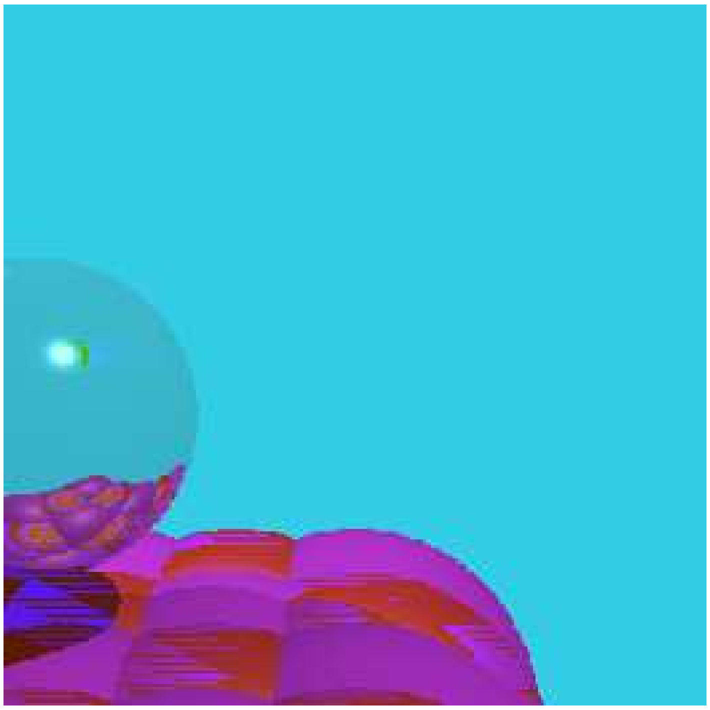
Layered Depth Images - Explored In Depth

Leon James Derman

Abstract

This paper describes the concept of a layered depth image and its implementation in the paper by Shade et al [1]. Motivation for this technique is discussed. Finally, the author's implementation of a layered depth image constructor and renderer is discussed in detail.

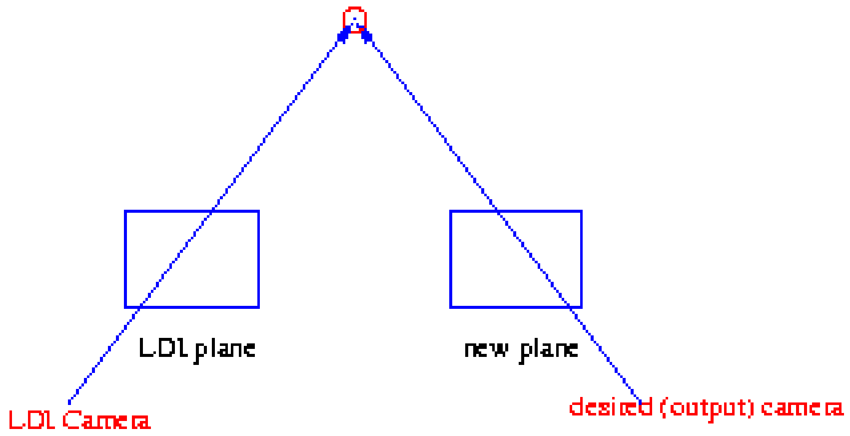


1 INTRODUCTION

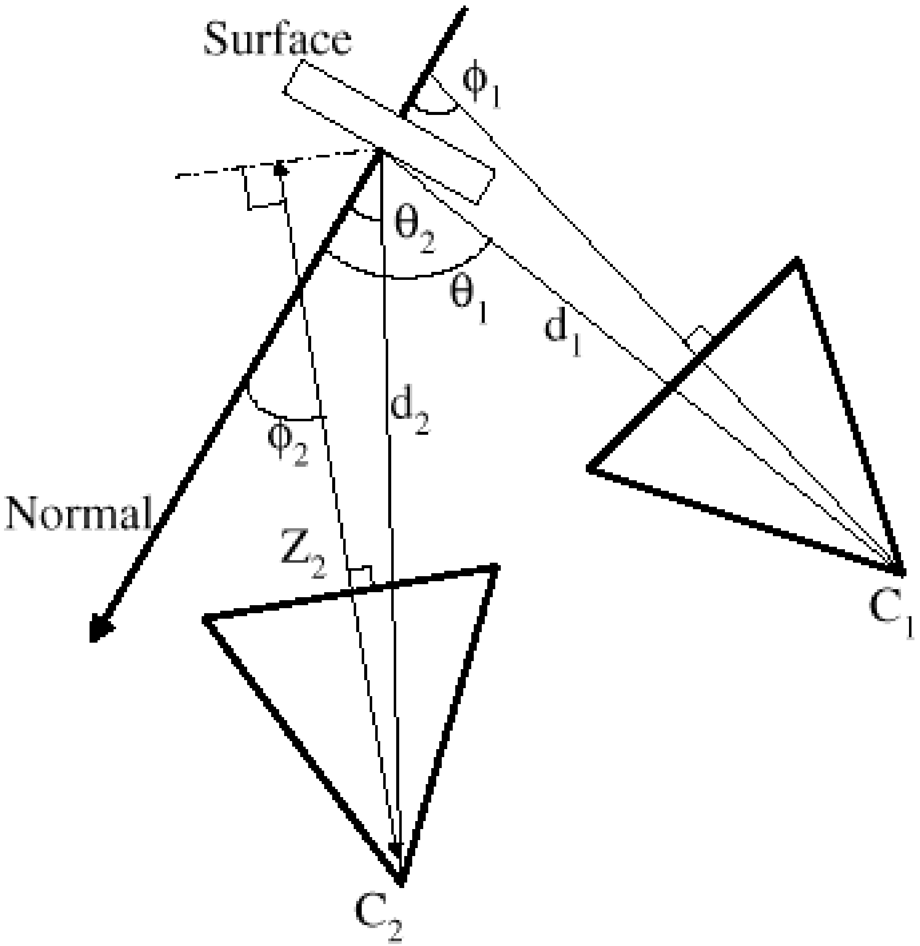
Every type of renderer faces a fundamental design decision - namely, the trade-off between space efficiency and time efficiency. Scenes which are stored as polygon vertices, surface colors, etc. have a fairly compact representation but require a large amount of resources to render realistically (by ray-tracing, for instance). Whereas scenes stored as a database of pre-rendered images from various viewpoints require considerably more resources to store, but can be quickly rendered.

For this project, an approach is described that trades some space efficiency for a great increase in time efficiency. This approach, called layered depth images, was first described in a paper by Shade et al [1]. The basic idea behind layered depth images is to augment what we typically think of as a two dimensional image (2D array of pixel colorings) by adding, where appropriate, additional pixels that are occluded by the current viewpoint. Depth is preserved for each pixel, hence a novel view can quickly be reconstructed by warping the pixels in for the layered depth image viewpoint to the desired viewpoint. Consider the case where one has modified a ray-tracer to output depth per pixel in addition to the pixel's color value. Given several renderings of the same scene from different viewpoints using the above ray-tracer, we can pick a viewpoint as the layered depth image viewpoint (or camera position) and warp the pixels from each of the given images to the desired layered depth image viewpoint, creating a layered depth image. This newly created layered depth image can then be rendered from novel viewpoints by warping from the layered depth image viewpoint to the desired (output) viewpoint.

The warping process involves several steps. We assume a general planar-pinhole camera model, so that each point in our image represents a ray that starts at the image's center of projection, passes through the given point and either intersects an object in the scene (where the background is also considered an object). Here image refers to the layered depth image, which is a two dimensional array of layered depth pixels. Each index into this array represents the color of the pixel of a (u,v) coordinate in the image plane created by the layered depth image's default camera projection.



As can be seen above, warping consists of converting a pixel from LDI image space to world space to desired viewpoint image space. Since we are mapping to a different viewpoint, the relative size of each pixel from the LDI image may increase or decrease depending upon the angles θ_1 and θ_2.



[*]

The exact formula used to calculate the splat size may be determined by the following equation:

$$\text{size} = \frac{(d_1)^2 \cos(\theta_2) \text{res}_2 \tan(\text{fov}_1/2)}{(d_2)^2 \cos(\theta_1) \text{res}_1 \tan(\text{fov}_2/2)}$$

Where d_1 is the distance from the sampled surface point to the LDI camera, fov_1 is the field of view of the LDI camera, res_1 is:

$$\frac{1}{w_1 h_1}$$

(where w_1 amd h_1 are the width and height, respectively of the LDI image, and θ_1 is the angle between the surface normal and the line of sight of the LDI camera.

So, once the layered depth image is constructed, novel viewpoints may be rendered by warping each pixel in the layered depth image to the new viewpoint's image plane as described above. Because multiple pixels are stored per (u,v) image position in the layered depth image plane, objects which are occluded in the LDI camera projection may become visable as the viewpoint changes, yielding the effect of a scene with depth.

[*] From Shade et al [1]

2 RELATED WORK

A large amount of research has gone into the subject of image-based representation of three-dimensional scenes. This is due, in part, to the fact that the best (most realistic and easiest to generate) method of acquiring realistic texture representations is to simply take of photograph of the object which is then transferred to a two-dimensional image. Research in the field of image-based scene representation is also being driven by parties interested in computer vision for robotics. Given a sparse set of two-dimensional images it only natural to attempt to construct the three-dimensional scene captured by these images. This is where warping comes into play.

There is currently a large body of research that has been and is being done on the subject of warping. Seitz et al [3] address of subject of which views may be inferred from a set of basis images. They prove that under certain assumptions, in specific that uniform regions are preserved in different view, image reconstruction is not an ill-posed problem. Hence, view morphing is not as difficult of a problem as three-dimensional scene reconstruction.

Another example of image based scene reconstruction is given by Seitz et al [4] in which they construct a three-dimensional voxel representation of an object from a given finite set of two-dimensional photographs. Here the images are taken from a camera in such a way as to facilitate the construction of the voxel representation of the object. The convex hull of the camera positions is calculated and used to define a plane, the normal of which (pointing towards the object) is the direction that the voxel construction progress. The resulting voxel can then be used to project an image of the object from a novel viewpoint. As, shall be described in the next section, this voxel construction algorithm may be modified to construct a layered depth image.

3 IMPLEMENTATION BY SHADE ET AL

Shade et al [1] describe the methods to both construct and render layered depth images. Rather than produce the most realistic scene reconstruction images, their goal is to instead minimize the time spent rendering the layered depth image (while still maintaining a reasonable approximation to the exact image). To this end, they employ several optimizations that improve the rendering time considerably.

A layered depth image is stored using the following structure:

```
DepthPixel =
ColorRGBA: 32 bit integer
  Z: 20 bit integer
SplatIndex: 11 bit integer

LayeredDepthPixel =
NumLayers: integer
Layers[0..numlayers-1]: array of DepthPixel

LayeredDepthImage =
Camera:  camera
Pixels[0..xres-1, 0..yres-1]:  array of LayeredDepthPixel
```

Several methods are discussed that can be used to construct a layered depth image. Multiple images with depth may be combined into a layered depth image by warping each image into a common camera view. Or a ray-tracer may be modified to construct a layered depth image. Shade et al discuss various sampling strategies, such as using a cosine weighted ray direction in a scene bounded by a cube, and a stratified stochastic method. Real images may also be used to construct a layered depth image. The voxel coloring algorithm as described by Seitz et al [4] may be adapted to construct a layered depth image directly from the input images. To achieve this, move outwards on rays from the LDI camera center and project candidate voxels back into the input images.

Once created, a layered depth image can be rendering by warping and splatting each depth pixel. Recall that in the general warping algorithm discussed in section 1, we performed the following actions:

Given: $C_1 = V_1 \cdot P_1 \dot{A}_1$ = LDI camera C_2 = novel viewpoint camera $T_{1,2} = C_2 \cdot C_1^{-1}$

We have:

$$T_{1,2} \cdot \begin{pmatrix} x_1 \\ y_1 \\ z_1 \\ 1 \end{pmatrix} = \begin{pmatrix} x_2 \cdot w_2 \\ y_2 \cdot w_2 \\ z_2 \cdot w_2 \\ w_2 \end{pmatrix} = \text{result}$$

Shade et al employ a technique which they refer to as incremental warping in order to reduce the amount of computation needed per depth pixel warp. By using the linearity of matrix operations, the above matrix multiply can be factored to reuse previous computation.

$$T_{1,2} \cdot \begin{pmatrix} x_1 \\ y_1 \\ z_1 \\ 1 \end{pmatrix} = \begin{pmatrix} x_1 \\ y_1 \\ 0 \\ 1 \end{pmatrix} + z_1 \cdot T_{1,2} \cdot \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \end{pmatrix} = \text{start} + z_1 + \text{depth}$$

Instead of exactly computing the splat size, an estimation is calculated as follows:

$$\sqrt{\text{size}} \approx z_2 \cdot \text{lookup[nx,ny,d1]}$$

To further speed things up, McMillan's ordering algorithm [2] is used to determine the order in which to render each depth pixel so as to guarantee that pixels closest to the novel viewpoint are rendered last.

In their paper, Shade et al describe a system which achieves a frame rate of 8 to 10 frames per second, on a Pentium II 300Mhz system, with the generated images having a resolution of 300 by 300.

However, given a complex layered depth image, which contained over 1.1 million depth pixels, the frame rate was reduced to 4 frames per second (which is still pretty good!)

4 IMPLEMENTATION

An attempt was made to implement a layered depth image constructor/renderer using openGL. All code is written in C, with the exception of the modified ray-tracer(originally written by McMillan† which is written in Java. Both the construction and rendering implementations shall now be discussed.

Data is organized in much the same way as the method described by Shade et al [1]. An array of layered depth pixels, a camera position, and an image resolution comprise a layered depth image. Each layered depth pixel consists of one or more depth pixels. Depth pixels consist of an rgba color, a depth value, a splat index. A summary of the C code (from ldi.h) follows:

†see McMillan's website, http://www.cs.unc.edu/ mcmillan/comp136/Project5/RayTrace.java

```
typedef struct DepthPixel *DepthPixelPtr;
struct DepthPixel
{
        int                     colorRGBA;       /* 32-bit integer */
        int                     z;
        int                     splatIndex;
};

typedef struct LayeredDepthPixel *LayeredDepthPixelPtr;
struct LayeredDepthPixel
{
        int                     numLayers;  /* number of layers */
        DepthPixelPtr           layers;    /* pointer to array of DepthPixels w/ dim numLayers */
};

typedef struct LayeredDepthImage *LayeredDepthImagePtr;
struct LayeredDepthImage
{
        CameraGLPtr             camera;     /* pointer to the LDI camera */
        int                     xres;       /* x resolution (in pixels) of the LDI image */
        int                     yres;       /* y resolution (in pixels) of the LDI image */
        LayeredDepthPixelPtr    pixels;    /* pointer to array of LayeredDepthPixels w/ dim (xres * yres) */
};
```

where CameraGLPtr is defined (in camera.h) to be:

```
typedef struct CameraGL *CameraGLPtr;
struct CameraGL
{
        char    type;
        double  projMat[16];    /* perspective projection matrix */
        double  modelMat[16];   /* modelview matrix */
        int     viewMat[4];     /* viewport matrix */

        double  eye[3];
        double  center[3];
        double  up[3];

        double  du[3];
        double  dv[3];

        double  fov;
        double  aspect;
        double  near;
        double  far;

        double  frustum[6];  /* left, right, bottom, top, near, far */
};
```

In order to construct a layered depth image, the candidate two-dimensional images with camera position and depth must first be constructed. This is achieved by running a ray-tracer, originally written by McMillan [‡], which has been modified to save to file the camera configuration, as well as color and depth information per pixel (see section 8). The depth is computed by taking the length of the ray which extends from the eye position to the position intersected on the current object.

Once several such output files have been computed, they can be used to construct a corresponding layered depth image by the use of the jrt2ldi program. This program accepts 1 or more input files that are assumed to be output from the above modified ray-tracer. It picks the first input image's camera as the layered depth image camera, and warps all other given images to the layered depth image camera space. The program works as follows:

```
for(each y position in current image}
{
  for(each x position in current image}
```

[‡]see McMillan's website, http://www.cs.unc.edu/ mcmillan/comp136/Project5/RayTrace.java

```
    {
      read current pixel info from file
      if(current image is LDI image)
      {
         add current pixel to current layered depth pixel
      } else {
 warp current pixel to LDI image space
         if(pixel is in image)
         {
            add current pixel to current layered depth pixel
         }
      }
    }
  }
```

The resulting layered depth image is saved to a file with the same base-name as the first input image, but with the extension ".ldi".

Once a layered depth image has been computed, individual layers may be extracted by using the ldi2ppm program. This program simply reads in the given layered depth image and dumps as a ppm file the result of extracting each pixel in the image at the given depth. For example, ldi2ppm this.ldi 0, would extract all pixels at level 0 and write the resulting image in ppm format.

