Polyray Grammar

What follows is the complete YACC grammar used to parse Polyray input files. Only the actions taken at each rule have been deleted.   The conventions used in the grammar are: keywords (terminal symbols like "sphere") appear in all caps, i.e. SPHERE. All other grammar rules (nonterminals) appear in lower case.   The terminal symbols, including punctuation, are either recognized by the lexical analyzer or by lookup from one of the symbol tables.

Note: there is one ambiguity present in the grammar - the "if .. if .. else" construct.

B

background
bezier
bezier_points
blob
blobelement
blobelements
box

C

camera
camera_exper
camera_expers
conditional
cone
csg
csg_tree
cylinder

D

defined_token
definition
disc

E

element
elementlist
end_frame_decl
expression
expression_list

F

fexper
flush_statement
frame_decl
function

G

gridded

H

height_field

scene

```
scene
  : elementlist
  ;
```

elementlist

```
elementlist
    : elementlist element
    | element
    ;
```

element

```
element
    : background
    | camera
    | definition
    | flush_statement
    | frame_decl
    | if_statement
    | light
    | object
    | outfile
    | system_call
    ;
```

defined_token

```
defined_token
    : SURFACE_SYM
    | TEXTURE_SYM
    | OBJECT_SYM
    | EXPRESSION_SYM
    | TRANSFORM_SYM
    ;
```

definition

```
definition
    : DEFINE defined_token surface
    | DEFINE defined_token texture
    | DEFINE defined_token object
    | DEFINE defined_token transform
    | DEFINE defined_token expression
    | DEFINE TOKEN surface
    | DEFINE TOKEN texture
    | DEFINE TOKEN object
    | DEFINE TOKEN transform
    | DEFINE TOKEN expression
    ;
```

object

```
object
    : OBJECT '{' object_decls '}'
    | OBJECT_SYM
    | OBJECT_SYM '{' object_modifier_decls '}'
    ;
```

object_modifier_decls

```
object_modifier_decls
    : object_modifier_decl object_modifier_decls
    | object_modifier_decl
    ;
```

object_modifier_decl

```
object_modifier_decl
    : texture
    | transform
    | DITHER fexper
    | ROTATE point
    | ROTATE point ',' fexper
    | SHEAR fexper ',' fexper ',' fexper ',' fexper ','
            fexper ',' fexper
    | TRANSLATE point
    | SCALE point
    | U_STEPS fexper
    | V_STEPS fexper
    | SHADING_FLAGS fexper
    | BOUNDING_BOX point ',' point
    | ROOT_SOLVER FERRARI
    | ROOT_SOLVER VIETA
    | ROOT_SOLVER STURM
    ;
```

object_decls

```
object_decls
   : shape_decl
   | shape_decl object_modifier_decls
   ;
```

shape_decl

```
shape_decl
    : bezier
    | blob
    | box
    | cone
    | cylinder
    | csg
    | disc
    | function
    | gridded
    | height_field
    | height_fn
    | lathe
    | parabola
    | polygon
    | polynomial
    | ppatch
    | smooth_height_field
    | smooth_height_fn
    | sphere
    | sweep
    | torus
    ;
```

camera_exper

```
camera_exper
    : ANGLE fexper
    | APERTURE fexper
    | AT point
    | ASPECT fexper
    | MAX_TRACE_DEPTH fexper
    | DITHER_RAYS fexper
    | DITHER_OBJECTS fexper
    | FOCAL_DISTANCE fexper
    | FROM point
    | HITHER fexper
    | YON fexper
    | RESOLUTION fexper ',' fexper
    | UP point
    ;
```

camera_expers

```
camera_expers
    : camera_expers camera_exper
    | camera_exper
    ;
```

camera

```
camera
   : VIEWPOINT '{' camera_expers '}'
   ;
```

light_modifier_decl

```
light_modifier_decl
   : COLOR expression
   | transform
   | ROTATE point
   | ROTATE point ',' fexper
   | SHEAR fexper ',' fexper ',' fexper ',' fexper ','
         fexper ',' fexper
   | TRANSLATE point
   | SCALE point
   ;
```

light_modifier_decls

```
light_modifier_decls
    : light_modifier_decl light_modifier_decls
    |
    ;
```

light

```
light
    : LIGHT point ',' point
    | LIGHT point
    | SPOT_LIGHT point ',' point
    | SPOT_LIGHT point ',' point ',' point ',' fexper ','
             fexper ',' fexper
    | TEXTURED_LIGHT '{' light_modifier_decls '}'
    ;
```

background

```
background
    : BACKGROUND expression
    ;
```

surface_declaration

```
surface_declaration
    : COLOR expression
    | COLOR_MAP '(' map_entries ',' expression ')'
    | COLOR_MAP '(' map_entries ')'
    | AMBIENT expression ',' expression
    | AMBIENT expression
    | BUMP_SCALE expression
    | DIFFUSE expression ',' expression
    | DIFFUSE expression
    | FREQUENCY expression
    | LOOKUP_FUNCTION expression
    | MICROFACET PHONG expression
    | MICROFACET BLINN expression
    | MICROFACET GAUSSIAN expression
    | MICROFACET REITZ expression
    | MICROFACET COOK expression
    | MICROFACET expression
    | NORMAL expression
    | OCTAVES expression
    | PHASE expression
    | POSITION_FUNCTION expression
    | POSITION_SCALE expression
    | REFLECTION expression ',' expression
    | REFLECTION expression
    | SPECULAR expression ',' expression
    | SPECULAR expression
    | TRANSMISSION expression ',' expression ',' expression
    | TRANSMISSION expression ',' expression
    | TURBULENCE expression
    ;
```

surface_declarations

```
surface_declarations
    : surface_declaration surface_declarations
    |
    ;
```

surface

```
surface
    : SURFACE '{' surface_declarations '}'
    | SURFACE_SYM
    | SURFACE_SYM '{' surface_declarations '}'
    ;
```

texture_modifier_decls

```
texture_modifier_decls
    : texture_modifier_decl texture_modifier_decls
    | texture_modifier_decl
    ;
```

texture_modifier_decl

```
texture_modifier_decl
    : transform
    | ROTATE point
    | ROTATE point ',' fexper
    | SHEAR fexper ',' fexper ',' fexper ',' fexper ','
            fexper ',' fexper
    | TRANSLATE point
    | SCALE point
    ;
```

texture_declarations

```
texture_declarations
    : texture_declaration texture_modifier_decls
    | texture_declaration
    ;
```

texture_declaration

```
texture_declaration
    : surface
    | SPECIAL surface
    | NOISE surface
    | CHECKER texture ',' texture
    | HEXAGON texture ',' texture ',' texture
    | LAYERED texture_list
    ;
```

texture

```
texture
    : TEXTURE '{' texture_declarations '}'
    | TEXTURE_SYM
    | TEXTURE_SYM '{' texture_modifier_decls '}'
    ;
```

texture_list

```
texture_list
    : texture
    | texture_list ',' texture
    ;
```

transform_declaration

```
transform_declaration
   : ROTATE point
   | ROTATE point ',' fexper
   | SCALE point
   | TRANSLATE point
   ;
```

transform_declarations

```
transform_declarations
    : transform_declaration
    | transform_declarations transform_declaration
    ;
```

transform

```
transform
    : TRANSFORM '{'
      transform_declarations '}'
    | TRANSFORM_SYM
    | TRANSFORM_SYM '{' transform_declarations '}'
    ;
```

bezier_points

```
bezier_points
    : bezier_points ',' point
    | point
    ;
```

bezier

```
bezier
    : BEZIER fexper ',' fexper ',' fexper ',' fexper ','
bezier_points
    ;
```

blob

```
blob
   : BLOB fexper ':' blobelements
   ;
```

blobelements

```
blobelements
    : blobelement
    | blobelements ',' blobelement
    ;
```

blobelement

```
blobelement
    : fexper ',' fexper ',' point
    | SPHERE point ',' fexper ',' fexper
    | CYLINDER point ',' point ',' fexper ',' fexper
    | PLANE point ',' fexper ',' fexper ',' fexper
    | TORUS point ',' point ',' fexper ',' fexper ',' fexper
    ;
```

box

```
box
    : BOX point ',' point
    ;
```

cone

```
cone
    : CONE point ',' fexper ',' point ',' fexper
    ;
```

csg

```
csg
    : csg_tree
    ;
```

csg_tree

```
csg_tree
    : '(' csg_tree ')'
    | csg_tree '+' csg_tree
    | csg_tree '-' csg_tree
    | csg_tree '*' csg_tree
    | '~' csg_tree
    | csg_tree '&' csg_tree
    | object
    ;
```

cylinder

```
cylinder
    : CYLINDER point ',' point ',' fexper
    ;
```

disc

```
disc
   : DISC point ',' point ',' fexper
   | DISC point ',' point ',' fexper ',' fexper
   ;
```

function

```
function
   : FUNCTION expression
   ;
```

gridded

```
gridded
   : GRIDDED sexper ',' object_list
   ;
```

object_list

```
object_list
    : object
    | object object_list
    ;
```

height_field

```
height_field
    : HEIGHT_FIELD sexper
    ;
```

height_fn

```
height_fn
    : HEIGHT_FN fexper ',' fexper ','
              fexper ',' fexper ',' fexper ',' fexper ','
              expression
    | HEIGHT_FN fexper ',' fexper ',' expression
    ;
```

lathe

```
lathe
   : LATHE fexper ',' point ',' fexper  ',' pointlist
   ;
```

parabola

```
parabola
    : PARABOLA point ',' point ',' fexper
    ;
```

polygon

```
polygon:
    POLYGON fexper  ',' pointlist
    ;
```

polynomial

```
polynomial
    : POLYNOMIAL expression
    ;
```

ppatch

```
ppatch
    : PATCH point ',' point ',' point ',' point ',' point ','
point
    ;
```

smooth_height_field

```
smooth_height_field
   : SMOOTH_HEIGHT_FIELD sexper
   ;
```

smooth_height_fn

```
smooth_height_fn
    : SMOOTH_HEIGHT_FN fexper ',' fexper ','
                fexper ',' fexper ',' fexper ',' fexper ','
                expression
    | SMOOTH_HEIGHT_FN fexper ',' fexper ',' expression
    ;
```

sphere

```
sphere
    : SPHERE point ',' fexper
    ;
```

sweep

```
sweep
   : SWEEP fexper ',' point ',' fexper  ',' pointlist
   ;
```

torus

```
torus
    : TORUS fexper ',' fexper ',' point ',' point
    ;
```

fexper

```
fexper
    : expression
    ;
```

point

```
point
    : expression
    ;
```

sexper

```
sexper
   : expression
   ;
```

pointlist

```
pointlist
    : point
    | pointlist ',' point
    ;
```

expression

```
expression
    : '(' expression ')'
    | '[' expression_list ']'
    | '<' expression ',' expression '>'
    | '<' expression ',' expression ',' expression '>'
    | expression '[' expression ']'
    | '(' conditional '?' expression ':' expression ')'
    | expression '^' expression
    | expression '%' expression
    | expression '*' expression
    | expression '.' expression
    | expression '/' expression
    | expression '+' expression
    | expression '-' expression
    | '-' expression %prec UMINUS
    | '|' expression '|'
    | COLOR_MAP '(' map_entries ',' expression ')'
    | COLOR_MAP '(' map_entries ')'
    | NOISE '(' expression ')'
    | NOISE '(' expression ',' expression ')'
    | ROTATE '(' expression ',' expression ')'
    | ROTATE '(' expression ',' expression ',' expression ')'
    | END_FRAME
    | START_FRAME
    | TOTAL_FRAMES
    | TOKEN '(' expression_list ')'
    | TOKEN
    | NUM
    | STRING
    | EXPRESSION_SYM
    ;
```

expression_list

```
expression_list
    : expression
    | expression ',' expression_list
    ;
```

conditional

```
conditional
    : '(' conditional ')'
    | expression '<' expression
    | expression '>' expression
    | expression LTEQ_SYM expression
    | expression GTEQ_SYM expression
    | expression EQUAL_SYM expression
    | conditional AND_SYM conditional
    | conditional OR_SYM conditional
    | '!' conditional
    ;
```

map_entry

```
map_entry
   : '[' fexper ',' fexper ',' point ',' point ']'
   | '[' fexper ',' fexper ',' point ',' fexper ',' point ','
fexper ']'
   ;
```

map_entries

```
map_entries
    : map_entry map_entries
    | map_entry
    ;
```

frame_decl

```
frame_decl
    : end_frame_decl
    | start_frame_decl
    | total_frames_decl
    ;
```

end_frame_decl

```
end_frame_decl
   : END_FRAME fexper
   ;
```

start_frame_decl

```
start_frame_decl
    : START_FRAME fexper
    ;
```

total_frames_decl

```
total_frames_decl
    : TOTAL_FRAMES fexper
    ;
```

outfile

```
outfile
    : OUTFILE TOKEN
    | OUTFILE STRING
    ;
```

flush_statement

```
flush_statement
    : FILE_FLUSH fexper
    ;
```

system_call

```
system_call
  : SYSTEM '(' expression_list ')'
  ;
```

statement

```
statement
    : '{' elementlist '}'
    | element
    ;
```

if_else_part

```
if_else_part
   : ELSE statement
   |
   ;
```

if_statement

```
if_statement
    : IF '(' conditional ')' statement if_else_part
    ;
```

ÿ   `;`

What's New in Version 1.7

Changes:

● Added parametric surfaces. Now possible to define a mesh object (surface) where each point in the mesh is a function
   of u and v.

● Added bump maps

● Added noeval to definitions to cure a particle system bug

● Added support for GIF and JPEG images in textures, etc.

● Added support for system calls in 386 version. Can now call an external program from within Polyray.

● The default shading flags for raytracing is now set to:

```
SHADOW_CHECK + REFLECT_CHECK + TRANSMIT_CHECK +UV_CHECK +
CAST_SHADOW
```

The difference is that it is no longer assumed that surfaces should be lit on both sides (normal correction), this boosts rendering speed at the cost of funny shading once in a while. UV_CHECK was added to allow turning off checking of u/v bounds on objects (also a speedup).

● Changed u_steps, v_steps back to the way it was in v1.5. Now for all objects it is uniformly subdivided by exactly the value of u_steps/v_steps.

● Significantly enhanced the function object. Polyray will now accept any predefined function as part of the function definition.

● Added NURBS objects. (Sorry, no trim curves yet.)

Close   Displacement functions are now possible on CSG objects. They also work in raytracing now (although you may need to really boost the u_steps and v_steps values to get good results).

Close   Now supports /* ... */ comments (C style). No you can't nest them. You can nest the single line // comments within them.

Close   Internal modifications made that should result in less memory usage by objects. Your mileage may vary.

Close   Modified numeric input to allow numbers like 1., .1

Close   Fixed up bug in polygon tracing that caused them to drop out if they were oriented in just the right way.

Close   Added a second raw triangle output format - only outputs the vertex coordinates. This makes it more compatible with RAW2POV. Previous style with normals and u/v still available.

Close   Removed BSP tree bounding. It wasn't any faster than slabs and locked up every once in a while.

Close   Added a glyph object to support TrueType style extruded surfaces. Glyphs are always

closed at the top and bottom, they may have both straight and curved sides in the same shape.   A program to extract TrueType information and write into Polyray glyph format is included with the data file archive.

Close   Support for simple particle systems.   Can define birth and death conditions, # of objects to generate, and ability to bounce off other objects.

Close   Added many more VESA display modes.   There are now 5 SVGA 256 color modes, 5 Hicolor modes, and 5 truecolor modes supported.   If your board doesn't support a selected mode, Polyray tries a lower res one having the same # of bytes per pixel.

Polyray v1.7 Help Contents

About Polyray Help

This Help file describes the input format and capabilities of the *Polyray* raytracing program, and gives an overview of the basic commands, functions and syntax of *Polyray.*

Close   If you find yourself deep inside this document without a clue as to your whereabouts, don't worry!   Just look for this **Up** button in the non-scrolling region at the top of the screen.   It steps you up one level at a time until you're back at one of the following main topics...

Topics:

Introduction & Shareware Information

Detailed Description of the Polyray Input Format

File Formats

Algorithms

Sample Files

Polyray Grammar

What's New in Version 1.7

Bibliography

Quick Reference

That's the Idea!

Unfortunately, this is only a demo button.
The real **Up** button will be found at the top of
the screen in the gray non-scrolling regions
of nested topics...

`Topics`

Introduction & Shareware Information

The program "Polyray" is a rendering program for producing scenes of 3D shapes and surfaces.   The means of description range from standard primitives like box, sphere, etc. to 3 variable polynomial expressions, and finally (and slowest of all) surfaces containing transcendental functions like sin, cos, log.   The files associated with Polyray are distributed in three pieces: the executable, a collection of document files, and a collection of data files.

Since version 1.5, Polyray has been a Shareware program, rather than Freeware.   If you enjoy this program, use it frequently, and can afford to pay a registration fee, then send $35 to:

> **Alexander Enzmann**
> **20 Clinton St.**
> **Woburn, Ma 01801**
> **USA**

Please include your name and mailing address.

If you formally register this program, you will receive free the next release of Polyray, when it occurs. In addition you will be contributing to my ability to purchase software tools to make Polyray a better program.   If you don't register this program, don't feel bad - I'm poor too - but you also shouldn't expect as prompt a response to questions or bugs.   Consider it guilt-free Shareware.   Note that all of the sample files in PLYDAT are Public Domain, you may use them freely.   Note that the Polyray executables and the Polyray documents (including this document) are copyrighted.

If you want a good reference book about raytracing, buy the book: "Introduction to Ray Tracing", edited by Andrew Glassner, Academic Press, 1989.   That book is excellent and will provide many of the background details that are not contained here.   For an abbreviated list of Polyray's syntax, see the topic Quick Reference.

This document contains a step by step description of a simple scene file to get you familiar with the structure of Polyray data.   By working through the examples and by reviewing the examples in the data archive, you will be able to see how the various features are used.

It is assumed that you are familiar with files compressed with PKZIP (version 2.04 is required) and setting the PATH variable to make an executable visible to DOS.   If you aren't, then get some help from someone who has used ZIP and is familiar with configuring a DOS system.

The data files are ASCII text, the output image file format (see Output Files for the supported input and output formats) supported is Targa.   Input images (for texturing, height fields, etc.) may be: Targa (all variants defined in the Truevision spec), GIF (both 87a and 89a should work), or JPEG (JFIF format).   The Targa format is supported by many image processing programs so if you need to translate between Targa and something else it is quite simple.   The utility CJPEG, which converts from Targa to JPEG has been included in the utility archive.   Polyray is case sensitive, so the following ways of writing foo are all considered different: Foo, FOO, and foo.   For an abbreviated list of Polyray's syntax, see the Quick Reference.

Polyray supports a number of VESA compliant display modes from the standard 320x200 VGA mode through to 24 bits at high resolution.   If you don't have a VESA compliant graphics board you will either be limited to standard VGA or you will need to find a VESA driver for your board.

The standard executable requires an IBM PC compatible with at least a 386 and 387, and a minimum of 2 Mbytes of RAM to run.   Other memory models will be made available if enough requests are made.   The distributed executable uses a DOS extender in order to grab as much memory as is available.   It has been successfully run with HIMEM.SYS, QEMM386.SYS, and 386MAX.SYS. Under Windows it will run in a DOS window, however if you use the graphics display it will need to be run full screen.

There have been a number of problems reported of conflicts between the DOS extender used in Polyray and memory managers.   These all involve DOS 6.0 or later.   In general the conflict results in an error message along the lines of:

Previously loaded software is neither VCPI or DPMI compliant

If you see this message, or one like it, you may need to disable EMS handling from you CONFIG.SYS file.   A typical change is:

```
EMM386 NOEMS
```

For some people, just running Polyray in a DOS box under Windows is sufficient to eliminate memory manager problems.   As a first step in resolving this type of problem, try setting up a CONFIG.SYS and AUTOEXEC.BAT with as few entries as possible.

I'm interested in any comments/bug reports.   I can be reached via email by:

CompuServe as: Alexander Enzmann 70323,2461
Internet as: xander@mitre.org

or via snailmail at:

Alexander Enzmann
20 Clinton St.
Woburn, Ma 01801
USA

Topics

Acknowledgments and Other Legal Stuff

There is no warranty, explicit or implied, of the suitability of this software for any purpose.   This software is provided as is, and the user of this software assumes the entire risk as to its quality, accuracy, and for any damages resulting from the use of this software.

No part of this package may be included as part of a commercial package without explicit written permission.

This original code for this program was based on the "mtv" ray - tracer, written (and placed in the public domain) by Mark VandeWettering.

This software is based in part on the work of the Independent JPEG Group.
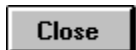
The Graphics Interchange Format © is the copyright property of CompuServe Incorporated.   The service marks Graphics Interchange Format(sm), and GIF(sm) are owned by CompuServe Incorporated.

Thanks to Rob McGregor at Screaming Tiki (tm) for this Help file.   Many thanks go to David Mason for his numerous comments and suggestions (and for adding a new feature to DTA every time I needed to test something).   Thanks also to the Cafe Pamplona in Cambridge Ma. for providing a place to get heavily caffeinated and rap about ray tracing and image processing for hours at a time.

Kudos go to Jeff Bowermaster, Alfonso Hermida, and Dan Richardson for extensive beta testing of Polyray and their many suggestions for improvements.   Additionally, many thanks to the folks that let me include some of their scenes in the data archive: Jeff Bowermaster, Dan Farmer, and Will Wagner.
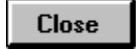
[Up] Useful Tools

There are several types of tools that you will want to have to work with this program (in relative order of importance):

[Close] An ASCII text editor for preparing and modifying input files.

