dynamic Content Using Ajax and Ruby on Rails

As I began to research alternate methods of implementing interactive content in websites, I learned from fellow computer science major Mike Wolk that a tool known as Ajax can be used to create dynamic effects online. At the time, Mike was using Ajax to create dynamic functionality for a web application he was developing with the Ruby on Rails. While his understanding of Ajax was entirely dependent on its implementation within the Rails framework, which he was using as means of managing the database for his system, he suggested that I investigate the possibility of extending my current implementation using Ajax. Since Ajax is a tool specifically designed for creating dynamic content in websites while eliminating the need for page reloads, I decided this would be a good place to start.

I spent a number of days researching Ajax and trying to determine how I could incorporate it into my already functional website. I encountered countless tutorials, articles, and examples online that indicated the widespread popularity of the tool, yet to some degree this plethora of information only served to make the process of learning Ajax more difficult. Much like the POV-Ray documentation, the resources available for learning Ajax were disorganized and often incorrect. I experienced a great deal of frustration while trying to learn the tool, mainly because I was trying to accomplish a task that was more complicated than most of the examples discussed. While I found endless articles describing the JavaScript code needed to create dynamic text and forms, I could not find any documentation explaining how to use Ajax to display the information returned by a Perl CGI in a more dynamic manner.

After a significant amount of research and frustration, I decided to try to redevelop the POV-IT system using Ruby on Rails instead of sinking deeper into the
complex JavaScript functions needed to create Ajax effects from scratch. The Rails framework was designed specifically to reduce the amount of code needed to develop a complex dynamic web application, and Mike indicated that he had no trouble implementing Ajax effects within the Rails environment. I therefore set aside my Perl implementation and began developing with Ruby on Rails.

4.5.1 Developing Interactive Capabilities with Ruby on Rails

My initial goal as I began to work with Ruby on Rails was simply to determine whether I would be able to create the dynamic functionality I desired quickly and efficiently. I therefore set out to create a small application that would simply replace one image with another at the press of a button. Mike advised me to develop my application using the open source software Locomotive, a program designed to reduce the hassle of configuring a server to run Ruby on Rails projects on Mac OS X during the development and testing process. Locomotive runs its own local server already linked to run Ruby on Rails applications, so I was able to start implementing my website immediately after installing the software.

The Ruby on Rails framework utilizes the Model-View-Controller architectural pattern to organize the content of web applications. In this system, the Model is used to organize and maintain the system’s data, while the View describes how information will be visually displayed. The Controller is used to process events such as requests for information. Since the interactivity of the POV-IT system is not dependent on information stored in a database, I did not need to define a Model for my system. The View for my system was essentially an HTML file describing the visual contents of the
Ruby code was incorporated into the View, however, to call functions defined in the Controller and to control the dynamic and interactive aspects of the page. The Controller was a Ruby file containing the definitions of the functions that were used to respond to events such as button clicks. The Controller can be thought of as a server-side program with a purpose similar to that of the Perl CGIs in the previous implementation.

To implement my simple Ruby on Rails project, I defined a View containing an HTML form similar to that described in the previous implementation. In this case, I wanted to make an Ajax request when the “Render” button was clicked instead of a typical request to a server-side script. To do this, I utilized the Rails helper methods designed for that purpose. The Ruby code I used to define the HTML form is as follows:

```
<% form_remote_tag(:url => {:action => :render_input}) do %>
  <%= text_area_tag(:input, @params[:input], :size=>"50x15") %>
  <br>
  <%= submit_tag "Render" %>
<% end %>
```

Here, the Rails helper function `form_remote_tag` is used to define the HTML form. This form is then linked to the function `render_input` that appears in the page’s associated Controller. This provides similar functionality as when the form tag’s `action` parameter was linked to the `getCode.cgi` program in the previous implementation, but this form generates an Ajax request instead of a regular server-side request. The `text_area_tag` helper is used to describe the text area where the user will write and edit POV-Ray code, and the `submit_tag` defines the form’s “Render” submit button. While this is a seemingly complicated way to describe an HTML form, utilizing the Rails helper functions to create an Ajax request significantly decreases the amount of work for the
programmer because the requests are generated automatically without the need for complex JavaScript calls.

After designing the View so that the form would operate as I desired, I began to write the `render_input` Ruby function in the Controller that would implement the website’s dynamic effects. This proved a challenging task, but was still significantly less frustrating than trying to learn to implement similar functionality outside of the Ruby on Rails framework. As mentioned above, I began by trying to write a simple script that would replace the originally displayed image with a new one when the form’s “Render” button was pressed. This removed most of the complex functionality of the script, such as creating a POV-Ray file and running the `povray` system command, which allowed me to focus on reproducing the dynamic functionality I envisioned. The Ruby script was therefore solely responsible for recognizing that an Ajax request had been made and responding accordingly.

When a function in the Controller is called, it usually executes a few instructions and then reloads the main page to display its associated View, which describes how the newly processed information should be visualized. When an Ajax request is made, however, the function passes the control to its associated Ruby JavaScript (RJS) template instead of loading a new View. The RJS template contains calls to JavaScript functions that define how the content of the webpage should change dynamically, and is therefore at the heart of Ajax. The RJS template I wrote contained a call to the `replace_html` function, which substitutes part of the HTML code in the original webpage with a new block of code. This function was ideal for my system because it would allow me to replace the original HTML `img` definition with a `img` tag referencing a new image. In order to reference the section of code I wished to replace with the `replace_html` method,
I used a Ruby on Rails tool referred to as a partial. A partial is simply a block of HTML code that is placed in a separate file from the rest of the View. For example, the POV-Ray image originally displayed in the interactive section is defined using a partial called render_input that contains the following code:

```html
<div id="pov_image">
  <img src="images/povray.png" id="image" alt="POV-Ray Image">
</div>
```

This partial is displayed within the View by inserting the following Ruby statement:

```ruby
<%= render :partial => 'render_input' %>
```

When the HTML code is processed, this Ruby statement is replaced by the HTML code that appears in the partial file. This system of organization is useful because it allows for the programmer to easily reference the block of code that she wishes to replace dynamically using Ajax. I was able to write a very simple RJS template to describe the dynamic changes I wished to make after each render request, containing only the following code:

```ruby
page.replace_html("pov_image", :partial => "new_partial")
```

This statement indicates that the div with the id pov_image should be replaced by the partial new_partial. The new_partial file contains another div definition identical to that in the render_input partial except that the img tag references the new image that will dynamically replace the previous one described in the render_input partial.
While implementing this simple Ruby on Rails program required some time and effort, I was able to get the website up and running successfully within one afternoon. After I had accomplished my initial goal of dynamically modifying the content of the webpage without reloading the page, I worked on extending the render_input function so that it would generate a POV-Ray file from the code entered into the text area and render that file using the povray system command. The text entered into the HTML textarea was passed to the render_input function as a parameter, so the script was able to grab this code using the Ruby param() method. The code was printed to a POV-Ray file that had been created using the File.new() function, and then rendered using the Ruby system() function. These modifications added the desired functionality to the already dynamic server-side script.

Once I had created the desired dynamic functionality for the interactive section of the tutorial within the Ruby on Rails framework, I moved all of the lessons I had already developed to my Rails project and left the Perl CGI implementation behind forever. While it took a significant amount of work to generate an entirely new implementation for the POV-IT system, my efforts produced a more intuitive interactive environment that was significantly easier to use.

### 4.5.2 Designing a CSS Layout

Once I had achieved my goal of implementing the interactive functionality of POV-IT so that the newly rendered POV-Ray image would appear dynamically without requiring the page to reload, I was able to begin designing a new interface for the
website. Since I no longer needed to accommodate the interactive section’s frequent reloading, I could eliminate my previous use of the ugly, confusing, and outdated HTML frames. Instead, I decided to use Cascading Style Sheets (CSS) to design a layout for POV-IT. CSS is a widely popular web development technology utilized by all professional web developers, and due to its frequent use it is well documented and easy to learn.

In order to speed up the process of developing a look for my tutorial, I downloaded a few free layouts from CSS Tinderbox and modified them to create the design I wanted (6). I wanted to develop a layout that would retain the two-panel organization of my previous implementation, with the tutorial’s textual descriptions in the left half of the window and the Ajax-enabled interactive section in the right half of the window. To accomplish this using CSS, I created one div style for each section. The interactive div was called rightcontent and had a fixed position 55% from the left of the window. It had the following CSS definition:

```css
#rightcontent {
  position:fixed;
  left:55%;
  padding:10px 10px 10px 10px;
  background-color:#cccccc;
}
```

