Appendix B

Mg Interface Specifications and Source Code

B.1. The Mg Interface

```
/*
 *    Mg output interface definition
 *
 */


typedef     double     xyz[3];
typedef     double     rgb[3];
typedef     double     rgba[4];

enum SurfaceType {Constant, Matte, Plastic, Reflective};
enum LightType {Ambient, Distant, Point};
enum BasisType {Bezier, Bspline, Catrom};


/*
 *       Initialization
 * OutputBegin and OutputEnd are the first and last output
 * procedures called by any scene generator.
 * For convenience, they may write any prologue, epilogue, or
 * auxiliary files required by the target system.
 * OutputBegin takes the name of the picture as an argument.
 */

OutputBegin (name)
char *name;

OutputEnd ()



/*
 *       Frame Setup
 */

OutputViewpoint (position, look_at, up, field_of_view, near, far)
xyz position, look_at, up;
double field_of_view, near, far;

OutputResolution(xres, yres)
int xres, yres;


static int nlights = 0;

OutputLight (model, intensity, color, position, direction)
enum LightType model;
double intensity;
rgb color;
xyz position, direction;
```

```
/*
 *       Matrix  commands
 */


OutputSave()


OutputRestore()


OutputIdentity()


OutputTranslate(dx,dy,dz)
double  dx,  dy,  dz;


OutputScale(sx,  sy,  sz)
double  sx,  sy,  sz;


OutputRotate(angle,  dx,  dy,  dz)
double  angle,  dx,  dy,  dz;
```

```
/*
 *      Object Geometry
 */


/*
 *              vertex routines
 *
 *      OutputVertex(), OutputVertexNormal(), and OutputVertexColor()
 *      set values for the current vertex.
 *      OutputVertex() increments vcount.
 *
 */

OutputVertexReset()

OutputVertex(v)
xyz v;

OutputVertexNormal(n)
xyz n;

OutputVertexST(s, t)
double s, t;

OutputVertexColor(c)
rgba c;


OutputPolygon (n, v, nflag, cflag, tflag)       /* specify vertices in clockwise order */
int n, v[], nflag, cflag, tflag;

OutputPointsPolygons(np, nv, v, nflag, cflag, tflag)
int np, nv[], v[], nflag, cflag, tflag;

OutputPatch (basis, v)
enum BasisType basis;
int v[];

OutputSphere (radius)
double radius;

OutputCone (radius, height)
double radius, height;

OutputCylinder (radius, bottom, top)
double radius, bottom, top;
```

```
/*
 * Shading interface.  All shading properties stay in effect until overwritten.
 *      Color
 *      Opacity (percent opaque, index of refraction)
 *      Surface characteristics (surface model;
 *           coefficients of ambient, diffuse, and specular reflection;
 *           roughness for specular reflection)
 *      Texture map information
 */

OutputColor(color)
xyz color;

OutputOpacity(percent, refraction)
rgb percent;
double refraction;

OutputSurface(model, ka, kd, ks, roughness)
enum SurfaceType model;
double  ka, kd, ks, roughness;

/*
 *       s,t texture coordinates for parametric or quadric surface
 */

OutputTxCoords(s, t)
double s[4], t[4];

/*
 *       specify texture map(s) for a surface
 *  texture shading model specifies number of textures, Ka, and Kd.
 */

OutputTxSurface(n, ka, kd)
int n;
double  ka, kd;
```

B.2. mg_defs.h

```
/*
 *    Constants
 */

#define     TRUE          1
#define     FALSE         0
#define     EPSILON       1.e−10

#define     ONEDEGREE     0.01745329252
#define     ONERADIAN     57.2957795131
#define     PI            3.14159265359

#define     MAXSTRLEN     40    /* string length for picture names, etc */
#define     MAXVERTEX     1000  /* length of vertex lists */
#define     MAXMATRIX     50    /* size of matrix stack */
#define     MAXTEXTURE    10    /* maximum #textures per surface */

/*
 *    Typedefs, Data Structures, and Defined Functions
 */

enum SurfaceType {Constant, Matte, Plastic, Reflective};
enum LightType {Ambient, Distant, Point};
enum BasisType {Bezier, Bspline, Catrom};
static char *BasisName[] = {
    "Bezier", "b−spline", "Catmull−Rom"
};

typedef     double     xyz[3];
typedef     double     xyzw[4];
typedef     double     st[2];
typedef     double     rgb[3];
typedef     double     rgba[4];
typedef     double     Matrix[4][4];

#define     COPYXYZ(d,s)        { d[0] = s[0]; d[1] = s[1]; d[2] = s[2]; }
#define     COPYXYZW(d,s)       { d[0] = s[0]; d[1] = s[1]; d[2] = s[2]; d[3] = s[3]; }
#define     DOT(a,b)            ( a[0] * b[0] + a[1] * b[1] + a[2] * b[2] )
#define     ABS(a)              (((a) < 0.) ?( −(a)) : (a))

/*
 *    Function types
 */

long        setseed();
double      drand();
double      gaussrand();
double      urand();
```

B.3. mg_lib.c

```
#include <stdio.h>
#include <math.h>
#include "mg_defs.h"


/*
 *      Matrix routines
 */

ident(m)                /* makes a 4x4 identity matrix of the argument */
Matrix m;
{
        int i, j;

        for (i = 0; i < 4; ++i) {
                for (j = 0; j < 4; ++j)
                        m[i][j] = 0;
                        m[i][i] = 1;
                }
        }

zero(m)                 /* makes a 4x4 zero matrix */
Matrix m;
{
        int i, j;

        for (i = 0; i < 4; ++i)
                for (j = 0; j < 4; ++j)
                        m[i][j] = 0;
}

mxcopy(src, dest)       /* 4x4 dest matrix gets a copy of src matrix */
Matrix src, dest;
{
        int i, j;

        for (i = 0; i < 4; i++)
                for (j = 0; j < 4; j++)
                        dest[i][j] = src[i][j];
}

mxprint(fp, m)          /* prints a matrix in row major order */
FILE *fp;
Matrix m;
{
        int i, j;

        for (i = 0; i < 4; ++i)
                for (j = 0; j < 4; j++)
                        fprintf(fp, " %g", m[i][j]);
}
```

```
mxmult(r, a, b)          /* 4x4 matrix multiplication:  r = a * b */
Matrix r, a, b;
{
int i, j;

        for (i = 0; i < 4; i++)
                for (j = 0; j < 4; j++)
                        r[i][j] = a[i][0] * b[0][j]
                                + a[i][1] * b[1][j]
                                + a[i][2] * b[2][j]
                                + a[i][3] * b[3][j];
}

transform(r, v, m)       /* multiply vector v by matrix m, store result in vector r */
xyz r, v;
Matrix m;
{
        r[0] = v[0]*m[0][0] + v[1]*m[1][0] + v[2]*m[2][0] + v[3]*m[3][0];
        r[1] = v[0]*m[0][1] + v[1]*m[1][1] + v[2]*m[2][1] + v[3]*m[3][1];
        r[2] = v[0]*m[0][2] + v[1]*m[1][2] + v[2]*m[2][2] + v[3]*m[3][2];
        r[3] = v[0]*m[0][3] + v[1]*m[1][3] + v[2]*m[2][3] + v[3]*m[3][3];
}

mxnormalize(v)           /* Normalize a vector */
xyzw v;
{
        double length;

        length = sqrt(v[0] * v[0] + v[1] * v[1] + v[2] * v[2]);
        if (length > 0.) {
                v[0] /= length;
                v[1] /= length;
                v[2] /= length;
        }
        return(length);
}

mxcross(r, a, b)         /* assumes a[3] == b[3] */
xyzw r, a, b;
{
        double r0, r1, r2;

        r0 = a[1]*b[2] – a[2]*b[1];
        r1 = a[0]*b[2] – a[2]*b[0];
        r2 = a[0]*b[1] – a[1]*b[0];
        r[0] = r0; r[1] = –r1; r[2] = r2;
        r[3] = a[3];
}
```

```
/*
 *        Transformation Routines
 *   Scale, translate, and rotate set the argument matrix
 *   to represent a specified single transformation.
 */

scale(m, sx, sy, sz)
Matrix m;
double sx, sy, sz;
{
        ident(m);
        m[0][0] = sx; m[1][1] = sy; m[2][2] = sz;
}

translate(m, tx, ty, tz)
Matrix m;
double tx, ty, tz;
{
        ident(m);
        m[3][0] = tx; m[3][1] = ty; m[3][2] = tz;
}

/*
 *        rotate
 * about an axis, through a given angle
 * Takes as arguments the index of the axis and the angle in degrees.
 * Assumes lefthand coordinate system in setting up the rotation matrix.
 */

rotate(m, about, degrees)
Matrix m;
int about;
double degrees;
{
        double s, c;
        int from, to;

        s = sin(degrees * ONEDEGREE);
        c = cos(degrees * ONEDEGREE);
        from = (about + 1) % 3;
        to = (about + 2) % 3;
        ident (m);
        m[from][from] = m[to][to] = c;
        m[from][to] = s;
        m[to][from] = –s;
}
```

```
/*
 *    change of basis from Catmull–Rom to Bezier
 */

Matrix convert = {
        0.,        6./6.,     0.,        0.,
        –1./6.,    6./6.,     1./6.,     0.,
        0.,        1./6.,     6./6.,     –1./6.,
        0.,        0.,        6./6.,     0.
};

Matrix transpose = {
        0.,        –1./6.,    0.,             0.,
        6./6.,     6./6.,     1./6.,          0.,
        0.,        1./6.,     6./6.,          6./6.,
        0.,        0.,        –1./6.,         0.
};

CatromToBez(to, from)
Matrix to, from;
{
        Matrix temp;

        mxmult(temp, convert, from);
        mxmult(to, temp, transpose);
}
```

```
/*
 *    error message routines
 */

error(message, s)
char *message, *s;
{
        fprintf(stderr, message, s);
        exit(–1);
}

warning(message, s)
char *message, *s;
{
        fprintf(stderr, message, s);
}
```

```
/*
 * portable uniform random number generator
 * based on Forsythe, Malcolm, and Moler
 *
 * change seed with setseed; returns previous seed
 * reinitialize with setseed(1)
 */
static long m2 = 0;
static long seed = 1;

double
drand()
{
	long iseed;
	static long a, c, m, mc;
	static double halfm, scale;

	iseed = seed;
	if (!m2) {
		for (m = 1; m2 < m; ) {
			m2 = m;
			m = 2 * m2;
		}
		halfm = m2;
		a = 8 * (long) (halfm * atan(1.)/8.) +5;
		c = 2 * (long) (halfm * (0.5 - sqrt(3.)/6.)) + 1.;
		mc = (m2 - c) + m2;
		scale = 0.5/halfm;
	}
	iseed *= a;
	if (iseed > mc)
		iseed = (iseed - m2) - m2;
	iseed += c;
	if (iseed/2 > m2)
		iseed = (iseed - m2) - m2;
	if (iseed < 0)
		iseed = (iseed + m2) + m2;
	seed = iseed;
	return ((double)iseed * scale);
}

long
setseed(new)
long new;
{
	long old;

	old = seed;
	seed = new;
	return(old);
}
```

```
/*
 *    procedures that use random numbers
 */

double
gaussrand()              /* Gaussian, returns a double distributed N(0,1) */
{
      double v1, v2, r, s, g1;
      static double g2;
      static int extra = 0;

      if (extra) {
            extra = 0;
            return (g2);
      }
      do {
            v1 = urand();
            v2 = urand();
            r = v1 * v1 + v2 * v2;
      } while (r >= 1.);
      s = sqrt(–2. * log(r)/r);
      g1 = v1 * s;
      g2 = v2 * s;
      extra = 1;
      return(g1);
}

double
urand()                  /* return uniform random number between (–1,1) */
{
      return((drand() * 2.) – 1.0);
}

/*
 *     select random colors
 */

#define COLORSCALE 1.7
double drand();

RandColor(color)
rgba color;
{
      color[0] = drand() / COLORSCALE;
      color[1] = drand() / COLORSCALE;
      color[2] = drand() / COLORSCALE;
      color[3] = drand() / COLORSCALE;
}
```

