The Materials and Geometry Format

Greg Ward
Lawrence Berkeley Laboratory

1. Introduction

The Materials and Geometry Format (referred to henceforth as MGF) is a description language for 3-dimensional environments expressly suited to visible light simulation and rendering. The materials are physically-based and rely on standard and well-accepted definitions of color, reflectance and transmittance for good accuracy and reproducibility. The geometry is based on boundary representation using simple geometric primitives such as polygons, spheres and cones. The file format itself is terse but human-readable ASCII text.

1.1. What makes MGF special?

There are three principal reasons to use MGF as an input language for lighting simulation and physically-based rendering:

1. It's the only existing format that describes materials physically.
2. It is endorsed by the Illuminating Engineering Society of North America (IESNA) as part of their LM-63-1995 standard for luminaire data.
3. It's easy and fun to support since it comes with a standard parser and sample scenes and objects at the web site, "http://radsite.lbl.gov/mgf/HOME.html".

The standard parser provides both immediate and long-term benefits, since it presents a programming interface that is more stable even than the language itself. Unlike AutoCAD DXF and other de facto standards, a change to the language will not break existing programs. This is because the parser gives the calling software only those entities it can handle. If the translator understands only polygons, it will be given only polygons. If a new geometric primitive is included in a later version of the standard, the new parser that comes with it will still be able to express this entity as polygons. Thus, the urgency of modifying code to support a changing standard is removed, and long-term stability is assured.

This notion of *extensibility* is a cornerstone of the format, and it goes well beyond the extensibility of other languages because is guarantees that new versions of the standard will not break existing programs, and the new information will be used as much as possible. Other languages either require that all translators stay up to date with the latest standard, or allow forward compatibility by simply *ignoring* new entities. In MGF, if NURBS are added at some point and the translator or loader does not handle them directly, the new version of the parser will automatically convert them to smoothed polygons without changing a single line of the calling program. It is merely necessary to link to the new library, and all the new entities are supported†.

†If an old version of the parser encounters new entities it does not recognize, the default action is to ignore them, printing a warning message. This may be overridden to support custom entities, but such practice is discouraged because it weakens the standard.

1.2. What does MGF look like?

MGF has a simple entity-per-line structure, with a similar appearance to Wavefront's .OBJ format. Each entity is specified by a short keyword, and arguments are separated by white space (tabs and/or spaces). A newline may be escaped with a backslash ('\'), in which case it counts as a space. Lines and continued lines may have up to 4096 characters, including newlines, tabs and spaces. A comment is an ignored entity whose keyword is the pound sign ('#').

Here is an MGF file that describes a simple two-drawer file cabinet:

```
# Conversion from inches to meters
xf -s .0254
# Surface material
m burgundy_formica =
	c
		cxy .362 .283
	rd .0402
	c
	rs .0284 .05
	sides 1
# Cabinet vertices
v fc.xy =
	p .05 0 0
v fc.xY =
	p .05 18 0
v fc.XY =
	p 35.95 18 0
v fc.Xy =
	p 35.95 0 0
# Cabinet
prism fc.xy fc.xY fc.XY fc.Xy 24
# Drawer vertices
v fcd.Xz =
	p 34 0 0
v fcd.XZ =
	p 34 0 10
v fcd.xZ =
	p 0 0 10
v fcd.xz =
	p 0 0 0
# Two drawers
o drawer
	xf -t 1 18.1 2 -a 2 -t 0 0 11
	prism fcd.xz fcd.Xz fcd.XZ fcd.xZ .9
	xf
o
# End of units conversion
xf
```

1.3. MGF's place in the world of standards

MGF was developed initially to support detailed geometric description of light fixtures for the IESNA luminaire data standard, publication LM-63†. Existing standards for geometric

†To obtain the latest version of this standard, write to: Illuminating Engineering Society of North America, 345 East 47th St., New York, NY 10017.

description were either too cumbersome (e.g. *Radiance)* or did not include physical materials (e.g. IGES). It was noted early on that a standard able to fully describe luminaires would necessarily be capable of describing other objects as well; indeed whole environments could be defined this way. Since the descriptions would be physical, they could serve as input to both lighting simulation and rendering software. A standard language for describing the appearance of physical objects has been lacking for some time, and current efforts in this direction (i.e. STEP) seem several years away from fruition. (There are other languages for describing realistic scenes that deserve mention here, such as VRML and the Manchester Scene Description Language, but none give specific attention to physical material properties and are thus unsuitable for lighting simulation.)

In short, we saw this as an opportunity to offer the lighting and rendering community a simple and easy-to-support standard for describing environments in a physically valid way. Our hope is that this will promote sharing color, material and object libraries as well as complete scene descriptions. Sharing libraries is of obvious benefit to users and software developers alike. Sharing scenes should also permit comparisons between rendering systems and intervalidation of lighting calculations. As anyone who works in this field knows, modeling is the most difficult step in creating any simulation or rendering, and there is no excuse for this data being held prisoner by a proprietary data format.

2. MGF Basics

The default coordinate system in MGF is right-handed with distances given in meters, though this can be effectively changed by specifying a global transformation. The transformation context is affected by the xf entity, and the whole of MGF can be understood in terms of entities and contexts.

2.1. Entities and Contexts

An *entity* in MGF is any non-blank line, which must be one of a finite set of command keywords followed by zero or more arguments. (As mentioned previously, an entity may continue over multiple lines by escaping the newline with a backslash.) Table 1 gives a list of entities and their expected arguments. Section 3 gives more detailed information on each entity.

A *context* describes the current state of the interpreter, and affects or is affected by certain entities as they are read in. MGF contexts can be divided into two types, *hierarchical contexts* and *named contexts* .

Hierarchical contexts are manipulated by a single entity and have an associated "stack" onto which new contexts are "pushed" using the entity. The last context may be "popped" by giving the entity again with no arguments. The two hierarchical contexts in MGF are the current transformation, manipulated with the xf entity, and the current object, manipulated with the o entity.

Named contexts in contrast hold sets of values that are swapped in and out one at a time. There are three named contexts in MGF, the current material, the current color and the current vertex. Each one may be associated with an identifier (any non-white sequence of printing ASCII characters beginning with a letter), and one of each is in effect at any given time. Initially, these contexts are unnamed, and invoking an unnamed context always returns to the original (default) values. (See Table 2 for a list of contexts, their related entities and defaults.)

It is easiest to think of a context as a "scratch space" where values are written by some entities and read by others. Naming a context allows us to reestablish the same scratch space later, usually for reference but it can be altered as well. Let us say we wanted to create a smooth blue plastic material with a diffuse reflectance of 20% and a specular reflectance of 4%:

Keyword	Arguments	Interpretation
#	[anything ...]	a comment
o	[name]	begin/end object context
xf	[xform]	begin/end transformation context
i	pathname [xform]	include file (with transformation)
ies	pathname [-m f][xform]	include IES luminaire (with transformation)
c	[id [= [template]]]	get/set color context
cxy	x y	set CIE (x,y) chromaticity for current color
cspec	l_min l_max v1 v2 ...	set relative spectrum for current color
cct	temperature	set spectrum based on black body temperature
cmix	w1 c1 w2 c2 ...	mix named colors to make current color
m	[id [= [template]]]	get/set material context
sides	{1|2}	set number of sides for current material
rd	rho_d	set diffuse reflectance for current material
td	tau_d	set diffuse transmittance for current material
ed	epsilon_d	set diffuse emittance for current material
rs	rho_s alpha_r	set specular reflectance for current material
ts	tau_s alpha_t	set specular transmittance for current material
ir	n_real n_imag	set index of refraction for current material
v	[id [= [template]]]	get/set vertex context
p	x y z	set point position for current vertex
n	dx dy dz	set surface normal for current vertex
f	v1 v2 v3 ...	polygon using current material, spec. vertices
fh	v1 v2 v3 - ...	face with explicit holes
sph	vc radius	sphere
cyl	v1 radius v2	truncated right cylinder (open-ended)
cone	v1 rad1 v2 rad2	truncated right cone (open-ended)
prism	v1 v2 v3 ... length	truncated right prism (closed solid)
ring	vc rmin rmax	circular ring with inner and outer radii
torus	vc rmin rmax	circular torus with inner and outer radii

Table 1. MGF entities and their arguments. Arguments in brackets are optional. Arguments in curly braces mean one of the given choices must appear. Ellipsis (...) mean that any number of arguments may be given.

Context	Cntl. Entity	Default Value	Field Entities	Affects
Object	o	-	-	-
Transform	xf	-	-	f, fh, sph, cyl, cone, ring, torus, prism
Material	m	2-sided black	sides, rd, td, ed, rs, ts, ir	f, fh, sph, cyl, cone, ring, torus, prism
Color	c	neutral grey	cxy, cspec, cct, cmix	rd, td, ed, rs, ts
Vertex	v	(0,0,0), no normal	p, n	f, fh, sph, cyl, cone, ring, torus, prism

Table 2. MGF contexts and their related entities and default values.

```
# Establish a new material context called "blue_plastic"
m blue_plastic =
        # Reestablish a previous color context called "blue"
        c blue
```

```
        rd .20
        # Get the unnamed color context (always starts out grey)
        c
        # Set the specular reflectance, which is uncolored
        rs .04 0
    # We're done, the current material context is now "blue_plastic"
```

Note that the above assumes that we have previously defined a color context named "blue". If we forgot to do that, the above description would generate an "undefined" error. The color context affects the material context indirectly because it is read by the specular and diffuse reflectance entities, which are in turn written to the current material. It is not necessary to indent the entities that affect the material definition, but it improves readability. Note also that there is no explicit end to the material definition. As long as a context remains in effect, its contents may be altered by its field entities. This will not affect previous uses of the context, however. For example, a surface entity following the above definition will have the specified color and reflectance, and later changes to the material "blue_plastic" will have no effect on it.

Each of the three named contexts has an associated entity that controls it. The material context is controlled by the m entity, the color context is controlled by the c entity, and the vertex context is controlled by the v entity. There are exactly four forms for each entity. The first form is the keyword by itself, which establishes an unnamed context with predetermined default values. This is a useful way to set values without worrying about saving them for recall later. The second form is to give the keyword with a previously defined name. This reestablishes a prior context for reuse. The third form is to give the keyword with a name followed by an equals sign. (There must be a space between the name and the equals sign, since it is a separate argument.) This establishes a new context and assigns it the same default values as the unnamed context. The fourth and final form gives the keyword followed by a name then an equals then the name of a previous context definition. This establishes a new context for the first name, assigning the values from the second named context rather than the usual defaults. This is a convenient way create an alias or to modify a context under a new name (i.e. "save as").

2.2. Hierarchical Contexts and Transformations

As mentioned in the last subsection, there are two hierarchical contexts in MGF, the current object and the current transformation. We will start by discussing the current object, since it is the simpler of the two.

2.2.1. Objects

There is no particular need in lighting simulation or rendering to name objects, but it may help the user to know what object a particular surface is associated with. The o entity provides a convenient mechanism for associating names with surfaces. The basic use of this entity is as follows:

```
o object_name
        [object entities...]
        o subobject_name
                [subobject entities...]
        o
        [more object entities and subobjects...]
o
```

The o keyword by itself marks the end of an object context. Any number of hierarchical context levels are supported, and there are no rules governing the choice of object names except that they begin with a letter and be made up of printing, non-white ASCII characters. Indentation is not necessary of course, but it does improve readability.

2.2.2. Transformations

MGF supports only rigid-body (i.e. non-distorting) transformations with uniform scaling. Unlike the other contexts, transformations have no associated name, only arguments. Thus, there is no way to reestablish a previous transformation other than to give the same arguments over again. Since the arguments are concise and self-explanatory, this was thought sufficient. The following transformation flags and parameters are defined:

-t dx dy dz	translate objects along the given vector
-rx degrees	rotate objects about the X-axis
-ry degrees	rotate objects about the Y-axis
-rz degrees	rotate objects about the Z-axis
-s scalefactor	scale objects by the given factor
-mx	mirror objects about the Y-Z plane
-my	mirror objects about the X-Z plane
-mz	mirror objects about the X-Y plane
-i N	repeat the following arguments N times
-a N	make an array of N geometric instances

Transform arguments have a cumulative effect. That is, a rotation about X of 20 degrees followed by a rotation about X of -50 degrees results in a total rotation of -30 degrees. However, if the two rotations are separated by some translation vector, the cumulative effect is quite different. It is best to think of each argument as acting on the included geometric objects, and each subsequent transformation argument affects the objects relative to their new position/orientation.

For example, rotating an object about its center is most easily done by translating the object back to the origin, applying the desired rotation, and translating it again back to its original position, like so:

```
# rotate an included object 20 degrees clockwise looking down
# an axis parallel to Y and passing through the point (15,0,-35)
xf -t -15 0 35 -ry -20 -t 15 0 -35
i object.mgf
xf
```

Note that the include entity, i, permits a transformation to be given with it, so the above could have been written more compactly as:

```
i object.mgf -t -15 0 35 -ry -20 -t 15 0 -35
```

Rotations are given in degrees counter-clockwise about a principal axis. That is, with the thumb of the right hand pointing in the direction of the axis, rotation follows the curl of the fingers.

The transform entity itself is cumulative, but in the reverse order to its arguments. That is, later transformations (i.e. enclosed transformations) are prepended to existing (i.e. enclosing) ones. A transform command with no arguments is used to return to the previous condition. It is necessary that transforms and their end statements ("xf" by itself) be balanced in a file, so that later or enclosing files are not affected.

Transformations apply only to geometric types, e.g. polygons, spheres, etc. Vertices and the components that go into geometry are not directly affected. This is to avoid confusion and the inadvertent multiple application of a given transformation. For example:

```
xf -t 5 0 0
v v1 =
        p 0 10 0
        n 0 0 1
xf -rx 180
# Transform now in effect is "-rx 180 -t 5 0 0"
ring v1 0 2
xf
xf
```

The final ring center is (5,-10,0) -- note that the vertex itself is not affected by the transformation, only the geometry that calls on it. The normal orientation is (0,0,-1) due to the rotation about X, which also reversed the sign of the central Y coordinate.

