1 Introduction

Preface

Documentation Conventions

The information in this guide is presented in the following way:

- Square brackets [] indicate options; parenthesis () indicate arguments.
- Each command and function is addressed separately. The discussion includes a description of the command or function's purpose and operation. This is followed by its syntax and command usage format and, finally, by an explanation of the arguments; for example:

  RAYatom(x,y,z,r)
  float x,y,z,r;

x,y,z	=	the coordinates of the centerpoint
r	=	the radius

- Where appropriate, examples and illustrations are included to further clarify the use of a command or function.

Pixel Machine Features

The Pixel Machines are graphics generation and display systems that provide high quality image computing. The systems are programmable and modular, and are designed to execute complex graphics functions at very high speeds. (For a detailed description of the Pixel Machine hardware and software features, refer to the *Pixel Machines User's Guide*.)

The Pixel Machine offers a complete set of system commands and a powerful graphics library, PIClib, for generating a multitude of images. PIClib's functions reside on the host computer and provide an interface between your application program and the Pixel Machine. Some of the highlights of PIClib include:

- high-level, 3D object generation (including patches, quadrics, and superquadrics)
- flat and Gouraud shading
- texture mapping onto 2D or 3D surfaces
- multiple light sources of different types
- antialiasing by supersampling for photorealistic 3D rendering
- 32-bit floating point z-buffer for highly accurate depth precision
- 32-bit double buffering
- a robust set of interactive 3D graphics functions
- a unique set of rgbz buffer copy routines

In addition to the graphics library, PIClib, a ray tracing library, RAYlib, is available for the Pixel Machine. RAYlib is composed of a powerful set of tools for generating high-quality graphics images and includes primitives for generating and manipulating 3D computer models and describing their physical attributes. It can be used as a high-end renderer to create realistic images for use in 3D visualization, industrial design, and television/motion picture production.

RAYlib includes many sophisticated features that enable you to achieve superb visual realism while maintaining the high rendering speed made possible by the parallel architecture of the Pixel Machine. These features include:

- **Realism** - RAYlib generates the shadows, reflections, and transparency that make a computer-generated image more realistic. The user can control the reflective and specular components of an object as well as the object's degree of transparency. All of these features (shadows, reflections, and transparency) can be disabled in the user's program for a quick preview of the scene before rendering the final image.
- **Light Sources** - Multiple light sources of any color can be used to produce a variety of lighting effects. The types of light sources available are point, direct (infinite), and area. Direct light sources produce lighting that is similar to sunlight. Area light sources have the effect of producing soft, natural shadows.
- **Texture Mapping** - Any 2D texture can be mapped onto objects created by RAYlib. These textures can be used to generate realistic surfaces, such as wood grains, clouds, marble, etc. The texture maps can have surface properties, such as reflectance and transparency, on a pixel by pixel basis. As many as 64 texture maps can be used at one time. Texture maps can be as large as 4K x 4K.
- **Antialiasing** - Adaptive stochastic antialiasing can be turned on to eliminate rough, jagged edges. The user can control the minimum and maximum number of samples and the contrast threshold.

- **Double Buffering** - Double buffering can be used to create smooth animations. When double buffering is enabled, objects are rendered into the off-screen buffer. This buffer is swapped with the on-screen buffer when instructed to do so by the user.

All application programs for the Pixel Machine are written in C. As a result, life-like images can be created quickly and easily without the need for machine-specific knowledge. For more information on the C programming language, refer to *The C Programming Language* by Brian W. Kernighan and Dennis M. Ritchie (1978, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, or the updated 1988 edition).

The sections that follow provide an overview of RAYlib's functions and a brief discussion of the differences between PIClib and RAYlib.

RAYlib Functions

RAYlib functions are grouped into the following categories:

Category	Functions in this category are used to . . .
Control	initialize the machine, start ray tracing, and terminate a ray trace session.
Graphics primitives	generate three dimensional polygons and atoms, render superquadrics (spheres, cylinders, ellipsoids, toroids, and hyperboloids of one and two sheets) and generate patches. Surface properties for these three dimensional objects are supported.
Bounding volumes	initialize, terminate, and record three dimensional extents of objects. Proper use can greatly enhance the ray tracing execution speed.
Transformations	perform a wide variety of operations such as controlling the translation, scale and rotation of objects. Viewing and projection are also included in this category.
Shading and Lighting	control the position, orientation, and intensity of light sources. Ambient light and light switch control are handled in this category. Surface properties for objects as well as shading modes are defined using these functions.
Viewports	create and manipulate viewports.
Antialiasing	eliminate jagged edges in the objects of a scene through the use of stochastic sampling.
Video	enable and disable the video map from the shadow map, load and retrieve color rgb maps as well as alpha overlay color maps.

Differences Between RAYlib and PIClib

The internal operation of RAYlib is fundamentally different from PIClib. While PIClib renders each geometric primitive as it is received by the Pixel Machine, RAYlib maintains a database of all geometry being rendered. Consequently, when using RAYlib you must define all objects and viewing parameters prior to rendering. If a change is necessary in a scene, the entire image must be redefined and a complete ray trace invoked again. Although slower than PIClib, RAYlib provides the realism often required for high quality output.

Despite the internal differences, PIClib users will find that their programs are easily ported to RAYlib because PIClib and RAYlib share a common syntax, common functionality (although not every RAYlib function has a corresponding PIClib function and vice versa) and a common set of structure definitions (typedefs).

There are, of course, some differences, and the remainder of this section is devoted to helping the PIClib user quickly become a RAYlib user. To this end, the first subsection, "Differences in Common Structure Definitions", briefly describes differences in the way elements of some common structure definitions are used by each library. The second subsection, "Differences in Common Functions", discusses the ways in which some RAYlib functions differ from their PIClib counterparts. The final subsection, "Functions Unique to RAYlib" provides an overview of the RAYlib functions that do not have a PIClib counterpart. For a detailed description of each RAYlib function, refer to Chapter 3 of this guide.

Differences in Common Structure Definitions

The following structure definitions are accessed differently by each library:

Structure	Usage
RAYsurface_model	The **RAYsurface_model** structure contains the same elements as the **PICsurface_model** structure, but these elements are used slightly differently. In RAYlib, the *a_** and *s_** color components are ignored, and the *specularity, reflectivity*, and *refraction_index* elements are used. The reverse is true in PIClib.
RAYlight_source	The RAYlib structure **RAYlight_source** has a structure element, *intensity*, that applies to all RAYlib light sources. Additionally, the fields *samples*, *vertices*, and *vertex* have been added to support area light sources.

Differences in Common Functions

The following RAYlib functions are applied differently than their PIClib counterparts:

Function	Usage
RAYput_surface_model()	In RAYlib a call to **RAYput_surface_model()** actually allocates memory for a new surface model. To reuse a surface model, **RAYset_surface_model()** should be called with the value that the function call to **RAYput_surface_model()** returned. Note that only **RAYput_surface_model()** returns a meaningful value. The corresponding PIClib function does not.
RAYatom()	The radius of the atom primitive defined by **RAYatom()** is scaled by the average scale factor determined by the current transform. In PIClib, no modeling transformations are applied to the radius of an atom, even though the projection transform is applied.
RAYshade_mode()	The **RAYshade_mode()** function is used in RAYlib to control shading effects such as shadows, reflections, and antialiasing.

Functions Unique to RAYlib

The following functions exist only in RAYlib:

Function	Description
RAYtrace()	Begins the ray tracing process. Nothing is rendered until **RAY-trace()** is called.
RAYstatistics()	Enables/disables the printing of ray tracing statistics.
RAYopen_bounding_volume()	Begins the computing of bounding volumes. Proper use of bounding volumes improves RAYlib's performance.
RAYclose_bounding_volume()	Ends the computing of bounding volumes.
RAYambient_intensity()	Sets the intensity of the ambient light.
RAYbackground_color()	Sets the color of a primary ray when it does not intersect any object in a 3D scene.

Function	Description
RAYclear_viewport()	Clears the current viewport to a specified color. This function is primarily used to clear the entire screen or to display drop shadows. Because RAYlib will set every pixel in the current viewport when it ray traces, there is no need to clear the viewport being ray traced.
RAYsamples()	Defines the minimum and maximum number of samples to take within a pixel when antialiasing is being done. It also defines the contrast threshold to be used to determine if the maximum amount of antialiasing is needed.
RAYput_texture()	Allocates regions of resident texture memory or host memory for virtual textures.
RAYset_texture()	Sets the current texture map to the specified texture id; the texture id should be the value returned by **RAYput_texture.**
RAYset_surface_model()	Sets the current surface model to the specified surface id; the id should be the value returned by the **RAYput_surface_model** call.

Getting Started

Before you can compile and run your programs, you need to make sure that the hardware is initialized and the software environment is set up correctly. When you first turn on the Pixel Machine, you must initialize the hardware to a known state. This is accomplished by executing the **hypinit** command. Once the hardware is initialized, you must boot the Pixel Machine by executing the **rayboot** command before you can run RAYlib graphics programs. For more information about **hypinit** and **rayboot**, refer to Chapter 2 of this guide.

The software environment must be set up at installation time and *after* any changes to the system's configurations (for example, upgrading the Pixel Machine or changing the Transformation Pipeline configuration). The procedures for setting up the software environment are described below.

Defining the Software Environment

Before using the Pixel Machine, the proper environment must be created. The /usr/hyper directory contains files for defining the Pixel Machine environment. For csh users, a **.login** (.hyper_login) and a **.cshrc** (.hyper_cshrc) are provided in /usr/hyper. **ksh** users will find a **.profile** (.hyper_profile) and a **.env** (.hyper_env) residing there as well.

.hyper_login and **.hyper_profile** define Pixel Machine-specific environment variables and update some standard UNIX system environment variables in order to provide easy access to Pixel Machine software and manual pages. **.hyper_cshrc** and **.hyper_env** establish *aliases* (or shortcuts) for redefining environment variables and performing system initialization functions.

If you are using **csh**, you should **source** .hyper_login and .hyper_cshrc into your **.login** and **.cshrc** files, respectively. To do so, edit your **.login** file, and add the following to the end of the file:

```
source /usr/hyper/.hyper_login
```

Then edit your **.cshrc** file and add the following to the end of the file:

```
source /usr/hyper/.hyper_cshrc
```

If you are using **ksh**, you should **.** (dot) .hyper_profile and .hyper_env into your **.profile** and **.env** files, respectively. To do so, edit your **.profile** and add the following to the end of the file:

```
. /usr/hyper/.hyper_profile
```

Then edit your **.env** file and add the following to the end of the file:

```
. /usr/hyper/.hyper_env
```

It is important to note that the files provided in /usr/hyper are generic, and the environment variables defined in these files may not initially correspond to your specific machine configuration. Before using the files provided, your system administrator should make any necessary modifications to ensure that the Pixel Machine environment variables are defined appropriately for your machine. A description of each environment variable and a list of its possible values is given below.

The **HYPER_MODEL** variable specifies the Pixel Machine model and Transformation Pipeline configuration. The table below describes the values that should be assigned to this variable, depending on what mode. and configuration you have.

A value of . . .	Denotes a . . .
916	Pixel Machine 916, single Pipe, 1024x1024
916d	Pixel Machine 916, dual Pipe, 1024x1024
920	Pixel Machine 920, single Pipe, 1280x1024
920d	Pixel Machine 920, dual Pipe, 1280x1024
932	Pixel Machine 932, single Pipe, 1024x1024
932d	Pixel Machine 932, dual Pipe, 1024x1024
940	Pixel Machine 940, single Pipe, 1280x1024
940d	Pixel Machine 940, dual Pipe, 1280x1024
964	Pixel Machine 964, single Pipe, 1024x1024
964d	Pixel Machine 964, dual Pipe, 1024x1024
964X	Pixel Machine 964, single Pipe, 1280x1024
964dX	Pixel Machine 964, dual Pipe, 1280x1024

NOTE A lower case "n" appended to the model number denotes an NTSC model whose resolution is 720x486. A lower case "p" appended to the model number denotes a PAL model whose resolution is 720x576.

The **HYPER_PATH** variable specifies the full pathname to the host directory that contains the Pixel Machine software (for example, */usr/hyper*)

The **HYPER_PIPE** variable specifies the Pipeline configuration (serial or parallel) for systems with two Transformation Pipelines.

The **HYPER_UNIT** variable specifies the Pixel Machine unit number. Up to four machines (numbered 0, 1, 2, 3) can be connected to a host computer.

Writing RAYlib Programs

At the beginning of each application C program you write, you need to include the RAYlib header file. This file includes type definitions, constants, and external definitions, and is included by the following statement:

```
#include   "raylib.h"
```

The first RAYlib function called within an application program should be **RAYInit**. This function initializes the viewport to a full screen (either 1024x1024, 1280x1024, 720x480 or 720x576 depending on your Pixel Machine configuration) and sets default precisions. **RAYInit** returns a value of RAY_ERR_OK if the initialization is successful, or a value of RAY_ERR_ARG if it failed. For a complete description of **RAYInit**, see that manual page in the *RAYlib Reference Manual*.

The last RAYlib function called within an application program is usually **RAYexit**. It performs various clean up functions, and unlocks the Pixel Machine making it accessible to other users. Be sure to include it at the end of your program.

Compiling RAYlib Programs

To compile your RAYlib program, link **raylib.a** and the **math** library as follows:

cc -I$HYPER_PATH/include *file*.c $HYPER_PATH/lib/raylib.a -lm -o *file.exe*

where, *file.c* is the name of the file containing the program. You can also link with raylib_ffpa.a to run on a Sun with a floating point accelerator board.

The system will compile your program and create an executable file called *file.exe* (if the **-o** flag is omitted, the executable file will be called **a.out**). To run the program, type the name of this executable file.

2 Commands and Utilities

Pixel Machine System Commands and Utilities

The system commands and utilities allow you to perform utility and administrative functions, such as initializing the hardware, loading the RAYlib processor programs into the Transformation Pipeline(s) and Pixel Nodes, or simply locking your Pixel Machine.

The system **commands** described in this section are:

Command	Function
hypenv	Displays current settings of environment variables.
hypfree	Releases a locked unit.
hypid	Displays node ID data.
hypinit	Initializes the hardware.
hyplock	Locks a unit.
hypstat	Displays system status.

The system **utilities** described in this section are:

Utility	Function
rayboot	Loads RAYlib software into the Pixel Machine.

hypenv

The **hypenv** command displays the current values of the Pixel Machine environment variables. The environment variables must be set on the host workstation either in a login procedure or on the command line before using the Pixel Machine. (See Chapter 1 of this guide for procedures for setting Pixel Machine environment variables.) If no options are specified, the status of all environment variables are displayed.

Command usage is:

hypenv [-D] [-M] [-P] [-U] [-u]

The options are as follows:

- **-D** Print current value of **HYPER_PIPE** (serial or parallel)
- **-M** Print current value of **HYPER_MODEL** environment variable
- **-P** Print current value of **HYPER_PATH** environment variable
- **-U** Print current value of **HYPER_UNIT** environment variable
- **-u** Print command usage format

If you enter **hypenv**, the system displays the following typical response:

```
Model: 964d  Pipe: parallel  Unit: 0  Path: /usr/hyper
```

hypfree

The **hypfree** command releases one or more Pixel Machines that were locked with the **hyplock** command. If no options are specified, the command releases only the current unit.

Command usage:

hypfree [-a] [-u]

The options are as follows:

- **-a** Free all units
- **-u** Print command usage format

hypid

The **hypid** command generates a list of ID data on the Nodes in the Pixel Machine.

Command usage:

hypid [-a] [-d *node***] [-g** *node***] [-w] [-u]**

The options are as follows:

-a	Print ID data on all Nodes
-d*node*	Print ID data of Pixel Node number *node* or *all*
-g*node*	Print ID data of Transformation Node number *node* or *all*
-w	Write the ID data into the selected Node.
-u	Print command usage format

If you enter **hypid -d1**, the system displays the following typical response for a Pixel Machine 964 model booted with PIClib:

```
Drawing node 1 identification data:
    node id:  1
    x nodes:  8
    y nodes:  8
    x offset:  0
    y offset:  1
    program:  'pic964.dsp'
    semaphore: 0
```

The ID data provides the following information:

- **node id** contains the sequential numbering of the Transformation and Pixel Nodes. The Pixel Nodes range from 0 to *n* (n = 63 on a model 964). The Transformation Nodes range from 0 to 8 for a single Pipe configuration; from 0 to 17 for a dual Pipe configuration.
- **x nodes** and **y nodes** indicate the configuration of the buffer in an N x M array.
- **x offset** and **y offset** indicate the position of the processor in the 2D array.
- **program** lists the name of the DSP executable program that is loaded into memory.
- **semaphore** contains system information.

hypinit

Each time you power up the system, you need to initialize it to a known state. The **hypinit** command initializes the Pixel Machine to its default state. If no options are specified, **hypinit** initializes the Transformation Nodes and FIFOs, the Pixel Nodes, the drawing mode register, the Transformation Pipeline, and the video.

You can also use this command to reinitialize the Pixel Machine whenever you want the system to return to its initial state.

