RADIANCE

User's Manual

DRAFT

Lighting Systems Research Group
Lawrence Berkeley Laboratory
Berkeley, California

Cindy Larson
February 1, 1991

Acknowledgements

Greg Ward	Program Manager Principal Author
Francis Rubinstein	Program Manager
Sam Berman	Principal Investigator
Rudy Verderber	Group Leader
Steve Selkowitz	Program Leader
Robert Clear	Technical Advisor
Charles Ehrlich	Technical Advisor
Paul Heckbert	Technical Advisor
Cindy Larson	Documentation
Anat Grynberg	Program Validation
Jennifer Schuman	CAD Interfaces
Ning Zhang	CAD Interfaces

This work was supported by the Assistant Secretary of Conservation and Renewable Energy, Office of Building Energy Research and Development, Buildings

Equipment Division of the U.S. Department of Energy
under Contract No. DE-AC-03-76SF00098.

Introduction to Radiance

Radiance is a software package for accurately calculating and displaying lighting. The program takes a scene description with light sources, sun, sky, buildings, rooms, furniture, etc. and produces spectral radiance values which can be collected in a "photo-accurate" color image. As a lighting design tool, Radiance represents a significant advance in the state of the art. Since Radiance is a research tool, it lacks many of the user-friendly features found in commercial software packages. However, Radiance is more versatile than other lighting simulations and faster than other ray tracing programs, with some important capabilities not found in either.

By simulating the behavior of light, Radiance calculates radiance (or luminance) and predicts the appearance of any geometrically described scene, usually an architectural space. The software fulfills the traditional role of a renderer, except that this renderer provides an accurate simulation of lighting from lamp photometric data and advanced surface reflectance models which can correctly account for both diffuse and specular interreflection in complicated spaces. Because the calculations are accurate, they can be used for making design decisions, and the images produced by Radiance provide a client with a genuine preview of how the design will appear when it is completed.

Radiance was written for lighting designers who want to produce energy efficient designs that require some innovation, and are hampered by the limitations of conventional methods. When designers want to use fixtures with unusual distributions, non-standard fixture placements, indirect lighting, task lighting, or daylighting, they currently have to draw solely on their own resources and experience to create a good solution. Radiance allows lighting designers, engineers, architects, and their clients to visualize their solutions to these more difficult problems during the design phase, and to try out alternative designs and novel approaches without risk.

You might say "This sounds great, but why not just use conventional CAD rendering systems?" The problem with current computer aided drafting (CAD) software is that it produces images that are not predictive of lighting. For example, a few CAD systems will calculate

shadows from a small number of light sources, but they will not consider light source distributions or interreflections between objects. Commercial lighting programs, on the other hand, are incapable of simulating realistic spaces in their full detail. Some of the more advanced lighting programs calculate interreflection only in limited circumstances such as empty rectangular spaces, and they do not account for obstructions (such as partitions), color, and the effect of non-Lambertian surfaces (such as metal and glass). Since all the subtleties of lighting and reflections the viewer normally associates with real scenes are present in the final images, there can be a very strong psychological impact when viewing a computer graphic image generated by Radiance.

Lighting simulation is typically a two or three step process. The first step is describing the geometry, which is usually done within a CAD system. Note that this step may already have been carried out by the architect, so only a small extra effort is required on the part of the lighting designer. The second step, if it was not included in the geometric description, is the addition of materials and fixtures from libraries and manufacturer's catalogs to complete the model for the simulation software. The third step is running the lighting simulation program and evaluating the output. As a result of this evaluation, the designer will probably return to step two and, in some cases, will go all the way back to step one for another iteration. This process continues until the designer and client are satisfied that the choice and layout of the lighting system will provide the level and quality of illumination desired for the space.

Radiance has been compared to scale model measurements and different lighting calculation programs in validation studies and comparisons of empty rooms and diffuse surfaces that show good correlation with measured data and conventional lighting simulations. The program's accuracy has also been verified for unempty spaces, but more work is needed in the validation of non-Lambertian surfaces. Radiance is capable of going beyond the limitations of other programs and the simple cases presented in this documentation.

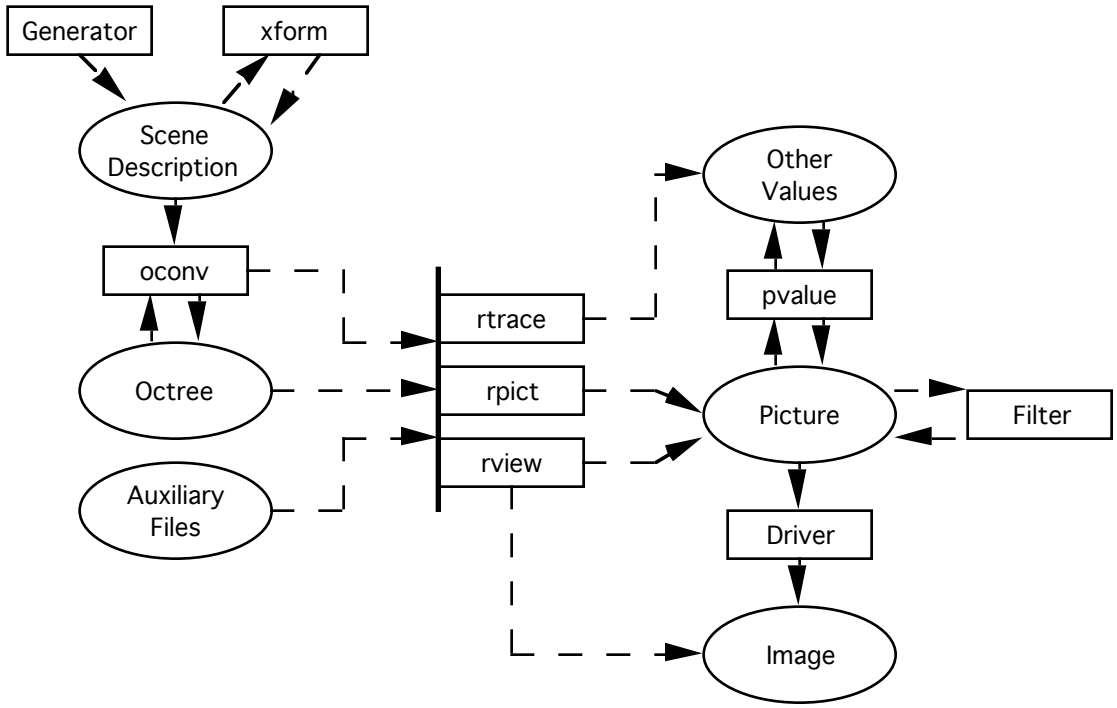
Radiance Capabilities

Radiance uses the simulation technique called image-oriented ray tracing, which is particularly well-suited to computer graphics and

visualization. The path of light is tracked from its presumed destination to one or more sources, taking into account specular reflection, transmission, and virtually any geometry. Following light paths in reverse is a much more efficient approach than tracing them from the light sources, because only a minute percentage of the photons that are emitted enter a viewer's eyes. If light were followed from the sources, most of the calculation would be wasted on rays that do not contribute to the desired image.

The input to the program is the scene geometry, which describes the location and shape of every surface, and the materials, which describe how light interacts with each surface. Rays are followed from the viewpoint into the scene, and then are traced to other surfaces and light sources to calculate luminance. In this way, ray tracing calculates luminance directly, which is ideal for the visualization of illuminated spaces since images are just collections of luminance values. Lighting calculation based on illuminance must convert to luminance prior to display, which results in a significant loss of information. Recent advances in ray tracing techniques have resulted in a complete lighting calculation that incorporates diffuse interreflection and daylight. Since the quality of the images created by Radiance depends on the number of rays traced, a large number of operations must be performed to produce a high-quality image.

The three main Radiance programs use ray tracing to calculate luminance and then (1) display images interactively with RVIEW, or (2) produce picture files in batch mode with RPICT, or (3) compute specific values for other purposes with RTRACE. Numerous other programs provide "filtering" (translation) within and between various formats, image processing and display functions, procedural object generation, light distribution calculation, and so forth.



The Radiance system allows the user to describe a scene and generate images based on three basic surface types: polygons, spheres, and cones. From these primitive shapes, compound surfaces of arbitrary complexity can be constructed and then manipulated and displayed. Through a process called instancing, hierarchical scenes containing millions of surfaces have been constructed. The interface to Radiance is command-based for maximum flexibility. Input files are created by CAD programs, text editors, and object generator programs. There is a library of useful generators to make prisms, patches, and so on, and objects such as furniture and light sources are provided. The UNIX utility can be used to automate scene creation and rendering, and several other bundled programs simplify the creation of custom generators. Import programs for a few popular CAD systems are provided.

Basic Radiance material types include composite, metal, glass, and self-luminous surfaces. Each material type describes the basic interaction of light with a surface, and variable parameters determine things such as color, polish, refractive index, and intensity. To these materials one can add procedural and scanned textures and patterns that add local variations to the surface orientation, color or intensity. By increasing the realism of the

reflection model in this way, the viewer gets a much better feel for the lighting present in a space.

The Radiance system is composed of a few dozen C programs that have been compiled and run on DEC and Sun workstations, Apple Mac II's (under A/UX), CRAY's, and a number of other UNIX machines. The software was developed at Lawrence Berkeley Laboratory (LBL), and has been in use since 1987 by the Architecture Department at the University of California at Berkeley, which augments its computer modeling courses with rendering and simulation.

The computer aided drafting (CAD) system used most frequently to produce geometric descriptions for Radiance is GDS (Graphical Design System) from McDonnell Douglas, because it happens to be installed on the UCB Architecture Department machines. Although it takes little effort to write translators from other CAD formats, limited access to these systems has curtailed LBL's translator development.

System Requirements

Radiance is used by architectural and engineering firms that have the necessary computer equipment and expertise, and a desire to produce better designs. Radiance is best suited for color UNIX workstations, but it will work on any machine running the UNIX operating system including IBM's running AIX, and Macintoshes running A/UX. Ideally, a computer system for the prediction of lighting should consist of a color workstation with a pointer device such as a tablet or mouse, optional input devices such as scanners or frame grabbers for obtaining material properties and textures, and optional output devices such as color printers or film recorders for recording simulation results. The platform must have enough memory and computing power to provide good interactive response time and reasonable turnaround of large batch jobs. The system should have access to large amounts of secondary storage such as bulletin boards and CD-ROM catalogs to aid in the development of building descriptions.

Radiance includes C source code for compiling on BSD or AT&T UNIX platforms, and has been compiled for the following hardware platforms:

- Sun 3, Sun 4 workstations
- DECstation running ULTRIX
- Silicon Graphics IRIS
- Mac II running A/UX

Radiance currently supports the following graphics interfaces:

- X11 8-bit color or greyscale and 24-bit color displays
- X10 8-bit color or greyscale displays
- SunView 8-bit color or greyscale
- NeWS color or greyscale
- AED 512 color graphics terminal

Radiance comes with translators for the following file types:

- GDS Things File
- IES Luminaire Data
- Sun 8 and 24-bit Rasterfiles
- Architrion Text File
- AutoCAD DXF (next release)
- Targa 8, 16, 24 and 32-bit images

Using this Manual

Since the Radiance system requires complete scene descriptions before images can be generated and then manipulated, this manual follows this same basic organization:

Scene Descriptions
Image Generation (Rendering)
Image Manipulation
Advanced Topics

Readers who are already familiar with the basic capabilities of Radiance and would like to learn more about using more sophisticated features will find examples throughout this document, and may benefit from a special section devoted entirely to advanced topics (such as auxiliary files, simulation options, and animation).

Tutorial examples have been provided at the end of the Scene Description, Image Generation, and Image Manipulation sections for readers who are interested in getting started with Radiance.

Glossary, Terms, and Programs sections are included at the end of this document to help the reader understand some of the technical terminology used in this document, as well as some of the basic concepts required for understanding ray tracing in general and Radiance in particular. Since this user's manual does not contain detailed descriptions of how to use the various hardware platforms, operating systems, graphics interfaces, and translators, it is expected that readers already have a working knowledge of the systems they are using to run Radiance.

It is recommended that readers use the Radiance Reference Manual as a complete guide to Radiance commands and input primitives, since this user's manual does not include descriptions of all commands and primitives.

Radiance Installation

Getting Started

The most current release of the Radiance synthetic imaging system is the fourth release, Version 1.3, and it includes all source files.

You may want to start by reading the basic Radiance Reference Manual documentation contained in "doc/ray.1". Use troff with the "-ms" macro package. Individual manual pages may be found in the subdirectory "doc/man1". Use the "-man" macro package for these documents. If you have a question, the answer is probably in one of these manuals, but you must read everything very carefully. Also, read the file called "doc/notes/ReleaseNotes".

The executables in bin.sun3 should be installed on all Sun 3 machines, those in bin.sun4 on Sun 4 machines, bin.IRIS on SGI Iris, and bin.DEC on DECstations, and bin.mac on MacIntoshes running A/UX 2.0. Not all of the programs in the source directory have been compiled on every machine, and not all are documented in the manual pages. In general, the documentation of the source is non-existent. If the binaries for your hardware are not present, you will have to compile the programs yourself; check the Makefiles carefully before compiling. The X drivers in this distribution work with X10R4 and X11Rany.

The files in the "lib" subdirectory should be installed in "/usr/local/lib/ray". If this location is different on your system, you will have to define RAYPATH to override the default, or change the LIBDIR variable in "src/rt/Makefile" and rebuild the renderers. RAYPATH is an environment variable, similar to PATH, that tells the programs where and in what order to look for auxiliary files. The default RAYPATH in this distribution is ":/usr/local/lib/ray" which searches in the current directory, then "/usr/local/lib/ray".

The first thing you should do is run the script in this directory called "RUNME" that requests you to provide some information about your use of Radiance. It will take only a moment to provide some basic information from and e-mail it back.

Good luck!

Scene Descriptions

Introduction

The first step in creating any Radiance images is to describe the scene you are designing using three basic geometric shapes: polygons, spheres, and cones. You can choose to either describe your scene mathematically for Radiance, or import this information to Radiance from a CAD system. It is most efficient to use a CAD system to create the descriptions of building and room geometries used by Radiance, if you have access to a CAD system capable of describing your scene. CAD descriptions are translated into Radiance input with the addition of material and light source descriptions, which you will need to define. Radiance can then be used to quickly and interactively view the scene from different perspectives, and can render high-quality images which can be displayed on the screen or on hardcopy devices such as film recorders and color printers for client evaluations and presentations.

The best way to get started with Radiance is to envision a scene you would like to describe, and either use a CAD system to sketch it out in three-dimensions, or start jotting down some notes on paper. The minimum requirements for creating a Radiance scene are a light source, a type of material, a surface, and a view. As you read the examples and tutorial example in this manual, think about your own scene and what you'll need to do to describe it for Radiance in terms of input files, materials, surfaces, textures, patterns, and light sources. The best way to appreciate Radiance's capabilities is to see how it can be used to bring your ideas to life!

Basic Format of Input Files

Input to the Radiance program comes from one or more *object description files*. These files specify the location, size, shape, and makeup of the objects and sources in the scene. The object descriptions can be created directly with a text editor, although this process is time consuming. Object libraries and object generators make the work go faster, but a three-dimensional graphics editor or CAD system is really needed to make the input process user-

friendly. It's a good idea to understand how to read and understand Radiance input files in any event, since even scenes developed on CAD systems will require some additional work (such as defining materials and light sources) before they can be used by Radiance.

Primitives are the basic building blocks of Radiance input files, which combine together to make up the scene description. All the materials you plan to use in your scene will need to be defined as primitives, according to a specific input format, and all the surfaces that describe objects in your scene also need to be defined as primitives. It helps to start thinking in advance about how you plan to organize your primitives together into Radiance input files, since good organization early on when you describe the scene can make the task of generating images much easier later on. For example, you might decide to put all the material definitions together in a file called "materials", so you can easily find the materials you've defined together in one place when you decide to add, change, or delete materials from your scene description.

A Radiance scene description file lists the materials and surfaces that are used to create a specific scene. Radiance scene descriptions represent the scene in three-dimensional Cartesian (x,y,z rectilinear) coordinates. The scene description file is stored in ascii text, according to the format shown here. For a more complete description of Radiance primitives, refer to the Radiance Reference Manual.

Primitive Format
modifier type identifier *n S1 S2 S3 ... Sn* 0 *m R1 R2 R3 ... Rm*

Primitive Example
void **plastic** fireplace_stone 0 0 5 .6 .05 .02 0 0

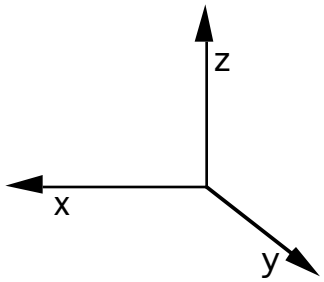
Modifiers are either the word "void", which indicates no modifier, or some previously defined primitive identifier. (The most recent definition of a modifier is the one used, and later definitions do not cause relinking of loaded primitives. Thus, the same identifier can be used repeatedly, and each new definition will apply to the primitives following it.)

The type can be the material type (eg: plastic, metal, glass) or surface type (eg: sphere, polygon, cone, cylinder), as well as one of a few other type categories (pattern, texture, or mixture), and the identifier is the name of the primitive being described.

String arguments (words) are listed next (S1,S2,S3,...Sn) for variable names, file names, or transformations being used, preceeded by the number (n) of arguments there are. No integer arguments are yet being used by Radiance, so the next line is always zero. Real arguments are always numeric (eg: red, blue, green, specularity, roughness), and are listed last (R1,R2,R3,...Rm), preceeded by the number (m) of real arguments to be listed.

Note: Coordinate Systems & Units

Radiance uses a right-handed coordinate system, which means this is the only real restriction on the definition of world coordinates.



Dimensions are in whatever units the user desires. The units can be thought of as meters, inches, feet, microns, or anything else, but they must be used consistently within a given scene. It is also important to keep to reasonable values (ie: between 10^{-5} and 10^{8} in magnitude) to avoid calculation anomolies. However, it will be much easier to use other Radiance programs if a few conventions are followed.

First, the z-vector should be the pointing in the zenith (up) direction. Second, if there is a North direction, it should be aligned with the y-axis if possible (which implies that the x-axis would point East). Finally, when creating a separate object file, it is best to place the origin at the most logical point for rotation, and explain the dimensions and units in a comment at the beginning of the file.

Comments are preceeded by a pound sign, '#', and continue to the end of the line.

Aliases can be defined for modifiers (material, pattern, or texture) that have the same characteristics as another (such as the materials xzbrick and yzbrick, which are both the exact same brick color, specularity, and roughness). Aliases relate identifiers that share identical attributes but are identified by different names and may have different modifiers, and can be used to organize identifiers referring to the same thing in different places. It's important to remember to fully specify the required data in the primitive for the reference part of the alias, so there will be some data to be referred to somewhere (and not just an alias primitive).

Alias Format

 # Comment

modifier **alias** *identifier reference*

Alias Example

```
# Cabin porch material definition
# (the same as fireplace material)

void alias yzbrick
      xzbrick
```

Inline commands are useful for quickly and easily generating complicated objects with many surfaces (instead of manually determining and specifying every vertex), and placing objects correctly within the scene, among other things. Inline commands must begin with an exclamation point, '!', in order to be identified as commands. These commands are executed by the shell, and their output is read as input to the program. The command must not try to read from its standard input, or confusion will result. A command may be continued over multiple lines using a backslash, '\', to escape the newline.

Command Format

 # Comment

!command

The formats for Radiance primitives may look a bit strange at first, but will become familiar to you as you gain experience using them.

Materials

When the scene geometry is being described, the user must be prepared to assign material names to each surface. The definition of these materials will determine how each surface interacts with light. Radiance provides several material types for this purpose; the

four basic classes of surface types supported are: light, normal, dielectric, and BRDF (bidirectional reflectance distribution function).

Light	The basic type of emissive surfaces source is simply called ***light***. Variations are provided for efficient modeling of spotlights, called ***spotlight***; secondary emitters, called ***illum***; and weak sources, called ***glow***. A light source can be a polygon, a sphere, a disk, or a source.
Normal	A normal surface is typically ***metal*** or ***plastic***. It has a diffuse and specular component, and a color. A roughness factor is also given. If the material is purely specular and has a roughness of zero, it is a mirror. If the material is purely diffuse, it is Lambertian. Most real materials are neither purely specular nor purely diffuse. Metal differs from plastic in that its specular component is influenced by the metal's color. A variation called ***trans*** is translucent.
Dielectric	A ***dielectric*** material, such as crystal or water, has an index of refraction and a spectral absorbence. Snell's laws and Fresnel's equations are used to compute the reflected and transmitted components and directions. A variation of dielectric called ***glass*** is optimized for efficient computation of thin glass surfaces such as windows, and another variation used to describe the surface between two dielectric materials (such as between crystal and water) is called ***interface***.
BRDF	Materials with arbitrary bidirectional reflectance distribution functions (BRDF's) can be modeled as one of four types. ***Plasfunc*** is used for a plastic-like material (white highlights) whose specular BRDF can be described as a function. ***Metfunc*** is identical except that the highlights will be given the material's color. Likewise, ***plasdata*** and ***metdata*** are used for BRDF's with white and colored highlights, based on data file input.