2.2.3. Arrays

The -a N transform specification causes the following transform arguments to be repeated along with the contents of the included objects N times. The first instance of the geometry will be in its initial location; the second instance will be repositioned according to the named transformation; the third instance will be repositioned by applying this transformation twice, and so on up to N-1 applications.

Multi-dimensional arrays may be specified with a single include entity by giving multiple array commands separated by their corresponding transforms. A final transformation may be given by preceding it with a -i 1 specification. In other words, the scope of an array command continues until the next -i or -a option.

The following MGF description places 60 spheres at a one unit spacing in a 3x4x5 array, then moves the whole thing to an origin of (15,30,45):

```
v v0 =
        p 0 0 0
xf -a 3 -t 1 0 0 -a 4 -t 0 1 0 -a 5 -t 0 0 1 -i 1 -t 15 30 45
sph v0 0.1
xf
```

Note the "-i 1" in the specification, which marks the end of the third array arguments before the final translation.

2.3. Detailed MGF Example

The following example of a simple room with a single door and six file cabinets shows MGF in action, with copious comments to help explain what’s going on.

```
# "ceiling_tile" is a diffuse white surface with 75% reflectance:
# Create new named material context and clear it
m ceiling_tile =
	# Specify one-sided material so we can see through from above
	sides 1
	# Set neutral color
	c
	# Set diffuse reflectance
	rd .75
# "stainless_steel" is a mostly specular surface with 70% reflectance:
m stainless_steel =
	sides 1
	c
	# Set specular reflectance to 50%, .08 roughness
	rs .5 .08
	# Other 20% reflectance is diffuse
	rd .2

# The following materials were measured with a spectrophotometer:
m beige_paint =
	sides 1
	# Set diffuse spectral reflectance
	c
		# Spectrum measured in 10 nm increments from 400 to 700 nm
		cspec 400 700 35.29 44.87 47.25 47.03 46.87 47.00 47.09 \
			47.15 46.80 46.17 46.26 48.74 51.08 51.31 51.10 \
			51.11 50.52 50.36 51.72 53.61 53.95 52.08 49.49 \
			48.30 48.75 49.99 51.35 52.75 54.44 56.34 58.00
	rd 0.5078
	# Neutral (grey) specular component
	c
	rs 0.0099 0.08000
m mottled_carpet =
	sides 1
	c
		cspec 400 700 11.23 11.28 11.39 11.49 11.61 11.73 11.88 \
			12.02 12.12 12.19 12.30 12.37 12.37 12.36 12.34 \
			12.28 12.22 12.29 12.45 12.59 12.70 12.77 12.82 \
			12.88 12.98 13.24 13.67 14.31 15.55 17.46 19.75
	rd 0.1245
m reddish_cloth =
	# 2-sided so we can observe it from behind
	sides 2
	c
		cspec 400 700 28.62 27.96 27.86 28.28 29.28 30.49 31.61 \
			32.27 32.26 31.83 31.13 30.07 29.14 29.03 29.69 \
			30.79 32.30 33.90 34.56 34.32 33.85 33.51 33.30 \
			33.43 34.06 35.26 37.04 39.41 42.55 46.46 51.00
	rd 0.3210
m burgundy_formica =
	sides 1
	c
		cspec 400 700  3.86  3.74  3.63  3.51  3.34  3.21  3.14 \
			3.09  3.08  3.14  3.13  2.91  2.72  2.74  2.72 \
```

```
                2.60  2.68  3.40  4.76  6.05  6.65  6.75  6.68 \
                6.63  6.56  6.51  6.46  6.41  6.36  6.34  6.34
        rd 0.0402
        c
        rs 0.0284 0.05000
m speckled_grey_formica =
        sides 1
        c
                cspec 400 700 30.95 44.77 51.15 52.60 53.00 53.37 53.68 \
                        54.07 54.33 54.57 54.85 55.20 55.42 55.51 55.54 \
                        55.46 55.33 55.30 55.52 55.81 55.91 55.92 56.00 \
                        56.22 56.45 56.66 56.72 56.58 56.44 56.39 56.39
        rd 0.5550
        c
        rs 0.0149 0.15000

# 40' x 22' x 9' office space with no windows and one door

# All measurements are in inches, so enclose with a metric conversion:
xf -s .0254

# The room corner vertices:
v rc.xyz =
        p 0 0 0
v rc.Xyz =
        p 480 0 0
v rc.xYz =
        p 0 264 0
v rc.xyZ =
        p 0 0 108
v rc.XYz =
        p 480 264 0
v rc.xYZ =
        p 0 264 108
v rc.XyZ =
        p 480 0 108
v rc.XYZ =
        p 480 264 108

# The floor:
# Push object name
o floor
        # Get previously defined carpet material
        m mottled_carpet
        # Polygonal face using defined vertices
        f rc.xyz rc.Xyz rc.XYz rc.xYz
# Pop object name
o

# The ceiling:
o ceiling
        m ceiling_tile
        f rc.xyZ rc.xYZ rc.XYZ rc.XyZ
o
```

```
# The door outline vertices:          v do.xz =                p 216 0 0          v do.Xz =
        p 264 0 0              v do.xZ =                p 216 0 84              v do.XZ =
        p 264 0 84

# The walls:                o wall                    m beige_paint                    o x
                f rc.xyz rc.xYz rc.xYZ rc.xyZ                    o                    o X
                f rc.Xyz rc.XyZ rc.XYZ rc.XYz                    o                    o y
                f rc.xyz rc.xyZ rc.XyZ rc.Xyz do.Xz do.XZ do.xZ do.xz                    o
        o Y                    f rc.xYz rc.XYz rc.XYZ rc.xYZ                o          o

# The door and jam vertices:                v djo.xz =                p 216 .5 0
v djo.xZ =                p 216 .5 84          v djo.XZ =                p 264 .5 84
v djo.Xz =                p 264 .5 0           v dji.Xz =                p 262 .5 0
v dji.XZ =                p 262 .5 82          v dji.xZ =                p 218 .5 82
v dji.xz =                p 218 .5 0           v door.xz =               p 218 0 0
v door.xZ =               p 218 0 82           v door.XZ =               p 262 0 82
v door.Xz =               p 262 0 0

# The door, jam and knob                o door                    m burgundy_formica
        f door.xz door.xZ door.XZ door.Xz                                        o jam
                m beige_paint
                f djo.xz djo.xZ djo.XZ djo.Xz dji.Xz dji.XZ dji.xZ dji.xz
                f djo.xz do.xz do.xZ djo.xZ
                f djo.xZ do.xZ do.XZ djo.XZ
                f djo.Xz djo.XZ do.XZ do.Xz
                f dji.xz dji.xZ door.xZ door.xz
                f dji.xZ dji.XZ door.XZ door.xZ
                f dji.Xz door.Xz door.XZ dji.XZ                o                    o knob
                m stainless_steel
                # Define vertices needed for curved geometry                    v kb1 =
                        p 257 0 36                                              v kb2 =
                        p 257 .25 36                                            n 0 1 0
                v kb3 =                                                         p 257 2 36
                # 1" diameter cylindrical base from kb1 to kb2
                cyl kb1 1 kb2                                   # Ring at base of knob stem
                ring kb2 .4 1                                                   # Knob stem
                cyl kb2 .4 kb3                                                  # Spherical knob
                sph kb3 .85                o          o

# Six file cabinets (36" wide each)
# ("filecab.inc" was given as an earlier example in Section 1.2)                o filecab.x
        # include a file as an array of three 36" apart                i filecab.inc -t -36 0 0 -
rz -90 -t 1 54 0 -a 3 -t 0 36 0                    o                    o filecab.X
        # the other three cabinets                i filecab.inc -rz 90 -t 479 54 0 -a 3 -t 0 36 0
o

# End of transform from inches to meters:          xf

# The 10 recessed fluorescent ceiling fixtures                    ies hlrs2gna.ies -
t 1.2192 2.1336 2.74 -a 5 -t 2.4384 0 0 -a 2 -t 0 2.4384 0
```

3. MGF Entity Reference

There are currently 28 entities in the MGF specification. For ease of reference we have broken these into five categories:

1. General

#	[anything ...]	a comment
o	[name]	begin/end object context
xf	[xform]	begin/end transformation context
i	pathname [xform]	include file (with transformation)
ies	pathname [-m f][xform]	include IES luminaire (with transformation)

2. Color

c	[id [= [template]]]	get/set color context
cxy	x y	set CIE (x,y) chromaticity for current color
cspec	l_min l_max v1 v2 ...	set relative spectrum for current color
cct	temperature	set spectrum based on black body temperature
cmix	w1 c1 w2 c2 ...	mix named colors to make current color

3. Material

m	[id [= [template]]]	get/set material context
sides	{1|2}	set number of sides for current material
rd	rho_d	set diffuse reflectance for current material
td	tau_d	set diffuse transmittance for current material
ed	epsilon_d	set diffuse emittance for current material
rs	rho_s alpha_r	set specular reflectance for current material
ts	tau_s alpha_t	set specular transmittance for current material
ir	n_real n_imag	set index of refraction for current material

4. Vertex

v	[id [= [template]]]	get/set vertex context
p	x y z	set point position for current vertex
n	dx dy dz	set surface normal for current vertex

5. Geometry

f	v1 v2 v3 ...	polygon using current material, spec. vertices
fh	v1 v2 v3 - ...	face with explicit holes
sph	vc radius	sphere
cyl	v1 radius v2	truncated right cylinder (open-ended)
cone	v1 rad1 v2 rad2	truncated right cone (open-ended)
prism	v1 v2 v3 ... length	truncated right prism (closed solid)
ring	vc rmin rmax	circular ring with inner and outer radii
torus	vc rmin rmax	circular torus with inner and outer radii

NAME

- a comment

SYNOPSIS

[*anything*]

DESCRIPTION

A comment is a bit of text explanation. Since it is an entity like any other (except that it has no effect), there must be at least one space between the keyword (which is a pound sign) and the "arguments," and the end of line may be escaped as usual with the backslash character ('\').

A comment may actually be used to hold auxiliary information such as view parameters, which may be interpreted by some destination program. Care should be taken under such circumstances that the user does not inadvertently mung or mimic this information in other comments, and it is therefore advisable to use an additional set of identifying characters to distinguish such data.

EXAMPLE

```
# The following include file is in inches, so convert to meters
i cubgeom.inc -s .0254
# Stuff we don't want to see at the moment:
# i person.mgf -t 3 2 0
# ies hlrs3gna.ies -rz 90 -t 1.524 1.8288 2.74 \
        -a 6 -t 1.8288 0 0 -a 2 -t 0 3.048 0
```

NAME

o - begin or end object context

SYNOPSIS

o [*name*]

DESCRIPTION

If *name* is given, we push a new object context onto the stack, which is to say that we begin a new subobject by this name†. If the o keyword is given by itself, then we pop the last object context off the stack, which means that we leave the current subobject.

All geometry between the start of an object context and its matching end statement is associated with the given name. This may be used in modeling software to help identify objects and subobjects, or it may be ignored altogether.

Object begin and end statements should be balanced in a file, and care should be taken not to overlap transform (xf) contexts with object contexts, especially when arrays are involved. This is because the standard parser will assign object contexts to instanced geometry, which can get confused with other object contexts if a clear enclosure is not maintained.

EXAMPLE

```
o body
        o torso
                i torso.mgf
        o
        o arm
                o left
                        i leftarm.mgf
                o
                o right
                        i leftarm.mgf -mx
                o
        o
o
```

SEE ALSO

xf

†A name is any sequence of printing, non-white ASCII characters beginning with a letter.

NAME

xf - begin or end transformation context

SYNOPSIS

xf [*transform*]

DESCRIPTION

If a set of *transform* arguments are given, we push a new transformation context onto the stack. If the xf keyword is given by itself, then we pop the last transformation context off the stack. The total transformation in effect at any given time is computed by prepending each set subcontext arguments onto those of its enclosing context. This and other details about transformation specifications are explained in some detail in section 2.2.2.

The following transformation flags and parameters are defined:

-t dx dy dz	translate objects along the given vector
-rx degrees	rotate objects about the X-axis
-ry degrees	rotate objects about the Y-axis
-rz degrees	rotate objects about the Z-axis
-s scalefactor	scale objects by the given factor
-mx	mirror objects about the Y-Z plane
-my	mirror objects about the X-Z plane
-mz	mirror objects about the X-Y plane
-i N	repeat the following arguments N times
-a N	make an array of N geometric instances

EXAMPLE

```
# Create 3x5 array of evenly-spaced spheres (grid size = 3)
v vc =
        p 0 0 0
xf -t 1 1 10 -a 3 -t 3 0 0 -a 5 -t 0 3 0
        sph vc .5
xf
```

SEE ALSO

i, ies, o

NAME

i - include MGF data file

SYNOPSIS

i *pathname* [*transform*]

DESCRIPTION

Include the information contained in the file *pathname.* If a *transform* specification is given, then it will be applied as though the include statement were enclosed by beginning and ending xf entities with this transformation.

The *pathname* will be interpreted relative to the enclosing MGF file. That is, if the file containing the include statement is in some parent or subdirectory, then the given pathname is appended to this directory. It is illegal to specify a *pathname* relative to the root directory, and the MGF standard requires that all filenames adhere to the ISO-9660 8.3 name format for maximum portability between systems. The directory separator is defined to be slash ('/'), and drive specifications (such as "c:") are not allowed. All pathnames should be given in lower case, and will be converted to upper case on systems that require it. (That way, there are no accidental name collisions.)

The suggested suffix for MGF-adherent files is ".mgf". Files that are not in metric units but are in MGF may be given any suffix, but we suggest using ".inc" as a convention.