Command usage is:

hypinit [-b] [-d] [-g] [-m] [-p] [-q] [-Q] [-r] [-v] [-V] [-u]

The following options may be used to limit initialization:

- **-b** Initialize the VME bus repeater
- **-d** Initialize the Pixel Nodes
- **-g** Initialize the Transformation Nodes
- **-m** Initialize the drawing mode register to the current configuration model, disable overlay video, and turn off testing mode.
- **-p** Reconfigure Pipelines in series or parallel based on the environment variable
- **-q** enables pipelined writes
- **-Q** disables pipelined writes
- **-r** Reset input and output Pipeline FIFOs
- **-v** Initialize video registers and lookup table
- **-V** Do not initialize video
- **-u** Print command usage format

If you enter **hypinit**, the system displays the following typical response.

```
System configuration:
     geometry cards: 2 nodes: 18
     geometry pipes: multiple in parallel
     drawing cards: 16 nodes: 64
     drawing node dram: 256 [kbytes] vram: 256 [kbytes]
     drawing pixel interleaving x: 8 y: 8
     drawing node/screen scale x: 0.125 y: 0.125
     video format: high resolution
     video screen size x: 1024 y: 1024

VMEbus-repeater csr register: active [ no_pipeline lights: 0 no_reset cool_temperature ].

Geometry nodes[0-17]: active [ halted pir16 eni dma auto pdf ].

Drawing nodes[0-63]: active [ halted pir16 eni dma auto ] errors [ sync ].

Geometry output (write   ) fifo[0] flags: active [ empty ].
Geometry input  (feedback) fifo[0] flags: active [ empty ].
Geometry output (write   ) fifo[1] flags: active [ empty ].
Geometry input  (feedback) fifo[1] flags: active [ empty ].

Draw mode registers[0-15]: active.

Video csr register: active [ type: 964 shadow no_refresh no_shift yo:
  964X no_psync0 no_psync1 hsize: 1280 ].
```

hyplock

The **hyplock** command locks the current Pixel Machine and prevents other users who are timesharing the system from accessing it. (The Pixel Machine is not multitasking.) Before you log off, remember to unlock the system by executing the **hypfree** command.

Command usage:

hyplock [**-u**]

The options are as follows:

-u Print command usage format

hypstat

The **hypstat** command displays the system status of the Pixel Machine.

Command usage is:

hypstat [-u]

The options are as follows:

-u Print command usage format

If you enter **hypstat**, the system displays the following typical response. If you get an error message, enter the **hypinit** command first and then **hypstat**:

```
System configuration:
    geometry cards: 2 nodes: 18
    geometry pipes: multiple in parallel
    drawing cards: 16 nodes: 64
    drawing node dram: 256 [kbytes] vram: 256 [kbytes]
    drawing pixel interleaving x: 8 y: 8
    drawing node/screen scale x: 0.125 y: 0.125
    video format: high resolution
    video screen size x: 1024 y: 1024

VMEbus-repeater csr register: active [ no_pipeline lights: 0 no_reset cool_temperature ].

Geometry nodes[0-7]: active [ halted pir16 eni dma auto pdf ].
Geometry node[8]: active [ halted pir16 eni dma auto ].
Geometry nodes[9-16]: active [ halted pir16 eni dma auto pdf ].
Geometry node[17]: active [ halted pir16 eni dma auto ].

Geometry output (write    ) fifo[0] flags: active [ empty ].
Geometry input  (feedback) fifo[0] flags: active [ empty ].
Geometry output (write    ) fifo[1] flags: active [ empty ].
Geometry input  (feedback) fifo[1] flags: active [ empty ].

Drawing nodes[0-63]: active [ halted pir16 eni dma auto ] errors [ sync ].

Draw mode registers[0-15]: active.

Video csr register: active [ type: 964 shadow no_refresh no_shift yo:
  964X no_psync0 no_psync1 hsize: 1280 ].
```

rayboot

rayboot initializes the machine (with the exception of the video) and loads the Transformation and Pixel Nodes with the appropriate software for the machine based on the value of the **HYPER_MODEL** and **HYPER_PIPE** variables. The software that is downloaded to the Transformation and Pixel Nodes resides in the directory specified by **HYPER_PATH**. To initialize the Pixel Machine video, use the **hypinit** command discussed earlier in this chapter.

rayboot should be executed:

- when the machine is powered up
- after changing any of the environment variables; **HYPER_MODEL**, **HYPER_PATH**, **HYPER_PIPE** and **HYPER_UNIT**
- after running PIClib demos
- after using DEVtools
- in the event of RAYlib software failure

It is important to note that after changing the HYPER_MODEL environment variable, you must first initialize the Pixel Machine by executing **hypinit -v** in order to initialize the video registers and lookup tables. The **rayboot** command can then be executed to download the appropriate software to the Transformation and Pixel Nodes. Because the video registers and lookup tables are re-initialized by the first call to **hypinit**, you need to re-execute **picrt** and **picgamma** if you are using these utilities.

NOTE

For more information about **picrt** and **picgamma**, refer to Chapter 2 of the *PIClib User's Guide*.

3 RAYlib Functions

Control Functions

The RAYlib system control functions perform basic setup and "housekeeping" operations to allow applications to communicate with the Pixel Machine. Prior to using these programs, the Pixel Machine ray tracing software must be loaded into the Pixel Machine hardware (see Chapter 2 of this guide). These control functions are:

- **RAYinit()**
- **RAYexit()**
- **RAYtrace()**
- **RAYstatistics()**
- **RAYexit_immediate()**
- **RAYhalt()**

RAYinit()

int RAYinit()

RAYinit is always the first function called in every RAYlib program, and should be invoked only once. It initializes the viewport to full screen (1024x1024 or 1280x1024 for high resolution models, 720x480 for NTSC models and 720x576 for PAL), initializes the transformation matrix to the identity matrix, and sets various system parameters to their default values. **RAYinit** also sets up a signal handler to catch the following signals:

- hangup
- interrupt
- software termination

When the signal handler is invoked, it calls **RAYexit_immediate** and terminates ray tracing on the Pixel Machine. **RAYinit** will not override previously established signal handlers. If the user has established a signal handler for any of the above signals prior to calling **RAYinit**, that signal handler will remain in effect.

RAYinit returns an integer value of RAY_ERR_OK if the initialization succeeds and RAY_ERR_OPEN if it fails.

RAYexit()

---

int RAYexit()

RAYexit is usually the last RAYlib function called in a program. It halts all transformation and drawing node processors in the Pixel Machine hardware and closes the device. Signal handlers established during **RAYinit** are reset to their default actions.

NOTE: **RAYexit** should not be used in a signal handler; instead use **RAYexit_immediate**.

RAYtrace()

---

int RAYtrace()

RAYtrace initiates ray tracing in the Pixel Machine. Calls to RAYlib prior to **RAYtrace** define the objects, viewing, light sources, and other aspects of a scene. When **RAYtrace** is invoked, the data that has been collected is rendered into the frame buffer. **RAYtrace** does not return until the entire ray tracing process is completed. The RAYlib database is then re-initialized to accommodate animation sequences.

NOTE: Unlike PIClib which renders objects as they are defined, RAYlib renders nothing until **RAYtrace** is called. Refer to the section, "Differences between RAYlib and PIClib" in Chapter 1 for more information.

RAYtrace returns an integer value indicating the completion status of the ray tracer. The possible return codes are:

A return code of . . .	**indicates that . . .**
RAY_ERR_OK	the ray tracer completed successfully
RAY_WARN_NO_OBJ	no objects were passed to the ray tracer
RAY_ERR_BAD_BVOL	the number of bounding volume opens does not match the number of bounding volume closes
RAY_HALTED	the ray tracer was suspended at the user's request
RAY_ERR_INTERNAL	an internal error (hardware or software) has occurred

Example:

```
#include "raylib.h"
main()
{
        if (RAYinit() != RAY_ERR_OK) exit(1);
        ...
        ...
        ...
        RAYtrace();
        RAYexit();
}
```

RAYstatistics()

void RAYstatistics(mode)
int mode;

mode = statistics to be printed at the end of the ray tracing run.

RAYstatistics determines what ray tracing statistics, if any, will be printed at the end of a ray tracing run. This function is called with an argument, *mode*, which can be set to any one or combination of values described in the table below. Modes are combined by adding them together. The default mode is RAY_TIMINGS + RAY_STATISTICS.

A mode of . . .	prints . . .
RAY_OFF	no statistics
RAY_STATISTICS	total pages and object counts
RAY_TIMINGS	timing statistics
RAY_PAGE_STATISTICS	page and page fault statistics
RAY_ALL_STATISTICS	print all of the above statistics

RAYexit_immediate()

---

void **RAYexit_immediate()**

RAYexit_immediate replaces **RAYexit** for signal handlers. If an application signal handler is going to exit immediately, it should call **RAYexit_immediate** to halt the pixel nodes and clean up the system. If the pixel nodes have begun ray tracing, they will continue to do so even after the program exits, unless a **RAYexit_immediate** is called.

NOTE: **RAYexit** should still be used for normal program exits.

Example:

```
/* example of a signal handler that exits immediately */

#include    <signal.h>
#include    "raylib.h"

void
mysignal(sig,code,scp)
        int         sig, code;
        struct sigcontext    *scp;
{
        /* in the event of an interrupt, this signal handler will
          terminate ray tracing and exit immediately */

        RAYexit_immediate();
        exit();
}

main()
{
        int         return_value;

        signal(SIGINT, mysignal);

        if ( RAYinit() != RAY_ERR_OK) exit(1);

        ...
        ...
        ...

        return_value = RAYtrace();

        RAYexit();
}
```

RAYhalt()

---

void RAYhalt()

RAYhalt is used to post a request to halt the ray tracer. Typically, it is called from a signal handler in response to a user request. The halt request is only recognized by the routine **RAYtrace**. If a halt request is encountered at any time during the execution of a call to **RAYtrace**, ray tracing is terminated and **RAYtrace** returns RAY_HALTED. Execution may then continue in the application program as though **RAYtrace** had completed normally.

If **RAYhalt** is called from a signal handler, it is important that the signal handler return to continue execution, or else the halt request will never be seen and the Pixel Machine may continue ray tracing. If a signal handler must exit, **RAYexit_immediate** should be called to halt the Pixel Machine.

Example:

```
/* example of a signal handler that requests a halt */

#include   <signal.h>
#include   "raylib.h"

void
mysignal(sig,code,scp)
        int             sig, code;
        struct sigcontext     *scp;
{
        /* in the event of an interrupt, this signal handler will
          post a request to terminate ray tracing and then return */

        RAYhalt();
}

main()
{
        int             return_value;

        signal(SIGINT, mysignal);

        if ( RAYinit() != RAY_ERR_OK) exit(1);

        ...
        ...
        ...

        return_value = RAYtrace();

        if (return_value == RAY_HALTED)
                printf("Ray tracing halted because of user interrupt\n");

        RAYexit();
}
```

Graphics Primitives - Polygons and Atoms

Graphic primitives allow an application to draw three dimensional polygons and surfaces. The specific topology is defined in this section. Surface characteristics are covered in the "Shading and Lighting" section. All surfaces are rendered using the current surface model, which is established by a call to **RAYput_surface_model** or **RAYset_surface_model.**

Polygons may be texture mapped and/or have normal vectors defined at the vertices. Associated functions are:

- **RAYpoly_close()**
- **RAYpoly_point_3d(x,y,z)**
- **RAYpoly_point_nv(x,y,z,nx,ny,nz)**
- **RAYpoly_point_uv(x,y,z,u,v)**
- **RAYpoly_point_nv_uv(x,y,z,nx,ny,nz,u,v)**
- **RAYatom(x,y,z,r)**

NOTE

When using the **RAYpoly_point** functions, note that all polygons must be convex and should be planar

RAYpoly_close()

---

void **RAYpoly_close()**

This function closes a polygon by connecting the last polygon point to the first vertex. The polygon is rendered using the current surface model. This function must be used after a series of polygon defining calls such as:

- **RAYpoly_point_3d**
- **RAYpoly_point_nv**
- **RAYpoly_point_uv**
- **RAYpoly_point_nv_uv**

RAYpoly_point_3d()

```
void RAYpoly_point_3d(x,y,z)
float x,y,z;
```

x,y,z = the x,y and z coordinates of a vertex

RAYpoly_point_3d is used in sequence to define a series of 3D vertices that compose a polygon. The sequence of coordinates defined by each call to a **RAYpoly_point_3d** function is not connected until a **RAYpoly_close** function is specified. The polygon is drawn using the current surface model.

RAY_poly_point_nv()

```
void RAYpoly_point_nv(x,y,z,nx,ny,nz)
float x,y,z,nx,ny,nz;
```

x,y,z = the x,y and z coordinates of a vertex

nx,ny,nz = normal vector at the vertex

RAYpoly_point_nv is used in sequence to define a series of 3D vertices that compose a polygon with normals at each vertex. The sequence of coordinates defined by each call to a **RAYpoly_point_nv** function is not connected until **RAYpoly_close** is specified. The polygon is drawn using the current surface model.

A surface normal is specified at each vertex. The normal vector points outward in a closed solid object.

RAY_poly_point_uv()

```
void RAYpoly_point_uv(x,y,z,u,v)
float x,y,z,u,v;
```

x,y,z = the x,y and z coordinates of a vertex

u,v = texture indices at the vertex

RAYpoly_point_uv is used in sequence to define a series of 3D vertices that compose a polygon with texture indices at each vertex. The sequence of coordinates defined by each call to a **RAYpoly_point_uv** function is not connected until **RAYpoly_close** is specified. The polygon is drawn using the current surface model.

The texture indices are specified at each vertex. These indices must be non-negative floating point values. The edges of the texture are defined to be from 0.0 to 1.0, regardless of the size or aspect ratio of the texture. Indices greater than 1.0 cause the texture to wrap.

NOTE: Texture indicies are currently limited to the range 0.0 to 16.0.

This function is unique to RAYlib and has no equivalent in PIClib.

RAY_poly_point_nv_uv()

```
void RAYpoly_point_nv_uv(x,y,z,nx,ny,nz,u,v)
float x,y,z,nx,ny,nz,u,v;
```

x,y,z	=	the x,y and z coordinates of a vertex
nx,ny,nz	=	normal vector at the vertex
u,v	=	texture indices at the vertex

RAYpoly_point_nv_uv is used in sequence to define a series of 3D vertices that compose a polygon with normals and texture indices at each vertex. The sequence of coordinates defined by each call to **RAYpoly_point_nv_uv** is not connected until **RAYpoly_close** is specified. The polygon is drawn using the current surface model.

The texture indices are specified at each vertex. These indices must be non-negative floating point values. The edges of the texture are defined to be from 0.0 to 1.0, regardless of the size or aspect ratio of the texture. Indices greater than 1.0 cause the texture to wrap.

NOTE: Texture indicies are currently limited to the range 0.0 to 16.0.

RAYatom()

```
void RAYatom(x,y,z,r)
float x,y,z,r;
```

x,y,z = center of the atom

r = radius

RAYatom draws a spherical atom centered at a given location with a specified radius. The atom's center and radius are transformed by the current transformation matrix. The radius of the atom is scaled by the average scale factor determined by the current transform. Thus, if the modeling transform has explicit distortion, the atom will still be round when rendered. The atom is rendered using the current surface model.

Whenever possible, **RAYatom** should be used in place of **RAYsphere**. **RAYatom** is faster and more accurate than **RAYsphere** because **RAYatom** is rendered as a single primitive, while **RAYsphere** (and all the other superquadrics and patches) are tessellated into polygons. **RAYsphere** should only be used when independent scaling is required along each axis of the sphere.

NOTE: The radius (*r*) must be positive.

Graphics Primitives - Quadrics and Superquadrics

The quadrics and superquadrics functions draw atoms, spheres, ellipsoids, toroids, and hyperboloids of one and two sheets. All the primitives in this section are tesselated into polygons. The degree of tesselation is controlled by the quadric precision.

NOTE The maximum precision for superquadrics is limited to 160 divisions in each direction.

Functions in this section are:

- **RAYquadric_precision(nu,nv)**
- **RAYsphere()**
- **RAYsuperq_ellipsoid(x,y,z,exp1,exp2)**
- **RAYsuperq_torus(x,y,z,r,exp1,exp2)**
- **RAYsuperq_hyper1(x,y,z,exp1,exp2)**
- **RAYsuperq_hyper2(x,y,z,exp1,exp2)**

NOTE When using the **quadric** and **superquadric** functions, one bounding volume is implicitly defined around the entire primitive.

RAYquadric_precision()

```
int RAYquadric_precision(nu,nv)
int nu,nv;
```

nu = the number of line segments (or points) used to approximate the quadric in the u direction

nv = the number of line segments (or points) used to approximate the quadric in the v direction

The **RAYquadric_precision** function sets the precision used to render quadrics and superquadrics. The precision is defined by the number of line segments (or points) used to approximate the quadric in both the u and v directions. If the values for either direction are less than zero, the function returns RAY_ERR_ARG, otherwise it returns RAY_ERR_OK; the default precision is RAY_QUADRIC_DEFAULT in both the u and v directions.

RAYsphere()

---

void RAYsphere()

Using the current surface model, the **RAYsphere** function renders a sphere that is centered at the origin and has a unit radius. Its precision is set by the **RAYquadric_precision** function.

Whenever possible, **RAYatom** should be used in place of **RAYsphere**. **RAYatom** is faster and more accurate than **RAYsphere** because **RAYatom** is rendered as a single primitive, while **RAYsphere** (and all the other superquadrics and patches) are tessellated into polygons. **RAYsphere** should only be used when independent scaling is required along each axis of the sphere.

