Chapter 1

Introduction

The *Blue Moon Rendering Tools* (BMRT) are a collection of programs that render 3-D scene models.

BMRT uses some APIs that are very similar to those described in the published RenderMan Interface Specification. However, BMRT is not associated with Pixar, and no claims are made that BMRT is in any way a compatible replacement for RenderMan. Those who want a licensed implementaion of RenderMan should contact Pixar directly.

Despite these technical/legal terms, you may find that most applications, scene files, and shaders written to conform to the RenderMan Interface can also use BMRT without modification.

This document is intended for the reader who is familiar with the concepts of computer graphics and already is fluent in both the RenderMan procedural interface and the RIB archival format (due to BMRT's similarities to that published specification). For more detailed information about the RenderMan standard, we recommend *Advanced RenderMan: Creating CGI for Motion Pictures* by Anthony Apodaca and Larry Gritz, *The RenderMan Companion* by Steve Upstill, or the official RenderMan Interface Specification, available from Pixar. All of these texts are fully detailed and clearly written, and no attempt will be made here to duplicate the information in these references.

The parts of BMRT you'll most likely use are outlined below:

rgl A previewer for RIB files which runs on top of OpenGL. Primitives display as lines or Gouraud-shaded polygons.

rendrib A high quality renderer which uses some of the latest techniques of radiosity and ray tracing to produce near photorealistic images.

slc A compiler for shaders, allowing you to write your own procedures for defining the appearance of surfaces, lights, displacements, volume attenuation, and pixel operations.

mkmip A program to pre-process texture, shadow, and environment map files for more efficient access during rendering.

```
/*
   SGI:
     cc -n32 -I../include colorspheres.c -L../lib -lribout -o colorspheres

   Linux:
     cc -I../include colorspheres.c -L../lib -lribout -o colorspheres

   Win32:
     cl /I..\include /c colorspheres.c
     link colorspheres.obj ..\lib\libribout.lib /out:colorspheres.exe
*/

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <ri.h>

#define NFRAMES 100
#define NSPHERES  4
#define FRAMEROT 15.0f


void
ColorSpheres(int n, float s)
{
    int x, y, z;
    RtColor color;

    if(n <= 0) {
        return;
    }

    RiAttributeBegin();

    RiTranslate(-0.5f, -0.5f, -0.5f );
    RiScale(1.0f/n, 1.0f/n, 1.0f/n);

    for (x = 0; x < n; x++) {
        for (y = 0; y < n; y++) {
            for (z = 0; z < n; z++) {
                color[0] = ((float) x+1) / ((float) n);
                color[1] = ((float) y+1) / ((float) n);
                color[2] = ((float) z+1) / ((float) n);

                RiColor(color);
                RiTransformBegin();
                RiTranslate(x+.5f, y+.5f, z+.5f);
                RiScale(s, s, s);
                RiSphere(0.5f, -0.5f, 0.5f, 360.0f, RI_NULL);
                RiTransformEnd();
            }
        }
    }

    RiAttributeEnd();

    return ;
}


int main(int argC, char** argV)
{
    RtInt  frame;
    float  scale;
    char   filename[64];
    char*  renderer = RI_NULL;
    if (argC != 2) {
        fprintf(stderr,
                "USAGE: %s ribFile|rgl|rendrib\n\n",
                argV[0]);
        exit (-1);
    }

    renderer = argV[1];

    /* if the variable renderer is "rgl" or "rendrib", RiBegin() will
       attempt to start up that renderer and pipe its output directly to
       that renderer.  Since RiDisplay is set to put its output to the
       framebuffer, that renderer will attempt to open the framebuffer
       and render directly to it.  If the variable renderer is set to
       some other value, RiBegin() will open that as a file and put the
       RIB commands in it.
    */

    RiBegin(renderer);

    for (frame = 0; frame <= NFRAMES; frame++) {
        sprintf(filename, "colorSpheres.%03d.tif", frame );
        RiFrameBegin(frame);

          RiProjection("perspective", RI_NULL);
          RiTranslate(0.0f, 0.0f, 1.5f);
          RiRotate(40.0f, -1.0f, 1.0f, 0.0f);

          RiDisplay(filename, RI_FRAMEBUFFER, RI_RGBA, RI_NULL);
          RiFormat((RtInt)256, (RtInt)192, -1.0f);
          RiShadingRate(1.0);

          RiWorldBegin();

            RiLightSource("distantlight", RI_NULL);

            RiSides((RtInt)1);

            scale = (float)(NFRAMES-(frame-1)) / (float)NFRAMES;
            RiRotate(FRAMEROT*frame, 0.0f, 0.0f, 1.0f);
            RiSurface("plastic", RI_NULL);
            ColorSpheres(NSPHERES, scale);
          RiWorldEnd();
        RiFrameEnd();
    }

    RiEnd();

    return 1;
}
```

