Contents

Figures and Tables

P Preface

Important Notice to Users

This manual is intended for use by qualified computer and engineering professionals in accordance with generally accepted engineering practices and principles. AT&T reserves the right to revise this manual for any reason, including, but not limited to, conformity with standards promulgated by IEEE, ANSI, EIA, CCITT, ECMA or similar agencies; utilization of new advances in the state of technical arts; or to reflect changes in design of equipment or services described therein. While every effort has been made to ensure the accuracy of all information in this manual, AT&T expressly and absolutely disclaims any liability to any party of any kind in this PIClib *User's Guide*, its updates, supplements, or special editions, whether such errors are omissions or statements resulting from negligence, accident or any other cause.

Warning

The software described in this *User's Guide* runs on equipment that generates, uses, and can radiate radio frequency energy, and if not installed and used in accordance with the instructions manual, may cause interference to radio communications. It has been tested and found to comply with the limits for a Class A computing device pursuant to Subpart J of Part 15 of FCC Rules, which are designed to provide reasonable protection against such interference when operated in a commercial environment. Operation of this equipment in a residential area is likely to cause interference in which case the users at their own expense will be required to take whatever measures may be required to correct the interference.

LIMITED WARRANTY

AT&T warrants this Pixel Machine is to be in good working order for a period of ninety (90) days from the date of purchase from AT&T or an authorized AT&T dealer. Should this product fail to be in good working order at any time during this 90-day warranty period, AT&T will, at its option, repair or replace this product at no additional charge except as set forth below. Repair parts and replacement Products will be furnished on an exchange basis and will either be new, remanufactured or refurbished, at the discretion of AT&T. All replaced parts and Products become the property of AT&T. This limited warranty does not include repair or damage to the Product resulting from accident, disaster, misuse, abuse, unauthorized modification of the Product, or other events outside AT&T's reasonable control or not arising under normal operating conditions.

Limited warranty service may be obtained by returning the Product during the 90-day warranty period to an authorized AT&T dealer, or by mail or carrier, to AT&T in accordance with the instructions provided to you by the Pixel Machines Hotline (1-800-544-0097 or (201) 563-2288) and providing proof of purchase date. If this Product is returned to AT&T, you agree to insure the Product or assume the risk of loss or damage in transit, to prepay shipping charges to the designated warranty service location and to ship the Product in the original shipping container or equivalent. Contact your authorized AT&T dealer or, if purchased directly from AT&T, your AT&T Account Executive for further information.

All express or implied warranties for this product including the warranties of merchantability and fitness for a particular purpose are limited in duration to a period of 90 days from the date of purchase, and no warranties, whether express or implied, will apply after this period. Some states do not allow limitations on how long an implied warranty lasts, so the above limitations may not apply to you.

If this product is not in good working order as warranted above, your sole remedy shall be repair or replacement as provided above, in no event will AT&T or its dealers or suppliers be liable to you for any damages, including any lost profits, lost savings or other incidental or consequential damages arising out of the use of or inability to use such product, even if AT&T or an authorized dealer or supplier has been advised of the possibility of such damages, or for any claim by any other party.

Some states do not allow the exclusion or limitation of incidental or consequential damages so the above limitation or exclusion may not apply to you.

This warranty gives you specific legal rights, and you may also have other rights which may vary from state to state.

About Trademarks

Sun Workstation® is a registered trademark of Sun Microsystems, Inc.

UNIX® System, Operating System, is a registered trademark of AT&T.

WE® DSP32 Digital Signal Processor is a registered trademark of AT&T.

PIClib™ is a registered trademark of AT&T Pixel Machines.

RAYlib™ is a registered trademark of AT&T Pixel Machines.

RTSlib™ is a registered trademark of AT&T Pixel Machines.

IMGlib™ is a registered trademark of AT&T Pixel Machines.

DEVtools™ is a registered trademark of AT&T Pixel Machines.

Preface

This document presents a description of the PXM 900 Series system commands and PIClib functions and is intended for users who are familiar with C language and experienced in developing programs. The information presented here is not introductory and assumes that the reader has knowledge of basic programming concepts and the UNIX® Operating System.

1 Introduction

Pixel Machine Features

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.

The Pixel Machines offer a complete set of system commands and a powerful graphics library, PIClib, for generating a multitude of images. PIClib's functions reside in 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, Gouraud and Phong 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

The application programs are written in C. 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).

Documentation Conventions

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

- Square brackets [] indicate options; parenthesis () indicate arguments.
- Variables and user–supplied names are printed in italics.
- Constants and return values are printed in helvetica.
- 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:

  PICcircle(*x,y,r*)
  float *x,y,r*;

x,y	=	the coordinates of the circle's centerpoint
r	=	the circle's radius

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

Software Structure Overview

To set-up your software environment on the Pixel Machine for PIClib, you need two tapes: PXMtools and PIClib. PXMtools contains the system commands and data files that are not unique to any one Pixel Machine software library. These commands include what you need to initialize the machine and to set-up your environment variables, along with files containing cursor and font data.

These tapes must be loaded according to the instructions in their accompanying *Release Notes.*

The following two sections describe the directory structures and contents of PXMtools and PIClib.

PXMtools Directory Structure

PXMtools has the following directory structure:

pxmtools:
- **bin** – PXMtools commands
- **boot** – Pixel Machine DSP executables
- **cpic** – gamma correction data for calibrating various monitors
- **cursors** – bitmaps that define cursors
- **fonts** – vector fonts used with various demos
- **icons** – icons for some of the demos
- **include** – user include files
- **locks** – Pixel Machine lockfile directory–permissions must be 777.
- **man** – manual pages:
  - **man1** – source for command man pages
  - **man4** – source for image header man pages
  - **cat1** – command man pages
  - **cat4** – image header man pages

PIClib Directory Structure

PIClib has the following directory structure:

piclib:
- **bin** – Shell level PIClib utilities
  - **picclear, picdisp, picsave, piccompress, pictexture, picbroadv**
- **boot** – Pixel Machine DSP executables
- **demo** – demonstration programs
  - **bin** – executable demo programs with scripts
  - **data** – object and image data files
  - **obj** – object files
  - **src** – source files
- **include** – user include files

lib – library directory contains the following libraries:

- **piclib.a** – host library
- **piclib_p.a** – profiled host library
- **piclib_ffpa.a** – host library, floating point (Sun 3 only)
- **piclib_ffpa_p.a** – profiled host library, floating point (Sun 3 only)

– in the FORTRAN version, this directory contains:

- **piclib_ffpa_pf.a** – profiled host library, floating point, FORTRAN version (Sun 3 only)
- **piclib_ffpaf.a** – host library, floating point, FORTRAN version (Sun 3 only)
- **piclib_pf.a** – profiled host library, FORTRAN version
- **piclibf.a** – host library, FORTRAN version

man – manual pages:

- **whatis** – list of PIClib functions
- **man1** – source for command man pages
- **man3** – source for function man pages
- **man4** – source for image header man pages
- **cat1** – command man pages
- **cat3** – function man pages
- **cat4** – image header man pages

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. For more information about **hypinit**, see the manual page that came with the PXMtools commands and the section on **hypinit** in Chapter 2 of this *User's Guide.*

The software environment must be set up at installation time, after power-up, 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. You must set the Pixel Machine environment variables for each login on the host computer. These variables are set in one of the following three ways:

1. As commands typed manually from the UNIX® system prompt.
2. As statements in your **.login** and **.cshrc** files (C shell [csh]) or as statements in your **.profile** and **.env** files (Korn shell [ksh]).
3. As statements in a *system* file, such as **/etc/profile.**

Environment Variables

The **/usr/hyper** directory contains sample files for defining the Pixel Machine environment. These files are named:

.hyper_login
.hyper_cshrc
.hyper_profile
.hyper_env

If you are using **csh**, you can **source .hyper_login** and **.hyper_cshrc** into your **.login** file. Edit your **.login** file, and add the following to the end of the file:

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

If you are using **ksh**, you can . (dot) **.hyper_profile** and **.hyper_env** into your **.profile**. Edit your **.profile** and add the following to the end of the file:

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

When setting up your environment, refer to the variable descriptions below. These variables should be included in your **.profile, .login** or system file.

The **HYPER_MODEL** variable specifies the Pixel Machine model and Transformation Pipeline configuration. The table below describes the values that can be assigned to this variable.

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 indicates an NTSC model whose resolution is 720x486.

A lower case "p" appended to the model number indicates a PAL model whose resolution is XXX.

A lower case "z" appended to the model number indicates zero pipes.

An "X" appended to the model number indicates a high resolution monitor.

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. The **HYPER_GAMMA** variable controls how the color tables are updated by **hypinit**. If **HYPER_GAMMA** is set and is not null, it is used as the the name of a file that contains a gamma correction table. If **HYPER_GAMMA** is not set or is null, a linear ramp is loaded into the color tables. If **HYPER_GAMMA** does not contain an absolute pathname, it is used as a filename in the *$HYPER_PATH*/**cpic** directory. Relative pathnames are not supported.

The video control parameters are set based on the **HYPER_MODEL** and **HYPER_VIDEO** environment variables. The **HYPER_VIDEO** variable contains a string that is parsed to produce a value that is passed to **DEVset_video_options()**. The string in **HYPER_VIDEO** must be of the format:

```
sync_source={int,ext}
sync_on_green={on,off}
```

The value after the equal sign must be one of the values listed in braces. The first value is the default; spaces in the string are ignored.

Examples:

```
HYPER_VIDEO="sync_source=ext sync_on_green=off"
HYPER_VIDEO="sync_source = int"
```

In addition to defining the Pixel Machine–specific environment variables, you can also update your PATH variable(s) to provide easy access to Pixel Machine software, demos and manual pages.

To update your PATH variable, add the following directories:

$HYPER_PATH/bin — Allows you to run the PXMtools system commands (e.g., **hypinit** (see Chapter 2 for more information)) and the PIClib utilities (e.g., **picclear** (see Chapter 2 for more information)) from your current working directory.

$HYPER_PATH/piclib/demo/bin — Allows you to run PIClib demos from your current working directory.

To update your MANPATH variable, add the following directories:

$HYPER_PATH/man

$HYPER_PATH/piclib/man — Allows you to access PXMtools manual pages as well as PIClib manual pages from your current working directory.

To see a list of what your environment variables are set to, type the **hypenv** command. For more information about **hypenv**, refer to Chapter 2 of this *User's Guide* and the **hypenv(1)** manual page that came with the PXMtools.

NOTE: If you upgrade or change your present Pixel Machine, you need to redefine the environment variables.

Setting Environment Variables Using csh

The following example illustrates the **csh** commands you need to specify to define the Pixel Machine environment for a Model 964d with the transformation pipelines configured in serial mode.

NOTE: Be sure to enter each environment variable on a *separate* line.

The following can be added to your **.login** file:

```
setenv HYPER_MODEL    964d
setenv HYPER_PATH     /usr/hyper
setenv HYPER_PIPE     serial
setenv HYPER_UNIT     0
setenv MANPATH        ${MANPATH}:$HYPER_PATH/man:$HYPER_PATH/piclib/man
setenv HYPER_GAMMA
setenv HYPER_VIDEO
```

```
set path = ( ${path} $HYPER_PATH/bin $HYPER_PATH/demo/piclib/bin \
$HYPER_PATH/demo/raylib/bin)
```

Alias definitions provide a ''short–cut'' for defining variables. The following lines can be added to your **.cshrc** file.

```
alias   hypmodel   'setenv HYPER_MODEL   \!*'
alias   hypipe     'setenv HYPER_PIPE    \!*'
alias   hypunit    'setenv HYPER_UNIT    \!*'
alias   hypath     'setenv HYPER_PATH    \!*'
alias   hypgamma   'setenv HYPER_GAMMA   \!*'
alias   hypvideo   'setenv HYPER_VIDEO   \!*'
```

Once the above *aliases* have been established, you can use them to define environment variables. For example, if you need to redefine the **HYPER_PIPE** variable to designate a parallel pipeline configuration, you can type **hypipe parallel** instead of **setenv HYPER_PIPE parallel** .

Setting Environment Variables Using ksh

The following example illustrates the **ksh** commands you need to specify to define the Pixel Machine environment for a Model 964d with the transformation pipelines configured in serial mode.

```
HYPER_MODEL=964d
HYPER_PATH=/usr/hyper
HYPER_PIPE=serial
HYPER_UNIT=0
HYPER_GAMMA
HYPER_VIDEO
PATH=$PATH:$HYPER_PATH/bin:$HYPER_PATH/piclib/demo/bin:
MANPATH=$MANPATH:$HYPER_PATH/man:$HYPER_PATH/piclib/man
export HYPER_MODEL HYPER_PATH HYPER_PIPE HYPER_UNIT HYPER_GAMMA HYPER_VIDEO
export MANPATH PATH
```

Alias definitions provide a "short–cut" for defining variables. The following lines can be added to your **.env** file.

```
hypmodel() { HYPER_MODEL=$1; }
hypipe() { HYPER_PIPE=$1; }
hypunit() { HYPER_UNIT=$1; }
hypath() { HYPER_PATH=$1; }
hypgamma() { HYPER_GAMMA=$1; }
hypvideo() { HYPER_VIDEO=$1; }
```

Compiling and Running Programs

At the beginning of each C application program, you must include a header file for PIClib. This file includes type definitions, constants, and external definitions, and is included by the following statement:

```
#include     <piclib.h>
```

The *first* graphics function called within an application program is usually **PICinit()**. This function initializes and resets the PIClib library, and opens the Pixel Machine device, stapic all the nodes in the system, and resets all graphical parameters to their default values as described in **PICreset()** manual page. **PICinit()** returns a value of PIC_ERR_OK if the initialization is successful, or a PIC_ERR_OPEN if it failed. For a complete description of the functionality, see the **PICinit(3)** manual page in the PIClib *Reference Manual.*

The *last* graphics function called within an application program is usually **PICexit()**. Be sure to include it at the end of your program.

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

```
cc -I$HYPER_PATH/include [file.c] $HYPER_PATH/lib/piclib.a -lm
```

where, *[file.c]* is the name of the file containing the program. You can also link with versions of PIClib that include profiling and different floating point options.

The system will compile your program and create an executable file called **a.out**. To run the program, rename the file to whatever *file* you have chosen and type the name of this executable file.

Using Macros

The file **PICMAC.h** contains macros that you can use to speed up the processing of your programs (see Appendix A for a list of the macros. These macros avoid the overhead of calling and returning from functions and of converting floating point arguments into double precision and back into single precision. The file is located in **/usr/hyper/include**. Be sure to include it at the top of your program if you want to use the macros for faster command execution. The macro for a PIClib command is identical to the PIClib command, except the **PIC** prefix is replaced with **PICMAC** (e.g., **PIC_point_3d** becomes **PICMAC_point_3d**).

2 Commands and Utilities

Pixel Machine System Commands and Utilities

The system commands (UNIX system commands typed on the command line) and utilities (PIClib programs) allow you to perform utility and administrative functions, such as initializing the hardware, loading the PIClib 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
hypdis	Disables selected pixel boards.
hypenv	Displays current settings of environment variables.
hypfree	Releases a locked unit.
hypid	Displays node ID data.
hypinit	Initializes the current Pixel Machine.
hyplock	Locks the current Pixel Machine.
hypstat	Displays the status of the current Pixel Machine.
hypunlock	Unconditionally unlocks a Pixel Machine.
hypversion	Prints the software version.

For more information on the use of these commands, see the manual pages that came with your PXMtools software.

The system utilities described in this section are:

Utility	Function
picalpha	Turns the alpha channel display on/off
picbars	Displays the color bars on the screen
picboot	Loads the PIClib modules into the geometry and drawing nodes
picbroadv	Broadcasts a buffer of data to VRAM
picbroadz	Broadcasts a buffer of data to DRAM
picbtof	Copies contents of the back buffer to the front buffer
picdisp	Downloads and/or displays an image
picetof	Copies the contents of the extended VRAM buffer to the front buffer
picftob	Copies the contents of the front buffer to the back buffer
picftoe	Copies the contents of the front buffer to extended VRAM
picgamma	Creates gamma corrected lookup tables
picinit	Resets the Pixel Machine to its default values
piclear	Clears the screen
piclens	Interactive tool that roams around and magnifies the display
picsave	Saves an image to disk
pictexture	Displays current texture loaded into VRAM

System Commands

hypdis

hypdis writes a zero to the mode register of each pixel board in a system, thereby effectively removing the board from consideration during writes to the broadcast FIFO and during processor synchronization operations.

hypdis is typically used to reconfigure a Pixel Machine to a lower model number for testing purposes or when a pixel board becomes inoperative.

The following example shows how to configure a Pixel Machine with 64 nodes as a 940:

```
HYPER_MODEL=940d
hypdis
```

The **hypdis** command should always be followed by a **hypinit**.

NOTE: It is important to note that Pixel Machines equipped with the serial I/O feature will not work when a system is configured as a smaller model using **hypdis**.

Also note that pipe boards are unaffected by **hypdis**, therefore **hypdis** cannot be used to configure a 964d as a 964, for example.

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 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][-G][-V][-A][-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
- **−G** Print current value of **HYPER_GAMMA** environment variable
- **−V** Print current value of **HYPER_VIDEO** environment variable
- **−A** Print current value of **HYPER_ADDRESS** 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
```

WARNING

The **HYPER_ADDRESS** environment variable should **ONLY** be used with the SGI Power Series host. Please read the *SGI Release Notes* for more information on this variable. **HYPER_ADDRESS** should **NOT** be used or set with any other Pixel Machines hosts.

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 is:

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 is:

hypid [−a][−d *node***][−g** *node***][−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*
−u	Print command usage format

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

```
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] [−n] [−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 pipe nodes
−m	Initialize the pixel mode register to the current configuration model, disable overlay video, and turn off testing mode
−n	Do not initialize the video
−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	enable verbose mode
−V	Initialize video registers and lookup tables
−u	Print command usage format

If you enter **hypinit −v**, 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 no_broadcast no_fbaddress no_
reset no_local_interrupt no_bus_busy new_repeater cool_temperature no_half_full_
low no_half_full_hi no_full_low no_full_hi ].

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: 1024 ].
```

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 execute **hypstat**, the system displays the the same message as **hypinit −v**, which is described above. If you get an error message, enter the **hypinit** command first and then **hypstat**.

hypunlock

The **hypunlock** shell script can be used to unconditionally unlock a particular Pixel Machine. If *unit* is specified, that Pixel Machine is unlocked, otherwise the machine specified by **$HYPER_UNIT** is unlocked. The **hypunlock** command is typically used when someone has previously locked the Pixel Machine (using **hyplock**) and forgotten to free the Pixel Machine after using it.

Command usage is:

hypunlock [unit]

This command should only be used as a last resort when you are sure that no one else is currently using the Pixel Machine.

hypversion

hypversion with no options displays the software product, version, and date of the installed software. Specific products can have version information displayed by using the appropriate option.

Command usage is:

hypversion [−p] [−d] [−r] [−s] [−u]

The options are as follows:

- **−p** Print version of PIClib
- **−d** Print version of DEVtools
- **−r** Print version of RAYlib
- **−s** Print version of RTSlib (Simulation Library)
- **−u** Print command usage format

PIClib Utility Programs

Following are brief descriptions of the PIClib system commands. For more detailed information, see the manual pages in section 1 of the PIClib*Reference Manual.*

picalpha

picalpha turns the display of the alpha channel on or off. With an argument, it turns on the display; without an argument, the display is turned off.

picbars

picbars displays color bars on the screen. Followed by an argument, it displays logarithmic color bars, and with no argument, it displays linear color bars.

NOTE: This tool is useful for calibrating equipment such as cameras and monitors.

picboot

picboot downloads PIClib into the Pixel Machine. This command checks each pipe and pixel node to see if the correct PIClib module is loaded. If it isn't, **picboot** loads it.

picbroadv

picbroadv broadcasts pixels to extended VRAM in many formats. The following image formats are supported:

PIC_RGB_PACKED_PIXELS
PIC_RGBA_PACKED_PIXELS
PIC_ABGR_PACKED_PIXELS

This command is useful for loading texture maps.

Command usage is:

picbroadv [-p x y] [-s npixels nlines] [-o xoffset yoffset] [-d] [-v] filename

The options are as follows:

−p *x y*	Specifies the starting pixel location in the plane of memory in *x* and *y* pixel coordinates. The default is 0 0.
−s *npixels nlines*	Specifies the number of pixels and number of scan lines to download. The default is the size of the image as specified in the image header.
−o *xoffset yoffset*	Specifies the number of pixels (in the *x* and *y* direction) to offset the image by before downloading. This number of pixels and lines will be skipped in the image file. The default is 0 0.
−d	Uses the default starting pixel location that is specified in the image file. This overrides the **−p** option.
−v	Print verbose output

picbroadz

picbroadz braodcasts pixels to extended DRAM in many different formats. The following image formats are supported:

PIC_RGB_PACKED_PIXELS
PIC_RGBA_PACKED_PIXELS
PIC_ABGR_PACKED_PIXELS

Command usage is:

picbroadz [−p x y] [−s npixels nlines] [−o xoffset yoffset] [−d] [−v] filename

The options are as follows:

-p *x y* — Specifies the starting pixel location in the plane of memory in *x* and *y* pixel coordinates. The default is 0 0.

-s *npixels nlines* — Specifies the number of pixels and number of scan lines to download. The default is the size of the image as specified in the image header.

-o *xoffset yoffset* — Specifies the number of pixels (in the *x* and *y* direction) to offset the image by before downloading. This number of pixels and lines will be skipped in the image file. The default is 0 0.

-d — Uses the default starting pixel location that is specified in the image file. This overrides the -p option.

-v — Print verbose output

picbtof

picbtof copies the contents of the back buffer to the front buffer; the result is immediately seen on the display.

NOTE: This command does not take any options.

picdisp

picdisp downloads and/or displays an image in many different formats.

The image formats supported are:

PIC_RGB_PIXELS
PIC_RGB_PACKED_PIXELS
PIC_RGBA_PACKED PIXELS
PIC_ABGR_PACKED_PIXELS
PIC_RGB_ENCODED_PIXELS

Command usage is:

```
picdisp [-p initx inity] [-s npixl nline] [-o xoffset yoffset] -b [front|back|extended]
        -[vdc] filename [-C composite_mode] filename
```

The options are as follows:

−p *initx inity*	Specifies the starting pixel location on the screen in *x* and *y* pixel coordinates. The default is 0 0.
−s *npixl nline*	Specifies the number of pixels and number of scan lines to download. The default is the size of the image as specified in the image header.
−o *xoffset yoffset*	Specifies the number of pixels (in the *x* and *y* directions) to offset the image by before downloading. This number of pixels and lines will be skipped in the image file. The default is 0 0.
−b *buffer*	Specifies to which segment of memory pixels should be downloaded (front is the default). If **front** is specified, pixels are downloaded to the visible buffer of VRAM. If **back** is specified, pixels are downloaded to the invisible buffer of VRAM. If **extended** is specified, pixels are downloaded to the invisible buffer of extended VRAM.
−d	**picdisp** uses the default starting pixel location as specified in the image file. This option overrides the **−p** option.
−c	Copies the image to the front buffer. After the copy, the front and back buffers are identical.
−C	Specifies composite mode. The following modes, in either upper or lower case, are supported. NO_COMPOSITE A_OVER_B B_OVER_A A_IN_B B_IN_A A_OUT_B B_OUT_A A_ATOP_B B_ATOP_A A_XOR_B A_PLUS_B
−v	Prints verbose output

Examples:

picdisp −c −b back *filename*

displays the image in both front and back buffers.

picdisp *test.img*

In this example the image is displayed on the screen from (0,0).

picdisp **−d** *test.img*

The image is displayed on the screen in the way it was stored. That is, if the image was displayed in screen coordinate space from (500,500) to (800,800) when it was saved, the image will be displayed in the same coordinate space.

picdisp **−p 0 0 −s 255 255 −o 0 0 −b front −v** *test.img*

test.img is displayed on the screen at the starting point (0,0) with a size of 255,255. The offset into the image is (0,0), and the image is displayed in the front buffer.

NOTE

For this release ONLY, **picdisp** can also display images stored in the old image format.

picdisp *filename [initx inity [finalx finaly [ifromx ifromy [itox itoy]]]]*

picetof

picetof copies the contents of the extended VRAM buffer to the front buffer; the result is immediately seen on the display.

NOTE

picetof does not take any arguments.

picftob

picftob copies the contents of the front buffer to the back buffer. The display on the screen remains unchanged.

NOTE

picftob does not take any arguments.

picftoe

picftoe copies the contents of the front buffer to the extended VRAM buffer.

NOTE: **picftoe** does not take any arguments.

picgamma

picgamma creates gamma corrected lookup tables and loads these tables into the Pixel Machine.

Without any arguments, the gamma values used for **r, g, b** are 1.0. If one argument is specified, **r, g, b** are set to that argument. If three arguments are specified, the gamma values for **r, g, b** are set to the arguments, respectively.

picinit

picinit resets the Pixel Machine to its default values. This command is useful for returning the machine to a normal state.

NOTE: **picinit** does not take any arguments.

piclear

piclear clears the front and back buffers with the specified **rgb** on the command line. If **rgb** is not specified, it clears the screen to black. The alpha plane is set to zero, and the z-buffer is cleared to its default.

Command usage is:

