Magic Camera

COLLABORATORS			
	TITLE : Magic Camera		
ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY		February 28, 2022	

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME

Contents

Chapter 1

Magic Camera

1.1 Magic Camera User's Guide

```
                Table of Contents


                About Magic Camera

                About Ray Tracers

                Writing Scripts

                Run Time Options

                Technical Description

                Contacting the Author
```

1.2 About the Author

```
    To contact the author for registration/bug reports:

    U.S. Mail:
        Dan Wesnor
        107 Sleepy Hollow Lane
        Madison, Al. 35758

    On-line Networks:
        GEnie:
            Mail Address: D.WESNOR

    Internet:
        d.wesnor@genie.geis.com

One final note...
    Please do not call me on the phone.  I work full time,
do freelance computer work, and still have to take care
```

of a house and yard. I can't afford to give telephone
support at the prices I'm charging. Please contact me
ONLY through the means listed above. Thank you.

1.3   About Magic Camera

    Magic Camera © 1990-1994 Dan Wesnor

    This project is huge. It took thousands of hours of programming,
researching, and testing. I stopped counting lines of code at 12,000, and that
was a long time ago.

    Many thanks to the people at the SAS Institute for the best compiler
I've ever used. If you program your Amiga (and I hope you do), buy it.

    I hate having to do this, but having worked so hard, I feel I deserve
some recognition, and if you feel the effort was good and the product useful to
you, I also feel I deserve some compensation for the work I've done. So read
the following to avoid any legal problems.

    The rights to profit from this program belong completely to the sole
author, Dan Wesnor. Remember, only the demo version is freely distributable.
The registered version may only be used by the person to whom it is registered.
Registration is traceable, and any "loose" copies of the registered version
found on bulletin boards and so forth will be considered illegal. The
registree, not the sysop, will be considered responsible for all illegally
distributed copies registered under his/her name. Prosecution is always an
option. Copies which cross state lines become a federal case. (The version you
are using will inform you as to whether it is a demo or registered version, and
who the registree is.)

    Registration may in no way be transferred. If you no longer want this
program, you may not sell it. Destroy all copies of all files distributed to
you.

    USE THIS SOFTWARE AT YOUR OWN RISK. The author assumes no liability for
damages resulting from the use of this program. Period. Using this software
constitutes waiver for any damages which you might incur in the process.

    About This Documentation

    This documentation is intended for use with AmigaGuide and compatible
document readers. You can try one of the programs which converts Guide files to
readable text. I've never used one, so good luck.

    In short, I hate writing documentation. I'm also no good at it. I feel
I made a good effort, and I detested every minute of it. Unfortunately, it
shows in the text. I'll probably make several cuts at the documentation and
release new versions with each new version of the code.

    If you notice any bugs in the documentation, send them to me, just as
you would a program bug. Hopefully, you'll find neither.

1.4   About Raytracers

A ray tracer is a program which creates highly realistic three dimensional scenes. Although a ray tracer is slow compared to other forms of 3D rendering, the basic algorithm used is simple, elegant, powerful, and quite easy to modify to produce highly realistic results. For instance, the same basic subroutines solve the problems of object visibility, local lighting (shadows), reflections, and transparency with refraction. This would explain why ray tracers are popular programs for many interested in 3D computer graphics – they are simple to write.

It seems as if the ray tracing algorithm has everything going for it: power, elegance, expandability. But there's a down side to everything. Ray tracing's primary down side is speed. A ray tracer must perform many calculations, called ray-object intersection tests. Each of these tests can involve from twenty to one hundred floating point calculations (when optimized). If a moderately detailed scene (5,000 objects) is rendered at a resolution of 640x480 pixels, then as many as 150 BILLION floating point calculations can be performed. And this doesn't include the calculations for shadows, reflections, transparency, anti-aliasing (getting the "jaggies" out), and pattern and texture mapping. Add it all up, and your computer ends up doing a lot of thinking.

So, what can't a ray tracer do. Well, in computer graphics, there is a problem that's referred to as "global lighting". Put simply, global lighting takes into account all the light in the scene. Light in a scene comes not only from sources like the lamps, but it is also reflected from every point on every object in the scene (the term "global" means just that – everything, everywhere). A ray tracer does not take this reflected light into account when computing shadows and shading. There are just too many possible light sources in a global solution. Some programmers have tried, using an algorithm called "ray bouncing", but even mediocre results slow rendering by a factor of 100 or more.

The current "great hope" of solving the global lighting problem is an algorithm called "radiosity". You may have heard this buzzword once or twice. If you've ever seen pictures generated using radiosity (especially indoor scenes), they're incredible. I would love to implement this algorithm, but unfortunately, it's just too slow, requires too much memory, and requires that the person building the scene know quite a bit about the internals of the algorithm. I have some ideas about finding some middle ground, but this is way down on my "to do" list.

If you'd like to know more about how Magic Camera implements the ray tracing algorithm, see the
                Technical Description
                 section.

1.5   Writing Scripts

Magic Camera takes its input in the form of script files.   ↩
                Scripts
describe the scene which is to be rendered, including the shape and location of objects in the scene and the surface characteristics of those objects.

Script Basics

Data Types

Using Variables and Expressions

Built-in Functions

Global Elements

Surface Characteristics

Primitive Elements

Constructed Elements

Named Objects/Instances

1.6   Scripts Writing Basics

Scripts consist of descriptions of objects, the surface ↩
                    characteristics
of those objects, scene lighting, the arrangement of the camera, and general
instructions to Magic Camera as to how the scene is to be rendered.

   Scripts are stored in ASCII files, so they may be written using any text
editor, or any word processor which allows files to be saved in ASCII. Scripts
may also be generated by another program, such as programs which convert objects
from other popular renderers into Magic Camera format.

   Although scripts follow a free-format, it is always a good idea to
follow a basic format so you know where everything is when you have to change
the script. I usually put control statements such as
                maxobcube
                 or
                maxaadepth
                 at
the beginning of the file. These are followed by the
                camera
                , then the
                lamps
                ,
and then the
                sky
                . After this,
                patterns
                 and
                textures
                 are declared, and finally,
the objects themselves. More important objects, such as those that are animated
or those which serve as a focal point for the scene usually come first, and the
background objects last. Of course, you can organize your scripts any way you

want.

    It's easier to keep track of what your script is doing by using
comments. Two types of comments are allowed, C-style, and line comments.
C-style comments are enclosed between '/*' and '*/', and may continue over many
lines. Example:

    /* this is a comment */

    Line comments are initiated by a semicolon, and continue until the end
of the line. They may start anywhere on the line. They may not extend past the
end of the line, but more than one may be used in a row:

; this is a line comment
; and this is another

    Use comments to write notes to yourself, so you can remember how a
script works months from now. It prevents a lot of confusion.

    Sometimes, scripts get too long to manage. In this case, you may want
to break the script down into several files. To rejoin them, use the include
statement:

include "filename"

    Typically, I put
               objects
                in separate files, and include them into the
main file. This keeps scripts uncluttered, and allows objects to be easily
shared by more than one script.

    Scripts consist of keywords and elements. Keywords are usually followed
by a value, such as:

maxobcube 3

    The value of "maxobcube" would be set to 3. Elements are more complex,
and consist of a keyword followed by a list of subelements in braces ('{' and
'}'). Some subelements are mandatory and must be included. Some are optional,
and have default values. If you forget a mandatory subelement, the parser will
tell you. Subelements, like keywords, consist of a keyword followed by a value.
 An example of an element declaration (this one's a sphere) is given below:

sphere {
    loc   <0, 5, 0>
    radius 3
    patt   red
}

    This sphere element uses three subelements, 'loc', 'radius', and 'patt'.
 The subelements 'loc' and 'patt' are actually shorthand for 'location' and
'pattern'. Any time a subelement has a shorthand listed, feel free to use it to
save typing. If you want, you can also use the longhand term.

    In reading the descriptions of elements below, note that some
subelements are surrounded by brackets ('[' and ']'). This indicates that these

subelements are optional. DO NOT INCLUDE THE BRACKETS IN THE SCRIPT. You'll
get a syntax error if you do.

    When subelements are referred to in context with their elements, a
notation like "sphere.radius" is used. This indicates the "radius" subelement
of the "sphere" element. This notation is used only by the documentation and
cannot be used in scripts.

    While on the subject of errors, if Magic Camera detects an error in a
script, it will inform you and try to continue reading the script. Sometimes it
will not be able to continue after the error. Correct the errors and run the
script again.

    Included in the Magic Camera package, you should have received three
files, called "scan.l", "parse.y", and "defaults.c". These are the definitive
definitions of the scripting language. They are also difficult to read. Scan.l
contains a list of keywords that the parser recognizes, and also the regular
expressions for things like
                Real
                 numbers. Parse.y contains the grammar of the
language, and will show the organization of the language, and which subelements
are acceptable with which elements. Defaults.c is a definitive list of the
default values for all subelements; use it as a reference. If it disagrees with
other documentation you received, let me know. Defaults.c is always correct (as
are all three of these files). If you learn to read these files, you may find
some interesting things. Experimental features show up here, but may not be
documented if they are not ready for use. If you try to use these experimental
features, do so with caution. They may not work properly. That's why they're
not documented.

1.7 Data Types

                This section lists the various types of values that Magic ←
                    Camera will
accept. It describes exactly how the types are formed. The data types accepted
are:

            Int

            Real

            Point

            Vector

            Color

            Identifier

            Filename

            Text

1.8 Integer

Int denotes an integer number. For those who can read such things, the regular expression for an integer is:

    [0-9]+

In English, this translates to a string of one or more digits (0-9).

Examples:

    52   Valid.
    52.0 Not Valid - Integers may not contain decimal points.

1.9 Real

Real denotes a real number. The regular expression is:

    [0-9]*"."[0-9]+

Which means zero or more optional digits, followed by a decimal point, followed by one or more digits.

Real numbers using exponential notation may also be used:

    [0-9]*"."[0-9]+[eE][+-]?[0-9]+

This more confusing regular expression simply means that any Real number may end with an "e" notation.

Also, an
            Int
             may be substituted anywhere a Real is used.

Examples:

    11.45   Valid.
    1.145e1   Valid. Same as 11.45 above.
    1.145e+1 Valid. Same as 11.45 above.
    11.45e-10 Valid.
    .1145   Valid. No leading zero required.
    1145.   Invalid. Decimal point must be followed by
              digits.
    1145   Valid.
              Ints
               may be used in
              place of Reals.
    11.4.5   Invalid. Only one decimal point allowed.

1.10   Point

Point denotes a three dimensional point. A Point consists of ←
three
legal
Real
numbers contained in less-than/greater-than signs ('<' and '>') ←
and
separated by commas. The numbers are in X, Y, Z order.

Examples:

```
<1.5, -7.0, 0.3>    Valid.
<1.5,-7.0,0.3>      Valid.  Spaces not required.
<1.5, -7, .3>       Valid. '-7' and '.3' are legal
```

Real
numbers.

1.11   Vector

Vector denotes a three dimensional vector. The basic ←
definition of
Vector is the same as for
Point
, with the exception that at least one dimension
of the Vector must be non-zero.

Examples:

```
<1.5, -7, .3>    Valid.
<0, 0, 0>        Invalid.  Must have one non-zero dimension.
<0, 1.0, 0>      Valid.
```

1.12   Color

Color denotes a set of
RGB
values. Its basic definition is the same as
for
Point
, except that all three values must be between 0.0 and 1.0, ←
inclusive.
The values are in red, green, blue order. Zero denotes no color, one denotes
full color.

Examples:

```
<0, 0, 0>        Valid.  This would be black.
<1, 1, 1>        Valid.  White.
```

```
<1, 0, 0>          Valid.  Red.
<1, 1, 0>          Valid.  Yellow
<0, 0, .5>         Valid.  A darkish blue.
<.6, 1, .6>        Valid.  A pastel green
<.5, -.1, .6>      Invalid.  No negative numbers.
<.5, 1.1, .6>      Invalid.  No numbers greater than 1.0.
```

Hint: Play with a color requester (like the one found in Preferences) to find the colors you want to use.

1.13 Identifier

Identifiers (IDs) are used to name patterns, objects, and just about anything else that can be named. Once an element has been named, it is referenced by that name. The regular expression for an Identifier is:

```
[_]*[a-zA-Z]+[_.a-zA-Z0-9]*
```

This means that an Identifier consists of any number of optional leading underscores, followed by any alphabetic character, followed by any number of alphanumerics, underscores, or periods.

Examples:

```
horse       Valid.
_horse      Valid.
__horse     Valid.  Any number of leading underscores may
            be used.
hor5se      Valid.
hor_se      Valid.
5horse      Invalid.  First non-underscore character must
            be alphabetic.
_5horse     Invalid.  First non-underscore character must
            be alphabetic.
```

1.14 Filename

Filenames are used when a file (such as a bitmap) must be loaded. The regular expression for a filename is:

```
"[a-zA-Z0-9/'._\]+"
```

This means any string of one or more alphanumeric characters, slashes (either forward or backwards), single quotes, periods, or underscores. Note that your operating system may allow more different characters in filenames than Magic Camera will currently parse.

1.15 Text Strings

The Text data type is used primarily by the
note
element. Text consists
of any characters between double quotes ("). However, Text may not carry over
from one line to the next.

Example:

    "Boy, have we got some text here!"   Valid

1.16   Using Variables and Expressions

Magic Camera scripts support the use of variables. Variables ←⌐
are used
to make managing related values easier and to allows simple animation.

    Variables make managing related values easier by allowing the value to
be specified once in the file, then used many times. If the value changes, then
the file only needs to be changed in one place. For example:

```
real sphere_radius = 5.0

sphere {
    ...
    radius    sphere_radius
    ...
}

sphere {
    ...
    radius    sphere_radius*2
    ...
}
```

    This example creates two spheres, one twice as big as the other. If the
writer decides that the spheres should be larger, then he only changes the
statement

```
real sphere_radius = 5.0
```

    to

```
real sphere_radius = 7.5
```

    causing all spheres which use sphere_radius to control their size to
change.

    Variables can facilitate animation by using the built-in variables
"FRAME", "FIRSTFRAME", and "LASTFRAME". These variable can be changed each time
the program is run (see
                Run Time Options
). They may be used in calculations
involving the location or size of objects. For example:

```
int numframes = 20

sphere {
    loc    <0, FRAME*10.0/numframes, 0>
    ...
}
```

    In this example, the Y coordinate of the sphere is dependent on the
value of FRAME. All that the user needs to do is change the value of FRAME, or
FIRSTFRAME and LASTFRAME each time Magic Camera is run, and several frames will
be produced in which the sphere appears to move vertically. See the demo script
"balls.a" for a more complex example of using variables for animation.

Declaring Variables

    There are 6 different variable types: real, int, vector, array, slice,
and path. They are described below.

    Type | Description
    --------+--

            real
                | same as
            Real

            int
                | same as
            Integer

            vector
                | a three dimensional vector

            array
                | a one dimensional array of
            Reals

            slice
                | a one dimensional array of 2D points

            path
                | a two dimensional array of 3D points

    Variables of type "real", "int", and vector are declared by:

type var_name = value

    where type is either "real", "int", or "vector", var_name is an

            Identifier
            , and value is an appropriate
            Real
            ,
            Integer
```

```
                    , or
                Vector
                 value.  All variables
MUST be initialized.

    Variables of type "array", "int", and "path" are defined by:

type var_name = { value1, value2, value3, ..., valueN }

    See the details on each type for examples.


Pre-defined Variables
---------------------

    Variable   |     Value
    -----------+-----------------------
    PI         |  3.1415...
    E          |  2.718281828...
    TINY       |  An extremely small, positive value
    HUGE       |  An extremely large, positive value
    FRAME      |  Determined by
                run-time options
                    FIRSTFRAME |  Determined by
                run-time options
                    LASTFRAME  |  Determined by
                run-time options
                Expressions
-----------
```

    As you may have noticed in some examples, expressions may be used. The
follow the standard order of operation for most programming languages, and
should be familiar to many. For the rest of you the order is as follows:

    1. Any value or expression in parenthesis is evaluated first.
    2. Expressions are evaluated left to right.
    3. Multiplication ("$*$") and division ("/") are evaluated next.
    4. Addition ("+") and subtraction ("-") are evaluated last.

    A
                built-in function
                 or variable name may be used in any appropriate
place in an expression. Expressions may freely mix real and integer values.
However, any expression which contains a real value ANYWHERE within it is
considered a real expression, and may not be used in place of an integer.
    The expression may be used anywhere a value is accepted. An expression
may be assigned to a variable by using:

var_name = expression

    For example:

x = 4*y

    which assigns the value of 4*y to the (previously declared) variable x.
Expressions may be embedded in complex types such as vectors or points...