[Close] A picture viewer for viewing images.

[Close] An image processing program for translation between Targa and some other format.

[Close] An animation generator that will take a series of Targa images and produce an animation file.

[Close] An animation viewer for playing an animation on the screen.

There are a number of tools out there that can be used in conjunction with Polyray.   I've somewhat arbitrarily divided them into categories, with no particular order of importance.   I'm pretty sure all of the following are available from CompuServe in either the GRAPHDEV or GRAPHSUP forums.

Animation Utilities

Many of the features in Polyray are specifically to support the generation of multiple frames.   Once the frames are generated, additional tools are necessary to format and play the resulting animation. Top picks in this area are:

[Close] DTA (David Mason)

[Close] DFV (David Mason)

[Close] PLAY (Trilobyte)

Dave's Targa Animator (DTA) may be the only tool you need for creating FLI/FLCs from images, converting images between formats, compositing images, etc.   For building animations with Polyray, all you need is DTA and a FLI/FLC player.

DFV and PLAY80 do a single thing.   They play FLI/FLC animations on a VGA or SVGA screen. Either one will do the job for 8 bit FLI and FLC animations.   However, DFV can handle a few formats that aren't supported by PLAY (in particular the 16 and 24 bit animation formats produced by DTA).

Modelers

There are a couple of really good Shareware modelers out there.   Both of them allow the creation of objects in native form (rather than as hundreds of polygons), and provide fast, interactive, development of scenes using wireframe displays.   The modelers are:

[Close] POVCAD   (Alfonso Hermida and Rob McGregor)

[Close] MORAY   (Lutz Kretzschmar)

POVCAD is a graphical modeling program with both Windows and DOS versions.   POVCAD allows you to quickly build models through point and click in a wireframe environment.   Both Polyray and POV-Ray data files can be generated.   A full featured text editor with a one-click rendering interface is included.

Actually, MORAY is a DOS modeler for POV-Ray, but it is mentioned here since it is a very good shareware modeler.   Perhaps if enough people ask Lutz to add Polyray support...

Miscellaneous Programs

[Close] CTDS   (Truman Brown)

[Close] RAW2POV   (Steve Anger)

[Close] 3DS2POV   (Steve Anger)

[Close] SPD   (Eric Haines, updates by Eduard Schwan & Alexander Enzmann)

CTDS is dot-to-dot for renderers.   By creating a file listing lots of points and associated radii, this program will connect them together using spheres and cones.   It can output to the native input of several renderers, including: Polyray, POV-Ray, and Vivid.   This programs really fun for creating abstract shapes.

RAW2POV is a conversion program that takes ASCII files of triangle vertices and creates an input for various renderers.   It can also examine the triangles and calculate normals for the vertices, giving a very smooth looking object when rendered.   This is a must have utility if you are going to be converting polygonal objects into a form that can be used by Polyray.

3DS2POV converts model files built by the costly 3DS modeller/renderer into the native input of various renderers.   Really nice to have if you are getting models in 3DS format or if you are fortunate enough to have 3DS yourself.

SPD is a library of C code originally developed to benchmark various renderers.   Similar in nature to OORT (described below), it allows you to write short C programs that will generate scene files for a number of different renderers.   Currently supported are: Polyray, POV-Ray, Vivid Rayshade Rend 386, RTrace, NFF, and QRT.   There are also routines for displaying the models in wireframe (have to be tuned for individual compilers and platforms).   New output modes and rendering formats are still being added.

Other Raytracers

If you want to explore the world of raytracers, below is a list of what I consider the best of the Shareware/Freeware world.   (No flame wars please, these are just ones I happen to have and like.) Each has it's own strengths and weaknesses.   Overall, you will learn something from working with each one of them.

[Close] POV-Ray   (The POV-Ray Team)

[Close] Vivid   (Stephen Coy)

[Close] Rayshade   (Craig Kolb)

[Close] OORT   (Nicholas Wilt)

POV-Ray is the camel that you let warm it's nose in your tent.   It's starting to really dominate the scene.   It's strengths are in a broad set of features together with a huge level of support.   The authors are available on CompuServe (Graphdev) to answer questions, there are an astonishing number of utilities, and the source code is available for porting it to different platforms.

For a really fast raytracer, Vivid can't be beat.   It's DOS based only, but the registered version comes with a DOS extender for those really huge scene files.   Shape primitives are somewhat limited, but the texturing capabilities and camera lens features are very strong.

Rayshade is an old standard for the UNIX workstation crowd.   It has just about all of the shapes you would want, texturing approximately on a par with POV-Ray, and an interesting gridded optimization technique.   It doesn't do graphics, and it's really slow if you don't manually tune the optimization. On the plus side it has a long background of people using it and will compile on just about anything (source code is available).

A real newcomer, OORT is a whole bunch of C++ code (classes, objects, whatever) for creating and then rendering scenes.   No native input language yet (other than minimal NFF support), so you really need a C++ compiler to use it.   (This is likely to change as Nicholas is really energetic and seems to want to throw just about everything into OORT.)

References

Over the last year or so there have been several books published that describe ways to use Polyray. Although these books were written about v1.6a, almost all of the content is still valid for v1.7.   The differences are pretty well described in the topic <u>Whats New</u>.   Other differences are buried here and there in this document.   (I don't remember what they all are, so I won't try to list them.)

Of the books published, the three that really dig into using Polyray for various tasks, from modeling to animations are:

Making Movies on Your PC
David Mason & Alexander Enzmann
Waite Group Press, 1993
ISBN 1-878739-41-7

Adventures in Ray Tracing
Alfonso Hermida
Que, 1993
ISBN 1-56529-555-2

Animation How-To CD
Jeff Bowermaster
Waite Group Press, 1994
ISBN 1-878739-54-9

There's good stuff in each of these books, so it's not unreasonable to actually get all of them.   (I don't get royalties from any of them, so this is a reasonably shameless plug.)

`Up` **Overview of Polyray**

The following features are supported:

`Close`   Viewpoint (camera) characteristics

`Close`   Positional (point), directional (spotlight) light sources, and functional (textured) lights

`Close`   Background color

`Close`   Surface characteristics of objects

`Close`   Shape primitives: Bezier patch, blob, box, cone, cylinder, disc, glyph, implicit function, height field, lathe surface, NURBS, parabola, parametric function, polygon, polynomial function, sphere, sweep surface, torus, triangular patches

`Close`   Animation support

`Close`   Conditional processing

`Close`   Include files

`Close`   Named values and objects

`Close`   Constructive Solid Geometry (CSG)

`Close`   Grids of objects

`Close`   User definable (functional) textures

`Close`   Initialization file for default values

`Copy`

`Up` # Quick Demo

This section describes one of the simplest possible data files: a single sphere and a single light source.   In what follows; anything following a double slash, "//" is a <u>comment</u>, and is ignored by Polyray.   The data file follows in monospace type.   You can either use the file "sphere.pi" in the data archive, or copy and paste these declarations into a new file (with the copy button above).

```
// We start by setting up the camera.  This is where the eye
// is located, and describes how wide a field of view will
// be used, and the resolution (# of pixels wide and high)
// of the resulting image.

viewpoint {
   from <0,0,-8>         // The location of the eye
   at <0,0,0>            // The point that we are looking at
   up <0,1,0>            // The direction that will be up
   angle 45              // The vertical field of view
   resolution 160, 160  // The image will be 160x160 pixels
}

// Next define the color of the background.  This is the
// color that is used for any pixel that isn't part of an
// object.  In this file it will color the image around the
// sphere.

background skyblue

// Next is a light source.  This is a point light source a
// little above, to the left, and behind the eye.
light <-10, 3, -20>

// The following declaration is a "texture" that will be
// applied to the sphere.  It will be red at every point but
// where the light is reflecting directly towards the eye -
// at that point there will be a bright white spot.
define shiny_red
texture {
   surface {
      color red
      ambient 0.2          // Always a little red in the sphere
      diffuse 0.6          // Where the light is hitting the
                           // sphere, the color of the sphere
                           // will be a little brighter.
      specular white, 0.5 // There will be a white highlight.
      microfacet Cook 5   // The white highlight drops to half
                           // its intensity at an angle of 5
                           // degrees.
   }
}
```

```
// Finally we declare the sphere.  It sits right at the origin,
// and has a radius of two.  Following the declaration of the
// sphere, we associate the texture declared above.
object {
   sphere <0, 0, 0>, 2
   shiny_red
   }
```

Now that we have a data file, lets render it and show it on the screen.   First of all we can look at a wireframe representation of the file.   Enter the following commands (DOS prompts are in capitals).

```
C> polyray sphere.pi -r 2 -V 1 -W
```

An outline of the sphere will be drawn on the screen, press any key to get back to DOS
Next lets do a raytrace.   Enter the following:

```
C> polyray sphere.pi -V 1
```

The sphere will be drawn a line at a time, when it is done you will be returned to DOS.

The output image will be the default output file, "out.tga".   You can view it directly with VPIC or CSHOW (although VPIC will not have the full range of colors that were generated by the raytrace).   If you have PICLAB, then the following commands will load the image, map its colors into a spectrum that matches the colors in the image, then will display it:

```
C> piclab
> tload out
> makepal
> map
> show
```

Hit any key to get back to PICLAB's command line

```
> quit
C>
```

You should see a greatly improved image quality over the display that is shown during tracing.

Now that you have seen a simple image, you have two options:

> **1.** Go render some or all of the files in the data archive, or

> **2.** Continue reading the documents.

For those of you that prefer immediately getting started, there are a series of DOS batch files that pretty well automate the rendering of the sample scenes.   (When you unzip the data files, remember to use the **-d** switch so that all the subdirectories will be installed properly.)   The sample scenes in PLYDAT will render on a 33Mhz 486 in less than a day.   The animation examples will take an additional 4-5 days (and chew up quite a few megabytes of disk space).

Up Command Line Options

A number of operations can be manipulated through values in an <u>initialization file</u>, within the <u>data file</u>, or from the command line (processed in that order, with the last read having the highest precedence). The command line values will be displayed if no arguments are given to Polyray.

The values that can be specified at the command line, with a brief description of their meaning are:

-a n	Antialiasing (0=none, 1=filter, 2-4=adaptive)
-b pixels	Set the maximum number of pixels that will be calculated between file flushes
-B	Flush the output file every scan line
-d	Output the image as a depth map
-o filename	The output file name, the default output file name if not specified is "out.tga"
-p bits/pixel	Set the number of bits per pixel in the output file (must be one of 8, 16, 24, 32)
-P palette	Which palette option to use [0=grey, 1=666, 2=884]
-q flags	Turn on/off various global shading options
-Q	Abort if any key is hit during trace
-r renderer	Which rendering method: [0=raytrace, 1=scan convert, 2=wireframe, 3=raw tri, 4=u/v tri]
-R	Resume an interrupted trace.
-S samples	# of samples per pixel when performing focal blur
-t status_vals	Status display type [0=none, 1=totals, 2=line, 3=pixel].
-T threshold	Threshold to start adaptive antialiasing
-u	Write the output file in uncompressed form
-v	Trace from bottom to top
-V mode	Use VGA display while tracing [0=none,1-5=8 bit, 6-10=15 bit, 11-15=24 bit]
-W	Wait for key before clearing display
-x columns	Set the x resolution
-y lines	Set the y resolution
-z start_line	Start a trace at a specified line

If no arguments are given then Polyray will give a brief description of the command line options.

Up Initialization File

The first operation carried out by Polyray is to read the initialization file "polyray.ini".   This file can be used to tune a number of the default variables used during rendering.   This file must appear in the current directory.   This file doesn't have to exist, it is typically used as a convenience to eliminate retyping command line parameters.

Each entry in the initialization file must appear on a separate line, and have the form:

```
default_name    default_value
```

The names are text.   The values are numeric for some names, and text for others.   The allowed names and values are:

abort_test	true/false/on/off
alias_threshold	[Min value to start adaptive antialiasing]
antialias	none/filter/adaptive1/adaptive2
display	none/vga1-vga5/hicolor1-hicolor5/truecolor1-truecolor5
max_level	[max depth of recursion]
max_samples	[# samples for focal blur]
optimizer	none/slabs
pixel_size	[8, 16, 24, 32]
pixel_encoding	none/rle
renderer	ray_trace/scan_convert/wire_frame/ raw_triangles/uv_triangles
shade_flags	[default/bit mask of flags, (see topic Shading Quality Flags)]
shadow_tolerance	[minimum distance for blocking objects]
status	none/totals/line/pixel
warnings	on/off

A typical example of "polyray.ini" would be:

```
abort_test      on
alias_threshold 0.05
antialias       adaptive
display         vga
max_samples     8
pixel_size      24
status          line
```

If no initialization file exists, then Polyray will use the following default values:

```
abort_test      on
alias_threshold 0.2
antialias       none
display         none
max_level       5
max_samples     4
```

```
optimizer         slabs
pixel_size        16
pixel_encoding    rle
renderer          ray_trace
shade_flags       default
shadow_tolerance  0.001
status            none
warnings          on
```

Rendering Options

Polyray supports four very distinct means of rendering scenes: raytracing, polygon scan conversion, wireframe, and raw triangle output.   Raytracing is often a very time consuming process, however the highest quality of images can be produced this way.   Scan conversion is a very memory intensive method, but produces a good quality image quickly.   Wireframe gives a very rough view of the scene in the fastest possible time.   (Note that there is no output file when wireframe is used.)   Raw triangle output produces an ASCII file of triangles describing the scene.

Raytracing

Raytracing is a very "compute intensive" process, but is the method of choice for a final image.   The quality of the images that can be produced is entirely a function of how long you want to wait for results.   There are certain options that allow you to adjust how you want to make tradeoffs between: quality, speed, and memory requirements.

The basic operation in raytracing is that of shooting a ray from the eye, through a pixel, and finally hitting an object.   For each type of <u>primitive</u> there is specialized code that helps determine if a ray has hit that primitive.   The standard way that Polyray generates an image is to use one ray per pixel and to test that ray against all of the objects in the data file.   Antialiasing or focal blur will result in more than one ray per pixel, increasing rendering time.

Antialiasing

The representation of rays is as a 1 dimensional line.   On the other hand, pixels and objects have a definite size.   This can lead to a problem known as "aliasing", where a ray may not intersect an object because the object only partially overlaps a pixel, or the pixel should have color contributed by several objects that overlap it, none of which completely fills the pixel.   Aliasing often shows up as a staircase effect on the edges of objects.

Polyray offers a couple of ways to reduce aliasing:

> **1.** <u>Filtering</u>
>
> **2.** <u>Adaptive oversampling</u>.

The two initialization (and command line) variables that will affect the performance of adaptive antialiasing are "alias_threshold" and "max_samples".   The first is a measure of how different a pixel must be with respect to its neighbors before antialiasing will kick in.   If a pixel has the value: <r1, g1, b1>, and it's neighbor has the value <r2, g2, b2> (RGB values between 0 and 1 inclusive), then the "distance" between the two colors is:

> **dist = sqrt((r1 - r2)^2 + (b1 - b2)^2 + (g1 - g2)^2)**

This is the standard Pythagorean formula for values specified in RGB.   If "dist" is greater than the value of "alias_threshold", then extra rays will be fired through the pixel in order to determine an averaged color for that pixel.

Focal Blur

The default viewpoint declaration is a pinhole camera model.   All objects are in sharp focus, no matter how near or far they are.   By adjusting the aperture and focal_distance parameters it is possible to simulate a real world camera.   Objects close to the focal distance will be in focus and those closer or farther will be blurred.   Three parameters affect the blur: aperture, focal_distance, and max_samples.

Adjusting the size of aperture allows you to adjust how much of the scene is in focus.   Adjusting

focal_distance gives you control over what part of the scene is in focus.   If focal_distance isn't set, Polyray will set it so that the point defined by the 'at' declaration is in focus.   The declaration max_samples allows you to fine tune how smooth the final image appears.   If you use a large value for aperture, you will probably need to increase max_samples as well to avoid a grainy appearance in the image.

Also note that focal blur and adaptive antialiasing interact with each other.   Polyray attempts to distribute the focal blur rays into the antialiasing rays, however there will be more rays cast than would have been using either method alone.   It's best to leave the antialiasing off and the value of max_samples small until producing the final image.

Shading Quality Flags

By specifying a series of bits, various shading options can be turned on and off.   The value of each flag, as well as the meaning are:

1 - shadow_check	Shadows will be generated
2 - reflect_check	Reflectivity will be tested
4 - transmit_check	Check for refraction
8 - two_sides	If on, highlighting will be performed on both sides of a surface.
16 - uv_check	Calculate the uv-coordinates of each point.
32 - cast_shadow	Determines if an object can cast a shadow.

The default settings of these flags are:

raytracing	Shadow_Check + Reflect_Check + Transmit_Check + UV_Check (= 23)
scan conversion	[Two_Sides, = 8]
wireframe	Not applicable
raw triangles	Not applicable

Note that the flag, cast_shadow, is only meaningful for declarations of shading flags for an object, not for the renderer.   See the topic Shading Quality Flags for more information.   If you want to turn off shadows in a raytrace, then you would leave out the shadow_check flag (conversely if you wanted shadows in scan conversion you would add it).

The assumption made is that during raytracing the highest quality is desired, and consequently every shading test is made.   The assumption for scan conversion is that you are trying to render quickly, and hence most of the complex shading options are turned off.

If any of the three flags Shadow_Check, Reflect_Check, or Transmit_Check are explicitly set and scan conversion is used to render the scene, then at every pixel that is displayed, a recursive call to the raytracer will be made to determine the correct shading.   Note that due to the faceted nature of objects during scan conversion, shadowing, and especially refraction can get messed up during scan conversion.

For example if you want to do a scan conversion that contains shadows, you would use the following:

```
polyray xxx.pi -r 1 -q 1
```

or if you wanted to do raytracing with no shadows, reflection turned on, transparency turned off, and with diffuse and specular shading performed for both sides of surfaces you would use options 2 and

8:

```
polyray xxx.pi -q 10
```

NOTE: Texturing cannot be turned off.

Scan Conversion

In order to support a quicker render of images Polyray can render most <u>primitives</u> as a series of <u>polygons</u>.   Each polygon is scan converted using a Z-Buffer for depth information, and a S-Buffer for color information.

The scan conversion process does not by default provide support for: shadows, reflectivity, or transparency.   It is possible to instruct Polyray to use raytracing to perform these shading operations through the use of either the global shade flags or by setting shade flags for a specific object.   An alternative method for quickly testing shadows involves using depth mapped lights (see the topic <u>Depth Mapped Lights</u>).   Reflections can be simulated with a multipass approach where an image map or environment map is calculated and then applied to the reflective object.

The memory requirements for performing scan conversion can be quite substantial.   You need at least as much memory as is required for raytracing, plus at least 7 bytes per pixel for the final image. In order to correctly keep track of depth and color - 4 bytes are used for the depth of each pixel in the Z-Buffer (32 bit floating point number), and 3 bytes are used for each pixel in the S-Buffer (1 byte each for red, green, and blue).

During scan conversion, the number of polygons used to cover the surface of a primitive is controlled using the keywords "u_steps", and "v_steps" (or combined with uv_steps).   These two values control how many steps around and along the surface of the object values are generated to create polygons. (see the topic <u>UV Bounds</u>)   The higher the number of steps, the smoother the final appearance. Note however that if the object is very small then there is little reason to use a fine mesh - hand tuning is sometimes required to get the best balance between speed and appearance.

Generating iso-surfaces for <u>blobs</u>, <u>polynomial functions</u> and <u>implicit functions</u>, followed by polygonalization of the surfaces is performed using a <u>marching cubes</u> algorithm.   Currently the value of "u_steps" determines the number of slices along the x- axis, and the value of "v_steps" controls the number of slices along the y-axis and along the z-axis.   Future versions may introduce a third value to allow independent control of y and z.

Note that running Polyray in a Windows DOS box is a way to increase the memory available for performing scan conversion.   Polyray will take advantage of the virtual memory capabilities of Windows, allowing for larger image to be rendered.

Also note that there are images that render slower in scan conversion and wireframe that raytracing! These are typically those files that contain large numbers of spheres and cones such as data files generated by CTDS or large gridded objects.   The scan conversion process generates every possible part of each object, whereas the raytracer is able to sort the objects and only display the visible parts of the surfaces.   If you run into this situation, a speedup trick you can use is to set the values of uv_steps very low for the small objects (e.g., uv_steps 6, 4 for spheres).

Wireframe

In most cases the fastest way to display a model is by showing a wireframe representation.   Polyray supports this as a variant of the scan conversion process.   When drawing a wireframe image only the edges of the polygons that are generated as part of scan conversion are drawn onto the screen.   A big problem with wireframe representations is that <u>CSG</u> operations are not performed.   If you use CSG intersection, difference, or clipping, you may see a lot of stuff that will not appear in the final image.

Depth Files

Instead of creating an image, it is possible to instruct Polyray to create a file that contains depth information.   Each pixel in the resulting Targa will contain the distance from the eye to the first surface that is hit.   The format of the files is identical to the formats used by height fields (see File Based Height Fields).

For example, to create a 24 bit depth file, the following command could be used:

```
polyray sphere.pi -p 24 -d
```

Polyray will disable any antialiasing when creating depth files(it is inappropriate to average depths).

