CMSC 23700 **Introduction to Computer Graphics** **Handout 3**
Fall 2005 **October 18**

The GML Specification

1 The GML language

The ray tracer in Project-1 takes as input a *scene description* (or *model*) written in a functional modeling language called GML. The language has a syntax and execution model that is similar to PostScript (and Forth), but GML is *lexically* scoped and does not have side effects. This document specifies the syntax and semantics of GML.

1.1 Syntax

A GML program is written using a subset of the printable ASCII character set (including space), plus tab, return, linefeed and vertical tab characters. The space, tab, return, linefeed and vertical tab characters are called *whitespace*.

The characters `%`, `[`, `]`, {, } are *special* characters.

Any occurrence of the character "`%`" not inside a string literal (see below) starts a comment, which runs to the end of the current line. Comments are treated as whitespace when tokenizing the input file.

The syntax of GML is given in Figure 1 (an *opt* superscript means an optional item and a * superscript means a sequence of zero or more items). A GML program is a *token list*, which is a sequence of zero or more *token groups*. A token group is either a single token, a *function* (a token list enclosed in '{' '}'), or an *array* (a token list enclosed in '`[`' '`]`'). Tokens do not have to be separated by white space when it is unambiguous. Whitespace is not allowed in numbers, identifiers, or binders.

Identifiers must start with an letter and can contain letters, digits, dashes ('`-`'), and underscores ('_'). A subset of the identifiers are used as predefined *operators*, which may not be rebound. A list of the operators can be found in the appendix. A binder is an identifier prefixed with a '`/`' character.

Numbers are either integers or reals. The syntax of numbers is given by the following grammar:

Number
::= *Integer*
| *Real*

Integer
::= `-`^{opt} DecimalNumber

TokenList
::= *TokenGroup**

TokenGroup
::= *Token*
| { *TokenList* }
| **[** *TokenList* **]**

Token
::= Operator
| Identifier
| Binder
| *Number*
| String

Figure 1: GML grammar

Real
::= **–**^{opt} DecimalNumber **.** *DecimalNumber Exponent*^{opt}
| **–**^{opt} DecimalNumber *Exponent*

Exponent
::= **e** **–**^{opt} DecimalNumber
| **E** **–**^{opt} DecimalNumber

where a DecimalNumber is a sequence of one or more decimal digits. Integers represented by 32-bit 2's complement values and reals by 64-bit IEEE floating-point values.

Strings are written enclosed in double quotes (‘**"**’) and may contain any printable character other than the double quote (but including the space character). There are no escape sequences.

1.2 Evaluation

We define the evaluation semantics of a GML program using an abstract machine. The state of the machine is a triple $\langle \Gamma;\ \alpha;\ c \rangle$, where Γ is an environment mapping identifiers to values, α is a stack of values, and c is a sequence of token groups. More formally, we use the following semantic definitions:

$$
\begin{array}{rcl}
i & \in & \text{Int} \\
\iota & \in & \text{BaseValue} = \text{Bool} \cup \text{Int} \cup \text{Real} \cup \text{String} \\
v & \in & \text{Value} = \text{BaseValue} \cup \text{Closure} \cup \text{Array} \cup \text{Point} \cup \text{Object} \cup \text{Light} \\
(\Gamma, c) & \in & \text{Closure} = \text{Env} \times \text{Code} \\
a, [v_1\ \ldots\ v_n] & \in & \text{Array} = \text{Value}^* \\
\Gamma & \in & \text{Env} = \text{Id} \xrightarrow{\text{fin}} \text{Value} \\
\alpha, \beta & \in & \text{Stack} = \text{Value}^* \\
c & \in & \text{Code} = \textit{TokenList}
\end{array}
$$

$$\langle \Gamma;\ \alpha;\ \iota\ c\rangle \Longrightarrow \langle \Gamma;\ \alpha\,\iota;\ c\rangle \tag{1}$$

$$\langle \Gamma;\ \alpha\ v;\ /x\ c\rangle \Longrightarrow \langle \Gamma \pm xv;\ \alpha;\ c\rangle \tag{2}$$

$$\langle \Gamma;\ \alpha;\ x\ c\rangle \Longrightarrow \langle \Gamma;\ \alpha\,\Gamma(x);\ c\rangle \tag{3}$$

$$\langle \Gamma;\ \alpha;\ \{c'\}\ c\rangle \Longrightarrow \langle \Gamma;\ \alpha\,(\Gamma, c');\ c\rangle \tag{4}$$

$$\frac{\langle \Gamma';\ \alpha;\ c'\rangle \Longrightarrow^* \langle \Gamma'';\ \beta;\ \varepsilon\rangle}{\langle \Gamma;\ \alpha\,(\Gamma', c');\ \texttt{apply}\ c\rangle \Longrightarrow \langle \Gamma;\ \beta;\ c\rangle} \tag{5}$$