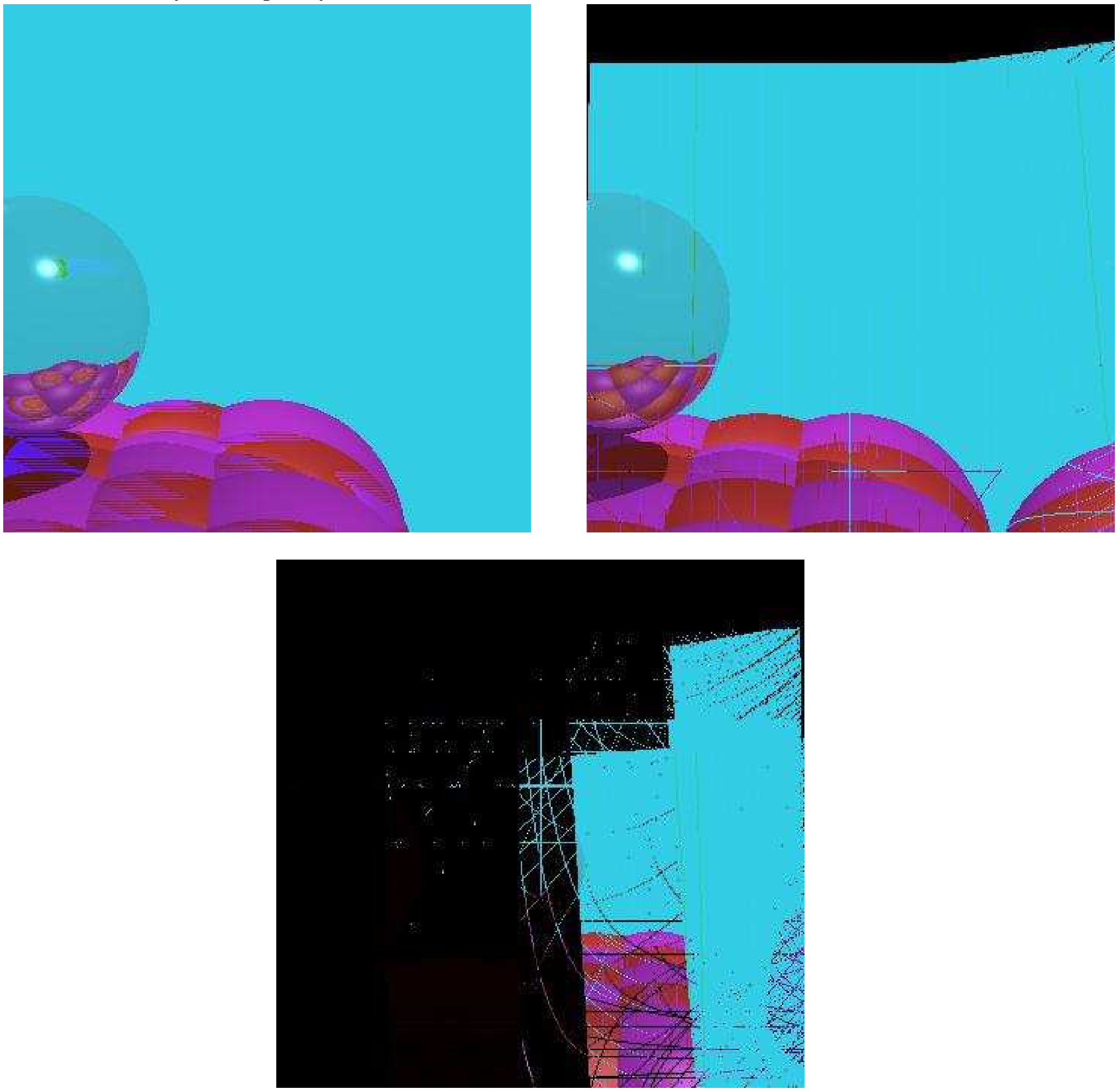
Besides chopping layers off of the given layered depth image, one may also render it by using the ldiview program. This program reads in the given layered depth image and renders it from the LDI camera viewpoint. It then allows the user to move the camera position, direction, etc. via the mouse and keyboard so as to specify a different viewpoint. This new viewpoint is used as the viewpoint to warp the layered depth image to. Warping is achieved by the following code:

```
void
Warp(LayeredDepthPixelPtr ldpPtr, double start[4], double depth[4], double xincr[4])
{
  int l;
  double result[4];
  double ir[4];
  int z1;
  char sqrtSize;

  for(l = 0; l < ldpPtr->numLayers; l++)
  {
    z1 = ldpPtr->layers[l].z;

    /* compute z1 * depth */
    ir[0] = z1 * depth[0]; ir[1] = z1 * depth[1];
    ir[2] = z1 * depth[2]; ir[3] = z1 * depth[3];

    /* compute result (start + ir) */
    result[0] = ir[0] + start[0]; result[1] = ir[1] + start[1];
    result[2] = ir[2] + start[2]; result[3] = ir[3] + start[3];

    /* cull if pixel goes behind output camera or out of output cam's frustum */
    if((result[3] > 0) && (IsInViewPort(result)))
    {
      result[0] /= result[3]; result[1] /= result[3];
      result[2] /= result[3]; result[3] /= result[3];
      sqrtSize = result[2] * lookupTable[ldpPtr->layers[l].splatIndex];
      Splat(ldpPtr->layers[l].colorRGBA, result[0], result[1], result[2], sqrtSize);
    }

    /* increment start */
    start[0] += xincr[0]; start[1] += xincr[1];
    start[2] += xincr[2]; start[3] += xincr[3];
  }
}
```

So, incremental warping is used for efficiency. Splat estimation is not currently implemented - everything splats to 1. Neither is McMillan's ordering algorithm implemented - the z-buffer is used instead. No clipping is performed other than a test to see if the warped pixel is a valid index in the novel viewpoint's image plane.

5 RESULTS

There are problems with the layered depth image construction code that preclude the rendering code from functioning correctly. From analyzing the layered depth image a depth slice at a time, starting from the front and moving back, it is clear that there are problems with calibration (i.e. transformation) between the openGL camera model and the camera model used by the simple ray-tracer.



6 FUTURE WORK


§

There are many ways in which to refine the work presented in this paper. The obvious refinement is to debug the construction and rendering code so that it works in its simple state. This involves debugging the camera transformations between non-openGL and openGL cameras, and proper computation of the splat table. Next, implementation of McMillan's warp ordering algorithm when rendering depth pixels would greatly speed display by no longer requiring the use of the Z-buffer. Further refinements involve building a more general layered depth image construction utility capable of building layered depth images in all of methods described by Shade et al, such as via a modified stratified stochastic ray-tracer, a modified Seitz and Dyer voxel construction algorithm and other image based techniques. An alternate structure for storing layered depth image data may be more adaptable to the current desired use - i.e. in some instances it may be of desire to sacrifice speed for image quality.

In conclusion, a simple ray-tracer was modified to store depth information and current camera position as well as compute pixel color values. Several views from the same static scene produced by the above ray-tracer were then used in the construction of a layered depth image representation of the scene. This constructed layered depth image could then be loaded by the viewer. However, due to problems with camera calibration and splatting, the resulting rendered images bare little resemblance to the original scene.

Appendix

7 SOURCE CODE SUMMARY

Here is a brief description of each file used for this project:

§ stereogram created by McMillan

Makefile	makefile used to build project. to rebuild on new system: make depend.
ldi.[hc]	layered depth image ADT routines.
camera.[hc]	camera ADT routines.
proc_ppm.[hc]	ppm file manipulation routines.
ldi_display.[hc]	layered depth image openGL rendering code.
matrix.[hc]	various matrix routines.
mymatrix.[hc]	various other matrix routines.
mlist.[hc]	singly-linked list ADT routines.
render_main.c	ldiview's main - render given layered depth image.
ldi2ppm.c	ldi2ppm's main - extract layer from layered depth image.
jrt2ldi.c	jrt2ldi's main - convert ray-tracer output to ldi.
vs.c	vs's main - render given scene def. file.