B.4. mg_output.c

```
#include <stdio.h>
#include <math.h>
#include "mg_defs.h"

/*
 *  mg_output.c – object output routines
 *  Photorealistic RenderMan version, RIB output
 *
 *  does not support:
 *      reflective surfaces (uses "metal" shader)
 *      refraction
 */


/*
 *      Initialization
 * OutputBegin and OutputEnd are the first and last output
 * procedures called by any scene generator.
 * For convenience, they may write any prologue, epilogue, or
 * auxiliary files required by the target system.
 * OutputBegin takes the name of the picture as an argument.
 */

FILE *ReyesCmdFile, *ModelFile;
char picture_name[MAXSTRLEN];

OutputBegin (name)
char *name;
{
	char temp_name[MAXSTRLEN];

	strcpy (picture_name, name);
	sprintf(temp_name, "%s.rip", picture_name);
	ModelFile = fopen (temp_name, "w");
	fprintf(ModelFile,
		"display \"%s.pic\" \"framebuffer\" \"rgba\"\n",
		picture_name);
	fflush(ModelFile);
}

OutputEnd ()
{
	fprintf(ModelFile, "WorldEnd\n");
	fclose(ModelFile);
}

OutputFlush()   /* for debugging */
{
	fflush(ModelFile);
}
```

```
/*
 *      Frame Setup
 */

OutputViewpoint (position, look_at, up, field_of_view, near, far)
xyz position, look_at, up;
double field_of_view, near, far;
{
        int i;
        xyzw u, v, n, V;
        Matrix m;
        double upn;

        fprintf(ModelFile,
                "Clipping %g %g\n", near, far);
        fprintf(ModelFile,
                "Projection \"perspective\" \"fov\" %g \n", field_of_view);

        /* viewpoint uses position, look at, up vectors */
        OutputIdentity();
        V[0]=position[0]; n[0]=look_at[0]−position[0]; v[0]=up[0]−position[0];
        V[1]=position[1]; n[1]=look_at[1]−position[1]; v[1]=up[1]−position[1];
        V[2]=position[2]; n[2]=look_at[2]−position[2]; v[2]=up[2]−position[2];
        V[3] = n[3] = v[3] = 1.;
        mxnormalize(n);             /* get unit normal */
        mxcross(u, n, v);
        mxnormalize(u);
        /* check for degenerate case */
        if ((ABS(u[0])<EPSILON) && (ABS(u[1])<EPSILON) && (ABS(u[2])<EPSILON))
                error("Up collinear with normal\n");
        upn = DOT(v, n);            /* get unit up */
        v[0] −= upn*n[0];
        v[1] −= upn*n[1];
        v[2] −= upn*n[2];
        v[3] = 1.;
        mxnormalize(v);
        u[3] = v[3] = n[3] = V[3] = 1.;
        m[0][0] = u[0]; m[0][1] = v[0]; m[0][2] = n[0];
        m[1][0] = u[1]; m[1][1] = v[1]; m[1][2] = n[1];
        m[2][0] = u[2]; m[2][1] = v[2]; m[2][2] = n[2];
        m[0][3] = m[1][3] = m[2][3] = 0.;
        m[3][0] = −DOT(u,V); m[3][1] = −DOT(v,V); m[3][2] = −DOT(n,V);
        m[3][3] = 1.;
        fprintf(ModelFile, "ConcatTransform [ ");
        mxprint(ModelFile, m);
        fprintf(ModelFile, "]\n");
        fprintf(ModelFile, "WorldBegin\n");
}

OutputResolution(xres, yres)       /* assumes pixel aspect ratio = 1 */
int xres, yres;
{
        fprintf(ModelFile, "format %d %d 1\n", xres, yres);
}
```

```
/*
 *    initialize light sources
 */

static int nlights = 0;

OutputLight (model, intensity, color, position, direction)
enum LightType model;
double intensity;
rgb color;
xyz position, direction;
{
        ++nlights;
        switch(model) {
        case Ambient:
                fprintf(ModelFile,
                        "LightSource \"ambientlight\" %d ",
                        nlights);
                fprintf(ModelFile,
                        "\"intensity\" [%g] \"lightcolor\" [%g %g %g]\n",
                        intensity, color[0], color[1], color[2]);
                break;
        case Distant:
                fprintf(ModelFile,
                        "LightSource \"distantlight\" %d ",
                        nlights);
                fprintf(ModelFile,
                        "\"intensity\" [%g] \"lightcolor\" [%g %g %g] ",
                        intensity, color[0], color[1], color[2]);
                fprintf(ModelFile,
                        "\"from\" [%g] \"to\" [%g %g %g]\n",
                        position[0], position[1], position[2],
                        direction[0], direction[1], direction[2]);
                break;
        case Point:
                fprintf(ModelFile,
                        "LightSource \"pointlight\" %d ", nlights);
                fprintf(ModelFile,
                        "\"intensity\" [%g] \"lightcolor\" [%g %g %g] ",
                        intensity, color[0], color[1], color[2]);
                fprintf(ModelFile,
                        "\"from\" [%g %g %g]\n",
                        position[0], position[1], position[2]);
                break;
        default:
                warning("Nonstandard light source model ignored (%s)\n", model);
        }
        return(nlights);
}
```

```
/*
 *      Matrix commands
 */



OutputSave()
{
      fprintf(ModelFile, "TransformBegin\n");
}

OutputRestore()
{
      fprintf(ModelFile, "TransformEnd\n");
}

OutputIdentity()
{
      fprintf(ModelFile, "Identity\n");
}

OutputTranslate(dx,dy,dz)
double dx, dy, dz;
{
      if ((ABS(dx)<EPSILON) && (ABS(dy)<EPSILON) && (ABS(dz)<EPSILON))
            return;
      fprintf(ModelFile,
            "Translate %g %g %g\n",
            dx, dy, dz);
}

OutputScale(sx, sy, sz)
double sx, sy, sz;
{
      fprintf(ModelFile,
            "Scale %g %g %g\n",
            sx, sy, sz);
}

OutputRotate(angle, dx, dy, dz)
double angle, dx, dy, dz;
{
      fprintf(ModelFile,
            "Rotate %g %g %g %g\n",
            angle, dx, dy, dz);
}
```

```
/*
 *       Object  Geometry
 */

/*
 *    vertex  routines
 *          OutputVertex(),  OutputVertexNormal(),  and  OutputVertexColor()
 *                set  values  for  the  current  vertex.
 *          OutputVertex()  increments  vcount  (current  vertex  index).
 *          OutputVertexReset()  sets  it  to  0.
 */

int vcount = –1;        /* index of current vertex */
xyz vertex[MAXVERTEX];
xyz vertex_normal[MAXVERTEX];
st vertex_st[MAXVERTEX];
rgba vertex_color[MAXVERTEX];

OutputVertexReset()
{
      vcount = –1;
      return(vcount+1);
}

OutputVertex(v)
xyz v;
{
      vcount++;
      COPYXYZ(vertex[vcount], v);
      return(vcount);
}

OutputVertexNormal(n)
xyz n;
{
      COPYXYZ(vertex_normal[vcount], n);
      return(vcount);
}

OutputVertexST(s, t)
double s, t;
{
      vertex_st[vcount][0] = s;
      vertex_st[vcount][1] = t;
      return(vcount);
}

OutputVertexColor(c)
rgba c;
{
      COPYXYZW(vertex_color[vcount], c);
      return(vcount);
}
```

```
OutputPolygon (n, v, nflag, cflag, tflag)          /* vertices in clockwise order */
int n, v[], nflag, cflag, tflag;
{
        int i;

        fprintf(ModelFile, "Polygon \"P\" [");
        for (i = 0; i < n; i++)
                fprintf(ModelFile, " %g %g %g",
                        vertex[v[i]][0], vertex[v[i]][1], vertex[v[i]][2]);
        fprintf(ModelFile, " ]");
        if (nflag) {
                fprintf(ModelFile, " \"N\" [");
                for (i = 0; i < n; i++)
                fprintf(ModelFile, " %g %g %g",
                        vertex_normal[v[i]][0], vertex_normal[v[i]][1], vertex_normal[v[i]][2]);
                fprintf(ModelFile, " ]");
        }
        if (tflag) {
                fprintf(ModelFile, " \"st\" [");
                for (i = 0; i < n; i++)
                        fprintf(ModelFile, " %g %g",
                                vertex_st[v[i]][0], vertex_st[v[i]][1]);
                fprintf(ModelFile, " ]");
        }
        if (cflag) {
                fprintf(ModelFile, " \"Cs\" [");
                for (i = 0; i < n; i++)
                        fprintf(ModelFile, " %g %g %g",
                                vertex_color[v[i]][0], vertex_color[v[i]][1], vertex_color[v[i]][2]);
                fprintf(ModelFile, " ]");
        }
        fprintf(ModelFile, "\n");
}

OutputPointsPolygons(np, nv, v, nflag, cflag, tflag)
int np, nv[], v[], nflag, cflag, tflag;
{
        int i, j, totv = 0, vlast = –1;

        fprintf(ModelFile, "PointsPolygons [ ");
        for (i = 0; i < np; ++i) {                          /* print nvertices */
                fprintf(ModelFile, "%d ", nv[i]);
                if ((i+1)%20 == 0 && i != np–1)
                        fprintf(ModelFile, "\n");
                totv += nv[i];
        }
        fprintf(ModelFile, "]\n    [ ");
        for (i = 0; i < totv; ++i) {                        /* print vertex indexes */
                fprintf(ModelFile, "%d ", v[i]);
                if (v[i] > vlast)
                vlast = v[i];
                if ((i+1)%20 == 0 && i != totv–1)
                        fprintf(ModelFile, "\n");
        }
```

```
	fprintf(ModelFile, "]\n   \"P\"  [ ");
	for (i = 0; i <= vlast; i++) {
		fprintf(ModelFile, " %g %g %g",
			vertex[i][0], vertex[i][1], vertex[i][2]);
		if ((i+1)%3 == 0 && i < vlast)
		fprintf(ModelFile, "\n");
	}
	fprintf(ModelFile, " ]\n");
	if (nflag) {
		fprintf(ModelFile, "  \"N\"  [");
		for (i = 0; i <= vlast; i++) {
			fprintf(ModelFile, " %g %g %g ",
				vertex_normal[i][0], vertex_normal[i][1], vertex_normal[i][2]);
			if ((i+1)%3 == 0 && i < vlast)
				fprintf(ModelFile, "\n");
		}
		fprintf(ModelFile, " ]\n");
	}
	if (tflag) {
		fprintf(ModelFile, "  \"st\"  [");
		for (i = 0; i <= vlast; i++) {
			fprintf(ModelFile, " %g %g",
				vertex_st[i][0], vertex_st[i][1]);
			if ((i+1)%3 == 0 && i < vlast)
				fprintf(ModelFile, "\n");
		}
		fprintf(ModelFile, " ]\n");
	}
	if (cflag) {
		fprintf(ModelFile, "  \"Cs\"  [");
		for (i = 0; i <= vlast; i++) {
			fprintf(ModelFile, " %g %g %g",
				vertex_color[i][0], vertex_color[i][1], vertex_color[i][2]);
			if ((i+1)%3 == 0 && i < vlast)
				fprintf(ModelFile, "\n   ");
		}
		fprintf(ModelFile, " ]\n");
	}
}
```

```
OutputPatch (basis, v)
enum BasisType basis;
int v[];
{
	int i;
	static savebasis, init = FALSE;

	if (!init || basis != savebasis) {
		init = TRUE;
		savebasis = basis;
		fprintf(ModelFile, "Basis ");
		switch (basis) {
		case Bezier:
			fprintf(ModelFile, "\"bezier\"\n");
			break;
		case Bspline:
		case Catrom:
			error("%s patches not implemented\n", BasisName[(int) basis]);
		default:
			error("Unrecognized patch type\n");
		}
	}
	fprintf(ModelFile, "Patch \"bicubic\" \"P\" [");
	for (i = 0; i < 16; i++)
		fprintf(ModelFile, " %g %g %g",
			vertex[v[i]][0], vertex[v[i]][1], vertex[v[i]][2]);
	fprintf(ModelFile, " ]\n");
}

OutputSphere (radius)
double radius;
{
	fprintf(ModelFile, "Sphere %g %g %g 360\n",
		radius, radius, -radius) ;
}

OutputCone (radius, height)
double radius, height;
{
	fprintf(ModelFile, "Cone %g %g 360\n",
		height, radius) ;
}

OutputCylinder (radius, bottom, top)
double radius, bottom, top;
{
	fprintf(ModelFile, "Cylinder %g %g %g 360\n",
		radius, bottom, top) ;
}
```

```
/*
 * Shading interface.  All shading properties stay in effect until overwritten.
 *      Color
 *      Opacity (percent opaque, index of refraction)
 *      Surface characteristics (surface model;
 *           coefficients of ambient, diffuse, and specular reflection;
 *           roughness for specular reflection)
 *      Texture map information
 */

OutputColor(color)
xyz color;
{
	fprintf(ModelFile,
		"Color [%g %g %g]\n", color[0], color[1], color[2]);
}

OutputOpacity(percent, refraction)	/* refraction is not supported in this implementation */
rgb percent;
double refraction;
{
	fprintf(ModelFile,
		"Opacity [%g %g %g]\n", percent[0], percent[1], percent[2]);
}

OutputSurface(model, ka, kd, ks, roughness)
enum SurfaceType model;
double  ka, kd, ks, roughness;
{
	switch (model) {
	case Constant:
		fprintf(ModelFile, "Surface \"constant\"\n");
		break;
	case Matte:
		fprintf(ModelFile,
			"Surface \"matte\" \"Ka\" %g \"Kd\" %g \n",
			ka, kd);
		break;
	case Reflective:
		fprintf(ModelFile,
			"Surface \"metal\" \"Ka\" %g \"Ks\" %g \"roughness\" %g \n",
			ka, ks, roughness);
		break;
	case Plastic:
		/* use default of white for color of specular reflection */
		fprintf(ModelFile,
			"Surface \"plastic\" \"Ka\" %g \"Kd\" %g \"Ks\" %g \"roughness\" %g \n",
			ka, kd, ks, roughness);
		break;
	default:
		warning("Nonstandard surface model ignored (%s)\n", model);
	}
}
```

```
/*
 *      s,t texture coordinates for parametric or quadric surface
 */

OutputTxCoords(s, t)
double s[4], t[4];
{
        int i;

        fprintf(ModelFile, "TextureCoordinates [");
        for (i = 0; i < 4; i++)
                fprintf(ModelFile, " %g %g", s[i], t[i]);
        fprintf(ModelFile, " ]\n");
}

/*
 *      specify texture map(s) for a surface
 *  texture shading model specifies number of textures, Ka, and Kd.
 *
 *  Rib implementation keeps a texture counter, and varies
 *  the texture when ntx == 1.
 *  Texture maps are name mg_tex0_t, ..., mg_tex9_t.
 *  Shaders are named mg_shade1, ..., mg_shade10, indicating
 *  number of textures applied to the surface.  Only mg_shade1
 *  can take a texture name as an argument.
 */

OutputTxSurface(n, ka, kd)
int n;
double  ka, kd;
{
        static int txcount = 1;
        char name[MAXSTRLEN];

        if (n < 1 || n > MAXTEXTURE)
                error("Specify from 1 to %d textures.\n", MAXTEXTURE);
        if (n == 1) {
                sprintf (name, "mg_tex%d_t", txcount++);
                fprintf(ModelFile,
                        "Surface \"mg_shade1\" \"Ka\" %g \"Kd\" %g \"texname\" \"%s\"\n",
                        ka, kd, name);
        }
        else {
                sprintf (name, "mg_shade%d", n);
                fprintf(ModelFile,
                        "Surface \"%s\" \"Ka\" %g \"Kd\" %g \n",
                        name, ka, kd);
        }
}
```