The left-hand textual panel had an associated style called leftcontent, which gave the section an absolute position with an automatic overflow, in other words automatic scrolling. It has the following CSS definition:

```css
#leftcontent {
  position:absolute;
}
Using CSS styles significantly improved the appearance of POV-IT. The lessons have a much more intuitive flow than they did when I used HTML frames to divide the content because each lesson page only has one scrollbar that controls the left-hand textual section. The right-hand section remains static as the left side of the window scrolls, ensuring that the interactive capabilities of each lesson will be readily available for the user to practice writing code as learns new concepts. Additionally, I was able to create a footer that spanned the entire window providing links to the next lesson, previous lesson, and table of contents. These modifications created a much more professional-looking final product.

4.6 Conclusion

The process I underwent to implement POV-IT displays some important developments that have been made in the technologies available for interactive and dynamic web design. While tools such as Perl CGI scripts and HTML layout options
were used widely in the past, these have recently been replaced by new technologies such as CSS, Ruby on Rails, and Ajax. While I was able to implement the POV-IT system using older technologies, the final functionality and beauty of the application was gained by using newer tools that were able to provide me with the intuitive desktop-like environment I had envisioned for the tutorial. While the process of learning both methods of implementation taught me about the history of web development and made me fully appreciate the advantages of newer tools, future developers should learn from this process and avoid the hassle involved with using restrictive implementations such as Perl CGIs.
REFERENCES


CHAPTER 5
The Formation of the POV-IT Lessons

5.1 Introduction

The process of composing, organizing, and writing the textual descriptions for POV-IT’s nine interactive lessons required an extensive amount of research. Before beginning to develop the POV-Ray Interactive Tutorial, I had only used the ray tracer for a few specific projects and had not yet developed an in-depth understanding of even the beginning concepts. As a result, I spent a long period of time at the beginning of the semester reading the POV-Ray online documentation available at povray.org and perusing the user-made tutorials available for the language. This process helped me gain a better understanding of the POV-Ray syntax and the visual effects that can be created with the language, yet it also helped me formulate an idea of the material that should be covered by a beginning POV-Ray tutorial. While I learned a great deal from the resources available, I also noticed a number of problems and deficiencies in the way that material was presented to the reader that I hoped to improve upon with the formation of the POV-IT.

5.2 The Insufficiencies of Available POV-Ray Documentation

In my research of POV-Ray I encountered a number of websites providing documentation of the scene description language (1, 2, 3, 4, 5, 6). I spent the majority of my time reading those that proved most useful as I myself attempted to learn the language, and these are therefore the resources that I analyzed as a means of determining
the deficiencies of the current POV-Ray documentation and the steps I could take to improve upon these tutorials as I wrote the POV-IT lessons. While I gained a relatively clear idea of the POV-Ray syntax from these tutorials and learned a number of useful programming tips, I had a difficult time gaining a complete understanding of the beginning concepts from any one resource. I therefore found myself reading multiple descriptions of the same topic before trusting that I had fully understood it. The process that I underwent to learn the scene description language signifies that the available documentation does not provide a sufficiently in-depth explanation of concepts, a deficiency that I strived to improve upon in my own tutorial.

As I investigated the available POV-Ray tutorials, one of my main goals was to analyze both their intended audiences and the extent of beginning topics they covered. I began by looking at the “Introduction to POV-Ray” section of the official POV-Ray documentation, intended to serve as a tutorial for beginning users. In some ways this was the most useful resource I encountered, to the extent that it covers all beginning topic extensively. While this may be the case, the tutorial is not particularly accessible to a wide range of users. The descriptions provided in this introduction are entirely textual, almost to an excessive degree. While POV-Ray is intended for the production of visual media, this tutorial does not include a single rendered image as an example to display the effects that can be created with the scene description language. This seems surprising since learning how to visualize a scene in three dimensions is so central to learning POV-Ray, and graphics in general. The writing style utilized in this introductory tutorial indicates that the users should practice rendering the example code provided in her desktop version of POV-Ray. While this creates a pseudo-interactive environment, I
would argue that this approach is ineffective. This tutorial requires a certain level of
dedication on the part of the user because he must open and render the POV-Ray code
discussed in the lesson in a separate window to fully understand the textual descriptions
provided. Additionally, the Introduction’s text suggests that the user simply copy and
paste the example code into a POV-Ray document to view the rendered result, which
does not promote user participation or practice in any way. Providing the rendered image
in the body of the lesson would prove just as effective and would significantly reduce the
time and effort needed to complete this introductory tutorial. Therefore, while the
documentation available at povray.org is extensive and usually the most reliable resource
for learning the scene description language, the introductory lessons are not designed to
appeal to a broad array of potential users. Instead the descriptions cater to the practiced
programmers that enjoy writing and reading code, excluding the artists that may be more
interested in visualizing the effects that can be created with POV-Ray code.

In addition to reading the official POV-Ray documentation, I examined a number
of the POV-Ray tutorials that have been developed by users. I found many resources
simply by searching for POV-Ray online, and I also utilized the links page at povray.org
that contains references to additional tutorials. The user-developed resources differed
from the POV-Ray documentation in that they used rendered POV-Ray images to support
their descriptions of syntax and programming concepts. This approach made it
significantly easier to understand the lessons provided, especially when describing topics
such as lighting or textures that require explanations of a large number of complex
keywords. Although the user-developed tutorials may have improved on the official
POV-Ray documentation in this sense, not one covered the beginning topics
exhaustively, making it difficult for a potential reader to gain a full understanding of the 
POV-Ray scene description language from any one tutorial. Additionally, none of the 
tutorials I encountered attempted to incorporate interactivity and individual practice into 
its lessons as a means of teaching POV-Ray.

As a result of the problems I encountered as I tried to learn the POV-Ray scene 
description language, I decided to design the POV-IT lessons as a means of improving 
the insufficiencies of the currently available documentation. I developed a series of 
lessons for beginners that rely on a combination of textual explanations, visual examples, 
and interactive exercises to teach the language.

5.3 Writing the POV-IT Lessons

The information I utilized to write each of the nine lessons that make up the POV-Ray Interactive tutorial was compiled from a number of resources, including the official 
POV-Ray documentation, a number of user-generated tutorials, and my own 
understanding of the scene description language. The sources I used directly in my 
research included the *POV-Ray 3.6.1 Documentation* maintained by the POV-Team, *An 
Introduction to POVRAY* written by C. K. Shene, *The Raytracing Index, The Online 
POV-Ray Tutorial* maintained by the ThinkQuest Team, *Descriptions and Samples for 
the Raytracer POV-Ray* written by Friedrich A. Lohmueller, and *Mike’s POV-Ray Page* 
written by Michael Kost (1, 2, 3, 4, 5, 6). While other POV-Ray tutorials and 
documentation are available, these are the resources I found most accurate and useful 
over the course of my investigation.
The sources I referred to while developing POV-IT can be divided into two categories – tutorials providing incremental descriptions of concepts and reference manuals providing alphabetical descriptions of POV-Ray keywords. As I strived to learn the scene description language, I found both of these methods of organizing information useful in different situations. I therefore strived to create lessons that combined textual descriptions and examples with reference-like explanations of the many complex keywords that make up the POV-Ray language. I found that some topics, such as transformations and Constructive Solid Geometry (CSG), were syntactically very simple but could be conceptually difficult to grasp. For these lessons I developed an organization modeled after a typical tutorial, with short syntactical descriptions, many visual examples, and interactive exercises to inspire users to practice writing their own code so they could gain a more thorough understanding of the concepts described.

I found that other topics, such as lighting and textures, benefited more from a reference-like organization due to the large number of keywords that needed to be explained. These lessons are not conceptually or spatially difficult to understand, yet they require knowledge of complex syntax that can be difficult to remember. Additionally, while many beginning tutorials exclude some of the keywords that can be used to create complex textures from their lessons to simplify their textual descriptions, I wanted to cover all of the potentially useful options so that users would not have to refer to another source to learn how to create a particular effect. For this reason, I decided to organize some of my descriptions in these lessons similar to reference manuals so that users can read the keyword descriptions that relate to their interests while skipping over others. While I modified my writing style in these later lessons, interactivity continued to
play an important role. The tutorial text encourages users to practice using each keyword described in the lesson in the interactive panel to gain a clearer understanding of the effects that it can create.

By combining the styles of a number of different resources intended to teach the POV-Ray scene description language, I was able to create an tutorial that provides extensive coverage of beginning topics.

5.4 Conclusion

I developed the lessons for the POV-Ray Interactive Tutorial by combining the strengths of the available resources and creating a new learning environment that caters to the user’s personal learning style through interactivity. I devoted a significant amount of time to analyzing the available tutorials to gain a clear understanding of their insufficiencies so I could improve upon their mistakes. The success of my descriptions is in many ways a result of the fact that I approached writing a POV-Ray tutorial from the user’s perspective, first learning the language myself by reading the existing documentation, and then generating a tutorial that more realistically adheres to the experience of the user.
REFERENCES


CHAPTER 6
The POV-Ray Interactive Tutorial

This chapter contains a printed version of the body of the POV-Ray Interactive Tutorial. The layout of the lessons has been modified so that the text can be read as a printed document. Instead of the two-paneled format that is used online, the printed version of the tutorial contains one panel with the textual description, followed by another containing the interactive content that appears at the start of that lesson. The nine lessons of POV-IT are included here, as well as the introductory pages providing a background on POV-Ray, ray tracing, and POV-IT. See the table of contents below:

POV-IT Table of Contents

Introductory Pages
About POV-IT
Ray Tracing Overview
About POV-Ray

Lessons
Introduction
Lesson 1: A POV-Ray scene
Lesson 2: Basic Shapes
Lesson 3: Transformations
Lesson 4: Constructive Solid Geometry Objects
Lesson 5: The Light Source Object
Lesson 6: An Introduction to Textures
Lesson 7: The Pigment Texture Statement
Lesson 8: The Finish Texture Statement
Lesson 9: The Normal Texture Statement
This is the home page of the POV-Ray Interactive Tutorial, also known as POV-IT. To begin learning the POV-Ray scene description language, visit the Table of Contents for a list of interactive lessons.

POV-Ray is an open source ray tracer that can be used to create beautiful and realistic images. Its intuitive syntax is easy to learn and fun to use. This tutorial is designed for beginning users, especially those without much previous programming experience. It provides an interactive setting that allows users to write and render POV-Ray code through the site's AJAX enabled interface, facilitating the concurrent learning and practice of new concepts.

© 2007, Nell O'Rourke
About POV-IT

There is an extensive online POV-Ray community, and many tutorials have been written to help teach beginning users the nuances of the POV-Ray scene description language. This tutorial, however, provides an interactive environment that I have not yet encountered in another POV-Ray tutorial. If you are new to POV-Ray and interested in trying it out before going through the process of downloading and installing the program, or simply interested in improving your POV-Ray skills in an online interactive environment, this tutorial is for you.

This site provides a server-side program to analyze and render POV-Ray code, allowing you to practice skills as you learn them and complete the various programming problems and exercises provided over the course of the tutorial. While the scene description language is intuitive and the syntax simple, learning to utilize the options to their full potential can be difficult. This tutorial therefore tries to explain concepts simply and succinctly, placing more emphasis on writing code than lengthy descriptions.

I hope you enjoy this tutorial, and please direct any questions, comments, or suggestions to Nell O'Rourke: emorourk@colby.edu

© 2007, Nell O'Rourke
What is Ray Tracing?

The acronym POV-Ray stands for the "Persistence of Vision Ray Tracer." It is an open source ray tracing program that can be used to create highly sophisticated photo-realistic images. Before delving into POV-Ray, it may be useful to learn a little bit about how ray tracing works.

Ray tracing is a rendering technique that creates realistic images by simulating the physical movement of light through a three dimensional scene. In the real world, you see an object when the light emanating from a light source bounces off its surface into your eyes. While ray tracing programs attempt to simulate this effect, they traditionally trace rays backwards. This is because a huge number of the rays that propagate from a light source will never actually reach the eyes of the observer. It is therefore much more time efficient to trace rays backwards from the camera, which simulates the observer's viewpoint, through the scene and back to the light source.

A scene to be rendered using a ray tracer contains a camera as well as a collection of light sources and objects, all of which are arranged in a three dimensional world according to a particular coordinate system. While ray tracing only supports simple objects that can be described mathematically (spheres, cones, cylinders), more complex objects may be created by combining these simple building blocks. Each object can be given certain surface properties, such as transparency and reflectiveness, which describe how light interacts with the object.

To calculate the color of each pixel in the final rendered image of a scene, the ray tracer sends one or more rays from the camera through the viewing screen (the final image) into the scene to determine whether they intersect with any of the defined objects. If a ray does intersect with an object, the color at that point on the object's surface is calculated by sending reflected rays from its surface to determine whether it is in view of a light source, or if it reflects any other objects in the scene. After this calculation is completed, the color at this point on the object's surface is transferred to the corresponding pixel in the final image. Using this method, the color of each pixel is painstakingly calculated to create the ultimate realistic image.

This is only a very brief introduction to ray tracing, so if you are interested in learning more about the technique more information can be gathered from the resources listed below.
More Resources:

- [http://www.geocities.com/jamisbuck/raytracing.html](http://www.geocities.com/jamisbuck/raytracing.html) -- A great overview of ray tracing that is both in-depth and comprehensible.

- [http://fuzzyphoton.tripod.com/whatisrt.htm](http://fuzzyphoton.tripod.com/whatisrt.htm) -- This description includes an explanation of the physics of light.


POV-Ray stands for "The Persistence of Vision Ray-Tracer." It is an open source ray-tracing program that began as an extension of DKBTrace, a ray-tracer developed by David K. Buck and Aaron A. Collins in the 1980s (1). In 1989 with the release of DKBTrace 2.12, Buck and Collins were feeling the pressure of their increasingly popular ray-tracer as their users expressed desires for more and more new features. To remedy the increasing demand for a fully capable ray-tracing program, Buck offered the DKBTrace system as the base for the POV-Ray project (1). A group of volunteers known as the POV-Team, mostly previous users of the DKBTrace system including both Buck and Collins, combined forces to add the desired features to the program and update its documentation. While both Buck and Collins left the project shortly after its creation to pursue other interests, the POV-Team continues to expand and modify POV-Ray's capabilities to this day (1).

The Persistence of Vision Ray-Tracer reads in a text file written in a language known as the POV-Ray scene description language (2). The scene description language provides all of the functionality necessary to describe a three-dimensional scene including objects, lights, and a camera. Additionally, the POV-Ray package provides a number of include files supplying predefined colors, textures, pigments, shapes, and functions. As POV-Ray's capabilities has grown since the early 90s, many advanced features such as lathe objects, isosurfaces, and radiosity have also been incorporated into its syntax. POV-Ray therefore provides both advanced modeling and lighting capabilities, making it an extremely useful program.
The popularity of POV-Ray is evidenced by the large number of registered users. The POV-Ray community map available at povray.org/community/map/ provides listings of all registered users in different areas of the world, including 1,054 users within the United States (3). The actual number of POV-Ray users is most likely much larger, since registration is only necessary for posting questions and comments on the community forums. The artistic applications of POV-Ray are widespread, ranging from the Visual Music animations of Dennis Miller to the still images of Tor Olav Kristensen, Jaime Vives Piqueres and others who have submitted work to the POV-Ray hall of fame (4, 5, 6). Many computer science departments use POV-Ray to teach computer graphics, including those at the California Institute of Technology, University of Illinois-Springfield, and Smith College (7, 8). While POV-Ray is widely used, the documentation available for the application is confusingly disorganized. The official website povray.org provides extensive resources, but because they have been produced gradually over a number of years by volunteers, the resulting documents can difficult to parse for useful information. For this reason, POV-IT has been developed as a resource for beginning POV-Ray users that aims to provide clear and concise documentation.

Works Cited:

(1) www.povray.org/documentation/view/3.6.1/7/
(2) www.povray.org/documentation/view/3.6.1/3/
(3) www.povray.org/community/map/
(4) www.dennismiller.neu.edu/
(5) www.home.no/t-o-k/povray/
More Resources:

- www.wikipov.org/ -- The unofficial POV-Ray wiki.
POV-IT
The POV-Ray Interactive Tutorial

POV-IT Table of Contents

Introduction: Tutorial instructions

Lesson 1: The coordinate system, the camera, and simple lighting

Lesson 2: Basic shapes

Lesson 3: Translating, rotating, and scaling objects

Lesson 4: Constructive Solid Geometry (CSG) Objects

Lesson 5: The Light Source Object
Lesson 6: An Introduction to Textures

Lesson 7: The Pigment Texture Statement

Lesson 8: The Finish Texture Statement

Lesson 9: The Normal Texture Statement

© 2007, Nell O'Rourke
This interactive tutorial is designed to teach the POV-Ray scene description language through practice, allowing users to write and render POV-Ray code as they learn new techniques. It is based on the theory that by writing code in a controlled environment designed specifically to teach certain skills, users will gain a comfortable understanding of POV-Ray quickly and will be able to apply the concepts they learn here to their own projects more easily. For this reason, user participation is strongly suggested, and the textual descriptions are designed with the expectation that concurrent practice will strengthen the user's understanding of the concepts described.

The main window of each lesson is divided into two sections. The left section describes POV-Ray syntax and concepts, while the right section provides an interactive environment where you can practice writing code. The interactive section contains a text-box holding POV-Ray code and the image generated by rendering that code. There are two buttons at the bottom of the text-box, a Render button which will render the code in the text-box and display the resulting image, and a Reset button which resets the text-box to hold original code for that lesson.

The interactive section is equipped with error checking, so if you have a syntax error in your code you will be notified. Try deleting one of the curly braces "{" or "}" in the text-box to the right and click the Render button to see what an error message looks like. Then click the Reset button to reset the code and click Render again to see the original image again.

Finally, throughout the tutorial you will be presented with various practice problems. The answers to these exercises are provided through drop down answer sections. Simply click on the red "Answer" link, such as the one below, to view the answer. Click "Answer" again to hide the code.

Answer

Your answer here!
```cpp
#include "colors.inc"