$$\frac{\langle \Gamma;\ \varepsilon;\ c'\rangle \Longrightarrow^* \langle \Gamma';\ v_1\ \ldots\ v_n;\ \varepsilon\rangle}{\langle \Gamma;\ \alpha;\ [c']\ c\rangle \Longrightarrow \langle \Gamma;\ \alpha\,[v_1\ \ldots\ v_n];\ c\rangle} \tag{6}$$

$$\frac{\langle \Gamma_1;\ \alpha;\ c_1\rangle \Longrightarrow^* \langle \Gamma'';\ \beta;\ \varepsilon\rangle}{\langle \Gamma;\ \alpha\,\mathbf{true}\,(\Gamma_1, c_1)\,(\Gamma_2, c_2);\ \texttt{if}\ c\rangle \Longrightarrow \langle \Gamma;\ \beta;\ c\rangle} \tag{7}$$

$$\frac{\langle \Gamma_2;\ \alpha;\ c_2\rangle \Longrightarrow^* \langle \Gamma'';\ \beta;\ \varepsilon\rangle}{\langle \Gamma;\ \alpha\,\mathbf{false}\,(\Gamma_1, c_1)\,(\Gamma_2, c_2);\ \texttt{if}\ c\rangle \Longrightarrow \langle \Gamma;\ \beta;\ c\rangle} \tag{8}$$

$$\frac{\alpha\quad \texttt{OPERATOR}\quad \alpha'}{\langle \Gamma;\ \beta\,\alpha;\ \texttt{OPERATOR}\,c\rangle \Longrightarrow \langle \Gamma;\ \beta\,\alpha';\ c\rangle} \tag{9}$$

Figure 2: Evaluation rules for GML

Evaluation from one state to another is written as $\langle \Gamma;\ \alpha;\ c\rangle \Longrightarrow \langle \Gamma';\ \alpha';\ c'\rangle$. We define $\Longrightarrow^*$ to be the transitive closure of $\Longrightarrow$. Figure 2 gives the GML evaluation rules. In these rules, we write stacks with the top to the right (*e.g.*; $\alpha\ x$ is a stack with x as its top element) and token sequences are written with the first token on the left. We use ε to signify the empty stack and the empty code sequence.

Rule 1 (Section 1) describes the evaluation of a literal token, which is pushed on the stack. The next two rules describe the semantics of variable binding and reference. Rules 4 (Section 4) and 5 (Section 5) describe function-closure creation and the `apply` operator. Rule 6 (Section 6) describes the evaluation of an array expression; note that body of the array expression is evaluated on an initially empty stack. The semantics of the `if` operator are given by Rules 7 (Section 7) and 8 (Section 8). The last evaluation rule (Rule 9 (Section 9)) describes how an operator (other than one of the control operators) is evaluated. We write

$$\alpha\quad \texttt{OPERATOR}\quad \alpha'$$

to mean that the operator `OPERATOR` transforms the stack α to the stack α'. This notation is used below to specify the GML operators.

We write $\mathrm{Eval}(c, v_1, \ldots, v_n) = (v'_1, \ldots, v'_n)$ for when a program c yields $(v'_1, \ldots, v'_n)$ when applied to the values $v_1, \ldots, v_n$; *i.e.*, when $\langle \{\};\ v_1\ \cdots\ v_n;\ c\rangle \Longrightarrow^* \langle \Gamma;\ v'_1\ \cdots, v'_n;\ \varepsilon\rangle$.

There is no direct support for recursion in GML, but one can program recursive functions by

explicitly passing the function as an extra argument to itself (see Section 1.8 for an example).

1.3 Control operators

GML contains two *control* operators that can be used to implement control structures. These operators are formally defined in Figure 2, but we provide an informal description here.

The `apply` operator takes a function closure, (Γ, c), off the stack and evaluates c using the environment Γ and the current stack. When evaluation of c is complete (*i.e.*, there are no more instructions left), the previous environment is restored and execution continues with the instruction after the `apply`. Argument and result passing is done via the stack. For example:

```
1 { /x x x } apply addi
```

will evaluate to 2. Note that functions bind their variables according to the environment where they are defined; not where they are applied. For example the following code evaluates to 3:

```
1 /x          % bind x to 1
{ x } /f      % the function f pushes the value of x
2 /x          % rebind x to 2
f apply x addi
```

The `if` operator takes two closures and a boolean off the stack and evaluates the first closure if the boolean is **true**, and the second if the boolean is **false**. For example,

```
b { 1 } { 2 } if
```

will result in 1 on the top of the stack if b is **true**, and 2 if it is **false**

1.4 Booleans

GML supports the standard boolean type with three operators:

ε **`false`** **false**
: pushes the boolean **false** value.

b_1 **`not`** b_2
: computes the negation b_2 of the boolean b_1.

ε **`true`** **true**
: pushes the boolean **true** value.