There are three common applications for depth files: shadow maps (Depth Lights), height fields, and creating Random Dot Stereograms (RDS).

RDS images are an interesting way to create a 3D picture that doesn't require special glasses.   The program RDSGEN (available in the GRAPHDEV forum on CompuServe) will accept Polyrays 24 bit depth format and create an RDS image.

Raw Triangles

A somewhat odd addition to the image output formats for Polyray is the generation of raw triangle information.   What happens is very similar to the scan conversion process, but rather than draw polygons, Polyray will write a text description of the polygons (after splitting them into triangles).   The final output is a (usually long) list of lines, each line describing a single smooth triangle.   The format of the output is one of the following:

```
x1 y1 z1 x2 y2 z2 x3 y3 z3
```
or
```
x1 y1 z1 x2 y2 z2 x3 y3 z3 nx1 ny1 nz1 nx2 ny2 nz2 nx3 ny3 nz3
u1 v1 u2 v2 u3 v3
```

If the output is raw triangles then only the three vertices are printed.   If uv_triangles are being created, then the normal information for each of the vertices follows the vertices and the u/v values follow them.   The actual u/v values are specific to the type of object being generated.

Currently I don't have any applications for this output, but the first form is identical to the input format of the program RAW2POV. The intent of this feature is to provide a way to build models in polygon form for conversion to the input format of another renderer.

For example, to produce raw triangle output describing a sphere, and dump it to a file you could use the command:

```
polyray sphere.pi -r 3 > sphere.tri
```

For full vertex, normal, and uv information, use the following command:

```
polyray sphere.pi -r 4 > uvsphere.tri
```

Nothing is drawn on screen while the raw triangles are generated, so typically you would turn off the graphics display (-V 0) when using this option.

Display Options (IBM format only)

Polyray supports 15 distinct VESA display modes.   The table below lists the modes, the command line value to use with '-V x', and the entry in the file polyray.ini to set a default video mode.

Resolution	Colors	Command Line	Initialization Name
N/A	none	-V 0	none
320x200	256	-V 1	vga/vga1
640x480	256	-V 2	vga2
800x600	256	-V 3	vga3
1024x768	256	-V 4	vga4
1280x1024	256	-V 5	vga5
320x200	32K	-V 6	hicolor/hicolor1
640x480	32K	-V 7	hicolor2
800x600	32K	-V 8	hicolor3
1024x768	32K	-V 9	hicolor4
1280x1024	32K	-V 10	hicolor5
320x200	16M	-V 11	truecolor1
640x480	16M	-V 12	truecolor2
800x600	16M	-V 13	truecolor3
1024x768	16M	-V 14	truecolor4
1280x1024	16M	-V 15	truecolor5

It is pretty unlikely you have a board that will do all of the resolutions listed above.   For that reason, Polyray will check to see if the requested mode is possible on your board.   If not, another mode will be selected.   The first priority is to find a mode with the same # of bits per pixel (even if the screen resolution is lower).   The second priority is to drop to the next lower # of bits per pixel, starting at the highest resolution possible.

Setting the default display mode in polyray.ini is done by inserting a line of the form:

```
display hicolor1
```

To override a display mode from the command line you would do something like:

```
C> polyray sphere.pi -V 12
```

Note that using the line status display will mess up some of the higher graphics modes.   If this bothers you, use the flag -t 1 to limit statistics to only startup and totals (or -t 0 for no statistics).

Filtering

Filtering smoothes the output image by averaging the color values of neighboring pixels to the pixel being rendered.   Oversampling is performed by adding extra rays through random jittering of the direction of the ray within the pixel being traced.   By averaging the result of all of the rays that are shot through a single pixel, aliasing problems can be greatly reduced.

The filtering process adds little overhead to the rendering process, however, the resolution of the image is degraded by the averaging process. It simply averages the colors found at the four corners of a pixel.

Adaptive Oversampling

Adaptive antialiasing starts by sending a ray through the four corners of a pixel.   If there is a sufficient contrast between the corners then Polyray will fire five more rays within the pixel.   This results in four subpixels.   Each level of adaptive antialiasing repeats this procedure.   If adaptive1 is used in polyray.ini, or -a 2 from the command line, then the antialiasing stops after a single subdivision.   If adaptive2 is used then each subpixel is checked and if the corners are sufficiently different then it will be divided again.

Rendering Options

Raytracing

Antialiasing

Focal Blur

Bounding Slabs

Shading Quality Flags

Scan Conversion

Wireframe

Depth Files

Raw Triangles

File Formats

Output Files

Currently the output file formats are 8, 16, 24, and 32 bit uncompressed and RLE compressed Targa . If no output file is defined with the **-o** command line option, the file "out.tga" will be used.   The command line option **-u** specifies that no compression should be used on the output file.

The default output format is an RLE compressed 16-bit color format.   This format holds 5 bits for each of red, green, and blue.   The reason for choosing this format is that very few can afford 24-bit color boards and this format meets the limits of the typical 8 and 15 bit color boards.   If you have the hardware to display more than 32K colors then set the command line switches (i.e. **-p 24** or **-p 32**) or set the defaults in the initialization file **polyray.ini** (i.e. **pixelsize 24** or **pixelsize 32**) to generate 24 or 32 bit Targa files.

Algorithms

Processing Polynomial Expressions

The program in its current form allows the definition of polynomial surfaces of degree less than 33 (Due to numerical problems, don't bother with polynomials of degree more than around 10, and if possible keep them less than 6.)

The way that polynomial expressions are processed and manipulated follows the following steps:

1. Parse the expression as taken from the input file.   The YACC parser that is used to process the input file creates a parse tree of the expression as it is read.   The format for allowed expressions is described in the topic <u>Allowed Formula Syntax.</u>

2. Expand the expression into a sum of simple subexpressions.   After the parse tree has been built, the expression is expanded so that the expression is in expanded form. (i.e. $(x+1)^2$ is rewritten as $x^2 + 2*x + 1$.)   The rules for doing this are explained in the topic <u>Rules for Processing Formulas.</u>

3. Determine what order polynomial is represented by the expression.   Every term in the expanded expression is examined to see what the largest order of x, y, and z is used.   This determines the order N of the polynomial.   From this value, the number of possible terms of that order is computed.

4. Create an array of coefficients for the general three variable polynomial equation of order N.   There are a total of $(N+1)*(N+2)*(N+3)/6$ terms in the general equation.   The values of each of the coefficients in the expanded expression are then substituted into the array of coefficients.

NOTE: The storage of expressions is not particularly efficient for high orders of N.

Example Equation Representation

As an example of how the parse tree is expanded, given the input:

```
(x+y)*(1-x-z)  +  (x-z)^2.
```

The expanded representation is:

```
-1*x*y + -3*x*z + x + -1*y*z + y + z^2
```

This is a polynomial of order 2, and the array of coefficients is:

```
0 0 1 -1 -3 -1 1 1 0 0
```

Allowed Formula Syntax

The keyword here is "simple" algebraic expressions.   The only allowed variables are "x", "y", and "z" (and only lower case). Constants are floating point numbers.   The operations allowed between constants and the three variable symbols are: addition, subtraction, multiplication, unary minus, and exponentiation by an positive integer power.   Examples of expressions that can be processed are:

```
x^2+y-42
(1+x+y+z)^2 * 2*(x-y)*(z-x*(133-z^2))
27*x-y*42*y^2
(x^2+y^2+z^2+3)^2 - (x*y-x^2)*z^2
```

Examples of expressions that cannot be processed are:

```
x/y
x^0.5+y
2(x-y)
```

```
x^-2
```

Rules for Processing Formulas

There are only a few rules needed to simplify a polynomial expression to the point that it can be easily turned into an array of coefficients. They appear below, with the letters A, B, C, and D standing for an arbitrary sub-expression.   The rules used to simplify an expression are:

Addition:
No simplification needed for the term: A + B

Subtraction:

A-B   A + (-B)

Unary minus:

-(A*B)   (-A * B)

-(A+B)   (-A + -B)

Multiplication:

A*(B+C)   A*B + A*C

(A+B)*C   A*C + B*C

(A+B)*(C+D)   A*C + A*D + B*C + B*D

Exponentiation:

(A*B)^n   A^n * B^n

(A+B)^n   C(n,0) * A^n + C(n,1) * A^(n-1) * B + ... C(n,n-1) * A * B^(n-1) +
               C(n,n) * B^n { C(n, r) is the binomial coefficient }

Each of the subexpressions is further simplified until every term in the expression has the form:

```
const * x^i * y^j * z^k.
```

| Close | # Processing of Arbitrary Functional Surfaces |

The basic algorithm for determining a point of intersection using interval math is quite simple:

1. Given a range of allowed values for the ray parameter **T**, a range of function values is calculated.

2. If the range does not include 0 then there is no intersection.

3. If the range includes 0, then:

   a. The range of values of the derivative of the function is calculated for the allowed values of the ray parameter T.

   b. If the range of values of the derivative includes 0, then the interval for T is bisected and we process each subinterval starting with step 1.

   c. If the range of values of the derivative does not include 0, then we know that there is exactly one solution of the function in the current interval.   A standard root solver using the regula-falsi method is called to find the root.

There is of course much more to this...

Spherical Coordinates

To ray-trace functions that are defined in terms of spherical coordinates use the substitutions:

```
r     = sqrt(x^2 + y^2 + z^2),
theta = atan(y/x),
phi   = atan(sqrt(x^2 + y^2)/z)
```

Sample files: sectorl.pi, zonal.pi

Three Dimensional Noise Generation

The way Polyray generates noise values from vector inputs follows these steps:

1. The integer part of the x, y, and z values of the vector are determined.

2. The eight integer points surrounding the input vector are determined.

3. A hashing function (described below) is applied to the integer points surrounding the input vector, resulting in eight random numbers.

4. A final number is determine by weighting the values associated with the surrounding points with the distance from the input vector to those points.

The result of this is that each point on an integer lattice has a random value, but each fractional value inside a unit cube in the lattice is related to its surrounding points.

The hashing function I use takes the bottom 10 bits of the integer parts of the input vector, squishes them into a single number, then crunches this value through a series of adds, multiplies, and mods to get a single result.   The algorithm is:

```
K = ((x & 0x03ff) << 20) |
    ((y & 0x03ff) << 10) |
     (z & 0x03ff);
Kt = 0;
for (i=0;i<mult_table_size;i++)
   Kt = ((Kt + K) * mult_table[i]) % HASH_SIZE;
result = (Flt)Kt / (Flt)(HASH_SIZE - 1);
```

The number and value of the entries in "mult_table" were chosen at random, as was the value HASH_SIZE.   I haven't bothered to determine if the results from this hashing function really are spread evenly, however the visual results from using it are quite good.

Close Marching Cubes

The marching cubes algorithm puts a lattice over the three dimensional space that an object sits in. By stepping through each subcube in the lattice and looking for places where the surface passes through a subcube, a good guess at a polygonal cover for the surface can be generated. This algorithm is used for scan converting blobs, polynomial functions, and implicit functions.

As each cube is processed, every vertex of the cube is tested to see if it is inside the object "hot", or outside the object "cold". There are a total of 2^8 = 256 possible combinations of hot and cold vertices.

The processing of each subcube in the lattice takes these steps:

Close Each of the 8 vertices of the cube corresponds to a bit in an unsigned char. If the vertex is hot then the bit is set to 1, if the vertex is cold the bit is set to 0.

Close Using a little combinatorics and group theory, it is possible to classify, under the group of solid rotations of the cube, each of the 256 possibilities to one of 23 unique base cases. (You can actually drop down to 14, but it would have made parts of the algorithm more complicated.)

Close Each of the base cases has an entry in a table of triangles that separate the hot vertices from the cold vertices for that case.

Close By using the inverse of the rotations that took the cube into the base case, the triangles are rotated to separate the hot and cold vertices of the original cube.

Close The amount of hot and amount of cold that are separated by each triangle is examined in order to see how much to push the triangle up or down. This is a linear interpolation that helps to keep the triangles close to the actual surface.

Close Now that the triangles are known and in approximately the right place, the normal scan conversion routine is called for each triangle.

The biggest drawback of this algorithm is that the number of evaluations goes up with the cube of the resolution of the lattice. For example, if u_steps is 20 and v_steps is 20 (the default), then 20x20x20 = 8000 individual subcubes are visited. If you were to increase the resolution by a factor of 5, there would be 100x100x100 = 1,000,000 subcubes visited.

Another problem is one of aliasing. If the surface is very complex and/or contains many small features, and the number of steps along the axes is insufficient to properly determine inside/outside, then large portions of the surface can disappear. The solution is to increase the step sizes, or to raytrace, either option will increase processing time.

> **NOTE:** When I built the table of triangles, I made the assumption that if there are two ways to split hot from cold, that hot would always connect to hot. One example of this choice is when the vertices: (0, 0, 0), (1, 0, 0), (0, 1, 1), and (1, 1, 1) are all hot and the rest are cold.

Sample Files

A number of sample files are referenced in this document.   These files are contained in a separate archive, and demonstrate various features of Polyray.

Simple demo files:
boxes.pi, cone.pi, cossph.pi, cwheel.pi, cylinder.pi, disc.pi, gsphere.pi, sphere.pi, spot0.pi

Sample color/texture definitions:
colors.inc, textures.inc

Polynomial surfaces:
bicorn.pi, bifolia.pi, cassini.pi, csaddle.pi, devil.pi, folium.pi, helix.pi, hyptorus.pi, kampyle.pi, lemnisca.pi, loop.pi, monkey.pi, parabol.pi, partorus.pi, piriform.pi, qparab.pi, qsaddle.pi, quarcyl.pi, quarpara.pi, steiner.pi, strophid.pi, tcubic.pi, tear5.pi, torus.pi, trough.pi, twincone.pi, twinglob.pi, witch.pi

Implicit function surfaces:
sectorl.pi, sombrero.pi, sinsurf.pi, superq.pi, zonal.pi

Bezier patches:
bezier0.pi, teapot.pi, teapot.inc

Height fields:
hfnoise.pi, sombfn.pi, sinfn.pi, wake.pi

Image mapping:
map1.pi

Texturing, CSG, etc.:
lens.pi, lookpond.pi, marble.pi, polytope.pi, spot1.pi, wood.pi, xander.pi

Data file generators:
balls.c, coil.c, gears.c, hilbert.c, mountain.c, sphcoil.c, tetra.c

Animation files:
plane.pi, squish.pi, whirl.pi

Bibliography

"Introduction to Ray Tracing"
Edited by Andrew Glassner
Academic Press, 1989

"Illumination and Color in Computer Generated Imagery"
Roy Hall
Springer Verlag, 1989

"Numerical Recipes in C"
Press, et al.
Cambridge University Press, 1988

"CRC Handbook of Mathematical Curves and Surfaces"
David H. von Seggern
CRC Press, 1990

"Robust Ray Intersection with Interval Arithmetic"
D.P. Mitchell,
from:
   "Proceedings Graphics Interface '90"
   Canadian Information Processing Society

Detailed Description of the Polyray Input Format

An input file describes the basic components of an image:

A viewpoint that characterizes where the eye is, where it is looking and what its orientation is.

Objects, their shape, placement, and orientation.

Light sources, their placement and color.

Textures for coloring and patterning surfaces.

Beyond the fundamentals, there are many components that exist either as a convenience such as definable expressions, or textures.   This section of the document describes in detail the syntax of all of the components of an input file.

Close

Close

Expressions

There are six basic types of expressions that are used in Polyray:

1. **float:** Floating point expression (i.e. 0.5, 2 * sin(1.33)).   These are used at any point a floating point value is needed, such as the radius of a sphere or the amount of contribution of a part of the lighting model.

2. **vector:** Vector valued expression   (i.e. <0, 1, 0>, red, 12 * <2, sin(x), 17> + P).   Used for color expressions, describing positions, describing orientations, etc.

3. **arrays:** Lists of expressions (i.e. [0, 1, 17, 42], [<0,1,0>, <2*sin(theta), 42, -4>, 2*<3, 7, 2>]

4. **cexper:** Conditional expression (i.e. x < 42).

5.   **string:** Strings used for file names or systems calls

6.   **images:** A Targa, GIF, or JPEG image.

The following sections describe the syntax for each of these types of expressions, as well as how to define variables in terms of expressions.   See also the description of color maps, image maps (from which you can retrieve color or vector values), indexed maps, and height maps.

Numeric Expressions

In most places where a number can be used (i.e. scale values, angles, RGB components, etc.) a simple floating point expression (float) may be used.   These expressions can contain any of the following terms:

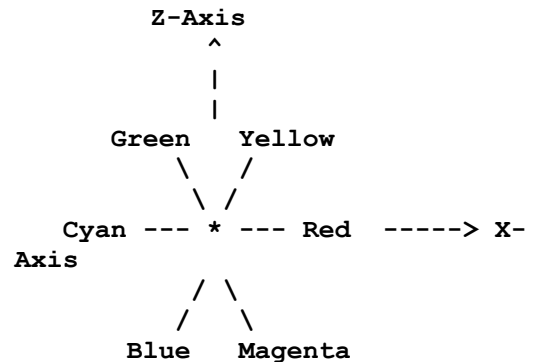|	
-0.1, 5e-3, ab, ...	A floating point number or defined value
'(' float ')'	Parenthesised expression
float ^ float	Same as pow(x, y)
float * float	Multiplication
float / float	Division
float + float	Addition
float - float	Subtraction
-float	Unary minus
acos(float)	Arccosine, (radians for all trig functions)
asin(float)	Arcsin

atan(float)	Arctangent
ceil(float)	Ceiling function
cos(float)	Cosine
cosh(float)	Hyperbolic cosine
degrees(float)	Converts radians to degrees
exp(float)	e^x, standard exponential function
fabs(float)	Absolute value
floor(float)	Floor function
fmod(float, float)	Modulus function for floating point values
heightmap(image, vector)	Height of an pixel in an image
ln(float)	Natural logarithm
indexed(image, vector)	Index of an pixel in an image
legendre(l, m, n)	Legendre function
log(float)	Logarithm base 10
min(float,float)	Minimum of the two arguments
max(float,float)	Maximum of the two arguments
noise(vector) noise(vector, float)	Solid texturing (noise) function. If the second argument is given, it is used as the number of octaves (repetitions) of the 3D noise function.
noise(vector,vector)	Second arg provides more flexible operation using: <pos scale, noise scale, octaves>
pow(float, float)	Exponentiation (x^y)
radians(float)	Converts degrees to radians
sawtooth(float)	Sawtooth function (range 0 - 1)
sin(float)	Sine
sinh(float)	Hyperbolic sine
sqrt(float)	Square root
tan(float)	Tangent
tanh(float)	Hyperbolic tangent
visible(vector, vector)	Returns 1 if second point visible from first.
vector[i]	Extract component i from a vector (0<=i<=3).
vector . vector	Dot product of two vectors
|float|	Absolute value (same as fabs)
|vector|	Length of a vector

Vector Expressions

In most places where a vector can be used (i.e. color values, rotation angles, locations, etc.), a vector

expression is allowed.   The expressions can contain any of the following terms:

vector + vector	Addition
vector - vector	Subtraction
vector * vector	Cross product
vector * float	Scaling of a vector by a scalar
float * vector	Scaling of a vector by a scalar
vector / float	Inverse scaling of a vector by a scalar
brownian(vector)	Random displacement of up to 0.1
brownian(vector, vector)	Makes a random displacement of the first point by an amount proportional to the components of the second point
color_wheel(x, y, z)	RGB color wheel using x and z (y ignored), the color returned is based on <x, z> using the diagram below:

```
              Z-Axis
                ^
                |
                |
         Green     Yellow
            \     /
             \   /
    Cyan --- * --- Red   -----> X-
Axis
             /   \
            /     \
         Blue    Magenta
```

Intermediate colors are generated by interpolation.

dnoise(vector) **dnoise(vector,float)**	Returns a vector (gradient) based on the location given in the first argument. If the second argument is given, it is used as the number of octaves (repetitions) of the 3D noise function.
dnoise(vector,vector)	Second arg provides more flexible operation using: <pos scale, noise scale, octaves>.
planar_imagemap(image, vector [, rflag]) **cylindrical_imagemap(image, vector [, rflag])** **spherical_imagemap(image, vector [, rflag])** **environment_map(vector, environment)**	Image map lookup functions. If the third argument is given, then the image will be tiled, otherwise black is used outside the bounds of the image. Note: for planar image maps only the x and z coordinates of the second argument are used.
rotate(vector, vector)	Rotate the point specified in the first argument by the angles specified in the second argument. (angles in degrees)
rotate(vector, vector, float)	Rotate the point specified in the first argument around the axis specified in the

	second argument, by the angle specified in the third argument. (angle in degrees)
reflect(vector, vector)	Reflect the first vector about the second vector (particularly useful in environment maps)
trace(vector, vector)	Color resulting from tracing a ray from the the point given as the first argument in the direction given by the second argument.

Arrays

Arrays are a way to represent data in a convenient list form.   A good use for arrays is to hold a number of locations for polygon vertices or as locations for objects in successive frames of an animation.