camera {
    location <0, 2, -6>
    look_at <0, 1, 0>
}

light_source { <20, 20, -20> color White }

plane {
    y, -1
    pigment { checker color White color Black }
}
sphere {
    <1, 1.5, 0>, 1.5
    pigment { color Red }
    finish {
        reflection 0.4
        phong 1
        phong_size 50
    }
}

render button { Render Reset }
```

Table of Contents  Next Lesson
Lesson 1
An introduction to POV-Ray

A POV-Ray Scene

To create an image using the ray tracer POV-Ray, you have to define the elements that will appear in your three dimensional "scene". A scene is comprised of a camera, a light source, and the objects you will be viewing. The camera defines your viewpoint in relation to the scene, and is comprised of both a location and a viewing direction. The light source determines how the scene will be lit. Light sources are essential scene components because ray tracers depend on light to model a scene; without a light source your scene will be completely black. Finally, a scene contains the three dimensional objects that are the focus of your piece. All of these scene components are described using a simple programming language called the POV-Ray scene description language.

The POV-Ray Scene Description Language

The POV-Ray scene description language is an intuitive object-oriented programming language providing users with the syntax needed to describe scenes. While the complexity of POV-Ray code increases with scene complexity, the basic syntax of all objects is the same. Each object is defined using a block of code that begins with the object name followed by curly braces "{" and "}". All of the information needed to describe that object appears inside the curly braces. This information could include the object's location, color, size or it's surface properties. Over the course of this tutorial we will learn the specific syntax of many of the objects that can be described using POV-Ray.

For now, take a minute to browse through the code in the interactive section to the right. Don't worry about trying to understand exactly what it means yet, but note that the code is broken up into a number of object statements separated by curly braces. As you might notice, there are many object definitions in this scene, not just definitions for the sphere, cone, and torus that are directly visible. This is because some objects are out of view of the camera. If you look closely at the red sphere, you will notice that there are a few non-visible objects reflected in its surface.

The Coordinate System

The locations of all POV-Ray objects are described using a three dimensional coordinate system. POV-Ray uses what is called a "left-handed" coordinate system. Consider the image below, which displays the coordinate system:
This coordinate system is "left-handed" because you can create it with the fingers on your left hand. Make an "L" shape with your thumb and pointer fingers. Your pointer finger is the y-axis, and your thumb is the x-axis. If you stick out your middle finger perpendicular to your pointer finger, you have your z-axis, which points away from you. You can also think about the coordinate system in terms of your computer screen, which is sometimes easier to visualize. If the origin is in the lower left-hand corner of your screen, the y-axis points up the left side of your screen and the x-axis points along the bottom of your screen. The z-axis points directly into your monitor.

A position in POV-Ray are described using a three component vector. The three coordinates, \( x, y, \) and \( z \), are surrounded by angle brackets \((< \text{ and } >)\) to represent a point in space. For example, \(<0,0,0>\) represents the origin. For convenience, POV-Ray has predefined normal vectors for each of the axes. In other words, you can use:

- \( x \) in place of \(<1,0,0>\)
- \( y \) in place of \(<0,1,0>\)
- \( z \) in place of \(<0,0,1>\)

Take another look at the code in the interactive section. Note the frequency with which three coordinate points appear in the code. These points are all used to describe various aspects relating to the position and orientation of the objects.

---

The Camera

The camera is an essential part of every POV-Ray scene. It determines the observer's viewpoint in relation to the other objects in the scene. There are a number of ways to describe a camera in POV-Ray, the simplest of which consists of a location and viewing direction:

```plaintext
camera {
  location \(<x,y,z>\)
  look_at \(<a,b,c>\)
}
```
The keyword `camera { ... }` indicates that you are describing a camera object. The `location` keyword is used to define the position of the camera. It is always followed by a three coordinate point `<x,y,z>` which determines the location of the camera in three dimensional space. The statement `look_at <a,b,c>` determines the camera's viewing direction. It tells the camera that it should look along the vector between the point `<x,y,z>` where the camera is located and the point `<a,b,c>` where the camera is looking. In other words, the camera should orient itself such that the point `<a,b,c>` is directly in the center of the final image.

The `location` and `look_at` keywords allow you to describe the location and orientation of the camera with relatively little work. Alternately, if you would like to manually describe the orientation of the camera, you can use the following definition:

```
camera {
  location <x,y,z>
  direction <a,b,c>
  up <i,j,k>
  right <l,m,n>
}
```

The keywords in this `camera` definition are used as follows:

- **location**: As with the simpler `camera` syntax, this keyword is used to describe the camera's coordinate location in three dimensional space.
- **direction**: This keyword is used to describe the vector that determines the direction in which the camera points. The length of the vector describes the focal length of the camera, allowing for the simulation of various types of lenses. For example, a larger focal length signifies a telephoto lens, while a smaller one signifies a wide-angle lens.
- **up**: This keyword is used to define the vector that points in the camera's "up" direction.
- **right**: This keyword is used to describe the vector that points to the right of the camera.

These three additional keywords allow you to describe the orientation of your camera more precisely. I would suggest, however, that beginners use the `look_at` keyword to start, since it provides a much more intuitive way to think about camera orientation.

Locate the camera definition in the code to the right. Try modifying the location and `look_at` points to view the scene from a different position and direction. For example, what happens if you change the camera's `location` to `<0,2,-16>`? Or if you change it to `<5,2,10>`?

### Simple Lighting

As a ray tracer, POVRay can generate very realistically-lit scenes. Understanding and describing light sources is therefore one of the most important, and most difficult, aspects of creating a POVRay scene. We will wait to delve into the nuances of complex lighting until lesson 5, but for now we will learn to create a simple light source so that at the very least we can see our scene. This is the syntax of a simple light source:
light_source {
  \{x,y,z\}
  color C
}

The point \{x,y,z\} determines the location of the light. The statement color C is used to define the color of the light source. This is the syntax for using a pre-defined color in POV-Ray, which will be discussed in more depth in the next section. Usually, however, we will simply replace the variable \(C\) with the color "White" to create a white light source. Locate the light_source definition in the interactive section to the right to see an example of the code needed to describe a light.

Try changing the location of the light source and rendering the code. What effects are caused by moving the light?

**Include Files**

The POVRay package includes a number of libraries containing predefined colors and textures. These can be very useful, especially for beginning users who want to make nice realistic images without going through the work of creating their own complex textures. The syntax for including a file is as follows:

```
#include "package_name.inc"
```

Include statements always appear at the very beginning of your code. Find the include statement in the code to the right to get an idea of what an include statement looks like. The most commonly used package is the "colors.inc" package, which includes all of the predefined colors. As we saw in the light_source definition above, predefined colors are preceded by the keyword color. Other packages, such as "woods.inc" and "stones.inc" can be included to use predefined textures, which will be discussed in later lessons. To see the colors that are included in the "colors.inc" package, you will want to use a reference source. Some of the POVRay GUIs contain a list of all of the colors in their help sections. You can also use the reference web page povray.tashcorp.net/library/, which contains visual renderings of all the predefined colors.
Interactive Content
```c
#include "colors.inc"

camera {
    location <0, 2, -6>
    look_at <0, 0.5, 0>
}

light_source { <20, 20, -20> color White }

plane {
    y, -1
    pigment { checker color White color Black }
}

sphere {
    <0.5, 0, 0>, 1
    pigment { color Red }
    finish {
        reflection 0.4
        phong 1
        phong_size 50
    }
}

cone {
    <2.0, 2>, 1.5,
    <2.3, 2>, 0.2
    pigment { color Violet }
}
torus {
    1.2, 0.5
    pigment { color Green }
    rotate <-40, -20, 0>
    translate <-2, 1.5, 1>
}

sphere {
    <0, 2, -10>, 1.5
    pigment { color NavyBlue }
    finish {
        reflection 0.7
    }
}

cone {
    <-4, 0, -5>, 1.5,
    <-4.3, -5>, 0.2
    pigment { color Yellow }
}
torus {
    1.2, 0.5
    pigment { color Orange }
    rotate <-20, -20, 0>
    translate <-5, 2.5, -8>
}
```

Render  Reset
Lesson 2
Basic POV-Ray Shapes

In this lesson, you will learn how to use POV-Ray's predefined shape objects. These basic shapes are the simplest POV-Ray objects, with straightforward syntax that is easy to learn. Using only the objects covered in this lesson, you will be able to model a relatively complex sledding snowman figure.

The Plane Object

The **plane** object statement is used to define an infinite plane. In order to describe an infinite plane, you must provide POV-Ray with the plane's normal vector as well as the plane's distance from the origin along this vector. The syntax is as follows:

```plaintext
plane {
    <x,y,z>, d
    pigment { color C }
}
```

Here the vector \(<x,y,z>\) represents the plane's normal vector. A normal vector is a vector perpendicular to the plane. The variable \(d\) defines the position of the plane along that normal vector. For example, to add a horizontal plane serving as the "ground" to your scene, you would use the y-axis as your normal vector. If you would like this plane to lie at the origin, you could use 0 as your \(d\) value. The **pigment** statement describes the color of the plane.

Try adding a plane to your scene in the interactive section. Think of it as a snowy ground. To view one possible solution, click the "Answer" link below.

**Answer**

```plaintext
plane {
    <0,1,0>, 0
    pigment { color White }
}
```
The Sphere Object

The sphere object is defined by its center point and its radius. The syntax is as follows:

```plaintext
sphere {
    <x, y, z>, r
    pigment { color C }
}
```

In this definition, \(<x,y,z>\) represents the point at the sphere's center, and \(r\) represents the sphere's radius. As with the plane object, we can give the sphere a color by defining its `pigment` value. To get a feel for the sphere object, try adding three spheres to your code to create the snowman's body.

**Answer**

