CMSC 23700 **Introduction to Computer Graphics** **Project 1**
Fall 2003 **October 14**

Ray tracing

Due: Friday, October 31

1 The problem

The second project is to implement a ray tracer. The input to the ray tracer is 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. Execution of a GML program produces zero, or more, *image files*, which are in PPM format (Section 3.7). A detailed description of GML is given in separate handout.

GML has primitives for defining simple geometric objects (*e.g.*, planes, spheres, and cubes) and lighting sources. The surface properties used to render the objects are specified as functions in GML itself. In addition to supporting scene description, GML also has a `render` operator that renders a scene to an image file. For each pixel in the output image, the `render` command must compute a color. Conceptually, this color is computed by tracing the path of the light backwards from the eye of the viewer, to where it bounced off an object, and ultimately back to the light sources.

We will provide the implementation of GML; it is your task to implement a collection of 16 C functions that make up the modelling and rendering parts of the ray tracer.

This document is organized as follows. Section 2 gives a quick introduction to the GML modeling language. It is followed by Section 3, which describes your task. Section 4 specifies the submission requirements and Section 5 provides hints about algorithms and pointers to online resources to get you started.

2 A quick introduction to GML

The "*GML Specification*" gives a detailed description of the syntax and semantics of the modeling language. For this handout, we limit ourselves to a quick introduction to GML.

GML is a dynmically-typed, postfix stack-based language. Operations take their arguments from the stack and deposit their results on the stack. GML supports a small collection of different value types: *booleans*, *integers*, *reals*, *strings*, *points*, *closures* (*i.e.*, functions), *lights*, *geometrical objects*, and *arrays* of values.

GML supports binding values to variables. For example, the following code pushes the real value `3.1415` on to the stack and then pops it and binds it to the variable `pi`.

```
3.1415 /pi
```

We can then use this value as in the following code that defines `twopi`:

```
2.0 pi mulf /twopi
```

The syntax "`/a`" binds the variable `a` to a value poped off the stack, while the syntax "`a`" pushes the value of the variable.

GML functions are anonymous and are defined by enclosing code in "{ ... }." For example, a function that adds one to its argument is defined by

```
{ 1 addi } /inc
```

Functions can bind local variables as in the following example that duplicates the top of the stack:

```
{ /x x x } /dup
```

This function works by binding the top of the stack to the local variable `x` and then pushing the value of `x` twice. Functions are first-class values, so to apply them we must use the `apply` operator:

```
{ dup apply muli } /sq
```

The other control operator is the `if` operator that takes a boolean and two functions as arguments. Here is the definition of *logical or* in GML.

```
{ /a /b a { true } { b } if } /or
```

GML has no looping construct, but recursion can be used to implement loops. To write and apply a recursive function requires explicitly tying the recursive knot. For example, the GML version of the recursive factorial function is as follows:

```
{ /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
```

3 The task

In this section, we describe the functions that implement the rendering aspects of the GML interpreter. We group these functions into five classes: geometric primitives, transformations, lighting, CSG operators, and the render command. An implementation of the lighting functions is provided in the sample code, you are responsible for the rest. We have supplied a file `objects.c` that contains stubs for the functions that you are responsible for.

3.1 GML graphics operations

There are 18 GML graphics operators. The following table summarizes them with cross references to the section where the corresponding interpreter functions are described:

Name	Description	Section
`cone`	a unit cone	3.3
`cube`	a unit cube	3.3
`cylinder`	a unit cylinder	3.3
`difference`	difference of two solids	3.6
`intersect`	intersection of two solids	3.6
`light`	defines a directional light source	3.5
`plane`	the XZ-plane	3.3
`pointlight`	defines a point-light source	3.5
`render`	render a scene to a file	3.7
`rotatex`	rotation around the X-axis	3.4
`rotatey`	rotation around the Y-axis	3.4
`rotatez`	rotation around the Z-axis	3.4
`scale`	scaling transform	3.4
`sphere`	a unit sphere	3.3
`spotlight`	defines a spotlight source	3.5
`translate`	translation transform	3.4
`union`	union of two solids	3.6
`uscale`	uniform scaling transform	3.4

3.2 Interpreter types

The GML interpreter manipulates a number of different value types, which are defined in the file `gml.h`.

3.3 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 GML function as an argument, which defines the primitive's surface properties. As part of your implementation, you will need to define the representation of objects as a C `struct` type:

```
struct struct_object {
    HEADER
    ...
};
```

The `HEADER` macro expands to some common member definitions that support memory management.