As an example, a way to define a tetrahedron (4 sided solid) is to define its vertices, and which vertices make up its faces.   By using this information in an object declaration, we can make a tetrahedron out of polygons very easily.

```
define tetrahedron_faces
   [<0, 1, 2>, <0, 2, 3>, <0, 3, 1>, <1, 3, 2>]

define tetrahedron_vertices
   [<0, 0, sqrt(3)>, <0, (2*sqrt(2)*sqrt(3))/3, -sqrt(3)/3>,
    <-sqrt(2), -(sqrt(2)*sqrt(3))/3, -sqrt(3)/3>,
    <sqrt(2), -(sqrt(2)*sqrt(3))/3, -sqrt(3)/3>]

define tcf tetrahedron_faces
define tcv tetrahedron_vertices
define tetrahedron
object {
   object { polygon 3, tcv[tcf[ 0][0]],
            tcv[tcf[ 0][1]], tcv[tcf[ ][2]] } +
   object { polygon 3, tcv[tcf[ 1][0]],
            tcv[tcf[ 1][1]], tcv[tcf[ 1][2]] } +
   object { polygon 3, tcv[tcf[ 2][0]],
            tcv[tcf[ 2][1]], tcv[tcf[ 2][2]] } +
   object { polygon 3, tcv[tcf[ 3][0]],
            tcv[tcf[ 3][1]], tcv[tcf[ 3][2]] }
   }
```

What happened in the object declaration is that each polygon grabbed a series of vertex indices from the array "tetrahedron_faces", then used that index to grab the actual location in space of that vertex.

Another example is to use an array to store a series of view directions so that we can use animation to generate a series of very distinct renders of the same scene (the following example is how the views for an environment map are generated):

```
define location <0, 0, 0>
define at_vecs [<1, 0, 0>, <-1, 0, 0>, < 0, 1, 0>, < 0,-1, 0>,
               < 0, 0,-1>, < 0, 0, 1>]
define up_vecs [< 0, 1, 0>, < 0, 1, 0>, < 0, 0, 1>,
               < 0, 0,-1>, < 0, 1, 0>, < 0, 1, 0>]

// Generate six frames
start_frame 0
```

```
   end_frame 5

   // Each frame generates the view in a specific direction.   The
   // vectors stored in the arrays "at_vecs", and "up_vecs" turn
   // the camera in such a way as to generate image maps correct
   // for using in an environment map.
   viewpoint {
      from location
      at location + at_vecs[frame]
      up up_vecs[frame]
      ...
      }
   ... Rest of the data file ...
```

Sample files: plyhdrn.pi, environ.pi

Conditional Expressions

Conditional expressions are used in one of two places: conditional processing of *declarations* or *conditional value functions*.

cexper has one of the following forms:

> **!cexper**
> **cexper && cexper**
> **cexper || cexper**
> **float < float**
> **float <= float**
> **float > float**
> **float >= float**
> **float == float**

A use of conditional expressions is to define a texture based on other expressions, the format of this expression is:

```
(cexper ? true_value : false_value)
```

Where true_value/false_value can be either floating point or vector values.   This type of expression is taken directly from the equivalent in the C language.   An example of how this is used (from the file "spot1.pi") is:

```
special surface {
  color white
  ambient (visible(W, throw_offset) == 0 ? 0
              : (P[0] < 1 ? 1
                : (P[0] > throw_length ? 0
                  : (throw_length - P[0]) / throw_length)))
   transmission (visible(W, throw_offset) == 1
                ? (P[0] < 1 ? 0
                  : (P[0] > throw_length ? 1
                    : P[0] / throw_length))
                : 1), 1
   }
```

In this case conditional statements are used to determine the surface characteristics of a one defining the boundary of a spotlight.   The amount of ambient light is modified with distance from the apex of the cone, the visibility of the cone is modified based on both distance and on a determination if the cone is behind an object with respect to the source of the light.

Run-Time Expressions

There are a few expressions that only have meaning during the rendering process:

I	The direction of the ray that struck the object
P	Point of intersection in object coordinates.
N	The normal to the point of intersection in "world" coordinates.
W	The point of intersection in "world" coordinates.
U	The u/v/w coordinate of the intersection point.
x,y,z	Components of the point in object coordinates.
u,v,w	Components of the uv-coordinate of the point.

These expressions describe the interaction of a ray and an object.   To use them effectively, you need to understand the distinction between "world" coordinates, "object" coordinates, and "u/v" coordinates.   Object coordinates describe a point or a direction with respect to an object as it was originally defined.   World coordinates describe a point with respect to an object after it has been rotated/scaled/translated.   u/v coordinates describe the point in a way natural to a particular object type (e.g., latitude and longitude for a sphere).   Typically texturing is done in either object coordinates or u/v coordinates so that as the object is moved around the texture will move with it.   On the other hand shading is done in world coordinates.

The variables u, v, and w are specific to each surface type and in general are the natural mapping for the surface (e.g., latitude and longitude on a sphere)   In general u varies from 0 to 1 as you go around an object and v varies from 0 to one as you go from the bottom to the top of an object.   These variables can be used in a couple of ways, to tell Polyray to only render portions of a surface within certain uv bounds, or they can be used as arguments to expressions in textures or displacement functions.

Not all primitives set meaningful values for u and v, those that do are:

```
bezier, cone, cylinder, disc, height fields, NURB, parabola, parametric,
sphere, torus, patch
```

Other surface types will simply have u=x, v=y, w=z.   An intersection with a gridded object will use the u/v coordinates of the object within the grid.

See the file uvtst.pi in the data archives for an example of using uv bounds on objects.   The file spikes.pi demonstrates using uv as variables in a displacement surface.   The file bezier1.pi demonstrates using uv as variables to stretch an image over the surface of a bezier patch.

The meanings of some of these variables are slightly different when creating particle systems (see 2.3.5), or when coloring the background of an image (see 2.4.2)

NOTE: Capitalization of these variables is important.

Named Expressions

A major convenience for creating input files is the ability to create named definitions of surface models, object definitions, vectors, etc.   The way a value is defined takes one of the following forms:

```
define token expression        float, vector, array, cexper
define token "str..."              string expression
define token object { ... }
```

```
define token surface { ... }
define token texture { ... }
define token transform { ... }   Each entry may be one of
        scale/translate/rotate/shear
define token particle { ... }
```

Objects, surfaces, and textures can either be instantiated as-is by using the token name alone, or it can be instantiated and modified:

```
token,
```
or
```
token { ... modifiers ... },
```

Polyray keeps track of what type of entity the token represents and will parse the expressions accordingly.

> **NOTE: It is not possible to have two types of entities referred to by the same name.   If a name is reused, then a warning will be printed, and all references to that name will use the new definition from that point on.**

Static Variables

Static variables are a way to retain variable values from frame to frame of an animation.   Instead of the normal declaration of a variable:

```
define xyz 32 * frame
```

you would do something like this:

```
if (frame == start_frame)
    static define xyz 42
else
    static define xyz (xyz + 0.1)
```

The big differences between a "static define" and a "define" are that the static will be retained from frame to frame, and the static actually replaces any previous definitions rather than simply overloading them.

The static variables have an additional use beyond simple arithmetic on variables.   By defining something that takes a lot of processing at parse time (like height fields and image maps), you can make them static in the first frame and simply instantiate them every frame after that.

One example of this would be to spin a complex height field, if you have to create it every frame, there is a many second long wait while Polyray generates the field.   The following declarations would be a better way:

```
if (frame == start_frame)
    static define sinsf
        object {
            smooth_height_fn 128, 128, -2, 2, -2, 2, 0.25 *
                sin(18.85 * x * z + theta_offset)
            shiny_red
            }
    ...
    sinsf
    ...
```

Several examples of how static variables can be used are found in the animation directory in the data file archive.   Two that make extensive use of static variables are movsph.pi, which bounces several spherical blob components around inside a box, and cannon.pi which points a cannon in several directions, firing balls once it is pointed.

> **Warning: A texture inside a static object should ONLY be static itself.   The reason is that between frames, every non-static thing is deallocated.   If you have things inside a static object that point to a deallocated pointer, you will most certainly crash the program.   Sorry, but detecting these things would be too hard and reallocating all the memory would take up too much space.**

Lazy Evaluation

Normally, Polyray tries to reduce the amount of time and space necessary for evaluating and storing variables and expressions.   For example, if you had the following definition of x:

```
define x 2 * 3
```

Polyray would store 6 rather than the entire expression.   This can cause problems in certain circumstances if Polyray decides to reduce the expression too soon.   The problem is most notable in particle systems since the expression that is used in it's declaration is used for the life of the particle system.   An example of the problem would be a declaration like:

```
define t (frame - start_frame) / (end_frame - start_frame)
```

When Polyray encounters this it will store a value for t that is between 0 (for the first frame) and 1 (for the last frame).   This is great, up until you declare a particle system with something like:

```
if (frame == start_frame)
define partx
particle {

death (t > 0.5 ? 1 : 0)
}
```

Clearly the intent here is that the particles created by this system will die halfway through the animation (t > 0.5).   This won't happen, since Polyray will evaluate the value of t at the time that partx is declared, and store the expression (0 > 0.5 ? 1 : 0), which then reduces to 0.   This means the particle will never die.

To avoid this difficulty, the keyword `noeval` can be added to a definition to force Polyray to postpone evaluation.   The declaration of t would now be:

```
define noeval t (frame - start_frame) / (end_frame - start_frame)
```

When `partx` is declared, the entire expression is now used and the particles will die at the appropriate time.   Note that `noeval` is always the last keyword in a definition, and it works correctly with static definitions as well as normal ones.

Sample file: part6.pi

Types of Expressions

Close │ **Definition of the Viewpoint**

The viewpoint and its associated components define the position and orientation the view will be generated from.

The format of the declaration is:

```
viewpoint {
    from vector
    at vector
    up vector
    angle float
    hither float
    resolution float, float
    aspect float
    yon float
    max_trace_depth float
    aperture float
        max_samples float
focal_distance float
        image_format float
        pixel_encoding float
        pixelsize float
        antialias float
        antialias_threshold float
    }
```

All of the entries in the viewpoint declaration are optional and have reasonable default values (see below).   Order of the entries defining the viewpoint is not important, unless you redefine some field. (In which case the last is used.)

The parameters are:

aspect	The ratio of width to height. (Default: 1.0.)
at	A position to be at the center of the image, in XYZ world coordinates. (Default: <0, 0, 0>)
angle	The field of view (in degrees), from the center of the top row to the center of the bottom row. (Default: 45 degrees)
antialias	Level of antialiasing to use (Default: 0).
antialias_threshold	Threshold to start antialiasing (Default: 0.01)
aperture	If larger than 0, then extra rays are shot (controlled by "max_samples" in the initialization file) to produce a

	blurred image. Good values are between 0.1 and 0.5. (Default: 0)
focal_distance	Distance from the eye to the point that things are in focus, this defaults to the distance between "from" and "at".
from	The eye location in XYZ. (Default: <0, 0, -1>)
hither	Distance to front of view pyramid. Any intersection less than this value will be ignored. (Defaults: 1.0e-3)
image_format	If 0 then normal image, if 1 then depth image.
max_samples	Number of rays/pixel when performing focal blur (Default: 4).
max_trace_depth	This allows you to tailor the amount of recursion allowed for scenes with reflection and/or transparency. (Default: 5)
resolution	The number of pixels wide and high of the raster. (Default: 256x256)
up	A vector defining which direction is up, as an XYZ vector. (Default: <0, 1, 0>)
yon	Distance to back of view pyramid. Any intersection beyond this distance will be ignored. (Defaults to 1.0e5)

The view vectors will be coerced so that they are perpendicular to the vector from the eye (from) to the point of interest (at).

A typical declaration is:

```
viewpoint {
   from <0, 5, -5>
   at   <0, 0,  0>
   up   <0, 1,  0>
   angle 30
   resolution 320, 160
   aspect 2
   }
```

In this declaration the eye is five units behind the origin, five units above the x-z plane and is looking at the origin. The up direction is aligned with the y-axis, the field of view is 30 degrees and the output file will default to 320x160 pixels.

In this example it is assumed that pixels are square, and hence the aspect ratio is set to width/height. If you were generating an image for a screen that has pixels half as wide as they are high then the aspect ratio would be set to one.

> **Note that you can change from left handed coordinates (the default for Polyray) to right handed by using a negative aspect ratio (e.g., aspect -4/3).**

Objects and Surfaces

In order to make pictures, the light has to hit something.   Polyray supports several underlined primitive objects, and the following sections describe the syntax for describing the primitives, as well as how more complex primitives can be built from simple ones.

The "object" declaration is how Polyray associates a surface with its lighting characteristics, and its orientation.   This declaration includes one of the primitive shapes (sphere, polygon, ...), and optionally: a texture declaration (set to a matte white if none is defined), orientation declarations, or a bounding box declaration.

The format the declaration is:

```
object {
    shape_declaration
    [texture_declaration]
    [transformation declarations]
    [subdivision declarations]
    [bounding box declaration]
    [u/v bounds declaration]
    }
```

There are three separate ways that an object can be rendered by Polyray: raytracing, scan conversion, or wireframe.   The steps taken by each of these methods are:

Raytracing

1. Find the object that seems closest to the eye along the current ray.   The bounding slab process is what determines what order objects will be tested against the ray.

2. Find all intersections of the ray with the object.   (Up to the maximum number of intersections.)

Scan Conversion

1. The object is broken up into a number of rectangles (the actual number depends on the values of "u_steps" and "v_steps" in the object declarations).

2. The rectangle is clipped so that only the visible part will be rendered.

3. The clipped polygon is scan converted a line at a time.   As the scan conversion takes place, the object and world coordinates of the polygon are interpolated from vertices.

4. As each pixel is generated by scan conversion, its distance from the eye is checked against a Z-Buffer.

5. If the distance is less than the value in the Z-Buffer, then any CSG and clipping operations associated with the object are performed.

6. If the pixel passes the depth check and CSG checks, then the pixel is shaded using the texture information associated with the object.   The depth to the pixel is stored in the Z-Buffer and the color associated with the pixel is stored in the S-Buffer.

Wireframe

1. The object is broken up into a number of rectangles (the actual number depends on the values of "u_steps" and "v_steps" in the object declarations).

2. The rectangle is clipped so that only the visible part will be rendered.

3. The clipped rectangle is displayed on the screen. <u>CSG</u> is not taken into consideration in a wireframe image.

Determining if a point is inside an object is performed by evaluating the function that describes the object using the point of intersection. If the value is less than or equal to 0 then the point is inside. Some primitives do not have an inside (triangular patches), others do not have a well defined inside (cylinder).

The following sub-sections describe the format of the individual parts of an object declaration.

NOTE: The shape declaration MUST be first in the declaration, as any operations that follow have to have data to work on.

Close

Close

Object Modifiers

Object modifiers are statements within an object declaration that are used to move and/or change the shape of the object.

The declarations are processed in the order they appear in the declaration, so if you want to resize and object before moving it, you need to put the scale statement before the translation statement. The exception to this are displacement and u/v bounds. These act on the underlying shape and are applied prior to any other object modifiers, regardless of where they appear in the object declaration.

Bounding Box

In order to speed up the process of determining if a ray intersects an object and in order to define good bounds for surfaces such as polynomials and implicit surfaces, a bounding box can be specified. A short example of how it is used to define bounds of a polynomial:

```
define r0 3
define r1 1
define torus_expression (x^2 + y^2 + z^2 - (r0^2 + r1^2))^2 -
                          4 * r0^2 * (r1^2 - z^2)
object {
   polynomial torus_expression
   shiny_red
   bounding_box <-(r0+r1), -(r0+r1), -r1>,
                <(r0+r1), (r0+r1), r1>
   }
```

The test for intersecting a ray against a box is much faster than performing the test for the polynomial equation. In addition the box helps the scan conversion process determine where to look for the surface of the torus.

Subdivision of Primitives

The amount of subdivision of a primitive that is performed before it is displayed as polygons is tunable. These declarations are used for scan conversion of object, when creating displacement surfaces, and to determine the quality of an implicit function. The declarations are:

```
u_steps n
v_steps m
w_steps l
uv_steps n, m
uv_steps n, m, l
```

Where U generally refers to the number of steps "around" the primitive (the number of steps around the equator of a sphere for example). The parameter V refers to the number of steps along the primitive (latitudes on a sphere). Cone and cylinder primitives only require 1 step along V, but for

smoothness may require many steps in U.

For blobs, polynomials, and implicit surfaces, the u_steps component defines how many subdivisions along the x-axis, the v_steps component defines how many subdivisions along the y-axis, and the w_steps component defines how many subdivision along the z-axis.

Shading Flags

It is possible to tune the shading that will be performed for each object.   The values of each bit in the flag has the same meaning as that given for global shading in the topic <u>Shading Quality Flags</u>:

|	
1	Shadow_Check - Shadows will be generated
2	Reflect_Check - Reflectivity will be tested
4	Transmit_Check - Check for refraction
8	Two_Sides - If on, highlighting will be performed on both sides of a surface.
16	Cast_Shadow - Determines if an object casts shadows.
32	Primary_Rays - If off, then primary rays (those from the eye) are not checked against the object.

By default, all objects have the following flags set: Shadow_Check, Reflect_Check, Transmit_Check, UV_Check, and Cast_Shadow.   The two sides check is normally off.

The declaration has the form:

```
shading_flags xx
```

i.e. if the value 50 (32 + 16 + 2) is used for "xx" above, then this object can be reflective and will cast shadows, however there will be no tests for transparency, there will be no shading of the back sides of surfaces, and there will be no shadows on the surface.

NOTE: The shading flag only affects the object in which the declaration is made.   This means that if you want the shading values affected for all parts of a <u>CSG</u> object, then you will need a declaration in every component.
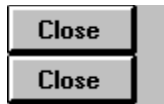
Position and Orientation Modifiers

The position, orientation, and size of an object can be modified through one of four linear transformations:

Translation

Rotation

Scaling

Shear.

Other modifications that can be made to the shape are displacement and u/v bounding.

Translation

Translation moves the position of an object by the number of units specified in the associated vector. The format of the declaration is:

```
translate <xt, yt, zt>
```

For example the declaration:

```
translate <0, 20, -4>
```

will move the entire object up 20 units along the y axis and back 4 units along the (negative) z axis.

Rotation

Rotation revolves the position of an object about the x, y, and z axes (in that order).   The amount of rotation is specified in degrees for each of the axes.   The direction of rotations follows a left-handed convention: if the thumb of your left hand points along the positive direction of an axis, then the direction your fingers curl is the positive direction of rotation.   (A negative aspect ratio in the viewpoint flips everything around to a right handed system.)

The format of the declaration is:

```
rotate <xr, yr, zr>
```

For example the declaration:

```
rotate <30, 0, 20>
```

will rotate the object by 30 degrees about the x axis, followed by 20 degrees about the z axis.
***Remember:** Left Handed Rotations.*

Scaling

Scaling alters the size of an object by a given amount with respect to each of the coordinate axes. The format of the declaration is:

```
scale <xs, ys, zs>
```

Note that using 0 for any of the components of a scale is a bad idea.   It may result in a division by zero in Polyray, causing a program crash.   Use a small number like 1.0e-5 to flatten things.   Usually you will just want to use 1 for components that don't need scaling.

Shear

A less frequently used, but occasionally useful transformation is linear shear.   Shear scales along one axis by an amount that is proportional to the location on another axis.   The format of the declaration is:

```
shear yx, zx, xy, zy, xz, yz
```

Typically only one or two of the components will be non-zero, for example the declaration:

```
shear 0, 0, 1, 0, 0, 0
```

will shear an object more and more to the right as y gets larger and larger.   The order of the letters in the declaration is descriptive, shear ... ab, ... means shear along direction a by the amount "ab" times the position b.

This declaration should probably be split into three: one that associates shear in x with the y and z values, one that associates shear in y with x and z values, and one that associates shear in z with x and y values.

You might want to look at the file "xander.pi" - this uses shear on boxes to make diagonally slanted parts of letters.

Displace

The displacement operation causes a modification of the shape of an object as it is being rendered. The amount and direction of the displacement are specified by the statement:

```
displace vector
```
or
```
displace float
```

If a vector expression is given, then Polyray will displace each vertex in the surface by the vector.   If a floating point expression is given, then Polyray will displace the surface along the direction of the normal, by the amount given in the expression.

For effective results, the value of u_steps and v_steps should be set fairly high.   Polyray subdivides the surface, then performs the displacement.   If there are too few subdivisions of the surface then the result will be a very coarse looking object.

An example of an object that has a displacement is shown below.   The displacement in the outer object is applied to each of the two spheres within.

```
object {
   object {
      sphere <0,-1.5, 0>, 2
      u_steps 32
      v_steps 64
      shiny_red
      }
 + object {
      sphere <0, 1.5, 0>, 2
      u_steps 32
      v_steps 64
      shiny_blue
      translate <0, -1.5, 0>
```

```
        rotate <-20, 0, 30>
        translate <0, 2.25, 0>
        }
    displace 0.5 * sin(5*y)
    }
```

Sample Files: disp2.pi, disp3.pi, legen.pi, spikes.pi

UV Bounds

By adjusting the value of the u/v bounds, it is possible to only render a selected portion of a surface.
The format of the declaration is:

```
uv_bounds low_u, high_u, low_v, high_v
```

For example, the following declaration will result in a wedge shaped portion of a sphere:

```
object {
    sphere <-2, 2, 0>, 1
    uv_bounds 0.3, 1.0, 0.0, 0.8
    }
```

The same effect could be achieved through the use of CSG with one or more clipping planes.   For
most purposes (e.g., creating a hemisphere by setting u_low to 0.5) the uv_bounds will easier to
create and faster to render.