Plastic

```
# This defines a plastic type of
# material called red_plastic
# with Red=.7, Blue=.06,
# Green=.08, Specularity=.05
# and Roughness=.005

void plastic red_plastic
0
0
5  .7  .06  .08  .05  .005
```

Examples of material name assignments to Radiance are given for a type of plastic called "red_plastic", and a type of glass called "window_glass". Once a material name has been defined to Radiance, it may be used as many times as it's needed in the scene description by simply using the material name as a modifier to a surface. The number of parameters required for each material definition should always immediately preceed those parameters; the number five (5) shows up on the plastic example, right before the five required real parameters, and the number three (3) appears immediately before the three color transmission parameters in the glass example.

Glass

```
# This defines a glass type of
# material called
# window_glass with Red
# transmission=.96, Green
# transmission=.96, and
# Blue transmission=.96

void glass window_glass
0
0
3  .96  .96  .96
```

Any of the above materials can be modified by a set of textures and patterns to simulate a broad range of light interactions. For example, plastic can take on the appearance of any paint, paper, ceramic, or wood simply by varying parameters and applying optional textures and patterns. A texture modifies the surface normal direction. A pattern modifies the surface color. Textures and patterns are defined as functions of the surface normal, intersection point, and ray direction. The functions are stored in separate files which are read in and interpreted as necessary. Source distributions are implemented as light patterns, for example (please refer to the section on light sources).

Surfaces

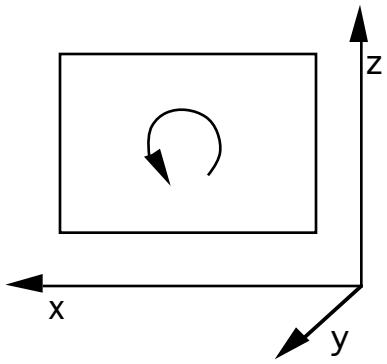
At the lowest level, Radiance models *polygons*, *spheres* and *cones*. From these basic boundary representations (surfaces), varied and complicated shapes can be produced. An *object* is a collection of one or more surfaces contained in a seperate description file. In addition to these geometric entities, there is a special type for distant sources, called *source*. Surface normals need to be considered when describing objects, since their description to Radiance defines how the object is viewed (where the "front" of the

object is). The "front" of an object is typically the side it is viewed from.

It's also very important to follow the right-hand rule to put vertices in the proper order on polygons, since mistakes can create very unusual and undesirable self-intersecting polygons that look something like bowties!

Note: Right Hand Rule

A method for determining or specifying the surface normal direction (the thumb) from the curved direction of vertices across the other two dimensions of space (our fingers). In the following illustration, vertices entered in the circular direction shown (counter-clockwise) would result in a surface normal pointing out of the page in the y-direction.



Most users will be familiar with polygonal representations of objects. Providing a variety of surface types (ie: spheres, cones, & polygons) reduces both the user's time entering complicated object approximations and the program time required for the extra surfaces.

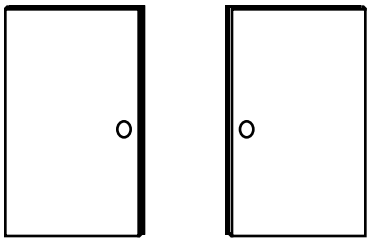
Polygons	***Polygons*** are specified as a list of at least three coordinate triplets. These vertices must lie in the same plane to be interpreted correctly. The order in which they are listed defines the surface normal orientation. A polygon can be concave or convex, and there is no upper limit to the number of vertices. Self-intersecting polygons should be avoided.
Spheres	A ***sphere*** is defined as a center point and a radius. By default, the surface normal points away from the center. Spheres with inward pointing normals are specified as ***bubbles***.
Cones	In the family of cones, several surface shapes are represented. A ***cylinder*** has a starting point, an ending point, and a radius. A ***ring*** has a center, a normal direction, and an inner and outer radius. A ***cone*** has a starting point and a radius, and an ending point and a radius. A ***tube*** is a cylinder whose surface normal is directed inwards rather than outwards. A ***cup*** is an inverted cone.
Sources	A light ***source*** is represented as a very distant disc. It is not really a surface, but a direction and an angular diameter.

Scenes are composed of many objects, and these objects may in turn contain many objects. This hierarchy is constructed through the simple mechanism of command expansion. When the program comes across a command in a scene file, it executes the command and interprets the output as more scene input. The ability to have a command within a scene file also simplifies the use of generator programs. Rather than including the output of a program that produces a box, for example, the command itself can be included in the scene file. Changes are straightforward, and the scene description is more compact.

The ability to move, rotate, and scale objects is essential to the creation of complex environments. XFORM allows these simple transformations, along with the mirroring of objects about any axis, and the generation of object arrays. Arbitrary transformations such as non-uniform scaling and skewing are not allowed simply because they don't make sense for all surface types. The main XFORM options are:

-t x y z	Translate the scene along the vector x y z. Since most objects generated by Radiance start out at the origin, they need to be repositioned to their final location.
-rx *degrees* (ry,rz)	Rotate the scene *degrees* about the x axis (respectively y or z axis). An object created at the origin may not be facing the correct direction, and might even be on its side, requiring rotation to lift it up or swivel it into place.
-s *factor*	Scale the scene by *factor*. The conversion factor between inches and feet is .08333 feet equals one inch.
-mx,my,mz	Mirror the scene respectively about the yz, xz, xy plane. Rather than recreating an object that shares some kind of symmetry with an existing object (such as doors that open different directions), it's easier to use the mirror option of xform.

Mirroring a Door with xform



xform Command for Mirroring Door

```
# This command is executed from inside a file, and mirrors the
# contents of the file called "door" about the y-axis, with a
# translation of 1.375 inches in the y-direction, to account for
# the thickness of the door.

!xform -my -t 0 1.375 0 door
```

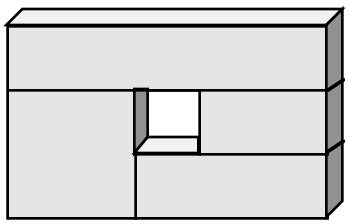
If the file name "door" was left out in the example above, xform would try to read from standard input all the expected data, and would then undoubtedly confuse the user with unusual results.

The specialized programs designed to create certain classes of objects are called *generators*. Their primary task is to simplify the task of entering descriptions of common or complex objects, and they are useful as inline commands in input files. Some examples of generators are:

GENBOX	Creates a parallelopiped with sharp, beveled, or rounded corners.
GENPRISM	Produces a Radiance scene description of a prism (an extruded polygon).
GENREV	Creates a surface of revolution using cones.
GENSKY	Creates a sky for use in daylighting calculations or outdoor simulation.
GENSURF	Creates an arbitrary curved surface patch by breaking a parametric function into polygons.
GENWORM	Creates a three-dimensional curve of varying thickness made up of cones and spheres.

When we want to describe a three-dimensional wall that contains a window, we can use either genbox or genprism to help us create the wall. Using genbox, we would build four boxes that surround the window and whose outer edges correspond to the perimeter edges of our wall. We'll need to issue four seperate genbox commands, one for each of the boxes in the wall. Genbox commands require specification of the material type and an identifying name for the surface being described, as well as the (x,y,z) size dimensions.

Building a Wall with genbox



genbox command example

```
# This is the genbox command for creating the first of four
# boxes that together form a wall with a window.

!genbox wood_panel top_wall 2  .2  .333 | xform  -t  0  0  .667
```

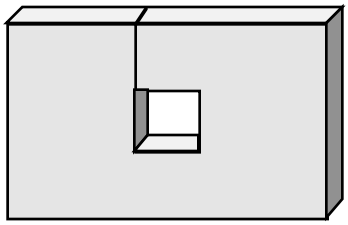
Note: Coinciding Surfaces

When Radiance surfaces coincide, so that two or more surfaces are located on top of one another, interesting and often undesirable effects can result when the objects have different modifiers (eg: different materials and patterns).

There is no problem in assigning different surfaces the same coordinates, as long as this "bleed-through" at the intersection is acceptable. For example, it is generally a good idea to place furniture up off the floor and away from walls, and the concrete foundation a little bit separate from the floor, but boxes created with genbox for a fireplace can be placed right on top of each other, since they share the same materials and patterns.

Using genprism to build our wall, we would build one prism that surrounds the window. Genprism requires specification of the material type and an identifying name for the surface being described, as well as the number of vertices to be entered, the vertices themselves (x,y), and finally an extrusion vector (-l option, for "length") that provides genprism with information about the missing (z) dimension. The order of the vertices and the extrusion vector determines the surface orientations, according to the right-hand rule (where the right thumb points in the direction of the surface normal as the right hand fingers curl around with the vertices).

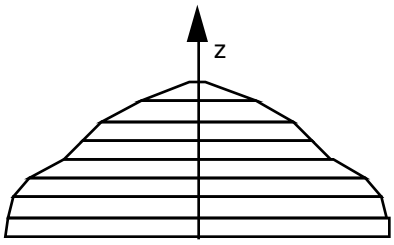
Building a Wall with genprism



genprism command example

```
!genprism wood_panel wall 10  0  1  0  0  2  0  2  1  .8  1  \
          .8  .333  1.2  .333  1.2  .667  .8  .667  .8  1  -l  0  0  -.2
```

Creating Curved Surface with genrev

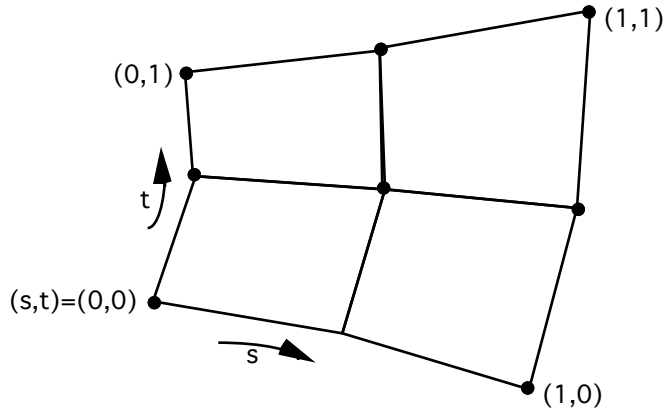


Genrev is very useful for generating surfaces of revolution that contain radial symmetry, such as glasses and vases. Surfaces are always rotated around the z-axis with genrev, and the surface is described in terms of z(t), the radius r(t) where t is an independent variable, and the number of segments. The example shown above consists of eight segments.

Gensurf could also be used to build a wall, although using gensurf to create such a rectangular object would be a tricky task; gensurf is better suited to curved surface patches. Any curved surface that can be defined parametrically can be built with gensurf, using (s,t) curved coordinates that might mean there are more than one "x" value for any given "y" value. Since gensurf allows the user to create as many sub-sections as desired along the s-axis and t-axis, it's very useful for subdividing surfaces automatically, as is sometimes desirable for large light sources. The "-s" option in the genrev command indicates that smoothing is desired for the

generated curve, so the ridges between segments don't show up as sharp corners.

Creating Curved Surfaces with gensurf



We'll use gensurf in the tutorial example at the end of this section to allow us to subdivide illuminated windows, and in the tutorial example at the end of the Advanced Topics chapter to generate a bumpy ground surface around the cabin. The surfaces gensurf is capable of creating are only limited by one's imagination (and mathematical prowess).

Note: Hermite Curves

Hermite functions are useful for specifying curves to generator functions. With a hermite curve function in Radiance, we only have to specify the starting point, end point, starting

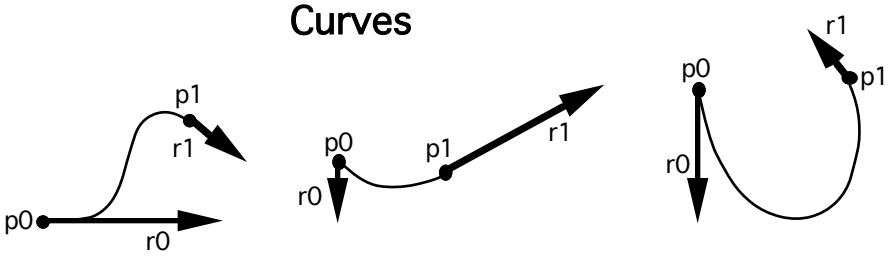
Hermite Function

```
# Hermite Curve Arguments
# p0, p1   - start & end points
# r0, r1   - start & end direction
# t        - independent
#              parameter (0 to 1)

hermite (p0, p1, r0, r1, t)
```

direction vector, and ending direction vector. The specifics for a hermite curve are predefined for us.

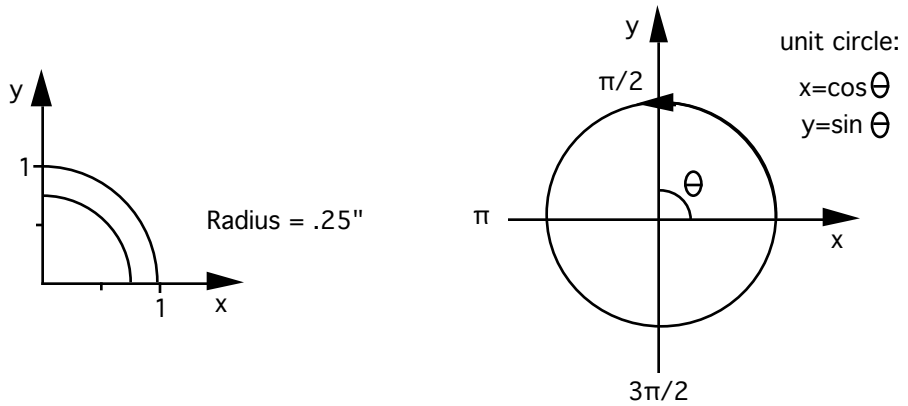
Each hermite function can be used to describe a curve containing up to one inflection point, where the second derivative changes sign (positive to negative, or vice versa).

Example Hermite Curves



The genworm command is very handy for creating curved elongated surfaces, given a formula of some kind describing the path and radius of the curve. To create a cylindrical shape that curves like a quarter-circle, one must first be familiar with the formulas for a unit circle. The number of segments must be decided upon for all genworm creations, since the genworm curves are not perfectly smooth. The higher the number of segments, the greater the realism and the more time required for generating the image.

Creating Curved Surface with genworm



genworm Command Example

```
!genworm brass right_bend 'cos(t*PI/2 + PI/2)+1'\
        'sin(t*PI/2 + PI/2) + 1'   '0'  .25  5
```

NOTE: Hierarchy

In a hierarchy, the XFORM command is used to read other scene files, transforming them to new positions. These scene files may contain other XFORM commands, thus producing a tree of transformations. XFORM can also be used to create arrays of objects, such as furniture or light sources.

Hierarchy Example

In a file called "tableset":
```
# A table with four chairs
!xform table
!xform -t 0 2 0 -a 4 -rz 90 chair
```

In the main scene file:
```
# A row of 5 tables with chairs
!xform -a 5 -t 10 0 0 tableset
```

If desired, XFORM may be used to expand all commands, creating a "flat" scene description.

Instances

Instancing is a way to construct complicated or repetitive scenes by taking an object that has been described once and duplicating it as many times as are required, such as creating a forest by defining one pine tree and selecting locations for duplicate pine trees to be placed in the scene. Using this technique tells Radiance that these objects are identical, except that they are placed in different locations and possibly consist of different material types.

Instancing helps to minimize memory requirements. Although the limit to the number of surfaces in the expanded geometric model is large -- around 18,000, depending on memory -- it is not infinite. Since the same objects are repeated many times throughout the space, instances can share the data of those objects, requiring memory for only individual transformations. Using hierarchical instancing (instances that in turn contain other instances), scenes

with hundreds of billions of primitives can be modeled. Although rendering time does increase with the scene complexity, it is a sub-linear relationship $O(n^{1/3})$ such that an image with 1,000 surfaces takes roughly twice as long as a scene with 125 surfaces, and a scene with 8,000 surfaces takes four times as long as a scene with 125 surfaces.

Instancing is accomplished by creating an octree of the object which is repeated in the scene, so that subsequent instances of that object refer back to the same octree. For the pine tree forest example, an octree for the pine tree would be created (called "tree.oct" in this case), and the locations of each instance of the pine tree would be defined in primitives.

Instancing Example

```
# Planting a couple of trees
# (both from tree.oct)

void instance first_tree
9 tree.oct -rz 50 -s .9\
    -t 35 -10 -1.5
0
0

void instance second_tree
9 tree.oct -rz 125 -s .8\
    -t -8 20 -1.5
0
0
```

Each tree can be rotated a different amount about the z-axis with the "-rz" option, and scaled with the "-s" option before being placed with the translation coordinates ("-t" option) desired.

Slightly different rotations and sizes of trees help provide the scene with some diversity in its trees, so it's not too apparent that all the trees are actually identical to each other.

Textures

Textures add important visual detail to surfaces without adding substantially to the model complexity. We define a *texture* as a perturbation of the surface normal (as opposed to a perturbation of the material color), giving the object the appearance of having a bumpy surface that could be felt if touched. A texture affects the illumination and highlights of an object. There are two types of textures: texfunc, and texdata.

texfunc	is the most commonly used type of texture. It uses a functional procedure to give surfaces a bumpy (non-smooth) appearance.
texdata	can be used if you have a data file corresponding to surface normal perturbations of a surface.

Patterns

We define a *pattern* as a perturbation of the material color (as opposed to a perturbation of the surface normal). A pattern affects the reflectance (or transmittance) of an object. Radiance provides several means for pattern specification. A procedural pattern is given as a formula that defines the pattern's value in terms of the current intersection point, ray direction, surface normal, distance, etc. Patterns may also be scanned from photographs or video. The coordinate mapping for a pattern is determined by the user, and mappings from rectangular images to rectangles, cylinders, and spheres are provided.

There are three general classes of pattern types used by Radiance: functions, data, and text.

Functions	**Brightfunc** and **colorfunc** describe patterns as functional procedures. Brightfunc changes the overall reflectance variable for an object, while colorfunc changes the colors (red, green, and blue).
Data	**Brightdata** and **colordata** take information from data files (rather than from functional procedures) to change the overall reflectance colors of an object. **Colorpict** is the same as colordata, but it takes a Radiance picture file as input, rather than three data files.
Text	**Brighttext** and **colortext** produce text (eg: page from a book) when the text and font are specified. Brighttext requires foreground and background reflectivity, while colortext requires foreground and background color.

Note: Auxiliary Files

There are several auxiliary files which are useful for creating patterns; these files are stored in the system Radiance directory (/usr/local/lib/ray, on most machines).

Data Pattern Example

```
void colorpict needlepat
9 red green blue forestfl.pic
picture.cal\
      match_u  match_v  -s 2
0
0

needlepat plastic groundmat
0
0
5  .5  .3  .2  0  0
```

This example of a ground material modified by a pattern called "needlepat" shows how a data function can be used to modify a material. The needle pattern called "needlepat" is taken from a picture file called "forestfl.pic", which is a scanned image. The "picture.cal" file is used to specify how the picture file will be used, and contains different variables for different options (such as match_u and match_v for tiling so the edges match). The scaling for this data pattern has been set to 2 with the "-s" option. Note how the modifier "void" is now used for the pattern called "needlepat", and the material in turn uses "needlepat" for its modifier.

A List of Modifiers

```
void brightfunc filthy
4  dirt  dirt.cal  -s 2
0
1  .6

filthy colorpict needlepat
9 red green blue forestfl.pic
picture.cal\
      match_u match_v -s 2
0
0

needlepat plastic groundmat
0
0
5  .5  .3  .2  0  0
```

If we now added a functional pattern called "filthy" to modify our forest floor pattern so that the appearance of regularly spaced tiles would be minimized, we would have a list of modifiers influencing our final "groundmat" material. The functional pattern used to create dirty surfaces can be scaled to whatever factor is desired with the "-s" option (2 in this example), and the intensity of dirt can be provided as a real argument (60% dirt for "filthy"). The "needlepat" modifier for our "groundmat" material is in turn modified by the functional dirt pattern called "filthy", whose modifier is now "void", since it is modified by nothing else.

This dirt brightness pattern is an extremely valuable companion to any scanned-in pattern for introducing a more natural, uneven look. The dirt pattern is based on a noise function, so it produces an unpredictably varied pattern.

It's possible to have as large a string of modifiers as you wish (within reason), so that patterns or textures modify other patterns or textures that in turn modify some material.

> **Note: Noise Functions**
>
> There are two basic types of noise functions available that can be used to good effect with patterns. **Fractal noise** is the rougher type of noise, that contains a lot of high frequencies, while **noise** is smoother. Both types of noise are useful for introducing an element of randomness to the appearance of things, and both two-dimensional and three-dimensional noise functions are available for use with Radiance. (Look at "rayinit.cal" in /usr/local/lib/ray to learn what functions are predefined and find out variable names).

Mixtures

Mixtures allow the combination of textures and patterns in unique and exciting ways (eg: weaving leather strips in burlap cloth)†. By specifying in the mixture a coefficient that describes a combination of patterns and textures, it's a lot easier to generate pattern and texture combinations (by "toggling" them off or on) than it would be if different pieces of polygons were pieced together, and some effects can be produced that can't be produced any other way (such as creating a "soft blend"). There are three basic types of mixtures: mixfunc, mixdata, and mixtext.

mixfunc	allows specification of a function that determines the mixture coefficients.
mixdata	takes information about the mixture from a data file.
mixtext	handles mixtures of text, such as bumpy lettering on a piece of wood.

