A Concurrent Ray Tracing Algorithm Using Distributed Symmetric Processes

by

Delano Junior McFarlane

Submitted to the Department of Electrical Engineering and Computer Science in partial fulfillment of the requirements for the degree of Masters of Engineering in Computer Science and Engineering at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 1996

© Delano Junior McFarlane, MCMXCVI. All rights reserved.

The author hereby grants to MIT permission to reproduce and distribute publicly paper and electronic copies of this thesis document in whole or in part, and to grant others the right to do so.

Author .................................................................

Department of Electrical Engineering and Computer Science

May 23, 1996

Certified by ..................................................

Seth Teller

Assistant Professor of Computer Science and Engineering Thesis Supervisor

Accepted by ........................................................

F.R. Morgenthaler

Chairman, Departmental Committee on Graduate Students

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

JUN 11 1996
A Concurrent Ray Tracing Algorithm Using Distributed Symmetric Processes

by

Delano Junior McFarlane

Submitted to the Department of Electrical Engineering and Computer Science on May 23, 1996, in partial fulfillment of the requirements for the degree of Masters of Engineering in Computer Science and Engineering

Abstract

Ray tracing is an algorithm that is used to create a two dimensional image from a definition of a three dimensional scene. The two dimensional image is produced by casting imaginary rays from a location in a scene, through a window in an arbitrary projection plane in the scene, and then recording an estimate of the energy that returns to the eye along each ray.

A standard implementation of a ray tracing algorithm uses recursive function calls to calculate the energy of each ray that is cast. Most of the calculations performed by such an algorithm are independent of one another. This computational independence leads to the possibility of concurrent implementations of the ray tracing algorithm. This document describes such a concurrent algorithm.

DIraytrace, the concurrent implementation described in this document, utilizes Distributed Symmetric Processes that delegate the work that is required to calculate the energy of each ray that is cast in a three dimensional scene. The implementation and performance of this algorithm will be compared and contrasted with Braytrace, a standard recursive implementation of the ray tracing algorithm, and DRRaytrace, a simple parallelization of the recursive algorithm.

Thesis Supervisor: Seth Teller
Title: Assistant Professor of Computer Science and Engineering
Contents

1 Introduction ......................................................... 13

2 The Ray Tracing Algorithms ...................................... 17
   2.1 Braytrace ................................................. 17
   2.2 DRaytrace .................................................. 18
   2.3 DRaytrace .................................................. 21
   2.4 Extra Features ............................................ 22
       2.4.1 Supersampling ........................................ 22
       2.4.2 Spectral Distribution Calculations ................. 25

3 Implementations of the Ray Tracing Algorithms .............. 27
   3.1 Braytrace ................................................. 27
       3.1.1 Braytrace: Data Objects and Functions ............... 27
       3.1.2 Braytrace: Calling Sequence ......................... 29
   3.2 DRaytrace and DRaytrace .................................. 29
       3.2.1 DRaytrace: Objects and Functions .................... 30
       3.2.2 DRaytrace: Calling Sequence ......................... 34
       3.2.3 DRaytrace: Objects and Functions .................... 36
       3.2.4 DRaytrace: Calling Sequence ......................... 38
       3.2.5 DRaytrace: Convergence of CTree Data ................. 39

4 User Interface ..................................................... 41
   4.1 Graphic User Interface .................................... 41
4.2 Commandline Interface ............................................. 42

5 Debugging and Testing ............................................. 45

6 Analysis of the Concurrent Algorithms ............................. 47
   6.1 Initial Analysis and Optimizations .......................... 47
   6.2 Runtime Profile Statistics .................................... 49

7 CTPlayer Demonstration Application ............................... 53
   7.1 CTPlayer Interface ............................................. 53
      7.1.1 Button Menu .............................................. 53
      7.1.2 Instruction Window ....................................... 55
      7.1.3 Tree Display .............................................. 55
   7.2 CTPlayer Design and Implementation .......................... 59

8 Conclusion .................................................................. 61

A Profile Statistics for TraceProcRecursive and TraceProcIncremental ............................................. 63
   A.1 Braytrace .......................................................... 64
   A.2 TraceProcRecursive ............................................. 66
   A.3 TraceProcIncremental .......................................... 68

B File Formats .................................................................. 71
   B.1 CTrec Specification File ......................................... 71
   B.2 CT Log File format ............................................... 72
   B.3 Eye definition file ............................................... 73

C DTrace: Makefile, main() and Initialization Functions ............... 75

D Basic Ray Tracer Functions & Data Objects ......................... 115

E Concurrent Ray Trace Functions and Data Objects ................. 157
F  CTPlayer Code  191

Bibliography  225
List of Figures

1-1 A sample scene illustrating a viewing location and direction, an eye ray, a projection window and a light source. .......................... 14

2-1 Pseudocode Braytrace’s controlling function. .......................... 18
2-2 Pseudocode for Braytrace’s Trace function. .......................... 19
2-3 Pseudocode for DRraytrace’s controlling function. ...................... 20
2-4 Pseudocode for DRraytrace’s TraceProcRecurse function. .............. 21
2-5 Pseudocode for DIraytrace’s controlling function. ...................... 23
2-6 Pseudocode for DIraytrace’s TraceProcIncremental function. .......... 24

3-1 Braytrace Flow Diagram ............................................. 30
3-2 Object Inheritance and Data Dependency diagram for DRraytrace and DIraytrace. Broken lines indicate a data dependency, solid lines indicate a derived class. .............................................. 31
3-3 DRraytrace Flow Diagram ............................................. 34
3-4 Simple Linked List Pseudocode ....................................... 35
3-5 Ilmutex Pseudocode .................................................. 36
3-6 DIraytrace Flow Diagram ............................................. 38

4-1 The Graphical User Interface of DTrace. .................................. 42

7-1 The Graphic User Interface for the CTPlayer demonstration application. 54
7-2 The Graphic User Interface for the CTPlayer demonstration application after skipping to an instruction. .............................. 56
7-3 The Graphic User Interface for the CTPlayer demonstration application after advancing to a new location in the CT file.  

7-4 The Graphic User Interface for the CTPlayer demonstration application after skipping to the end of the CT file.
List of Tables

4.1 Commandline options for DTrace ....................... 43
6.1 Runtime durations for the ray tracing algorithms. ............ 47
6.2 Profile Statistics ................................................. 50
Chapter 1

Introduction

Ray tracing is an algorithm that creates a two dimensional image from a geometric definition of a three dimensional scene. The two dimensional image is produced by casting imaginary rays from a location in the scene (the viewer's eye), through a window on an arbitrary viewing plane within the scene.

As a ray interacts with the objects in a scene, energy is conveyed back along that ray to the pixel associated with the subdivision that defined that ray\(^1\). Once all the calculations pertaining to the rays cast through a particular subdivision are complete, the value of the pixel associated with that subdivision is set to the composite energy of the rays cast through that subdivision (see Figure 1-1).

The images that a ray tracing algorithm is capable of producing are both realistic and detailed. Shadows, reflection and refraction are just some of the effects that a ray tracing algorithm can simulate. These effects are produced by performing extensive calculations on the interactions between objects and rays.

As a ray intersects an object, that object's description and orientation are used to create new rays which may pass through the intersected object (simulating refracted light) or bounce off that object (simulating reflected light). These new rays, as they intersect other objects in the scene, may themselves spawn new reflective and refractive rays. A series of rays that is spawned from a single *root* ray is referred to

---

\(^1\)The energy conveyed along a ray can also be thought of as the color of light that traverses that ray.
as a *ray tree*. The final value of a pixel in the output image is then calculated by converging the data contained in the one or more ray trees that are associated with that pixel's corresponding subdivision.

The traditional implementation of a ray tracing algorithm uses a recursive calling strategy to calculate and converge the data in a root ray's ray tree. The recursive functions employed by such an implementation are controlled by a function that iterates over each pixel in the output image. During each iteration of the controlling function, one or more root rays are created and dispatched to the recursive ray tracing function. This implementation of the ray tracing algorithm will be referred to as Braytrace\(^2\).

One fact of importance regarding the execution of Braytrace is that most of the calculations performed to compute the energy along each ray are independent of all other calculations. This computational independence leads to a concurrent implemen-

\(^2\)For a more detailed discussion on the basic ray tracing algorithm refer to [5].
tation of the ray tracing algorithm which uses asynchronous symmetric processes to calculate and converge the contribution of each individual ray that is created during the rendering of a scene. This implementation of the ray tracing algorithm will be referred to as DIraytrace.

The computational independence described above also leads to a second, simpler concurrent implementation of the ray tracing algorithm. This second implementation uses the recursive strategy of Braytrace as the basis of its concurrent processes. The concurrent processes used by this implementation work in parallel, each recursively calculating the final value of a pixel in the output image. This concurrent implementation of the ray tracing algorithm will be referred to as DRraytrace.

The remainder of this document describes the design and implementation of the three ray tracing algorithms described above. The emphasis of this document will be directed towards comparing and contrasting the DIraytrace algorithm with the simpler, more straightforward DRraytrace algorithm, and the original Braytrace algorithm.

The chapters in this document are organized as follows. Chapter 2 describes each algorithm in more detail, giving pseudocode examples where necessary to illustrate how the algorithms work. Chapter 3 gives a detailed description of the design and implementation of the three algorithms. This chapter will include detailed descriptions of all important functions and data objects. Chapter 4 describes DTrace, the user interface to the ray tracing algorithms. Chapter 5 describes the debugging and testing of the three algorithms. Chapter 6 describes the analysis conducted on the concurrent algorithms. The aim of this analysis is to compare and contrast the running time and efficiency of the algorithms. Chapter 7 describes CTPlayer, an application whose purpose is to demonstrate the expansion and convergence of ray trees as they created and used by the DIraytrace algorithm. Then Chapter 8 discusses some conclusions that can be drawn from the work described in this document.
Chapter 2

The Ray Tracing Algorithms

2.1 Braytrace

Braytrace stands for Basic Ray Tracing algorithm. Figure 2-1 shows the pseudocode for RayExt, the controlling function of the Braytrace algorithm. As you can see RayExt iterates over the entire projection window. During each iteration, RayExt generates a ray defined by the pixel referenced in the current iteration. If the ray fails to intersect an object in the scene, then the pixel is set to the background color. If an intersection does occur, RayExt sets the current pixel to the value returned by RayExt’s call to the Trace function (see Figure 2-2). Once all the locations in the output window have been iterated over and filled, the algorithm terminates and returns the output buffer.

Trace is the entry point to the recursive ray tracing functions that are used by RayExt to calculate the energy along each root ray. As can be seen in Figure 2-2, Trace returns the value that is returned by its call to the Shade function. Shade first calculates the local energy of the intersection point (due to direct lighting), then adds to that the contributions due to reflected and refracted light. The reflected and refracted contributions are calculated by returning the value that is returned by Shade’s recursive call to Trace.
/* RayExt is the function that controls the execution of BRayExt
the recursive raytracing algorithm. */
RayExt()
{
  get the location of the eye and the size and location of
  the window on the view plane;
  for (each scan line y)
    for (each pixel x) {
      create a ray that goes from the eye through pixel x,y;
      OBJintersected = 0;
      for (each object in the scene)
        if (the object is intersected by the current ray)
          OBJintersected = 1;
      // set the value of the pixel
      if (OBJintersected) outMatrix[x,y] = Trace(ray, 1);
      else outMatrix[x,y] = BACKGROUND;
    } // end for
  } // end for
  Save or Return the outMatrix;
}

Figure 2-1: Pseudocode Braytrace’s controlling function.

2.2 DRraytrace

DRraytrace is a straightforward parallelized extension of Braytrace. DRraytrace
stands for the Delegated Recursive Ray Tracer. It uses identical processes to concur-
rently calculate the values of individual pixels in the output image. These processes
are named Recursive Trace (or TPrecurse) processes. Each TPrecurse process runs
an extended version of the recursive functions listed in Figure 2-2. The new function
is called TraceProcRecurse. The pseudocode for DRraytrace’s controlling function
and TraceProcRecurse function can be found in Figure 2-3 and Figure 2-4.

As can be seen in Figure 2-3, RayExt once again iterates over the entire output
image, creating the root rays that will be used to calculate the pixel values. However,
since DRraytrace calculates the pixel values not by calling Trace, but by sending the
rays to the TPrecurse processes, a new form of communication between the calculation
functions and the controlling function is needed.

Communication between RayExt and TPrecurse processes is provided by an in-
stance of an object named compQ. A compQ object is a singly linked list, where the
Radiance Trace(ray, depth)
{
    determine closest intersection of ray with an object;
    if (the intersection point is valid) {
        compute normal at intersection;
        return Shade(object, ray, intersection,
                     normal, depth);
    } else return BACKGROUND_VALUE;
}

/* Compute shade at point on object, tracing rays for shadows,
reflection, refraction. */
Radiance Shade(object, // Object intersected
    ray, // Incident ray
    ipoint, // Point of intersection to shade
    inormal, // Normal at point
    depth // Depth in ray tree
    )
{
    Radiance Result_Color; // Color of ray
    ray rRay, tRay, sRay; // Reflected, refracted, and shadow rays
    Radiance rColor, tColor; // Reflected and refracted ray colors

    // Calculate the local color
    Result_Color = The ambient lighting in the scene;
    for (each light in the scene)
        Result_Color += contribution of light source on intersection point;

    if (depth < maxDepth) { // if we are not too deep in the raytree
        if (object is reflective) { // calculate the reflected color
            rRay = ray in reflection direction from point;
            rColor = Trace ( rRay, depth + 1);
            Result_Color += rColor * specular coefficient of object;
        }

        if (object is transparent) { // calculate the refracted color
            tRay = ray in refraction direction from point;
            if (total internal reflection does not occur) {
                tColor = Trace ( tRay, depth + 1);
                Result_Color += tColor * transmission coefficient;
            }
        }
    }
    RETURN Result_Color;
}

Figure 2-2: Pseudocode for Braytrace's Trace function.
/* RayExt controls the execution of the concurrent raytracing algorithms */
RayExt()
{
    GlobalCompQ = new compQ;
    Initialize n TPrecurse() processes and pass each process
    a pointer to GlobalCompQ, and the object list;

    select location of eye and the window on the view plane;
    for (each scan line in image)
        for (each pixel in scan line){
            eye = new ray; /* from the center of projection through the pixel*/
            ct = new ConvergenceObject; /* for the current pixel*/
            /* wait until GlobalCompQ is small enough */
            while (GlobalCompQ.size > a predefined minimum);
            /* then add a new computation */
            GlobalCompQ.add(eye, ct);
        }

    /* wait for the destruction of all ConvergenceObjects */
    while (ConvergenceObjects::Instances > 0);

    /* Kill all TPrecurse processes by adding n NULL
        computations to GlobalCompQ*/
    for (int k=0; k < n; k++) GlobalCompQ.add(NULL)

    Clean up all data structures;
    Report Termination;
}

/* TPrecurse is a process that can be run in parallel with itself
    to calculate the output image of a raytrace execution */
TraceProcRecurse()
{
    for(;;) {
        C = GlobalCompQ.get();
        /* if the computation was a NULL, then break*/
        if (C == NULL) break;

        r = the ray associates with C;
        for (each object in scene)
            if ((object is intersected by eye ray) &&
                (object is closer than the closest
                intersected object thus far))
                record intersection and object;

        x, y = the location of the pixel associated to C;
        outMatrix[x,y] = Trace(ray, intersection point, object)
    }
}

Figure 2-3: Pseudocode for DRaytrace's controlling function.
/* TPrecurse is a process that can be run in parallel with itself to calculate the output image of a raytrace execution */

TraceProcRecurse()
{
    for(;;) {
        C = GlobalCompQ.get();
        /* if the computation was a NULL, then break */
        if (C == NULL) break;

        r = the ray associates with C;
        for (each object in scene)
            if ((object is intersected by eye ray) &&
                (object is closer than the closest intersected object thus far))
                record intersection and object;
        // end for
        x, y = the location of the pixel associated to C;
        outMatrix[x,y] = Trace(ray, intersection point, object)
    } // end for
} // end of TraceProcRecurse

Figure 2-4: Pseudocode for DRaytrace’s TraceProcRecurse function.

data at each location of the list is what will be referred to as a Ray Intersect Computation (RIC). An RIC is an object that contains a pointer to a ray, and a pointer to a location where the result of the ray trace calculations can be stored.

A single instance of the compQ data object is shared by all concurrent processes, this instance is called GlobalCompQ. Once a TPrecurse process extracts an RIC from GlobalCompQ, it uses the ray within the RIC as an argument to the Trace function. The value returned by Trace is then placed into the output buffer location defined in the RIC.

2.3 DIraytrace

As was stated in the introduction, all rays can be calculated independently of one another. DIraytrace seeks to exploit the independence of computation associated each individual ray. The concurrent processes that DIraytrace controls are called Incremental Trace (or TPincremental) processes. The main function of a TPincremental
process is called TraceProcIncremental.

As can be seen in the pseudocode listed in Figure 2-5, Dlraytrace introduces two new data objects, the Convergence Tree (CTree) and the Ray Path. Instances of both of these objects are sent with each ray that RayExt places in the GlobalCompQ. When a TPincremental process finishes a calculation, it uses the Ray Path to reference the location in the CTree where the calculated data should be placed.

CTrees are shared data objects, this allows TPincremental processes to concurrently place data into different locations in a particular CTree. This is important due to the fact that a number of processes may be concurrently calculating values for a single CTree.

When enough information has been placed in a CTree, a TPincremental process will converge the data in the CTree. The converged result is then used as a contribution to the final value of the pixel associated with the CTree.

2.4 Extra Features

A number of visual effects can be produced by making simple modifications to the basic and concurrent ray tracing algorithms. Two such modifications, Supersampling and Spectral Distribution calculations, are realized in the implementations of Braytrace, DRraytrace and DIraytrace.

2.4.1 Supersampling

Supersampling refers to the casting of multiple rays through a single subsection in the projection window. The thing to keep in mind is that each subsection in the projection window possesses an area. Supersampling is performed by projecting multiple, slightly offset rays through each location in the window, and then averaging the results of the calculations associated with each ray.

The visual result of supersampling is that edges are “softer” or less distinct, and details that may have not contributed to the output image due to the sparseness of the subsections in the projection window, will now have some effect on the output
RayExt()
{
  GlobalCompQ = new compQ;
  /* The Recursive flag equals 1 then we will be using TPrecurse processes */
  if (Recursive)
    Initialize n TPrecurse() processes and pass each process
    a pointer to GlobalCompQ, and the object list;
  else
    Initialize n TIncremental() processes and pass each process
    a pointer to GlobalCompQ, and the object list;

  select location of eye and the window on the view plane;
  for (each scan line in image)
    for (each pixel in scan line){
      eye = new ray; /* from the center of projection through the pixel*/
      if (Recursive)
        ct = new ConvergenceObject;
      else
        ct = new CTree;
      /* wait until GlobalCompQ is small enough */
      while (GlobalCompQ.size > a predefined minimum);
      /* then add a new computation */
      GlobalCompQ.add(eye, ct);
    }

  /* wait for the destruction of all ConvergenceObjects or CTrees*/
  while (ConvergenceObjects::Instances > 0);
  /* Kill all TPrecurse processes by adding n NULL
 computations to GlobalCompQ*/
  for (int k=0; k < n; k++) GlobalCompQ.add(NULL)
  Clean up all data structures;
  Report Termination;
}

Figure 2-5: Pseudocode for DIraytrace’s controlling function.
TraceProcIncremental()
{
    for(;;) { /* infinitely loop */
        if (a new computation is needed) {
            C = a new computation from the GlobalCompQ;
            extract r (the ray), ct (the CTree) and rp (the RayPath) from C;
        }
        Find the object closest to the eye location that intersects r;
        if (Such an intersection occurs) {
            /* Get Local light Contribution */
            Result_Color = The ambient lighting in the scene;
            for (each light in the scene)
                Result_Color += contribution of light source on intersection point;
            // end for
            Set the local contribution at rp to Result_Color;
        if (depth < MaxDepth) { // we aren't too deep
            if (object is transparent) {
                tRay = ray in refraction direction from point;
                /* create a new RayPath that points to a node in the CTree that is
                down from the node that rp points to. Also flag the so that it
                knows it is the refractive branch of its parent. */
                newPath = rp->down_refraction(ct);
                /* add the new computation to the global computation queue */
                GlobalCompQ->addcomp(tRay, rp, ct);
            } else // if the object is not transparent
                set the refractive contribution at rp to NOCONTRIBUTION;
        if (object is reflective) {
            /* We will compute the reflection contribution when we loop back.
            so set a flag telling the function not to get a new RIC */
            r = the new reflection direction from point;
            /* create a new RayPath that points to a new Reflective
            Child of the current CTree node */
            rp = rp->down_reflective;
        } else set the reflection contribution at tp to NOCONTRIBUTION;
        } else // we are too deep
            Set the reflected and refracted contributions at rp to NOCONTRIBUTION;
    } else // no intersection occurred
        set the reflection, refraction and local contributions at rp to NOCONTRIBUTION;
    if (ct->Converged()) { // if ct fully converges then writeout the data
        x, y = the pixel location stored in ct;
        outMatrix[x,y] = ct->FinalResult;
        delete ct;
    }
} // for(;;)
} // end of TPIncremental

Figure 2-6: Pseudocode for DIraytrace's TraceProcIncremental function.
To perform the supersampling of an output image, Braytrace must create the supersample rays, then average their return values.