Sample File: uvtst.pi

Position and Orientation Topics

Object Modifiers

Close
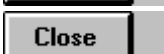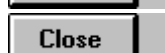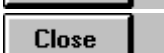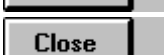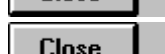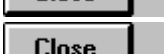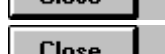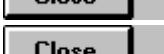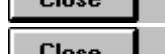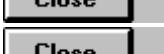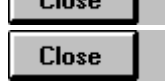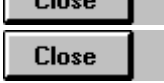
Close # Primitives

Primitives are the lowest level of shape description.   Typically a scene will contain many primitives, appearing either individually or as aggregates using either Constructive Solid Geometry (CSG) operations or gridded objects.

The primitive shapes that can be used in Polyray include the following:

Close	Bezier Patch	**Close**	Lathe Surface
Close	Blob	**Close**	NURBS
Close	Box	**Close**	Parabola
Close	Cone	**Close**	Polygon
Close	Cylinder	**Close**	Polynomial Surface
Close	Disc	**Close**	Sphere
Close	Glyph	**Close**	Sweep Surface
Close	Height Field	**Close**	Torus
Close	Implicit Surface	**Close**	Triangular Patch

Descriptions of each of these primitive shapes, as well as references to data files that demonstrate them are given in the following subsections.   Following the description of the primitives are descriptions of how CSG and grids can be built.

Bezier Patch

A Bezier patch is a form of bicubic patch that interpolates its control vertices.   The patch is defined in

terms of a 4x4 array of control vertices, as well as several tuning values.

The format of the declaration is:

```
bezier subdivision_type, flatness_value,
       u_subdivisions, v_subdivision,
       [ 16 comma-separated vertices, i.e.
          <x0, y0, z0>, <x1, y1, z1>, ..., <x15, y15, z15> ]
```

The subdivision_type and flatness_value are no longer used by Polyray.   They are retained in the declaration for backwards compatibility.

The number of levels of subdivision of the patch, in each direction, is controlled by either the u_subdivisions and v_subdivisions given in the bezier shape declaration or by the value of uv_steps (see the topic Subdivision of Primitives) if given later in the object declaration.   The more subdivisions allowed, the smoother the approximation to the patch, however storage and processing time go up.

An example of a bezier patch is:

```
object {
bezier 2, 0.05, 3, 3,<0, 0, 2>, <1, 0, 0>, <2, 0, 0>, <3, 0,-2>,
                      <0, 1, 0>, <1, 1, 0>, <2, 1, 0>, <3, 1, 0>,
                      <0, 2, 0>, <1, 2, 0>, <2, 2, 0>, <3, 2, 0>,
                      <0, 3, 2>, <1, 3, 0>, <2, 3, 0>, <3, 3,-2>
rotate <30, -70, 0>
shiny_red
}
```

Sample files: bezier0.pi, teapot.pi, teapot.inc

Blob

A blob describes a smooth potential field around one or more spherical, cylindrical, or planar components.

The format of the declaration is:

```
blob threshold:
   blob_component1
   [, blob_component2 ]
   [, etc. for each component ]
```

The threshold is the minimum potential value that will be considered when examining the interaction of the various components of the blob. Each blob component one of two forms:

```
sphere <x, y, z>, strength, radius
cylinder <x0, y0, z0>, <x1, y1, z1>, strength, radius
plane <nx, ny, nz>, d, strength, dist
```

The strength component describes how strong the potential field is around the center of the component, the "radius" component describes the maximum distance at which the component will interact with other components.   For a spherical blob component the vector <x,y,z> gives the center of the potential field around the component.   For a cylindrical blob component the vector <x0, y0, z0> defines one end of the axis of a cylinder, the vector <x1, y1, z1> defines the other end of the axis of a cylinder.   A planar blob component is defined by the standard plane equation with <nx, ny, nz> defining the normal and "d" defining the distance of the plane from the origin along the normal.

NOTE: The ends of a cylindrical blob component are given hemispherical caps.

An example of a blob is:

```
object {
    blob 0.5:
        cylinder <0,  0, 0>, <5, 0, 0>, 1, 0.7,
        cylinder <1, -3, 0>, <3, 2, 0>, 1, 1.4,
        sphere <3, -0.8, 0>, 1, 1,
        sphere <4,  0.5, 0>, 1, 1,
        sphere <1,  1,   0>, 1, 1,
        sphere <1,  1.5, 0>, 1, 1,
        sphere <1,  2.7, 0>, 1, 1

    texture {
        surface {
            color red
            ambient 0.2
            diffuse 0.8
            specular white, 1
            microfacet Phong 5
            }
        }
    }
```

Sample file: blob.pi

Box

A box is rectangular solid that has its edges aligned with the x, y, and z axes.   it is defined in terms of two diagonally opposite corners.   The alignment can be changed by rotations after the shape declaration.

The format of the declaration is:

```
box <x0, y0, z0>, <x1, y1, z1>
```

Usually the convention is that the first point is the front-lower- left point and the second is the back-upper-right point.   The following declaration is four boxes stacked on top of each other:

```
define pyramid
object {
      object { box <-1, 3, -1>, <1, 4, 1> }
    + object { box <-2, 2, -2>, <2, 3, 2> }
    + object { box <-3, 1, -3>, <3, 2, 3> }
    + object { box <-4, 0, -4>, <4, 1, 4> }
      matte_blue
      }
```

Cone

A cone is defined in terms of a base point, an apex point, and the radii at those two points.   Note that cones are not closed (you must use discs to cap them).

The format of the declaration is:

```
cone <x0, y0, z0>, r0, <x1, y1, z1>, r1
```

An example declaration of a cone is:

```
object {
    cone <0, 0, 0>, 4, <4, 0, 0>, 0
    shiny_red
    }
```

Cylinder

A cylinder is defined in terms of a bottom point, a top point, and its radius.   Note that cylinders are not closed.

The format of the declaration is:

```
cylinder <x0, y0, z0>, <x1, y1, z1>, r
```

An example of a cylinder is:

```
object {
    cylinder <-3, -2, 0>, <0, 1, 3>, 0.5
    shiny_red
    }
```

Disc

A disc is defined in terms of a center, a normal, and either a radius, or using an inner radius and an outer radius.   If only one radius is given, then the disc has the appearance of a (very) flat coin.   If two radii are given, then the disc takes the shape of an annulus (washer) where the disc extends from the first radius to the second radius.   Typical uses for discs are as caps for cones and cylinders, or as ground planes (using a really big radius).

The format of the declaration is:

```
disc <cx, cy, cz>, <nx, ny, nz>, r
```
or
```
disc <cx, cy, cz>, <nx, ny, nz>, ir, or
```

The center vector <cx,cy,cz> defines where the center of the disc is located, the normal vector <nx,ny,nz> defines the direction that is perpendicular to the disc.   i.e. a disc having the center <0,0,0> and the normal <0,1,0> would put the disc in the x-z plane with the y-axis coming straight out of the center.

An example of a disc is:

```
object {
    disc <0, 2, 0>, <0, 1, 0>, 3
```

```
      rotate <-30, 20, 0>
      shiny_red
      }
```

NOTE: A disc is infinitely thin.   If you look at it edge-on it will disappear.

Sample file: disc.pi

Glyphs (TrueType fonts)

The glyph primitive creates shapes similar to the sweep primitive.   The big difference is that straight line and curved line segments may appear in the same contour.   Additionally, the glyph can be made from several contours with exterior ones defining the outsides of the shape and interior ones defining holes within the glyph.

Typically the glyph primitive will be used in conjunction with the utility TTFX (described below) to translate TrueType font information into Polyray glyph information.   The modeler POVCAD can create Polyray glyphs, and also allows visual manipulation of the resulting object.

The following declaration creates a box with a square hole cut out of it.

```
object {
   glyph 2
      contour 4,
         <0, 0>, <4, 0>, <4, 4>, <0, 4>
      contour 4,
         <1, 1>, <1, 3>, <3, 3>, <3, 1>
   texture { shiny { color red reflection 0.2 } }
   translate <-2, 0, 0>
   }
```

The default placement of glyph coordinates is in the x-y plane and the glyph has a depth of one in z (starting at z=0 and going to z=1).   To change the depth, just use something like: scale <1, 1, 0.2>.

Each entry in a contour is a 2D point.   If a non-zero z component is added then the point is assumed to be an off-curve point and will create a curved segment within the contour.   For example, the following declaration makes a somewhat star shaped contour with sharp corners for the inner parts of the star and rounded curves for the outer parts of the star:

```
object {
glyph 1 contour 14,
   <r0*cos( 1*dt),r0*sin( 1*dt)>,<r1*cos( 2*dt),r1*sin( 2*dt),1>,
   <r0*cos( 3*dt),r0*sin( 3*dt)>,<r1*cos( 4*dt),r1*sin( 4*dt),1>,
   <r0*cos( 5*dt),r0*sin( 5*dt)>,<r1*cos( 6*dt),r1*sin( 6*dt),1>,
   <r0*cos( 7*dt),r0*sin( 7*dt)>,<r1*cos( 8*dt),r1*sin( 8*dt),1>,
   <r0*cos( 9*dt),r0*sin( 9*dt)>,<r1*cos(10*dt),r1*sin(10*dt),1>,
<r0*cos(11*dt),r0*sin(11*dt)>,<r1*cos(12*dt),r1*sin(12*dt),1>,
   <r0*cos(13*dt),r0*sin(13*dt)>,<r1*cos(14*dt),r1*sin(14*dt),1>
   }
```

The program TTFX.EXE has been included to help with the conversion of TrueType fonts from their .TTF format (typically found in the /WINDOWS/SYSTEM directory) into the sort of declaration that Polyray can understand.   For example:

```
   ttfx \windows\system\times.ttf Foo > temp.pi
```
or,
```
   ttfx \windows\system\times.ttf Foo 0.2 > temp.pi
```

By default, the characters in the string being converted (the word Foo above) are packed right next to each other. If a number follows the text string, then the characters are separated by that amount. A spacing of 0.2 has worked pretty well for the fonts I've tried.

If you then add the following line to the top of temp.pi (leaving everything else like viewpoint to their defaults)

```
light <0, 100, -100>
```

and render it with

```
polyray temp.pi -r 1
```

you will get a nice rendered image of the word Foo.

> **Note: the combination of TTFX or POVCAD, Polyray raw triangle output, and RAW2POV is a way you can create TrueType characters for use in a number of renderers.**

Sample files: timestst.pi, glyph1.pi, glyph2.pi, glyph3.pi, wingtst.pi

Implicit Surface

The format of the declaration is:

```
function f(x,y,z)
```

The function f(x,y,z) may be any algebraic expression composed of the variables: x, y, z, a numerical value (e.g. 0.5), the operators: +, -, *, /, ^, and the functions: cos, cosh, exp, fabs, ln, log, sin, sinh, tan, tanh.   The code is not particularly fast, nor is it totally accurate, however the capability to ray-trace such a wide class of functions by a SW program is (I believe) unique to Polyray.

The distance along the ray that solutions will be found in are determined by the following:

Close   If there is a bounding box, then the entry and exit points of the ray with the box will be used as the search interval.

Close   If there is no bounding box then the interval from 0.01 to 100 will be used.

Close   Solutions must be more than 1.0e-4 units distant from each other.

> **NOTE: The absolute value can be written with vertical bars, i.e. |x|^0.75.**

The following object is taken from "sombrero.pi" and is a surface that looks very much like diminishing ripples on the surface of water:

```
define a_const 1.0
define b_const 2.0
define c_const 3.0
define two_pi_a 2.0 * 3.14159265358 * a_const

// Define a diminishing cosine surface (sombrero)
object {
   function y - c_const * cos(two_pi_a * sqrt(x^2 + z^2)) *
                          exp(-b_const * sqrt(x^2 + z^2))
   matte_red
   bounds object { box <-4, -4, -4>, <4, 4, 4> }
   }
```

It is quite important to have the bounding box declaration within the object where the function is declared.   If that isn't there, Polyray will be unable to determine where to look for intersections with the surface.   (The bounding box of an implicit function defaults to <-1, -1, -1>, <1, 1, 1>.)

Sample files: noisefn.pi, sinsurf.pi, sectorl.pi, sombrero.pi, superq.pi, zonal.pi

Lathe Surfaces

A lathe surface is a polygon that has been revolved about the y- axis.   This surface allows you to build objects that are symmetric about an axis, simply by defining 2D points.

The format of the declaration is:

```
lathe type, direction, total_vertices,
    <vert1.x,vert1.y,vert1.z>
    [, <vert2.x, vert2.y, vert2.z>]
    [, etc. for total_vertices vertices]
```

The value of "type" is either 1, or 2.   If the value is 1, then the surface will simply revolve the line segments.   If the value is 2, then the surface will be represented by a spline that approximates the line segments that were given.   A lathe surface of type 2 is a very good way to smooth off corners in a set of line segments.

The value of the vector "direction" is used to change the orientation of the lathe.   For a lathe surface that goes straight up and down the y-axis, use <0, 1, 0> for "direction.   For a lathe surface that lies on the x-axis, you would use <1, 0, 0> for the direction.

Sample file: lathe1.pi, lathe2.pi

Note that CSG will really only work correctly if you close the lathe - that is either make the end point of the lathe the same as the start point, or make the x-value of the start and end points equal zero. Lathes, unlike polygons are not automatically closed by Polyray.

NOTE: Since a splined lathe surface (type = 2) is a 4th order polynomial, it is possible to specify which quartic root solver to use.   See the topic Polynomial Surface for a description of the "root_solver" statement.

NURBS

Polyray supports the general patch type, Non-Uniform Rational B-Splines (NURBS).   All that is described here is how they are declared and used in Polyray.   For further background and details on NURBS, refer to the literature.

They are declared with the following syntax:

```
nurb u_order, u_points, v_order, v_points,
    u_knots, v_knots, uv_mesh
```
or
```
nurb u_order, u_points, v_order, v_points, uv_mesh
```

Where each part of the declaration has the following format and definition,

Name	Format	Definition
u_order	integer	One more than the power of the spline in the u direction. (If u_order = 4, then it will be a cubic patch.)
u_points	integer	The number of vertices in each row of the patch mesh
v_order	integer	One more than the power of the spline in v direction.

v_points	integer	The number of rows in the patch mesh
u_knots	[...]	Knot values in the u direction
v_knots	[...]	Knot values in the v direction
uv_mesh	[...]	An array of arrays of vertices.

Each vertex may have either three or four [...]] components.   If the fourth component is set then the spline will be rational.   If the vertex has only three components then the homogenous (fourth) component is assumed to be one. The homogenous component must be greater than 0.

For example, the following is a complete declaration of a NURB patch:

```
object {
 nurb 4, 6, 4, 5,
       [0, 0, 0, 0, 1.5, 1.5, 3, 3, 3, 3], // Non-uniform knots
       [0, 0, 0, 0, 1, 2, 2, 2, 2],          // Uniform open knots
       [[<0,0,0>, <1,0, 3>, <2,0,-3>,     <3,0, 3>, <4,0,0>],
       [<0,1,0>, <1,1, 0>, <2,1, 0>,     <3,1, 0>, <4,1,0>],
       [<0,2,0>, <1,2, 0>, <2,2, 5,2>,   <3,2, 0>, <4,2,0>],
       [<0,3,0>, <1,3, 0>, <2,3, 5,0.5>, <3,3, 0>, <4,3,0>],
       [<0,4,0>, <1,4, 0>, <2,4, 0>,     <3,4, 0>, <4,4,0>],
       [<0,5,0>, <1,5,-3>, <2,5, 3>,     <3,5,-3>, <4,5,0>]]
    translate <-2, -2.5, 0>
    rotate <-90, -30, 0>
    uv_steps 32, 32
    shiny_red
    }
```

The preceding patch was both non-uniform and rational.   If you don't want to bother declaring the knot vector you can simply omit it.   This will result in a open uniform B-Spline patch.   Most of the time the non-uniform knot vectors are unnecessary and can be safely omitted.   The preceding declaration with uniform knot vectors, and non-rational vertices could then be declared as:

```
object {
   nurb 4, 6, 4, 5,
   [[< 0, 0, 0>, < 1, 0, 3>, < 2, 0,-3>, < 3, 0, 3>, < 4, 0, 0>],
   [< 0, 1, 0>, < 1, 1, 0>, < 2, 1, 0>, < 3, 1, 0>, < 4, 1, 0>],
   [< 0, 2, 0>, < 1, 2, 0>, < 2, 2, 5>, < 3, 2, 0>, < 4, 2, 0>],
   [< 0, 3, 0>, < 1, 3, 0>, < 2, 3, 5>, < 3, 3, 0>, < 4, 3, 0>],
   [< 0, 4, 0>, < 1, 4, 0>, < 2, 4, 0>, < 3, 4, 0>, < 4, 4, 0>],
   [< 0, 5, 0>, < 1, 5,-3>, < 2, 5, 3>, < 3, 5,-3>, < 4, 5, 0>]]
   translate <-2, -2.5, 0>
   rotate <-90, -30, 0>
   uv_steps 32, 32
   shiny_red
}
```

Note that internally NURBS are stored as triangles.   This can result in a high memory usage for a finely diced NURB (uv_steps large).

Sample files: nurb0.pi, nurb1.pi

Parabola
A parabola is defined in terms of a bottom point, a top point, and its radius at the top.

The format of the declaration is:

```
parabola <x0, y0, z0>, <x1, y1, z1>, r
```

The vector <x0,y0,z0> defines the "top" of the parabola - the part that comes to a point.   The vector <x1,y1,z1> defines the "bottom" of the parabola, the width of the parabola at this point is "r".

An example of a parabola declaration is:

```
object {
    parabola <0, 6, 0>, <0, 0, 0>, 3
    translate <16, 0, 16>
    steel_blue
    }
```

This is sort of like a salt shaker shape with a rounded top and the base on the x-z plane.

Parametric Surface

A parametric surface allows the creation of surfaces as a mesh of triangles.   By defining the vertices of the mesh in terms of functions of u and v, Polyray will automatically create the entire surface.   The smoothness of the surface is determined by the number of steps allowed for u and v.

The mesh defaults to 0 <= u <= 1, and 0 <= v <= 1.   By explicitly defining the uv_bounds for the surface it is possible to create only the desired parts of the surface.

The format of the declaration is:

```
parametric <fx(u,v), fy(u,v), fz(u,v)>
```

For example, the following declarations could be used to create a torus:

```
define r0 1.25
define r1 0.5

define torux (r0 + r1 * cos(v)) * cos(u)
define toruy (r0 + r1 * cos(v)) * sin(u)
define toruz r1 * sin(v)

object {
    parametric <torux,toruy,toruz>
    rotate <-20, 0, 0>
    shiny_red
    uv_bounds 0, 2*pi, 0, 2*pi
    uv_steps 16, 8
    }
```

Polygon

Although polygons are not very interesting mathematically, there are many sorts of objects that are much easier to represent with polygons.   Polyray assumes that all polygons are closed and automatically adds a side from the last vertex to the first vertex.

The format of the declaration is:

```
polygon total_vertices,
    <vert1.x,vert1.y,vert1.z>
    [, <vert2.x, vert2.y, vert2.z>]
    [, etc. for total_vertices vertices]
```

As with the sphere, note the comma separating each vertex of the polygon.

I use polygons as a floor in a lot of images.   They are a little slower than the corresponding plane, but for scan conversion they are a lot easier to handle.   An example of a checkered floor made from a polygon is:

```
object
   {
   polygon 4, <-20, 0, -20>, <-20, 0,  20>,
              < 20, 0,  20>, < 20, 0, -20>
   texture {
      checker matte_white, matte_black
      translate <0, -0.1, 0>
      scale <2, 1, 2>
      }
   }
```

Polynomial Surface

The format of the declaration is:

```
polynomial f(x,y,z)
```

The function f(x,y,z) must be a simple polynomial, i.e., $x^2+y^2+z^2-1.0$ is the definition of a sphere of radius 1 centered at (0,0,0).   See the topic Processing Polynomial Expressions for a little more detail on restrictions on the form of the polynomial.

For quartic equations, there are three available ways to solve for roots, by specifying which one is desired, it is possible to tune for quality or speed.   The method of Ferrari is the fastest, but also the most numerically unstable.   By default the method of Vieta is used.   Sturm sequences (which are the slowest) should be used where the highest quality is desired.

The declaration of which root solver to use takes one of the forms:

```
root_solver Ferrari
root_solver Vieta
root_solver Sturm
```

(Capitalization is important - these are proper nouns after all.)

> **NOTE: Due to unavoidable numerical inaccuracies, not all polynomial surfaces will render correctly from all directions.**

The following example, taken from "devil.pi" defines a quartic polynomial.   The use of the CSG clipping object is to trim uninteresting parts of the surface.   The bounding box declaration helps the scan conversion routines figure out where to look for the surface.

```
// Variant of a devil's curve in 3-space.  This figure has a
// top and bottom part that are very similar to a hyperboloid
// of one sheet, however the central region is pinched in the
// middle leaving two teardrop shaped holes.
object {
    object {
       polynomial x^4 + 2*x^2*z^2 - 0.36*x^2 - y^4 +
                  0.25*y^2 + z^4
       root_solver Ferrari
       }
    & object { box <-2, -2, -0.5>, <2, 2, 0.5> }
    bounding_box <-2, -2, -0.5>, <2, 2, 0.5>
```

```
    rotate <10, 20, 0>
    translate <0, 3, -10>
    shiny_red
    }
```

> **NOTE: As the order of the polynomial goes up, the numerical accuracy required to render the surface correctly also goes up. One problem that starts to rear its ugly head starting at around 3rd to 4th order equations is a problem with determining shadows correctly.   The result is black spots on the surface.   You can avoid this problem to a certain extent by making the value of "shadow_tolerance" larger.   For 4th and higher equations, you will want to use a value of at least 0.05, rather than the default 0.001.**

Sample file: torus.pi, many others

Sphere

Spheres are the simplest 3D object to render and a sphere primitive enjoys a speed advantage over most other primitives.