EXAMPLE

```
# Define vertices for 62x30" partition
i pv62x30.inc
# Insert 2 62x30" partitions
o cpart1
        i partn.inc -t 75 130.5 0
o
o cpart3
        i partn.inc -t 186 130.5 0
o
# Define vertices for 62x36" partition
i pv62x36.inc
# Insert 62x36" partition
o cpart2
        i partn.inc -t 105 130.5 0
o
```

SEE ALSO

ies, o, xf

NAME

ies - include IESNA luminaire file

SYNOPSIS

ies *pathname* [–**m** *multiplier*] [*transform*]

DESCRIPTION

Load the IES standard luminaire information contained in the file *pathname.* If a *multiplier* is given, all candela values will be multiplied by this factor. (This option must appear first if present.) If a *transform* specification is given, then it will be applied as though the statement were enclosed by beginning and ending xf entities with this transformation.

The *pathname* will be interpreted relative to the enclosing MGF file, and all restrictions discussed under the i entity also apply to the IES file name. The suggested suffix is ".ies", but this has not been followed consistently by lighting manufacturers.

EXAMPLE

```
# Insert 10 2x4' fluorescent troffers in two groups
ies cf9pr240.ies -t 3.6576 2.1336 2.74 -a 3 -t 2.4384 0 0 -a 2 -t 0 2.4384 0
ies cf9pr240.ies -rz 90 -t 1.2192 1.8288 2.74 \
        -a 2 -t 9.7536 0 0 -a 2 -t 0 3.048 0
```

SEE ALSO

i, o, xf

NAME

c - get or set the current color context

SYNOPSIS

c [*id* [**=** [*template*]]]

DESCRIPTION

If the c keyword is given by itself, then it establishes the unnamed color context, which is neutral (i.e. equal-energy) grey. This context may be modified, but the changes will not be saved.

If the keyword is followed by an identifier *id,* then it reestablishes a previous context. If the specified context was never defined, an error will result.

If the entity is given with an identifier followed by an equals sign ('='), then a new context is established, and cleared to the default neutral grey. (Note that the equals sign must be separated from other arguments by white space to be properly recognized.) If the equals sign is followed by a second identifier *template,* then this previously defined color will be used as a source of default values rather than grey. This is most useful for establishing a color alias.

EXAMPLE

```
# Define the color "red32"
c red32 =
        cxy .42 .15
# Make "cabinet_color" an alias for "red32"
c cabinet_color = red32

# Later in another part of the description...

# Get our cabinet color
c cabinet_color
# Get the geometry
i cabgeom.mgf
```

SEE ALSO

cct, cmix, cspec, cxy, m

NAME

cxy - set the CIE (x,y) chromaticity for the current color

SYNOPSIS

cxy *x y*

DESCRIPTION

This entity sets the current color using (x,y) chromaticity coordinates for the 1931 CIE standard 2 degree observer. Legal values for *x* and *y* are greater than zero and sum to less than one, and more specifically they must fit within the curve of the visible spectrum. The *x* coordinate roughly corresponds to the red part of the spectrum and the *y* coordinate corresponds to the green. The CIE z coordinate is implicit, since it is equal to (1-x-y).

All colors in MGF are absolute, thus colorimeter measurements should be conducted the same for surfaces as for light sources. Applying a standard illuminant calculation is redundant and introduces inaccuracies, and should therefore be avoided if possible.

Conversion between CIE colors and those more commonly used in computer graphics are described in the application notes section 6.1.1.

EXAMPLE

```
# Set unnamed color context
c
# Set CIE chromaticity to a bluish hue
cxy .15 .2
# Apply color to diffuse reflectance of 15%
rd .15
```

SEE ALSO

c, cct, cmix, cspec

NAME

cspec - set the relative spectrum for the current color

SYNOPSIS

cspec *l_min l_max o1 o2 ... oN*

DESCRIPTION

Assign a relative spectrum measured between *l_min* and *l_max* nanometers at evenly spaced intervals. The first value, *o1* corresponds to the measurement at *l_min*, and the last value, *oN* corresponds to the measurement at *l_max*. Values in between are separated by *(l_max-l_min)/(N-1)* nanometers. All values should be non-negative unless defining a component for complementary color mixing, and the spectrum outside of the specified range is assumed to be zero. (The visible range is 380 to 780 nm.) The actual units and scale of the measurements do not matter, since the total will be normalized according to whatever the color is modifying (e.g. photometric reflectance or emittance).

EXAMPLE

```
# Color measured at 10 nm increments from 400 to 700
m reddish_cloth =
        c
                cspec 400 700 28.62 27.96 27.86 28.28 29.28 30.49 31.61 \
                        32.27 32.26 31.83 31.13 30.07 29.14 29.03 29.69 \
                        30.79 32.30 33.90 34.56 34.32 33.85 33.51 33.30 \
                        33.43 34.06 35.26 37.04 39.41 42.55 46.46 51.00
        rd 0.3210
```

SEE ALSO

c, cct, cmix, cxy

NAME

cct - set the current color to a black body spectrum

SYNOPSIS

cct *temperature*

DESCRIPTION

The cct entity sets the current color to the spectrum of an ideal black body radiating at *temperature* degrees Kelvin. This is often the most convenient way to set the color of an incandescent light source, but it is not recommended for fluorescent lamps or other materials that do not fit a black body spectrum.

EXAMPLE

```
# Define an incandescent source material at 3000 degrees K
m incand3000k =
        c
                cct 3000
        ed 1500
```

SEE ALSO

c, cmix, cspec, cxy

NAME

cmix - mix two or more named colors to make the current color

SYNOPSIS

cmix *w1 c1 w2 c2 ...*

DESCRIPTION

The cmix entity sums together two or more named colors using specified weighting coefficients, which correspond to the relative photometric brightness of each. As in all color specifications, the result is normalized so the absolute scale of the weights does not matter, only their relative values.

If any of the colors is a spectral quantity (i.e. from a cspec or cct entity), then all the colors are first converted to spectral quantities. This is done with an approximation for CIE (x,y) chromaticities, which may be problematic depending on their values. In general, it is safest to add together colors that are either all spectral quantities or all CIE quantities.

EXAMPLE

```
# Define RGB primaries for a standard color monitor
c R =
        cxy 0.640 0.330
c G =
        cxy 0.290 0.600
c B =
        cxy 0.150 0.060
# Mix them together in appropriate amounts for white
c white =
        cmix 0.265 R 0.670 G 0.065 B
```

SEE ALSO

c, cct, cspec, cxy

NAME

m - get or set the current material context

SYNOPSIS

m [*id* [= [*template*]]]

DESCRIPTION

If the m keyword is given by itself, then it establishes the unnamed material context, which is a perfect two-sided black absorber. This context may be modified, but the changes will not be saved.

If the keyword is followed by an identifier *id*, then it reestablishes a previous context. If the specified context was never defined, an error will result.

If the entity is given with an identifier followed by an equals sign ('='), then a new context is established, and cleared to the default material. (Note that the equals sign must be separated from other arguments by white space to be properly recognized.) If the equals sign is followed by a second identifier *template*, then this previously defined material will be used as a source of default values instead. This may be used to establish a material alias, or to modify an existing material and give it a new name.

The sum of the diffuse and specular reflectances and transmittances must not be greater than one (with no negative values, obviously). These values are assumed to be measured at normal incidence. If an index of refraction is given, this may modify the balance between diffuse and specular reflectance at other incident angles. If the material is one-sided (see sides entity), then it may be a dielectric interface. In this case, the specular transmittance given is that which would be measured at normal incidence for a pane of the material 5 mm thick. This is important for figuring the actual transmittance for non-planar geometries assuming a uniformly absorbing medium. (Diffuse transmittance will not be affected by thickness.) If the index of refraction has an imaginary part, then the surface is a metal and this implies other properties as well. The default index of refraction is that of a vacuum, i.e. (1,0).

EXAMPLE

```
# Define a blue enamel paint
m blue_enamel =
        c
                cxy 0.2771 0.2975
        rd 0.5011
        c
        rs 0.0100 0.0350
# Assign blue_enamel to be the color of the south wall
m swall_mat = blue_enamel
# ...
# South wall face
m swall_mat
f sv1 sv2 sv3 sv4
```

SEE ALSO

ed, ir, rd, rs, sides, td, ts

NAME

sides - set the number of sides for the current material

SYNOPSIS

sides { **1** | **2** }

DESCRIPTION

The sides entity is used to set the number of sides for the current material. If a surface is two-sided, then it will appear identical when viewed from either the front or the back. If a surface is one-sided, then it appears invisible when viewed from the back side. This means that a transmitting object will affect the light coming in through the front surface and ignore the characteristics of the back surface, unless the index of refraction is set. If the index of refraction is set, then the object will act as a solid piece of dielectric material. In either case, the transmission properties of the exiting surface should be the same as the incident surface for the model to be physically valid.

The default number of sides is two.

EXAMPLE

```
# Describe a blue crystal ball
m blue_crystal =
        ir 1.650000 0
          # Solid dielectrics must use one-sided materials
        sides 1
        c
        rs 0.0602 0
        c
                cxy 0.3127 0.2881
        ts 0.6425 0
v sc =
          p 10 15 1.5
sph sc .02
```

SEE ALSO

ed, ir, m, rd, rs, td, ts

NAME

rd - set the diffuse reflectance for the current material

SYNOPSIS

rd *rho_d*

DESCRIPTION

Set the diffuse reflectance for the current material to *rho_d* using the current color to determine the spectral characteristics. This is the fraction of visible light that is reflected from a surface equally in all directions according to Lambert's law, and is often called the "Lambertian component." Photometric reflectance is measured according to v(lambda) response function of the 1931 CIE standard 2 degree observer, and assumes an equal-energy white light source. The value must be between zero and one, and may be further restricted by the luminosity of the selected color. (I.e. it is impossible to have a violet material with a photometric reflectance close to one since the eye is less sensitive in this part of the spectrum.)

The default diffuse reflectance is zero.

EXAMPLE

```
# An off-white paint with 70% reflectance
m flat_white70 =
        c
                cxy .3632 .3420
        rd .70
```

SEE ALSO

c, ed, ir, m, rs, sides, td, ts

NAME

td - set the diffuse transmittance for the current material

SYNOPSIS

td *tau_d*

DESCRIPTION

Set the diffuse transmittance for the current material to *tau_d* using the current color to determine the spectral characteristics. This is the fraction of visible light that is transmitted through a surface equally in all (transmitted) directions. Like reflectance, transmittance is measured according to the standard v(lambda) curve, and assumes an equal-energy white light source. It is probably not possible to create a material with a diffuse transmittance above 50%, since well-diffused light will be reflected as well.

The default diffuse transmittance is zero.

EXAMPLE

```
# Model a perfect spherical diffuser, i.e. light hitting either side will be scattered equally in all directions
m wonderland_diffuser =
        c
        td .5
        rd .5
```

SEE ALSO

c, ed, ir, m, rd, rs, sides, ts

NAME

ed - set the diffuse emittance for the current material

SYNOPSIS

ed *epsilon_d*

DESCRIPTION

Set the diffuse emittance for the current material to *epsilon_d* lumens per square meter using the current color to determine the spectral characteristics. Note that this is emittance rather than exitance, and therefore does not include reflected or transmitted light, which is a function of the other material settings and the illuminated environment.

The total lumen output of a convex emitting object is the radiating area of that object multiplied by its emittance. Therefore, one can compute the appropriate *epsilon_d* value for an emitter by dividing the total lumen output by the radiating area (in square meters).

The default emittance is zero.

EXAMPLE

```
# A 100-watt incandescent bulb (1600 lumens) modeled as a sphere
m
        c
                cct 3000
        ed 87712
v cent =
        p 0 0 0
sph cent .0381
```

SEE ALSO

c, ir, m, rd, rs, sides, td, ts

NAME

rs - set the specular reflectance for the current material

SYNOPSIS

rs *rho_s alpha_r*

DESCRIPTION

Set the specular reflectance for the current material to *rho_s* using the current color to determine the spectral characteristics. The surface roughness parameter is set to *alpha_r*, which is the RMS height of surface variations over the autocorrelation distance (equivalent to RMS facet slope). A roughness value of zero means a perfectly smooth surface, and values greater than 0.2 are unusual. (See application notes section 6.1.2 for a comparison between the roughness parameter and Phong specular power.)

The default specular reflectance is zero.

EXAMPLE

```
# Define a slightly rough brass metallic surface
m rough_brass =
        c
                cxy .3820 .4035
        # 30% specular, 9% diffuse
        rs .30 .08
        rd .09
```

SEE ALSO

c, ed, ir, m, rd, sides, td, ts

NAME

ts - set the specular transmittance for the current material

SYNOPSIS

ts *tau_s alpha_t*

DESCRIPTION

Set the specular transmittance for the current material to *tau_s* using the current color to determine the spectral characteristics. The effective surface roughness is set to *alpha_t.* Rays will be transmitted with the same distribution as they would have been reflected with if this roughness value were given to the rs entity.

The default specular transmittance is zero.

EXAMPLE

```
# Define a green glass material (58% transmittance)
m glass =
        sides 2
      ir 1.52 0
      c
      rs 0.0725 0
        c
              cxy .23 .38
      ts 0.5815 0
# Define an uncolored translucent plastic (40% transmittance)
m translucent =
        sides 2
        ir 1.4 0
        c
        rs .045 0
        ts .40 .05
```

SEE ALSO

c, ed, ir, m, rd, rs, sides, td

NAME

ir - set the complex index of refraction for the current material

SYNOPSIS

ir *n_real n_imag*

DESCRIPTION

Set the index of refraction for the current material to *(n_real,n_imag).* If the material is a dielectric (as opposed to metallic), then *n_imag* should be zero. For solid dielectric objects, the material should be made one-sided. If it is being used for thin objects, then a two-sided material is appropriate. (See the sides entity.)

The default index of refraction is that of a vacuum, (1,0).