To accomplish supersampling in the DRraytrace algorithm, RayExt must create the supersample rays, and then send a single instance of a new data object, the supersample Convergence object (supConvObj), to the TPrecursive processes. This new object possesses functionality which facilitates the convergence of supersampled data.

Dlraytrace works in a way similar to that of DRraytrace. However, instead of using the supConvObj object, Dlraytrace uses an object derived from CTree. This new object is called a Supersample Convergence Tree object (supCTree).

### 2.4.2 Spectral Distribution Calculations

Spectral Distribution (SPD) calculations take into account the fact that materials possess different refractive properties for different wavelengths of light. One example of this phenomenon is the way in which a crystal refracts white light. Since a crystal has a different refractive index for different wavelengths of light, the light that exits a crystal is a spectrum that is spread over an area (as opposed to a simple beam of white light).

To perform SPD calculations, each object in the scene is given three refractive indices corresponding to red, green or blue light. Then, when a ray hits an object, three refraction rays (one for each index) are cast rather than one. When converging the ray tree to which the refractive rays belong, only the color of light for which a ray was cast will contribute to the final result. For example if a white light strikes a crystal, three refractive rays (a red ray, a green ray and a blue ray) are cast. Then, when converging the ray tree, only the red contribution of the red transmission ray, the green contribution of the green ray and the blue contribution of the blue ray will be used in final result of the ray tree.

To implement SPD calculations, the way in which recursive rays are created must be modified. Braytrace and Dlraytrace must now spawn three transmission rays,
then scale the values that are returned for each ray. The scaled results can then be used as contributions to the ray tree's final value.

The modifications to DIraytrace are very similar to the modification made to Braytrace and DDraytrace. DIraytrace must create three refractive rays instead of one, then converge the resulting contributions of the three rays. The difference between DIraytrace and the other two algorithms is that instead of making a recursive function call to calculate the energy of each recursive ray, DIraytrace places a new ray into the GlobalCompQ so that an idle process can calculate that ray's contribution.
Chapter 3

Implementations of the Ray Tracing Algorithms

The pseudocode in the previous chapter hides many important implementational issues. For example what primitives are used in the Braytrace algorithm? What mechanisms are employed by DRraytrace and DIraytrace to synchronize the concurrent processes? How does DIraytrace converge a CTree?

This chapter will describe the functions and data objects that are used to realize the algorithms detailed in the previous chapter.

3.1 Braytrace

Let us first discuss the implementation of the Braytrace algorithm. Let us begin by describing the data structures and functions used by Braytrace. Then we shall describe the calling sequence of the functions that control the algorithm.

3.1.1 Braytrace: Data Objects and Functions

Many of the data objects used in Braytrace are provided by or derived directly from objects that are defined in the Silicon Graphics InventorGL graphics library [4].

---

1The code and data objects for the Braytrace algorithm are taken from code used in a Computer Graphics course taught by Professor Seth Teller at the Massachusetts Institute of Technology.
Below is a list describing the data objects and functions used by Braytrace\textsuperscript{2}.

**Point:** Point encapsulates the three coordinates that define a location in a scene.

**ObjectData:** ObjectData describes an object in a scene. It encapsulates such information as the type of material that the object is made of (used for the calculation of transmitted and reflected rays), the location and orientation of the object, and the faces, vertices and edges that define the object.

**LightData:** LightData describes a light source in the scene. It encapsulates such information as the light's location, direction, color and intensity.

**Octree:** Octree is a data object that is used to store and retrieve all the entities in a scene. It possesses member functions that allow a caller to add objects to a scene, search for objects in a scene and list all the objects in a scene.

**Ray:** Ray encapsulates information used to describe rays that are cast by the ray tracing algorithms. The information that Ray encapsulates includes a ray's origin and direction, a Spectral Distribution mask (used for Spectral Distribution Calculations), and an ObjectData pointer that points to an object that the ray may be propagating through (used for transmission calculations).

**Hit:** Hit describes an intersection that has occurred between a ray and an object. It contains a pointer to the object, the location of the intersection, the normal vector at the point of the intersection, and whether the ray was entering or leaving the object.

**ViewVolume:** ViewVolume contains information on how the scene will be viewed. The information it encapsulates includes the location and orientation of the viewer of the scene, the distance from the location of the viewer to the projection plane and the distance from the viewer to the farthest objects that will appear in the output image (these distances are often referred to as the near and far clipping planes).

**Radiance:** Radiance is used to encapsulate data that describes a color. The information contained in a Radiance object is composed of three floating point values ranging from 0 to 1. Each value corresponds to the amount of red, green or blue light in the composite color.

**setPixel(…):** The setPixel function is used to set the value of a pixel in the output image. Its arguments are an \(<x,y>\) coordinate and a Radiance. A pointer to a setPixel function is passed to a ray tracing algorithm during that algorithm's initialization. Then, as the algorithm executes, the setPixel function is used to store the pixel values that are calculated by the algorithm.

\textsuperscript{2}In this list and all lists to follow, "<obj1> -> <obj2>" means that obj2 inherits all data and functionality encapsulated in obj1.
RayExt(...): RayExt is the controlling function of all three ray tracing algorithms. Its arguments include an array of ObjectData (containing the objects in the scene), an array of LightData (containing the lights in the scene), a pointer to a ViewVolume and a pointer to a setPixel function.

ivTrace(...): The ivTrace function is an implementation of the Trace function described in Section 2.1. It takes as its arguments a Ray, an Octree containing the objects in a scene, a depth, a weight and an index of refraction. It returns the Radiance along the given Ray. The depth and weight are used for recursion termination.

ivIntersect(...): The ivIntersect function takes as its arguments a Ray, an Octree containing the objects in a scene, and a pointer to a Hit object. Its return type is an integer. If the ray intersects an object in the scene, then the Hit object pointed to by the Hit pointer is modified to describe the intersection, and a 1 is returned to the caller. Otherwise 0 is returned.

ivShade(...): The ivShade function takes as its arguments a pointer to a Hit object, a viewing direction, a Spectral Distribution mask, an Octree containing a list of objects in the scene, an array of LightData containing the lights in the scene and the depth and weight values from ivTrace. The ivShade function then returns a Radiance object that contains the color at the intersection contained in the Hit object.

The code for these objects and functions can be found in Appendix D.

### 3.1.2 Braytrace: Calling Sequence

Figure 3-1 contains a flow diagram describing the calling sequence of the Braytrace algorithm. The figure illustrates the recursive function calling that is employed by the Braytrace algorithm.

### 3.2 DRraytrace and DIraytrace

A number of new data objects and functions are needed to realize the DRraytrace and DIraytrace algorithms. Figure 3-2 contains an inheritance diagram that shows the objects that are used by the concurrent algorithms.

As you can see, most of the new objects use the shared data object lock. The lock object is what is referred to as a Mutual Exclusion (mutex) object. Mutex
objects are one of the most basic synchronization\textsuperscript{3} mechanisms used by concurrent algorithms. The validity and efficiency of a mutex object is central to the performance and correctness of concurrent algorithms. For a more thorough discussion on mutual exclusion objects refer to [7].

3.2.1 DRraytrace: Objects and Functions

Below is a list containing all the new data objects and functions used by the DRraytrace algorithm.

\textbf{ConvergenceObject}: ConvergenceObject is a base object (i.e. other objects will be derived from it) that encapsulates functionality and information needed to converge ray tracing calculations to an output Radiance. ConvergenceObject possesses a pointer to a setPixel function. It also contains data describing the number of ConvergenceObject instances in existence (the number of existing instances is used to check for termination of the algorithm).

\textsuperscript{3}Synchronization refers to the action of getting asynchronous, concurrently executing processes to access data or execute code in a synchronous manner.
Figure 3-2: Object Inheritance and Data Dependency diagram for DRraytrace and DIRaytrace. Broken lines indicate a data dependency, solid lines indicate a derived class.
supConvergenceObject: (ConvergenceObject->supConvergenceObject) supConvergenceObject is a ConvergenceObject that possess the additional functionality and data needed to converge ray tracing calculations that are performed during the creation of a supersampled output image.

RIC: RIC stands for Ray Interception Computation. It describes a concurrent ray tracing computation. RIC is a base class from which specialized RICs are defined. The only data that is encapsulated by a base RIC object is a Ray.

RIC_CO: (RIC->RIC_CO) RIC_CO is an RIC that additionally possesses a pointer to a ConvergenceObject (this pointer may also point to a supConvergenceObject). RIC_CO objects are used to convey root rays to TPrecurse processes.

lock: A lock is a mutual exclusion object. These objects are shared by concurrent processes. They are used in situations where concurrent processes may interfere with one another, or where the synchronization of concurrently executing processes is needed. A lock can be "acquired" or "released". Once a lock is acquired it cannot be acquired again until it is released. The lock object is implemented using an ablock.t data structure. These data structures are primitive mutual exclusion objects that are built into Silicon Graphics' Iris Unix operating system [1].

llmutex: The llmutex object is a sharable singly linked list. It can be accessed simultaneously by multiple concurrently executing processes. It uses lock objects to ensure the integrity of the list even when the list is being accessed simultaneously by multiple processes. The llmutex object is a base class for more specialized sharable linked lists.

llEntry: The llEntry object is a node in an llmutex list.

compQEntry: (llEntry->compQEntry) The compQEntry object is an llEntry that contains a pointer to an RIC object (note that a pointer to an RIC object can also point to any object derived from an RIC object).

compQ: (llmutex->compQ) The compQ object is a linked list of compQEntries. This new linked list provides functions for adding and retrieving RIC objects. It further provides functions for waiting on the availability of RICs in the list. These wait functions are used so that a process that is waiting for an RIC, can be informed when an RIC is available.

pidEntry: (pidEntry->llEntry) A pidEntry is an llEntry that contains a process identifier, and a lock.

PidAwakenList (llmutex->PidAwakenList) A PidAwakenList is an llmutex object that allows a calling process to add itself to a list of processes that are waiting to be awakened by some other process. It also allows a process to awaken some or all of the processes that are in the list. PidAwakenList uses pidEntry nodes to keep track of what processes are in the list. The mechanism for awakening
processes in a PidAwakenList uses the lock that is contained in the pidEntry object. When a process wishes to be added to a PidAwakenList, it calls a member function of the list which returns a lock object that has already been acquired. The process then attempts to reacquire the lock. Since the lock is already acquired, the process must wait until the lock is release before it can continue executing its code. The list, when informed that it should awaken a process, will release the lock that is associated with that process, thereby enabling the process to acquire its lock and proceed with its computations. The compQ object uses the PidAwakenList as a means of telling processes that RICs are available.

**TraceArg:** TraceArg encapsulates information that is needed when initializing TPre-cursive processes. The TraceArg data object contains pointers to a compQ object, an Octree data object and a LightData array which contains the lights in the scene.

**sproc(...):** The sproc function [2] is a function call that “forks” off new processes. It is used to initialize the TPrecurcive and TPincremental processes. One of its arguments is a pointer that will be passed to the forked process. This pointer is used to transmit a TraceArg object from RayExt to the TPrecurcive and TPincremental processes.

**RayExt(...):** RayExt is the function that supervises the execution of all the ray tracing algorithms. When RayExt is controlling the DRaytrace algorithm, it creates a single GlobalCompQ object. Then it uses sproc to initialize the TPrecurcive processes. RayExt then iterates over the pixels in the output image, creating root rays and RICs that will be sent to the current processes via the GlobalCompQ. Lastly, when all pixels have been iterated over, RayExt reports the termination of the algorithm.

**TraceProcRecurse(...):** TraceProcRecurse is the main function of a TPrecurcive process. Upon initialization it gets the TraceArg object that was sent to it by RayExt, and from that object it extracts all the information that it needs to initialize itself. Then it waits for an RIC to become available in the GlobalCompQ. Once an RIC is available it calculates the radiance along the ray contained in the RIC, then records the result in the output image location specified in the RIC. Once a computation is complete, TraceProcRecurse loops back and waits for the next RIC to become available. TraceProcRecurse interprets a NULL RIC as a signal that RayExt wants the process to terminate.

These objects and functions possess the functionality that is needed to perform the DRaytrace algorithm. The source code for these objects and functions can be found in Appendix E.
3.2.2 DRraytrace: Calling Sequence

Figure 3-3 contains a flow diagram describing the calling sequence for the DRraytrace algorithm. This figure illustrates the division of labor that exists in the DRraytrace algorithm. This division is facilitated by the communication that takes place via the GlobalCompQ.

One important quality that the GlobalCompQ must possess is that it must guarantee that only one process can retrieve any particular RIC. Let us look at an example of why such a guarantee is needed.

Figure 3-4 contains the pseudocode for a function that extracts the first element from a standard singly linked list. In the case where only one process will execute this function, the code in the figure is valid. However it is not difficult to derive examples where if two processes are executing this code nearly simultaneously, the code yields...
LinkedListEntry * LinkedList::get_top()
{
    /* Assuming that the _size of the list is consistent with 
       what is actually in the list, check that the list is non- 
       empty by checking the size of the list. */
    if (_size == 0) // if the list is empty than return NULL
        return NULL;
    // Otherwise decrease the _size
    _size--;          
    // manipulate the top pointer
    LinkedListEntry * e = head;
    // move the head pointer
    head = head->next;
    // return the entry
    return e;
}

Figure 3-4: Simple Linked List Pseudocode

incorrect results.

Example:

1. Process i enters the get_top function and checks the _size variable. The _size variable is currently set to 1.

2. Process j enters the get_top function and checks the _size variable. The _size variable is still 1.

3. Processes i and j then both attempt to get the element in the list. Since they both decrease the _size variable, the _size is now set to -1.

4. Both i and j then set their respective e pointers to head.

5. i then sets head = head->next. The result of this instruction is that head is now set to NULL.

6. j tries to set head = head->next, but since head is already set to NULL this instruction causes an invalid pointer reference.

This example shows the incorrectness of this function when it is being concurrently executed by multiple processes.

Now look at the pseudocode for llmutex's get_top member function (refer to Figure 3-5). This code prevents errors like the ones in the example just given by limiting the critical retrieval code to one process at a time. Such exclusion policies are present
llEntry *llmutex::get_top()
{
    // check if the list is nonempty
    if (_size == 0) return NULL;
    // lock the _size
    sizeLock.lock_resource(TRUE);
    // recheck _size
    if (_size == 0) {
        sizeLock.unlock_resource();
        return NULL;
    }
    // manipulate _size
    _size--;
    // lock, manipulate and unlock top
    headLock.lock_resource();
    llEntry *e = head;
    head = head->next;
    headLock.unlock_resource();
    sizeLock.unlock_resource();
    // return the element
    return e;
}

Figure 3-5: llmutex Pseudocode

in most of the shared object classes used in the DRraytrace and DIraytrace algorithms.

3.2.3 DIraytrace: Objects and Functions

Below is a list containing the new data objects and functions that are used by DIraytrace.

CTree: (ConvergenceObject->CTree) CTree is an object that represents a ray tree. It provides functions to traverse the tree, and set values within the tree.

CTreeNode: CTreeNode is a node in a CTree. It contains node pointers that point to a node’s reflective child, refractive children, and parent. It also contains Spectral Distribution, weight and scaling coefficient information. Lastly it contains a lock object and a integer flag. These last two data objects are used to ensure the correct convergence of the data in the CTree.

CTreeRec: (CTree->CTreeRec) CTreeRec is a CTree that has the added capability of saving a log of the creation and convergence of the CTree to an external
data file. This log is created by augmenting all the functions that manipulate the CTree object with commands that write to a log file (see Chapter 7 and Appendix B for more information on CTreeRec and log files).

**RayPath:** RayPath is a data object that specifies a location in a CTree or CTreeRec object. It contains a pointer to a node in a CTree as well as functions that allow a caller to traverse and create new nodes in a CTree.

**RIC_CT:** (RIC->RIC.CT) RIC.CT is an RIC object that includes a pointer to a CTree object and a pointer to a RayPath object. RIC.CT objects are created by RayExt when RayExt is controlling TPincremental processes. RIC.CT objects are also created by TPincremental processes when they need to delegate work to other TPincremental process.

**supCTree:** (CTree->supCTree) The supCTree object is a CTree that possesses additional data and functionality to support the supersampling of a scene. It does this by storing an array of CTrees, and then converging the final result of each CTree to a composite radiance for the entire supCTree object.

**supRayPath:** (RayPath->supRayPath) The supRayPath object is a RayPath that is used when referencing supCTree objects. In addition to the data that is contained in a standard RayPath object, supRayPath also indicates which CTree within a supCTree object is being referred to.

**RayExt(...):** RayExt, when controlling the DIFraytrace algorithm, creates CTree, CTreeRec or supCTree objects for the RICs that are sent to the TPincremental processes (supCTree objects are created if supersampling is enabled, and CTreeRec objects are created when a log of the creation and convergence of the CTree is needed).

**TraceProcIncremental(...)**: TraceProcIncremental is the main function of a TPincremental process. Its execution is best described as a merging of the TraceProcRecursive and ivShade functions. TraceProcIncremental extracts the next available RIC.CT object from GlobalCompQ. It then checks if the ray in the RIC intersects an object in the scene. If an intersection occurs, then the local radiance at the intersection point is calculated (by calling ivShade) and stored in the CTree. If needed, reflective and refractive rays, as well as RIC.CT objects to store those rays, are created and added to GlobalCompQ. TraceProcIncremental then loops back and if the last computation spawned a reflection ray, it calculates the radiance of that ray and places the result into the CTree. When TraceProcIncremental can no longer calculate or converge data for the current RIC.CT, it discards that RIC and attempts to acquire a new RIC.CT object from the GlobalCompQ. Just as with TraceProcRecursive, TraceProcIncremental interprets the retrieval of a NULL RIC.CT from the GlobalCompQ as an indication that RayExt wants the process to terminate.
These objects and functions possess the functionality that is needed to execute the DIraytrace algorithm. The source code for these objects and functions can be found in Appendix E.

### 3.2.4 DIraytrace: Calling Sequence

Figure 3-6 contains a flow diagram that describes the calling sequence of the DIraytrace algorithm. As you can see, the execution of the RayExt function remains primarily unchanged. The major difference between the execution of this algorithm and the execution of the DRraytrace algorithm is the non recursive nature of the TPIncremental process.
3.2.5 DInRaytrace: Convergence of CTree Data

One important aspect of the DInRaytrace implementation that is hidden by the CTree object class is the issue of which process actually converges the data in a CTree. The answer to this question lies in the implementation of the CTree object. In particular the SetReflected, SetRefracted and SetLocal member functions of CTree and its derived types.

CTree::SetReflected, CTree::SetRefracted and CTree::SetLocal take as arguments a Radiance, and a RayPath\(^4\). Once the radiance is recorded in the node, a private member function named Converge is called\(^5\).

Converge attempts to converge the data in the CTree. Converge's only argument is a RayPath. Converge extracts from that RayPath object the CTreeNode from which convergence should be attempted. The mutex lock that is contained within each CTreeNode is used to figure out what process converges the data that is contained in the node.