8 MODIFIED RAY-TRACER LISTING

```
package traceldi;

import java.awt.*;
import java.awt.image.*;
import java.awt.event.*;
import java.applet.*;
import java.io.*;
import java.net.*;
import java.util.*;


// A raster object
class Raster implements ImageConsumer {
        public int width, height;
        public int pixel[];
        public float depth[];
        private ImageProducer producer;


    //////////////////////// Constructors /////////////////////

        /**
         *  This constructor which takes no arguments
         *  allows for future extension.
         */
        public Raster()
        {
        }

        /**
         *  This constructor creates an uninitialized
         *  Raster Object of a given size (w x h).
         */
        public Raster(int w, int h)
        {
            width = w;
            height = h;
            pixel = new int[w*h];
            depth = new float[w*h];
        }

         /**
         *  This constructor creates an Raster initialized
         *  with the contents of an image.
         */
        public Raster(Image img)
        {
            producer = img.getSource();
            producer.addConsumer(this);
            producer.startProduction(this);
            waitForImage();
            depth = null;
        }

    public synchronized void waitForImage() {
                try {
                    wait( );
                } catch (Exception e) {
                };
        }

    public synchronized void imageComplete(int status)
        {
            producer.removeConsumer(this);
            notifyAll();
        }

        public void setProperties(Hashtable props)
        {
        }

        public void setColorModel(ColorModel model)
        {
        }

        public void setHints(int hintflags)
```

```
    {
    }

    public void setDimensions(int w, int h) {
        width = w;
        height = h;
        pixel = new int[w*h];
        depth = new float[w*h];
    }

public void setPixels(int x, int y, int w, int h, ColorModel cm, byte p[], int offset, int stride) {
        int i, j, delta;

        i = y*width+x;
        delta = width - w;
        while (h > 0) {
            for (j = 0; j < w; j++) {
                pixel[i++] = cm.getRGB(p[offset+j] & 255);
            }
            offset += stride;
            i += delta;
            h -= 1;
        }
    }

  public void renderRaster(Graphics g) {
        int x, y;

        for(y = 0; y < height; y++) {
            for (x = 0; x < width; x++) {
                g.setColor(this.getColor(x, y));
                g.drawLine(x,y, x,y);
               /*
               if(this.getDepth(x,y) != (float)0.0)
               {
                System.out.println("["+Integer.toString(x)+","+Integer.toString(y)+"d="+Float.toString(this.getDepth(x,y)));
               }
               */
            }
        }
    }

    public void setPixels(int x, int y, int w, int h, ColorModel cm, int p[], int offset, int stride) {
        int i, j, delta;

        i = y*width+x;
        delta = width - w;
        while (h > 0) {
            for (j = 0; j < w; j++) {
                pixel[i++] = cm.getRGB(p[offset+j]);
            }
            offset += stride;
            i += delta;
            h -= 1;
        }
    }
///////////////////////// Methods /////////////////////////

    /**
     *  Returns the number of pixels in the Raster
     */
    public final int size( )
    {
        return width*height;
    }

    /**
     *  Fills a Raster with a solid color
     */
    public final void fill(Color c)
    {
        int s = size();
        int rgb = c.getRGB();
        for (int i = 0; i < s; i++)
            pixel[i] = rgb;
    }
/**
     *  Converts Rasters to Images
     */
    public final Image toImage(Component root)
    {
        return root.createImage(new MemoryImageSource(width, height, pixel, 0, width));
    }

/**
     *  Gets a pixel from a Raster
     */
    public final int getPixel(int x, int y)
    {
        return pixel[y*width+x];
    }

    /**
     *  Gets a color from a Raster
     */
    public final Color getColor(int x, int y)
    {
        return new Color(pixel[y*width+x]);
    }

         public final Color getColorDepth(int x, int y)
    {
        return new Color((int)depth[y*width+x] & 255);
    }

    /**
```

```
         *  Sets a pixel to a given value
         */
        public final boolean setPixel(int pix, int x, int y)
        {
            pixel[y*width+x] = pix;
            return true;
        }

        /**
         *  Sets a pixel to a given color
         */
         public final boolean setColor(Color c, int x, int y)
         {
            pixel[y*width+x] = c.getRGB();
            return true;
         }

         public final void setColorDepth(DepthColor c, int x, int y, Vector3D eye)
         {
            depth[y*width+x] = c.getDepth(eye);
            pixel[y*width+x] = c.color.getRGB();
         }

         public final float getDepth(int x, int y)
         {
           return(depth[y*width+x]);
         }


        // public void save(
  }

// A simple vector class
class Vector3D {
    public float x, y, z;

    // constructors
    public Vector3D( )
    {
    }

    public Vector3D(float x, float y, float z)
    {
        this.x = x; this.y = y; this.z = z;
    }

    public Vector3D(Vector3D v)
    {
        x = v.x;
        y = v.y;
        z = v.z;
    }

    // methods
    public final float dot(Vector3D B)
    {
        return (x*B.x + y*B.y + z*B.z);
    }

    public final float dot(float Bx, float By, float Bz)
    {
        return (x*Bx + y*By + z*Bz);
    }

    public static final float dot(Vector3D A, Vector3D B)
    {
        return (A.x*B.x + A.y*B.y + A.z*B.z);
    }

    public final Vector3D cross(Vector3D B)
    {
        return new Vector3D(y*B.z - z*B.y, z*B.x - x*B.z, x*B.y - y*B.x);
    }

    public final Vector3D cross(float Bx, float By, float Bz)
    {
        return new Vector3D(y*Bz - z*By, z*Bx - x*Bz, x*By - y*Bx);
    }

    public final static Vector3D cross(Vector3D A, Vector3D B)
    {
        return new Vector3D(A.y*B.z - A.z*B.y, A.z*B.x - A.x*B.z, A.x*B.y - A.y*B.x);
    }

    public final float length( )
    {
        return (float) Math.sqrt(x*x + y*y + z*z);
    }

    public final static float length(Vector3D A)
    {
        return (float) Math.sqrt(A.x*A.x + A.y*A.y + A.z*A.z);
    }

    public final void normalize( )
    {
        float t = x*x + y*y + z*z;
        if (t != 0 && t != 1) t = (float) (1 / Math.sqrt(t));
        x *= t;
        y *= t;
        z *= t;
    }

    public final static Vector3D normalize(Vector3D A)
    {
        float t = A.x*A.x + A.y*A.y + A.z*A.z;
```

```
        if (t != 0 && t != 1) t = (float)(1 / Math.sqrt(t));
        return new Vector3D(A.x*t, A.y*t, A.z*t);
    }

    public final Vector3D scale(float s)
    {
        return new Vector3D(s*x, s*y, s*z);
    }

    public final Vector3D plus(Vector3D A)
    {
        return new Vector3D(x+A.x, y+A.y, z+A.z);
    }

    public final Vector3D minus(Vector3D A)
    {
        return new Vector3D(x-A.x, y-A.y, z-A.z);
    }

    public String toString()
    {
        return new String("["+x+", "+y+", "+z+"]");
    }
}


class DepthColor {
  Vector3D ip;
  public Color color;

  public DepthColor(Vector3D i, Color c)
  {
    ip = i;
    color = c;
  }

  public DepthColor()
  {
    ip = null;
    color = null;
  }
  public float getDepth(Vector3D eye)
  {
    return(Vector3D.length(ip.minus(eye)));
  }

  public Color getColor()
  {
    return(color);
  }
}


class Ray {
    public static final float MAX_T = Float.MAX_VALUE;
    Vector3D origin;
    Vector3D direction;
    float t;
    Renderable object;
    public int level;
    public Ray(Vector3D eye, Vector3D dir)
    {
        origin = new Vector3D(eye);
        direction = Vector3D.normalize(dir);
level = 1;
    }

    public boolean trace(Vector objects)
    {
        Enumeration objList = objects.elements();
        t = MAX_T;
        object = null;
        while (objList.hasMoreElements()) {
            Renderable object = (Renderable) objList.nextElement();
            object.intersect(this);
        }
        return (object != null);
    }

    // The following method is not strictly needed, and most likely
    // adds unnecessary overhead, but I prefered the syntax
    //
    //            ray.Shade(...)
    // to
    //            ray.object.Shade(ray, ...)
    //
    /*
    public final Color Shade(Vector lights, Vector objects, Color bgnd)
    {
        return object.Shade(this, lights, objects, bgnd);
    }
    */
    public final DepthColor Shade(Vector lights, Vector objects, Color bgnd)
    {
        return object.Shade(this, lights, objects, bgnd);
    }

    public String toString()
    {
        return ("ray origin = "+origin+"  direction = "+direction+"  t = "+t);
    }
}

// All the public variables here are ugly, but I
// wanted Lights and Surfaces to be "friends"
class Light {
```

```
    public static final int AMBIENT = 0;
    public static final int DIRECTIONAL = 1;
    public static final int POINT = 2;

    public int lightType;
    public Vector3D lvec;           // the position of a point light or
                                    // the direction to a directional light
    public float ir, ig, ib;        // intensity of the light source

    public Light(int type, Vector3D v, float r, float g, float b)
    {
        lightType = type;
        ir = r;
        ig = g;
        ib = b;
        if (type != AMBIENT) {
            lvec = v;
            if (type == DIRECTIONAL) {
                lvec.normalize();
            }
        }
    }
}


class Surface {
    public float ir, ig, ib;        // surface's intrinsic color
    public float ka, kd, ks, ns;    // constants for phong model
    public float kt, kr, nt;
    private static final float TINY = 0.00001f;
    private static final float I255 = 0.00392156f;  // 1/255

    public Surface(float rval, float gval, float bval, float a, float d, float s, float n, float r, float t, float index)
    {
        ir = rval; ig = gval; ib = bval;
        ka = a; kd = d; ks = s; ns = n;
        kr = r*I255; kt = t*I255; nt = index;
    }

    public Color Shade(Vector3D p, Vector3D n, Vector3D v, Vector lights, Vector objects, Color bgnd, boolean inside, int level)
    {
        Enumeration lightSources = lights.elements();

        float r = 0;
        float g = 0;
        float b = 0;
        while (lightSources.hasMoreElements()) {
            Light light = (Light) lightSources.nextElement();
            if (light.lightType == Light.AMBIENT) {
                r += ka*ir*light.ir;
                g += ka*ig*light.ig;
                b += ka*ib*light.ib;
            } else {
                Vector3D l;
                if (light.lightType == Light.POINT) {
                    l = new Vector3D(light.lvec.x - p.x, light.lvec.y - p.y, light.lvec.z - p.z);
                    l.normalize();
                } else {
                    l = new Vector3D(-light.lvec.x, -light.lvec.y, -light.lvec.z);
                }

                // Check if the surface point is in shadow
                Vector3D poffset = new Vector3D(p.x + TINY*l.x, p.y + TINY*l.y, p.z + TINY*l.z);
                Ray shadowRay = new Ray(poffset, l);
                if (shadowRay.trace(objects))
                    break;

                float lambert = Vector3D.dot(n,l);
                if (lambert > 0) {
                    if (kd > 0) {
                        float diffuse = kd*lambert;
                        r += diffuse*ir*light.ir;
                        g += diffuse*ig*light.ig;
                        b += diffuse*ib*light.ib;
                    }
                    if (ks > 0) {
                        lambert *= 2;
                        float spec = v.dot(lambert*n.x - l.x, lambert*n.y - l.y, lambert*n.z - l.z);
                        if (spec > 0) {
                            spec = ks*((float) Math.pow((double) spec, (double) ns));
                            r += spec*light.ir;
                            g += spec*light.ig;
                            b += spec*light.ib;
                        }
                    }
                }
            }
        }

        // Compute illumination due to reflection
        if (kr > 0) {
            float t = v.dot(n);
            if (t > 0) {
                t *= 2;
                Vector3D reflect = new Vector3D(t*n.x - v.x, t*n.y - v.y, t*n.z - v.z);
                Vector3D poffset = new Vector3D(p.x + TINY*reflect.x, p.y + TINY*reflect.y, p.z + TINY*reflect.z);
                if(level < 8){
Ray reflectedRay = new Ray(poffset, reflect);
                reflectedRay.level = level +1;
if (reflectedRay.trace(objects)) {
                    DepthColor rcolor = reflectedRay.Shade(lights, objects, bgnd);
                    r += kr*rcolor.color.getRed();
                    g += kr*rcolor.color.getGreen();
                    b += kr*rcolor.color.getBlue();
                } else {
                    r += kr*bgnd.getRed();
```

```
                    g += kr*bgnd.getGreen();
                    b += kr*bgnd.getBlue();
}
            }
    }
        }

        // Have the students add code for refraction here
//Eq. 14-18, p.510, Hearne: T=(nt(v dot n) - cos(r))n - nt*v(Gregory)
if(kt > 0){
if(inside)nt=1/nt;
float t = v.dot(n);
if (t > 0){
float cr = (float)Math.sqrt(1-nt*nt*(1-t*t));  //refracted vector cosin
float nc = nt*t - cr; //normal component coefficient
Vector3D refract = new Vector3D(nc*n.x - nt*v.x, nc*n.y - nt*v.y, nc*n.z - nt*v.z);
                Vector3D poffset = new Vector3D(p.x + TINY*refract.x, p.y + TINY*refract.y, p.z + TINY*refract.z);
                if(level < 8){
Ray refractedRay = new Ray(poffset, refract);
refractedRay.level = level +1;
if (refractedRay.trace(objects)) {
                    DepthColor rcolor = refractedRay.Shade(lights, objects, bgnd);
                    r += kt*rcolor.color.getRed();
                    g += kt*rcolor.color.getGreen();
                    b += kt*rcolor.color.getBlue();
                } else {
                    r += kt*bgnd.getRed();
                    g += kt*bgnd.getGreen();
                    b += kt*bgnd.getBlue();
}
}
}
}
        r = (r > 1f) ? 1f : r;
        g = (g > 1f) ? 1f : g;
        b = (b > 1f) ? 1f : b;
        return new Color(r, g, b);
    }
 }


// An object must implement a Renderable interface in order to
// be ray traced. It should be straight forward to add new objects
abstract interface Renderable {
    public abstract boolean intersect(Ray r);
//    public abstract Color Shade(Ray r, Vector lights, Vector objects, Color bgnd);
    public abstract DepthColor Shade(Ray r, Vector lights, Vector objects, Color bgnd);
    public String toString();
}


// A example "Renderable" object
class Sphere implements Renderable {
    Surface surface;
    Vector3D center;
    float radius;
    float radSqr;
    boolean inside;     // this tricky variable is set by intersect()
                        // and tested by a subsequent Shade()

    public Sphere(Surface s, Vector3D c, float r)
    {
        surface = s;
        center = c;
        radius = r;
        radSqr = r*r;
    }

    public boolean intersect(Ray ray)
    {
        float dx = center.x - ray.origin.x;
        float dy = center.y - ray.origin.y;
        float dz = center.z - ray.origin.z;
        float v = ray.direction.dot(dx, dy, dz);
        float s, t;

        // compare the distance from the ray origin
        // to the sphere's center with its radius
        t = radSqr - dx*dx - dy*dy - dz*dz;
        if (t > 0) {
            // if we're inside we better be transparent
            if (surface.kt == 0) return false;
            inside = true;
        } else {
            // Check if a ray starting outside has any chance
            // to have an intersection closer than a previous one
            if (v - radius > ray.t) return false;

            // if we're outside we had better be heading
            // toward the sphere
            if (v < 0) return false;
            inside = false;
        }
        t += v*v;

        // Test if the ray actually intersects the sphere
        if (t < 0)
            return false;

        s = (float) Math.sqrt(t);
        if (inside) {
            t = v + s;
        } else {
            t = v - s;
        }
```

```
        if (t > ray.t)
            return false;

        ray.t = t;
        ray.object = this;
        return true;
    }

    /*
     public Color Shade(Ray ray, Vector lights, Vector objects, Color bgnd)
    {
        // An object shader doesn't really do too much other than
        // supply a few critical pieces of geometric information
        // for a surface shader. It must must compute
        //   1. the point of intersection (p)
        //   2. a unit-length surface normal (n)
        //   3. a unit-length vector towards the ray's origin (v)
        float px = ray.origin.x + ray.t*ray.direction.x;
        float py = ray.origin.y + ray.t*ray.direction.y;
        float pz = ray.origin.z + ray.t*ray.direction.z;

        Vector3D p = new Vector3D(px, py, pz);
        Vector3D v = new Vector3D(-ray.direction.x, -ray.direction.y, -ray.direction.z);
        Vector3D n;
        if (inside) {
            n = new Vector3D(center.x - px, center.y - py, center.z - pz);
        } else {
            n = new Vector3D(px - center.x, py - center.y, pz - center.z);
        }
        n.normalize();

        // The illumination model is applied
        // by the surface's Shade() method
        return surface.Shade(p, n, v, lights, objects, bgnd, inside, ray.level);
    }
    */


    public DepthColor Shade(Ray ray, Vector lights, Vector objects, Color bgnd)
    {
    DepthColor dc = new DepthColor();
        // An object shader doesn't really do too much other than
        // supply a few critical pieces of geometric information
        // for a surface shader. It must must compute
        //   1. the point of intersection (p)
        //   2. a unit-length surface normal (n)
        //   3. a unit-length vector towards the ray's origin (v)
        float px = ray.origin.x + ray.t*ray.direction.x;
        float py = ray.origin.y + ray.t*ray.direction.y;
        float pz = ray.origin.z + ray.t*ray.direction.z;

        Vector3D p = new Vector3D(px, py, pz);
        dc.ip = p;

        Vector3D v = new Vector3D(-ray.direction.x, -ray.direction.y, -ray.direction.z);
        Vector3D n;
        if (inside) {
            n = new Vector3D(center.x - px, center.y - py, center.z - pz);
        } else {
            n = new Vector3D(px - center.x, py - center.y, pz - center.z);
        }
        n.normalize();

        // The illumination model is applied
        // by the surface's Shade() method
        dc.color = surface.Shade(p, n, v, lights, objects, bgnd, inside, ray.level);
        return(dc);
    }

    public String toString()
    {
        return ("sphere "+center+" "+radius);
    }
}


//Try a triangle renderable
class Triangle implements Renderable {
    Surface surface;
    Vector3D A;
    Vector3D B;
    Vector3D C;
    boolean inside;     // this tricky variable is set by intersect()
    Vector3D n;        // and tested by a subsequent Shade()
    float D;

    public Triangle(Surface s, Vector3D ptA,Vector3D ptB,Vector3D ptC)
    {
        surface = s;
        A = ptA;
        B = ptB;
C = ptC;

    }

    public float area(Vector3D ab,Vector3D ac,Vector3D bc){
float area, lab, lac, lbc;
lab = Vector3D.length(ab);
lac = ac.length();
lbc = bc.length();
if ((lab > lac) && (lab > lbc))
area = (lab/2)*(float)Math.sqrt((lac*lac) - (ac.dot(ab)/lab)*(ac.dot(ab)/lab));
else if (lac > lbc)
area = (lac/2)*(float)Math.sqrt((lab*lab) - (ac.dot(ab)/lac)*(ac.dot(ab)/lac));
else
area = (lbc/2)*(float)Math.sqrt((lab*lab) - (bc.dot(ab)/lbc)*(bc.dot(ab)/lbc));
```

```
return area;
    }
    public boolean intersect(Ray ray)
    {
boolean InSphere = true; //for bounding sphere
//from http://www.wri.com/MathSource/.aliases/0207-324/TWJPACKS.SHR
Vector3D AB = new Vector3D(A.x - B.x, A.y - B.y, A.z - C.z);
Vector3D AC = new Vector3D(A.x - C.x, A.y - C.y, A.z - C.z);
Vector3D BC = new Vector3D(B.x - C.x, B.y - C.y, B.z - C.z);
Vector3D n = Vector3D.cross(AB,AC);
n.normalize();
         float d1, d2, d3, c1, c2, c3, c,rad;
         Vector3D mAC = new Vector3D(-AC.x, -AC.y, -AC.z);
         Vector3D mAB = new Vector3D(-AB.x, -AB.y, -AB.z);
                Vector3D mBC = new Vector3D(-BC.x, -BC.y, -BC.z);
                d1 = mAC.dot(mAB);
                d2 = AB.dot(mBC);
                d3 = BC.dot(AC) ;
                c1 = d2 * d3 ;
                c2 = d3 * d1 ;
                c3 = d1 * d2 ;
                c = c1 + c2 + c3 ;
                rad = (float) Math.sqrt( (d1+d2)*(d2+d3)*(d3+d1)/c)/2 ;
                Vector3D cent = new Vector3D((c2+c3)*A.x + (c3+c1)*B.x + (c1+c2)*C.x,
(c2+c3)*A.y + (c3+c1)*B.y + (c1+c2)*C.y,
(c2+c3)*A.z + (c3+c1)*B.z + (c1+c2)*C.z);
float C2 = 1/(2*c);
                cent = cent.scale(C2);
float radSqr = rad * rad;
        float dx = cent.x - ray.origin.x;
        float dy = cent.y - ray.origin.y;
        float dz = cent.z - ray.origin.z;
        float v = ray.direction.dot(dx, dy, dz);
        float s, t;

        // compare the distance from the ray origin
        // to the sphere's center with its radius
        t = radSqr - dx*dx - dy*dy - dz*dz;
        if (t > 0) {
            // if we're inside we better be transparent
            //if (surface.kt == 0) return false; ------but for the bounding sphere we don't care
            InSphere = true;
    inside = true;
        } else {
            // Check if a ray starting outside has any chance
            // to have an intersection closer than a previous one
            if (v - rad > ray.t) InSphere = false;

            // if we're outside we had better be heading
            // toward the sphere
            if (v < 0) InSphere = false;
    inside = false;
        }
        t += v*v;

        // Test if the ray actually intersects the sphere
        if (t < 0)
            InSphere = false;

        s = (float) Math.sqrt(t);
        if (inside) {
            t = v + s;
        } else {
            t = v - s;
        }

        if (t > ray.t)
            InSphere = false;
//-------------Put stuff here
//Now, for a triangle, we have to do the plane equation

        if ((ray.direction.dot(n) < 0) && InSphere) { //check to see if we're facing the eye
float D = -n.dot(A); //find the value of D in plane equation by
//dot product of normal with point of triangle
float TriT = -D /(n.dot(ray.direction)); //coefficient to get point on plane
        float px = ray.origin.x + TriT*ray.direction.x;
        float py = ray.origin.y + TriT*ray.direction.y;
        float pz = ray.origin.z + TriT*ray.direction.z;
        Vector3D p = new Vector3D(px, py, pz);

        Vector3D Ap = new Vector3D(A.x - p.x, A.y - p.y, A.z - p.z);
        Vector3D Bp = new Vector3D(B.x - p.x, B.y - p.y, B.z - p.z);
        Vector3D Cp = new Vector3D(C.x - p.x, C.y - p.y, C.z - p.z);

float areaPrime,areaSecond, area1, area2, area3;
areaPrime = area(AB,AC,BC);
area1 = area(AB,Ap,Bp);
area2 = area(AC,Ap,Cp);
area3 = area(BC,Bp,Cp);
areaSecond = area1 + area2 + area3;
if (areaPrime == areaSecond){
//------------------------
        ray.t = TriT;
        ray.object = this;
        return true;
    }
else return false;
    }
else return false;
    }
    /*
    public Color Shade(Ray ray, Vector lights, Vector objects, Color bgnd)
    {
        // An object shader doesn't really do too much other than
        // supply a few critical pieces of geometric information
        // for a surface shader. It must must compute
```

```
        //   1. the point of intersection (p)
        //   2. a unit-length surface normal (n)
        //   3. a unit-length vector towards the ray's origin (v)

        // The illumination model is applied
        // by the surface's Shade() method

        float px = ray.origin.x + ray.t*ray.direction.x;
        float py = ray.origin.y + ray.t*ray.direction.y;
        float pz = ray.origin.z + ray.t*ray.direction.z;

        Vector3D p = new Vector3D(px, py, pz);
        Vector3D v = new Vector3D(-ray.direction.x, -ray.direction.y, -ray.direction.z);
Vector3D n = this.n;
        return surface.Shade(p, n, v, lights, objects, bgnd, inside,ray.level);
    }
    */
    public DepthColor Shade(Ray ray, Vector lights, Vector objects, Color bgnd)
    {
        DepthColor dc = new DepthColor();

        // An object shader doesn't really do too much other than
        // supply a few critical pieces of geometric information
        // for a surface shader. It must must compute
        //   1. the point of intersection (p)
        //   2. a unit-length surface normal (n)
        //   3. a unit-length vector towards the ray's origin (v)

        // The illumination model is applied
        // by the surface's Shade() method

        float px = ray.origin.x + ray.t*ray.direction.x;
        float py = ray.origin.y + ray.t*ray.direction.y;
        float pz = ray.origin.z + ray.t*ray.direction.z;

        Vector3D p = new Vector3D(px, py, pz);
        dc.ip = p;

        Vector3D v = new Vector3D(-ray.direction.x, -ray.direction.y, -ray.direction.z);
Vector3D n = this.n;
        dc.color = surface.Shade(p, n, v, lights, objects, bgnd, inside,ray.level);
        return(dc);
    }


    public String toString()
    {
        return ("Triangle ");
    }
}

/*An attempt to create another Renderable
//above, triangle first
class Torus implements Renderable {
    Surface surface;
    Vector3D center;
    Vector3D up;
    float orbit;
    float radius;
    float radSqr;
    boolean inside;     // this tricky variable is set by intersect()
                        // and tested by a subsequent Shade()

    public Torus(Surface s, Vector3D c, Vector3D u, float o, float r)
    {
        surface = s;
        center = c;
up = u;
orbit = o;
        radius = r;
        radSqr = r*r;
    }

    public boolean intersect(Ray ray)
    {
        float dx = center.x - ray.origin.x;
        float dy = center.y - ray.origin.y;
        float dz = center.z - ray.origin.z;
        float v = ray.direction.dot(dx, dy, dz);
        float s, t;

        // compare the distance from the ray origin
        // to the sphere's center with its radius
        t = radSqr - dx*dx - dy*dy - dz*dz;
        if (t > 0) {
            // if we're inside we better be transparent
            if (surface.kt == 0) return false;
            inside = true;
        } else {
            // Check if a ray starting outside has any chance
            // to have an intersection closer than a previous one
            if (v - radius > ray.t) return false;

            // if we're outside we had better be heading
            // toward the sphere
            if (v < 0) return false;
            inside = false;
        }
        t += v*v;

        // Test if the ray actually intersects the sphere
        if (t < 0)
            return false;

        s = (float) Math.sqrt(t);
        if (inside) {
```

```
            t = v + s;
        } else {
            t = v - s;
        }

        if (t > ray.t)
            return false;

        ray.t = t;
        ray.object = this;
        return true;
    }

    public Color Shade(Ray ray, Vector lights, Vector objects, Color bgnd)
    {
        // An object shader doesn't really do too much other than
        // supply a few critical pieces of geometric information
        // for a surface shader. It must must compute
        //   1. the point of intersection (p)
        //   2. a unit-length surface normal (n)
        //   3. a unit-length vector towards the ray's origin (v)
        float px = ray.origin.x + ray.t*ray.direction.x;
        float py = ray.origin.y + ray.t*ray.direction.y;
        float pz = ray.origin.z + ray.t*ray.direction.z;

        Vector3D p = new Vector3D(px, py, pz);
        Vector3D v = new Vector3D(-ray.direction.x, -ray.direction.y, -ray.direction.z);
        Vector3D n;
        if (inside) {
            n = new Vector3D(center.x - px, center.y - py, center.z - pz);
        } else {
            n = new Vector3D(px - center.x, py - center.y, pz - center.z);
        }
        n.normalize();

        // The illumination model is applied
        // by the surface's Shade() method
        return surface.Shade(p, n, v, lights, objects, bgnd, inside);
    }

    public String toString()
    {
        return ("sphere "+center+" "+radius);
    }
}
**********************************************************************/




//impoublic class RayTracer extends Applet {
public class RayTracer extends Component implements Runnable {

 // XYLayout xYLayout1 = new XYLayout();
  //boolean isStandalone = false;

  //Construct the applet

   final static int CHUNKSIZE = 100;
    Raster raster;
    Vector objectList;
    Vector lightList;
    Surface currentSurface;

    Vector3D eye, lookat, up;
    Vector3D Du, Dv, Vp;
    float fov;

    Color background;

    int width, height;
    FileOutputStream os = null;

    protected Frame frame;
    protected PopupMenu popup;

    String iFileName, oFileName;

    /*
    private void jbInit() throws Exception {
      xYLayout1.setWidth(400);
      xYLayout1.setHeight(300);
      this.setLayout(xYLayout1);
    }
    */
    public RayTracer(Frame frame, int width, int height, RayTracer refRt, String outfile, int which)
    {
      this.frame = frame;
      this.width = width;
      this.height = height;

      raster = new Raster(this.width, this.height);

      this.enableEvents(AWTEvent.MOUSE_EVENT_MASK);
      this.enableEvents(AWTEvent.MOUSE_MOTION_EVENT_MASK);

      iFileName = refRt.iFileName;
      oFileName = outfile;

      init(refRt, which);
    }

    public RayTracer(Frame frame, int width, int height, String infile, String outfile, float ex, float ey, float ez, float fov)
    {
      this.frame = frame;
```

```
    this.width = width;
    this.height = height;

    raster = new Raster(this.width, this.height);

    this.enableEvents(AWTEvent.MOUSE_EVENT_MASK);
    this.enableEvents(AWTEvent.MOUSE_MOTION_EVENT_MASK);

    iFileName = infile;
    oFileName = outfile;

    init();

    eye.x = ex;
    eye.y = ey;
    eye.z = ez;
    fov = fov;
  }

  public void init()
  {
      // initialize the off-screen rendering buffer
      raster.fill(Color.black);


      fov = 30;               // default horizonal field of view

      // Initialize various lists
      objectList = new Vector(CHUNKSIZE, CHUNKSIZE);
      lightList = new Vector(CHUNKSIZE, CHUNKSIZE);
      currentSurface = new Surface(0.8f, 0.2f, 0.9f, 0.2f, 0.4f, 0.4f, 10.0f, 0f, 0f, 1f);

      // Parse the scene file
      String filename = iFileName;
      System.out.println("Parsing " + filename);
      try {
          FileInputStream is = new FileInputStream(filename);
          ReadInput(is);
          is.close();
      } catch (IOException e) {
          System.err.println("Error reading "+filename);
          System.exit(-1);
      }

      /*
      filename = oFileName;
      System.out.println("Parsing " + filename);
      try {
          os = new FileOutputStream(filename);
      } catch (IOException e) {
          System.err.println("Error error opening "+filename+" for writing");
          System.exit(-1);
      }
      */

      // Initialize more defaults if they weren't specified
      if (eye == null) eye = new Vector3D(0, 0, 10);
      if (lookat == null) lookat = new Vector3D(0, 0, 0);
      if (up  == null) up = new Vector3D(0, 1, 0);
      if (background == null) background = new Color(0,0,0);

      // Compute viewing matrix that maps a
      // screen coordinate to a ray direction
      Vector3D look = new Vector3D(lookat.x - eye.x, lookat.y - eye.y, lookat.z - eye.z);
      Du = Vector3D.normalize(look.cross(up));
      Dv = Vector3D.normalize(look.cross(Du));
      float fl = (float)(width / (2*Math.tan((0.5*fov)*Math.PI/180)));
      Vp = Vector3D.normalize(look);
      Vp.x = Vp.x*fl - 0.5f*(width*Du.x + height*Dv.x);
      Vp.y = Vp.y*fl - 0.5f*(width*Du.y + height*Dv.y);
      Vp.z = Vp.z*fl - 0.5f*(width*Du.z + height*Dv.z);

  }

  public void init(RayTracer refRt, int which)
  {
      init();
      if(which == 2)
      {
/*
        // case 1 - rotate eye 90 deg. about up
        Vector3D rr1 = new Vector3D((float)Math.cos(90), (float)0.0, (float)-Math.sin(90));
        Vector3D rr2 = new Vector3D((float)0.0, (float)1.0, (float)0.0);
        Vector3D rr3 = new Vector3D((float)Math.sin(90), (float)0.0, (float)Math.cos(90));
        Vector3D orig = new Vector3D((float)0, (float)0, (float)0);
        eye.x = eye.x + orig.dot(rr1);
        eye.y = eye.y + orig.dot(rr2);
        eye.z = eye.z + orig.dot(rr3);
  */
        eye.x = (float)1.0;
        eye.y = (float)2.0;
        eye.z = (float)10.0;
        System.out.println("Eye = " + Float.toString(eye.x) + ", " + Float.toString(eye.y) + ", " + Float.toString(eye.z) + ".");
      }
      else if(which == 3)
      {
        eye.x = (float)1.0;
        eye.y = (float)1.0;
        eye.z = (float)9.0;


        System.out.println("Eye = " + Float.toString(eye.x) + ", " + Float.toString(eye.y) + ", " + Float.toString(eye.z) + ".");
      }
      else if(which == 4)
      {
        eye.x = (float)2.0;
```

```
      eye.y = (float)1.0;
      eye.z = (float)7.0;

      System.out.println("Eye = " + Float.toString(eye.x) + ", " + Float.toString(eye.y) + ", " + Float.toString(eye.z) + ".");
    }
   else if(which == 5)
    {
      eye.x = (float)0.0;
      eye.y = (float)2.0;
      eye.z = (float)8.0;

      System.out.println("Eye = " + Float.toString(eye.x) + ", " + Float.toString(eye.y) + ", " + Float.toString(eye.z) + ".");
    }
   else if(which == 6)
    {
      eye.x = (float)1.0;
      eye.y = (float)0.0;
      eye.z = (float)9.0;

      System.out.println("Eye = " + Float.toString(eye.x) + ", " + Float.toString(eye.y) + ", " + Float.toString(eye.z) + ".");
    }
   else if(which == 7)
    {
      eye.x = (float)-1.0;
      eye.y = (float)0.0;
      eye.z = (float)9.0;

      System.out.println("Eye = " + Float.toString(eye.x) + ", " + Float.toString(eye.y) + ", " + Float.toString(eye.z) + ".");
    }
    else if(which == 8)
    {
      eye.x = (float)-1.0;
      eye.y = (float)-1.0;
      eye.z = (float)10.0;
      System.out.println("Eye = " + Float.toString(eye.x) + ", " + Float.toString(eye.y) + ", " + Float.toString(eye.z) + ".");
    }
    else if(which == 9)
    {
      eye.x = (float)0.0;
      eye.y = (float)-2.0;
      eye.z = (float)10.0;
      System.out.println("Eye = " + Float.toString(eye.x) + ", " + Float.toString(eye.y) + ", " + Float.toString(eye.z) + ".");
    }
     else if(which == 10)
    {
      eye.x = (float)2.0;
      eye.y = (float)-2.0;
      eye.z = (float)7.0;
      System.out.println("Eye = " + Float.toString(eye.x) + ", " + Float.toString(eye.y) + ", " + Float.toString(eye.z) + ".");
    }
     else if(which == 11)
    {
      eye.x = (float)0.0;
      eye.y = (float)2.0;
      eye.z = (float)-5.0;
      System.out.println("Eye = " + Float.toString(eye.x) + ", " + Float.toString(eye.y) + ", " + Float.toString(eye.z) + ".");
    }
     else
    {
      System.err.println("unknown RayTracer.init()");
    }
}


double getNumber(StreamTokenizer st) throws IOException
{
    if (st.nextToken() != StreamTokenizer.TT_NUMBER) {
        System.err.println("ERROR: number expected in line "+st.lineno());
        throw new IOException(st.toString());
    }
    return st.nval;
}

void WriteOutput(String outputFileName) throws IOException
{
  FileWriter fw = new FileWriter(outputFileName);

  // write out camera settings
  // format:  eye lookat up Du Dv fov
  fw.write(eye.toString());
  fw.write(lookat.toString());
  fw.write(up.toString());
  fw.write(Du.toString());
  fw.write(Dv.toString());
  fw.write(Float.toString(fov));
  fw.write("*");
  // write out pixel,depth info
  // format:  width, height
  // for each pixel, write color, depth
  Color cc;
  fw.write(Integer.toString(width)+","+Integer.toString(height));
  for(int y = 0; y < height; y++)
  {
    for(int x = 0; x < width; x++)
    {
      cc = raster.getColor(x,y);
      fw.write("["+Integer.toString(cc.getRed())+","+Integer.toString(cc.getGreen())+","+Integer.toString(cc.getBlue())+"]");
      fw.write(Float.toString(raster.getDepth(x, y))+"*");
    }
  }
  fw.close();
}
void ReadInput(InputStream is) throws IOException
{
        StreamTokenizer st = new StreamTokenizer(is);
    st.commentChar('#');
```

```
        scan: while (true) {
                switch (st.nextToken()) {
                  default:
                        break scan;
                  case StreamTokenizer.TT_WORD:
                    if (st.sval.equals("sphere")) {
                    Vector3D v = new Vector3D((float) getNumber(st), (float) getNumber(st), (float) getNumber(st));
                            float r = (float) getNumber(st);
                            objectList.addElement(new Sphere(currentSurface, v, r));
                            } else
                    if (st.sval.equals("triangle")) {
                    Vector3D a = new Vector3D((float) getNumber(st), (float) getNumber(st), (float)
getNumber(st));
                    Vector3D b = new Vector3D((float) getNumber(st), (float) getNumber(st), (float)
getNumber(st));
                    Vector3D c = new Vector3D((float) getNumber(st), (float) getNumber(st), (float)
getNumber(st));
                            objectList.addElement(new Triangle(currentSurface, a, b, c));
                            } else
                            if (st.sval.equals("eye")) {
                            eye = new Vector3D((float) getNumber(st), (float) getNumber(st), (float) getNumber(st));
                            } else
                            if (st.sval.equals("lookat")) {
                            lookat = new Vector3D((float) getNumber(st), (float) getNumber(st), (float) getNumber(st));
                            } else
                            if (st.sval.equals("up")) {
                            up = new Vector3D((float) getNumber(st), (float) getNumber(st), (float) getNumber(st));
                            } else
                            if (st.sval.equals("fov")) {
                    fov = (float) getNumber(st);
                            } else
                            if (st.sval.equals("background")) {
                    background = new Color((float) getNumber(st), (float) getNumber(st), (float) getNumber(st));
                            } else
                            if (st.sval.equals("light")) {
                                float r = (float) getNumber(st);
                                float g = (float) getNumber(st);
                                float b = (float) getNumber(st);
                            if (st.nextToken() != StreamTokenizer.TT_WORD) {
                        throw new IOException(st.toString());
                    }
                            if (st.sval.equals("ambient")) {
                                lightList.addElement(new Light(Light.AMBIENT, null, r, g, b));
                            } else
                            if (st.sval.equals("directional")) {
                                Vector3D v = new Vector3D((float) getNumber(st), (float) getNumber(st), (float)
getNumber(st));
                                lightList.addElement(new Light(Light.DIRECTIONAL, v, r, g, b));
                            } else
                            if (st.sval.equals("point")) {
                                Vector3D v = new Vector3D((float) getNumber(st), (float) getNumber(st), (float)
getNumber(st));
                                lightList.addElement(new Light(Light.POINT, v, r, g, b));
                            } else {
                                System.err.println("ERROR: in line "+st.lineno()+" at "+st.sval);
                                throw new IOException(st.toString());
                            }
                            } else
                            if (st.sval.equals("surface")) {
                                float r = (float) getNumber(st);
                                float g = (float) getNumber(st);
                                float b = (float) getNumber(st);
                            float ka = (float) getNumber(st);
                            float kd = (float) getNumber(st);
                            float ks = (float) getNumber(st);
                            float ns = (float) getNumber(st);
                            float kr = (float) getNumber(st);
                            float kt = (float) getNumber(st);
                            float index = (float) getNumber(st);
                            currentSurface = new Surface(r, g, b, ka, kd, ks, ns, kr, kt, index);
                            }
                            break;
                }
            }
        is.close();
            if (st.ttype != StreamTokenizer.TT_EOF)
                throw new IOException(st.toString());
        }

    public void paint(Graphics g)
    {
 //       Image output = raster.toImage(this);
  //      g.drawImage(output, 0, 0, this);
  //      output.dispose();
        raster.renderRaster(g);
    }

    // this overide avoid the unnecessary clear on each paint()
    public void update(Graphics g)
    {
        paint(g);
    }



    synchronized void renderPixel(int i, int j)
    {
        Vector3D dir = new Vector3D(
                                i*Du.x + j*Dv.x + Vp.x,
                                i*Du.y + j*Dv.y + Vp.y,
                                i*Du.z + j*Dv.z + Vp.z);
        Ray ray = new Ray(eye, dir);
        if (ray.trace(objectList)) {
            raster.setColorDepth(ray.Shade(lightList, objectList, background), i, j, eye);
        } else {
            DepthColor dc = new DepthColor(eye, background);
```

```
            raster.setColorDepth(dc, i, j, eye);
        }
    }

    public void run()
    {
        long time = System.currentTimeMillis();
        for (int j = 0; j < height; j++) {
            // System.out.println("Scanline "+j);
            for (int i = 0; i < width; i++) {
                renderPixel(i, j);
            }
            repaint();
        }
        time = System.currentTimeMillis() - time;
        System.out.println("Rendered in "+(time/60000)+":"+((time%60000)*0.001));

        // save file
  //      String filename = getParameter("outputfile");
   //     showStatus("Writing to file " + filename);
    //    OutputStream os = null;
     //   try {
      //      os = new URL(getDocumentBase(), filename).openStream();
       //     WriteOutput(os);
        //    is.close();
       // } catch (IOException e) {
       //     showStatus("Error reading "+filename);
       //     System.err.println("Error reading "+filename);
       // }
    }


    public boolean mouseDown(Event e, int x, int y)
    {
        return true;
    }

    public boolean mouseDrag(Event e, int x, int y)
    {
        return true;
    }


    public void processMouseEvent(MouseEvent e) {
      if(e.getID() == MouseEvent.MOUSE_RELEASED)
      {
        System.out.println("Eye = " + Float.toString(eye.x) + ", " + Float.toString(eye.y) + ", " + Float.toString(eye.z) + ".");
        System.out.println("Lookat = " + Float.toString(lookat.x) + ", " + Float.toString(lookat.y) + ", " + Float.toString(lookat.z) + ".");
       }
       else if(e.getID() == MouseEvent.MOUSE_PRESSED)
       {
          System.out.println("depth for "+Integer.toString(e.getX())+","+
              Integer.toString(e.getY())+" = " + Float.toString(raster.getDepth(e.getX(),e.getY())));
       }
    }
  }
```