The format of the declaration is:

```
sphere <center.x, center.y, center.z>, radius
```

Note the comma after the center vector, it really is necessary.

My basic benchmark file is a single sphere, illuminated by a single light.   The definition of the sphere is:

```
object {
    sphere <0, 0, 0>, 2
    shiny_red
    }
```

Sample file: sphere.pi

Sweep Surface

A sweep surface, also referred to as an extruded surface, is a polygon that has been swept along a given direction.   It can be used to make multi-sided beams, or to create ribbon-like objects.

The format of the declaration is:

```
sweep type, direction, total_vertices,
    <vert1.x,vert1.y,vert1.z>
    [, <vert2.x, vert2.y, vert2.z>]
    [, etc. for total_vertices vertices]
```

The value of "type" is either 1, or 2.   If the value is 1, then the surface will be a set of connected squares.   If the value is 2, then the surface will be represented by a spline that approximates the line segments that were given.

The value of the vector "direction" is used to change the orientation of the sweep.   For a sweep surface that is extruded straight up and down the y-axis, use <0, 1, 0> for "direction.   The size of the vector "direction" will also affect the amount of extrusion; i.e., if |direction| = 2, then the extrusion will be two units in that direction.

An example of a sweep surface is:

```
// Sweep made from connected quadratic splines.
object {
```

```
sweep 2, <0, 2, 0>, 16,
    <0, 0>, <0, 1>, <-1, 1>, <-1, -1>, <2, -1>, <2, 3>,
    <-4, 3>, <-4, -4>, <4, -4>, <4, -11>, <-2, -11>,
    <-2, -7>, <2, -7>, <2, -9>, <0, -9>, <0, -8>
translate <0, 0, -4>
scale <1, 0.5, 1>
rotate <0,-45, 0>
translate <10, 0, -18>
shiny_yellow
}
```

Sample file: sweep1.pi, sweep2.pi

> **NOTE: CSG will really only work correctly if you close the sweep - that is make the end point of the sweep the same as the start point.   Sweeps, unlike polygons are not automatically closed by Polyray.**

Torus

The torus primitive is a doughnut shaped surface that is defined by a center point, the distance from the center point to the middle of the ring of the doughnut, the radius of the ring of the doughnut, and the orientation of the surface.

The format of the declaration is:

```
torus r0, r1, <center.x, center.y, center.z>,
           <dir.x, dir.y, dir.z>
```

As an example, a torus that has major radius 1, minor radius 0.4, and is oriented so that the ring lies in the x-z plane would be declared as:

```
object {
    torus 1, 0.4, <0, 0, 0>, <0, 1, 0>
    shiny_red
    }
```

> **NOTE: Since a torus is a 4th order polynomial, it is possible to specify which quartic root solver to use.   See the topic Polynomial Surface for a description of the "root_solver" statement.**

Sample file: torus.pi

Triangular Patch

A triangular patch is defined by a set of vertices and their normals. When calculating shading information for a triangular patch, the normal information is used to interpolate the final normal from the intersection point to produce a smooth shaded triangle.

The format of the declaration is:

```
patch <x0,y0,z0>, <nx0,ny0,nz0>,   [UV u0, v0,]
      <x1,y1,z1>, <nx1,ny1,nz1>,   [UV u1, v1,]
      <x2,y2,z2>, <nx2,ny2,nz2> [, UV u2, v2]
```

The vertices and normals are required for each of the three corners of the patch.   The u/v coordinates are optional.   If they are omitted, then they will be set to the following values:

```
u0 = 0, v0 = 0
u1 = 1, v1 = 0
```

```
u2 = 0, v1 = 1
```

Smooth patch data is usually generated as the output of another program.

Primitives

Bezier Patch	Lathe Surface
Blob	NURBS
Box	Parabola
Cone	Polygon
Cylinder	Polynomial Surface
Disc	Sphere
Glyph	Sweep Surface
Height Field	Torus
Implicit Surface	Triangular Patch

Close

Close

Height Field

There are two ways that height fields can be specified, either by using data stored in a Targa file, or using an implicit function of the form y = f(x, z).

The default orientation of a height field is that the entire field lies in the square
 0 <= x <= 1, 0 <= z <= 1.
File based height fields are always in this orientation, implicit height fields can optionally be defined over a different area of the x-z plane.   The height value is used for y.

File Based Height Fields

Height fields data can be read from a Targa, GIF, or JPEG format file.   A GIF image will be treated as an 8 bit format image as described below.   Any color information in the GIF file is ignored.

Note that if you use JPEG images, a grayscale JPEG will be treated as an 8 bitformat and a color JPEG will be treated as a 24 bit format height field.   Due to the lossy nature of JPEG, it is extremely unlikely that a color JPEG will be useful as a height field.   It is possible that reasonable results can be obtained using grayscale JPEG images as height fields (no guarantees).

By using "smooth_" in front of the declaration, an extra step is performed that calculates normals to the height field at every point within the field.   The result of this is a greatly smoothed appearance, at the expense of around three times as much memory being used.

The format of the declaration is:

```
height_field "filename"
smooth_height_field "filename"
```

The sample program "wake.c" generates a Targa file that simulates the wake behind a boat.

Sample files: wake.pi, spiral.pi, islands.pi

8 Bit Format

Each pixel in the file (Type 3 Targa, raw greyscale image) is represented by a single byte.   The byte is treated as a signed integer, giving a range of values from -127 to 128.

16 Bit Format

Each pixel in the file is represented by two bytes, low then high.   The high component defines the integer component of the height, the low component holds the fractional part scaled by 255. The entire value is offset by 128 to compensate for the unsigned nature of the storage bytes.   As an example the values   high = 140, low = 37 would be translated to the height:

```
(140 + 37 / 256) - 128 = 12.144
```

similarly if you are generating a Targa file to use in Polyray, given a height, add 128 to the height, extract the integer component, then extract the scaled fractional component.   The following code fragment shows a way of generating the low and high bytes from a floating point number.

```
unsigned char low, high;
float height;
FILE *height_file;

...

height += 128.0;
high = (unsigned char)height;
height -= (float)high;
low = (unsigned char)(256.0 * height);
fputc(low, height_file);
fputc(high, height_file);
```

24 Bit Format

The red component defines the integer component of the height, the green component holds the fractional part scaled by 255, the blue component is ignored.   The entire value is offset by 128 to compensate for the unsigned nature of the RGB   values.   As an example the values   r = 140, g = 37, and b = 0 would be translated to the height:

```
(140 + 37 / 256) - 128 = 12.144
```

similarly if you are generating a Targa file to use in Polyray, given a height, add 128 to the height, extract the integer component, then extract the scaled fractional component.   The following code fragment shows a way of generating the RGB components from a floating point number.

```
unsigned char r, g, b;
float height;
FILE *height_file;

...

height += 128.0;
r = (unsigned char)height;
height -= (float)r;
g = (unsigned char)(256.0 * height);
b = 0;
fputc(b, height_file);
fputc(g, height_file);
fputc(r, height_file);
```

32 Bit Format

The four bytes of the 32 bit Targa image are used to hold the four bytes of a floating point number. The format of the floating point number is machine specific, and the order of the four bytes in the image correspond exactly to the order of the four bytes of the floating number when it is in memory.

For example, the following code shows how to take a floating point number and store it to a file in the format that Polyray will expect.   You will also need to write the Targa header, etc.

```
unsigned char *byteptr;
float depth;
FILE *ofile;

calculations for depth ...

/* Store a floating point number as a 32 bit color- this is obviously a
machine specific result for the floating point number that is stored.
There is also an assumption here that a float type is exactly 4 bytes
and that the size of an unsigned char is exactly 1 byte. */
byteptr = (unsigned char *)&depth;
fputc(byteptr[0], ofile);
fputc(byteptr[1], ofile);
fputc(byteptr[2], ofile);
fputc(byteptr[3], ofile);

...
```

Implicit Height Fields

Another way to define height fields is by evaluating a mathematical function over a grid.   Given a function y = f(x, z), Polyray will evaluate the function over a specified area and generate a height field based on the function.   This method can be used to generate images of many sorts of functions that are not easily represented by collections of simpler primitives.

The valid formats of the declaration are:

```
height_fn xsize, zsize, min_x, max_x, min_z, max_z, expression
height_fn xsize, zsize, expression

smooth_height_fn xsize, zsize, min_x, max_x, min_z, max_z, expression

smooth_height_fn xsize, zsize, expression
```

If the four values min_x, max_x, min_z, and max_z are not defined then the default square 0 <= x <= 1, 0 <= z <= 1 will be used.

For example:

```
// Define constants for the sombrero function
define a_const 1.0
define b_const 2.0
define c_const 3.0
define two_pi_a 2.0 * 3.14159265358 * a_const

// Define a diminishing cosine surface (sombrero)
object {
   height_fn 80, 80, -4, 4, -4, 4,
      c_const * cos(two_pi_a * sqrt(x^2 + z^2)) *
               exp(-b_const * sqrt(x^2 + z^2))
   shiny_red
   }
```

will build a height field 80x80, covering the area from -4 <= x <= 4, and -4 <= z <= 4.

Compare the run-time performance and visual quality of the sombrero function as defined in "sombfn.pi" with the sombrero function as defined in "sombrero.pi".   The former uses a height field

representation and renders quite fast.   The latter uses a very general function representation and gives smoother but very slow results.

Sample file: sombfn.pi, sinfn.pi

Close # Constructive Solid Geometry (CSG)

Objects can be defined in terms of the union, intersection, and inverse of other objects.   The operations and the symbols used are:

csgexper + csgexper	Union
csgexper * csgexper	Intersection
csgexper - csgexper	Difference
csgexper & csgexper	Clip the first object by the second
~csgexper	Inverse
(csgexper)	Parenthesised expression

Union simply collects two objects together.   Union is a convenient way to group objects in such a way as to be able to transform them as a group.   Intersection keeps only those parts of each object that are "inside" the other object.   Difference keeps all parts of the two surfaces that are inside the first object and outside the second.   Clipping removes all parts of the first object that are outside the second.   The inverse operation inverts the "inside" to be the outside and vice-versa.   (i.e. a - b is the same as a * ~b).

Note that intersection and difference require a clear inside and outside.   Not all primitives have well defined sides.   Those that do are:

Cylinders, Cones, Discs, Height Fields, Lathes, Parabola, Polygons, and Sweeps.

Using Cylinders, Cones, and Parabolas works correctly, but the open ends of these surfaces will also be open in the resulting CSG.   To close them off you can use a disc shape.

Using Discs and Polygons in a CSG is really the same as doing a CSG with the plane that they lie in. If fact, a large disc is an excellent choice for clipping or intersecting an object, as the inside/outside test is very fast.

Lathes, and Sweeps use Jordan's rule to determine if a point is inside.   This means that given a test point, if a line from that point to infinity crosses the surface an odd number of times, then the point is inside.   The net result is that if the lathe (or sweep) is not closed, then you may get odd results in a CSG intersection (or difference).

CSG involving height fields only works within the bounds of the field.

As an example, the following file will generate a view of a sphere of radius 1 with a hole of radius 0.5 through the middle:

```
define cylinder_z object { cylinder <0,0,-1.1>, <0,0,1.1>, 0.5 }
define unit_sphere object { sphere <0, 0, 0>, 1 }
```

```
// Define a CSG shape by deleting a cylinder from a sphere
object {
  unit_sphere - cylinder_z
  shiny_red
  }
```

Sample files: csg1.pi, csg2.pi, csg3.pi, roman.pi, lens.pi, polytope.pi

Gridded Objects

A gridded object is a way to compactly represent a rectangular arrangement of objects by using an image map. Each object is placed within a 1x1 cube that has its lower left corner at the location <i, 0, j> and its upper right corner at <i+1, 1, j+1>. The color index of each pixel in the image map is used to determine which of a set of objects will be placed at the corresponding position in space.

The gridded object is much faster to render than the corresponding layout of objects. The major drawback is that every object must be scaled and translated to completely fit into a 1x1x1 cube that has corners at <0,0,0> and <1,1,1>.

The size of the entire grid is determined by the number of pixels in the image. A 16x32 image would go from 0 to 16 along the x- axis and the last row would range from 0 to 16 at 31 units in z out from the x-axis.

The format of the declaration is:

```
gridded "image.tga",
    object1
    object2
    object3
    ...
```

An example of how a gridded object is declared is:

```
define tiny_sphere object { sphere <0.5, 0.4, 0.5>, 0.4 }
define pointy_cone object { cone <0.5, 0.5, 0.5>, 0.4,
                                 <0.5, 1, 0.5>, 0 }
  object {
  gridded "grdimg0.tga",
      tiny_sphere { shiny_coral }
      tiny_sphere { shiny_red }
      pointy_cone { shiny_green }
  translate <-10, 0, -10>
  rotate <0, 210, 0>
```

In the image "grdimg0.tga", there are a total of 3 colors used, every pixel that uses color index 0 will generate a shiny "coral" colored sphere, every pixel that uses index will generate a red sphere, every pixel that uses index 2 will generate a green cone, and every other color index used in the image will leave the corresponding space empty.

The normal image format for a gridded object is either grayscale or color mapped. To determine which object will be used, the value of the pixel itself in the grayscale image and the color index is used in the mapped image. If a 16 bit Targa is used, then the second byte of the color is used. If a 24 bit Targa is used, then the value of the red component is used. This limits the # of objects for all images to 256.

A color JPEG is treated the same as a 24 bit Targa (unlikely to be useful, due to the lossy nature of JPEG).

Sample files: grid.pi, river.pi

Particle Systems

There are two distinct pieces of information that are used by Polyray to do particles. The first is a particle declaration, this is the particle generator. The second is the particle itself. It is important to retain the distinction, you generally only want to have one particle generator, but you may want that

generator to produce many particles.

The form of the declaration for a particle generator is:

```
particle{
   object "name"
   position vector
   velocity vector
   acceleration vector
   birth float
   death float
   count float
   avoid float
}
```

The wrapper for the particle must be the name of an object appearing in a define statement.   If there isn't an object with that name, Polyray will write an error message and abort.   Everything else is optional, but if you don't set either the velocity or acceleration then the object will just sit at the origin.

The default values for each component are:

```
position     - <0, 0, 0>
velocity     - <0, 0, 0>
acceleration - <0, 0, 0>
birth        - frame == start_frame
death        - false (never dies)
count        - 1
avoid        - false (doesn't bounce)
```

As an example, the following declaration makes a starburst of 50 spheres.   Note the conditional before the particle declaration.   You almost always want to do this, otherwise you will create a new particle generator at every frame of the animation. Since the default birth condition is to generate particles only on the first frame, the only side effect here would be to suck up more memory every frame to retain the particle generator definition.

```
frame_time 0.05

define gravity -1

// Star burst
if (frame == start_frame)
particle {
position <0, 5, 0>
velocity brownian(<0, 0, 0>, <1, 1, 1>)
acceleration gravity
object "bsphere"
count 50
}
```

The value in the velocity declaration generates a random vector that ranges from <-1, -1, -1> to <1, 1, 1>.   Each particle of the 50 is given a different initial velocity, which is what gives the bursting effect.

An additional declaration, frame_time xx, has been added specifically for tuning particle animations. This declaration determines how much time passes for every frame.   Each particle starts with an age of 0, after each frame it's age is incremented by the value xx in the frame_time declaration. Additionally the position and velocity of the particle is updated after every frame according to the

formula:

```
V = V + frame_time * A
P = P + frame_time * V
```

The status of a particle is set by making use of some of the standard Polyray variables.   The names and meanings are:

P	Current location of the particle as a vector
x	X location of the particle (or P[0])
y	Y location of the particle (or P[1])
z	Z location of the particle (or P[2])
I	Current velocity of the particle as a vector
u	Age of the particle (frame_time * elapsed frames since birth)

These values are used in two situations, when checking the death condition of a particle and when calculating the acceleration of the particle.

If an avoid statement is given then before every frame the position of the particle is checked to see if it hits any objects in the scene (non-particle objects).   If so, then the particle will bounce off the object.

Note: See the topic Lazy Evaluation for information on lazy evaluation of expressions.   This is necessary for some particle systems to behave correctly.

Sample files: part1.pi, part2.pi, part3.pi, part6.pi

Bounding Slabs

For scenes with large numbers of small objects, there are optimization tricks that can take advantage of the distribution of the objects. An easy one to implement, and one that often results in huge time savings are bounding slabs. The basic idea of a bounding slab is to sort all objects along a particular direction. Once this sorting step has been performed, then during rendering the ray can be quickly checked against a slab (which can represent many objects rather than just one) to see if intersection tests need to be made on the contents of the slab.

The most important caveats with Polyray's implementation of bounding slabs are:

Scenes with only a few large objects will derive little speed benefits.

If there is a lot of overlap of the positions of the objects in the scene, then there will be little advantage to the slabs.

If the direction of the slabs does not correspond to a direction that easily sorts objects, then there will be little speed gained.

However, for data files that are generated by another program, the requirements for effective use of bounding slabs are often met. For example, most of the data files generated by the SPD programs will be rendered orders of magnitude faster with bounding slabs than without.

Slabs are turned on by default. To turn them off either use the command line flag, `-O 0`, or add the line, `optimizer none`, to polyray.ini.

Color and Lighting

The color space used in Polyray is RGB, with values of each component specified as a value from 0 -> 1. The way the color and shading of surfaces is specified is described in the following sections.

RGB colors are defined as either a three component vector, such as <0.6, 0.196078, 0.8>, or as one of the X11R3 named colors (which for the value given is DarkOrchid). One of these days when I feel like typing and not thinking (or if I find them on line), I'll put in the X11R4 colors.

The coloring of objects is determined by the interaction of lights, the shape of the surface it is striking, and the characteristics of the surface itself.
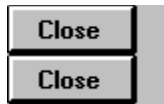
Light Sources

Light sources are one of: simple positional light sources, spot lights, or textured lights.   None of these lights have any physical size.   The lights do not appear in the scene, only the effects of the lights.

Positional Lights

A positional light is defined by its RGB color and its XYZ position.

The format of the declaration is:

```
light color, location
```
or
```
light location
```

The second declaration will use white as the color.

Spot Lights

The format of the declaration is:

```
spot_light color, location, pointed_at, Tightness, Angle, Falloff
```
or
```
spot_light location, pointed_at
```

The vector "location" defines the position of the spot light, the vector "pointed_at" defines the point at which the spot light is directed.   The optional components are:

Color	The color of the spotlight
Tightness	The power function used to determine the shape of the hot spot
Angle	The angle (in degrees) of the full effect of the spot light
Falloff	A larger angle at which the amount of light falls to nothing

A sample declaration is:

```
spot_light white, <10,10,0>, <3,0,0>, 3, 5, 20
```

Sample file: spot0.pi

Textured Lights

Textured lights are an extension of point lights that use a function (including image maps) to describe the intensity & color of the light in each direction.   The format of the declaration is:

```
textured_light {
   color float
   [scale/translate/rotate/shear]
   }
```

Any color expression is allowed for the textured light, and is evaluated at run time.

A rotating slide projector light from the data file ilight.pi is shown below:

```
define block_environ
environment("one.tga", "two.tga", "three.tga","four.tga",
           "five.tga", "six.tga")
textured_light {
```

```
    color environment_map(P, block_environ)
    rotate <frame*6, frame*3, 0>
    translate <0, 2, 0>
    }
```

Sample file: environ.pi, ilight.pi

Directional Lights

The directional light means just that, light coming from some direction.

An example would be: directional_light <2, 3, -4>, giving a white light coming from the right, above, and behind the origin.

```
    directional_light color, direction
    directional_light direction
```

Depth Mapped Lights

Depth mapped lights are very similar to spotlights, in the sense that they point from one location and at another information.   The primary use for these is for doing shadowing in scan converted scenes. The format of their declaration is:

```
    depthmapped_light {
        [ angle fexper ]
        [ aspect fexper ]
        [ at vexper ]
        [ color expression ]
        [ depth "depthfile.tga" ]
        [ from vexper ]
        [ hither fexper ]
        [ up vexper ]
        }
```

You may notice that the format of the declaration is very similar to the viewpoint declaration.   This is intentional, as you will usually generate the depth information for "depthfile.tga" as the output of a run of Polyray.

To support output of depth information, a new statements was added to the viewpoint declaration, image_format.

A viewpoint declaration that will output a depth file would have the form:

```
    viewpoint {
        from [ location of depth mapped light ]
        at [ location the light is pointed at ]
        ...
        image_format 1
        }
```

Where the final statement tells Polyray to output depth information instead of color information.   Note that if the value in the image_format statement is 0, then normal rendering will occur.

If a hither declaration is used, then the value given is used as a bias to help prevent self shadowing. The default value for this bias is the value of shadow_tolerance in polyray.ini.

Sample File: room1.pi, depmap.pi

Area Lights

By adding a sphere declaration to a textured_light, it is turned into an area light.   The current implementation is a bit rough for long and narrow shadows, but in general gives very good results.   A typical declaration of an area light is:

```
textured_light {
    color white
    sphere <8, 10, 0>, 1
    }
```

Sample file: spoty.pi

| Close |

Background Color

The background color is the one used if the current ray does not strike any objects.   The color can be any vector expression, although is usually a simple RGB color value.