```
piclear [-b] [r g b]

float r,g,b;
```

The options are as follows:

−b	Clears the back buffer only
r g b	Specifies the color to be used in clearing the buffers. r, g and b range from 0.0 to 1.0.

piclens

piclens is an interactive tool that allows the user to roam around the display and magnify segments of it. The image on the screen can be magnified up to 128 times. The image cannot be scaled down below its original size (i.e., 1). The size of the window is 256 X 256 pixels.

When **piclens** is invoked, a *mouse playground* window appears on the host machine. The three buttons on the mouse do the following:

Right button: the magnification factor is doubled

Left button: the magnification factor is halved

Middle button: sets the point to be magnified

Keyboard keys: the keyboard keys can be upper or lower case, and they do the following:

- G – toggle grid on/off in magnification window. The current pixel is highlighted with a red boundary.
- **arrow keys** – move position by one pixel in the appropriate direction. Pre-fixing the arrow keys with a number, moves the position by given amount.
- Q – quit

NOTE: **piclens** does not take any arguments.

picsave

picsave saves an image to disk in many different formats.

Command usage is:

picsave [−p initx inity] [−s npixels nlines] [−m rgb|rgba|agbr] [−b front|back] [−v] filename

The options are as follows:

−p *x y*	Specifies the starting pixel location on the screen in pixel coordinates. The default is 0,0.
−s *npixels nlines*	Specifies the number of pixels and number of scan lines to be saved. The default is the entire screen.
−m *mode*	Specifies how pixels are stored. *mode* is one of the following (**rgba** is the default): **rgb** – Pixels are stored in RED, GREEN, BLUE format, 24–bits per pixel (PIC_RGB_PACKED_PIXELS) **rgba** – Pixels are stored in RED, GREEN, BLUE, ALPHA format, 32–bits per pixel (PIC_RGBA_PACKED_PIXELS). **agbr** – Pixels are stored in ALPHA, GREEN, BLUE, RED format, 32–bits per pixel (PIC_AGBR_PACKED_PIXELS)
−b *buffer*	Specifies from which segment of VRAM pixels should be saved (**front** is the default): **front** – Save pixels from the visible buffer of VRAM **back** – Save pixels from the invisible buffer of VRAM
−v	Print verbose output

pictexture

pictexture maps the current texture loaded into offscreen VRAM on a 1K by 1K polygon and displays it in the front buffer.

NOTE: **pictexture** does not take any arguments.

3 Overview of PIClib Functions

Picking and Selecting

Overview of PIClib Functions

The Pixel Machine's graphics library, PIClib, consists of C-callable functions that allow you to create graphics primitives, curves, surface patches, transformations, texture maps, projections, zbuffering, Gouraud shading, double buffering, and anti-aliased lines and polygons and much more.

The PIClib functions are grouped into the following categories:

- **Control Functions** – initialize and exit the graphics library; load PIClib program modules into the Pipes and Pixel Nodes; reset graphical parameters to default values; Transformation Pipes (multiple Pipe configurations); enable or disable the use of DSP32 floating point format; and wait for vertical sync or Pixel Node processor sync.
- **Graphics Primitives** – render objects with points, lines or filled polygons. These functions are categorized as follow:
  - **basic functions** – render arcs and circles with specified precisions and rectangles. They also render lines and points (2D, 3D, 4D), and move the current graphics position to a new point (2D, 3D, 4D).
  - **polygon functions** – define sequentially the vertices of a polygon in 2D, 3D, or 4D coordinates and close the polygon. These functions also allow normal vectors, rgb colors, and texture map indices to be attached to individual polygon vertices.
  - **quadric/superquadric functions** – render spheres, hemispheres, cones, cylinders, ellipsoids, toroids, and hyperboloids of one or two sheets.
  - **curve and patch functions** – render 3D curve segments and surface patches based on bicubic basis matrices. A basis matrix can be defined and saved. A basis matrix and a precision is selected before rendering the curve or patch.
- **Font and Character Functions** – allow you to select a font type and display text using that font.
- **Transformation Functions** – control the modeling and viewing transformations. These functions are categorized as follows:
  - **transformation control functions** – store and retrieve transformation matrices to/from the transformation stack, get inverse transformation matrices, pre- or post-multiply the current transformation matrix with the specified matrix, and push or pop the transformation stack. Transformation control functions can operate on either the projection transformation stack or the modeling/viewing transformation stack.
  - **modeling functions** – translate, rotate and/or scale the geometric mode. These operations may be done with absolute or incremental values. Modeling transformation functions can be applied to one coordinate axis or to all three simultaneously.
  - **viewing functions** – define view points and view directions. A camera view can be defined in terms of pan, tilt, and swing angles. Look at and look up views can be defined in terms of a view point and a reference point. View points and view directions can be defined in polar coordinates.

  - **projection functions** – define a 2D or 3D projection. The projection can be a 3D perspective pyramid, a 3D perspective window, a 3D orthographic projection, or a 2D orthographic projection.

- **Viewport Functions** – specify a rectangular viewing space that becomes the active area of the screen. Using these functions, the viewport's near and far boundaries are defined and retrieved. These definitions can be stored on the viewport stack along with their corresponding depth ranges. The viewport stack can be manipulated through push and pop operations.
- **Shading and Depth Cueing Functions** – select shading modes and light configurations and enable/disable depth cueing. The possible shading modes are flat, Gouraud and Phong. The light commands define light sources (direct, point, spot), set a light's intensity value, and turn off/on any or all light sources. A surface shading model, such as ambient, diffuse, and specular coefficients can also be specified, as well as the object's opacity and specular exponent. Enabling depth cueing causes points and lines to be rendered with intensities that vary as a function of depth. The z limits and boundary colors of depth cueing can be changed.
- **Color Functions** – define the current rgb and alpha colors. These values are used for current color, point, line, polygon and clear screen commands.
- **Display Functions** – determine what modes of operation are in effect in the frame buffer and control certain aspects of the display. The different modes of operation are double buffering, overlays, anti-aliasing and alpha channel reading. Other display functions swap buffers; clear the rgb or alpha colors in the current viewport; define, draw and erase cursors; write or read a scan line of rgb pixels; and copy image/z buffer memory from on-screen to off-screen memory (and vice-versa).
- **Hidden Surface Removal Functions** – enable and disable the use of zbuffering and backface surface removal algorithms.
- **Video Functions** – control the updating of the video color maps. A complete rgb or alpha color map can be loaded, or one entry of the map can be loaded. The current color maps or a color entry can be retrieved from either the rgb map or the alpha map.
- **Raster Functions** – perform logical operations on pixels, such as adds and multiplies.
- **Picking and Selecting Functions** – can be used to identify objects on the screen based on the object's coordinates. The picking and selecting functions can be enabled or disabled. Identifiers are maintained in a stack with load, push and pop commands. The size of the pick window or selection volume may be set.
- **Input Device Functions** – query for the current value of a locator or the current state of a mouse or keyboard button. These input device events can be queued or sampled. These functions also control the definition and display of cursors associated with a mouse.

- **Compositing Functions** – provide a full set of image compositing functions using the alpha channel of the image.

Control Functions

The PIClib control functions perform operations that initialize and exit PIClib; reset graphical parameters to default values; swap Transformation Pipelines (dual Pipeline configuration); wait for a vertical sync, and wait for a Pixel Node sync.

The control functions are:

- **PICinit()**
- **PICdsp_float()**
- **PICexit()**
- **PICreset()**
- **PICresume()**
- **PICswap_pipe()**
- **PICwait_vsync()**
- **PICwait_psync()**

PICinit()

PICinit() is usually the first graphics function call in every graphics program. The function initializes the viewport to a full screen (1024x1024 or 1280x1024 in high resolution mode, 720x480 in NTSC mode) and the transformation matrix to a 2D, full-screen, orthographic projection. **PICinit()** also locks the Pixel Machine from other users, though it is still accessible to you from any windows you have open.

PICinit() calls **PICreset()** to set all graphical parameters to default values. **PICinit()** also sets up a signal handler for the following signals:

- hangup
- interrupt
- software termination.

When invoked, the signal handler calls the **PICexit()** function and disconnects *all* forked processes, shared memories, and semaphores.

The DEVtools automatic loading facility figures out what is loaded and what additional modules need to be loaded, therefore, the user does not have to remember what modules are already loaded into the Pixel Machine. This makes switching between libraries transparent.

PICinit() returns an integer value of PIC_ERR_OK if the initialization is successful and should be called only once at the beginning of a program and *before* calling any PIClib functions.

Example:

```
#include <piclib.h>
main()
{
     if (PICinit()) exit 1;
     .
     .
     .
     PICexit();
     exit(0);
}
```

PICdsp_float()

The **PICdsp_float()** function enables or disables DSP floating point format and can be used to send DSP floating point data into the Pixel Machine. When floating point format is enabled (mode = PIC_ON), floating point numbers on the host are assumed to be in DSP32 format and no conversion is made before downloading this data to the Pixel Machine. When floating point format is disabled (mode = PIC_OFF), floating point numbers on the host are assumed to be in IEEE format and are converted to DSP32 format after being downloaded. The default mode is PIC_OFF.

PICdsp_float(mode)
int mode;

mode = PIC_ON or PIC_OFF

PICexit()

The **PICexit()** function halts all Transformation and Pixel Nodes and closes the device. If the exit is successful, **PICexit()** returns an integer value of PIC_ERR_OK. This function is always the last graphics function in an application program, and unlocks the Pixel Machine making it accessible to other users.

Example:

```
#include      <piclib.h>
main()
{
     if (PICinit()) exit 1;
     .
     .
     .
     PICexit();
     exit(0);
}
```

PICreset()

The **PICreset()** function resets all possible graphical parameters to their default values as follows:

Function	Default
PICalpha()	PIC_OFF
PICantialias_lines()	PIC_OFF
PICarc_precision()	PIC_ARC_DEFAULT
PICbackface()	PIC_OFF
PICcircle_precision()	PIC_CIRCLE_DEFAULT
PICclockwise()	PIC_OFF
PICcolor_alpha()	0
PICcolor_rgb()	PIC_WHITE
PICcomposite_mode()	PIC_NO_COMPOSITE
PICcurve_precision()	PIC_CURVE_DEFAULT
PICdefine_cursor()	mouse
PICdepth_cue()	PIC_OFF
PICdepth_cue_limits()	PIC_ZMIN_DEFAULT,PIC_WHITE,PIC_ZMAX_DEFAULT,PIC_WHITE
PICdisplay_cursor()	PIC_OFF

PICdisplay_overlay()	PIC_OFF
PICdouble_buffer()	PIC_OFF
PICdsp_float()	PIC_OFF
PICeuclid_mode()	PIC_EUCLID_LINE
PICflip()	PIC_OFF
PIClight_ambient()	PIC_WHITE
PICopen_vector_font()	standard1
PICortho_2d_project()	full screen
PICpatch_precision()	PIC_PATCH_DEFAULT, PIC_PATCH_DEFAULT
PICpercent_texture()	0.7
PICput_alpha_map_entry()	[128 - 255], PIC_WHITE
PICput_depth()	PIC_ZMIN_DEFAULT,PIC_ZMAX_DEFAULT
PICput_picking_region()	8, 8
PICput_rotate_dx()	0.0
PICput_rotate_dy()	0.0
PICput_rotate_dz()	0.0
PICput_viewport()	full screen
PICquadric_precision()	PIC_QUADRIC_DEFAULT, PIC_QUADRIC_DEFAULT
PICselect_curve_basis()	PIC_BEZIER_BASIS
PICselect_patch_basis()	PIC_BEZIER_BASIS,PIC_BEZIER_BASIS
PICshade_mode()	PIC_SHADE_OFF
PICupdate_map()	PIC_OFF (if NTSC mode) PIC_ON (if high resolution mode)
PICzbuffer()	PIC_OFF
PICzbuffer_lines()	PIC_OFF

PICresume()

The **PICresume()** function initializes PIClib *without* resetting any graphical parameters. **PICresume()** functions as **PICinit()** without calling **PICreset()** and is called once at the beginning of a PIClib program.

PICresume() returns PIC_ERR_OK if the initialization succeeded.

PICswap_pipe()

The **PICswap_pipe()** function swaps the two Transformation Pipelines. This function operates only in systems with parallel dual Pipeline configurations. By enabling the user to route commands to different Pipelines, this function helps to optimize program performance by allowing for the simultaneous generation and transformation of various objects.

PICwait_vsync()

The **PICwait_vsync()** function waits for a video vertical sync before executing the next graphics function.

PICwait_psync()

The **PICwait_psync()** function waits for all Pixel Node processors to sync on the same instruction before continuing. This function is used by **PICexit()** before halting the Transformation and Pixel Nodes.

Graphics Primitives – Basic Functions

The **Basic** graphics primitives functions let you render points, lines, arcs, circles, and rectangles. They also allow you to specify the drawing precision used to generate arcs and circles (i.e., you can define the number of points, lines, or filled polygons to be used in rendering an arc or circle); specify the drawing mode (point, line, or polygon); move the current drawing position.

The Basic functions described in this section are as follows:

- **PICeuclid_mode(mode)**
- **PICarc(x,y,r,start,end)**
- **PICcircle_precision(n)**
- **PICrectangle(x0,y0,x1,y1)**
- **PICdraw_2d(x,y)**
- **PICdraw_3d(x,y,z)**
- **PICdraw_4d(x,y,z,w)**
- **PICidraw_2d(ix,iy)**
- **PICidraw_3d(ix,iy,iz)**
- **PICidraw_4d(ix,iy,iz,iw)**
- **PICmove_2d(x,y)**
- **PICmove_3d(x,y,z)**
- **PICmove_4d(x,y,z,w)**
- **PICimove_2d(ix,iy)**
- **PICimove_3d(ix,iy,iz)**
- **PICimove_4d(ix,iy,iz,iw)**
- **PICpoint_2d(x,y)**
- **PICpoint_3d(x,y,z)**
- **PICpoint_4d(x,y,z,w)**
- **PICipoint_2d(ix,iy)**
- **PICipoint_3d(ix,iy,iz)**
- **PICipoint_4d(ix,iy,iz,iw)**

PICeuclid_mode()

The **PICeuclid_mode()** function sets the drawing mode for generating arcs, circles, rectangles, polygons, curves, patches, quadrics, and superquadrics. The default drawing mode is PIC_EUCLID_LINE. Available modes are:

PIC_EUCLID_POINT	primitives are rendered with points
PIC_EUCLID_LINE	primitives are rendered with lines
PIC_EUCLID_POLYGON	primitives are rendered with *filled* polygons
PIC_EUCLID_TEXTURE	primitives are rendered with *textured* polygons

PICeuclid_mode(mode)
int mode;

mode	=	PIC_EUCLID_POINT PIC_EUCLID_LINE PIC_EUCLID_POLYGON PIC_EUCLID_TEXTURE (currently used only for rendering bicubic patches)

PICarc()

The **PICarc()** command draws a circular arc in the xy plane, at z = 0, using the current attributes. The arc is generated according to the mode specified by **PICeuclid_mode()**.

To draw an arc, you must specify the coordinates (*x,y*) of the arc's center; the radius of the arc, *r*; and the starting and ending angles, *start* and *end*. The angles are measured from the positive x axis and are specified in positive floating point degrees. The arc is rendered counterclockwise from the *start* angle to the *end* angle.

The number of points, lines, or polygons used in rendering the arc is set by the **PICarc_precision()** function.

PICarc(x,y,r,start,end)
float x,y,r,start,end;

x,y	=	the x,y coordinates of the arc's center point
r	=	the arc's radius
start	=	the arc's starting angle measured in degrees
end	=	the arc's ending angle measured in degrees

Example:

The following program renders an arc in the positive x,y quadrant. The arc has a center point of 200.0,200.0, a radius of 100, a starting angle of 0.0, and an ending angle of 90.0. The arc is red (specified by the **PICcolor_rgb()** function). By default, the drawing mode is set to PIC_EUCLID_LINE and, therefore, the arc is composed of line segments.

```
        /*render an arc*/

#include      <piclib.h>

main()
{
        if(PICinit()) exit (1);

        /*clear the screen to blue*/

        PICcolor_rgb(0.0,0.0,1.0);
        PICclear_rgb();

        /*select red drawing color*/

        PICcolor_rgb(1.0,0.0,0.0);
        PICarc(200.0,200.0,100.0,0.0,90.0);
        PICexit();
        exit(0);
}
```

PICarc_precision()

The **PICarc_precision()** function is used to set the precision at which the arc will be drawn. The number of points, lines, or polygons used in rendering the arc is specified by the argument, *n*. The larger *n* is, the smoother the arc will be. The default value for *n* is PIC_ARC_DEFAULT.

PICarc_precision(n)
int n;

n = the number of points, lines, or polygons used in rendering the arc

Example:

The following program is the same as the one shown for rendering an arc. This time, however, the arc's precision is set at 100, which results in a smoother arc.

```
        /*render an arc*/

#include     <piclib.h>

main()
{
        if(PICinit()) exit (1);

        /*clear the screen to blue*/

        PICcolor_rgb(0.0,0.0,1.0);
        PICclear_rgb();

        /*select red drawing color*/

        PICcolor_rgb(1.0,0.0,0.0);

        /*set arc precision*/

        PICarc_precision(100);
        PICarc(200.0,200.0,100.0,0.0,90.0);
        PICexit();
        exit(0);
}
```

PICcircle()

The **PICcircle()** function draws a circle in the xy plane at z = 0, using the current attributes. The circle is generated according to the mode specified by **PICeuclid_mode()**.

To draw a circle, you need to specify the circle's center point (*x,y*) and its radius, *r*. The circle's precision is set by the **PICcircle_precision()** function.

PICcircle(x,y,r)
float x,y,r;

x,y = the x,y coordinates of the circle's center point

r = the circle's radius

Example:

The following program renders a red circle with a center point of 200.0,200.0 and a radius of 100.0. Since the circle's precision is not specified, the default setting of PIC_CIRCLE_DEFAULT line segments will be used.

```
    /*render a red circle*/
#include    <piclib.h>
main ()
{
    if (PICinit()) exit (1);
    /*clear the screen to blue*/
    PICcolor_rgb(0.0,0.0,1.0);
    PICclear_rgb();
    /*select red drawing color*/
    PICcolor_rgb(1.0,0.0,0.0);
    PICcircle(200.0,200.0,100.0);
    PICexit();
    exit(0);
}
```

PICcircle_precision()

The **PICcircle_precision()** function is used to set the precision at which a circle will be rendered. The number of points, lines, or polygons used in rendering the circle is specified by the argument, *n*. The larger *n* is, the smoother the circle will be. The default value for *n* is PIC_CIRCLE_DEFAULT

PICcircle_precision(n)
int n;

n = specifies the number of points, lines, or polygons used to render the circle

---

Example:

The following program is the same as the one shown for rendering a circle. This time, however, the precision is set to 100, which results in a smoother circle.

```
        /*render a red circle*/
#include     <piclib.h>
main ()
{
        if (PICinit()) exit (1);
        /*clear the screen to blue*/
        PICcolor_rgb(0.0,0.0,1.0);
        PICclear_rgb();
        /*select red drawing color*/
        PICcolor_rgb(1.0,0.0,0.0);
        /*set circle precision and render circle*/
        PICcircle_precision(100);
        PICcircle(200.0,200.0,100.0);
        PICexit();
        exit(0);
}
```

PICrectangle()

The **PICrectangle()** function renders a rectangle in the xy plane, at z = 0, using the current attributes. The rectangle is generated according to the mode specified by **PICeuclid_mode()**.

To render a rectangle, you must specify its lower left corner *(x0,y0)* and its upper right corner *(x1,y1)*. The sides of the rectangle are parallel with the x and y axes of the coordinate system. In line mode, the current graphics position is *(x0,y0)* after the rectangle is drawn.

PICrectangle(x0,y0,x1,y1)
float x0,y0,x1,y1;

x0,y0 = define the lower left corner of the rectangle

x1,y1 = define the upper right corner of the rectangle

Example:

In the following example, the drawing mode is set to PIC_EUCLID_POLYGON. The lower left and upper right corners of the rectangle are defined as 400.0, 300.0 and 800.0, 500.0, respectively.

```
    /*render a green rectangle*/
#include      <piclib.h>
main ()
{
    if (PICinit()) exit (1);
    /*clear screen to white*/
    PICcolor_rgb(1.0,1.0,1.0);
    PICclear_rgb();
    /*select drawing color*/
    PICcolor_rgb(1.0,0.0,0.0);
    /*select drawing mode*/
    PICeuclid_mode(PIC_EUCLID_POLYGON);
    /*render rectangle*/
    PICrectangle(400.0,300.0,800.0,500.0);
    PICexit();
    exit(0);
}
```

PICdraw()

The **PICdraw** functions draw a line from the current graphics position to the given point using the current attributes. There are six **PICdraw** functions:

- PICdraw_2d(x,y)
- PICidraw_2d(ix,iy)
- PICdraw_3d(x,y,z)
- PICidraw_3d(ix,iy,iz)
- PICdraw_4d(x,y,z,w)
- PICidraw_4d(ix,iy,iz,iw)

PICdraw_4d() uses homogeneous coordinates to draw a 3D line from the current graphics position to the point represented in 3-space as $(x/w, y/w, z/w)$. If w =1, the homogeneous coordinates represent the physical coordinates (x, y, z).

PICdraw_2d(x,y)
float x,y;

x,y = the x and y floating point coordinates of the 2D point to which the line is drawn

PICidraw_2d(ix,iy)
int ix,iy;

ix,iy = the ix and iy integer coordinates of the 2D point, to which the line is drawn

PICdraw_3d(x,y,z)
float x,y,z;

x,y,z = the x,y, and z floating point coordinates of the 3D point to which the line is drawn

PICidraw_3d(ix,iy,iz)
int ix,iy,ix;

ix, iy, iz = the ix, iy, and iz integer coordinates of the 3D point to which the line is drawn

PICdraw_4d(x,y,z,w)
float x,y,z,w;

x,y,z,w = the x, y, z, and w floating point coordinates of the 4D point to which the line is drawn

PICidraw_4d(ix,iy,iz,iw)
int ix,iy,iz,iw;

ix,iy,iz,iw = the ix, iy, iz, and iw floating point coordinates of the 4D point to which the line is drawn

PICmove()

The **PICmove** functions move the current drawing position to a specified one. There are six **PICmove** functions:

- **PICmove_2d(x,y)**
- **PICmove_3d(x,y,z)**
- **PICmove_4d(x,y,z,w)**
- **PICimove_2d(ix,iy)**
- **PICimove_3d(ix,iy,iz)**
- **PICimove_4d(ix,iy,iz,iw)**

PICmove_4d() changes the current graphics position to the 3D point (*x/w,y/w,z/w*). If *w* = 1, the homogeneous coordinates of this point represent the physical coordinates (*x,y,z*).

PICmove_2d(x,y)
float x,y;

x,y = the x and y floating point coordinates of the 2D point

PICimove_2d(ix,iy)
int ix,iy;

ix,iy = the x and y integer coordinates of the 2D point

PICmove_3d(x,y,z)
float x,y,z;

x,y,z = the x,y, and z floating point coordinates of the 3D point

PICimove_3d(ix,iy,iz)
int ix,iy,iz;

ix,iy,iz = the x, y, and z integer coordinates of the 3D point

PICmove_4d(x,y,z,w)
float x,y,z,w;

x,y,z,w = the x, y, z, and w floating point coordinates of the 4D point

PICimove_4d(ix,iy,iz,iw)
int ix,iy,iz,iw;

ix,iy,iz,iw = the x, y, z, and w integer coordinates of the 4D point

---

PICpoint()

The **PICpoint** function(s) draw a point at a specified location, defined by the coordinates, using the current color.

The **PICpoint** functions are:

- **PICpoint_2d(x,y)**
- **PICpoint_3d(x,y,z)**
- **PICpoint_4d(x,y,z,w)**
- **PICipoint_2d(ix,iy)**
- **PICipoint_3d(ix,iy,iz)**
- **PICipoint_4d(ix,iy,iz,iw)**

The **PICpoint_4d()** function draws a point the size of a pixel at location $(x/w,y/w,z/w)$. If $w = 1$, the physical coordinates of this point are the same as the homogeneous coordinates (x,y,z,w). If $w = 0$, the homogeneous point represents a point at infinity and the new graphics position becomes the point $(x/w,y/w,z/w)$.

PICpoint_2d(x,y)
float x,y;

x,y = the x and y floating point coordinates of the 2D point

PICipoint_2d(ix,iy)
int ix,iy;

ix,iy = the x and y integer coordinates of the 2D point

```
PICpoint_3d(x,y,z)
float x,y,z;
```

x,y,z = the x, y, and z floating point coordinates of the 3D point

```
PICipoint_3d(ix,iy,iz)
int ix,iy,iz;
```

ix,iy,iz = the x, y, and z integer coordinates of the 3D point

```
PICpoint_4d(x,y,z,w)
float x,y,z,w;
```

x,y,z,w = the x, y, z, and w floating point coordinates of the 4D point

```
PICipoint_4d(ix,iz,iy,iw)
int ix,iz,iy,iw;
```

ix,iz,iy,iw = the x, y, z, and w integer coordinates of the 4D point

Graphics Primitives – Polygons

The *Polygon* functions let you render 2D, 3D, or 4D polygons by defining a sequence of coordinates and then closing the polygon. These functions also allow normal vectors, rgb colors, and texture map indices to be attached to individual polygon vertices.

The maximum number of points in a polygon is defined in the PIC_MAX_POINTS constant. After all vertices have been specified, the **PICpoly_close()** function must be called to close the polygon by connecting the last polygon vertex to the first polygon vertex.

The functions described in this section are:

- **PICclockwise(mode)**
- **PICpoly_close()**
- **PICpoly_normal(nx,ny,nz)**
- **PICpoly_point_2d(x,y)**
- **PICpoly_point_3d(x,y,z)**
- **PICpoly_point_4d(x,y,z,w)**
- **PICpoly_point_nv(x,y,z,nx,ny,nz)**
- **PICpoly_point_uv(x,y,z,u,v)**
- **PICpoly_point_rgb(x,y,z,r,g,b)**
- **PICpoly_point_nv_uv(x,y,z,nx,ny,nz,u,v)**

PICclockwise()

The **PICclockwise()** function defines how a normal vector of a polygon is computed. The calculation of the normal vector affects backface removal and normal shading. The first three vertices (P_0, P_1, P_2) of a polygon are used to form two vectors. When this function is set to PIC_ON, the normal vector is computed as

$$N = (P_1 - P_2) \times (P_0 - P_2).$$

When this function is set to PIC_OFF, the normal vector is computed as

$$N = (P_0 - P_2) \times (P_1 - P_2)$$

The default mode is counterclockwise.

NOTE: The direction of the vector is defined by the right-hand rule.