When a process calls one of the set functions, the lock for that node will be acquired, thus preventing other processes from setting a value in that node, or converging the data in that node. Once a process has the lock for a node, it will, after setting the data in the node, attempt to converge the data in that node to the node's parent. If successful the node's flag is set to a value that indicates that the node has been converged. Once a process has attempted to converge the data in a node, it will release the node's lock and exit the Set function.

Once a CTree is filled, one of the last processes (but not necessarily the last process) that attempts to converge the tree will succeed.

---

\(^4\)The SetRefracted function also takes an integer that indicates which Spectral Distribution contribution needs to be set.

\(^5\)Private refers to the fact that Converge is not visible outside the object.
Chapter 4

User Interface

4.1 Graphic User Interface

All three of the rendering algorithms just described are incorporated into a single program named DTrace. DTrace's user interface is mostly based on an interface that was provided with the code from Seth Teller's class. This interface was created using the OpenGl application development toolkit [3].

To provide access to the DRraytrace and DIraytrace algorithms, various additions have been made to the original interface.

Figure 4-1 shows the graphic user interface of DTrace. Small modifications have been made to the button list on the left side of the window. The three buttons that appear on the left side are now linked to the three different rendering algorithms (Braytrace, DRraytrace and DIraytrace). Other than this change, the interface is the same as it was originally.

Some of the other controls that are available through the graphic interface provide ways of manipulating the lighting in the scene, and changing the orientation of the eye of the viewer.
4.2 Commandline Interface

The original code also implemented a very limited command line interface. This interface simply set certain flags which defined how Braytrace would execute. To help with the testing and debugging of the concurrent algorithms, the command line interface was expanded.

As you can see in Table 4.1 the commandline parameters are divided into old and new options. The old options mostly pertain to the common elements of all three algorithms, while the new options are mostly needed for the concurrent algorithms.

Some of the most important new options allow a user to not only set the parameters used by the concurrent algorithms, but to indicate that the image should be rendered immediately using the information on the command line. The options that provide for the immediate rendering of an image are -eye <eye file> and -UI <interface mode>.

By creating and indicating an eye file (the format of which is described in Ap-
OLD OPTIONS:

[-script <file name> [-c]]
[-read <file name> [-i N] [-c]]
[-m rendermode]
   Inventor
   Raytrace
[-f <base file name>]
[-o]
[-d maxraydepth >= 0]
[-e environ.iv]
[-j(itter samples)]
[-p N (cast N x N rays/pixel) >= 1]
[-eye <filename>]
[-s shademode ( Flat
   Normal
   Ambient
   Local
   Direct
   Recursive)]
[-w minrayweight >=0]
[-D debuglevel >=0]
[-d (- for stdin)]

NEW OPTIONS:

[-del] (delegate computations)
[-n nprocesses] (nprocess >= 1; default = 2)
[-UI <line | gl>]
[-size w h]  (in glmode just resize the window,
                otherwise default = 50x50)
[file.iv (- for stdin)]
[-CT <file name>]
[-old] (use the old raytracer tracer)
[-help]

Table 4.1: Commandline options for DTrace
appendix B.3), and then by setting the -UI option to “line” DTrace will render the image without starting the interactive graphic user interface (if no eye-file is used, a default location is set).

Another new option that is important to debugging the concurrent algorithms is the -CT <Convergence Tree File> option. When this option is set, DTrace reads the designated Convergence Tree File (see Appendix B), then, for every pixel listed in the Convergence Tree file, RayExt creates a CTreeRec object (see Section 3.2.3) as opposed to a CTree object.
Chapter 5

Debugging and Testing

This chapter will describe the methods used to test and debug the code that implements DRraytrace and DIraytrace. The simplest and most effective way to test these concurrent algorithms is to have them render images to which the output is known. Since the code for Braytrace was provided, the output from that algorithm can be used as a control to which the output of the other algorithms can be compared.

Once a scene is rendered by both Braytrace and one of the concurrent algorithms, a comparison of the output file sizes invariably indicates if the output files are identical. However even if the sizes of the output files are the same, a second, more reliable comparison can be made. This second comparison is a bit by bit comparison of the output files using the Unix cmp program [6, p. 2.17].

During testing, when a discrepancy in the output of one of a concurrent algorithms was discovered, a number of steps could be taken to track down the incorrect code. The X Windows graphic image viewing utility xv [8] was used to locate the actual <x,y> coordinates of pixels that had incorrect values. Then a CT file (refer to Appendix B) containing the incorrectly calculated pixel locations could be created and used to trace the creation and convergence of the CTree associated with the incorrect pixels.

Dbx [6, pgs. 20.1-20.13], a source level debugger, could then be used. Dbx would provide a way of tracing the executions of the old or new algorithms as they calculated the values of the pixels in question. Although tedious, this technique invariably
indicated where the incorrect code could be found.

Initially the code was written without supersampling or Spectral Distribution calculations. The initial test scenes were simple and did not possess any reflective or refractive objects. These test cases uncovered most of the implementation errors in the compQ, lmutex, lock, ConvergenceObject, RayExt and TraceProc components of the implementation. Due to the code sharing between DRaytrace and Braytrace, the DRaytrace algorithm worked well and was completely debugged by the end of this stage.

The next test cases used reflective objects. These test cases uncovered a number of errors in the TraceProcIncremental function, and the CTree and RayPath objects.

At this point Spectral Distribution calculation code was added to TraceProcIncremental\(^1\). To test this code, as well as all the remaining untested code, scenes that possessed refractive objects were rendered. These tests uncovered the bulk of the implementational errors. As stated earlier DRaytrace was unaffected by these tests, but DRaytrace had a number of minor implementational errors that needed to be addressed. Some examples of these errors were flaws in the constructor and destructor sequences for some of the derived data objects, locking sequences which resulted in the dead lock of TPincremental processes, and a number of simple pointer manipulation errors.

Once the output of all three algorithms agreed for all test cases, supersampling was added to the DRaytrace and DRaytrace algorithms. To test the supersampling code, all that was needed was to enable supersampling in the old and new algorithms and compare the resultant images. The only errors that were found were in the supConvergenceObject and supCTree data objects.

\(^1\)Spectral Distribution code already existed in DRaytrace because of the code it shares with Braytrace.
Chapter 6

Analysis of the Concurrent Algorithms

A runtime analysis was conducted on the three ray tracing algorithms. This analysis was done to achieve a better understanding of the issues involved in an efficient implementation of the original ray tracing algorithm. This chapter will describe the statistics that were compiled for the analysis, and will detail a number of optimizations and conclusions that were drawn from the statistics.

6.1 Initial Analysis and Optimizations

During the debugging stage of this project, observations of the running times of the ray tracing algorithms seemed to indicate that given the same test cases, DRraytrace, and in some cases Braytrace, were terminating earlier than DIraytrace. To verify these observations, the Unix time program [6] was used to get a precise measurement of

<table>
<thead>
<tr>
<th></th>
<th>Braytrace</th>
<th>DRraytrace</th>
<th>DIraytrace</th>
</tr>
</thead>
<tbody>
<tr>
<td>One Processor</td>
<td>29.6 secs</td>
<td>30.5 secs</td>
<td>44.2 secs</td>
</tr>
<tr>
<td>Two Processors</td>
<td>24.0 secs</td>
<td>35.3 secs</td>
<td></td>
</tr>
<tr>
<td>Four Processors</td>
<td>16.6 secs</td>
<td>23.0 secs</td>
<td></td>
</tr>
</tbody>
</table>

Table 6.1: Runtime durations for the ray tracing algorithms.
the exact running time of each algorithm\textsuperscript{1}. The results of these tests are listed in Table 6.1.

The results in Table 6.1 indicated that the two concurrent algorithms used well over their ideal running times\textsuperscript{2}. The obvious question of where the overhead in the concurrent algorithms was coming from, needed to be answered.

The first step was to make sure that all algorithms were executing similar numbers of calculations. In order to verify this, counters were placed in the code of all algorithms. These counters accumulated various statistics like the number of rays cast, the number of rays that didn't intersect an object, the number of rays that did intersect an object and the number of calls to ivIntersect (refer to section 3.1.1).

The initial test case was the rendering of a small image (only 25 by 25), where the concurrent algorithms would use 4 concurrent processes. Braytrace and DRaytrace returned the same results; 12037 rays cast, 3690 rays hit, 2988 rays missed and 275694 calls to ivIntersect. DiRaytrace returned the same number of rays cast, rays hit and rays missed. But it indicated that only 276389 calls to ivIntersect had been made.

It turns out that due to a small inefficiency in the original code, DiRaytrace is capable of performing fewer intersection checks. Obviously whatever overhead existed in the concurrent algorithms was not to be found in the number of calculations performed.

Another possibility was that the GlobalCompQ was not feeding enough RICs to the concurrent processes. To check this theory another counter was added. This counter was placed in the member function of compQ whose purpose is to retrieve RICs from the list. This counter would increment every time a request for a new RIC resulted in the calling process having to wait for an RIC to be added to the computation queue. This statistic could be interpreted as the number of times a TPIncremental process had to wait on RayExt or another TPIncremental process to add a new RIC to GlobalCompQ.

\textsuperscript{1}These tests were run on a Silicon Graphics Symmetric Multiprocessing machine that possessed 4 parallel processors.

\textsuperscript{2}If OLD equals the non concurrent running time, and \( n \) is the number of concurrent processes, then the ideal running time of a concurrent algorithm is \( \frac{OLD}{n} \).
While rendering a 100 by 100 image (again using 4 concurrent processes), DRraytrace caused TPrecursive processes to wait 434 times, and DImraytrace cause TPincremental processes to wait 673. These numbers were far too high.

The policy that RayExt was using for inserting new RICs into the GlobalCompQ was that RayExt would wait for the size of the queue to be less than the number of parallel processes running in the execution. When this condition was met RayExt would add twice as many RICs as there were processes to GlobalCompQ. To get the number of waits down, this policy was changed so that RayExt would add 4 times as many RICs as there were processes. This insertion would be carried out when there were less than 4 times as many RICs in the queue as there were concurrent processes.

This new policy resulted in DRraytrace causing only 17 waits, and DImraytrace causing 23 waits. However, even after this optimization, the concurrent algorithms continued to ran noticeably longer than their ideal running times.

6.2 Runtime Profile Statistics

The next step taken to locate the overhead in the concurrent algorithms was to employ the use of the Unix runtime profiling facilities.

Profiling is a method of automatically gathering runtime statistics. These statistics are accumulated by code that is added to a program during the linking\(^3\) of that program.

The profiling code is added when a commandline parameter of the compiler or linker is set. If a program is created with the profiling code attached to it, after that program terminates an execution, a profile table will be saved to a file\(^4\). This table contains all the runtime statistics that were gathered during the run of the program. The table that is saved can then be read using the Unix prof [6] program.

One type of statistical gathering method that is supported by the UNIX profiling

---

\(^3\)Linking refers to a stage during the creation of an executable program. After a program is compiled, it it linked to system and runtime libraries which provide code for all the system and library references within a program.

\(^4\)When profiling is performed on programs that fork processes, each process, upon termination saves its own profile.
facilities is called pc-sampling. PC-sampling interrupts the execution of a program every 10 milliseconds. During each interruption the location of the program counter is recorded. When the program terminates, a profile table with the pc-sampling information is accumulated and saved to a file.

For the analysis of the concurrent algorithms, profile statistics were gathered on a TPIncremental process, a TPrecurrsive process and an entire Braytrace execution. Table 6.2 contains a condensed version of the profile information gathered during these tests 5.

The values in the each column contain the percentage of the running time, and the number of pc-samples taken in the type of functions associated with that column.

The profile statistics listed in the table indicate that a considerable portion of the computation time of a TPIncremental process is spent in procedures that support memory usage and synchronization.

After examining the implementations of the concurrent algorithms it became clear that the increased computation times used by TPIncremental processes are due to DRaytrace's increased usage for dynamic memory and mutual exclusion objects.

TPrecursive processes have no significant need for dynamic memory, and only need to synchronize themselves when they are accessing the GlobalCompQ. Therefore the overall running time of the DRaytrace algorithm is consistently lower than that of DRaytrace.

TPIncremental processes must synchronize themselves not only when they access

\[\text{Table 6.2: Profile Statistics}\]

<table>
<thead>
<tr>
<th>Braytrace</th>
<th>Radiance Calculations</th>
<th>Memory</th>
<th>System and Synchronization</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>98% (2525)</td>
<td>0.2% (5)</td>
<td>0.5% (12)</td>
<td>98.7% (2542)</td>
</tr>
<tr>
<td>DRaytrace</td>
<td>98% (1790)</td>
<td>0.8% (15)</td>
<td>0.4% (7)</td>
<td>99.2% (1812)</td>
</tr>
<tr>
<td>DRaytrace</td>
<td>76% (1902)</td>
<td>13% (316)</td>
<td>6.6% (167)</td>
<td>95.6% (2385)</td>
</tr>
</tbody>
</table>

5The Braytrace statistics in this table are taken from an entire execution of DTrace, while the statistics for DRaytrace and DRaytrace are taken from the execution of a single concurrent process. This means that the Braytrace statistics include an overhead associated with the initialization of the ray tracing algorithm, while the concurrent statistics do not. Also note that the totals in the table do not add up to 100% because the default settings of the prof utility do not print procedures that had very small sampling sums. For a complete listing of the profile tables, and for further discussion on how to read profiling information, refer to Appendix A.
the GlobalCompQ, but when they converge a CTree. These processes also execute more dynamic memory operations due to their need to create and destroy Ray objects, RIC objects and CTree objects.

Another contribution to the overhead of TPincremental processes stems from TPincremental’s increased use of the GlobalCompQ. The GlobalCompQ possesses hidden internal costs due to its usage of dynamic memory. Whenever an entry is added to or removed from the compQ structure, memory needs to be allocated or deallocated. Since, in the DlRaytrace algorithm, TPincremental processes as well as the RayExt function, can add entries to GlobalCompQ, the hidden costs of the compQ structure become more noticeable.
Chapter 7

CTPlayer Demonstration Application

To help visualize the creation and convergence of CTrees by the DRaytrace algorithm, a demonstration application was written. This application, named CTPlayer, takes as its input a log file that was saved by a CTreeRec object. CTPlayer then provides the user with a means of graphically replaying the creation and convergence of the CTree described by the log file.

7.1 CTPlayer Interface

Figure 7-1 shows the interface of CTPlayer application. As you can see in the figure, the interface is divided into 3 sections. At the top of the interface is the Button Menu, at the bottom is the Instruction Window, and in the center is the Tree Display.

7.1.1 Button Menu

The Button Menu consists of buttons that are all linked to simple operations that can be performed on the information that was parsed from the CT file. These operations are listed below.
Figure 7-1: The Graphic User Interface for the CTPlayer demonstration application.
Play: Play will begin a graphic play back of the creation and convergence of the CTree as specified by the CT file.

Stop: Stop the playback of a CT file.

FStep: Update the display with the next instruction in the CT file.

BStep: Update the display after undoing the last instruction that was executed.

Skip: Skip will bring up the skip dialog so that a new location in the CT file can be immediately skipped to.

Quit: Exit the application.

These buttons provide a user with the ability to skip or play to any location in the CT file.

7.1.2 Instruction Window

At the bottom of the interface is the a window that displays a readable list of the instructions in the CT file. The first column of each row contains the number of the instruction. The second column contains a process identifier, indicating what process executed the instruction on that line. The last column contains text describing the instruction on that line.

A number of different types of instructions can be listed in the Instruction Window. Refer to Appendix B.2 for a list of the types of instructions that may appear in a CT file.

7.1.3 Tree Display

The Tree Display contains the current state of the CTree. Nodes in the tree are displayed from left to right (nodes that are farther left are higher in the tree). A node is drawn as a rectangular region that has some fraction of its interior filled. A completely filled interior indicates that all the data fields in that node have been filled, and that the node's data can be or has been converged to it parent.

One node in the tree is always highlighted. The highlighted node is the node being referred to by the instruction that is highlighted in the instruction window.
Finally, although it is not clear in the figure, the lines that connect the nodes are colored to indicate the relationship of a child node to its parent. A gray line indicates that a child was created to converge the reflective contribution of its parent. Red, Green and Blue lines are used to indicate nodes that were created to find the Red, Green and Blue spectral distribution contributions of their parent.

Figure 7-2 shows the Tree Display after the application has skipped to a location in the middle of a CT file. This figure illustrates how the Tree Display renders the state of a CTree.

As the CTPlayer goes from one instruction to the next, the instruction window scrolls in order to keep the current instruction in the window. The current instruction is then highlighted to indicate the current position in the CT file.

At the same time the Tree Display will highlight the node associated with the
current instruction, expand the tree if new nodes are created, and fill in a node if new information has been added to a node in the tree.

Figure 7-3 show CTPlayer after using the FStep instruction to advance to instruction 120. As you can see, the CTree has expanded by adding new nodes to the lower portion of the tree.

If the BStep button is pressed the instruction window will scroll up a line. At the same time the Tree Display may empty a filled node or remove a node from the tree.

If the skip option is used, the skip dialog is brought up. In this dialog a user may type a numeric value associated with an instruction in the CT file. If a valid number is entered and the accept button is pressed, the CTree in the Tree Display will be updated to indicate what the tree looks like at the instruction that was skipped to. If the cancel button in the dialog is pressed, then no skip takes place.
Figure 7-4: The Graphic User Interface for the CTPlayer demonstration application after skipping to the end of the CT file.
If during the playing of a CT file, the CTree becomes too large to be displayed in the Tree Display, the scroll bar to the right of the Tree Display window can be used to move the display up or down.

Figure 7-4 illustrates a case where the tree is too large to fit in the Tree Display. As you can see, the tree has completely converged.

### 7.2 CTPlayer Design and Implementation

CTPlayer was implemented using the Xlib [9] and X Toolkit Intrinsics [10] application development interfaces. The widgets provided in the standard MIT Athena Widget Set were used to create the basic user interface elements (e.g. the command buttons, and text window) that can be seen in figures of the last section.

The design of the application was not a very complicated one. Below is a list of the new data objects that were defined and implemented to realize the CTPlayer application.

**strll**: The strll object is a simple object oriented definition of a linked list of character strings. In addition to the standard linked list commands like add entry, next entry, etc. strll has the additional functionality needed to condense the data in the list into either an array of character strings, or a single contiguous block of characters.

**strllNode**: The strllNode object is a node in an strll string list.

**CTreeNodePlayer**: (CTreeNode->CTreeNodePlayer) CTreeNodePlayer is a CTreeNode (see section 3.2.3) that has extra data which allows the node to keep track of its location in the Tree Display. A CTreeNodePlayer also contains the maximum y location of any of its children. This information is used to place new nodes below preexisting nodes. Lastly the CTreeNodePlayer object contains two flags, HIGH and REDRAW. HIGH is used to store whether the node should be drawn in high or low intensity (high intensity indicates that this node is being referred to by the current instruction). REDRAW is used to indicate if a node should be redrawn at all.

**CTreePlayer**: (CTree->CTreePlayer) CTreePlayer is a CTree (see section 3.2.3) with added functionality that enables it to parse a CT file, update the locations of the CTreeNodePlayer nodes within the CTree, and execute or undo any instruction that appear in a CT file. Additional data included in CTreePlayer
allows it to store two versions of the text in the CT file. One version is the raw file, while the other is the text that will be displayed in the Instruction Window.

**RayPathPlayer:** (RayPath->RayPathPlayer) RayPathPlayer is a RayPath object whose node traversal and creation functions will honor the CTree location information in CTreeNodePlayer objects.

**GraphicRenderer:** GraphicRenderer encapsulates all the functionality and information needed to create and support CTPlayer’s graphic user interface.

**main(…):** The main function of this program first creates an instance of the CTreePlayer object (the CTreePlayer is initialized with the name of the CT file that it should read), then creates an instance of GraphicRenderer (the GraphicRenderer is initialized with a pointer to the previously created CTreePlayer), then calls the GraphicRenderer member function that brings up the user interface.

The code for this application can be found in Appendix F.
Chapter 8

Conclusion

The hope behind all concurrent algorithms is that by parallelizing the computations of a given algorithm, the running time of that algorithm will decrease proportionally to the number of concurrent processes executing the algorithm. The reality is that in most cases the overhead involved in synchronizing and maintaining the concurrent processes takes away from the benefits of concurrent computation.