1.5 Numbers

GML supports both integer and real numbers (which are represented by IEEE double-precision floating-point numbers). Many of the numeric operators have both integer and real versions, so we combine their descriptions in the following:

n_1 n_2 **`addi`**/**`addf`** n_3
: computes the sum n_3 of the numbers n_1 and n_2.

r_1 **acos** r_2
computes the arc cosine r_2 in degrees of r_1. The result is undefined if $r_1 < -1$ or $1 < r_1$.

r_1 **asin** r_2
computes the arc sine r_2 in degrees of r_1. The result is undefined if $r_1 < -1$ or $1 < r_1$.

r_1 **clampf** r_2
computes $r_2 = \begin{cases} 0.0 & r_1 < 0.0 \\ 1.0 & r_1 > 1.0 \\ r_1 & \text{otherwise} \end{cases}$.

r_1 **cos** r_2
computes the cosine r_2 of r_1 in degrees.

n_1 n_2 **divi**/**divf** n_3
computes the quotient n_3 of dividing the number n_1 by n_2. The `divi` operator rounds its result towards 0. For the `divi` operator, if n_2 is zero, then the program halts. For `divf`, the effect of division by zero is undefined.

n_1 n_2 **eqi**/**eqf** b
compares the numbers n_1 and n_2 and pushes **true** if n_1 is equal to n_2; otherwise **false** is pushed.

r **floor** i
converts the real r to the greatest integer i that is less than or equal to r.

r_1 **frac** r_2
computes the fractional part r_2 of the real number r_1. The result r_2 will always have the same sign as the argument r_1.

n_1 n_2 **lessi**/**lessf** b
compares the numbers n_1 and n_2 and pushes **true** if n_1 is less than n_2; otherwise **false** is pushed.

i_1 i_2 **modi** i_3
computes the remainder i_3 of dividing i_1 by i_2. The following relation holds between `divi` and `modi`:

$$i2(i1 \texttt{ divi } i2) + (i1 \texttt{ mod } i2) = i1$$

n_1 n_2 **muli**/**mulf** n_3
computes the product n_3 of the numbers n_1 and n_2.

n_1 **negi**/**negf** n_2
computes the negation n_2 of the number n_1.

i **real** r
converts the integer i to its real representation r.

r_1 **sin** r_2
computes the sine r_2 of r_1 in degrees.

r_1 **sqrt** r_2
computes the square root r_2 of r_1. If r_1 is negative, then the interpreter should halt.

$n_1\ n_2$ **subi/subf** n_3
computes the difference n_3 of subtracting the number n_2 from n_1.

1.6 Points

A *point* is comprised of three real numbers. Points are used to represent positions, vectors, and colors (in the latter case, the range of the components is restricted to $[0.0, 1.0]$). There are four operations on points:

p **getx** x
gets the first component x of the point p.

p **gety** y
gets the second component y of the point p.

p **getz** z
gets the third component z of the point p.

$x\ y\ z$ **point** p
creates a point p from the reals x, y, and z.

1.7 Arrays

There are two operations on arrays:

$arr\ i$ **get** v_i
gets the ith element of the array arr. Array indexing is zero based in GML. If i is out of bounds, the GML interpreter should terminate.

arr **length** n
gets the number of elements in the array arr.

The elements of an array do not have to have the same type and arrays can be used to construct data structures. For example, we can implement lists using two-element arrays for cons cells and the zero-length array for nil.

```
[] /nil
{ /cdr /car [ car cdr ] } /cons
```

We can also write a function that "*pattern matches*" on the head of a list.

```
{ /if-cons /if-nil /lst
  lst length 0 eqi
  if-nil
  { lst 0 get lst 1 get if-cons apply }
  if
}
```

1.8 Examples

Some simple function definitions written in GML:

```
{ } /id                              % the identity function
{ 1 addi } /inc                      % the increment function
{ /x /y x y } /swap                  % swap the top two stack locations
{ /x x x } /dup                      % duplicate the top of the stack
{ dup apply muli } /sq               % the squaring function
{ /a /b a { true } { b } if } /or    % logical-or function
{ /p                                 % negate a point value
  p getx negf
  p gety negf
  p getz negf point
} /negp
```

A more substantial example is the GML version of the recursive factorial function:

```
{ /self /n
  n 2 lessi
  { 1 }
  { n 1 subi self self apply n muli }
  if
} /fact
```

Notice that this function follows the convention of passing itself as the top-most argument on the stack. We can compute the factorial of `12` with the expression

```
12 fact fact apply
```

2 Ray tracing operations

In this section, we describe how the GML interpreter supports ray tracing.

2.1 Coordinate systems

GML models are defined in terms of two coordinate systems: *world coordinates* and *object coordinates*. World coordinates are used to specify the position of lights while object coordinates are used to specify primitive objects. There are six *transformation* operators (described in Section 2.3) that are used to map object space to world space.

The world-coordinate system is *left-handed*. The X-axis goes to the right, the Y-axis goes up, and the Z-axis goes away from the viewer.