B.5. mg_output.c, differences in Opal version

```
#include <stdio.h>
#include <math.h>
#include "tk_defs.h"

/*
 * tk_output.c  – object output routines
 * Opal ray tracer version, RIB output
 * This listing contains only the procedures that differ from the prman/rib version
 */

/*
 * Shading interface.   All shading properties stay in effect until overwritten.
 *      Color
 *      Opacity (percent opaque, index of refraction)
 *      Surface characteristics (surface model;
 *           coefficients of ambient, diffuse, and specular reflection;
 *           roughness for specular reflection)
 *      Texture map information
 */

static int transparent;
static double rindex;

OutputOpacity(percent, refraction)
rgb percent;
double refraction;
{
        fprintf(ModelFile,
                "Opacity [%g %g %g]\n", percent[0], percent[1], percent[2]);
        if (percent[0] < 1. || percent[1] < 1. || percent[2] < 1.) {
                transparent = TRUE;
                rindex = refraction;
        }
        else
              transparent = FALSE;
}

OutputSurface(model, ka, kd, ks, roughness)
enum SurfaceType model;
double  ka, kd, ks, roughness;
{
        switch (model) {
        case Constant:
                fprintf(ModelFile, "Surface \"constant\"\n");
                break;
        case Matte:
                fprintf(ModelFile,
                        "Surface \"matte\" \"Ka\" %g \"Kd\" %g \n",
                        ka, kd);
                break;
        case Reflective:
```

```
            fprintf(ModelFile,
                  "Surface \"tk_reflect\" \"Ka\" %g \"Ks\" %g \"roughness\" %g \n",
                  ka, ks, roughness);
            break;
      case Plastic:
      /*
      *     use default of white for color of specular reflection
      */
            if (transparent)
                  fprintf(ModelFile,
                        "Surface \"tk_refract\" \"Ka\" %g \"Kd\" %g \"Ks\" %g\
 \"roughness\" %g \"eta\" %g\n",
                        ka, kd, ks, roughness, rindex);
            else
                  fprintf(ModelFile,
                        "Surface \"plastic\" \"Ka\" %g \"Kd\" %g \"Ks\" %g\
 \"roughness\"%g \n",
                        ka, kd, ks, roughness);
            break;
      default:
            warning("Unrecognized surface model ignored (%s)\n", model);
      }
}
```