```plaintext
sphere {
    <0, 0.8, 0>, 1
    pigment { color White }
}
sphere {
    <0, 2.1, 0>, 0.9
    pigment { color White }
}
sphere {
    <0, 3.3, 0>, 0.7
    pigment { color White }
}
```

Now that your snowman has a body, try adding two "coal" spheres for eyes.

**Answer**

```plaintext
sphere {
    <-0.2, 3.5, -0.65>, 0.1
    pigment { color Black }
}
sphere {
    <0.2, 3.5, -0.65>, 0.1
    pigment { color Black }
}
```

The Cone Object

The cone object is described using the following syntax:
In the first line, the point \(<x,y,z>\) defines the center point of one end of the cone and the variable \(r_1\) defines the radius of that same end of the cone. In the second line, the variables \(<a,b,c>\) and \(r_2\) define the center point and radius of the other end of the cone. The optional open keyword can be added to the cone definition to create a cone with open ends. To create a cone with closed ends, simply omit this keyword. Finally, the pigment statement is included to describe the color of the object.

Try using the cone object to give your snowman a carrot nose. You may have to move the camera closer to the snowman to get a good view of his face.

**Answer**

```plaintext
cone {
  <0, 3.3, -0.65>, 0.12,
  <0, 3.3, -1.2>, 0.01
  pigment {color Orange}
}
```

---

**The Cylinder Object**

The cylinder object can be thought of as a simpler version of the cone. It has two center points (one for each end), but only one radius. See the syntax description below:

```plaintext
cylinder {
  <x,y,z>,
  <a,b,c>,
  r
  open //optional
  pigment {color C}
}
```

Here, the first point \(<x,y,z>\) defines the center of one end of the cylinder, and the second point \(<a,b,c>\) defines the center of the other end of the cylinder. The variable \(r\) determines the cylinder's radius. Again, add the optional open keyword to create a cylinder with open ends, and the pigment statement is used to define the cylinder's color.

Practice using the cylinder object by giving your snowman stick arms.

**Answer**
The Box Object

The box object is defined using only two points, its near lower left corner and its far upper right corner:

```
box {
    <x,y,z>,
    <a,b,c>,
    pigment { color C }
}
```

In the definition above, the box's near lower left corner is given by the point \( <x,y,z> \) and its far upper right corner is given by the point \( <a,b,c> \). Try using the box object to create the rim for the snowman's hat. Then add another cylinder to form the body of the hat.

**Answer**

```
box {
    <-0.7, 3.8,-0.7>,
    <0.7, 4, 0.7>
    pigment { color Black}
}
cylinder {
    <0, 4, 0>,
    <0, 4.8, 0>,
    0.5
    pigment { color Black }
}
```

The Torus Object

The torus object is the last simple shape predefined in POV-Ray. A torus can be thought of as a donut shaped object that is defined by its two radii. The syntax for describe a torus object is as follows:
The first radius, \( r_1 \), is called the *major radius* while the second, \( r_2 \) is called the *minor radius*. The major radius can be thought of as the radius of the large circular area taken up by the donut. The minor radius determines the "thickness" of the donut, and can be thought of as the radius of the circular sliver created by cutting the donut. As you may have noticed, the torus syntax does not require you to define any points. This is because the torus object is always created by rotating a circle of radius \( r_2 \) about a circle of radius \( r_1 \) lying in the x-z plane. Therefore, each torus will be centered at the point \( <0,0,0> \).

As you might imagine, tori are not particularly useful if they cannot be moved to other locations. As you will learn in Lesson 3, the transformations can be applied POV-Ray objects to move them to other locations. For now, try adding a torus inner tube to your scene in its generic location centered at \( <0,0,0> \). You may have to move the ground plane to be able to see the entire torus object.

**Answer**

\[
\text{torus} \{ \\
\quad r_1, \\
\quad r_2 \\
\quad \text{pigment \{ color Black \}} \\
\}
\]

Change your plane to:

\[
\text{plane} \{ <0,1,0>, -0.7 \\
\quad \text{pigment \{ color White \}} \\
\}
\]
Interactive Content

#include "colors.inc"

camera {
    location <0, 3, -7>
    look_at <0, 2.5, 0>
}

background { color SkyBlue }

light_source { <20, 20, -20> color White }
light_source { <0, 3, -17> color White }

[Render] [Reset]
This lesson discusses how to use the POVRay keywords `translate` and `rotate` to move objects in a scene, and the `scale` keyword to change an object's size. The syntax of these three transformation operations is simple, but learning how to use them correctly takes time and practice.

**Translation**

The `translate` keyword can be used to move objects in a POVRay scene.

```
translate <x,y,z>
```

The vector `<x,y,z>` specifies the magnitude of movement in the x, y, and z directions. Translations are made relative to the object's current location. Consider the following example:

```
sphere {
    <10, 10, 10>, 1
    translate <5,-10,-5>
}
```

This sphere, initially centered at `<10, 10, 10>`, moves to `<15, 0, 5>` after translation (rather than the absolute point `<5,-10,-5>`).

To practice translating objects, try to make the body of this caterpillar. Define each sphere to be centered at `<0,0,0>`, and then translate it to the desired location. Try adding eyes using the same method.

**Answer**

```
// The caterpillar's body
sphere {
```
Rotation

The \texttt{rotate} keyword can be used to change the orientation of an object by rotating it about the x, y and z axes. The syntax is as follows:

\begin{verbatim}
rotate \langle x, y, z \rangle
\end{verbatim}

The vector $\langle x, y, z \rangle$ determines the number of degrees to rotate about the x, y, and z axes respectively. Ordering is important, because POV-Ray rotates the object first about the x axis, then the y, then finally the z. As you will remember from Lesson 1, POV-Ray uses a left-handed coordinate system. You can determine the direction of rotation about an axis by pointing the thumb of your left hand in the positive direction of that axis. Your fingers will curl about the axis in the direction of rotation. For example, if the camera is positioned at $\langle 0, 0, -5 \rangle$ looking at $\langle 0, 0, 0 \rangle$:
- Objects rotate away from the camera about the x axis.
- Objects rotate to the left about the y axis.
- Objects rotate counter clockwise about the z axis.

Consider the following example:

```plaintext
sphere {
  <2, -2, 0>, 1
  pigment { color Red }
  rotate <0, 45, 270>
}
```

Here the sphere does not rotate about the x axis since the vector’s x value is 0. Therefore, it first rotates 45 degrees about the y axis, and then 270 degrees about the z axis.

Sphere’s original location before rotation

Sphere rotates 45 degrees about the y axis

Sphere rotates 270 degrees about the z axis

When rotating an object about only one axis, you can use this shorthand notation instead:

`rotate 45*x`
This has the same effect as writing `rotate <45,0,0>`.

Now that you have learned how to rotate objects, try to add red spots to the caterpillar by rotating and translating tori.

**Answer**

```plaintext
torus {
  0.4, 0.1
  pigment { color Red }
  rotate <-45,-20,0>
  translate<1.1,0.3,-0.25>
}
torus {
  0.4, 0.1
  pigment { color Red }
  rotate<-30,20,10>
  translate<1.9,0.7,-0.2>
}
torus {
  0.4, 0.1
  pigment { color Red }
  rotate<-20,20,-24>
  translate<3.1,0.65,-0.1>
}
torus {
  0.4, 0.1
  pigment { color Red }
  rotate<-55,0,-20>
  translate<4.1,0.2,-0.3>
}
```

**Scaling**

The `scale` keyword in POV-Ray can be used to modify the scale of an object in three dimensions.

```
scale <x,y,z>
```

Consider a point `<a,b,c>` in the object. After scaling, it will be changed to `<a*x, b*y, c*z>`. Remember that if you don't want to scale in a particular direction, use 1 instead of 0 (multiplying all coordinates by 0 in that direction will have disastrous results).

Here is an example of how scaling objects can be useful:
This pile of stone shaped objects was created by scaling spheres. All of these spheres were initially the same size, but their shapes have been greatly modified due to scaling.

Additionally, if you wish to scale an object by the same factor in the x, y and z directions (in other words, change the size of the object while maintaining its proportions) you can write `scale 3` instead of `scale <3,3,3>`.

Now that you know a little bit about scaling objects, try to add some grass to your scene by scaling spheres to create grass-like objects.

**Answer**

```plaintext
sphere {
  <0,0,0>, 0.5
  pigment { color DarkGreen }
  scale<0.1,3,0.1>
  rotate z*10
  translate<4.6,0.75,0>
}
sphere {
  <0,0,0>, 0.5
  pigment { color DarkGreen }
  scale<0.1,4,0.1>
  rotate z*2
  translate<4.7,1,0>
}
sphere {
  <0,0,0>, 0.5
  pigment { color DarkGreen }
  scale<0.1,3,0.1>
  rotate z*-4
  translate<4.7,0.75,-0.7>
}
sphere {
  <0,0,0>, 0.5
  pigment { color DarkGreen }
  scale<0.1,2,0.1>
  rotate z*2
  translate<4.3,0.5,-1>
}
```
sphere {
  <0,0,0>, 0.5
  pigment { color DarkGreen }
  scale<0.1,1,0.1>
  rotate z*5
  translate<4,0,-1.1>
}
sphere {
  <0,0,0>, 0.5
  pigment { color DarkGreen }
  scale<0.1,1.5,0.1>
  rotate z*-5
  translate<4.1,0.25,-1.1>
}
#include "colors.inc"
#include "woods.inc"
camera {
    location <2, 1, -5>
    look_at <2, 1, 0>
}
light_source { <20, 20, -20> color White }
light_source { <0, 3, -17> color White }
plane {
y, -0.6
    pigment { color DarkGreen }
}
box {
    <-10,-1,1>, <10,10,2>
    texture { T_Wood2 }
}
Lesson 4
Constructive Solid Geometry Objects

This lesson discusses how to use Constructive Solid Geometry (CSG) to make complex objects by combining predefined POV-Ray objects in various ways. Over the course of the lesson, you will learn how to make this lamp using nothing other than the simple objects described in Lesson 2 and a couple of new keywords.

Overview

A CSG object is an object that has been created by combining other objects in some way. These other objects may be either primitive objects such as sphere, cone, and box, or they may be other CSG objects. There are four different ways to combine objects. First, the union keyword allows you to add objects together. The difference keyword takes the first listed object and subtracts all subsequent objects from it. The intersection statement creates a new object comprised of the area common to both of the original objects. Finally, the merge statement works much like the union statement in that it adds two objects together, except that all interior surfaces are removed from the resulting object. Merging can be very useful when creating transparent objects.

These combinations of objects can be deeply nested. For example, you may have a union of a difference and an intersection. This allows you to create very complex objects using a relatively simple syntax.

Union

To add two object together, use the union keyword. A union can be used to combine as many objects as desired, but the basic syntax for two objects is as follows:

```plaintext
union {
    object1
    object2
    pigment { color C }
    /** Optional transformations such as translate, rotate, or scale */
}
```
Here, *object1* and *object2* are the objects to be joined. If you wish to join more than two objects, simply include subsequent objects after *object2*. The *pigment* statement defines the color of the combined CSG object. After defining the object’s color, you can use transformations such as *translate*, *rotate*, and *scale* to modify the location, orientation, and dimensions of the object. Note that the pigment statement and transformations apply to the entire union, not just to *object1* or *object2*. Consider the following example for further clarification:

```plaintext
union {
    sphere{<0,2,0>, 1 }
    cone {
        <0,2,0>, 1
        <0,0,0>, 0.1
    }
    pigment { color Red }
    rotate <0,0,25>
}
```

In the example above, a sphere and a cone are joined together using the *union* keyword, and then the CSG object is colored and rotated as a unit.

The CSG lamp we will be making in this lesson can be thought of as one big union -- a number of different pieces created using various CSG keywords are added together to create the final object using a union statement.

The base of the lamp can be created by using *union* to combine a cylinder and a cone. Try making this base in the right hand half of the window to practice using unions. Remember that in order to add two objects together, they should overlap a little bit.

**Answer**