```
PICclockwise(mode)
int mode;
mode    =    PIC_ON or PIC_OFF
```

PICpoly_close()

The **PICpoly_close()** function closes a polygon by connecting the last polygon vertex to the first polygon vertex.

NOTE: This function must be used *after* a sequence of **PICpoly_point** functions.

```
PICpoly_close()
```

PICpoly_normal()

The **PICpoly_normal()** function is used to define a surface normal *(nx, ny, nz)* for a polygon. The surface normal should point outward in closed solid objects and is used for backface culling, flip tests and flat shading. **PICpoly_normal()** has to be specified before the corresponding PICpoly_point functions. The surface normal does not need to be normalized.

NOTE: This function must be specified *before* the corresponding **PICpoly_point** commands.

```
PICpoly_normal (nx, ny, nz)
float nx, ny, nz;
```

nx, ny, nz = normal vector

PICpoly_point()

The **PICpoly_point** functions are used in a sequence to render 2D, 3D, or 4D polygons. The sequence of coordinates defined by each call to a **PICpoly_point** function are not connected until the **PICpoly_close()** function is invoked. The polygon is rendered using the current attributes. If shading is disabled, the specified polygon is rendered using the current color.

The **PICpoly_point** functions are:

- **PICpoly_point_2d(x,y)**
- **PICpoly_point_3d(x,y,z)**
- **PICpoly_point_4d(x,y,z,w)**

The **PICpoly_point_3d(x,y,z)** function operates in each shading mode as follows:

- If shading is disabled, then the current color (specified with **PICcolor_rgb**) is used to fill the polygon.
- If flat shading is enabled and a user-specified normal vector (**PICpoly_normal()**) precedes the definition of the polygon points, then that definition is used to compute the shade of the polygon. If a normal vector for the polygon is *not* specified, then a normal vector for the polygon is computed in the Transformation Pipeline. (The points must be in counterclockwise order to obtain an outward-pointing normal vector unless **PICclockwise**(PIC_ON) has been called; this vector is then used to compute the shade of the polygon.)
- If Gouraud or Phong shading is enabled, the normal vector to the polygon is computed in the Transformation Pipeline and copied to each vertex for use in shading.

```
PICpoly_point_2d(x,y)
float x,y;
```

x,y = the x, y coordinates of the 2D polygon point (z = 0.0)

PICpoly_point_3d(x,y,z)
float x,y,z;

x,y,z = the x, y, and z coordinates of the 3D polygon point

PICpoly_point_4d(x,y,z,w)
float x,y,z,w;

x,y,z,w = the x, y, z, and w coordinates of the 4D polygon point

NOTE: It is recommended that polygons be planar and convex. Currently, there is a limit of PIC_MAX_POLY_PNTS points per polygon.

Example:

The following program fragment illustrates the commands used to render 2D and 3D polygons.

```
    /*render a 2D polygon*/
PICpoly_point_2d(400.0,100.0);
PICpoly_point_2d(800.0,100.0);
PICpoly_point_2d(800.0,1000.0);
PICpoly_point_2d(400.0,1000.0);
PICpoly_close();
    /*render a 3D polygon*/
PICpoly_point_3d(10.0,10.0,-100.0);
PICpoly_point_3d(700.0,10.0,-50.0);
PICpoly_point_3d(700.0,700.0,-25.0);
PICpoly_point_3d(10.0,700.0,-30.0);
PICpoly_close();
```

PICpoly_point_nv()

The**PICpoly_point_nv()** function is used in a sequence to draw a 3D polygon with a normal *(nx, ny, nz)* defined at each polygon vertex *(x, y, z)*. The vertex normal should point outward in closed solid objects and is used by the Gouraud shading routines (it is ignored by flat shading routines). A sequence of **PICpoly_point_nv()** function calls must be followed by a **PICpoly_close()** function.

This function operates as follows in each shading mode:

- If shading is disabled, then the current color (specified with **PICcolor_rgb()**) is used to fill the polygon.
- If flat shading is enabled, then the normal vector (nx,ny,nz) specified at each vertex is ignored. If a user-specified normal vector (**PICpoly_ normal()**) precedes the definition of the polygon points, then it is used to compute the shade of the polygon. If a normal vector is *not* specified, then a normal vector is computed in the Transformation Pipeline. (The points must be in counterclockwise order to obtain an outward-pointing normal vector unless **PICclockwise**(PIC_ON) has been called; this vector is used to compute the shade of the polygon.)
- If Gouraud shading is enabled, the normal vector (nx,ny,nz) is used to compute an rgb intensity at each vertex.
- If Phong shading is enabled, the vertex and its normal vector are sent to the pixel nodes for shading.

PICpoly_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 the 3D point
nx,ny,nz	=	normal vector

NOTE: It is recommended that polygons be planar and convex. Currently, there is a limit of PIC_MAX_POLY_PNTS points per polygon. The vertex normals do not need to be normalized.

PICpoly_point_uv()

PICpoly_point_uv() is used in a sequence to render a 3D polygon with a texture index (*u,v*) defined at each polygon vertex (*x,y,z*). The intensity of each pixel is a combination of the texture value and the shading value. The combination of these values can be set with **PICpercent_texture()**.

A sequence of **PICpoly_point_uv()** functions must be followed by a **PICpoly_close()** function.

---

void PICpoly_point_uv(x, y, z, u, v)
float x, y, z, u, v;

x,y,z = the x, y, and z coordinates of the 3D point

u,v = texture index

NOTE

It is recommended that polygons be planar and convex. Currently, there is a limit of PIC_MAX_POLY_PNTS points per polygon.

Polygons rendered with this function are flat shaded.

PICpoly_point_rgb()

The **PICpoly_point_rgb()** function is used in a sequence to draw a 3D polygon with a rgb color (*r, g, b*) defined at each polygon vertex point (*x, y, z*). Each *r*, *g*, and *b* color parameter can range from 0.0 to 1.0. A sequence of **PICpoly_point_rgb()** functions must be followed by a **PICpoly_close()** function.

This function operates as follows in each shading mode:

- If shading is disabled, then the *rgb* color is used to color each vertex. (The vertex colors are interpolated over the polygon.)
- If flat shading is enabled, then the color at each vertex (*r,g,b*) is ignored. If a user-specified normal vector precedes the definition of the polygon points, then it is used to compute the shade of the polygon. If a normal vector is *not* specified, then it is computed in the Transformation Pipeline. (The points must be in counterclockwise order to obtain an outward-pointing normal vector unless **PICclockwise**(PIC_ON) has been called.)

- If Gouraud or Phong shading is enabled, the normal vector to the polygon is computed in the Transformation Pipeline and copied to each vertex for shading.

PICpoly_point_rgb(x,y,z,r,g,b)
float x,y,z,r,g,b;

x,y,z = the x, y, and z coordinates of the 3D vertex

r,g,b = the color at a polygon vertex, where each primary component is a floating point number in the range 0.0 to 1.0 (i.e., a normalized color)

NOTE: It is recommended that polygons be planar and convex. Currently, there is a limit of PIC_MAX_POLY_PNTS points per polygon.

PICpoly_point_nv_uv()

The **PICpoly_point_nv_uv()** function is used in sequence to render 3D polygon points with normal vectors and texture indices. This function operates as follows in each shading mode:

- If shading is disabled, then the current color (specified with **PICcolor_rgb()**) is copied to each vertex, and the normal vector value (nx,ny,nz) is ignored.
- If flat shading is enabled, the user-specified normal vector or Transformation Pipeline computed normal vector is used to compute a shade for the polygon. The normal vector at each vertex is ignored.
- If Gouraud shading is enabled, the normal vector is used to compute an rgb intensity at each vertex. The intensity at each pixel within the polygon is a combination of the texture map and the shading value.
- If Phong shading is enabled, the vertex, normal, and texture map indices are sent to the pixel nodes for shading.
- The intensity value at each pixel is computed according to the following equation:

$$Intensity_{pixel} = texture\_percent * texture\_color + (1.0 - texture\_percent) * surface\_intensity$$

The value *texture_percent* can be set with the **PICpercent_texture()** function.

NOTE: When using perspective projection, objects composed of this type of polygon point should be tessellated into many small polygons to ensure minimal perspective distortion.

PICpoly_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 the 3D point
nx,ny,nz	=	normal vector
u,v	=	texture map indices

The *u,v* values are not restricted to the range [0.0,1.0]. Assigning values greater than 1.0 will repeat the texture map over the surface, while assigning values less than 1.0 will allow for the extraction of a portion of the texture map.

NOTE: It is recommended that polygons be planar and convex. Currently, there is a limit of PIC_MAX_POLY_PNTS points per polygon. The vertex normals do not need to be normalized.

Graphics Primitives – Quadrics and Superquadrics

The **Quadrics and Superquadrics** functions render cylinders, ellipsoids, toroids, and hyperboloids of one or two sheets.

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

The functions discussed in this section are:

- **PICquadric_precision(nu,nv)**
- **PICsphere()**
- **PICsuperq_ellipsoid(x,y,z,exp1,exp2)**
- **PICsuperq_toroid(x,y,z,r,exp1,exp2)**
- **PICsuperq_hyper1(x,y,z,exp1,exp2)**
- **PICsuperq_hyper2(x,y,z,exp1,exp2)**

PICquadric_precision()

The **PICquadric_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 is less than zero, the function returns a value of PIC_ERR_ARG. The default is 16x16.

PICquadric_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

---

PICsphere()

The **PICsphere()** function renders a sphere using the current color and drawing mode (i.e., point, line, or polygon). The sphere is centered at the *current* graphics position and has a unit radius. Its precision is set by the **PICquadric_precision()** function.

If polygon mode is on, the sphere is shaded according to the current shading mode.

PICsphere()

PICsuperq_ellipsoid()

The **PICsuperq_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 follows:

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

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

If polygon mode is on, the ellipsoid is shaded according to the current shading mode.

PICsuperq_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

NOTE: Make sure all arguments specified for this function are greater than or equal to zero.

Example:

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

```
#include    <piclib.h>
main()
{
     .
     .
     /*render a sphere*/
     PICsuperq_ellipsoid(100.0,100.0,100.0,1.0,1.0);
     /*render an ellipsoid that's stretched in the y direction*/
     PICsuperq_ellipsoid(100.0,200.0,100.0,1.0,1.0);
     /*render a cube*/
     PICsuperq_ellipsoid(100.0,100.0,100.0,0.01,0.01);
     /*render a cylinder*/
     PICsuperq_ellipsoid(100.0,100.0,100.0,0.0,1.0);
     .
     .
     PICexit();
     exit(0);
}
```

PICsuperq_toroid()

The **PICsuperq_toroid()** function renders a superquadric toroid using the current attributes. The toroid is represented mathematically as follows:

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

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:

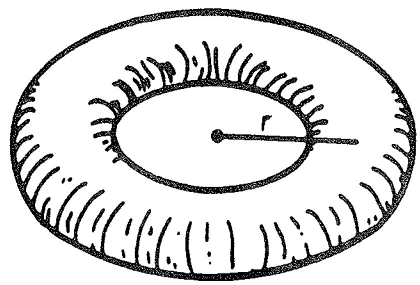
exp < 1	Square-shaped toroids
exp ≃ 1	Round toroids
exp = 2	Flat-beveled toroids
exp > 2	Pinched toroids

If polygon mode is on, the toroid is shaded according to the current shading mode.

PICsuperq_toroid(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

Figure 3-1: A Superquadric Toroid



PICsuperq_hyper1()

The PICsuperq_hyper1() function renders a superquadric hyperboloid of one sheet using the current attributes. The hyperboloid is represented mathematically as follows:

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

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

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

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

If polygon mode is on, the hyperboloid is shaded according to the current shading mode.

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

PICsuperq_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 η = 45°
exp1	=	the squareness parameter in the longitudinal direction
exp2	=	the squareness parameter in the latitudinal direction

PICsuperq_hyper2()

The **PICsuperq_hyper2()** function renders a superquadric hyperboloid of two sheets using the current attributes. The hyperboloid is represented mathematically as follows:

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

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

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

Values for ω are in the range: -pi/2 < ω < pi/2 (piece 1), pi/2 < ω < 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

```
PICsuperq_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 $\eta = 45°$

exp1 = the squareness parameters in the longitudinal direction

exp2 = the squareness parameters in the latitudinal direction

Graphics Primitives – Curve Functions

The **Curve** functions generate parametric curves which can be displayed as a set of points or connected line segments. A **parametric curve** is a set of points obtained by interpolating or approximating a set of control points. The coordinates of the points that define a parametric curve are of the form

$$x = x(u) \quad y = y(u) \quad z = z(u)$$

where u is a parametric variable with an interval of $u \in [0,1]$.

In PIClib, curves are rendered by first specifying a basis matrix and then defining a set of four 3D control points that determine the shape of the curve. The basis matrix determines how the control points will be used to render the curve. Complex curves are rendered by connecting several curve segments to form one curve. However, care must be taken at curve boundaries to ensure continuity.

For more information on ensuring the continuity of a curve, refer to *Mathematical Elements for Computer Graphics* by David F. Rogers and J. Alan Adams (1976, McGraw-Hill, Inc.) or *Geometric Modeling* by Michael E. Mortenson (1985, John Wiley & Sons, Inc.).

The Curve functions described in this section are:

- **PICcurve_geometry_3d()**
- **PICcurve_precision()**
- **PICput_basis()**
- **PICselect_curve_basis()**

Generating Curves

PIClib offers basis matrices for four predefined classes of curves; **Bezier Curves, Hermite Curves, Four-Point Curves** and **B-Spline Curves.** Each curve is cubic (third order polynomial) and generated using the method of forward differences. More complicated curves can be constructed from several smaller curves. Each of the predefined classes of curves is described below.

To define basis matrices for other classes of curves, use the **PICput_basis()** function discussed later in this section.

Bezier Curves

A **Bezier** curve defines the position of the curve's end points and uses two other points (not on the curve) to define indirectly the tangents at the curve's end points. Bezier curves are defined with a set of four *control points* (p_0,p_1,p_2, and p_3) representing the vertices of a polygon. Each point (p) consists of the components (x,y,z). The tangent at p_0 is $p_1 - p_0$ and the tangent at p_3 is $p_2 - p_3$. The curve always passes through p_0 and p_3.

The control points can be easily manipulated to change the shape of the curve as desired. (Any local changes are strongly propagated throughout the entire curve.) For example, by specifying the first and last control points to the same position, a closed curve is generated. The control points define a convex polygon called a *convex hull*, which bounds the Bezier curve and ensures that it follows the specified control points. The matrix for this type of curve is:

$$\begin{bmatrix} x_0 & x_1 & x_2 & x_3 \\ y_0 & y_1 & y_2 & y_3 \\ z_0 & z_1 & z_2 & z_3 \end{bmatrix}$$

Hermite Curves

A **Hermite** curve is a cubic curve. The left half of the input curve matrix is filled with the end points of the curve p_0 and p_1; The right half is filled with tangent vectors at the end points of the curve p_0^u and p_1^u. A Hermite curve always passes through the end points (interpolates) and approximates the two inner points. The matrix is as follows:

$$\begin{bmatrix} x_0 & x_1 & x_0^u & x_1^u \\ y_0 & y_1 & y_0^u & y_1^u \\ z_0 & z_1 & z_0^u & z_1^u \end{bmatrix}$$

B-spline Curves

A **B-spline** curve is a class of spline curves that approximates the end points, allowing both the first and second derivatives to be continuous at the segment's end points. This type of curve uses a set of blending functions to allow localized changes to be made easily by manipulating only a few neighboring control points. No part of the curve passes through the control points.

Local changes are propagated only in the area of change. For example, if you change the position of the first control point, the shape of the curve changes near the first point without significantly affecting the rest of the curve.

Four-point Curves

A **Four-point** curve is an interpolating curve that passes through four distinct points in space. The control points (p_0,p_1,p_2, and p_3) are assigned the parametric u values 0, 1/3, 2/3, and 1, respectively. This type of curve is cubic (third order polynomial).

PICcurve_geometry_3d()

The **PICcurve_geometry_3d()** function renders a 3D curve using the current curve precision, color, and drawing mode. The curve is rendered using the current color.

```
PICcurve_geometry_3d(geom)
float geom[3][4];
```

geom = a set of four 3D control points that determine the shape of the curve

---

PICcurve_precision()

The **PICcurve_precision()** function specifies the number of points, lines, or polygons used in rendering the curve. The precision is expressed as a positive integer between 4 and infinity. The higher the precision specified, the smoother the curve that is rendered.

```
PICcurve_precision(n)
int n;
```

n = the number of points or lines used to render the curve

PICput_basis()

The **PICput_basis()** function defines a 4x4 basis matrix and an associated index number, which can subsequently be used in rendering curves. The index numbers are defined by the following constants:

```
PIC_USER_BASIS_0
PIC_USER_BASIS_1
      .
      .
      .
PIC_USER_BASIS_7
```

At initialization, the first four basis matrices contain the matrix definitions for **Bezier curves, Hermite curves, Four-point curves** and **B-spline curves** respectively. Unless you wish to overwrite these matrices, the *index* argument passed to **PICput_basis()** should range from PIC_USER_BASIS_4 to PIC_MAX_BASIS.

If *index* is less than zero or greater than or equal to PIC_MAX_BASIS, this function returns a value of PIC_ERR_ARG.

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

PICput_basis(basis,index)
PICmatrix basis;
int index;

basis = an matrix of 16 floating point numbers

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

PICselect_curve_basis()

The **PICselect_curve_basis()** function selects the basis matrix that is used to render curves. (The matrix and its index are defined with the **PICput_basis()** function.) Make sure the matrix and its index are defined *before* using the **PICselect_basis()** function. If *index* is less than zero or greater than or equal to PIC_MAX_BASIS, this function returns a value of PIC_ERR_ARG.

PICselect_curve_basis(index)
int index;

index = the index to the basis matrix

Example:

The following program defines the x, y, and z coordinates of the control polygon for a bicubic curve, sets the precision in u and v direction, selects basis PIC_BEZIER_BASIS; draws the control polygon; and, finally, generates a Bezier curve.

```
/* define x,y and z coordinates of the control polygon for a bicubic curve

             defined as :

             X0 X1 X2 X3
             Y0 Y1 Y2 Y3
             Z0 Z1 Z2 Z3
*/

PICmatrix   G = {
                   100.0, 100.0, 0.0, 0.0,
                   0.0, 0.0, 100.0, 100.0,
                   0.0, 100.0, 100.0, 0.0,
          };

main(argc,argv)
int       argc;
char      **argv;
{

          int       npixl,nline;
          int       precu,precv;
          int       basis;
          int       i,iter;

          if (PICinit()) exit(-1);

          /*  animate */

          PICdouble_buffer(PIC_ON);

          PICget_screen_size( &npixl, &nline );
          PICput_viewport( 0, npixl-1, 0, nline-1 );

          PICcopy_front_to_back();

          /*  make drop shadow */

          PICput_viewport( 290+20, 990+20, 80+20, 800+20 );
          PICpixel_add( -0.2, -0.2, -0.2, -0.2 );
          PICswap_buffer();

          PICput_viewport( 290+20, 990+20, 80+20, 800+20 );
          PICpixel_add( -0.2, -0.2, -0.2, -0.2 );
          PICswap_buffer();

          /*  create viewport and projection  */

          PICput_viewport( 290, 990, 80, 800 );
          PICput_depth( 0.0, 32767.0 );
```

(continued on next page)

```
/* set viewing and projection parameters */

PICpersp_project(45.0 , 1.25, 1.0, 2048.0);
PIClookup_view(150.0, 150.0, 150.0, 0.0, 0.0, 0.0, 0.0);

/*  set precision in u direction, v direction and
    select basis index 0 (bezier basis) */

precu = 25;
basis = 0;

PICcurve_precision(precu);
PICselect_curve_basis(basis);
PICeuclid_mode(PIC_EUCLID_LINE);

/* rotate curve around the z axis 2.5 degrees at each frame */

iter = 550;

    do {
                   PICcolor_rgb( 1.0, 1.0, 1.0 );
                   PICclear_rgbz();

/* rotate the patch */

                   PICrotate_z(2.5);

/* draw the control polygon */

                   PICcolor_rgb( 0.0 , 0.0 , 1.0);
                   PICmove_3d(G[0][0],G[1][0],G[2][0]);

                   PICdraw_3d(G[0][1],G[1][1],G[2][1]);
                   PICdraw_3d(G[0][2],G[1][2],G[2][2]);
                   PICdraw_3d(G[0][3],G[1][3],G[2][3]);

                   PICcolor_rgb( 0.0 , 1.0 , 0.0);

/* draw some axes */

                   PICmove_3d(0.0,0.0,0.0);
                   PICdraw_3d(0.0,0.0,100.0);

                   PICmove_3d(0.0,0.0,0.0);
                   PICdraw_3d(0.0,100.0,0.0);

                   PICmove_3d(0.0,0.0,0.0);
                   PICdraw_3d(100.0,0.0,0.0);

                   PICcolor_rgb( 1.0 , 0.0 , 0.0);
```

(continued on next page)

```
        /* generate a Bezier curve */

                        PICcurve_geometry_3d(G);
                        PICswap_buffer();

            }   while (iter--);

        PICexit();
        exit();
}
```

Graphics Primitives – Patch Functions

A *patch* is a bounded collection of points and is the simplest mathematical element used to model a surface. The coordinates of the points that define the patch have two parameters and are of the form:

$$x = x(u,w) \quad y = y(u,w) \quad z = z(u,w)$$

where u and w are parametric variables with an interval of $u,w \in [0,1]$.

In PIClib 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.

The patch functions discussed in this section are:

- **PICpatch_geometry3d(xgeom,ygeom,zgeom)**
- **PICpatch_precision(nu,nv)**
- **PICput_basis(basis,index)**
- **PICselect_patch_basis(uindex,vindex)**

Generating Patches

PIClib offers basis matrices for four predefined classes of patches; **Bezier Patches, Hermite Patches, B-Spline Patches** and **Sixteen-Point Form Patches.** Each of the predefined classes of patches is described below.

To define basis matrices for other classes of patches, use the **PICput_basis()** function discussed later in this section. Patches can be generated as a:

- cloud of points
- line mesh
- shaded polygon mesh

- texture mapped shaded patch

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 as follows:

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

Hermite Patch

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

$$\begin{bmatrix} P_{00} & P_{01} & P^w_{00} & P^w_{01} \\ P_{10} & P_{11} & P^w_{10} & P^w_{11} \\ P^u_{00} & P^u_{01} & P^{uw}_{00} & P^{uw}_{01} \\ P^u_{10} & P^u_{11} & P^{uw}_{10} & P^{uw}_{11} \end{bmatrix}$$

NOTE $P_{00}{}^u$ is the derivative of the point with respect to the parametric variable u; P^w is the derivative of the point with respect to w; P^{uw} is the derivative of the point with respect to u and w.

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 w 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

A **B-spline** patch is defined by a characteristic polyhedron. The shape of the entire surface approximates the polyhedron.