Antimatter

Antimatter allows the "subtraction" or removal of one surface from another. Once a surface has been defined as antimatter to another type of surface, the subsequent intersection of these two surfaces

---

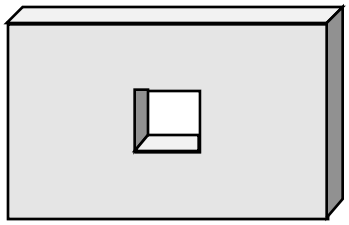
† Radiance does not currently support the mixing of different materials (such as glass and plastic, or two different plastics).

will result in a "hole" where the two surfaces meet, resulting in the subtraction of a volume from the original surface.

This feature does not work with all material types (the trans material type, specifically), and care must be taken to avoid intersecting two or more antimatter surfaces in a scene (intersecting antimatter surfaces don't work properly). Additionally, the antimatter surface can't be used with a viewpoint located inside the volume (simply ensure that the viewpoint is located elsewhere).

Antimatter can also be used to describe our previous example of a three-dimensional wall containing a window; an antimatter window can be placed inside a genbox-defined wall.

Building a Wall with Antimatter



Light Sources

Because light sources are critical to the illumination of a scene, they are given special attention by the program. In general, a light source is differentiated from other surfaces by its material type. Currently polygons, spheres, and disks may be used for local sources; area sources and odd shapes may be modeled with polygonal meshes.

Distant light sources such as the sun are modeled as special cases. A non-Lambertian source distribution (that doesn't emit light equally in all directions) is another kind of special case.

gensky Example

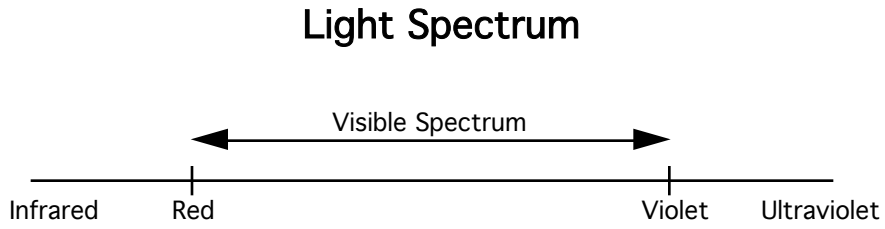
```
# Hemispherical Blue Sky

!gensky  4  1  14

skyfunc glow skyglow
0
0
4  .9  .9  1  0

skyglow source sky
0
0
4  0  0  1  180
```

The generator program called gensky creates a sky that can be used for daylighting calculations or outdoor simulations. Gensky can create a Radiance scene description for the CIE standard sky distribution at the given date and time; for gensky, the x-axis points east, the y-axis points north, and the z-axis points up (towards the zenith). The sky can be sunny with or without sun, or cloudy. The materials and surfaces used for describing the sky are left up to the user; this example describes a simple hemispherical blue sky. Since Radiance has defaults set for the San Francisco bay area, longitude, lattitude and meridian information need to be supplied for different locations.

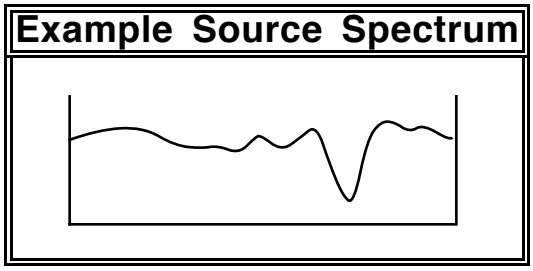
IES lighting files are available to all Radiance users that contain Radiance descriptions for a bunch of IES fixtures (and can be found under /usr/local/lib/ray/source/ies). The files were converted from standard format to Radiance using the "ies2rad" conversion program (see Radiance Reference Manual). Lamp color was not used in these files (all are white), because colored lights have to be balanced before they can provide color-balanced renderings. Light source colors are especially important when comparing incandescents and fluorescents -- the natural color balancing in your eyes compensates for this effect. The ies2rad program converts one ies file (usually ending in ".ies") into a Radiance file (ending ".rad"), and an auxiliary file used by Radiance (ending ".dat") that contains output distribution data.

If a light source is missing some frequency (part of the color spectrum), it looks color-balanced, but not very colorful. Incandescents can't render blue, violet, and purple very well because they don't have as many higher frequencies (which fluorescents do have). Color-balancing is the technique of seeing lights that are not heavily weighted in blue, green, or any one color. If color-balancing doesn't take place somewhere (in your eyes, on camera film, with a software filter), the resulting images don't look natural. Using all white instead of colored lights is kind of a cheat, but Radiance doesn't have any easier way to color balance (other than pfilt). Mixed source types require consideration of the colors

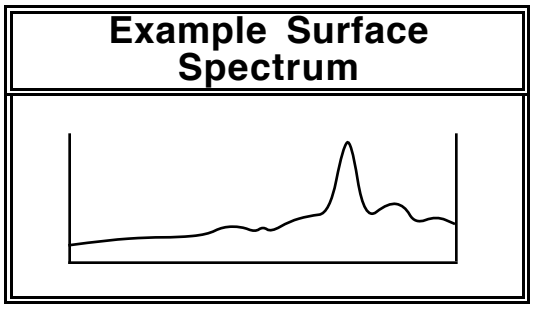
of each contributing source, where converging areas show gradiation.



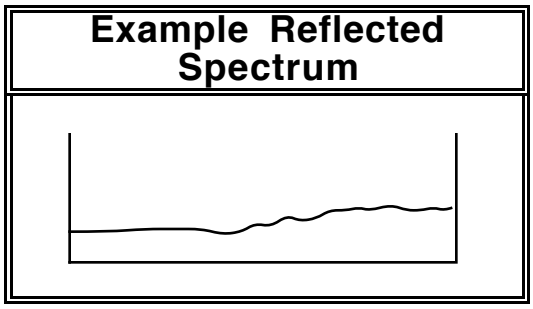
Radiance uses a RGB (red, green, blue) color model. Our eyes have receptors for these three colors. Three well-chosen colors will represent almost all of the colors we're able to see, but it's still not the same as representing all the full spectral content of the scene (since we're only sampling three colors, instead of continuously sampling the entire spectrum). An example showing how a light source and surface can interact helps to illustrate this point.



This example of a light source spectrum illustrates the situation where a light source clearly shows a deficiency in the blue range of the spectrum when we look at a graph showing its performance over the entire visual spectrum. Even though this dip shows up clearly on our graph, the light source will still appear mostly white to Radiance, since its RGB values would be something like (10,10,8).



An example surface that just happens to have a reflectance peak in the blue range would look blue under white light, and might have RGB values something like (.5,.5,.7), which don't fully describe the striking peak in the blue range that we can see when we look at the entire visual spectrum.



When our example surface is viewed under our example light source, the peak and the deficiency cancel each other out, and the surface that should appear bluish instead appears grey, with RGB values on the order of (5,5,5.6). This example illustrates the point that taking only three color samples is inadequate; the RGB system is the best we have to work with, but it's important to understand its limitations as well.

The file called "lamp.tab" (located under /usr/local/lib/ray) contains all sorts of useful lamp information, including: lamp types, xy chromaticity coordinates, and depreciation lists. This information is useful for looking up the RGB values of different types of lamps. For example, an incandescent lamp has an x-chromaticity of .453 (Red) and a y-chromaticity of .405 (Green) with a depreciation factor of .95. Since the chromaticity factors have been normalized to one, the Blue value can be computed (Blue = 1 - .453 - .405 = .152).

When you choose to determine your light source's radiance yourself, without aid from the IES files, there is a four-step process you can follow. Starting with the lumen value of the light source you wish to describe to Radiance, you will need to convert the lumens to watts (power), and divide by the area of the source. The fourth and final step is compensating by some depreciation value. The result from using this computation will be a light value you can use for the RGB values in Radiance when you create your light source primitive†.

† You can use about 15 lumens/watt for an incandescent source, as a general rule of thumb. A 50 watt incandescent bulb is approximately equivalent to 500 lumens.

Computing radiance
(1) Determine light output in lumens. (2) Convert lumens to watts (multiply by 1 watt / 179 lumens) (3) Divide watts by emitting area of source in square meters and by π, resulting in final units of watts/steradians/meter^2 for R,G, & B. (4) Compensate for fixtures and lumen depreciation (in the range 5% to 20%).

Example radiance computation
(1) 860 lumens (2) 860 *(1/179)=4.80 watts (3) (4.80/(3.5"* (.0254 m/inch))^2*4* π)/π =15.4watts/steradian/meter^2 (4) 14.5 w/sr/m^2

An example of computing Radiance for an incandescent bulb with output of 860 lumens (which we looked up in a manufacturer's table) will help illustrate this process. We convert our 860 lumens to watts in step two, and divide the value in watts by 4π and the area of a 3.5 inch sphere. Our final result of 15.4 w/sr/m^2 is reduced to 14.5 w/sr/m^2, to compensate for lumen depreciation of the bulb and the fixture.

The computation outlined above is embodied in a program called LAMPCOLOR for your convenience. LAMPCOLOR asks questions about the lamp type, output and geometry and computes the radiance value accordingly.

Importing from CAD Systems

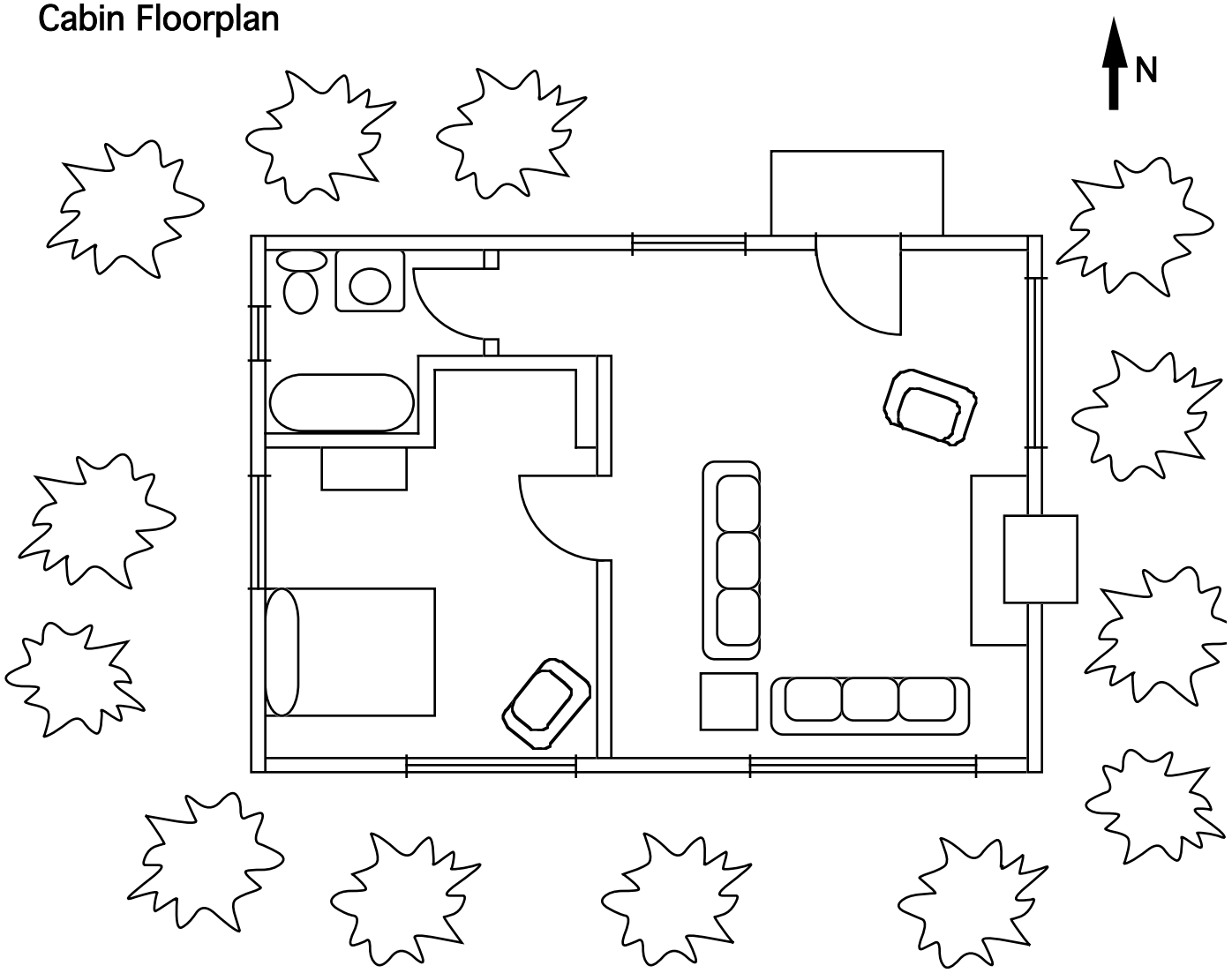
Radiance typically operates in conjunction with a commercial computer aided drafting (CAD) system, which is used to create the scene geometry. An import program takes the CAD scene description and converts it to the Radiance input format. Details such as surface properties are added, analysis is performed, and design modifications are made based on the result.

Tutorial Example

We've chosen an example of a cabin in the woods for this tutorial, which we'll start in this section and continue working on in the following chapters. The floorplan is shown below, and some of the pine forest surroundings are also shown. Since we'll need to describe the scene before we can generate and manipulate images, we'll be starting off by defining a few materials for the scene, setting up the basic geometry (walls, the door, and windows), and describing the basic setting of sky and ground surrounding the cabin. Once we've gotten the basics defined, we'll also be experimenting with patterns and textures in the cabin, and using instancing to create a forest of pine trees.

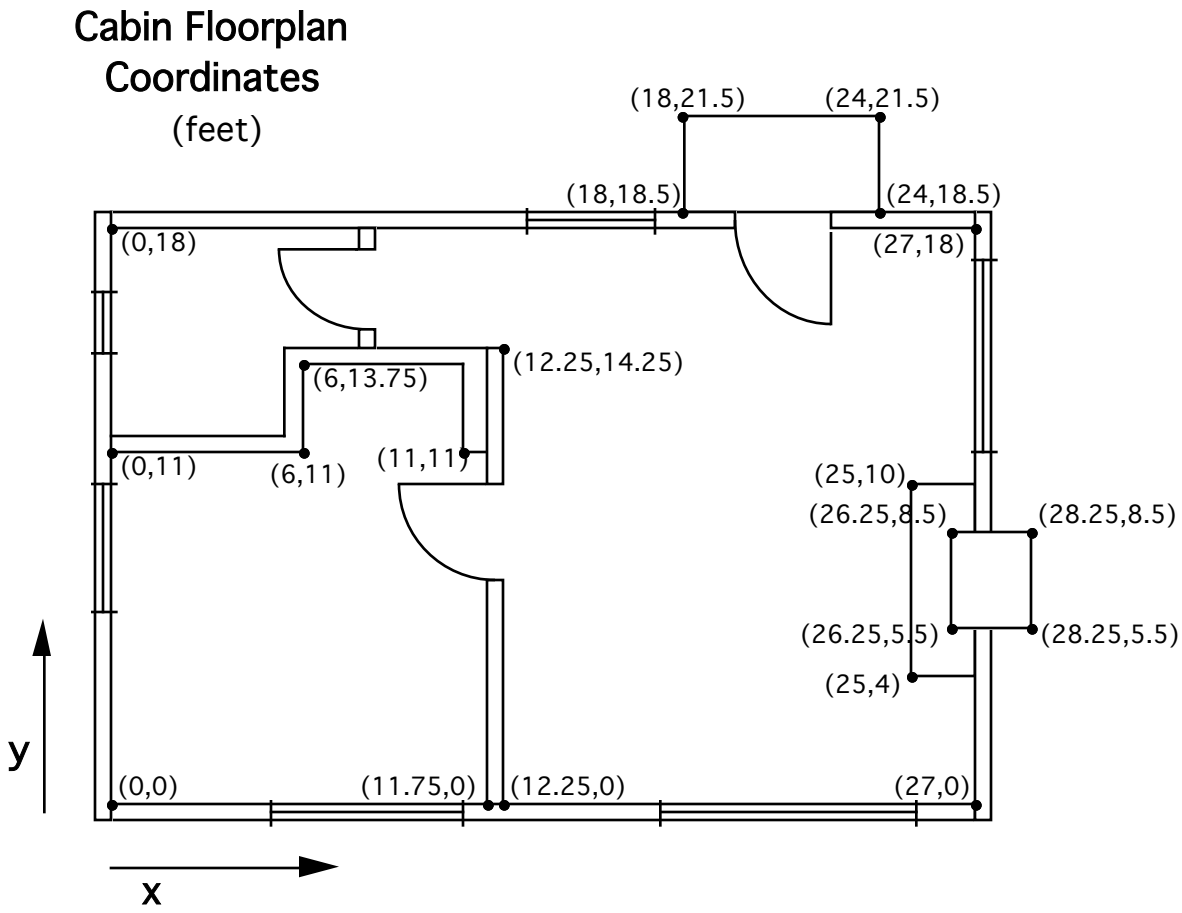
This example is a fairly simple building structure, yet it will allow us to learn the skills necessary to complete much more complex scene descriptions. In the following chapters, we will be using this scene description to learn about Radiance's image generation and image manipulation capabilities.

Cabin Floorplan



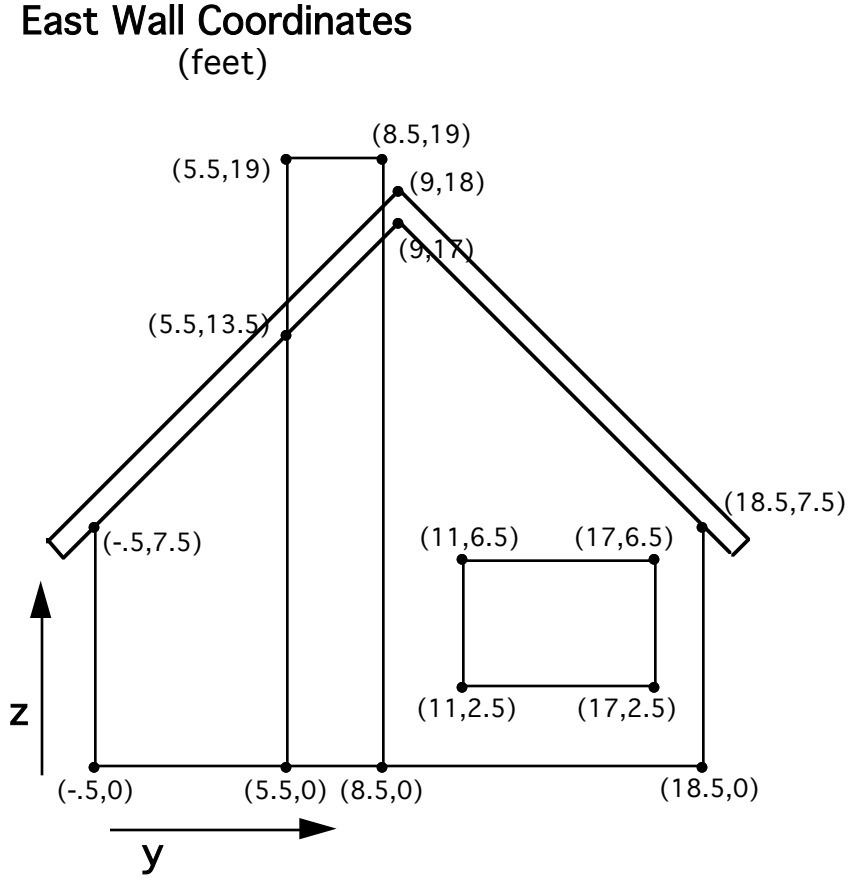
one inch = six feet

This floorplan shows a three room cabin, surrounded by pine trees. The entrance to the cabin is the door on the north side of the livingroom, next to the porch. The livingroom has three windows, a fireplace on the eastern wall, and two interior doors on the western wall which lead to the bedroom and the bathroom. The bedroom has two windows, and the bathroom has one window. The overall inside dimensions of the cabin are 18 feet by 27 feet.



One of the first things to determine is the axis orientation and scale in our scene. The cabin was drafted in feet on a right-handed Cartesian coordinate system that placed the origin (0,0,0) at the floor in the most southwest corner of the cabin's interior for convenience. The y-axis points north, the x-axis points east, and the z-axis points up, out of the page, corresponding to the axes we'll need to set when we describe the daylighting conditions using gensky.

The cabin's roof slopes down on the north and south sides, so that the east and west walls rise to an interior height (on the z-axis) of 17 feet. Coordinates for location of some key points in the scene have been determined and written here for future reference when we start generating surfaces.



Input Files

We will be creating this scene description directly with a text editor. It's a good idea to master the skills of working with the Radiance input files, where the object descriptions are contained. We'll need to put the material primitives and surface primitives into input files, as well as information about the sky conditions and so forth.

Materials

We'll need to define the materials we want to use for the foundation, walls, floor, roof, and ceiling of our cabin, as well as the porch, fireplace, windows, doors, and furniture. We can start a file that will contain only cabin material primitives, and call it "materials". We'll add to this file as we think of new materials we may need along the way, and we can

Foundation Material

```
# Cabin Foundation Material

void plastic concrete
0
0
5  .3  .3  .3  0  0
```

start by defining our material for the foundation. Since we have no other identifiers yet defined, the modifier for our foundation material must be **void**. We use ***plastic*** as the material type, and name our material "concrete". There are no string arguments for materials, so the next two entries in our primitive are zeroes. All materials require five real arguments for red, green, blue, specularity, and roughness; our concrete's RGB color values are all equal for a grey tone, and about a third the fullest color intensity one might find in a very white material (.3). There is no specularity for this material (zero), and likewise, we need not provide any roughness (also zero).