9 ldi2ppm LISTING (extract given depth layer of ldi)

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <assert.h>
#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glut.h>
#include <math.h>

#include "proc_ppm.h"
#include "ldi.h"

#define DEFAULT_Z       0
#define DEFAULT_SPLAT   1

#define TEST_FILENAME1   "test1.ldi"
#define TEST_FILENAME2   "test2.ldi"


int level = 0;

int
BuildDepthImage(LayeredDepthImagePtr curLdiPtr,
MyImagePtr imgp, FILE *depthFP)
{
  int x, y, yoffset, rgba;
  struct DepthPixel curP;
  double depth, depthScaled;
  double avgDepth = 0, minDepth = 0, maxDepth = 0;
  int    totalDepthPixels = 0;
  double eye[3], center[3], up[3], frustum[6];
  int    view[4];
  int    cr, cg, cb;

  curLdiPtr->xres = imgp->width;
  curLdiPtr->yres = imgp->height;
  curLdiPtr->pixels = malloc(sizeof(struct LayeredDepthPixel) * (curLdiPtr->xres * curLdiPtr->yres));
  assert(curLdiPtr->pixels != NULL);
  curLdiPtr->camera = CameraGL_Create();


  for(y = 0; y < curLdiPtr->yres; y++)
  {
```

```
    yoffset = y * curLdiPtr->xres;
    for(x = 0; x < curLdiPtr->xres; x++)
    {
curLdiPtr->pixels[ yoffset + x ].numLayers = 0;
curLdiPtr->pixels[ yoffset + x ].layers = NULL;
rgba = 0;
cr = GET_RED(imgp->data[yoffset + x]);
cg = GET_GREEN(imgp->data[yoffset + x]);
cb = GET_BLUE(imgp->data[yoffset + x]);

SET_RED(rgba, cr);
SET_GREEN(rgba, cg);
SET_BLUE(rgba, cb);
/*
fprintf(stdout, "[r g b m]:  %d %d %d %d\n", cr, cg, cb, imgp->maxvalue);
*/
DepthPixel_SetRGBA(&curP, rgba);
if(depthFP != NULL)
{
if(fscanf(depthFP, "%lg %lg\n", &depth, &depthScaled) != 2)
{
DepthPixel_SetZ(&curP, DEFAULT_Z);
fprintf(stderr, "Error - using DEFAULT_Z [%d %d]\n", x, y);
}
else
{
DepthPixel_SetZ(&curP, (int)depthScaled);
avgDepth += depthScaled;
if(totalDepthPixels == 0)
{
  minDepth = maxDepth = depthScaled;
}
else
{
  if(minDepth > depthScaled)
  {
    minDepth = depthScaled;
  }
  if(maxDepth < depthScaled)
  {
    maxDepth = depthScaled;
  }
}
totalDepthPixels++;
}
}
else
{
DepthPixel_SetZ(&curP, DEFAULT_Z);
}
DepthPixel_SetSplatIndex(&curP, DEFAULT_SPLAT);

if(LayeredDepthPixel_AddDepthPixelLayer(&(curLdiPtr->pixels[ yoffset + x ]), &curP))
{
fprintf(stderr, "Error adding depth pixel\n");
return(1);
}
    }
  }

  if(totalDepthPixels > 0)
  {
    avgDepth /= totalDepthPixels;
  }


  /* configure camera */
  center[0] = curLdiPtr->xres / 2;
  center[1] = curLdiPtr->yres / 2;
  center[2] = avgDepth;

  eye[0] = center[0];
  eye[1] = center[1];
  eye[2] = minDepth + 10;

  up[0] = 0.0;
  up[1] = 1.0;
  up[2] = 0.0;
  CameraGL_SetLookAt(curLdiPtr->camera, eye, center, up);

  frustum[0] = 0.0; /* left is 0 */
  frustum[1] = curLdiPtr->xres;
  frustum[2] = curLdiPtr->yres;
  frustum[3] = 0.0; /* top is 0 */
  frustum[4] = 9.0;
  frustum[5] = fabs(maxDepth) + 11.0;
  CameraGL_SetFrustum(curLdiPtr->camera, frustum);

  view[0] = 0;
  view[1] = 0;
  view[2] = curLdiPtr->xres;
  view[3] = curLdiPtr->yres;
  CameraGL_SetViewMat(curLdiPtr->camera, view);