Example:

Generate a viewport with dropped shadows. A red Bezier bicubic patch rotates around the z axis.

```
#include    <piclib.h>

        /* define x,y and z coordinates of the control mesh for a bicubic patch */

PICmatrix   GX = {
                        0.0, 0.0, 0.0, 0.0,
                        25.0, 25.0, 25.0, 25.0,
                        50.0, 50.0, 50.0, 50.0,
                        75.0, 75.0, 75.0, 75.0
                };

PICmatrix   GY = {
                        0.0, 25.0, 50.0, 75.0,
                        0.0, 25.0, 50.0, 75.0,
                        0.0, 25.0, 50.0, 75.0,
                        0.0, 25.0, 50.0, 75.0
                };

PICmatrix   GZ = {
                        5.0, 45.0, 25.0, -30.0,
                        5.0, 55.0, 65.0, -20.0,
                        5.0, 65.0, 25.0, -10.0,
                        5.0, 35.0, 15.0, 0.0
                };


main(argc,argv)
int     argc;
char    **argv;
{

        int             npixl,nline;
        int             precu,precv;
        int             basis;
        int             i,iter;
        void            draw_mesh();


        if (PICinit()) exit(-1);

        /* animate */

        PICdouble_buffer( PIC_ON );
        PICzbuffer( PIC_ON );
```

(continued on next page)

```
        PICget_screen_size( &npixl, &nline );
        PICput_viewport( 0, npixl-1, 0, nline-1 );

        PICcopy_front_to_back();

        /* make drop shadow */

        PICput_viewport( 290+20, 990+20, 80+20, 800+20 );
        PICpixel_add( -0.2, -0.2, -0.2, -0.2 );
        PICswap_buffer();

        PICput_viewport( 290+20, 990+20, 80+20, 800+20 );
        PICpixel_add( -0.2, -0.2, -0.2, -0.2 );
        PICswap_buffer();

        /* create viewport and projection */

        PICput_viewport( 290, 990, 80, 800 );
        PICput_depth( 0.0, 32767.0 );

        /* set viewing and projection parameters */

        PICpersp_project(45.0 , 1.25, 1.0, 2048.0);
        PIClookup_view(150.0, 150.0, 150.0, 0.0, 0.0, 0.0, 0.0);

        /* set precision in u direction, v direction and select basis index 0 (bezier basis) */

        precu = 25;
        precv = 20;
        basis = 0;

        PICpatch_precision(precu,precv);
        PICselect_patch_basis(basis,basis);
        PICeuclid_mode(PIC_EUCLID_LINE);

/* rotate patch around the z axis 2.5 degrees at each frame */

        iter = 55;

            do {

                        PICcolor_rgb( 1.0, 1.0, 1.0 );
                        PICclear_rgbz();

                        PICrotate_z(2.5);
                        PICcolor_rgb( 1.0 , 0.0 , 0.0);
                        PICpatch_geometry_3d(GX,GY,GZ);

                        draw_mesh();
```

(continued on next page)

```
                    PICswap_buffer();

            }   while (iter--);

        PICexit();
        exit();
}

void    draw_mesh()
{
        int     i,j;

        PICcolor_rgb(1.0,0.0,1.0);

        for(i = 0; i < 4; i++)
        {
                PICmove_3d(GX[i][0],GY[i][0],GZ[i][0]);

                for(j = 1; j < 4; j++)
                        PICdraw_3d(GX[i][j],GY[i][j],GZ[i][j]);
        }

        for(j = 0; j < 4; j++)
        {
                PICmove_3d(GX[0][j],GY[0][j],GZ[0][j]);

                for(i = 1; i < 4; i++)
                        PICdraw_3d(GX[i][j],GY[i][j],GZ[i][j]);
        }
}
```

PICpatch_geometry_3d()

The **PICpatch_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 shape of a 3D patch is defined by a set of user-specified 3D control points. The surface patch is rendered using the current color and drawing mode. If polygon mode is on, the patch is shaded according to the current shading mode.

```
PICpatch_geometry_3d(xgeom,ygeom,zgeom)
PICmatrix xgeom,ygeom,zgeom
```

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

PICpatch_precision()

The PICpatch_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. If the arguments *nu,nv* are less than zero, the function returns a value of PIC_ERR_ARG.

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

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

PICput_basis()

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

PIC_USER_BASIS_0
PIC_USER_BASIS_1
.
.
.
PIC_USER_BASIS_7

At initialization, the first four basis matrices contain the matrix definitions for **Bezier patches**, **Hermite patches**, **B-spline patches** and **Four-point patches** respectively. Unless you wish to overwrite these matrices, the *index* argument passed to **PICput_basis()** should range from 4 to PIC_MAX_BASIS.

If *index* is less than zero or greater than or equal to PIC_MAX_BASIS, this function returns a value of PIC_ERR_ARG.

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

PICput_basis(basis,index)
PICmatrix basis;
int index;

basis	=	an matrix of 16 floating point numbers
index	=	the index number associated with the *basis* matrix

PICselect_patch_basis()

The **PICselect_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 indexes must have been previously defined by **PICput_basis()**. If *uindex* or *vindex* are less than zero or ≥ PIC_MAX_BASIS, **PICselect_patch_basis()** returns PIC_ERR_ARG.

PICselect_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

Graphics Primitives – Template Functions

The **Template** functions create precalculated atoms that can be quickly rendered on the screen. These atoms are not affected by geometric distortions, such as perspective projection or aspect ratio changes. sphere and user defined templates can be created and stamped on the screen. They can be Zbuffered, vary in size up to 256 X 256, and the user can define pixel alpha opacity percentages for transparency. Because templates are a raster primitive, there is little I/O overhead and transformation pipe computation involved, therefore, you can stamp many templates at high speed.

The **Template** functions are:

- **PICatom(x,y,z,r)**
- **PICatom_light(light)**
- **PICatom_surface(surface)**
- **PICget_template(templ_ptr,index)**
- **PICmake_template(index,size,ix,iy,data,zdata,npixl)**
- **PICmake_sphere_template(index,ix,iy,z,radius)**
- **PICstamp_template(index,xyz,npoint,mode)**

PICatom()

The **PICatom()** function draws a 3D spherical atom centered at the point *(x,y,z)* and with radius *r*. The atom will be Phong shaded and will be lit according to the light source specified by **PICatom_light()** and surface model specified by **PICatom_surface()**. **PICatom()** is a template function, and, thus, will quickly render a spherical atom that is not affected by geometrical distortions.

Atoms are a special primitive and are not handled in the same manner as the standard primitives. Atoms do not work correctly with perspective projection; orthographic projection should be used. To render atoms correctly it is recommended that the viewport volume be a cube, the viewport a square, and the depth range set with *near* = 0.0 and *far* = the width of the viewport.

NOTE: Atoms are not clipped in the pipeline. To clip atoms, set the z depth outside of the current viewport to a negative value, then clear the viewport to a positive z value. Also note that modeling transformations are applied to the center of the atom but not to the radius. The atom's radius should be specified with respect to World Coordinates. Transparent atoms have not been implemented.

```
PICatom(x,y,z,r)
float x,y,z,r;
```

x,y,z = the coordinates of the atom's center point

r = the atom's radius

PICatom_light()

PICatom_light() specifies a light source for a spherical atom. It has one argument, a pointer to PIClight_source.

---

void PICatom_light(light)
PIClight_source *light;

light = pointer to PIClight_source

PICatom_surface()

PICatom_surface() takes one argument, a pointer to PICsurface_model. An atom's shading is computed using the following equation:

```
I = Ka * Li + Kd * Li * (V_normal.V_light) + Ks * Li *
(V_eye.V_reflection) ** S_exponent)
```

where:

Li = intensity of light
Kd = diffuse coefficient of surface
Ks = specular coefficient of surface
Ka = ambient coefficient of surface
S_exponent = specular exponent of surface
V_normal = surface normal vector
V_light = vector to light source
V_eye = vector to the eye
V_reflection = reflection vector

NOTE

S_exponent is currently set to 10.

---

void PICatom_surface(surface)
PICsurface_model *surface;

surface = pointer to PICsurface_model

PICget_template()

PICget_template() takes a pointer to a PICtemplate structure and stores the template associated with *index* in the location pointed to by *templ_ptr*. The PICtemplate structure is defined as follows:

```
typedef struct
        {
        short   type;       /* sphere (type=0) or user defined (type=1) */
        int     ix;         /* template position in vram in x */
        int     iy;         /* template position in vram in y */
        int     size;       /* size of template in pixels in x and y dimensions */
        float radius;       /* size of radius (spheres only)                     */
        }PICtemplate;
```

PICget_template() returns PIC_TRUE on success and PIC_FALSE on failure.

---

int *PICget_template(templ_ptr,index)
int index
PICtemplate *templ_ptr

templ_ptr = pointer to PICtemplate structure

index = template to be stored

PICmake_template()

PICmake_template() takes *index* and *size*, broadcasts a user defined template consisting of *npixl* bitmapped RGBA values and their associated z depth information into off screen memory and associates that template with *index*. *index* is an integer that ranges from 0 to PIC_MAX_STAMP. The template's position is (*ix, iy*). The template has a height and width of variable size; *size* is the same in the x and y dimensions. The maximum size of a template is PIC_MAX_UDTEMPLATE.

The alpha value for each pixel in the template determines the opacity of the pixel. Alpha values range from 0 to 255, where 0 is completely transparent and 255 is completely opaque.

```
void PICmake_template(index,size,ix,iy,data,zdata,npixl)
int  index,ix,iy,size,npixl
PICrgba_pixel  *data
float  *zdata
```

index = template to be broadcast

size = height and width of template

ix, iy = position of the template

data = bitmapped RGBA values

npixl = number of RGBA values

PICmake_sphere_template()

PICmake_sphere_template() takes an index, radius and z-depth, renders a sphere template into offscreen memory and associates that template with *index*. *index* is an integer that ranges from 0 to PIC_MAX_STAMP. The template's position is (*ix, iy*). The maximum size of a sphere template is PIC_MAX_STEMPLATE, and the maximum radius is PIC_MAX_STEMPLATE/2 . In order for sphere templates to be stamped correctly, zbuffering must be enabled.

---

void PICmake_sphere_template(index,ix,iy,z,radius)
int index,ix,iy,radius
float z

index	=	sphere template to be rendered
ix,iy	=	template's position
z	=	z-depth of the template
radius	=	radius of the template

PICstamp_template()

PICstamp_template() takes a template index, a point array, a point count, and a mode and stamps the template associated with *index* at the locations specified by *xyz*. The variable *xyz* consists of *npoint* (x,y,z) locations. No more than PIC_MAX_STAMP points can be stamped in one call to **PICstamp_template()**. If the template is a user defined template and *mode* = PIC_ON, then alpha opacity percentages are generated for each pixel. If *mode* = PIC_OFF, then the alpha values are ignored. Alpha values are ignored for sphere templates. Templates can be zbuffered or non-zbuffered. The maximum size of a non-zbuffered template is PIC_MAX_UDTEMPLATE X PIC_MAX_UDTEMPLATE. The maximum size of a zbuffered template is PIC_MAX_UDTEMPLATE/2 X PIC_MAX_UDTEMPLATE/2 .

---

void PICstamp_template(index,xyz,npoint,mode)
int index,npoint,mode
float *xyz

index	=	template to be stamped
xyz	=	locations at which template is stamped
npoint	=	number of x, y, z locations
mode	=	PIC_ON – alpha opacity percentages are generated
	=	PIC_OFF – alpha values are ignored

Fonts and Characters

PIClib supports two types of fonts, **raster** fonts and **vector** fonts. **Vector** fonts supported by PIClib are the standard hershey vector fonts and reside in **$HYPER_PATH/fonts**. Vector fonts are composed of a series of connected 3D lines, and are affected by the current line mode and the current projection and modeling matrices. Vector fonts are clipped by the viewing pyramid.

Raster fonts are a series of bit patterns displayed on the screen. Raster fonts are not affected by the projection or the modeling matrices, however they are clipped by the viewport. In regular rgb mode the current color is used to display the raster fonts. When the alpha channel is enabled, raster fonts are drawn in the alpha channel using the current alpha color. A set of raster font files reside in /usr/lib/fonts/fixedwidthfonts. You may also create your own raster fonts with the fontedit program supplied by Sun.

The **Fonts and Characters** functions allow you to select a font type and write text using the font you selected. This section discusses the following functions:

- **PICopen_raster_font(font)**
- **PICput_raster_font(font)**
- **PICraster_text(ix,iy,string)**
- **PICraster_font_text(font,ix,iy,string)**
- **PICopen_vector_font(font)**
- **PICput_vector_font(font)**
- **PICvector_text(string)**
- **PICvector_font_text(font,string)**

PICopen_raster_font()

The **PICopen_raster_font()** selects (opens) the specified raster font and returns a pointer to the raster font structure, PICraster_font. If the font cannot be opened, a null pointer is returned.

The following raster fonts are currently available:

- apl.r.10
- cmr.b.8, cmr.b.14, cmr.r.8, cmr.r.14
- cour.b.10, cour.b.12, cour.b.14, cour.b.16, cour.b.18, cour.b.24, cour.r.10, cour.r.12 cour.r.14, cour.r.16, cour.r.18, cour.r.24
- gacha.b.8, gacha.r.7, gacha.r.8
- gallant.r.10, gallant.r.19
- sail.r.6
- screen.b.12, screen.b.11, screen.r.7, screen.r.12, screen.r.13, screen.r.14

- serif.r.10, serif.r.11, serif.r.12, serif.r.14, serif.r.16

The fonts listed above are the standard fonts available with Sun's system software. You can generate new fonts by using the *fontedit* routine supplied by *suntools.*

PICopen_raster_font(font)
char *font;

Example:

In the following example, the **serif.r.10** font is selected:

```
font1 = PICopen_raster_font("/usr/lib/fonts/fixedwidthfonts/serif.r.10");
```

PICput_raster_font()

The **PICput_raster_font()** function sets the current raster font to a previously opened raster font *font.* The current raster font is used by the **PICraster_text()** function.

PICput_raster_font(font)
PICraster_font *font;

PICraster_text()

The **PICraster_text()** function writes a text string, *string*, using the current raster font. The upper left corner of the text is located at point *(ix,iy)* with respect to the current viewport

Raster text is clipped by the viewport, but is not affected by the projection or modeling transformations. If alpha channel rendering is enabled, the raster text will be displayed in the alpha channel using the current alpha color. **PICraster_text()** returns the x position of the end of the string.

PICraster_text(ix,iy,string)
int ix, iy;
char *string;

Example:

In the following example, the string "**hello**" is written at location 100,100.

```
PICraster_text(100, 100, "hello");
```

PICraster_font_text()

The **PICraster_font_text()** function writes a text string, *string*, using the specified raster font *font*. The raster font must have been previously opened by **PICopen_raster_font()**. The upper left corner of the text is located at point (*ix,iy*) with respect to the current viewport

Raster text is clipped by the viewport, but is not affected by the projection or modeling transformations. If alpha channel rendering is enabled, the raster text will be displayed in the alpha channel using the current alpha color. **PICraster_font_text()** returns the x position of the end of the string.

```
PICraster_font_text(font,ix,iy,string)
PICraster_font *font;
int ix, iy;
char *string;
```

Examples:

In the following example, the specified string "**hello**" will be output using **serif.r.10** at location 100,100. Note that in a previous example, the **serif.r.10** font was opened and stored in the PICraster_font structure named *font1*.

```
PICraster_font_text(font1, 100, 100, "hello");
```

The following program illustrates the use of raster fonts for displaying text.

```
main()
{
    PICraster_font      *font;
    int       x,y;

    PICinit();

    /* open font */

    font = PICopen_raster_font("/usr/lib/fonts/fixedwidthfonts/cour.b.16");

    if (font == NULL) {
       printf("Could not open font file \n");
       exit(0);
    }

    y = 10;
    x = 50;

    PICcolor_rgb(1.0,1.0,0.5);

    /* print the text */

    PICraster_font_text(font,x,y,"Message #1");

    /* set the current font and print the text */

    PICput_raster_font(font);
    PICraster_text(x,y+50,"Message #2");

    PICexit();
}
```

PICopen_vector_font()

The **PICopen_vector_font()** selects (opens) the specified vector font and returns a pointer to the vector font structure, PICvector_font. If the font cannot be opened, a null pointer is returned.

The following vector fonts are currently available:

- greek1
- italic1, italic2
- lombardic
- roman1, roman2
- script1, script2
- special1
- standard1
- texture

The font types that end with a "2" indicate **boldface** versions of those fonts.

PICopen_vector_font(font)
char *font;

Example:

In the following example, the **italic1** font is selected:

```
font1 = PICopen_vector_font("italic1");
```

PICput_vector_font()

The **PICput_vector_font()** function sets the current vector font to a previously opened vector font *font*. The current vector font is used by the **PICvector_text()** function.

PICput_vector_font(font)
PICvector_font *font;

PICvector_text()

The **PICvector_text()** function writes a text string, *string*, using the current vector font. Because vector fonts are a series of 3D lines, the text being displayed is transformed by the current transformation matrix and affected by the current color and line mode. The text starts at location (0.0,0.0,0.0) and the x position of the end of the string is returned.

PICvector_text(string)
char *string;

Example:

In the following example, the string "**hello**" is written.

```
PICvector_text("hello");
```

PICvector_font_text()

The **PICvector_text()** function writes a text string, *string*, using the specified vector font *font*. The vector font must have been previously opened by **PICopen_vector_font()**. Because vector fonts are a series of 3D lines, the text being displayed is transformed by the current transformation matrix and affected by the current color and line mode. The text starts at location (0.0,0.0,0.0) and the x position of the end of the string is returned.

PICvector_font_text(font,string)
PICvector_font *font;
char *string;

Example:

In the following example, the specified string "**hello**" will be output using **italic1**. Note that in a previous example, the **italic1** font was opened and stored in the PICvector_font structure named *font1*.

```
PICvector_font_text(font1, "hello");
```

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 may 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 raster operations, cursors and viewports, 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 are two matrix stacks and two current matrices, which can be operated on separately. One stack contains the Modeling and Viewing transformations, the other holds the Projection transformations. Objects are transformed by the product of the two current matrices: Modeling and Viewing (MV) matrix and Projection (P) 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

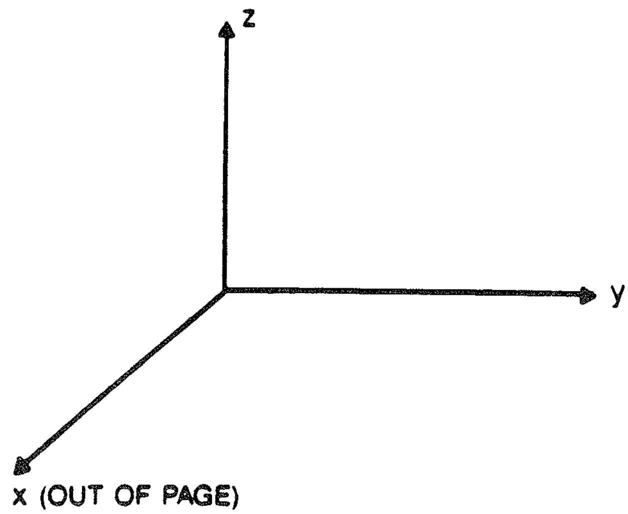


Figure 3-3: Eye Coordinate System

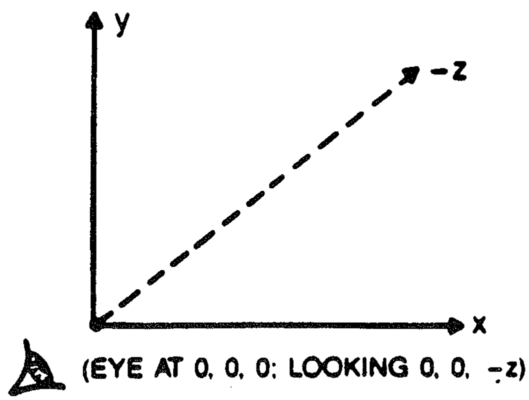


Figure 3-4: Screen Coordinate System

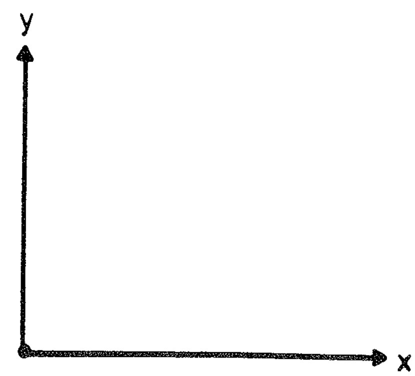
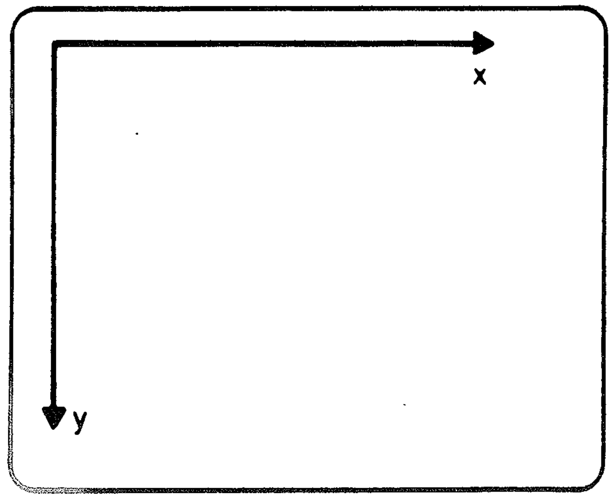


Figure 3-5: Pixel Coordinate System



Transformations – 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 **PICpush_transform()** function and restored with the **PICpop_transform()** function.

This section describes the following modeling transformation functions:

Rotation Functions

- PICrotate_x(x)
- PICrotate_y(y)
- PICrotate_z(z)
- PICrotate_vector(x,y,z,nx,ny,nz,angle)
- PICput_rotate_dx(dx)
- PICput_rotate_dy(dy)
- PICput_rotate_dz(dz)
- PICrotate_dx()
- PICrotate_dy()
- PICrotate_dz()

Translation Functions

- PICtranslate_x(x)
- PICtranslate_y(y)
- PICtranslate_z(z)
- PICtranslate(x,y,z)
- PICput_translate_dx(tx)
- PICput_translate_dy(ty)
- PICput_translate_dz(tz)
- PICtranslate_dx()
- PICtranslate_dy()
- PICtranslate_dz()

Scaling Functions

- PICscale_x(x)
- PICscale_y(y)
- PICscale_z(z)
- PICscale(x,y,z)
- PICput_scale_dx(sx)
- PICput_scale_dy(sy)
- PICput_scale_dz(sz)
- PICscale_dx()
- PICscale_dy()
- PICscale_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 3-6).

Rotations may be absolute or incremental. Absolute rotations rotate about the x or y or z axis by θx, θy, and θz degrees. Also, arbitrary axis rotations allow you to specify an axis of rotation with a point, x,y,z and a direction, nx,ny,nz. 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.