EXAMPLE

```
# Define polished aluminum material
m polished_aluminum =
        # Complex index of refraction (from physics table)
        ir .770058 6.08351
        c
        rs .75 0
```

SEE ALSO

c, ed, m, rd, rs, sides, td, ts

NAME

v - get or set the current vertex context

SYNOPSIS

v [*id* [**=** [*template*]]]

DESCRIPTION

If the v keyword is given by itself, then it establishes the unnamed vertex context, which is the origin with no normal. This context may be modified, but the changes will not be saved. (The unnamed vertex is never used except as a source of default values since all geometric entities call their vertices by name.)

If the keyword is followed by an identifier *id*, then it reestablishes a previous context. If the specified context was never defined, an error will result.

If the entity is given with an identifier followed by an equals sign ('='), then a new context is established, and cleared to the default vertex (the origin). (Note that the equals sign must be separated from other arguments by white space to be properly recognized.) If the equals sign is followed by a second identifier *template*, then this previously defined vertex will be used as a source of default values instead. This may be used to establish a vertex alias, or to modify an existing vertex and give it a new name.

A non-zero vertex normal must be given for certain entities, specifically ring and torus require a normal direction. An f or fh entity will interpolate vertex normals if given, and use the polygon plane normal otherwise. See the prism entry for an explanation of how it interprets and uses vertex normals. The other entities ignore vertex normals if present.

The actual position and normal direction for a vertex is determined at the time of use by a geometric entity. Specifically, the transformation in effect at the time the vertex is defined is irrelevant. The only transformation that matters is the one that is applied to the geometry itself. This prevents double-transformation of vertices and allows one set of vertices to be used for multiple purposes, e.g. the front and back sides of a drawer.

EXAMPLE

```
# Make a capped cylinder
v end1 =
        p 0 0 0
        n 0 0 -1
v end2 =
        p 0 0 1
cyl end1 1.2 end2
# Forgot normal for end2
v end2
        n 0 0 1
ring end1 0 1.2
ring end2 0 1.2
```

SEE ALSO

cone, cyl, f, fh, n, p, prism, ring, sph, torus

NAME

p - set the point location for the current vertex

SYNOPSIS

p *px py pz*

DESCRIPTION

Set the 3-dimensional position for the current vertex to *(px,py,pz).* The actual position of the vertex will be determined by the transformation in effect at the time the vertex is applied to a geometric surface entity. The transform current when the position is set is irrelevant.

The default vertex position is the origin, (0,0,0).

EXAMPLE

```
# Make a small circle of 6 spheres
v scent =
        p 1 0 0
xf -a 6 -rz 60
        sph scent .05
xf
```

SEE ALSO

cone, cyl, f, fh, n, prism, ring, sph, torus, v

NAME

n - set the surface normal direction for the current vertex

SYNOPSIS

n *dx dy dz*

DESCRIPTION

Set the 3-dimensional surface normal for the current vertex to the normalized vector along *(dx,dy,dz)*. If this vector is zero, then the surface normal is effectively unset. The actual surface normal orientation of the vertex will be determined by the transformation in effect at the time the vertex is applied to a geometric surface entity. The current transform when the normal is set is irrelevant.

The default vertex normal is the zero vector (i.e. no normal).

EXAMPLE

```
# Make a chain of 10 interlocking doughnuts
v tcent =
        p 0 0 0
        n 0 1 0
xf -a 10 -rx 90 -t .2 0 0
        torus tcent .1 .2
xf
```

SEE ALSO

f, fh, p, prism, ring, torus, v

NAME

f - create an N-sided polygonal face

SYNOPSIS

f *v1 v2 ... vN*

DESCRIPTION

Create a polygonal face made of the current material by connecting the named vertices in order, and connecting the last vertex to the first. There must be at least three vertices, and if any vertex is undefined, an error will result.

The surface orientation is determined by the right-hand rule; when the curl of the fingers follows the given order of the vertices, the surface normal points in the thumb direction. Face vertices should be coplanar, though this is difficult to guarantee in a 3-dimensional specification.

If any vertices have associated surface normals, they will be used instead of the average plane normal, though it is safest to specify either all normals or no normals, and to stick with triangles when normals are used. Also, specified normals should point in the general direction of the surface for best results.

There is no explicit representation of holes in this entity, but see the fh entity for an alternative specification.

A hole may be represented implicitly in a face entity by connecting vertices to form "seams." For example, a wall with a window in it might look as shown in Figure 1. In many systems, the wall itself would be represented with the first list of vertices, (v1,v2,v3,v4) and the hole associated with that wall as a second set of vertices (v5,v6,v7,v8). Using the face entity, we must give the whole thing as a single polygon, connecting the vertices so as to create a "seam," as shown in Figure 2. This could be written as "f v1 v2 v3 v4 v5 v6 v7 v8 v5 v4".

It is very important that the order of the hole be opposite to the order of the outer perimeter, otherwise the polygon will be "twisted" on top of itself. Note also that the seam was traversed in both directions, once going from v4 to v5, and again returning from v5 to v4. This is a necessary condition for a proper seam.

The choice of vertices to make into a seam is somewhat arbitrary, but some rendering systems may not give sane results if you cross over a hole with part of your seam. If we had chosen to create the seam between v2 and v5 in the above example instead of v4 and v5, the seam would cross our hole and may not render correctly†.

†For systems that are sensitive to this, it is probably safest for their MGF loader/translator to re-expresses seams in terms of holes again, which can be done easily so long as vertices are shared in the fashion shown.

Replace this page with the first page from "figures.ps".

EXAMPLE

```
# Make a pyramid
v apex =
        p 1 1 1
v base0 =
        p 0 0 0
v base1 =
        p 0 2 0
v base2 =
        p 2 2 0
v base3 =
        p 2 0 0
# Bottom
f base0 base1 base2 base3
# Sides
f base0 apex base1
f base1 apex base2
f base2 apex base3
f base3 apex base0
```

SEE ALSO

cone, cyl, fh, m, prism, ring, sph, torus, v

NAME

fh - create a polygonal face with explicit holes

SYNOPSIS

fh *p1 p2 ... - h1.1 h1.2 ... - h2.1 h2.2 ...*

DESCRIPTION

Create a polygonal face with optional holes made of the current material. The first contour is the outer perimeter, with vertices given in counter-clockwise order as seen from the front side (the same as the f entity). A hole is indicated by a hyphen ('-') followed by the hole's vertices, given in clockwise order as seen from the front side. Multiple hole contours are separated by additional hyphens. There must be at least three vertices for each contour, and the last vertex is implicitly connected to the first. If any vertex is undefined, an error will result.

If any vertices have associated surface normals, they will be used instead of the average plane normal, though it is safest to specify either all normals or no normals, and to stick with triangles when normals are used. Also, specified normals should point in the general direction of the surface for best results.

Vertices should not be shared between any two contours. I.e., a hole should not share a vertex or edge with the perimeter or another hole, or incorrect rendering may result.

EXAMPLE

```
# Make a wall with a window using an explicit hole.
# (See Figures 1 and 2.)
fh v1 v2 v3 v4 - v5 v6 v7 v8
```

SEE ALSO

cone, cyl, f, m, prism, ring, sph, torus, v

NAME

sph - create a sphere

SYNOPSIS

sph *vc rad*

DESCRIPTION

Create a sphere made of the current material with its center at the named vertex *vc* and a radius of *rad.* If the vertex is undefined an error will result.

The surface normal is usually directed outward, but will be directed inward if the given radius is negative. (This typically matters only for one-sided materials.) A zero radius is illegal.

EXAMPLE

```
# Create a thick glass sphere with a hollow inside
m glass =
        sides 1
        ir 1.52 0
        c
        rs .06 0
        ts .88 0
v cent =
        p 0 0 1.1
# The outer shell
sph cent .1
# The inner bubble
sph cent -.08
```

SEE ALSO

cone, cyl, f, fh, m, prism, ring, torus, v

NAME

cyl - create an open-ended, truncated right cylinder

SYNOPSIS

cyl *v1 rad v2*

DESCRIPTION

Create a truncated right cylinder of radius *rad* using the current material, starting at the named vertex *v1* and continuing to *v2*. The ends will be open, but may be capped using the ring entity if desired.

The surface normal will usually be directed outward, but may be directed inward by giving a negative value for *rad*. A zero radius is illegal, and *v1* cannot equal *v2*.

EXAMPLE

```
# A stylus with one rounded and one pointed end
o stylus
        v vtip0 =
                p 0 0 0
        v vtip1 =
                p 0 0 .005
        v vend =
                p 0 0 .05
        cyl vtip1 .0015 vend
        sph vend .0015
        cone vtip0 0 vtip1 .0015
o
```

SEE ALSO

cone, f, fh, m, prism, ring, sph, torus, v

NAME

cone - create an open-ended, truncated right cone

SYNOPSIS

cone *v1 rad1 v2 rad2*

DESCRIPTION

Create a truncated right cone using the current material. The starting radius is *rad1* at *v1* and the ending radius is is *rad2* at *v2*. The ends will be open, but may be capped using the ring entity if desired.

The surface normal will usually be directed outward, but may be directed inward by giving negative values for both radii. (It is illegal for the signs of the two radii to disagree.) One but not both radii may be zero, indicating that the cone comes to a point.

Although it is not strictly forbidden to have equal cone radii, the cyl entity should be used in such cases. Likewise, the ring entity must be used if *v1* and *v2* are equal.

EXAMPLE

```
# A parasol
o parasol
        v v1 =
                p 0 0 0
        v v2 =
                p 0 0 .75
        v v3 =
                p 0 0 .7
        m handle_mat
        cyl v1 .002 v2
        m parasol_paper
        cyl v2 0 v3 .33
o
```

SEE ALSO

cyl, f, fh, m, prism, ring, sph, torus, v

NAME

prism - create a closed right prism

SYNOPSIS

prism *v1 v2 ... vN length*

DESCRIPTION

Create a closed right prism using the current material. One end face will be enclosed by the named vertices, and the opposite end face will be a mirror image at a distance *length* from the original. The edges will be extruded into N quadrilaterals connecting the two end faces.

The order of vertices determines the original face orientation according to the right-hand rule as explained for the f entity. Normally, the prism is extruded in the direction opposite to the original surface normal, resulting in faces that all point outward. If the specified *length* is negative, the prism will be extruded above the original face and all surface normals will point inward.

If the vertices have associated normals, they are applied to the side faces only, and should generally point in the appropriate direction (i.e. in or out depending on whether *length* is negative or positive).

EXAMPLE

```
# Make a unit cube starting at the origin and \
        extending to the positive octant
v cv0 =
        p 0 0 0
v cv1 =
        p 0 1 0
v cv2 =
        p 1 1 0
v cv3 =
        p 1 0 0
# Right hand rule has original face looking in -Z direction
prism cv0 cv1 cv2 cv3 1
```

SEE ALSO

cyl, cone, f, fh, m, ring, sph, torus, v

NAME

ring - create a circular ring with inner and outer radii

SYNOPSIS

ring *vc rmin rmax*

DESCRIPTION

Create a circular face of the current material centered on the named vertex *vc* with an inner radius of *rmin* and an outer radius of *rmax*. The surface orientation is determined by the normal vector associated with *vc*. If this vertex is undefined or has no normal, an error will result. The minimum radius may be equal to but not less than zero, and the maximum radius must be strictly greater than the minimum.

EXAMPLE

```
# The proverbial brass ring
o brass_ring
        m brass
        v end1 =
                p 0 -.005 0
                n 0 -1 0
        v end2 =
                p 0 .005 0
                n 0 1 0
        ring end1 .02 .03
        cyl end1 .03 end2
        ring end2 .02 .03
        cyl end2 -.02 end1
o
```

SEE ALSO

cyl, cone, f, fh, m, prism, sph, torus, v

NAME

torus - create a regular torus

SYNOPSIS

torus *vc rmin rmax*

DESCRIPTION

Create a torus of the current material centered on the named vertex *vc* with an inner radius of *rmin* and an outer radius of *rmax.* The plane of the torus will be perpendicular to the normal vector associated with *vc.* If this vertex is undefined or has no normal, an error will result.

If a torus with an inward facing surface normal is desired, *rmin* and *rmax* may be negative. The minimum radius may be zero, but may not be negative when *rmax* is positive or vice versa. The magnitude or *rmax* must always be strictly greater than that of *rmin.*

EXAMPLE

```
# The proverbial brass ring -- easy grip version
o brass_ring
        m brass
        v center =
                p 0 0 0
                n 0 1 0
        torus center .02 .03
o
```

SEE ALSO

cyl, cone, f, fh, m, prism, ring, sph, v

4. MGF Translators

Initially, there are six translators for MGF data, and three of these are distributed with the MGF parser itself, *mgfilt, mgf2inv* and *3ds2mgf*. Two of the other translators, *mgf2rad* and *rad2mgf* convert between MGF and the Radiance scene description language, and are distributed for free with the rest of the Radiance package†. The sixth translator, *mgf2meta,* converts to a 2-dimensional line plot, and is also distributed with Radiance.

Mgfilt is a simple but useful utility that takes MGF on its input and produces MGF on its output. It uses the parser to convert entities that are not wanted or understood, and produces only the requested ones. This is useful for seeing what exactly a program must understand when it supports a given set of entities, and may serve as a substitute for linking to the parser library for programmers who wish to interpret the ASCII input directly but without all the unwanted entities. In future releases of MGF, this utility will also be handy for taking new entities and producing older versions of MGF for translators that have not yet been updated properly.

Mgf2inv converts from MGF to Inventor or VRML format. Some information is lost, because these formats do not support physical light sources or materials.

3ds2mgf converts from 3D Studio binary format to MGF. Care must be taken to correct for errors in the material descriptions, since 3D Studio is completely non-physical.

†Radiance is available by anonymous ftp from hobbes.lbl.gov and nestor.epfl.ch, or by WWW from "http://radsite.lbl.gov/radiance/HOME.html"

NAME

mgfilt - get usable MGF entities from input

SYNOPSIS

mgfilt version [**input**..]
or
mgfilt e1,e2,.. [**input**..]

DESCRIPTION

Mgfilt takes one or more MGF input files and converts all the entities to the types listed. In the first form, a single integer is given for the *version* of MGF that is to be produced. Since MGF is in its first major release, this is not yet a useful form, but it will be when the second major release comes out. This has the necessary side-effect of expanding all included files. (See the i entity.)