  /* indicate success */
  return(0);
}


MyImagePtr
BuildPpm(LayeredDepthImagePtr curPtr)
{
  int y, x, yoffset;
  PixelType rgba;
  MyImagePtr imgp;
```

```
  imgp = malloc(sizeof(struct MyImage));
  assert(imgp != NULL);

  imgp->width = curPtr->xres;
  imgp->height = curPtr->yres;
  imgp->data = malloc(sizeof(PixelType) * (imgp->width * imgp->height));
  assert(imgp->data != NULL);
  imgp->maxvalue = 255;

  for(y = 0; y < imgp->height; y++)
    {
      yoffset = y * imgp->width;
      for(x = 0; x < imgp->width; x++)
{
  rgba = 0;
  if(curPtr->pixels[yoffset + x].numLayers >= (level-1))
  {
  SET_RED(rgba, GET_RED(curPtr->pixels[yoffset + x].layers[level].colorRGBA));
  SET_GREEN(rgba, GET_GREEN(curPtr->pixels[yoffset + x].layers[level].colorRGBA));
  SET_BLUE(rgba, GET_BLUE(curPtr->pixels[yoffset + x].layers[level].colorRGBA));
  }
  else
  {
  SET_RED(rgba, 0);
  SET_GREEN(rgba, 0);
  SET_BLUE(rgba, 0);

  }
  imgp->data[yoffset + x] = rgba;
}
    }

  return(imgp);
}


void
Usage(char *name)
{
 fprintf(stdout, "%s {ldifilename} {level-to-splat}\n", name);

}


void init(void)
{
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  glMatrixMode(GL_MODELVIEW);
  glLoadIdentity();
}

void display(void)
{
  int length = 5;

   glClearColor(1,1,1,1);
   glClear(GL_COLOR_BUFFER_BIT);
   glClearDepth(1.0);
   glClear(GL_DEPTH_BUFFER_BIT);

        glColor3d (1.0, 0.0, 0.0);
        glBegin(GL_LINES);
        glVertex3d(0.0,0.0,0.0);                /* x axis */
        glVertex3d(length,0.0,0.0);
        glColor3d (0.0, 1.0, 0.0);
        glVertex3d(0.0,0.0,0.0);                /* y axis */
        glVertex3d(0.0,length,0.0);
        glColor3d (0.0, 0.0, 1.0);
        glVertex3d(0.0,0.0,0.0);                /* z axis */
        glVertex3d(0.0,0.0,length);
        glEnd();

   glutSwapBuffers();

}
int
main(int argc, char *argv[])
{
  LayeredDepthImagePtr curLdiPtr;
  FILE *depthFP;
  MyImagePtr imgp;
  char newName[256];
  char *ch;

  if(argc != 3)
    {
      Usage(argv[0]);
      exit(1);
    }

  level = atoi(argv[2]);

  strcpy(newName, argv[1]);
  ch = strrchr(newName, '.');
  if(ch != NULL)
  {
    *ch = '\0';
  }
  strcat(newName, ".ppm");


   glutInit(&argc, argv);
   glutInitDisplayMode (GLUT_DOUBLE | GLUT_RGB);
   glutInitWindowSize (400, 300);
   glutInitWindowPosition (100, 100);
```

```
   glutCreateWindow (argv[0]);
   glutDisplayFunc(display);

  curLdiPtr = LayeredDepthImage_Create();

  fprintf(stdout, "Reading Layered Depth Image\n");
  if(LayeredDepthImage_ReadFromFile(curLdiPtr, argv[1]))
  {
    fprintf(stderr, "Error building depth image\n");
    exit(1);
  }

  fprintf(stdout, "Creating ppm\m");
  imgp = BuildPpm(curLdiPtr);

  fprintf(stdout, "Initial write of ppm\n");
  writeppm(imgp, newName, argv[0]);


  fprintf(stdout, "Destroying current LDI\n");


  LayeredDepthImage_Destroy(curLdiPtr);
  /*
  curLdiPtr = LayeredDepthImage_Create();
  fprintf(stdout, "Reading written LDI\n");
  if(LayeredDepthImage_ReadFromFile(curLdiPtr, TEST_FILENAME1))
  {
fprintf(stderr, "Error reading ldi file %s\n", TEST_FILENAME1);
  }

  fprintf(stdout, "Writing just-read LDI\n");
  if(LayeredDepthImage_WriteToFile(curLdiPtr, TEST_FILENAME2))
  {
fprintf(stderr, "Error writing ldi to file %s\n", TEST_FILENAME2);
exit(1);
  }

  fprintf(stdout, "Destroying current LDI\n");
  LayeredDepthImage_Destroy(curLdiPtr);
  */

  exit(0);
}
```

10 ldiview LISTING (layered depth image renderer)

```
/*
** ldi_display.c
**
** $Header: /cse/grads/ljderman/Cmps260/FinalProj/LdiCode/RCS/ldi_display.c,v 1.1 1999/03/02 04:03:26 ljderman Exp ljderman $
**
** DESCRIPTION:
**   Implementation of routines which implement LDI rendering.
**
** MODIFICATION HISTORY:
**   $Log: ldi_display.c,v $
**   Revision 1.1  1999/03/02 04:03:26  ljderman
**   Initial revision
**
**
*/


#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glut.h>
#include <stdlib.h>
#include <stdio.h>
#include <assert.h>



#include "ldi.h"
#include "proc_ppm.h"
#include "matrix.h"


#define DEBUG_LDI_DISPLAY 1


/*
#define DEBUG_LDI_DISPLAY_SPLAT
*/

char lookupTable[2048];


void
ComputeSplatLookup()
{
  int i;

  for(i = 0; i < 2048; i++)
  {
    lookupTable[i] = 5;
  }
}


void
Splat(int rgba, int x, int y, int z, char sqrtSize)
```

```
{
  /*
  glColor3d((double)GET_RED(rgba) / 255.0,
    (double)GET_GREEN(rgba) / 255.0,
    (double)GET_BLUE(rgba) / 255.0);
  glPushMatrix();
  glTranslated((double)x, (double)y, 0.0);
  glutSolidCube(sqrtSize * sqrtSize);
  glPopMatrix();
  */
#ifdef DEBUG_LDI_DISPLAY_SPLAT
  fprintf(stdout, "Splatting [x y rgba sqrtsize] [%d %d %d %d]\n",
  x, y, rgba, sqrtSize);
#endif

  glPointSize(sqrtSize * sqrtSize);
  glBegin(GL_POINTS);
  glColor3d((double)GET_RED(rgba) / 255.0,
    (double)GET_GREEN(rgba) / 255.0,
    (double)GET_BLUE(rgba) / 255.0);
  glVertex3i(x, y, z);
  glEnd();
}


int
IsInViewPort(double cp[4])
{
  return(1);
}


void
Warp(LayeredDepthPixelPtr ldpPtr, double start[4], double depth[4], double xincr[4])
{
  int l;
  double result[4];
  double ir[4];
  int z1;
  char sqrtSize;

  for(l = 0; l < ldpPtr->numLayers; l++)
  {
    z1 = ldpPtr->layers[l].z;

    /* compute z1 * depth */
    ir[0] = z1 * depth[0]; ir[1] = z1 * depth[1];
    ir[2] = z1 * depth[2]; ir[3] = z1 * depth[3];

    /* compute result (start + ir) */
    result[0] = ir[0] + start[0]; result[1] = ir[1] + start[1];
    result[2] = ir[2] + start[2]; result[3] = ir[3] + start[3];

    /* cull if pixel goes behind output camera or out of output cam's frustum */
    if((result[3] > 0) && (IsInViewPort(result)))
    {
      result[0] /= result[3]; result[1] /= result[3];
      result[2] /= result[3]; result[3] /= result[3];
      sqrtSize = result[2] * lookupTable[ldpPtr->layers[l].splatIndex];
      Splat(ldpPtr->layers[l].colorRGBA, result[0], result[1], result[2], sqrtSize);
    }

    /* increment start */
    start[0] += xincr[0]; start[1] += xincr[1];
    start[2] += xincr[2]; start[3] += xincr[3];
  }
}

void
SetGLToCamera(CameraGLPtr outCam)
{
    glMatrixMode(GL_PROJECTION);
    glLoadMatrixd(outCam->projMat);
    glMatrixMode(GL_MODELVIEW);
    glLoadMatrixd(outCam->modelMat);
    glViewport(outCam->viewMat[0], outCam->viewMat[1],
       outCam->viewMat[2], outCam->viewMat[3]);
}


void
WarpGL(LayeredDepthPixelPtr ldpPtr, CameraGLPtr ldiCamPtr, int x, int y)
{
  int l;
  double obj[3];
  int z1;
  char sqrtSize;

  /*
  ** perform warp by doing an unProject from x,y,depth in ldi view.
  ** then projecting corresponding winx,winy,winz coord to outCam view.
  */

  for(l = 0; l < ldpPtr->numLayers; l++)
  {
    z1 = ldpPtr->layers[l].z;
    assert(gluUnProject(x, y, z1, ldiCamPtr->modelMat, ldiCamPtr->projMat, ldiCamPtr->viewMat,
 &obj[0], &obj[1], &obj[2]) == GL_TRUE);
    sqrtSize = obj[2] * lookupTable[ldpPtr->layers[l].splatIndex];
    Splat(ldpPtr->layers[l].colorRGBA, obj[0], obj[1], obj[2], sqrtSize);
  }
}



void
```

```
RenderLdi_WarpGL(LayeredDepthImagePtr curPtr, CameraGLPtr outCam)
{
  GLint w, h, x, y, yoffset, l, z;
  GLint r, g, b;
  GLdouble rd, gd, bd;


  /* precompute splat lookup table */
  ComputeSplatLookup();


  w = curPtr->xres;
  h = curPtr->yres;


  /* set openGL to output camera */
  SetGLToCamera(outCam);

  for(y = 0; y < h; y++)
  {
      yoffset = y * w;

      for(x = 0; x < w; x++)
      {
WarpGL(&(curPtr->pixels[yoffset + x]), curPtr->camera, x, y);
      }
  }

#ifdef DEBUG_LDI_DISPLAY
  fprintf(stderr, "LDI rendered (RenderLdi_WarpGL)\n");
#endif
}


void
RenderLdi_Warp(LayeredDepthImagePtr curPtr, CameraGLPtr outCam)
{
  GLint w, h, x, y, yoffset, l, z;
  GLint r, g, b;
  GLdouble rd, gd, bd;

  double c1[16];
  double c1i[16];
  double c2[16];
  double t12[16];

  double ipt[4];

  double start[4];
  double depth[4];
  double result[4];
  double xincr[4];
  double d0[4] = {0.0, 0.0, 1.0, 0.0};
  double x0[4] = {1.0, 0.0, 0.0, 0.0};

  int numLDPs = 0;

  /* compute matrix t12, which is C2 * C1^-1 */
  multiplyMatrices(c1,  curPtr->camera->projMat, curPtr->camera->modelMat);
  /* multiply c1 by its viewport matrix !!! */

  invertMatrix(c1i, c1);

  multiplyMatrices(c2, outCam->projMat, outCam->modelMat);
  /* multiply c2 by its viewport matrix !!! */

  multiplyMatrices(t12, c2, c1i);



  /* precompute splat lookup table */
  ComputeSplatLookup();


  w = curPtr->xres;
  h = curPtr->yres;


  /* compute depth component */
  transformPoint(depth, d0, t12);

  /* compute xincr component */
  transformPoint(xincr, x0, t12);

  /* set openGL to output camera */
  SetGLToCamera(outCam);

  for(y = 0; y < h; y++)
  {
      yoffset = y * w;

      /* compute start */
      ipt[0] = x;
      ipt[1] = y;
      ipt[2] = 0;
      ipt[3] = 1;
      transformPoint(start, ipt, t12);

      for(x = 0; x < w; x++)
      {
numLDPs++;
Warp(&(curPtr->pixels[yoffset + x]), start, depth, xincr);
      }
  }

  fprintf(stderr, "LDI rendered (%d LDP's)\n", numLDPs);
```

```
}



void
RenderLdi_Simple(LayeredDepthImagePtr curPtr)
{
  GLint w, h, x, y, yoffset, l, z;
  GLint r, g, b;
  GLdouble rd, gd, bd;


  /* precompute splat lookup table */



  /* not implemented - just render as points */
  w = curPtr->xres;
  h = curPtr->yres;

  glBegin(GL_POINTS);
  for(y = 0; y < h; y++)
  {
      yoffset = y * w;
      for(x = 0; x < w; x++)
      {
/* for each layer */
for(l = 0; l < curPtr->pixels[x + yoffset].numLayers; l++)
{
r = (GLint)GET_RED(curPtr->pixels[x + yoffset].layers[l].colorRGBA);
g = (GLint)GET_GREEN(curPtr->pixels[x + yoffset].layers[l].colorRGBA);
b = (GLint)GET_BLUE(curPtr->pixels[x + yoffset].layers[l].colorRGBA);
z = (GLint)(curPtr->pixels[x + yoffset].layers[l].z);

#ifdef DEBUG_LDI_DISPLAY
/*
fprintf(stdout, "[%d %d %d]: [%d %d %d %d]\n", x, y, l, r, g, b, z);
*/
#endif
glColor3d((double)r / 255.0, (double)g / 255.0, (double)b / 255.0);
glVertex3d((GLdouble)x, (GLdouble)y, (GLdouble)z);
}
      }
  }
  glEnd();

#ifdef DEBUG_LDI_DISPLAY
  fprintf(stderr, "LDI rendered\n");
#endif
}


void
RenderLdi_GLonly(LayeredDepthImagePtr curPtr, CameraGLPtr outCamPtr)
{
  GLint w, h, x, y, yoffset, l, z;
  GLint r, g, b, a;
  GLdouble rd, gd, bd;
  double N[4][4];
  double c[3];
  char sqrtSize = 1;


  c[0] = curPtr->camera->center[0] - curPtr->camera->eye[0];
  c[1] = curPtr->camera->center[1] - curPtr->camera->eye[1];
  c[2] = curPtr->camera->center[2] - curPtr->camera->eye[2];

  glMatrixMode(GL_MODELVIEW);
  glLoadIdentity();
  SetGLToCamera(outCamPtr);
  N[0][0] = curPtr->camera->du[0]; N[1][0] = curPtr->camera->dv[0]; N[2][0] = c[0]; N[3][0] = curPtr->camera->eye[0];
  N[0][1] = curPtr->camera->du[1]; N[1][1] = curPtr->camera->dv[1]; N[2][1] = c[1]; N[3][1] = curPtr->camera->eye[1];
  N[0][2] = curPtr->camera->du[2]; N[1][2] = curPtr->camera->dv[2]; N[2][2] = c[2]; N[3][2] = curPtr->camera->eye[2];
  N[0][3] = 0; N[1][3] = 0; N[2][3] = 0; N[3][3] = 1;
  glMultMatrixd(N);


  w = curPtr->xres;
  h = curPtr->yres;

  glBegin(GL_POINTS);
  for(y = 0; y < h; y++)
  {
      yoffset = y * w;
      for(x = 0; x < w; x++)
      {
/* for each layer */
for(l = 0; l < curPtr->pixels[x + yoffset].numLayers; l++)
{
  /*
r = (GLint)GET_RED(curPtr->pixels[x + yoffset].layers[l].colorRGBA);
g = (GLint)GET_GREEN(curPtr->pixels[x + yoffset].layers[l].colorRGBA);
b = (GLint)GET_BLUE(curPtr->pixels[x + yoffset].layers[l].colorRGBA);
z = (GLint)(curPtr->pixels[x + yoffset].layers[l].z);

a = (GLint)GET_ALPHA(curPtr->pixels[x + yoffset].layers[l].colorRGBA);
*/
/*
glColor3d((double)r / 255.0, (double)g / 255.0,
  (double)b / 255.0);
glVertex3d((GLdouble)x, (GLdouble)y, (GLdouble)l);
*/
Splat(curPtr->pixels[x + yoffset].layers[l].colorRGBA, x, y, z,
      sqrtSize);
}
```

```
      }
  }
  glEnd();

#ifdef DEBUG_LDI_DISPLAY
  fprintf(stderr, "LDI rendered\n");
#endif
}
/*
** ldi.h
**
** $Header: /cse/grads/ljderman/Cmps260/FinalProj/RCS/ldi.h,v 1.1 1999/02/28 00:45:03 ljderman Exp ljderman $
**
** DESCRIPTION:
**   Interface to Ldi ADT.
**
** MODIFICATION HISTORY:
**   $Log: ldi.h,v $
**   Revision 1.1  1999/02/28 00:45:03  ljderman
**   Initial revision
**
*/

#ifndef __ldi_h__
#define __ldi_h__

#include "camera.h"


#define GET_Z(x)         (x->z)
#define SET_Z(x, y)      (x->z = y)
#define SET_SPLAT(x,y)   (x->splatIndex = y)
#define GET_SPLAT(x)     (x->splatIndex)
#define LDI_INDEX(x, y, xr, yr)   ((y * xr) + x)


typedef struct DepthPixel *DepthPixelPtr;
struct DepthPixel
{
int  colorRGBA;       /* 32-bit integer */
int  z;
        int                     splatIndex;
};

typedef struct LayeredDepthPixel *LayeredDepthPixelPtr;
struct LayeredDepthPixel
{
int  numLayers;  /* number of layers */
DepthPixelPtr  layers;    /* pointer to array of DepthPixels w/ dim numLayers */
};

typedef struct LayeredDepthImage *LayeredDepthImagePtr;
struct LayeredDepthImage
{
CameraGLPtr camera;     /* pointer to the LDI camera */
int  xres;       /* x resolution (in pixels) of the LDI image */
int yres;       /* y resolution (in pixels) of the LDI image */
LayeredDepthPixelPtr pixels;    /* pointer to array of LayeredDepthPixels w/ dim (xres * yres) */
};