2.2 Geometric primitives

There are five operations in GML for constructing primitive solids: `sphere`, `cube`, `cylinder`, `cone`, and `plane`. Each of these operations takes a single function as an argument, which defines the primitive's surface properties (see Section 2.6).

surface **`sphere`** *obj*
: creates a sphere of radius 1 centered at the origin with surface properties specified by the

function *surface*. Formally, the sphere is defined by $x^2 + y^2 + z^2 \leq 1$.

surface **cube** *obj*
: creates a unit cube with opposite corners $(0, 0, 0)$ and $(1, 1, 1)$. The function *surface* specifies the cube's surface properties. Formally, the cube is defined by $0 \leq x \leq 1$, $0 \leq y \leq 1$, and $0 \leq z \leq 1$.

surface **cylinder** *obj*
: creates a cylinder of radius 1 and height 1 with surface properties specified by the function *surface*. The base of the cylinder is centered at $(0, 0, 0)$ and the top is centered at $(0, 1, 0)$ (*i.e.*, the axis of the cylinder is the Y-axis). Formally, the cylinder is defined by $x^2 + z^2 \leq 1$ and $0 \leq y \leq 1$.

surface **cone** *obj*
: creates a cone with base radius 1 and height 1 with surface properties specified by the function *surface*. The apex of the cone is at $(0, 0, 0)$ and the base of the cone is centered at $(0, 1, 0)$. Formally, the cone is defined by $x^2 + z^2 - y^2 \leq 0$ and $0 \leq y \leq 1$.

surface **plane** *obj*
: creates a plane object with the equation $y = 0$ with surface properties specified by the function *surface*. Formally, the plane is the half-space $y \leq 0$.

2.3 Transformations

Fixed size objects at the origin are not very interesting, so GML provides *transformation* operations to place objects in world space. Each transformation operator takes an object and one or more reals as arguments and returns the transformed object. The operations are:

obj r_{tx} r_{ty} r_{tz} **translate** *obj*$'$
: translates *obj* by the vector (r_{tx}, r_{ty}, r_{tz}). I.e., if *obj* is at position (p_x, p_y, p_z), then *obj*$'$ is at position $(p_x + r_{tx}, p_y + r_{ty}, p_z + r_{tz})$.

obj r_{sx} r_{sy} r_{sz} **scale** *obj*$'$
: scales *obj* by r_{sx} in the X-dimension, r_{sy} in the Y-dimension, and r_{sz} in the Z dimension.

obj r_s **uscale** *obj*$'$
: uniformly scales *obj* by r_s in each dimension. This operation is called *Isotropic scaling*.

obj θ **rotatex** *obj*$'$
: rotates *obj* around the X-axis by θ degrees. Rotation is measured counterclockwise when looking along the X-axis from the origin towards $+\infty$.

obj θ **rotatey** *obj*$'$
: rotates *obj* around the Y-axis by θ degrees. Rotation is measured counterclockwise when looking along the Y-axis from the origin towards $+\infty$.

obj θ **rotatez** *obj*$'$
: rotates *obj* around the Z-axis by θ degrees. Rotation is measured counterclockwise when looking along the Z-axis from the origin towards $+\infty$.

$$\begin{bmatrix} 1 & 0 & 0 & r_{tx} \\ 0 & 1 & 0 & r_{ty} \\ 0 & 0 & 1 & r_{tz} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$
Translation

$$\begin{bmatrix} r_{sx} & 0 & 0 & 0 \\ 0 & r_{sy} & 0 & 0 \\ 0 & 0 & r_{sz} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$
Scale matrix

$$\begin{bmatrix} r_s & 0 & 0 & 0 \\ 0 & r_s & 0 & 0 \\ 0 & 0 & r_s & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$
Isotropic scale matrix

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\theta) & -\sin(\theta) & 0 \\ 0 & \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$
Rotation (X-axis)

$$\begin{bmatrix} \cos(\theta) & 0 & \sin(\theta) & 0 \\ 0 & 1 & 0 & 0 \\ -\sin(\theta) & 0 & \cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$
Rotation (Y-axis)

$$\begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 & 0 \\ \sin(\theta) & \cos(\theta) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$
Rotation (Z-axis)

Figure 3: Transformation matrices

For example, if we want to put a sphere of radius 2.0 at $(5.0, 5.0, 5.0)$, we can use the following GML code:

```
{ ... } sphere
2.0 uscale
5.0 5.0 5.0 translate
```

The first line creates the sphere (as described in Section 2.2, the `sphere` operator takes a single function argument). The second line uniformly scales the sphere by a factor of 2.0, and the third line translates the sphere to $(5.0, 5.0, 5.0)$.

These transformations are all *affine* transformations and they have the property of preserving the straightness of lines and parallelism between lines, but they can alter the distance between points and the angle between lines. Using *homogeneous coordinates*, these transformations can be expressed as multiplication by a 4×4 matrix. Figure 3 describes the matrices that correspond to each of the transformation operators. For example, translating the point $(2.6, 3.0, -5.0)$ by $(-1.6, -2.0, 6.0)$ is expressed as the following multiplication:

$$\begin{bmatrix} 1.0 & 0.0 & 0.0 & -1.6 \\ 0.0 & 1.0 & 0.0 & -2.0 \\ 0.0 & 0.0 & 1.0 & 6.0 \\ 0.0 & 0.0 & 0.0 & 1.0 \end{bmatrix} \begin{bmatrix} 2.6 \\ 3.0 \\ -5.0 \\ 1.0 \end{bmatrix} = \begin{bmatrix} 1.0 \\ 1.0 \\ 1.0 \\ 1.0 \end{bmatrix}$$

Observe that points have a fourth coordinate of 1, whereas vectors have a fourth coordinate of 0. Thus, translation has no effect on vectors.