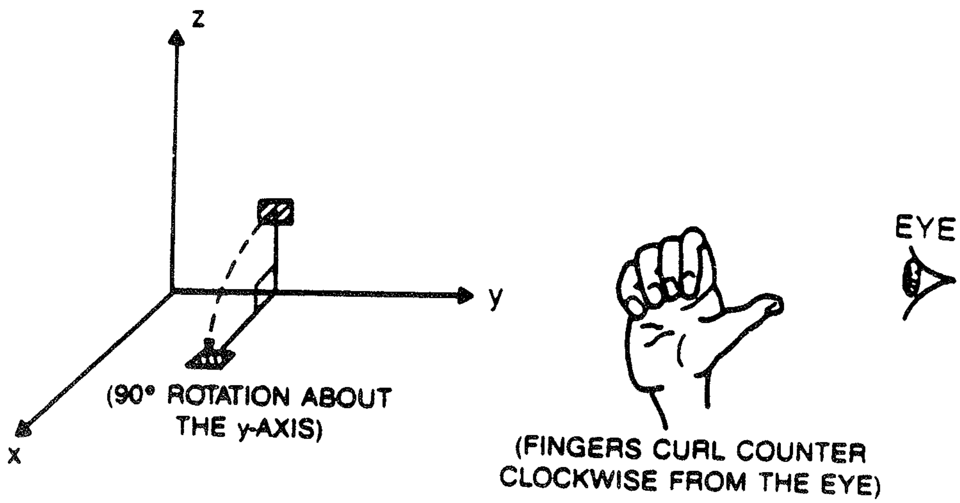
NOTE

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

The rotation functions are:

- PICrotate_x(x)
- PICrotate_y(y)
- PICrotate_z(z)
- PICrotate_vector(x,y,z,nx,ny,nz,angle)
- PICput_rotate_dx(dx)
- PICput_rotate_dy(dy)
- PICput_rotate_dz(dz)
- PICrotate_dx()
- PICrotate_dy()
- PICrotate_dz()

Figure 3-6: Right-Hand Rule Rotation



PICrotate Functions

The **PICrotate** functions (**PICrotate_x()**, **PICrotate_y()** and **PICrotate_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.

```
PICrotate_x(x)
float x;
```

x = the angle of rotation about the x axis

PICrotate_y(y)
float y;

y = the angle of rotation about the y axis

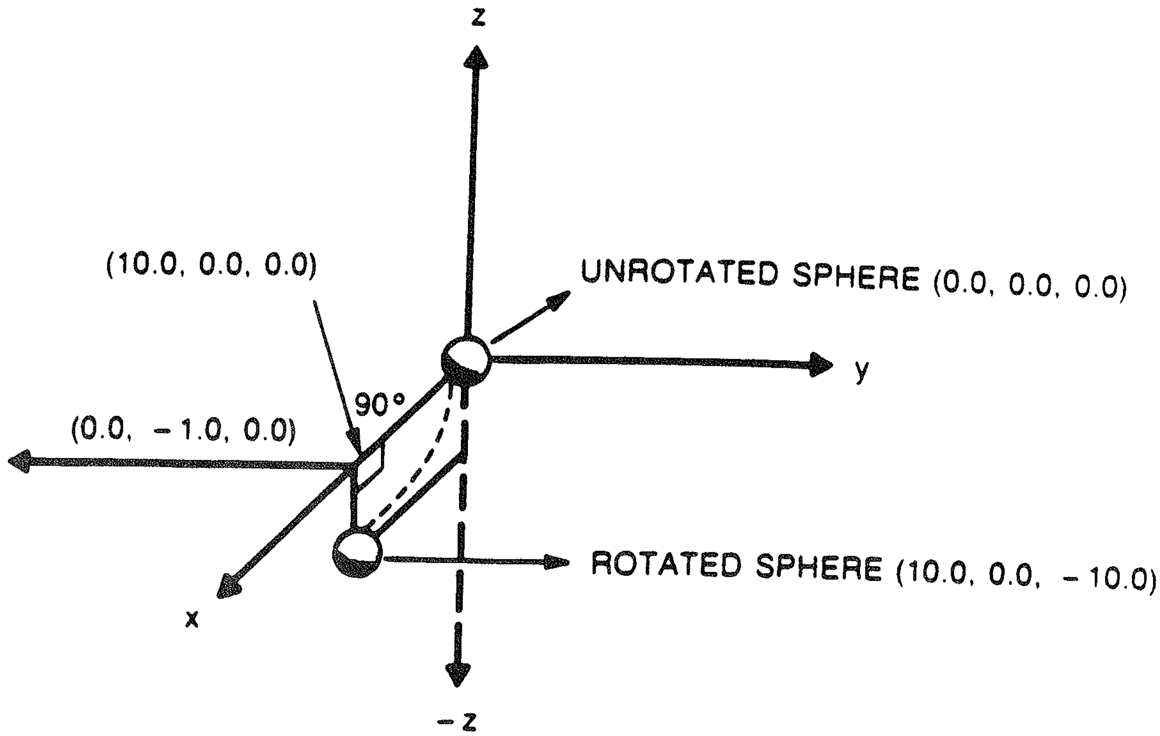
PICrotate_z(z)
float z;

z = the angle of rotation about the z axis

PICrotate_vector()

The **PICrotate_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 3-7: Arbitrary Axis Rotation (PICrotate_vector(10.0,0.0,0.0,0.0,-1.0,0.0,90.0);



PICrotate_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

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].

```
{
        .
        .
        .
PICmove_3d(10.0, 0.0, 0.0);

        /* draw the axis of rotation */

PICdraw_3d(10.0, 10.0, 10.0);

PICrotate_vector(10.0, 0.0, 0.0, 0.0, 1.0, 1.0, 90.0);

PICsphere();
        /* draw a unit sphere at the origin */

        .
        .
        .
}
```

PICput_rotate_d Functions

The **PICput_rotate_d** functions (**PICput_rotate_dx()**, **PICput_rotate_dy()** and **PICput_rotate_dz()**) define a constant that specifies increments of rotation in Δ degrees. Objects can then be rotated in increments about a World Space axis (x, y, or z) using the **PICrotate_d** functions.

PICput_rotate_dx(dx)
float dx;

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

PICput_rotate_dy(dy)
float dy;

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

PICput_rotate_dz(dz)
float dz;

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

PICrotate_d Functions

The **PICrotate_d** functions (**PICrotate_dx()**, **PICrotate_dy()** and **PICrotate_dz()**) rotate objects about the x, y, and/or z axis by a predefined, incremental rotation. Before using any of the **PICrotate_d** functions, be sure to specify the incremental angle with one of the **PICput_rotate_d** functions.

PICrotate_dx()

PICrotate_dy()

PICrotate_dz()

Translation

Objects may 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:

- PICtranslate_x(x)
- PICtranslate_y(y)
- PICtranslate_z(z)
- PICtranslate(x,y,z)
- PICput_translate_dx(tx)
- PICput_translate_dy(ty)
- PICput_translate_dz(tz)
- PICtranslate_dx()
- PICtranslate_dy()
- PICtranslate_dz()

PICtranslate Functions

The **PICtranslate** functions (**PICtranslate()**, **PICtranslate_x()**, **PICtranslate_y()** and **PICtranslate_z()**) apply a translation along *x* or *y* or *z* to the current transformation matrix.

PICtranslate(x,y,z)
float x,y,z;

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

PICtranslate_x(x)
float x;

x = the x translation

PICtranslate_y(y)
float y;

y = the y translation

PICtranslate_z(z)
float z;

z = the z translation

PICput_translate_d Functions

The **PICput_translate_d** functions (**PICput_translate_dx()**, **PICput_translate_dy()** and **PICput_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 **PICtranslate_d** functions.

PICput_translate_dx(tx)
float tx;

tx = the incremental translation in x

PICput_translate_dy(ty)
float ty;

ty = the incremental translation in y

PICput_translate_dz(tz)
float tz;

tz = the incremental translation in z

PICtranslate_d Functions

The **PICtranslate_d** functions (**PICtranslate_dx()**, **PICtranslate_dy()** and **PICtranslate_dz()**) translate the objects along the *x* or *y* or *z* axis by a predefined, incremental translation. Before using any of the **PICtranslate_d** functions, be sure to specify the incremental angle with one of the **PICput_translate_d** functions.

PICtranslate_dx()

PICtranslate_dy()

PICtranslate_dz()

Scaling

Objects may 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:

- PICscale_x(x)
- PICscale_y(y)
- PICscale_z(z)
- PICscale(x,y,z)
- PICput_scale_dx(sx)
- PICput_scale_dy(sy)
- PICput_scale_dz(sz)
- PICscale_dx()
- PICscale_dy()
- PICscale_dz()

PICscale Functions

The **PICscale** functions (**PICscale()**, **PICscale_x()**, **PICscale_y()** and **PICscale_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 one expand the object; less then one, reduce the object. Negative scaling factors mirror the scaled object across an axis.

PICscale(x,y,z)
float x,y,z;

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

PICscale_x(x)
float x;

x = the x scaling factor

PICscale_y(y)
float y;

y = the y scaling factor

PICscale_z(z)
float z;

z = the z scaling factor

PICput_scale_d Functions

The **PICput_scale_d** functions (**PICput_scale_dx()**, **PICput_scale_dy()** and **PICput_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 **PICscale_d** functions.

PICput_scale_dx(sx)
float sx;

sx = the incremental scaling factor in x

PICput_scale_dy(sy)
float sy;

sy = the incremental scaling factor in y

PICput_scale_dz(sz)
float sz;

sz = the incremental scaling factor in z

PICscale_d Functions

The **PICscale_d** functions (**PICscale_dx()**, **PICscale_dy()** and **PICscale_dz()**) scale the objects in *x* or *y* or *z* by a predefined scale factor. Before using any of the **PICscale_d** functions, be sure to specify the incremental angle with one of the **PICput_scale_d** functions.

PICscale_dx()

PICscale_dy()

PICscale_dz()

Example:

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

```
{
        .
        .
        .
PICpersp_project( 45.0, 1.25, 1.0, 1000.0 );
PIClookup_view( 150.0, 150.0, 150.0, 0.0, 0.0, 0.0, 0.0 );

PICput_scale_dx(3.0);

        /* set the incremental x scale value */
PICput_rotate_dz(20.0);

        /* set the incremental y rotation value */

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

        PICcolor_rgb( BLACK );
        PICclear_rgbz();

        PICrotate_dz();
        PICscale_dx();

        PICcolor_rgb( WHITE );
        PICpatch_geometry_3d(GX,GY,GZ);

        PICswap_buffer();

        .
        .
        .
}
```

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. PIClib provides four viewing functions for specifying the view point and viewing direction:

- **PICcamera_view(x,y,z,pan,tilt,swing)**
- **PIClookat_view(vx,vy,vz,px,py,pz,twist)**
- **PIClookup_view(vx,vy,vz,px,py,pz,twist)**
- **PICpolar_view(dist,azim,inc,twist)**

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

PICcamera_view(), **PIClookat_view()**, **PIClookup_view()** and **PICpolar_view()** all replace the current transformation with the specified viewing matrix. In order to preserve the current modeling and viewing transformation, use the **PICpush_transform()** command.

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

PICcamera_view()

PICcamera_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 of the Camera Coordinate System, a *tilt* degree rotation about the x axis of the Camera Coordinate System, 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 3-8 and 3-9. Note that the view vector in Figure 3-9 points toward the origin.

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

PICcamera_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

Figure 3-8: PICcamera_view(100.0, 100.0, 0.0, 0.0, 0.0, 0.0)

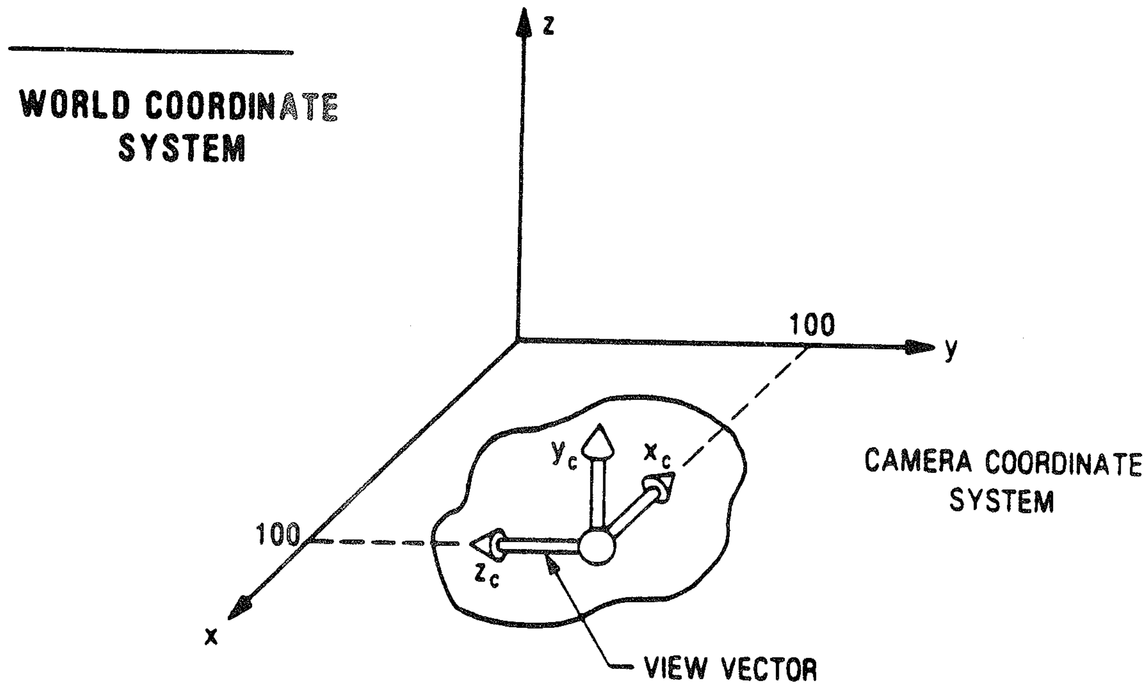
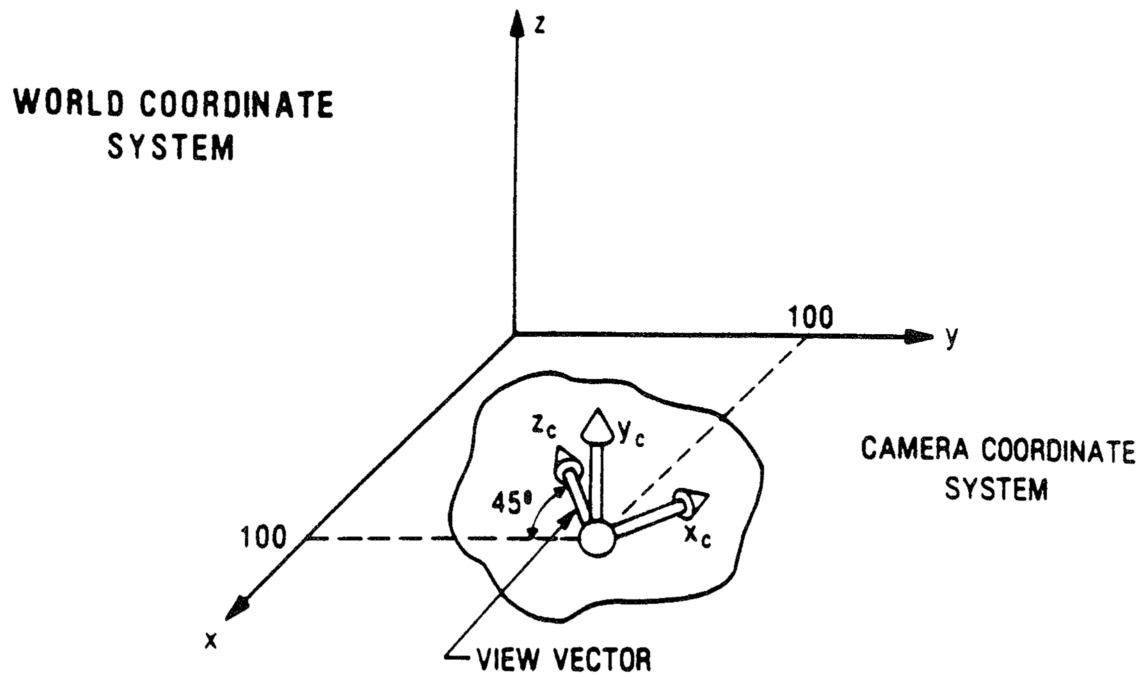


Figure 3-9: PICcamera_view(100.0, 100.0, 0.0, 45.0, 0.0, 0.0)



PIClookat_view()

PIClookat_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.

```
PIClookat_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)

PIClookup_view()

The **PIClookup_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 **PIClookup_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 **PIClookat_view()** function is used.

PIClookup_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)

PICpolar_view()

The **PICpolar_view()** function defines the viewpoint and direction in Polar Coordinates. The *dist* parameter is the distance from the view point 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.

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

- **Perspective pyramid**
- **Perspective window**
- **2D orthographic projection**
- **3D orthographic projection**

The projection functions described in this section are:

- **PICpersp_project(fovy,aspect,near,far)**
- **PICwindow_project(left,right,bottom,top,near,far)**
- **PICortho_project(left,right,bottom,top,near,far)**
- **PICortho_2D_project(left,right,bottom,top,near,far)**

PICpersp_project()

PICpersp_project() defines a 3D perspective viewing pyramid by specifying the field-of-view angle, *fovy*, in the *y* direction, the *aspect* ratio of the *x* and *y* Eye Space dimensions, and *near* and *far* clipping planes. The *z* clipping planes are specified by distances from the eye along the -*z* axis of the Eye Coordinate System. The *fovy* parameter and the *near* clipping plane establish the size of the projection frustum in the *y* direction. The size of the projection frustum in the *x* direction is multiplied by the *aspect* ratio. This ratio must match the aspect ratio of the current viewport in order to display data without distortions.

PICpersp_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 the *near* and *far* clipping planes along the view vector (the -z axis of the Eye Coordinate System)

PICwindow_project()

The **PICwindow_project()** function defines a 3D perspective projection by specifying a rectangular frustum between the *near* and *far* clipping planes. The parameters *left*, *right*, *bottom* and *top* define the position and size of the viewing window in the near clipping plane. These are specified in the *x* and *y* dimensions of the Eye Coordinate System. The parameters *near* and *far* define the distances from the eye to the clipping planes in the -z direction of the Eye Coordinate System.

PICwindow_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,far	=	the distances from the eye to the near and far clipping planes in the -z direction of the Eye Coordinate System

---

PICortho_project()

The **PICortho_project()** function defines a 3D orthographic projection with *left*, *right*, *bottom*, and *top* clipping planes in the *x* and *y* directions of the Eye Coordinate System. The *near* and *far* parameters represent the distances from the eye to the clipping planes in the -z direction of the Eye Coordinate System.

PICortho_project(left,right,bottom,top,near,far)
float left,right,bottom,top,near,far;

left,right,bottom,top	=	the clipping plane specified along the *x* and *y* axes of the Eye Coordinate System
near,far	=	the distances from the eye to the clipping planes in the -z direction of the Eye Coordinate System. Example: a *near* of **-10.0** is actually *behind* the eye, and a *far* of **1000.0** is 1000 units in from of the eye at -1000.0 z.

PICortho_2D_project()

The **PICortho_2D_project()** function defines a 2D orthographic projection by specifying the *left*, *right*, *bottom* and *top* clipping planes in the xy plane of the Eye Coordinate System.

PICortho_2D_project(left,right,bottom,top)
float left,right,bottom,top;

left,right,bottom,top	=	the *left*, *right*, *bottom* and *top* clipping planes specified along the x and y axes of the Eye Coordinate System

---

Transformations – Control Functions

The **Transformation Control** functions manipulate the transformation matrix stacks by pushing and popping matrices, pre and postmultiplying matrices, and loading or retrieving matrices. There are two transformation matrix stacks. One stack contains the modeling and viewing transformations, the other holds the projection transformations. Transformation Control operations are categorized by the stack they are manipulating.

Both the modeling and viewing transformation matrix and the projection transformation matrix are applied as follows:

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

The coefficients of a vector are contained in a column.

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 listed below:

- **PICget_inverse_transform(matrix)**
- **PICget_normal_transform(matrix)**
- **PICget_transform(matrix)**
- **PICpremultiply_transform(matrix)**
- **PICpostmultiply_transform(matrix)**
- **PICpush_transform()**
- **PICpop_transform()**
- **PICput_transform(matrix)**
- **PICput_identity_transform()**

PICget_inverse_transform()

The **PICget_inverse_transform()** function returns the *inverse* of the current MV transformation matrix.

PICget_inverse_transform(matrix)
PICmatrix matrix;

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

---

PICget_normal_transform()

The **PICget_normal_transform()** function returns the normal vector transformation matrix. This matrix is only available if shading or backface removal is on; otherwise, the identity matrix is returned. The normal vector transformation matrix is the inverse transpose of the upper 3x3 submatrix of the current transformation matrix.

PICget_normal_transform(matrix)
PICmatrix matrix;

matrix = indicates where to store the normal transformation matrix

---

PICget_transform()

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

PICget_transform(matrix)
PICmatrix matrix;

matrix = indicates where to store the current transformation matrix

---

PICpremultiply_transform()

The **PICpremultiply_transform()** function premultiplies the current MV transformation matrix by a specified matrix.

PICpremultiply_transform(matrix)
PICmatrix matrix;

matrix = a user-defined 4x4 matrix

PICpostmultiply_transform()

The **PICpostmultiply_transform()** function postmultiplies the current MV transformation matrix by a specified matrix.

PICpostmultiply_transform(matrix)
PICmatrix matrix;

matrix = a user-defined 4x4 matrix

PICpush_transform()

The **PICpush_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 PIC_MAX_TRANSFORM levels deep.

PICpush_transform()

PICpop_transform()

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

PICpop_transform()

Example:

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

```
{
        .
        .
        .
PICpersp_project( 45.0, 1.25, 1.0, 1000.0 );
PIClookup_view( 150.0, 150.0, 150.0, 0.0, 0.0, 0.0, 0.0 );

PICpush_transform();

        /* save the original coordinate system */

PICtranslate(10.0, 10.0, 10.0);
PICrotate_x(90.0);
PICsuperq_toroid(50.0, 50.0, 50.0, 90.0, 1.0, 2.0);

PICpop_transform();

        /* restore the original coordinate system */

PICsphere();

        .
        .
        .
}
```

PICput_transform()

The **PICput_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 **PICpush_transform()**.

PICput_transform(matrix)
PICmatrix matrix;

matrix = a user-defined 4x4 matrix

PICput_identity_transform()

The **PICput_identity_transform()** function places an *identity* matrix into the current MV transformation matrix.

PICput_identity_transform()

The identity matrix is of the form:

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

Projection Transformation Control Functions

The **Projection Transformation Control** functions operate on the current matrix and stack containing the projection transformations.

The Projection Transformation Control functions are:

- **PICget_inverse_project(matrix)**
- **PICget_project(matrix)**
- **PICpremultiply_project(matrix)**
- **PICpostmultiply_project(matrix)**
- **PICpush_project()**
- **PICpop_project()**
- **PICput_project(matrix)**

PICget_inverse_project()

The **PICget_inverse_project()** function returns the *inverse* of the current projection transformation matrix.

PICget_inverse_project(matrix)

matrix = indicates where to store the inverse of the current projection matrix

---

PICget_project()

The **PICget_project()** function returns the current projection transformation matrix.

PICget_project(matrix)
PICmatrix matrix;

matrix = indicates where to store the current projection matrix

---

PICpremultiply_project()

The **PICpremultiply_project()** function premultiplies the current projection transformation matrix by a specified matrix.

PICpremultiply_project(matrix)
PICmatrix matrix;

matrix = a user-defined 4x4 matrix

PICpostmultiply_project()

The **PICpostmultiply_project()** function postmultiplies the current projection transformation matrix by a specified matrix.

PICpostmultiply_project(matrix)
PICmatrix matrix;

matrix = a user-defined 4x4 matrix

PICpush_project()

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

```
PICpush_project()
```

PICpop_project()

The **PICpop_project()** function replaces the current projection transformation matrix with the matrix on top of the projection stack. If the projection stack is empty, this function has no effect.

```
PICpop_project()
```

PICput_project()

The **PICput_project()** function loads a specified 4x4 matrix into the current projection transformation matrix, replacing the original matrix. If you need to save a copy of the current projection transformation matrix on the projection stack, use **PICpush_project()**.