The runtime analysis discussed in Chapter 6 showed that the overhead involved in maintaining the data structures needed to implement the D1raytrace and DRraytrace algorithms, and the time needed to synchronize the executions of these algorithms' concurrent processes, consumed a considerable portion of their computation times.

One optimization that may help decrease the computational overhead of these concurrent algorithms is replacing the dynamic memory used by the D1raytrace implementation with static memory that is allocated once and reused throughout the execution of the algorithm. Another possible optimization is to either decrease the overall usage of mutual exclusion objects in the concurrent algorithms, or reimplement the mutual exclusion objects using a more efficient mutual exclusion algorithm (Lynch describes a number of mutual exclusion algorithms in [7]).

These and other modifications must be made in order to completely take advantage of the independence of computation that is inherent in the ray tracing algorithm.
Appendix A

Profile Statistics for TraceProcRecursive and TraceProcIncremental

Here are the raw profile statistics discussed in Chapter 6.2. These statistics were taken during successive runs of the Braytrace, DRraytrace and DIraytrace algorithms using the same scene definition, and 2 concurrent processes (for the concurrent algorithms). The image being rendered was of size 100 by 100. These tests were run on a Silicon Graphics symmetric multiprocessing machine.

The data in the table can be read as follows: Column 1 indicates the number of samples taken in a particular function. Column 2 is the amount and percentage of time spent in that function. Column 3 is the accumulated time and percentage for the function on that line and all functions that had greater total times. The 4th column lists the procedure and file that the ps-sample data on that row pertains to.
A.1 Braytrace

Profile listing generated Wed May 22 09:55:58 1996
with: prof DTrace

samples time CPU FPU Clock N-cpu S-interval Counts size
2542 25s R4400 R4010 250.0MHz 0 10.0ms 0(bytes)

Each sample covers 4 bytes for every 10.0ms (0.04% of 25.4200sec)

Sorted in descending order by the number of samples in each procedure.
Unexecuted procedures are excluded.

samples time(%) cum time(%) procedure (file)
1467 15s(57.7) 15s(57.7) ivIntersectBbox__FRC7SbBox3fR7SbVec3fN22 (DTrace:iv/Intersect.C)
178 1.8s(7.0) 16s(64.7) fmax__FfT1 (DTrace:iv/MathUtils.C)
174 1.7s(6.8) 18s(71.6) fmin__FfT1 (DTrace:iv/MathUtils.C)
142 1.4s(5.6) 20s(77.1) ivIntersect__FR3RayP6OctreeR3Hit (DTrace:iv/Intersect.C)
90 0.9s(3.5) 21s(80.7) ivShade__FR3HitR7SbVec3fPUcif7SbVec3fP6 OctreeP9LightDataT4 (DTrace:iv/Shade.C)
65 0.65s(2.6) 21s(83.2) ivIntersectBbox__FRC7SbBox3fR7SbVec3fT2 (DTrace:iv/Intersect.C)
47 0.47s(1.8) 22s(85.1) __sqrtf (/usr/lib/libm.so:fsqrt.s)
45 0.45s(1.8) 22s(86.9) __mi__7SbVec3fCFv (/usr/lib/libInventor.so.2:SbVec.c++)
41 0.41s(1.6) 22s(88.5) ivIntersectCube__FP100bjectDataR7SbVec3fN22 (DTrace:iv/Intersect.C)
28 0.28s(1.1) 23s(89.6) isDerivedFrom__6SoTypeCF6SoType (/usr/lib/libInventor.so.2:SoType.c++)
21 0.21s(0.8) 23s(90.4) normalize__7SbVec3fFv (/usr/lib/libInventor.so.2:SbVec.c++)
19 0.19s(0.7) 23s(91.1) multVecMatrix__8SbMatrixCFRC7SbVec3fR7SbVec3f (/usr/lib/libInventor.so.2:SbMatrix.c++)
17 0.17s(0.7) 23s(91.8) length__7SbVec3fCFv (/usr/lib/libInventor.so.2:SbVec.c++)
17 0.17s(0.7) 24s(92.5) multDirMatrix__8SbMatrixCFRC7SbVec3fR7SbVec3f
12 0.12s (0.5) 24s (93.0) ivTrace__FR3Rayif7SbVec3fP6OctreeP9LightDataT2 (DTrace:iv/Trace.C)
12 0.12s (0.5) 24s (93.4) ivIntersectFace__FiP7SbVec3fR7SbVec4fP7SbVec4fR7SbVec3fN25 (DTrace:iv/Intersect.C)
12 0.12s (0.5) 24s (93.9) __pl__FRC7SbVec3fT1 (/usr/lib/libInventor.so.2:SbVec.c++)
12 0.12s (0.5) 24s (94.4) getParent__6SoTypeCFv (/usr/lib/libInventor.so.2:SoType.c++)
12 0.12s (0.5) 24s (94.8) __pow (/usr/lib/libm.so:pow.s)
11 0.11s (0.4) 24s (95.3) __ml__FRC7SbVec3ff (/usr/lib/libInventor.so.2:SbVec.c++)
8 0.08s (0.3) 24s (95.6) RayExt__FP100bjectDataiP9LightDataT2P10ViewVolume7SbVec2sPFiT17SbColor_v (DTrace:RayExt.C)
8 0.08s (0.3) 24s (95.9) __apl__7SbVec3f7SbVec3f7SbVec3f (/usr/lib/libInventor.so.2:SbVec.c++)
8 0.08s (0.3) 24s (96.2) dot__7SbVec3fCFRC7SbVec3f (/usr/lib/libInventor.so.2:SbVec.c++)
7 0.07s (0.3) 25s (96.5) TransmissionDirection__FR7SbVec3fT1RfT3T1 (DTrace:iv/Shade.C)
7 0.07s (0.3) 25s (96.8) mult__FRC7SbColorT1 (DTrace:iv/Shade.C)
6 0.06s (0.2) 25s (97.0) __amu__7SbVec3ff (/usr/lib/libInventor.so.2:SbVec.c++)
5 0.05s (0.2) 25s (97.2) ReflectionDirection__FR7SbVec3fN21 (DTrace:iv/Shade.C)
4 0.04s (0.2) 25s (97.4) ivIntersectPlane__FR7SbVec4fR7SbVec3fN22 (DTrace:iv/Intersect.C)
4 0.04s (0.2) 25s (97.5) distance__FR7SbVec4fR7SbVec3f (DTrace:iv/MathUtils.C)
3 0.03s (0.1) 25s (97.6) Matches (/usr/lib/libXt.so:GCManager.c)
3 0.03s (0.1) 25s (97.8) _smallloc (/usr/lib/libc.so.1:malloc.c)
3 0.03s (0.1) 25s (97.9) __write (/usr/lib/libc.so.1:write.s)
3 0.03s (0.1) 25s (98.0) __mi__FRC7SbVec3fT1 (/usr/lib/libInventor.so.2:SbVec.c++)
2 0.02s (0.1) 25s (98.1) isOfType__6SoBaseCF6SoType (/usr/lib/libInventor.so.2:SoBase.c++)
2 0.02s (0.1) 25s (98.2) __malloc (/usr/lib/libc.so.1:malloc.c)
2 0.02s (0.1) 25s (98.2) __close (/usr/lib/libc.so.1:close.s)
2 0.02s (0.1) 25s (98.3) __writev (/usr/lib/libc.so.1:writev.s)
2 0.02s (0.1) 25s (98.4) ivIntersectFaceSet__FP100bjectDataR7SbVec3fN22 (DTrace:iv/Intersect.C)
2 0.02s (0.1) 25s (98.5) _open (/usr/lib/libc.so.1:open.s)
2 0.02s (0.1) 25s (98.5) __fabsf (/usr/lib/libm.so:fabsf.s)
2542 25s(100.0) 25s(100.0) TOTAL
### A.2 TraceProcRecursive

Profile listing generated Wed May 22 07:45:37 1996

with: prof DTrace DTrace.mon.16298

<table>
<thead>
<tr>
<th>samples</th>
<th>time</th>
<th>CPU</th>
<th>FPU</th>
<th>Clock</th>
<th>N-cpu</th>
<th>S-interval</th>
<th>Counts</th>
<th>size</th>
</tr>
</thead>
<tbody>
<tr>
<td>1826</td>
<td>18s</td>
<td>R4400</td>
<td>R4010</td>
<td>250.0MHz</td>
<td>0</td>
<td>10.0ms</td>
<td>0(bytes)</td>
<td></td>
</tr>
</tbody>
</table>

Each sample covers 4 bytes for every 10.0ms (0.05% of 18.2600sec)

- procedures using pc-sampling.
  Sorted in descending order by the number of samples in each procedure.
  Unexecuted procedures are excluded.

<table>
<thead>
<tr>
<th>samples</th>
<th>time(%)</th>
<th>cum time(%)</th>
<th>procedure (file)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1175</td>
<td>12s(64.3)</td>
<td>12s(64.3)</td>
<td>ivIntersectBbox__FRC7SbBox3fr7SbVec3fN22 (DTrace:iv/Intersect.C)</td>
</tr>
<tr>
<td>118</td>
<td>1.2s(6.5)</td>
<td>13s(70.8)</td>
<td>fmax__FfT1 (DTrace:iv/MathUtils.C)</td>
</tr>
<tr>
<td>105</td>
<td>1.1s(5.8)</td>
<td>14s(76.6)</td>
<td>fmin__FfT1 (DTrace:iv/MathUtils.C)</td>
</tr>
<tr>
<td>86</td>
<td>0.86s(4.7)</td>
<td>15s(81.3)</td>
<td>ivShade__FR3HitR7SbVec3fFUCrIf7SbVec3fP6octreeP9LightDataT4 (DTrace:iv/Shade.C)</td>
</tr>
<tr>
<td>73</td>
<td>0.73s(4.0)</td>
<td>16s(85.3)</td>
<td>ivIntersect__FR3RayP6octreeR3Hit (DTrace:iv/Intersect.C)</td>
</tr>
<tr>
<td>32</td>
<td>0.32s(1.8)</td>
<td>16s(87.0)</td>
<td>ivIntersectBbox__FRC7SbBox3fr7SbVec3fT2 (DTrace:iv/Intersect.C)</td>
</tr>
<tr>
<td>23</td>
<td>0.23s(1.3)</td>
<td>16s(88.3)</td>
<td>__sqrte (/usr/lib/libm.so:sqrtf)</td>
</tr>
<tr>
<td>19</td>
<td>0.19s(1.0)</td>
<td>16s(89.3)</td>
<td>__mi__7SbVec3fCFv (/usr/lib/libInventor.so.2:SbVec.c++)</td>
</tr>
<tr>
<td>15</td>
<td>0.15s(0.8)</td>
<td>16s(90.1)</td>
<td>length__7SbVec3fCFv (/usr/lib/libInventor.so.2:SbVec.c++)</td>
</tr>
<tr>
<td>13</td>
<td>0.13s(0.7)</td>
<td>17s(90.9)</td>
<td>isDerivedFrom__6SoTypeCF6SoType (/usr/lib/libInventor.so.2:SoType.c++)</td>
</tr>
<tr>
<td>13</td>
<td>0.13s(0.7)</td>
<td>17s(91.6)</td>
<td>normalize__7SbVec3fFv (/usr/lib/libInventor.so.2:SbVec.c++)</td>
</tr>
<tr>
<td>10</td>
<td>0.1s(0.5)</td>
<td>17s(92.1)</td>
<td>is0fType__6SoBaseCF6SoType (/usr/lib/libInventor.so.2:SoBase.c++)</td>
</tr>
<tr>
<td>9</td>
<td>0.09s(0.5)</td>
<td>17s(92.6)</td>
<td>__ml__FRC7SbVec3ff</td>
</tr>
</tbody>
</table>
9 0.09s (0.5) 17s (93.1) TraceProcRecurse__FPv (DTrace:TraceProc.C)
9 0.09s (0.5) 17s (93.6) ivIntersectCube__FP100bjectDataR7SbVec3fN22 (DTrace:iv/Intersect.C)
8 0.08s (0.4) 17s (94.0) __free (/usr/lib/libc.so.1:malloc.c)
8 0.08s (0.4) 17s (94.5) TransmissionDirection__FR7SbVec3fT1RfT3T1 (DTrace:iv/Shade.C)
7 0.07s (0.4) 17s (94.9) realfree (/usr/lib/libc.so.1:malloc.c)
7 0.07s (0.4) 17s (95.2) ivIntersectFace__FiP7SbVec3fR7SbVec4fP7 SbVec4fR7SbVec3fN25 (DTrace:iv/Intersect.C)
6 0.06s (0.3) 17s (95.6) ivTrace__FR3Rayif7SbVec3fP6OctreeP9 LightDataT2 (DTrace:iv/Trace.C)
6 0.06s (0.3) 18s (95.9) __pow (/usr/lib/libm.so:pow.s)
6 0.06s (0.3) 18s (96.2) mult__FRC7SbColorT1 (DTrace:iv/Shade.C)
5 0.05s (0.3) 18s (96.5) get_entry__711mutexFv (DTrace:llmutex.C)
5 0.05s (0.3) 18s (96.8) multVecMatrix__8SbMatrixCFRC7SbVec3fR7SbVec3f (DTrace:iv/Intersect.C)
4 0.04s (0.2) 18s (97.0) distance__FR7SbVec4fR7SbVec3f (DTrace:iv/MathUtils.C)
4 0.04s (0.2) 18s (97.2) ivIntersectFaceSet__FP100bjectDataR7 SbVec3fN22 (DTrace:iv/Intersect.C)
4 0.04s (0.2) 18s (97.4) __amu__7SbVec3f (DTrace:iv/Intersect.C)
4 0.04s (0.2) 18s (97.6) __pl__FRC7SbVec3fT1 (DTrace:iv/Intersect.C)
4 0.04s (0.2) 18s (97.9) getParent__6SoTypeCFv (DTrace:iv/Intersect.C)
4 0.04s (0.2) 18s (98.1) __dt__17ConvergenceObjectFv (DTrace:ConvObj.C)
3 0.03s (0.2) 18s (98.2) DISTsetpixel__FiT17SbColor (DTrace:RayExt.C)
3 0.03s (0.2) 18s (98.4) ReflectionDirection__FR7SbVec3fN21 (DTrace:iv/Shade.C)
3 0.03s (0.2) 18s (98.6) multDirMatrix__8SbMatrixCFRC7SbVec3fR7SbVec3f (DTrace:iv/Intersect.C)
3 0.03s (0.2) 18s (98.7) dot__7SbVec3fCFRC7SbVec3f (DTrace:iv/Intersect.C)
2 0.02s (0.1) 18s (98.8) __apl__7SbVec3f (DTrace:iv/Intersect.C)
2 0.02s (0.1) 18s (99.0) __sginap (/usr/lib/libc.so.1:sginap.s)
2 0.02s (0.1) 18s (99.1) getTypeId__6SoCubeCFv (DTrace:iv/Intersect.C)
2 0.02s (0.1) 18s (99.2) __spin_lock (/usr/lib/libc.so.1:libmutexs.s)
2 0.02s (0.1) 18s (99.3) SetRadiance__20supConvergenceObjectF7SbVec3f (DTrace:iv/Intersect.C)
1 0.01s (0.1) 18s (99.3) __powf (/usr/lib/libm.so:fpow.c)

1826 18s (100.0) 18s (100.0) TOTAL
A.3 TraceProcIncremental

Profile listing generated Wed May 22 08:11:58 1996
with: prof DTrace DTrace.mon.16351

samples time CPU FPU Clock N-cpu S-interval Counts size
2509 25s R4400 R4010 250.0MHz 0 10.0ms 0(bytes)

Each sample covers 4 bytes for every 10.0ms (0.04% of 25.0900sec)

Sorted in descending order by the number of samples in each procedure.
Unexecuted procedures are excluded.

samples time(%) cum time(%) procedure (file)
1154 12s( 46.0) 12s( 46.0) ivIntersectBbox__FRC7SbBox3fR7SbVec3fW22
          (DTrace:iv/Intersect.C)
126 1.3s( 5.0) 13s( 51.0) _r4kmp_setlock (/usr/lib/libc.so.1:r4k.s)
109 1.1s( 4.3) 14s( 55.4) ivIntersect__FR3RayP60cmtreeR3Hit
           (DTrace:iv/Intersect.C)
109 1.1s( 4.3) 15s( 59.7) fmax__fT1 (DTrace:iv/MathUtils.C)
96 0.96s( 3.8) 16s( 63.5) fmin__fT1 (DTrace:iv/MathUtils.C)
85 0.85s( 3.4) 17s( 66.9) __realfree (/usr/lib/libc.so.1:malloc.c)
69 0.69s( 2.8) 17s( 69.7) TraceProcIncremental__FPv (DTrace:TraceProc.C)
62 0.62s( 2.5) 18s( 72.1) __malloc (/usr/lib/libc.so.1:malloc.c)
55 0.55s( 2.2) 19s( 74.3) ivShade__FR3HitR7SbVec3fFUcif7SbVec3fP6
                  Octree9LightDataT4 (DTrace:iv/Shade.C)
44 0.44s( 1.8) 19s( 76.1) t_splay (/usr/lib/libc.so.1:malloc.c)
42 0.42s( 1.7) 20s( 77.8) ivIntersectBbox__FRC7SbBox3fR7SbVec3fT2
                  (DTrace:iv/Intersect.C)
33 0.33s( 1.3) 20s( 79.1) __free (/usr/lib/libc.so.1:malloc.c)
30 0.3s( 1.2) 20s( 80.3) __mi__7SbVec3fCFv
                  (/usr/lib/libInventor.so.2:SbVec.c++)
27 0.27s( 1.1) 20s( 81.3) __spin_lock (/usr/lib/libc.so.1:libmutexs.s)
22 0.22s( 0.9) 21s( 82.2) cleanfree (/usr/lib/libc.so.1:malloc.c)
20 0.2s( 0.8) 21s( 83.0) ivIntersectCube__FP100bjectDataR7SbVec3fW22
                  (DTrace:iv/Intersect.C)
<table>
<thead>
<tr>
<th>Time</th>
<th>Function</th>
<th>File</th>
</tr>
</thead>
<tbody>
<tr>
<td>20 s</td>
<td>0.2s( 0.8) __sqrtf (/usr/lib/libm.so:fsqrt.s)</td>
<td></td>
</tr>
<tr>
<td>20 s</td>
<td>0.2s( 0.8) isDerivedFrom__6SoTypeCF6SoType</td>
<td>(/usr/lib/libInventor.so.2:SoType.c++)</td>
</tr>
<tr>
<td>19 s</td>
<td>0.19s( 0.8) Converge__5CTreeFP7RayPathP7SbVec3f</td>
<td>(DTrace:CTree.C)</td>
</tr>
<tr>
<td>18 s</td>
<td>0.18s( 0.7) length__7SbVec3fCFv</td>
<td>(/usr/lib/libInventor.so.2:SbVec.c++)</td>
</tr>
<tr>
<td>16 s</td>
<td>0.16s( 0.6) t_delete (/usr/lib/libc.so.1:malloc.c)</td>
<td></td>
</tr>
<tr>
<td>14 s</td>
<td>0.14s( 0.6) get_entry__711mutexFv (DTrace:11mutex.C)</td>
<td></td>
</tr>
<tr>
<td>14 s</td>
<td>0.14s( 0.6) normalize__7SbVec3fFv</td>
<td>(/usr/lib/libInventor.so.2:SbVec.c++)</td>
</tr>
<tr>
<td>13 s</td>
<td>0.13s( 0.5) _sginap (/usr/lib/libc.so.1:sginap.s)</td>
<td></td>
</tr>
<tr>
<td>12 s</td>
<td>0.12s( 0.5) _malloc (/usr/lib/libc.so.1:malloc.c)</td>
<td></td>
</tr>
<tr>
<td>12 s</td>
<td>0.12s( 0.5) multVecMatrix__8SbMatrixCFRC7SbVec3fr7SbVec3f</td>
<td>(/usr/lib/libInventor.so.2:SbMatrix.c++)</td>
</tr>
<tr>
<td>11 s</td>
<td>0.11s( 0.4) _smalloc (/usr/lib/libc.so.1:malloc.c)</td>
<td></td>
</tr>
<tr>
<td>11 s</td>
<td>0.11s( 0.4) _free (/usr/lib/libc.so.1:malloc.c)</td>
<td></td>
</tr>
<tr>
<td>11 s</td>
<td>0.11s( 0.4) multDirMatrix__8SbMatrixCFRC7SbVec3fr7SbVec3f</td>
<td>(/usr/lib/libInventor.so.2:SbMatrix.c++)</td>
</tr>
<tr>
<td>10 s</td>
<td>0.1s( 0.4) SetLocal__5CTreeFP7RayPathR7SbVec3fi</td>
<td>(DTrace:CTree.C)</td>
</tr>
<tr>
<td>10 s</td>
<td>0.1s( 0.4) mult__FRC7SbColorT1 (DTrace:iv/Shade.C)</td>
<td></td>
</tr>
<tr>
<td>9 s</td>
<td>0.09s( 0.4) __ct__9CTreeNodeFv (DTrace:CTree.C)</td>
<td></td>
</tr>
<tr>
<td>8 s</td>
<td>0.08s( 0.3) __pl__FRC7SbVec3fF1</td>
<td>(/usr/lib/libInventor.so.2:SbVec.c++)</td>
</tr>
<tr>
<td>8 s</td>
<td>0.08s( 0.3) _dl__FPv (/usr/lib/libC.so:../new/_delete.c++)</td>
<td></td>
</tr>
<tr>
<td>7 s</td>
<td>0.07s( 0.3) dot__7SbVec3fCFRC7SbVec3f</td>
<td>(/usr/lib/libInventor.so.2:SbVec.c++)</td>
</tr>
<tr>
<td>7 s</td>
<td>0.07s( 0.3) getLink__7RayPathFP7RayPath</td>
<td>(DTrace:CTree.C)</td>
</tr>
<tr>
<td>7 s</td>
<td>0.07s( 0.3) __ct__7RayPathFP7RayPath (DTrace:CTree.C)</td>
<td></td>
</tr>
<tr>
<td>2509</td>
<td>25s(100.0) TOTAL</td>
<td></td>
</tr>
</tbody>
</table>
Appendix B

File Formats

B.1 CTrec Specification File

CTrec Specification Files are text files that are created by the user if the user wants to save a log of the creation and convergence of a particular CTree during the calculation of the Radiance of a pixel in the output image. The format of the file is as follows.