Cabin Floor Material

```
# Cabin Floor Material

void plastic wood_floor
0
0
5  .3  .15  .05  .02  .05
```

The material for our cabin floor will have some shine, or specularity and some roughness (to prevent a perfect mirror-like shine). Since we don't need to modify our wood floor material by any previously defined primitive, we'll start off again with a modifier of **void**. *Plastic* is again the best choice of material type to create our wood floor, and we'll identify this material as "wood_floor". The cabin walls and roof will be made of similar types of wood material, with slightly different real arguments for RGB color, specularity, and roughness.

Brass Material

```
# Brass Material

void metal brass
0
0
5  .68  .27  .002  .95  0
```

We'll need brass material for the metal fixtures in the cabin (such as the door knobs), and brass provides us with an interesting example of a material that has lots of specularity and no roughness. The material type for brass is ***metal***, and it's color is mostly red with a little bit of green and hardly any blue at all.

White Enamel Material

```
# White Enamel

void plastic white_enamel
0
0
5  .5  .5  .5  .02  .05
```

The cabin's doors, windows, and roof trim are all made up of a white enamel material. This white enamel paint has equal red, green, and blue arguments (.5). There is some specularity (.02) and roughness (.05), to provide the material with the slightly glossy look of enamel paint.

Cabin Door Materials

```
# Cabin Door Materials

void alias door_paint
    white_enamel
```

Since this same material is used for many surfaces, we can create primitives for these other material identifiers with aliases that refer back to our "white_enamel" material. This example shows how we've created a primitive called "door_paint" which will share all of its characteristics with "white_enamel".

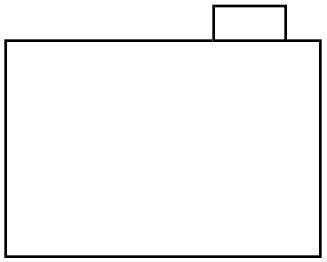
Dresser Mirror Material

```
# Dresser Mirror

void metal mirror
0
0
5  .8  .8  .8  .9  0
```

The mirror surface on the bedroom dresser has very high and equal red green and blue values, since most light hitting the mirror is reflected. The specularity is very high (.9), and the roughness is negligble, since we want a shiny mirror. The material type we're using is metal this time, since metal has the properties we desire in a mirror.

Surfaces

Foundation, Porch & Floor

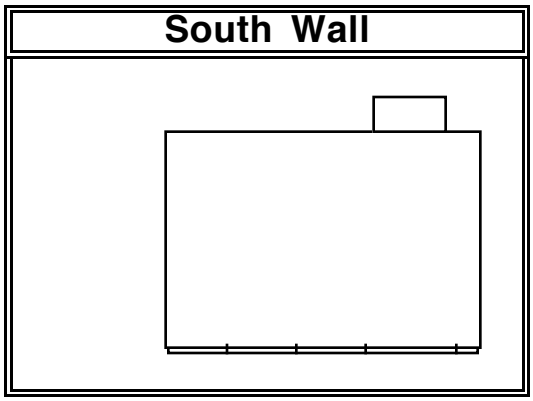


The cabin's foundation is a fairly simple surface to generate, and since it's shape is basically that of a box, we can use the genbox command to build the foundation for us. The material for the foundation has already been defined in our "materials" file as concrete.

Keeping in mind that the cabin walls are 6 inches thick, and that our foundation will be set in by about .1 feet in each direction, we can determine the foundation dimensions. Our generated box from genbox will end up sitting with one corner on the origin, and will be sitting inside the interior of the cabin, so we'll need to use xform to translate the box down a little more than 3 feet (so the concrete foundation doesn't show through the interior floor), and back a bit in both the x-axis and y-axis directions to allow for the thickness of the walls. The cabin's porch is another simple box shape, and the floor can be modeled as a polygon.

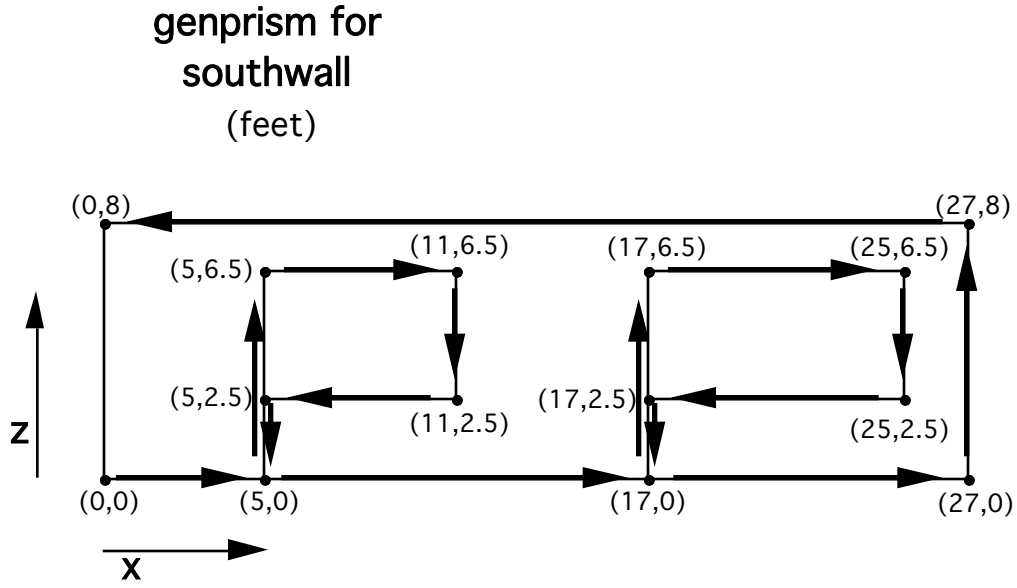
genbox Command for Building Foundation

```
!genbox concrete foundation 27.8  18.8  3 \
        | xform  -t  -.4  -.4  -3.0001
```



We'll use genprism to build the cabin's walls, starting with the south wall, which has two windows. Genprism is not the only generator capable of building three-dimensional walls that have windows, but it is the simplest for us to work with.

Once the principle for building a wall is understood, is very easy to raise all the rest of the cabin's walls in a similar fashion.



Although we intend for this wall to end up standing, it's actually starting out sitting on the floor of the cabin, since genprism accepts all coordinates in (x,y) with z equal to zero. (The sketch above shows the x-axis and z-axis directions of the wall once it's been lifted up into place.) We'll need to specify that the thickness of the wall will be 6 inches; this extrusion along the z-axis will be down, into the page, since the wall's front face is in the up direction. This can be verified using the right hand rule; our right hand's thumb

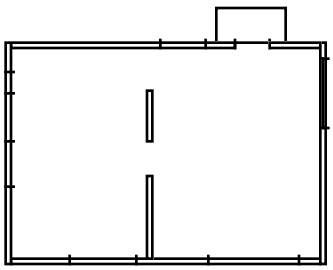
(corresponding to the surface normal) points up, out of the page if we curl our right hand fingers in the direction of the vertices shown in the figure above (counterclockwise). Extrusion along the z-axis is always in the opposite direction of the surface normal on genprism, since all it's generated surfaces should have surface normals that point outward (including both the interior and exterior wall surfaces).

We will "lift" the southwall into place using the xform command to translate the wall up along the z-axis by 6 inches, and to rotate the wall by 90°. The rotation is positive, and this can be verified using the right-hand rule (our right-hand fingers curl up with the lifting wall, as our right-hand thumb points in the positive direction along the x-axis). We will pipe the results of our genprism command into the xform command, using the backslash, "\", to continue the command over multiple lines. All other walls can be built in similar fashion.

genprism Command for Building Southwall

```
!genprism wood_panel southwall 16   0 0   5 0   5 6.5   11 6.5 \
        11 2.5   5 2.5   5 0   17 0   17 6.5   25 6.5   25 2.5 \
        17 2.5   17 0   27 0   27 8   0 8 \
        -l 0 0 -.5 | xform -t 0 0 .5 -rx 90
```

North East Windows



Windows for the cabin have already been defined in a file called "window.norm", so all we need to do is place them in the window openings. The window measures 2 feet in width by 4 feet high, and we can be place three of them side by side to create 4-foot and 6-foot wide windows for the northeast windows in the livingroom. The xform command will do the work of placing the window in the window opening for us. We need to include the "-e" option to indicate we want inline commands to be expanded, and the "-n" option to give the window a name. Since the window is starting out sitting on the x-axis with it's lower left corner at the origin, we'll need to rotate it 90° around the z-axis and translate it to the starting coordinates for the window (27.5, 11, 0). The x dimension of 27.5 is necessary to allow for placement of the window inside the wall correctly, since the

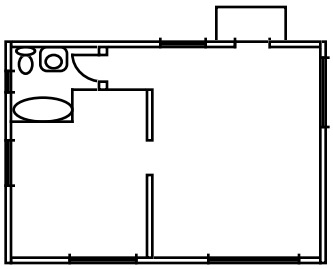
thickness of the walls needs to be taken into account. We can use xform's array option, "-a", to indicate we want the window to include three of the window segments, and follow the array option with the translation vector (0,2,0) that indicates we'll be installing window segments 2 feet apart along the y-axis. All other windows in the cabin can be built in similar fashion.

xform Command for Building Northeast Window

```
!xform -e -n northeast_window  -rz 90  -t  27.5  11  0 \
        -a  3  -t  0  2  0  window.norm
```

Our cabin model takes advantage of a previously defined bathroom in a file called "bathroom", so we won't need to to model any bathroom surfaces, and can just drop the whole bathroom scene description into our model without going to the trouble of describing every surface.

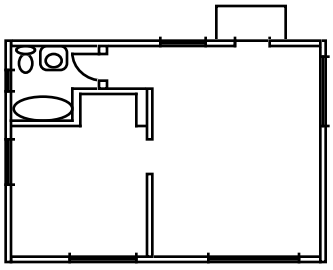
Bathroom



We will need to define the north bedroom wall to go around the bathroom, and can do this easily using a polygon description of all the vertices on either side and above the closet (which is 6.5 feet high). We'll need to more fully describe the bedroom closet next.

We need to add some substance for the bedroom closet, since all that currently exists is the perimeter of the bathroom model with the polygon we've just defined showing it's infinitely thin thickness around the closet door opening. The bedroom closet can be built using the genbox command, provided we then open up the polygon face of the box that should open into the bedroom so we end up with a five-sided box.

Bedroom Closet & Wall



North Bedroom Polygon Wall

```
wood_panel polygon nbed_wall
0
0
24      0       11      0
        6       11      0
        6       11      6.5
        11      11      6.5
        11      11      0
        11.75   11      0
        11.75   11      15
        0       11      15
```

We will build the closet out of wood_panel material, with dimensions of (5, 2.75, 6.5), and invert it so the surface normals point inwards (where it will be seen) using the "-i" option of genbox. From the vi editor we can issue the ":r" read command in front of our inline genbox command, so the output of the genbox command will be entered directly into the edited cabin file, allowing us to comment out the closet door opening (y values will be equal to 11). We'll need to translate the generated box using xform, since it is created with it's lower left corner sitting on the origin.

genbox Command for Building Bedroom Closet

```
:r  !genbox wood_panel closet  5  2.75  6.5  -i | xform -t  6  11  0
```

We also need to remember to change the material type for the closet floor, since genbox automatically assigned the wood_panel material, and that won't match our other floors made of wood_floor. All we need to do is find the closet polygon that has z values equal to zero, and change the material type from wood_panel to wood_floor.

The finishing touch on the closet is to add a closet shelf and a closet dowel. We can use the genbox command for the shelf, and create a cylindrical type of surface.

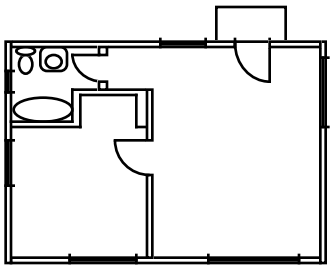
Closet Shelf

```
!genbox light_wood closet_shelf  5  1  .08333  | xform  -t  6  12.25
5.33
```

Closet Dowel

```
light_wood cylinder closet_dowel
0
0
7       6       12.25           5
        11      12.25           5
        .052
```

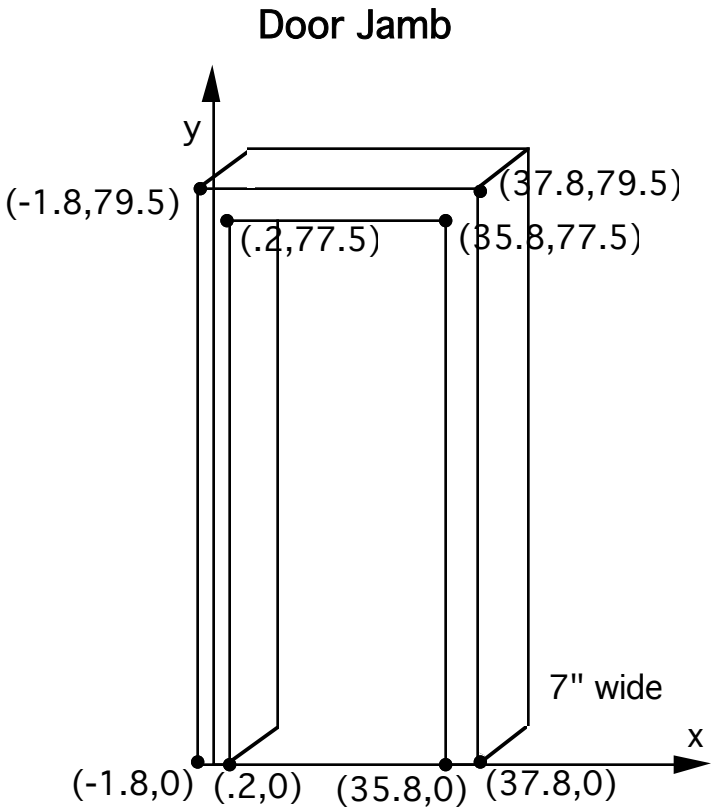
Doors



Details help make models look "real". By adding doors with brass doorknobs, we'll add an element of realism to our cabin scene. Genbox can be easily used to generate the doors, and xform can be used to "hang" the door correctly in the doorframe. Since there's more than one door in the cabin scene, and the doors don't all open the same way, we'll make a separate "door.norm" file. The door.norm file will allow us to easily place the doors where we want them. The door jamb primitives can be included in our "cabin" file of other scene structures, since door jambs are relatively simple structures that don't require much effort to place correctly in the scene.

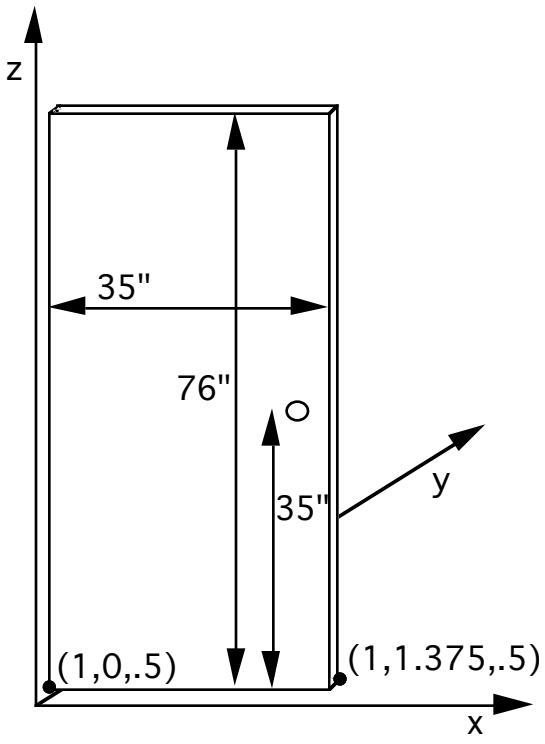
The door jambs can be created using either genbox or genprism. Genprism is a bit more straightforward, since one command can be used to create the entire structure.

When creating the door jamb, it's important to follow the right hand rule to determine where the surface normal will point (away from the extrusion direction), and therefore what sequence to use for entering vertices. Our extrusion direction will be downward, into the page, so our surface normal is up, and that means when our right hand's thumb points up, the fingers curl around in a counter-clockwise fashion in the xy plane. (Genprism always works with xy coordinates). Because the extrusion is 7 inches down (in the negative z direction), the length "-l" vector will be (0,0,7).

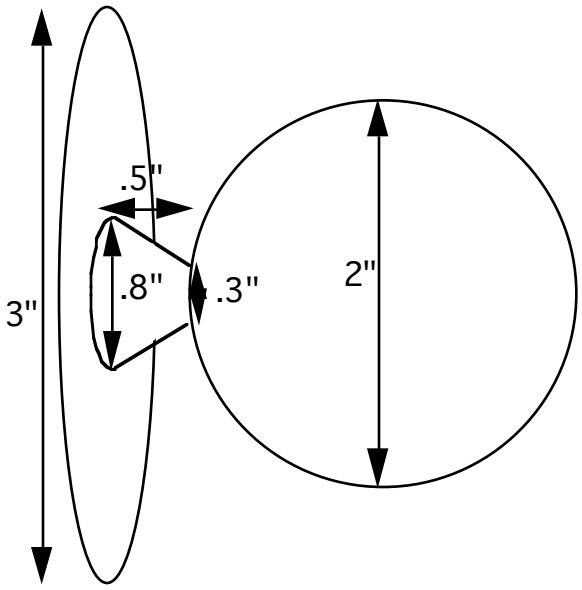


Bedroom Door Jamb
Bedroom Door Jamb !genprism door_paint beddoor_jamb 8 -1.8 0 .2 0 .2 77.5\ 35.8 77.5 35.8 0 37.8 0 37.8 79.5 -1.8 79.5\ -l 0 0 -7 l xform -t 0 0 .5 -s .08333\ -rx 90 -rz 90 -t 12.25 7 0

Cabin Door



Brass Doorknob



The measurements of our doorknob components are illustrated here. The doorknob consists of three distinct pieces: the spherical

doorknob, the conical knobstem, and the knobplate ring. These three simple surfaces can be generated using the sphere, cone, and ring surface types, respectively.

Cabin Door

```
# Cabin Door File for a door, called "door.norm".
# Scale is in inches.  Origin is at floor, 1" below bottom of door
# and .5" from hinge edge of door.  This door fits in a frame
# slightly less than 36" by 78".  The materials door_paint and
# brass need to be defined.

!genbox door_paint door 35  1.375  76  |  xform  -t  .5  0  1

brass sphere inner_doorknob
0
0
4   33.125   -1.5   36   1

brass cone inner_knobstem
0
0
8       33.125          0       36
        33.125          -.53    36
        .8              .3

brass ring inner_knobplate
0
0
8       33.125          -.01    36
        0               -1      0
        1.5             .8
```

Now that our basic doorknob parts are defined for one side of the door (the inner side), we'll need to add a doorknob to the other side of the door. This can be fairly easily accomplished before the "door.norm" file has been saved, using xform's mirror option as an inline command, to read in the entire door.norm file so far, and mirror it about the y-axis by the thickness of the door (1.375 inches).

Mirroring Inner Door Knob

```
:.,$w!
:$r !xform  -a 2   -my  -t   0   1.375   0   door.norm
```

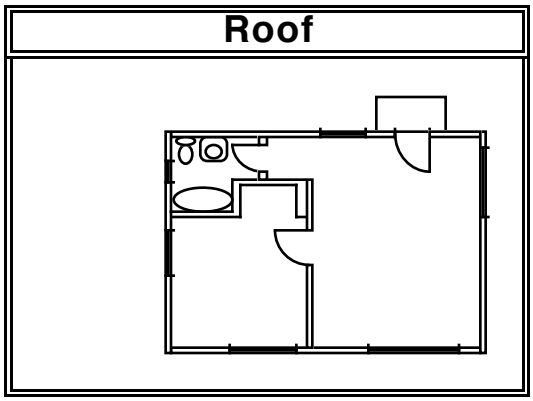
These commands must be executed from the start of the doorknob description (the line that reads "brass sphere inner_doorknob") in the "door.norm" file, to indicate that only the doorknob parts should be written to a file, and then used for the following xform command. The results from this command will produce primitives containing the correct values for the mirrored doorknob parts, which we can edit in vi to names that begin with "outer", to indicate their position on the other side of the door in the "door.norm" file.

Now we're ready to place a door in the cabin. The bedroom door can use the "door.norm" file, as long as we remember to use a scaling factor with xform's "-s" option to convert from inches to feet. We'll put the door in the scene in an open position using the "-rz" rotation around the z-axis, so we'll be able to look through the bedroom door when we start generating images of our cabin. When we place the door in the bedroom, we'll need to remember to provide xform with the translation coordinates of (11.9, 10, 0). The x coordinate is slightly less than 12, to allow for the door to open a little distance away from the door jamb. We'll use the "-e" option to expand the results of our xform command, and we'll name our bedroom door "bedroom_door".