```
sphere {
    ...
    loc    <5*cos(angle), 0, 5*sin(angle)>
    ...
}
```

1.17 Declaring Reals

A variable of type real can be used anywhere a
real number
may be used.
Before using a real variable, it must be declared and initialized:

real variable_name = initial_value

    where "variable_name" is an
                Identifier
                by which the variable will be
referenced in the future and "initial value" is an expression of type
                Real
                .

Example:

real foo = 5.2

1.18 Declaring Ints

A variable of type int can be used anywhere a
integer number
may be
used. Before using a int variable, it must be declared and initialized:

int variable_name = initial_value

    where "variable_name" is an
                Identifier
                by which the variable will be
referenced in the future and "initial value" is an expression of type
                Integer
                .

Example:

int foo = 5

1.19   Declaring Vectorss

                    A variable of type vector can be used anywhere a
                vector
                ,
                point
                , or
                color
                may be used. Before using a vector variable, it must be declared ←
                    and
initialized:

int variable_name = initial_value

    where "variable_name" is an
                Identifier
                 by which the variable will be
referenced in the future and "initial value" is a 3D value.

Example:

vector foo = <5, 4, -1>

NOTE: Unlike the
                vector
                 type, the vector variable may have a value of all zeros
(<0,0,0>). However, when used in place of a
                vector
                 type or
                color
                 type, the
vector variable must have an acceptable value.

1.20   Declaring Arrays

                    An array is a list of
                Real
                 values which can be referenced by an "index"
which specifies which value in the list is to be used. To declare an array:

array array_name = { value0, value1, ..., valueN }

    where "variable_name" is an
                Identifier
                 by which the array will be
referenced in the future and "value0" through "valueN" are expressions of type

                Real
                . Optionally, the keyword "closed" may be used as the last ←
                    element to indicate
that the array is closed (see below).

    To reference the values stored in an array, use:

```
array_name[integer_expression]
```

    where "integer expression" is any expression of type
                Integer
                . Note that
the first value in an array has an index of zero, so it would be referenced by:

```
array_name[0]
```

    If the array was declared using "closed", then referencing an element
beyond the last element of the array is allowed, and the index will "roll over"
to the beginning of the array. For example, if an array is declared with 10
values, then:

```
array_name[15]
```

    is the same as

```
array_name[5]
```

    Example of a closed array:

```
array foo = {
    1, 2, 3, 4, 5
    closed
}
```

NOTE: Currently, an array may hold only 100 values.

1.21 Declaring Slices

                    Slices are primarily used to describe cross sections of ↩
                        objects to be
built using certain
                constructed primitives
                , such as
                spins
                 and
                extrusions
                .

    A slice is a list of 2D vectors. To declare a slice:

```
slice slice_name = {
    <Xvalue0, Yvalue0>,
    <Xvalue1, Yvalue1>,
    ...,
    <XvalueN, YvalueN>
}
```

    where "variable_name" is an
                Identifier

```
                  by which the slice will be
referenced in the future and "value0" through "valueN" are expressions of type

              Real
              .  Optionally, the keyword "closed" may be used as the last  ←
                 element to indicate
that the slice is closed (see below).  The vectors need not be put on separate
lines.  The example does so only for clarity.

    Example of a closed slice:

slice foo = {
    <1, 1>,
    <1, 0>,
    <0, 0>,
    <0, 1>
    closed
}

NOTE:  Currently, a slice may hold only 100 points.
```

1.22 Declaring Paths

```
                    A path is a list of 3D
                Points
                 which can be referenced by an "index"
which specifies which point in the list is to be used.  To declare a path:

path path_name = {
    point0,
    point1,
    ...
    pointN
}

    where "variable_name" is an
                Identifier
                 by which the path will be
referenced in the future and "point0" through "pointN" are expressions of type

              Point
              .  Optionally, the keyword "closed" may be used as the last  ←
                 elements to indicate
that the path is closed (see below).  The points need not be put on separate
lines.  The example does so only for clarity.

    To reference the values stored in a path, use:

path_name[integer_expression]

    where "integer expression" is any expression of type
                Integer
                 .  Note that
the first value in a path has an index of zero, so it would be referenced by:
```

```
path_name[0]
```

If the path was declared using "closed", then referencing an element
beyond the last element of the path is allowed, and the index will "roll over"
to the beginning of the path. For example, if an path is declared with 10
values, then:

```
path_name[15]
```

is the same as

```
path_name[5]
```

A path can be used anywhere a
Point
or
Vector
can be used, for example,
the location of a sphere:

```
sphere {
    loc        path_name[3]
    radius     3
    patt       red
}
```

Example of a closed path:

```
path foo = {
    <0, 0, 0>,
    <0, 1, 0>,
    <0, 1, 1>,
    <1, 0, 0>
    closed
}
```

NOTE: Currently, a path may hold only 100 values.

1.23   Built-in Functions

The following built in functions may be used in expressions (note that
all angular values are in degrees):

```
    Function          |    Description
    ------------------+------------------------------------------
    cos(degrees)      |  Returns the cosine of the angle "degrees"
    sin(degrees)      |  Returns the sine of the angle "degrees"
    tan(degrees)      |  Returns the tangent of the angle "degrees"
    acos(value)       |  Returns the arccosine of the "value"
    asin(value)       |  Returns the arcsine of the "value"
    atan(value)       |  Returns the arctangent of the "value"
```

```
cosh(degrees)     | Returns the hyperbolic cosine of the angle "degrees"
sinh(degrees)     | Returns the hyperbolic sine of the angle "degrees"
tanh(degrees)     | Returns the hyperbolic tangent of the angle
                  | "degrees"
exp(value)        | Returns "e" to the power of "value"
log(value)        | Returns the natural logarithm of "value"
log10(value)      | Returns the base 10 logarithm of "value"
pow(x,y)          | Returns "x" to the power of "y"
power(x,y)        | Same as pow(x,y) above
sqrt(x)           | Returns the square root of "x"
ceil(x)           | Rounds "x" upwards to the nearest integer
floor(x)          | Truncates "x" downwards to the nearest
                  | integer
deg2rad(degrees)  | Converts the value "degrees" to the
                  | equivalent in radians
rad2deg(radians)  | Converts the value "radians" to the
                  | equivalent in degrees
```

1.24 Global Scripts Elements

```
          Setting the Camera

          The Sky

          Lighting the Scene

          Changing the Atmosphere

          Vertices

          Smoothing

          Anti-aliasing

          Octree control

          Importing Bitmaps

          Annotating Scripts

          Miscellaneous Flags
```

1.25 Setting Up the Camera

```
          Definition:

camera {

          location
```

```
                Point

                target

                Point
                    [
                up

                Vector
                ]
    [
                resolution

                Int

                Int
                ]
    [
                aspect

                Real
                ]
    [
                hfov

                Real
                ]
    [
                vfov

                Real
                ]
}
```

NOTE: All scripts must contain a camera definition.

1.26   camera.location

```
                Subelement:  location
Shorthand:  loc
Data type:
                Point
                Legal values:  All values legal for
                Point
                Default value:  No default.
```

This subelement is mandatory.

Description:

    Positions the camera in three dimensional space.

See Also:

                camera

1.27 camera.target

```
              Subelement:  target
Shorthand:  None
Data type:
              Point
              Legal values:  All values legal for
              Point
              Default value:  No default
```

This subelement is mandatory.

Description:

   Defines the point in space which the camera looks at. Currently, no
depth-of-field information is derived from the target, but may be in the future.

See Also:
```
              camera
```

1.28 camera.up

```
              Subelement:  up
Shorthand:  None
Data type:
              Vector
              Legal values:  All values legal for
              Vector
              .
   However, the "up" vector must be distinct from the direction
   the camera is pointing.
Default value:  <0, 1, 0>
```

This subelement is optional.

Description:

   Defines the direction which will be considered up for camera pointing
purposes only. The "up" vector must not be parallel to the direction the camera
is pointing.

See Also:
```
              camera
```

1.29 camera.resolution

```
                   Subelement:  resolution
Shorthand:  res
Data type:
                Int

                Int
                Legal values:  Both
                Ints
                 must be greater than or
     equal to one.
Default values:  320 400
```

This subelement is optional.

Description:

Determines the resolution of the output image. The first value indicates the horizontal resolution, the second vertical. Values of up to 65535 are supported, but this would cause extremely time and disk space consuming renderings.

```
     See Also:
                camera
```

1.30 camera.aspect

```
                   Subelement:  aspect
Shorthand:  None
Data type:
                Real
                Legal values:  All positive
                Reals
                 greater than
     zero
Default value:  0.56
```

This subelement is optional.

Description:

Determines pixel aspect ratio. Pixel aspect is defined as the vertical pixel size divided by the horizontal pixel size. The default value is the standard for 320x400 Amiga displays. Here are some other values:

```
     320x200 Amiga:  1.12
     640x400 Amiga:  1.12
     640x200 Amiga:  2.24
     Square Pixel (320x240, 640x480, 800x600, etc.):  1.0

     Note:  If aspect is not specified and
                hfov
                 and
                vfov
```

```
                    are, the default
value for aspect is computed from those two values and the image resolution.

See Also:
                    camera
```

1.31 camera.hfov

```
                    Subelement:  hfov
Shorthand:  None
Data type:
                    Real
                    Legal values:  1.0 to 179.0, inclusive
Default value:  45.0

This subelement is optional.

Description:

    Specifies the horizontal field-of-view, in degrees.  Larger numbers
indicate wider camera shots.  Smaller numbers give a zoom effect.

    Note:  If
                    vfov
                     is specified, hvof is computed from
                    aspect
                     and
                    vfov
                    and the image resolution.

See Also:
                    camera
```

1.32 camera.vfov

```
                    Subelement:  vfov
Shorthand:  None
Data type:
                    Real
                    Legal values:  1.0 to 180.0, inclusive
Default value:  Computed

This subelement is optional.

Description:

    Specifies the vertical field-of-view, in degrees.  Larger numbers
indicate wider camera shots.  Smaller numbers give a zoom effect.

    Note:  If vfov is not specified, the default value for vfov is computed
```

```
from
                camera.hfov
                 and
                camera.aspect
                 and the image resolution.
```

```
See Also:
                camera
```

1.33 The Sky

```
                Definition:

sky {
    [
                zenith

                Color
                ]
    [
                horizon

                Color
                ]
    [
                up

                Vector
                ]
    [
                numstars

                Int
                ]
    [
                starcolor

                Color
                ]
}
```

```
NOTE:  Although all subelements are optional, there must be at least one in the
sky definition.
```

1.34 sky.zenith

```
                Subelement:  zenith
Shorthand:  None
Data type:
                Color
```

```
                    Legal values:  All legal
                    Color
                     values
Default value:  <0.5, 0.0, 0.5> (deep magenta)

This subelement is optional.

Description:

    Specifies the color of the sky at the zenith.  Normally the zenith is
directly overhead, but can be changed using the
                    up
                     subelement of
                    sky
                    .

See Also:
                    sky.horizon
```

1.35 sky.horizon

```
                    Subelement:  horizon
Shorthand:  None
Data type:
                    Color
                    Legal values:  All legal
                    Color
                     values
Default value:  <0.0, 0.0, 0.5> (deep blue)

This subelement is optional.

Description:

    Specifies the color of the sky at the horizon.  The horizon is defined
to be the plane passing through <0, 0, 0> and perpendicular to the
                    up
                     vector of

                    sky
                    .

See Also:
                    sky.zenith
```

1.36 sky.up

```
                    Subelement:  up
Shorthand:  None
Data type:
```

```
                Vector
                Legal values:  All legal
                Vector
                 values
Default value:  <0, 1, 0>
```

This subelement is optional.

Description:

    Specifies the direction towards the
                zenith
                . Note that this "up" vector
is different from
                camera.up
                . This "up" vector is used only in computing sky
color.

See Also:
                sky

                sky.horizon

1.37 sky.numstars

```
                Subelement:  numstars
Shorthand:  nstars
Data type:
                Int
                Legal values:  Positive integers
Default value:  0
```

This subelement is optional.

Description:

    Specifies the total number of stars which should appear in the picture.

    Note: Because of the way the stars algorithm works, stars tend to
disappear when several levels of
                anti-aliasing
                 are used. I hope to get this
cleared up soon.

See Also:
                sky

                sky.starcolor

1.38 sky.starcolor

```
                  Subelement:  starcolor
Shorthand:  scolor
Data type:
                  Color
                  Legal values:  All legal
                  Color
                   values
Default value:  <1, 1, 1>

This subelement is optional.

Description:

    Specifies the brightest color which will be used for stars.  The stars
algorithm will "dim" this color to lesser values to simulate stars of different
intensities.

    Note:  Because of the way the stars algorithm works, stars tend to
disappear when several levels of
                  anti-aliasing
                   are used.  I hope to get this
cleared up soon.

See Also:
                  sky

                  sky.numstars
```

1.39   Changing the Atmosphere

```
                  Definition:

fog {
    [
                  color

                  Color
                  ]
    [
                  thinness

                  Real
                  ]
    [
                  distance

                  Real
                  ]
    [
                  power

                  Real
                  ]
```

```
}
```

NOTE: Although all subelements are optional, at least one must be included in
the definition.

1.40   fog.color

```
                Subelement:  color
Shorthand:  None
Data type:
                Color
                Legal values:  All legal
                Color
                 values
Default value:  <0.5, 0.5, 0.5>
```

This subelement is optional.

Description:

    Specifies the color of fog in the picture. The more something is
obscured by fog, the closer to this color it will become.

See Also:
                fog

1.41   fog.thinness

```
                Subelement:  thinness
Shorthand:  thin
Data type:
                Real
                Legal values:  Greater than zero
Default value:  1.0
```

This subelement is optional.

Description:

    Specifies how thick the fog will be. Higher values decrease visibility.
 Lower values increase visibility.

See Also:
                fog

1.42   fog.distance

```
                Subelement:  distance
Shorthand:  dist
Data type:
                Real
                Legal values:  Greater than zero
Default value:  10.0

This subelement is optional.

Description:

    Specifies how far away the fog begins to take effect.  The fog will not
obscure any objects within this many units of the camera.

See Also:
                fog
```

1.43 fog.power

```
                Subelement:  power
Shorthand:  pow
Data type:
                Real
                Legal values:  Greater than zero
Default value:  2.0

This subelement is optional.

Description:

    Specifies how linear the fog thickness will be.  A value of 1.0
indicates that the fog will be uniformly thick over any distance, i.e. the
obscuring effect of the fog will be proportional to the distance from the
camera.  Higher values cause the fog to thicken as objects move away from the
camera.

See Also:
                fog
```

1.44 Smoothing Polygons

```
                    Objects made of polygons (triangles and parallelograms) tend  ←
                         to have a
flat look to them.  Magic Camera is capable of using a shading technique (called
"Phong shading") to give these polygons the appearance of being rounded (they
are not actually rounded, they just appear to be through optical trickery).
This gives the objects the apearance of being smooth instead of facetted.