<number of CTrees to record>
<x1> <y1>
<x2> <y2>
...
<xn> <yn>

For example:

3
3 5
10 7
7 25

The example above tells the algorithm to record the activity for the convergence trees associated with the 3 pixel locations <3,5> <10,7> and <7,25>.

Notes and Bugs: Be aware that CTrec files and the -CT command line option are ignored if either Supersampling or recursive Ray Tracing is enabled.
B.2 CT Log File format

CT Log files are the output file that is generated from logging the construction and convergence of a CTree object. The format of the CT file is as follows:

\[ <x,y> \]
\[ <pid> \text{COMMAND COMMANDargs ||PATH} \]
\[ <pid> \text{COMMAND COMMANDargs ||PATH} \]
\[ <pid> \text{COMMAND COMMANDargs ||PATH} \]
\[ \ldots \]
\[ \ldots \]

**Final Result:** \(<\text{Red}> <\text{Green}> <\text{Blue}>\)

The \(<x,y>\) pair on the first line is the location of the pixel to which the log file pertains. The \(<pid>\) on the next and subsequent lines refers to the processor id of the process that performed the command on that line.

**COMMAND** and **COMMANDargs** refer to one of the following operations that a process could have performed.

**SetLocal** \(<\text{Red}> <\text{Green}> <\text{Blue}>\): Set the local contribution in the node indicated by the path (described below) to the values in the 3 arguments.

**SetWeightScale** \(<\text{Weight}> <\text{Scale}>\): Set the weight and scale at the node indicated by the path to the values in the 2 arguments.

**SetSPD** \(<\text{Red}> <\text{Green}> <\text{Blue}>\): Set the value of the spectral distribution mask in the node indicated by the path to the values in the 3 arguments.

**SetReflection** \(<\text{Red}> <\text{Green}> <\text{Blue}>\): Set the reflection contribution of the node indicated by the path to the values in the 3 arguments.

**CSetReflection** \(<\text{Red}> <\text{Green}> <\text{Blue}>\): This is the same as SetReflection except that it is used in the case where the reflection was set during convergence of the tree.

**SetRefraction** \(<\text{SPD#}> <\text{Red}> <\text{Green}> <\text{Blue}>\): Set the refraction contribution of the node indicated by the path to the values in the 3 arguments. The \(<\text{SPD#}>\) argument indicates which of the 3 Spectral Distribution contributions is being referred to by the command. Correct values for \(<\text{SPD#}>\) are 0, 1 and 2.
CSetRefraction <SPD#> <Red> <Green> <Blue>: This is the same as SetRefraction except that it is used in the case where the refraction was set during convergence of the tree.

A path specification follows the command and the command arguments on each line. "||" indicates the root of the tree. Each number after "||" indicates the direction that should be taken at a particular level in the tree in order to reach the desired node. Valid numbers are 2, 3, 4 and 5. 2 means to go to the current node’s reflected child. 3, 4, and 5 mean to go to the current node’s red, green or blue refractive child.

For example “|| 2 4 3 5”, would mean go to the root’s reflected child, then to the reflected child’s green refracted child, then to that node’s red refracted child, then finally to the blue refracted child.

B.3 Eye definition file

If the -eye <eyefile> command line option is enabled, then DTrace will read the indicated eye definition file. This file has the following format:

```
<x,y,z>
<dx,dy,dz>
<xUP,yUP,zUP>
```

The first three values indicate the location of the viewer in the scene, the second three indicate the direction that the viewer is facing, and the last three indicate the orientation of the viewer.

The orientation of the viewer refers to what the up vector of the viewer is. (e.g. if the viewer is standing upright the up vector is <0,1,0>, while if the viewer is standing on their head it is <0,-1,0>)
Appendix C

DTrace: Makefile, main() and Initialization Functions

# Makefile for DTrace
#

# C++ sources
NewC++SRCS =
RayExt.C\nTraceProc.C\CTree.C\nsupCTree.C

OldC++SRCS=
iv/RayTraceTypes.C \ iv/Camera.C

# iv/RayTrace.C

altC++SRCS = \ ConvObj.C \ lock.C \ CTree.C \ CTreePlayer.C \ Renderer.C \ CTreePlayerMain.C

C++SRCS = $(oldC++SRCS) $(NewC++SRCS)

# C sources
CSRCS=

# all sources
ALLSRCS=$(C++SRCS) $(CSRCS)

# header files
HDRS=${C++SRCS:.C=.h} ${CSRCS:.c=.h}
altHDRS = ${altC++SRCS:.C=.h}

# objects
OBJS=${C++SRCS:.C=.o} ${CSRCS:.c=.o}
MyOBJS=${NewC++SRCS:.C=.o}
altOBJS=${altC++SRCS:.C=.o}

# app
TARGET=DTrace
altTARGET=CTPlayer

# debugging
DEBUG=
CCDEBUG=

TARGET_COMPILER=$(CC)
TARGET_COMPILER_FLAGS=$(DEBUG)
altTARGET_COMPILER_FLAGS=$(altDEBUG)

# Put any libs you use here. These will be included on the link line.
LIBS=-nostdlib -L/lib -L/usr/lib -L./ivwraplib -L./ivwidgets \ -lInventorXt -lGL -lGLw -livwrap -lInventorWidget
altLIBS=-nostdlib -L/lib -L/usr/lib -lInventorXt -lXaw -lXt -lX11

# Put any include directories (-I...) here
INC=-nostdinc -I/usr/include/CC -I/usr/include

# Put any defines or misc. C compiler flags here.
CDEF=-prototypes -Xcpluscomm -xansi -fullwarn
# Put any defines or misc. C++ compiler flags here.
altCDEF=-xansi -fullwarn -use_cfront
CCDEF=-xansi -fullwarn

CFLAGS=$(INC) $(CDEF) $(DEBUG)
CCFLAGS=$(INC) $(CCDEF) $(CCDEBUG)
altCCFLAGS=$(INC) $(altCDEF) $(altDEBUG)
CC=/usr/bin/CC
cc=/usr/bin/cc

# should be first target
default:
$(MAKE) $(TARGET)

# optimized executable
opt:
make $(TARGET) \
    CFLAGS="$(INC) $(CDEF) -O -DNDEBUG" \
    CCFLAGS="$(INC) $(CCDEF) -O -DNDEBUG"

# make executable
$(TARGET): $(OBJ)
$(TARGET_COMPILER) -o $@ $(TARGET_COMPILER_FLAGS) \$
    $(OBJ)\$
    $(LIBS)
@echo $@ is made.

$(altTARGET): $(altOBJ)
$(TARGET_COMPILER) -o $@ $(altTARGET_COMPILER_FLAGS) \$$(altOBJ)\$
    $(altLIBS)
@echo $@ is made.

cleanme:
rm -f $(MyOBJ)
altcleanme:
rm -f $(altOBJS)

# clean up cruft
clean:
rm -f $(OBJS) \\ 
  *.slo *.o *~ *.s \\ 
  *.a *..c TAGS \\ 
  core a.out /*.*.bak *.BAK *.CKP \\ 
  *.Addr *.Counts *.pixie ./**

# clean up everything
clobber: clean
rm -f $(TARGET)

# known file types
.SUFFIXES: .c .C

# make a .o from a .c (C file)
.c.o:
$(cc) $(CFLAGS) -c $< -o $@

# make a .o from a .C (C++ file)
.C.o:
$(CC) $(CCFLAGS) -c $< -o $@

/* DTraceMain.C

Scene Viewer, modified to OpenInventor ray tracer
*/

// base include
#include <sys/types.h>
#include <sys/resource.h>
#include <sys/prctl.h>
#include <stdio.h>

// X includes
#include <X11/intrinsic.h>

// Inventor includes
#include <Inventor/SoDB.h>
#include <Inventor/Xt/SoXt.h>
#include <Inventor/nodes/SoSelection.h>
#include <Inventor/nodes/SoCamera.h>
#include <Inventor/sensors/SoFieldSensor.h>
#include <Inventor/sensors/SoTimerSensor.h>
// local includes
#include "ivwraplib/SoSceneViewer.h"
#include "iv/RayTraceWrap.h" // for callbacks
#include "iv/Camera.h" // for callbacks
#include "lineUI.h" // line mode entrance

extern int ivIntersectBboxn;

// Load the CTreeRec file
static SbBool LoadCTreeRecConfig(char *fn)
{
  FILE *f = fopen(fn, "r");
  if (!f) {
    fprintf(stderr, "Error Opening CTreeRecConfig file %s\n", fn);
    return FALSE;
  }

  int *size = &RayTrace::CTreeRecSize;
  fscanf(f, "%d", size);

  if ((*size < 1) || (*size > 10)) return FALSE;

  for (int k = 0; (k < *size) && (!feof(f)) && (!ferror(f)); k++)
    fscanf(f, "%d %d", (RayTrace::CTreeRecIndices + k * 2),
           (RayTrace::CTreeRecIndices + k * 2 + 1));

  if ((k == --*size) && (!ferror(f))) {
    fclose(f);
    return TRUE;
  } else return FALSE;
}

// process command line arguments
// set envFile, filename, RayTrace:: statics
static SbBool getArgs(int argc, char **argv, char *&envFile, char *&filename,
                       char *&scriptFile)
{
  envFile = NULL;
  filename = NULL;
  scriptFile = NULL;

  for (int k = 1; k < argc; k++) {
    if (!strcmp (argv[k], "-help")) return FALSE;
    if (!strcmp (argv[k], "-b")) { // -batch mode
      RayTrace::batchmode = TRUE;
      fprintf(stderr, "batch mode not yet implemented.\n");
      return FALSE;
    } else if (!strcmp(argv[k], "-script")) { // -script (mode)
      RayTrace::script = TRUE;
      if (k == argc - 1)
        return FALSE;
  }
scriptFile = argv[++k];
if (!strcmp(argv[k+1], "-c")) {
    k++;
    RayTrace::continuous = TRUE;
}
}
else if ( !strcmp(argv[k], "-read") ) {
    // - read ( mode )
    RayTrace::read = TRUE;
    if ( k == argc - 1 )
        return FALSE;
    scriptFile = argv[++k];
    if (!strcmp(argv[k+1], "-i") ) {
        // - i ( interval )
        k += 2;
        RayTrace::continuous = TRUE;
        RayTrace::interval = atof(argv[k]);
        if (!strcmp(argv[k+1], "-c") ) {
            k++;
        }
    }
    if (!strcmp(argv[k+1], "-c") ) {
        // - c ( ontinuous )
        k++;
        RayTrace::continuous = TRUE;
        if (!strcmp(argv[k+1], "-c") ) {
            // - c ( ontinuous )
            k++;
            RayTrace::continuous = TRUE;
            if (!strcmp(argv[k+1], "-c") ) {
                // - c ( ontinuous )
                k++;
                RayTrace::continuous = TRUE;
            }
        }
    }
}
else if ( !strcmp(argv[k], "--mode") ) {
    // - m ( ode )
    if (!strcmp(argv[++k], "Raytrace") ) {
        RayTrace::mode = TRUE;
    }
}
else if ( !strcmp(argv[k], "--old") )
    RayTrace::NewTrace = FALSE;
else if ( !strcmp(argv[k], "--file base") ) {
    // - f ( ile base )
    RayTrace::basename = argv[++k];
}
else if ( !strcmp(argv[k], "--output") ) {
    // - o ( utput )
    RayTrace::out = TRUE;
}
else if ( !strcmp ( argv[k], "--maxdepth") ) {
    // - d maxdepth
    if ( k == argc - 1 )
        return FALSE;
    if ( (RayTrace::maxdepth = atoi( argv[++k] )) < 1 )
        return FALSE;
}
else if ( !strcmp ( argv[k], "--envfile" ) ) {
    // - e envfile
    if ( k == argc - 1 )
        return FALSE;
    envFile = argv[++k];
}
else if ( !strcmp ( argv[k], "--eye" ) ) {
    // - e yel oc and dir
    if ( k > argc - 1 )
return FALSE;
RayTrace::OverRideEye = TRUE;
RayTrace::CamFile = argv[++k];
}
else if (!strcmp ( argv[k], "-j") ) { // -j(itter samples)
if ( k == argc - 1)
    return FALSE;
if ( (RayTrace::ssr = atoi ( argv[++k] )) < 1 ) // N
    return FALSE;
RayTrace::jitter = TRUE;
}
else if ( !strcmp ( argv[k], "-p") ) { // -p N (supersample)
if ( k == argc - 1)
    return FALSE;
if ( (RayTrace::ssg = atoi ( argv[++k] )) < 1)
    return FALSE;
RayTrace::ssr = RayTrace::ssg * RayTrace::ssg; // N^2
}
else if ( !strcmp ( argv[k], "-s") ) { // - s shademode
if ( k == argc - 1)
    return FALSE;
switch ( toupper(argv[++k][0]) )
{
    case 'F':
        RayTrace::shademode = SHADE_FLAT; break;
    case 'N': RayTrace::shademode = SHADE_NORMAL; break;
    case 'A': RayTrace::shademode = SHADE_AMBIENT; break;
    case 'L': RayTrace::shademode = SHADE_LOCAL; break;
    case 'D': RayTrace::shademode = SHADE_DIRECT; break;
    case 'R': RayTrace::shademode = SHADE_RECURSIVE; break;
    default : return FALSE;
}
}
else if ( !strcmp ( argv[k], "-w") ) { // - w minrayweight
if ( k == argc - 1)
    return FALSE;
if ( ( RayTrace::minweight = atof ( argv[++k] ) ) < 0. )
    return FALSE;
}
else if ( !strcmp ( argv[k], "-D") ) { // -D debuglevel
if ( k == argc - 1)
    return FALSE;
RayTrace::debug = atoi ( argv[++k] );
}
else if ( !strcmp ( argv[k], "-del") ) // -del
    RayTrace::Incremental = TRUE;
else if ( !strcmp ( argv[k], "-n") ) { // - n nprocesses
if ( k == argc - 1)
    return FALSE;
RayTrace::nProcesses = atoi ( argv[++k] );
if (RayTrace::nProcesses < 1) return FALSE;
}
else if ( !strcmp ( argv[k], "--CT") ) { // ctree config file name
if (k == argc - 1) return FALSE;
if (!LoadCTreeRecConfig(argv[++k])) return FALSE;
}
else if (!strcmp (argv[k], "-size")) { // viewport size
  if (k == (argc - 2)) return FALSE;
  int x, y;
  if ((x = atoi ( argv[++k] )) < 1 ) return FALSE;
  else if ((y = atoi ( argv[++k] )) < 1 ) return FALSE;
  SbViewportRegion region((short) x, (short) y);
  RayTrace::winSize = region.getViewportSizePixels();
}
else if (!strcmp (argv[k], "-UI")) { // UI mode
  if ( k == argc - 1 )
    return FALSE;
  if (!strcmp (argv[++k], "line")) // line mode
    RayTrace::UImode = RTT_LINE_MODE;
  else if (!strcmp(argv[k], "gl")) // gl display mode
    RayTrace::UImode = RTT_GL_MODE;
  else return FALSE;
}
else {
  // error if more than one
  if ( filename )
    return FALSE;
  // assume arg is iv filename
  filename = argv[k];
}

// args seem ok
if (((RayTrace::ssr>1) && (RayTrace::CTreeRecSize>0)) { 210
  fprintf(stderr,"You cannot record trees while supper sampling!\n");
  return FALSE;
}

return TRUE;
}

static
void DisplayUsage() {
  fprintf( stderr, "OLD OPTIONS: \n");
  fprintf( stderr, "\t[-script <file name> [-c]]\n" );
  fprintf( stderr, "\t[-read <file name> [-i N] [-c]]\n" );
  fprintf( stderr, "\t[-m rendermode]\n" );
  fprintf( stderr, "\t\tInventor\n" );
  fprintf( stderr, "\t\tRaytrace\n" );
  fprintf( stderr, "\t[-f <base file name>]\n" );
  fprintf( stderr, "\t[-o]\n" );
  fprintf( stderr, "\t[-d maxraydepth >= 0]\n" );
  fprintf( stderr, "\t[-e environ.iv]\n" );
  fprintf( stderr, "\t[-p N (cast N x N rays/pixel) >= 1]\n" );
  fprintf( stderr, "\t[-eye <filename>]\n" );
  fprintf( stderr, "\t[-j(itter samples)]\n" );
  fprintf( stderr, "\t[-f <base file name>]\n" );
  fprintf( stderr, "\t[-o]\n" );
  fprintf( stderr, "\t[-d maxraydepth >= 0]\n" );
  fprintf( stderr, "\t[-e environ.iv]\n" );
  fprintf( stderr, "\t[-p N (cast N x N rays/pixel) >= 1]\n" );
  fprintf( stderr, "\t[-eye <filename>]\n" );

82
fprintf(stderr, \"\t[-a shademode (\tFlat\n\n\thing{\nfprintf(stderr, \"\t\t\tNormal\n\thing{\n fprintf(stderr, \"\t\t\tAmbient\n\thing{\n fprintf(stderr, \"\t\t\tLocal\n\thing{\n fprintf(stderr, \"\t\t\tDirect\n\thing{\n fprintf(stderr, \"\t\t\tRecursive)\n\thing{\n\r\n fprintf(stderr, \"\t\-w minrayweight >=0\n\thing{\n fprintf(stderr, \"\t\-D debuglevel >=0\n\thing{\n fprintf(stderr, \"\t\-d (- for stdin)\n\thing{\n NEW OPTIONS:\n\thing{\n fprintf(stderr, \"\t\-del\) (delegate computations)\n\thing{\n fprintf(stderr, \"\t[-n nprocesses] (nprocess >= 1; default = 2)\n\thing{\n fprintf(stderr, \"\t[-UI <line | gl>\n\thing{\n fprintf(stderr, \"\t[-size w h]\{in glmode just resize the window,\n\thing{\n fprintf(stderr, \"\t\t\totherwise default = 50x50\n\thing{\n fprintf(stderr, \"\t[file.iv (- for stdin)\n\thing{\n fprintf(stderr, \"\t[-CT <file name]\n\thing{\n fprintf(stderr, \"\t[-old] (use the old raytracer tracer)\n\thing{\n fprintf(stderr, \"\t[-help]\n\thing{\n void main(int argc, char **argv)
\thing{\n SoSceneViewer *sv;
 Solnput in;
 char *envFile, *filename, *scriptFile;
 Widget mainWindow;