Bedroom Door

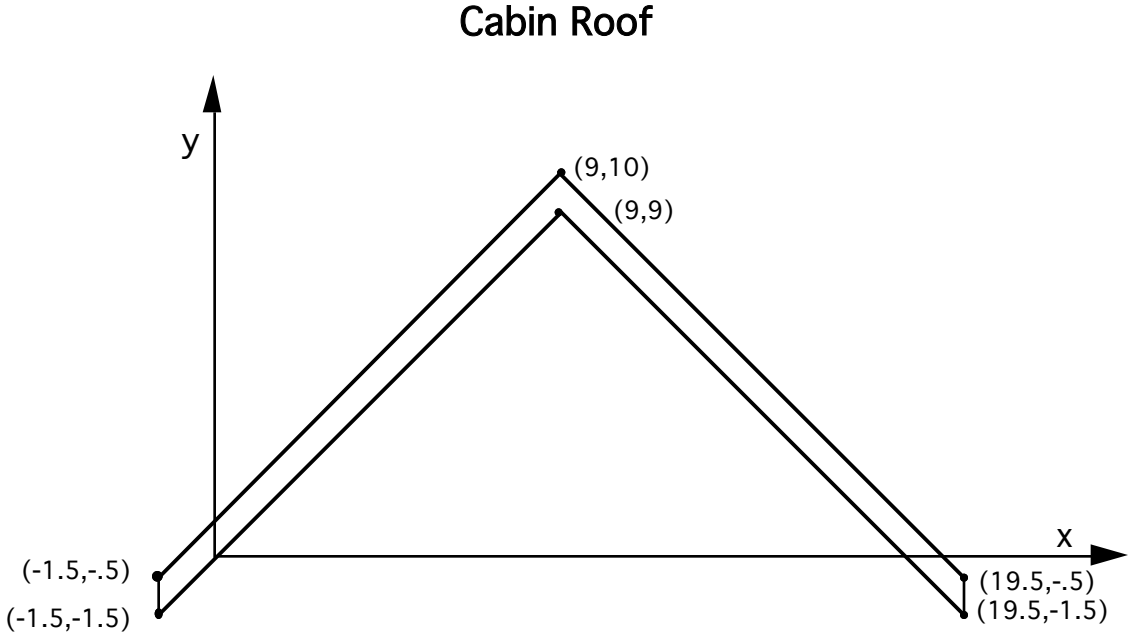
```
!xform -e -n bedroom_door -s .08333 -rz -165 \
        -t 11.9  10  0  door.norm
```

The front door can be created using the "door.norm" file also, using xform's mirror option to move the doorknob to the other side of the door, and to place the door in the correct location. The front door can be placed in a closed position, or it can be rotated around the z axis by some amount.



The cabin's roof can be created using genprism, since it will have a thickness. We could use two-dimensional polygons instead of genprism, but genprism is quick and easy. We'll use a lighter color wood for our interior ceiling, for better light distribution, and the white enamel material for the trims. The top of the roof will be shingles eventually (when we get around to doing patterns), so we'll call the north roof and south roof different names.

We'll start the genprism command with the material type "white_enamel", even though we'll be changing the material type for the top two roof surfaces and the bottom two ceiling surfaces, since six of the ten surfaces will use the white_enamel material. We'll set the roof on the xy plane in such a way that the origin will intersect the inside top part of the cabin's wall; this means the roof will need to extend below the x-axis and into the negative along the y-axis. We'll use the right hand rule to determine what sequence to enter our vertices, so that the surface normal points down, and can therefore determine whether our extrusion length value of 29 feet along the z-axis will be the opposite direction (positive).



Creating a Roof with genprism

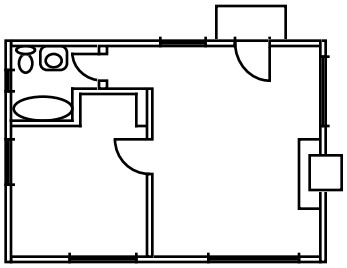
```
:r  !genprism white_enamel roof  6  -1.5  -1.5  -1.5  -.5\
        9  10   19.5  -.5   19.5  -1.5   9  9   -I  0   0   29\
        I xform  -rz 90  -ry 90  -t  -1  0  8
```

By using the ":r" command in our "cabin" file, and entering the genprism command, we'll get the resulting primitives written into the "cabin" file immediately. We'll need to remember to change the material types for the north and south roofs, and the north and south ceilings. This is also a good time to change the names of the roof surfaces to meaningful names (north_roof, south_roof, north_ceiling, etc).

Roof Materials

```
# Cabin Roof Materials

void plastic white_enamel
0
0
5       .5      .5      .5      .02     .08

void plastic light_wood
0
0
5       .5      .3      .2      0       0

void plastic north_shingle
0
0
5       .3      .2      .1      0       0
```

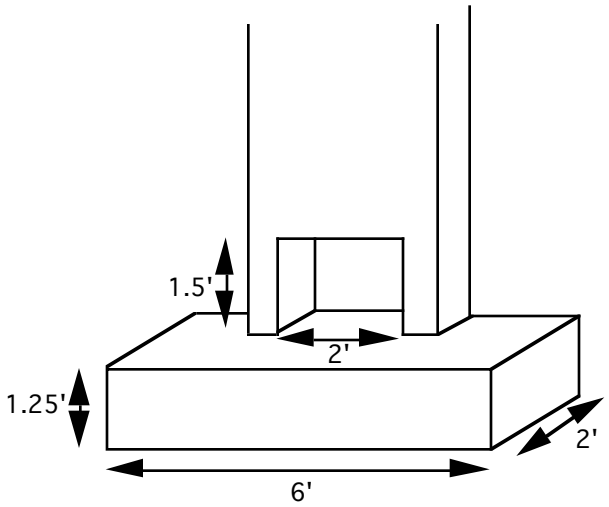
Fireplace



The fireplace is simply a combination of genbox-generated surfaces, with a hole in the chimney for the hearth and fireplace opening. The tricky part of building the fireplace is remembering to invert the fireplace opening surfaces so their surface normals point inwards, and to cut "holes" in the boxes where the fireplace opening is.

Once the fireplace box has been inserted into the chimney box, we can comment out the surface of the fireplace that should open out into the inside of the cabin. Even after we've done this, we'll need to remember to open up the corresponding portion of the chimney box.

Cabin Fireplace

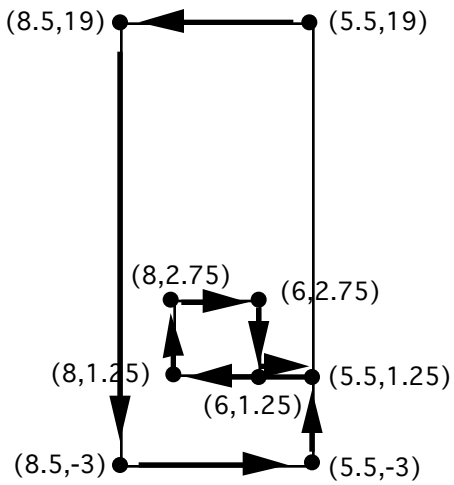


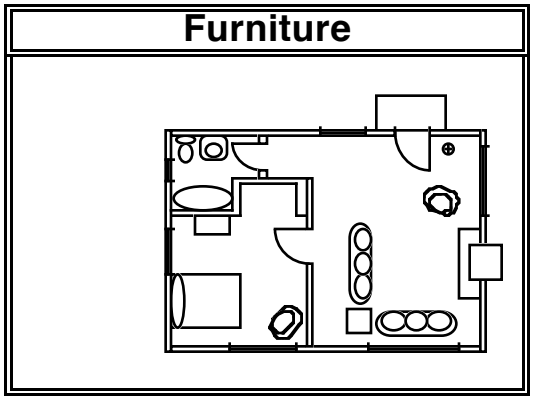
Generating a Fireplace with genbox

```
:r !genbox cinderblock fireplace 1.5  2  1.5  -i \
        | xform  -t  26.25  6  1.25
```

We can insert the necessary vertices into the chimney's polygon description, using the right hand rule (surface normal points into the room, so points need to be entered counter-clockwise around the outer edge of the chimney surface).

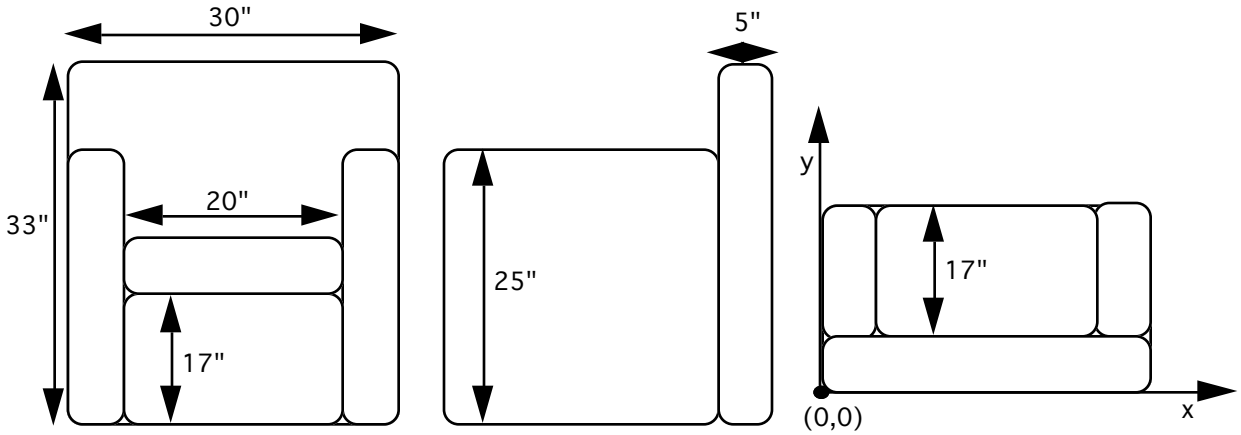
Cabin Fireplace



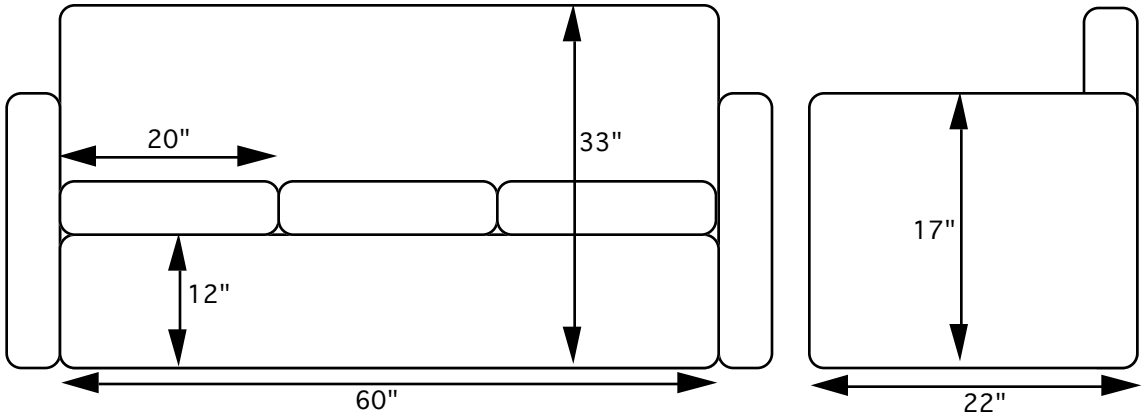


The cabin furniture can be made quickly from genbox commands, for the most part. The end table in the livingroom has a two foot by two foot square top (two inches thick), and has two inch by two inch legs. The genbox "-r" rounding radius option is very useful in creating softer looking cushions for the sofas, chairs, and the bed (simply follow the "-r" with the rounding radius desired: "-r 1"). The surface primitives for the sofa and chair can be put into files called "sofa.norm" and "chair.norm", to more easily accommodate placement of the completed objects (since there is more than one sofa and chair) in the cabin scene using xform.

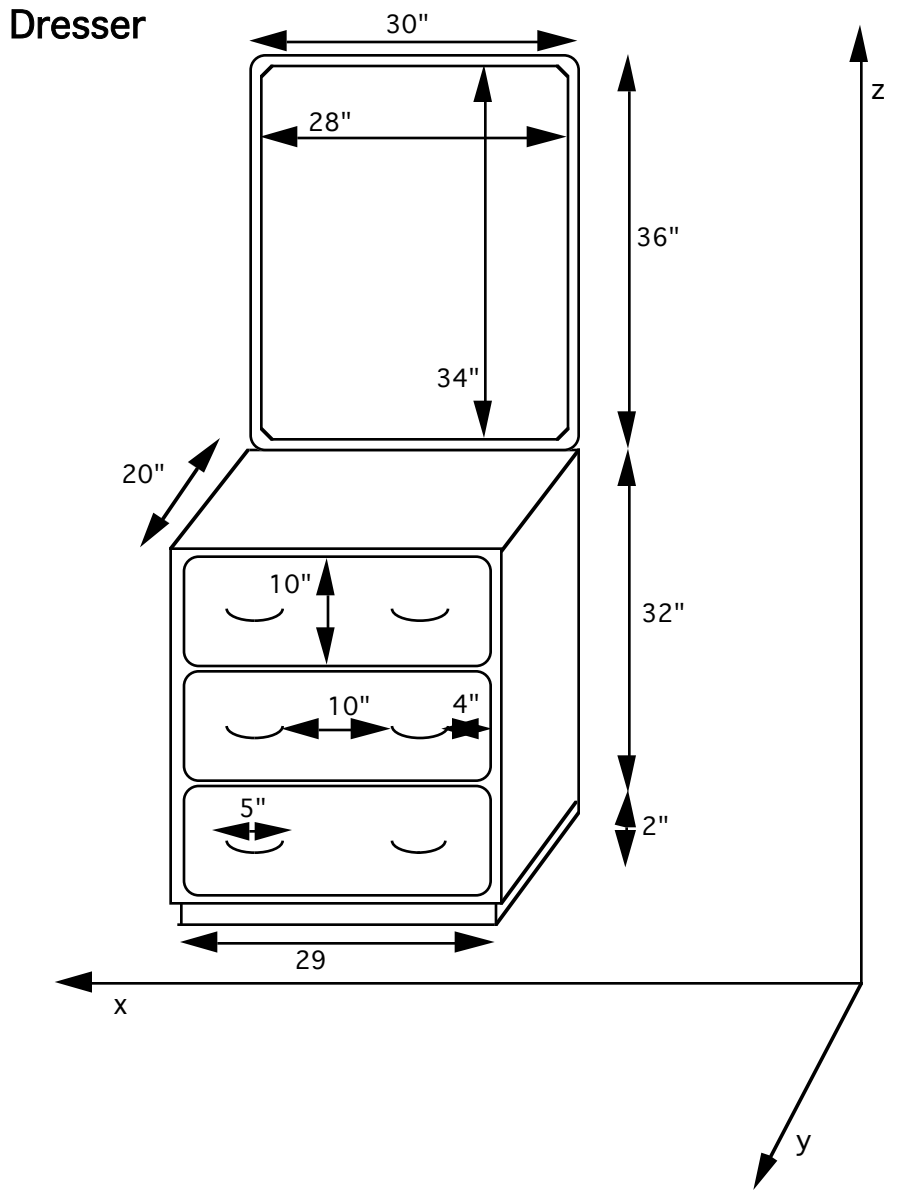
Cabin Chair



Cabin Sofa

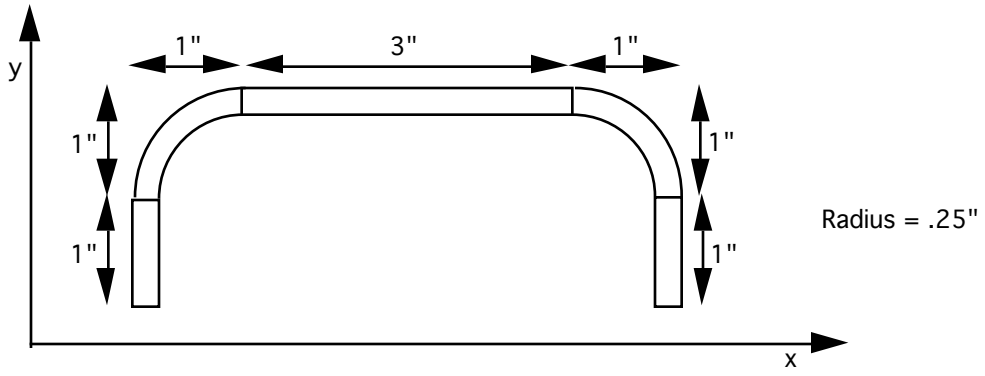


The most interesting piece of furniture, and by far the most complex, is the bedroom dresser. We can create a separate file for the dresser ("dresser.norm"), to keep this dresser for possible future use in other scenes. Since we'll want to add a woodgrain pattern to the dresser later on, we can edit the output of genbox to change the material type identifiers as we create the dresser. Woodgrain patterns run in a straight line; we'll need to select which way we want the grain to run for the dresser and livingroom table (eg: xpine along the x-axis for the top and front of the dresser, and zpine along the z-axis for the dresser's sides). Another detail to keep in mind with the dresser is the mirror surface; it should be located just in front of the wooden frame (which can be made of zpine).



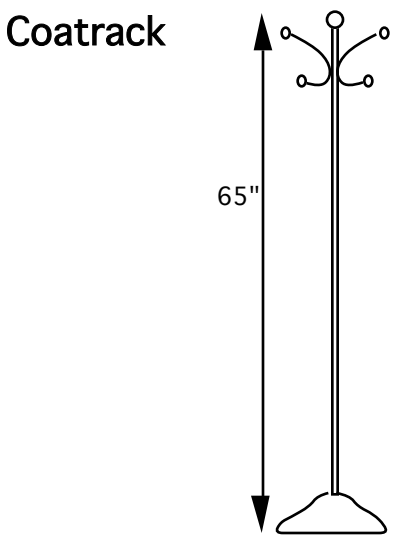
Most of the dresser can be built using the genbox commands, but the brass handles have curved corners than can't be built with cylinders or boxes. The handles can be thought of as consisting of five separate pieces: three cylinders, and two quarter-circles. We'll want to create a temporary file for this handle and one for the drawer, so we can easily place the handles on the drawers, and then place the drawers on the dresser.

Dresser Drawer Handle



Genworm can be used to create the quarter-circles, since we know the equation for a unit circle (from 0 to 2 π) to be: x=cosine theta, y=sine theta. The right and left bends on the handle can be created separately, since they correspond to different parts of the circle equation (zero to π/2, and π/2 to π). The x(t) and y(t) functions are given in terms of the unit circle formula, as the circle ranges from zero to π/2, and z(t) is zero. The radius of the handle is .25 inches, and five segments will be created (the curve isn't perfectly smooth).

A coatrack has been included in the cabin scene, to provide us with an opportunity to practice both our genworm and genrev commands. The coatrack stands 65 inches tall, and has a curved base that measures 14 inches across and 5 inches in height. The four curved top portions of the coatrack have two prongs each, the top measuring 4 inches extended from the center pole, and the bottom measuring 3 inches. The unusual shape of our coatrack necessitates that we guess at the formula required for specifying its curve; we can use a hermite curve function, a third order polynomial, to assist us.



We'll specify two hermite curves to describe the bell-like surface of the coatrack base, because the base has two inflection points (it's a combination of two S-shaped curves). We'll give genrev the functions describing the outline of the base's curve, and genrev will break up the surface of revolution into a number of segments, each of which is a cone. We'll give genrev the functions z(t) and r(t) along with the number of segments we want, and we'll use hermite functions to define the z and r functions.

We know the starting point for our coatrack base curve is at (z,r)=(0,7), and the final end point of the second curve is at (z,r)=(5,0). We can pick some point in the middle as an end point for the first curve, and a starting point for the second curve, such as (z,r)=(3,2). We can now define our hermite curves for genrev, either with two separate genrev commands, or one combined genrev command. The advantage of using one combined genrev command is that we can avoid a base with a crease in it, where two genrev command curves meet. The combined genrev command requires an "if then else" format, in order to specify which curve should be used as the independent variable t increases from zero to one. The connecting point for the two curves occurs when t is approximately .7, so for values of t equal to or less than .7, we want to use the first hermite curve, and for values of t greater than .7 we'll use the second hermite curve. The "-e" option in the genrev command indicates that an expression follows (in this case, our hermite curve functions). The "-s" option in the genrev command indicates that we desire smoothing for the generated curve, so the ridges between

segments don't show up as sharp corners. The starting and ending direction vectors are picked by trial and error; those shown here are the final values used in the cabin scene.

genrev Command for Brass Base

```
!genrev brass base 'if (t-.7, z2((t-.7)/.3), z1(t/.7))'\
        'if (t-.7, r2((t-.7)/.3), r1(t/.7))'  11  -s\
        -e  'z1(t)=hermite(0,3,5,3,t); r1(t)=hermite(7,2,0,-1,t)'\
        -e  'z2(t)=hermite(3,5,3,0,t); r2(t)=hermite(2,0,-1,-5,t)'
```

The brass hooks can also be generated using hermite curves, since once again, we know start and end points, but have no real function for how the hooks curve around. The hooks will be generated with genworm, and once again we'll just use trial and error to get the correct starting and ending direction vectors. Since there is more than one hook (there are four), we can create a "hook" file, and use xform to place the hooks around the central cylindrical pole. For our hook, we'll work in the xz plane, so that all y values are zero. We'll use (x,z)=(4,6.5) for the top part of the hook, and (x,z)=(3,2) for the bottom part, working in an xz plane that will have to be adjusted by adding 56 to all z values before we place the hook on the pole. The radius of the hook will be .25 inches, and we'll use 10 segments.

genworm Command for Brass Coat Hook

```
!genworm brass hook 'hermite(3,4,-12,12,t)' '0'\
        'hermite(58,62.5,-9,6,t)'  .25  10
```

Since we've written the coatrack hook data to a "hook" file, we can now use xform with it's array option to place four of these hooks evenly around the top of the coatrack pole. We can use the "read" command in the vi editor to bring the results of this command into our file for the coatrack, where the results are named "hook", and the "hook" file is read in as input:

Using xform Array to Place Hooks

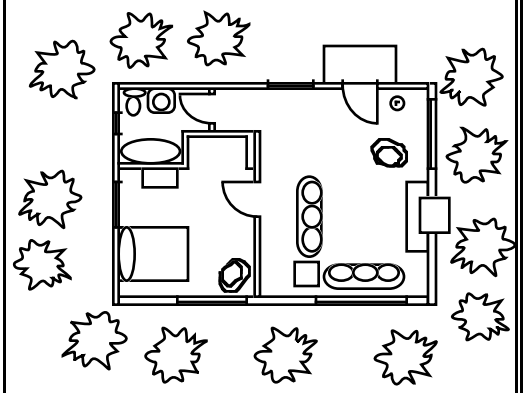
```
:r  !xform -n hook -a 4 -rz 90 hook
```

We can finish the coatrack by putting spherical brass balls at the ends of our hooks, and on the top part of the coatrack pole.