B.6. mg_texture.c

```
/*
 *        mg_texture.c
 *
 * This program generates model files for ten texture maps.
 * Render with an orthographic projection,
 * and convert image to system–specific texture map format.
 */

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "mg_defs.h"

#define TEXRES       512
#define HALFWIDTH  0.03125

char picname[100];

rgb background = {
      0.15,
      0.15,
      0.15
};

rgb texcolor[10] = {
      0.4,  0.,   0.,
      0.,   0.4,  0.,
      0.,   0.,   0.4,
      .5,   .5,   0.,
      .4,   0.,   0.,
      0.,   0.4,  0.,
      0.,   0.,   0.4,
      .5,   .5,   0.,
      .5,   0.,   .1,
      .5,   0.,   .1
};

double x[33] = {
      1.0,        0.9375,     0.875,      0.8125,
      0.75,       0.6875,     0.625,      0.5625,
      0.5,        0.4375,     0.375,      0.3125,
      0.25,       0.1875,     0.125,      0.0625,
      0.0,        –0.0625,    –0.125,     –0.1875,
      –0.25,      –0.3125,    –0.375,     –0.4375,
      –0.5,       –0.5625,    –0.625,     –0.6875,
      –0.75,      –0.8125,    –0.875,     –0.9375,
      –1.
};
```

```
main(argc, argv)
int argc;
char *argv[];
{
	int i, j, start;
	xyz v, position, look, up;
	int vlist[4];
	rgb ocolor = {1., 1., 1.};

	for (i = 0; i < 10; ++i) {
		sprintf(picname, "try.%d", i+1);
		OutputBegin(picname);
		OutputResolution(TEXRES, TEXRES);

	/* set viewing position */

		position[0] = 0.; position[1] = 0.; position[2] = –5.;
		look[0] = 0.; look[1] = 0.; look[2] = 0.;
		if (i < 4 || i == 8) {		/* vertical stripes */
			up[0] = 0.;
			up[1] = 1.;
			up[2] = –5.;
		}
		else {				/* horizontal stripes */
			up[0] = 1.;
			up[1] = 0.;
			up[2] = –5.;
		}
		OutputViewpoint(position, look, up, 45., 1., 10.);

		OutputSurface(Constant);
		OutputOpacity(ocolor, 0.);
		vlist[0] = 0; vlist[1] = 1; vlist[2] = 2; vlist[3] = 3;

	/* background polygon*/

		OutputVertexReset();
		v[2] = 0.1;
		v[0] = –1.; v[1] = 1.;
		OutputVertex(v);
		v[0] = 1.;
		OutputVertex(v);
		v[1] = –1.;
		OutputVertex(v);
		v[0] = –1.;
		OutputVertex(v);
		OutputColor(background);
		OutputPolygon(4, vlist, FALSE, FALSE, FALSE);
```

```
	/* stripes for current texture */

		OutputColor(texcolor[i]);
		v[2] = 0.;
		if (i < 8)
			start = i%4;
		else
			start = 2;
		for (j = start; j < 32; j+=4){
			OutputVertexReset();
			v[0] = x[j];
			v[1] = 1.;
			OutputVertex(v);
			v[1] = -1.;
			OutputVertex(v);
			if (i < 8)
				v[0] = x[j+1];
			else
				v[0] = x[j] + HALFWIDTH;
			OutputVertex(v);
			v[1] = 1.;
			OutputVertex(v);
			OutputPolygon(4, vlist, FALSE, FALSE, FALSE);
		}
		OutputEnd();
	}
}
```

B.7. mg_spheres.h

```
#define NC_BEZIER 7
double c_bezier[NC_BEZIER] = {   /* coordinates for bezier patches */
         0.0,
        –1.0,
         1.0,
        –0.305018,
         0.305018,
        –0.552285,
         0.552285
};

#define NV_BEZIER 62
int nv_bezier = NV_BEZIER;

int v_bezier [NV_BEZIER][3] = {
        /* vertices:  value is index into coordinate list */
        0, 0, 1,    6, 0, 1,         6, 4, 1,         6, 3, 1,
        5, 0, 1,    5, 4, 1,         5, 3, 1,         0, 6, 1,
        4, 6, 1,    3, 6, 1,         0, 5, 1,         4, 5, 1,
        3, 5, 1,    0, 0, 2,         6, 0, 2,         6, 4, 2,
        6, 3, 2,    5, 0, 2,         5, 4, 2,         5, 3, 2,
        0, 6, 2,    4, 6, 2,         3, 6, 2,         0, 5, 2,
        4, 5, 2,    3, 5, 2,         2, 0, 0,         2, 6, 0,
        2, 5, 0,    1, 0, 0,         1, 6, 0,         1, 5, 0,
        0, 2, 0,    6, 2, 0,         5, 2, 0,         0, 1, 0,
        6, 1, 0,    5, 1, 0,         2, 0, 6,         2, 6, 6,
        2, 5, 6,    1, 0, 6,         1, 6, 6,         1, 5, 6,
        0, 2, 6,    6, 2, 6,         5, 2, 6,         0, 1, 6,
        6, 1, 6,    5, 1, 6,         2, 0, 5,         2, 6, 5,
        2, 5, 5,    1, 0, 5,         1, 6, 5,         1, 5, 5,
        0, 2, 5,    6, 2, 5,         5, 2, 5,         0, 1, 5,
        6, 1, 5,    5, 1, 5
};

#define NP_BEZIER 8
int np_bezier = NP_BEZIER;
int p_bezier[NP_BEZIER][16] = {
        0, 0, 0, 0, 1, 2, 8, 7, 50, 51, 57, 56, 26, 27, 33, 32,
        0, 0, 0, 0, 7, 9, 5, 4, 56, 58, 54, 53, 32, 34, 30, 29,
        0, 0, 0, 0, 4, 6, 12, 10, 53, 55, 61, 59, 29, 31, 37, 35,
        0, 0, 0, 0, 10, 11, 3, 1, 59, 60, 52, 50, 35, 36, 28, 26,
        26, 27, 33, 32, 38, 39, 45, 44, 14, 15, 21, 20, 13, 13, 13, 13,
        32, 34, 30, 29, 44, 46, 42, 41, 20, 22, 18, 17, 13, 13, 13, 13,
        29, 31, 37, 35, 41, 43, 49, 47, 17, 19, 25, 23, 13, 13, 13, 13,
        35, 36, 28, 26, 47, 48, 40, 38, 23, 24, 16, 14, 13, 13, 13, 13
};
```

```
#define NC_POLY 19
double c_poly[NC_POLY] = {        /* coordinates for polygons  */
        −1,
        −0.92388,
        −0.862856,
        −0.707107,
        −0.678598,
        −0.57735,
        −0.382683,
        −0.357407,
        −0.281085,
        0,
        0.281085,
        0.357407,
        0.382683,
        0.57735,
        0.678598,
        0.707107,
        0.862856,
        0.92388,
        1
};

#define NV_POLY 98
int nv_poly = NV_POLY;

int v_poly [NV_POLY][3] = {
      /* vertices:  value is index into coordinate list */
        0, 9, 9,        1, 6, 9,        1, 9, 6,        1, 9, 12,       1, 12, 9,
        2, 7, 7,        2, 7, 11,       2, 11, 7,       2, 11, 11,      3, 3, 9,
        3, 9, 3,        3, 9, 15,       3, 15, 9,       4, 8, 4,        4, 8, 14,
        4, 4, 8,        4, 4, 10,       4, 10, 4,       4, 10, 14,      4, 14, 8,
        4, 14, 10,      5, 5, 5,        5, 5, 13,       5, 13, 5,       5, 13, 13,
        6, 1, 9,        6, 9, 1,        6, 9, 17,       6, 17, 9,       7, 7, 2,
        7, 7, 16,       7, 2, 7,        7, 2, 11,       7, 11, 2,       7, 11, 16,
        7, 16, 7,       7, 16, 11,      8, 4, 4,        8, 4, 14,       8, 14, 4,
        8, 14, 14,      9, 6, 1,        9, 6, 17,       9, 3, 3,        9, 3, 15,
        9, 1, 6,        9, 1, 12,       9, 0, 9,        9, 9, 0,        9, 9, 18,
        9, 12, 1,       9, 12, 17,      9, 15, 3,       9, 15, 15,      9, 17, 6,
        9, 17, 12,      9, 18, 9,       10, 4, 4,       10, 4, 14,      10, 14, 4,
        10, 14, 14,     11, 7, 2,       11, 7, 16,      11, 2, 7,       11, 2, 11,
        11, 11, 2,      11, 11, 16,     11, 16, 7,      11, 16, 11,     12, 1, 9,
        12, 9, 1,       12, 9, 17,      12, 17, 9,      13, 5, 5,       13, 5, 13,
        13, 13, 5,      13, 13, 13,     14, 8, 4,       14, 8, 14,      14, 4, 8,
        14, 4, 10,      14, 10, 4,      14, 10, 14,     14, 14, 8,      14, 14, 10,
        15, 3, 9,       15, 9, 3,       15, 9, 15,      15, 15, 9,      16, 7, 7,
        16, 7, 11,      16, 11, 7,      16, 11, 11,     17, 6, 9,       17, 9, 6,
        17, 9, 12,      17, 12, 9,      18, 9, 9
};
```

```
#define NP_POLY 192
int np_poly = NP_POLY;
int p_poly[NP_POLY][3] = {
    7, 17, 23,      23, 19, 7,      2, 10, 17,      17, 7, 2,
    5, 13, 10,      10, 2, 5,       15, 21, 13,     13, 5, 15,
    4, 7, 19,       19, 12, 4,      0, 2, 7,        7, 4, 0,
    1, 5, 2,        2, 0, 1,        9, 15, 5,       5, 1, 9,
    8, 4, 12,       12, 20, 8,      3, 0, 4,        4, 8, 3,
    6, 1, 0,        0, 3, 6,        16, 9, 1,       1, 6, 16,
    18, 8, 20,      20, 24, 18,     11, 3, 8,       8, 18, 11,
    14, 6, 3,       3, 11, 14,      22, 16, 6,      6, 14, 22,
    31, 37, 21,     21, 15, 31,     45, 43, 37,     37, 31, 45,
    63, 57, 43,     43, 45, 63,     79, 73, 57,     57, 63, 79,
    25, 31, 15,     15, 9, 25,      47, 45, 31,     31, 25, 47,
    69, 63, 45,     45, 47, 69,     85, 79, 63,     63, 69, 85,
    32, 25, 9,      9, 16, 32,      46, 47, 25,     25, 32, 46,
    64, 69, 47,     47, 46, 64,     80, 85, 69,     69, 64, 80,
    38, 32, 16,     16, 22, 38,     44, 46, 32,     32, 38, 44,
    58, 64, 46,     46, 44, 58,     74, 80, 64,     64, 58, 74,
    33, 39, 23,     23, 17, 33,     50, 52, 39,     39, 33, 50,
    65, 59, 52,     52, 50, 65,     81, 75, 59,     59, 65, 81,
    26, 33, 17,     17, 10, 26,     48, 50, 33,     33, 26, 48,
    70, 65, 50,     50, 48, 70,     86, 81, 65,     65, 70, 86,
    29, 26, 10,     10, 13, 29,     41, 48, 26,     26, 29, 41,
    61, 70, 48,     48, 41, 61,     77, 86, 70,     70, 61, 77,
    37, 29, 13,     13, 21, 37,     43, 41, 29,     29, 37, 43,
    57, 61, 41,     41, 43, 57,     73, 77, 61,     61, 57, 73,
    91, 83, 75,     75, 81, 91,     96, 88, 83,     83, 91, 96,
    92, 84, 88,     88, 96, 92,     82, 76, 84,     84, 92, 82,
    94, 91, 81,     81, 86, 94,     97, 96, 91,     91, 94, 97,
    95, 92, 96,     96, 97, 95,     87, 82, 92,     92, 95, 87,
    89, 94, 86,     86, 77, 89,     93, 97, 94,     94, 89, 93,
    90, 95, 97,     97, 93, 90,     78, 87, 95,     95, 90, 78,
    79, 89, 77,     77, 73, 79,     85, 93, 89,     89, 79, 85,
    80, 90, 93,     93, 85, 80,     74, 78, 90,     90, 80, 74,
    35, 19, 23,     23, 39, 35,     28, 12, 19,     19, 35, 28,
    36, 20, 12,     12, 28, 36,     40, 24, 20,     20, 36, 40,
    54, 35, 39,     39, 52, 54,     56, 28, 35,     35, 54, 56,
    55, 36, 28,     28, 56, 55,     53, 40, 36,     36, 55, 53,
    67, 54, 52,     52, 59, 67,     72, 56, 54,     54, 67, 72,
    68, 55, 56,     56, 72, 68,     60, 53, 55,     55, 68, 60,
    83, 67, 59,     59, 75, 83,     88, 72, 67,     67, 83, 88,
    84, 68, 72,     72, 88, 84,     76, 60, 68,     68, 84, 76,
    34, 18, 24,     24, 40, 34,     27, 11, 18,     18, 34, 27,
    30, 14, 11,     11, 27, 30,     38, 22, 14,     14, 30, 38,
    51, 34, 40,     40, 53, 51,     49, 27, 34,     34, 51, 49,
    42, 30, 27,     27, 49, 42,     44, 38, 30,     30, 42, 44,
    66, 51, 53,     53, 60, 66,     71, 49, 51,     51, 66, 71,
    62, 42, 49,     49, 71, 62,     58, 44, 42,     42, 62, 58,
    82, 66, 60,     60, 76, 82,     87, 71, 66,     66, 82, 87,
    78, 62, 71,     71, 87, 78,     74, 58, 62,     62, 78, 74
};
```