/*
*************
*************
** prototypes
*************
*************
*/




/*
**************
** depth pixel
**************
*/

/* constructor */
DepthPixelPtr
DepthPixel_Create(void);


/* destructor */
void
DepthPixel_Destroy(DepthPixelPtr curPtr);

/* state manipulation */
int
DepthPixel_GetRGBA(DepthPixelPtr curPtr);

int
DepthPixel_GetZ(DepthPixelPtr curPtr);

int
DepthPixel_GetSplatIndex(DepthPixelPtr curPtr);

int
DepthPixel_SetRGBA(DepthPixelPtr curPtr, int n);

int
DepthPixel_SetZ(DepthPixelPtr curPtr, int n);

int
DepthPixel_SetSplatIndex(DepthPixelPtr curPtr, int n);
```

```
/* fileio */
int
DepthPixel_WriteToStream(DepthPixelPtr curPtr, FILE *fp);

int
DepthPixel_ReadFromStream(DepthPixelPtr curPtr, FILE *fp);



/*
**********************
** layered depth pixel
**********************
*/

/* constructor */
LayeredDepthPixelPtr
LayeredDepthPixel_Create(void);

/* destructor */
void
LayeredDepthPixel_Destroy(LayeredDepthPixelPtr curPtr);


int
LayeredDepthPixel_AddDepthPixelLayer(LayeredDepthPixelPtr curPtr, DepthPixelPtr curDPPtr);

int
LayeredDepthPixel_GetNumLayeres(LayeredDepthPixelPtr curPtr);

DepthPixelPtr
LayeredDepthPixel_GetDepthPixel(LayeredDepthPixelPtr curPtr, int layer);

/* fileio */
int
LayeredDepthPixel_WriteToStream(LayeredDepthPixelPtr curPtr, FILE *fp);

int
LayeredDepthPixel_ReadFromStream(LayeredDepthPixelPtr curPtr, FILE *fp);



/*
**********************
** layered depth image
**********************
*/

/* constructor */
LayeredDepthImagePtr
LayeredDepthImage_Create(void);


/* destructor */
void
LayeredDepthImage_Destroy(LayeredDepthImagePtr curPtr);


/* fileio */
int
LayeredDepthImage_WriteToStream(LayeredDepthImagePtr curPtr, FILE *fp);

int
LayeredDepthImage_ReadFromStream(LayeredDepthImagePtr curPtr, FILE *fp);

int
LayeredDepthImage_ReadFromFile(LayeredDepthImagePtr curPtr, char *fileName);

int
LayeredDepthImage_WriteToFile(LayeredDepthImagePtr curPtr, char *fileName);

/*
** DESCRIPTION:
**   There should be 3 files:
**          {basename}.ppm  -  rgb intensity values of original image
**          {basename}.cam  -  camera (GL) specification file
**          {basename}.dep  -  pixel depth file
**
**  Information from these three files is read in and an LDI is constructed.
*/
int
LayeredDepthImage_ReadFromImageWDepth(LayeredDepthImagePtr curPtr, char *baseName);


void
LayeredDepthImage_Display(LayeredDepthImagePtr curPtr, CameraGLPtr outputCameraPtr);

void
LayeredDepthImage_PrintVitalStats(LayeredDepthImagePtr curPtr, FILE *fp);

#endif


/*
** ldi.c
**
** $Header: /cse/grads/ljderman/Cmps260/FinalProj/RCS/ldi.c,v 1.1 1999/02/28 00:45:22 ljderman Exp ljderman $
**
** DESCRIPTION:
**   Implementation of Ldi ADT.
**
** MODIFICATION HISTORY:
**   $Log: ldi.c,v $
**   Revision 1.1  1999/02/28 00:45:22  ljderman
```

```
**   Initial revision
**
*/


#include <stdlib.h>
#include <stdio.h>
#include <assert.h>

#include "ldi.h"
#include "ldi_display.h"

/*
#define DEBUG_LDI
*/
/*
#define DEBUG_LDI_STATUS
*/

/*
**************
** depth pixel
**************
*/

/* constructor */
DepthPixelPtr
DepthPixel_Create(void)
{
  DepthPixelPtr curPtr;


  curPtr = malloc(sizeof(struct DepthPixel));
  assert(curPtr != NULL);

  curPtr->colorRGBA = 0;
  curPtr->z = 0;
  curPtr->splatIndex = 0;

  return(curPtr);
}


/* destructor */
void
DepthPixel_Destroy(DepthPixelPtr curPtr)
{
  free(curPtr);
}



/* state manipulation */
int
DepthPixel_GetRGBA(DepthPixelPtr curPtr)
{
  return(curPtr->colorRGBA);
}

int
DepthPixel_GetZ(DepthPixelPtr curPtr)
{
  return(GET_Z(curPtr));
}


int
DepthPixel_GetSplatIndex(DepthPixelPtr curPtr)
{
  return(GET_SPLAT(curPtr));
}


int
DepthPixel_SetRGBA(DepthPixelPtr curPtr, int n)
{
  curPtr->colorRGBA = n;
}


int
DepthPixel_SetZ(DepthPixelPtr curPtr, int n)
{
  SET_Z(curPtr, n);
}


int
DepthPixel_SetSplatIndex(DepthPixelPtr curPtr, int n)
{
  SET_SPLAT(curPtr, n);
}


/* fileio */
int
DepthPixel_WriteToStream(DepthPixelPtr curPtr, FILE *fp)
{
  fwrite(curPtr, sizeof(struct DepthPixel), 1, fp);
  return(0);
}

int
DepthPixel_ReadFromStream(DepthPixelPtr curPtr, FILE *fp)
{
  int numI;
```

```
  numI = fread(curPtr, sizeof(struct DepthPixel), 1, fp);
  if(numI != sizeof(struct DepthPixel))
  {
    if(ferror(fp))
    {
      fprintf(stderr, "Error during DepthPixel_ReadFromStream\n");
      return(1);
    }
    if(feof(fp))
    {
      return(1);
    }
  }

  return(0);
}



/*
**********************
** layered depth pixel
**********************
*/

/* constructor */
LayeredDepthPixelPtr
LayeredDepthPixel_Create(void)
{
  LayeredDepthPixelPtr curPtr;


  curPtr = malloc(sizeof(struct LayeredDepthPixel));
  assert(curPtr != NULL);

  curPtr->numLayers = 0;
  curPtr->layers = NULL;
}


/* destructor */
void
LayeredDepthPixel_Destroy(LayeredDepthPixelPtr curPtr)
{
  if(curPtr->layers != NULL)
  {
    free(curPtr->layers);
  }
  free(curPtr);
}

int
LayeredDepthPixel_AddDepthPixelLayer(LayeredDepthPixelPtr curPtr, DepthPixelPtr curDPPtr)
{
curPtr->numLayers++;

if(curPtr->layers == NULL)
{
curPtr->layers = malloc(sizeof(struct DepthPixel));
assert(curPtr->layers);
}
else
{
curPtr->layers = realloc(curPtr->layers, curPtr->numLayers * sizeof(struct DepthPixel));
assert(curPtr->layers);
}

DepthPixel_SetRGBA(&(curPtr->layers[curPtr->numLayers - 1]), DepthPixel_GetRGBA(curDPPtr));
DepthPixel_SetZ(&(curPtr->layers[curPtr->numLayers - 1]), DepthPixel_GetZ(curDPPtr));
DepthPixel_SetSplatIndex(&(curPtr->layers[curPtr->numLayers - 1]), DepthPixel_GetSplatIndex(curDPPtr));

return(0);
}

int
LayeredDepthPixel_GetNumLayeres(LayeredDepthPixelPtr curPtr)
{
  return(curPtr->numLayers);
}

DepthPixelPtr
LayeredDepthPixel_GetDepthPixel(LayeredDepthPixelPtr curPtr, int layer)
{
  return(&(curPtr->layers[layer]));
}


/* fileio */
int
LayeredDepthPixel_WriteToStream(LayeredDepthPixelPtr curPtr, FILE *fp)
{
  int i;


  /* write number of layers */
  fwrite(&(curPtr->numLayers), sizeof(int), 1, fp);

  /* loop, writing each DepthPixel */
  for(i = 0; i < curPtr->numLayers; i++)
  {
    if(DepthPixel_WriteToStream(&(curPtr->layers[i]), fp))
    {
      fprintf(stderr, "Error writing depth pixel to stream\n");
      return(1);
    }
```

```
  }

  return(0);
}

int
LayeredDepthPixel_ReadFromStream(LayeredDepthPixelPtr curPtr, FILE *fp)
{
  int i, num;
  struct DepthPixel dp;

  /* read number of layers */

  fread(&num, sizeof(int), 1, fp);
#ifdef DEBUG_LDI
fprintf(stdout, " %d depth layers\n", curPtr->numLayers);
#endif

  curPtr->numLayers = 0;
  /* loop, reading each DepthPixel */
  for(i = 0; i < num; i++)
  {
    if(DepthPixel_ReadFromStream(&dp, fp))
    {
      fprintf(stderr, "Error reading depth pixel from stream\n");
      return(1);
    }
    if(LayeredDepthPixel_AddDepthPixelLayer(curPtr, &dp))
    {
      fprintf(stderr, "Error adding depth pixel layer (LayeredDepthPixel_ReadFromStream)\n");
      return(1);
    }
  }

  /* check that curPtr->numLayers matches value it should */
  assert(curPtr->numLayers == num);

  return(0);
}


/*
**********************
** layered depth image
**********************
*/

/* constructor */
LayeredDepthImagePtr
LayeredDepthImage_Create(void)
{
  LayeredDepthImagePtr curPtr;


  curPtr = malloc(sizeof(struct LayeredDepthImage));
  assert(curPtr != NULL);

  curPtr->camera = NULL;
  curPtr->xres = 0;
  curPtr->yres = 0;
  curPtr->pixels = NULL;

  return(curPtr);
}


/* destructor */
void
LayeredDepthImage_Destroy(LayeredDepthImagePtr curPtr)
{
  int x, y;

  if(curPtr->camera)
  {
    CameraGL_Destroy(curPtr->camera);
  }

  if(curPtr->pixels)
  {
    for(y = 0; y < curPtr->yres; y++)
    {
      for(x = 0; x < curPtr->xres; x++)
      {
LayeredDepthPixel_Destroy(&(curPtr->pixels[LDI_INDEX(x, y, curPtr->xres, curPtr->yres)]));
      }
    }
  }
}


/* fileio */
int
LayeredDepthImage_WriteToStream(LayeredDepthImagePtr curPtr, FILE *fp)
{
  int x, y, yoffset;


  /* write camera to stream */
  if(CameraGL_WriteToStream(curPtr->camera, fp))
  {
    fprintf(stderr, "Error writing CameraGL to stream\n");
    return(1);
  }


  /* write layered depth pixels to stream */
```

```
  fwrite(&(curPtr->xres), sizeof(int), 1, fp);
  fwrite(&(curPtr->yres), sizeof(int), 1, fp);

  for(y = 0; y < curPtr->yres; y++)
  {
    yoffset = y * curPtr->xres;

#ifdef DEBUG_LDI_STATUS
    fprintf(stdout, "Ypos %d\n", y);
#endif

    for(x = 0; x < curPtr->xres; x++)
    {
      if(LayeredDepthPixel_WriteToStream(
/* &(curPtr->pixels[LDI_INDEX(x, y, curPtr->xres, curPtr->yres)]), */
   &(curPtr->pixels[yoffset + x]),
      fp))
      {
fprintf(stderr, "Error writing layered depth pixel to stream\n");
return(1);
      }
    }
  }

  return(0);
}

int
LayeredDepthImage_ReadFromStream(LayeredDepthImagePtr curPtr, FILE *fp)
{
  int x, y, yoffset;


  /* read camera from stream */
#ifdef DEBUG_LDI
fprintf(stdout, "Reading CameraGL\n");
#endif
  curPtr->camera = CameraGL_Create();

  if(CameraGL_ReadFromStream(curPtr->camera, fp))
  {
    fprintf(stderr, "Error reading CameraGL from stream\n");
    return(1);
  }


  /* read layered depth pixels from stream */
  fread(&(curPtr->xres), sizeof(int), 1, fp);
  fread(&(curPtr->yres), sizeof(int), 1, fp);
  curPtr->pixels = malloc(sizeof(struct LayeredDepthPixel) * (curPtr->xres * curPtr->yres));
  assert(curPtr->pixels != NULL);

#ifdef DEBUG_LDI
fprintf(stdout, "dim [%d %d]\n", curPtr->xres, curPtr->yres);
#endif

  for(y = 0; y < curPtr->yres; y++)
  {
    yoffset = y * curPtr->xres;

#ifdef DEBUG_LDI_STATUS
    fprintf(stdout, "Ypos %d\n", y);
#endif

    for(x = 0; x < curPtr->xres; x++)
    {
      if(LayeredDepthPixel_ReadFromStream(
/*    &(curPtr->pixels[LDI_INDEX(x, y, curPtr->xres, curPtr->yres)]), */
&(curPtr->pixels[yoffset + x]),
   fp))
      {
fprintf(stderr, "Error reading layered depth pixel from stream\n");
return(1);
      }
    }
  }

  return(0);
}


int
LayeredDepthImage_ReadFromFile(LayeredDepthImagePtr curPtr, char *fileName)
{
  FILE *fp;

  fp = fopen(fileName, "r");
  if(fp == NULL)
    {
      fprintf(stderr, "Error opening file %s\n", fileName);
      return(1);
    }

  if(LayeredDepthImage_ReadFromStream(curPtr, fp))
  {
    fprintf(stderr, "Error reading from stream\n");
    return(1);
  }
  fclose(fp);

  return(0);
}


int
LayeredDepthImage_WriteToFile(LayeredDepthImagePtr curPtr, char *fileName)
```

```
{
  FILE *fp;

  fp = fopen(fileName, "w");
  if(fp == NULL)
    {
      fprintf(stderr, "Error opening file %s\n", fileName);
      return(1);
    }

  if(LayeredDepthImage_WriteToStream(curPtr, fp))
  {
    fprintf(stderr, "Error reading from stream\n");
    return(1);
  }
  fclose(fp);

  return(0);
}

/*
** DESCRIPTION:
**   There should be 3 files:
**          {basename}.ppm  -  rgb intensity values of original image
**          {basename}.cam  -  camera (GL) specification file
**          {basename}.dep  -  pixel depth file
**
**  Information from these three files is read in and an LDI is constructed.
*/
int
LayeredDepthImage_ReadFromImageWDepth(LayeredDepthImagePtr curPtr, char *baseName)
{
  fprintf(stdout, "LayeredDepthImage_ReadFromImageWDepth():  not-implemented\n");
  return(1);
}


void
LayeredDepthImage_Display(LayeredDepthImagePtr curPtr, CameraGLPtr outputCameraPtr)
{
  fprintf(stdout, "LayeredDepthImage_Display():  not-implemented\n");
}