Instances

The environment immediately surrounding our cabin model provides us with the opportunity to use instancing (for planting the trees), and one very large polygon for creating the ground surface surrounding the cabin. We saw how instances of trees could be taken from the "tree.oct" file in the example given earlier in this chapter, and planted wherever we wanted them in our scene (be careful not to plant any trees inside the cabin).

Environment



The ground we'll be planting trees in is a flat polygonal surface, centered around the cabin's foundation, whose center coordinates are (13,9,-1.5), and extending a sufficient distance in every direction (about 100 feet from the cabin to each edge) to give the appearance of continuous ground around the cabin.

Textures

One of the simplest textures available for our use is one that perturbs all three coordinates randomly with a three-dimensional noise function, without requiring us to keep track of surface normal orientations. We can use this "noise3" function to add a puckering texture to our bedroom bedspread. We'll start by writing the "puckered" primitive, which is a texfunc, and using it to modify our "rose_spread" material on the bed. We start by naming the three variables we'll need for perturbing our three dimensions, "puck_dx", "puck_dy", and "puck_dz", and naming the calculation file where we'll define these

Puckered Bedspread

```
void texfunc puckered
6 puck_dx puck_dy puck_dz \
      pucker.cal -s .5
0
1  .2

puckered plastic rose_spread
0
0
5   .4        .03      .03      0         0
```

functions, "pucker.cal". We'll space the textures about six inches apart, on average, so we'll use the "-s" scaling option with .5 feet.

The Texture File "pucker.cal"

```
{
        This puckered texture is used on the Cabin
        Bedspread to provide it with a bumpy look.
        A1      - The degree of puckeredness
}
puck_dx = A1 * noise3a(Px,Py,Pz);
puck_dy = A1 * noise3b(Px,Py,Pz);
puck_dz = A1 * noise3c(Px,Py,Pz);
```

Our "pucker.cal" file defines each of the three coordinate variables ("puck_dx", "puck_dy", and "puck_dz") in terms of our degree of puckeredness, A1, entered as the real variable with a value of .2 in our texfunc primitive above. The noise3 function has an autocorrelation distance of 1 and a magnitude of 1 (ranging from values of -1 to 1) which we multiply our A1 variable by. The comments for ".cal" files are enclosed in curly brackets.

Patterns

Our cabin model contains a variety of possibilities for using patterns; we can hang pictures of previously scanned images (or Radiance images) on the walls, use brick patterns for the chimney and porch, apply an oak floor pattern to the floors, put a shake pattern on the roof for shingles, and use a procedural woodgrain pattern on the bedroom dresser and livingroom table.

The picture on the livingroom wall will be taken from a scanned image of a photograph taken in Alberta, called "alberta.pic". The x and y dimensions for the picture can be displayed with the:

```
getinfo -d!$
```

command, when in the directory containing the image. In this case, the y dimension is 231, and the x is 150. The smaller dimension is scaled to be equal to one, so the larger dimension is 231/150.

We'll start by putting a flat polygonal matte behind the picture, to prevent bleedthrough of the wall into our picture. We're going to

blow the image up to a width of three feet, for a poster-sized image. The poster should be placed just slightly in front of the wall (which is located at x=12.25), and we can figure out the height of the poster by multiplying the 231/150 ratio by our width of three feet (to get 7.62 feet).

Photo Paper Material

```
void plastic photo_paper
0
0
5  .82  .82  .82  0  0
```

We'll use a material called "photo_paper" to print our photo on, and we can save this primitive in our "materials" file. We can now use this same material for any other pictures we'd like to use in the cabin (such as a forest picture called "richgrove.pic" in the bedroom).

The data pattern that takes an image, such as our picture file, is called colorpict. We'll call our poster "alberta_image", based on the "alberta.pic" scanned image. The file containing standard coordinates for pictures from scanned patterns is called "picture.cal", and is stored in the system Radiance directory (/usr/local/lib/ray, on most machines).

Alberta Poster Data Pattern

```
# Pictures

void colorpict alberta_image
17  clip_r  clip_g  clip_b  alberta.pic picture.cal\
        pic_u pic_v  -s 3  -rx 90  -rz 90\
        -t 12.251  2.5  3

alberta_image alias alberta_photo
        photo_paper

alberta_photo polygon alberta_poster
0
0
12      12.251          2.5     3
        12.251          5.5     3
        12.251          5.5     7.62
        12.251          2.5     7.62
```

The pattern primitive contains a scaling factor of 3, since the "alberta.pic" data file is scaled to one, and our poster is three feet wide. The red, green, and blue values are "clipped", so that they won't exceed reflectance of one and start glowing, and the "pic_u"

and "pic_v" variables indicate that we want to start our picture at the xy intersection point (0,0), face-up.

Our wood floor pattern comes from a data file called "oakfloor.pic", which has y/x dimensions of 96/166. This image contains seven slats, and we'll scale this smaller dimension so that these seven slats measure 14 inches (s=1.1667). The larger dimension will then be 14*(166/96), or 24.208 inches. We had already defined a primitive for the wood_floor in materials with a modifier of void; now we will change the word "void" to "oakfloor_pat", and change the RGB values to an overall reflectance of 20%, so the material has a neutral effect at first (we can change the values later on, to adjust color intensities).

Wood Floor Pattern

```
void brightfunc dusty
4  dirt  dirt.cal  -s  2
0
1  .15

dusty colorpict oakfloor_pat
9  red green blue  oakfloor.pic  picture.cal
          tile_u tile_v  -s 1.1667
0
1   .578313253

oakfloor_pat plastic wood_floor
0
0
5  .2  .2  .2  .02  .05
```

The one real argument in the "wood_floor" primitive is the aspect ratio of our oakfloor pattern (96/166), and tile_u and tile_v need these ratio figures to place the pattern on the floor. The "dusty" primitive throws dirt on the floor, making a slightly uneven (and therefore, more natural) coloration. The dirt periodicity is spaced every 2 feet, and the degree of dirtiness is 15%.

The roof shingles can be applied to the roof in a similar fashion, for the most part, except that we need to pay special attention to orienting the pattern to match the angle of the roof (45°), and the direction (either north or south facing) will require rotation about the z-axis for the north shingles.

The woodgrain patterns for furniture, such as the bedroom dresser and the livingroom table, can be generated procedurally, using brightfunc. We have already given careful consideration to the direction we want our woodgrain to go for the dresser and table, using the material names xpine, ypine, and zpine to correspond to the direction of the grain. Now we can name our procedural wood patterns names like "ywoodpat", to correspond to the grain that runs along the y-axis. The scaling of .025 is the distance between the woodgrains we've set, and we'll use a darkness magnitude of 30%. We can use aliases to relate the wood patterns with our xpine, ypine, and zpine material modifiers.

Procedural Woodgrain Pattern

```
void brightfunc xwoodpat
4 xgrain woodpat.cal -s .025
0
1 .3

xwoodpat plastic xpine
0
0
5      .7      .25      .08     0       0

void brightfunc ywoodpat
4  ygrain  woodpat.cal  -s .025
0
1  .3

ywoodpat alias ypine
        xpine
```

Light Sources

Until we add a light source, our scene will appear completely dark. It's generally a good idea to add the sun as a light source, and the sun's light contributions to our scene will depend on where and when the scene is being viewed.

One of the first input files we'll want to build before we start generating images, regardless of whether or not we've used a CAD system, is the file containing the information about our scene's location. Radiance uses this information to correctly position the sun for the location and time provided. Our cabin is located in Jasper, Alberta, with lattitude 52.53° and longitude 118.05°, and

standard meridian 105°. Radiance has defaults set for the San Francisco bay area, so we'll need to provide gensky the information about our cabin's location. We'll select a date and time of July 12th at noon for a summer day, and put all the information into a file called "summerday". We can easily create other files based on different times of day and year, to view our model under a variety of daylight conditions.

In addition to the gensky command, the sky distribution is given as a brightness function, skyfunc, and defined in two primitives. For a hemispherical blue sky, a skyfunc primitive for a blue sky ("sky_glow") is defined, and then used to modify the source called "sky" that has a hemispherical (180°) shape. There can also be some light from the ground, which we can call "ground_glow", eminating from a source called "ground".

Summerday File Information

```
# The sky over our little cabin scene in Jasper, Alberta
# Canada at 12 noon (standard time) on July 12th.

!gensky  7  12  12  -a  52.53  -o  118.05  -m  105

skyfunc glow ground_glow
0
0
4  1  .8  .5  0

ground_glow source ground
0
0
4  0  0  -1  180

skyfunc glow sky_glow
0
0
4  .8  .8  1  0

sky_glow source sky
0
0
4  0  0  1  180
```

The cabin windows currently don't include interreflection calculations from the sky, so we'll add an optional optimization (illum) to the windows to turn them into secondary light sources.

This addition of secondary light sources is not really absolutely essential for viewing the scene, but it does improve the efficiency of the program's lighting calculations, and since we know that some light is coming through the windows after first bouncing around, our simulation will benefit from our use of this knowledge. Without any light sources whatsoever, everything in our scene is dark. With just the sun, the program checks to see if the object is in shadow; if not, the program calculates brightness at that point. If the object being viewed is in shadow, the program uses the constant ambient value we specified. This results in the scene appearing to consist of flat colors, if only ambient light exists with no direct light. The ambient value for outdoor scenes must correlate to the actual amount of light bouncing around, which is much higher for outdoor scenes than indoor scenes.

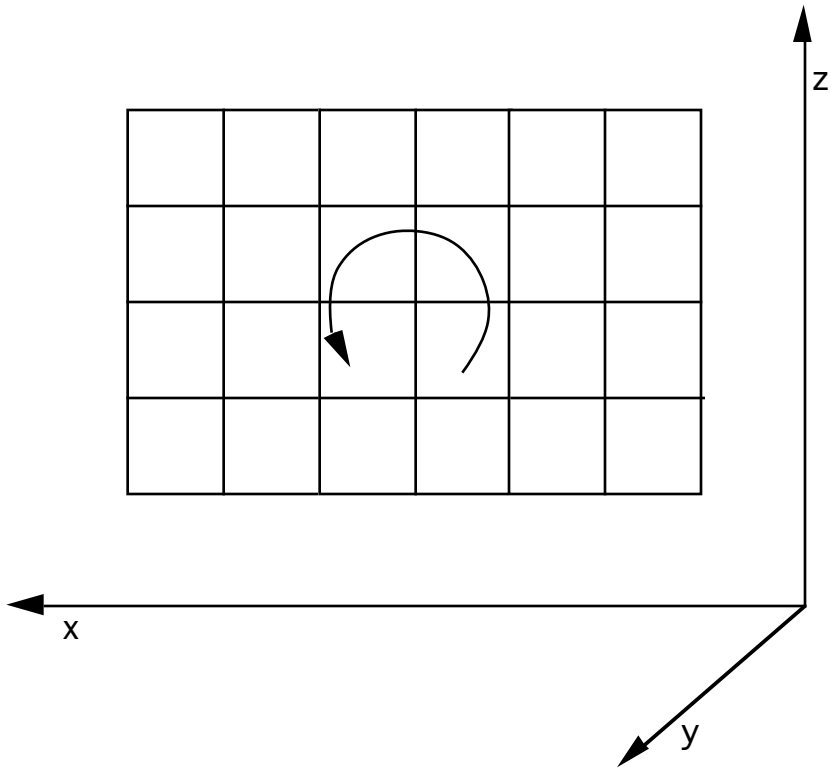
Currently, our cabin model has windows that only let in the direct light from the sun, and doesn't allow for the windows to provide an additional brightness from interreflection. If we give the window panes the same brightness as the sky at that point, we will help to illuminate more than just the bright patches of sunshine from direct sunlight in the cabin's interior. We still want to be able to see outside the window (and not be viewing a white light source), so we'll use illum. Illum allows us to see through the window panes, and allows other rays in the calculation to pass through it so the sun's rays still come through, even though illum is a light source. We'll use "plain_glass" as the default glass material, which will be visible whenever being viewed through (so we can see to the other side). To give the windows the same distribution as the sky, we'll use a modifier (pattern and brightness) with gensky. Gensky always produces a description of a modifier, called skyfunc, for that particular time of day and year.

Changing Window Glass to Illum in "daywindows"

```
# Adding illumination sources to cabin windows
# to provide sky light for the cabin scene interior.

skyfunc brightfunc winbright
2  winxmit  winxmit.cal
0
0

winbright illum window_illum
0
0
3  .88  .88  .88
```

It's important to note that the glass surfaces must be oriented with their surface normals pointing toward the interior of the cabin space, to function properly as light sources for the interior. We'll use the right hand rule to determine surface normals for these window illumination sources, and we can use gensurf to create these illuminating windows, since we may decide later on to change the number of illumination sources per window from one to some more accurately representative number (although that will sacrifice speed and efficiency for accuracy).

Illuminating South Facing Windows



The above diagram shows how the surface normal points towards us (the thumb) when we curl our fingers up along the z-axis, and then out to the left, along the x-axis. For the southeast and southwest windows then, t in gensurf will correspond with the x-axis, and s with the z-axis.

Polygons for Window Illumination

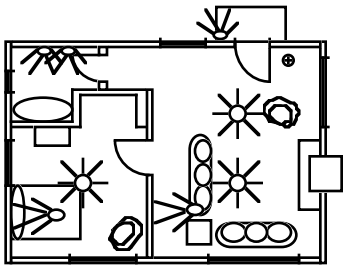
```
# Polygons for window illumination surfaces,
# for the south_east window, using gensurf.

!gensurf window_illum southwest_window \
        '5.5 + t*(10.5-5.5)'  '0'  '3 + s*(6-3)'  1  1
```

Gensurf provides us with a very useful way to break these illum light sources into sub-sources later on, since gensurf defines a curved surface which is defined parametrically in terms of (s,t) functions. Since our s and t correspond to z and x, all y-values will be zero.

The width of the window along the x-axis varies from 5.5 to 10.5 feet, and the window's height along the z-axis varies from 3 to 6 feet as s and t both vary from zero to one. A six-inch allowance has been made along the x-axis for the window sills. The number of s divisions and t divisions will be one this time, since we're starting off with one illumination source per window.

We can add lights to our cabin scene using both IES files and our own light source creations. We can put floodlights in the bedroom and livingroom of the cabin to illuminate the posters, and use overhead lights in the bedroom and livingroom for general illumination. We can use the same sorts of lights to mount on the side of the wall in the bathroom and over the front porch, giving us a total of three different types of lights in this scene.

Cabin Lights



We can use an IES file ("ies06") that contains information for the floodlights, and run the ies2rad program from the command line to convert the ies file to Radiance format. We can rename the file "flood.ies", to provide a slightly more descriptive name.

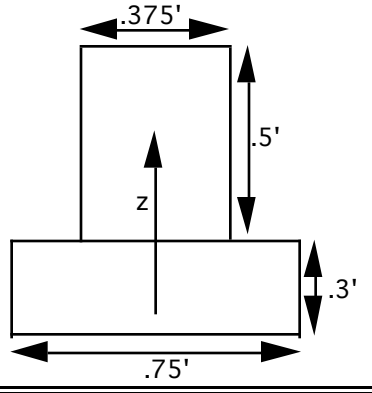
Converting IES Files to Radiance Format

```
%ies2rad -m .25 -df -t default create/flood.ies
```

The "-m" option in the ies2rad command indicates that a multiplier of 25% is being used on the ies file values; the "-df" option specifies that we want our dimensions in feet (the default is meters); and the "-t" option stands for lamp type information (such as color, depreciation, and lamp specifications). The "default" indicates that we want a white fixture (so we would need to provide color with the "-c" option, if we didn't want white). The "create/flood.ies" grabs the input file "flood.ies" from our create subdirectory and puts the output files in our current directory (unless we use the "-l" or "-p" options), automatically naming the output files "flood.rad" and "flood.dat".

We'll want to surround the light source with a fixture of some type, so we'll build a shroud for the floodlight in a file called "flood.shroud", using two cylinders and two rings. The cylinders and rings will rise up along the z-axis, and we can consider the z origin to be near the bottom of the shroud, perhaps .05' inside of the larger cylinder (since we'll want our light recessed into the shroud just a bit).

Floodlight Shroud



The next step will be centering the floodlight over each of our posters, which are centered at (x,y,z)=(12.251,4,5.31) and (.001,4,5.2205) for the livingroom and bedroom posters, respectively. We'll be placing the floodlight in our "lights" file, using the "flood.rad" and "flood.shroud" files as input. Assuming that we want to place our floodlights about 3 feet away from the posters (in the x direction), we can determine the desirable height above the poster to be about five feet. Once we're happy with where we want to place the floodlights, we can use xform to name the light, drop the floodlight down to center (where the light source will be recessed), and rotate and place the floodlight right where we want it. We'll also want to add the finishing touch of a support for the floodlight, so it doesn't appear to be dangling in thin air.

Bedroom Spotlight in "lights" file

```
!xform -n bedroomspot -e -t 0 0 -.5 -ry 30 -t 3 4 11\
        flood.rad flood.shroud

brass cylinder bedroomspot.support
0
0
7       3       4       11
        3       4       12.1
        .02
```

Bedroom Overhead Light

```
void light big_globe_light
0
0
3   10   10   10

big_globe_light sphere bedlight
0
0
4   6        5.5     8        .5

brass cylinder bedlight.support
0
0
7   6        5.5     8.5
    6        5.5     13.6
    .02
```

The bedroom overhead light can be created fairly quickly, using the radiance calculation discussed earlier in this chapter for a spherical source. Our 100 watt bulbs of 6 inch radius have an output of 1750 lumens, and if we work through all the calculations for radiance, we end up with RGB values of 10 w/sr/m^2. We can now create the primitives describing our bedroom overhead light that describe the light source, the shape of the spherical source, and the brass support pole that attaches the overhead light to the ceiling. The two overhead lights in the livingroom can be built exactly the same way we built the bedroom overhead light, using the "big_globe_light" modifier for the spheres used in the livingroom, and adding their supports.

Our front porch light and the identical bathroom lights can be created similarly to how we built the overhead lights, with the exception that they will be side-mounted to walls with a brass cone. The small globe lights used for the porch and bathroom can be given greater output (using 14.5, instead of 10 for RGB values).

Image Generation

Introduction

Ray tracing simulates the behavior of light by tracking the path of light from its presumed destination to one or more sources, following the ray's path around the scene. Image generation is the process where this simulation takes place, based on the input files provided to the ray tracing software that contain the material, object, and light source descriptions.

The most frequently desired output, and the type for which Radiance is tailored, is the color image. A color image is a two-dimensional array of radiance values rendered by projection from a three-dimensional scene, similar to a photographic image. The Radiance values are broken into spectral components, nominally red, green, and blue. The renderer RPICT produces picture files in batch mode, and RVIEW calculates and displays images interactively. RTRACE provides other lighting calculations and information, such as illuminance or light source distributions.

There are commonalities between the Radiance rendering programs (RPICT, RVIEW, and RTRACE), which share many of the same command options (please refer to the RPICT manual page for a list).

Scene Compilation

Radiance uses octrees to generate its images in the most efficient way possible, since this technique handles complex environments very well by sorting objects in the scene before the rays are traced. In an octree sort, the space in the scene is recursively divided into cubes which contain no more than a certain number of objects specified to the program. A ray is then followed through the resulting "octree", and intersection calculations are performed only on those objects which lie in the cubes intercepting the ray, instead of across the whole scene. Octrees can greatly reduce the time required for ray-tracing complex scenes; the time required for this type of intersection calculation increases as the cube root of the number of objects, making it suitable for very densely crowded environments.

OCONV is used to create an octree from a list of object files. Each surface is recursively intersected with the cubes in the growing octree until all objects have been done and the octree is complete. The requirement that surfaces be intersected with cube faces makes the task of implementing a new surface type somewhat more involved than it would be without an octree conversion. This drawback is more than compensated for by the increased speed during ray tracing using an octree sort. Options for changing the maximum octree resolution and the maximum number of objects per elemental cube are provided. Optimal speed of ray tracing is related to these values, but they are not critical.