2.4 Illumination model

When the ray that shoots from the eye position through a pixel hits a surface, we need to apply the illumination equation to determine what color the pixel should have. Figure 4 shows a situation where a ray from the viewer has hit a surface. The illumination at this point is given by the following equation:

$$I = k_a I_a C + k_d \sum_{j=1}^{ls} \max(\mathbf{N} \cdot \mathbf{L}_j, 0) I_j C + k_s \sum_{j=1}^{ls} (\mathbf{N} \cdot \mathbf{H}_j)^n I_j + k_s I_s \tag{10}$$

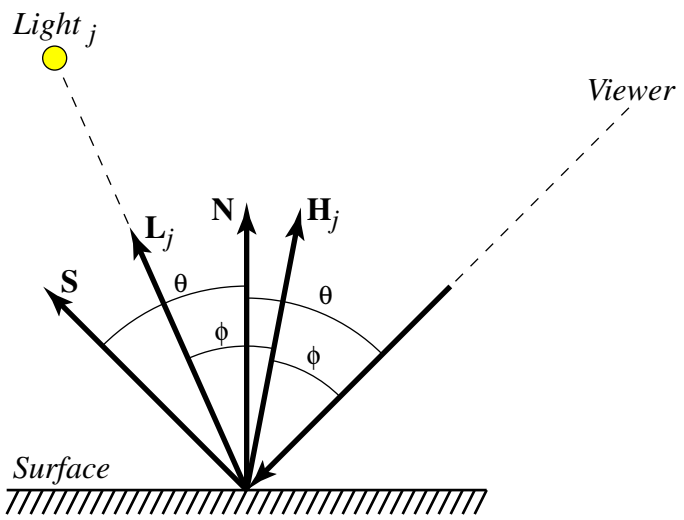

Figure 4: A ray intersecting a surface

where

k_a	=	ambient reflection coefficient
I_a	=	intensity of ambient lighting
C	=	surface color
k_d	=	diffuse reflection coefficient
$\mathbf{N}$	=	unit surface normal
$\mathbf{L}_j$	=	unit vector in direction of jth light source
I_j	=	intensity of jth light source
k_s	=	specular reflection coefficient
$\mathbf{H}_j$	=	unit vector in the direction halfway between the viewer and $\mathbf{L}_j$
n	=	Phong exponent
$\mathbf{S}$	=	reflection vector
I_s	=	intensity of light from direction $\mathbf{S}$

The four components of this equation correspond to the local ambient lighting, the local diffuse lighting, the local specular lighting, and the global reflection (resp.). The view vector, $\mathbf{N}$, and $\mathbf{S}$ all lie in the same plane. The vector $\mathbf{S}$ is called the *reflection* vector and forms the same angle with $\mathbf{N}$ as the vector to the viewer does (this angle is labeled θ in Figure 4). Light intensity is represented as point in GML and multiplication of points is component wise. The values of C, k_a, k_d, k_s, and n are the *surface properties* of the object at the point of reflection. Section 2.6 describes the mechanism for specifying these values for an object.

Computing the contribution of lights (the I_j part of the above equation) requires casting a *shadow ray* from the intersection point to the light's position. If the ray hits an object that is closer than the light, then the light does not contribute to the illumination of the intersection point.

Ray tracing is a recursive process. Computing the value of I_s requires shooting a ray in the direction of S and seeing what object (if any) it intersects. To avoid infinite recursion, we limit the tracing to some *depth*. The depth limit is given as an argument to the `render` operator (see Section 2.8).

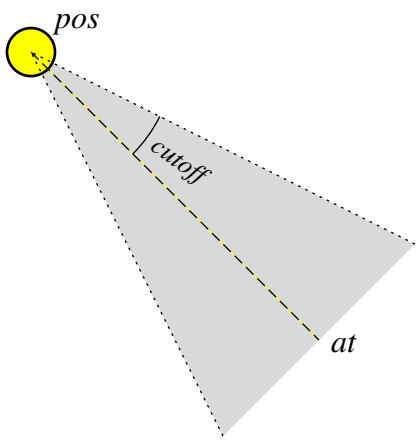

Figure 5: Spotlight

2.5 Lights

GML supports three types of light sources: *directional lights*, *point lights* and *spotlights*. Directional lights are assumed to be infinitely far away and have only a direction. Point lights have a position and an intensity (specified as a color triple), and they emit light uniformly in all directions. Spotlights emit a cone of light in a given direction. The light cone is specified by three parameters: the light's direction, the light's cutoff angle, and an attenuation exponent (see Figure 5). Unlike geometric objects, lights are defined in terms of world coordinates.