```
PICput_project(matrix)
PICmatrix matrix;
```

matrix = a user-defined 4x4 matrix

Viewport Functions

The *Viewport* functions let you define an active area on the screen. Viewports are defined by specifying the four limits of the viewport rectangle in the Pixel Coordinate System (see Figure 3-5). Depending on your Pixel Machine configuration, the screen area may be 1024x1024 or 1280x1024 for high resolution monitors and 720x480 for NTSC monitors.

In addition to defining viewports, these functions allow you to manipulate the viewport stack; set and retrieve the current viewport; set and retrieve the current depth ranges; and retrieve the current screen size.

The functions discussed in this section are:

- **PICget_screen_size(ix,iy)**
- **PICget_depth(near,far)**
- **PICget_viewport(left,right,top,bottom)**
- **PICpop_viewport()**
- **PICpush_viewport()**
- **PICput_depth(near,far)**
- **PICput_viewport(left,right,top,bottom)**

PICget_screen_size()

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

PICget_screen_size(&ix,&iy)
int ix,iy;

ix,iy = the screen's dimensions (1024x1024 or 1280x1024 for high resolution monitors and 720x480 for NTSC monitors)

PICget_depth()

The **PICget_depth()** function returns the z depth range associated with the current viewport. The z depth of the near plane is written into **near**; the z depth of the far plane is written into **far**.

PICget_depth(&near,&far)
float near,far;

near = the near (hither) plane

far = the far (yon) plane

PICget_viewport()

The **PICget_viewport()** function returns the coordinates of the current viewport. The viewport's initial and final *x* Pixel Coordinates are written into the **left** and **right** arguments, respectively; the initial and final *y* Pixel Coordinates are written into the **top** and **bottom** arguments, respectively.

PICget_viewport(&left,&right,&top,&bottom)
int left,right,top,bottom;

left,right
top,bottom = coordinates of the current viewport

PICpop_viewport()

The **PICpop_viewport()** replaces the current viewport with the viewport that is on top of the viewport stack. If the viewport stack is empty, this function has no effect. The depth values associated with each viewport are maintained on the stack with the viewport.

PICpop_viewport()

PICpush_viewport()

The **PICpush_viewport()** function copies the current viewport matrix to the top of the viewport stack. If the viewport stack is full, this function has no effect. The maximum number of viewports that can be stored is PIC_MAX_VIEWPORT.

PICpush_viewport()

PICput_depth()

The **PICput_depth()** function defines z range associated with the current viewport, thus establishing the z range between the near and far clipping planes. With a floating point buffer, a z range of 0.0 to 1.0 is usually sufficient. The **PICclear_z()** function clears the z buffer to the current value of *far*.

PICput_depth(near,far)

near = the minimum z value

far = the maximum z value

PICput_viewport()

The **PICput_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 (i.e., 1024x1024 or 1280x1024 in high resolution mode and 720x480 for NTSC mode). The left and right coordinates range from 0 to screen_width - 1, the top and bottom coordinates range from 0 to screen_height -1.

PICput_viewport(left,right,top,bottom)

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

Example:

To calculate the coordinates of a viewport of size 801x801 in the screen's center (given a model whose screen dimensions are 1280x1024) do the following:

left	=	(1279 - 801)/2	=	239
right	=	1279 - 239	=	1040
top	=	(1023 - 801)/2	=	111
bottom	=	1023 - 111	=	912

The coordinates of the viewport, then, are 239, 1040, 111, 912. Therefore,

```
PICput_viewport(239,1040,111,912);
```

Shading and Depth Cueing

The *Shading* and *Depth Cueing* functions allow you to use different shading modes, depth cueing, different types of light sources, and different surface properties.

This section discusses the following functions:

- **PICshade_mode(mode)**
- **PICget_shade_mode()**
- **PICflip(mode)**
- **PICclockwise(mode)**
- **PIClight_ambient(red,green,blue)**
- **PICput_light_source(type,index,light)**
- **PIClight_switch(index,state)**
- **PICpercent_texture(texture_contribution)**
- **PICput_surface_model(model)**
- **PICdepth_cue(mode)**
- **PICdepth_cue_limits(z0,r0,g0,b0,z1,r1,g1,b1)**
- **PICput_texture(type,offset_x,offset_y,size_x,size_y)**
- **PICset_texture(index)**
- **PICreset_texture()**
- **PICtexture_precision(mode)**

PICshade_mode()

The **PICshade_mode()** function allows you to select one of the following modes:

- Flat
- Gouraud
- Phong
- No shade

PICshade_mode(mode) int mode;

```
mode  =  PIC_SHADE_FLAT
      =  PIC_SHADE_GOURAUD
      =  PIC_SHADE_PHONG
      =  PIC_SHADE_OFF
```

Whenever you switch to PIC_SHADE_FLAT, PIC_SHADE_GOURAUD or PIC_SHADE_PHONG, you need to specify at least one light source (see **PICput_light_source()**) and a surface model (see **PICput_surface_model()**). The shading mode you select remains active and will affect all object rendered until a new shading mode is specified. See the description of Phong shading at the end of this section for further information on the use of this shading mode.

PICget_shade_mode()

The **PICget_shade_mode()** function returns a value that corresponds to one of the shade modes. These values are:

- PIC_SHADE_OFF for no shading
- PIC_SHADE_FLAT for flat shading
- PIC_SHADE_GOURAUD for Gouraud shading
- PIC_SHADE_PHONG for Phong shading

PICget_shade_mode()

PICflip()

The **PICflip()** function reverses all surface normals of polygons that face away from the viewer. The sign of the normal vectors that face away from the viewer is reversed. This causes polygons that face *away* from the viewer to be illuminated so as to appear to be facing *toward* the viewer.

NOTE: In order for **PICflip()** to operate properly, the polygons must be planar. **PICflip()** must be disabled before **PICbackface()** is enabled.

PICflip(mode)
int mode;

mode = PIC_ON or PIC_OFF

PICclockwise()

The **PICclockwise()** function defines how a normal vector of a polygon is computed. The calculation of the normal vector affects backface removal and normal shading. The first three vertices (P_0, P_1, P_2) of a polygon are used to form two vectors. When this function is set to PIC_ON, the normal vector is computed as

$$N = (P_0 - P_2) \times (P_1 - P_2)$$

When this function is set to PIC_OFF, the normal vector is computed as

$$N = (P_1 - P_2) \times (P_0 - P_2).$$

The default mode is counter-clockwise (PIC_OFF).

NOTE: The direction of the vector is defined by the right-hand rule.

PICclockwise(mode)
int mode;

mode = PIC_ON or PIC_OFF

PIClight_ambient()

The **PIClight_ambient()** function sets the ambient light intensity for a 3D scene or a group of objects. Ambient light (also called background color) is the illumination that is produced by the combination of light reflections from objects in a scene. You can specify an ambient light intensity only if shading is on (i.e., if you have selected PIC_SHADE_FLAT, PIC_SHADE_GOURAUD or PIC_SHADE_PHONG mode). The default setting is black (0.0, 0.0, 0.0).

PIClight_ambient(red,green,blue)
float red,green,blue;

red,green,blue = value of ambient light intensity

PIClight_switch()

The **PIClight_switch()** function allows you to selectively turn on or off any or all of the light sources you have defined for a scene. The following constants can be used to manipulate *all* light sources simultaneously:

PIC_TYPE_ALL	select all light types
PIC_LIGHT_ALL	select all light sources
PIC_BLACKOUT	switch all light sources off
PIC_SUNGLASSES	switch all light sources on

PIClight_switch(type,index,state)
int type,index,state;

type = PIC_LIGHT_DIRECT
PIC_LIGHT_SPOT
PIC_LIGHT_POINT

index = a user-defined number assigned to a light source and used to control an array of light sources

state = PIC_ON or PIC_OFF

PICput_light_source()

The **PICput_light_source()** function lets you select a light source. You can choose one of three types:

- **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 of the surface being illuminated. To calculate the diffuse light contribution (*Cd*) from directional light source, the following equation is used:†

*Cd = Kd * Lc * (Vn • Vl)*

To calculate the specular light contribution (*Cs*) from directional light source, the following equation is used:†

*Cs = Ks * Lc * (Ve • Vr)**Oe*

where,

Kd is the coefficient of diffuse reflection (from **PICput_surface_model()**)
Ks is the coefficient of specular reflection (from **PICput_surface_model()**)
Vn is the normal vector at a point on the object surface
Vl is the vector from the light source (from **PICput_light_source()**)
Lc is the color of the light source (from **PICput_light_source()**)
Ve is the vector from the object to the eye point
Vr is the reflection vector from the object
Oe is the object specular exponent (from **PICput_surface_model()**)

- **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. To calculate the diffuse light contribution (*Cd*) from directional light source, the following equation is used:†

  *Cd = Kd * Lc * (Vn • Vl)*

  To calculate the specular light contribution (*Cs*) from directional light source, use the following equation:†

  *Cs = Ks * Lc * (Ve • Vr)**Oe*

  where,

  Kd is the coefficient of diffuse reflection (from **PICput_surface_model()**)
  Ks is the coefficient of specular reflection (from **PICput_surface_model()**)
  Vn is the normal vector at a point on the object surface
  Vl is the vector from the object to the light source
  Lc is the color of the light source (from **PICput_light_source()**)
  Ve is the vector from the object to the eye point
  Vr is the reflection vector from the object
  Oe is the object specular exponent (from **PICput_surface_model()**)

- **Spot**— a unidirectional light source that is used to simulate localized lighting effects, but restricts the zone of illumination to a cone. As with the point light source, the calculation of spot light depends on the orientation and relative position of the surface being illuminated. The size of the cone, however, can vary as the light source concentration exponent is varied. To calculate the diffuse contribution (*Cd*) of spot light source, use the following equation:†

  *Cd = Kd * Lc * (Vn • Vl) * (Ld • Vl)**Le*

  To calculate the specular contribution (*Cs*) of spot light source, the following equation is used:†

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

where,

Kd is the coefficient of diffuse reflection (from **PICput_surface_model()**)
Ks is the coefficient of specular reflection (from **PICput_surface_model()**)
Lc is the light source color (from **PICput_light_source()**)
Vn is the normal vector at a point on the object surface
Ld is the direction of the light source
Vl is the vector from the object to the light source
Le is the light source concentration exponent (from **PICput_light_source()**)
Ve is the vector from the object to the eye point
Vr is the reflection vector from the object
Oe is the object specular exponent (from **PICput_surface_model()**)

† Adapted from PHIGS+ Functional Description; Revision 2.0; July 20, 1987; Andries van Dam.

PICput_light_source(type,index,light)
int type,index;
PIClight_source *light;

type	=	PIC_LIGHT_DIRECT
	=	PIC_LIGHT_SPOT
	=	PIC_LIGHT_POINT
index	=	a user-defined number assigned to a light source and used to control an array of light sources
light	=	a data structure defining the light's position, direction, color, concentration exponent, and angle

---

Please keep the following points in mind:

- You need to define a light source *after* you define the projection for a scene.
- Each time you change the projection, you need to redefine the light source.
- Once a light source is turned on, it remains on until it is turned off.
- You can define up to 50 light sources for each light type (i.e., directional, spot, or point).
- There is no default setting for **PICput_light_source()**. Therefore, you need to specify a light source.

PICput_surface_model()

The **PICput_surface_model()** function lets you define a data structure of surface characteristics.

PICput_surface_model(model)
PICsurface_model *model;

model = a data structure defining the object's ambient color (for red, green, and blue), diffuse color (for red, green, and blue), specular color (for red, green, and blue), specular exponent, and transparency

PICdepth_cue()

The **PICdepth_cue()** function allows you to turn depth cueing mode on or off. Depth cueing applies to points and vectors. When in depth cueing mode, points and vectors vary according to colors defined at the depth cueing limits.

PICdepth_cue(mode)
int mode;

mode = PIC_ON (turn depth cueing on)
= PIC_OFF (turn depth cueing off)

PICdepth_cue_limits()

The **PICdepth_cue_limits()** function sets the z limits and color range of depth cueing.

PICdepth_cue_limits($z_0, r_0, g_0, b_0, z_1, r_1, g_1, b_1$)
float $z_0, r_0, g_0, b_0, z_1, r_1, g_1, b_1$;

z_0 = the z depth at which to begin depth cueing
r_0,g_0,b_0 = the color at the beginning of the depth cueing limits
z_1 = the z depth at which to end depth cueing
r_1,g_1,b_1 = the color at the end of the depth cueing limits

---

All points or lines that fall within the specified z depth range (z_0,z_1) will have their color calculated by the following equation:

$$i = i_0 + \frac{z - z_0}{z_1 - z_0} * (i_1 - i_0)$$

Where z_0 and z_1 are the z limits described above, i_0 and i_1 represent the intensities at the z limits, z is the depth of the current point, and i is the computed intensity.

These z limits and colors are active until another **PICdepth_cue_limits()** is defined. z_0,z_1 are not necessarily the same as near and far clipping planes.

The colors are linearly interpolated based on the position of the objects relative to the z depth limits.

PICput_texture()

PICput_texture() defines an area of offscreen memory to be used as a single texture. There can be sixty-four of these texture areas. *type* is the type of texture, resident or virtual. Currently, only resident is supported. *type*=null represents resident texture. *offset_x, offset_y* is the starting location of texture in off-screen video RAM. *size_x, size_y* is the size of texture in pixels in off-screen video RAM.

```
PICput_texture(type, offset_x, offset_y, size_x, size_y)
unsigned long *type;
unsigned long offset_x, offset_y, size_x, size_y;
```

type = type of texture

offset_x, offset_y = starting location of texture in offscreen VRAM

size_x, size_y = size of texture in pixels in offscreen VRAM

PICset_texture()

PICset_texture() sets the current area to be used for texture mapping as defined by **PICput_texture()**.

index is the value returned by **PICput_texture()**, and it is used to reference the texture area defined by the associated **PICput_texture()** routine for all commands that follow and use texturing.

PIC_DEFAULT_TEXTURE can be used as *index* to reference the entire 256x256 texture area.

NOTE: If the texture maps are less than 256 X 256, PIClib supports multiple texture maps. They can be used simultaneously.

PICset_texture(index)
int index;

index = value used to reference the texture area

PICreset_texture()

PICreset_texture() sets the current area to be used for texture mapping.

The *current_texture_id* and the *next_texture_id* are set to PIC_DEFAULT_TEXTURE.

The texture area is set to 256X256.

---

void PICreset_texture()

PICtexture_precision()

PICtexture_precision() sets the precision for the display of texture mapped polygons. This is used in perspective mode and determines the number of times that a polygon is split when displayed to correct the perspective distortion of the texture. The *mode* argument is defined as follows:

PIC_LOW (default)
PIC_MEDIUM
PIC_HIGH

mode is the number of times that the polygon is split.

PICtexture_precision() takes any integer as *mode*; we have defined PIC_LOW, PIC_MEDIUM, and PIC_HIGH, but you can specify whatever integer you want.

NOTE The default setting corresponds to no splitting of polygons and causes the texture mapped polygons to appear as they have in previous releases. It should be noted that setting the *mode* to anything other than PIC_LOW will impact the speed of display of texture mapped polygons.

This function does not support the **PICpoly_point** macros.

---

int PICtexture_precision(mode)
int mode;

mode = PIC_LOW (default)
PIC_MEDIUM
PIC_HIGH

PICpercent_texture()

The **PICpercent_texture()** function indicates the contribution of the texture map's intensity value at each pixel with a floating point argument between 0.0 and 1.0. The compliment of this argument is the contribution of the interpolated Gouraud shaded value at each pixel. An argument of 0.0 indicates that the surface intensity is all Gouraud shaded. An argument of 1.0 means the surface intensity is all texture map.

PICpercent_texture(texture_contribution)
float texture_contribution

texture_contribution = contribution of the texture map's intensity value at each pixel. This value ranges from 0.0 to 1.0.

Phong Shading

PIClib allows you to select Phong shading as the shade mode.

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

Ia(x) is the ambient light intensity for the scene (from **PIClight_ambient()**).
Kd(x) is a component of the object's diffuse reflection coefficient (from the *d_** elements of **PICsurface_model()**).
Ka(x) is a component of the object's ambient coefficient (from the *a_** elements of **PICsurface_model()**).
Ks(x) is a component of the object's specular reflection coefficient (from the *s_** elements of **PICsurface_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 **PIClight_source()**).

Lc(x) is a component of the color of the light source (from the *r, g, b* elements of **PIClight_source()**).
Ve is the vector from the object to the eye point.
R is the reflection vector from the object which is the mirror vector of *Vl* about *Vn*.
S is the object's specular exponent (from the *exp* element of **PICsurface_model()**).
x is the red, green, and blue components of light.
ls number of lights; 5 point light sources and 5 directional lights.

The following formula is used to determine the shading of a pixel:

$$Color(x) = Ia(x) * Ka(x) + Kd(x) * \sum_{i=l}^{ls} Lc_i(x) * (\vec{N} \bullet \vec{L_i}) + Ks(x) * \sum_{i=l}^{ls} Lc_i(x) * (\vec{E} \bullet \vec{R_i})^S$$

The first term of the equation is the global ambient contribution to the pixel. This depends on the global illumination and the ambient coefficient of the object's surface model.

The second term is the diffuse elimination of all light sources and is controlled by the diffuse coefficient of the object's surface model (*Kd*) and the color and intensity of the light (*Lc*) and relative orientation of each light source compared to the normal of the object at that point (*N . L*).

The last term is the specular contribution of all light sources, and it is controlled by the specular coefficient of the object's surface model (*Ks*), the color and intensity of each light (*Lci*) and the dot product of (*E . R)**S*, where *E* is the vector in the eye direction and *R* is the reflection vector from the object which is the mirror of *L* about *N*. *S* is the specular coefficient in the object's surface model. The higher the value of S the sharper and smaller the area of the specular highlights.

Using Phong Shading

Because the pixel nodes contain a fixed amount of memory allocated for program storage, PIClib uses a pixel node code overlay mode facility. This allows PIClib to download code into the pixel nodes whenever it is needed. For the most part, it is transparent, that is the user does not have to keep track of what is loaded into the nodes; downloading of code is an automatic process. However, Phong shading is a special case. In order to render polygons as quickly as possible, the Phong shading code must be manually downloaded. This is done to avoid checking the overlay mode in **PICpoly_point()** commands, which can slow polygonal rendering considerably.

For Phong shading to work correctly, Phong overlay mode must be downloaded, while in PIC_SHADE_PHONG mode, before any surface model, light source (direct or point), ambient light or poly_point command is called. This is done by calling **PICput_overlay_mode**(PIC_PIXEL_PHONG). Once **PICput_overlay_mode()** is called, lights, ambient light and surfaces can be defined and Phong shaded polygons can be rendered. The data structures for lights and surfaces are static, so **PICput_overlay_mode()** need only be called when surface or lights change and *must* be called before rendering polygons. It is important to remember that other PIClib calls can download code over the Phong shading code. You should check to make sure that Phong overlay mode is loaded. This can be done by using the **PICget_overlay_mode()** function. For example to render a Phong shaded polygon:

```
          .
          .
          .
PICeuclid_mode(PIC_EUCLID_POLYGON);
PICshade_mode(PIC_SHADE_PHONG);
          .
          .
          .
mode = PICget_overlay_mode();
if (mode != PIC_PIXEL_PHONG)
        PICput_overlay_mode(PIC_PIXEL_PHONG);
set_surface();
set_lights();
draw_object();
          .
          .
          .
```

The functions that download code are:

1. Overlay mode 1: everything rendered in points or lines
2. Overlay mode 2: **PICatom_surface(), PICatom_light(), PICatom(), PICpixel_add(), PICpixel_multiply(), PICput_scan_line(), PICbroadcast_data()**
3. Overlay mode 3: **PICmake_sphere_template(), PICmake_template(), PICstamp_template()**
4. Overlay mode 4: Phong shading

To optimize program performance, it is recommended that switching between overlay modes be kept to a minimum. Phong shading always generates alpha mattes.

Color Functions

The Pixel Machine uses the rgb color system. All colors are specified as percentages of red, green, and blue. You can choose from a palette of 2^{24} colors.

PIClib offers the following color functions:

- PICcolor_rgb()
- PICcolor_alpha()

PICcolor_rgb()

The **PICcolor_rgb()** function defines the current color. The current color is used to color all objects subsequently specified by the user (i.e., points, lines, polygons, etc.).

PICcolor_rgb(r,g,b)
float r,g,b;

r,g,b = the specified percentages of red, green, and blue (between 0.0 and 1.0)

All colors are specified as normalized floating point numbers. A default color map is loaded each time **hypinit** is executed. The specified percentages of red, green and blue are multiplied by 255 and used as an index into a color lookup table. rgb color tables are used primarily for gamma correction. The lookup table does not affect the frame buffer, only the contents displayed on the video screen.

PICcolor_alpha()

The **PICcolor_alpha()** function defines the current alpha color. You can choose from 256 colors.

PICcolor_alpha(alpha)
int alpha;

alpha = the index that selects the current alpha color (between 0 and 255)

The current alpha color is used when writing into the alpha channel.

NOTE

PICenable_alpha() must be set *before* you can write into the alpha channel.

Display Functions

The **Display** functions perform operations on pixels, images, viewports, and data memory, such as, read or write a scan line of rgb pixels and enable or disable the alpha planes, overlay modes, or double buffer mode.

These functions are grouped into the following categories:

- **Clear** functions clear the current viewport to a specified color (rgb or alpha) and clear the z depth settings:
  - **PICclear_alpha()**
  - **PICclear_rgb()**
  - **PICclear_z()**
  - **PICclear_rgbz()**
- **Buffer** functions return data on the buffer and buffer mode and provide double buffering operations:
  - **PICget_buffer()**
  - **PICget_buffer_mode()**
  - **PICdouble_buffer(mode)**
  - **PICswap_buffer()**
- **Overlay** functions enable or disable writing to the alpha channel and select overlay mode:
  - **PICalpha(mode)**
  - **PICdisplay_overlay(mode)**
  - **PICoverlay_mode(mode)**
  - **PICput_overlay_mode(mode)**
  - **PICget_overlay_mode(mode)**
- **Scan Line** functions read and write from video and floating point memory banks:
  - **PICput_scan_line(ix,iy,red,green,blue,alpha,npixl,mode)**
  - **PICget_scan_line(ix,iy,red,green,blue,alpha,npixl,mode)**
  - **PICbroadcast_data(memory,ix,iy,data,nword,mode)**
  - **PICcomposite_mode(mode)**

- **Copy** functions copy screen and/or *z* data between buffers:
  - **PICcopy_front_to_back()**
  - **PICcopy_back_to_ext(buffer,ix,iy)**
  - **PICcopy_ext_to_back(buffer,ix,iy)**
  - **PICcopy_z_to_ext()**
  - **PICcopy_ext_to_z()**

PICclear_alpha()

The **PICclear_alpha()** function clears the alpha planes of the current viewport to the current alpha color.

PICclear_alpha()

PICclear_rgb()

The **PICclear_rgb()** function clears the rgb planes of the current viewport to the current rgb color.

PICclear_rgb()

PICclear_z()

The **PICclear_z()** function clears the *z* depth of the current viewport to the *far* value specified by the **PICput_depth()** function.

PICclear_z()

PICclear_rgbz()

The **PICclear_rgbz()** function clears the rgb planes of the current viewport to the current rgb color and clears the *z* depth to the *far* value specified by the **PICput_depth()** function.

PICclear_rgbz()

PICget_buffer()

The **PICget_buffer()** function returns an integer indicating the number of the current display buffer. The number is either PIC_BUFFER_ZERO or PIC_BUFFER_ONE. When you initialize PIClib, the front buffer is PIC_BUFFER_ZERO (this buffer is displayed on the screen) and the back buffer is PIC_BUFFER_ONE.

PICget_buffer()

PICget_buffer_mode()

The **PICget_buffer_mode()** function returns an integer indicating which buffer mode is being used (single or double). PIC_SINGLE_BUFFER indicates single buffer mode; PIC_DOUBLE_BUFFER indicates double buffer mode.

PICget_buffer_mode()

PICdouble_buffer()

The **PICdouble_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 **PICswap_buffer()** function after completing a frame.) When disabled, objects are drawn into the front buffer only, which is displayed on the screen.

PICdouble_buffer(mode)
int mode;

mode = PIC_ON or PIC_OFF

PICswap_buffer()

The **PICswap_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 first enable double buffering (**PICdouble_buffer()**) *before* using **PICswap_buffer()**.

PICswap_buffer()

PICdisplay_overlay()

The **PICdisplay_overlay()** function enables or disables the display of overlays. If overlays are disabled (**mode** = PIC_OFF) the rgb channels are always displayed. If overlays are enabled (**mode** = PIC_ON) the rgb or alpha channels are conditionally displayed according to the mode set by **PICoverlay_mode()**.

PICdisplay_overlay(mode)
int mode;

mode = PIC_ON or PIC_OFF

PICoverlay_mode()

The **PICoverlay_mode()** function selects the overlay mode to be used when overlays are enabled. When overlays are disabled, the rgb signal is always displayed; when enabled, the alpha channel and inverted rgb can be displayed, or you can toggle between the alpha and rgb channels. When rendering into the alpha channel, it is suggested that you use mode PIC_OVERLAY_NON_ZERO and avoid the alpha entry 255. When using the cursor, the PIC_OVERLAY_HIGH_BIT mode should be used.

PICoverlay_mode(mode)
int mode;

mode	=	PIC_OVERLAY_OFF	Disable overlays; rgb signal always displayed
	=	PIC_OVERLAY_NON_ZERO	If the alpha channel is non-zero, display it; otherwise, display the rgb signal; if the alpha channel is all 1's (α = 255), display inverted rgb
	=	PIC_OVERLAY_HIGH_BIT	Toggle mode; if the most significant bit of the alpha channel is set (i.e., bit 7 = 1), display the contents of the alpha channel; if it is not set (i.e., bit 7 = 0), display the rgb signal

---

NOTE

Be sure to enable writing into the alpha channel before using overlays. See **PICalpha()**.
Be sure to enable the display of overlays. See **PICdisplay_overlay()**.

PICput_overlay_mode()

PICput_overlay_mode() takes an overlay mode and downloads the code associated with the overlay. At present, only a mode equal to PIC_PIXEL_PHONG will download any code. The overlay must be loaded before any PIClib function can be called.

---

void PICput_overlay_mode(mode)
int mode

mode = PIC_PIXEL_PHONG

PICget_overlay_mode()

PICget_overlay_mode() returns the overlay mode that is currently loaded into the pixel nodes.

---

int PICput_overlay_mode()

PICalpha()

The **PICalpha()** function enables or disables rendering into the alpha channel. When disabled, rendering is done in the rgb channels. You should refer to **PICdisplay_overlay()** and **PICoverlay_mode()** to display contents of the alpha channel. Objects are rendered using the current alpha color set by **PICcolor_alpha()**. To load a 24 bit color into the alpha channel lookup table, use **PICput_alpha_map_entry()**. Lines are not rendered into the overlay modes, however, cursors, raster text, flat and filled polygons are.

PICalpha(mode)
int mode;

mode = PIC_ON or PIC_OFF

Example:

The following program illustrates alpha channel rendering.