void
LayeredDepthImage_PrintVitalStats(LayeredDepthImagePtr curPtr, FILE *fp)
{
  fprintf(fp, "LDI dim[%d %d]\n", curPtr->xres, curPtr->yres);
  if(curPtr->camera != NULL)
  {
    CameraGL_PrintPrettyToStream(curPtr->camera, fp);
  }
  else
  {
    fprintf(stdout, "<No camera>\n\n");
  }
}
```

11 jrt2ldi LISTING (convert ray-tracer output file to ldi)

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <assert.h>
#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glut.h>
#include <math.h>

#include "proc_ppm.h"
#include "ldi.h"
#include "matrix.h"

#define DEFAULT_Z       0
#define DEFAULT_SPLAT   1


LayeredDepthImagePtr theLdiPtr;
int lookupTable[2048];

void
readVector(FILE *curFP, double *v)
{
  int ch;
  char buffer[256];
  char *b, *nb;

  ch = getc(curFP);
  if(ch == EOF)
    {
      return;
    }
  assert(ch == '[');

  for(b = &buffer[0]; (ch = getc(curFP)) != EOF;)
  {
    if(ch == ']')
    {
break;
    }
    else
    {
```

```
      *b++ = ch;
    }
  }

  b = strchr(buffer, ',');
  assert(b != NULL);
  *b = '\0';
  v[0] = atof(buffer);
  nb = strchr(b + 1, ',');
  assert(nb != NULL);
  *nb = '\0';

  v[1] = atof(b + 1);
  v[2] = atof(nb + 1);
}


void
readVectorInt(FILE *curFP, int *v)
{
  int ch;
  char buffer[256];
  char *b, *nb;

  ch = getc(curFP);
  if(ch == EOF)
    {
      return;
    }
  assert(ch == '[');

  for(b = &buffer[0]; (ch = getc(curFP)) != EOF;)
  {
    if(ch == ']')
    {
break;
    }
    else
    {
      *b++ = ch;
    }
  }

  b = strchr(buffer, ',');
  assert(b != NULL);
  *b = '\0';
  v[0] = atoi(buffer);
  nb = strchr(b + 1, ',');
  assert(nb != NULL);
  *nb = '\0';

  v[1] = atoi(b + 1);
  v[2] = atoi(nb + 1);
}


void
readUntilVector(FILE *curFP, char *b)
{
  int ch;

  for(; (ch = getc(curFP)) != EOF;)
  {
    if(ch == '[')
    {
ungetc(ch, curFP);
break;
    }
    else
    {
      *b++ = ch;
    }
  }
}

void
readThruStar(FILE *curFP, char *b)
{
  int ch;

  for(; (ch = getc(curFP)) != EOF;)
  {
    if(ch == '*')
    {
break;
    }
    else
    {
      *b++ = ch;
    }
  }
}



void
SetGLToCamera(CameraGLPtr outCam)
{
    glMatrixMode(GL_PROJECTION);
    glLoadMatrixd(outCam->projMat);
    glMatrixMode(GL_MODELVIEW);
    glLoadMatrixd(outCam->modelMat);
    glViewport(outCam->viewMat[0], outCam->viewMat[1],
       outCam->viewMat[2], outCam->viewMat[3]);
}
```

```
void
WarpGL_DepthPixel(DepthPixelPtr ldpPtr, CameraGLPtr curCamPtr, int x, int y,
       double *wx, double *wy, double *depth, int *splatSize)
{
  double obj[3];
  int z1;
  char sqrtSize;

  /*
  ** perform warp by doing an unProject from x,y,depth in ldi view.
  ** then projecting corresponding winx,winy,winz coord to outCam view.
  */

    z1 = 1;
    assert(gluUnProject(x, y, z1, curCamPtr->modelMat, curCamPtr->projMat, curCamPtr->viewMat,
 &obj[0], &obj[1], &obj[2]) == GL_TRUE);
    sqrtSize = obj[2] * lookupTable[ldpPtr->splatIndex];

    /*
fprintf(stdout, "Pixel [%d %d %d] unprojects to [%.2lf %.2lf %.2lf] [%d %d %d]\n",
x, y, z1, obj[0], obj[1], obj[2], (int)obj[0], (int)obj[1], (int)obj[2]);
    */

    *wx = obj[0];
    *wy = obj[1];
    *depth = obj[2];
    *splatSize = sqrtSize;
}




void
ComputeSplatLookup(LayeredDepthImagePtr curPtr, CameraGLPtr outCam)
{

}


int
IsInImage(int *viewMat, int lix, int liy)
{
  if((lix >= viewMat[0]) && (lix < viewMat[2]) &&
     (liy >= viewMat[1]) && (liy < viewMat[3]))
  {
    return(1);
  }
  return(0);
}


void
BuildViewMatrix(double *vm, int *ivm, double near, double far)
{
  double ox, oy;


  ox = (double)ivm[0] + (double)(ivm[2] / 2);
  oy = (double)ivm[1] + (double)(ivm[3] / 2);

  vm[0] = ((double)ivm[2] / (double)2.0);
  vm[1] = (double)0.0;
  vm[2] = (double)0.0;
  vm[3] = (double)0.0;

  vm[4] = (double)0.0;
  vm[5] = ((double)ivm[3] / (double)2.0);
  vm[6] = (double)0.0;
  vm[7] = (double)0.0;

  vm[8] = (double)0.0;
  vm[9] = (double)0.0;
  vm[10] = (far - near) / (double)2.0;
  vm[11] = (double)0.0;

  vm[12] = ox;
  vm[13] = oy;
  vm[14] = (near + far) / (double)2.0;
  vm[15] = (double)1.0;
}

void
ComputeCompositeMatrix(double *projMat, double *modelMat,
       int *viewMat, double *rs, double near, double far)
{
  double temp[16];
  double viewM[16];

  BuildViewMatrix(viewM, viewMat, near, far);
  multiplyMatrices(temp, viewM, projMat);
  multiplyMatrices(rs, temp, modelMat);
}

int
BuildDepthImage(LayeredDepthImagePtr curLdiPtr,
FILE *curFP)
{
  int x, y, yoffset, rgba;
  struct DepthPixel curP;
  double depth;
  double avgDepth = 0, minDepth = 0, maxDepth = 0;
  int    totalDepthPixels = 0;
  double eye[3], lookat[3], up[3], du[3], dv[3], fov, frustum[6];
  int    view[4];
```

```
int    cr, cg, cb;
char   curLine[1024], *ch, *nch, buffer[512];
int    ppmYoffset;
double near, far;
double result[16];

int    fsRet;
int    w,h;
int    cols[3];

int    isLdi;

double    lix, liy, lid;

double wx, wy;
int splatSize;

CameraGLPtr curCam;
int warpCollisions = 0;
int maxDepthNum = 0;
double ldiFor[16];
double otherBak[16];
double otherFor[16];
double compWarp[16];
double dist;

double curPoint[4];
double warpPt[4];



curCam = CameraGL_Create();

/*
** parse given file (curFP)
**   determine camera info and dimensions of image.
*/
readVector(curFP, eye);
readVector(curFP, lookat);
readVector(curFP, up);
readVector(curFP, du);
readVector(curFP, dv);
readUntilVector(curFP, buffer);
ch = strchr(buffer, '*');
assert(ch != NULL);
*ch = '\0';
fov = atof(buffer);
nch = strchr(ch + 1, ',');
assert(nch != NULL);
*nch = '\0';
w = atoi(ch + 1);
h = atoi(nch + 1);




/* compute distance:  we know size (yres) and fov */
dist = h / ( tan((fov * M_PI)/360.0));

near = (double)dist;
far = (double)dist + 100;


/* configure camera */
CameraGL_SetLookAt(curCam, eye, lookat, up);
CameraGL_SetPerspective(curCam, fov, w / h, near, far);
CameraGL_SetDuv(curCam, du, dv);

view[0] = 0;
view[1] = 0;
view[2] = w;
view[3] = h;
CameraGL_SetViewMat(curCam, view);

/*
fsRet = fscanf(curFP, "[%.lf, %.lf, %.lf][%.lf %.lf %.lf][%.lf %.lf %.lf][%.lf %.lf %.lf][%.lf %.lf %.lf]%.lf*%d,%d", &(eye[0]), &(eye[1]), &(eye[2]),
&(lookat[0]), &(lookat[1]), &(lookat[2]),
&(up[0]), &(up[1]), &(up[2]), &(du[0]), &(du[1]), &(du[1]),
&(dv[0]), &(dv[1]), &(dv[2]), &fov, &w, &h);
assert(fsRet == 18);
*/


if((curLdiPtr->xres == 0) && (curLdiPtr->yres == 0))
{
  isLdi = 1;
curLdiPtr->xres = w;
curLdiPtr->yres = h;
curLdiPtr->pixels = malloc(sizeof(struct LayeredDepthPixel) * (curLdiPtr->xres * curLdiPtr->yres));
assert(curLdiPtr->pixels != NULL);
curLdiPtr->camera = curCam;

ComputeCompositeMatrix(curLdiPtr->camera->projMat, curLdiPtr->camera->modelMat,
 curLdiPtr->camera->viewMat, ldiFor, near, far);

}
else
{
  isLdi = 0;


  /* make sure images have same dimensions */
  assert((w == curLdiPtr->xres) && (h == curLdiPtr->yres));

  ComputeCompositeMatrix(curCam->projMat, curCam->modelMat, curCam->viewMat,
 otherFor, near, far);
```

```
    invertMatrix(otherBak, otherFor);
    multiplyMatrices(compWarp, ldiFor, otherBak);
  }

    /* precompute splat lookup table */
    ComputeSplatLookup(curLdiPtr, curCam);


  for(y = 0; y < curLdiPtr->yres; y++)
  {
    yoffset = y * curLdiPtr->xres;
    for(x = 0; x < curLdiPtr->xres; x++)
    {
      if(isLdi)
      {
curLdiPtr->pixels[ yoffset + x ].numLayers = 0;
curLdiPtr->pixels[ yoffset + x ].layers = NULL;
      }

/*
fscanf(curFP, "[%d,%d,%d]%.lf*",
      &cr, &cg, &cb, &depth);
*/
readVectorInt(curFP, cols);
readThruStar(curFP, buffer);
depth = atof(buffer);

rgba = 0;
if(depth == 0)
{
  SET_ALPHA(rgba,0);
}
else
{
  SET_ALPHA(rgba,1);
}

SET_RED(rgba, cols[0]);
SET_GREEN(rgba, cols[1]);
SET_BLUE(rgba, cols[2]);
DepthPixel_SetRGBA(&curP, rgba);
        DepthPixel_SetZ(&curP, (int)depth);
DepthPixel_SetSplatIndex(&curP, (int)depth);

avgDepth += depth;
totalDepthPixels++;

if(isLdi)
{
  if(LayeredDepthPixel_AddDepthPixelLayer(&(curLdiPtr->pixels[ yoffset + x ]), &curP))
  {
fprintf(stderr, "Error adding depth pixel\n");
return(1);
  }
}
else
{
  /* we have to warp to LDI space */


  curPoint[0] = (double)x;
  curPoint[1] = (double)y;
  curPoint[2] = (double)-depth;
  curPoint[3] = 1;

  transformPoint(warpPt, curPoint, compWarp);
  /* divide by w */
  warpPt[0] /= warpPt[3];
  warpPt[1] /= warpPt[3];
  warpPt[2] /= warpPt[3];

  /* need to adjust splatIndex */
  DepthPixel_SetSplatIndex(&curP, (int)warpPt[3]);
  warpPt[3] = (double)1.0;

  if(IsInImage(curLdiPtr->camera->viewMat, (int)warpPt[0],
       (int)warpPt[1]))
  {

    warpCollisions++;
    /*
fprintf(stdout, "warping collision at [%d,%d]\n", (int)lix, (int)liy);
    */

    if(LayeredDepthPixel_AddDepthPixelLayer(&(curLdiPtr->pixels[ ((int)warpPt[1] * (int)curLdiPtr->xres)  + (int)warpPt[0] ]), &curP))
      {
fprintf(stderr, "Error adding depth pixel\n");
return(1);
      }
    if(curLdiPtr->pixels[((int)warpPt[1] * (int)curLdiPtr->xres) + (int)warpPt[0]].numLayers > maxDepthNum)
      {
maxDepthNum = curLdiPtr->pixels[((int)warpPt[1] * (int)curLdiPtr->xres) + (int)warpPt[0]].numLayers;
      }
  }
}
    }
  }

  if(totalDepthPixels > 0)
  {
    avgDepth /= totalDepthPixels;
  }


  fprintf(stdout, "%d warping collisions (%d max depth)\n", warpCollisions,
  maxDepthNum);
```

```
  /* indicate success */
  return(0);
}



void
renderFlatLdi()
{
  int x, y, yoffset, l;
  DepthPixelPtr curDP;
  LayeredDepthPixelPtr curLDP;
  double r, g, b, a;
  unsigned int   ir, ig, ib;


  glBegin(GL_POINTS);
  for(y = 0; y < theLdiPtr->yres; y++)
  {
    yoffset = y * theLdiPtr->xres;
    for(x = 0; x < theLdiPtr->xres; x++)
    {
      curLDP = &(theLdiPtr->pixels[yoffset + x]);
      if(curLDP->numLayers > 0)
      {
for(l = curLDP->numLayers - 1; l >= 0; l--)
{
  curDP = &(curLDP->layers[curLDP->numLayers - 1]);

  r = (double)(GET_RED(curDP->colorRGBA)) / (double)255.0;
  g = (double)(GET_GREEN(curDP->colorRGBA)) / (double)255.0;
  b = (double)(GET_BLUE(curDP->colorRGBA)) / (double)255.0;
  a = (double)(GET_ALPHA(curDP->colorRGBA)); /* range 0-1 */
/*
fprintf(stdout, "[%d %d]rgb=[%.2lf %.2lf %.2lf\n", x, y, r, g, b);
*/
  glColor4d(r, g, b, a);
        glVertex3i(x, y, (curDP->z));
  /* glVertex3i(x, y, 0);*/
}
      }
    }
  }
  glEnd();
}

void
SortLDI()
{
  int x, y, yoffset, l;
  struct DepthPixel swapDP, cmpDP;
  LayeredDepthPixelPtr curLDP;
  double r, g, b, a;
  unsigned int   ir, ig, ib;

  for(y = 0; y < theLdiPtr->yres; y++)
  {
    yoffset = y * theLdiPtr->xres;
    for(x = 0; x < theLdiPtr->xres; x++)
    {
      curLDP = &(theLdiPtr->pixels[yoffset + x]);

      /* sort layers of given LDP */
      /* do a really bad job of it */


    }
  }
}


void
Usage(char *name)
{
 fprintf(stdout, "%s {jrtfile...}\n", name);

}


void init(void)
{
   glClearColor (0.0, 0.0, 0.0, 0.0);
   glShadeModel (GL_FLAT);

   SetGLToCamera(theLdiPtr->camera);
}

void reshape(int w, int h)
{
  init();
}


void display(void)
{
  int length = 5;

   glClearColor(1,1,1,1);
   glClear(GL_COLOR_BUFFER_BIT);
   glClearDepth(1.0);
   glClear(GL_DEPTH_BUFFER_BIT);

   renderFlatLdi();
```

```
   glutSwapBuffers();

}
int
main(int argc, char *argv[])
{
  FILE *curFP;
  char newName[256];
  char *ch;
  int  i;

  if(argc < 2)
    {
      Usage(argv[0]);
      exit(1);
    }


   glutInit(&argc, argv);
   glutInitDisplayMode (GLUT_DOUBLE | GLUT_RGB);
   glutInitWindowSize (500, 500);
   glutInitWindowPosition (300, 300);
   glutCreateWindow (argv[0]);
   glutDisplayFunc(display);
   glutReshapeFunc(reshape);


  strcpy(newName, argv[1]);
  ch = strrchr(newName, '.');
  if(ch != NULL)
  {
    *ch = '\0';
  }
  strcat(newName, ".ldi");

  theLdiPtr = LayeredDepthImage_Create();

  fprintf(stdout, "Building Layered Depth Image\n");
  for(i = 1; i < argc; i++)
  {
    fprintf(stdout, "Parsing file %s\n", argv[i]);
    curFP = fopen(argv[i], "r");
    assert(curFP != NULL);

    if(BuildDepthImage(theLdiPtr, curFP))
    {
  fprintf(stderr, "Error building depth image\n");
  exit(1);
    }
    fclose(curFP);
  }


  fprintf(stdout, "Finished reading LDI's\n");

  fprintf(stdout, "Initial write of LDI\n");
  if(LayeredDepthImage_WriteToFile(theLdiPtr, newName))
  {
fprintf(stderr, "Error writing ldi to file %s\n", newName);
exit(1);
  }

  init();

  glutPostRedisplay();
  glutMainLoop();

  exit(0);
}
```

12 vs LISTING (view ray-tracer scene file)

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <assert.h>
#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glut.h>
#include <math.h>

#include "proc_ppm.h"
#include "matrix.h"
#include "mlist.h"
#include "camera.h"

#define SPHERE_TYPE  1
#define TRI_TYPE     2


#define MOUSE_LEFT    0
#define MOUSE_MIDDLE  1
#define MOUSE_RIGHT   2


#define D_WIDTH  350
#define D_HEIGHT 350
#define FAR_DIST 100
```

```
typedef struct Obj *ObjPtr;
struct Obj
{
  char type;
  double p0[3];
  double p1[3];
  double p2[3];
  double col[3];
  double rad;
};

typedef struct Scene *ScenePtr;
struct Scene
{
  CameraGLPtr camera;
  double      bg[3];
  listPtr     objListPtr;
};


ScenePtr tsPtr;
GLUquadricObj *cs;

int    curAxis = 2;
int    curType = 0;
int    lw, lh;
int    lmx, lmy;
char   mbutton_state[3];
double rot[3];




ScenePtr
Scene_Create(void)
{
  ScenePtr curPtr;


  curPtr = malloc(sizeof(struct Scene));
  assert(curPtr != NULL);
  curPtr->camera = CameraGL_Create();
  curPtr->objListPtr = List_Initialize();
  assert(curPtr->objListPtr != NULL);
  curPtr->bg[0] = 0.0;
  curPtr->bg[1] = 0.0;
  curPtr->bg[2] = 0.0;

  return(curPtr);
}

ObjPtr
Sphere_Create(double *p0, double rad, double *c)
{
  ObjPtr curPtr;

  curPtr = malloc(sizeof(struct Obj));
  assert(curPtr != NULL);

  curPtr->p0[0] = p0[0];
  curPtr->p0[1] = p0[1];
  curPtr->p0[2] = p0[2];
  curPtr->col[0] = c[0];
  curPtr->col[1] = c[1];
  curPtr->col[2] = c[2];
  curPtr->rad = rad;
  curPtr->type = SPHERE_TYPE;

  return(curPtr);
}

ObjPtr
Tri_Create(double *p0, double *p1, double *p2, double *c)
{
  ObjPtr curPtr;

  curPtr = malloc(sizeof(struct Obj));
  assert(curPtr != NULL);

  curPtr->p0[0] = p0[0];
  curPtr->p0[1] = p0[1];
  curPtr->p0[2] = p0[2];
  curPtr->p1[0] = p1[0];
  curPtr->p1[1] = p1[1];
  curPtr->p1[2] = p1[2];
  curPtr->p2[0] = p2[0];
  curPtr->p2[1] = p2[1];
  curPtr->p2[2] = p2[2];
  curPtr->col[0] = c[0];
  curPtr->col[1] = c[1];
  curPtr->col[2] = c[2];
  curPtr->type = TRI_TYPE;

  return(curPtr);
}


void
drawAxes(double length)
{
        glColor3d (1.0, 0.0, 0.0);
        glBegin(GL_LINES);
        glVertex3d(0.0,0.0,0.0);                /* x axis */
        glVertex3d(length,0.0,0.0);
        glColor3d (0.0, 1.0, 0.0);
        glVertex3d(0.0,0.0,0.0);                /* y axis */
```

```
        glVertex3d(0.0,length,0.0);
        glColor3d (0.0, 0.0, 1.0);
        glVertex3d(0.0,0.0,0.0);                /* z axis */
        glVertex3d(0.0,0.0,length);
        glEnd();
}