RAYsuperq_ellipsoid()

---

void RAYsuperq_ellipsoid(x,y,z,exp1,exp2)
float x,y,z,exp1,exp2;

x,y,z	=	the radii of the ellipsoid in the x, y, and z directions
exp1	=	the squareness parameter in the longitudinal direction
exp2	=	the squareness parameter in the latitudinal direction

The **RAYsuperq_ellipsoid** function renders a superquadric ellipsoid using the current attributes. A superquadric ellipsoid is a single, closed volume that ranges from a cuboid to a spheroid to a pinched object, depending on the specified exponents, and is represented mathematically as:

$$\underline{e}(\eta, \omega) = \begin{bmatrix} x \cos^{exp1}(\eta)\cos^{exp2}(\omega) \\ y \cos^{exp1}(\eta)\sin^{exp2}(\omega) \\ z \sin^{exp1}(\eta) \end{bmatrix}$$

where, η and ω are the longitudinal and latitudinal angles, respectively.

Values for η are in the range: $-\pi/2 <= \eta <= \pi/2$.

Values for ω are in the range: $-\pi <= \omega < \pi$.

The shape of the ellipsoid can be modified by varying the exponents as follows:

exp < 1	Square shaped ellipsoids
exp = 1	Round ellipsoids
exp = 2	Flat beveled ellipsoids
exp > 2	Pinched ellipsoids

NOTE All arguments for this function must be greater than or equal to zero.

Example:

The following program fragments render a sphere, ellipsoid, cube, and cylinder, respectively:

```
#include    "raylib.h"
main()
{

   /*render a sphere*/
   RAYsuperq_ellipsoid(100.0,100.0,100.0,1.0,1.0);
   /*render an ellipsoid that's stretched in the y direction*/
   RAYsuperq_ellipsoid(100.0,200.0,100.0,1.0,1.0);
   /*render a cube*/
   RAYsuperq_ellipsoid(100.0,100.0,100.0,0.01,0.01);
   /*render a cylinder*/
   RAYsuperq_ellipsoid(100.0,100.0,100.0,0.0,1.0);


   RAYexit();
   exit(0);
}
```

RAYsuperq_torus()

void RAYsuperq_torus(x,y,z,r,exp1,exp2)
float x,y,z,r,exp1,exp2;

x,y,z = the radii of the toroid ring

r = the distance from the center of the torus to the center of the outer ring (see Figure 3-1)

exp1 = the squareness parameter in the longitudinal direction

exp2 = the squareness parameter in the latitudinal direction

The **RAYsuperq_torus** function renders a superquadric toroid using the current attributes. The toroid is represented mathematically as:

$$\underline{p}(\eta, \omega) = \begin{bmatrix} x(a + \cos^{exp1}(\eta)) \cos^{exp2}(\omega) \\ y(a + \cos^{exp1}(\eta)) \sin^{exp2}(\omega) \\ z \sin^{exp1}(\eta) \end{bmatrix}$$

where,

$$a = \frac{r}{\sqrt{x^2 + y^2}}$$

and where η and ω are the longitudinal and latitudinal angles, respectively.

Values for η are in the range: $-\pi <= \eta < \pi$.

Values for ω are in the range: $-\pi <= \omega < \pi$.

If x and y parameters are not the same, the toroid radius is "stretched" in the direction of the larger parameter. The shape of the toroid can be modified in each direction by varying the exponents as follows:

exp < 1	Square shaped toroids
exp = 1	Round toroids
exp = 2	Flat beveled toroids
exp > 2	Pinched toroids

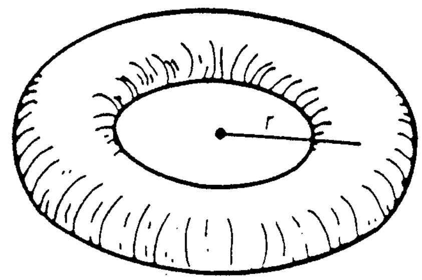

Figure 3-1. A Superquadric Toroid

RAYsuperq_hyper1()

void RAYsuperq_hyper1(x,y,z,exp1,exp2)
float x,y,z,exp1,exp2;

x,y	=	the radii of the xy cross-section of the hyperboloid at z = 0
z	=	the height of the hyperboloid when $\eta = 45°$
exp1	=	the squareness parameter in the longitudinal direction
exp2	=	the squareness parameter in the latitudinal direction

The **RAYsuperq_hyper1** function renders a superquadric hyperboloid of one sheet using the current attributes. The hyperboloid is represented mathematically as:

$$\underline{p}(\eta, \omega) = \begin{bmatrix} x \sec^{exp1}(\eta)\cos^{exp2}(\omega) \\ y \sec^{exp1}(\eta)\sin^{exp2}(\omega) \\ z \tan^{exp1}(\eta) \end{bmatrix}$$

where, η and ω are the longitudinal and latitudinal angles, respectively.

Values for η are in the range: $-\pi/2 < \eta < \pi/2$.

Values for ω are in the range: $-\pi <= \omega < \pi$.

The shape of the hyperboloid can be modified by varying the exponents as follows:

exp < 1	Square shaped hyperboloids
exp = 1	Round hyperboloids
exp = 2	Flat beveled hyperboloids
exp > 2	Pinched hyperboloids

RAYsuperq_hyper2()

```
void RAYsuperq_hyper2(x,y,z,exp1,exp2)
float x,y,z,r,exp1,exp2;
```

x,y	=	the radii of the xy cross-section of the hyperboloid at z = 0
z	=	the height of the hyperboloid when η = 45°
exp1	=	the squareness parameters in the longitudinal direction
exp2	=	the squareness parameters in the latitudinal direction

The **RAYsuperq_hyper2** function renders a superquadric hyperboloid of two sheets using the current attributes. The hyperboloid is represented mathematically as:

$$\underline{r}(\eta, \omega) = \begin{bmatrix} x \sec^{exp1}(\eta)\sec^{exp2}(\omega) \\ y \sec^{exp1}(\eta)\tan^{exp2}(\omega) \\ z \tan^{exp1}(\eta) \end{bmatrix}$$

where, η and ω are the longitudinal and latitudinal angles, respectively.

Values for η are in the range: $-\pi/2 < \eta < \pi/2$.

Values for ω are in the range: $-\pi/2 < \omega < \pi/2$ (piece 1), $\pi/2 < \omega < 3*\pi/2$ (piece 2)

The shape of the hyperboloid can be modified by varying the exponents as follows:

exp < 1	Square shaped hyperboloids
exp = 1	Round hyperboloids
exp = 2	Flat beveled hyperboloids
exp > 2	Pinched hyperboloids

Graphics Primitives - Patches

A **patch** is a bounded collection of points used to model a surface. In RAYlib patches are rendered by first specifying a basis matrix and then defining the patch as either:

1. a set of 16 control points
2. a set of four corner points with associated tangent and twist vectors
3. four boundary curves

The basis matrix determines how the control points will be used to render the patch. Complex surfaces can be created by connecting patches.

Patches in RAYlib are tesselated into polygons based on the current patch precision.

The patch functions discussed in this section are:

- **RAYpatch_geometry3d(xgeom,ygeom,zgeom)**
- **RAYpatch_precision(nu,nv)**
- **RAYput_basis(basis,index)**
- **RAYselect_patch_basis(uindex,vindex)**

Generating Patches

Bicubic patches are used to create individual surface fragments that can be connected together to form complete surfaces of complex objects. Traditionally, patches have been used in the field of computer-aided design, for example, ship designers use patches to model ship hulls and automobile designers use patches to experiment with different body styles.

Different types of patches allow varying degrees of control over surface design. Some patches exactly interpolate the control mesh defining the patch (such as Sixteen Point Form patches) while others only loosely approximate a surface (such as periodic B-Spline Patches). The type of patch used to represent a surface depends on the surface properties required by the designer.

RAYlib provides users with a set of predefined patch types; **Bezier**, **Hermite**, **periodic B-Spline** and **Sixteen point form**. Users can also define their own patch types with arbitrary properties.

Patches are described in RAYlib in **geometric form** and are generated using the technique of *forward differences* because this technique is very fast and generates polygons that can be transformed in the pipeline. Geometric form is a matrix representation of a parametric surface. The equation describing any point on a patch is:

$$p(u,v) = U \cdot M \cdot B \cdot M^t \cdot V^t$$

The U and V vectors indicate position in the patch, M is the matrix that defines the characteristic of a patch and B is a geometry matrix, which can hold point, tangent or twist information depending on what type of patch is being generated.

Each of the predefined classes of patches is described below. To define basis matrices for other classes of patches, use the **RAYput_basis()** function discussed later in this section.

Patches are always generated as polygonal meshes and are implicitly enclosed by one bounding volume. Once the desired shape is obtained, users can employ sophisticated lighting models and texture mapping to create photorealistic complex objects.

NOTE: Texture mapped patches are currently not available.

Bezier Patches

Bezier patches are formed from a mesh of 16 control points. The four corner points actually lie on the patch; the other control points are approximated. The Bezier surface has a characteristic polyhedron of 16 points. The matrices defining the patch are:

$$\begin{bmatrix} x_0 & x_{04} & x_{08} & x_{12} \\ x_{01} & x_{05} & x_{09} & x_{13} \\ x_{02} & x_{06} & x_{10} & x_{14} \\ x_{03} & x_{07} & x_{11} & x_{15} \end{bmatrix} \quad \begin{bmatrix} y_0 & y_{04} & y_{08} & y_{12} \\ y_{01} & y_{05} & y_{09} & y_{13} \\ y_{02} & y_{06} & y_{10} & y_{14} \\ y_{03} & y_{07} & y_{11} & y_{15} \end{bmatrix} \quad \begin{bmatrix} z_0 & z_{04} & z_{08} & z_{12} \\ z_{01} & z_{05} & z_{09} & z_{13} \\ z_{02} & z_{06} & z_{10} & z_{14} \\ z_{03} & z_{07} & z_{11} & z_{15} \end{bmatrix}$$

Hermite Patch

A **Hermite** patch is defined by the following matrix:

$$\begin{bmatrix} x_{00} & x_{01} & x^{v}_{00} & x^{v}_{01} \\ x_{10} & x_{11} & x^{v}_{10} & x^{v}_{11} \\ x^{u}_{00} & x^{u}_{01} & x^{uv}_{00} & x^{uv}_{01} \\ x^{u}_{10} & x^{u}_{11} & x^{uv}_{10} & x^{uv}_{11} \end{bmatrix} \quad \begin{bmatrix} y_{00} & y_{01} & y^{v}_{00} & y^{v}_{01} \\ y_{10} & y_{11} & y^{v}_{10} & y^{v}_{11} \\ y^{u}_{00} & y^{u}_{01} & y^{uv}_{00} & y^{uv}_{01} \\ y^{u}_{10} & y^{u}_{11} & y^{uv}_{10} & y^{uv}_{11} \end{bmatrix} \quad \begin{bmatrix} z_{00} & z_{01} & z^{v}_{00} & z^{v}_{01} \\ z_{10} & z_{11} & z^{v}_{10} & z^{v}_{11} \\ z^{u}_{00} & z^{u}_{01} & z^{uv}_{00} & z^{uv}_{01} \\ z^{u}_{10} & z^{u}_{11} & z^{uv}_{10} & z^{uv}_{11} \end{bmatrix}$$

NOTE: P_{00}^{u} is the derivative of the point with respect to the parametric variable u; P^{v} is the derivative of the point with respect to v; P^{uv} is the derivative of the point with respect to u and v.

The matrix is split into four quarters. The upper left quarter defines the four corner points; the lower left quarter contains the u tangent vectors at the four corner points; the upper right quarter contains the v tangent vectors at the four corner points; the lower right corner contains the twist vector. If twist is set to zero, then the patch is a Ferguson, or F-patch. This type of patch can only have first-order continuity with adjacent patches. An F-patch is easier to specify than a fully specified Hermite patch because the twist vectors can be difficult to compute.

B-Spline Patch

The **B-Spline** surface is defined by a characteristic polyhedron, where all of the points fall within the convex hull. The patch weakly approximates the polyhedra and local deformations of control points affect only local regions of the patch. The particular type of B-Spline used here is termed *periodic*, which refers to the symmetry of the blending function used to generate the patch.

Sixteen-Point Form Patch

The **Sixteen-Point Form** patch is defined as a patch whose 16 control points actually lie on the patch. Sixteen-Point Form patches are easy to specify, particularly if the input geometry for the patch can be obtained from a device like a 3D digitizer that can accurately interpolate the points on an object.

Sixteen-Point Form patches can be contrasted with Bezier patches, where only the four corner points actually lie on the patch. The control points for Sixteen-Point Form patches are interpolated, whereas the control points for Bezier patches are approximated. Sixteen-Point Form patches do not require input of tangent or twist vectors.

RAYpatch_geometry_3d()

void RAYpatch_geometry_3d(xgeom,ygeom,zgeom)
RAYmatrix xgeom,ygeom,zgeom;

xgeom,ygeom,zgeom = a set of 3D control points

The **RAYpatch_geometry_3d** function renders a 3D surface patch using the current basis matrix and the current patch precision.

The shape of a 3D surface patch is defined by a set of user-specified 3D control points. The surface patch is rendered using the current surface model.

NOTE: One bounding volume is implicitly defined around the entire primitive.

RAYpatch_precision()

```
int RAYpatch_precision(nu,nv)
int nu,nv;
```

nu,nv = the curve's precision in the u and v directions

The **RAYpatch_precision** function specifies the number of points, lines, or polygons used to represent segments of a surface patch. The precision is specified for both the u and v directions and can be a different value for each direction. The arguments are specified as integers and must be greater than or equal to zero. Remember, the higher the number (*nu,nv*), the smoother the patch, but the longer it takes to render. If the arguments *nu,nv* are less than zero, the function returns RAY_ERR_ARG, otherwise it returns RAY_ERR_OK. The default patch precision is RAY_PATCH_DEFAULT in both the u and v directions.

RAYput_basis()

```
int RAYput_basis(basis,index)
RAYmatrix basis;
int index;
```

basis = a matrix of 16 floating point numbers

index = the index number associated with the *basis* matrix

The **RAYput_basis** function defines a 4x4 basis matrix and an associated *index* number, that can subsequently be used in rendering patches. The *index* numbers are defined by the following constants:

```
RAY_USER_BASIS_0
RAY_USER_BASIS_1
      .
      .
      .
RAY_USER_BASIS_7
```

At initialization, the first four basis matrices contain the matrix definitions for **Bezier**, **Hermite**, **B-spline** and **Sixteen-point patches**, respectively. Unless you wish to overwrite these matrices, the *index* argument passed to **RAYput_basis**() should range from RAY_USER_BASIS_4 to RAY_MAX_BASIS - 1.

If *index* is less than zero or greater than or equal to RAY_MAX_BASIS, this function returns a value of RAY_ERR_ARG, otherwise this function returns a value of RAY_ERR_OK.

Once defined, the basis matrix is selected by passing its associated *index* to the **RAYselect_patch_basis** function.

RAYselect_patch_basis()

int RAYselect_patch_basis(uindex,vindex)
int uindex,vindex;

uindex = the index to the basis matrix for the u direction

vindex = the index to the basis matrix for the v direction

The **RAYselect_patch_basis** function selects the basis matrices to be used in drawing a surface patch. A basis matrix is selected for both the u and v parametric directions of the patch. The basis matrices and their indices must have been previously defined by **RAYput_basis.** If *uindex* or *vindex* are less than zero or greater than or equal to RAY_MAX_BASIS, **RAYselect_patch_basis** returns RAY_ERR_ARG, otherwise this function returns a value of RAY_ERR_OK.

NOTE: At present, *uindex* and *vindex* must be set to the same value.

Example:

Generate a viewport with a shaded Bezier bicubic patch.

```
#include   "raylib.h"

#define          PI       3.14159265358979323846

RAYmatrix  GX = {
                          -100.0, -100.0, -100.0, -100.0,
                          -50.0, -50.0, -50.0, -50.0,
                          50.0, 50.0, 50.0, 50.0,
                          100.0, 100.0, 100.0, 100.0
                 };

RAYmatrix  GY = {
                          -100.0, -50.0, 50.0, 100.0,
                          -100.0, -50.0, 50.0, 100.0,
                          -100.0, -50.0, 50.0, 100.0,
```

continued

```
                        -100.0, -50.0, 50.0, 100.0
                };

RAYmatrix  GZ = {
                        0.0, 30.0, 70.0, 110.0,
                        -20.0, 0.0, 80.0, 90.0,
                        30.0, -30.0, 40.0, 60.0,
                        60.0, 80.0, 90.0, 20.0
                };


main(argc,argv)
        int         argc;
        char        **argv;
{
        int                     precu,precv;
        RAYlight_source         light;
        RAYsurface_model    poly_surface;


        if (RAYinit()) exit(-1);

        /* --        set patch precision                    -- */

        precu = 13;
        precv = 13;

        /* --        setup surface characteristics    -- */

        poly_surface.d_red   = 0.7;
        poly_surface.d_green = 0.3;
        poly_surface.d_blue  = 0.2;



        poly_surface.exp = 1.0;
        poly_surface.specularity = 1.0;
        poly_surface.transparent = 0.0;
        poly_surface.reflectivity = 0.0;
        poly_surface.refraction_index = 1.0;
        RAYput_surface_model( &poly_surface );

        RAYshade_mode( RAY_TRACE );

        /* -- make drop shadow                        --*/

        RAYput_viewport( 290+20, 690+20, 80+20, 400+20 );
        RAYclear_viewport( 0.1, 0.15, 0.5, 0.0 );

        /* -- create viewport and projection          --*/

        RAYput_viewport( 290, 690, 80, 400 );

        RAYpersp_project(30.0 , 1.25, 1.0, 2048.0);
```

continued

```
RAYlookup_view(185.0, 185.0, 185.0, 0.0, 0.0, 0.0, 0.0);

/* --        setup light source            -- */

RAYclear_viewport( 0.2, 0.3, 0.7, 0.0 );
RAYbackground_color( 0.5, 0.7, 0.9 );
RAYlight_ambient( 1.0, 1.0, 1.0 );
RAYambient_intensity( 0.3 );

light.nx = 1.0;
light.ny = -0.5;
light.nz = 1.0;
light.r = 0.7;
light.g = 0.7;
light.b = 0.7;

RAYput_light_source( RAY_LIGHT_DIRECT, 1, &light );
RAYlight_switch( RAY_LIGHT_DIRECT, 1, RAY_ON );

RAYpatch_precision(precu,precv);
RAYselect_patch_basis(RAY_BEZIER_BASIS, RAY_BEZIER_BASIS);

RAYscale(0.4, 0.4, 0.4);

/* --      display shaded patch    -- */

RAYpatch_geometry_3d(GX,GY,GZ);

RAYtrace();

RAYexit();
exit();
}
```

Bounding Volumes

Bounding volumes are 3D extents used to group objects that are near each other. A bounding volume is defined as the extents of the smallest cube that encloses all the objects being grouped. Bounding volumes are parallel to each coordinate axis in Viewing Space.