```plaintext
union {
    cylinder {
        <0,0,0>,<0,1,0>, 4
    }
    cone {
        <0,0.99,0>, 4
        <0,2,0>, 2.5
    }
}
```
Difference

The `difference` keyword can be used to subtract one object from another. It has a very similar syntax to the `union` keyword:

```plaintext
difference {
  object1
  object2
  pigment { color C }
  /** Optional transformations such as
   translate, rotate, or scale */
}
```

Here, `object1` is the initial object, and all subsequent objects described in the `difference` statement will be subtracted away from `object1`. In this case, `object2` will be subtracted from `object1`. The resulting CSG object will be given the color described in the `pigment` statement, and will be transformed as a unit if transformations are included. Let's look at an example:

```plaintext
difference {
  box {
    <-1,-1,-1>,
    <1,1,1>
  }
  cylinder {
    <0,0,-2>,
    <0,0,2>,
    0.5
  }
  pigment { color Red }
  rotate <0,15,0>
}
```

Here, a cylinder is subtracted away from a box, resulting in a box with a cylindrical hole cut through it. The CSG object is then colored and rotated as a unit.

We can use `difference` to create the next piece in the lamp (shown in red) by taking a box and subtracting a torus from it, so that all that is left of the box is the curvy hole at the center of the torus. Create this shape at the origin and then translate it into place. Add this difference definition as the third object in your main union statement, after the cylinder and cone that make up the base. For decoration, add a thin torus to the top of this curved piece, as shown in blue.
Answer

Add these lines to your main union, after the cone and cylinder that make up the lamp's base.

difference {
box {
<-2.5,-1.5, -2.5>,
<2.5,1.5,2.5>
}
torus { 3, 1.6}
translate <0,3.5,0>
}
torus {
2.1,0.5
translate <0,5.4,0>
}

Intersection

The `intersection` keyword can be used to create a CSG object made up of the area common to two or more overlapping objects.

```
intersection {
object1
object2
pigment { color C }
/** Optional transformations such as
 translate, rotate, or scale */
}
```

Here, the CSG object resulting from the intersection is made up of any overlapping area between `object1` and `object2`. The object is then colored and transformed as a unit. See the following example:

```
intersection {
  sphere { <-1,0,0>, 2 }  
sphere { <1,0,0>, 2 }  
pigment { color Red }
}
```

In this example, two overlapping spheres are combined using `intersection` to create an almond-shaped object formed by the area common to both spheres.
You can use the `intersection` keyword to add a half-sphere to the lamp body by taking the intersection of a box and a sphere. Try it now to practice using intersections. Again, add the half-sphere as to your main `union` statement.

``` Answer
intersection {
    sphere { <0,0,0>, 4
    box { <5,0,5> } 
    translate <0,8.8,0>
}
Answer

To finish off the lamp, use another box-torus `difference` to create the curvy piece shown in red. Then add a tall thin cylinder as the lamp's "neck" and a large open cone (shown in blue) as the lamp shade. Don't forget to color the lamp shade a nice white color. To color one piece of the `union` differently from the rest, add a `pigment` declaration inside the cone definition as you do for normal objects.

``` Answer
difference {
    box {
        <3.65,1.9, 3.65> 
        < 3.65,-1.9, -3.65>
    }
    torus { 4.5, 2
    translate <0,10.7,0>
}
cylinder {
    <0,20,0>, 0.5
}
cone {
    <0,14,0> 10,
    <0,25,0> 5
    open
    pigment { color White }
}
### Merge

The CSG keyword `merge` adds two or more objects together just as the `union` keyword does. The difference is that all interior geometry is not drawn, making it useful for transparent objects.

```plaintext
merge {
    object1
    object2
    pigment { color C }
    /** Optional transformations such as translate, rotate, or scale */
}
```

Here, `object1` and `object2` are added together, and then all edges and surfaces that appear inside the object are removed to create a clean, solid geometry. The object is then colored and transformed as a unit. Consider the following example, where a `union` statement is used with a transparent object:

```plaintext
union {
    sphere{ <0,2,0>, 1 }
    cone {
        <0,2,0>, 1
        <0,0,0>, 0.1
    }
    pigment { Red filter 0.5 }
    rotate <0,0,25>
}
```

This image shows what happens when we make the CSG union discussed in the first section transparent by adding the statement `filter 0.5` to the `pigment` definition. Filters will be discussed in further detail in Lesson 6.1. When we make this object transparent, its interior geometry becomes clearly noticeable. To solve this problem, we can replace the `union` keyword with `merge`:

```plaintext
merge {
    sphere{ <0,2,0>, 1 }
    cone {
        <0,2,0>, 1
        <0,0,0>, 0.1
    }
    pigment { Red filter 0.5 }
    rotate <0,0,25>
}
```

The `merge` keyword removes all interior geometry so that the object appears solid and seamless.
To get a better understanding of `merge` and its uses, try adding `filter 0.5` to your lamp's pigment definition. Don't forget to add it to the lamp shade's white pigment definition too. Then try changing your main `union` to a `merge` to see the difference.
Interactive Content

#include "colors.inc"

camera {
    location <0, 14, -40>
    look_at <0, 13, 0>
}

light_source { <20, 20, -20> color White }
light_source { <0, 3, -30> color White }

background { color DarkSlateGray }

[Button: Render] [Button: Reset]
Lesson 5
The Light Source Object

In Lesson 1, you learned about the simplest variation of the light_source object - the pointlight. This light source is defined by its location and color only. However, there are a number of other variations of the light_source object that provide many more options. Learning to utilize these more advanced light sources can greatly improve your ability to create interesting effects with your lighting.

Overview

There are a number of different types of light sources that can be defined in POV-Ray, but they all have the same basic syntax which makes it easier to learn how to use them. This lesson will introduce a number of new keywords, which you should practice using in the interactive section since writing your own POV-Ray code is the only way to become truly comfortable with this material. In this lesson, the following light source definitions will be covered:

- pointlight
- spotlight
- cylindrical light
- area light
- spotlight/cylindrical light that is also an area light

The Pointlight

The pointlight, which was also introduced in Lesson 1, is the simplest light source definition.

```
light_source { 
  <x,y,z>
  color C
}
```

The pointlight is defined by its location <x,y,z> and its color C. It is invisible and has no volume; it is simply a point in space that emits light in all directions. A pointlight illuminates all objects in the scene equally no matter how far they are from it. A pointlight is currently used to light the scene in the right half of the window, defined as follows:
light_source {
  20, 20, -20
  color White
}

If you look closely at the image, you will notice that the checkered plane is illuminated equally out to the horizon. To convince yourself that the light doesn't dissipate as objects are moved farther away from the light source, try translating the green torus away from the camera (to a location in the positive z direction) and render the scene. You will notice that the torus appears just as brightly illuminated as when it was closer to the light source. While this type of lighting can be useful in some cases, it is not particularly realistic.

Answer

torus {
  1.2, 0.5
  pigment { color Green }
  rotate <-40, -20, 0>
  translate <-2, 1.5, 1>
  translate <0,0,30>
}

The Spotlight

A spotlight can be used to highlight a certain area of a scene. Unlike the pointlight, it illuminates objects based on their distance from the source, creating a more natural effect. A spotlight produces a conical light emanating from a point in space. The syntax for the spotlight is slightly complicated, but this is beneficial because it gives you a lot of control over the size and brightness of the light.

light_source {
  x, y, z
  color C
  spotlight
  radius r
  falloff f
  tightness t
  point_at <a, b, c>
}

Since the spotlight introduces so many new keywords, we are going to begin with a sample light and walk through the purpose of each keyword interactively. First, copy the following light definition and paste it into the code on the right half of the screen (delete the original pointlight definition), then render the scene to get an idea of the effect a spotlight can create.
light_source {
  <10,10,-10>
  color White
  spotlight
  radius 15
  falloff 18
  tightness 10
  point_at <1,0,1>
}

The first three parameters are relatively simple:

- `<x,y,z>` Determines the location of the light (as with the pointlight).
- `color c` Defines the light's color (as with the pointlight).
- `spotlight` Refers to the type of light source. Here, we are defining a spotlight so we use the `spotlight` keyword.

The next four parameters are a little bit trickier. Read the descriptions of each parameter and play around with its value in the interactive section until you understand how it works.

- `radius r` Determines the size of the cone that is fully illuminated. The variable `r` should be replaced with the desired angle, in degrees, between the center of the cone and the cone's outer edge.
- `falloff f` This keyword determines the size of the cone outside of the fully illuminated cone defined using the `radius` keyword where the light gradually falls off into darkness. The falloff value `f` should always be greater than or equal to the radius value `r`. If `f` is equal to `r`, then the resulting spotlight will have very sharp edges since the cone defined by radius `r` will be completely illuminated and the area outside this cone will be completely dark. As the fallout value `f` increases with respect to `r`, the edges of the spotlight will become fuzzier and fuzzier.
- `tightness t` Tightness is an optional keyword that can be used to modify a spotlight's falloff rate. Lower tightness values produce spotlights with sharp edges that fade quickly, while higher tightness values produce spotlights with more gradually fading edges. As the tightness value increases, the fully illuminated cone defined using the `radius` keyword may become faded around the edges as well. The default tightness value is 10, but it can range anywhere from 0 to 100, or even larger if desired.
- `point_at` The `point_at` parameter for spotlights is very similar to the `look_at` parameter for cameras. The center of the cone will pass through the point `<a,b,c>`.
Cylindrical light

The cylindrical light source is essentially the same as the spotlight, except that it generates a cylindrically shaped light instead of a conical one. This can be useful because while the apparent size of a spotlight is dependent on an object's distance from the light, a cylindrical light will always have the same apparent radius and falloff. To create a cylindrical light source, simply replace the `spotlight` keyword with the `cylinder` keyword. See the syntax below for further clarification.

```plaintext
light_source {
    <x,y,z>
    color C
    cylinder
    radius r
    falloff f
    tightness t
    point_at <a,b,c>
}
```

To better understand what a cylindrical light might look like, replace the `spotlight` keyword with the `cylinder` keyword in your light definition. You might notice that the scene appears much darker when you render it, so give your light source larger radius and fallout values. Try modifying the other parameters as well to see what kind of effects it has.

The Area Light

As you may have noticed, all of the lights we have used thus far create very sharp shadows. While we have some control over the falloff and tightness of the light, the shadows created by objects are very stark. This is because pointlights, spotlights, and cylindrical lights all originate from one point in space. Objects that are in plain view of the light will be completely illuminated, while objects that are hidden behind others or are outside of the illuminated area will be in full shadow. While this can create an interesting effect, it is not particularly realistic since in the real world light never emanates from a volumeless point.

To create more realistic lighting with softer shadows, we can use the area light. An area light can be thought of as a two dimensional array of pointlights. In other words, it is a rectangular area containing a number of pointlights that can be used to create the illusion of a single light with volume. Here is the most basic syntax for the area light:

```plaintext
light_source {
    <x,y,z>
    color C
    area_light <a,b,c>, <i,j,k>, num1, num2
    adaptive amount
    jitter
}
```
As with other light sources we have seen, the first vector \( <x,y,z> \) determines the location of the light source. Since the area light is a rectangle rather than a point, this vector defines where the center of the rectangle will lie. Again, the color keyword determines the light source's color.

The area_light keyword takes four parameters. The first two, \( <a,b,c> \) and \( <i,j,k> \), are vectors that determine the plane in which your rectangular area light lies. The lengths of these vectors determine the size of your area light. The variables \( num_1 \) and \( num_2 \) determine how many lights will be on each side of your area light. Consider the diagram below for further clarification:

![Diagram showing matrix of pointlights](image)

This image shows the matrix of pointlights that would be created with this area light definition:

```
area_light <4,0,0>, <0,0,3>, 5, 4
```

The first vector \( <4,0,0> \) declares that one side of the rectangle lies along the x-axis and has a length of 4 units. The second vector \( <0,0,3> \) declares that the other side lies along the z-axis and has a length of 3 units. The vectors used to describe the sides of an area light should always be perpendicular to each other, since we want to create a rectangle. The next variable 5 states that the first side (along the x-axis) should have 5 pointlights, and the last variable 4 states that the second side (along the z-axis) should have 4 pointlights. As you can see in the diagram, the resulting area light has a total of 20 pointlights, laid out in a 5x4 matrix.

If you define a rectangular area light that lies in the x-z plane as in the example above, but you decide to center the light at another point in space, say \( <10,10,-10> \), the area light will be translated such that it is centered at \( <10,10,-10> \) and will lie in a plane parallel to the x-z plane where it was originally defined. Therefore the vectors given in the area_light definition are used only to describe the orientation of the light source, and not its location in the scene.

The adaptive keyword is used to speed up the rendering of an scene with an area light. As you might imagine, using an area light significantly slows the rendering process. For our 4x5 matrix, for example, POVRay will need to send out 20 times more rays than if we had used a simple pointlight. Using the adaptive keyword tells POVRay to only send those rays needed to determine the color of a particular pixel, instead of sending one ray for each pointlight in the area light. Usually, an adaptive value of 1 is used, however higher values of 2 or 3 can also be used. The higher the adaptive value, the more rays are sent.

Finally, the optional jitter keyword is used to soften the shadow created by our area light. Without jitter, POVRay will create a number of sharp overlapping shadows, one for each pointlight in your area_light matrix. Using the jitter keyword tells POVRay to "jitter", or slightly move, the position of each light in the area light to create a truly soft shadow.

To see what your scene looks like when it is illuminated using an area light, replace your current light definition with the sample area light definition below, and render the scene. Be patient while you wait for the scene to render - even though we are using the adaptive keyword, rendering a scene with an area light will take much longer than one with a pointlight or spotlight!
In order to gain a better understanding of how the keywords used in the area_light actually work, try modifying the values of each keyword individually to see how it modifies the effect created by the light. Also, try rendering the scene without using the jitter keyword.

---

### Spotlights and Cylindrical Lights as Area Lights

An area light does not have to be made up of pointlights, as seen in the examples above. If you would like to create a spotlight or cylindrical light that also creates soft shadows, you can simply combine the syntax of the spotlight and area light as follows:

```plaintext
light_source {
    <x,y,z>
    color C
    spotlight
    radius r
    falloff f
    tightness t
    point_at <a,b,c>
    area_light <l,m,n>, <i,j,k>, num1, num2
    adaptive amount
    jitter
}
```

As you can see, this light source definition simply combines the keywords used for spotlights and cylindrical lights with the keywords used for area lights. Try adding a spotlight that is also an area light to your scene, and see what effect this creates.

---

### Conclusion

This is a long lesson and a number of new keywords have been introduced, but now you know most of what there is to know about lighting in POV-Ray. If you are finding all of these lighting definitions confusing, it may be useful to look at the compiled syntax of the light source object:
light_source {
    /* Required for all light sources */
    <x,y,z>
    color C

    /* Optional specifiers for a spotlight
    or cylindrical light*/
    type /* Either spotlight or cylinder */
    radius r
    falloff f
    tightness t
    point_at <a,b,c>

    /* Optional specifiers for an area light */
    area_light <l,m,n>, <i,j,k>, num1, num2
    adaptive amount
    jitter
}

All light sources must include a location and color definition. The simplest light source, the
pointlight, contains only these. Other more complicated light sources are made up of combinations
of the optional parameters: spotlights and cylindrical lights use only the first set of optional
parameters; area lights use only the second set of optional parameters; and spotlights and
cylindrical lights that are also area lights use all of the parameters.
#include "colors.inc"

camera {
    location <0, 5, -10>
    look_at <0, 0.5, 0>
}

light_source { <20, 20, -20> color White }

plane {
y, -1
    pigment { checker color White color Black }
}
sphere {
    <0.5, 0, 0>, 1
    pigment { color Red }
    finish {
        reflection 0.4
        phong 1
        phong_size 50
    }
}
cone {
    <2, 0, 2>, 1.5,
    <2, 3, 2>, 0.2
    pigment { color Violet }
}
torus {
    1.2, 0.5
    pigment { color Green }
    rotate <-40, -20, 0>
    translate <-2, 1.5, 1>
}
Lesson 6
Introduction to POV-Ray Textures

In POV-Ray, the word **texture** is used to encompass all options that describe the look of an object. It is comprised of three pieces: **pigment**, **finish**, and **normal**. As we have seen in previous lessons pigment can control the color of an object, however it can also describe a pattern of colors. Finish determines the surface properties of the object, such as its reflectiveness and its highlights. Normal allows you to modify the way light reflects and refracts off the surface of an object so that it appears bumpy. Mastering these texture options will greatly increase your ability to create interesting and realistic objects. Here is the texture syntax:

```
texture {
    pigment { myPigment }
    finish { myFinish }
    normal { myNormal }
}
```

Earlier in this tutorial, we defined pigments without placing them within a texture definition. This is a shortcut provided by the POV-Ray language, which allows you to define pigments, finishes, and normals without including the within a texture definition. However, it is important to understand that these three pieces make up a POV-Ray texture, and for clarity it can be helpful to include the texture statement.

Before you start making your own textures, it is important to understand that textures are three dimensional, and can be translated, rotated, and scaled in the same way as POV-Ray objects. A texture can be thought of as a massive three dimensional object that permeates all space. When a texture is applied to an object, that object is "carved" out of the texture at that location. For this reason, if you apply a texture to an object before translating it, the object will not appear the same as when you apply the texture after this transformation. It is therefore important maintain a consistent order of texturing and transformation. Usually, textures are applied to objects before transformation.

When performing a transformation on a texture there are a number of details to consider. In the simplest case the entire texture will be transformed, however it is also possible to transform one piece of the texture independently. Note that individual transformations can only be applied to the texture's pigment or normal -- a finish cannot be rotated, translated, or scaled. When performing these individual transformations, the order in which they are applied is very important. A transformation within the **texture** block affects all previously defined statements. Consider the following example:

```
texture {
    pigment { myPigment }
    scale x // affects the pigment only
    normal { myNormal }
    translate y // affects both the normal and the pigment
    finish { myFinish }
}
```
As you can see in the example above, transformation keywords can be placed in varying locations throughout the texture definition to create different effects. To make things even more complicated, transformations can be defined within the pigment and normal definitions themselves, as will be seen in the following lessons. For now, it is most important to understand the three-dimensional nature of textures and the importance of order when performing transformations on textures.

The following three lessons will cover pigments, finishes, and normals in depth:

Lesson 6.1: Pigment
Lesson 6.2: Finish
Lesson 6.3: Normal
Lesson 7
Defining Pigments in POV-Ray

Thus far, we have been using the pigment keyword to determine the surface color of our objects. A pigment, however, does not have to be a solid color. In fact, POV-Ray provides a large number of patterning options that can be used to create more complex, and more interesting, pigments.

Solid Colors

The simplest type of pigment is a solid color. As we have already seen, colors can be determined by using the predefined colors described in the "colors.inc" package as follows:

```
pigment { color color }
```

However, if you do not wish to be limited by the predefined colors provided in the POV-Ray package, you can also define your own colors using the RGB definition. RGB colors are defined by providing three parameters which determine the amount of Red, Green, and Blue in the color. Each value is a number between 0 and 1, where 0 is the lowest intensity and 1 is the highest intensity. See the following example:

```
pigment { rgb <r,g,b> }
```

If you would like to add transparency to your pigment, there are two options: add a fourth filter parameter or add a fourth transmit parameter to your RGB definition. Both parameters can have values between 0 and 1, where 0 represents a completely opaque pigment and 1 represents a completely transparent pigment. A filtered transparent pigment is one where the light that passes through an object is filters according to the surface color. A filter value of 1 means that the object is completely transparent, but still filters the light so that the pigment color is visible. A transmittance transparency, on the other hand, does not filter the light according to the pigment color and instead lets light pass through the object unchanged. A filtered pigment is defined as seen below, using the rgbf keyword:

```
pigment { rgbt <r,g,b,f> }
```
A transmitted pigment is defined as follows, using the rgbt keyword:

\[
\text{pigment \{ rgbt } \langle r, g, b, t \rangle \}\]

Finally, there is a shortcut available if you would like to add a filter value to a predefined color in the "colors.inc" package. The filter keyword can be added at the end of a color definition followed by a filter value as can be seen below:

\[
\text{pigment \{ color color filter value\}}
\]

To get a better understanding of these three methods for defining solid and transparent colors, try modifying the pigment of the sphere in the interactive right half of the window.

- Give the sphere an rgb color pigment.

\[
\text{Answer}
\]

\[
\text{pigment \{ rgb<0,1,0> }\]
\]

- Modify the pigment so that it has a filtered transparency.

\[
\text{Answer}
\]

\[
\text{pigment \{ rgbf<0,1,0,0.9> }\]
\]

- Modify the pigment again so that it has a transmittance transparency.

\[
\text{Answer}
\]

\[
\text{pigment \{ rgbt<0,1,0,0.9> }\]
\]

Color List Pigments

The simplest type of pigment pattern comes in the form of color list pigments. As you may have noticed in the code in the interactive section to the right, the plane in our scene has a checkered pigment. This is an example of a color list pigment. A color list pigment is defined by providing a pattern keyword followed by a list of the colors to be displayed in that pattern. There are a number of different color list patterns, as can be seen in the table below:
The *checker* pattern takes two colors and is defined as follows:

```plaintext
pigment {
  checker
  color color1
  color color2
}
```

The *hexagon* pattern takes three colors and is defined as follows:

```plaintext
pigment {
  hexagon
  color color1
  color color2
  color color3
}
```

The *brick* pattern takes two colors and is defined as follows:

```plaintext
pigment {
  brick
  color color1
  color color2
}
```

These three list pigments provide simple but useful patterns. Practice using color list pigments by applying each pattern to the sphere and box in code to the right.

**Color Map Patterns and the Gradient Pattern Type**

Color maps provide the most control over the appearance of a patterned pigment. Using a color map allows you to specify the colors that will appear in your pigment as well as the percentage of space they will occupy. Pattern Modifiers can be used to determine how the colors specified in the color map will blend together. For example the *gradient* modifier that we will look at in this section creates a striped pattern. The syntax for a color map pigment is as follows:

```plaintext
pigment {
  pattern_type
  pattern_modifier
  color_map {
    [control_point_0, color color0]
    [control_point_1, color color1]
    [control_point_2, color color2]
  }
}
```
Here, the variable `control_point_0` holds a double value specifying the control point for the first color in the color map. The control point values increase from 0.0 to 1.0 across the range of colors, usually with the first control point having a value of 0.0 and the last having a value of 1.0. A control point can be thought of as a marker stating that the color appearing at a particular point in the color map should have the value defined by `color color0`. The colors between control points are determined through linear interpolation. The control points can be used to specify both the order in which the pattern will use the colors and the percentage of space each color will have in the pattern. For example, if you have the following color map containing two colors:

```perl
color_map {
    [0.8, color Red]
    [1.0, color Blue]
}
```

80% of the pattern will be red and about 20% of the pattern will be blue, with some blending between the two colors. Notice that in this color map, the first defined color has a control point of 0.8 instead of 0.0. This means that the first 80% of the pattern will be solid red, without any blending, while the color in the last 20% will be determined by a linear interpolation between red and blue. If another color, say yellow, had been defined first with a control point of 0.0, the first 80% of the pattern would contain a blending between yellow and red instead of a solid color. While slightly confusing, in this way control points are very useful because they provide low-level control the pigment's appearance.

To create a striped pattern, we will add `gradient vector` in place of `pattern_type` in the above code. The `gradient` keyword states that we would like to create a striped pigment. The `vector` variable is a three component vector that determines the normal to the plane that the stripes lie in. In other words, the `vector` variable determines the direction of the stripes. At the moment we will not be using any pattern modifiers, so we can omit the `pattern_modifier` variable. Consider the following example:

```perl
pigment {
    gradient x
    color_map {
        [0.0, color Red]
        [0.33, color Yellow]
        [1.0, color Blue]
    }
    scale 0.6
}
```

In the image above, a striped pigment has been applied to the sphere and the box objects. Note that a transformation, `scale 0.6`, has been added at the end of the pigment definition. This is used
to create a pattern with more stripes. In this image, the scale transformation was applied to the box but not to the sphere. As you can see, only two iterations of the color map appear in the sphere while more appear in the box. This is a good example of a situation where transformations can be useful when defining textures.

To practice using color maps, try adding a pigment with a very large number of stripes and colors to the sphere object. Then, create a pigment for the box object that utilizes the color map's control points to make a pattern with unevenly distributed colors. Change the gradients of both pigments to see what effect it has.

Sphere pigment: **Answer**

Change the sphere's pigment definition to:

```plaintext
pigment {
  gradient y
  color_map {
    [0.0, color Red]
    [0.2, color Yellow]
    [0.4, color Blue]
    [0.6, color Violet]
    [0.8, color Orange]
    [1.0, color Green]
  }
  scale 0.15
}
```

Box pigment: **Answer**

Change the box's pigment definition to:

```plaintext
pigment {
  gradient <0,2,1>
  color_map {
    [0.5, color Red]
    [0.8, color Yellow]
    [1.0, color Blue]
  }
  scale 0.6
}
```

---

**The Frequency and Turbulence Warp Modifiers**

A pattern modifier is a statement that can be added to a pigment definition to modify the color map pattern in some way or another. Pattern modifiers are optional, but very useful when creating complex pigments. There are two particularly useful pattern modifiers, frequency and turbulence, which apply to most pattern types.
**Frequency**

The **frequency** modifier is used to determine the number of times a color map will repeat over each unit of size in the object. The default frequency value is 1. If a frequency value between 0 and 1 is used, the color map will be stretched, while a value greater than 1 will compress the color map. In the exercise above, the **frequency** modifier could have been used instead of the **scale** to create a pigment with a large number of thin stripes. As with all pattern modifiers, the **frequency** modifier appears between the pattern type definition and the color map definition, as can be seen in the syntax definition below:

```latex
pigment {
  pattern_type
  frequency value
  color_map {
    [control_point_0, color color0]
    [control_point_1, color color1]
    [control_point_2, color color2]
  }
}
```

Practice using the **frequency** pattern modifier by recreating your thin striped sphere pigment without using the **scale** transformation. Create a new **gradient** pigment for the box using a frequency value between 0 and 1 to see what a stretched pattern looks like.

**Sphere pigment: Answer**

Change the sphere's pigment definition to:

```latex
pigment {
  gradient y
  frequency 6
  color_map {
    [0.0, color Red]
    [0.2, color Yellow]
    [0.4, color Blue]
    [0.6, color Violet]
    [0.8, color Orange]
    [1.0, color Green]
  }
}
```

**Box pigment: Answer**

Change the box's pigment definition to:

```latex
pigment {
  gradient z
  frequency 0.8
}
```
Turbulence

The syntax for defining turbulence in POVRay has changed recently. Originally, turbulence could be defined simply by including the pattern modifier keyword \texttt{turbulence} followed by a turbulence value. However, with this definition turbulence could not be affected by transformations such as \texttt{rotate, scale, or translate}. The syntax for including turbulence in a pigment has therefore changed, and is now included inside a \texttt{warp} definition. This lesson will discuss the new syntax, but it is important to be aware of the previous syntax because much of the POVRay documentation available has not been updated to include this change.

The \texttt{turbulence} pattern modifier allows you to "mix up" the pattern to some degree in a random manner using a noise function called \texttt{DNoise}. It has three optional parameters defined by the keywords \texttt{octaves, lambda, and omega}. These parameters are not necessary, but they can provide additional control over the way in which turbulence is computed. The general syntax for a pigment that uses turbulence warp is as follows:

\begin{verbatim}
pigment {
  pattern_type
  warp {
    turbulence turb_value
    octaves oct_value
    lambda lam_value
    omega om_value
  }
  color_map {
    [control_point_0, color color0]
    [control_point_1, color color1]
    [control_point_2, color color2]
  }
}
\end{verbatim}

The \texttt{turbulence} keyword can be followed by either a double value or a vector. If double value is used, turbulence occurs equally in the x, y, and z directions. If a vector such as $\langle a, b, c \rangle$ is used, a turbulence value of $a$ will be applied in the x direction, a value of $b$ in the y direction, and a value of $c$ in the z direction.

To get a feel for the type of effects turbulence can have on a pigment, try adding a simple turbulence warp to the sphere's pigment, without worrying about the optional parameters \texttt{octaves, lambda, and omega}.

\textbf{Answer}

\begin{verbatim}
pigment {
  pattern_type
  warp {
    turbulence turb_value
    octaves oct_value
    lambda lam_value
    omega om_value
  }
  color_map {
    [control_point_0, color color0]
    [control_point_1, color color1]
    [control_point_2, color color2]
  }
}
\end{verbatim}
As you can see, including turbulence in a pigment definition creates a drastic and interesting effect. The \textit{DNoise} function used to create this turbulence determines the color at a certain point on the object's surface by taking "steps" away from the original to other nearby points in a random manner. Each successive step is shorter than the previous step, and the initial magnitude of these steps is determined by the turbulence value. This creates an effect where points that are close together have similar color values, while points that are far apart can be randomly different. The additional parameters provided for warp turbulence give you some control over this function.

- \textbf{octaves}: This parameter takes a float value in the range 1 to $< 10$. It determines the number of steps that the \textit{DNoise} function will take to calculate the color of each point. The default value is 6, and since the steps get successively smaller, increasing the \textit{octaves} value will not have a particularly drastic effect. Decreasing the value, however, can change the pigment significantly.

- \textbf{lambda}: This parameter determines how statistically different one random step taken during the \textit{DNoise} calculation is from the previous random step. The default value is 2. Smaller values will result in a less random pattern, while larger values will increase randomization.

- \textbf{omega}: This parameter takes a float value between 0 and 1, which determines how large one octave step is as compared to the previous step. The default value is 0.5, indicating that each successive step is half the size of the previous step.

To get a better feel for how these parameters work, try adding each one to your previous \textit{warp} definition separately. Try using both small and large values to see what effects it has on the pigment's appearance.

\textbf{More Pattern Types}

There are a number of different pattern types other than the \textit{gradient} pattern we have been using thus far. Most of these patterns can be tweaked the \textit{frequency} and \textit{turbulence} pattern modifiers, although in some cases this is purposefully forbidden. Each pattern type will be discussed briefly below.

The \textit{agate} pattern type creates a marble-like pigment. It is unique in that it contains its own build-in turbulence function. Additional turbulence can be added using the \textit{turbulence} pattern modifier, but this is usually not necessary. The agate pattern's build in turbulence can be controlled using a special pattern modifier called \textit{agate_turb}, which works exactly like \textit{turbulence}. 
The bozo pattern type creates smooth random pigment that varies greatly with different turbulence values. It responds to both the frequency and the turbulence pattern modifiers.

The granite pattern type creates a pigment that makes the object appear as if it was carved out of granite. It responds to both the frequency and the turbulence pattern modifiers. In fact, the granite pattern type has a default turbulence of 0, so without adding additional turbulence it will create a simple striped pattern much like gradient. Therefore turbulence should almost always be included.

The marble pattern type creates a pigment that makes the object appear as if it was carved out of marble. Like the granite pattern type, it has a default turbulence of 0, so additional turbulence should usually be included. As can be seen in the cube below, which has no turbulence, the blending pattern of marble is slightly different than that of gradient. Instead of creating a sharp transition between one iteration of the color map and the next, it blends them smoothly together.
The *onion* pattern type creates a pigment formed of concentric spheres each of unit length, much like a physical onion. This pattern responds to both the *frequency* and the *turbulence* modifiers.

The *wood* pattern type creates a pigment formed of concentric cylinders centered about the *z* axis. Intended to be used for creating wood-like pigments, this pattern can be used with a bit of turbulence to create realistic tree growth rings. It responds to both the *frequency* and the *turbulence* modifiers.

While the POVRay language does incorporate a few other pigment pattern types, this is an overview of the most useful patterns that should provide a wide range of patterning options. To gain some practice creating complex pigments, practice trying to make some realistic marble, granite, and wood textures.

Granite pigment: Answer
Marble pigment: Answer

```plaintext
color_map {
  [0.0, color Black]
  [0.2, color Gray20]
  [0.8, color Gray80]
  [1.0, color Black]
}
```

Wood pigment: Answer

```plaintext
color_map {
  [0.0, color MediumWood]
  [0.6, color DarkWood]
  [1.0, color MediumWood]
}
```
Interactive Content

```c
#include "colors.inc"

camera {
    location <0, 2,-5>
    look_at <0, 0.5, 0>
}

light_source { <20, 20, -20> color White }
lighLsource { <2,10,-10> color White spotlight radius 15 falloff 18 tightness 10 point_at <1,0,1> }

plane { y, -1 pigment { checker color White color Black } }

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

box { <-2.1, -1.5> <0-0.5, -0.5, -0.5> pigment { color Blue } rotate y*20 }
```

Render Reset
Lesson 8
Defining Finishes in POV-Ray

The finish component of a texture determines the surface properties of the object. In most cases, these properties control how light interacts with the surface of the object, for example whether it is reflective or flat, shiny or dull. This lesson discusses the keywords necessary to define complex surface finishes.

Ambient and Diffuse

The ambient and diffuse keywords are used to simulate ambient lighting in a scene. Ambient light is the light that partially illuminates the shadowed areas of a scene. As you may have noticed, the shadowed side of a sphere or box is not completely black, but rather darkened slightly. In reality, this ambient light is created by scattered light rays that bounce throughout a scene, eventually illuminating even those areas that are not directly lit by a light source. This effect would be difficult and costly to calculate using ray tracing, so the keywords ambient and diffuse have been incorporated into the POV-Ray scene description language to simulate this ambient lighting.

The ambient keyword is used to approximate ambient lighting by adding a small amount of white to the surface of an object so that it appears to glow. This ensures that the shadowed areas of the object still display some of their surface color. The default ambient value is 0.1, which is relatively small. It can have any value between 0.0 and 1.0. Its syntax is as follows:

```
finish { ambient value }
```

To get a feel for the effect that ambient has on the appearance of an object, try adding an ambient statement to the sphere's finish in the interactive section. First use an ambient value of 0.0 to see what the sphere looks like without an ambient lighting. Next change the value to 0.4, and the 1.0, to see how the sphere's appearance changes as you modify the ambient value.

Diffuse reflection is the light that originates from a light source and reflects in a scattered manner off the surface of an object. The majority of visible light results from diffuse reflection. The diffuse keyword controls how much light is reflected via diffuse reflection. The default diffuse value is 0.6, and while any number is a legal, usually a value between 0.0 and 1.0 is used. Another
keyword, **brilliance**, can be used to modify how the diffuse reflection is calculated. The amount of light diffused from the surface of an object is dependent on the angle with which it hits that surface. The **brilliance** keyword controls how light falls off the object's surface depending on the angle of intersection. The default **brilliance** value is 1.0, but any range of values can be used. Objects with higher brilliance values may appear more metallic, while objects with lower values may appear flatter. Generally, **brilliance** is more useful when including highlights, which will be discussed in the next section. The syntax for these two keywords is as follows:

```plaintext
finish {
    diffuse d_value
    brilliance b_value
}
```

To learn how the **diffuse** and **brilliance** keywords work in practice, remove the **ambient** keyword from your current **finish** definition. First try using **diffuse** alone, with a value of 0.1. This creates a much darker sphere. Now try a **diffuse** value of 0.9 to see what strong diffuse reflection looks like. After you have a feel for **diffuse**, try adding the **brilliance** keyword in as well. Give it first a value of 0.2, and then a value of 2 to see how the sphere's finish changes.

---

**Highlights**

Highlights are bright spots that are created by light reflecting off the surface of a smooth object. There are two different types of highlights that can be created in POV-Ray, phong highlights and specular highlights. The two models are very similar, but the specular model creates slightly more physically realistic highlights. Both can be extremely useful depending on the context, however, so be sure to try out both models.

**Phong Highlights**

There are two keywords related to phong highlights, the **phong** keyword which controls the brightness of the phong highlight, and the **phong_size** keyword that determines the size of the phong highlight. The syntax is as follows:

```plaintext
finish {
    phong value
    phong_size size
}
```

The default **phong** value is 0.0, indicating no highlight, but it can hold any value in the range 0.0 to 1.0, where 1.0 signifies complete reflection of the light source's color. The **phong_size** keyword can hold any value, but it usually ranges between 1 and 350. Finishes with smaller phong sizes produce duller-looking objects, while larger phong sizes produce highly polished objects. The default **phong_size** value is 40, which simulates the highlight of a plastic material.
Specular Highlights

The specular model creates slightly brighter highlights than the phong model, and can therefore be most useful when creating metal and glass finishes. As with phong highlighting, specular highlighting has two associated keywords, `specular` which controls the brightness of the highlight and `roughness` which controls the size of the highlight. The syntax is as follows.