The format of the declaration is:

```
background <R,G,B>
```
or
```
background color
```

If no background color is set black will be used.

An interesting trick that can be performed with the background is to use an image map as the background color (it is also a way to translate from one Targa format to another).   The way this can be done is:

```
background planar_imagemap(image("test1.tga", P)
```

The background also affects the opacity channel in 16 and 32 bit Targa output images.   If any background contributes directly to the pixel (not as a result of reflection or refraction), then the attribute bit is cleared in a 16 bit color and the attribute byte is set to the percentage of the pixel that was covered by the background in a 32 bit color.

In order to extend the flexibility of the background, the meanings of various runtime variables are set uniquely.

u,x	How far across the output image (from 0 at left to 1 at right)
v,z	How far down the output image (from 0 at bottom to 1 at top)
W	<0,0,0>
P	Same as <x, 0, z>
N	Direction of the current ray
I	<0,0,0>
U	Same as <u, v, w>
w	Level of recursion (0 for eye rays, higher for reflected and refracted rays)

As an example, suppose you wanted to have an image appear in the background, but you didn't want to have the image appear in any reflections.   Then you could define the background with the following expression:

```
background (w == 0 ? planar_imagemap(img1, P) : black)
```

If you wanted to have one image in the background, and another that appears in reflections (or refracted rays), then you could use the following expression:

```
background (w == 0 ? planar_imagemap(img1, P)
spherical_imagemap(img2, N))
```

The previous background expression fits img1 exactly into the output image and maps img2 completely around the entire scene.   This might be useful if you have a map of stars that you want to appear in reflections, but still want to have an image as the background.

Global Haze

The global haze is a color that is added based on how far the ray traveled before hitting the surface. The format of the expression is:

```
haze coeff, starting_distance, color
```

The color you use should almost always be the same as the background color. The only time it would be different is if you are trying to put haze into a valley, with a clear sky above (this is a tough trick, but looks nice). A example would be:

```
haze 0.8, 3, midnight_blue
```

The value of the coeff ranges from 0 to 1, with values closer to 0 causing the haze to thicken, and values closer to 1 causing the haze to thin out.   I know it seems backwards, but it is working and I don't want to break anything.
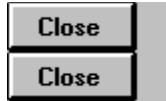
Textures

Polyray supports a few simple procedural textures: a standard shading model, a checker texture, a hexagon texture, and a general purpose (3D noise based) texture.   In addition, a very flexible (although slower) functional texture is supported.   Individual textures can be combined in various ways to create new ones.   Texture types that help to combine other textures include: layered textures, indexed textures, and summed textures.   (See the topics Indexed Textures and Texture Maps, Layered Textures and Summed Textures).

```
    texture { [texture declaration] }
or
    texture_sym
```

Where "texture_sym" is a previously defined texture declaration.

`Close`

`Close` **Procedural Textures**

Procedural textures (i.e. checker, matte_white, shiny_red, ...) are ones that are completely defined at the time the data file is read.   Textures that are evaluated during rendering are described in the topic Functional Textures.

Standard Shading Model

Unlike many other ray-tracers, surfaces in Polyray do not have a single color that is used for all of the components of the shading model.   Instead, a number of characteristics of the surface must be defined (with a matte white being the default).

A surface declaration has the form:

```
surface {
   [ surface definitions ]
   }
```

For example, the following declaration is a red surface with a white highlight, corresponding to the often seen "plastic" texture:

```
define shiny_red
texture {
   surface {
      ambient red, 0.2
      diffuse red, 0.6
      specular white, 0.8
      microfacet Reitz 10
      }
   }
```

The allowed surface characteristics that can be defined are:

color	Color if not given in another component.
ambient	Light given off by the surface
diffuse	Light reflected in all directions
specular	Amount and color of specular highlights
reflection	Reflectivity of the surface
transmission	Amount and color of refracted light
microfacet	Specular lighting model (see below)

The lighting equation used is (in somewhat simplified terms):

```
L = ambient + diffuse + specular + reflected + transmitted
```

or

```
L = Ka + Kd * (l1 + l2 + ...) + Ks * (l1 + l2 + ...) + Kr + Kt
```

Where l1, l2, ... are the lights, Ka is the ambient term, Kd is the diffuse term, Ks is the specular term, Kr is the reflective term, and Kt it the transmitted (refractive) term. Each of these terms has a scale value and a filter value (the filter defaults to white/clear if unspecified).

See the file "colors.inc" for a number of declarations of surface characteristics, including: mirror, glass, shiny, and matte.

For lots of detail on lighting models, and the theory behind how color is used in computer generated images, run (don't walk) down to your local computer bookstore and get: **"Illumination and Color in Computer Generated Imagery"**, Roy Hall, 1989 *Springer Verlag.*

Source code in the back of that book was the inspiration for the microfacet distribution models implemented for Polyray.

Note that you don't really have to specify all of the color components if you don't want to.   If the color of a particular part of the surface declaration is not defined, then the value of the "color" component will be examined to see if it was declared. If so, then that color will be used as the filter.   As an example, the declaration above could also be written as:

```
define shiny_red
texture {
   surface {
      color red
      ambient 0.2
      diffuse 0.6
      specular white, 0.8
      microfacet Reitz 10
      }
   }
```

Ambient Light

Ambient lighting is the light given off by the surface itself.   This will be a constant amount, independent of any lights that may be in the scene.

The format of the declaration is:

```
ambient color, scale
```
or
```
ambient scale
```

As always, color indicates either an RGB triple like <1.0,0.7,0.9>, or a named color.   Scale is the contribution that ambient gives to the overall amount light coming from the pixel.   The scale values should lie in the range 0.0 -> 1.0

Diffuse Light

Diffuse lighting is the light given off by the surface under stimulation by a light source.   The intensity of the diffuse light is directly proportional to the angle of the surface with respect to the light.

The format of the declaration is:

```
diffuse color, scale
```
or
```
diffuse scale
```

The only information used for diffuse calculations is the angle of incidence of the light on the surface.

Specular Highlights
The format of the declaration is:

```
specular color, scale
```
or
```
specular scale
```

The means of calculating specular highlights is by default the Phong model.   Other models are selected through the Microfacet distribution declaration.

Reflected Light
Reflected light is the color of whatever lies in the reflected direction as calculated by the relationship of the view angle and the normal to the surface.

The format of the declaration is:

```
reflection scale
```
or
```
reflection color, scale
```

Typically, only the scale factor is included in the reflection declaration, this corresponds to all colors being reflected with intensity proportional to the scale.   A color filter is allowed in the reflection definition, and this allows the modification of the color being reflected (I'm not sure if this is useful, but I included it anyway).

Transmitted Light
Transmitted light is the color of whatever lies in the refracted direction as calculated by the relationship of the view angle, the normal to the surface, and the index of refraction of the material.

The format of the declaration is:

```
transmit scale, ior
```
or
```
transmit color, scale, ior
```

Typically, only the scale factor is included in the transmitted declaration, this corresponds to all colors being transmitted with intensity proportional to the scale.   A color filter is allowed in the transmitted definition, and this allows the modification of the color being transmitted by making the transmission filter different from the color of the surface itself.

It is possible to have surfaces with colors very different than the one closest to the eye become apparent.   (See "gsphere.pi" for an example, a red sphere is in the foreground, a green sphere and a blue sphere behind.   The specular highlights of the red sphere go to yellow, and blue light is transmitted through the red sphere.)

A more complex file is "lens.pi" in which two convex lenses are lined up in front of the viewer.   The magnified view of part of a grid of colored balls is very apparent in the front lens.

Microfacet Distribution
The microfacet distribution is a function that determines how the specular highlighting is calculated for an object.

The format of the declaration is:

```
microfacet Distribution_name falloff_angle
```

or

```
    microfacet falloff_angle
```

The distribution name is one of: Blinn, Cook, Gaussian, Phong, Reitz.   The falloff angle is the angle at which the specular highlight falls to 50% of its maximum intensity.   (The smaller the falloff angle, the sharper the highlight.)   If a microfacet name is not given, then the Phong model is used.

The falloff angle must be specified in degrees, with values in the range 0 to 45.   The falloff angle corresponds to the roughness of the surface, the smaller the angle, the smoother the surface.   A very wide falloff angle will give the same sort of shading that diffuse shading gives.

> **NOTE: As stated before, look at the book by Hall.   I have found falloff values of 5-10 degrees to give nice tight highlights.   Using falloff angle may seem a bit backwards from other raytracers, which typically use a value defining the power of a cosine function to define highlight size.   When using a power value, the higher the power, the smaller the highlight. Using angles seems a little tidier since the smaller the angle, the smaller the highlight.**

Checker

The checker texture has the form:

```
    texture {
       checker texture1, texture2
       }
```

where texture1 and texture2 are texture declarations (or texture constants).

A standard sort of checkered plane can be defined with the following:

```
    // Define a matte red surface
    define matte_red
    texture {
       surface {
          ambient red, 0.1
          diffuse red, 0.5
          }
       }

    // Define a matte blue surface
    define matte_blue
    texture {
       surface {
          ambient blue, 0.2
          diffuse blue, 0.8
          }
       }

    // Define a plane that has red and blue checkers
    object {
       disc <0, 0.01, 0>, <0, 1, 0>, 5000
       texture {
          checker matte_red, matte_blue
          }
       }
```

For a sample file, look at "spot0.pi".   This file has a sphere with a red/blue checker, and a plane with a black/white checker.

> **NOTE:** A distinct problem with checkerboards is one of aliasing at great distances. To help with aliasing, either use larger checks (by adding a scale to the texture), or make sure there is a very low light level at great distances.

Hexagon

The hexagon texture is oriented in the x-z plane, and has the form:

```
texture {
   hexagon texture1, texture2, texture3
   }
```

This texture produces a honeycomb tiling of the three textures in the declaration. Remember that this tiling is with respect to the x-z plane, if you want it on a vertical wall you will need to rotate the texture.

Noise Surfaces

The complexity and speed of rendering of the noise surface type lies between the standard shading model and the special surfaces described below. It is an attempt to capture a large number of the standard 3D texturing operations in a single declaration.

A noise surface declaration has the form:

```
texture {
   noise surface {
      [ noise surface definition ]
      }
   }
```

The allowed surface characteristics that can be defined are:

color <r, g, b>	Basic surface color (used if the noise function generates a value not contained in the color map)
ambient scale	Amount of ambient contribution
diffuse scale	Diffuse contribution
specular color, scale	Amount and color of specular highlights, if the color is not given then the body color is used.
reflection scale	Reflectivity of the surface
transmission scale, ior	Amount of refracted light
microfacet kind angle	Specular lighting model (see the description of a standard surface)
color_map(map_entries)	Define the color map (for further details see the topic on <u>color map definitions</u>)
bump_scale fexper	How much the bumpiness affects the normal to the surface
frequency fexper	Affects the wavelength of ripples and waves
phase fexper	Affects the phase of the ripples and waves
lookup_fn index	Selects a predefined lookup function
normal_fn index	Selects a predefined normal modifier

octaves fexper	Number of octaves of noise to use
position_fn index	How the intersection point is used in the process of generating a noise texture
position_scale fexper	Amount of contribution of the position value to the overall texture
turbulence fexper	Amount of contribution of the noise to the overall texture.

The way the final color of the texture is decided is by calculating a floating point value using the following general formula:

index = lookup_fn(position_scale * position_fn + turbulence * noise3d(P, octaves))

The index value that is calculated is then used to lookup a color from the color map.   This final color is used for the ambient, diffuse, reflection and transmission filters.   The functions that are currently available, with their corresponding indices are:

Position functions:

Index	Effect
1	x value in the object coordinate system
2	x value in the world coordinate system
3	Distance from the z axis
4	Distance from the origin
5	Distance around the y-axis (ranges from 0 -> 1) default: 0.0

Lookup functions:

Index	Effect
1	sawtooth function, result from 0 -> 1
2	sin function, result from 0->1
3	ramp function, result from 0->1 default: no modification made

Definitions of these function numbers that make sense are:

```
define position_plain       0
define position_objectx     1
define position_worldx      2
define position_cylindrical 3
define position_spherical   4
define position_radial      5

define lookup_plain    0
define lookup_sawtooth 1
define lookup_sin      2
define lookup_ramp     3
```

An example of a texture defined this way is a simple white marble:

```
define white_marble_texture
```

```
texture {
   noise surface {
      color white
      position_fn position_objectx
      lookup_fn lookup_sawtooth
      octaves 3
      turbulence 3
      ambient 0.2
      diffuse 0.8
      specular 0.3
      microfacet Reitz 5
      color_map(
         [0.0, 0.8, <1, 1, 1>, <0.6, 0.6, 0.6>]
         [0.8, 1.0, <0.6, 0.6, 0.6>, <0.1, 0.1, 0.1>])
      }
   }
```

In addition to coloration, the bumpiness of the surface can be affected by selecting a function to modify the normal.   The currently supported normal modifier functions are:

Index	Effect
1	Make random bumps in the surface
2	Add ripples to the surface
3	Give the surface a dented appearance default: no change

Definitions that make sense are:

```
define default_normal 0
define bump_normal    1
define ripple_normal  2
define dented_normal  3
```

See also the file "texture.txt" for a little more explanation and a few more texture definitions.

Sample file: textures.inc

Indexed Textures and Texture Maps

A texture map is declared in a manner similar to color maps.   There is a list of value pairs and texture pairs, for example:

```
define index_tex_map
    texture_map ([-2, 0, red_blue_check, bumpy_green],
               [0, 2, bumpy_green, reflective_blue])
```

Note that for texture maps there is a required comma separating each of the entries.

These texture maps are complimentary to the indexed texture (see below).   Two typical uses of indexed textures are to use solid texturing functions to select (and optionally blend) between complete textures rather than just colors, and to use image maps as a way to map textures to a surface.

For example, using the texture map above on a sphere can be done accomplished with the following:

```
object {
```

```
sphere <0, 0, 0>, 2
texture { indexed x, index_tex_map }
}
```

The indexed texture uses a lookup function (in example above a simple gradient along the x axis) to select from the texture map that follows.   See the data file **indexed1.pi** for the complete example.

As an example of using an image map to place textures on a surface, the following example uses several textures, selected by the color values in an image map.   The function indexed_map returns the color index value from a color mapped image (or uses the red channel in a raw image).   The example below is equivalent to creating a material map in the POV-Ray raytracer.

```
object {
  sphere <0, 0, 0>, 1
  texture {
    indexed indexed_map(image("txmap.tga"), <x, 0, y>, 1),
             texture_map([1, 1, mirror, mirror],
                         [2, 2, bright_pink, bright_pink],
                         [3, 3, Jade, Jade])
    translate <-0.5, -0.5, 0> // center image
  }
}
```

In this example, the image is oriented in the x-y plane and centered on the origin.   The only difference between a indexed_map and a planar_imagemap is that the first (indexed_map) returns the index of the color in the image and the second returns the color itself.   Note that the texture map shown above has holes in it (between the integer values), however this isn't a problem as the indexed_map function will only produce integers.

Layered Textures

Layered textures allow you to stack multiple textures on top of each other.   If a part of the texture is not completely opaque (non-zero alpha), then the layers below will show through.   For example, the following texture creates a texture with a marble outer layer and a mirrored bottom layer and applies it to a sphere:

```
include "colors.inc"

define marble_alpha_map
   color_map([0.0, 0.2, white, 0,    white, 0]
             [0.2, 0.5, white, 0,    black, 0.2]
             [0.6, 1.0, black, 0.2, black, 1])
define mirror_veined_marble
texture {
   layered
      texture {
         special shiny { color marble_alpha_map[marble_fn] }
         },
      mirror
   }

object {
   sphere <0, 0, 0>, 2
   mirror_veined_marble
   }
```

Sample files: Stone1-Stone24, layer1.pi, layer2.pi

Summed Textures

Summed textures simply add weighted amounts of a number of textures together to make the final color.   The syntax is:

```
texture {
    summed f1, tex1, f2, tex2, ...
    }
```

The expressions f1, f2, ... are numeric expressions.   The expressions tex1, ... are textures.

Sample file: blobtx.pi

Procedural Texture Topics

Functional Texture Topics

Close

Close
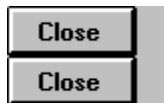
Functional Textures

The most general textures in Polyray are functional textures. These textures are evaluated at run-time based on the expressions given for the components of the lighting model.   The general syntax for a surface using a functional texture is:

```
special surface {
    [ surface declaration ]
}
```

In addition to the components usually defined in a surface declaration, it is possible to define a function that deflects the normal, and a "body" color that will be used as the color filter in the ambient, diffuse, etc. components of the surface declaration.   The format of the two declarations are:

```
    color vector
```
and
```
    normal vector
```

An example of how a functional texture can be defined is:

```
define sin_color_offset (sin(3.14*fmod(x*y*z,1)+otheta)+1)/2
define sin_color <sin_color_offset, 0, 1 - sin_color_offset>
```

```
define xyz_sin_texture
   texture {
   special surface {
      color sin_color
      ambient 0.2
      diffuse 0.7
      specular white, 0.2
      microfacet Reitz 10
      }
   }
```

In this example, the color of the surface is defined based on the location of the intersection point using the vector defined as "sin_color".

Sample files: cossph.pi, cwheel.pi.

Color Maps

Color maps are generally used in noise textures and functional textures.   They are a way of representing a spectrum of colors that blend from one into another.   Each color is represented as RGB, with an optional alpha (transparency) value.   The format of the declaration is:

```
color_map([low0, high0, <r0, g0, b0>, a0, <r1, g1, b1>, a1]
          [low1, high1, <r2, g2, b2>, a2, <r3, g3, b3>, a3]
          ...
          [lowx, highx, <rx, gx, bx>, ax, <ry, gy, by>, ay])
```

Note that there are no commas between entries in the color map even though commas are required within each entry.   (This is a holdover to retain compatibility with earlier versions of Polyray.)   If you don't need any transparency in the color map, then the following declaration could be used:

```
color_map([low0, high0, <r0, g0, b0>, <r1, g1, b1>]
          [low1, high1, <r2, g2, b2>, <r3, g3, b3>]
          ...
          [lowx, highx, <rx, gx, bx>, <ry, gy, by>])
```

In this case, the alpha is set to 0 for all colors created by the color map.   Note that it is possible to mix colors with and without alpha value.

> **Note: If there is an alpha value in the color map, then the amount of alpha is used as the scale for the transmit component of the surface.   To turn off this automatic transparency, use transmit 0.**

Using CMAPPER

A good way to build color maps for layered textures is with ColorMapper,

Written by: SoftTronics, Lutz + Kretzschmar

This is available as CMAP.ZIP in the forum Graphdev on CompuServe.   This program allows you to build color maps with varying colors and transparency values.   The output of this program does have to be massaged a little bit to make it into a color map as Polyray understands it.   In order to help with this process an IBM executable, makemap.exe has been included.   To use this little program, you follow these steps:

1) Run CMAPPER to create a color map in the standard output (not the POV-Ray output format).

2) run makemap on that file, giving a name for the new Polyray color map definition

3) Add this definition to your Polyray data file.

If you saved your map as **foo.map**, and you wanted to add this color map to the Polyray data file foo.inc, with the name of **foox_map**, you would then run makemap the following way:

```
makemap foo.map foox_map >> foo.inc
```

This makes the translation from CMAPPER format to Polyray format, and appends the output as, `define foox_map color_map(...)`, to the file foo.inc.

Image Maps

A type of coloring that can be used involves the use of an existing image projected onto a surface. There are four types of projection supported, planar, cylindrical, spherical, and environment. Input files for use as image maps may be 8, 16, 24, and 32 bit uncompressed, RLE compressed, or color mapped Targa files.

The declaration of an image map is:

```
image("imfile.tga")
```

Typically, an image will be associated with a variable through a definition such as:

```
define myimage image("imfile.tga")
```

The image is projected onto a shape by means of a "projection". The four types of projection are declared by:

```
planar_imagemap(image, coordinate [, repeat]),
cylindrical_imagemap(image, coordinate [, repeat]),
spherical_imagemap(image, coordinate)
environment_map(environment(image1, image2, image3, image4,
                                 image5, image6))
```

The planar projection maps the entire raster image into the coordinates 0 <= x <= 1, 0 <= z <= 1. The vector value given as "coordinate" is used to select a color by multiplying the x value by the number of columns, and the z value by the number of rows.

The color appearing at that position in the raster will then be returned as the result. If a "repeat" value is given then entire surface, repeating between every integer Value of x and/or z.

The cylindrical projection wraps the image about a cylinder that has one end at the origin and the other at <0, 1, 0>. If a "repeat" value is given, then the image will be repeated along the y-axis, if none is given, then any part of the object that is not covered by the image will be given the color of pixel (0, 0). The spherical projection wraps the image about an origin centered sphere. The top and bottom seam are folded into the north and south poles respectively. The left and right edges are brought together on the positive x axis.

The environment map wraps six images around a point. This method is a standard way to fake reflections by wrapping the images that would be seen from a point inside an object around the object. A sample of this technique can be seen by rendering "room1.pi" (which generates the images) followed by rendering "room0.pi" (which wraps the images around a sphere).