    Enclose groups of polygons to be smoothed in between "smoothon" and
```

"smoothoff" statements. There may be more than one "smoothon"/"smoothoff" pair,
but pairs may not be nested. The "smoothon" and "smoothoff" statements take no
arguments.

Example:

smoothon

/* Polygon definitions here... */

smoothoff

smoothon

/* More polygons here ... */

smoothoff

    All types of statements may be put between "smoothon" and "smoothoff".
However, only
                 triangles
                 ,
                 parallelograms
                 , and
                 compound objects
                  will be affected.
Polygons in one "smoothon"/"smoothoff" pair will be handled separately from
those in other pairs.

1.45 ws_g_antialias

                    "Jaggies" on the edges of polygons may be smoothed by invoking ←
                       the
anti-aliasing routines. Magic Camera uses an adaptive anti-aliasing algorithm
which traces additional rays only when necessary. When the corners of a pixel
are found to have different colors, the pixel is subdivided, and more rays are
traced. If the difference can still not be resolved, the pixel is subdivided
again, and so on, until either the difference is resolved or a pre-determined
limit is reached.
    The maximum number of subdivisions per pixel allowed is determined by
the "maxaadepth" statement. The "maxaadepth" statement takes one argument, an

                 Int
                 . The value given for "maxaadepth" must be greater than zero.   ←
                    The default
value is one, and a value of three gives excellent anti-aliasing (at the expense
of rendering time). Larger values of "maxaadepth" cause extremely time
consuming ray traces without producing noticeably better results.

    The difference in color which the algorithm considers to be significant
is controlled by the "maxaadiff" statement. This statement takes a
                 Color
                  as its
only argument. The default value is <0.1, 0.1, 0.1>, and rarely needs to be

changed.

Examples:

```
maxaadepth 3              /* go deep... */
maxaadiff <0.3, 0.3, 0.3>    /* but only when necessary */
```

    For more information on anti-aliasing, see
                Anti-Aliasing
                 in the

                Technical Description
                section.

1.46   Controlling the Octree

                    Magic Camera uses a data structure called an "octree" to ←
                        reduce the
number of intersection tests the ray tracer needs to perform. An octree
subdivides the three-dimensional world until either there are fewer than
"maxobcube" cubes in each division of the world, or each octant of the world has
been subdivided "maxoctdepth" times.
    The "maxoctdepth" statement takes a single
                Int
                 as its argument. The
default value is five. Larger numbers increase the size of the octree by as
much as eight times for each one added to the value.
    The "maxobcube" statement has one argument, a single
                Int
                 . Its default
value is two. Larger values will build a smaller octree.
    Larger octrees may speed up ray tracing, but for some scenes, they can
actually slow processing down. Larger octrees always use more memory.
    If the ray tracer runs out of memory while building the octree, it will
not abort. However, an "unbalanced" octree will be built, which may cause
longer trace times. Also, the ray tracer may need more memory after the octree
has been built. If the octree has used all the available memory, and the ray
tracer needs more later on, the ray tracer will quit before finishing the scene.
 It's always a good idea to specify an octree which leaves at least some memory
free.

Examples:

```
maxobcube 5    /* good settings for low memory */
maxoctdepth 3
```

    For more information on octrees, see
                How an Octree Works
                 in the

                Technical Description
                section.

1.47   Declaring Bitmaps

```
                  When using
             bitmapped patterns
              and
             bitmapped textures
             , it is necessary to
declare bitmaps before using them.  Bitmaps are stored in files on disk, and
loaded into memory when declared.  Two types of file formats are recognized,
             IFF
             and
             raw 24 bit data
             .
```

```
To declare a bitmap using an
             IFF
              file:
```

```
iff filename bitmapname
```

```
To declare a bitmap using
             raw 24 bit data
             :
```

```
raw24 filename bitmapname
```

```
    In both cases, the argument "filename" is of type
             Filename
              and the
argument "bitmapname" is of type
             Identifier
             .  Data is loaded from a file called
"filename" and stored in a bitmap called "bitmapname".  When the bitmap is
referenced in other parts of the script, it will be referenced by "bitmapname".
```

```
    If the same bitmap file is used several times, it needs to be declared
only once.  Using the same file in several bitmap declarations wastes memory.
```

```
Example:

/* load an image from the file "mapfiles/duffslabel.iff" and use it in a
spherical wrap */

iff mapfiles/duffslabel.iff duffslabel

wrapsphere foo {
    bitmap    duffslabel
    /* other wrapsphere subelements here */
}
```

See also:

                wrapsphere

                wrapcylinder

                wrapflat

                bumpsphere

                bumpcylinder

                bumpflat
                NOTE: Bitmap are memory hungry. The approximate amount of memory $\hookleftarrow$
                    needed by a
bitmap can be computed as follows:

For 24 bit images:

    memory (bytes) = (X image size) x (Y image size) x 3

For images which are 8 bits or less deep:

    memory (bytes) = (X size) x (Y size) x (depth) / 8

Examples:

A 640 by 480 24 bit image uses:

    memory = 640 x 480 x 3 = 921,600 bytes

A 320 by 400 HAM image uses:

    memory = 320 x 400 x 6 / 8 = 96,000 bytes
    (HAM images are 6 bits deep)

1.48   ws_g_note

                Sometimes you may want to include notes in a script and have $\hookleftarrow$
                    those notes
printed out as that script is executed. A good use of this is displaying the
author's name or any copyright information. To include notes in a script, use:

note
                Text
                For example:

note "This script by The Imagination Guys"

    The message between quotes would be displayed as the script is executed.

1.49 ws_g_flags

Magic Camera scripts may contain flags which cause certain features of
the ray tracer to either be disabled or altered. Flags are made up of a single
word and take no arguments. The following table lists Magic Camera's script
flags and their functions.

```
noshadows       Causes shadows to not be computed
noshad             "       "      "   "   "      "
notransparent   Causes no transparencies to be computed
notrans            "       "         "           "   "      "
noreflect       Causes no reflections to be computed
norefl             "       "        "          "   "      "
illuminate      Causes all objects to be fully lit
illum              "       "        "       "   "    "    "
```

1.50 Lighting

There are two types of lamps in Magic camera, local lamps and
directional lamps. Local lamps exist at a fixed location in space. Directional
lamps have no fixed location. Light from directional lamps appears to have come
from a source placed an infinite distance away in a given direction.

Local lamps are more powerful and offer a wider range of options.
Directional lamps are "quick and dirty", allowing simple light sources to be
quickly defined.

Local lamps can further be divided into three groups: simple, extended,
and spotlight. Simple local lamps have no outstanding features. Extended lamps
can be used to create more realistic shadows at the expense of increased trace
time, and spotlights create an effect similar to that of a theater spotlight.
It is possible, but not terribly useful, to create an "extended spotlight" lamp.

The definition of a local lamp is:

```
lamp {

                location

                Point
                    [
                color

                Color
                ]
    [
                distance

                Real
                ]
    [
                radius

                Real
```

```
                    ]     /* for extended lamps */
        [
                    numrays

                    Int
                    ]     /* for extended lamps */
        [
                    pointat

                    Point
                    ]    /* for spotlights */
        [
                    spread

                    Real
                    ]    /* for spotlights */
        [
                    power

                    Real
                    ]    /* for spotlights */
        [
                    noshad
                    ]
}
```

The definition of a directional lamp is:

```
lamp {

                    direction

                    Vector
                        [
                    color

                    Color
                    ]
        [
                    noshad
                    ]
}
```

Unless indicated above, you may not mix the definitions for local and directional lamps. In particular, the "direction" and "location" elements may not exist within the same lamp definition.

1.51 lamp.location

```
                    Subelement:  location
Shorthand:  loc
Data type:
```

```
                    Point
                    Legal values:  All values legal for
                    Point
                    .
Default value:  None
```

This subelement is mandatory for local lamps and illegal for directional lamps.

Description:

Defines the position in three dimensional space at which the lamp is to be located.

See Also:

lamp

1.52 lamp.color

```
                    Subelement:  color
Shorthand:  None
Data type:
                    Color
                    Legal values:  All values legal for
                    Color
                    .
Default value:  <1, 1, 1>
```

This subelement is optional for all lamps.

Description:

Indicates the color of light emitted by the lamp. A value of <0, 0, 0> (black) is pointless since it essentially turns off the lamp.

See Also:

lamp

1.53 lamp.spread

```
                    Subelement:  spread
Shorthand:  None
Data type:
                    Real
                    Legal values:  greater than zero
Default value: 15.0
```

This subelement is optional for local lamps and ineffective for directional lamps.

Description:

    For "spotlight" type lamps, defines the spread angle of the spotlight in degrees. For a value of 45.0, the lamp will emit a cone of light which is 45 degrees in angle.

    The

               pointat

                subelement must be defined in order for "spread" to be effective.

See Also:

               lamp

               power

1.54   lamp.power

               Subelement: power
Shorthand: pow
Data type:

               Real
               Legal values: Greater than zero
Default value: HUGE

    This subelement is optional for local lamps and ineffective for directional lamps.

Description:

    For "spotlight" type lamps, defines how sharp the edges of the spotlight circle will be. Higher values increase sharpness.

See Also:

               lamp

               spread

1.55   lamp.radius

               Subelement: radius
Shorthand: None
Data type:

               Real
               Legal values: Greater than or equal to zero
Default value: 0.0

    This subelement is optional for local lamps and ineffective for directional lamps.

Description:

    Causes the lamp to be a spherical extended light source with a radius of
the given value. If a radius other than 0.0 is specified, than
                numrays
                 controls
the number of shadow rays cast to this lamp.

    Using extended light sources will causes shadows to "soften", making
them more realistic. However, trace time increases dramatically, just as if
several lamps were added to the scene. This subelement is recommended for use
in final images only.

See Also:
                lamp

1.56 lamp.distance

                Subelement: distance
Shorthand: dist
Data type:
                Real
                Legal values: Greater than or equal to zero
Default value: HUGE

    This subelement is optional for local lamps and ineffective for
directional lamps.

Description:

    Specifies the distance from the lamp at which the light fades away.
Objects farther away from the source than the indicated value will not be lit by
the lamp. Objects closer to the lamp will be more brightly lit. The default
value of HUGE is sufficiently large to cause all objects to be fully lit in most
scenes.

See Also:
                lamp

1.57 lamp.numrays

                Subelement: numrays
Shorthand: nrays
Data type:
                Int
                Legal values: Greater than zero
Default value: 1

    This subelement is optional for local lamps and ineffective for
directional lamps.

Description:

    Indicates how many rays will be traced to the lamp in order to compute
shadows. The actual number of shadow rays per lamp is the square of the value
given; i.e., a value of 3 causes 9 rays to be cast in shadow computation for
this lamp. The default value of one essentially cause the lamp to be a point,
rather than extended, source.

See Also:

                lamp

                radius

1.58 lamp.pointat

                Subelement: pointat
Shorthand: None
Data type:
                Point
                Legal values: All values legal for
                Point
                .
Default value: <0, 0, 0>

    This subelement is optional for local lamps and ineffective for
directional lamps.

Description:

    For "spotlight" type lamps, defines the location in three dimensional
space at which the spotlight should point.

See Also:

                lamp

                spread

                power

1.59 lamp.noshad

                Subelement: noshad
Shorthand: None
Data type: None
Legal values: None
Default value: None

    This subelement is optional for all lamps.

Description:

    Causes no shadow computation to be done for this lamp. Full shading computations are still done, but no shadows will be cast by the lamp. (Great for rendering vampires.)

See Also:
                lamp

1.60   lamp.direct

                Subelement: direction
Shorthand: direct
Data type:
                Vector
                Legal values: All values legal for
                Vector
                .
Default value: None

    This subelement is mandatory for directional lamps and illegal for local lamps.

Description:

    Indicates the direction in which light travels from a directional lamp.

See Also:
                lamp

1.61   Vertices

                Some primitives, namely
              triangles
             can be described by using named
vertices. This form of declaration is useful in creating polygon meshes and is most commonly used by programs which translate other object formats into Magic Camera scripts.

To declare a named vertex:

vertex
                Identifier

                Point
                Example:

vertex vert1 <13, 2, 6>

    This will create a vertex named "vert1" at the point <13, 2, 6>.

Once a group of vertices is no longer needed, the "flushverts" command
should be used. This frees up the memory used by all the named vertices which
have been declared. The "flushverts" command takes no arguments.

1.62   Surface Characteristics

Obviously, there must be some way to control the appearance of ←↩
objects
which are in the scene. Magic Camera provides two basic ways in which an
object's characteristics can be altered: surface color, and surface texture.

The color of a surface is controlled by
pattern
elements. Pattern
elements can range from simple colors to procedural textures to bitmapped
patterns created using your favorite paint program or scanner.

Normally, all objects appear smooth. However, this produces unrealistic
images, since objects in the real world have different textures. Magic camera
allows
texture
definitions to be attached to objects. Textures may be either
procedural or bump-mapped.

1.63   Patterns

Magic Camera has very powerful pattern generation features.   ←↩
Patterns
can be defined in heirarchial trees. This means that one pattern can be used to
define how other patterns are applied. For example, a
marble
pattern may be
used to combine two
color
patterns to produce a marble surface. The
brick
pattern may then be used to combine the resulting marble surface ←↩
with another

color
pattern, generating a wall made up of marble bricks.

Hopefully, this diagram will help explain:

```
     brick         <---- the top level is applied to an object
        |
   -------------
   |           |
  color     marble   <---- any pattern, no matter how complex,
```

```
                      |             may be combined into other patterns
      -------------
      |           |
   color       color
```

By carefully combining patterns, you can create extremely complex patterns. Use your imagination!!

There are three types of patterns used by Magic Camera: simple, procedural, and bitmapped.

Simple patterns are always on the lowest levels of pattern trees. They are the building blocks for all other pattern types. Simple patterns are define using
                  color
                  .

Procedural patterns use some type of built-in algorithm to mix other patterns. The following are Magic Camera's pre-defined procedural patterns:

              check

              brick

              marble

              wood

              clouds

              blotch
                  The last four procedural patterns can often be used to create ←
                      some very
interesting patterns. Remember, clouds need not be white on a blue sky (they don't even need to be on the sky!!), and wood can be so mangled as to not look even remotely like wood, if that's what you want. Play with these, and you'll find you can do some creative things with them.

Bitmapped patterns are also very useful. They can be used to put pictures on wall, create street signs, or even put labels on soda bottles. The various bit-mapped patterns all use information loaded by the
                  bitmap
                   command.
However, they differ in how the bitmap is applied to the object. The bitmap may be "stamped" onto the object using the
                  wrapflat
                   pattern. The
                  wrapsphere
                   and

                  wrapcylinder
                  patterns have more of a "shrink wrap" effect, as if a spherical or ←
                      cylindrical
picture were placed around the object and shrunk onto it until it fit tightly.
```

1.64 Pattern Name

This field is of type
identifier
, and "names" the pattern. When the
pattern is attached to objects, it will be referenced by this
identifier
.

1.65 color

The color pattern is the simplest of all pattern definitions, ←
and is
used as a basis for defining all other patterns. The format of a color
definition is:

color
patt_name
{
[
diffuse

color
]
[
ambient

color
]
[
reflect

color
]
[
filter

color
]
[
transmit

color
]
[
index

real
]

```
    [
                speccoef

                real
                ]
    [
                specrefl

                real
                ]
}
```

Please note that although no subelements are mandatory, there must be at least one subelement within the definition.

1.66   color.diffuse

```
                Subelement:  diffuse
Shorthand:  diff
Data type:
                Color
                Legal values:  All legal values for
                Color
                Default value:  <1, 1, 1>
```

This subelement is optional.

Description:

Defines the diffuse color of the pattern. The diffuse color is the color which the pattern appears to be when fully lit.

See also:
                color

                The Magic Camera Color Model

1.67   color.ambient

```
                Subelement:  ambient
Shorthand:  amb
Data type:
                Color
                Legal values:  All legal values for
                Color
                Default value:  <.5, .5, .5>
```

This subelement is optional.

Description:

Defines the amount of ambient light falling on the pattern. Ambient
light is light that does not appear to come from any particular light source.
This represents the minimum amount of light which will fall on objects with this
pattern, even if no lamps are present in the scene.

A good trick for glowing objects, such as neon signs or car tail lights,
is to raise the ambient value to <1, 1, 1>, causing the pattern to glow in the
dark.

See also:

            color

            The Magic Camera Color Model

1.68   color.reflect

            Subelement: reflect
Shorthand: refl
Data type:

            Color
            Legal values: All legal values for
            Color
            Default value: <0, 0, 0>

This subelement is optional.

Description:

Defines the amount of reflection in the pattern. The default (black)
causes no reflections to appear on this pattern. Changing the value causes
reflections to be computed, which heightens realism but also increases trace
time.

See also:

            color

            The Magic Camera Color Model

1.69   color.filter

            Subelement: filter
Shorthand: filt
Data type:

            Color
            Legal values: All legal values for
            Color
            Default value: <1, 1, 1>

This subelement is optional.

Description:

    Defines the filter color of the pattern. For transparent objects, this
defines how much light will be filtered as rays pass through the object on a per
unit length basis. The default value (white) causes all light passing through
the object to be filtered if the ray travels one or more unit length inside the
object.

    If a pattern with a filter value of <.5, 0, 0> is applied to a sphere of
radius 1.0, then rays traveling through the center of the sphere will have all
red light filtered out, since the ray travels 2.0 units through the center
(radius*2) and half (0.5) of the light is filtered per unit. Rays traveling
through the sphere near its edges will have less red light filtered out. Other
green and blue light will not be affected.

    This subelement is primarily used to create more convincing shadows of
transparent objects.

See also:
                color

                color.transmit

                The Magic Camera Color Model

1.70   color.transmit

                Subelement: transmit
Shorthand: trans
Data type:
                Color
                Legal values: All legal values for
                Color
                Default value: <0, 0, 0>

    This subelement is optional.

Description:

    Defines the color of light transmitted by the pattern. A pattern with a
transmit value of <0, 0, 1> will appear as blue glass. The default value
(black) makes the pattern opaque. Changing the value will increase trace time.

    Note that, unlike
                filter
                , there is no dependence on the length of a ray
passing through the object.

See also:
                color

                color.filter

                    The Magic Camera Color Model

1.71 color.index

                    Subelement: index
Shorthand: None
Data type:
                    Real
                    Legal values: Greater than zero
Default value: 1.0

    This subelement is optional.