All objects defined after a **RAYopen_bounding_volume** function call and before its corresponding **RAYclose_bounding_volume** function call are included in the same bounding volume. When a bounding volume is encountered while tracing a given ray, the bounding volume is tested for intersection with that ray. The **contents** of the bounding volume are tested for intersections with the ray if, and only if, the ray intersects the bounding volume. Bounding volumes can be nested to further improve performance.

Although they are not required, bounding volumes are highly recommended for any RAYlib program, because they can significantly reduce the time it takes to ray trace a given scene. Proper selection of bounding volumes is largely trial and error, however, using the following guidelines should improve rendering time:

1. If an entire scene fits in a viewport, bound the entire scene.
2. Nest bounding volumes into a hierarchy of scene, groups of objects, objects, groups of primitives. Each of these levels can be further nested if they are sufficiently complex.
3. Bounding volumes work best when they envelope groups of 7 tó 20 objects (an object may be a nested bounding volume or a primitive).
4. Bounding volumes work best when the objects they bound are grouped closely together in viewing space. The smaller the volume, the better the performance.
5. Do **NOT** bound individual atoms (**RAYatom**), but try to collect those that are close to each other. Higher level objects, such as superquadrics and patches, are tessellated into simpler triangles by RAYlib. Also, RAYlib automatically creates a bounding volume for each superquadric and patch.

The effort involved in generating good, tight, nested bounding volumes will more than pay for itself with significantly reduced rendering times.

Functions in this section are:

- **RAYopen_bounding_volume()**
- **RAYclose_bounding_volume()**

RAYopen_bounding_volume()

int **RAYopen_bounding_volume()**

This function opens a bounding volume. The extents of the bounding volume are defined by the cumulative extents of each object defined up to the matching **RAYclose_bounding_volume**. Bounding volumes can be nested up to RAY_MAX_BVOL_NEST levels deep. Upon successful completion, this function returns a value of RAY_ERR_OK. If a call exceeds the maximum nesting level, RAY_MAX_BAD_BVOL is returned.

RAYclose_bounding_volume()

int RAYclose_bounding_volume()

This function closes a bounding volume opened in a matching **RAYopen_bounding_volume** call. Once this function is called no other objects are considered as part of this bounding volume, and the current extents of the bounding volume are saved. Upon successful completion, this function returns a value of RAY_ERR_OK. If you attempt to close more bounding volumes than are opened, RAY_ERR_BAD_BVOL is returned.

Example:

```
#include "raylib.h"
main()
{
        ...
        /* raylib initialization */
        if (RAYinit()) exit(100);
        ...
        RAYopen_bounding_volume();
        RAYpoly_point_3d(x1,y1,z1);
        RAYpoly_point_3d(x2,y2,z2);
        RAYpoly_point_3d(x3,y3,z3);
        RAYpoly_point_3d(x4,y4,z4);
        RAYpoly_close();
        RAYclose_bounding_volume();
        ...
        RAYtrace();
        RAYexit();
}
```

Transformations

The list below describes the three major types of transformations: **Modeling**, **Viewing** and **Projection**.

- **Modeling** transformations manipulate the **Object Coordinate System** with respect to the **World Coordinate System**. Objects are first defined in their own space, the Object Coordinate System, and then placed in the World Coordinate System by applying the modeling transformations (rotate, translate, and scale). The Object Coordinate System can be the same as the World Coordinate System, thus eliminating the transformation from Object to World Space. The World Coordinate System is a right-hand system with y to the right, z up, and x out of the page (see Figure 3-2).
- **Viewing** transformations transform World Space to Eye Space. The **Eye Coordinate System** is a right-hand system with x to the right, y up, and z out of the page. The eye is at the origin and the viewing direction is down the negative z axis (see Figure 3-3).
- **Projection** transformations map eye space into the **Screen Coordinate System**. The origin of the Screen Coordinate System is in the lower left corner with x to the right and y up (see Figure 3-4).

Primitives that are not transformed by the current transformation matrix, such as viewport definitions, are specified in the Pixel Coordinate System. The origin of the **Pixel Coordinate System** is in the upper left corner with x to the right and y down (see Figure 3-5).

Transformation Matrices

There is a matrix stack and current matrix that can be operated on. The stack contains the Modeling and Viewing transformations. Objects are transformed by the current Modeling and Viewing (MV) matrix. Viewing commands replace the current MV matrix with the specified viewing matrix. Modeling functions cause the current MV matrix to be premultiplied by the matrix representing the specified transformation. For this reason, transformations should be specified in the reverse order in which they will be applied. Typically, transformations are specified in the following order:

1. Projection transformations
2. Viewing transformations
3. Modeling transformations

Object vertices and light positions are transformed by the current set of transformation matrices. Push and pop functions can be used to localize operations by saving and restoring transformations.

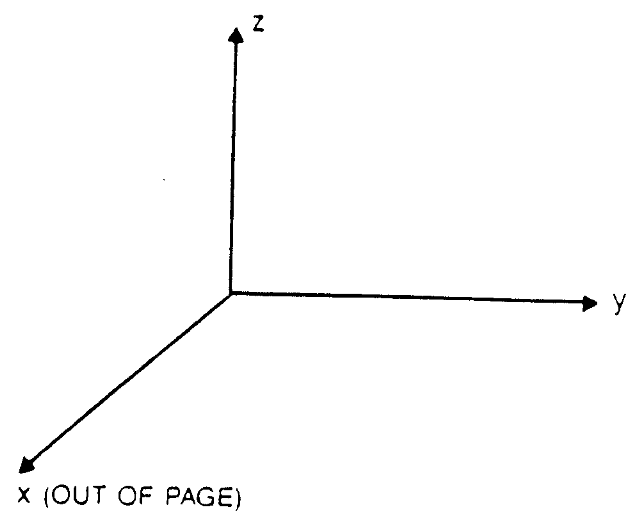

Figure 3-2. World Coordinate System

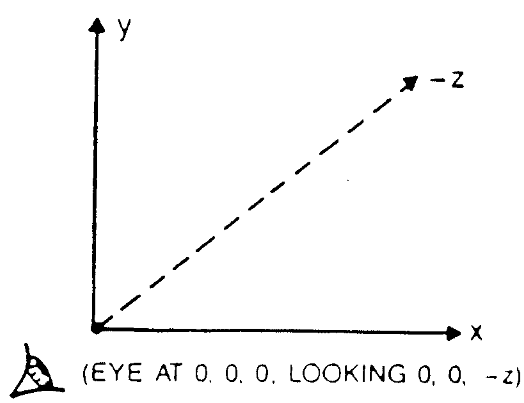

Figure 3-3. Eye Coordinate System

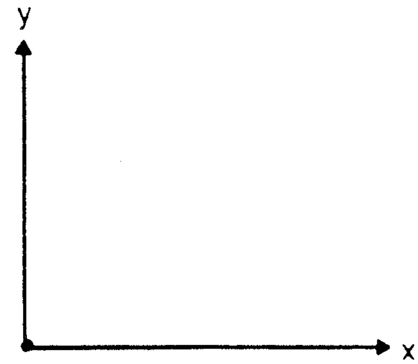

Figure 3-4. Screen Coordinate System

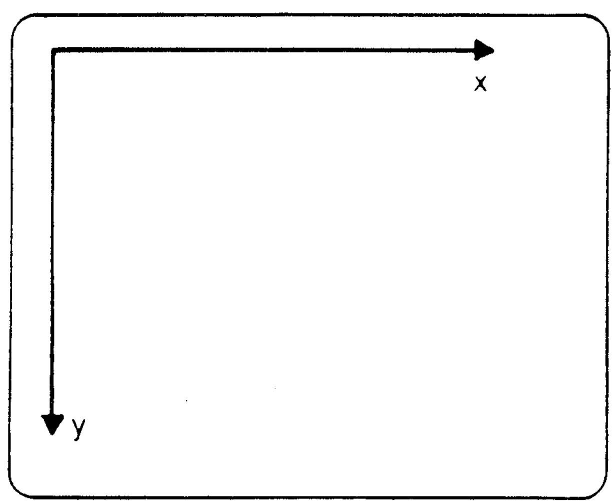

Figure 3-5. Pixel Coordinate System

Transformations - Projection Functions

The RAYlib **Projection Transformation** functions define the viewing volume and type of projection. The projection transformation maps Eye Space to Screen Space. RAYlib provides the following types of projections:

- Perspective pyramid
- Perspective window

Functions in this group are:

- **RAYpersp project(foy,aspect,near,far)**
- **RAYwindow_project(left,right,bottom,top,near,far)**

RAYpersp_project()

void RAYpersp_project(fovy,aspect,near,far)
float fovy,aspect,near,far;

fovy	=	the field-of-view angle in the *y* direction of the Eye Coordinate System
aspect	=	the ratio of the *x* and *y* dimensions of the Eye Coordinate System
near,far	=	the distances form the origin to *near* and *far* planes along the view vector. These arguments are present only for compatibility with PIClib and are ignored by RAYlib.

RAYpersp_project defines a 3D perspective viewing pyramid by specifying the field-of-view angle, *fovy*, in the *y* direction and the *aspect* ratio of the x and y directions of the Eye Coordinate System. The *fovy* and *aspect* parameters determine the size of the projection frustrum. The *aspect* ratio of the projection frustum should match the aspect ratio of the current viewport in order to display data without distortion.

NOTE

The *near* and *far* arguments are present only for compatability with PIClib and are ignored by RAYlib.

RAYwindow_project()

void RAYwindow_project(left,right,bottom,top,near,far)
float left,right,bottom,top,near,far;

left,right,bottom,top	=	the position and size of the viewing window in the *near* clipping plane, defined in the x and y dimensions of the Eye Coordinate System
near	=	the distances from the eye to the near plane. This argument is used to position the projection window.
far	=	included only for PIClib compatability. This argument is ignored by RAYlib.

The RAYwindow_project function defines a 3D perspective projection by specifying a rectangular frustum in the *near* plane. The parameters *left*, *right*, *bottom* and *top* define the position and size of the viewing window in the *near* plane. These are specified in the *x* and *y* directions of the Eye Coordinate System.

NOTE The *near* and *far* arguments have no meaning other than positioning the projection window. These arguments are included for compatability with PIClib and are otherwise ignored.

Transformations - Viewing Functions

Viewing Transformations map World Space into Eye Space, given the user's view specified by an eye position and a view direction in the World Coordinate System. RAYlib provides four viewing functions for specifying the viewpoint and viewing direction:

- **RAYlookat_view(vx,vy,vz,px,py,pz,twist)**
- **RAYlookup_view(vx,vy,vz,px,py,pz,twist)**
- **RAYcamera_view(x,y,z,pan,tilt,swing)**
- **RAYpolar_view(dist,azim,inc,twist)**

The viewing transformations are kept on the transformation stack and are pre-multiplied by the modeling transformations as they are defined. Therefore, the viewing transformations must be specified before any modeling transformations are applied.

RAYlookat_view, RAYlookup_view, RAYcamera_view, and RAYpolar_view all *replace* the current transformation with the specified viewing matrix. To preserve the current modeling and viewing transformation, use the **RAYpush_transform** command.

NOTE: All rotations discussed in this section follow the right-hand rule, unless otherwise noted. All rotations are specified in degrees.

RAYlookat_view()

void RAYlookat_view(vx,vy,vz,px,py,pz,twist)
float vx,vy,vz,px,py,pz,twist;

vx,vy,vz	=	the coordinates of the viewpoint
px,py,pz	=	the coordinates of the reference (*at*) point
twist	=	the rotation about the view vector (the -z axis of the Eye Coordinate System)

RAYlookat_view defines a viewpoint and a reference (lookat) point in World Coordinates. The viewpoint is at (*vx, vy, vz*) and the reference point is (*px, py, pz*). These two points define the view direction or view vector. The *twist* angle specifies a rotation about the view vector (directed from the viewpoint to the reference point). The view vector defines the -z axis of the Eye Coordinate System.

NOTE: **RAYlookat_view** maintains the y axis of the World Coordinate System as the up vector.

RAYlookup_view()

void RAYlookup_view(vx,vy,vz,px,py,pz,twist)
float vx,vy,vz,px,py,pz,twist;

vx,vy,vz	=	the coordinates of the viewpoint
px,py,pz	=	the coordinates of the reference (*at*) point
twist	=	the rotation about the view vector, (the -z axis of the Eye Coordinate System)

The **RAYlookup_view** function specifies the viewpoint and view direction with a *from* point and an *at* point in the World Coordinate System. These two points define the view direction or view vector. The twist angle specifies a rotation about the view vector (directed from the viewpoint to the reference point). The **RAYlookup_view** transformation ensures that the +y (up) vector of Eye Space and the +z (up) vector of World Space form an acute angle. If the view direction is (0,0,±z), then the results are the same as if the **RAYlookat_view** function had been used.

NOTE: **RAYlookup_view** maintains the z axis of the World Coordinate System as the up vector.

RAYcamera_view()

void RAYcamera_view(x,y,z,pan,tilt,swing)
float x,y,z,pan,tilt,swing;

x,y,z	=	the x, y, and z coordinates of the viewpoint
pan	=	the **left-hand rule** rotation about the y axis of the Camera Coordinate System
tilt	=	the **left-hand rule** rotation about the x axis of the Camera Coordinate System
swing	=	the left-hand rule rotation about the z axis of the Camera Coordinate System

RAYcamera_view defines a viewing transformation in terms of pan, tilt, and swing angles. The arguments to this function define a viewpoint (x,y,z) and specify a view direction by applying a *pan* degree rotation about the y axis, a *tilt* degree rotation about the x axis, and a *swing* degree rotation about the z axis of the Camera Coordinate System.

In its initial orientation, the x, y, z axes of the Camera Coordinate System are parallel to the -x, z, -y axes of the World Coordinate System. The eye is positioned at the origin of the Camera Coordinate System (defined by *x, y, z*) and the viewing vector is the positive z axis of the Camera Coordinate System. The orientation of the view vector is determined by the *pan*, *tilt* and *swing* parameters. See Figures 3-6 and 3-7. Note that the view vector in Figure 3-7 points toward the origin.

NOTE: The Camera Coordinate System is a left-hand system and all rotations in it are left-hand rotations.