ScenePtr
ReadScene(char *fname)
{
  ScenePtr theScenePtr;
  FILE *fp;
  char line[256];
  char *tokstr;
  int i;
  char *ch;
  double fov;
  double eye[3], lookat[3], up[3], cc[3];
  double p0[3], p1[3], p2[3], rad;
  double center[3], aspect, near, far;
  double dist[3];
  int vm[4];
  ObjPtr curObjPtr;


  vm[0] = -(D_WIDTH / 2);
  vm[1] = 0;
  vm[2] = D_WIDTH;
  vm[3] = D_HEIGHT;



  fp = fopen(fname, "r");
  if(fp == NULL)
  {
    perror("Error opening file");
    assert(0);
  }

  theScenePtr = Scene_Create();

  for(; fgets(line, 256, fp);)
  {
    if((ch = strstr(line, "eye")) != NULL)
    {
      ch += 4;
      for(i = 0, tokstr = strtok(ch, " "); tokstr; i++, tokstr = strtok(NULL, " "))
      {
switch(i)
  {
  case 0:
    eye[0] = atof(tokstr);
    break;
  case 1:
    eye[1] = atof(tokstr);
    break;
  case 2:
    eye[2] = atof(tokstr);
    break;
  case 3:
    break;
  default:
    break;
  }
      }
    }
    else if((ch = strstr(line, "lookat")) != NULL)
    {
      ch += 7;

      for(i = 0, tokstr = strtok(ch, " "); tokstr; i++, tokstr = strtok(NULL, " "))
      {
switch(i)
  {
  case 0:
    lookat[0] = atof(tokstr);
    break;
  case 1:
    lookat[1] = atof(tokstr);
    break;
  case 2:
    lookat[2] = atof(tokstr);
    break;
  case 3:
    break;
  default:
    break;
  }
      }

    }
    else if((ch = strstr(line, "up")) != NULL)
    {
      ch += 3;
      for(i = 0, tokstr = strtok(ch, " "); tokstr; i++, tokstr = strtok(NULL, " "))
      {
switch(i)
  {
  case 0:
    up[0] = atof(tokstr);
    break;
  case 1:
    up[1] = atof(tokstr);
    break;
  case 2:
```

```
    up[2] = atof(tokstr);
    break;
  case 3:
    break;
  default:
    break;
  }
      }

    }
    else if((ch = strstr(line, "fov")) != NULL)
    {
      ch += 4;
      fov = atof(ch);
      fprintf(stdout, "** FOV = %.2lf\n", fov);
    }
    else if((ch = strstr(line, "surface")) != NULL)
    {
      ch += 8;
      for(i = 0, tokstr = strtok(ch, " "); tokstr; i++, tokstr = strtok(NULL, " "))
      {
switch(i)
  {
  case 0:
    cc[0] = atof(tokstr);
    break;
  case 1:
    cc[1] = atof(tokstr);
    break;
  case 2:
    cc[2] = atof(tokstr);
    break;
  case 3:
    break;
  default:
    break;
  }
      }

    }
    else if((ch = strstr(line, "background")) != NULL)
    {
      ch += 11;
      for(i = 0, tokstr = strtok(ch, " "); tokstr; i++, tokstr = strtok(NULL, " "))
      {
switch(i)
  {
  case 0:
    theScenePtr->bg[0] = atof(tokstr);
    break;
  case 1:
    theScenePtr->bg[1] = atof(tokstr);
    break;
  case 2:
    theScenePtr->bg[2] = atof(tokstr);
    break;
  case 3:
    break;
  default:
    break;
  }
      }
    }
    else if((ch = strstr(line, "sphere")) != NULL)
    {
      ch += 7;
      for(i = 0, tokstr = strtok(ch, " "); tokstr; i++, tokstr = strtok(NULL, " "))
      {
switch(i)
  {
  case 0:
    p0[0] = atof(tokstr);
    break;
  case 1:
    p0[1] = atof(tokstr);
    break;
  case 2:
    p0[2] = atof(tokstr);
    break;
  case 3:
    rad = atof(tokstr);
    break;
  default:
    break;
  }
      }
      curObjPtr = Sphere_Create(p0, rad, cc);
      List_Append(theScenePtr->objListPtr, (void *)curObjPtr);
    }
    else if((ch = strstr(line, "triangle")) != NULL)
    {
      ch += 9;
      for(i = 0, tokstr = strtok(ch, " "); tokstr; i++, tokstr = strtok(NULL, " "))
      {
switch(i)
  {
  case 0:
    p0[0] = atof(tokstr);
    break;
  case 1:
    p0[1] = atof(tokstr);
    break;
  case 2:
    p0[2] = atof(tokstr);
    break;
  case 3:
```

```
    p1[0] = atof(tokstr);
    break;
  case 4:
    p1[1] = atof(tokstr);
    break;
  case 5:
    p1[2] = atof(tokstr);
    break;
  case 6:
    p2[0] = atof(tokstr);
    break;
  case 7:
    p2[1] = atof(tokstr);
    break;
  case 8:
    p2[2] = atof(tokstr);
    break;
  default:
    break;
  }
      }
      curObjPtr = Tri_Create(p0, p1, p2, cc);
      List_Append(theScenePtr->objListPtr, (void *)curObjPtr);
    }
  }

  fclose(fp);

  aspect = (double)D_WIDTH / (double)D_HEIGHT;
  dist[0] = eye[0] - lookat[0];
  dist[1] = eye[1] - lookat[1];
  dist[2] = eye[2] - lookat[2];

  near = sqrt((dist[0] * dist[0]) + (dist[1]*dist[1]) + (dist[2]*dist[2]));
  far = FAR_DIST;


  center[0] = lookat[0] + 0.9;
  center[1] = lookat[1];
  center[2] = lookat[2];


  /* now init scene w/ camera info */
  CameraGL_SetPerspective(theScenePtr->camera, fov, aspect, near, far);
  CameraGL_SetLookAt2(theScenePtr->camera, eye, center, up);
  CameraGL_SetViewMat(theScenePtr->camera, vm);

  return(theScenePtr);
}



void
Scene_Print(ScenePtr curPtr, FILE *fp)
{
  ObjPtr curObjPtr;
  listIterPtr curLIPtr;



  curLIPtr = ListIter_Initialize();
  assert(curLIPtr != NULL);

  fprintf(fp, "Scene:\n");
  fprintf(fp, "Eye[%.2lf %.2lf %.2lf] Lookat[%.2lf %.2lf %.2lf]\n",
  curPtr->camera->eye[0], curPtr->camera->eye[1], curPtr->camera->eye[2],
  curPtr->camera->center[0], curPtr->camera->center[1], curPtr->camera->center[2]);

  CameraGL_PrintPrettyToStream(curPtr->camera, stdout);

  for(curObjPtr = (ObjPtr)ListIter_Start(curLIPtr, curPtr->objListPtr);
      curObjPtr;
      curObjPtr = (ObjPtr)ListIter_Next(curLIPtr))
  {
    fprintf(fp, "[type=%d][%.2f %.2f %.2f]\n",
    curObjPtr->type, curObjPtr->p0[0], curObjPtr->p0[1], curObjPtr->p0[2]);
  }

  ListIter_Destroy(curLIPtr);
}




void
SetGLToCamera(CameraGLPtr outCam)
{
    glMatrixMode(GL_PROJECTION);
    glLoadMatrixd(outCam->projMat);
    glMatrixMode(GL_MODELVIEW);
    glLoadMatrixd(outCam->modelMat);
    glViewport(outCam->viewMat[0], outCam->viewMat[1],
       outCam->viewMat[2], outCam->viewMat[3]);
}




int
IsInImage(int *viewMat, int lix, int liy)
{
  if((lix >= viewMat[0]) && (lix < viewMat[2]) &&
     (liy >= viewMat[1]) && (liy < viewMat[3]))
  {
    return(1);
  }
```

```
  return(0);
}


void
BuildViewMatrix(double *vm, int *ivm, double near, double far)
{
  double ox, oy;


  ox = (double)ivm[0] + (double)(ivm[2] / 2);
  oy = (double)ivm[1] + (double)(ivm[3] / 2);

  vm[0] = ((double)ivm[2] / (double)2.0);
  vm[1] = (double)0.0;
  vm[2] = (double)0.0;
  vm[3] = (double)0.0;

  vm[4] = (double)0.0;
  vm[5] = ((double)ivm[3] / (double)2.0);
  vm[6] = (double)0.0;
  vm[7] = (double)0.0;

  vm[8] = (double)0.0;
  vm[9] = (double)0.0;
  vm[10] = (far - near) / (double)2.0;
  vm[11] = (double)0.0;

  vm[12] = ox;
  vm[13] = oy;
  vm[14] = (near + far) / (double)2.0;
  vm[15] = (double)1.0;
}

void
ComputeCompositeMatrix(double *projMat, double *modelMat,
       int *viewMat, double *rs, double near, double far)
{
  double temp[16];
  double viewM[16];

  BuildViewMatrix(viewM, viewMat, near, far);
  multiplyMatrices(temp, viewM, projMat);
  multiplyMatrices(rs, temp, modelMat);
}


void
Usage(char *name)
{
 fprintf(stdout, "%s {scenefile}\n", name);

}


void init(void)
{
   glClearColor (0.0, 0.0, 0.0, 0.0);
   glShadeModel (GL_FLAT);

   SetGLToCamera(tsPtr->camera);
}

void reshape(int w, int h)
{
  int vm[4];

  lw = w;
  lh = h;
  vm[0] = 0;
  vm[1] = 0;
  vm[2] = w;
  vm[3] = h;
  CameraGL_SetViewMat(tsPtr->camera, vm);
}


void
RenderObj(ObjPtr curObjPtr)
{
  switch(curObjPtr->type)
  {
  case SPHERE_TYPE:
    glPushMatrix();
    glTranslated(curObjPtr->p0[0], curObjPtr->p0[1], curObjPtr->p0[2]);
    glColor3d(curObjPtr->col[0], curObjPtr->col[1], curObjPtr->col[2]);
    gluSphere(cs, curObjPtr->rad, 16, 19);
    glPopMatrix();
    break;
  case TRI_TYPE:
    glColor3d(curObjPtr->col[0], curObjPtr->col[1], curObjPtr->col[2]);
    glBegin(GL_TRIANGLES);
    glVertex3d(curObjPtr->p0[0], curObjPtr->p0[1], curObjPtr->p0[2]);
    glVertex3d(curObjPtr->p1[0], curObjPtr->p1[1], curObjPtr->p1[2]);
    glVertex3d(curObjPtr->p2[0], curObjPtr->p2[1], curObjPtr->p2[2]);
    glEnd();
    break;
  default:
    fprintf(stdout, "Unknown render type\n");
    break;
  }
}

void
renderScene()
{
```

```
  ObjPtr curObjPtr;
  listIterPtr curLIPtr;


  curLIPtr = ListIter_Initialize();

  for(curObjPtr = (ObjPtr)ListIter_Start(curLIPtr, tsPtr->objListPtr);
      curObjPtr;
      curObjPtr = (ObjPtr)ListIter_Next(curLIPtr))
  {
    RenderObj(curObjPtr);
  }

  ListIter_Destroy(curLIPtr);
}


void display(void)
{
  int length = 5;

   glClearColor(tsPtr->bg[0],tsPtr->bg[1],tsPtr->bg[2],0.0);
   glClear(GL_COLOR_BUFFER_BIT);
   glClearDepth(1.0);
   glClear(GL_DEPTH_BUFFER_BIT);

   glPushMatrix();

   glRotated(rot[0], 1.0, 0.0, 0.0); /* x rotation */
   glRotated(rot[1], 0.0, 1.0, 0.0); /* y rotation */
   glRotated(rot[2], 0.0, 0.0, 1.0); /* z rotation */

   renderScene();
   drawAxes(1);
   glPopMatrix();

   glutSwapBuffers();
}



void keyboard (unsigned char key, int x, int y)
{
  double inp[3];
  double frust[6];
  char buffer[256];
  double fov, near, far;

/* Exit on escape */
   switch (key) {
      case 27:
        exit(0);
        break;
   case 'x':
        curAxis = 0;
fprintf(stdout, "curaxis = x\n");
        break;
   case 'y':
        curAxis = 1;
fprintf(stdout, "curaxis = y\n");
        break;
   case 'z':
        curAxis = 2;
fprintf(stdout, "curaxis = z\n");
        break;
   case 'v':
     fprintf(stdout, "Enter view[0]: ");
     tsPtr->camera->viewMat[0] = atoi(gets(buffer));
     fprintf(stdout, "Enter view[1]: ");
     tsPtr->camera->viewMat[1] = atoi(gets(buffer));
     SetGLToCamera(tsPtr->camera);
     glutPostRedisplay();
     break;
   case '0':
     glMatrixMode(GL_PROJECTION);
     glLoadIdentity();
     gluPerspective(30, 1.0, 0.001, 4000);
     glMatrixMode(GL_MODELVIEW);
     glLoadIdentity();
     gluLookAt(0.0, 5.0, 10.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);
     fprintf(stdout, "reset view\n");
     reshape(lw, lh);
     glutPostRedisplay();
     break;
   case '1':
     SetGLToCamera(tsPtr->camera);
     fprintf(stdout, "reset to camera\n");
     reshape(lw, lh);
     glutPostRedisplay();
     break;
   case 'p':
     fprintf(stdout, "Enter fov (%.2lf): ", tsPtr->camera->fov);
     fov = atof(gets(buffer));
     fprintf(stdout, "Enter near (%.2lf): ", tsPtr->camera->near);
     near = atof(gets(buffer));
     fprintf(stdout, "Enter far (%.2lf): ", tsPtr->camera->far);
     far = atof(gets(buffer));
     CameraGL_SetPerspective(tsPtr->camera, fov, tsPtr->camera->aspect, near, far);
     SetGLToCamera(tsPtr->camera);
     reshape(lw, lh);
     glutPostRedisplay();
     break;
   case 'e':
     fprintf(stdout, "Enter eye X (%.2lf): ", tsPtr->camera->eye[0]);
     tsPtr->camera->eye[0] = atof(gets(buffer));
     fprintf(stdout, "Enter eye Y (%.2lf): ", tsPtr->camera->eye[1]);
```

```
     tsPtr->camera->eye[1] = atof(gets(buffer));
     fprintf(stdout, "Enter eye Z (%.2lf): ", tsPtr->camera->eye[2]);
     tsPtr->camera->eye[2] = atof(gets(buffer));
     CameraGL_SetEye(tsPtr->camera, tsPtr->camera->eye);
     SetGLToCamera(tsPtr->camera);
     reshape(lw, lh);
     glutPostRedisplay();
     break;
   default:
     break;
   }
}


/*
** DESCRIPTION:
**   Scales world uniformly
*/
void
mouse_ztrans(int x, int y)
{
  double prop[2], mv;

  prop[0] = (lmx - x);
  prop[1] = (lmy - y);  /* origin is upper left */

  mv = prop[0] + prop[1] / 2;

  tsPtr->camera->eye[2] += mv;
  CameraGL_SetEye(tsPtr->camera, tsPtr->camera->eye);
  fprintf(stdout, "camera[eye] = [%.2lf %.2lf %.2lf]\n",
  tsPtr->camera->eye[0],
  tsPtr->camera->eye[1],
  tsPtr->camera->eye[2]);
}


/*
** DESCRIPTION:
**   Translates world
*/
void
mouse_trans(int x, int y)
{
  double sc;
  double prop[2];


  prop[0] = (lmx - x);
  prop[1] = (lmy - y);

  tsPtr->camera->eye[0] += (prop[0] / 2);
  tsPtr->camera->eye[1] += (prop[1] / 2);
  CameraGL_SetEye(tsPtr->camera, tsPtr->camera->eye);
}

/*
** DESCRIPTION:
**   Rotates world
*/
void
mouse_rot(int x, int y)
{

  if(curAxis == 0)
    {
      rot[0] += (lmx - x);
    }
  else if(curAxis == 1)
    {
      rot[1] += (lmx - x);
    }
  else
    {
      rot[2] += (lmx - x);
    }

fprintf(stdout, "Rotate about axis[%d] [%.2lf %.2lf %.2lf]\n", curAxis,
rot[0], rot[1], rot[2]);

}

/************************************************************************/
/* call back for clicking any mouse button */
/* remembers the state of button and which window was in */
 void mouseclick(int button, int state, int x, int y)
{
  lmx = x;
  lmy = y;
  mbutton_state[button] = state;
}

/************************************************************************/
/* this routine is the callback for moving the mouse while a button
   is down.  it calls a routine to handle result of motion. */
void mousemove(int x, int y)
{

  if (mbutton_state[MOUSE_RIGHT] == 0)
  {
    mouse_ztrans(x,y);
  }
  else if (mbutton_state[MOUSE_MIDDLE] == 0)
  {
    mouse_trans(x,y);
  }
```

```
  else if (mbutton_state[MOUSE_LEFT] == 0)
  {
    mouse_rot(x,y);
  }

  lmx = x;
  lmy = y;
  reshape(lw, lh);
  glutPostRedisplay();
}


int
main(int argc, char *argv[])
{
  FILE *curFP;
  char newName[256];
  char *ch;
  int  i;

  if(argc != 2)
    {
      Usage(argv[0]);
      exit(1);
    }

  rot[0] = 0.0;
  rot[1] = 0.0;
  rot[2] = 0.0;

   glutInit(&argc, argv);
   glutInitDisplayMode (GLUT_DOUBLE | GLUT_RGB);
   glutInitWindowSize (D_WIDTH, D_HEIGHT);
   glutInitWindowPosition (300, 300);
   glutCreateWindow (argv[0]);
   glutDisplayFunc(display);
   glutReshapeFunc(reshape);
   glutKeyboardFunc(keyboard);
   glutMouseFunc(mouseclick);
   glutMotionFunc(mousemove);

   cs = gluNewQuadric();

   tsPtr = ReadScene(argv[1]);
   Scene_Print(tsPtr, stdout);

  init();

  glutMainLoop();

  exit(0);
}
```

References

[1] R. Szeliski J. Shade, L.He. Layered depth images. *Siggraph '98*, pages 231–242, 1998.

[2] L. McMillan. A list-priority rendering algorithm for redisplaying projected surfaces. *Tech. Report 95-005, University of North Carolina*, 1995.

[3] C. Dyer S. Seitz. Toward image-based scene representation using view morphing. *Tech. Report 1298, University of Wisconsin*, 1996.

[4] C. Dyer S. Seitz. Photorealistic scene reconstruction by voxel coloring. *Proceedings of Computer Vision and Pattern Recognition Conference*, pages 1067–1073, 1997.