Description:

    For transparent patterns, defines the index of refraction. The default
value causes no refraction to occur. Larger values increase the amount of
refraction.

    Refraction, put simply, is the bending of light as it passes through a
transparent object.

See also:
                    color

1.72 color.specrefl

                    Subelement: specrefl
Shorthand: srefl
Data type:
                    Real
                    Legal values: Between zero and one, inclusive
Default value: 0.5

    This subelement is optional.

Description:

    This is the percentage of light reflected by "specular" reflection.
Higher values produce brighter "hot spots" on the pattern.

    For the purposes of Magic Camera, specular reflection means reflection
of the light source on the pattern. This makes a pattern appear to be shiny.

See also:
                    color

                    color.speccoef

                    The Magic Camera Color Model

1.73   color.speccoef

                  Subelement: speccoef
Shorthand: scoef
Data type:
                  Real
                  Legal values: Greater than zero
Default value: 50

    This subelement is optional.

Description:

    This element controls the "tightness" of the pattern's specular
reflection. Higher values produce tighter "hot spots" on the pattern.

    For the purposes of Magic Camera, specular reflection means reflection
of the light source on the pattern. This makes a pattern appear to be shiny.

See also:
                  color

                  color.specrefl

                  The Magic Camera Color Model

1.74   check

                  This pattern produces a checkerboard type pattern, with ←
                      alternating
squares of two different surfaces. The format is:

check
                  patt_name
                   {

                  pattern1

                  Identifier

                  pattern2

                  Identifier
                      [
                  xsize

                  Real
                  ]
        [

```
                ysize

                Real
                ]
    [
                zsize

                Real
                ]
}
```

1.75   check.pattern1, check.pattern2

```
                Subelement:  pattern1, pattern2
Shorthand:  patt1, patt2
Data type:
                Identifier
                Legal values:  The name of any previosly defined pattern
Default value:  None
```

These subelements are mandatory.

Description:

These subelements specify the alternating patterns used in the
checkerboard.

See also:
                check

1.76   check.xsize, check.ysize, check.zsize

```
                Subelement:  xsize, ysize, zsize
Shorthand:  None
Data type:
                Real
                Legal values:  Greater than zero
Default value:  1.0 for each
```

These subelements are optional.

Description:

These subelements specify the size of the alternating blocks in the X,
Y, and Z dimensions.

See also:
                check

1.77 brick

This pattern simulates a brick surface. The format is:

```
brick
                patt_name
                 {

                brick

                Identifier

                mortar

                Identifier
                    [
                xsize

                Real
                ]
    [
                ysize

                Real
                ]
    [
                zsize

                Real
                ]
    [
                msize

                Real
                ]
    [
                yoffset

                Real
                ]
    [
                zoffset

                Real
                ]
}
```

1.78 brick.brick, brick.mortar

```
                Subelement:  brick, mortar
Shorthand:  None
Data type:
                Identifier
```

```
                     Legal values:  The name of any previosly defined pattern
Default value:   None
```

    These subelements are mandatory.

Description:

    These subelements specify the patterns used for the "bricks" and the
"mortar" in the pattern.

See also:
                brick

1.79   brick.xsize, brick.ysize, brick.zsize

```
                Subelement:  xsize, ysize, zsize
Shorthand:  None
Data type:
                Real
                Legal values:  Greater than zero
Default value:  xsize = 1.0, ysize = 0.5, zsize = 0.5
```

    These subelements are optional.

Description:

    These subelements specify the size of the bricks in the X, Y, and Z
dimensions.

See also:
                brick

1.80   brick.msize

```
                Subelement:  msize
Shorthand:  None
Data type:
                Real
                Legal values:  Greater than zero
Default value:  0.2
```

    This subelement is optional.

Description:

    This subelement specifies the thickness of the mortar between the
bricks.

See also:
                brick

1.81   brick.yoffset, brick.zoffset

```
              Subelement:  yoffset, zoffset
Shorthand:  None
Data type:
              Real
              Legal values:  Any valid
              Real
              Default value:  yoffset = 0.5, zoffset = 0.5

    These subelements are optional.

Description:

    These subelements specify the offset between rows of bricks.  As bricks
are "stacked" in the Y dimension, each successive row is displaced in the X
dimension by "yoffset".  As bricks are "stacked" in the Z dimension, each
successive row is displaced in the X dimension by "zoffset".  Note that there is
currently no subelement called "xoffset".

See also:
              brick
```

1.82   marble

```
              This pattern creates a rather realistic marble effect.  The  ←
                 format is:

marble

              patt_name
               {

              pattern

              Identifier

              grain

              Identifier
                  [
              scale

              Real
              ]
    [

              power

              Real
              ]
```

```
}
```

1.83   marble.pattern, marble.grain

```
              Subelement:  pattern, grain
Shorthand:  patt, None
Data type:
              Identifier
              Legal values:  The name of any previosly defined pattern
Default value:  None
```

    These subelements are mandatory.

Description:

    These subelements specify the patterns used. The pattern named by
"pattern" is the primary pattern of the marble, while the pattern named by
"grain" is used for the grain running through the marble.

See also:
              marble

1.84   marble.scale

```
              Subelement:  scale
Shorthand:  None
Data type:
              Real
              Legal values:  Greater than zero
Default value: 1.0
```

    This subelement is optional.

Description:

    This subelement changes the apparent size of the pattern on the surface
of the object. Increasing values cause the pattern to become smaller on the
object, while decreasing values cause the pattern to become larger on the
object.

See also:
              marble

1.85   marble.power

```
                Subelement:  power
Shorthand:  None
Data type:
                Real
                Legal values:  Greater than zero
Default value:  1.0

    This subelement is optional.

Description:

    This subelement changes the sharpness of the transition between the
"pattern" and "grain" patterns of the marble.  It also influences the thickness
of the "grain".  Increasing values cause sharper transitions, and more "grain".
Decreasing values cause transitions to be less sharp, and less "grain" will
appear.

See also:
                marble
```

1.86 wood

```
                Creates a simulated wood pattern.  The format is:

wood
                patt_name
                 {

                pattern

                Identifier

                grain

                Identifier
                    [
                scale

                Real
                ]
}
```

1.87 wood.pattern, wood.grain

```
                Subelement:  pattern, grain
Shorthand:  patt, None
Data type:
                Identifier
                Legal values:  The name on any previosly defined pattern
```

```
Default value:  None
```

These subelements are mandatory.

```
Description:
```

These subelements specify the patterns used. The pattern named by
"pattern" is the primary pattern of the wood, while the pattern named by "grain"
is used for the grain running through the wood.

```
See also:
                wood
```

1.88   wood.scale

```
                Subelement:  scale
Shorthand:  None
Data type:
                Real
                Legal values:  Greater than zero
Default value: 1.0
```

This subelement is optional.

```
Description:
```

This subelement changes the apparent size of the pattern on the surface
of the object. Increasing values cause the pattern to become smaller on the
object, while decreasing values cause the pattern to become larger on the
object.

```
See also:
                wood
```

1.89   clouds

```
                Creates a cloud pattern.  The format is:

clouds
                patt_name
                 {

                sky

                Identifier

                clouds

                Identifier
                    [
```

```
                    scale

                    Real
                    ]
    [
                    power

                    Real
                    ]
    [
                    perturb

                    Real
                    ]
    [
                    xphase

                    Real
                    ]
    [
                    yphase

                    Real
                    ]
    [
                    zphase

                    Real
                    ]
}
```

1.90 clouds.sky, clouds.clouds

```
                Subelement:  sky, clouds
Shorthand:  None
Data type:
                Identifier
                Legal values:  The name of any previosly defined pattern
Default value:  None
```

These subelements are mandatory.

Description:

These subelements specify the patterns used. The pattern named by "sky" is the background pattern of the sky, while the pattern named by "clouds" is used for the clouds in the sky.

See also:
```
                clouds
```

1.91   clouds.scale

```
                Subelement:  scale
Shorthand:  None
Data type:
                Real
                Legal values:  Greater than zero
Default value: 1.0
```

   This subelement is optional.

Description:

   This subelement changes the apparent size of the pattern on the surface
of the object. Increasing values cause the pattern to become smaller on the
object, while decreasing values cause the pattern to become larger on the
object.

See also:
                clouds

1.92   clouds.power

```
                Subelement:  power
Shorthand:  pow
Data type:
                Real
                Legal values:  Greater than zero
Default value: 1.0
```

   This subelement is optional.

Description:

   This subelement causes the clouds to become more or less defined in the
sky, and also causes more or less clouds to be present. Increasing values cause
fewer, but more sharply defined, clouds. Decreasing values have the opposite
effect.

See also:
                clouds

1.93   clouds.perturb

```
                Subelement:  perturb
Shorthand:  turb
Data type:
                Real
                Legal values:  Greater than zero
Default value: 1.0
```

```
    This subelement is optional.

Description:

    This subelement changes the amount of turbulence in the clouds.  Higher
values cause more turbulence.  Values above two or so give unrealistic results,
but sometimes this is desired.  Low values may cause too much repetition in the
cloud pattern.

See also:
                clouds
```

1.94   clouds.xphase, clouds.yphase, clouds.zphase

```
                Subelement:  xphase, yphase, zphase
Shorthand:  None
Data type:
                Real
                Legal values:  All valid values for
                Real
                Default value:  0.0

    These subelements are optional.

Description:

    These subelements change the phase of the sine and cosine waves used to
create the clouds.  By cycling one or more of these values through 360 degrees,
the clouds may be animated.

See also:
                clouds
```

1.95   wrapsphere

```
                This pattern wraps a bitmapped image onto the object using a  ←
                    sphere as
the intermediate surface.  The format is:

wrapsphere
                patt_name
                 {

                bitmap

                Identifier

                pattern
```

```
                Identifier
                    [
                substitute

                Integer

                Identifier
                ]
    [
                dodiffuse
                ]
    [
                dotransmit
                ]
    [
                doreflect
                ]
    [
                xrepeat

                Real
                ]
    [
                yrepeat

                Real
                ]
    [
                filter

                Integer
                ]
}
```

The shorthand "wrapsp" may be used in place of "wrapsphere".

One of [
                dodiffuse
                ], [
                dotransmit
                ], or [
                doreflect
                ] must be included.

1.96   wrapsphere.bitmap, wrapcylinder.bitmap, wrapflat.bitmap

```
                Subelement:  bitmap
Shorthand:  None
Data type:
                Identifier
                Legal values:  The name of any previosly defined bitmap
Default value:  None
```

This subelement is mandatory.

Description:

This subelement specifies the bitmap which is to be applied to the
object.

See also:

            wrapsphere

            wrapflat

            wrapcylinder

            Declaring Bitmaps

1.97 wrapsphere.pattern, wrapcylinder.pattern, wrapflat.pattern

            Subelement: pattern
Shorthand: patt
Data type:
            Identifier
            Legal values: The name of any previosly defined pattern
Default value: None

    This subelement is mandatory.

Description:

    This subelement specifies the pattern which is used as a basis for the
new pattern. Since the bitmap will supply only the diffuse, transmit, and/or
reflect values of the new pattern, all other values come from the pattern
specified by this subelement.

See also:

            wrapsphere

            wrapflat

            wrapcylinder

            dodiffuse

            dotransmit

            doreflect

1.98 wrapsphere.substitute, wrapcylinder.substitute, wrapflat.substitute

```
                 Subelement:  substitute
Shorthand:  sub
Data type:
                 Integer

                 Identifier
                 Legal values:  Any integer greater than or equal to zero followed  ←
                     by
        the name of any previously defined pattern
Default value:  None
```

    This subelement is optional.

Description:

    If the bitmap uses a color look-up table (bitmaps with 256 or fewer
colors), it is possible to substitute a previously defined pattern for a color
in the look-up table. For example:

```
wrapsp foo {
    ...
    sub 4 white_marble
    ...
}
```

    This would cause the pattern "white_marble" to be used where ever the
color specified by pen number 4 appeared in the bitmap.

See also:

                 wrapsphere

                 wrapflat

                 wrapcylinder

1.99   wrapsphere.dodiffuse, wrapsphere.dotransmit, wrapsphere.doreflect

```
                 Subelement:  dodiffuse, dotransmit, doreflect
Shorthand:  dodiff, dotrans, dorefl
Data type:  None
Legal values:  None
Default value:  None
```

    At least one of these subelements must be used.

Description:

    These subelements cause the color found in the bitmapped image to be
used for the diffuse, transmitted, or reflected color of the pattern. These may
be used in any combination, so long as at least one appears in the defintion.

See also:

wrapsphere

wrapflat

wrapcylinder

1.100 wrapsphere.xrepeat, wrapsphere.yrepeat

Subelement: xrepeat, yrepeat
Shorthand: xrep, yrep
Data type:

Real
Legal values: Greater than zero
Default value: 1.0

These subelements are optional.

Description:

These subelements control how many times the bitmap is repeated around
the sphere. The bitmap is repeated "xrepeat" number of times around the equator
of the sphere and "yrepeat" number of times between the poles of the sphere.

See also:

wrapsphere

1.101 wrapsphere.filter

Subelement: filter
Shorthand: filt
Data type:

Integer
Legal values: 1, 3, 5, or 7
Default value: 3

This subelement is optional.

Description:

Controls the size of the filter used when applying the bitmap. Either
no filter is used or a 3x3, 5x5, or 7x7 Gaussian filter is applied (see table).
This is necessary because aliasing occurs when mapping the bitmap from object
space to screen space. This aliasing manifests itself in the form of jaggies
and thin lines which disappear and reappear. Changing this values may improve
the quality of the bitmap's appearance.

The values given cause the following filters to be used:

```
 value | filter
-------+-------------
   1   | no filter
   3   | 3x3 Gaussian
   5   | 5x5 Gaussian
   7   | 7x7 Gaussian
```

Since the aliasing depends on final image resolution and the possibility
of magnification through reflections and transmissions by curved surfaces, the
only way to find the best filter to use is by playing with the values.

See also:
                    wrapsphere

                    wrapflat

                    wrapcylinder

1.102   wrapflat

                    This pattern wraps a bitmapped image onto the object using a ←
                      plane as
the intermediate surface. The format is:

wrapflat
                    patt_name
                     {

                    bitmap

                    Identifier

                    pattern

                    Identifier
                        [
                    substitute

                    Integer

                    Identifier
                    ]
      [
                    dodiffuse
                    ]
      [
                    dotransmit
                    ]
      [
                    doreflect
                    ]
      [
                    filter
```

```
                Integer
                ]
    [
                repeatx
                ]
    [
                repeaty
                ]

                location

                Point

                xaxis

                Vector

                yaxis

                Vector
                   [
                xlength

                Point
                ]
    [
                ylength

                Point
                ]
}
```

The shorthand "wrapfl" may be used in place of "wrapflat".

```
    One of [
                dodiffuse
                ], [
                dotransmit
                ], or [
                doreflect
                ] must be included.
```

1.103   wrapflat.location

```
                Subelement:  location
Shorthand:  loc
Data type:
                Point
                Legal values:  All legal values for
                Point
                Default value:  None
```

This subelement is mandatory.

Description:

    This subelement defines the location of the plane which is used as an
intermediate surface for the pattern mapping. This location corresponds to the
<0,0> pixel coordinate of the bitmap.

See also:
                wrapflat

                wrapflat.xaxis

                wrapflat.yaxis

1.104   wrapflat.xaxis, wrapflat.yaxis

                Subelement: xaxis, yaxis
Shorthand: None
Data type:
                Vector
                Legal values: All legal values for
                Vector
                Default value: None

    These subelements are mandatory.

Description:

    These subelements define the X and Y axes for the plane which is to be
used as the intermediate surface for the pattern mapping. They correspond to
the X and Y axes of the bitmap used in the pattern mapping. If
                xlength
                 or

                ylength
                are not given, then the length of these vectors also determine the ←
                    length in
object space that the bitmap extends.

See also:
                wrapflat

                wrapflat.location

1.105   wrapflat.xlength, wrapflat.ylength

                Subelement: xlength, ylength
Shorthand: xlen, ylen

```
Data type:
                Real
                Legal values:  Greater than zero
Default value:  Computed from
                xaxis
                 and
                yaxis
                    These subelements are optional.
```

Description:

These subelements define the length of the X and Y axis of the bitmap in
object space. If these values are not specified, then they are computed from
the length of the vectors specified by
```
                xaxis
                 and
                yaxis
                .  Using these subelements
```
is usually easier than trying to compute the proper length of
```
                xaxis
                 and
                yaxis
                 by
```
hand.

See also:
```
                wrapflat
```


1.106   wrapflat.repeatx, wrapflat.repeaty

```
                Subelement:  repeatx, repeaty
Shorthand:  repx, repy
Data type:  None
Legal values:  None
Default value:  None
```

These subelements are optional.

Description:

If either of these flags are set, then the bitmap will be caused to
repeat along its X or Y axis (as in a wallpaper pattern). The default action is
to not repeat the bitmap along either axis (as in a decal).

See also:
```
                wrapflat
```


1.107   wrapcylinder

```
                    This pattern wraps a bitmapped image onto the object using a  ←
                    cylinder as
the intermediate surface.   The format is:

wrapcylinder
                patt_name
                 {

                bitmap

                Identifier

                pattern

                Identifier
                    [
                substitute

                Integer

                Identifier
                ]
    [
                dodiffuse
                ]
    [
                dotransmit
                ]
    [
                doreflect
                ]
    [
                xrepeat

                Real
                ]

                height

                Real
                    [
                filter

                Integer
                ]
}

    The shorthand "wrapcy" may be used in place of "wrapcylinder".