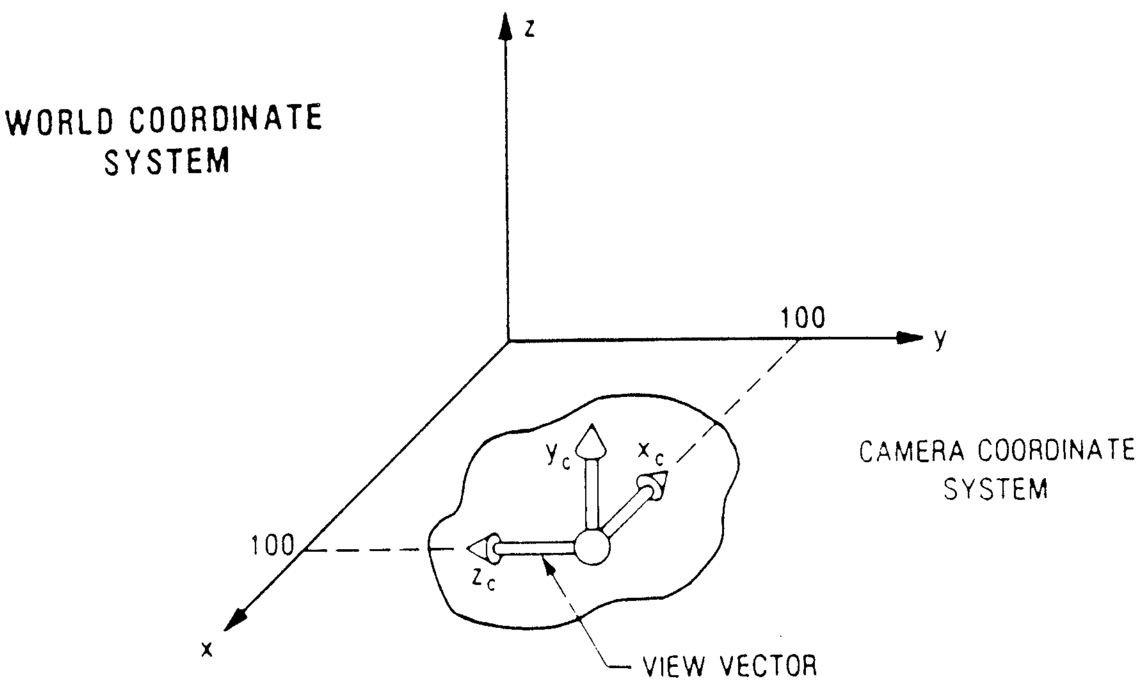

Figure 3-6. RAYcamera_view(100.0, 100.0, 0.0, 0.0, 0.0, 0.0)

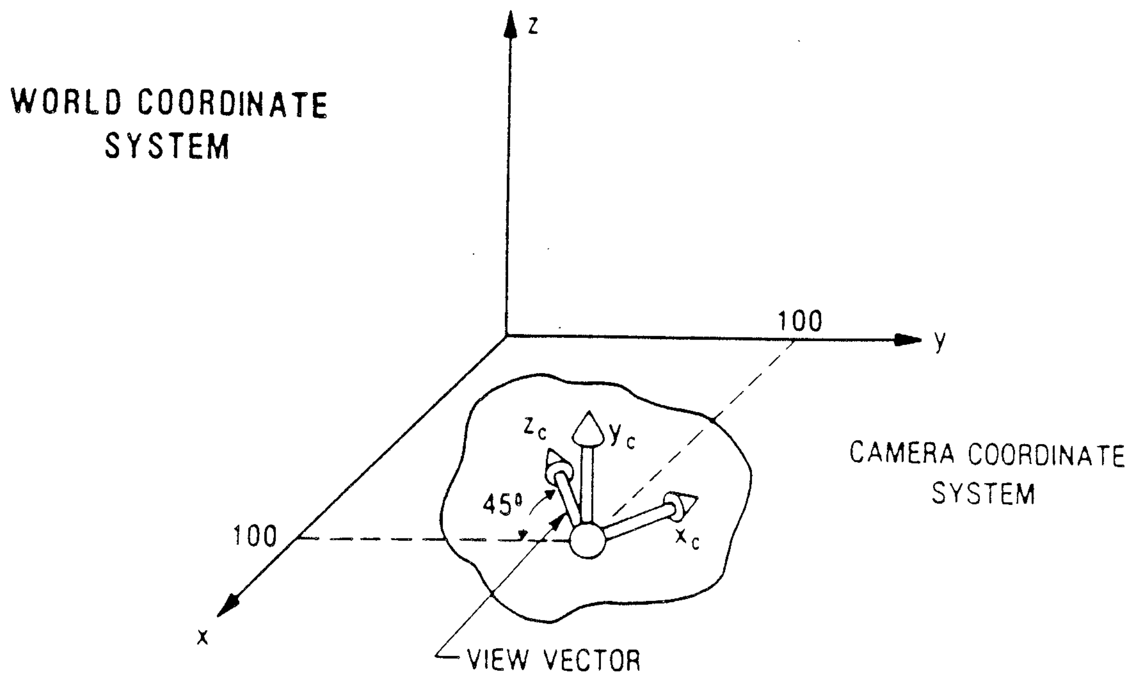

Figure 3-7. RAYcamera_view(100.0, 100.0, 0.0, 45.0, 0.0, 0.0)

RAYpolar_view()

```
void RAYpolar_view(dist,azim,inc,twist)
float dist,azim,inc,twist;
```

dist	=	the distance from the viewpoint to the origin of the World Coordinate System
azim	=	the azimuthal angle of the viewpoint in the xy plane measured from the y axis
inc	=	the incidence angle of the viewpoint in the yz plane measured from the z axis
twist	=	the rotation about the view vector (the -z axis of the Eye Coordinate System)

The **RAYpolar_view** function defines the viewpoint and direction in Polar Coordinates. The *dist* parameter is the distance from the viewpoint to the origin of the World Coordinate System. The *azim* parameter is the azimuthal angle in the xy plane, measured from the y axis. The *inc* parameter is the incidence angle in the yz plane measured from the z axis. The *twist* parameter specifies a rotation about the view vector. The view vector is directed from the viewpoint to the origin of the World Coordinate System and defines the *-z* axis of the Eye Coordinate System.

Transformations - Modeling Functions

The **Modeling Transformations** rotate, translate, and scale objects relative to the World Coordinate System. Modeling functions cause the current MV matrix to be premultiplied by the matrix representing the specified function. Because of this, modeling transformations are applied to all objects drawn after the modeling transformation is requested. The current Modeling and Viewing matrix can be saved with the **RAYpush_transform** function and restored with the **RAYpop_transform** function.

This section describes the following modeling transformation functions:

Rotation Functions

- **RAYrotate_x(x)**
- **RAYrotate_y(y)**
- **RAYrotate_z(z)**
- **RAYrotate_vector(x,y,z,nx,ny,nz,angle)**
- **RAYput_rotate_dx(dx)**
- **RAYput_rotate_dy(dy)**
- **RAYput_rotate_dz(dz)**
- **RAYrotate_dx()**
- **RAYrotate_dy()**
- **RAYrotate_dz()**

Translation Functions

- **RAYtranslate_x(x)**
- **RAYtranslate_y(y)**
- **RAYtranslate_z(z)**
- **RAYtranslate(x,y,z)**
- **RAYput_translate_dx(tx)**
- **RAYput_translate_dy(ty)**
- **RAYput_translate_dz(tz)**
- **RAYtranslate_dx()**
- **RAYtranslate_dy()**
- **RAYtranslate_dz()**

Scaling Functions

- **RAYscale_x(x)**
- **RAYscale_y(y)**
- **RAYscale_z(z)**
- **RAYscale(x,y,z)**
- **RAYput_scale_dx(sx)**
- **RAYput_scale_dy(sy)**
- **RAYput_scale_dz(sz)**
- **RAYscale_dx()**
- **RAYscale_dy()**
- **RAYscale_dz()**

NOTE: All modeling commands operate with respect to the World Coordinate System.

Rotation

Objects may be rotated with respect to *x* or *y* or *z* or an arbitrary axis. All rotations follow the right-hand rule. Positive rotations are counterclockwise when looking from the positive axis toward the origin (see Figure 3-8).

Rotations may be absolute or incremental. Absolute rotations rotate about the *x* or *y* or *z* axis by *x*, *y*, and *z* degrees. Also, arbitrary axis rotations allow you to specify an axis of rotation with a point, *x*,*y*,*z*, a direction, *nx*,*ny*,*nz*, and an angle θ. This produces a rotation of θ degrees about the specified axis with the center of rotation at *x*,*y*,*z*.

Incremental rotations rotate about the *x*,*y*, or *z* axis by a prespecified Δx, Δy, and Δz degrees.

NOTE: Positive degrees cause counterclockwise rotation; negative degrees cause clockwise rotation.

The rotation functions are:

- **RAYrotate_x(x)**
- **RAYrotate_y(y)**
- **RAYrotate_z(z)**
- **RAYrotate_vector(x,y,z,nx,ny,nz,angle)**
- **RAYput_rotate_dx(dx)**
- **RAYput_rotate_dy(dy)**
- **RAYput_rotate_dz(dz)**
- **RAYrotate_dx()**
- **RAYrotate_dy()**
- **RAYrotate_dz()**

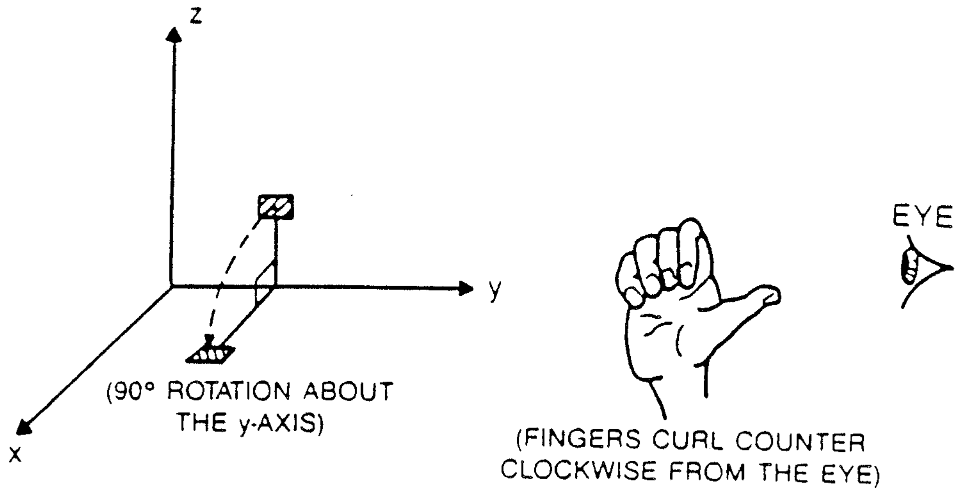

Figure 3-8. Right-Hand Rule Rotation

RAYrotate Functions

---

void **RAYrotate_x(x)**
float x;

x = the angle of rotation about the x axis

void **RAYrotate_y(y)**
float y;

y = the angle of rotation about the y axis

void **RAYrotate_z(z)**
float z;

z = the angle of rotation about the z axis

The **RAYrotate** functions (**RAYrotate_x**, **RAYrotate_y** and **RAYrotate_z**) rotate objects by a specified angle about the x or y or z axis. The angle is specified in degrees according to the right-hand rule.

RAYrotate_vector()

```
void RAYrotate_vector(x,y,z,nx,ny,nz,angle)
float x,y,z,nx,ny,nz,angle;
```

x,y,z,nx,ny,nz	=	the point (x,y,z) and direction (nx,ny,nz) that define the axis about which the object will rotate
angle	=	the angle of the rotation expressed in degrees

The **RAYrotate_vector** function rotates objects by a specified angle about an arbitrary axis. The axis of rotation is defined by a point and a direction as shown below:

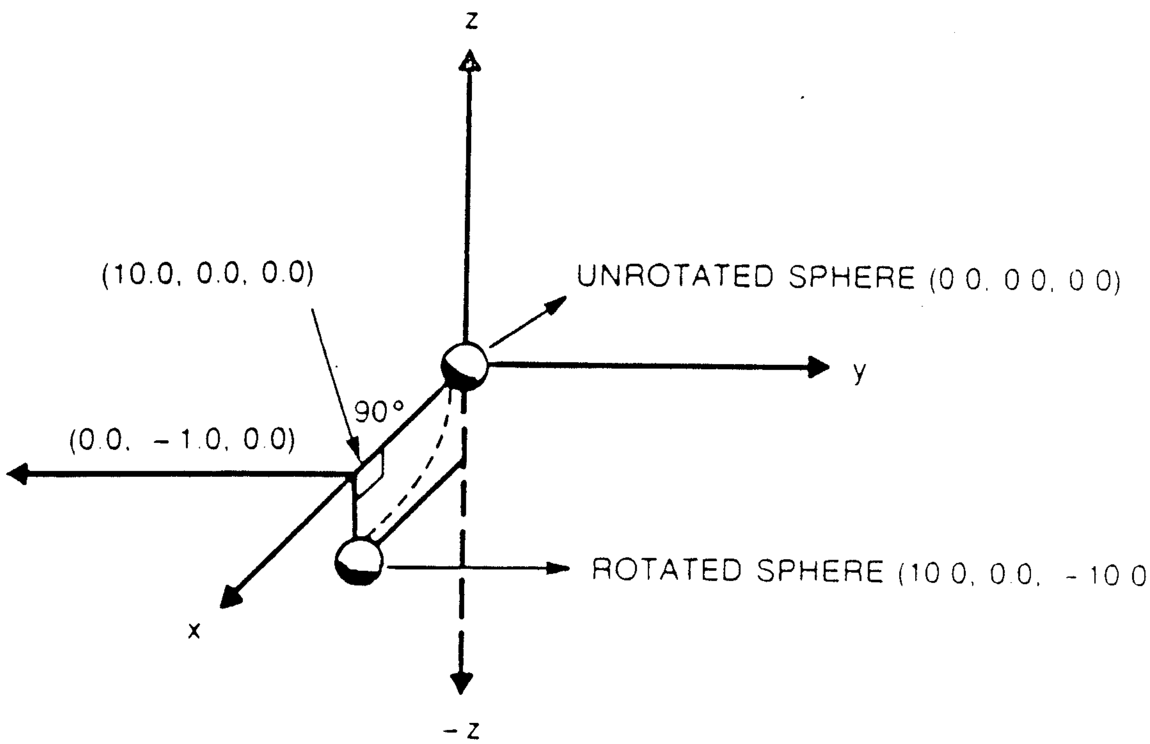

Figure 3-9. Arbitrary Axis Rotation
(RAYrotate_vector(10.0,0.0,0.0,0.0,-1.0,0.0,90.0);

Example:

The following example demonstrates how to specify a rotation of 90° about the vector defined by the point [10.0, 0.0, 0.0] and the direction [0.0, 1.0, 1.0].

```
{
.
.
RAYrotate_vector(10.0, 0.0, 0.0, 0.0, 1.0, 1.0, 90.0);
RAYsphere();                        /* draw a unit sphere at the origin */
.
.
}
```

RAYput_rotate_d Functions

void RAYput_rotate_dx(dx)
float dx;

dx = the incremental angle of rotation, in degrees, about the x axis

void RAYput_rotate_dy(dy)
float dy;

dy = the incremental angle of rotation, in degrees, about the y axis

void RAYput_rotate_dz(dz)
float dz;

dz = the incremental angle of rotation, in degrees, about the z axis

The **RAYput_rotate_d** functions (**RAYput_rotate_dx**, **RAYput_rotate_dy** and **RAYput_rotate_dz**) specify the Δ rotation about each axis. Objects can then be rotated in increments about a World Space axis (x, y, or z) using the **RAYrotate_d** functions.

RAYrotate_d Functions

void **RAYrotate_dx()**

void **RAYrotate_dy()**

void **RAYrotate_dz()**

The **RAYrotate_d** functions (**RAYrotate_dx**, **RAYrotate_dy** and **RAYrotate_dz**) rotate objects about the x, y, and/or z axis using a predefined incremental rotation. Before using any of the **RAYrotate_d** functions, be sure to specify the incremental angle with one of the **RAYput_rotate_d** functions.

Translation

Objects can be translated independently in *x* or *y* or *z* or in *xyz*. There are two types of translations: absolute and incremental. Absolute translations are applied along *x* or *y* or *z*. Incremental translations are applied along the *x* or *y* or *z* axis by a specified Δx, Δy and Δz.

The translation functions are:

- **RAYtranslate_x(x)**
- **RAYtranslate_y(y)**
- **RAYtranslate_z(z)**
- **RAYtranslate(x,y,z)**
- **RAYput_translate_dx(tx)**
- **RAYput_translate_dy**(ty)
- **RAYput_translate_dz**(tz)
- **RAYtranslate_dx()**
- **RAYtranslate_dy()**
- **RAYtranslate_dz()**

RAYtranslate Functions

void **RAYtranslate(x,y,z)**
float x,y,z;

x,y,z = the x, y, z translation

void **RAYtranslate_x(x)**
float x;

x = the x translation

void **RAYtranslate_y(y)**
float y;

y = the y translation

```
void RAYtranslate_z(z)
float z;
```

z = the z translation

The **RAYtranslate** functions (**RAYtranslate**, **RAYtranslate_x**, **RAYtranslate_y** and **RAYtranslate_z**) apply a translation along *x* or *y* or *z* to the current transformation matrix.

RAYput_translate_d Functions

```
void RAYput_translate_dx(tx)
float tx;
```

tx = the incremental translation in x

```
void RAYput_translate_dy(ty)
float ty;
```

ty = the incremental translation in y

```
void RAYput_translate_dz(tz)
float tz;
```

tz = the incremental translation in z

The **RAYput_translate_d** functions (**RAYput_translate_dx**, **RAYput_translate_dy** and **RAYput_translate_dz**) specify the delta translation along each axis. Objects can then be translated in increments along a World Space axis (x,y, or z) using the **RAYtranslate_d** functions.

RAYtranslate_d Functions

```
void RAYtranslate_dx()
void RAYtranslate_dy()
void RAYtranslate_dz()
```

The **RAYtranslate_d** functions (**RAYtranslate_dx**, **RAYtranslate_dy** and **RAYtranslate_dz**) translate the objects along the *x* or *y* or *z* axis by a predefined incremental translation. Before using any of the **RAYtranslate_d** functions, be sure to specify the incremental angle with one of the **RAYput_translate_d** functions.

Scaling

Objects can be scaled independently about *x* or *y* or *z* or about *xyz*, simultaneously. Scale commands can shrink (sx < 1), expand (sx > 1), and mirror (sx < 0) objects.

There are two types of scaling transformations: absolute and incremental. Absolute scaling is applied about *x* or *y* or *z*. Incremental scaling is applied about the *x* or *y* or *z* axis by a specified Δx, Δy, and Δz.

The scaling functions are:

- **RAYscale_x(x)**
- **RAYscale_y(y)**
- **RAYscale_z(z)**
- **RAYscale(x,y,z)**
- **RAYput_scale_dx(sx)**
- **RAYput_scale_dy(sy)**
- **RAYput_scale_dz(sz)**
- **RAYscale_dx()**
- **RAYscale_dy()**
- **RAYscale_dz()**

RAYscale Functions

```
void RAYscale(x,y,z)
float x,y,z;
```

x,y,z = the x, y, and z scaling factors

```
void RAYscale_x(x)
float x;
```

x = the x scaling factor

```
void RAYscale_y(y)
float y;
```

y = the y scaling factor

```
void RAYscale_z(z)
float z;
```

z = the z scaling factor

NOTE: A common mistake is to call **RAYscale** with only one argument. Be sure to supply *x*, *y* and *z* scaling factors even if uniform scaling is desired.