dir color **`light`** *l*
: creates a directional light source at infinity with direction dir and intensity $color$. Both dir and $color$ are specified as point values.

pos color **`pointlight`** *l*
: creates a point-light source at the world coordinate position pos with intensity $color$. Both pos and $color$ are specified as point values.

pos at color cutoff exp **`spotlight`** *l*
: creates a spotlight source at the world coordinate position pos pointing towards the position at. The light's color is given by $color$. The spotlight's cutoff angle is given in degrees by $cutoff$ and the attenuation exponent is given by exp (these are real numbers). The intensity of the light from a spotlight at a point Q is determined by the angle between the light's direction vector (*i.e.*, the vector from pos to at) and the vector from pos to Q. If the angle is greater than the cutoff angle, then intensity is zero; otherwise the intensity is given by the equation

$$I = \left(\frac{at - pos}{|at - pos|} \cdot \frac{Q - pos}{|Q - pos|} \right)^{exp} color \tag{11}$$

The light from point lights and spotlights is attenuated by the distance from the light to the surface. The attenuation equation is:

$$I_{surface} = \frac{I}{a_0 + a_1 d + a_2 d^2} \tag{12}$$

Table 1: Texture coordinates for primitives

	SPHERE	
$(0, u, v)$	$(\sqrt{1-y^2}\sin(360u), y, \sqrt{1-y^2}\cos(360u))$,	where $y = 2v - 1$
	CUBE	
$(0, u, v)$	$(u, v, 0)$	front
$(1, u, v)$	$(u, v, 1)$	back
$(2, u, v)$	$(0, v, u)$	left
$(3, u, v)$	$(1, v, u)$	right
$(4, u, v)$	$(u, 1, v)$	top
$(5, u, v)$	$(u, 0, v)$	bottom
	CYLINDER	
$(0, u, v)$	$(\sin(360u), v, \cos(360u))$	side
$(1, u, v)$	$(2u - 1, 1, 2v - 1)$	top
$(2, u, v)$	$(2u - 1, 0, 2v - 1)$	bottom
	CONE	
$(0, u, v)$	$(v\sin(360u), v, v\cos(360u))$	side
$(1, u, v)$	$(2u - 1, 1, 2v - 1)$	base
	PLANE	
$(0, u, v)$	$(u, 0, v)$	

where I is the intensity of the light, d is the distance from the light to the surface, and the a_i are the attenuation coefficients given to the `render` command (see Section 2.8). Note that the light reflected from surfaces (the $k_s I_s$ term in Equation 10) is *not* attenuated; nor is the light from directional sources.

2.6 Surface functions

GML uses *procedural texturing* to describe the surface properties of objects. The basic idea is that the model provides a function for each object, which maps positions on the object to the surface properties that determine how the object is illuminated (see Section 2.4).

A surface function takes three arguments: an integer specifying an object's face and two texture coordinates. For all objects, except planes, the texture coordinates are restricted to the range $0 \leq u, v \leq 1$. The Table 1 specifies how these coordinates map to points in object-space for the various builtin graphical objects. Note that the arguments to the $\sin$ and $\cos$ functions are in degrees. The GML implementation is responsible for the inverse mapping; *i.e.*, given a point on a solid, compute the texture coordinates.

A surface function returns a point representing the surface color (C), and four real numbers: the ambient reflection coefficient (k_a), the diffuse reflection coefficient (k_d), the specular reflection coefficient (k_s), and the Phong exponent (n). For example, the code in Figure 6 defines a cube with a matte 3×3 black and white checked pattern on each face.

```
0.0 0.0 0.0 point /black
1.0 1.0 1.0 point /white

[                                   % 3x3 pattern
  [ black white black ]
  [ white black white ]
  [ black white black ]
] /texture

{ /v /u /face                       % bind parameters
  {                                 % toIntCoord : float -> int
    3.0 mulf floor /i                 % i = floor(3.0*r)
    i 3 eqi { 2 } { i } if             % make sure i is not 3
  } /toIntCoord
  texture u toIntCoord apply get    % color = texture[u][v]
    v toIntCoord apply get
  1.0                               % ka = 1.0
  1.0                               % kd = 1.0
  0.0                               % ks = 0.0
  1.0                               % n = 1.0
} cube
```

Figure 6: A checked pattern on a cube

2.7 Constructive solid geometry

Solid objects may be combined using boolean set operations to form more complex solids. There are three composition operations:

obj_1 obj_2 **`union`** obj_3
: forms the union obj_3 of the two solids obj_1 and obj_2.

obj_1 obj_2 **`intersect`** obj_3
: forms the intersection obj_3 of the two solids obj_1 and obj_2.

obj_1 obj_2 **`difference`** obj_3
: forms the solid obj_3 that is the solid obj_1 minus the solid obj_2.

We can determine the intersection of a ray and a compound solid by recursively computing the intersections of the ray and the solid's pieces (both entries and exits) and then merging the information according to the boolean composition operator. Figure 7 illustrates this process for two objects (this picture is called a *Roth diagram*).