B.8. mg_spheres.c

```
/*
 *    mg_spheres.c      – model generator for sphere scenes
 */

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "mg_defs.h"
#include "mg_spheres.h"

/*
 *    static definitions
 */

#define CRAND (double) (drand() /   1.5)
#define YRAND (double) (drand() / 2.5) –0.2        /* (–.2,.2) */

/*
 *    scene parameters
 */

#define N   100
#define GRID     10
#define STACK   10
#define G_RANDgaussrand()/2.5
#define NLIGHT 4
#define SIZE      1.

enum {Uniform, Gaussian, Grid, Stack} distn;
enum {Sphere, Patch, Polygon, PointsPolygons} prim;
char picname[100] = "";
static int n;                            /* number of spheres */
static int ng, ns;                       /* number of spheres per grids, stacked grids */
static double tpct = 0.0, mpct = 0.0;    /* fraction transparent or reflective */
static int nt = 0, nm = 0;               /* number transparent or reflective */
static int constant = FALSE;             /* constant shading */
struct {                                 /* light source description */
      int              off;
      enum LightType  model;
} light[NLIGHT] = {
      0, Ambient,
      0, Point,
      0, Point,
      0, Point
};

/*
 *    viewing controls
 */
int xres = 512, yres = 512;
double camera = 6.;
```

```
main(argc, argv)
int argc;
char *argv[];
{
	int i, l;
	int back = FALSE;		/* if TRUE, ouput background polygon */
	double r;			/* radius of spheres */
	double gap, offset;		/* spacing for grids */
	double size = SIZE;		/* scale factor */
	xyz position, look, up, scale;
	double fov = 25.;
	double near = 1.;
	double far = 11.;
	rgb c;

	/*
	 * initialize options
	 */
	distn = Uniform;
	prim = Sphere;
	while (--argc > 0) {
		argv++;
		if (argv[0][0] == '-')
			switch (argv[0][1]) {
			case 'b':	/* output background polygon */
				back = TRUE;
				break;
			case 'c':	/* constant shader */
				if (tpct > 0. || mpct > 0.) {
					warning("-c overides -m and -t\n");
					tpct = mpct = 0.;
				}
				constant = TRUE;
				break;
			case 'd':	/* spatial distribution */
				if (argv[0][2] == ' ')
					error("No spatial distribution specified.\n");
				switch (argv[0][2]) {
				case 'g':
					/* usage:  -dg n
					 * n x n grid
					 */
					if (n)
						error("Only use one of -n, -dg, -ds, -do\n");
					distn = Grid;
					if (argc < 2 || argv[1][0] == '-')
						error("Usage:  -dg n\n");
					argv++; argc--;
					ng = atoi(argv[0]);
					if (ng < 1)
						error("Usage:  -dg n\n");
					n = ng * ng;
					break;
```

```
                case 'G':
                        distn = Gaussian;
                        break;
                case 's':
                        /*
                        * usage:  -ds n m
                        * n x n grids, stacked m deep
                        */
                        distn = Stack;
                        if (n)
                                error("Only use one of -n, -dg, -ds, -do");
                        if (argc < 3 || argv[1][0] == '-' || argv[2][0] == '-')
                                error("Usage:  -ds n m\n");
                        argv++; argc--;
                        ng = atoi(argv[0]);
                        if (ng < 1)
                                error("Usage:  -ds n m\n");
                        n = ng * ng;
                        argv++; argc--;
                        ns = atoi(argv[0]);
                        if (ns < 1)
                                error("Usage:  -ds n m\n");
                        break;
                case 'u':   /* Default: scattered with uniform distribution */
                        break;
                default:
                        error ("Spatial distribution must be one of g, G, o, s, u\n");
                }
                break;
        case 'f':   /* field of view */
                if (argc < 2 || argv[1][0] == '-')
                        error("No angle specified for -f.\n");
                argv++; argc--;
                fov = atof(argv[0]);
                break;
        case 'L':   /* turn off light */
                if (argv[0][2] == ' ')
                        error("Usage:  -Li, 0 <= i <= %d\n", NLIGHT-1);
                l = atoi(&argv[0][2]);
                if (l < 0 || l >= NLIGHT)
                        warning("Usage:  -Li, 0 <= i <= %d\n", NLIGHT-1);
                else
                        ++light[l].off;
                break;
        case 'm':
                /*
                * usage:  -m [percent]
                * make some spheres metallic; default = 20%
                */
                if (constant) {
                        warning("-c overides -m and -t\n");
                        if (argc > 1 && argv[1][0] != '-') {
                                argv++; argc--;
                        }
```

```
			break;
		}
		if (argc > 1 && argv[1][0] != '−') {
			argv++; argc−−;
			mpct = atof(argv[0]);
			if (mpct < 0. || mpct > 1.)
				error("−m argument must be between 0 and 1.0);
		}
		else
			mpct = 0.2;
		if (mpct + tpct > 1.)
			error("Total metallic and transparent must be <=1.0);
		break;
	case 'n':
		if (argc < 2 || argv[1][0] == '−')
			error("No number specified for −n\n");
		argv++; argc−−;
		if (ng)
			error("Use only one of −n, −dg, −ds, −do");
		n = atoi(argv[0]);
		if (n < 0)
			error("Choose n >= 0\n");
		break;
	case 'N':	/* picture name */
		if(argc < 2 || argv[1][0] == '−')
			error("No name\n");
		else {
			strcpy(picname, argv[1]);
			argv++; argc−−;
		}
		break;
	case 'p':	/* type of primitive */
		if(argv[0][2] == ' ')
			error("No primitive type specified for −p.\n");
		switch (argv[0][2]) {
		case 'b':
			prim = Patch;
			break;
		case 'p':
			prim = Polygon;
			break;
		case 'P':
			prim = PointsPolygons;
			break;
		case 's':
			prim = Sphere;
			break;
		default:
			error("Primitive type must be one of b, p, or s.");
		}
		break;
	case 'r':	/* resolution */
		if (argc < 3 || argv[1][0] == '−' || argv[2][0] == '−')
			error("Usage:  −r xres yres\n");
```

```
			argv++; argc−−;
			xres = atoi(argv[0]);
			argv++; argc−−;
			yres = atoi(argv[0]);
			break;
		case 's':	/* size factor */
			if (argc < 2 || argv[1][0] == '−')
				error("No size factor specified for −s.\n");
			argv++; argc−−;
			size = atof(argv[0]);
			break;
		case 't':
		/*
		* usage:  −t [percent]
		* make some spheres transparent; default = 20%
		*/
			if (constant) {
				warning("−c overides −m and −t\n");
				if (argc > 1 && argv[1][0] != '−') {
					argv++; argc−−;
				}
				break;
			}
			if (argc > 1 && argv[1][0] != '−') {
				argv++; argc−−;
				tpct = atof(argv[0]);
				if (tpct < 0.0 || tpct > 1.0)
					error("−t argument must be between 0 and 1.0);
			}
			else
				tpct = 0.2;
			if (mpct + tpct > 1.0)
				error("Total metallic and transparent must be <=1.0);
			break;
		default:
			error("Unrecognized option %s\n", argv[0]);
		}
}

/*	set up frame */

if (!n)
	n = N;
r = size / sqrt((double) n);
if (distn != Gaussian && distn != Uniform)
	gap = (0.4 * size) / ng;
if (!strcmp(picname, ""))
	strcpy(picname,"spheres");
OutputBegin(picname);
OutputResolution(xres, yres);
position[0] = 0.; position[1] = 0.; position[2] = −camera;
look[0] = 0.; look[1] = 0.; look[2] = 0.;
up[0] = 0.; up[1] = 1.; up[2] = −camera;
```

```
	if (distn == Stack)
		far = camera + 2. * r * (double) ns + ((double) ns − 1.) * gap;
	else
		far = camera + r + 4. * size;
	if (back)
		far += 1.;
	OutputViewpoint(position, look, up, fov, near, far);
	c[0] = c[1] = c[2] = 1.;
	if (!light[0].off)			/* white ambient light */
		OutputLight(light[0].model, 0.3, c, NULL, NULL);
	if (!light[1].off) {			/* white point light source */
		position[0] = 1.; position[1] = −1.; position[2] = −6.;
		OutputLight(light[1].model, 24., c, position, NULL);
	}
	c[0] = 1.;  c[1] = 0.5; c[2] = 0.5;
	if (!light[2].off) {			/* red point light source */
		position[0] = 3.; position[1] = 4.; position[2] = −4.;