The **RAYscale** functions (**RAYscale**, **RAYscale_x**, **RAYscale_y** and **RAYscale_z**) reduce, enlarge, and mirror objects by scaling the object's x or y or z coordinates by the scaling factors x, y, and z, respectively. Objects can be scaled about one axis only or about all three axes.

NOTE: Positive scaling factors larger than *1* expand the object; less then *1*, reduce the object. Negative scaling factors mirror the scaled object across an axis.

RAYput_scale_d Functions

```
void RAYput_scale_dx(sx)
float sx;
```

sx = the incremental scaling factor in x

```
void RAYput_scale_dy(sy)
float sy;
```

sy = the incremental scaling factor in y

```
void RAYput_scale_dz(sz)
float sz;
```

sz = the incremental scaling factor in z

The **RAYput_scale_d** functions (**RAYput_scale_dx**, **RAYput_scale_dy** and **RAYput_scale_dz**) specify the delta scaling factor about each axis. Objects can then be scaled about a World Space axis (x, y or z) using the **RAYscale_d** functions.

RAYscale_d Functions

```
void RAYscale_dx()

void RAYscale_dy()

void RAYscale_dz()
```

The **RAYscale_d** functions (**RAYscale_dx, RAYscale_dy** and **RAYscale_dz**) scale the objects in *x* or *y* or *z* by a predefined incremental scale factor. Before using any of the **RAYscale_d** functions, be sure to specify the incremental angle with one of the **RAYput_scale_d** functions.

Example:

The following code fragment illustrates the use of the incremental scaling and rotation functions.

```
{
.
.
.
RAYpersp_project( 45.0, 1.25, 1.0, 1000.0 );
RAYlookup_view( 150.0, 150.0, 150.0, 0.0, 0.0, 0.0, 0.0 );

RAYput_scale_dx(3.0);       /* set the incremental x scale value   */
RAYput_rotate_dz(20.0);     /* set the incremental y rotation value */

for ( i = 0; i < MAX_ITERATIONS; i ++) {

        RAYrotate_dz();
        RAYscale_dx();

        RAYpatch_geometry_3d(GX,GY,GZ);
        RAYtrace();

        RAYswap_buffer();
.
.
}
```

Transformations - Control Functions

The **Transformation Control** functions manipulate the modeling and viewing transformation stack by pushing and popping, pre- and postmultiplying, and loading or retrieving matrices.

NOTE: RAYlib does not have the projection matrix stack that PIClib has.

The modeling and viewing transformation matrix is applied as follows:

$$\begin{bmatrix} x & y & z & w \end{bmatrix} \begin{bmatrix} C_{00} & C_{01} & C_{02} & C_{03} \\ C_{10} & C_{11} & C_{12} & C_{13} \\ C_{20} & C_{21} & C_{22} & C_{23} \\ C_{30} & C_{31} & C_{32} & C_{33} \end{bmatrix} = \begin{bmatrix} x' & y' & z' & w' \end{bmatrix}$$

Modeling and Viewing Transformation Control

The **Modeling and Viewing Transformation Control** functions operate on the current MV (Modeling and Viewing) matrix and MV stack containing the modeling and viewing transformations. These functions are:

- **RAYget_inverse_transform(matrix)**
- **RAYget_normal_transform(matrix)**
- **RAYget_transform(matrix)**
- **RAYpremultiply_transform(matrix)**
- **RAYpostmultiply_transform(matrix)**
- **RAYpush_transform()**
- **RAYpop_transform()**
- **RAYput_transform(matrix)**
- **RAYput_identity_transform()**

RAYget_inverse_transform()

int RAYget_inverse_transform(matrix)
RAYmatrix matrix;

matrix = indicates where to store the inverse of the current MV transformation matrix

The **RAYget_inverse_transform** function returns the *inverse* of the current MV transformation matrix. The inverse is computed on the host from the current MV matrix obtained from the Pixel Machine. This function *does not* change the MV transformation stack or current transformation matrix.

RAYget_inverse_transform returns RAY_ERR_INVERSE if the matrix is singular (not invertable), and RAY_ERR_OK if successful.

RAYget_normal_transform()

```
void RAYget_normal_transform(matrix)
RAYmatrix matrix;
```

matrix = indicates where to store the normal transformation matrix

---

The **RAYget_normal_transform** function returns the normal vector transformation matrix. The normal vector transformation matrix is the inverse transpose of the upper 3x3 submatrix of the current transformation matrix. This function *does not* change the MV transformation stack or current transformation matrix.

RAYget_transform()

```
void RAYget_transform(matrix)
RAYmatrix matrix;
```

matrix = indicates where to store the current transformation matrix

---

The **RAYget_transform** function returns the current 4x4 modeling and viewing transformation matrix. This function *does not* change the MV transformation stack or current transformation matrix.

RAYpremultiply_transform()

```
void RAYpremultiply_transform(matrix)
RAYmatrix matrix;
```

matrix = a user-defined 4x4 matrix

The **RAYpremultiply_transform** function premultiplies the current MV transformation matrix by a specified matrix. The result becomes the current MV matrix.

RAYpostmultiply_transform()

void RAYpostmultiply_transform(matrix)
RAYmatrix matrix;

matrix = a user-defined 4x4 matrix

The **RAYpostmultiply_transform** function postmultiplies the current MV transformation matrix by a specified matrix. The result becomes the current MV matrix.

RAYpush_transform()

void RAYpush_transform()

The **RAYpush_transform** function places a copy of the current MV transformation matrix on top of the stack. (The stack is not changed if it is full.) The MV transformation stack can be RAY_MAX_TRANSFORM levels deep.

This function is useful for saving the current transformation on the matrix stack, modifying this transformation temporarily, and then restoring its original contents by popping the transformation stack with **RAYpop_transform**.

RAYpop_transform()

void RAYpop_transform()

The **RAYpop_transform** function replaces the current transformation matrix with the transformation matrix on top of the MV stack. If the MV transformation stack is empty, **RAYpop_transform** has no effect.

Example:

The following code fragment illustrates the use of the push and pop operations on the transformation stack.

```
{
.
.
.
RAYpersp_project( 45.0, 1.25, 1.0, 1000.0 );
RAYlookup_view( 150.0, 150.0, 150.0, 0.0, 0.0, 0.0, 0.0 );

RAYpush_transform();   /* save the original coordinate system */

RAYtranslate(10.0, 10.0, 10.0);
RAYrotate_x(90.0);
RAYsuperq_torus(50.0, 50.0, 50.0, 90.0, 1.0, 2.0);

RAYpop_transform();   /* restore the original coordinate system */

RAYsphere();
}
```

RAYput_transform()

void **RAYput_transform(matrix)**
RAYmatrix matrix;

matrix = a user-defined 4x4 matrix

The **RAYput_transform** function loads a specified 4x4 matrix into the current MV transformation matrix. This function *replaces* the current MV transformation matrix with the specified matrix. If you need to save a copy of the current transformation matrix on the stack, use **RAYpush_transform** first.

RAYput_identity_transform()

void **RAYput_identity_transform()**

The **RAYput_identity_transform** function places an *identity* matrix into the current MV transformation matrix. This function *replaces* the current MV transformation matrix with the specified matrix. If you need to save a copy of the current transformation matrix on the stack, use **RAYpush_transform** first.

The identity matrix is of the form:

$$I = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Viewports

The viewport functions allows you to define an area on the screen that displays the rendered image. Viewports are defined by specifying the four limits of the rectangular area in screen units. Depending on your Pixel Machine model and configuration, the screen may be 1024x1024 or 1280x1024 in high resolution mode, 720x480 in NTSC mode and 720x576 in PAL mode. Application developers should remember that the screen space starts in the upper left with the +y axis going down.

Functions in this group are:

- **RAYget_screen_size(ix,iy)**
- **RAYput_viewport(left,right,top,bottom)**

RAYget_screen_size()

---

void RAYget_screen_size(ix,iy)
int *ix,*iy;

ix,iy	=	pointers to the memory locations that contain the screen's dimensions

---

The **RAYget_screen_size** function returns the dimensions of the screen in the x and y directions. The *x* dimension is stored in **ix**; the *y* dimension is stored in **iy**.

RAYput_viewport()

---

void RAYput_viewport(left,right,top,bottom)
int left,right,top,bottom;

left,right	=	initial and final *x* Pixel Coordinates
top,bottom	=	initial and final *y* Pixel Coordinates

The **RAYput_viewport** function defines the coordinates of the current rectangular viewport and loads it into the current viewport.

NOTE: Viewports must be defined in accordance with the screen's coordinates. The left and right coordinates range from 0 to screen_width - 1, the top and bottom coordinates range from 0 to screen_height - 1

Example:

To calculate the coordinates of a viewport of size 401x401 in the screen's center, do the following:

```
{
        int x,y;

        RAYget_screen_size(&x,&y);          /* assume 1280,1024 */

        max_x = x - 1;
        max_y = y - 1;

        left  = (max_x - 401) / 2;          /* 439 */
        right = max_x - left;               /* 840 */
        top   = (max_y - 401) / 2;          /* 311 */
        left  = max_y - top;                /* 712 */

        RAYput_viewport(left,right,top,bottom);
}
```

Shading and Lighting

RAYlib provides a wide variety of light sources and surface types. In addition to ambient light, three light source types (directional, point, and area) can simulate many lighting situations found in nature.

The light sources cast realistic shadows based on physical light models. Accurate color rendering is achieved through ray tracing methods that account for light and surface characteristics, including transparent and reflective surfaces. Texture maps can also be applied to surfaces.

A description of each type of light source is provided below:

- **Directional:** a unidirectional light source used to simulate global lighting effects. The intensity of the light reflected from the light source depends only on the orientation of the surface relative to the light source. It is independent of the relative position and distance of the surface being illuminated.

  Directional light sources are specified by color and a vector pointing toward the light.

- **Point:** an omnidirectional light source that is used to simulate localized lighting effects. The intensity of the light reflected from the light source depends on the orientation and relative position of the surface being illuminated. Point lights attenuate with distance.

  Point light sources are specified by color, position, and intensity of the light.

- **Area:** an area light source simulates diffuse light that radiates from a polygonal area, and is used to generate soft shadows in a scene. Samples are taken stochasticly from within the polygonal area. Each sample is treated as an independent point light source and behaves as such.

  Area light sources are specified by color, intensity, samples per triangle, number of vertices and list of vertices.

Up to RAY_MAX_LIGHT light sources of each type can be defined. Light sources can be turned on and off for a given scene.

NOTE! In PIClib, lights can be turned on and off for any object in a scene. In RAYlib, however, only the lights that are on when **RAYtrace** is called are used to render the scene.

The following variables are used to describe the lighting calculations presented below:

Ia	is the ambient light intensity for the scene (from **RAYambient_intensity**).
Kr	is the object's reflection coefficient (from the *reflectivity* element of RAYsurface_model).
Kt	is the object's transmission coefficient (from the *transparent* component of RAYsurface_model).
Kd(x)	is a component of the object's diffuse reflection coeffient (from the *d_** elements of RAYsurface_model).
Ks	is the object's specular reflection coefficient (from the *specularity* element of RAYsurface_model).

Vn is the normal vector at a point on the object surface.
Vl is the vector from the light source to the point on the object's surface (derived from the *x,y,z* or *nx,ny,nz* elements of RAYlight_source).
Lc(*x*) is a component of the color of the light source (from the *r, g, b* elements of RAYlight_source).
Ve is the vector from the object to the eye point.
Vr is the reflection vector from the object which is the mirror vector of *Vl* about *Vn*.
Oe is the object's specular exponent (from the *exp* element of RAYsurface_model).
I is the intensity of the light source (from the *intensity* element of RAYlight_source).
td is temporarily hardwired to 1.0.
d is the distance from the light source to the point being shaded.
d(*s*) is the distance from a sample of an anea light source to the point being shaded
x is the red, green, and blue components of light.

If a ray intersects an object, the following formula is used to determine the *x* component of the color of that ray:

$$Color(x) = Ia \cdot Kd(x) + (Kr \cdot reflect(x)) + (Kt \cdot transmitted(x)) + \textit{contribution of each light source}$$

where,

reflect(*x*) is color produced by the reflected ray, if any.
transmitted(*x*) is the the color produced by the transmitted ray, if any. The direction of the transmitted ray is affected by the relative measurement of the permativity of two surfaces (from the *refraction_index* component of RAYsurface_model)

The colors produced by the reflected and transmitted rays are evaluated as if they were primary rays at the point of ray intersection. The same calculations described here are used for the reflected and transmitted rays.

The *contribution of each light source* is determined as follows; a ray is shot from the point of intersection of the current ray and the object, toward each light source. If this shadow ray strikes an opaque object, the light source has no contribution on the current object, because it is in shadow. If the shadow ray does not intersect any objects, the contribution of the light source is defined by the formula:

$$Contribution(x) = A(d) \cdot Cd(x) + A(d) \cdot Cs(x)$$

where,

Cd(*x*) is the diffuse component of the light source given by the equation:

$$Cd(x) = Kd(x) \cdot Lc(x) \cdot (Vn \bullet Vl)$$

$Cs(x)$ is the specular component of the light source given by the equation:

$$Cs(x) = Ks * Lc(x) * (Ve \bullet Vr)^{**}Oe$$

$A(d)$ is an attenuation function which is dependent on the distance (d) of a given light source from the point being shaded.

The value of $A(d)$ for each type of light source is given as:

for direct light sources: $A(d) = 1.0$

for point light sources: $A(d) = I / (d + td)$

for area light sources: $A(d(s)) = I / (d(s) + td)$

If the shadow ray intersects one or more transparent objects, the light's contribution is computed as described above, but attenuated for each object by the formulas:

$$Contribution(red) \quad *= Kt * (1.0 - 0.5 * (Kd(green) + Kd(blue)))$$
$$Contribution(green) \quad *= Kt * (1.0 - 0.5 * (Kd(red) + Kd(blue)))$$
$$Contribution(blue) \quad *= Kt * (1.0 - 0.5 * (Kd(red) + Kd(green)))$$

NOTE: In these calculations, Kt and $Kd(x)$ are taken from the surface model of the object that intersects the shadow ray.

In the case of area light sources, a shadow ray is shot at random points in the area polygon, one shadow ray for each sample specified in the light source definition.

If a primary ray does not intersect any objects, the pixel color is set to the color defined by **RAYbackground_color**. If a reflected or transmitted ray fails to intersect any objects, its color components are set to the color defined by **RAYlight_ambient**.

If a surface is textured, the diffuse components are taken from the texture rather than the surface model. In addition, the *transparent* and *reflectivity* components may also be modified by the alpha value of the texture map. If a pixel of a texture has an 8 bit alpha value ranging from 0 to 127, it redefines the *reflectivity* component to (alpha / 127). If the textured pixel has an alpha value ranging from 128 to 255, it redefines the *transparent* component to ((alpha - 128) / 127). These components are only temporarily modified for the current pixel of the texture.

The Shading and Lighting functions available in RAYlib are:

- **RAYambient_intensity(lval)**
- **RAYlight_ambient(red,green,blue)**
- **RAYput_light_source(type,index,light)**
- **RAYlight_switch(type,index,light)**
- **RAYput_surface_model(surface)**
- **RAYset_surface_model(surface)**
- **RAYput_texture(texture,offx,offy,sizex,sizey)**
- **RAYset_texture(texture_id)**
- **RAYshade_mode(mode)**

RAYambient_intensity()

---

void **RAYambient_intensity(lval)**
float lval;

lval = intensity of the ambient light for a 3D scene. Should range from 0.0 to 1.0.

This function sets the intensity of white ambient light in a scene. Ambient light has no specific direction and does not cast a shadow in a scene. The ambient intensity should be between 0.0 and 1.0.

RAYlight_ambient()

---

void RAYlight_ambient(red,green,blue)
float red,green,blue;

red,green,blue = light component values for ambient color

This function sets the color of the ambient light for a 3D scene. **RAYlight_ambient** defines the color when a *reflected* or *transmitted* ray does not intersect any objects. **RAYbackground_color** defines the color when a *primary* ray does not intersect any objects.

Valid values for the light color components range from 0.0 to 1.0

NOTE Non-reflecting, non-transparent objects will **not** be affected by **RAYlight_ambient** as they are with the corresponding PIClib function, **PIClight_ambient**.

RAYbackground_color()

void RAYbackground_color(red,green,blue)
float red,green,blue;

red,green,blue = color values for background color of the current viewport

RAYbackground_color specifies the *rgb* color of the background. This is the color that is used if a *primary* ray does not intersect any objects in the scene. The colors are specified using an additive *rgb* color system with color components ranging from 0.0 to 1.0. **RAYlight_ambient** can be used to define the color if a *reflected* or *transmitted* ray does not intersect any objects.

NOTE It is recommended that **RAYbackground_color** be set to the same color as **RAYlight_ambient**.

RAYput_light_source()

void RAYput_light_source(type,index,light)
int type,index;
RAYlight_source *light;

type	=	RAY_LIGHT_POINT
	=	RAY_LIGHT_DIRECT
	=	RAY_LIGHT_AREA
index	=	a user-specified number ranging from 0 to RAY_MAX_LIGHT used to identify a light source.
***light**	=	pointer to the RAYlight_source structure

RAYput_light_source selects a light source. The *type* of light selected can be either point, direct or area. Up to RAY_MAX_LIGHT light sources of each *type* can be defined.