    One of [
                dodiffuse
                ], [
                dotransmit
                ], or [
                doreflect
                ] must be included.
```

1.108   wrapcylinder.xrepeat

```
                Subelement:  xrepeat
Shorthand:  xrep
Data type:
                Real
                Legal values:  Greater than zero
Default value:  1.0
```

    This subelement is optional.

Description:

    This subelement controls how many times the bitmap is repeated around
the circumference of the cylinder.

See also:
                wrapcylinder

1.109   wrapcylinder.height

```
                Subelement:  height
Shorthand:  None
Data type:
                Real
                Legal values:  Greater than zero
Default value:  None
```

    This subelement is mandatory.

Description:

    This subelement controls how far the bitmap extends vertically.

See also:
                wrapcylinder

1.110   blotch

```
                This pattern produces a "patchwork" of other patterns.  It is  ←
                    difficult
to describe, so see some of the included sample scripts and images to get an
idea of what this pattern does.  The format is:

blotch
                patt_name
```

```
                    {

                    scale

                    Real

                    pattern

                    Identifier
                    }
```

1.111   blotch.scale

```
              Subelement:  scale
Shorthand:  None
Data type:
              Real
              Legal values:  Greater than zero
Default value:  1.0
```

    This subelement is optional.

Description:

    This subelement changes the apparent size of the pattern on the surface
of the object. Increasing values cause the pattern to become smaller on the
object, while decreasing values cause the pattern to become larger on the
object.

See also:
              blotch

1.112   blotch.pattern

```
              Subelement:  pattern
Shorthand:  patt
Data type:
              Identifier
              Legal values:  The name of any previously defined pattern.
Default value:  None
```

    This subelement must be used at least once. It may be repeated as many
as 32 times.

Description:

    This subelement adds a pattern to the list of those used by "blotch".
As many as 32 different patterns may be used by "blotch".

See also:

```
                    clouds
```

1.113 Textures

                   Like patterns, textures are used to add detail to the surfaces ←
                    of
objects. Unlike patterns, which essentially change the color of an object,
textures control the apparent "roughness" of the object.

    There are two types of textures available in Magic Camera: procedural
and bump mapped. Currently, the only procedural texture available is
                    waves
                    .
The "waves" texture gives the appearance of an undulating, wavy surface.

    Bump mapped textures are similar to bit mapped patterns. The only
difference is that the intensity of the pixels is used to determine the apparent
height of that area of the object, instead of the pixels determining the color
of the object. The three available bump mapped textures are
                    bumpsphere
                    ,

                    bumpcylinder
                    , and
                    bumpflat
                    .

    Textures do their work by "tricking" the ray tracer's shading algorithms
into believing that the surface of the object is not flat. This shading trick
produces the optical illusion that the surface is not flat. However, the actual
surface of the object remains flat. This is obvious if the object is viewed
"edge on". Because the surface distortions causes by textures are only
illusions, at times textures will appear unrealistic. This will happen when
they are viewed from a shallow angle (nearly edge on), when lit from certain
angles, or if you try to make them appear too extreme.

1.114 Texture Name

                   This field is of type
                    identifier
                    , and "names" the texture. When the
texture is attached to objects, it will be referenced by this
                    identifier
                    .

1.115 Waves

This texture produces a wave-like surface. The format is:

```
waves

          text_name
           {
   [
          ncenters

          Integer
          ]
   [
          scale

          Real
          ]
   [
          phase

          Real
          ]
   [
          size

          Real
          ]
}
```

Although all subelements are optional, at least one must be present.

1.116   waves.ncenters

```
          Subelement:  ncenters
Shorthand:  None
Data type:
          Integer
          Legal values:  1 to 50
Default value:  10
```

This subelement is optional.

Description:

This subelement controls the complexity of the waves. Higher numbers
cause more complex waves. Lower numbers cause simpler waves. Using a value of
one causes a single ring of waves. Higher values increase the time required to
compute the wave shapes.

See also:

          waves

1.117   waves.scale

```
              Subelement:  scale
Shorthand:  None
Data type:
              Real
              Legal values:  Greater than 0.0
Default value:  1.0
```

    This subelement is optional.

Description:

    This subelement controls the scale of the waves on the object. Higher
numbers produce a smaller wave pattern. Lower numbers have the opposite effect.

See also:
```
              waves
```

1.118   waves.phase

```
              Subelement:  phase
Shorthand:  None
Data type:
              Real
              Legal values:  Any value
Default value:  0.0
```

    This subelement is optional.

Description:

    This subelement controls the phase, in degrees, of the waves. By
cycling this value through 360 degrees, the waves can be made to appear to
"roll".

See also:
```
              waves
```

1.119   waves.size

```
              Subelement:  size
Shorthand:  None
Data type:
              Real
              Legal values:  Greater than 0.0
Default value:  1.0
```

    This subelement is optional.

Description:

    This subelement controls the height of the waves on the object. Higher
numbers produce taller waves. Lower numbers have the opposite effect.

    If the is number is too far from zero, the result may become
unrealistic. Typical values are between -1.0 and 1.0.

See also:
                waves

1.120 Spherical Bump Maps

                    This texture applies a bit map texture to the object using the ←
                        same
geometry as
                wrapsphere
                . The format is:

bumpsphere
                text_name
                 {

                bitmap

                Identifier
                    [
                xrepeat

                Real
                ]
    [
                yrepeat

                Real
                ]
    [
                size

                Real
                ]
}

    The shorthand "bumpsp" is an acceptable substitution for "bumpsphere".

1.121 bumpsphere.xrepeat, bumpsphere.yrepeat

                Subelement: xrepeat, yrepeat
Shorthand: xrep, yrep
Data type:

```
                Real
                Legal values:  Greater than zero.
Default value:  1.0
```

    These subelements are optional.

Description:

    These subelements control how many times the bitmap is repeated around
the sphere. The bitmap is repeated "xrepeat" number of times around the equator
of the sphere and "yrepeat" number of times between the poles of the sphere.

See also:
                bumpsphere

1.122   bumpsphere.bitmap, bumpcylinder.bitmap, bumpflat.bitmap

```
                Subelement:  bitmap
Shorthand:  None
Data type:
                Identifier
                Legal values:  Any previously defined bitmap.
Default value:  None
```

    This subelement is mandatory.

Description:

    This subelement defines which bitmap should be used to apply the texture
to the object.

See also:
                bumpsphere

                bumpcylinder

                bumpflat

1.123   bumpsphere.size, bumpcylinder.size, bumpflat.size

```
                Subelement:  size
Shorthand:  None
Data type:
                Real
                Legal values:  Greater than 0.0
Default value:  1.0
```

    This subelement is optional.

Description:

    This subelement controls the height of the texture on the object.
Higher numbers produce taller bumps. Lower numbers have the opposite effect.

    If the is number is too far from zero, the results may become
unrealistic. Typical values are between -1.0 and 1.0.

See also:

                bumpsphere

                bumpcylinder

                bumpflat

1.124 Cylindrical Bump Maps

                This texture applies a bit map texture to the object using the ←
                    same
geometry as
                wrapcylinder
                . The format is:

bumpcylinder
                text_name
                 {

                bitmap

                Identifier
                    [
                xrepeat

                Real
                ]
    [
                height

                Real
                ]
    [
                size

                Real
                ]
}

    The shorthand "bumpcy" is an acceptable substitution for
"bumpcylinder".

1.125 bumpcylinder.xrepeat

```
                 Subelement:  xrepeat
Shorthand:  xrep
Data type:
                 Real
                 Legal values:  Greater than zero
Default value:  1.0
```

    This subelement is optional.

Description:

    This subelement controls how many times the bitmap is repeated around
the circumference of the cylinder.

See also:
                 bumpcylinder

1.126   wrapcylinder.height

```
                 Subelement:  height
Shorthand:  None
Data type:
                 Real
                 Legal values:  Greater than zero
Default value:  None
```

    This subelement is mandatory.

Description:

    This subelement controls how far the bitmap extends vertically.

See also:
                 bumpcylinder

1.127   Flat Bump Maps

```
                 This texture applies a bit map texture to the object using the ←
                     same
geometry as
                 wrapflat
                 .  The format is:

bumpflat
                 text_name
                  {

                 bitmap
```

```
                    Identifier

                    size

                    Real
                        [
                    repeatx
                    ]
        [
                    repeaty
                    ]

                    location

                    Point

                    xaxis

                    Vector

                    yaxis

                    Vector
                        [
                    xlength

                    Point
                    ]
        [
                    ylength

                    Point
                    ]
}
```

The shorthand "bumpfl" is an acceptable substitution for "bumpflat".

1.128 bumpflat.location

```
                    Subelement:  location
Shorthand:  loc
Data type:
                    Point
                    Legal values:  All legal values for
                    Point
                    Default value:  None
```

This subelement is mandatory.

Description:

This subelement defines the location of the plane which is used as an
intermediate surface for the bump mapping. This location corresponds to the
<0,0> pixel coordinate of the bitmap.

See also:

                bumpflat

                bumpflat.xaxis

                bumpflat.yaxis

1.129   bumpflat.xaxis, bumpflat.yaxis

                Subelement: xaxis, yaxis
Shorthand: None
Data type:
                Vector
                Legal values: All legal values for
                Vector
                Default value: None

    These subelements are mandatory.

Description:

    These subelements define the X and Y axes for the plane which is to be
used as the intermediate surface for the bump mapping. They correspond to the X
and Y axes of the bitmap used in the bump mapping. If
                xlength
                 or
                ylength
                 are
not given, then the length of these vectors also determine the length in object
space that the bitmap extends.

See also:

                bumpflat

                bumpflat.location

1.130   bumpflat.xlength, bumpflat.ylength

                Subelement: xlength, ylength
Shorthand: xlen, ylen
Data type:
                Real
                Legal values: Greater than zero
Default value: Computed from
                xaxis
                 and
                yaxis

These subelements are optional.

Description:

    These subelements define the length of the X and Y axis of the bitmap in
object space. If these values are not specified, then they are computed from
the length of the vectors specified by
                xaxis
                 and
                yaxis
                . Using these subelements
is easier than trying to compute the proper length of
                xaxis
                 and
                yaxis
                 by hand.

See also:
                bumpflat

1.131 bumpflat.repeatx, bumpflat.repeaty

                Subelement: repeatx, repeaty
Shorthand: repx, repy
Data type: None
Legal values: None
Default value: None

    These subelements are optional.

Description:

    If either of these flags are set, then the bitmap will be caused to
repeat along its X or Y axis (as in a wallpaper pattern). The default action is
to not repeat the bitmap along either axis (as in a decal).

See also:
                bumpflat

1.132 Primitive Object Types

                Primitives are the basic building blocks of the objects found ←
                    lying
about ray traced scenes. No matter what you see, all Magic Camera scenes are
filled with triangles, parallelograms, planes, spheres, and rings.
    Triangles and parallelograms make up the bulk of the objects seen,
although they are the least common found in scripts. Most often, they are
specified indirectly by using

```
                constructed elements
                 such as boxes, spins, and
extrudes.
```

The primitive elements are:

```
                Triangles

                Parallelograms

                Planes

                Spheres

                Rings
```

1.133 pattern

```
                Subelement:  pattern
Shorthand:  patt
Data type:
                Identifier
                Legal values:  The name of any previously declared
                pattern
                Default value:  None
```

```
    This subelement is mandatory for all
                primitives
                 and all

                constructed elements
                 .
```

Description:

```
    Attaches a named
                pattern
                 to a primitive.
```

1.134 texture

```
                Subelement:  texture
Shorthand:  text
Data type:
                Identifier
                Legal values:  The name of any previously declared
                texture
                Default value:  None
```

```
    This subelement is optional for all
                primitives
                 and all

                constructed elements
                .
```

Description:

```
    Attaches a named
                texture
                 to an object.
```

1.135   origin

```
                Subelement:  origin
Shorthand:  None
Data type:
                Point
                Legal values:  Any legal value for
                Point
                Default value:  None

    This subelement is optional for all
                primitives
                 and all

                constructed elements
                .
```

Description:

```
    Causes the given point to be considered the origin for
                pattern
                 and

                texture
                 computation.  If you are building an object and require no  ←
                    discontinuities in

                patterns
                 and
                textures
                 across the boundaries of individual elements, then use the " ←
                    origin"
statement to give them the same reference point for pattern computation.

    Normally, the origin is derived from some part of the element's
description, normally the "location" statement.  The "origin" statement
overrides this, giving better user control.
```

1.136 offtree

```
                  Subelement:  offtree
Shorthand:  None
Data type:  None
Legal values:  None
Default value:  None
```

This subelement is optional for all
                primitives
                 and all

                constructed elements
                .

Description:

   Occasionally, a single primitive will be so large as to interfere with

                octree
                 construction. In this case, use the "offtree" flag to cause the ←
                    object to be
considered a global object and not be added to the
                octree
                .

   The only example I can give of the usefulness of this is a script I was
working on in which a spacecraft was in the foreground with a planet (a large
sphere) in the background. The sphere was large enough to dwarf the spacecraft,
and caused the
                octree
                 to become heavily unbalanced. I added this flag to
prevent this, and trace time was reduced considerably.

   Keep in mind that this flag causes the object to generate an intersect
test for ALL rays traced (which doesn't normally happen), so use it sparingly.

1.137 Triangles

                    The triangle primitive creates a single triangle. Like the

                parallelogram
                 primitive, it is rarely used directly in scripts. Its most ←
                    common use is by
programs which translate other object formats into Magic Camera scripts.

   There are two formats for the triangle primitive. One uses named
vertices, the other does not.

Without named vertices:

triangle {

```
                location

                Point

                v1

                Vector

                v2

                Vector

                pattern

                Identifier
                    [
                texture
                ]
                Identifier
                ]
    [
                origin

                Point
                ]
    [
                offtree
                ]
}

Using named vertices:

triangle {

                p1

                Identifier

                p2

                Identifier

                p3

                Identifier

                pattern

                Identifier
                    [
                texture

                Identifier
                ]
    [
                origin
```

```
                        Point
                        ]
        [
                        offtree
                        ]
}
```

In either case, the shorthand "tri" can be used in place of the full
word "triangle"

1.138 triangle.location

```
                        Subelement:  location
Shorthand:  loc
Data type:
                        Point
                        Legal values:  Any legal value for
                        Point
                        Default value:  None
```

This subelement is mandatory.

Description:

Defines the location of the first corner of the triangle. This point is
also considered the origin for
                        pattern
                         and
                        texture
                         computation.

See Also:
                        triangle

                        origin

1.139 triangle.v1, triangle.v2

```
                        Subelement:  v1, v2
Shorthand:  None
Data type:
                        Vector
                        Legal values:  Any legal value for
                        Vector
                        Default value:  None
```

These subelements are mandatory.

Description:

```
    Defines the vectors from the
                location
                 of the triangle to the second and
third corners of the triangle.  Note that these are vectors, and not absolute
points.
```

Example:

```
    If the triangle to be created has the corners <3, 4, 2>, <4, 2, 1>, and
<4, 3, 3>, the
                location
                 would be <3, 4, 2>, v1 would be <1, -2, -1>, and v3
would be <1, -1, 1>.
```

See Also:

```
                triangle
```

1.140   triangle.p1, triangle.p2, triangle.p3

```
                Subelement:  p1, p2, p3
Shorthand:  None
Data type:
                Identifier
                Legal values:  Any previously declared
                vertex
                .
Default value:  None
```

These subelements are mandatory.

Description:

```
    Defines the vertices to be used as the three corners of the triangle.
The vertex named by p1 is considered to be the origin for pattern and texture
computations.
```

See Also:

```
                triangle

                origin
```

1.141   Parallelograms

```
                 The parallelogram primitive is very similar to the
                triangle
                 primitive,
except that a four-sided parallelogram is created.  Also, the parallelogram
declaration does not allow vertex based definitions.
```

    When defining a parallelogram, only three points are explicitly

specified. The fourth is computed from these three. This is because, for
parallelograms, when three points are given, there is only one possible point
which will complete the parallelogram.

The definition:

parallelogram {

                location

                Point

                v1

                Vector

                v2

                Vector

                pattern

                Identifier
                    [
                texture
                ]
                Identifier
                ]
    [
                origin

                Point
                ]
    [
                offtree
                ]
}

    Sharp observers will note that this is exactly the same as for

                triangles
                    .

    The shorthand "paragram" can be used in place of the full word
"parallelogram". The term "quad" will also be accepted for historical reasons.

1.142 parallelogram.location

                Subelement: location
Shorthand: loc
Data type:
                Point
                Legal values: Any legal value for
                Point

```
                Default value:  None
```

    This subelement is mandatory.

Description:

    Defines the location of the first corner of the parallelogram. This
point is also considered the origin for
                pattern
                 and
                texture
                 computation.

See Also:

                parallelogram

                origin

1.143   parallelogram.v1, parallelogram.v2

```
                Subelement:  v1, v2
Shorthand:  None
Data type:
                Vector
                Legal values:  Any legal value for
                Vector
                Default value:  None
```

    These subelements are mandatory.

Description:

    Defines the vectors from the
                location
                 of the parallelogram to the second
and third corners of the parallelogram. The fourth corner of the parallelogram
is computed by adding
                location
                , v1, and v2 together.

    Note that these are vectors, not absolute points.

Example:

    If the parallelogram to be created has the corners <3, 4, 2>, <4, 2, 1>,
<4, 3, 3>, and <5, 1, 2>, the
                location
                 would be <3, 4, 2>, v1 would be <1, -2,
-1>, and v3 would be <1, -1, 1>.

See Also:

                parallelogram

1.144 Planes

A plane is an infinite flat surface. The most common use of a ←
plane is
to define the "ground" or "floor" of a scene. The definition of a plane looks
similar to that of a triangle or a parallelogram. Note that a plane does not
have the "offtree" subelement, since a plane is infinite and therefore cannot be
bounded by an
octree
.