		OutputLight(light[2].model, 20., c, position, NULL);
	}
	c[0] = 0.5;  c[1] = 1.; c[2] = 0.5;
	if (!light[3].off) {			/* green point light source */
		position[0] = 6.; position[1] = −4.; position[2] = −2.;
		OutputLight(light[3].model, 16., c, position, NULL);
	}
	c[0] = c[1] = c[2] = 1.;

	/*	generate model	*/

	OutputOpacity(c, 0);
	if (back)
		background(far − camera − 0.5);
	if (constant)
		OutputSurface(Constant);
	else
		OutputSurface(Plastic, 0.33, 1., 1., 0.1);
	switch (distn) {
	case Gaussian:
	case Uniform:
		makerandom(n, r);
		break;
	case Grid:
		makegrid(ng, r, gap);
		break;
	case Stack:
		for (i = 0; i < ns; i++) {
			makegrid(ng, r, gap);
			OutputTranslate(0., 0., 2. * r + gap);
		}
		break;
	}
	OutputEnd();
	report(r, size, fov, back);
}
```

```
makerandom(n, r)        /* generate randomly-scattered spheres */
int n;
double r;
{
        int i;
        xyz position;

        for (i = 0; i < n; i++) {
                setsurface();
                if (distn == Gaussian) {
                        position[0] = G_RAND * 1.2;
                        position[1] = G_RAND * 1.2;
                        position[2] = G_RAND * 1.2;
                }
                else if (distn == Uniform) {
                        position[0] = urand() * 1.2;
                        position[1] = urand() * 1.2;
                        position[2] = urand() * 1.2;
                }
                sphere_at(r, position);
        }
}

makegrid(ng, r, gap)            /* generate spheres in a grid */
int ng;
double r, gap;
{
        int i, j;
        xyz position;
        double offset, start;

        offset = (2. / sqrt((double) n)) + (0.4 / ng);
        start = (((double) ng - 1.)/ 2.) * offset;
        position[1] = start;
        position [2] = 0.;
        for (i = 0; i < ng; i++) {
                position[0] = - start;
                for (j = 0; j < ng; j++) {
                      setsurface();
                      sphere_at(r, position);
                      position[0] += offset;
                }
                position[1] -= offset;
        }
}
```

```
sphere_at(r, position)
double r;
xyz position;
{
        OutputSave();
        OutputTranslate(position[0], position[1], position[2]);
        switch (prim) {
        case Sphere:
                OutputSphere(r);
                break;
        case Patch:
                beziersphere(r);
                break;
        case Polygon:
        case PointsPolygons:
                polysphere(r);
                break;
        }
        OutputRestore();
}

#define CMAX 20
double r_save = −1, c[CMAX];
polysphere(r)                   /* generate a set of triangles to approximate a sphere */
double r;
{
        int i, j, k, vlist[4*NP_POLY], nvertices[NP_POLY];
        static int init = 0;
        xyz v;

        if (!init) {
                for (i = 0; i < NP_POLY; ++i)
                        nvertices[i] = 3;
                ++init;
        }
        if (r != r_save) {
                r_save = r;
                for (i = 0; i < NC_POLY; ++i)
                        c[i] = c_poly[i] * r;
                OutputVertexReset();
                for(i = 0; i < NV_POLY; ++i) {
                        v[0] = c[v_poly[i][0]];
                        v[1] = c[v_poly[i][1]];
                        v[2] = c[v_poly[i][2]];
                        OutputVertex(v);
                        k = OutputVertexNormal(v);
                }
        }
        if (prim == Polygon)
                for (i = 0; i < NP_POLY; ++i) {
                        for (j = 0; j < 3; ++j)
                                vlist[j] = p_poly[i][j];
                        OutputPolygon(3, vlist, TRUE, FALSE, FALSE);
                }
```

```
	else if (prim == PointsPolygons) {
		for (i = 0, k = 0; i < NP_POLY; ++i) {
			for (j = 0; j < 3; ++j)
				vlist[k++] = p_poly[i][j];
		}
		OutputPointsPolygons(NP_POLY, nvertices, vlist, TRUE, FALSE, FALSE);
	}
}

beziersphere(r)		/* generate a set of bezier patches to approximate a sphere */
double r;
{
	int i, j, vlist[16];
	xyz v;

	if (r != r_save) {
		r_save = r;
		for (i = 0; i < NC_BEZIER; ++i)
			c[i] = c_bezier[i] * r;
		OutputVertexReset();
		for(i = 0; i < NV_BEZIER; ++i) {
			v[0] = c[v_bezier[i][0]];
			v[1] = c[v_bezier[i][1]];
			v[2] = c[v_bezier[i][2]];
			OutputVertex(v);
		}
	}
	for (i = 0; i < NP_BEZIER; ++i) {
		for (j = 0; j < 16; ++j)
			vlist[j] = p_bezier[i][j];
		OutputPatch(Bezier, vlist);
	}
}

setsurface()	/* generate surface characteristics for a sphere */
{
	rgb color;
	double a;
	static int mlast = FALSE;
	static int tlast = FALSE;
	int tflag, mflag;
	rgb tcolor = { 0.6, 0.6, 0.6 };
	rgb ocolor = { 1.0, 1.0, 1.0 };

	/* a sphere may be opaque plastic, transparent, or reflective */
	color[0] = CRAND; color[1] = CRAND; color[2] = CRAND;
	OutputColor(&color);
	if (tpct < EPSILON && mpct < EPSILON)
		return;
	tflag = mflag = FALSE;
	a = drand();
```

```
	if (tpct > 0.0 && a <= tpct) {
		tflag = TRUE;
		++nt;
	}
	else if (mpct > 0.0 && a <= tpct + mpct) {
		mflag = TRUE;
		++nm;
	}
	if (tflag != tlast) {
		if (tflag)
			OutputOpacity(tcolor, 1.5);
		else
			OutputOpacity(ocolor, 0.);
		OutputSurface(Plastic, 0.33, 1., 1., 0.1);
		tlast = tflag;
	}
	if (mflag != mlast) {
		if (mflag)
			OutputSurface(Reflective, 1., 0., 1., 0.1);
		else
			OutputSurface(Plastic, 0.33, 1., 1., 0.1);
		mlast = mflag;
	}
}

background(d)		/* set up background polygon */
double d;
{
	xyz v;
	int vlist[4];
	rgb color;

	if (constant) {
		color[0] = CRAND; color[1] = CRAND; color[2] = CRAND;
		OutputColor(&color);
		OutputSurface(Constant);
	}
	else
		OutputTxSurface(1, 0.33, 1.);
	v[2] = d;
	v[0] = –5.;
	v[1] = 5.;
	OutputVertex(v);
	v[0] = 5.;
	OutputVertex(v);
	v[1] = –5.;
	OutputVertex(v);
	v[0] = –5.;
	OutputVertex(v);
	vlist[0] = 0; vlist[1] = 1; vlist[2] = 2; vlist[3] = 3;
	OutputPolygon(4, vlist, FALSE, FALSE, FALSE);
}
```

```
report(r, s, fov, back)
double r, s, fov;
int back;
{
	int i;

	printf("Model name:  %s\n", picname);
	if (ns > 0)
		n *= ns;
	printf("Number of spheres:  %d\n", n);
	printf("Spatial distribution:  ");
	switch(distn) {
	case Uniform:
		printf("scattered, uniform distribution\n");
		break;
	case Gaussian:
		printf("scattered, gaussian distribution\n");
		break;
	case Grid:
		printf("%d by %d grid\n", ng, ng);
		break;
	case Stack:
		printf("%d by %d grids, stacked %d deep\n", ng, ng, ns);
		break;
	}
	printf("Radius: %g (scale factor = %g)\n", r, s);
	printf("Output primitive:  ");
	switch(prim) {
	case Patch:
		printf("bezier patch\n");
		break;
	case Polygon:
		printf("polygon\n");
		break;
	case PointsPolygons:
		printf("points-polygon\n");
		break;
	case Sphere:
		printf("sphere\n");
		break;
	}
	if (back)
		printf("Textured background polygon\n");
	printf("Viewing:  field-of-view = %g, resolution = %d x %d\n",
	fov, xres, yres);
	printf("Lighting:\n");
	for (i = 0; i < NLIGHT; ++i) {
		printf("    light source %d is %s, ", i, light[i].off?"off":"on");
		switch(light[i].model) {
		case Ambient:
			printf("ambient\n");
			break;
		case Point:
			printf("point\n");
```

```
                break;
            }
        }
        if (constant)
            printf("Constant shading\n");
        else {
            printf("Surface:  goals were ");
            if (tpct > 0)
                printf("%g%% transparent, ", 100. * tpct);
            if (mpct > 0.)
                printf("%g%% mpct, ", 100. * mpct);
            if (tpct + mpct < 1.0)
                printf("%g%% plastic", 100. * (1. - (mpct + tpct)));
            printf("\n");
            printf("generated ");
            if (nt > 0)
                printf("%d transparent, ", nt);
            if (nm > 0)
                printf("%d metallic, ", nm);
            if (nt + nm < n)
                printf("%d plastic", n - (nm + nt));
            printf("\n");
        }
}
```