The *index* is a user-defined number ranging from 0 to RAY_MAX_LIGHT - 1 that selects a light from an array of light sources of the specified *type*. Each light source needs to be defined according to the following RAYlight_source data structure:

```
typedef struct {

        float   x, y, z;
        float   nx, ny, nz;
        float   r, g, b;
        float   exp, angle;
        float   intensity;
        long    samples;
        long    vertices;
        float   *vertex;

} RAYlight_source;
```

The elements *x*, *y*, and *z* define the position of a point light source. *nx*,*ny*, and *nz* define a vector from any object toward a direct light source. The color of light is defined by the elements *r*, *g*, and *b*. The *exp* and *angle* elements are currently unused. The intensity of point and area lights used in attenuation calculations is defined by *intensity*. *samples* defines the number of samples per triangle of area light, and *vertices* defines the number of vertices in the polygon defining an area light source. **vertex* is a pointer to the x,y and z positions of those vertices.

For point light sources, the light's position (*x*, *y*, *z*), color (*r*, *g*, *b*), and *intensity* must be defined; for direct light sources, the light's direction vector (*nx*, *ny*, *nz*) and color (*r*, *g*, *b*) must be defined; for area light sources, the color (*r*, *g*, *b*), *intensity*, number of *samples* per triangle, number of *vertices* in the area, and a list of vertices must be defined.

NOTE | Area light sources with more than 3 vertices are tesselated to multiple triangular area lights, each of which counts as 1 light source. The *intensity* and *samples* are specified per triangle.

Please keep the following points in mind:

- The viewing transformation for a scene should be defined before defining area light sources.
- Once a light source is turned on, it remains on until it is turned off.
- Only the lights that are on when **RAYtrace** is called are used to render the scene.
- There is no default setting for **RAYput_light_source**, therefore you need to specify a light source.

RAYlight_switch()

```
void RAYlight_switch(type,index,state)
int type,index,state;
```

type	=	RAY_LIGHT_POINT
	=	RAY_LIGHT_DIRECT
	=	RAY_LIGHT_AREA
	=	RAY_TYPE_ALL
index	=	user-defined number to identify light in other operations
state	=	flag indicating on/off state of the light

RAYlight_switch selectively turns light sources on (*state* = RAY_ON) or off (*state* = RAY_OFF) . Light sources are defined with **RAYput_light_source.**

The *type* argument specifies the light source type:

- RAY_LIGHT_DIRECT
- RAY_LIGHT_POINT
- RAY_LIGHT_AREA
- RAY_TYPE_ALL

The *index* argument is a user-defined number assigned to a light source that exists in a user-defined array of light sources. Indices may range from 0 to RAY_MAX_LIGHT. The constants RAY_TYPE_ALL and RAY_LIGHT_ALL can be used to manipulate all light sources simultaneously. The following constants can also be used to turn lights on and off:

- RAY_BLACKOUT - switch all light sources off
- RAY_SUNGLASSES - switch all light sources on

NOTE: Only the lights that are on when **RAYtrace** is called are used to render the scene.

RAYput_surface_model()

int RAYput_surface_model(model)
RAYsurface_model *model;

***model** = pointer to the RAYsurface_model structure

RAYput_surface_model defines a data structure,RAYsurface_model, of surface characteristics and sets the current surface model. The current surface model will remain in effect until it is overwritten by either another call to **RAYput_surface_model** or a call to **RAYset_surface_model**. The RAYsurface_model data structure is defined as follows:

```
typedef struct {

        float   a_red, a_green, a_blue;
        float   d_red, d_green, d_blue;
        float   s_red, s_green, s_blue;
        float   exp;
        float   transparent;
        float   specularity;
        float   reflectivity;
        float   refraction_index;

} RAYsurface_model;
```

The elements (*d_red, d_green, d_blue*) define the object's diffuse color. The object's degree of specularity is defined by *specularity*. The object's specular exponent is defined by *exp*. The object's transparency level is defined by *transparent* and ranges from 0.0 (no transparency) to 1.0 (full transparency). The object's reflectivity is defined by *reflectivity* and ranges from 0.0 (no reflection) to 1.0 (full reflection). The index of refraction is defined by *refraction_index*. The other components of RAYsurface_model are only present for PIClib compatibility and are ignored by RAYlib.

If a surface is textured, the diffuse components are taken from the texture as opposed to the surface model. In addition, the *transparent* and *reflectivity* components may also be modified by the alpha value of the texture map. If a pixel of a texture has an 8 bit alpha value ranging from 0 to 127, it redefines the *reflectivity* component to (alpha / 127). If the textured pixel has an alpha value ranging from 128 to 255, it redefines the *transparent* component to ((alpha - 128) / 127). These components are only temporarily modified for the current pixel of the texture.

Each time **RAYput_surface_model** is called, it allocates memory for the specified surface model and returns an integer that can be used to access that model. If an application program needs to reuse a surface model, it is most efficient to pass the integer returned by **RAYput_surface_model** to the **RAYset_surface_model** function described below. Keep in mind that each time **RAYput_surface_model** is called, memory is allocated for the new surface model description. No checking is done for duplicate surface models.

RAYset_surface_model()

```
void RAYset_surface_model(model)
int model;
```

model = index returned from the **RAYput_surface** model call

The **RAYset_surface_model** function is unique to RAYlib. It sets the current surface model to the specified *model*; which should be a value returned by **RAYput_surface_model**. To reduce memory requirements and improve the efficiency of application code, **RAYset_surface_model** should be called whenever you want to reuse a previously defined surface model.

RAYput_texture()

```
int RAYput_texture(texture,offx,offy,sizex,sizey)
unsigned long *texture;
unsigned long offx,offy,sizex,sizey;
```

texture = format of the texture

offx,offy = coordinates of the beginning of a texture residing in extended video memory

sizex,sizey = size of a texture

RAYput_texture defines a texture map. A texture can either be resident in extended video memory or in a memory array on the host. **RAYput_texture** returns an index identifying a texture that can be passed to **RAYset_texture** to set the current texture. **RAYput_texture** returns RAY_ERR_TEXTURE if you attempt to define more than RAY_MAX_TEXTURES. The format of the texture is determined by *texture*:

texture = RAY_RESIDENT_TEXTURE

The texture resides in extended video memory. The texture begins at x = *offx*, y = *offy*, and is of size *sizex* by *sizey*. Both *sizex* and *sizey* must be positive and **not** greater than 256. Both *offx* and *offy* must be positive and **not** greater than 255. The sum of *offx* and *sizex* and the sum of *offy* and *sizey* must be positive and **not** greater than 256. Resident textures may be loaded with **RAYbroadcast_data**. Once a texture is loaded, it stays in memory until the machine is powered down, another texture is loaded, or a program overwrites the texture. Textures will usually be overwritten by copies to external memory or by antialiasing in PIClib.

texture != RAY_RESIDENT_TEXTURE

The texture resides on the host and is considered to be a *virtual texture*. *texture* is a pointer to an array of *sizex* by *sizey* longwords each containing an RGBA value. Each byte of the longword is a value from 0 to 255 defining the red, green, blue, or alpha component of the texture. *offx* and *offy* are unused in this mode. *sizex* and *sizey* must be positive and less than or equal to 4096.

RAYset_texture must be called to use a texture defined with **RAYput_texture**.

If a surface is textured, the diffuse components are taken from the texture rather than the surface model. In addition, the *transparent* and *reflectivity* components may also be modified by the alpha value of the texture map. If a pixel of a texture has an 8-bit alpha value ranging from 0 to 127, it redefines the *reflectivity* component to (alpha / 127). If the textured pixel has an alpha value ranging from 128 to 255, it redefines the *transparent* component to ((alpha - 128) / 127). These components are only temporarily modified for the current pixel of the texture.

NOTE: *texture* = RAY_RESIDENT_TEXTURE is not supported on Pixel Machine models 916 and 920 in high resolution mode, however it is supported in NTSC mode on all models.

RAYset_texture()

void RAYset_texture(texture_id)
int texture_id;

texture_id = texture index

RAYset_texture sets the current texture map to the specified texture id; *texture_id* should be the value returned by the **RAYput_texture** call. Any polygons subsequently defined with **RAYpoly_point_uv** or **RAYpoly_point_nv_uv** will be textured using the specified texture map.

The default *texture_id* is 0, which is the entire 256 x 256 pixel resident texture map.

RAYshade_mode()

void RAYshade_mode(mode)
int mode;

mode = shading mode for ray tracing

This function selects the shading mode used for ray tracing. The possible values for the *mode* argument are:

- RAY_TRACE
- RAY_SHADOWS
- RAY_NOSHADOWS
- RAY_ANTIALIAS
- RAY_NO_ANTIALIAS

modes may be combined by adding them together. The default is:

RAY_TRACE + RAY_SHADOWS + RAY_NO_ANTIALIAS

NOTE: Each of the default shading modes (RAY_TRACE, RAY_SHADOWS and RAY_NO_ANTIALIAS) are defined to be 0. A mode only needs to be specified if it is not a default. For example, use RAYshade_mode(RAY_NO_SHADOWS) instead of RAYshade_mode(RAY_TRACE + RAY_NO_SHADOWS + RAY_NO_ANTIALIAS).

Antialiasing

To reduce the jagged edges that occur between objects and within textures, you can use antialiasing, which takes several stochastic (randomly placed) samples and averages them together to obtain the final value for a pixel. RAYlib uses adaptive antialiasing to achieve better quality with a minimum number of samples. In many cases, the adaptive nature of the antialiasing technique used allows for a picture quality equivalent to 100 samples per pixel at only 16 times the rendering speed of a typical unantialiased image.

Sampling passes are performed sequentially, i.e., the entire image is rendered for a sampling pass, then those pixels that require further sampling are recomputed and their values averaged, and so on. Thus, the entire image is first rendered with no antialiasing, and then it is iteratively improved. Samples are taken stochasticly within each pixel.

The function used to control antialiasing is:

- **RAYsamples(min,max,threshold)**

RAYsamples()

```
void RAYsamples(min,max,threshold)
int min,max;
float threshold;
```

min	=	the minimum number of passes the ray tracer will make in an attempt to antialias the image
max	=	the maximum number of passes the ray tracer will make in an attempt to antialias the image
threshold	=	the minimum contrast needed within a given pixel to require further antialiasing

This function establishes the minimum (*min*) and maximum (*max*) number of samples taken in any pixel during ray tracing to antialias an image. The *threshold* defines a minimum contrast needed within a pixel to require further processing; its value should range between 0.0 and 1.0

The contrast of a given pixel is computed as:

$$\frac{\textit{Max Luminence in a pixel} - \textit{Min Luminence}}{\textit{Max Luminence} + \textit{Min Luminence}}$$

where luminance is defined as:

$$0.3 * \textit{red} + 0.59 * \textit{green} + 0.11 * \textit{blue}$$

If **RAYshade_mode** has RAY_ANTIALIAS set, the minimum number of samples is taken at every pixel. Further samples are taken only for those pixels that exceed the contrast threshold. The contrast is checked after each additional sample. Sampling occurs until each pixel falls within the specified threshold or the maximum number of samples is reached.

Display Control

This set of functions performs operations on pixels, images, viewports, and data memory, such as reading/writing scan line operations. The functions are:

- RAYclear_viewport()
- RAYdouble_buffer(mode)
- RAYswap_buffer()
- RAYget_buffer_mode()
- RAYget_buffer()
- RAYput_scan_line()
- RAYget_scan_line()
- RAYbroadcast_data()
- RAYcopy_front_to_back()
- RAYcopy_back_to_ext()
- RAYcopy_ext_to_back()

RAYclear_viewport()

void RAYclear_viewport(r,g,b,a)
float r,g,b,a;

r,g,b,a = red, green, blue, and alpha indices for the viewport

RAYclear_viewport clears the current viewport to the specified *rgb* color and the overlay plane to the specified alpha (*a*) index. This function is primarily used to clear the entire screen or to display drop shadows.

NOTE: Because RAYlib will set every pixel in the current viewport when it ray traces, there is no need to clear the viewport being ray traced.

RAYdouble_buffer()

void RAYdouble_buffer(mode)
int mode;

mode = RAY_ON or RAY_OFF

The **RAYdouble_buffer** function enables or disables the use of double buffering. When enabled, objects are drawn into the *back* buffer, which is not displayed on the screen. (When in double buffering mode, use the **RAYswap_buffer** function after completing a frame.) When disabled, objects are drawn into the front buffer only, which is displayed on the screen. The default setting is RAY_OFF.

RAYswap_buffer()

void RAYswap_buffer()

The **RAYswap_buffer** function swaps the *back* and *front* buffers. This function is called during animation. Objects are drawn in the *back* buffer and displayed in the *front* buffer. (The *back* buffer is not displayed.)

NOTE Be sure to enable double buffering *before* using **RAYswap_buffer**.

RAYget_buffer_mode()

int RAYget_buffer_mode()

The **RAYget_buffer_mode** function returns an integer indicating which buffer mode is being used (single or double). RAY_SINGLE_BUFFER indicates single buffer mode; RAY_DOUBLE_BUFFER indicates double buffer mode. The default setting is RAY_SINGLE_BUFFER.

RAYget_buffer()

int RAYget_buffer()

The **RAYget_buffer** function returns an integer indicating the number of the current display buffer. The number is either RAY_BUFFER_ZERO or RAY_BUFFER_ONE. When you initialize RAYlib, the front buffer is RAY_BUFFER_ZERO (this buffer is displayed on the screen) and the back buffer is RAY_BUFFER_ONE.

RAYput_scan_line()

void RAYput_scan_line(ix,iy,red,green,blue,alpha,npixl,mode)
int ix,iy;
RAYpixel *red, *green, *blue, *alpha;
int npixl,mode;

ix,iy	=	the coordinates of the scan line. The left-most pixel of the scan line is positioned at Pixel Coordinates (*ix,iy*) (see Figure 3-5).
red,green,blue,alpha	=	arrays that determine the color of each pixel
npixl	=	the number of pixels in the scan line. **RAYput_scan_line** can write an individual pixel by setting *npixl* to one.

mode	=	RAY_RGB_PIXELS	Each pixel is 24 bits of rgb; 8 bits from each *red, green, blue* array.
	=	RAY_RGBA_PIXELS	Each pixel is 32 bits of rgbα; 8 bits from each *red, green, blue, alpha* array.
	=	RAY_RGBA_PACKED_PIXELS	Each pixel is 32 bits of rgbα from a packed array pointed to by *red*. The pixel components are stored in *rgbα* order. The first byte in *red* contains the red component of the first pixel.
	=	RAY_ABGR_PACKED_PIXELS	Each pixel is 32-bits of rgbα from a packed array pointed to by *red*. The pixel components are stored in *αbgr* order. The first byte in *red* contains the alpha component of the first pixel.
	=	RAY_RGB_ENCODED_PIXELS	Each pixel is 24 bits of rgb; 8 bits from each *red, green, blue* array. The *alpha* array contains count numbers that determine how many pixels of the same color are to be written. A count number can range from 0, indicating that the run is 1 pixel long, to 255, indicating that the run is 256 pixels long. In this mode, *npixl* refers to the number of runs in the scan line.
	=	RAY_EXTENDED_VRAM	If RAY_EXTENDED_VRAM is added to *mode*, the scan line is written into the extended video memory.
	=	RAY_COMPOSITE	Combines current image on screen with input scan line according to the formula:

$$RGB_{scrn} = RGB_{scrn} * \alpha + RGB_{in} * (1 - \alpha)$$

The **RAYput_scan_line** function lets you write a scan line of rgb or rgbα pixels to the screen by specifying the location of the first (left-most) pixel (*ix,iy*); the number of pixels, *npixl*; the color of each pixel, *red, green, blue, alpha*, which are arrays of length *npixl*; and the format of the pixels (*mode*).

NOTE: If the system is in double-buffer mode, the scan line will be written to the write buffer *not* the display buffer.

RAYget_scan_line()

int RAYget_scan_line(ix,iy,red,green,blue,alpha,npixl,mode)
int ix,iy;
RAYpixel *red, *green, *blue, *alpha;
int npixl;
int mode;

ix,iy	=	the coordinates of the scan line. The left-most pixel of the scan line is positioned at Pixel Coordinates (*ix,iy*). (See Figure 3-5).
red,green,blue,alpha	=	arrays to store the scan line
npixl	=	the number of pixels in the scan line. **RAYget_scan_line** can read an individual pixel by setting *npixl* to one.

mode	=	RAY_RGB_PIXELS	Each pixel is 24 bits of rgb (8 bits stored to each *red, green, blue* array).
	=	RAY_RGBA_PIXELS	Each pixel is 32 bits rgbα (8 bits stored to each *red, green, blue, alpha* array)
	=	RAY_RGBA_PACKED_PIXELS	Each pixel is 32 bits of rgbα stored to a packed array pointed to by *red*. The pixel components are stored in rgbα order. The first byte in *red* contains the red component of the first pixel.
	=	RAY_ABGR_PACKED_PIXELS	Each pixel is 32 bits of rgbα written to an array pointed to by *red*. The pixel components are stored in αbrg. order. The first byte in *red* contains the alpha component of the first pixel.

mode	=	RAY_RGB_ENCODED_PIXELS	Each pixel is 24 bits of rgb; 8 bits from each *red, green, blue* array. The *alpha* array contains count numbers that determine how many pixels of the same color were read. A count number can range from 0, indicating that the run is 1 pixel long, to 255, indicating that the run is 256 pixels long. In this mode, *npixl* refers to the number of runs in the scan line.
	=	RAY_EXTENDED_VRAM	If RAY_EXTENDED_VRAM is added to *mode*, the scan line is read from the extended video memory.