 // parse arguments, complain if bad usage
 if (! getArgs(argc, argv, envFile, filename, scriptFile)) {
 fprintf(stderr, \"usage: \%s\n", argv[0]);
 DisplayUsage();
 exit(1);
 }

 // open script file to write
 if ((RayTrace::script) &&
 (RayTrace::scriptFILE = fopen(scriptFile, \"w\") == NULL) {
 fprintf(stderr, \"error opening: \%s\n", scriptFile);
 exit(0);
 }

 // open script file to read
 if ((RayTrace::read) &&
 (RayTrace::scriptFILE = fopen(scriptFile, \"r\") == NULL) {
 fprintf(stderr, \"error opening: \%s\n", scriptFile);
 exit(0);
 }

 if (RayTrace::UImode == RTT_GL_MODE)
 // workaround for bug 200909 -- this will force OpenGL to create
 // a connection with the X server to receive delete window events
 // (so that OpenGL can delete Accumulation and other software

83
buffers when the window is destroyed).
if (putenv("GL_CHECK_WINDOW_DESTROY=y"))
    fprintf(stderr, "Couldn't set GL_CHECK_WINDOW_DESTROY env.\n");

// init Inventor Database
SoDB::init();

// create selection node as scene graph root
SoSelection *selRoot = new SoSelection;

// read inventor file
SbBool doRead = FALSE;
// if filename is "-"; read from stdin
if ( filename && !strcmp(filename,"-")) {
    doRead = TRUE;
    in.setFilePointer(stdin);
}
else if (filename) {
    doRead = in.openFile(filename);
}

selRoot->ref();

if (doRead) {
    fprintf(stderr, "Reading input file... ");
    SoNode *n;
    while ((SoDB::read(&in, n) != FALSE) && (n != NULL))
        selRoot->addChild(n);

    fprintf(stderr, selRoot->getNumChildren() ?
        "done.\n": "No data read; creating empty scene.\n");
}

// initialize inventor window
mainWindow = SoXt::init(argv[0]);

if (RayTrace::UImode == RTT_GL_MODE) {
    // install ray trace buttons and callbacks
    installSVbuttonCB ( "DR", "RayTrace",
                        RayTraceWrap::DRRayTrace );
    installSVbuttonCB ( "DI", "RayTrace",
                        RayTraceWrap::DIRayTrace );
    installSVbuttonCB ( "B", "OriginalRayTrace",
                        RayTraceWrap::BRayTrace );
    if (RayTrace::script)
        installSVbuttonCB ( "R\nE\nC", "RecordCameraData",
                            Camera::recordCB );
    if (RayTrace::read)
        installSVbuttonCB ( "P\nL\nA\nY", "PlayCameraData",
                            Camera::playCB );
}

// Create (modified) Scene Viewer
sv = new SoSceneViewer( mainWindow, NULL, TRUE, selRoot, envFile );
selRoot->unref();

if (RayTrace::UImode == RTT_LINE_MODE) {
    DTraceLineMode(sv);
}  else if (RayTrace::UImode == RTT_GL_MODE) {
    // Continuous record
    if ((RayTrace::script) & (RayTrace::continuous)) {
        SoCamera *cam = sv->getCamera();
        SoFieldSensor *mySensor1 = new SoFieldSensor(Camera::recordCB1, sv);
        SoFieldSensor *mySensor2 = new SoFieldSensor(Camera::recordCB1, sv);
        mySensor1->attach(&cam->position);
        mySensor2->attach(&cam->orientation);
    }

    // Build and show (modified) SceneViewer
    sv->show();
    XtRealizeWidget( mainWindow );

    // Continuous play
    if (((RayTrace::read) & (RayTrace::continuous)) ||
        ((RayTrace::read) & (RayTrace::out))) {
        SoCamera *cam = sv->getCamera();
        SoTimerSensor *mySensor = new SoTimerSensor(Camera::playCB1, sv);
        mySensor->setInterval(RayTrace::interval);
        mySensor->schedule();
    }

    // Loop until done
    SoXt::mainLoop();
}
fprintf(stderr, "ivIntersectBBox called %d times.\n", ivIntersectBbboxn);
fprintf(stderr,"DONE!\n");
exit(1);
}

// RayTraceTypes.h
#define _RT_TYPES_
#include <stdio.h>
#include <Inventor/SbBasic.h>
#include <Inventor/SbLinear.h>
// tag for function parameters
#define OUTPUT  // no-op

// tags for display params
#define RTT_GLMODE 0
#define RTT_LINE_MODE 1
#define RTT_X_MODE 2

// typedef boolean, point, vector, color, radiance, 4x4 matrix
typedef SbBool      Binary;
typedef SbVec3f     Point;
typedef Point      *PointP;
typedef SbVec2f     Vector2D;
typedef SbVec3f     Vector;
typedef Vector     *VectorP;
typedef SbVec4f     Plane;
typedef Plane      *PlaneP;
typedef SbColor     Color;
typedef SbVec3f     Radiance;
typedef SbBox3f     Bbox;
typedef Bbox       *BboxP;
typedef void       *VoidP;
typedef SbMatrix    Matrix;

// supported shade modes (could add other shaders here)
enum ShadeMode   {  // controls ivShade()'s lighting model
    SHADE_NORMAL, // use surface normal as color
    SHADE_FLAT,   // use material diffuse color
    SHADE_AMBIENT, // ambient term only
    SHADE_LOCAL,  // AMBIENT + diffuse, specular
    SHADE_DIRECT, // LOCAL with shadow rays
    SHADE_RECURSIVE // spawn reflection, refraction rays
    };

// supported light types (could add area lights later)
enum LightType   {  // affects ivShade()'s lighting model
    INFINITE_LIGHT, // iv light with direction only
    LOCAL_LIGHT, // iv light with location only
    SPOT_LIGHT    // iv light with location, direction
    };

struct ObjectData {
    SoShape *object;  // An Inventor Scene Object.
    Matrix modelMat;  // OS points -> WS
    Matrix modelMatInv;  // WS points -> OS
    Matrix modelMatTrans;  // WS normals -> OS
    Matrix modelMatInvTrans;  // OS normals -> WS

    // Environment for ambient illumination, fog (eta), et cetera.
    SoEnvironment *environment;

    // Current coords for SoIndexedFaceSet and SoIndexedNurbsSurface
    SoCoordinate3 *coordinate3;
    SoCoordinate4 *coordinate4;

    // these fields are nonzero iff object is face set
    int nfaces;  // number of faces
    Plane *fplanes;  // plane of the kth face
    int *flengths;  // # of vertices in kth face
    Point **fvertices;  // vertices of kth face
    Plane **fedgeplanes;  // edge planes of kth face

    // Material information.
    Color ambient;
    Color diffuse;
    Color specular;
    Color emission;
    float shininess;
    float transparency;
    Vector eta;  // index of refraction at lambda = r,g,b
    float kd, ks;  // coefficients of diffuse, specular reflection

    // optimization fields follow

    // bounding box info, for accelerating ray-casting
    Bbox bboxOS;  // axial bounding box in object coords.
    Bbox bboxWS;  // axial bounding box in world coords.

    // unique id of last ray to be checked against this object
    long generation;
};

struct LightData {
    SoLight *light;  // the light as an OpenInventor object
    LightType lighttype;  // INFINITE_LIGHT, LOCAL_LIGHT, SPOT_LIGHT
    Point locationWS;  // light's WS location, if POINT/SPOT
    Vector directionWS;  // light's WS direction, if INFINITE/SPOT
    Matrix modelMat;  // OS points -> WS
    Matrix modelMatInv;  // WS points -> OS
Matrix modelMatTrans;  // WS normals -> OS
Matrix modelMatInvTrans; // OS normals -> WS

// Viewing frustum, along with camera orientation specifier.
struct ViewVolume {
    Point eye; // Eye position, in WS
    Vector viewDir; // normalized View direction, in WS
    Vector upDir; // normalized Up vector, in WS
    float aspectRatio; // width / height on near clipping plane
    float nearDistance; // Distance to near clipping plane.
    float farDistance; // Distance to far clipping plane.
    float fovy; // Vertical field of view, in radians.
    SbRotation orientation; // Required by certain iv constructors.
};

// a ray
class Ray {
    private:
        static long uid; // unique ray ids
    public:
        Ray( void )
        { generation = uid++;
            object = 0;  }
        Ray( Point &R, Vector &D)
        { origWS = R; dirWS = D;
            generation = uid++;;
            object = 0;  }
        Ray( Point &R, Vector &D, unsigned char r, unsigned char g, unsigned char b )
        { origWS = R; dirWS = D;
            generation = uid++;;
            spd[0] = r; spd[1] = g; spd[2] = b;
            object = 0;  }
        Ray(Point &R, Vector &D, ObjectData *O) { origWS = R;
            dirWS = D; generation = uid++; object = O; }
        static int NumRaysCast( void ) { return uid; }

        Point origWS; // WS ray origin
        Vector dirWS; // WS ray direction, normalized
        long generation; // unique generation number
        unsigned char spd[3]; // spectral distribution mask
        ObjectData *object; // medium ray is in, or NULL
    };

// a hit
class Hit {
    public:

Point surfpWS; // WS point of intersection
Vector surfhWS; // WS surface normal at point
int entering;  // 1 if entering, 0 if leaving
ObjectData *object; // medium ray is in, or NULL
// CCC -- add statistics for hits, misses, depth, etc.
};

// RayTrace control parameters

class RayTrace {

public:
// public class statics
static int debug;    // Set by raytraceDebugCB.
static ShadeMode shademode; // used by ivShade();
static int maxdepth;  // spawn no ray deeper than maxdepth
static float minweight; // spawn no ray weighted by < minweight
static int ssg;       // cast (ssg X ssg) rays per pixel
static int ssr;       // ssr = (ssg * ssg)
static Binary jitter; // use jittered sampling
static Binary batchmode; // run offscreen (not yet implemented)
static Binary script;  // write camera data to file
static Binary read;    // read camera data from file
static Binary continuous; // default to single step mode
static float interval; // interval between steps in seconds
static FILE *scriptFILE; // FILE * to use for scripting
static Binary mode;    // Inventor rendering mode
static char *basename; // output base file name
static Binary out; // don't output
static int UImode; // line mode, glwindow mode,
// simple X display mode

static SbVec2s winSize;      // if needed this will hold the size of the
    // window
static int Incremental;      // use ctrees and delegate work
static int nProcesses;       // the number of tprocs to create
static int NewTrace;         // non 0 when using distributed tracing
static Binary OverRideEye;   // eye data given on commandline
static char *CamFile; // eye rotation
static int CTreeRecSize; // the number of indices
static int CTreeRecIndices[20]; /* The indices for which RayExt should
        use CTreeRec as the convergence tree */
};
#endif /* !_RT_TYPES */

// RayTraceTypes.C

#include "RayTraceTypes.h"

// initializers for Ray:: class statics

89
long Ray::uid = 0;

// initializers (default values) for RayTrace:: class statics
int RayTrace::debug = 1; // debug/verbose level
int RayTrace::maxdepth = 4; // bounds raytree recursion
float RayTrace::minweight = 0.01; // no rays with weight < minweight
ShadeMode RayTrace::shademode = SHADE_RECURSIVE;
int RayTrace::ssg = 1; // trace (ssg X ssg) rays/pixel
int RayTrace::ssr = 1; // ssr = (ssg * ssg)
Binary RayTrace::jitter = FALSE; // use jittered sampling
Binary RayTrace::batchmode = FALSE; // run offscreen renderer
Binary RayTrace::script = FALSE; // don't write out camera data
Binary RayTrace::read = FALSE; // don't read in camera data
Binary RayTrace::continuous = FALSE; // default to single step mode
float RayTrace::interval = 1.0; // interval between steps in seconds
FILE *RayTrace::scriptFILE = NULL; // null file pointer
Binary RayTrace::mode = FALSE; // Inventor rendering mode
char *RayTrace::basename = NULL; // output base file name
Binary RayTrace::out = FALSE; // don't output
int RayTrace::Incremental = FALSE; // use ctrees and delegate work
int RayTrace::nProcesses = 2; // the number of tprocs to create
int RayTrace::Ulmode = RTT_LINE_MODE; // default to linemode
SbVec2s RayTrace::winSize(50,50); // default winsize if not in glmode
int RayTrace::NewTrace = 1; // default to newtrace
Binary RayTrace::OverRideEye = FALSE;
char * RayTrace::CamFile = NULL;

int RayTrace::CTreeRecIndices[20];
int RayTrace::CTreeRecSize = 0;

// RayTraceWrap.h

#ifndef _RTRAYTRACEWRAP
#define _RTRAYTRACEWRAP

// X includes
#include <Xm/Xm.h>

// inventor includes
#include <Inventor/nodes/SoSeparator.h>
#include <Inventor/SbColor.h>
#include <Inventor/SbLinear.h>
#include <Inventor/actions/SoCallbackAction.h>
#include <Inventor/nodes/SoEnvironment.h>
#include <Inventor/nodes/SoCoordinate3.h>
#include <Inventor/nodes/SoCoordinate4.h>

// openGL includes
#include <GL/gl.h>

// local include
#include "RayTraceTypes.h"
void
testParameters(ObjectData *objects, int nObjects,
   LightData *lights, int nLights,
   ViewVolume *vVolume,
   SbVec2s vpSize);

// Wrapper for user-supplied ray-tracing functions.
class RayTraceWrap {

public:
static void RayTraceWrap::saveWindow(unsigned long *pixels, int width,
   int height, char *fName, int inspect);

// Called when Scene Viewer RTD, RT buttons are pressed.
static void raytraceCB(Widget,XtPointer,XtPointer);
static void raytraceDebugCB(Widget,XtPointer,XtPointer);
static void BRayTrace(Widget,XtPointer,XtPointer);
static void DRRayTrace(Widget,XtPointer,XtPointer);
static void DIRayTrace(Widget,XtPointer,XtPointer);

// write pixel in immediate, delayed framebuffer mode
static void setPixellmmed(int x,int y,SbColor color);
static void setPixelDelayed(int x,int y,SbColor color);

private:
static GLuint ABGRtoRGBA(GLuint abgr) {
   return (((abgr >> 0) & 0xff) << 24) // R
   (((abgr >> 8) & 0xff) << 16) // G
   (((abgr >> 16) & 0xff) << 8) // B
   (((abgr >> 24) & 0xff) << 0); // A
}

// Find all the objects and lights in sceneGraph. Puts results in
// objectData and lightData. Allocates memory that must be freed.
static void getObjectsAndLights(SoSeparator *sceneGraph);

// Free up the allocated memory from getobjArrayAndlightArray.
static void freeObjectsAndLights();

// Count nodes of different types callbacks. userData will be a pointer to
// an integer that will be incremented.
static SoCallbackAction::Response
countObjectsCB(void *userData,SoCallbackAction *action,const SoNode *node);
static SoCallbackAction::Response
countLightsCB(void *userData,SoCallbackAction *action,const SoNode *node);

// Add objects, lights to objectData, lightData. Used by getObjectsAndLights().
static SoCallbackAction::Response
addObjectsCB(void *userData,SoCallbackAction *action,const SoNode *node);
static SoCallbackAction::Response
addLightsCB(void *userData,SoCallbackAction *action,const SoNode *node);

// private class statics
static SbVec2s viewportSize; // viewport w, h
static GLuint *pixelData;               // framebuffer store

static ObjectData *objectData;
static int numObjects;               // Number of elements in objectData
static LightData *lightData;
static int numLights;                // Number of elements in lightData
static SoEnvironment *currentEnvironment;     // Used only by addObjectsCB.
static SoCoordinate3 *currentCoordinate3;   // Used only by addObjectsCB.
static SoCoordinate4 *currentCoordinate4;   // Used only by addObjectsCB.
static float currentkd;               // coefficient of diffuse reflection
static float currentks;              // coefficient of specular reflection

#endif /* !_RT_RAYTRACE_WRAP_ */

/* RayTraceWrap.C
wrapper code by Steve Hardt 10/95
substantial modification Seth 10/95

references:
OIC++RM
Coplien, Advanced C++ etc.
Siggraph '88 Course Notes, Heckbert
*/

// system include files
#include <assert.h>
#include <math.h>
#include <iostream.h>   // for strstreams
#include <iomanip.h>
#include <strstream.h>
#include <fstream.h>
#include <string.h>
#include <sys/types.h>  // for getpid()
#include <unistd.h>

// X include files
#include <Xm/Xm.h>

// OpenGL include files
#include <GL/gl.h>
#include <GL/glx.h>
#include <GL/GLwDrawA.h>

// Inventor include files
#include <Inventor/Xt/SoXt.h>
#include <Inventor/nodes/SoSeparator.h>
#include <Inventor/nodes/SoSelection.h>
#include <Inventor/SbColor.h>

#include <Inventor/nodes/SoCamera.h>
#include <Inventor/SbLinear.h>
#include <Inventor/SbBox.h>
#include <Inventor/SbString.h>
#include <Inventor/SbViewportRegion.h>
#include <Inventor/SoPickedPoint.h>
#include <Inventor/nodes/SoSelection.h>
#include <Inventor/actions/SoCallbackAction.h>
#include <Inventor/actions/SoGLRenderAction.h>
#include <Inventor/actions/SoRayPickAction.h>
#include <Inventor/actions/SoGetBoundingBoxAction.h>
#include <Inventor/nodes/SoShape.h>
#include <Inventor/nodes/SoCube.h>
#include <Inventor/nodes/SoSphere.h>
#include <Inventor/nodes/SoCone.h>
#include <Inventor/nodes/SoCylinder.h>
#include <Inventor/nodes/SoLight.h>
#include <Inventor/nodes/SoDirectionalLight.h>
#include <Inventor/nodes/SoPointLight.h>
#include <Inventor/nodes/SoSpotLight.h>
#include <Inventor/nodes/SoPerspectiveCamera.h>
#include <Inventor/nodes/SoTransform.h>
#include <Inventor/nodes/SoMaterial.h>
#include <Inventor/nodes/SoTransformSeparator.h>
#include <Inventor/nodes/SoBaseColor.h>
#include <Inventor/nodes/SoEnvironment.h>
#include <Inventor/nodes/SoCoordinate3.h>
#include <Inventor/nodes/SoCoordinate4.h>
#include <Inventor/nodes/SoFaceSet.h>
#include <Inventor/nodes/SoIndexedFaceSet.h>
#include <Inventor/nodes/SoIndexedNurbsSurface.h>
#include <Inventor/nodes/SoLabel.h>
#include <Inventor/fields/SoSFName.h>

// local include files
#include "./ivwraplib/SoSceneViewer.h"
#include "./ivwraplib/SaveWindow.h"
#include "MathUtils.h"
#include "RayTraceTypes.h"
#include "RayTraceWrap.h"
#include "./RayExt.h"

// prototypes at file scope

static SoSeparator *
makeSceneGraph(ObjectData *objectData, int nObjects,
               LightData *lightData, int nLights,
               ViewVolume *vVolume);

/*
the following code comprises the “wrapper” around
your ray tracer which will manage its interaction with
inventor, opengl, the window system, etc. you should
not need to modify it; however, you may wish to read
it for a deeper understanding of the application.
*/
void RayTraceWrap::saveWindow(unsigned long *pixels, int width, int height, char *fName, int inspect)
{
    static char fname[128];
    static int fnum = 0;
    ostrstream filename( fname, sizeof fname);
    if (fName)
    {
        filename << fName << (int)getpid() << "."; filename.fill( '0' ); filename.width( 4 );
        filename << fnum++ << ".rgb" << ends;
    }
    else
    {
        // construct unique filename
        filename << "ivray" << (int)getpid() << ".";
        filename.fill( '0' ); filename.width( 4 );
        filename << fnum++ << ".rgb" << ends;
    }

    // write pixels as .rgb image file 'filename'
    savebuffer( fname, pixels, width, height );

    // display saved image with ipaste, scope, etc.
    if ( inspect )
    {
        char cbuffer[128], *pwd;
        ostrstream command( cbuffer, sizeof cbuffer );
        if( pwd = getenv( "PWD" ) )
            command << "cd " << pwd << " ; \n\t\t";
        command << "/usr/sbin/ipaste " << fname << ends;
        cout << "saveWindow(): executing " << cbuffer << " " << endl;
        system( cbuffer );
    }
}

// construct new scene graph with specified
// objects, lights, and camera (from vVolume)
static SoSeparator *
makeSceneGraph(ObjectData *objectData, int nObjects,
               LightData *lightData, int nLights,
               ViewVolume *vVolume)
{
    // Create root of scene graph.
    SoSeparator *sep = new SoSeparator;
    sep->ref();

    // Create perspective camera from given ViewVolume.