The operations that are used to create the basic primitives are as follows:

`Object_t *coneAct (Closure_t *`*`surf`*`)`

creates a cone with base radius 1 and height 1 and with surface properties specified by the function *`surf`*. 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 $0 \leq y \leq 1$.

`Object_t *cubeAct (Closure_t *`*`surf`*`)`

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

`Object_t *cylinderAct (Closure_t *surf)`
: creates a cylinder of radius 1 and height 1 with surface properties specified by the function *surf*. 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 $0 \leq y \leq 1$.

`Object_t *planeAct (Closure_t *surf)`
: creates a plane object with the equation $y = 0$ with surface properties specified by the function *surf*. Formally, the plane is the half-space $y \leq 0$.

`Object_t *sphereAct (Closure_t *surf)`
: creates a sphere of radius 1 centered at the origin with surface properties specified by the function *surf*. Formally, the sphere is defined by $x^2 + y^2 + z^2 \leq 1$.

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

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 (as always in GML), the arguments to the $\sin$ and $\cos$ functions are in degrees. Your 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 three real numbers: the diffuse reflection coefficient (k_d), the specular reflection coefficient (k_s), and the Phong exponent (n). We provide the following helper function (defined in `gml.h`) for applying a surface function to texture coordinates:

```
extern void EvalSurfaceFn (
    Closure_t *surf,       /* surface function */
    int face,              /* object face */
    Real_t u, Real_t v,    /* texture coordinates */
    Vec3_t color,          /* surface color (output) */
    Real_t *kd,            /* diffuse reflection coeff. (output) */
    Real_t *ks,            /* specular reflection coeff. (output) */
    Real_t *n);            /* Phong exponent (output) */
```

3.4 Transformations

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

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	
$(0,u,v)$	$(u,0,v)$	

`Object_t *rotatexAct (Object_t *`*`obj`*`, Real_t `*`theta`*`)`
: rotates *`obj`* around the X-axis by θ degrees. Rotation is measured counterclockwise when looking along the X-axis from the origin towards $+\infty$.

`Object_t *rotateyAct (Object_t *`*`obj`*`, Real_t `*`theta`*`)`
: rotates *`obj`* around the Y-axis by θ degrees. Rotation is measured counterclockwise when looking along the Y-axis from the origin towards $+\infty$.

`Object_t *rotatezAct (Object_t *`*`obj`*`, Real_t `*`theta`*`)`
: rotates *`obj`* around the Z-axis by θ degrees. Rotation is measured counterclockwise when looking along the Z-axis from the origin towards $+\infty$.

`Object_t *scaleAct (Object_t *`*`obj`*`, Vec3_t `*`s`*`)`
: scales *`obj`* by r_{sx} in the X-dimension, r_{sy} in the Y-dimension, and r_{sz} in the Z dimension.

`Object_t *translateAct (Object_t *`*`obj`*`, Vec3_t `*`t`*`)`
: 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 $(p_x + r_{tx}, p_y + r_{ty}, p_z + r_{tz})$.

`Object_t *uscaleAct (Object_t *`*`obj`*`, Real_t `*`s`*`)`
: uniformly scales *`obj`* by r_s in each dimension. This operation is called *Isotropic scaling*.

3.5 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. Unlike geometric objects, lights are defined in terms of world coordinates. The following C functions are used to implement GML lights:

`Light_t *lightAct (Vec3_t` *`dir`*`, Vec3_t` *`color`*`)`
: creates a directional light source at infinity with direction *`dir`* and intensity *`color`*. Both *`dir`* and *`color`* are specified as point values.

`Light_t *pointlightAct (Vec3_t` *`pos`*`, Vec3_t` *`color`*`)`
: creates a point-light source at the world coordinate position *`pos`* with intensity *`color`*. Both *`pos`* and *`color`* are specified as point values.

`Light_t *spotlightAct (Vec3_t` *`pos`*`, Vec3_t` *`at`*`, Vec3_t` *`color`*`,`
`Real_t` *`cutoff`*`, Real_t` *`exp`*`)`

: 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

$$I = \left(\frac{at - pos}{|at - pos|} \cdot \frac{Q - pos}{|Q - pos|} \right)^{exp} color \tag{1}$$

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

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

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 passed to the `renderAct` command (see Section 3.7).

3.6 Constructive solid geometry

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

`Object_t *differenceAct (Object_t *`*`obj1`*`, Object_t *`*`obj2`*`)`
: returns an object that is the object *`obj1`* minus the solid *`obj2`*.

`Object_t *intersectAct (Object_t *`*`obj1`*`, Object_t *`*`obj2`*`)`
: returns an object that is the intersection of objects *`obj1`* and *`obj2`*.