When rendering a composite object, the surface properties are determined by the primitive that defines the surface. If the surfaces of two primitives coincide, then which primitive defines the surface properties is unspecified.

2.8 Rendering

The `render` operator causes the scene to be rendered to a file.

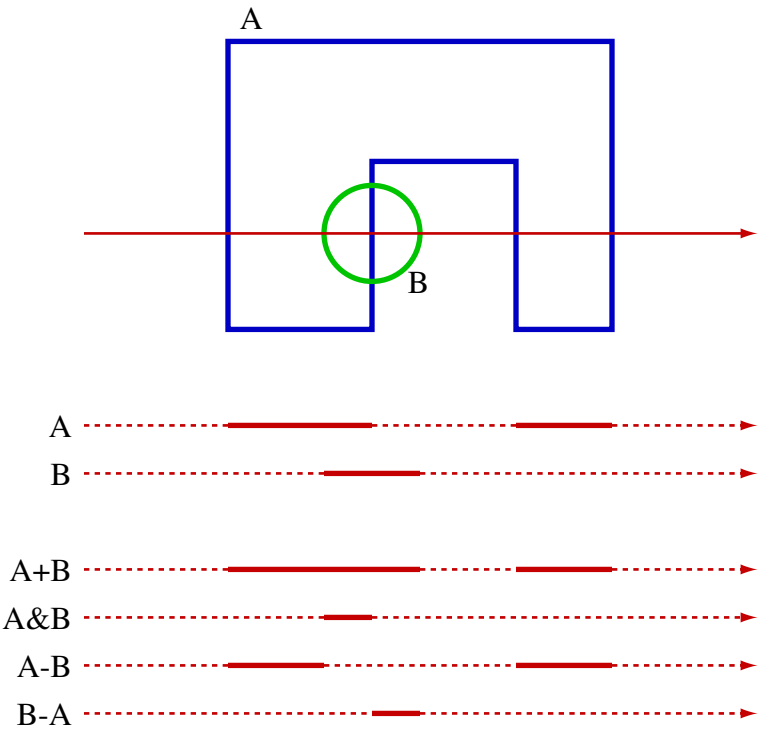

Figure 7: Tracing a ray through a compound solid

amb lights attn obj depth fov wid ht file **`render`** —

The render operator renders a scene to a file. It takes nine arguments:

amb the intensity of ambient light (a point).

lights is an array of lights used to illuminate the scene.

attn is a point that represents the light-attenuation coefficients (see Section 2.5), with the a_0 coefficient corresponding to the x component of *attn*, a_1 corresponding to the y component, and a_1 corresponding to the z component.

obj is the scene to render.

depth is an integer limit on the recursive depth of the ray tracing owing to specular reflection. *I.e.*, when $depth = 0$, we do not recursively compute the contribution from the direction of reflection (**S** in Figure 4).

fov is the horizontal field of view in degrees (a real number).

wid is the width of the rendered image in pixels (an integer).

ht is the height of the rendered image in pixels (an integer).

file is a string specifying output file for the rendered image.

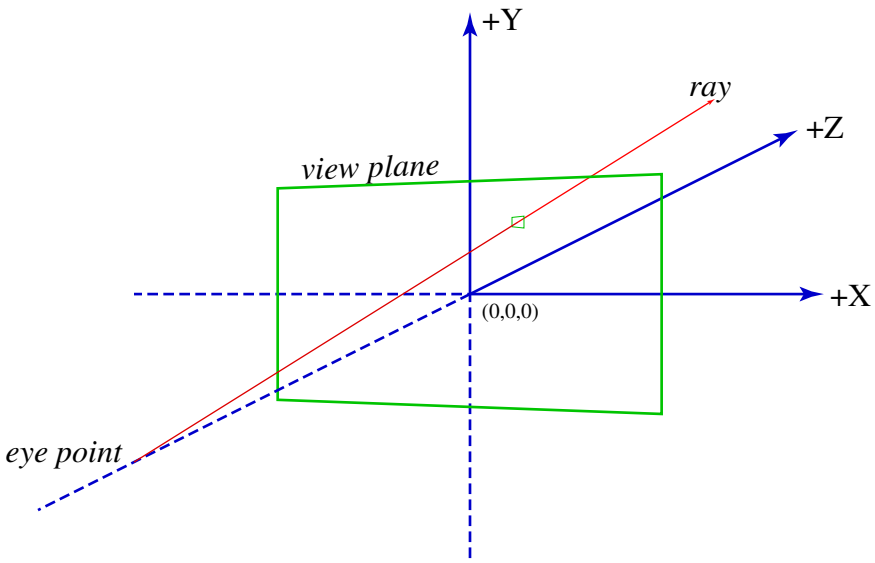

Figure 8: View coordinate system

The `render` operator is the only GML operator with side effects (*i.e.*, it modifies the host file system). A GML program may contain multiple `render` operators (for animation effects).