```
plane {

                location

                Point

                v1

                Vector

                v2

                Vector

                pattern

                Identifier
                    [
                texture
                ]
                Identifier
                ]
    [

                origin

                Point
                ]
}
```

1.145 plane.location

Subelement: location
Shorthand: loc
Data type:
                Point
                Legal values: Any legal value for
                Point
                Default value: None

   This subelement is mandatory.

Description:

Defines one point through which the plane will pass. This point is also
considered the origin for
                pattern
                 and
                texture
                 computation.

See Also:
                plane

                origin

1.146   plane.v1, plane.v2

                Subelement: v1, v2
Shorthand: None
Data type:
                Vector
                Legal values: Any legal value for
                Vector
                Default value: None

    These subelements are mandatory.

Description:

    Defines the vectors with which the plane will be parallel. Another way
to think of it is that the plane will pass through the points specified by

                location
                ,
                location
                 plus v1, and
                location
                 plus v2.

Example:

    If the plane to be created is to pass through the points <0, 5, 0>, <3,
4, 1>, and <2, 2, 3>, the
                location
                 would be <0, 5, 0>, v1 would be <3, -1, 1>,
and v2 would be <2, -3, 3>.

To define a ground plane:

```
plane {
    loc     <0, 0, 0>
    v1      <1, 0, 0>
    v2      <0, 0, 1>
    patt    ground.patt
}
```

See Also:

                    plane

1.147 Spheres

                    Using the "sphere" element creates a "perfect" sphere (as ←
                        opposed to a
"geometric" sphere made up of several triangles).

The definition of a sphere is:

sphere {

                location

                Point

                radius

                Real

                pattern

                Identifier
                    [
                texture
                ]
                Identifier
                ]
    [
                origin

                Point
                ]
    [
                offtree
                ]
}

Note: A sphere cannot be rotated. If a sphere is rotated, the
                pattern
                 or

                texture
                 attached to the sphere will not appear to rotate with the sphere ←
                    (it will not
move at all), so the sphere will appear as if it has not rotated. To produce
spheres which behave properly under all transformations, use
                psphere
                .

1.148 sphere.location

```
              Subelement:  location
Shorthand:  loc
Data type:
              Point
              Legal values:  Any legal value for
              Point
              Default value:  None
```

    This subelement is mandatory.

Description:

    Defines the center of the sphere. This point is also considered the
origin for
```
              pattern
               and
              texture
               computation.
```

See Also:
```
              sphere

              origin
```

1.149 sphere.radius

```
              Subelement:  radius
Shorthand:  None
Data type:
              Real
              Legal values:  Greater than zero
Default value:  None
```

    This subelement is mandatory.

Description:

    Defines the radius of the sphere.

See Also:
```
              sphere
```

1.150 Rings

              To create a ring or disk shape, use the "ring" element.

The definition of a ring is:

```
ring {

                 location

                 Point

                 v1

                 Vector

                 v2

                 Vector

                 in

                 Real

                 out

                 Real

                 pattern

                 Identifier
                     [
                 texture

                 Identifier
                 ]
     [
                 origin

                 Point
                 ]
     [
                 offtree
                 ]
}
```

Note: Rotating a ring has unpredictable results. If a
                 pattern
                  or
                 texture
                  is
attached to the ring, the
                 pattern
                  or
                 texture
                  may appear to rotate in an unusual
manner. Otherwise, the ring itself will rotate normally. Fully transformable
rings can be created using the
                 spin construct
                 .

1.151   ring.location

```
                Subelement:  location
Shorthand:  loc
Data type:
                Point
                Legal values:  Any legal value for
                Point
                Default value:  None
```

   This subelement is mandatory.

Description:

   Defines the center of the ring. This point is also considered the
origin for
```
                pattern
                 and
                texture
                 computation.
```

See Also:
```
                ring

                origin
```

1.152   ring.v1, ring.v2

```
                Subelement:  v1, v2
Shorthand:  None
Data type:
                Vector
                Legal values:  Any legal value for
                Vector
                Default value:  None
```

   These subelements are mandatory.

Description:

   Defines the vectors with which the plane on which the ring lies will be
parallel. Another way to think of it is that the ring will lie on a plane which
will pass through the points specified by
```
                location
                ,
                location
                 plus v1, and

                location
                 plus v2.
```

Example:

If the ring to be created is to pass through the points <0, 3, 0>and be
parallel to the plane y=0, the
                location
                 would be <0, 3, 0>, v1 would be <1, 0,
0>, and v2 would be <0, 0, 1>.

See Also:
                ring

1.153   ring.in, ring.out

                Subelement: in, out
Shorthand: None
Data type:
                Real
                Legal values: "in" must be greater than or equal to zero.
        "out" must be greater than "in".
Default value: None

    These subelements are mandatory.

Description:

    Defines the inner and outer radii or the ring. If the inner radius is
equal to zero, a disk will be formed instead of a ring.

See Also:
                ring

1.154   Primitive Constructions

                    As you can imagine, building complex shapes from triangles can ←
                    be a
trying task. In order to help out, the Magic Camera parsing routines can build
some shapes for you. These shapes include rotational solids ("lathed") objects,
extruded objects, etc. These are the primary means of building objects using
Magic Camera.

    The types of constructed objects supported are:

                Height Fields

                Rotational Solids

                Boxes

                Extrusions

                Filled Polygons

```
                Skinned Polygon Frames

                Polygon Spheres
```

1.155 Smoothing Constructions

```
                Subelement:  smooth
Shorthand:  None
Data type:  None
Legal values: None
Default value: None

    This subelement is optional.

Description:

    This flag causes the smoothing algorithms to be applied to the
construction.  It is equivalent the placing the construction inside of a
"smoothon/smoothoff" pair.

See Also:
                Smoothing
```

1.156 Height Fields

```
                A height field is used to build terrain-like objects.   ←
                Essentially, a
plane is built from a grid of triangles.  The vertices of the triangles are the
perturbed vertically from the plane according to data read in from a separate
file (see
                hfield.file
                 for the format of this file).  The format for a height
field is:

hfield {

                location

                Point

                v1

                Vector

                v2

                Vector
                    [
                up
```

```
                Vector
                ]
    [
                height

                Real
                ]
    [
                floor

                Real
                ]

                file

                Filename

                pattern

                Identifier
                    [
                texture

                Identifier
                ]
    [
                origin

                Point
                ]
    [
                smooth
                ]
}
```

1.157   hfield.location

```
                Subelement:  location
Shorthand:  loc
Data type:
                Point
                Legal values: any legal value
Default value:  None

    This subelement is mandatory.

Description:

    Defines the location of the plane which is used to build the height
field.  It is similar to
                plane.location
                See Also:
                hfield
```

1.158   hfield.v1, hfield.v2

```
                Subelement:  v1, v2
Shorthand:  None
Data type:
                Vector
                Legal values: any legal value
Default value:  None
```

    These subelement are mandatory.

Description:

    Defines the two vectors along which lies the plane used to build the
height field. It is similar to
                plane.v1 and plane.v2
                See Also:
                hfield

1.159   hfield.up

```
                Subelement:  up
Shorthand:  None
Data type:
                Vector
                Legal values: any legal value
Default value:  Computed from hfield.v1 and hfield.v2
```

    This subelement is optional.

Description:

    Defines the direction which is considered "up" from the plane of the
height field. If it is not specified, it is computed to be perpendicular to the
plane (the cross product of
                hfield.v1 and hfield.v2
).

See Also:
                hfield

1.160   hfield.height

```
                Subelement:  height
Shorthand:  None
Data type:
```

```
                Real
                Legal values: any legal value
Default value:  1.0
```

    This subelement is optional.

Description:

    Defines the height of the height field. This values read from

```
                hfield.file
                 are multiplied by this number, providing easy scaling of the  ↩
                    field without
having to rewrite the (potentially large) file.  The height of any point in the
field is computed by:
```

```
    height = data * hfield.height -
                hfield.floor
                    Where "data" is the value read from
                hfield.file
                 for that point. If the
value of height is less than zero, it is set to zero.
```

See Also:
```
                hfield
```

1.161   hfield.floor

```
                Subelement:  floor
Shorthand:  None
Data type:
                Real
                Legal values: any legal value
Default value:  0.0
```

    This subelement is optional.

Description:

    Defines the minimum value used in the height field. This value is
subtracted from all data points in the height field. The height of any point in
the field is computed by:

```
    height = data *
                hfield.height
                 - hfield.floor
```

    Where "data" is the value read from
```
                hfield.file
                 for that point. If the
value of height is less than zero, it is set to zero.
```

See Also:

```
                    hfield
```

1.162 hfield.file

```
                Subelement:  file
Shorthand:  None
Data type:
                Filename
                Legal values:  the name of any existing file
Default value:  None
```

This subelement is mandatory.

Description:

Tells the parser which file to read the height data from. All numbers
in the file are ASCII, so you can create the file with a text editor. The data
in the file must have the following format:

1. An integer value (dim) giving the dimension of the data. If this
value is 8, then the data forms an 8x8 array.
2. Dim squared floating point values. If dim=8, then there must be 64
floating point values in the file. These values form an array, with the X
dimension increasing across the rows, and the Y dimension increasing down the
columns.

There are no provisions for comments in the file. An example file
follows:

```
4
1.0 .7 .7 1.4
.8 .2 .2 .6
.2 .3 .4 .2
.5 .4 .6 .6
```

See Also:
```
                hfield
```

1.163 Rotational Solids (Spins)

```
                A rotational solid is an object created by spinning a shape  ←
                    about an
axis.  These are the types of objects often created on a woodworker's lathe.
Any object that is circularly symmetrical can be created using the "spin"
command.  The format is:

spin {

                location
```

```
                        Point
                            [
                        segments

                        Integer
                        ]

                        slice

                        Identifier
                            [
                        start

                        Real
                        ]
        [
                        end

                        Real
                        ]
        [
                        rise

                        Real
                        ]
        [
                        fillfirst
                        ]
        [
                        filllast
                        ]

                        pattern

                        Identifier
                            [
                        texture

                        Identifier
                        ]
        [
                        origin

                        Point
                        ]
        [
                        smooth
                        ]
}

NOTE: Currently, all spins are made around the Y axis.
```

1.164 spin.location

```
                Subelement:  location
Shorthand:  loc
Data type:
                Point
                Legal values:  all legal values
Default value:  None
```

    This subelement is mandatory.

Description:

    Gives the location at which the spin is placed. This point will be the
point at which any point <0, 0> in the given
                slice
                 is located.

See Also:
                spin

1.165   spin.segments

```
                Subelement:  segments
Shorthand:  segs
Data type:
                Integer
                Legal values:  greater than or equal to 3
Default value:  8
```

    This subelement is optional.

Description:

    Tells how many different "segments" the spin will be broken into. Since
the spin is made from individual triangles, it has flat sides. This subelement
determines how many flat sides there are to the spin. Increasing this number
produces rounder looking spins, but causes more triangles to be generated,
increasing memory consumption and rendering time.

See Also:
                spin

1.166   spin.slice

```
                Subelement:  slice
Shorthand:  None
Data type:
                Identifier
                Legal values:  the name of any previously defined
                slice
```

```
                    Default value:  None
```

    This subelement is mandatory.

Description:

    Gives the name of the slice which defines the shape to be rotated. In
this slice, the first coordinate of each 2-D point indicates the radius, the
second of each pair indicates the height above (or below, for negative numbers)

```
                    spin.location
                    .
```

See Also:
```
                    spin
```

1.167   spin.start, spin.end

```
                    Subelement:  start, end
Shorthand:  None
Data type:
                    Real
                    Legal values:  all legal values, end must be larger than start
Default value:  start=0.0, end=360.0
```

    These subelements are optional.

Description:

    These two subelements allow spins which are not exactly 360 degrees
around. Each takes a value in degrees of where the spin is to start or to end.
Zero degrees corresponds to the positive X axis.
    Note that the values can be greater than 360 degrees, this is
particularly useful in creating spiral shapes which spiral around for more than
one revolution.

See Also:
```
                    spin

                    spin.rise

                    spin.fillfirst, spin.filllast
```

1.168   spin.rise

```
                    Subelement:  rise
Shorthand:  None
Data type:
                    Real
                    Legal values:  all legal values
```

```
Default value:  0.0
```

    This subelement is optional.

```
Description:
```

    If a value for rise is specified, spiraling shapes can be made. As the
slice is rotated about the axis, it is shifted upwards (or downwards, for
negative values) so that the end of the spin is "rise" units higher than the
first.

```
See Also:
                spin
```

1.169   spin.fillfirst, spin.filllast

```
                Subelement:  fillfirst, filllast
Shorthand:  first, last
Data type:  None
Legal values:  None
Default value:  None
```

    These subelements are optional.

```
Description:
```

    When using
                spin.start, spin.end
                 or
                spin.rise
                , the start and end of the
spin will be unfilled, or open ended. Specifying "fillfirst" or "filllast"
closes the starting slice or ending slice, respectively.

```
See Also:
                spin
```

1.170   Boxes

                Sometimes you just want to create a simple box. This is how ←
                    you do it:

```
box {

                location

                Point

                v1
```

```
                Vector

                v2

                Vector

                v3

                Vector

                pattern

                Identifier
                   [
                texture

                Identifier
                ]
      [
                origin

                Point
                ]
      [
                smooth
                ]
}
```

1.171 box.location

```
                Subelement:  location
Shorthand:  loc
Data type:
                Point
                Legal values:  All legal values
Default value:  None

    This subelement is mandatory.

Description:

    Determines the location of one corner of the box.  The other locations
are determined by
                box.v1, box.v2, and box.v3
                .