```plaintext
finish {
    specular value
    roughness size
}
```

The default `specular` value is 0.0, indicating no highlight, but again it can hold any value in the range 0.0 to 1.0, where 1.0 signifies complete saturation at the center of the highlight. The `roughness` keyword typically holds a value between 1.0, which creates a very rough highlight, and 0.0005, which creates a smooth highlight. The default value is 0.5, which simulates the highlight of a plastic material.

Practice using highlights by adding first a phong highlight and then a specular highlight to your sphere. Try some different brightness and size values for each to see what effects it has.

Phong highlight: **Answer**

```plaintext
finish {
    phong 0.3
    phong_size 70
}
```

Specular highlight: **Answer**

```plaintext
finish {
    specular 0.3
    roughness 0.01
}
```

There is an additional keyword that can be added to the `finish` definition when either a phong highlight or a specular highlight is specified, `metallic`. The `metallic` keyword modifies the way in which the highlight is calculated to more closely represent the reflective properties of metal. Instead of simply reflecting the color of the light source, the highlight also takes some color from the surface pigment.

Try adding the `metallic` keyword to your finish definition to see what effects it has.

Metallic highlight: **Answer**
Specular Reflection

Specular reflection occurs when a surface reflects light perfectly instead of diffusing it. The resulting surface reflects other objects in the scene much like a mirror. Reflection is controlled by the `reflection {...}` block, and contains an optional keyword `metallic` which produces a reflection more typical of metals. The syntax is as follows:

```plaintext
finish {
  reflection {
    value
    metallic /* Optional */
  }
}
```

The `reflection` value can range from 0.0, which signifies no reflection, to 1.0, which signifies complete reflection. By default there is no reflection.

Try adding some reflection to your sphere. Practice using a few different reflection values to see what the effects are.

Reflective finish: **Answer**

```plaintext
finish {
  phong 0.3
  phong_size 30
  reflection {
    0.3
  }
}
```

Iridescence

Iridescence is the effect that occurs when a thin reflective layer lines on the surface of an object. For example, the rainbow reflections of an oily puddle or the colored reflections of the surface of a bubble. Iridescence is controlled by the `irid {...}` block, which contains two optional parameters `thickness` and `turbulence`. The syntax is as follows:
The \textit{irid} variable determines the level to which iridescence will affect the color of the object's surface. It can hold any value between 0.0 (no effect) to 1.0 (full effect). The optional keyword \textit{thickness} determines the thickness of the iridescent layer. It can hold any value, but usually a value between 0.25 and 1.0 works best. The \textit{turbulence} keyword can be used to modify the thickness slightly, and usually has values between 0.25 and 1.0 as well.

Try adding iridescence to your sphere's \textit{finish} statement, finding the values that create the most "realistic" iridescent layer (since solid objects are not usually iridescent, nothing will look too realistic).

Iridescent finish: \textbf{Answer}

```
finish {
    irid {
        irid_value
        thickness thickness /* Optional */
        turbulence turbulence /* Optional */
    }
}
```

\textbf{Refraction}

Refraction is the bending that occurs when light passes through a transparent object such as a glass. Different substances bend, or refract, light differently depending on their "index of refraction". The index of refraction is dependent on a number of factors, including the density and thickness of the object. In POVRay, refraction is controlled through two statements. First, the \textit{refraction} keyword in the \textit{finish} block determines whether or not the object has refraction. It should have a value of either 0.0 (refraction turned off) or 1.0 (refraction turned on). Second, a separate \textit{interior} block outside of the \textit{finish} block is used to control the properties of the interior of the object. This \textit{interior} block contains a keyword \textit{ior}, which determines the desired index of refraction. The syntax for a transparent sphere with refraction is as follows. Note that the \textit{pigment} definition includes a \textit{filter} statement to make the sphere transparent.
sphere {
  <x,y,z>, radius
  pigment { color mycolor filter value }
  finish {
    refraction 1.0
  }
  interior {
    ior 1.5
  }
}

The ior value is usually between 1.0 (no refraction) and 3.0. The index of refraction of air is 1.0, of water is 1.33, of glass is 1.5, and of diamond is 2.4.

Try modifying your sphere definition to create a transparent pigment with refraction.

Glass sphere: Answer

```plaintext
sphere {
  <0.7, 1.5, 0>, 1.2
  pigment { color White filter 0.8 }
  finish {
    refraction 1.0
  }
  interior {
    ior 1.5
  }
}
```
Interactive Content

```c
#include "colors.inc"

camera {
  location <0, 2, -5>
  look_at <0, 1, 0>
}

light_source {<-10, 10, -10> color White }
light_source {
  <-10,10,-10>
  color White
  spotlight
  radius 15
  falloff 18
  tightness 10
  point_at <1,0,1>
}

plane {
  y, -1
  pigment { checker color Violet color White }
}

sphere {
  <0.7, 1.5, 0>, 1.2
  pigment { color Aquamarine }
  finish {} 
}
```

Render  Reset
The normal statement can be used to modify the surface normals of an object to make it appear textured. While the physical geometry of the object is not changed, the normal modification controls the way light reflects off its surface, therefore changing its appearance.

Overview

It can be geometrically difficult to model complex surfaces such as bumps and ripples, but with ray tracing we can simulate the appearance of these types of surfaces by modifying the way light reflects off them. The light reflection calculation at a particular point on the surface of an object is dependent on the surface normal at that point. A surface normal is a vector that is perpendicular to the surface of an object at a particular point. By modifying this surface normal at different locations on the surface of an object, we can change the way that light interacts with its surface and thus create textures that appear complex.

In recent versions of POV-Ray (since 3.0), any pattern can be applied to a pigment or a normal block, while originally patterns could only apply to one or the other. Therefore, any of the patterns discussed in Lesson 6.1 (excluding the list pigments checker, hexagon, and brick) can also be used as normal patterns. To avoid repetition, only the new patterns that usually pertain to normal statements will be discussed here.

Normal Syntax and Patterns

The normal block has a relatively simple syntax:

```
normal {
    pattern amount
    pattern_modifiers /* Optional */
    transformations /* Optional */
}
```

There are a number of different patterns that can be used in the normal statement. Each pattern
can be followed by an optional amount parameter, usually lying in the range 0.0 to 2.0, which specifies the depth of the pattern's bumps. Below each of the patterns will be described briefly:

The bumps pattern creates a bumpy surface. The normal statement for this object was defined as follows:

```
normal {
    bumps 1
    scale 0.1
}
```

The dents pattern creates an indented surface. The normal statement for this object was defined as follows:

```
normal {
    dents 1.2
    scale 0.2
}
```

The wrinkles pattern creates a chapped looking surface. The normal statement for this object was defined as follows:

```
normal {
    wrinkles 0.8
    scale 0.5
}
```

The ripples pattern creates a rippled watery surface. The normal statement for this object was defined as follows:

```
normal {
    ripples 1.2
    scale 0.2
}
```

The waves pattern creates a wavy looking surface. The normal statement for this object was defined as follows:

```
normal {
    waves 0.8
    scale 0.15
}
```
Try adding a `normal` statement to your sphere and test out some of the patterns above. How do the patterns change as you modify their parameter values?

---

**Pattern Modifiers**

The `turbulence` pattern modifier may be applied to `normal` statements just as with `pigment` statements. The syntax is shown below:

```plaintext
normal {
    pattern amount
    turbulence value
}
```

Try adding some turbulence to your `normal` statement.
Interactive Content

#include "colors.inc"

camera {
    location <0, 2, -5>
    look_at <0, 1, 0>
}

light_source { <-10, 10, -10> color White }
light_source {
    <-10,10,-10>
    color White
    spotlight
    radius 15
    falloff 18
    tightness 10
    point_at <1,0,1>
}

background { color Violet }

plane {
    y = -1
    pigment { checker color Firebrick color White }
}

sphere {
    <0.7, 1.5, 0>, 1.2
    pigment { color Aquamarine }
    finish {
        phong 0.5
    }
    normal {}
CHAPTER 7
Conclusions and Future Work

While the process of developing POV-IT was long and its implementation changed drastically over the course of the semester, I would argue that the resulting web application achieved most of the goals set out at the beginning of the project. I was able to successfully implement an online interactive tutorial that allows users to practice writing POV-Ray code as they learn new concepts and syntax. I also created a dynamic user interface that utilized the capabilities of modern web development tools such as Ajax, Ruby on Rails, and Cascading Style Sheets. The result is an in-depth tutorial teaching the beginning concepts of the POV-Ray scene description language formed from a combination of textual explanations, concise reference descriptions, and interactivity. The tutorial serves to accurately describe the functionality of the language, even for those users without extensive programming experience.

Although POV-IT achieved most of the goals originally set out for the project, there are still many improvements that could be made to the system. Ideally, I would have liked to include many more lessons covering the more advanced features of POV-Ray such as isosurfaces, lathe objects, and radiosity. While POV-IT provides a substantial introduction to the POV-Ray scene description language, it hardly serves as a replacement for the disorganized documentation provided at povray.org since so many topics are not discussed in the tutorial. Inspiring beginning users to start using POV-Ray is one of the central goals of this tutorial, but not only the beginning users are suffering
from a lack of accessible documentation. I would therefore like POV-IT to be able to provide documentation for more advanced features in the future.

While it would be great if I had the time to generate a new group of advanced lessons to include in POV-IT, it is likely that I will not be able to complete these lessons, especially because my knowledge of the advanced POV-Ray features is not particularly strong. One way to remedy this situation would be to provide a way for other POV-Ray users to generate advanced lessons to be included in POV-IT. For example, if one user is very good at creating realistic water with POV-Ray, she could write her own interactive lesson describing this process that could then be included in the POV-IT advanced lessons. I would then serve as an administrator and editor of the new lessons submitted by users to make sure that they were clear and fitting to be included in POV-IT. This option could greatly expand the number of topics discussed in the POV-Ray Interactive Tutorial, making it a more useful resource for a large variety of users.

One of the main problems with the POV-IT system that could eventually harm its ability to reach a large number of users is the length of time that it takes for an image to render in POV-Ray. As a ray tracer, POV-Ray needs to trace thousands of light rays each time it renders an image to be displayed in the interactive section for one user. While this is not a significant issue when only a few users are rendering images on the server simultaneously, if POV-IT becomes a popular tool for learning POV-Ray the server may eventually grind to a halt due to the number of lengthy render requests. One way to solve this problem would be to allow users to render code on the client-side if they have the proper version of POV-Ray installed. If a user notices that it is taking an unusually long time for the server to render his code, he could switch to client-side rendering mode and
continue to interact with the system at a much faster rate. While this would potentially be a rather difficult feature to implement, it could greatly improve the experience of POV-IT users.

While there are a number of improvements that could be made to the POV-IT system and many lessons that could be added to improve its value to a larger variety of users, the tutorial is a useful addition to the available POV-Ray documentation in its current state. I hope to publicize POV-IT by sending the link to the POV-Team so that it may be included on the POV-Ray links page and also by posting the link on the POV-Ray forum. In time, I hope that POV-Ray users will come to enjoy the tutorial as much as I have enjoyed writing it.