In the second form, *mgfilt* produces only the entities listed in the first argument, which must be comma-separated. The listed entity order is not important, but all entities given must be defined in the current version of MGF. Unknown entities will be summarily discarded on the input, and a warning message will be printed to the standard error.

EXAMPLES

To take an MGF version 3 file and send it to a version 2 translator:

```
mgfilt 2 input.mgf | mgf2rad > input.rad
```

To take an MGF file and produce only flat polygonal faces with no materials:

```
mgfilt f,v,p,xf input.mgf > flatpoly.mgf
```

SEE ALSO

i, mgf2inv, mgf2rad, rad2mgf

NAME

mgf2inv - convert from MGF to Inventor or VRML format

SYNOPSIS

mgf2inv [**-1**|**-2**|**-vrml**] [**input**..]

DESCRIPTION

Mgf2inv takes one or more MGF input files and converts it to Inventor or VRML format. If the *–1* option is used, then Inventor 1.0 ASCII output is produced. If the *–2* option is used, then Inventor 2.0 ASCII output is produced. (This is the default.) If the *–vrml* option is used, then VRML 1.0 ASCII output is produced.

This converter does not work properly for light sources, since the output formats do not support IES-type luminaires with recorded distributions. Also, some material information may be lost because Inventor lacks a physically valid reflectance model.

EXAMPLES

To take an MGF file and convert it to VRML format:

```
mgf2inv -vrml myscene.mgf > myscene.iv
```

SEE ALSO

mgf2rad(1), mgfilt(1), 3ds2mgf(1), rad2mgf(1)

NAME

3ds2mgf - convert 3D Studio binary file to Materials and Geometry Format

SYNOPSIS

3ds2mgf input [**output**] [**-lMatlib**][**-xObjname**][**-sAngle**][**-aAnimfile**][**-fN**]

DESCRIPTION

3ds2mgf converts a 3D Studio binary scene description to the Materials and Geometry Format (MGF). If no output file name is given, the input root name will be taken as the output root, and an "mgf" extension will be added. This file will contain any light sources and materials, and an include statement for a similarly named file ending in "inc", which will contain the MGF geometry of all the translated 3DS meshes.

The MGF material names and properties for the surfaces will be those assigned in 3D Studio, unless they are named in one or more MGF material libraries given in a *-l* option.

The *-x* option may be used to exclude a named object from the output.

The *-s* option may be used to adjust automatic mesh smoothing such that adjacent triangle faces with less than the given angle between them (in degrees) will be smoothed. A value of zero turns smoothing off. The default value is 60 degrees.

The *-a* option may be used to specify a 3D Studio animation file, and together with the *-f* option, *3ds2mgf* will generate a scene description for the specified frame.

Note that there are no spaces between the options and their arguments.

LIMITATIONS

Obviously, since 3D Studio has no notion of physical materials, the translation to MGF material descriptions is very ad hoc, and it will usually be necessary to edit the materials and light sources in the output file or replace materials with proper entries from a material library using the *-l* option.

With smoothing turned on (i.e., a non-zero value for the *-s* option), vertices in the MGF output will not be linked in a proper mesh for each object. This is due to the way the automatic smoothing code was originally written, and is too difficult to repair. If a good mesh is needed, then smoothing must be turned off.

EXAMPLES

To convert a 3D Studio robot model to MGF without smoothing. (Output will be put into "robot.mgf" and "robot.inc".)

```
3ds2mgf robot.3ds -s0
```

To convert a DC10 jet model to MGF using a hand-created material library:

```
3ds2mgf dc10.3ds -ldc10mat.mgf
```

AUTHORS

Steve Anger, Jeff Bowermaster and Greg Ward
Extended from 3ds2pov 1.8.

SEE ALSO

mgf2inv(1), mgf2meta(1), mgf2rad(1)

NAME

mgf2rad - convert Materials and Geometry Format file to RADIANCE description

SYNOPSIS

mgf2rad [–**m matfile**][–**e mult**][–**g dist**] [**input**..]

DESCRIPTION

Mgf2rad converts one or more Materials and Geometry Format (MGF) files to a RADIANCE scene description. By definition, all output dimensions are in meters. The material names and properties for the surfaces will be those assigned in MGF. Any materials not defined in MGF will result in an error during translation. Light sources are described inline as IES luminaire files, and *mgf2rad* calls the program *ies2rad(1)* to translate these files. If an IES file in turn contains an MGF description of the local fixture geometry, this may result in a recursive call to *mgf2rad,* which is normal and should be transparent. The only side-effect of this additional translation is the appearance of other RADIANCE scene and data files produced automatically by *ies2rad.*

The *–m* option may be used to put all the translated materials into a separate RADIANCE file. This is not always advisable, as any given material name may be reused at different points in the MGF description, and writing them to a separate file loses the contextual association between materials and surfaces. As long as unique material names are used throughout the MGF description and material properties are not redefined, there will be no problem. Note that this is the only way to get all the translated materials into a single file, since no output is produced for unreferenced materials; i.e. translating just the MGF materials does not work.

The *–e* option may be used to multiply all the emission values by the given *mult* factor. The *–g* option may be used to establish a glow distance (in meters) for all emitting surfaces. These two options are employed principally by *ies2rad,* and are not generally useful to most users.

EXAMPLES

To translate two MGF files into one RADIANCE materials file and one geometry file:

```
mgf2rad -m materials.rad building1.mgf building2.mgf > building1+2.rad
```

To create an octree directly from two MGF files and one RADIANCE file:

```
oconv '\!mgf2rad materials.mgf scene.mgf' source.rad > scene.oct
```

FILES

tmesh.cal	Used to smooth polygonal geometry
*.rad	RADIANCE source descriptions created by ies2rad
*.dat	RADIANCE source data created by ies2rad
source.cal	Used for IES source coordinates

AUTHOR

Greg Ward

SEE ALSO

ies2rad(1), mgf2meta(1), obj2rad(1), oconv(1), rad2mgf(1), xform(1)

NAME

rad2mgf - convert RADIANCE scene description to Materials and Geometry Format

SYNOPSIS

rad2mgf [–**dU**] [**input**..]

DESCRIPTION

Rad2mgf converts one or more RADIANCE scene files to the Materials and Geometry Format (MGF). Input units are specified with the *–mU* option, where *U* is one of 'm' (meters), 'c' (centimeters), 'f' (feet) or 'i' (inches). The assumed unit is meters, which is the required output unit for MGF (thus the need to know). If the input dimensions are in none of these units, then the user should apply *xform(1)* with the *–s* option to bring the units into line prior to translation.

The MGF material names and properties for the surfaces will be those assigned in RADIANCE. If a referenced material has not been defined, then its name will be invoked in the MGF output without definition, and the description will be incomplete.

LIMITATIONS

Although MGF supports all of the geometric types and the most common material types used in RADIANCE, there is currently no support for advanced BRDF materials, patterns, textures or mixtures. Also, the special types "source" and "antimatter" are not supported, and all light source materials are converted to simple diffuse emitters (except "illum" materials, which are converted to their alternates). These primitives are reproduced as comments in the output and must be replaced manually if necessary.

The RADIANCE "instance" type is treated specially. *Rad2mgf* converts each instance to an MGF include statement, using the corresponding transformation and a file name derived from the octree name. (The original octree suffix is replaced by ".mgf".) For this to work, the user must separately create the referenced MGF files from the original RADIANCE descriptions. The description file names can usually be determined using the *getinfo(1)* command run on the octrees in question.

EXAMPLES

To convert three RADIANCE files (in feet) to one MGF file:

```
mgf2rad -df file1.rad file2.rad file3.rad > scene.mgf
```

To translate a RADIANCE materials file to MGF:

```
mgf2rad materials.rad > materials.mgf
```

AUTHOR

Greg Ward

SEE ALSO

getinfo(1), ies2rad(1), mgf2meta(1), mgf2rad(1), obj2rad(1), oconv(1), xform(1)

NAME

mgf2meta - convert Materials and Geometry Format file to Metafile graphics

SYNOPSIS

mgf2meta [**-t threshold**] **{x|y|z} xmin xmax ymin ymax zmin zmax** [**input**..]

DESCRIPTION

Mgf2meta converts one or more Materials and Geometry Format (MGF) files to a 2-D orthographic projection along the selected axis in the *metafile(1)* graphics format. All geometry is clipped to the specified bounding box, and the resulting orientation is as follows:

Projection	Orientation
x	Y-axis right, Z-axis up
y	Z-axis right, X-axis up
z	X-axis right, Z-axis up

If multiple input files are given, the first file prints in black, the second prints in red, the third in green and the fourth in blue. If more than four input files are given, they cycle through the colors again in three other line types: dashed, dotted and dot-dashed.

The *–t* option may be used to randomly throw out line segments that are shorter than the given *threshold* (given as a fraction of the plot width). Segments are included with a probability equal to the square of the line length over the square of the threshold. This can greatly reduce the number of lines in the drawing (and therefore improve the drawing speed) with only a modest loss in quality. A typical value for this parameter is 0.005.

All MGF material information is ignored on the input.

EXAMPLES

To project two MGF files along the Z-axis and display them under X11:

```
mgf2meta z 0 10 0 15 0 9 building1.mgf building2.mgf | x11meta
```

To convert a RADIANCE scene to a line drawing in RADIANCE picture format:

```
rad2mgf scene.rad | mgf2meta x ‘getbbox -h scene.rad‘ | meta2tga | ra_t8 -r > scene.pic
```

AUTHOR

Greg Ward

SEE ALSO

getbbox(1), meta2tga(1), metafile(5), mgf2rad(1), pflip(1), protate(1), psmeta(1), ra_t8(1), rad2mgf(1), t4014(1), x11meta(1)

5. MGF Parser Library

The principal motivation for creating a standard parser library for MGF is to make it easy for software developers to offer some base level of compliance. The key to making MGF easy to support in fact is the parser, which has the ability to express higher order entities in terms of lower order ones. For example, tori are part of the MGF specification, but if a given program or translator does not support them, the parser will convert them to cones. If cones are not supported either, it will convert them further into smoothed polygons. If smoothing (vertex normal information) is not supported, it will be ignored and the program will just get flat polygons. This is done in such a way that future versions of the standard may include new entities that old software does not even have to know about, and they will be converted appropriately. Forward compatibility is thus built right into the parser loading mechanism itself -- the programmer simply links to the new code and the new standard is supported without any further changes.

Language

The provided MGF parser is written in ANSI-C. This language was chosen for reasons of portability and efficiency. Almost all systems support some form of ANSI-compatible C, and many languages can cross-link to C libraries without modification. Backward compatibility to Kernighan and Ritchie C is achieved by compiling with the -DNOPROTO flag.

All of the data structures and prototypes needed for the library are in the header file "parser.h". This file is the best resource for the parser and is updated with each MGF release.

Mechanism

The parser works by a simple callback mechanism to routines that actually interpret the individual entities. Some of these routines will belong to the calling program, and some will be entity support routines included in the library itself. There is a global array of function pointers, called *mg_ehand.* It is defined thus:

```
extern int    (*mg_ehand[MG_NENTITIES])(int argc, char **argv);
```

Before parsing begins, this dispatch table is initialized to point to the routines that will handle each supported entity. Every entity handler has the same basic prototype, which is the same as the *main* function, i.e:

```
extern int    handler(int argc, char **argv);
```

The first argument is the number of words in the MGF entity (counting the entity itself) and the second argument is an array of nul-terminated strings with the entity and its arguments. The function should return zero or one of the error codes defined in "parser.h". A non-zero return value causes the parser to abort, returning the error up through its call stack to the entry function, usually *mg_load.*

A special function pointer for undefined entities is defined as follows:

```
extern int    (*mg_uhand)(int argc, char **argv);
```

By default, this points to the library function *mg_defuhand,* which prints an error message on the first unknown entity and keeps a count from then on, which is stored in the global unsigned integer *mg_nunknown.* If the *mg_uhand* pointer is assigned a value of NULL instead, parsing will abort at the first unrecognized entity. The reason this is not the default action is that ignoring unknown entities offers a certain base level of forward compatibility. Ignoring things one does not understand is not the best approach, but it is usually better than quitting with an error message if the input is in fact valid, but is a later version of the standard. The real solution is to update the interpreter by linking to a new version of the parser, or use a new version of the *mgfilt* command to convert the new MGF input to an older standard.

The *mg_uhand* pointer may also be used to customize the language for a particular application by adding entities, though this is discouraged because it tends to weaken the standard.

The skeletal framework for an MGF loader or translator is to assign function pointers to the *mg_ehand* array, call the parser initialization function *mg_init*, then call the file loader function *mg_load* once for each input file. This will in turn make calls back to the functions assigned to *mg_ehand*. To give a simple example, let us look at a translator that understands only flat polygonal faces, putting out vertex locations immediately after each "face" keyword:

```
#include <stdio.h>
#include "parser.h"

int
myfaceh(ac, av)                 /* face handling routine */
int     ac;
char    **av;
{
        C_VERTEX        *vp;    /* vertex structure pointer */
        FVECT       vert;       /* vertex point location */
        int     i;

        if (ac < 4)             /* check # arguments */
                return(MG_EARGC);
        printf("face\n");       /* begin face output */
        for (i = 1; i < ac; i++) {
                if ((vp = c_getvert(av[i])) == NULL) /* vertex from name */
                        return(MG_EUNDEF);
                xf_xfmpoint(vert, vp->p);               /* apply transform */
                printf("%15.9f %15.9f %15.9f\n",
                        vert[0], vert[1], vert[2]);     /* output vertex */
        }
        printf(";\n");                  /* end of face output */
        return(MG_OK);                  /* normal exit */
}

main(argc, argv)            /* translate MGF file(s) */
int     argc;
char    **argv;
{
        int     i;
                                /* initialize dispatch table */
        mg_ehand[MG_E_FACE] = myfaceh;          /* ours */
        mg_ehand[MG_E_VERTEX] = c_hvertex;      /* parser lib */
        mg_ehand[MG_E_POINT] = c_hvertex;       /* parser lib */
        mg_ehand[MG_E_XF] = xf_handler;         /* parser lib */
        mg_init();                      /* initialize parser */
        for (i = 1; i < argc; i++)      /* load each file argument */
                if (mg_load(argv[i]) != MG_OK)      /* and check for error */
                        exit(1);
        exit(0);                        /* all done! */
}
```

Hopefully, this example demonstrates just how easy it is to write an MGF translator. Of course, translators get more complicated the more entity types they support, but the point is that one does not *have* to support every entity -- the parser handles what the translator does not. Also, the library includes many general entity handlers, further reducing the burden on the programmer.