See Also:
                box
```

1.172 box.v1, box.v2, box.v3

```
                Subelement:  v1, v2, v3
Shorthand:  None
Data type:
                Vector
                Legal values:  All legal values
Default value:  None
```

    These subelements are mandatory.

Description:

    Determines the location of the corners of the box in relation to

          box.location
          . The exact locations of the eight corners of the box are ←
             determined by:

```
                                          7--------8
    Corner 1: loc                        /|       /|
    Corner 2: loc+v1                     / |      / |
    Corner 3: loc+v2                    4--|-----6  |
    Corner 4: loc+v3                    |  |     |  |
    Corner 5: loc+v1+v2              v3|  3-----|--5
    Corner 6: loc+v1+v3                | /v2    | /
    Corner 7: loc+v2+v3                |/       |/
    Corner 8: loc+v1+v2+v3             1--------2
                                          v1
```

    Note that v1, v2, and v3 do not have to be orthogonal. That is, the box
can be slanted.

See Also:
                box

1.173 Filled Slices

          Filling a previously defined slice is an easy way to create a ←
           complex
polygon from triangles. The format for a fill is:

fill {

          location

          Point

          slice

          Identifier
            [
          hole

```
                Identifier
                ]

                xaxis

                Vector

                yaxis

                Vector

                pattern

                Identifier
                    [
                texture

                Identifier
                ]
    [
                origin

                Point
                ]
}
```

1.174   fill.location

```
                Subelement:  location
Shorthand:  loc
Data type:
                Point
                Legal values:  All legal values
Default value:  None
```

This subelement is mandatory.

Description:

Determines the location of the fill. The point <0, 0> in
                fill.slice
                would be placed here.

See Also:
                fill

1.175   fill.slice

```
                Subelement:  slice
Shorthand:  None
```

```
Data type:
                Identifier
                Legal values:  Any previously defined
                slice
                Default value:  None
```

    This subelement is mandatory.

Description:

    Determines which slice is to be used in the fill. This slice defines
the outline of the polygon. The slice must have been defined using the "closed"
subelement. The slice may not cross itself, that is, no edge of the slice may
cross any other edge of the slice.

See Also:
                fill

1.176   fill.hole

```
                Subelement:  hole
Shorthand:  None
Data type:
                Identifier
                Legal values:  Any previously defined
                slice
                Default value:  None
```

    This subelement is optional.

Description:

    This slice is used to cut a hole from inside the filled polygon. The
slice must have been defined using the "closed" subelement. The hole must be
completely contained within the slice being filled and may not cross itself,
that is, no edge of the hole may cross any other edge of the hole. Currently,
only one hole may be used per fill.

See Also:
                fill

1.177   fill.xaxis, fill.yaxis

```
                Subelement:  xaxis, yaxis
Shorthand:  None
Data type:
                Vector
                Legal values:  All legal values
Default value:  None
```

These subelements are mandatory.

Description:

    Determines the orientation of the polygon in 3D space. The xaxis
corresponds with the first coordinate in the slice's 2D pair, the yaxis
corresponds with the second.

See Also:
                fill

1.178 Skinned Polygon Frames

                    This command will allow you to build very complex shapes in ↩
                    much the
same manner as is used to build wooden model airplanes. Previously defined
slices are used as "forms" or "ribs", and a skin is stretched over these slices.
 The format is:

```
skin {

                slice

                Identifier

                Point

                Vector

                Vector

                fillfirst

                filllast

                pattern

                Identifier
                    [
                texture

                Identifier
                ]
    [
                origin

                Point
                ]
    [
                smooth
                ]
}
```

1.179   skin.slice

```
              Subelement:  slice
Shorthand:  None
Data type:
              Identifier

              Point

              Vector

              Vector
              Legal values:  Any previously define
              slice
              , any legal values for the point and
the vectors
Default value:  None
```

   This subelements is mandatory. It may be repeated as many times as is
necessary.

Description:

   This complex subelement takes four arguments. The first is the name of
a slice which is to be used as the form for this section of the skin. The
second is a point which gives the location of the form in 3D space. The two
vectors are the X and Y axes for on which the slice is oriented. The skin will
be built over the slices in the order that they appear. At least two slices
must appear. All slices must be either closed or not closed.

See Also:
              skin

1.180   skin.fillfirst, skin.filllast

```
              Subelement:  fillfirst, filllast
Shorthand:  first, last
Data type:  None
Legal values:  None
Default value:  None
```

   These subelements are optional.

Description:

   Causes either the first and/or the last of the
              slices
               to be filled.

See Also:

                skin

1.181 Extrusions

                This is a simple version of the
                skin
                 command. It allows only one slice
to be used. Essentially, that slice is stretched, or extruded, from two
dimensions into three. This could be used, for example, to create a 3D font.
The format is:

extrude {

                location

                Point

                xaxis

                Vector

                yaxis

                Vector

                slice

                Identifier

                direct

                Vector

                length

                Real

                fillfirst

                filllast

                pattern

                Identifier
                   [
                texture

                Identifier
                ]
      [
                origin

                Point

```
                    ]
    [
                    smooth
                    ]
}
```

1.182 extrude.location

```
                Subelement:  location
Shorthand:  loc
Data type:
                Vector
                Legal values:  All legal values
Default value:  None
```

    This subelement is mandatory.

Description:

    The location of the extrude.

See Also:

                extrude

1.183 extrude.xaxis, extrude.yaxis

```
                Subelement:  xaxis, yaxis
Shorthand:  None
Data type:
                Vector
                Legal values:  All legal values
Default value:  None
```

    These subelements are mandatory.

Description:

    Determines the orientation of the extrude in 3D space.

See Also:

                extrude

1.184 extrude.slice

```
                Subelement:  slice
Shorthand:  None
Data type:
```

```
                Identifier
                Legal values:  The name of a previously defined slice
Default value:  None

    This subelement is mandatory.
```

Description:

```
    Names the slice to be used in the extrude.
```

See Also:

```
                extrude
```

1.185   extrude.direct

```
                Subelement:  direct
Shorthand:  dir
Data type:
                Vector
                Legal values:  All legal values
Default value:  None

    This subelement is mandatory.
```

Description:

```
    Determines the direction (away from
                extrude.loc
                ) in which the extrusion
will be done.  If
                extrude.length
                 is not specified, the length of this vector
determines the length of the extrude.
```

See Also:

```
                extrude
```

1.186   extrude.length

```
                Subelement:  length
Shorthand:  len
Data type:
                Real
                Legal values:  Greater than zero
Default value:  Computed from
                extrude.direct
                    This subelement is optional.
```

Description:

```
    Determines the length of the extrusion.  If this subelement is defined,
the length of the
                extrude.direct
                 is irrelevant.

See Also:
                extrude
```

1.187   extrude.fillfirst, extrude.filllast

```
                Subelement:  fillfirst, filllast
Shorthand:  first, last
Data type:  None
Legal values:  None
Default value:  None

    These subelements are optional.

Description:

    These subelements cause either the front and/or rear faces of the
extrude to be filled.

See Also:
                extrude
```

1.188   Spheres

```
                Using the "psphere" element creates a "geometric" sphere made  ←
                    up of
several triangles.

The definition of a psphere is:

psphere {

                location

                Point

                radius

                Real

                pattern

                Identifier

                hsegments
```

```
                    Integer

                    vsegments

                    Integer
                        [
                    texture
                    ]
                    Identifier
                    ]
        [
                    origin

                    Point
                    ]
        [
                    offtree
                    ]
        [
                    smooth
                    ]
}
```

1.189   psphere.location

```
                    Subelement:  location
Shorthand:  loc
Data type:
                    Point
                    Legal values:  Any legal value for
                    Point
                    Default value:  None
```

   This subelement is mandatory.

Description:

   Defines the center of the sphere. This point is also considered the
origin for

```
                    pattern
                     and
                    texture
                     computation.
```

See Also:

```
                    psphere

                    origin
```

1.190   psphere.radius

```
                  Subelement:  radius
Shorthand:  None
Data type:
                  Real
                  Legal values:  Greater than zero
Default value:  None
```

    This subelement is mandatory.

Description:

    Defines the radius of the sphere.

See Also:

```
                  psphere
```

1.191   psphere.hsegments, psphere.vsegments

```
                  Subelement:  hsegments, vsegments
Shorthand: hsegs, vsegs
Data type:
                  Integer
                  Legal values:  3 or higher
Default value: 16, 8
```

    This subelement is optional.

Description:

    Defines how many division will make up the sphere. The value of
hsegments defines the number of divisions around the equator. The value of
vsegments defines the number of divisions between the poles. Higher values make
better spheres but increase trace time.

See Also:

```
                  psphere
```

1.192   Named Objects and Instancing

    In order to allow animation, Magic Camera is capable of naming ←
                  and
instancing objects. This works by giving an object, or a group of objects, a
name, and then invoking that object by instancing it. This allows an object to
be described once and used many times, and it allows an object to be moved,
rotated, and scaled.

    To name an object, enclose it inside an object/endobject pair:

```
object object_name

/* primitives and constructions here */

endobject
```

Where "object_name" is an
                Identifier
                 which will be used as the name of
the object. All primitives and constructions inside the object/endobject pair
will belong to the named object. Other elements, such as patterns and textures,
may appear inside the object/endobject pair, but they will not specifically
belong to that object.

When primitives and constructions appear inside an object/endobject
pair, they will not be immediately added to the scene. To make them a part of
the scene, they must be instanced:

```
instance object_name
```

A more complex form of instancing allows the patterns and textures to be
changed:

```
instance {
    object      object_name
    subpatt     old_pattern_name new_pattern_name    /* optional, may
                                                        be repeated */
    subtext     old_texture_name new_texture_name    /* optional, may
                                                        be repeated */
}
```

Any primitive using a pattern named "old_pattern_name" will be given the
pattern named "new_pattern_name". Likewise for textures. If the original
object has no textures, use the texture name "NULLTEXT" for "old_texture_name".

Once an object has been named, it may be transformed before being
instanced:

```
    /* move the object by the point value given */
translate object_name
                Point
                    /* scale the object */
scale object_name
                Point
                    /* rotating around the X, Y, or Z axes */
xrotate object_name
                Real
                    /* value in degrees */
yrotate object_name
                Real
                    /* value in degrees */
zrotate object_name
                Real
                    /* value in degrees */

    /* to return an object to its original state */
```

```
reset object_name
```

    Transformations of an object take affect only if it appears before
instancing. Use "reset" to get an object back to its original state if you
intend to instance the object again. That way, you'll be sure of where the
object is when it is instanced or transformed again.

    When an object is transformed, its pattern and texture are not
transformed with it. This causes the object to appear to move "through" the
pattern/texture. To prevent this, use:

```
lockobject object_name
```

    before transforming. Locking an object prevents most pattern/texture
problems in transformation. However, spheres and rings may still suffer
problems under rotation due to the way they are formed. If you must rotate a
sphere or a ring, try building it by using the
                spin
                 element instead.

    To free up the memory used by the object after it has been instanced,
use:

```
killobject object_name
```

    Do not instance an object after it has been freed. This command only
destroys the back-up copy of the object. Instances of the object will not be
affected.


```
Child Objects
-------------
```

    This next section explains the most powerful part of instancing objects.
 Imagine you've built an object which is an airplane, and you want to animate
the plane. As the plane moves, the propeller spins; that is, the propeller is
part of the plane, but it rotates independently of the plane. This is
accomplished by using the "child" command inside of an object/endobject pair:

```
object airplane
child propeller
    /* the rest of the plane here */
endobject
```

    The propeller has now become a child of the airplane. Where ever the
plane goes, the propeller goes with it. But the propeller can also be
transformed independently, so it can be rotated:

```
zrotate propeller 5
```

    In general, any child of an object does all the same transformations as
its parent does, and, in addition, does all of its own also. This is different
from:

```
object airplane
instance propeller
```

```
    /* the rest of the plane */
endobject
zrotate propeller 5
```

In this example, the propeller is not a child of the airplane, it is a part of the airplane. The "zrotate" has no effect on the propeller (until it's instanced again, and then the propeller is rotate only for instances following "zrotate").

Keep in mind that, after an object has been made into a child, any transformations on that object will affect the child. So if you're using child objects, and strange transformations occur, check for transformations later in the file that could affect the child.

1.193 'Undocumented' Features

This section contains features which I consider experimental and extremely risky to use. They may not work, they may cause the renderer to crash or lock up, they may cause syntax errors, or they may do nothing at all.

You should consider this a preview of coming attractions, but don't expect them right away. Some of these are causing me a headache, which is why they're not finished.


```
Constructed Solid Geometry (CSG)
--------------------------------
```

This needs much work. It is on the priority list somewhere right after finishing this documentation.
    In short, CSG allows objects to be built by adding one object to or subtracting it from another. For example, using a cylinder to drill a hole in a sphere. CSG allows very complex shapes to be built.
    The basic framework for CSG is written. However, shadows and reflections do no work properly with CSG. I also need to work out how to transform and instance objects built with CSG.
    Everything you need to know in order to use CSG (in its limited current implementation) is in the parser and scanner description files included in the package. If you can read these files, try CSG (on VERY simple objects) and see how it works.

1.194 Technical Description

                    This section of the documentation was intended to give you a ←
                        better
understanding of some of Magic Camera's peculiarities. It will probably end up being a mish-mash of topics that are written as they come up. In either case, it will hopefully explain why some things happen and other things don't. Most likely, it will bore you.

```
    Topics Discussed
    ----------------

                The Magic Camera Color Model

                Anti-Aliasing

                Ray-intersect Optimization Using an Octree
```

1.195   The Magic Camera Color Model

```
                    A rendering program's color model is the method it uses to  ↩
                        shade the
objects it renders.  Typically, this model combines diffuse color, reflection,
transparency, and specular reflection, and is written up in one long equation
which uses lots of confusing symbols.  Here is Magic Camera's long equation with
lots of confusing symbols (it'll take a few paragraphs to get it all in):

    color = diffuse + reflect + transmit + specular         (eq. 1)

    First of all, all symbols designate an
                RGB
                 triplet.  The equation is
actually repeated three times, once each for red, green, and blue.

    Second, all values are always between 0.0 and 1.0.  This causes some
problems, but is necessary.  There is a minimum and a maximum intensity of any
color that your computer can generate at each pixel.  I say this causes problems
because, if each of the elements of the color model can be 1.0, how can the be
added together and still be guaranteed to be less than 1.0?  They can't.  If
Magic camera ever finds a color value which is greater than 1.0, it is reset to
1.0.  Solution:  think a little bit when lighting scenes and specifying colors.
Typically, the sum of
                color.diffuse
                ,
                color.reflect
                , and
                color.transmit
                 should
not be too much more than 1.0.  You should also consider the lamps in the scene.
 The intensities of all lamps added together and multiplied by the sum of all
elements of the color concerned should not be much more than 1.0.  I try to use
1.5 or so.  To put this into an equation:

    (sum of lamp.color for all lamps)
        * (color.diff+color.refl+color.trans) < 1.5 or so

There's no exact maximum limit to the above equation because the lamps will not
always shine on all objects with their maximum intensities.

    Here's how each of the values which go into equation 1 are formed.


    Diffuse
```

```
------
```

When a ray hits an object, and it is determined that that object is visible, more rays are cast to each of the lamps in order to find shadows. (In the case of soft shadows, many shadow rays are cast and the results are averaged.) This value is multiplied by the shading coefficient, which is the dot product of the surface normal of the object and the vector from the intersect point to the lamp. This is done for all lamps, and these are added together, along with
                color.ambient
                 to get the total light falling from the surface. This looks like:

    lamp_light = sum of (lamp.color * (surf_normal dot lamp_ray))
    total_light = sum of (lamp_light [for each non-obstructed lamp]) + color.ambient
    diffuse = total_light * color.diffuse

    Reflect

    If a color is reflective, that is
                color.reflect
                 is not <0, 0, 0>, then a
ray is cast in the appropriate direction to see what is visible. This ray returns reflect_color, which is multiplied by color.reflect.

    reflect = reflect_color * color.reflect

    Transmit

    Transmit is similar to reflect, but more complex. First you must understand that a ray is either inside a transmissive object or not. Rays from the camera are always considered to be not inside of glass. This condition is marked by an "inglass" flag. Every time a ray hits a transmissive surface, "inglass" is toggled. This allows the length that the ray travels inside of glass to be tracked (I'll call this glass_len).

    If the surface is transmissive, that is
                color.transmit
                 is not <0, 0, 0>,
then a ray is cast to get transmit_color. If the original (parent) ray is not in glass, then glass_len is multiplied by
                color.filter
                 . This is subtracted from
color.transmit. The result is then multiplied by color_transmit. If the parent ray is in glass, then it is leaving glass at this intersection, and color.filter is ignored. So:

    For rays entering glass:

    transmit = (color.transmit - (glass_len * color.filter)) * transmit_color

```
    For rays leaving glass:

    transmit = color.transmit * transmit_color


    Specular
    --------

    This component is responsible for those shiny spots on objects.  It
approximates the reflection of a light source on an object.

    First of all, the same ray as cast in "Shadow" above is used to
determine if the light is obstructed or not.  If so, nothing is done.  If it is
not, then a dot product is taken between the incident ray and the vector to the
lamp.  This result is raised to the power of
                color.speccoef
                , then multiplied by

                color.specrefl
                , and then by lamp.color.  The result is:

    specular = lamp.color * color.specrefl * (in_ray DOT
lamp_ray)**color.speccoef



    Conclusion (or should I say, Confusion)
    ----------

    Well, there it is, the Magic Camera color model.  This is how all the
elements of the
                color
                 pattern are brought together.  I hope you understood it.
```

1.196 Anti-Aliasing

```
                    Aliasing is a problem in computer image generation which is  ←↩
                      most
commonly characterized by "jaggies", or stair step distortions along straight
lines.  Anti-aliasing is the process of removing and/or preventing these
distortions.