```
typedef   struct {
                float     red;
                float     green;
                float     blue;
          }  alpha_rgb;

static alpha_rgb   pink = { 0.9, 0.4, 0.7 };

main()
{
          PICinit();

          PICcolor_alpha(0);

          /* current alpha color = entry 0 */

          PICclear_alpha();

          /* clear alpha channel to zero          */

          PICalpha(PIC_ON);

          /* enable alpha rendering      */

          /* select overlay mode         */

          PICoverlay_mode( PIC_OVERLAY_NON_ZERO );

          /* display overlays                     */

          PICdisplay_overlay(PIC_ON);

          /*  disable updating from the shadow lookup table               */
          /*  set alpha entry #5 to pink in the shadow lookup table       */
          /*  enable updating from the shadow look up table               */

          PICupdate_map(PIC_OFF);

          PICput_alpha_map_entry(5, pink.red, pink.green, pink.blue);

          PICupdate_map(PIC_ON);

          PICcolor_alpha(5);

          /* current alpha color = entry 5 */

          PICeuclid_mode(PIC_EUCLID_POLYGON);

          PICshade_mode(PIC_SHADE_OFF);

          /* SHADE_FLAT or SHADE_OFF renders into the alpha channel */
```

(continued on next page)

Composite Mode	F_A	F_B
PIC_NO_COMPOSITE	1	0
PIC_A_OVER_B	1	$1 - \alpha_A$
PIC_B_OVER_A	$1 - \alpha_B$	1
PIC_A_IN_B	α_B	0
PIC_B_IN_A	0	α_A
PIC_A_OUT_B	$1 - \alpha_B$	0
PIC_B_OUT_A	0	$1 - \alpha_A$
PIC_A_ATOP_B	α_B	$1 - \alpha_A$
PIC_B_ATOP_A	$1 - \alpha_B$	α_A
PIC_A_XOR_B	$1 - \alpha_B$	$1 - \alpha_A$
PIC_PLUS	1	1

The compositing operation requires two source images. One is the image in the current buffer (the back buffer in double buffer mode and the front buffer in single buffer mode). The other image is sent to the Pixel Machine via the Pipeline using the **PICput_scan_line()** function. The two images are composited using the current compositing mode, and the result is stored in the current buffer, overwriting the original image.

PICcomposite_mode(mode)
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 that determine the color of each pixel
npixl	=	the number of pixels in the scan line. **PICput_scan_line()** can write an individual pixel by setting *npixl* to one.

mode	=	PIC_RGB_PIXELS	Each pixel is 24 bits of rgb; 8 bits from each *red, green, blue* array.
	=	PIC_RGB_PACKED_PIXELS	Each pixel is 24 bits of rgb from a packed array pointed to by *red*. The pixel components are stored in rgb order, and the pixels are stored in rgb order. The first byte in *red* contains the red component of the first pixel. Alpha remains unchanged.
	=	PIC_RGBA_PIXELS	Each pixel is 32 bits of rgbα; 8 bits from each *red, green, blue, alpha* array.
	=	PIC_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.
	=	PIC_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.
	=	PIC_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, which means that the run is 1 pixel long, to 255, which means that the run is 256 pixels long. In this mode, *npixl* refers to the number of runs in the scan line.
	=	PIC_EXTENDED_VRAM	If PIC_EXTENDED_VRAM is added to *mode*, the scan line is written into the extended video memory.

PICget_scan_line()

The **PICget_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 to call **PICwait_psync()** before the first call to **PICget_scan_line()**. This ensures that the entire frame has been drawn before any scan lines are read.

```
PICget_scan_line(ix,iy,red,green,blue,alpha,npixl,mode)
int ix,iy;
PICpixel *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. **PICget_scan_line()** can read an individual pixel by setting *npixl* to one.

mode	=	PIC_RGB_PIXELS	Each pixel is 24 bits of rgb (8 bits stored to each *red, green, blue* array).
	=	PIC_RGB_PACKED_PIXELS	Each pixel is 24 bits of rgb written to an array pointed to by *red*. The pixel components are stored in rgb order. The first byte in *red* contains the red component of the first pixel.
	=	PIC_RGBA_PIXELS	Each pixel is 32 bits rgbα (8 bits stored to each *red, green, blue, alpha* array)
	=	PIC_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.
	=	PIC_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.

=	PIC_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, which means that the run is 1 pixel long, to 255, which means that the run is 256 pixels long. In this mode, *npixl* refers to the number of runs in the scan line.
=	PIC_EXTENDED_VRAM	If PIC_EXTENDED_VRAM is added to *mode*, the scan line is read from the extended video memory.

PICput_image_header()

PICput_image_header() writes the PICimage_header and the optional user header (if one exists) to the specified file.

file is a file descriptor obtained from a previous call to *fopen(3)*. The file must have been successfully opened for writing and the file pointer should be pointing to the beginning of the file (i.e., no previous writes have been issued). Upon return from **PICput_image_header()**, the file pointer will be set to where the pixel data should start (i.e., past the image and optional headers).

PICput_image_header() will convert the PICimage_header structure pointed to by *image_header* into a string of decimal ASCII characters and write it to the file pointed to by *file*. If the *magic* structure member is 0, it will be set to PIC_IMAGE_MAGIC before being written. If *magic* is non-zero, it will be written as is.

If *optional_header* is non-zero, the characters pointed to it will be written to *file* immediately after the image header. *image_header->optional_header_size* bytes will be written.

PICput_image_header() returns 0 upon success and -1 on failure. **PICput_image_header()** will fail for the following reasons:

- the magic number is not PIC_IMAGE_MAGIC, or
- an error was returned by the *fwrite(3)* system call while writing either the image header or the optional header.

NOTE

No value in the PICimage_header should be greater than 1,2,000,000.

All Pixel Machine libraries share the same image header format.

```
#include &lt;stdio.h&gt;
#include &lt;picimage.h&gt;

int PICput_image_header(file, image_header, optional_header)
FILE              *file;
PICimage_header *image_header;
unsigned char        *optional_header;
```

file	=	file to which the header is written
image_header	=	pointer to the PICimage_header structure
optional_header	=	pointer to characters to be written following the header

---

PICget_image_header()

PICget_image_header() reads the PICimage_header and the optional header (if one exists) from the specified file and returns them to the caller.

file is a file descriptor obtained from a previous call to *fopen(3)*. The file must have been successfully opened for reading and the file pointer should be pointing to the beginning of the file (i.e., no previous reads have been issued). Upon return from **PICget_image_header()**, the file pointer will be set to the beginning of the pixel data (i.e., past the image and optional headers).

PICget_image_header() reads in the first PIC_IMAGE_HEADER_SIZE bytes from the file, converts them from ASCII into unsigned longs and place them into the correct locations in the structure pointed to by *image_header*. Except for the *magic* and *optional_header_size* fields, none of the information in the header is checked for validity.

If an optional header is present (image_header->optional_header_size is not 0), memory will be allocated (via *malloc(3)*) and *image_header->optional_header_size* bytes will be read. A pointer to the allocated memory will be returned in **optional_header*. If no optional header is present, **optional_header* will be set to NULL.

PICget_image_header() returns 0 upon success and -1 on failure. **PICget_image_header()** will fail for one of the following reasons:

- the magic number is not PIC_IMAGE_MAGIC, or
- an error was returned by the *fread(3)* system call while reading either the image header or the optional header.

NOTE: All Pixel Machine libraries share the same image header format.

```
#include <stdio.h>
#include <picimage.h>

int PICget_image_header(file, image_header, optional_header)
FILE            *file;
PICimage_header *image_header;
unsigned char        **optional_header;
```

file	=	file from which the header is read
image_header	=	location into which the header is read
optional_header	=	additional bytes to be read

PICbroadcast_data()

The **PICbroadcast_data()** function broadcasts a *line* of data to extended video memory (*memory* = PIC_BROADCAST_VRAM) or to z memory (*memory* = PIC_BROADCAST_ZRAM). 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 order or in αbgr order depending on the parameter *mode*. A common use of **PICbroadcast_data()** is to broadcast textures to VRAM so that all nodes receive the same data.

If the data is broadcast to the z memory, each 32-bit word can contain any data (floating point, long integer, 2 short integers or 4 bytes). 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*.

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

memory	=	PIC_BROADCAST_VRAM or	
	=	PIC_BROADCAST_ZRAM	
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	=	PIC_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.
	=	PIC_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.

PICcopy_front_to_back()

The **PICcopy_front_to_back()** function copies the contents of the current viewport from the front buffer to the back buffer.

PICcopy_front_to_back()

PICcopy_back_to_front()

PICcopy_back_to_front() copies the contents of the back buffer to the front buffer. This function only copies the contents of the current viewport.

---

void PICcopy_back_to_front()

PICcopy_back_to_ext()

The **PICcopy_back_to_ext()** function copies the contents of the current viewport from the back buffer to the extended screen buffer.
The coordinates *ix, iy* are used with the PIC_SCREEN_BUFFER constant to specify where in the off-screen image buffer to copy the contents of the current viewport. The size of the off-screen buffer varies, depending on the model, as follows:

Model	Off-screen Buffer Size
964x	2048x2048
964	2048x2048
964n	2048x2048
940	1280x2048
940n	2560x1024
932	1024x2048
932n	2048x1024
920	-
920n	1280x1024
916	-
916n	1024x1024

Because 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 PIC_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

There are two available extended buffers: PIC_TOP_BUFFER and PIC_BOTTOM_BUFFER. These are used for copying rgb planes to off-screen memory for 3D compositing and other purposes. When *buffer* is set to PIC_SCREEN_BUFFER, the extended memory is treated as a single large buffer and you need to specify the location indicating where to place the contents of the current viewport. Use PIC_SCREEN_BUFFER when you want to create flipbooks or scroll through a large image.

PICcopy_back_to_ext(buffer,ix,iy)
int buffer;
int ix, iy;

buffer = PIC_TOP_BUFFER, PIC_BOTTOM_BUFFER, or PIC_SCREEN_BUFFER

ix, iy = coordinates in an off-screen image buffer

PICcopy_ext_to_back()

The **PICcopy_ext_to_back()** function copies a region from the extended-screen buffer to the current viewport.

PICcopy_ext_to_back(buffer,ix,iy)
int buffer;
int ix, iy;

buffer = PIC_TOP_BUFFER, PIC_BOTTOM_BUFFER, or PIC_SCREEN_BUFFER

ix, iy = coordinates in an off-screen image buffer. These coordinates are used with the PIC_SCREEN_BUFFER constant to specify what part of the off-screen image buffer to copy into the current viewport. The size of the off-screen buffer varies, depending on the model. See the description of **PICcopy_back_to_ext()** above for the buffer sizes.

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 PIC_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 in Figure 3-11 lists the Nx and Ny values for the various Pixel Machine models.

There are two available extended buffers: PIC_TOP_BUFFER and PIC_BOTTOM_BUFFER. These are used for copying rgb planes to off-screen memory for 3D compositing and other purposes. When *buffer* is set to PIC_SCREEN_BUFFER, the extended memory is treated as a single large buffer and you need to specify the location indicating what part of the off-screen image buffer to copy into the current viewport. Use PIC_SCREEN_BUFFER when you want to create flipbooks or scrolling through a large image.

PICcopy_z_to_ext()

The **PICcopy_z_to_ext()** function copies the contents of the z buffer to the extended-screen z buffer. The region copied is defined by the current viewport.

```
PICcopy_z_to_ext()
```

PICcopy_ext_to_z()

The **PICcopy_ext_to_z()** function copies the contents of the extended-screen z buffer to the screen z buffer. The region copied is defined by the current viewport.

```
PICcopy_ext_to_z()
```

Hidden Surface Removal

The **Hidden Surface Removal** functions allow you to create realistic images by removing those surfaces that are hidden from view. These functions are:

- **PICzbuffer(mode)**
- **PICbackface(mode)**
- **PICzbuffer_lines(mode)**

PICzbuffer()

The **PICzbuffer()** function enables/disables hidden surface removal. The function removes hidden surfaces by comparing the z depth value of each pixel in a polygon to the contents of the z buffer for that pixel, and writes only those pixels that have a value less than that of the z buffer. The z buffer is initialized by the **PICclear_z()** function.

The table below describes the available z buffer modes:

Mode	Description
PIC_OFF	neither tests against nor writes the z buffer
PIC_ON	tests against and writes the z buffer
PIC_READ_ONLY	tests against but does not write the z buffer
PIC_WRITE_ONLY	writes the z buffer unconditionally

PICzbuffer(mode)
int mode;

mode = PIC_ON or PIC_OFF

PICbackface()

The **PICbackface()** function removes surfaces that face away from a specified viewing position. In order for backface removal to operate properly, the object must be closed and the polygons must be planar.

This function uses the eye position and the normal vector to the polygon to compute visibility. If no normal is given (with **PICpoly_normal()**), one is constructed from the first 3 vertices of the polygon. For this reason if it important to specify your vertices in a consistent order. The default order for specifying the vertices of a polygon is counterclockwise, viewing the polygon from the outside. For more information, refer to the description of the **PICclockwise()** function.

NOTE: In order for backface removal to operate properly, make sure the polygons are planar. Also note that **PICflip()** must be disabled before **PICbackface()** is enabled.

PICbackface(mode)
int mode;

mode = PIC_ON or PIC_OFF

PICzbuffer_lines()

The **PICzbuffer_lines()** functions controls whether lines are zbuffered or not. Zbuffered lines are aliased and can be rendered as depth-cued or current color lines. **PICinit()** initializes zbuffered lines to PIC_OFF.

NOTE: Because non-zbuffered lines are more efficient than zbuffered lines, it is recommended that you use **PICzbuffer_lines(**PIC_OFF**) when zbuffering is not required.**

PICzbuffer_lines(mode)
int mode;

mode = PIC_ON or PIC_OFF

Antialiasing

The *Antialiasing* functions allow you to eliminate jagged lines or edges in the objects of your scene. This section discusses the following functions:

- **PICantialias_lines(mode)**
- **PICinit_sampling(xsamples,ysamples,xscale,yscale,filter)**
- **PICenter_sampling_pass()**
- **PICexit_sampling_pass()**

NOTE: Antialiasing by supersampling (**PICinit_sampling()**, **PICenter_sampling()** and **PICexit_sampling()**) uses the external z memory, and therefore can be used only on models 932 and higher in high resolution mode and on all models in NTSC mode.

PICantialias_lines()

The **PICantialias_lines()** function determines whether lines are to be antialiased. To antialias an object, use the **PICinit_sampling()**, **PICenter_sampling_pass()** and **PICexit_sampling_pass()** functions described below.

PICantialias_lines(mode)
int mode;

mode = PIC_ON or PIC_OFF

PICinit_sampling()

PICinit_sampling() initializes super-sampling mode for use in antialiasing objects. Based on the arguments passed to it, **PICinit_sampling()** returns the number of sampling passes required on the scene. The arguments *xsamples* and *ysamples* are the number of samples in x and y, respectively, to take per pixel. The samples can be taken over a section of pixels, depending on *xscale* and *yscale*. For one pixel coverage, *xscale* and *yscale* should each be 1.0. Different filters can be defined for use in filtering the samples. The *filter* parameter should be an array of size (*xsamples* * *ysamples*).

The return value *npass* should be used to control the loop over the scene description with calls to **PICenter_sampling_pass()** and **PICexit_sampling_pass()** at the beginning and end of each iteration.

If *amode* = PIC_OFF, alpha matte generation is ignored. If *amode* = PIC_ON, an alpha matte is generated for the image. To generate an alpha matte correctly, the image must be Phong shaded.

If the function is called on a model 916 or a model 920 in high resolution mode, the return value will be zero.

NOTE: Because this function uses external z-memory, it can only be used with Pixel Machine models 932 and higher in high resolution mode, and on all models in NTSC mode.

xsamples,ysamples	=	the number of sampling points in the x and y directions
xscale,yscale	=	pixel scale factor.
filter	=	a matrix of size (xsamples x ysamples) which stores the coefficients to be applied to the samples
amode	=	PIC_ON – an alpha matte is generated for the image PIC_OFF – alpha matte generation is ignored

PICenter_sampling_pass()

The **PICenter_sampling_pass()** function marks the beginning of a sampling pass. This command alters the projection matrix.

NOTE: Remember to initialize the frame buffer and the z buffer before rendering the scene. (This can be done with any of the clear or copy functions).

PICenter_sampling_pass()

PICexit_sampling_pass()

The **PICexit_sampling_pass()** marks the end of a sampling pass. This command restores the projection matrix.

PICexit_sampling_pass()

Example:

```
#define     XSAMPLES     4
#define     YSAMPLES     4
#define     SAMPLES      (XSAMPLES * YSAMPLES)
#define     XSCALE       1.0
#define     YSCALE       1.0

        .
        .
        .
{
    int npass;
    float filter[SAMPLES];

    for ( i=0; i<SAMPLES;i++ ) filter[i]=1.0/(float)SAMPLES;

    npass=PICinit_sampling (XSAMPLES, YSAMPLES, XSCALE, YSCALE, filter);

    for ( i=0; i<npass;i++ ){
        PICenter_sampling_pass();
        PICclear_rgbz();
        draw_scene();
        PICexit_sampling_pass();
    }
        .
        .
        .
}
```

Video Functions

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

- PICupdate_map(mode)
- PICput_color_map(red,green,blue)
- PICput_color_map_entry(index,red,green,blue)
- PICput_alpha_map(red,green,blue)
- PICput_alpha_map_entry(index,red,green,blue)
- PICget_color_map(red,green,blue)
- PICget_color_map_entry(index,red,green,blue)
- PICget_alpha_map(red,green,blue)
- PICget_alpha_map_entry(index,red,green,blue)

PICupdate_map()

The **PICupdate_map()** function displays immediately any changes made to the video. The function is enabled by specifying the PIC_ON mode. When PIC_OFF, changes to the video are not visible until the function is re-enabled.

Whenever altering any of the color tables, it is suggested that you first call **PICupdate_map(PIC_OFF)**, and then call **PICupdate_map(PIC_ON)** after all your changes are complete.

```
PICupdate_map(mode)
int mode;
```

mode = PIC_ON or PIC_OFF

PICput_color_map()

The **PICput_color_map()** function loads an entire lookup table for each rgb channel. The values contained in these tables are in normalized form (between 0.0 and 1.0).

PICput_color_map(red,green,blue)
float *red,*green,*blue;

PICput_color_map_entry()

The **PICput_color_map_entry()** function loads a specified entry into the rgb color map. *Index* can range from 0 to PIC_VIDEO_TABLE - 1.

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

index = indicates which entry is being updated

PICput_alpha_map()

The **PICput_alpha_map()** function loads an entire lookup table for the alpha channel.

PICput_alpha_map(red,green,blue)
float *red,*green,*blue;

PICput_alpha_map_entry()

The **PICput_alpha_map_entry()** function loads a specified entry in the color map for the alpha channel. *index* can range from 0 to PIC_VIDEO_TABLE - 1.

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

index = indicates which entry is being updated

PICget_color_map()

The **PICget_color_map()** function returns arrays of r, g, and b values from the current rgb lookup map. These arrays (red, green, and blue) are of length PIC_VIDEO_TABLE.

PICget_color_map(red,green,blue)
float *red,*green,*blue;

PICget_color_map_entry()

The **PICget_color_map_entry()** function returns a specified rgb entry from the current rgb lookup table. *index* can range from 0 to PIC_VIDEO_TABLE - 1.

PICget_color_map_entry(index,red,green,blue)
int index;
float *red,*green,*blue;

PICget_alpha_map()

The **PICget_alpha_map()** function returns arrays for the current r, g, and b values in the alpha map. Each red, green, and blue array is of length PIC_VIDEO_TABLE.

```
PICget_alpha_map(red, green, blue)
float *red,*green,*blue;
```

PICget_alpha_map_entry()

The **PICget_alpha_map_entry()** function returns a specified rgb alpha map entry. *index* can range from 0 to PIC_VIDEO_TABLE - 1.

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

Raster Operations

The **Raster Operations** functions manipulate the intensities of pixels by adding, subtracting, or multiplying them by a constant value. This section discusses the following functions:

- **PICpixel_add(red,green,blue,alpha)**
- **PICpixels_multiply(red,green,blue,alpha)**

PICpixel_add()

The **PICpixel_add()** function adds a constant value to the intensities of all pixels in the current viewport.

PICpixel_add(red,green,blue,alpha)
float red,green,blue,alpha;

red,green,blue,alpha = the rgb and alpha values to be added to the pixel values

---

PICpixel_multiply()

The **PICpixel_multiply()** function multiplies the intensities of pixels in the current viewport by a constant value.

PICpixel_multiply(red,green,blue,alpha)
float red,green,blue,alpha;

red,green,blue,alpha = the rgb and alpha values to be multiplied by the pixel values

---

Input Device Functions

The **Input Device** functions let you control the operation of a mouse; query the state of a button, a valuator, or the current value of a 2D locator; query the event queue and sample keyboard buttons; and define a cursor and move it with or without the mouse. The functions discussed in this section are:

- PICattach_mouse()
- PICdetach_mouse()
- PICget_button(button)
- PICget_valuator(valuator)
- PICget_locator(x,y)
- PICquery_queue(event,value)
- PICflush_queue()
- PICput_mouse_playground(left,right,top,bottom)
- PICqueue_events(mode)
- PICget_event(event,value)
- PICdisplay_cursor(mode)
- PICdefine_cursor(cursor)
- PICposition_cursor(ix,iy)
- PICwait_event(event,value)
- PICget_host_screen_size(width,height)

PICattach_mouse()

The **PICattach_mouse()** function initializes the mouse and must be called before any other Input Device function.

PICattach_mouse()

PICdetach_mouse()

The **PICdetach_mouse()** function terminates the operation of the mouse and must be the last Input Device function called.

PICdetach_mouse()

PICget_button()

The **PICget_button()** function returns the state of a mouse button indicated by the argument *button*. If the button is *currently* pressed, the function returns a value of PIC_TRUE; if not, returns a value of PIC_FALSE.

PICget_button(button)
int button;

button	=	PIC_LEFTMOUSE
	=	PIC_RIGHTMOUSE
	=	PIC_MIDDLEMOUSE

PICget_valuator()

The **PICget_valuator()** function returns the current value of a valuator.

PICget_valuator(valuator)
int valuator;

valuator = PIC_XMOUSE,PIC_YMOUSE

PICget_locator()

The **PICget_locator()** function returns the current value of a locator's x and y position. The return values are stored in the locations pointed to by x and y respectively.

PICget_locator(x,y)
int *x, *y;

Input Device Functions

The **Input Device** functions let you control the operation of a mouse; query the state of a button, a valuator, or the current value of a 2D locator; query the event queue and sample keyboard buttons; and define a cursor and move it with or without the mouse. The functions discussed in this section are:

- PICattach_mouse()
- PICdetach_mouse()
- PICget_button(button)
- PICget_valuator(valuator)
- PICget_locator(x,y)
- PICquery_queue(event,value)
- PICflush_queue()
- PICput_mouse_playground(left,right,top,bottom)
- PICqueue_events(mode)
- PICget_event(event,value)
- PICdisplay_cursor(mode)
- PICdefine_cursor(cursor)
- PICposition_cursor(ix,iy)
- PICwait_event(event,value)
- PICget_host_screen_size(width,height)

PICattach_mouse()

The **PICattach_mouse()** function initializes the mouse and must be called before any other Input Device function.

PICattach_mouse()

PICdetach_mouse()

The PICdetach_mouse() function terminates the operation of the mouse and must be the last Input Device function called.

PICdetach_mouse()

PICget_button()

The **PICget_button()** function returns the state of a mouse button indicated by the argument *button*. If the button is *currently* pressed, the function returns a value of PIC_TRUE; if not, returns a value of PIC_FALSE.

PICget_button(button)
int button;

button = PIC_LEFTMOUSE
= PIC_RIGHTMOUSE
= PIC_MIDDLEMOUSE

PICget_valuator()

The **PICget_valuator()** function returns the current value of a valuator.

PICget_valuator(valuator)
int valuator;

valuator = PIC_XMOUSE,PIC_YMOUSE

PICget_locator()

The **PICget_locator()** function returns the current value of a locator's x and y position. The return values are stored in the locations pointed to by x and y respectively.

PICget_locator(x,y)
int *x, *y;

x,y = the x and y coordinates of the location.

PICqueue_events()

The **PICqueue_events()** function enables and/or disables the event queuing process.

PICqueue_events(mode)
int mode;

mode = PIC_ON or PIC_OFF.

PICget_event()

The **PICget_event()** function returns an event and its value. The return values are stored in the locations pointed to by *x* and *y* respectively. The **PICqueue_events()** function must be called to enable the queuing process before this function can be invoked. The possible events that can occur and their possible values are as follows:

Event	Value
PIC_LEFTMOUSE	PIC_UP PIC_DOWN
PIC_RIGHTMOUSE	PIC_UP PIC_DOWN
PIC_MIDDLEMOUSE	PIC_UP PIC_DOWN
PIC_XMOUSE	x screen coordinate
PIC_YMOUSE	y screen coordinate
PIC_KEYBOARD	keyboard event code

PICget_event(event,value)
short *event, *value;

PICdisplay_cursor()

The **PICdisplay_cursor()** function displays a cursor on the screen at a specified location.

PICdisplay_cursor(mode)
int mode;

mode = PIC_ON or PIC_OFF.

PICdefine_cursor()

The **PICput_cursor()** function defines a cursor to be displayed on the screen. The cursor is attached to the mouse input device and can be moved by moving the mouse.

The cursor is defined according to the PICcursor data structure. See Appendix B for a definition of the PICcursor structure. Once a cursor is defined, it can be displayed on the screen with the **PICdisplay_cursor()** function.

PICdefine_cursor(cursor)
PICcursor *cursor;

cursor = 32x4 byte array with a center point at *initx,inity*.

PICposition_cursor()

The **PICposition_cursor()** function positions the cursor on the screen.

PICposition_cursor(ix,iy)
int ix, iy;

ix,iy = the x and y screen coordinates of the position

PICquery_queue()

The **PICquery_queue()** function returns the state of the queue without altering the queue. The next event and value are returned in the location pointed to by *event* and *value*. A return event of *0* indicates that the queue is empty.

Event queuing must be enabled before invoking **PICquery_queue()**. To enable event queuing use the **PICqueue_events()** function.

PICquery_queue(event,value)
long *event, *value;

event = the event that occurred

value = the value associated with the event

PICwait_event()

The **PICwait_event()** function waits for a particular event to occur. The value of the *event* parameter indicates which event to wait for. A value of PIC_ANY_EVENT causes the function to return after any event occurs. The event and value of the event that occurred are stored in the location pointed to by *event* and *value* respectively.