This same principle means that it is not necessary to modify an existing program to accommodate a new version of MGF -- one need only link to the new parser library to comply with the new standard.

Division of Labor

As seen in the previous example, there are two parser routines that are normally called directly in an MGF translator or loader program. The first is *mg_init,* which takes no arguments but relies on the program having initialized those parts of the global *mg_ehand* array it cares about. The second routine is *mg_load,* which is called once on each input file. (A third routine, *mg_clear,* may be called to free the parser data structures after each file or after all files, if the program plans to continue rather than exit.)

The rest of the routines in a translator or loader program are called indirectly through the *mg_ehand* dispatch table, and they are the ones that do the real work of supporting the MGF entities. In addition to converting or discarding entities that the calling program does not know or care about, the parser library includes a set of context handlers that greatly simplify the translation process. There are three handlers for each of the three named contexts and their constituents, and two handlers for the two hierarchical context entities. To use these handlers, one simply sets the appropriate positions in the *mg_ehand* dispatch table to point to these functions. Additional functions and global data structures provide convenient access to the relevant contexts, and all of these are detailed in the following manual pages.

NAME

mg_init, mg_ehand, mg_uhand - initialize MGF entity handlers

SYNOPSIS

#include "parser.h"

void mg_init(**void**)

int mg_defuhand(**int** argc, **char** **argv)

extern int (*mg_ehand[MG_NENTITIES])(**int** argc, **char** **argv)

extern int (*mg_uhand)(**int** argc, **char** **argv)

extern unsigned mg_nunknown

DESCRIPTION

The parser dispatch table, *mg_ehand* is initially set to all NULL pointers, and it is the duty of the calling program to assign entity handler functions to each of the supported entity positions in the array. The entities are given in the include file "parser.h" as the following:

```
#define MG_E_COMMENT    0       /* #       */
#define MG_E_COLOR      1       /* c       */
#define MG_E_CCT        2       /* cct     */
#define MG_E_CONE       3       /* cone    */
#define MG_E_CMIX       4       /* cmix    */
#define MG_E_CSPEC      5       /* cspec   */
#define MG_E_CXY        6       /* cxy     */
#define MG_E_CYL        7       /* cyl     */
#define MG_E_ED         8       /* ed      */
#define MG_E_FACE       9       /* f       */
#define MG_E_INCLUDE    10      /* i       */
#define MG_E_IES        11      /* ies     */
#define MG_E_IR         12      /* ir      */
#define MG_E_MATERIAL   13      /* m       */
#define MG_E_NORMAL     14      /* n       */
#define MG_E_OBJECT     15      /* o       */
#define MG_E_POINT      16      /* p       */
#define MG_E_PRISM      17      /* prism   */
#define MG_E_RD         18      /* rd      */
#define MG_E_RING       19      /* ring    */
#define MG_E_RS         20      /* rs      */
#define MG_E_SIDES      21      /* sides   */
#define MG_E_SPH        22      /* sph*/
#define MG_E_TD         23      /* td      */
#define MG_E_TORUS      24      /* torus   */
#define MG_E_TS         25      /* ts      */
#define MG_E_VERTEX     26      /* v       */
#define MG_E_XF         27      /* xf      */

#define MG_NENTITIES    28      /* total # entities */
```

Once the *mg_ehand* array has been set by the program, the *mg_init* routine must be called to complete the initialization process. This should be done once and only once per invocation, before any other parser routines are called.

The *mg_uhand* variable points to the current handler for unknown entities encountered on the input. Its default value points to the *mg_defuhand* function, which simply increments the global variable *mg_nunknown*, printing a warning message on the standard error on the first offense. (This message may be avoided by incrementing *mg_nunknown* before processing begins.) If *mg_uhand* is assigned a value of NULL, then an unknown entity will return an *MG_EUNK* error, which will cause the parser to abort. (See the *mg_load* page for a list of errors.) If the *mg_uhand* pointer is assigned to another function, that function will receive any unknown entities and their arguments, and the parsing will abort if the new function returns a non-zero error value. This offers a convenient way to customize the language by adding non-standard entities.

DIAGNOSTICS

If an inconsistent set of entities has been set for support, the *mg_init* routine will print an informative message to standard error and abort the calling program with a call to *exit*. This is normally unacceptable behavior for a library routine, but since such an error indicates a fault with the calling program itself, recovery is impossible.

SEE ALSO

mg_load, mg_handle

NAME

mg_load, mg_clear, mg_file, mg_err - load MGF file, clear data structures

SYNOPSIS

#include "parser.h"

int mg_load(**char** *filename)

void mg_clear(**void**)

extern MG_FCTXT *mg_file

extern char *mg_err[MG_NERRS]

DESCRIPTION

The *mg_load* function loads the named file, or standard input if *filename* is the NULL pointer. Calls back to the appropriate MGF handler routines are made through the *mg_ehand* dispatch table.

The global *mg_file* variable points to the current file context structure, which may be useful for the interpretation of certain entities, such as ies, which must know the directory path of the enclosing file. This structure is of the defined type *MG_FCTXT,* given in "parser.h" as:

```
typedef struct mg_fctxt {
        char   fname[96];               /* file name */
        FILE  *fp;                      /* stream pointer */
        int    fid;                     /* unique file context id */
        char   inpline[4096];           /* input line */
        int    lineno;                  /* line number */
        struct mg_fctxt    *prev;       /* previous context */
} MG_FCTXT;
```

DIAGNOSTICS

If an error is encountered during parsing, *mg_load* will print an appropriate error message to the standard error stream and return one of the non-zero values from "parser.h" listed below:

```
#define MG_OK           0       /* normal return value */
#define MG_EUNK         1       /* unknown entity */
#define MG_EARGC        2       /* wrong number of arguments */
#define MG_ETYPE        3       /* argument type error */
#define MG_EILL         4       /* illegal argument value */
#define MG_EUNDEF       5       /* undefined reference */
#define MG_ENOFILE      6       /* cannot open input file */
#define MG_EINCL        7       /* error in included file */
#define MG_EMEM         8       /* out of memory */
#define MG_ESEEK        9       /* file seek error */
#define MG_EBADMAT      10      /* bad material specification */
#define MG_ELINE        11      /* input line too long */
#define MG_ECNTXT       12      /* unmatched context close */

#define MG_NERRS        13
```

If it is inappropriate to send output to standard error, the calling program should use the routines listed under *mg_open* for better control over the parsing process.

The *mg_err* array contains error messages corresponding to each of the values listed above in the native country's language.

SEE ALSO

mg_fgetpos, mg_handle, mg_init, mg_open

NAME

mg_open, mg_read, mg_parse, mg_close - MGF file loading subroutines

SYNOPSIS

#include "parser.h"

int mg_open(MG_FCTXT *fcp, **char** *filename)

int mg_read(**void**)

int mg_parse(**void**)

void mg_close(**void**)

DESCRIPTION

Most loaders and translators will call the *mg_load* routine to handle the above operations, but some programs or entity handlers require tighter control over the loading process.

The *mg_open* routine takes an uninitialized *MG_FCTXT* structure and a file name as its arguments. If *filename* is the NULL pointer, the standard input is "opened." The *fcp* structure will be set by *mg_open* prior to its return, and the global *mg_file* pointer will be assigned to point to it. This variable must not be destroyed until after the file is closed with a call to *mg_close.* (See the *mg_load* page for a definition of *mg_file* and the *MG_FCTXT* type.)

The *mg_read* function reads the next input line from the current file, returning the number of characters in the line, or zero if the end of file is reached or there is a file error. If the value returned equals MG_MAXLINE-1, then the input line was too long, and you should return an MG_ELINE error. The function keeps track of the line number in the current file context *mg_file,* which also contains the line that was read.

The *mg_parse* function breaks the current line in the *mg_file* structure into words and calls the appropriate handler routine, if any. Blank lines and unsupported entities cause a quick return.

The *mg_close* routine closes the current input file (unless it is the standard input) and returns to the previous file context (if any).

DIAGNOSTICS

The *mg_open* function returns *MG_OK* (0) normally, or *MG_ENOFILE* if the open fails for some reason.

The *mg_parse* function returns *MG_OK* if the current line was successfully interpreted, or one of the defined error values if there is a problem. (See the *mg_load* page for the defined error values.)

SEE ALSO

mg_fgetpos, mg_handle, mg_init, mg_load

NAME

mg_fgetpos, mg_fgoto - get current file position and seek to pointer

SYNOPSIS

#include "parser.h"

void mg_fgetpos(MG_FPOS *pos)

int mg_fgoto(MG_FPOS *pos)

DESCRIPTION

The *mg_fgetpos* gets the current MGF file position and loads it into the passed *MG_FPOS* structure, *pos.*

The *mg_fgoto* function seeks to the position *pos,* taken from a previous call to *mg_fgetpos.*

DIAGNOSTICS

If *mg_fgoto* is passed an illegal pointer or one that does not correspond to the current *mg_file* context, it will return the *MG_ESEEK* error value. Normally, it returns *MG_OK* (0).

SEE ALSO

mg_load, mg_open

NAME

mg_handle, mg_entity, mg_ename, mg_nqcdivs - entity assistance and control

SYNOPSIS

int mg_handle(**int** en, **int** ac, **char** *av)

int mg_entity(**char** *name)

extern char mg_ename[MG_NENTITIES][MG_MAXELEN]

extern int mg_nqcdivs

DESCRIPTION

The *mg_handle* routine may be used to pass entities back to the parser to be redirected through the *mg_ehand* dispatch table. This method is recommended rather than calling through *mg_ehand* directly, since the parser sometimes has its own support routines that it needs to call for specific entities. The first argument, *en*, is the corresponding entity number, or -1 if *mg_handle* should figure it out from the first *av* argument.

The *mg_entity* function gets an entity number from its name, using a hash table on the *mg_ename* list.

The *mg_ename* table contains the string names corresponding to each MGF entity in the designated order. (See the *mg_init* page for the list of MGF entities.)

The global integer variable *mg_nqcdivs* tells the parser how many subdivisions to use per quarter circle (90 degrees) when tesselating curved geometry. The default value is 5, and it may be reset at any time by the calling program.

DIAGNOSTICS

The *mg_handle* function returns *MG_OK* if the entity is handled correctly, or one of the predefined error values if there is a problem. (See the *mg_load* page for a list of error values.)

The *mg_entity* function returns -1 if the passed name does not appear in the *mg_ename* list.

SEE ALSO

mg_init, mg_load, mg_open

NAME

isint, isflt, isname - determine if string fits integer or real format, or is legal identifier

SYNOPSIS

int isint(**char** *str)

int isflt(**char** *str)

int isname(**char** *str)

DESCRIPTION

The *isint* function checks to see if the passed string *str* matches a decimal integer format (positive or negative), and returns 1 or 0 based on whether it does or does not.

The *isflt* function checks to see if the passed string *str* matches a floating point format (positive or negative with optional exponent), and returns 1 or 0 based on whether it does or does not.

The *isname* function checks to see if the passed string *str* is a legal identifier name. In MGF, a legal identifier must begin with a letter and contain only visible ASCII characters (those between decimal 33 and 127 inclusive). The one caveat to this is that names may begin with one or more underscores ('_'), but this is a trick employed by the parser to maintain a separate name space from the user, and is not legal usage otherwise.

Note that a string that matches an integer format is also a valid floating point value. Conversely, a string that is not a floating point number cannot be a valid integer.

These routines are useful for checking arguments passed to entity handlers that certain types in certain positions. If an invalid argument is passed, the handler should return an *MG_ETYPE* error.

SEE ALSO

mg_init, mg_load

NAME

c_hvertex, c_getvert, c_cvname, c_cvertex - vertex entity support

SYNOPSIS

#include "parser.h"

int c_hvertex(**int** argc, **char** **argv)

C_VERTEX *c_getvert(**char** *name)

extern char *c_vname

extern C_VERTEX *c_cvertex

DESCRIPTION

The *c_hvertex* function handles the MGF vertex entities, v, p and n. If either p or n is supported, then v must be also. The assignments are normally made to the *mg_ehand* array prior to parser initialization, like so:

```
mg_ehand[MG_E_VERTEX] = c_hvertex;      /* support "v" entity */
mg_ehand[MG_E_POINT] = c_hvertex;       /* support "p" entity */
mg_ehand[MG_E_NORMAL] = c_hvertex;      /* support "n" entity */
/* other entity handler assignments... */
mg_init();                  /* initialize parser */
```

If vertex normals are not understood by any of the program-supported entities, then the *MG_E_NORMAL* entry may be left with its original NULL assignment.

The *c_getvert* call takes the name of a defined vertex and returns a pointer to its *C_VERTEX* structure, defined in "parser.h" as:

```
typedef FLOAT  FVECT[3];    /* a 3-d real vector */

typedef struct {
        int     clock;              /* incremented each change -- resettable */
        char   *client_data;        /* pointer to private client data */
        FVECT       p, n;           /* point and normal */
} C_VERTEX;             /* vertex context */
```

The *clock* member will be incremented each time the value gets changed by a p or n entity, and may be reset by the controlling program if desired. This is a convenient way to keep track of whether or not a vertex has changed since its last use. To link identical vertices, one must also check that the current transform has not changed, which is uniquely identified by the global *xf_context->xid* variable, but only if one is using the parser library's transform handler. (See the *xf_handler* page.) The *client_data* pointer may be used to index private application data for vertex linking, etc. This pointer is initialized to NULL when the context is created, and otherwise ignored by the parser library.