The user should be aware of possible errors that may arise when using OCONV, as well as OCONV formats. It is important to use the correct order of files when running OCONV, so that modifiers are defined before they are used. If the files are out of order so that materials and other modifiers are referred to before they've been defined, an error message is printed about an undefined modifier, and OCONV aborts.

This example oconv command creates an octree called "oct/cabin" from the files called: "materials", "cabin", "bathroom" and "furniture".

Creating an Octree with OCONV

```
%oconv materials cabin bathroom furniture > oct/cabin
```

Octrees that are *frozen* contain data structures that can no longer be changed. The "-f" option of oconv freezes the octree by storing all the octree information as it currently exists in a binary file format at the end of the octree file (roughly doubling the file size), which facilitates faster scene loading. The chief disadvantage of freezing octrees is that any changes made to materials won't be reflected in an octree that has been frozen, so if you want to change a material to be more to your liking, you'll have to redo the octree as well.

When oconv is run, it creates a "bounding cube" minimum cube that surrounds the whole scene. The example of the octree for the cabin shown above has created a boundary that surrounds the cabin, but

does not include the forested areas to be added later around the cabin. When the forested areas are added, the bounding cube area will grow larger.

Unfrozen octrees are dependent on scene files; scene files are needed any time the octree is to be used. Care is needed any time files related to an unfrozen octree are changed or moved, to ensure that the octree will still be intact when needed. Frozen octrees are the best way to create libraries of objects, since they have the advantage of not requiring all the scene files to be stored with them (one frozen octree file contains all the information required to generate that scene).

Octrees are meant to be machine independent. OCONV creates binary, machine-independent files, so it's simple to move Radiance files from one machine to another.

Batch

RPICT follows rays of light through an image plane and into the scene compiled in an octree, to produce a high-quality picture file. To produce such a high-quality image of a scene, one starts by selecting a view point and view direction, much as we would do if we were to walk into the scene and take a picture with a camera. Generating an image can take anywhere from a few minutes for a low resolution test to many hours for a publication quality image; RPICT may be run overnight to produce high resolution pictures suitable for presentation.

RPICT Program variables fall into several categories:

View	The direction and field of view are selected for the image desired. A *view up* vector determines the orientation of the picture.
Resolution	The horizontal and vertical resolution determine the detail of the picture, and strongly influence the time required for its computation. Image plane sampling reduces the time for high resolution pictures. By specifying a grid of sample points, the program will only trace rays between points if they differ by more than a specified amount. Jittering rays through each pixel avoids aliasing at high contrast edges.

Direct Calculation	Normally, sources are treated as if they eminate from a point. Jittering a ray to a source by an amount proportional to the source's size produces a more accurate rendering of the scene. Unfortunately, this technique causes problems for image plane sampling, since adjacent pixels receive slightly different values. Pictures using source sampling therefore take longer than the same picture using point sources. New "-dj", "-dt", and "-dc" options are available; please refer to the RPICT manual page for a list.
Indirect Calculation	The ambient, or interreflected component, is calculated using a Monte Carlo technique. Each calculation produces a specified number of rays. Computed ambient levels are stored in a table and used for interpolation on nearby values. The distance at which an interpolated value is considered valid is set by the user. This method produces accurate interreflected components at a modest expense.
Limits on Recursion	To prevent tracing unnecessary rays, limits are placed on the spawning of new rays. A limit on the depth of recursion prevents more than a given number of reflections to occur. A minimum ray weight stops rays whose contribution to the final value would be below a certain level.

Based on our chosen view point and view direction coordinates, we can either calculate the view direction vector by subtracting the view point coordinates from the point being viewed (also called the "look at point", or the "view center"), or we can estimate some view direction vector that's close enough.

RPICT Example

```
% rpict -vp 5 4 6  -vd 0 1 0  test.oct > test.pic &
```

The "-vp" option stands for view point, and the "-vd" stands for view direction. This example shows how a view direction of looking due north up the y-axis has been selected, with a direction vector of (0,1,0). The resulting Radiance image is written to the file called "test.pic", and the "&" option indicates that we wish this job to run in the background, since it may take a while to complete. Once

you've started this job running in the background, you can log out and go home if you wish, and the job will keep running.

You will see a bracketed number printed by the C-shell command interpreter immediately preceding the RPICT command; this is the process id [PID] that can be used later to check the progress or kill the program. If for any reason you wish to kill the program, you may do so using the "kill" command, and following it with the process id number (12 in this example).

Killing an RPICT Process

```
% kill 12
```

When running a long RPICT job, it's a good idea to provide RPICT with an error file (named something like "errfile") where it can write out any errors and progress reports during the rendering.

Specifying an RPICT Error File

```
% rpict -e errfile
```

Progress and error reports can now be checked at any time by printing the file name you specified (in this case, "errfile") with the "cat" command.

Viewing an RPICT Error File

```
% cat errfile
```

RPICT has a "-av" option for setting the ambient value, but this option is best left to the default (of zeroes) if we don't have any idea what it should be. A correct ambient value selection produces shadows that aren't too light or too dark, but look pretty realistic for the scene.

If you have a view file you wish to use for generating a high-resolution image in batch mode, you can do so using the "-vf" option and providing RPICT with the name of the view file. If you

know the ambient values for the scene, those can also be specified with the "-av" option.

Using RPICT with a View File
% rpict -vf vf/myview -av .5 .5 .5 test.oct > test.pic &

Interactive

Interactive image generation is an important feature of Radiance that permits quick error checking, view determination, and lighting evaluation. Few things are more frustrating than waiting several hours for an image, only to discover errors in the input. In a matter of minutes, RVIEW produces a low-resolution image clear enough to spot any serious mistakes, and gives the user mobility to find errors visible only from certain vantage points. One may increase the resolution selectively to concentrate on more interesting parts of the image. The same freedom of movement and study permits crude lighting and visibility evaluation (eg: contrast and glare). Since a full calculation is being performed at a reduced resolution, absolute accuracy is maintained. Luminance values may be queried at any point in the scene, with instant results. Once the scene is in order and one or more good views have been determined, RPICT may be run overnight to produce high resolution pictures suitable for presentation.

RVIEW Example
% rview -vp 2.25 .375 1 -vd -.25 .125 -.125 -av .5 .5 .5 test.oct

Just as we used them in RPICT, the "-vp" and "-vd" options can be used to specify the view point and view direction vector, and the "-av" option specifies the ambient light present in our scene. A list of RVIEW's options and their default values may be reviewed using the following command:

Viewing RVIEW Option Defaults
% rview -defaults

Once you've found a view point and direction that you especially like, take the opportunity to save that view using the "view" command from within RVIEW, and select a name for that good view. Many different views can be saved this way, and higher-resolution images produced later on.

Saving a View in RVIEW
: view vf/ext.se

In the following example, the "-vf" option indicates that we will be getting view information from a file, and the "vf/ext.se" immediately following is the file name for the view file we saved. The octree being used is "oct/dayext", and the "&" symbol indicates that we want to start this rview process in the background, so we can check on it later.

RVIEW Example
% rview -vf vf/ext.se oct/dayext &

Other Lighting Calculations

Of course, images are not the only output desired from a lighting simulation. RTRACE provides a convenient interface for obtaining other kinds of information from the calculation. Individual radiance values may be computed and combined to get luminance, contrast rendering factors, equivalent sphere illumination, glare indices, or any other lighting metric. Besides radiance, irradiance, intersection point, surface normal direction, and other calculations are available. Typically, RTRACE is run from a shell script or other program. For example, the utility MAKEDIST calculates a source distribution from a geometric specification by calling RTRACE. By connecting certain display drivers to RTRACE, additional information may be computed interactively at selected image locations, such as the illuminance on a surface or the ray propagation tree.

Tutorial Example

Scene Compilation

Since we'll want to create many different octrees for viewing the interior and exterior of the cabin at different times of the day and year, we'll start by creating a basic cabin interior octree called "oct/cabin". The following oconv command creates this octree from the "pattmats", "cabin", "bathroom" and "furniture" files. This octree can now be used as a starting point for all future generated interior images of the cabin.

Creating a Cabin Octree

```
% oconv pattmats cabin bathroom furniture > oct/cabin
```

A second oconv command uses "oct/cabin" as input to change the octree into a frozen octree, and add new surfaces from the "summerday" and "daywindows" files, and to create the octree called "oct/summercabin". The "-i" option indicates that the input octree will be named next ("oct/cabin", in this case). The resulting octree will provide us with everything we need to generate images or perform calculations on the cabin, taking into account our illuminated windows and the gensky for a summer day.

Creating a Summer Day Octree

```
% oconv -f -i oct/cabin summerday landscape daywindows \
        > oct/summercabin
```

Provided that we had the gensky information for a winter day stored in a file called "winterday", we would be able to create a cabin octree for a snowy day in January:

Creating a Winter Day Octree

```
% oconv -f -i oct/cabin winterday landscape daywindows \
        > oct/wintercabin
```

Batch

Once an octree for our cabin on a summer day has been created with OCONV, a high-quality image can be rendered with RPICT. To produce such a high-quality image of our scene, we'll first want to examine the floor plan and pick a view point and view direction, much as we would do if we were to walk into the cabin and take a picture with a camera.

We can choose an example view point and view direction of standing in the bedroom doorway, looking towards the fireplace. The coordinates of a person standing in the doorway are approximately (12,8.5,5.5), and the approximate point we'd be looking at is located at (27,7,5.5). We can either calculate the view direction vector by subtracting the view point coordinates from the point being viewed, or we can estimate some view direction vector that's close enough, such as (1,-.2,0) since we're looking more or less straight ahead in the x direction, and only slightly down in the y direction.

Creating an RPICT Summer Cabin Picture

```
% rpict -vp 12 8.5 5.5 -vd 15 -1.5 0 \
        oct/summercabin > pic/summercabin &
```

The "-vp" option stands for view point, and the "-vd" stands for view direction. This example shows the more precise technique of subtracting view point coordinates from the view center, but a rougher approximation would work just as well. The resulting Radiance image is written to the file called "pic/summercabin", and the "&" option indicates that we wish this job to run in the background, since we expect it to take a while to complete.

Interactive

We can use RVIEW in much the same way we would use RPICT, except that once we start RVIEW, we'll be able to issue commands to move around in the scene and change our view point and view center. We'll also be able to change the ambient light settings, to see if the shadows are too dark or not dark enough.

Other Lighting Calculations

We can use RTRACE to examine our cabin scene and calculate luminance values at various points, at different times of the day and year. These calculations can help us determine how we might wish to change the light sources in our cabin.

Image Manipulation

Introduction

Several Radiance programs assist the user in viewing and understanding program output, such as pictures and data files. A picture is essentially a luminance map with color. With each picture element, known as a *pixel*, a red, green and blue value are associated. The green value corresponds to luminance and the red and blue values add color.

The storage required by an image is a function of its resolution and its complexity. It is not difficult to produce an image which takes more than 10 megabytes of disk space. Therefore, the representation of an image in a file is an important consideration. Currently, the programs use image files which represent colors in four bytes. The first three bytes are red, green, and blue values, and the fourth byte is a common exponent. By using a common exponent, it is possible to represent very large and very small luminance values in the same picture at minimal cost. Another means of saving space used by the programs is called *run length encoding*. If many adjacent pixels in a scanline have the same value, the value and a count is given instead of all the pixels. This saves a lot of space on pictures of low complexity, and some space on complex images.

The Radiance picture file format uses a floating point representation for greater accuracy. Pictures may be scaled in brightness, resized, anti-aliased, and composited digitally with different filters provided. Drivers display the pictures on monitors or make copies on film, video, or paper. Some display programs can superimpose numerical information on images at selected locations for more accurate evaluation.

Image Display & Conversion

A device driver is needed to make a picture file viewable on the device (piece of hardware) being used. Different display programs are required for different devices. The two basic categories of devices are *display* and *hardcopy*.

A display device, such as a graphics terminal, allows quick viewing of a picture. This type of device also permits interaction. For example, the driver for the X window system allows the user to display specific luminance values at points on the image.

Hardcopy devices produce permanent records of the Radiance calculation. Sometimes a hardcopy device is available as an adjunct to a display device. If this is not the case, it is necessary to have a separate driver to produce an image remotely. A simple driver for dot matrix printers is provided as well as a more sophisticated driver for the Dicomed film recorder. Usually, film recorder output is the highest quality available, and the most expensive.

Conversion to different image file formats is provided by a number of different utilities:

RA_PR	Convert to/from standard 8-bit Sun rasterfiles
RA_PR24	Convert to/from standard 24-bit Sun rasterfiles
RA_T8	Convert to/from 8-bit Targa images
RA_T16	Convert to/from 16 and 24-bit Targa images
RA_BN	Convert to/from Barneyscan image format

Image Processing Filters

Once a picture has been produced, it is usually desirable to filter it for viewing. Radiance provides the user with several picture filters: PFILT, PCOMPOS, and PCOMB.

PFILT allows a picture to be rescaled for a specific device, and will set the exposure for the frame. Rescaling allows a picture produced at one resolution to be viewed on a device with a different resolution. The exposure is set by normalizing the picture to a given average value; this avoids graphics output which is too light or too dark. The program also performs anti-aliasing using an algorithm which defocuses the image. (This process can be time consuming for high resolution pictures.) Another option provided by PFILT is the ability to produce star patterns around particularly bright areas of the picture. This simulates what happens in a conventional camera under extreme lighting conditions.

PCOMPOS provides the user with a "cut and paste" feature for assisting image composition. It crops an image, and allows sections to be more closely examined.

PCOMB combines Radiance images with mathematical functions on pixel values.

Other Utilities

PVALUE is a program which extracts red, green and blue values from a picture file. PVALUE converts between different image representations for analysis (not display). Its output can be used by calculation programs (such as rcalc or awk) to get specific information about an image.

PSIGN generates labels.

PROT rotates images.

To obtain the illuminance at a surface, one can run through the Lambertian approximation to luminance backward. This will be exact for purely diffuse surfaces, and in error for surfaces with specularity. A simple method for obtaining illuminance values at a number of points in a scene is to include small white objects with purely diffuse surfaces in the description. The illuminance at these objects will then be equal to their luminance multiplied by π.

The program to get information about a data file is called GETINFO. This comes in extremely handy, since most data files are unreadable.

Tutorial Example

Image Display & Conversion

Image Processing Filters

Other Utilities

Advanced Topics

Introduction

Radiance is an extremely powerful and complex set of programs, whose capabilities have only been touched on briefly so far in this manual. In this chapter, some of Radiance's more intriguing and useful features will be explored.

Auxiliary Files

Auxiliary files used in textures and patterns are accessed by the Radiance programs during image generation. These files may be located in the working directory, or in some library directory. Common auxiliary file types include data files, font files, and function files.

Data files contain n-dimensional arrays of real numbers used for interpolation. Typically, definitions in a function file determine how to index and use interpolated data values.

Font files list the polygons that make up a character set. There are no comments in font files, and all numbers are decimal integers. The ascii codes can appear in any order. N is the number of vertices, and the last is automatically connected to the first. Separate polygonal sections are joined by coincident sides. The character coordinate system is a square with a lower left corner at (0,0), lower right at (255,0), and upper right at (255,255).

Function files contain the definitions of variables and functions used by a primitive. The transformation that accompanies the file name contains the necessary rotations, translations, and scalings to bring the coordinates of the function file into agreement with the world coordinates. Many variables and functions are already defined by Radiance, and they are listed in the file "rayinit.cal". The following variables are particularly important:

Dx, Dy, Dz	Incident ray direction
Px, Py, Pz	Intersection point
Nx, Ny, Nz	Surface normal at intersection point
T	Total distance traveled by ray

Rdot	Cosine between ray and normal
arg(0)	Number of real arguments
arg(i)	primitive's real arguments

It's important to use unique variable and function names, since they are stored together in the same table. Also, the key variable in a primitive using a function file must only be defined in that file, since its presence is used to determine if the file has been loaded by the program.

Using Make

The Unix Makefile is a file that provides the Radiance user with an easy way to execute commands correctly and in the proper sequence, and is extremely useful for keeping programs updated. For our purposes, a series of oconv commands and an rview command can all be executed automatically for a given view and scene description with some ambient variable settings, once we've specified these commands to the file called "Makefile".

Unix Makefile

```
#
# Makefile for Cabin Scene
#

VIEW=		-vf vf/plan

SCENE=		summercabin

AMB=		-av .01 .01 .01

view:		oct/$(SCENE)
		rview $(VIEW) $(AMB) oct/$(SCENE)

oct/summercabin:	oct/cabin summerday landscape daywindows\
pattmats
	oconv -f -i oct/cabin summerday landscape daywindows >\
oct/summercabin

oct/wintercabin:	oct/cabin winterday landscape daywindows\
pattmats
	oconv -f -i oct/cabin winterday landscape daywindows >\
oct/wintercabin

oct/nightcabin:		oct/cabin lights pattmats
	oconv -f -i oct/cabin -r 8192 lights > oct/nightcabin

oct/cabin:	cabin bathroom furniture
	oconv -b -100 -100 -100 225 -r 8192 \
pattmats cabin bathroom furniture > oct/cabin

oct/cabin:	window.norm door.norm chair.norm sofa.norm\
coatrack.norm

oct/nightcabin:		flood.rad
```

The "VIEW", "SCENE", and "AMB" variables are assigned the values we choose (in this case, a floorplan view of our daycabin scene), which are then used later in the Makefile. We can easily change these values, in order to create a different scene, or include other file information. The "oct/cabin" entries are needed for "oct/summercabin", which in turn are needed for "view". The "view" filename didn't really exist until we defined it here in our Makefile; it contains the rview command that will generate a view of our scene for us.

To get our plan view of the cabin from the summercabin octree, we now can simply type "make". A big advantage of using the Makefile

is that it saves us from typing oconv every time we make a change to one of our input files; the Makefile maintains all changes for us automatically.

Simulation Options

Animation

PINTERP does interpolation for animation.

Tutorial Example

Coordinate Mapping

The roof shingles can be applied to the roof in a similar fashion, for the most part, except that we might want to use noise functions to vary our coordinate locations so that the lines between shingles don't look too unnaturally straight. We'll also need to pay special attention to orienting the pattern to match the angle of the roof (45°), and the direction (either north or south facing) will require rotation about the z-axis for the north shingles.

North Shingle Pattern

```
dirty colorpict nshake_pat
13 red green blue shingle.pic shake.cal shake_u shake_v
        -s .7  -rz 180 -rx -45
0
1  1.7037037

nshake_pat plastic north_shingle
0
0
5  .15  .08  .05  0  0
```

The "shake.cal" file being referred to will be our own customized version of the "picture.cal" file we've used before, with shake_u and shake_v acting as our variables. We can use the smooth noise function to vary the horizontal coordinates, and fractal noise (which has higher frequencies and looks rougher) for the vertical coordinate variation. Our "shake.cal" file will need to show how these noise functions will be defined for shake_u and shake_v. The ratio of the tile height to width for our shake pattern is 1.7037037, and it's important to use all the numbers after the decimal point, in order to assure the best possible job placing the tiles. This ratio will be used as a variable (A1) in our shake.cal file.

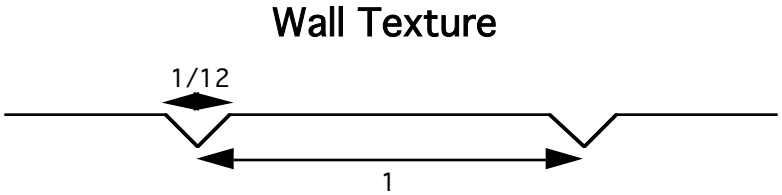
Using Noise Functions to Vary Shake Coordinates

```
{
        A1  -  Ratio of height to width for tiles
                  (where height is always equal to 1)
}

shake_u = mod(Px+.1*noise3(Px,Py,Pz), max(1,1/A1));
shake_v = mod(Py+.05*fnoise3(Px,Py,Pz), max(1,A1));
```

Shake_u will determine the coordinates for indexing the pattern in terms of its width (along the x-axis), and shake_v works on the pattern's height coordinates. We aren't using too much variation in either case; we've specified 10% variation along the width with the smooth noise function, and 5% variation along the height with the fractal noise. The max function takes two arguments and returns the larger of the two (either the max width or max height), and the mod function then takes two arguments (one of them being the coordinate plus some noise, and the other being the maximum) and returns the remainder of the first divided by the second. The result is the shake pattern being placed just a little up or down and to the left or right each time it is "tiled" in place.

Textures

We can add more realism to our cabin scene with a wall texture. Our wood paneled walls can be patterned with a procedural function, and textured with grooves between the boards. We'll start by visualizing what we want our wall panels to look like; the panels will be vertical, six inches wide, with V-shaped grooves .5 inches wide between the panels.



Ideally, we'd like to use this texture for all of our wall orientations, without ever having to specify "this is an xz texture" or "this is a yz texture". Even though it may seem more complex to set up such an adaptable texture, it's worth doing, because we didn't expand all our

genprism-generated wall surfaces (as we would need to do to determine which way the walls face).

We'll need some variable (either x or y) to break up our wall, using the mod function. We can use the surface normal orientation to determine if our wall is running along the x-axis or y-axis. The surface normal orientation is provided to us by Radiance as (Nx,Ny,Nz); the x-aligned walls have surface normals in the y-direction (Nx=0), while the y-aligned walls have surface normals in the x-direction (Ny=0). We'll edit a file called "paneltex.cal" to define our paneltex function.