SoPerspectiveCamera *cam = new SoPerspectiveCamera;
cam->position = vVolume->eye;
cam->orientation = vVolume->orientation;
cam->aspectRatio = vVolume->aspectRatio;
cam->nearDistance = vVolume->nearDistance;
cam->farDistance = vVolume->farDistance;
cam->heightAngle = vVolume->fovy;

// Add camera.
sep->addChild(cam);

// Add lights.
for (int l = 0; l < nLights; l++) {
    // Separate the transform for this light from the others.
    SoTransformSeparator *localSep = new SoTransformSeparator;
    sep->addChild(localSep);

    // Add transform.
    SoTransform *trans = new SoTransform;
    trans->setMatrix(lightData[l].modelMat);
    localSep->addChild(trans);

    // Add light source.
    localSep->addChild(lightData[l].light);
}

// Add objects, each with a flattened zform.
for (int ob = 0; ob < nObjects; ob++) {
    // Separate the transform for this object from the others.
    SoTransformSeparator *localSep = new SoTransformSeparator;
    sep->addChild(localSep);

    // material
    SoMaterial *mat = new SoMaterial;
    mat->ambientColor = objectData[ob].ambient;
    mat->diffuseColor = objectData[ob].diffuse;
    mat->specularColor = objectData[ob].specular;
    mat->emissiveColor = objectData[ob].emission;
    mat->shininess = objectData[ob].shininess;
    mat->transparency = objectData[ob].transparency;
    localSep->addChild(mat);

    // environment
    localSep->addChild(objectData[ob].environment);

    // transform
    SoTransform *trans = new SoTransform;
    trans->setMatrix(objectData[ob].modelMat);
    localSep->addChild(trans);

    // coordinates
    if (objectData[ob].coordinate3)
        localSep->addChild(objectData[ob].coordinate3);
    else if (objectData[ob].coordinate4)
localSep->addChild(objectData[ob].coordinate4);

// the object itself.
localSep->addChild(objectData[ob].object);
}

// return scene graph
return sep;

// Test ObjectData, LightData by rendering them as SceneGraph.
void
testParameters(ObjectData *objects, int nObjects,
    LightData *lights, int nLights,
    ViewVolume *vVolume,
    SbVec2s vpSize )
{
    // Pack objects, lights, and camera (derived from view volume) into a scene graph.
    SoSeparator *sep = makeSceneGraph ( objects, nObjects, lights, nLights, vVolume );

    // Create render action and apply it to scene graph.
    SoGLRenderAction render( SbViewportRegion( vpSize ), TRUE );
    render.apply( sep );
}

void RayTraceWrap::BRayTrace(Widget w,XtPointer data,XtPointer p)
{
    RayTrace::NewTrace = 0;
    RayTraceWrap::raytraceCB(w,data,p);
}

void RayTraceWrap::DRRayTrace(Widget w,XtPointer data,XtPointer p)
{
    RayTrace::Incremental = 0;
    RayTrace::NewTrace = 1;
    RayTraceWrap::raytraceCB(w,data,p);
}

void RayTraceWrap::DIRayTrace(Widget w,XtPointer data,XtPointer p)
{
    RayTrace::Incremental = 1;
    RayTrace::NewTrace = 1;
    RayTraceWrap::raytraceCB(w,data,p);
}

/*
raytraceCB() is called when any button are
pushed. You should not have to modify this function.
However, you may wish to examine it to understand how
it uses inventor state, and modifies openGL state,
to prepare for the invocation of your ray tracer.
*/
void RayTraceWrap::raytraceDebugCB(Widget, XtPointer data, XtPointer)
{
    cout << "RayTraceWrap::raytraceDebugCB\n";
    cout.flush();
    SoSceneViewer *sViewer = (SoSceneViewer *)data;

    // Make sure camera is perspective, not orthographic.
    assert(sViewer->getCamera());
    if (!sViewer->getCamera()->isOfType(SoPerspectiveCamera::getClassTypeId()))
    {
        cerr << "Error: Ray-tracing is enabled only for perspective cameras."
             << endl
             << "The current camera setting defines orthographic projection."
             << endl
             << "The frustum icon in the examiner-viewer toggles camera type." << endl;
        return;
    }

    // Create a view volume from the current camera.
    SoPerspectiveCamera *cam = (SoPerspectiveCamera *)sViewer->getCamera();
    ViewVolume vVolume;
    vVolume.eye = cam->position.getValue();

    // Get viewport size (width, height).
    assert(sViewer->getCurrentViewer());
    SbViewportRegion region =
        sViewer->getCurrentViewer()->getViewportRegion();
    viewportSize = region.getViewportSizePixels();

    // Get viewDir by applying camera orientation to (0,0,-1): -z axis.
    // For (inadequate) documentation, see SoCamera (OIC++RM p. 133).
    cam->orientation.getValue().multVec(SbVec3f(0,0,-1), vVolume.viewDir);

    // Get upDir by applying camera orientation to (0,1,0): +y axis.
    cam->orientation.getValue().multVec(SbVec3f(0,1,0), vVolume.upDir);

    // inventor camera reports aspect ratio 1, even with non-square viewport
    // also, it does not appear to be changed by resizing; docs are unclear
    // rather than doing vVolume.aspectRatio = cam->aspectRatio.getValue();
    // we derive aspect ratio directly from viewport width / height
    vVolume.aspectRatio = viewportSize[0] / (float)viewportSize[1];

    // fill in remaining view volume attributes
    vVolume.nearDistance = cam->nearDistance.getValue();
    vVolume.farDistance = cam->farDistance.getValue();
    vVolume.fovy = cam->heightAngle.getValue();
    vVolume.orientation = cam->orientation.getValue();

    // Move objects, lights into objectData, lightData arrays.
    SoSeparator *sceneGraph = sViewer->getSceneGraph();
    assert( sceneGraph );

    // Allocate & fill object, light arrays (must be released later).
    getObjectsAndLights( sceneGraph );

    // Force drawing to normal window, not overlay window.
GLwDrawingAreaMakeCurrent(sViewer->getNormalWidget(),
    sViewer->getNormalContext());

    // Save relevant openGL state.
    GLint mMode, dBuffer;
    glGetIntegerv(GL_MATRIX_MODE,&mMode);
    glGetIntegerv(GL_DRAW_BUFFER,&dBuffer);
    GLfloat clearColor[4];
    glGetFloatv(GL_COLOR_CLEAR_VALUE,clearColor);
    glDrawBuffer(GL_FRONT);

    // clear depth buffer
    glClearColor(0,0,0,0);
    glClear(GL_DEPTH_BUFFER_BIT);

    // set testParams to 1 to debug param passing;
    // set to 0 for normal (raytrace callback) use
    int testParams = 0;
    if (!testParams) {
        glOrtho(0,viewportSize[0],0,viewportSize[1],-1,1);

        // Allocate pixel array pixelData.
        pixelData = new GLuint[viewportSize[0] * viewportSize[1]];
        assert(pixelData);

        // Effect immediate or delayed pixel drawing.
        // ivRayTrace scene into pixelData array.
        RayExt(objectData, numObjects, lightData, numLights,
            &vVolume, viewportSize, RayTrace::debug ?
            setPixelImmed : setPixelDelayed);

        // Draw pixelData to active viewport.
        glRasterPos2i( 0, 0);
        glDrawPixels(viewportSize[0], viewportSize[1], GL_RGBA,
            GL_UNSIGNED_BYTE, pixelData);

        // save pixeldata to unique .rgb file
        RayTraceWrap::saveWindow((unsigned long *)pixelData, viewportSize[0],
            viewportSize[1], RayTrace::basename,
            RayTrace::debug);

        // Release pixel array.
        delete [] pixelData;
        pixelData = 0;

        glClear(GL_DEPTH_BUFFER_BIT);
    }

    else {
        // Test flattened object and light arrays by rendering them as Inventor scene graph.
        testParameters( objectData, numObjects, lightData, numLights, &vVolume, viewportSize );
    }

    // Restore openGL state.
    glClearColor(clearColor[0], clearColor[1], clearColor[2], clearColor[3]);
glMatrixMode(mMode);
glDrawBuffer(dBuffer);
glFlush();

// Release node arrays.
freeObjectsAndLights();
}

// Set debug to 0; call raytraceDebugCB; restore debug
void
RayTraceWrap::raytraceCB(Widget widget, XtPointer data, XtPointer cbs)
{
    cout << "no debug ";
    int savedebug = RayTrace::debug;
    RayTrace::debug = 0;
    raytraceDebugCB( widget, data, cbs );
    RayTrace::debug = savedebug;
}

// Write pixel to framebuffer for display via Swapbuffers().
void
RayTraceWrap::setPixelDelayed(int x, int y, SbColor color)
{
    // convert float vector to packed rgba format
    GLuint rgba = ABGRtoRGBA(color.getPackedValue());

    // store in our local framebuffer
    pixelData[x + y * viewportSize[0]] = rgba;

    // if reached end of raster, write raster to window
    if ( x == viewportSize[0] - 1 ) {
        glRasterPos2i( 0, y );
        glDrawPixels( viewportSize[0], 1, GLRGBA, GL_UNSIGNED_BYTE, &pixelData[y * viewportSize[0]] );
    }
}

// Draw pixel to front buffer in immediate mode.
void
RayTraceWrap::setPixelImmed(int x, int y, SbColor color)
{
    // convert float vector to packed rgba format
    GLuint rgba = ABGRtoRGBA(color.getPackedValue());

    // store in our local framebuffer
    pixelData[x + y * viewportSize[0]] = rgba;

    // write pixel to window
    glRasterPos2i( x, y );
    glDrawPixels( 1, 1, GL_RGBA, GL_UNSIGNED_BYTE, &rgba );

    // wait for pipeline to finish
void RayTraceWrap::getObjectsAndLights(SoSeparator *sep)
{
    numObjects = 0;
    numLights = 0;

    // Count number of objects and lights in order to size arrays.
    SoCallbackAction count;
    int maxObjects = 0;
    int maxLights = 0;
    count.addPreCallback(SoShape::getClassTypeId(), countObjectsCB, &maxObjects);
    count.addPreCallback(SoLight::getClassTypeId(), countLightsCB, &maxLights);
    count.apply(sep);

    // Allocate memory.
    objectData = new ObjectData[maxObjects];
    lightData = new LightData[maxLights];

    // Initialize data needed by the calls to addObjectsCB.
    currentEnvironment = 0;
    currentCoordinate3 = 0;
    currentCoordinate4 = 0;

    // Put objects and lights into ObjectData and LightData.
    // The add*CB routines increment numObjects, numLights.
    SoCallbackAction add;
    add.addPreCallback(SoNode::getClassTypeId(), addObjectsCB, NULL);
    add.addPreCallback(SoLight::getClassTypeId(), addLightsCB, NULL);
    add.apply(sep);

    // Check that we added only the counted objects and lights.
    assert( numObjects == maxObjects && numLights == maxLights );
}

// Release objectData, lightData storage.
void RayTraceWrap::freeObjectsAndLights()
{
    delete[] objectData;
    delete[] lightData;
}

// Called only on SoShape nodes.
// Increments passed counter.
SoCallbackAction::Response RayTraceWrap::countObjectsCB(void *userData, SoCallbackAction *,
const SoNode *)
{
  int *val = (int *)userData;
  (*val)++;

  return SoCallbackAction::CONTINUE;
}

// Called only on SoLight nodes.
// Increments passed counter.
SoCallbackAction::Response
RayTraceWrap::countLightsCB(void *userData, SoCallbackAction *,
const SoNode *node)
{
  // Only count lights that are on.
  if (((SoLight *)node)->on.getValue()) {
    int *val = (int *)userData;
    (*val)++;
  }

  return SoCallbackAction::CONTINUE;
}

// Called only on SoLight nodes.
// Copies away lightdata to array.
SoCallbackAction::Response
RayTraceWrap::addLightsCB(void *, SoCallbackAction *action,
const SoNode *node)
{
  // Only add viewer-enabled lights.
  if (((SoLight *)node)->on.getValue()) {
    LightData *light = &lightData[numLights];

    // get inventor light
    light->light = (SoLight *)node;

    // get OS to WS, WS to OS transformation matrices
    light->modelMat = action->getModelMatrix();  // matrix M
    light->modelMatInv = light->modelMat.inverse();  // WS points -> OS
    light->modelMatTrans = light->modelMat.transpose();  // WS normals -> OS
    light->modelMatInvTrans = light->modelMatInv.transpose();  // OS normals -> WS

    if (light->light->isOfType(SoDirectionalLight::getClassTypeId())) {
      SoDirectionalLight *dlight = (SoDirectionalLight *)node;

light->lighttype = INFINITE_LIGHT;
// infinite light; transform direction into world coords
light->modelMat.multDirMatrix (dlight->direction.getValue(), light->directionWS);
}
else if (light->light->isOfType( SoPointLight::getClassTypeId())) {
  SoPointLight *plight = (SoPointLight *)node;
  light->lighttype = LOCAL_LIGHT;
  // local light; transform location into world coords
  light->modelMat.multVecMatrix (plight->location.getValue(), light->locationWS);
}
else if (light->light->isOfType( SoSpotLight::getClassTypeId())) {
  SoSpotLight *slight = (SoSpotLight *)node;
  light->lighttype = SPOT_LIGHT;
  // spot light; transform location, direction into world coords
  light->modelMat.multVecMatrix (slight->location.getValue(), light->locationWS);
  light->modelMat.multDirMatrix (slight->direction.getValue(), light->directionWS);
}
numLights++;
}
return SoCallbackAction::CONTINUE;
}

// Called on all nodes, not just SoShape.
SoCallbackAction::Response
RayTraceWrap::addObjectsCB(void *, SoCallbackAction *action, const SoNode *node) {

  if (node->isOfType(SoShape::getClassTypeId())) {
    // Fill in data in objectData list.

    // Scene Viewer provides an environment.
    assert(currentEnvironment);
    ObjectData *obj = &objectData[numObjects];

    // get inventor object
    obj->object = (SoShape *)node;

    // get OS to WS, WS to OS transformation matrices
    obj->modelMat = action->getModelMatrix(); // matrix M
    obj->modelMatInv = obj->modelMat.inverse(); // matrix M^{-1}
    obj->modelMatTrans = obj->modelMat.transpose(); // matrix M^{T}
    obj->modelMatInvTrans = obj->modelMatInv.transpose(); // matrix (M^{-1})^{T}

    // get object material
    action->getMaterial(obj->ambient, obj->diffuse, obj->specular,

obj->emission, obj->shininess, obj->transparency);

// non-zero only for indexed face sets
obj->nf = 0;
obj->f = 0;
obj->fplanes = 0;
obj->fvertices = 0;
obj->fedgeplanes = 0;

// initialize bounding box to be vacuous
obj->bboxOS = Bbox( HUGE, HUGE, HUGE, -HUGE, -HUGE, -HUGE);

// if node is an Indexed face set, pull out faces & verts, and
// set bounding box, because inventor bbox code seems broken; it
// apparently does not take Coordinate{3,4} nodes into account
if (node->isOfType(SoIndexedFaceSet::getClassTypeId())) {
  SoIndexedFaceSet *faceset = (SoIndexedFaceSet *) (obj->object);

  // count faces
  obj->nf = 0;
  for (int nv = 0; nv < faceset->coordIndex.getNum(); nv++)
    if (faceset->coordIndex[nv] == -1)
      obj->nf++;

  // allocate array of lengths
  obj->f = new int[obj->nf];
  assert (obj->f);

  // allocate array of vertex pointers
  obj->fvertices = new PointP[obj->nf];
  assert (obj->fvertices);

  // allocate array of planes
  obj->fplanes = new Plane[obj->nf];
  assert (obj->fplanes);

  // allocate array of plane pointers
  obj->fedgeplanes = new PlaneP[obj->nf];
  assert (obj->fedgeplanes);

  // copy away OS face normal, vertices, edge planes
  int fid = 0;
  int nverts = 0;
  assert (currentCoordinate3);
  SoCoordinate3 *coords = (SoCoordinate3 *) (currentCoordinate3);
  for (int cid = 0; cid < faceset->coordIndex.getNum(); cid++) {
    // for each coord index
    if (faceset->coordIndex[cid] == -1) { // end of face
      // save away face length
      obj->f[fid] = nverts;

    // allocate space for, and copy, vertices
    obj->fvertices[fid] = new Point[nverts];
    for (int vid = cid - nverts; vid < cid; vid++)
      obj->fvertices[fid][vid - cid + nverts] = coords->point(faceset->coordIndex[vid]);
// compute face plane from vertices 0,1,2
if ( !plane_from_three_points ( obj->fvertices(fid)[0], obj->fvertices(fid)[1],
    obj->fvertices(fid)[2], obj->fplanes(fid) ) ) {
    fprintf ( stderr, "something's wrong; can't find plane for indexed face!\n" );
    assert( 0 );
}

// extract face normal from face plane
Vector fnormal(obj->fplanes(fid)[0],
    obj->fplanes(fid)[1], obj->fplanes(fid)[2]);

// allocate space for, and compute, edge planes
obj->fedgeplanes[fid] = new Plane[nverts];
Plane *plist = obj->fedgeplanes[fid];

// compute and store edge planes
int lag = obj->flengths[fid] - 1;
for ( int lead = 0; lead < obj->flengths[fid]; lead++ ) {
    if ( !plane_from_point_perpendicular ( obj->fvertices(fid)[lead],
        obj->fvertices(fid)[lead] - obj->fvertices(fid)[lag],
        fnormal, plist[lead] ) ) {
        fprintf ( stderr, "something's wrong; can't compute edge plane!\n" );
        assert( 0 );
    }
    lag = lead;
}

fid++; nverts = 0;
}
else {
    // expand bounding box to contain this vertex
    obj->bboxOS.extendBy ( coords->point[faceset->coordIndex[cid]] );
    // count the vertex
    nverts++;
}
} // for cid

else if ( node->isOfType ( SoIndexedNurbsSurface::getClassTypeId() ) ) {
    // similarly, hand-compute bounding box for nurb (e.g., bezier) patches
    SoIndexedNurbsSurface *nurb = (SoIndexedNurbsSurface *)obj->object;
    // check whether the nurb uses 3D or homogeneous 3D points
    assert ( currentCoordinate3 || currentCoordinate4 );
    SoCoordinate3 *coords3 = (SoCoordinate3 *)(currentCoordinate3);
    SoCoordinate4 *coords4 = (SoCoordinate4 *)(currentCoordinate4);
    for ( int cid = 0; cid < nurb->coordIndex.getNum(); cid++ ) { // for each index
        assert ( nurb->coordIndex[cid] != -1 );
        if ( coords3 )
            obj->bboxOS.extendBy ( coords3->point[nurb->coordIndex[cid]] );
        else {
            SbVec4f ph = coords4->point[nurb->coordIndex[cid]];
            // this fails for infinite control points -- cross your fingers!
obj->bboxOS.extendBy ( projp );
}
} // for
} // if nurb
else { // compute OS bbox using inventor machinery
static SoSelection *sroot = 0;

if ( !sroot ) {
    sroot = new SoSelection;
    sroot->ref();
}

// get object space bounding box
SbViewportRegion dummyvpr( 100, 100 );
SoGetBoundingBoxBoxAction bboxAction ( dummyvpr );
sroot->addChild( (SoNode *)node );
bboxAction.apply( sroot );
sroot->removeChild( (SoNode *)node );

// get OS bounding box from bbox action
obj->bboxOS = bboxAction.getBoundingBox();
}

// expand bounding box to handle planar, axial entities
// this is a hack; but a conservative one. that is, it will
// never cause a ray that should hit the object to miss it
Point bbmin = obj->bboxOS.getMin();
Point bbmax = obj->bboxOS.getMax();

if ( bbmax[0] <= bbmin[0] ) {
    float xavg = 0.5 * ( bbmax[0] + bbmin[0] );
    bbmin[0] = xavg - 0.1;
    bbmax[0] = xavg + 0.1;
}

    float yavg = 0.5 * ( bbmax[1] + bbmin[1] );
    bbmin[1] = yavg - 0.1;
    bbmax[1] = yavg + 0.1;
}

    float zavg = 0.5 * ( bbmax[2] + bbmin[2] );
    bbmin[2] = zavg - 0.1;
    bbmax[2] = zavg + 0.1;
}

// rebuild OS bbox from modified points
obj->bboxOS = Bbox (bbmin, bbmax);

// generate WS bbox from OS bbox, xform
obj->bboxWS = obj->bboxOS;
obj->bboxWS.transform ( obj->modelMat );

// Most recent env, coord, normal nodes encountered
obj->environment = currentEnvironment;
obj->coordinate3 = currentCoordinate3;
obj->coordinate4 = currentCoordinate4;

// coefficients of diffuse, specular reflection
obj->kd = currentkd;
obj->ks = currentks;

// encode object's index of refraction as fogcolor
// this complicates modeling fog effects in r/tracer...
// coincidentally, iv default color for fog is (1,1,1)
obj->eta = obj->environment->fogColor.getValue();
cout << "eta: (" << obj->eta[0] << " " << obj->eta[1] << " "
<< obj->eta[2] << ");" << endl;

// Warning messages.
if (node->isOfType(SoIndexedFaceSet::getClassTypeId))
    if ( !currentCoordinate3 ) {
        cerr << "Warning: There is a FaceSet or IndexedFaceSet with no " "preceding Coordinate3 node." << endl;
    }
    numObjects++;
}
else if (node->isOfType(SoEnvironment::getClassTypeId)) {
    // Set currentEnvironment.
    currentEnvironment = (SoEnvironment *)node;
}
else if (node->isOfType(SoCoordinate3::getClassTypeId())) {
    // Set currentCoordinate3.
    currentCoordinate3 = (SoCoordinate3 *)node;
    currentCoordinate4 = 0;
}
else if (node->isOfType(SoCoordinate4::getClassTypeId())) {
    // Set currentCoordinate4.
    currentCoordinate4 = (SoCoordinate4 *)node;
    currentCoordinate3 = 0;
}
else if (node->isOfType(SoLabel::getClassTypeId())) {
    SoLabel *label = (SoLabel *)node;
    SoSFName sfname = label->label;
    SbName name = sfname.getValue();
    const char *str = name.getString(); // whew!