Following are a couple of examples of objects and textures that make use of image maps:

```
define hivolt_image image("hivolt.tga")
define hivolt_tex
texture {
   special surface {
```

```
          color cylindrical_imagemap(hivolt_image, P, 1)
          ambient 0.9
          diffuse 0.1
          }
     scale <1, 2, 1>
     translate <0, -1, 0>
     }
object { cylinder <0, -1, 0>, <0, 1, 0>, 3 hivolt_tex }
```

and

```
define disc_image image("test.tga")
define disc_tex
texture {
   special surface {
      color planar_imagemap(disc_image, P)
      ambient 0.9
      diffuse 0.1
      }
   translate <-0.5, 0, -0.5>
   scale <7*4/3, 1, 7>
   rotate <90, 0, 0>
   }
object {
   disc <0, 0, 0>, <0, 0, 1>, 6
   u_steps 10
   disc_tex
   }
```

Bump Maps

Bump maps are declared using the same sort of projections as image maps (excepting environment maps).   The following are the valid declarations of bump maps:

```
planar_bumpmap(image, coordinate [, bump size]),
   cylindrical_bumpmap(image, coordinate [, bump size]),
   spherical_bumpmap(image, coordinate [, bump size])
```

Instead of an optional repeat argument, bumpmaps have an optional bump size argument.   If this argument is left out, then the bump size is set to one.   Note that negative bump values are allowed and cause the bumps to appear to project the other way.

Any Targa image can be used, but for best results greyscale or color mapped images are best.   The following declarations show how a bump map can be applied to objects:

```
include "colors.inc"

define tile_bumps image("tile1.tga")

define bumpmap_red1
texture {
   special shiny {
      color red
      normal planar_bumpmap(tile_bumps, <8*u, 0, 6*v>, 1)
      }
   }
object {
```

```
    object { torus 2, 0.75, <0, -1.25, 0>, <0, 1, 0> }
+ object { cone <0, -2, 0>, 1, <0, 3, 0>, 0 }
+ object { sphere <2, 0, 4>, 2 }
bumpmap_red1
}
```

The bumpmap is declared using u/v coordinates so that it will follow the natural coordinates of the object.   This ensures that it wraps properly around the torus, cone, and sphere in the example above. There is an automatic wrapping of bump maps, so there will be 8 tiles in the u direction and 6 tiles in the v direction of each object.

Comments

Comments follow the standard C/C++ formats.   Multiple line comments are enclosed by /* ... */ and single line comments are preceeded by //.

Single line comments are allowed and have the following format:

```
// [ any text to end of the line ]
```

As soon as the two characters "//" are detected, the rest of the line is considered a comment.

Close ☐ **Animation Support**

An animation is generated by rendering a series of frames, numbered from 0 to some total value. The declarations in Polyray that support the generation of multiple Targa images are:

total_frames val	The total number of frames in the animation
start_frame val	The value of the first frame to be rendered
end_frame val	The last frame to be rendered
outfile "name" **outfile name**	Polyray appends the frame number to this string in order to generate distinct Targa files.

The values of "total_frames", "start_frame", and "end_frame", as well as the value of the current frame, "frame", are usable in arithmetic expressions in the input file.   Note that these statements should appear before the use of: total_frames, start_frame, end_frame, or frame as a part of an expression. Typically I put the declarations right at the top of the file.

WARNING: if the string given for "outfile" is longer than 5 characters, the three digit frame number that is appended will be truncated by DOS.   Make sure this string is short enough or you will end up overwriting image files.

Sample files: whirl.pi, plane.pi, squish.pi, many others

System Calls

The system call allows you to invoke an external program from within Polyray.   This is useful if you have a program that will generate new data for every frame of an animation.   By using system, you can invoke the external program for every frame of the program, and possibly pass it the frame number so that it will know the specific data to generate.

The format of the declaration is:

```
system(arg1, ..., argn)
```

The arguments are concatenated together (no spaces added) and then passed to the system to execute.   Any numeric or string variable is allowed for an argument.   The number of arguments isn't limited, however the total number of characters in the final string must be less than 255.

One possible use for the system call is to use DTA to extract a particular frame of an animation for use in an image map.   By incrementing the frame number that is retrieved, you can embed an animation into your own animation.

Conditional Processing

In support of animation generation (and also because I sometimes like to toggle attributes of objects), Polyray supports limited conditional processing.   The syntax for this is:

```
if (cexper) {
   [object/light/... declarations]
   }
else {
   [other object/light/... declarations]
   }
```

The sample file "rsphere.pi" shows how it can be used to modify the color characteristics of a moving sphere.

The use of conditional statements is limited to the top level of the data file.   You cannot put a conditional within an object or texture declaration. i.e.

```
object {
   if (foo == 42) {
      sphere <0, 0, 0>, 4
      }
   else {
      disc <0, 0, 0>, <0, 1, 0>, 4
      }
   }
```

is not a valid use of an "if" statement, whereas:

```
if (foo == 42) {
   object {
      sphere <0, 0, 0>, 4
      }
   }
else {
   object {
      disc <0, 0, 0>, <0, 1, 0>, 4
      }
   }
```
or:
```
if (foo == 42)
   object { sphere <0, 0, 0>, 4 }
else if (foo = 12) {
   object { torus 3.0, 1.0, <0, 0, 0>, <0, 1, 0> }
   object { cylinder <0, -1, 0>, <0, 1, 0>, 0.5 }
   }
else
   object { disc <0, 0, 0>, <0, 1, 0>, 4 }
```

are valid.

NOTE: The curly braces "{}" are required if there are multiple statements within the conditional, and not required if there is a single statement.

Include Files

In order to allow breaking an input file into several files (as well as supporting the use of library files), it is possible to direct Polyray to process another file.   The syntax is:

```
include "filename"
```

Be aware that trying to use "#include ..." will fail as Polyray will consider it a comment.

File Flush

Due to unforeseen occurrences, like power outages, or roommates that hit the reset button so they can do word processing, it is possible to specify how often the output file will be flushed to disk.   The default is to wait until the entire file has been written before a flush (which is a little quicker but you lose everything if a crash occurs).

The format of the declaration is:

```
file_flush xxx
```

The number xxx indicates the maximum number of pixels that will be written before a file flush will occur.   This value can be as low as 1 - this will force a flush after every pixel (only for the very paranoid).

About Polyray Help

The text for Polyray Help was written by Alexander Enzmann.   All material herein is copyright © 1991 - 1994, Alexander Enzmann.

CompuServe ID: Alexander Enzmann 70323,2461
Internet Address: xander@mitre.org

This Help File was produced and edited by Robert McGregor at Screaming Tiki (tm).

CompuServe ID: Robert McGregor 73122,3125

POVCAD v4 for Windows

POVCAD is a 3-D modeler for generating data files in various formats and directly exports Polyray (.PI) and POV-Ray (.POV) scene files and include (.INC) files.   Shareware US$35, not crippled.   Requires a mouse.

Features include:

Close	2-D Curve Editor and spline creation
Close	Smart Text Editor that creates editable ray tracing scene file text
Close	Texture Builder to generate texture definitions
Close	Create native primitives and RAW data files
Close	Render scene files from within POVCAD
Close	Extrusion, translation, scaling, rotatation of objects
Close	Conversion of .DXF to .RAW or .RAW to .DXF for use with ray tracers
Close	Creation of Polyray lathe/sweep objects
Close	Conversion of TrueType fonts into Polyray glyph objects
Close	Creation of animation paths
Close	Generation and editing of bezier patches
Close	Creation of custom colors
Close	Supports positional, spot, directional, and area lights
Close	Mouse controlled zoom and object manipulation
Close	Context sensitive Help
Close	Includes tutorials and much more!

For more info send e-mail to:

Alfonso Hermida, CIS 72114,2060 or   **Rob McGregor, CIS 73122,3125**

MORAY

MORAY is a GUI modeller for POV-Ray 2.x on 386/486. Supports the plane, cube, sphere, cylinder, torus, cone, disc, heightfield and bezier patch primitives, and adds tapering, rotational and translational sweeps. You can add point-, area-, and spotlights, bounding boxes, textures and cameras, which show the scene in 3D. Shareware US$59, not crippled. Requires mouse and (S)VGA. Supports VESA.

For information about MORAY send e-mail to:

Lutz Kretzschmar, CIS 100023,2006

Topics

Close

Print

Quick Reference

This is an extremely abbreviated description of the command line flags, initialization file statements, and data file declarations for Polyray...

Syntax:

polyray datafile [options]

Command Line Options:

-a mode	Antialiasing (0=none,1=corner average,2-4=adaptive)
-b pixels	Set the maximum number of pixels that will be calculated between file flushes
-B	Flush the output file every scan line
-d	Generate a depth file instead of an image file
-o filename	Output file name (default "out.tga")
-p bits/pixel	Number of bits per pixel 8/16/24/32 (default 16)
-P palette	Which palette option to use [0=grey, 1=666, 2=884]
-q flags	Turn on/off various global shading options
-Q	Abort if any key is hit during trace
-r renderer	Which rendering method: [0=raytrace, 1=scan convert, 2=wireframe, 3=raw triangle information, 4=uv triangles]
-R	Resume an interrupted trace
-s samples	# of samples per pixel when performing focal blur

-t status_vals	Status display type [0=none,1=totals,2=line,3=pixel]
-T threshold	Threshold to start oversampling (default 0.2)
-u	Write the output file in uncompressed form
-v	Trace from bottom to top
-V mode	Display mode while tracing (0=none, 1-5=8bit, 6-10=16, 11-15=24)
-W	Wait for key before clearing display
-x columns	Set the x resolution
-y lines	Set the y resolution
-z start_line	Start a trace at a specified line

Initialization File ("polyray.ini") Keywords:

abort_test	true/false/on/off
alias_threshold	[Value to cause adaptive anitaliasing to start]
antialias	none/filter/adaptive1/adaptive2
display	none/vga1...vga5/hicolor1...hicolor5/truecolor1...truecolor5
max_level	[max depth of recursion]
max_samples	[# of samples when performing focal blur]
optimizer	none/slabs
pallette	884/666/grey
pixel_size	[8, 16, 24, 32]
pixel_encoding	none/rle
renderer	ray_trace/scan_convert/wire_frame/raw_triangles/ uv_triangles
shade_flags	[default/bit mask of flags]
shadow_tolerance	[minimum distance for blocking objects]
status	none/totals/line/pixel
warnings	on/off

Any lines starting with "//" will be treated as comments & ignored.
Any lines surrounded with "/*" ... "*/" will be ignored.

Input File Syntax:

```
[viewpoint statement]

[object declaration]

[conditional statement]

define token expression

define token [object declaration]

define token [surface declaration]

define token [texture declaration]
```

```
define token texture_map([a, b, texture1, texture2]
                         ...
                         [x, y, texturei, texturej])

define token particle { [particle declarations] }

define token transform { [rotate/translate/scale/shear statements] }

total_frames val

start_frame val

end_frame val

outfile "name"

outfile name

file_flush xxx

include "filename"

system(arg1, ..., argn)

background color

background expression

haze coeff, starting_distance, color

light color, location

light location

spot_light color, location, pointed_at, Tightness, Angle, Falloff

spot_light location, pointed_at
textured_light {
   color color_expression
   [sphere center, radius]
   [rotate/translate/... statements]

   }

directional_light color, direction

directional_light direction
depthmapped_light {
   [ angle fexper ]
   [ aspect fexper ]
   [ at vexper ]
   [ color expression ]
   [ depth "depthfile.tga" ]
   [ from vexper ]
   [ hither fexper ]
   [ up vexper ]
   }
```

Expressions:

Floating point operators:
```
+, -, *, /, ^
```

Functions returning floats:
```
acos(x), asin(x), atan(x), atan2(x, y), ceil(x), cos(x), cosh(x),
```

```
degrees(x), exp(x), fabs(x), floor(x), fmod(x, y), heightmap(image, P),
indexed(image,P), legendre(l, m, x), ln(x), log(x), max(x, y), min(x,
y), noise(P), noise(P, o), noise(P, <p, n, o>), pow(x, y), radians(x),
sawtooth(x), sin(x), sinh(x), sqrt(x), tan(x), tanh(x), visible(V1, V2),
V1 . V2, |x|
```

Vector operators:
+, -, * (cross product, or float times a vector)

Functions returning vectors/colors:
```
brownian(P), brownian(P, S), color_wheel(x, y, z), dnoise(P), dnoise(P,
o), dnoise(P, <p, n, o>), rotate(V1, <xdeg, ydeg, zdeg>), rotate(V1, V2,
deg), reflect(V1, V2) trace(P, D)
```

Predefined variables:
```
u, v, x, y, z, P, W, N, I, start_frame, frame, end_frame
```

Image file manipulation:
```
environment("file1", "file2", ..., "file6")

image("file")

cylindrical_imagemap(image, V [, repeat flag])

planar_imagemap(image, V [, repeat flag])

spherical_imagemap(image, V [, repeat flag])

heightmap(image, V)

indexed_map(image, V [, repeat flag])

environment_map(V, environment)
```

Color map:
```
color_map([v0, v1, Color0, Color1]
          [v2, v3, Color2, Color3]
          ...
          [vx, vy, Colorx, Colory])

color_map([v0, v1, Color0, alpha0, Color1, alpha1]
          [v2, v3, Color2, alpha2, Color3]
          ...
          [vx, vy, Colorx, Colory, alphay])
```

String manipulation (build single string from a set of string, numerical, or vector arguments):
```
concat(arg1, arg2, ..., arg3)
```

Viewpoint declaration:
```
viewpoint {
   [ from vexper ]                  // Default: <0, 0,-1>
   [ at vexper ]                    // Default: <0, 0, 0>
   [ up vexper ]                    // Default: <0, 1, 0>
   [ angle fexper ]                 // Default: 45
   [ resolution fexper, fexper ]    // Default: 256x256
   [ aspect fexper ]                // Default: 1.0
   [ hither fexper ]                // Default: 1.0e-3
   [ yon fexper ]                   // Default: 1.0e6
```

```
    [ max_trace_depth fexper ]        // Default: 5
    [ aperture fexper ]               // Default: 0
    [ focal_distance fexper ]         // Default: distance between from &
at
    [ image_format fexper ]           // 0 = image, 1 = depth
    [ pixelsize fexper ]              // valid: 8, 16, 24, 32
    [ pixel_encoding fexper ]         // 0 = normal, 1 = RLE
    [ antialias fexper ]              // 0 = none, 1 = filter, 2-4 =
adaptive
    [ antialias_threshold fexper ]  // Default: 0.02
    }
```

Object Declaration:

```
object {
    Shape declaration
    [ texture declaration ]
    [ Object modifier declaration ]
    }
```

Shape Declarations:

```
bezier subdivision_type, flatness_value,
  u_subdivisions, v_subdivision,
  // 16 comma-separated vertices
  [<x0, y0, z0>, <x1, y1, z1>, ..., <x15, y15, z15> ]

blob threshold:
blob_component1
[, sphere <x, y, z>, strength, radius ]
[, cylinder <x0, y0, z0>, <x1, y1, z1>, strength, radius ]
[, plane <nx, ny, nz>, d, strength, distance ]

box <x0, y0, z0>, <x1, y1, z1>

cone <x0, y0, z0>, r0, <x1, y1, z1>, r1

cylinder <x0, y0, z0>, <x1, y1, z1>, r

disc <cx, cy, cz>, <nx, ny, nz>, r
disc <cx, cy, cz>, <nx, ny, nz>, ir, or

function f(x,y,z)

glyph contour_count,
    contour num_points1, V11, ..., V1n
    ...
    contour num_pointsm, Vm1, ..., Vmn

gridded "filename", object1 object2 ...

height_field "filename"
smooth_height_field "filename"

height_fn xsize, zsize, min_x, max_x, min_z, max_z, expression
height_fn xsize, zsize, expression
smooth_height_fn xsize, zsize, min_x, max_x, min_z, max_z, expression
smooth_height_fn xsize, zsize, expression

lathe type, direction, total_vertices,
    <vert1.x,vert1.y,vert1.z>
    [, <vert2.x, vert2.y, vert2.z>]
    [, etc. for total_vertices vertices]
```

```
nurb u_order, u_vertices, v_order, v_vertices,
   [u_knot1, ..., u_knot(u_order+u_vertices)],
   [v_knot1, ..., v_knot(v_order+v_vertices)],
   [[<vert(1,1)>, ..., <vert(1,v_order)>],
   ...
   [<vert(u_order,1)>, ..., <vert(u_order,v_order)>]]

nurb u_order, u_vertices, v_order, v_vertices,
   [[<vert(1,1)>, ..., <vert(1,v_order)>],
    ...
   [<vert(u_order,1)>, ..., <vert(u_order,v_order)>]]

parabola <x0, y0, z0>, <x1, y1, z1>, r

parametric <fx(u,v), fy(u,v), fz(u,v)>

polygon total_vertices,
   <vert1.x,vert1.y,vert1.z>
   [, <vert2.x, vert2.y, vert2.z>]
   [, etc. for total_vertices vertices]

polynomial f(x,y,z)

sphere <center.x, center.y, center.z>, radius

sweep type, direction, total_vertices,
   <vert1.x,vert1.y,vert1.z>
   [, <vert2.x, vert2.y, vert2.z>]
   [, etc. for total_vertices vertices]

torus r0, r1, <center.x, center.y, center.z>, <dir.x, dir.y, dir.z>

patch <v1.x,v1.y,v1.z>, <n1.x,n1.y,n1.z>, [ UV u1, v1, ]
      <v2.x,v2.y,v2.z>, <n2.x,n2.y,n2.z>, [ UV u2, v2, ]
      <v3.x,v3.y,v3.z>, <n3.x,n3.y,n3.z> [, UV u3, v3 ]

object1 + object2  // Union
object1 * object2  // Intersection
object1 - object2  // Difference
object1 & object2  // Clipping
~object1           // Inverse
```

Root Solver Declarations (for blobs, polynomials, splined lathes, and tori):

```
root_solver Ferrari

root_solver Vieta

root_solver Sturm
```

Object Modifier Statements:

```
translate <tx, ty, tz>

rotate <rx,ry,rz>

scale <sx,sy,sz>

shear yx, zx, xy, zy, xz, yz

shading_flags flag1+flag2+...

u_steps u

v_steps v
```

```
    w_steps w

    uv_steps u, v

    uv_steps u, v, w

    uv_bounds u0, u1, v0, v1

    bounding_box <x0,y0,z0>, <x1,y1,z1>

    displace expression
```

Shading Flag Values:
```
    1 = Shadow_Check, 2 = Reflect_Check, 4 = Transmit_Check,
    8  = Two_Sides, 16 = UV_Check, 32 = Cast_Shadow
```

Textures:
```
    texture {
       [ texture declaration ]
       [ rotate/translate/scale/shear ]
       }
```

Texture Declarations:
```
    surface { [ surface declarations ] }

    noise surface { [ surface declarations ] }

    noise surface_sym

    noise surface_sym { [ surface declarations ] }

    special surface { [ surface declarations ] }

    special surface_sym

    special surface_sym { [ surface declarations ] }

    checker texture1, texture2

    hexagon texture1, texture2, texture3

    layered texture1, texture2, ..., textureN
    indexed fn, texture_map([a, b, texture1, texture2]
                          ...

                          [x, y, texturei, texturej])

    indexed fn, texture_map_sym
    summed fexper, texture1, ... fexper, textureN
```

Surface Declarations:
```
    color <r, g, b>

    ambient scale

    ambient color, scale

    diffuse scale

    diffuse color, scale

    specular color, scale

    specular scale
```

```
reflection color, scale

reflection scale

transmission color, scale, ior

transmission scale, ior

microfacet kind angle
```

Microfacet Kinds:
```
Blinn, Cook, Gaussian, Phong, Reitz.
```

Special Surface Declarations add the following to Surface Declarations:
```
position vexper
// Turbulence: P + x * (dnoise(P, o) - <0.5, 0.5, 0.5>)

normal vexper
// Bumps: N + x * (dnoise(P, o) - <0.5, 0.5, 0.5>)
```

Bumpmap Declarations to use in the normal component of special surfaces:
```
cylindrical_bumpmap(image, V [, bump size])

planar_bumpmap(image, V [, bump size])

spherical_bumpmap(image, V [, bump size])
```

Noise Surface Declarations include all surface declarations plus:
```
color_map(map_entries)

bump_scale fexper

frequency fexper

phase fexper

lookup_fn index

normal_fn index

octaves fexper

position_fn index

position_scale fexper

turbulence fexper
```

The output of the following function is passed through the lookup function, then into the color map:
```
nval = pos * position_scale + turbulence * noise(P, octaves);
```

Valid values for position_fn are:

0	No position function used (default)
1	X coordinate in object space
2	X coordinate in world space
3	Distance from z-axis
4	Distance from the origin

5   Radial measure (counter clockwise) around y-axis

Valid values for normal_fn are:
0   No modification made to normal (default)

1   Bumpy

2   Rippled

3   Dented

Valid values for lookup_fn are:
0   Use nval directly

1   sawtooth applied to nval

2   sin function

3   ramp function

Conditional Processing:
```
if (cexper)
    [single declaration]

if (cexper) {
    [object/light/... declarations]
    }
else {
    [other object/light/... declarations]
    }
```

Particles:
```
particle { [particle_declarations] }

particle_sym

particle_sym { [particle_declarations] }
```

Particle Declarations:
```
birth expression  // Particles are born when exper is non-zero

count fexper  // # of objects to create when birth is non-zero

death expression  // Dies when the expression is non-zero

position vexper   // Starting position

velocity vexper   // Starting velocity

acceleration vexper  // Acceleration added every frame

avoid expression     // Any non-null expression invokes avoid

object object_sym   // Previously defined object
```

Particle Variables:
P   Current location of the particle

x   X location of particle

y	Y location of particle
z	Z location of particle
l	Current velocity of the particle
u	Age of the particle