`Object_t *unionAct (Object_t *obj1, Object_t *obj)`
  returns an object that is the union of objects *obj1* and *obj2*.

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.

3.7 Rendering

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

```
   void renderAct (
       Vec3_t amb,
       int nLights,
       Light_t **lights,
       Vec3_t attn,
       Object_t *obj,
       int depth,
       double fov,
       int wid,
       int ht,
       char *file)
```

The `renderAct` function renders a scene to a file. It takes ten arguments:

amb the intensity of ambient light (a point).

nLights is the number of lights in the *lights* array.

lights is an array of *nLights* pointers to lights used to illuminate the scene.

attn is a vector of light-attenuation coefficients.

obj is the scene to render.

depth is an integer limit on the recursive depth of the ray tracing owing to specular reflection.

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.

The `render` operator is the only GML operator with side effects (*i.e.*, it modifies the host file system).

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`.”

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

- A width, formatted as ASCII characters in decimal.
- 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
```

3.8 Memory management

Most of the operations you must implement will allocate new graphical objects. The GML interpreter uses a reference counting scheme to keep track of when it is safe to free an object. It is your responsibility to follow the reference counting protocol for the objects you create.

In addition, you must implement the following function:

```
void FreeObject (Object_t *obj);
```

This function will need to determine the kind of object pointed to by `obj` and then decrement the reference counts of any other object it refers to. Finally, it should free the memory pointed to by `obj`.

4 Requirements

As with Project 0, we will create a module in your course CVS repository on the Computer Science CVS server. The module is named `project-1` and contains the implementation of a GML interpreter. Your task is to complete this interpreter by adding the implementation of the graphics operations. You should use this module to hold the source for your project. We will collect the projects at 9pm on Friday October 31st from the repositories, so make sure that you have committed your final version before then.

The Makefile in the repository builds a statically-linked library called `rtlib.a`. To build a raytracer from a given GML file, one uses the GML compiler (called **gmlc**) to translate the GML file to a C file that can then be compiled and linked with the `rtlib.a` library. For example, assume we have a GML file `scene.gml`, then the following commands will build a raytracer for the scene specified in the file:

```
% gmlc scene.gml
% cc -o scene.rt scene.c rtlib.a -lm
```

The **gmlc** command is located in `/usr/local/bin` on the Macs and in `/stage/cmsc237/bin` on the Linux machines. Running the `scene.rt` program will have the effect of running the GML program in `scene.gml`.

5 Hints

This section contains some hints that may help you in your implementation. We suggest that you start by implementing the transformations, spheres, union, and ambient lighting. Once these features are working, add the specular lighting components (including the recursive ray tracing). Finally, add the other shapes and CSG operators, testing each as you go.

5.1 Intersection testing

One approach to ray tracing with a modeling language that supports affine transformations (such as GML) is to transform the rays into object space and do the intersection tests there. This approach allows the intersection tests to be specialized to the standard objects, which can greatly simplify the tests. Remember, however, that affine transformations do not preserve lengths — applying an affine transformation to a unit vector will not yield a unit vector in general.

5.2 Surface acne

One problem that you are likely to encounter is called *surface acne* and results from precision errors. The problem arises from when the origin of a shadow ray is on the wrong side of its originating surface, and thus intersets the surface. The visual result is usually a black dot at that pixel. The sample images include an example that illustrates this problem. One solution is to offset the shadow ray's origin by a small amount in the ray's direction. Another solution is not to test intersection's against the originating surface.

5.3 Optimizations

There are opportunities for performance improvements both in the the implementation of the ray tracing engine.

The resources listed below include information on techniques for improving the efficiency of ray tracing. Most of these techniques focus on reducing the cost or number of ray/solid intersection tests. For example, if you precompute a bounding volume for a complex object, then a quick test against the bounding volume may allow you to avoid a more expensive test against the object. Another possible optimization is to have a version of your intersection testing code that is specialized for *shadow rays*.

5.4 Resources

Here are a few pointers to on-line sources of information about graphical algorithms and ray tracing.

`http://www.classes.cs.uchicago.edu/current/23700/project-1.html`
is a page of example GML specifications with the expected images.

`http://www.classes.cs.uchicago.edu/current/23700/gml-spec.pdf`
The GML specification.

`http://www.realtimerendering.com/int/`
is the *3D Object Intersection* page with pointers to papers and code describing various intersection algorithms.

`http://www.acm.org/tog/resources/RTNews/html/`
is the home page of the *Ray Tracing News*, which is an online journal about ray tracing techniques.

`http://www.cs.utah.edu/ bes/papers/fastRT/`
is a paper by Brian Smits on efficiency issues in implementing ray tracers.