It is possible but not recommended to alter the shared contents of the vertex structure returned by *c_getvert.* Normally it is read during the interpretation of entities using named vertices.

The name of the current vertex is given by the global *c_cvname* variable, which is set to NULL if the unnamed vertex is current. The current vertex value is pointed to by the global variable *c_cvertex,* which should never be NULL.

DIAGNOSTICS

The *c_hvertex* function returns *MG_OK* (0) if the vertex is handled correctly, or one of the predefined error values if there is a problem. (See the *mg_load* page for a list of errors.)

The *c_getvert* function returns NULL if the specified vertex name is undefined, at which point the calling function should return an *MG_EUNDEF* error.

SEE ALSO

c_hcolor, c_hmaterial, mg_init, mg_load, xf_handler

NAME

c_hcolor, c_getcolor, c_ccname, c_ccolor, c_ccvt, c_isgrey - color entity support

SYNOPSIS

#include "parser.h"

int c_hcolor(**int** argc, **char** **argv)

C_COLOR *c_getcolor(**char** *name)

extern char *c_ccname

extern C_COLOR *c_ccolor

void c_ccvt(C_COLOR *cvp, **int** cflags)

int c_isgrey(C_COLOR *cvp)

DESCRIPTION

The *c_hcolor* function supports the MGF entities, c, cxy, cspec, cct and cmix. It is an error to support any of the color field entities without supporting the c entity itself. The assignments are normally made to the *mg_ehand* array prior to parser initialization, like so:

```
mg_ehand[MG_E_COLOR] = c_hcolor;    /* support "c" entity */
mg_ehand[MG_E_CXY] = c_hcolor;      /* support "cxy" entity */
mg_ehand[MG_E_CSPEC] = c_hcolor;    /* support "cspec" entity */
mg_ehand[MG_E_CCT] = c_hcolor;      /* support "cct" entity */
mg_ehand[MG_E_CMIX] = c_hcolor;     /* support "cmix" entity */
/* other entity handler assignments... */
mg_init();              /* initialize parser */
```

If the loader/translator has no use for spectral data, the entries for cspec and cct may be left with their original NULL assignments and these entities will be re-expressed appropriately as tristimulus values.

The *c_getcolor* function takes the name of a defined color and returns a pointer to its *C_COLOR* structure, defined in "parser.h" as:

```
#define C_CMINWL    380                     /* minimum wavelength */
#define C_CMAXWL    780                     /* maximum wavelength */
#define C_CNSS      41                      /* number of spectral samples */
#define C_CWLI      ((C_CMAXWL-C_CMINWL)/(C_CNSS-1))
#define C_CMAXV         10000               /* nominal maximum sample value */
#define C_CLPWM         (683./C_CMAXV)      /* peak lumens/watt multiplier */

typedef struct {
        int     clock;          /* incremented each change */
        char    *client_data;   /* pointer to private client data */
        short   flags;          /* what's been set */
        short   ssamp[C_CNSS];  /* spectral samples, min wl to max */
        long    ssum;           /* straight sum of spectral values */
        float   cx, cy;         /* xy chromaticity value */
        float   eff;            /* efficacy (lumens/watt) */
} C_COLOR;              /* color context */
```

The *clock* member will be incremented each time the value gets changed by a color field entity, and may be reset by the calling program if desired. This is a convenient way to keep track of whether or not a color has changed since its last use. The *client_data* pointer may be used to

index private application data. This pointer is initialized to NULL when the context is created, and otherwise ignored by the parser library. The *flags* member indicates which color representations have been assigned, and is an inclusive OR of one or more of the following:

```
#define C_CSSPEC    01      /* flag if spectrum is set */
#define C_CDSPEC    02      /* flag if defined w/ spectrum */
#define C_CSXY      04      /* flag if xy is set */
#define C_CDXY      010     /* flag if defined w/ xy */
#define C_CSEFF     020     /* flag if efficacy set */
```

It is possible but not recommended to alter the contents of the color structure returned by *c_getcolor*. Normally, this routine is never called directly, since there are no entities that access colors by name other than c.

The global variable *c_ccname* points to the name of the current color, or NULL if it is unnamed. The variable *c_ccolor* points to the current color value, which should never be NULL.

The *c_ccvt* routine takes a *C_COLOR* structure and a set of desired flag settings and computes the missing color representation(s).

The *c_isgrey* function returns 1 if the passed color is very close to neutral grey, or 0 otherwise.

DIAGNOSTICS

The *c_hcolor* function returns *MG_OK* (0) if the color is handled correctly, or one of the predefined error values if there is a problem. (See the *mg_load* page for a list of errors.)

The *c_getcolor* function returns NULL if the specified color name is undefined, at which point the calling function should return an *MG_EUNDEF* error.

SEE ALSO

c_hmaterial, c_hvertex, mg_init, mg_load

NAME

c_hmaterial, c_getmaterial, c_cmname, c_cmaterial - material entity support

SYNOPSIS

#include "parser.h"

int c_hmaterial(**int** argc, **char** **argv)

C_MATERIAL *c_getmaterial(**char** *name)

extern char *c_cmname

extern C_MATERIAL *c_cmaterial

DESCRIPTION

The *c_hmaterial* function supports the MGF entities, m, ed, ir, rd, rs, sides, td, and ts. It is an error to support any of the material field entities without supporting the m entity itself. The assignments are normally made to the *mg_ehand* array prior to parser initialization, like so:

```
mg_ehand[MG_E_MATERIAL] = c_hmaterial;  /* support "m" entity */
mg_ehand[MG_E_ED] = c_hmaterial;        /* support "ed" entity */
mg_ehand[MG_E_IR] = c_hmaterial;        /* support "ir" entity */
mg_ehand[MG_E_RD] = c_hmaterial;        /* support "rd" entity */
mg_ehand[MG_E_RS] = c_hmaterial;        /* support "rs" entity */
mg_ehand[MG_E_SIDES] = c_hmaterial;     /* support "sides" entity */
mg_ehand[MG_E_TD] = c_hmaterial;        /* support "td" entity */
mg_ehand[MG_E_TS] = c_hmaterial;        /* support "ts" entity */
/* other entity handler assignments... */
mg_init();                  /* initialize parser */
```

Any of the above entities besides m may be unsupported, but the parser will not attempt to include their effect into other members, e.g. an unsupported rs component will not be added back into the rd member. It is therefore safer to support all of the relevant material entities and make final approximations from the complete *C_MATERIAL* structure.

The *c_getmaterial* function takes the name of a defined material and returns a pointer to its *C_MATERIAL* structure, defined in "parser.h" as:

```
#define C_1SIDEDTHICK     0.005         /* assumed thickness of 1-sided mat. */

typedef struct {
        int     clock;          /* incremented each change -- resettable */
        char    *client_data;         /* pointer to private client data */
        int     sided;          /* 1 if surface is 1-sided, 0 for 2-sided */
        float   nr, ni;         /* index of refraction, real and imaginary */
        float   rd;             /* diffuse reflectance */
        C_COLOR  rd_c;  /* diffuse reflectance color */
        float   td;             /* diffuse transmittance */
        C_COLOR  td_c;  /* diffuse transmittance color */
        float   ed;             /* diffuse emittance */
        C_COLOR  ed_c; /* diffuse emittance color */
        float   rs;             /* specular reflectance */
        C_COLOR  rs_c;  /* specular reflectance color */
        float   rs_a;           /* specular reflectance roughness */
        float   ts;             /* specular transmittance */
        C_COLOR  ts_c;  /* specular transmittance color */
        float   ts_a;           /* specular transmittance roughness */
} C_MATERIAL; /* material context */
```

The *clock* member will be incremented each time the value gets changed by a material field entity, and may be reset by the calling program if desired. This is a convenient way to keep track of whether or not a material has changed since its last use. The *client_data* pointer may be used to index private application data. This pointer is initialized to NULL when the context is created, and otherwise ignored by the parser library.

All reflectance and transmittance values correspond to normal incidence, and may vary as a function of angle depending on the index of refraction. A solid object is normally represented with a one-sided material. A two-sided material is most appropriate for thin surfaces, though it may be used also when the surface normal orientations in a model are unreliable.

If a transparent or translucent surface is one-sided, then the absorption will change as a function of distance through the material, and a single value for diffuse or specular transmittance is ambiguous. We therefore define a standard thickness, *C_1SIDEDTHICK*, which is the object thickness to which the given values correspond, so that one may compute the isotropic absorptance of the material.

It is possible but not recommended to alter the contents of the material structure returned by *c_getmaterial.* Normally, this routine is never called directly, since there are no entities that access materials by name other than m.

The global variable *c_cmname* points to the name of the current material, or NULL if it is unnamed. The variable *c_cmaterial* points to the current material value, which should never be NULL.

DIAGNOSTICS

The *c_hmaterial* function returns *MG_OK* (0) if the color is handled correctly, or one of the predefined error values if there is a problem. (See the *mg_load* page for a list of errors.)

The *c_getmaterial* function returns NULL if the specified material name is undefined, at which point the calling function should return an *MG_EUNDEF* error.

SEE ALSO

c_hcolor, c_hvertex, mg_init, mg_load

NAME

obj_handler, obj_clear, obj_nnames, obj_name - object name support

SYNOPSIS

int obj_handler(**int** argc, **char** **argv)

void obj_clear(**void**)

extern int obj_nnames

extern char **obj_name

DESCRIPTION

The *obj_handler* routine should be assigned to the *MG_E_OBJECT* entry of the parser's *mg_ehand* array prior to calling *mg_load* if the loader/translator wishes to support hierarchical object names.

The *obj_clear* function may be used to clear the object name stack and free any associated memory, but this is usually not necessary since o begin and end entities are normally balanced in the input.

The global *obj_nnames* variable indicates the number of names currently in the object stack, and the *obj_name* list contains the name strings in the same order as they were encountered on the input. (I.e. the most recently pushed name is last.)

DIAGNOSTICS

The *obj_handler* function returns *MG_OK* (0) if the color is handled correctly, or one of the predefined error values if there is a problem. (See the *mg_load* page for a list of errors.)

SEE ALSO

mg_init, mg_load, xf_handler

NAME

xf_handler, xf_clear, xf_context, xf_argend - transformation support

SYNOPSIS

int xf_handler(**int** argc, **char** **argv)

void xf_clear(**void**)

extern XF_SPEC *xf_context

extern char **xf_argend

DESCRIPTION

The *xf_handler* routine should be assigned to the *MG_E_XF* entry of the parser's *mg_ehand* array prior to calling *mg_load* if the loader/translator wishes to support hierarchical transformations. (Note that all MGF geometric entities require this support.)

The *xf_clear* function may be used to clear the transform stack and free any associated memory, but this is usually not necessary since xf begin and end entities are normally balanced in the input.

The global *xf_context* variable points to the current transformation context, which is of the type *XF_SPEC*, described in "parser.h":

```
typedef struct xf_spec {
        long   xid;                     /* unique transform id */
        short  xac;                     /* context argument count */
        short  rev;                     /* boolean true if vertices reversed */
        XF     xf;                      /* cumulative transformation */
        struct xf_array     *xarr;      /* transformation array pointer */
        struct xf_spec      *prev;      /* previous transformation context */
} XF_SPEC;              /* followed by argument buffer */
```

The *xid* member is a identifier associated with this transformation, which should be the same for identical transformations, as an aid to vertex sharing. (See also the *c_hvertex* page.) The *xac* member indicates the total number of transform arguments, and is used to indicate the position of the first argument relative to the last one pointed to by the global *xf_argend* variable.

The first transform argument starts at *xf_argv*, which is a macro defined in "parser.h" as:

```
#define xf_argv         (xf_argend - xf_context->xac)
```

Note that accessing this macro will result in a segmentation violation if the current context is NULL, so one should first test the second macro *xf_argc* against zero. This macro is defined as:

```
#define xf_argc         (xf_context==NULL ? 0 : xf_context->xac)
```

Normally, neither of these macros will be used, since there are routines for transforming points, vectors and scalars directly based on the current transformation context. (See the *xf_xfmpoint* page for details.)

The *rev* member of the *XF_SPEC* structure indicates whether or not this transform reverses the order of polygon vertices. This member will be 1 if the transformation mirrors about an odd number of coordinate axes, thus inverting faces. The usual thing to do in this circumstance is to interpret the vertex arguments in the reverse order, so as to bring the face back to its original orientation in the new position.

The *xf* member contains the transformation scalefactor (in xf.sca) and 4x4 homogeneous matrix (in xf.xfm), but these will usually not be accessed directly. Likewise, the *xarr* and *prev*

members point to data that should not be needed by the calling program.

DIAGNOSTICS

The *xf_handler* function returns *MG_OK* (0) if the color is handled correctly, or one of the predefined error values if there is a problem. (See the *mg_load* page for a list of errors.)

SEE ALSO

mg_init, mg_load, obj_handler, xf_xfmpoint

NAME

xf_xfmpoint, xf_xfmvect, xf_rotvect, xf_scale - apply current transformation

SYNOPSIS

void xf_xfmpoint(FVECT pnew, FVECT pold)

void xf_xfmvect(FVECT vnew, FVECT vold)

void xf_rotvect(FVECT nnew, FVECT nold)

double xf_scale(**double** sold)

DESCRIPTION

The *xf_xfmpoint* routine applies the current transformation defined by *xf_context* to the point *pold*, scaling, rotating and moving it to its proper location, which is put in *pnew*. (As for *xf_xfmvect* and *xf_rotvect*, the two arguments may point to the same vector.)

The *xf_xfmvect* routine applies the current transformation to the vector *vold*, scaling and rotating it to its proper location, which is put in *vnew*. The only difference between *xf_xfmpoint* and *xf_xfmvect* is that in the latter, the final translation is not applied.