Event queuing must be enabled before invoking **PICwait_event()**. To enable event queuing use the **PICqueue_events()** function.

PICwait_event(event,value)
long *event, *value;

event = PIC_LEFTMOUSE
= PIC_RIGHTMOUSE
= PIC_MIDDLEMOUSE
= PIC_XMOUSE
= PIC_YMOUSE
= PIC_KEYBOARD
= PIC_ANY_EVENT

value = the value of the event

PICflush_queue()

The **PICflush_queue()** function clears the event queue. Event queuing must be enabled before invoking **PICflush_queue()**. To enable event queuing use the **PICqueue_events()** function.

PICflush_queue()

PICget_host_screen_size()

The **PICget_host_screen_size()** function returns the x and y dimensions of the host screen. The width and height of the screen are stored in the locations pointed to by *width* and *height* respectively.

PICget_host_screen_size(width,height)
long *width, *height;

width = the x screen dimension in pixels

height = the y screen dimension in pixels

PICput_mouse_playground()

The **PICput_mouse_playground()** function initializes the mouse playground window. If this function is not called before the **PICattach_mouse()** function, the coordinates of the mouse playground will default to a pre-determined size and location.

PICput_mouse_playground(left,right,top,bottom)
int left, right, top, bottom;

left	=	the left x position of the playground in pixels
right	=	the right x position of the playground in pixels
top	=	the top y position of the playground in pixels
bottom	=	the bottom y position of the playground in pixels

Picking and Selecting

The **Picking and Selecting** functions enter and exit picking and selecting mode and manipulate the picking and selecting identifier stack. The identifier stack is used in picking and selecting operations.

The functions described in this section are:

- **PICattach_picking(nbuff,nstack)**
- **PICdetach_picking()**
- **PICenter_picking_mode(x,y)**
- **PICenter_selecting_mode()**
- **PICexit_picking_mode()**
- **PICexit_selecting_mode()**
- **PICinit_identifier_stack()**
- **PICpop_identifier()**
- **PICpush_identifier(id)**
- **PICput_identifier(id)**
- **PICput_picking_region(dx,dy)**

PICattach_picking()

The **PICattach_picking()** function starts the picking and selecting process, allocates space for the picking buffer and identifier stack, initializes a data structure of type PICbuffer and returns a pointer to that structure. This function must be called *before* any other Picking or Selecting function. The size of the picking/selecting buffer is specified by *nbuffer*; the size of the identifier stack is specified by *nstack*. For a definition of the PICbuffer structure, see Appendix B.

PICattach_picking(nbuffer,nstack)
int nbuffer,nstack;

nbuffer = the size of the buffer

nstack = the size of the stack

PICdetach_picking()

The **PICdetach_picking()** function terminates the picking and selecting process started by the **PICattach_picking()** function. **PICdetach_picking()** also frees the PICbuffer structure allocated by the **PICattach_picking()** function, the picking/selecting buffer, and the identifier stack. This function must be the *last* Picking or Selecting function called.

PICdetach_picking()

PICenter_picking_mode()

The **PICenter_picking_mode()** function enables picking mode. During picking mode no objects are rendered on the screen. Once picking mode is entered, if an identifier hits the picking region, the size of the identifier stack and its contents are written to the buffer. The buffer can be accessed through the PICbuffer() structure returned from a **PICattach_picking()** call.

PICenter_picking_mode() takes as arguments the coordinates of the center of the picking region. To specify the size of this region, use the **PICput_picking_region()** function described below.

NOTE: Note that atoms cannot be used as identifiers.

PICenter_picking_mode(x,y)
int x,y

x,y = the x,y location indicating the center of the picking region

PICenter_selecting_mode()

The **PICenter_selecting_mode()** function enables selecting mode. During selecting mode no objects are rendered on the screen. Once selecting mode is entered, if an identifier hits the selecting region, the size of the identifier stack and its contents are written to the buffer. The buffer can be accessed through the PICbuffer structure returned from a **PICattach_picking()** call.

The selecting region is the 3D volume specified by the current viewing projection. The viewing projection must be specified before entering selecting mode.

NOTE: Note that atoms cannot be used as identifiers.

PICenter_selecting_mode()

PICexit_picking_mode()

The **PICexit_picking_mode()** function exits picking mode. The picking/selecting buffer and the identifier stack are freed.

PICexit_picking_mode()

PICexit_selecting_mode()

The **PICexit_selecting_mode()** function exits selecting mode. The picking/selecting buffer and the identifier stack are freed.

PICexit_selecting_mode()

PICinit_identifier_stack()

The **PICinit_identifier_stack()** function initializes the identifier stack used in picking and selecting operations. This function is automatically performed when picking/selecting mode is entered, but can be used to reinitialize the identifier stack.

PICinit_identifier_stack()

PICpop_identifier()

The **PICpop_identifier()** function pops the *top* identifier from the identifier stack.

PICpop_identifier()

PICpush_identifier()

The **PICpush_identifier()** function pushes the identifier stack and places the identifier defined by the argument **id** on the *top* of the stack.

PICpush_identifier(id)
int id;

id = identifier

PICput_identifier()

The **PICput_identifier()** function replaces the top of the identifier stack with the identifier defined by the argument **id**.

PICput_identifier(id)
int id;

id = identifier

PICput_picking_region()

The **PICput_picking_region()** function sets the size of the picking region to a rectangle specified by the **dx** and **dy** arguments.

PICput_picking_region(dx,dy)
int dx,dy;

dx,dy = the size of the picking region rectangle in pixels

A Appendix A

Appendix A – Definition of Constants

Constant	Value
PIC_FALSE	0
PIC_TRUE	1
PIC_OFF	0
PIC_ON	1
PIC_ERR_OK	0
PIC_ERR_ARG	1
PIC_ERR_OPEN	2
PIC_ERR_NODE	3
PIC_ERR_FILE	4
PIC_ERR_LOAD	5
PIC_ERR_INVERSE	6
PIC_BEZIER_BASIS	0
PIC_HERMITE_BASIS	1
PIC_FOUR_POINT_BASIS	2
PIC_B_SPLINE_BASIS	3
PIC_USER_BASIS_0	0
PIC_USER_BASIS_1	1
PIC_USER_BASIS_2	2
PIC_USER_BASIS_3	3
PIC_USER_BASIS_4	4
PIC_USER_BASIS_5	5
PIC_USER_BASIS_6	6
PIC_USER_BASIS_7	7
PIC_EUCLID_POINT	1
PIC_EUCLID_LINE	2
PIC_EUCLID_POLYGON	3
PIC_EUCLID_TEXTURE	4
PIC_SCREEN_PIXELS	1280
PIC_SCREEN_LINES	1024
PIC_IMAGE_PIXELS	2048
PIC_IMAGE_LINES	2048

PIC_SINGLE_BUFFER	0
PIC_DOUBLE_BUFFER	1
PIC_BUFFER_ZERO	0
PIC_BUFFER_ONE	1
PIC_BUFFER_OVERFLOW_I	1
PIC_STACK_OVERFLOW_I	2
PIC_STACK_UNDERFLOW_I	3
#ifdef	_F77_
PIC_TOP_BUFFER	(2*16*16)
PIC_BOTTOM_BUFFER	((2*16+8)*16)
PIC_SCREEN_BUFFER	(6*16*16)
#else	
PIC_TOP_BUFFER	0x0200
PIC_BOTTOM_BUFFER	0x0280
PIC_SCREEN_BUFFER	0x0600
#endif	
PIC_LIGHT_DIRECT	1
PIC_LIGHT_POINT	2
PIC_LIGHT_SPOT	3
PIC_LIGHT_CONE	4
PIC_TYPE_ALL	-1
PIC_LIGHT_ALL	-1
PIC_BLACKOUT	PIC_OFF
PIC_SUNGLASSES	PIC_ON
PIC_SHADE_OFF	0
PIC_SHADE_FLAT	1
PIC_SHADE_GOURAUD	2
PIC_SHADE_PHONG	3
PIC_SHADE_DEPTH	4
PIC_MAX_BASIS	8
PIC_MAX_TRANSFORM	32
PIC_MAX_VIEWPORT	32
PIC_MAX_DIR_LIGHT	50
PIC_MAX_PNT_LIGHT	50
PIC_MAX_SPOT_LIGHT	50

PIC_MAX_POLY_PNTS	256
PIC_ARC_DEFAULT	64
PIC_CIRCLE_DEFAULT	64
PIC_CURVE_DEFAULT	16
PIC_QUADRIC_DEFAULT	16
PIC_PATCH_DEFAULT	16
PIC_LOW	1
PIC_MEDIUM	4
PIC_HIGH	7
PIC_TEXTURE_DEFAULT	PIC_LOW
PIC_ZMIN_DEFAULT	-1.0e+00
PIC_ZMAX_DEFAULT	0.0e+00
PIC_INTENSITY	32767.0
PIC_IINTENSITY	(1.0/PIC_INTENSITY)
PIC_VIDEO_TABLE	256
PIC_BLACK	0.0,0.0,0.0
PIC_RED	1.0,0.0,0.0
PIC_GREEN	0.0,1.0,0.0
PIC_BLUE	0.0,0.0,1.0
PIC_YELLOW	1.0,1.0,0.0
PIC_MAGENTA	1.0,0.0,1.0
PIC_CYAN	0.0,1.0,1.0
PIC_WHITE	1.0,1.0,1.0
#ifndef PIC_RGB_PIXELS	
PIC_RGB_PIXELS	11
PIC_RGB_PACKED_PIXELS	2
PIC_RGBA_PIXELS	12
PIC_RGBA_PACKED_PIXELS	1
PIC_ABGR_PACKED_PIXELS	15
PIC_RGB_ENCODED_PIXELS	14
PIC_RGB_PACKED_ENCODED_PIXELS	13
#endif	
PIC_EXTENDED_VRAM	0xf0

PIC_OVERLAY_OFF	0
PIC_OVERLAY_NON_ZERO	1
PIC_OVERLAY_HIGH_BIT	3
PIC_NO_COMPOSITE	0
PIC_A_OVER_B	1
PIC_B_OVER_A	2
PIC_A_IN_B	3
PIC_B_IN_A	4
PIC_A_OUT_B	5
PIC_B_OUT_A	6
PIC_A_ATOP_B	7
PIC_B_ATOP_A	8
PIC_A_XOR_B	9
PIC_PLUS	10
PIC_A_PLUS_B	10
PIC_SPHERE_TEMPLATE	0
PIC_UD_TEMPLATE	1
PIC_MAX_UDTEMPLATE	256
PIC_MAX_STEMPLATE	(PIC_MAX_UDTEMPLATE/2)
PIC_MAX_STAMP	19
PIC_BROADCAST_VRAM	0
PIC_BROADCAST_ZRAM	1
PIC_READ_ONLY	2
PIC_WRITE_ONLY	3
PIC_ANY_EVENT	0
PIC_LEFTMOUSE	1
PIC_MIDDLEMOUSE	2
PIC_RIGHTMOUSE	3
PIC_XMOUSE	4
PIC_YMOUSE	5
PIC_KEYBOARD	6
PIC_UP	0
PIC_DOWN	1
DEVcursor	PICcursor
PIC_PIXEL_PHONG	4

B Appendix B

Appendix B – Type Definitions

```
typedef float     PICmatrix[4][4];

typedef struct {
            int           initx, inity;
            int           bitmap[32];
        } PICcursor;


typedef struct {
            float x, y, z;
            float nx, ny, nz;
            float r, g, b;
            float exp, angle;
                 float    intensity;
                 long     samples, vertices;
                 float    *vertex;
        } PIClight_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 dissolve;
            float reflectivity;
            float refraction_index;
            float t_red, t_green, t_blue;
        } PICsurface_model;


typedef unsigned char  PICpixel;


typedef struct {
            PICpixel      red,
                          green,
                          blue;
        } PICrgb_pixel;


typedef struct {
            PICpixel      red,
```

```
                        green,
                        blue,
                        alpha;
        } PICrgba_pixel;


typedef struct {
            PICpixel    alpha,
                        blue,
                        green,
                        red;
        } PICabgr_pixel;


typedef struct {
            int         *buffer;
            int         *nused;
            int         *buffer_overflow;
            int         *stack_overflow;
            int         *stack_underflow;
        } PICbuffer;


#define PIC_RASTER_DISPATCH   256

typedef struct {
            short       magic;  /* Magic number VFONT_MAGIC */
            unsigned short  size;/* Total # bytes of bitmaps */
            short       maxx;   /* Maximum horizontal glyph size */
            short       maxy;   /* Maximum vertical glyph size */
            short       xtend;  /* (unused) */
        } raster_header;


typedef struct {
            unsigned short  addr[PIC_RASTER_DISPATCH];
            short   nbytes[PIC_RASTER_DISPATCH];
            char    *data;
        } raster_font;


typedef struct {
            raster_header   header;
            short   twobytes;    /* For aligning (char*)data below */
```

```
        raster_font        font;
    } PICraster_font;


#define PIC_VECTOR_FONT_SIZE    95

typedef struct {
        short   mm;
        short   pt;
        short   Lw;
        short   Rw;
    } vector_font;


typedef struct {
        vector_font        ptr[PIC_VECTOR_FONT_SIZE];
        char     *hshstr;
    } PICvector_font;

typedef struct
    {
    short   type;       /*    sphere or user defined            */
    int     ix;         /*    template position in vram in x    */
    int     iy;         /*    template position in vram in y    */
    int     size;       /*    size of template in pixels        */
    float   radius;     /*    size of radius (spheres only)     */
    } PICtemplate;
```

C Appendix C

Appendix C – Function Description

FUNCTION	DESCRIPTION
PICalpha (3)	- enable/disable writing to the alpha channel
PICantialias_lines (3)	- enable/disable the antialiasing of lines
PICarc (3)	- draw a circular arc
PICarc_precision (3)	- set precision of arc
PICatom (3)	- draw a spherical atom
PICatom_light (3)	- specify a light source for a spherical atom
PICatom_surface (3)	- specify a surface model for a spherical atom
PICattach_mouse (3)	- attach a mouse
PICattach_picking (3)	- start picking/selecting process
PICbackface (3)	- enable/disable backface removal mode
PICbroadcast_data (3)	- broadcast a buffer of data to pixel-node memories
PICcamera_view (3)	- define a viewing transformation in terms of pan, tilt, and swing angles
PICcircle (3)	- draw a circle
PICcircle_precision (3)	- set precision of circle
PICclear_alpha (3)	- clear the alpha channel of current viewport
PICclear_rgb (3)	- clear the rgb channels of current viewport
PICclear_rgbz (3)	- clear rgb and z depth of current viewport
PICclear_z (3)	- clear z depth of current viewport
PICclockwise (3)	- enable/disable normal vector definition in clockwise direction
PICcolor_alpha (3)	- define the current alpha color
PICcolor_rgb (3)	- define the current rgb color
PICcomposite_mode (3)	- set the current image compositing mode
PICcopy_back_to_ext (3)	- copy the back buffer to an extended screen buffer
PICcopy_back_to_front (3)	- copy back buffer to front buffer
PICcopy_ext_to_back (3)	- copy an extended screen buffer to the back buffer
PICcopy_ext_to_z (3)	- copy extended screen z buffer to screen z buffer
PICcopy_front_to_back (3)	- copy front buffer to back buffer
PICcopy_z_to_ext (3)	- copy z buffer to the extended z buffer
PICcurve_geometry_3d (3)	- draw a 3D curve
PICcurve_precision (3)	- set precision of curve
PICdefine_cursor (3)	- define the current cursor
PICdepth_cue (3)	- enable/disable depth cueing
PICdepth_cue_limits (3)	- set z limits and color range of depth cueing
PICdetach_mouse (3)	- terminate mouse process
PICdetach_picking (3)	- terminate picking/selecting process
PICdisplay_cursor (3)	- enable/disable cursor display
PICdisplay_overlay (3)	- enable/disable display of the alpha channel
PICdouble_buffer (3)	- enable/disable double buffer mode
PICdraw (3)	- draw a line
PICdsp_float (3)	- enable/disable DSP32 floating point format
PICenter_picking_mode (3)	- enter picking mode

FUNCTION	DESCRIPTION
PICenter_sampling_pass (3)	- start a super-sampling pass
PICenter_selecting_mode (3)	- enter selecting mode
PICeuclid_mode (3)	- set drawing mode
PICexit (3)	- exit the PIClib library
PICexit_picking_mode (3)	- exit picking mode
PICexit_sampling_pass (3)	- end a super-sampling pass
PICexit_selecting_mode (3)	- exit selecting mode
PICflip (3)	- enable/disable normal vector reversal
PICflush_queue (3)	- flush event queue
PICget_alpha_map (3)	- get current rgb entries from alpha map
PICget_alpha_map_entry (3)	- get a specified rgb alpha map entry
PICget_buffer (3)	- get the number of the current display buffer
PICget_buffer_mode (3)	- get buffer mode (single or double)
PICget_button (3)	- query current state of button
PICget_color_map (3)	- get current rgb color map
PICget_color_map_entry (3)	- get one rgb color map entry
PICget_depth (3)	- get the near and far depth limits
PICget_event (3)	- return an event and its value
PICget_host_screen_size (3)	- returns the dimensions of the host screen
PICget_image_header (3)	- read the Pixel Machine image header from a file
PICget_inverse_project (3)	- get the inverse of the current projection matrix
PICget_inverse_transform (3)	- get the inverse of the current transformation matrix
PICget_locator (3)	- query the current value of a locator
PICget_normal_transform (3)	- get normal vector transformation matrix
PICget_overlay_mode (3)	- get the pixel node overlay
PICget_project (3)	- get the current projection matrix
PICget_scan_line (3)	- read a scan line of pixels from the screen
PICget_screen_size (3)	- return the current screen size
PICget_shade_mode (3)	- return current shading mode
PICget_template (3)	- get a previously defined template
PICget_transform (3)	- get the current transformation matrix
PICget_valuator (3)	- returns the current value of a valuator
PICget_viewport (3)	- return the current viewport's definition
PICimage_header (4)	- format of a Pixel Machine image header file
PICinit (3)	- initialize and reset the PIClib library
PICinit_identifier_stack (3)	- initialize identifier stack
PICinit_sampling (3)	- initialize super-sampling mode
PIClight_ambient (3)	- set the ambient light's intensity value
PIClight_switch (3)	- turn all or one light source on or off
PIClookat_view (3)	- define a viewing transformation in terms of viewpoint, reference point and twist angle
PIClookup_view (3)	- define a viewing transformation in terms of viewpoint, reference point and twist angle

FUNCTION	DESCRIPTION
PICmake_sphere_template (3)	- create a sphere template
PICmake_template (3)	- create a user defined template
PICmove (3)	- move to a given point
PICopen_raster_font (3)	- select a raster font type
PICopen_vector_font (3)	- select a vector font type
PICortho_project (3)	- define an orthographic projection
PICoverlay_mode (3)	- select overlay mode to display the alpha channel
PICpatch_geometry_3d (3)	- draw a 3D surface patch
PICpatch_precision (3)	- set precision of patch
PICpercent_texture (3)	- determines the texture map's intensity value at each pixel
PICpersp_project (3)	- define a 3D perspective viewing pyramid
PICpixel_add (3)	- add a constant value to pixels in current viewport
PICpixel_multiply (3)	- multiply pixels in the current viewport by a constant
PICpoint (3)	- draw a point
PICpolar_view (3)	- define view point and view direction in Polar Coordinates
PICpoly_close (3)	- close a polygon
PICpoly_normal (3)	- define a polygon normal vector
PICpoly_point (3)	- draw a polygon
PICpoly_point_nv (3)	- draw a 3D polygon with normal vectors
PICpoly_point_nv_uv (3)	- draw a 3D polygon with normal vectors and texture indices
PICpoly_point_rgb (3)	- draw a 3D polygon with rgb color
PICpoly_point_uv (3)	- render a 3D polygon with texture indices
PICpop_identifier (3)	- pop top identifier from identifier stack
PICpop_project (3)	- replace the current projection matrix with the top of the projection stack
PICpop_transform (3)	- replace the current transformation matrix with the top of the transformation stack
PICpop_viewport (3)	- replace the current viewport with the top of the viewport stack
PICposition_cursor (3)	- position cursor (in Pixel Coordinates)
PICpostmultiply_project (3)	- post-multiply the current projection matrix by a specified matrix
PICpostmultiply_transform (3)	- post-multiply the current transformation matrix by a specified matrix
PICpremultiply_project (3)	- pre-multiply the current projection matrix by a specified matrix
PICpremultiply_transform (3)	- pre-multiply the current transformation matrix by a specified matrix
PICpush_identifier (3)	- push an identifier on identifier stack
PICpush_project (3)	- copy the current projection matrix onto the top of the projection stack
PICpush_transform (3)	- push transformation matrix onto the top of the transformtion stack
PICpush_viewport (3)	- copy the current viewport onto the top of the viewport stack
PICput_alpha_map (3)	- load the entire lookup table for the alpha channel
PICput_alpha_map_entry (3)	- load a single entry in the lookup table for the alpha channel
PICput_basis (3)	- define a basis matrix
PICput_color_map (3)	- load entire lookup table for each rgb channel
PICput_color_map_entry (3)	- load a specific entry into the rgb color map
PICput_depth (3)	- set the near and far depth limits

FUNCTION	DESCRIPTION
PICput_identifier (3)	- replace the top of identifier stack
PICput_identity_transform (3)	- load the current transformation matrix with the identity matrix
PICput_image_header (3)	- write a Pixel Machine image header to a file
PICput_light_source (3)	- load a light source
PICput_mouse_playground (3)	- initializes mouse playground window
PICput_overlay_mode (3)	- load overlay code into the pixel nodes
PICput_picking_region (3)	- set the size of picking region
PICput_project (3)	- load current projection transformation matrix with a specified matrix
PICput_raster_font (3)	- set the current raster font
PICput_scan_line (3)	- write a scan line of pixels to the screen
PICput_surface_model (3)	- specify a surface shading model
PICput_texture (3)	- define an area of offscreen memory to be used as a single texture
PICput_transform (3)	- load the current transformation matrix with a specified matrix
PICput_vector_font (3)	- set the current vector font
PICput_viewport (3)	- set the current viewport
PICquadric_precision (3)	- set precision of quadrics and superquadrics
PICquery_queue (3)	- query event queue
PICqueue_events (3)	- enable/disable event queueing
PICraster_font_text (3)	- write a text string using the specified raster font
PICraster_text (3)	- write a text string using the current raster font
PICrectangle (3)	- draw a rectangle
PICreset (3)	- reset graphical parameters to default values
PICreset_texture (3)	- set the current area to be used for texture mapping with the default values
PICresume (3)	- initialize the PIClib library
PICrotate (3)	- apply a rotation transformation to all objects
PICscale (3)	- apply a scaling transformation to all objects
PICselect_curve_basis (3)	- select the basis matrix used in drawing curves
PICselect_patch_basis (3)	- select the basis matrix to be used in drawing patches
PICset_texture (3)	- set the current texture mapping area
PICshade_mode (3)	- select a shading mode
PICsphere (3)	- draw a unit sphere
PICstamp_template (3)	- stamp a buffer of templates on the screen
PICsuperq_ellipsoid (3)	- draw a superquadric ellipsoid
PICsuperq_hyper1 (3)	- draw a superquadric hyperboloid of 1 sheet
PICsuperq_hyper2 (3)	- draw a superquadric hyperboloid of 2 sheets
PICsuperq_toroid (3)	- draw a superquadric toroid
PICswap_buffer (3)	- swap displayable buffers
PICswap_pipe (3)	- swaps Transformation Pipes in dual-pipe configurations
PICtexture_precision (3)	- set texture map precision of textured perspective polygons
PICtranslate (3)	- apply a translation transformation to all objects
PICupdate_map (3)	- enable/disable updating of video lookup tables

FUNCTION	DESCRIPTION
PICvector_font_text (3)	- write a text string using the specified vector font
PICvector_text (3)	- write a text string using current vector font
PICwait_event (3)	- wait for a specifed event to occur
PICwait_psync (3)	- wait for Pixel Node processor sync
PICwait_vsync (3)	- wait for a vertical sync
PICwindow_project (3)	- define a 3D perspective projection
PICzbuffer (3)	- enable/disable zbuffer mode
PICzbuffer_lines (3)	- enables/disables zbuffering of lines
picalpha (1)	- turn the display of the alpha channel on or off
picbars (1)	- display color bars on the screen
picboot (1)	- load the PIClib modules into the geometry and drawing nodes
picbroadv (1)	- broadcasts a buffer of data to the pixel node memory
picbroadz (1)	- broadcasts a buffer of data to the pixel node memory
picbtof (1)	- copy contents of the back buffer to the front buffer
picdisp (1)	- download and/or display an image
picetof (1)	- copy the contents of the extended VRAM buffer to the front buffer
picftob (1)	- copy the contents of the front buffer to the back buffer
picftoe (1)	- copy the contents of the front buffer to extended VRAM
picgamma (1)	- create gamma corrected lookup tables
picinit (1)	- resets the Pixel Machine to its default values
piclear (1)	- clear the screen
piclens (1)	- interactive tool that roams around and magnifies the display
picsave (1)	- save an image to disk
pictexture (1)	- display current texture loaded into VRAM