When rendering a scene, the eye position is fixed at $(0, 0, -1)$ looking down the Z-axis and the image plane is the XY-plane (see Figure 8). The horizontal field of view (fov) determines the width of the image in world space (*i.e.*, it is $2\tan(0.5fov)$), and the height is determined from the aspect ratio. If the upper-left corner of the image is at $(x, y, 0)$ and the width of a pixel is Δ, then the ray through the jth pixel in the ith row has a direction of $(x + (j + 0.5)\Delta, y - (i + 0.5)\Delta, 1)$. Objects that lie between the view plane and the eye position should not be rendered. One easy way to achieve this effect is to place the origin of the ray at the view plane (instead of the eye).

When the render operation detects that a ray has intersected the surface of an object, it must compute the texture coordinates at the point of intersection and apply the surface function to them. Let $(face, u, v)$ be the texture coordinates and $surf$ be the surface function at the point of intersection, and let

$$\text{Eval}(surf\ \texttt{apply}, face, u, v) = (C, k_a, k_d, k_s, n)$$

Then the surface properties for the illumination equation (see Section 2.4) are C, k_d, k_s, and n.

2.9 The output format

The output format is the *Portable Pixmap* (PPM) file format.[1] The format consists of a ASCII header followed by the pixel data in binary form. The format of the header is

- The magic number, which are the two characters "`P6`."
- A width, formatted as ASCII characters in decimal.

[1] On Linux systems, the **xv** program can be used to view these files and on MacOS X you can use the **GraphicsConverter** application.

- A height, again in ASCII decimal.
- The ASCII text "`255`," which is the maximum color-component value.

These items are separated by whitespace (blanks, TABs, CRs, and LFs). After the maximum color value, there is a single whitespace character (usually a newline), which is followed by the pixel data. The pixel data is a sequence of three-byte pixel values (red, green, blue) in row-major order. Light intensity values (represented as GML points) are converted to RGB format by clamping the range and scaling.

In the header, characters from a "`#`" to the next end-of-line are ignored (comments). This comment mechanism should be used to include the group's name immediately following the line with the magic number. For example, the sample implementation produces the following header:

```
P6
# GML Sample Implementation
256 256
255
```

Operator summary

The following is an alphabetical listing of the GML operators with brief descriptions. The third column lists the section where the operator is defined and the fourth column specifies whether the operator is provided by by us are to be implemented by you.

Name	Description	Section	Provided?
`acos`	arc cosine function	1.5	Yes
`addi`	integer addition	1.5	Yes
`addf`	real addition	1.5	Yes
`apply`	function application operator	1.3	Yes
`asin`	arc sine function	1.5	Yes
`clampf`	clamp the range of a real number	1.5	Yes
`cone`	a unit cone	2.2	No
`cos`	cosine function	1.5	Yes
`cube`	a unit cube	2.2	No
`cylinder`	a unit cylinder	2.2	No
`difference`	difference of two solids	2.7	No
`divi`	integer division	1.5	Yes
`divf`	real division	1.5	Yes
`eqi`	integer equality comparison	1.5	Yes
`eqf`	real equality comparison	1.5	Yes
`false`	push the **false** value	1.4	Yes
`floor`	real to integer conversion	1.5	Yes
`frac`	fractional part of real number	1.5	Yes
`get`	get an array element	1.7	Yes
`getx`	get x component of point	1.6	Yes
`gety`	get y component of point	1.6	Yes
`getz`	get z component of point	1.6	Yes
`if`	conditional control operator	1.3	Yes

Name	Description	Section	Provided?
`intersect`	intersection of two solids	2.7	No
`length`	array length	1.7	Yes
`lessi`	integer less-than comparison	1.5	Yes
`lessf`	real less-than comparison	1.5	Yes
`light`	defines a directional light source	2.5	Yes
`modi`	integer remainder	1.5	Yes
`muli`	integer multiplication	1.5	Yes
`mulf`	real multiplication	1.5	Yes
`negi`	integer negation	1.5	Yes
`negf`	real negation	1.5	Yes
`not`	boolean negation	1.4	Yes
`plane`	the XZ-plane	2.2	No
`point`	create a point value	1.6	Yes
`pointlight`	defines a point-light source	2.5	Yes
`real`	convert an integer to a real number	1.5	Yes
`render`	render a scene to a file	2.8	No
`rotatex`	rotation around the X-axis	2.3	No
`rotatey`	rotation around the Y-axis	2.3	No
`rotatez`	rotation around the Z-axis	2.3	No
`scale`	scaling transform	2.3	No
`sin`	sine function	1.5	Yes
`sphere`	a unit sphere	2.2	No
`spotlight`	defines a spotlight source	2.5	Yes
`sqrt`	square root	1.5	Yes
`subi`	integer subtraction	1.5	Yes
`subf`	real subtraction	1.5	Yes
`translate`	translation transform	2.3	No
`true`	push the **true** value	1.4	Yes
`union`	union of two solids	2.7	No
`uscale`	uniform scaling transform	2.3	No