The **RAYget_scan_line** function lets you read a scan line of rgb or rgbα pixels from the screen by specifying the location of the first (left-most) pixel of the scan line, (*ix,iy*); the number of pixels in the scan line, *npixl*; and the format used to read the pixels, *mode*.

NOTE: If the system is in double-buffer mode, the scan line will be read from the write buffer and *not* the display buffer. It is recommended that you call **RAYwait_psync()** before the first call to **RAYget_scan_line**. This ensures that the entire frame has been drawn before any scan lines are read.

RAYbroadcast_data()

```
void RAYbroadcast_data(memory,ix,iy,data,nword)
int memory, ix, iy;
int *data;
int nword;
int mode;
```

memory	=	RAY_BROADCAST_VRAM	
ix,iy	=	the starting *x* and *y* memory addresses	
data	=	an array of 32-bit words	
nword	=	the number of 32-bit words to be broadcast	
mode	=	RAY_RGBA_PACKED_PIXELS	Each pixel is 32 bits of rgbα from a packed array pointed to by *data*. The pixel components are stored in *rgbα* order. The first byte in *data* contains the red component of the first pixel.

mode = RAY_ABGR_PACKED_PIXELS Each pixel is 32-bits of rgbα from a packed array pointed to by *data*. The pixel components are stored in αbgr order. The first byte in *data* contains the alpha component of the first pixel.

The **RAYbroadcast_data** function broadcasts a *line* of data to extended video memory (*memory* = RAY_BROADCAST_VRAM). The data consists of 32-bit words stored in an array *data*.

If the data is broadcast to the extended video memory, each 32-bit word should be organized as four 8-bit pixel components. These components can be stored in rgbα or in αbgr order depending on the parameter *mode*. The number of 32-bit words of data to be broadcast is set by *nword*. The starting x and y memory addresses are *ix,iy*. A common use of **RAYbroadcast_data** is to broadcast textures to VRAM so that all nodes receive the same data.

RAYcopy_front_to_back()

---

void RAYcopy_front_to_back()

The **RAYcopy_front_to_back** function copies the contents of the current viewport from the front buffer to the back buffer. This function is useful when doing double buffered animation.

RAYcopy_back_to_ext()

---

void RAYcopy_back_to_ext(buffer,ix,iy)
int buffer;
int ix, iy;

buffer = RAY_TOP_BUFFER
= RAY_BOTTOM_BUFFER
= RAY_SCREEN_BUFFER

ix, iy = coordinates in an off-screen image buffer. These coordinates are used with the RAY_SCREEN_BUFFER constant to specify where in the off-screen image buffer to copy the contents of the current viewport.

The **RAYcopy_back_to_ext** function copies the contents of the current viewport from the back buffer to an extended buffer. There are two available extended buffers: RAY_TOP_BUFFER and RAY_BOTTOM_BUFFER. These are used for copying rgb planes to off-screen memory for 3D compositing and other purposes. When *buffer* is set to RAY_SCREEN_BUFFER, the extended memory is treated as a single large buffer, and you need to specify the location (*ix,iy*) indicating where to place the contents of the current viewport.

The size of the off-screen buffer varies, depending on the model being used. Consult the table below to determine the off-screen buffer size for your model. RAY_SCREEN_BUFFER should be used to create flipbooks or to scroll through large images.

Model	Off-screen Buffer Size
964x	2048x2048
964	2048x2048
964n	2048x2048
940	1280x2048
940n	1280x2048
932	1024x2048
932n	1024x2048
920	-
920n	1280x1024
916	-
916n	1024x1024

Figure 3-10.

Since each Pixel Node processor only has access to every other Nx x Ny pixels on the screen, the *ix,iy* values have to be chosen carefully when copying to/from RAY_SCREEN_BUFFER. For example, if the current viewport starts at a multiple of Nx x Ny pixels on the screen, then the *ix,iy* offset values would also have to be a multiple of Nx and Ny. The table below lists the Nx and Ny values for the various Pixel Machine models.

Model	Nx	Ny
964	8	8
940	10	8
932	8	8
920	10	8
916	8	8

Figure 3-11.

NOTE

x values in multiples of 40 and y values in multiples of 8 will work for all models.

RAYcopy_ext_to_back()

```
void RAYcopy_ext_to_back(buffer,ix,iy)
int buffer;
int ix, iy;
```

buffer = RAY_TOP_BUFFER
= RAY_BOTTOM_BUFFER
= RAY_SCREEN_BUFFER

ix, iy = coordinates in an off-screen image buffer. These coordinates are used with the RAY_SCREEN_BUFFER constant to specify what part of the off-screen image buffer to copy into the current viewport.

The **RAYcopy_ext_to_back** function copies a region from an extended buffer to the current viewport. For a description of the possible *buffer* types and use of the *ix,iy* coordinates, refer to the discussion on **RAYcopy_back_to_ext** above.

Video Functions

The *Video* functions allow you to manipulate the color lookup tables and query their current status. This section discusses the following functions:

- **RAYupdate_map(mode)**
- **RAYput_color_map(red,green,blue)**
- **RAYput_color_map_entry(index,red,green,blue)**
- **RAYput_alpha_map(red,green,blue)**
- **RAYput_alpha_map_entry(index,red,green,blue)**
- **RAYget_color_map(red,green,blue)**
- **RAYget_color_map_entry(index,red,green,blue)**
- **RAYget_alpha_map(red,green,blue)**
- **RAYget_alpha_map_entry(index,red,green,blue)**

NOTE: Before altering the color map entries or alpha map entries, **RAYupdate_map**(RAY_OFF) should be called. After the updates are complete, call **RAYupdate_map**(RAY_ON) to use the new color entries.

RAYupdate_map()

```
void RAYupdate_map(mode)
int mode;
```

mode = RAY_ON or RAY_OFF

RAYupdate_map enables updating of the video lookup tables from the shadow lookup tables when *mode* = RAY_ON. When *mode* = RAY_OFF, changes to the lookup tables are not displayed until the function is re-enabled. Updating of the video lookup tables should be disabled before calling any **RAYput_color** or **RAYput_alpha** map or map entry commands.

RAYput_color_map()

```
void RAYput_color_map(red,green,blue)
float *red,*green,*blue;
```

RAYput_color_map loads an entire lookup table, defined by the *red, green,* and *blue* arrays, for the rgb channel. The *red, green* and *blue* arrays are of length RAY_VIDEO_TABLE and consist of floating point values between 0.0 and 1.0.

RAYput_color_map_entry()

---

int RAYput_color_map_entry(index,red,green,blue)
int index;
float red,green,blue;

index = indicates which entry is being updated

RAYput_color_map_entry loads a specified color into the rgb color map. *index* can range from 0 to RAY_VIDEO_TABLE - 1. The parameters *red, green* and *blue* are floating point values between 0.0 and 1.0. **RAYput_color_map_entry** returns RAY_ERR_ARG if *index* is out of range, otherwise RAY_ERR_OK is returned.

RAYput_alpha_map()

---

void **RAYput_alpha_map(red,green,blue)**
float *red,*green,*blue;

RAYput_alpha_map loads an entire lookup table, defined by the *red, green,* and *blue* arrays, for the alpha channel. The *red, green* and *blue* arrays are of length RAY_VIDEO_TABLE and consist of floating point values between 0.0 and 1.0.

RAYput_alpha_map_entry()

---

int RAYput_alpha_map_entry(index,red,green,blue)
int index;
float red, green, blue;

index = indicates which entry is being updated

RAYput_alpha_map_entry loads a specified entry into the color map for the alpha channel. *index* can range from 0 to RAY_VIDEO_TABLE - 1. The parameters *red*, *green*, and *blue* are floating point values between 0.0 and 1.0. **RAYput_alpha_map_entry** returns RAY_ERR_ARG if *index* is out of range, otherwise it returns RAY_ERR_OK.

RAYget_color_map()

```
void RAYget_color_map(red,green,blue)
float *red,*green,*blue;
```

RAYget_color_map_entry returns a specified rgb entry from the current rgb lookup table. *index* can range from 0 to RAY_VIDEO_TABLE - 1.

RAYget_color_map_entry()

```
int RAYget_color_map_entry(index,red,green,blue)
int index;
float *red,*green,*blue;
```

RAYget_color_map_entry returns a specified rgb entry from the current rgb lookup table. *index* can range from 0 to RAY_VIDEO_TABLE - 1. **RAYget_color_map_entry** returns RAY_ERR_ARG if *index* is out of range, otherwise it returns RAY_ERR_OK.

RAYget_alpha_map()

```
void RAYget_alpha_map(red, green, blue)
float *red,*green,*blue;
```

RAYget_alpha_map returns arrays containing the current r, g, and b values in the alpha map. Each *red*, *green*, and *blue* array is of length RAY_VIDEO_TABLE.

RAYget_alpha_map_entry()

```
int RAYget_alpha_map_entry(index,red,green,blue)
int index;
float *red,*green,*blue;
```

RAYget_alpha_map_entry returns a specified rgb alpha map entry. *index* can range from 0 to RAY_VIDEO_TABLE - 1. **RAYget_alpha_map_entry** returns RAY_ERR_ARG if *index* is out of range, otherwise it returns RAY_ERR_OK.

A Appendix A

Definition of Constants

Constant	Value
RAY_FALSE	0
RAY_TRUE	(! RAY_FALSE)
RAY_OFF	0
RAY_ON	(! RAY_OFF)
RAY_ERR_TEXTURE	1
RAY_ERR_OK	0
RAY_ERR_ARG	1
RAY_ERR_OPEN	2
RAY_ERR_NODE	3
RAY_ERR_FILE	4
RAY_ERR_LOAD	5
RAY_ERR_INVERSE	6
RAY_WARN_NO_OBJ	7
RAY_ERR_INTERNAL	8
RAY_HALTED	9
RAY_ERR_BAD_BVOL	10
RAY_RESIDENT_TEXTURE	0
RAY_VIRTUAL_TEXTURE	2
RAY_MAX_TEXTURES	63
RAY_MAX_TRANSFORM	32
RAY_MAX_BASIS	8
RAY_QUADRIC_DEFAULT	16
RAY_PATCH_DEFAULT	16
RAY_BEZIER_BASIS	0
RAY_HERMITE_BASIS	1
RAY_FOUR_POINT_BASIS	2
RAY_B_SPLINE_BASIS	3
RAY_USER_BASIS_0	0
RAY_USER_BASIS_1	1
RAY_USER_BASIS_2	2
RAY_USER_BASIS_3	3
RAY_USER_BASIS_4	4
RAY_USER_BASIS_5	5
RAY_USER_BASIS_6	6
RAY_USER_BASIS_7	7

Constant	Value
RAY_LIGHT_DIRECT	1
RAY_LIGHT_POINT	2
RAY_LIGHT_SPOT	3
RAY_LIGHT_CONE	4
RAY_LIGHT_AREA	5
RAY_LIGHT_ALL	1
RAY_TYPE_ALL	1
RAY_BLACKOUT	RAY_OFF
RAY_SUNGLASSES	RAY_ON
RAY_MAX_LIGHT	16
RAY_VIDEO_TABLE	256
RAY_STATISTICS	1
RAY_TIMINGS	2
RAY_PAGE_STATISTICS	4
RAY_ALL_STATISTICS	(RAY_STATISTICS + RAY_TIMINGS + RAY_PAGE_STATISTICS)
RAY_TRACE	0
RAY_NO_SHADOWS	1
RAY_SHADOWS	0
RAY_ANTIALIAS	2
RAY_NO_ANTIALIAS	0
RAY_LIMIT_REFLECTIONS	1
RAY_LIMIT_TRANSPARENCY	2
RAY_MAX_TREE_DEPTH	16
RAY_MAX_BVOL_NEST	100
RAY_IMAGE_PIXELS	2048
RAY_RGB_PIXELS	0
RAY_RGBA_PIXELS	1
RAY_RGBA_PACKED_PIXELS	2
RAY_ABGR_PACKED_PIXELS	3
RAY_RGB_ENCODED_PIXELS	4
RAY_RGB_PACKED_ENCODED_PIXELS	5
RAY_COMPOSITE	6
RAY_TOP_BUFFER	0x0200
RAY_BOTTOM_BUFFER	0x0280
RAY_SCREEN_BUFFER	0x0600

Constant	Value
RAY_EXTENDED_VRAM	0xf0
RAY_BROADCAST_VRAM	0

B Appendix B

Type Definitions

Because RAYlib and PIClib are compatible in syntax, many of the supporting data structures (typedefs) are the same, or as close as possible. In some cases, particular variables may be meaningful for one library but meaningless in the other. To maintain compatibility, structures often include names to for both libraries. As a result, some variables necessary for one library are ignored by another.

```
typedef float   RAYmatrix[4][4];

typedef struct          {
                        float       x, y, z;
                        float       nx, ny, nz;
                        float       r, g, b;
                        float       exp, angle;
                        float       intensity;
                        long        samples,  vertices;
                        float       *vertex;
                        } RAYlight_source;


typedef struct          {
                        float       a_red, a_green, a_blue;
                        float       d_red, d_green, d_blue;
                        float       s_red, s_green, s_blue;
                        float       exp;
                        float       transparent;
                        float       specularity;
                        float       reflectivity;
                        float       refraction_index;
                        } RAYsurface_model;

typedef unsigned char   RAYpixel;

typedef struct { RAYpixel red, green, blue; } RAYrgb_pixel;

typedef struct { RAYpixel red, green, blue, alpha; } RAYrgba_pixel;

typedef struct { RAYpixel alpha, blue, green, red; } RAYabgr_pixel;
```

C Appendix C

RAYlib Function Return Types

RAYlib Function Return Types

extern void	RAYambient_intensity();
extern void	RAYantialias_range();
extern void	RAYatom();
extern void	RAYbackground_color();
extern void	RAYbroadcast_data();
extern void	RAYcamera_view();
extern void	RAYclear_viewport();
extern int	RAYclose_bounding_volume();
extern void	RAYcopy_back_to_ext();
extern void	RAYcopy_ext_to_back();
extern void	RAYcopy_front_to_back();
extern void	RAYdouble_buffer();
extern void	RAYexit();
extern void	RAYexit_immediate();
extern void	RAYget_alpha_map();
extern int	RAYget_alpha_map_entry();
extern int	RAYget_buffer();
extern int	RAYget_buffer_mode();
extern void	RAYget_color_map();
extern int	RAYget_color_map_entry();
extern int	RAYget_inverse_transform();
extern void	RAYget_normal_transform();
extern int	RAYget_scan_line();
extern void	RAYget_screen_size();
extern void	RAYget_transform();
extern void	RAYhalt();
extern int	RAYinit();
extern void	RAYlight_ambient();
extern void	RAYlight_switch();
extern void	RAYlookat_view();
extern void	RAYlookup_view();
extern int	RAYopen_bounding_volume();
extern void	RAYpatch_geometry_3d();
extern int	RAYpatch_precision();
extern void	RAYpersp_project();
extern void	RAYpolar_view();
extern void	RAYpoly_close();
extern void	RAYpoly_point_3d();
extern void	RAYpoly_point_nv();
extern void	RAYpoly_point_nv_uv();
extern void	RAYpoly_point_uv();
extern void	RAYpop_transform();
extern void	RAYpostmultiply_transform();
extern void	RAYpremultiply_transform();
extern void	RAYpush_transform();
extern void	RAYput_alpha_map();
extern int	RAYput_alpha_map_entry();
extern int	RAYput_basis();
extern void	RAYput_color_map();

```
extern int      RAYput_color_map_entry();
extern void     RAYput_identity_transform();
extern void     RAYput_light_source();
extern int      RAYput_limit();
extern void     RAYput_rotate_dx();
extern void     RAYput_rotate_dy();
extern void     RAYput_rotate_dz();
extern void     RAYput_scale_dx();
extern void     RAYput_scale_dy();
extern void     RAYput_scale_dz();
extern void     RAYput_scan_line();
extern int      RAYput_surface_model();
extern int      RAYput_texture();
extern void     RAYput_transform();
extern void     RAYput_translate_dx();
extern void     RAYput_translate_dy();
extern void     RAYput_translate_dz();
extern void     RAYput_viewport();
extern int      RAYquadric_precision();
extern void     RAYrotate_dx();
extern void     RAYrotate_dy();
extern void     RAYrotate_dz();
extern void     RAYrotate_vector();
extern void     RAYrotate_x();
extern void     RAYrotate_y();
extern void     RAYrotate_z();
extern void     RAYsamples();
extern void     RAYscale();
extern void     RAYscale_dx();
extern void     RAYscale_dy();
extern void     RAYscale_dz();
extern void     RAYscale_x();
extern void     RAYscale_y();
extern void     RAYscale_z();
extern int      RAYselect_patch_basis();
extern void     RAYset_surface_model();
extern void     RAYset_texture();
extern void     RAYshade_mode();
extern void     RAYsphere();
extern void     RAYstatistics();
extern void     RAYsuperq_ellipsoid();
extern void     RAYsuperq_hyper1();
extern void     RAYsuperq_hyper2();
extern void     RAYsuperq_torus();
extern void     RAYswap_buffer();
extern int      RAYtrace();
extern void     RAYtranslate_dx();
extern void     RAYtranslate_dy() ;
extern void     RAYtranslate_dz();
extern void     RAYtranslate_x();
extern void     RAYtranslate_y();
```

```
extern void    RAYtranslate_z();
extern void    RAYupdate_map();
extern void    RAYwindow_project();
```