Panel Texture Function

```
{
        This file creates a wall texture, using panels that run
        vertical (along the z-axis) with grooves that are 1/12
        as wide as the panel segments.  This texture is
        designed to work with all xz and yz wall orientations.
}
paneltex_dx = if(panel_isx, panel_pert(Px),0);
paneltex_dy = if(panel_isx, 0, panel_pert(Py));
paneltex_dz = 0;
panel_isx = .5-Nx*Nx;
panel_pert(v) = if     (1/24-frac(v),
                        -1,
                        if(frac(v)-23/24,
                                1,
                                0));
```

The purpose of our function is twofold; first, we need to determine what segment of the panel we're in (the left side, middle, or right side), and second, we need to apply the correct surface normal perturbation to each segment (computing the perturbation for the normal along v, regardless of whether we're on an x-aligned or y-aligned wall).

Once we've completed describing the texture function, we're ready to apply it to our walls with a procedural woodgrain pattern. This list of modifiers starts with our procedural function pattern, "zwoodpat", which modifies our functional texture "woodpaneltex", which in turn modifies our "wood_panel" material. The final result will be a much more realistic looking wall surface for our cabin, and all walls will be treated automatically without any need for us to expand the surface generators for all the walls.

Cabin Wall

```
# Cabin Wall Material

void brightfunc zwoodpat
4  zgrain  woodpat.cal  -s .05833
0
1  .15

zwoodpat texfunc woodpaneltex
6  paneltex_dx paneltex_dy \
     paneltex_dz paneltex.cal\
     -s .67
0
0

woodpaneltex plastic wood_panel
0
0
5  .5  .2  .1  0  .1
```

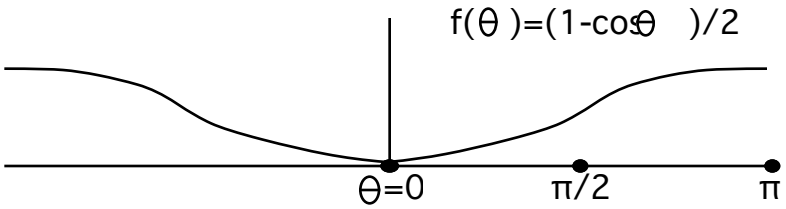
Instancing

The environment immediately surrounding our cabin model provides us with the opportunity to use instancing (for planting the trees), and gensurf for creating a bumpy ground.

For a bumpy ground surface, we can use the two-dimensional noise function to take the x and y values and produce a z value for our gensurf function. We'll want our cabin to be in a fairly flat spot, so we can modify the amplitude of the z-variation, so it flattens out the magnitude around the cabin. Since the ground material will be different in summer than in winter (needles in summer and snow in winter), we'll call our material something generic ("groundmat"), and then define it differently in our "summerday" and "winterday" files.

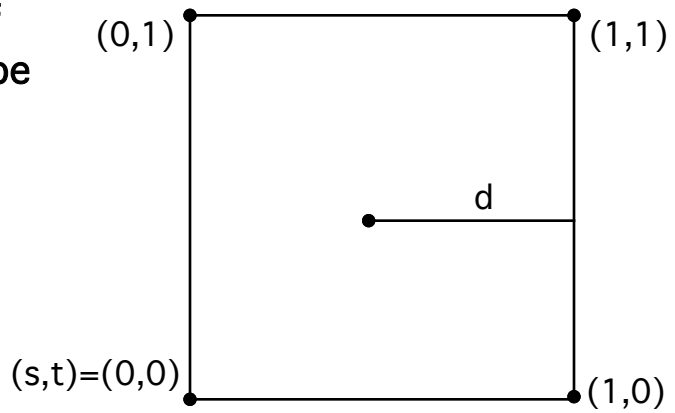
We'll center our landscape around the center of our cabin's foundation, which is located at (13,9,-1.5), and create a landscape that measures 200 feet square in the x and y dimensions, with a maximum height of eight (8) feet every fifty (50) feet. We'll can choose the type of function we want for gensurf to use to create these hills; a sinusoidal (smooth) function will look most natural (as opposed to a triangular function).

gensurf Magnitude Function



Since the gensurf function has an autocorrelation of 1 and a magnitude of 1, our 200' by 200' landscape will correspond to a gensurf surface that varies between (s,t)=(0,0) and (s,t)=(1,1).

gensurf Landscape



We'll want an equation to show as the distance (d) varies from 0 to .5 over our gensurf landscape, that our magnitude function for theta varies from 1 to π. The distance (d) is equal to the square root of the sum of the squares of (1/2-s) and (1/2-t), and this equation can be put in terms of theta using the conversion factor of (π/.5) on d, providing us with the final formula for the gensurf magnitude.

gensurf Landscape

```
!gensurf groundmat ground '-87+200*s'  '-91+200*t'\
        'mag(s,t)*noise2(s*4,t*4)*6-1.5'  15  15  -s\
        -e 'mag(s,t)=(1-cos(PI/.5*sqrt(sq(.5-s)+sq(.5-t))))/2'\
        -e 'sq(a)=a*a'  -f noise 2.cal
```

We can now define our forest floor pattern for our "summerdays" file, using the dirt function to create a pattern that will modify our scanned image of the forest floor, which in turn will modify our

"groundmat" material (this example was included as part of the Scene Description chapter).

Trees will be planted using rview, ximage, and rcalc. We need to start by running rview to place the trees in a scene that shows a box called "cabinbox" where the cabin is (for simplicity).

Using RVIEW to Place Trees

```
% oconv -f summerday landscape cabinbox > oct/dayland
% rview -vf vf/ext.se  -av 3  3  3  oct/dayland &
```

In RVIEW, we'll change to a parallel view by typing in the "view" command, and then typing "l" for parallel. We can start with a relatively high viewpoint (13 9 100) above the center of the cabin, with a view direction looking straight down (0 0 -1). Our view up will be (0 1 0), to match the same coordinate system we've been using for the cabin where the y-axis points north. The view horizontal and vertical size is (200 200), the dimensions of the whole scene.

Once we've selected all the parallel view parameters we need, we're ready to use the trace command "t" in RVIEW to pick a ray and get the (x,y,z) coordinates returned to us. We can either write down the resulting coordinates, or we can write out this file as "landsat", and then leave RVIEW with the "quit" command, and run ximage, with results piped in to RTRACE and our output file of tree locations, "tree.pts".

Using ximage to Plant Trees

```
% ximage pic/landsat | rtrace -op oct/dayland > tree.pts &
```

By using the "-op" option in RTRACE, we're requesting that output intersection points be given for the rays we pick at the places we want to plant our trees.

We can use RCALC to take the points we want to plant trees at, contained in our "tree.pts" file, and use a format file called "instance.fmt" to create the scene description instance primitives which can be appended to our "landscape" file.

"instance.fmt" File

```
void instance tree.${recno}
9  tree.oct  -rz  ${rand(recno)*360} -s ${.3+.2*rand(-recno)}\
        -t ${  $1  } ${  $2  } ${  $3  }
0
0
```

The variables for RCALC are preceded by the dollar sign symbol, "$", and are enclosed by curly brackets "{}". The record number in the input file will be used to number the tree instances, as well as assist in randomizing the rotation of the tree between 0° and 360° around the z-axis, and the scaling of the tree to somewhere between 30% and 50% of the tree size in "tree.oct". Now we're ready to perform the RCALC command to append the instances to our "landscape" file, based on the points we picked and saved in "tree.pts":

Using RCALC to Append "landscape" file

```
% rcalc  -o  instance.fmt  tree.pts  >>  ../landscape
```

This RCALC command indicates that output format is desired "-o", and that the output format file to be used is "instance.fmt". The input file is "tree.pts", and the ">>" characters indicate that we want the output to be appended to the following file, "../landscape", where the "../" characters indicate that the "landscape" file is in the directory just above.

If we thought we might want to change tree locations, we could use RCALC differently. Instead of appending instances for each tree to "landscape", we could have instead had one line command immediately following the gensurf command in the "landscape" file:

Alternate Use of RCALC for Tree Instancing

```
!rcalc  -o  treeinst.fmt  tree.pts
```

This use of the RCALC command would allow us the ability to change the "tree.pts" file, and not have to delete or change the rest of the "landscape" file.

Now we can change our Makefile to include the "landscape" file after "summerday". We'll also need to create a bounding cube around the cabin for "oct/cabin" that's big enough to show our surrounding landscape. We'll start by giving OCONV a bounding cube bigger than the default of the cabin size, and explicitly tell it to create a bounding cube that will include the entire landscape, where the x-minimum value is -87, the y-min is -91, and the zmin can be set to -100, since the bounding cube is a cube measuring 200' in each direction.

Changing Makefile to include "landscape"

```
oconv -b -87 -91 -100 200  -r 8192 \
        pattmats cabin bathroom furniture > oct/cabin
```

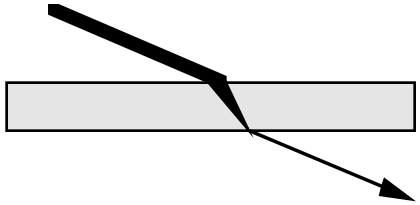
The "-r" option in the oconv command specifies the octree resolution. The resolution should be greater than or equal to the ratio of the largest and smallest dimensions in the scene (ie: surface size or distance between surfaces). The default resolution is 1024, or 2^{10}. The resolution works best with some 2^n value, because the cube gets subdivided by two's.

With a scene size set to 200' and the default resolution of 1024, our minimum cube size equals the size divided by the resolution, or 200/1024=0.1953', or 2.34". We increased the resolution by about eight times (2^3) to 2^{13}, or 8192. Now, our minimum cube size is 200/8192=.0244', or .293", which is pretty small. There should be no problem with any object in our cabin being smaller than .293 inches.

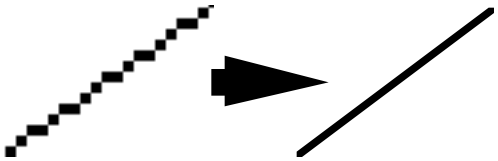
Glossary

Absorptance — A measurement of absorption.

Absorption — The diminishing of light's intensity at different wavelengths as it passes through a colored material, such as stained glass.



Anti-Aliasing — Programming techniques used to avoid "jaggies", unnatural pixel visibility which makes straight lines appear as staircases.



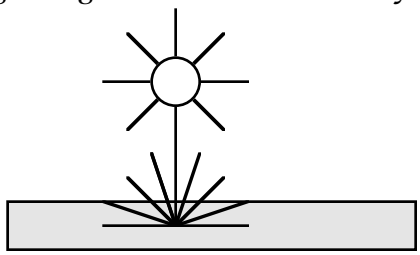
Bidirectional Reflectance Distribution Function — The ratio of the reflected radiance (intensity) in one direction to the incident irradiance (flux density) responsible for it from another direction is known as the bidirectional reflectivity, which is a function of the direction of reflection divided by the direction of incidence.

Boundary Representation — A description of an object in terms of its surface boundaries, rather than its volumes.

CAD Systems — Computer Aided Drafting systems used by designers, architects, and engineers.

Constructive Solid Geometry — An object definition system in which simple primitives are combined by means of regularized Boolean set operators that are included directly in the representation, where a whole object is the sum of its parts and their properties.

Diffuse Reflection — The equal scattering of light in all directions by a surface.



Geometric Model — The mathematical representation of objects in a picture. A polyhedral modeling system represents shapes as collections of polygonal faces.

Illuminance — A measure of the light arriving at a surface.

Image Plane — The two-dimensional viewing plane onto which a three-dimensional scene is projected.

Irradiance — The total light energy impinging on a surface.

Lambertian Surfaces — Surfaces that are purely diffuse, obeying Lambert's law of equal emission in all directions.

Luminance — The quantity of visible light passing through a point in a given direction.

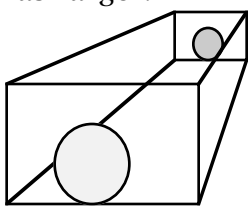
Mirror Reflection — The bouncing of a light ray by a surface in the mirror direction.

Monte Carlo — A stochastic, or random-sampling multi-dimensional integration technique.

Parametric Surface — Arbitrarily curved surfaces in three-dimensional space.

Parallel Projection — A mapping from the three-dimensional scene to the two-dimensional image plane in which parallel lines do not converge towards a vanishing point.

Perspective Projection — A mapping from the three-dimensional scene to the two-dimensional image plane which shows closer objects as larger.

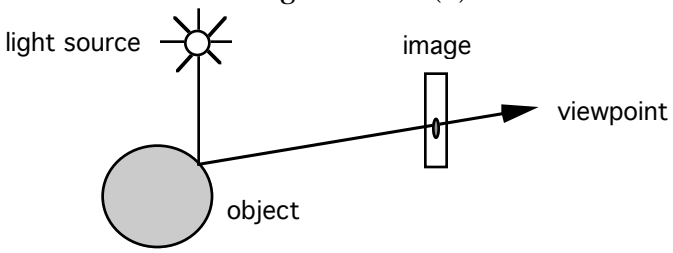


Pixel — The most fundamental picture element; the smallest alterable square in an image. A dot of color. Each pixel corresponds to a luminance value received through the lens by its position on the image plane.

Radiance — Radiometric brightness. The amount of light flux passing through a point in a particular direction.

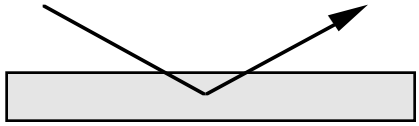
Ray Propagation Tree — The collection of an initial ray and all the rays that it spawned, and the rays that those rays spawned.

Ray Tracing — A technique used for rendering images of three-dimensional scenes from a computer by following individual rays of light from the viewpoint backwards to the light source(s).

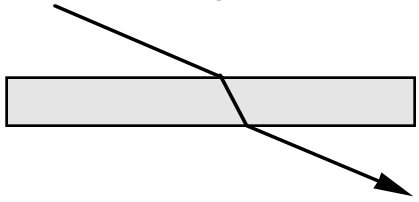


Recursion — Solving a problem by restating it as a simpler problem of the same type (eg: n! = n*(n-1)!, 0! = 1).

Reflection — The bouncing of a light ray by a surface in the mirror direction.



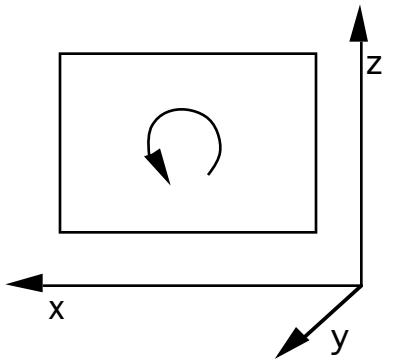
Refraction — The bending of a light ray as it enters or leaves a dielectric material, such as glass.



Resolution — The number of pixels in an image, usually given as the number of rows and columns. The color resolution is the number of possible colors for each pixel.

Right Hand Rule

A method for determining or specifying the surface normal direction (the thumb) from the curved direction of vertices across the other two dimensions of space (our fingers). In the following illustration, vertices entered in the circular direction shown (counter-clockwise) would result in a surface normal pointing out of the page in the y-direction.

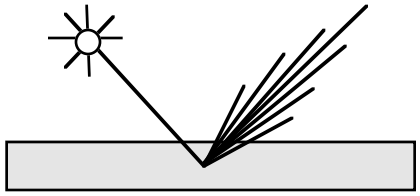


Run Length Encoding

A means of saving space used by ray tracing programs. If many adjacent pixels in a scanline have the same value, the value and a count is given instead of all the pixels.

Specular Reflection

The preferred scattering of light by a surface in the mirror direction. The shine from a surface.



Surface Normal

A three-dimensional vector pointing away from a surface at right angles.

Transmission

Light passing through a medium.

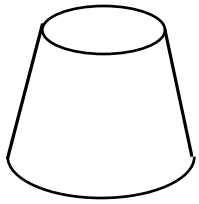
Terms

Argument
A string or real number associated with primitives in Radiance scene descriptions. The first integer following the identifier is the number of string arguments, and is followed by the arguments themselves.

Bounding Cube
A cube that contains all surfaces in the scene. OCONV must first determine or be given a bounding cube before it can generate an octree.

Command Expansion
The execution of commands contained in a scene description in lieu of the corresponding command output. If desired, the "-e" option of xform may be used to expand all commands, creating a "flat" scene description.

Cone
A cone is cut in two places perpendicular to its axis. The family of cones in Radiance includes *cylinders* and *rings* as special cases. A cone whose surface normal is directed inward is called a *cup*. A *cylinder* is like a cone whose radius is constant along its length. A *ring* is a flat object defined by a center, a normal direction, and an inner and outer radius.



Cup
An inverted cone (see cone).

Cylinder
A cylinder is like a cone whose radius is constant along its length. A *tube* is a cylinder whose surface normal is directed inwards rather than outwards.

Data File
A file containing data for Radiance programs; a scalar field in n-dimensions.

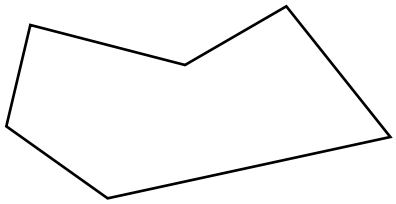
Direct Calculation	The calculation of illumination due to light coming directly from sources (rather than reflections "bounced" off other objects in the scene, or ambient light).
Frozen Octree	All data structures in an octree stay the same once that octree is frozen (with the "-f" option in oconv).
Function File	A file containing a functional procedure.
Generator	A program that creates compound objects such as prisms, patches and surfaces of revolution.
Hierarchy	A structure where an object is composed of smaller objects that may themselves be composed of even smaller objects. One method for creating scene hierarchy is to use the xform command to read in other scene files, transforming them to new positions. These scene files may contain other xform commands, thus producing a tree of transformations.
Indirect Calculation	A technique for computing interreflections from objects other than light sources.
Instancing	The use of one object many times (in many "instances") in a scene. Only one description of the object is required, along with information about the locations of all the identical objects, thus saving program memory. The layering of instances is called "hierarchical instancing".
Intersection Point	The point where a ray intersects the surface.
Light Source	An origin of illumination in a three-dimensional graphics scene.
Material	Materials determine how each surface in a Radiance scene interacts with light. Examples of materials are: light, metal, plastic, and glass.
Modifier	Either the identifier of a previously defined primitive, or "void", to get things started.
Object	A collection of one or more surfaces (polygons, spheres, cones, or light sources) contained in a description file.
Octree	The resulting data structure from a technique that sorts objects in a scene before the rays are traced, so the ray tracer can efficiently examine only those objects in the cubic segments of the scene where rays are intercepted.

Pattern — A perturbation of the material color (as opposed to a perturbation of the surface normal). A pattern affects its reflectance of an object.

Picture File — A type of file used by Radiance that contains information about a picture.

Pipe — A Unix facility for connecting the output of one program directly as input to another. In the Unix shell, the pipe facility is represented with the '|' character.

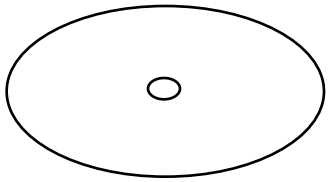
Polygon — An area in a plane enclosed by connected, non-crossing line segments defined by at least three distinct vertices.



Polygonal Mesh — A set of connected polygonally bounded planar surfaces. Polygonal meshes can be used to represent objects with curved surfaces, in an approximate fashion in which the curved surface is broken into a number of polygonal segments.

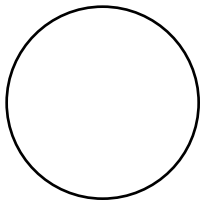
Primitive — The fundamental building block of a scene description that describes a material or pattern surface that's used by Radiance.

Ring — A ring is a flat object defined by a center, a normal direction, and an inner and outer radius. When the inner radius of a ring is zero, it is called a *disk*.



Shell Script — A file which contains commands to be executed by a command interpreter, such as the Bourne shell or the C-shell.

Sphere — A sphere is a three dimensional circular object, defined by its center point and radius. By default, the surface normal points away from the center. Spheres with inward pointing normals are called *bubbles* in Radiance.



Surface	The basic shapes used by Radiance: polygons, spheres, cones and light sources. An object is a collection of one or more surfaces contained in a description file.
Texture	A perturbation of the surface normal (as opposed to a perturbation of the material color). A texture affects the illumination and highlights of an object.
Translator	Converts files from one type to another (eg: converting a CAD file to a Radiance scene input file). Also called an importer or exporter.
Tube	A cylinder whose surface normal is directed inwards rather than outwards (see cylinder).
View	A particular set of parameters that define a two-dimensional projection from a three-dimensional scene.

Programs

OCONV	A Radiance command used to create an octree from a list of object files prior to producing picture files or other lighting calculation information in order to speed up the process of ray tracing.
RADIANCE	A software package for accurately calculating and displaying lighting. Radiance takes a scene description with light sources, sun, sky, buildings, rooms, furniture, etc. and produces spectral radiance values which can be collected in a "photo-accurate" color image.
RPICT	A Radiance rendering program that produces picture files in batch mode.
RTRACE	A Radiance program that provides a convenient interface for computing and combining individual radiance values to get luminance, contrast rendering factors, equivalent sphere illumination, glare indices, or any other lighting metric.
RVIEW	A Radiance rendering program that produces picture files interactively.
XFORM	A Radiance program that transforms a scene description from one coordinate space to another. XFORM performs rotation, translation, scaling, and mirroring.