    Ray tracers form an image using a process called sampling (a ray, is in
fact, a sample).  The rays used have an infinitely small area (actually zero
area), but the pixels in the final image have an extended area.  That is, rays
with an area of zero must be used to fill a pixel which has an area which is
greater than zero.  These zero-area samples must be extrapolated to form the
pixels.

    Matters are complicated by the fact that the edges of objects in the
scene almost never line up with the edges of the pixels.  Therefore, it is
possible for a pixel to contain objects of more than one color.  But the pixel
itself can only be one color, so the best that can be done is to form a weighted
average of all objects in the pixel (an object which occupies 30% of the pixels
```

contributes 30% to the final color of the pixel. Throw in the fact that, using patterns, an object may also be more than one color.

     Given the above two paragraphs, the problem becomes one of finding exactly how much of the pixel is occupied by a given object so that the weighted average may be computed. Magic Camera does this using an anti-aliasing method called adaptive supersampling. Supersampling simply means that more than one sample is taken for each pixel. Adaptive means that these samples are taken only when necessary.

     Magic Camera starts by taking one sample at each corner of the pixel, giving four samples per pixel. Using the corners of the pixel is highly efficient, since these samples may be shared with adjacent pixels. The samples are averaged, and then a check is then performed to see if one the corners of the pixel differs significantly from the average. If the difference exceeds a given amount (see

                maxaadiff
                ), then that quadrant of the pixel is subdivided,
each corner of the quadrant is sampled, the samples are averaged, and the corner samples are again compared with the average, and if necessary, the quadrant is again subdivided. This divide/sample/average/check cycle continues until either the maximum difference is not exceeded, or the pixel has been subdivided

                maxaadepth
                times. The following diagram applies:

             /
       a---+--/e-------b
       | | / | |
       +---h/--i |
       | / | |
       f--/j---g |
       | / |
       |/ |
       / |
      /c--------------d
     /
       edge of the object

     Samples are taken at points a, b, c, and d. They are averaged, and point a differs by more than maxaadiff. So that quadrant is divided, sampled (points a, e, f, and g) and averaged. This time point g differs, so its quadrant is divided, sampled (h, i, j & g). Since maxaadepth would be exceeded by continuing, no further sampling is done. The average of samples g, h, i & j are used in place of point g, then a, e, f and g are averaged and used in place of point a. Points a, b, c, and d are averaged and used as the pixel color. The final color is:

     (((a+e+f+(g+h+i+j)/4)/4)+b+c+d)/4

1.197 The Octree

First of all, let's compute amount of time it takes to ray ↩
                         trace a
scene. We'll say that there are 1000 objects in the scene, and the final image
will be 640 by 480 pixels. To simplify the example, we'll say that there are no
shadows, reflections, or transparencies, and no anti-aliasing is done. One ray
will be traced per pixel (307,200 total rays). Since, in a simple ray tracer,
each ray intersects each object, there will be a total of 307,200,000
intersection tests. On a fairly quick machine (25Mhz 68030), we may get 2000
intersection tests per second. This means that the scene will take about
153,600 seconds, or about 42 hours and 20 minutes to render. Not acceptable,
especially considering the simplicity of the scene (no shadows, etc.).

    So how can the rendering time be decreased? Simple, decrease the number
of objects in the scene (not acceptable), the resolution of the scene (not
acceptable), or buy a faster computer (get your checkbook out).

    Or make the ray tracer smarter. In the above example, each object is
tested for intersection with each ray. However, most rays will only come close
to a few objects. The ideal is to only perform intersection tests on objects
which the ray will actually hit (1 test per ray). This may never be achieved,
but we can sure try. The trouble is finding which objects can be intersected by
the rays. This is done by a process called "object bounding." In object
bounding, "bounding box" is formed around all objects in the scene. If, and
only if, the ray enters the bounding box the objects inside are tested for
intersection.

    Magic Camera implements object bounding using data structure called an
"octree". The basic process is, first a box is placed around the entire scene.
If the number of objects in the box exceed a preset value (see
                maxobcube
                ), then
that box is evenly divided into eight smaller boxes. The number of objects
inside each box is then counted, and any box which has more than maxobcube
objects inside is further subdivided. This continues until either no box
contains more than maxobcube objects inside, or there are
                maxoctdepth
                 levels of
subdivision. It is necessary to limit the number of divisions to prevent the
octree from becoming too large. It is possible for the octree to expand so much
that it will actually slow the rendering down (through overhead processing).
This diagram shows the process in two dimensions:

```
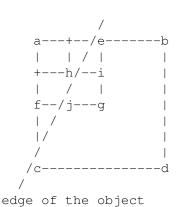    A---J---E-------B
    |o o|   |      o|    maxobcube = 1
    M---N---K       |    maxoctdepth = 2
    | o |   |       |
    H---L---I-------F
    | o     |       |
    |       |       |
    |       |   o   |
    D-------G-------C
```

    First, box ABCD is created around the whole scene. There are six
objects in the box ABCD (more than maxobcube), so the box is divided into four
quadrants (in three dimensions, there would be eight octants formed). Quadrant
AEIH still contains more than maxobcube objects, so it is divided again. Now
box AJNM has two objects, still more than maxobcube, but since there have been

maxoctdepth divisions, it will not be divided.

    An octree is not the ideal way to build bounding boxes, but it is a
solution to the problem which works well for most cases. It has the advantage
that it is adaptive, meaning that the scene is more finely divided only in areas
that require finer subdivisions.

1.198   Run-time Options

                    ToolType Options

                    Options Window

                    Command Line Options

1.199   ToolType Options

                    For a description of what tooltypes are and how to change them ←
                         , see your
Amiga Workbench user's manual.

    Tooltype options are always overridden by values read from input
scripts. In otherwords, they provide a means for setting new defaults.

The following tooltypes may be used:

NOSMOOTH   Cause no polygon smoothing to be done.

FRAME   Sets a value for the FRAME script variable. If
                only one value follows, a single frame will be
                rendered. Default is 0.

FIRSTFRAME   Indicates the first frame to be rendered.
                If not given, will be the same as for FRAME.
                If different from LASTFRAME, multiple frames
                will be rendered. The frame number will be
                appended to the output file's name.

LASTFRAME   Indicates the last frame to be rendered.
                If not given, will be the same as for FIRSTFRAME.
                If different from FIRSTFRAME, multiple frames
                will be rendered. The frame number will be
                appended to the output file's name.

OUTFILE   Indicates the name of the output file. If this
                option is not given, the output file name is
                "mc.out"

MAXRDEPTH   Sets the maximum resolve depth for ray-tracing.
Resolve depth is essentially the number of times
a ray may be reflected or refracted before dying.
Default is 6.

NOSHADOWS   Causes no shadows to be calculated.

NOREFLECT   Causes no reflections to be calculated.

NOTRANS   Causes no transparencies to be calculated.

ILLUMINATE   Causes all objects to appear fully illuminated.

HOLD   Causes Magic Camera to pause before exiting.
MC will wait for the closewindow box to be
clicked before exiting. Use this in
conjunction with NOTRACE to examine the
preview window.

NOTRACE   MC will not render the scene. Use this to see
the preview only.

XRES   Sets the X resolution of the image.
Overrides
camera.res
.

YRES   Sets the Y resolution of the image.
Overrides
camera.res
.

MAXOCTDEPTH   Sets the maximum octree depth. Overrides

maxoctdepth
. Default is 5.

MAXOBCUBE   Sets the maximum number of objects per
octree cube. Overrides
maxoxcube
.
Default is 2.

MAXAADEPTH   Sets the maximum depth for adaptive anit-
aliasing. Overrides
maxaadepth
.
Default is 1.

HFOV   Sets the horizontal field-of-view. Overrides

camera.hfov
. Default is
45 degrees.

VFOV   Sets the vertical field-of-view. Overrides

```
                  camera.vfov
                  .  Default is
                  computed from hfov, aspect, and image resoultion.

ASPECT            Sets the pixel aspect ratio.  Overrides

                  camera.aspect
                  .  Default is 0.56
                  (320x400 interlaced mode).

SQUARE            Sets the pixel aspect ratio to 1.0, and the
                  image resolution to 640x480.  These are
                  common settings for square-pixel images.

LACE              Sets the pixel aspect and resolution to create
                  a 320x400 interlaced Amiga image.

NOLACE            Sets the pixel aspect and resolution to create
                  a 320x200 non-interlaced Amiga image.

HIRES             Sets the pixels aspect and resoultion to create
                  a 640x200 non-interlaced Amiga image.

HIRESLACE         Sets the pixel aspect and resolution to create
                  a 640x400 interlaced Amiga image.

OUTPUT            Sets the format of the output file.  Accepted values
                  are
                  RAW
                  , "QRT" (output format same
                  as from the popular qrt ray-tracer), or "IFF24", for
                  24bit IFF.ILBM output (default).
```

1.200 Command Line Options

```
                  Usage:
------

mc [-nosmooth] [-frame frameno | -frame firstframe lastframe]
    [-quiet] [-terse] [-verbose] [-o outputfile]
    [-rd resolve_depth] [-ns] [-nr] [-nt] [-il] [-hold]
    [-r xres yres] [-notrace] [-mod maxoctdepth]
    [-moc maxobcube] [-aa maxaadepth] [-hfov horizfov]
    [-vfov vertfov] [-fov horizfov vertfov]
    [-aspect aspect_ratio] [-square] [-lace] [-nolace]
    [-out outputformat]
```

    Running Magic Camera from the command line without any options will
cause the options window to be used.

    In the case where an option is entered twice on the command line, the
latter occurence of the option will take precidence, as is the case with
conflicting options (i.e. -quiet -verbose).

Command line options always override any values found in input scripts.

```
Option Description
-----------------
```

```
-nosmooth      Cause no polygon smoothing to be done
```

```
-frame         Sets a value for the FRAME script variable.  If
               only one value follows, a single frame will be
               rendered.  If two values are given, they indicate
               the first and last frame to be rendered, and
               multiple frames will be rendered.  The frame number
               will be appended to the output file's name.
               FIRSTFRAME and LASTFRAME are also set by this option.
               Default is 0.
```

```
-quiet         MC will only print error messages.
```

```
-terse         MC will print only necessary messages.
               This is default.
```

```
-verbose       MC will become quite chatty, telling you
               more than you probably want to know.
```

```
-o             Indicates the name of the output file.  If this
               option is not given, the output file name is
               "mc.out"
```

```
-rd            Sets the maximum resolve depth for ray-tracing.
               Resolve depth is essentially the number of times
               a ray may be reflected or refracted before dying.
               Default is 7.
```

```
-ns            Causes no shadows to be calculated.
```

```
-nr            Causes no reflections to be calculated.
```

```
-nt            Causes no transparencies to be calculated.
```

```
-il            Causes all objects to appear fully illuminated.
```

```
-hold          Causes Magic Camera to pause before exiting.
               MC will wait for a carriage return at the
               command line before exiting.  Use this in
               conjuntion with -notrace to examine the
               preview window.
```

```
-notrace       MC will abort before rendering the scene.
```

```
-r             Sets the X and Y resolution of the image.
               Overrides
                 camera.res
                 .
               Default is 320x400.
```

```
-mod           Sets the maximum octree depth.  Overrides
```

```
                        maxoctdepth
                          .
                        Default is 5.

-moc            Sets the maximum number of objects per
                octree cube.  Overrides
                  maxoxcube
                  .
                Default is 2.

-aa             Sets the maximum depth for adaptive anit-
                aliasing.  Overrides
                  maxaadepth
                  .
                Default is 1.

-hfov           Sets the horizontal field-of-view.  Overrides

                  camera.hfov
                  .
                Default is 45 degrees.

-vfov           Sets the vertical field-of-view.  Overrides

                  camera.vfov
                  .
                Default is computed from hfov, aspect, and
                image resolution.

-fov            Sets the horizontal and vertical field-of-view.
                Overrides both
                  camera.hfov
                  .
                and
                  camera.vfov
                  .

-aspect         Sets the pixel aspect ratio.  Overrides

                  camera.aspect
                  .
                Default is 0.56 (Amiga 320x400 interlace).

-square         Sets the pixel aspect ratio to 1.0, and the
                image resolution to 640x480.  These are
                common settings for square-pixel images.

-lace           Sets the pixel aspect and resolution to create
                a 320x400 interlaced Amiga image.  This is
                default.

-nolace         Sets the pixel aspect and resolution to create
                a 320x200 non-interlaced Amiga image.

-hires          Sets the pixels aspect and resoultion to create
                a 640x200 non-interlaced Amiga image.
```

```
-hireslace     Sets the pixel aspect and resolution to create
               a 640x400 interlaced Amiga image.

-out           Sets the format of the output file.  Accepted values
               are
                 RAW
                 , "QRT" (output format same
               as from the popular qrt ray-tracer), or "IFF24", for
               24bit IFF.ILBM output (default).
```

1.201 Options Window

```
               The options window looks like this...
```

```
+-------------------------------------------------------------+
|                                                             |
| Filename _____ BROWSE     SMOOTHING   |
| Outfile  _____ BROWSE     SHADOWS     |
| Output IFF24...    Frames ___ to ___            REFLECT     |
| XRes ____ YRes ____ Aspect ____ SCREENMODE      TRANSMIT    |
| Octree Depth ____ Resolve Depth ____            ILLUMINATE  |
| Objects/Cube ____ Max AA Depth  ____        OVERRIDE SCRIPT |
|                                                             |
|    GO!                                          QUIT        |
+-------------------------------------------------------------+
```

```
Underscores (____) indicate a text entry gadget, ALL CAPS
indicates a button gadget.

The gadgets mean:

Filename       The filename of the script to read.  Hit
               browse to get an ASL file requester (if
               your version of AmigaDOS supports ASL
               file requesters).

Outfile        The filename of the output file.  Hit
               browse to get an ASL file requester (if
               your version of AmigaDOS supports ASL
               file requesters).

Output         Selects between 24 bit IFF files, QRT
               output files, or MC's own raw 24 output
               format.

Frames         Indicates starting and ending framenumbers.
               For use with animated scripts.

XRes, YRes,    Indicates the properties of the rendered
Aspect         image.  See
                 Camera
                 Screen Mode   Allows the use of the ASL screen mode requester
               to select the XRes, YRes, and Aspect values.
               Available only if you have the proper verison
```

```
                  of the ASL libraries.

Octree Depth, Fine tunes the
                Octree
                        .
Objects/Cube

Resolve Depth Sets the resolve depth, which is the maximum
              number of reflect/transmit generations a
              primary ray may generate.

Max AA Depth  See
                Anti-Aliasing
                Soothing,     Perform the same functions as the
                global flags
                        .
Shadows,
Reflect,
Transmit,
Illuminate

Override Script  When selected, causes options window values
                 to override any values found in the input
                 script file.

Go!              Start tracing!

Quit             Exit without doing anything.
```

1.202 RGB

RGB stands for red, green, and blue. This is the color model used by
Magic Camera (and most other programs which use color). In RGB various amounts
of red, green, and blue light are mixed to form different shades of color.

1.203 IFF Files

IFF stand for Interchange File Format, and is the file format used by
almost all Amiga programs to store bitmapped image data. IFF bitmaps come in
several flavors, including 24 bit, HAM, and color table (32 color, 16 color,
etc.) images. Magic Camera currently understands the following type of IFF
images:

```
    24 bit
    HAM8 (AGA Mode)
    HAM
    Halfbright
    256 color
    128 color
    64 color
```

```
32 color
16 color
8 color
4 color
2 color
```

   Magic camera doesn't understand the special effects, such as masking planes, etc., used in some IFF files.

1.204 misc_raw24

   Magic Camera can read raw 24 bit bitmaps stored in a special file format. If you have to write your own program to convert bitmaps for use by Magic Camera, it will be easier to convert the bitmaps into this format than to 24 bit IFF.

   The Raw24 format used by Magic Camera is:

1. The first 4 bytes are an unsigned long specifying the X dimension of the image (pixels per row). Bytes are stored in MSB first (Motorola) order.

2. The second 4 bytes are an unsigned long specifying the Y dimension of the image (number of rows). Bytes are stored in MSB first (Motorola) order.

3. The remaining data consists of triplets of unsigned bytes which contain the RGB data. These triplets are written by rows, so that all pixels of the first row are written (in increasing X order) before the second row is written. Rows are written in increasing Y order. The first pixel written (0, 0) is the upper left hand corner of the image. The following pseudo-code applies:

```
for y = 0 to num_rows
    for x = 0 to pix_per_row
        write red
        write green
        write blue
    next x
next y
```