The *xf_rotvect* routine rotates the vector *nold* using the current transformation, and stores the result in *nnew*. No translation or scaling is applied, which is the appropriate action for surface normal vectors for example.

The *xf_scale* function takes a scalar argument *sold* and applies the current scale factor, returning the result.

SEE ALSO

xf_handler

6. Application Notes

6.1. Relation to Standard Practices in Computer Graphics

For those coming from a computer graphics background, some of the choices in the material model may seem strange or even capricious. Why not simply stick with RGB colors and a Phong specular component like everyone else? What is the point in choosing the number of sides to a material?

In the real world, a surface can have only one side, defining the interface between one volume and another. Many object-space rendering packages (e.g. z-buffer algorithms) take advantage of this fact by culling back-facing polygons and thus saving as much as 50% of the preprocessing time. However, many models rely on an approximation whereby a single surface is used to represent a very thin volume, such as a pane of glass, and this also can provide significant calculational savings in an image-space algorithm (such as ray-tracing). Also, many models are created in such a way that the front vs. back information is lost or confused, so that the back side of one or more surfaces may have to serve as the front side during rendering. (AutoCAD is one easily identified culprit in this department.) Since both types of surface models are useful and any rendering algorithm may ultimately be applied, MGF provides a way to specify sidedness rather than picking one interpretation or the other.

The problem with RGB is that there is no accepted standard, and even if we were to set one it would either be impossible to realize (i.e. impossible to create phosphors with the chosen colors) or it would have a gamut that excludes many saturated colors. The CIE color system was very carefully conceived and developed, and is the standard to which all photometric measurements adhere. It is therefore the logical choice in any standard format, though it has been too often ignored by the computer graphics community.

Regarding Phong shading, this was never a physical model and making it behave basic laws of reciprocity and energy balance is difficult. More to the point, specular power has almost nothing to do with surface microstructure, and is difficult to set properly even if every physical characteristic of a material has been carefully measured. This is the ultimate indictment of any physical model -- that it is incapable of reproducing any measurement whatsoever.

Admittedly, the compliment of diffuse and specular component plus surface roughness and index of refraction used in MGF is less than a perfect model, but it is serviceable for most materials and relatively simple to incorporate into a rendering algorithm. In the long term, MGF shall probably include full spectral scattering functions, though the sheer quantity of data involved makes this burdensome from both the measurement side and the simulation side.

6.1.1. Converting between Phong Specular Power and Gaussian Roughness

So-called specular reflection and transmission are modeled using a Gaussian distribution of surface facets. The roughness parameters to the rs and ts entities specify the root-mean-squared (RMS) surface facet slope, which varies from 0 for a perfectly smooth surface to around .2 for a fairly rough one. The effect this will have on the reflected component distribution is well-defined, but predicting the behavior of the transmitted component requires further assumptions. We assume that the surface scatters light passing through it just as much as it scatters reflected light. This assumption is approximately correct for a two-sided transparent material with an index of refraction of 1.5 (like glass) and both sides having the given RMS facet slope.

Oftentimes, one is translating from a Phong exponent on the cosine of the half-vector-to-normal angle to the more physical but less familiar Gaussian model of MGF. The hardest part is translating the specular power to a roughness value. For this, we recommend the following approximation:

```
roughness = sqrt(2/specular_power)
```

It is not a perfect correlation, but it is about as close as one can get.

6.1.2. Converting between RGB and CIE Colors

Unlike most graphics languages, MGF does not use an RGB color model, simply because there is no recognized definition for this model. It is based on computer monitor phosphors, which vary from one CRT to the next. (There is an RGB standard defined in the TV industry, but this has a rather poor correlation to most computer monitors and it is impossible to express many real-world colors within its limited gamut.)

MGF uses two alternative, well-defined standards, spectral power distributions and the 1931 CIE 2 degree standard observer. With the CIE standard, any viewable color may be exactly represented as an (x,y) chromaticity value. Unfortunately, the interaction between colors (i.e. colored light sources and interreflections) cannot be specified exactly with any finite coordinate set, including CIE chromaticities. So, MGF offers the ability to give reflectance, transmittance or emittance as a function of wavelength over the visible spectrum. This function is still discretized, but at a user-selectable resolution. Furthermore, spectral colors may be mixed, providing (nearly) arbitrary basis functions, which can produce more accurate results in some cases and are merely a convenience for translation in others.

Conversion back and forth between CIE chromaticity coordinates and spectral samples is provided within the MGF parser. Unfortunately, conversion to and from RGB values depends on a particular RGB definition, and as we have said, there is no recognized standard. We therefore recommend that you decide yourself what chromaticity values to use for each RGB primary, and adopt the following code to convert between CIE and RGB coordinates.

```
#ifdef  NTSC
#define  CIE_x_r        0.670       /* standard NTSC primaries */
#define  CIE_y_r        0.330
#define  CIE_x_g        0.210
#define  CIE_y_g        0.710
#define  CIE_x_b        0.140
#define  CIE_y_b        0.080
#define  CIE_x_w        0.3333            /* monitor white point */
#define  CIE_y_w        0.3333
#else
#define  CIE_x_r        0.640       /* nominal CRT primaries */
#define  CIE_y_r        0.330
#define  CIE_x_g        0.290
#define  CIE_y_g        0.600
#define  CIE_x_b        0.150
#define  CIE_y_b        0.060
#define  CIE_x_w        0.3333            /* monitor white point */
#define  CIE_y_w        0.3333
#endif

#define CIE_D           (       CIE_x_r*(CIE_y_g - CIE_y_b) + \
                        CIE_x_g*(CIE_y_b - CIE_y_r) + \
                        CIE_x_b*(CIE_y_r - CIE_y_g)        )
#define CIE_C_rD        ( (1./CIE_y_w) * \
                        ( CIE_x_w*(CIE_y_g - CIE_y_b) - \
                          CIE_y_w*(CIE_x_g - CIE_x_b) + \
                          CIE_x_g*CIE_y_b - CIE_x_b*CIE_y_g   ) )
#define CIE_C_gD        ( (1./CIE_y_w) * \
                        ( CIE_x_w*(CIE_y_b - CIE_y_r) - \
                          CIE_y_w*(CIE_x_b - CIE_x_r) - \
```

```
                        CIE_x_r*CIE_y_b + CIE_x_b*CIE_y_r   ) )
#define CIE_C_bD        ( (1./CIE_y_w) * \
                        ( CIE_x_w*(CIE_y_r - CIE_y_g) - \
                          CIE_y_w*(CIE_x_r - CIE_x_g) + \
                          CIE_x_r*CIE_y_g - CIE_x_g*CIE_y_r    ) )

#define CIE_rf          (CIE_y_r*CIE_C_rD/CIE_D)
#define CIE_gf          (CIE_y_g*CIE_C_gD/CIE_D)
#define CIE_bf          (CIE_y_b*CIE_C_bD/CIE_D)

float  xyz2rgbmat[3][3] = {     /* XYZ to RGB */
        {(CIE_y_g - CIE_y_b - CIE_x_b*CIE_y_g + CIE_y_b*CIE_x_g)/CIE_C_rD,
         (CIE_x_b - CIE_x_g - CIE_x_b*CIE_y_g + CIE_x_g*CIE_y_b)/CIE_C_rD,
         (CIE_x_g*CIE_y_b - CIE_x_b*CIE_y_g)/CIE_C_rD},
        {(CIE_y_b - CIE_y_r - CIE_y_b*CIE_x_r + CIE_y_r*CIE_x_b)/CIE_C_gD,
         (CIE_x_r - CIE_x_b - CIE_x_r*CIE_y_b + CIE_x_b*CIE_y_r)/CIE_C_gD,
         (CIE_x_b*CIE_y_r - CIE_x_r*CIE_y_b)/CIE_C_gD},
        {(CIE_y_r - CIE_y_g - CIE_y_r*CIE_x_g + CIE_y_g*CIE_x_r)/CIE_C_bD,
         (CIE_x_g - CIE_x_r - CIE_x_g*CIE_y_r + CIE_x_r*CIE_y_g)/CIE_C_bD,
         (CIE_x_r*CIE_y_g - CIE_x_g*CIE_y_r)/CIE_C_bD}
};

float  rgb2xyzmat[3][3] = {     /* RGB to XYZ */
        {CIE_x_r*CIE_C_rD/CIE_D,CIE_x_g*CIE_C_gD/CIE_D,CIE_x_b*CIE_C_bD/CIE_D},
        {CIE_y_r*CIE_C_rD/CIE_D,CIE_y_g*CIE_C_gD/CIE_D,CIE_y_b*CIE_C_bD/CIE_D},
        {(1.-CIE_x_r-CIE_y_r)*CIE_C_rD/CIE_D,
         (1.-CIE_x_g-CIE_y_g)*CIE_C_gD/CIE_D,
         (1.-CIE_x_b-CIE_y_b)*CIE_C_bD/CIE_D}
};


cie_rgb(rgbcolor, ciecolor)             /* convert CIE to RGB */
register float  *rgbcolor, *ciecolor;
{
        register int  i;

        for (i = 0; i < 3; i++) {
                rgbcolor[i] =       xyz2rgbmat[i][0]*ciecolor[0] +
                                    xyz2rgbmat[i][1]*ciecolor[1] +
                                    xyz2rgbmat[i][2]*ciecolor[2] ;
                if (rgbcolor[i] < 0.0)          /* watch for negative values */
                        rgbcolor[i] = 0.0;
        }
}


rgb_cie(ciecolor, rgbcolor)             /* convert RGB to CIE */
register float  *ciecolor, *rgbcolor;
{
        register int  i;

        for (i = 0; i < 3; i++)
                ciecolor[i] =rgb2xyzmat[i][0]*rgbcolor[0] +
                                    rgb2xyzmat[i][1]*rgbcolor[1] +
```

```
                rgb2xyzmat[i][2]*rgbcolor[2] ;
}
```

An alternative to adopting the above code is to use the MGF "cmix" entity to convert from RGB directly by naming the three primaries in terms of their chromaticities, e.g:

```
c R =
        cxy 0.640 0.330
c G =
        cxy 0.290 0.600
c B =
        cxy 0.150 0.060
```

Then, converting from RGB to MGF colors is as simple as multiplying each component by its relative luminance in a cmix statement, for instance:

```
c white =
        cmix 0.265 R 0.670 G 0.065 B
```

For the chosen RGB standard, the above specification would result a pure white. The reason the coefficients are not all 1 as you might expect is that cmix uses relative luminance as the standard for its weights. Since blue is less luminous for the same energy than red, which is in turn less luminous than green, the weights cannot be the same to achieve an even spectral balance. Unfortunately, computing these relative weights is not straightforward, though it is given in the above macros as CIE_rf, CIE_gf and CIE_bf. (The common factors in these macros may of course be removed since cmix weights are all relative.) Alternatively, one could measure the actual full scale luminance of the phosphors with a luminance probe and get the same relative values.

6.2. Relation to IESNA LM-63 and Luminaire Catalogs

Recently, the Illuminating Engineering Society of North America (IESNA) adopted MGF as the official standard for representing luminaire geometry and materials. The way this works in an IES luminaire data file is through the addition of a keyword called LUMINOUSGEOMETRY, which is given on a line in the header portion of a file (before the TILT specification) like so:

[LUMINOUSGEOMETRY] *mgf_file*

The given MGF file must exist relative to the directory containing the IES file (i.e. the same stipulations and restrictions on pathnames apply as for the MGF i entity). Furthermore, the position of the MGF geometry must be such that the gross geometric specification of emitting surfaces in the IES file completely blocks or encloses the luminous portions of the MGF description. Specifically, any ray traced towards the MGF geometry must strike the IES gross geometry before it strikes any luminous surface in the MGF description. This provides a convenient way of preventing overcounting in the illumination calculation, while still allowing for accurate fixture appearance.

To give two examples, let us consider first a recessed can, followed by a hanging direct/indirect fluorescent fixture.

The most appropriate IES geometric specification for the emitting area of a can light would be a circular disk. Since the IES gross geometry gives only the diameter of the disk, the actual 3-dimensional placement is implicitly defined as having a center at the origin, with the radiating disk facing in the negative Z direction (nadir, downwards). The MGF geometry would then be placed such that any luminous portion was above this disk, and no portion of it would obstruct the IES geometry. The most sensible position therefore has the IES disk flush with the MGF can opening, as shown in Figure 3.

Replace this page with the second page from "figures.ps".

In the case of a direct/indirect fluorescent fixture, light will exit both the top and the bottom sides, and the IES geometry must enclose the radiating portion of the fixture entirely. It is acceptable to have additional MGF geometry above the fixture so long as it does not radiate, which is what we must do if we wish to include the support rods, as shown in Figure 4.

Note that the origin is always in the exact center of the IES geometry.

Not all fixtures will fit the simple IES geometry specification so nicely. For odd-shaped fixtures, it may be necessary to use an IES geometry that does not match the radiating area terribly well in order that it completely block or enclose the required MGF specification.

The unit of length in the MGF file is always meters, regardless of the units specified in the enclosing IES file. However, any and all multipliers applied to the candlepower data in the IES file will also be applied to the emittance of surfaces in the MGF specification, so that one MGF file may serve similar luminaires that differ in their total output.

7. Credits

The MGF language grew out of a joint investigation into physical representations for rendering undertaken by the author (Greg Ward of LBL) and Holly Rushmeier of the National Institute of Standards and Technology. After deciding that a complete and robust specification was an extreme challenge, we shelved the project for another time. A few months later, the author spoke with Ian Ashdown and Robert Shakespeare, who are both members of the IES Computing Committee, about the need for extending the existing data standard to include luminaire geometry and near-field photometry. We then moved forward as a team towards a somewhat less ambitious approach to physical materials and geometry that had the advantage of simplicity and the possibility of support with a standard parser library. The author went to work over the next two months on the detailed design of the language and an ANSI-C parser, with regular feedback from the other three team members. Several months and several versions later, we arrived at release 1.0, which is the occasion of this document's creation.

Funding for this work... would be nice.