B.9. mg_terrain.c

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "mg_defs.h"

/*
 *   mg_terrain.c
 *  generates terrain models with optional texture
 */

/*
 *   terrain grid definitions and variables
 */

#define MAKEXYZ(d,s) {d[0] = s.x; d[1] = s.y; d[2] = s.z; }
#define MAXGRID     100
#define NPTS    20
#define GSIZE   30
#define MAXNSAMP 40
#define MAXDGRID (MAXGRID/5)
#define DX 2
#define DY 2
double dist[MAXNSAMP];
int point[MAXNSAMP];
int minnsamp = 5, maxnsamp = 30;
double mrad = 5., super = 2.5;
double gridsamp();
int extra = TRUE;
float  dx = DX, dy = DY;
double xmax = 30., xmin = –30., ymax = 30., ymin = –30.;
double xlen, ylen;
int npts = NPTS;
int nx, ny;

struct pointstruct {
      float  x, y, z;
} grid[MAXGRID][MAXGRID], p[] =
{
      –27.8000,  –1.60545,  2.10363,
      1.13818,   34.4436,   5.55454,
      29.4273,   0.0218183, 0.627272,
      –13.2364,  22.4727,   5.70909,
      11.4727,   22.6,      6.52727,
      –21.4364,  –23.5273,  4.25455,
      –0.101818, –24.8909,  2.23636,
      20.9527,   –24.5455,  3.09091,
      –19.2546,  –11.0727,  0.765452,
      –0.345454, –11.6109,  0.589092,
      19.3927,   –10.8654,  0.830912,
      –18.2545,  –1.69093,  –0.036361,
      –3.832e–08,      –0.200005, 0.0181852,
```

```
	20.4364,	–1.61818,	0.0727231,
	–19.9655,	5.27273,	0.981818,
	0.30909,	8.89092,	1.52727,
	20.8000,	8.41819,	2.07273,
	–23.3818,	22.6909,	5.9091,
	0.981791,	21.8727,	6.25455,
	23.6000,	21.6545,	7.16364
};

enum { Patch, Polygon, PointsPolygons } prim;
int n;

/*
 *	controls for scene variation
 */
#define CRAND (double) (drand() / 1.5)
#define NT 10			/* number of pieces of terrain [or number per stack] */
#define ZGAP 5.			/* distance between pieces of terrain */
#define TXSCALE	2.	/* scale factor for s, t coordinates */
#define PLASTICSURFACE Plastic, 0.33, 1., 1., 0.1
#define NLIGHT 2

struct {
	int		off;
	enum LightType	model;
} light[NLIGHT] = {
	0,	Ambient,
	0,	Distant
};
rgb tcolor[NT];
char picname[100];

/*
 *	viewing specifications
 */

#define XRES 640
#define YRES 480
#define FOV 45.
#define NEAR 1.
#define FAR 800.
int xres = XRES, yres = YRES;
double fov = FOV, near = NEAR, far = FAR;
int frame = 0;
xyz cam[5] =
{
	31.,	–30.5,	5.,
	3.5,	–100.,	62.,
	1.,	–81.,	110.5,
	3.5,	–100.,	62.,
	1.,	–81.,	110.5
};
```

```
xyz look[5] =
{
	0.,	0.,	0.,
	−1.,	35.,	8.,
	0.,	17.,	2.7,
	−1.,	35.,	8.,
	0.,	17.,	2.7
};
xyz up[5] =
{
	33.,	−32.3,		6.4,
	3.5,	−100.,		63.,
	1.,	−80.,		111.2,
	3.5,	−100.,		63.,
	1.,	−80.,		111.2
};

main (argc, argv)
int argc;
char *argv[];
{
	int i, j, k, l, m;
	struct pointstruct cat[4][4], bez[4][4];
	int polylist[4*GSIZE*GSIZE], nvertices[GSIZE*GSIZE];
	int patchlist[16];
	xyz p, dir;
	rgb c;
	int nt = 0;			/* number of textured surfaces */
	int textures = 0;		/* number of textures per surface */
	int bottomup = FALSE;		/* direction to assign textures */
	enum SurfaceType surf = Plastic;
	double s[4], t[4];
	double zgap;

	prim = Patch;
	while (−−argc > 0) {
		argv++;
		if (argv[0][0] == '−')
			switch (argv[0][1]) {
			case 'c':	/* constant shader */
				if (nt)
					warning("−c overides −t and −T\n");
				surf = Constant;
				nt = textures = 0;
				break;
			case 'f':	/* field of view */
				if (argc < 2 || argv[1][0] == '−')
					error("No angle specified for −f.\n");
				argv++; argc−−;
				fov = atof(argv[0]);
				break;
			case 'L':	/* turn off light */
				if (argv[0][2] == ' ')
					error("Usage:  −Li, 0 <= i <= %d\n", NLIGHT−1);
```

```
        l = atoi(&argv[0][2]);
        if (l < 0 || l >= NLIGHT)
                error("Usage:  −Li, 0 <= i <= %d\n", NLIGHT−1);
        else
                ++light[l].off;
        break;
case 'n':
        if (argc < 2 || argv[1][0] == '−')
                error("No number specified for −n\n");
        argv++; argc−−;
        n = atoi(argv[0]);
        if (n < 1 || n > NT)
                error("Choose n between 1 and 10.\n");
        break;
case 'N':   /* picture name */
        if(argc < 2 || argv[1][0] == '−')
                error("No name.\n");
        else {
                strcpy(picname, argv[1]);
                argv++; argc−−;
        }
        break;
case 'p':   /* type of primitive */
        if(argv[0][2] == ' ')
                error("No primitive type specified for −p.\n");
        switch (argv[0][2]) {
        case 'b':
                prim = Patch;
                break;
        case 'p':
                prim = Polygon;
                extra = FALSE;
                break;
        case 'P':
                prim = PointsPolygons;
                for (i = 0; i < GSIZE*GSIZE; ++i)
                        nvertices[i] = 4;
                extra = FALSE;
                break;
        default:
                error("Primitive type must be one of b, p, or m.\n");
        }
        break;
case 'r':   /* resolution */
        if (argc < 3 || argv[1][0] == '−' || argv[2][0] == '−')
                error("Usage:  −r xres yres\n");
        argv++; argc−−;
        xres = atoi(argv[0]);
        argv++; argc−−;
        yres = atoi(argv[0]);
        break;
case 'T':   /* textures */
 /*
* usage:  −T m n
```

```
		 * same as -t, but assign textures from the bottom up
		 * if m < total number of surfaces
		 */
		bottomup = TRUE;
		case 't':	/* textures */
		 /*
		 * usage:  -t m n
		 * apply textures to m pieces of terrain,
		 * n textures per surface
		 *
		 */
			if (argc < 3 || argv[1][0] == '-' || argv[2][0] == '-')
				error("usage: -[tT] m n\n");
			argv++; argc--;
			nt = atoi(argv[0]);
			argv++; argc--;
			textures = atoi(argv[0]);
			if (surf == Constant)
				warning("-c overides -t and -T\n");
				nt = textures = 0;
			break;
		case 'v':	/* specify one of three views */
		 /*
		 *	frame 0	"landscape"
		 *	frame 1	front
		 *	frame 2	top
		 *	frame 3	front, more surfaces visible
		 *	frame 4	top, more surfaces visible
		 */
			if(argv[0][2] == ' ')
				error("Specify view 0 through 4.\n");
			frame = atoi(&argv[0][2]);
			if (frame < 0 || frame > 4)
				error("Specify view 0 through 4.\n");
			break;
		default:
			error("Unrecognized option %s\n", argv[0]);
		}
}
if (!n)
	if (frame == 0)
		n = 1;
	else
		n = NT;
if (n > nt)
	for (m = 0; m < NT; ++m) {
		tcolor[m][0]=CRAND;
		tcolor[m][1]=CRAND;
		tcolor[m][2]=CRAND;
	}
if (prim == Patch)
	for (i = 0; i < 16; ++i)
		patchlist[i] = i;
else if (prim == Polygon)
```