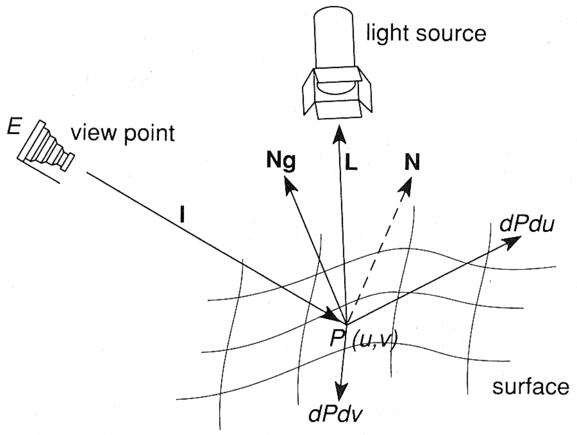

Figure 14.1 *Surface position, normal, and parametric derivatives*

Surface position and change

The **point** value P represents the position of the point being shaded in world space, and Ng is the **geometric normal vector**, perpendicular to the surface, at that point. The **shading normal vector** N is by default equal to Ng, but may be different for shading purposes. If a displacement shader changes the surface normal, it usually works on N and leaves Ng alone.

Parameter space

The floating-point values u and v give the position of the current point on the current surface in parameter space. The **points** dPdu and dPdv are parametric derivatives, giving the derivative of surface position P with respect to u and v, respectively. The surface normal Ng is defined to be the cross-product of these two vectors. u and v always range between exactly 0 and 1 on all surfaces except polygons.

Figure 14.1 illustrates P, Ng, dPdu and dPdv. The normal vector Ng is the cross product of dPdu and dPdv by definition.

Texture space

The floating-point values s and t give the texture-space coordinates of the current point on the surface. They may be used to

Table 14.2 lists each predefined global variable available to surface shaders, together with its data type, storage class and a summary of its meaning. Different sets of global variables are available to other shader types. They appear in Table 14.3.

Type	Name	Storage Class	Purpose
color	_Cs_	varying/uniform	Surface color (input)
color	_Os_	varying/uniform	Surface opacity (input)
point	_P_	varying	Surface position
point	_dPdu_	varying	Change in position with *u*
point	_dPdv_	varying	Change in position with *v*
point	_N_	varying	Surface shading normal
point	_Ng_	varying/uniform	Surface geometric normal
float	_u,v_	varying	Surface parameters
float	_du,dv_	varying/uniform	Change in *u,v* across element
float	_s,t_	varying	Surface texture coordinates
color	_L_	varying/uniform	Direction from surface to light source
color	_Cl_	varying/uniform	Light color
point	_I_	varying	Direction of ray impinging on surface point (often from camera)
color	_Ci_	varying	Color of light from surface (output)
color	_Oi_	varying	Opacity of surface (output)
point	_E_	uniform	Position of the camera

Table 14.2 *Global Variables Available to Surface Shaders*

Surface color and transparency

_Cs_ and _Os_ represent the current surface color and opacity, respectively, as declared in **RiColor**() and **RiOpacity**() and bound to the surface being shaded when it was created.

_Cs_ and _Os_ are used as filter values. The color of reflected light from a surface with surface color _Cs_ under incident light with color _Cl_ is often taken to be _Cl_ *_Cs_. In other words, each component of _Cs_ scales the corresponding component of the incoming light according to the absorption of the surface. _Os_ has the same effect on light passing *through* the surface. Normally, every component of _Cs_ and _Os_ lies in the range [0,1].

Geometric Functions

The shading language defines a variety of functions for geometric calculations. These are summarized in Table 15.2 and discussed below.

Return Type	Declaration	Meaning
float	**xcomp**(P)	return x component of **point** *P*
float	**ycomp**(P)	return y component of **point** *P*
float	**zcomp**(P)	return z component of **point** *P*
—	**setxcomp**(P, v)	set x component of **point** *P* to **float** *v*
—	**setycomp**(P, v)	set y component of **point** *P* to **float** *v*
—	**setzcomp**(P, v)	set z component of **point** *P* to **float** *v*
float	**length**(V)	return the Euclidean length of **point V**
float	**distance**(P1,P2)	return the Euclidean length of (*P1–P2*)
float	**area**(P)	return the surface element area at *P*, in pixels
point	**normalize**(V)	return **V/length**(**V**) for **point V**
point	**faceforward**(V,I)	return **V** flipped to point opposite **I**
point	**reflect**(**I**, **N**)	return reflection of incident ray **I** about normalized vector **N**
point	**refract**(**I**,**N**,eta)	return incident ray **I** refracted through surface with normal **N** and index of refraction **float** *eta*
	fresnel(**I**, **N**, eta, Kr, Kt [, **R**, **T**])	return reflectance coefficient **float** *Kr*, transmittance coefficient **float** *Kt*, reflected ray **R**, and refracted ray **T**, given incident ray **I**, surface normal **N** and relative index of refraction **float** *eta*
point	**transform**([fromspace,] tospace, P)	transform the **point** *P* from the coordinate system named by string *fromspace* to the coordinate space named by *tospace*
float	**depth**(P)	return depth of **point** *P* in camera space, normalized between 0 at the near clipping plane and 1 at the far one
point	**calculatenormal**(P)	return the normal to a surface at **point** *P*

Table 15.2 *Geometric Shading Functions*