```
        polylist[0] = 0; polylist[1] = 1; polylist[2] = 2; polylist[3] = 3;

if (!strcmp(picname, ""))
        strcpy(picname,"terrain");
OutputBegin(picname);
OutputResolution(xres, yres);
OutputViewpoint(cam[frame], look[frame], up[frame], fov, near, far);

c[0] = c[1] = c[2] = 1.;
if (!light[0].off)                /* white ambient light */
        OutputLight(light[0].model, 0.5, c, NULL, NULL);
if (!light[1].off) {              /* white distant light source */
        p[0] = 80.; p[1] = 50.; p[2] = 20.;
        dir[0] = dir[1] = dir[2] = 0.;
        OutputLight(light[1].model, 1., c, p, dir);
}
c[0] = c[1] = c[2] = 1.;
OutputOpacity(c, 0);
mkgrid();
xlen = xmax − xmin;
ylen = ymax − ymin;
if (frame < 4) {
        zgap = ZGAP;
        OutputTranslate(0., 0., (double) (n–1) * zgap);
}
else {
        zgap = (NT * ZGAP )/ (double) (n–1);
        OutputTranslate(0., 0., (double) (n–1) * zgap);
}
switch(prim) {
case Polygon:
case PointsPolygons:
        for (m = 0; m < n; m++) {
                OutputSave();
                if ((!bottomup && m < nt) || (bottomup &&  m >= n–nt))
                        OutputTxSurface(textures, 0.33, 1.);
                else if (surf == Plastic) {
                        OutputSurface(PLASTICSURFACE);
                        OutputColor(tcolor[m]);
                }
                else if (surf == Constant) {
                        OutputSurface(Constant);
                        OutputColor(tcolor[m]);
                }
                if (frame == 3)
                        OutputTranslate(–15. + 15. * (double) (m%3), 0., 0.);
                if (frame == 4)
                        OutputTranslate(–25. + 25. * (double) (m%3), 0., 0.);
                OutputTranslate(0., 0., – (double) m * zgap);
                if (prim == Polygon) {
                        for(j = 0; j < ny–1; j++) {
                                for(i = 0; i < nx–1; i++) {
                                        OutputVertexReset();
                                        polyvtx(grid[i][j], m < nt);
```

```
                        polyvtx(grid[i][j+1], m < nt);
                        polyvtx(grid[i+1][j+1], m < nt);
                        k =polyvtx(grid[i+1][j], m < nt);
                        if ((!bottomup && m < nt) || (bottomup &&  m >= n–nt))
                            OutputPolygon(4, polylist, FALSE, FALSE, TRUE);
                        else
                            OutputPolygon(4, polylist, FALSE, FALSE, FALSE);
                    }
                }
            }
            else if (prim == PointsPolygons) {
                OutputVertexReset();
                for(j = 0; j < ny; j++)
                    for(i = 0; i < nx; i++)
                        k = polyvtx(grid[i][j], m < nt);
                    k = 0;
                    for(j = 0; j < ny–1; j++) {
                        for(i = 0; i < nx–1; i++) {
                            polylist[k++] =   i +     j*ny;
                            polylist[k++] =   i + (j+1)*ny;
                            polylist[k++] = i+1 + (j+1)*ny;
                            polylist[k++] = i+1 +     j*ny;
                        }
                    }
                    if ((!bottomup && m < nt) || (bottomup &&  m >= n–nt))
                        OutputPointsPolygons(GSIZE*GSIZE, nvertices, polylist,
                        FALSE, FALSE, TRUE);
                    else
                        OutputPointsPolygons(GSIZE*GSIZE, nvertices, polylist,
                           FALSE, FALSE, FALSE);
            }
            OutputRestore();
        }
        break;
    case Patch:
        for(m = 0; m < n; m++) {
            OutputSave();
            if ((!bottomup && m < nt) || (bottomup &&  m >= n–nt))
                OutputTxSurface(textures, 0.33, 1.);
            else if (surf == Plastic) {
                OutputSurface(PLASTICSURFACE);
                OutputColor(tcolor[m]);
            }
            else if (surf == Constant) {
                OutputSurface(Constant);
                OutputColor(tcolor[m]);
            }
            if (frame == 3)
                OutputTranslate(–15. + 15. * (double) (m%3), 0., 0.);
            if (frame == 4)
                OutputTranslate(–25. + 25. * (double) (m%3), 0., 0.);
            OutputTranslate(0., 0., –(double) m * zgap);
            for(j = 1; j < ny–2; j += 1) {
                for(i = 1; i < nx–2; i += 1) {
```

```
                                OutputVertexReset();
                                for(l = i–1; l < i+3; l++) {
                                        for(k = j–1; k < j+3; k++) {
                                                cat[k–j+1][l–i+1] = grid[l][k];
                                        }
                                }
                                PatchCatToBez(bez, cat);
                                for(l = 0; l < 4; l++) {
                                        for(k = 0; k < 4; k++) {
                                                MAKEXYZ(p,bez[l][k]);
                                                OutputVertex(p);
                                        }
                                }
                                s[0] = TXSCALE * (bez[0][0].x − xmin) / xlen ;
                                t[0] = TXSCALE * (bez[0][0].y − ymin) / ylen ;
                                s[1] = TXSCALE * (bez[0][3].x − xmin) / xlen ;
                                t[1] = TXSCALE * (bez[0][3].y − ymin) / ylen ;
                                s[2] = TXSCALE * (bez[3][0].x − xmin) / xlen ;
                                t[2] = TXSCALE * (bez[3][0].y − ymin) / ylen ;
                                s[3] = TXSCALE * (bez[3][3].x − xmin) / xlen ;
                                t[3] = TXSCALE * (bez[3][3].y − ymin) / ylen ;
                                if ((!bottomup && m < nt) || (bottomup &&  m >= n–nt))
                                        OutputTxCoords(s, t);
                                OutputPatch(Bezier, patchlist);
                        }
                }
                OutputRestore();
        }
        break;
    }
    OutputEnd();
    report(nt, textures, bottomup, surf);
}

/*
 *      geometry routines
 *
 *  mkgrid() and gridsamp() adapted from code
 *          written by Bill Reeves (Pixar)
 */
mkgrid() {
    register int i, j;
    float x, y, z;

    nx = (xmax–xmin)/dx+1.5;
    ny = (ymax–ymin)/dy+1.5;
    if (extra) {
        nx += 2;
        ny += 2;
    }
    for(i = 0; i < nx; i++)
        for(j = 0; j < ny; j++) {
            if(extra) {
                if(i == 0)
```

```
                        x = xmin−dx;
                    else if(i == nx−1)
                        x = xmax+dx;
                    else
                        x = (xmax−xmin)*(i−1.)/(nx−3)+xmin;
                    if(j == 0)
                        y = ymin−dy;
                    else if(j == ny−1)
                        y = ymax+dy;
                    else
                        y = (ymax−ymin)*(j−1.)/(ny−3)+ymin;
                }
                else {
                    x = (xmax−xmin)*(i+0.)/(nx−1)+xmin;
                    y = (ymax−ymin)*(j+0.)/(ny−1)+ymin;
                }
                z = gridsamp(x, y);
                grid[i][j].x = x;
                grid[i][j].y = y;
                grid[i][j].z = z;
            }
}

double gridsamp(x, y)
    double x, y;
{
    register int k, l;
    register struct pointstruct *pp;
    double z, rad, d, dx, dy, denom, dsum;

    for(k = 0; k < maxnsamp; k++)
        dist[k] = 1e20;
    for(pp = p; pp < &p[npts]; pp++) {
        dx = (x−pp−>x);
        dy = (y−pp−>y);
        d = dx*dx+dy*dy;
        if(d >= dist[maxnsamp−1])
            continue;
        for(k = 0; k < maxnsamp; k++)
            if(d < dist[k]) {
                for(l = maxnsamp−1; l > k; l−−) {
                    dist[l] = dist[l−1];
                    point[l] = point[l−1];
                }
                dist[k] = d;
                point[k] = pp−p;
                break;
            }
    }
    if(dist[0] > 0.0) {
        z = 0.; dsum = 0;
        if((rad = mrad) > 0) {
            d = sqrt(dist[minnsamp]);
            if(d > rad)
```

```
				rad = d;
			}
			for(k = 0; k < maxnsamp; k++) {
				denom = sqrt(dist[k]);
				if(rad−denom < 0.)
					denom = 0.;
				else {
					denom = (rad−denom)/(rad*denom);
					denom = pow(denom, super/2.);
				}

				z += p[point[k]].z*denom;
				dsum += denom;
			}
			z /= dsum;
	} else
		z = p[point[0]].z;
	return(z);
}

polyvtx(p, tflag)
struct pointstruct p;
int tflag;
{
	xyz v;

	MAKEXYZ(v, p);
	OutputVertex(v);
	if (tflag)
		OutputVertexST(TXSCALE*(v[0]−xmin)/xlen, (TXSCALE*v[1]−ymin)/ylen);
}

PatchCatToBez(bezier, catrom)
struct pointstruct catrom[4][4], bezier[4][4];
{
	int i, j;
	Matrix cx, cy, cz, bx, by, bz;

	for (i = 0; i < 4; i++)
		for (j = 0; j < 4; j++) {
			cx[i][j] = catrom[i][j].x;
			cy[i][j] = catrom[i][j].y;
			cz[i][j] = catrom[i][j].z;
		}
	CatromToBez(bx, cx);
	CatromToBez(by, cy);
	CatromToBez(bz, cz);
	for (i = 0; i < 4; i++)
		for (j = 0; j < 4; j++) {
			bezier[i][j].x = bx[i][j];
			bezier[i][j].y = by[i][j];
			bezier[i][j].z = bz[i][j];
		}
}
```

```
report(nt, textures, bottomup, surf)
int nt, textures, bottomup;
enum SurfaceType surf;
{
	int i;

	printf("Model name:  %s\n", picname);
	printf("Number of terrain surfaces:  %d\n", n);
	printf("Output primitive:  ");
	switch(prim) {
	case Patch:
		printf("bezier patch\n");
		break;
	case Polygon:
		printf("polygon\n");
		break;
	case PointsPolygons:
		printf("points-polygon\n");
		break;
	}
	printf("View number: %d\n", frame);
	printf("Viewing:  field-of-view = %f, resolution = %d x %d\n",
		fov, xres, yres);
	printf("Lighting:\n");
	for (i = 0; i < NLIGHT; ++i) {
		printf("    light source %d is %s, ", i, light[i].off?"off":"on");
		switch(light[i].model) {
		case Ambient:
			printf("ambient\n");
			break;
		case Distant:
			printf("distant\n");
			break;
		}
	}
	printf("Surface:\n");
	if (nt) {
		printf("%d textured surface%s, %d texture%s per surface\n",
			nt, nt>1?"s":"", textures, textures>1?"s":"");
		printf("textures assigned from the %s\n",
			bottomup?"bottom up":"top down");
	}
	if (nt < n) {
		printf("%d non-textured surface%s with %s shading model\n",
			n - nt, (n - nt) > 1?"s":"",
			surf==Constant?"constant":"plastic");
	}
}
```