    // parse kd, ks from node label
    // expect "kd %f" or "ks %s"
    if ( str && (strlen(str) > 3) && str[0] == 'k' ) {
            if ( str[1] == 'd' )
                currentkd = atof ( &str[3] );
            else
                currentks = atof ( &str[3] );
        }
    }
else
    cerr << "Error: expecting 'ks %f' or 'kd %s' in Label";
}

return SoCallbackAction::CONTINUE;
}

// Initialization of class statics.

// for use inside inventor wrapper
GLuint *RayTraceWrap::pixelData = 0;
SbVec2s RayTraceWrap::viewportSize;
SoEnvironment *RayTraceWrap::currentEnvironment;
SoCoordinate3 *RayTraceWrap::currentCoordinate3;
SoCoordinate4 *RayTraceWrap::currentCoordinate4;
ObjectData *RayTraceWrap::objectData = 0;
int RayTraceWrap::numObjects = 0;
LightData *RayTraceWrap::lightData = 0;
int RayTraceWrap::numLights = 0;
float RayTraceWrap::currentkd = 0.5;
float RayTraceWrap::currentks = 0.5;

/*
   end of inventor, opengl, window system wrapper
*/

/***** lineUI.h *****/

#ifndef_lineUI_h
#define_lineUI_h

void DTraceLineMode(SoSceneViewer *sv);

#endif

// lineUI.C

// system include files
#include <fstream.h>
#include <iostream.h> // for strstreams
#include <string.h>

// local include files
#include "ivwrapplib/SoSceneViewer.h"
#include "iv/RayTraceWrap.h"
#include "RayExt.h"
```cpp
#include "lineUI.h"

void DTraceLineMode(SoSceneViewer *sViewer)
{
    cout << "DTraceLineMode\n";
    cout.flush();

    // Make sure camera is perspective, not orthographic.
    assert(sViewer->getCamera());
    if (!sViewer->getCamera()->isOfType(SoPerspectiveCamera::getClassTypeId()))
    {
        cerr << "Error: Ray-tracing is enabled only for perspective cameras."
        << " The current camera setting defines orthographic projection."
        << " The frustum icon in the examiner-viewer toggles camera type."
        << endl;
        return;
    }

    // Create a view volume from the current camera.
    SoPerspectiveCamera *cam = (SoPerspectiveCamera *)sViewer->getCamera();
    ViewVolume vVolume;
    RayTraceWrap::viewportSize = RayTrace::winSize;

    if (RayTrace::OverRideEye) {
        FILE *f = fopen(RayTrace::CamFile, "r");
        if (!f) {
            fprintf(stderr, "Could not open camfile %s \n",
                    RayTrace::CamFile);
            exit(1);
        }
        float a, b, c;
        fscanf(f, "%f %f %f", &a, &b, &c);
        vVolume.eye.setValue(a, b, c);
        fscanf(f, "%f %f %f", &a, &b, &c);
        vVolume.viewDir.setValue(a, b, c);
        vVolume.viewDir.normalize();
        fscanf(f, "%f %f %f", &a, &b, &c);
        vVolume.upDir.setValue(a, b, c);
        vVolume.upDir.normalize();
        fclose(f);
    } else {
        vVolume.eye = cam->position.getValue();
        cam->orientation.getValue().multVec(SbVec3f(0, 0, -1), vVolume.viewDir);
        cam->orientation.getValue().multVec(SbVec3f(0, 1, 0), vVolume.upDir);
        vVolume.orientation = cam->orientation.getValue();
    }
}
```
vVolume.aspectRatio = RayTraceWrap::viewportSize[0] / 
(float)RayTraceWrap::viewportSize[1];

// fill in remaining view volume attributes
vVolume.nearDistance = cam->nearDistance.getValue();
vVolume.farDistance = cam->farDistance.getValue();
vVolume.fovy = cam->heightAngle.getValue();

// Move objects, lights into objectData, lightData arrays.
SoSeparator *sceneGraph = sViewer->getSceneGraph();
assert( sceneGraph );

// Allocate & fill object, light arrays (must be released later).
RayTraceWrap::getObjectsAndLights( sceneGraph );

// set testParams to 1 to debug param passing;
// set to 0 for normal (raytrace callback) use
int testParams = 0;
if (!testParams) {
    // Allocate pixel array pixelData.
    assert(!RayTraceWrap::pixelData);
    RayTraceWrap::pixelData = new GLuint [RayTraceWrap::viewportSize[0] * 
            RayTraceWrap::viewportSize[1]];
    assert(RayTraceWrap::pixelData);

    // Effect immediate or delayed pixel drawing.
    // ivRayTrace scene into pixelData array.
    RayExt (RayTraceWrap::objectData, 
             RayTraceWrap::numObjects, 
             RayTraceWrap::lightData, 
             RayTraceWrap::numLights, 
             &vVolume, 
             RayTraceWrap::viewportSize, 
             RayTraceWrap::setPixelDelayed); 

    // save pixel data
    RayTraceWrap::saveWindow((unsigned long *)RayTraceWrap::pixelData, 
                            RayTraceWrap::viewportSize[0], 
                            RayTraceWrap::viewportSize[1], 
                            RayTrace::basename, 
                            RayTrace::debug);

    // Release pixel array.
    delete [] RayTraceWrap::pixelData;
    RayTraceWrap::pixelData = 0;
}
else {
    // Test flattened object and light arrays by rendering
    // them as Inventor scene graph.
    testParameters(RayTraceWrap::objectData, 
                  RayTraceWrap::numObjects, 
                  RayTraceWrap::lightData, 
                  RayTraceWrap::numLights, 
                  RayTraceWrap::viewportSize,
                  RayTraceWrap::setPixelDelayed); 

    // Release pixel array.
    delete [] RayTraceWrap::pixelData;
    RayTraceWrap::pixelData = 0;
}

}
&vVolume,
    RayTraceWrap::viewportSize);
}

// Release node arrays.
RayTraceWrap::freeObjectsAndLights();
}

/* Camera.C
   code by JP Mellor 11/95

   references:
   OIC++RM
   Coplien, Advanced C++ etc.
   Siggraph '88 Course Notes, Heckbert */

// system include files
#include <assert.h>
#include <math.h>
#include <iostream.h>
#include <iomanip.h>
#include <strstream.h>
#include <fstream.h>
#include <string.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

// X include files
#include <Xm/Xm.h>

// openGL include files
#include <GL/gl.h>
#include <GL/glx.h>
#include <GL/GLwDrawA.h>

// Inventor include files
#include <Inventor/Xt/SoXt.h>
#include <Inventor/Xt/viewers/SoXtFullViewer.h>
#include <Inventor/nodes/SoSeparator.h>
#include <Inventor/nodes/SoSelection.h>
#include <Inventor/SbColor.h>
#include <Inventor/nodes/SoCamera.h>
#include <Inventor/SbLinear.h>
#include <Inventor/SbBox.h>
#include <Inventor/SbViewportRegion.h>
#include <Inventor/SoPickedPoint.h>
Camera::recordCB is called when the REC button is pushed

```c
void Camera::recordCB(Widget, XtPointer data, XtPointer) {
    SoSceneViewer *sViewer = (SoSceneViewer *)data;

    // Recover the camera data for the current camera.
    SoPerspectiveCamera *cam = (SoPerspectiveCamera *)sViewer->getCamera();

    static float x0, y0, z0, a0, b0, c0, q = 0;
    SbVec3f position;
    float x, y, z;
    SbVec3f axis;
    float a, b, c, angle;

    position = cam->position.getValue0.getValue0;
}```
position.getValue(x, y, z);
cam->orientation.getValue(axis, angle);
axis.getValue(a, b, c);

if (q && (x == x) && (y == y) && (z == z) && (a == a) &&
(b == b) && (c == c)) {
    return;
}
q = 1;
x0 = x;
y0 = y;
z0 = z;
a0 = a;
b0 = b;
c0 = c;

// write out camera data
fprintf(RayTrace::scriptFILE, "position: %f %f %f\n", x, y, z);
fprintf(RayTrace::scriptFILE, "orientation: %f %f %f %f\n", a, b, c,
angle);

// print camera data out
if (RayTrace::debug) {
    fprintf(stderr, "position: %f %f %f\n", x, y, z);
    fprintf(stderr, "orientation: %f %f %f %f\n", a, b, c,
angle);
}
}

// Camera::playCB() is called when the PLAY button is pushed.

void
Camera::playCB(Widget widget, XtPointer data, XtPointer cbs)
{
    SoSceneViewer *sViewer = (SoSceneViewer *)data;

    if (RayTrace::read) {
        // Recover the current camera.
        SoPerspectiveCamera *cam = (SoPerspectiveCamera *)sViewer->getCamera();

        float x, y, z;
        SbVec3f axis;
        float a, b, c, angle;

        // read in camera data
        if (EOF ==
            fscanf(RayTrace::scriptFILE, "position: %f %f %f\n", &x, &y, &z))
            exit(0);
        if (EOF ==
            fscanf(RayTrace::scriptFILE, "orientation: %f %f %f %f\n", &a,
&b, &c, &angle))
exit(0);

// print camera data out
if (RayTrace::debug) {
    fprintf(stderr, "position: %f %f %f\n", x, y, z);
    fprintf(stderr, "orientation: %f %f %f %f\n", a, b, c, angle);
}

// update the camera parameters
cam->position.setValue(x, y, z);
axis.setValue(a, b, c);
cam->orientation.setValue(axis, angle);
}

// save the image
if (RayTrace::out) {
    if (RayTrace::mode) {
        RayTraceWrap::raytraceCB(widget, data, cbs);
    } else {
        // recover the current viewer and the region to render
        static SoXtFullViewer *cv = sViewer->getCurrentViewer();
        static SbViewportRegion region = cv->getViewportRegion();

        // force the scene to be rerendered so all changes take effect
        cv->renderO;

        // get framebuffer data
        SbVec2s viewportSize;
        viewportSize = region.getViewportSizePixels();
        GLuint *pixelData = new GLuint[viewportSize[0] * viewportSize[1]];
        glReadPixels(0, 0, viewportSize[0], viewportSize[1], GL_RGBA,
                    GL_UNSIGNED_BYTE, pixelData);

        // save image
        RayTraceWrap::saveWindow((unsigned long*)pixelData, viewportSize[0],
                                  viewportSize[1], RayTrace::basename,
                                  RayTrace::debug);

        delete [] pixelData;
    }
    if (!RayTrace::read) {
        exit(0);
    }
}

// Camera::recordCB1 is called whenever the camera changes
void Camera::recordCB( void *data, SoSensor * )
{
    Camera::recordCB(NULL, data, NULL);
}

// Camera::playCB1() is called every interval in continuous mode.

void Camera::playCB1( void *data, SoSensor * )
{
    Camera::playCB(NULL, data, NULL);
}

// Camera.h

#ifndef _CAMERA
#define _CAMERA

// X includes
#include <Xm/Xm.h>

// inventor includes
#include <Inventor/nodes/SoSeparator.h>
#include <Inventor/SbColor.h>
#include <Inventor/SbLinear.h>
#include <Inventor/actions/SoCallbackAction.h>
#include <Inventor/nodes/SoEnvironment.h>
#include <Inventor/nodes/SoCoordinate3.h>
#include <Inventor/nodes/SoCoordinate4.h>

// openGL includes
#include <GL/gl.h>

// local include
#include "RayTraceTypes.h"

// Wrapper for camera functions.
class Camera {

    public:
        // Called when SceneViewer RTD, RT buttons are pressed.
        static void recordCB(Widget, XtPointer, XtPointer);
        static void playCB(Widget, XtPointer, XtPointer);
        static void recordCB1( void *, SoSensor *);
        static void playCB1( void *, SoSensor *);

};

#endif /* !_CAMERA */
Appendix D

Basic Ray Tracer Functions & Data Objects

/***** RayExt.h ******/

#ifndef _RayExt_h_
#define _RayExt_h_

#include <Inventor/SbBasic.h>
#include <Inventor/SbColor.h>
#include <Inventor/SbLinear.h>
#include <Inventor/nodes/SoSeparator.h>
#include <Inventor/nodes/SoEnvironment.h>
#include <Inventor/nodes/SoCoordinate3.h>
#include <Inventor/nodes/SoNormal.h>
#include "iv/RayTraceTypes.h"

void RayExt(ObjectData *objects, int nObjects, LightData *lights, int nLights, ViewVolume *vV, SbVec2s vpSize, void setPixel(int x, int y, SbColor color));

#endif

// RayExt.C
#include "RayExt.h"

// system include files
#include <unistd.h>
#include <iostream.h>
#include <sys/types.h>
#include <sys/prctl.h>
```c
#include <signal.h>
#include <stdio.h>
#include <assert.h>

// local include files
#include "iv/MathUtils.h"
#include "iv/RayTraceTypes.h"
#include "iv/RayTraceWrap.h"
#include "iv/0ctree.h"
#include "iv/Trace.h"

#include "compQ.h"
#include "TraceProc.h"
#include "supCTree.h"

pid_t RayExtPid;
int * TProcComps = NULL;
lock ExtWaitingOnEmptyLock;
BOOLEAN ExtWaitingOnEmpty = FALSE;
pid_t * TProcPidArray = NULL;
extern int NumHits;
extern int NumMiss;

int waitTOT = 0;
int NumHits, NumMiss;  // global hit/miss counts

/* this function will return the name of the next CTreeRecFile the
name will always be of the form CTR<c> where c is a lowercase letter
representing the number of calls to this function (a being 2 call
etc... */

static char * NextCTreeRecFileName()
{
    static int i(0);
    static char name[10] = "CTR";

    name[3] = (i++) + 'a';
    return name;
}

void DISTsetpixel(int x, int y, SbColor color)
{
    // convert float vector to packed rgba format
    GLuint rgba = RayTraceWrap::ABGR_to_RGBA(color.getPackedValue());

    // store in our local framebuffer
    RayTraceWrap::pixelData[x + y * RayTraceWrap::viewportSize[0]] = rgba;
}

void RayExt(ObjectData *objects, int nObjects, LightData *lights, int
           nLights, ViewVolume *vV, SbVec2s vpSize,
           void setPixel(int x, int y, SbColor color))
{
    ShadeMode oldMode = RayTrace::shademode;
}
```
// BBB -- print initial diagnostics
cout << "RayExt() begins, with:" << endl
<< "n Objects << " object(s), " n Lights << " light(s);" << endl
<< "n Objects << " object(s), " n Lights << " light(s);" << endl
<< "nObjects << " object(s), " nLights << " light(s);" << endl
<< "nObjects << " object(s), " nLights << " light(s);" << endl
<< "samples per pixel: " << RayTrace::ssg << " x " << RayTrace::ssg << " = " << RayTrace::ssr
<< " (RayTrace::jitter ? " (jittered)" : " (non-jittered))" << endl
<< "max ray depth: " << RayTrace::maxdepth << endl
<< "min ray weight: " << RayTrace::minweight << endl
<< "shade mode: " << RayTrace::shademode << endl
<< "debug level: " << RayTrace::debug << endl;
if (RayTrace::NewTrace)
cout << "Processes will be "
<< "(RayTrace::Incremental ? "Incrementally Delegating Work."
<< "Recursively Delegating Working")" << endl;
else
cout << "Traditional Recursive Raytracing Enabled" << endl;

cout << "UI mode: 
<< ((RayTrace::UImode == RTT_GL_MODE) ? "GL": "line")
<< endl;
cout.flush();

Octree octree;
// create scene spatial data structure
// expand octree to include eye
octree.Grow ( vV->eye );
// insert all objects into octree
for ( int k = 0; k < nObjects; k++ )
{ octree.Grow ( objects[k].bboxWS );
octree.Insert ( &objects[k], &(objects[k].bboxWS) );
}
// expand octree to include local lights
for ( k = 0; k < nLights; k++ )
if ( lights[k].lighttype != INFINITE_LIGHT )
octree.Grow ( lights[k].locationWS );

// CCC -- for extra credit, recursively split the
// octree and distribute scene entities into it
// construct left-handed orthobasis xhat, yhat, zhat that
// spans the truncated pyramid which is the portion of the
// view pyramid between the eye and the near clipping plane.
// note: the hat vectors are orthogonal but not normalized.
// BBB -- this code is supplied
SbVec3f xhat, yhat, zhat;
/*
// zhat: vector from eye to view plane, normal to view plane
// yhat: vector parallel to up direction, lying in view plane
// if up width < up height, width constrains viewing frustum
// xhat: vector perpendicular to both zhat and yhat
// initialize all concurrent data objects

supConvergenceObject *sC(0);
CTree *C(0);
compQ GlobalCompQ;
TraceArg *procArg;

int RICBufSize;
RIC **RICBuf;

if (RayTrace::NewTrace) {
    TProcSyncLockArray = new lock[RayTrace::nProcesses];
    TProcPidArray = new pidt[RayTrace::nProcesses];
    TProcComps = new int[RayTrace::nProcesses];
    RICBufSize = RayTrace::nProcesses * 10;
    RICBuf = new RICp[RICBufSize];
    assert (TProcSyncLockArray);
    assert (TProcPidArray);
    assert (TProcComps);
    assert (RICBuf);
}

if (RayTrace::Incremental) RayTrace::shademode = SHADE_DIRECT;
RayExtPid = getpid();

if (RayTrace::Incremental ? sproc(TraceProcIncremental, PR_SALL, procArg) : sproc(TraceProcRecurse, PR_SALL, procArg));
cout << TProcPidArray[k] << " ";
if (TProcPidArray[k] == -1)
{
    perror("An error occurred while spawning the TProcs");
    exit(0);
}

// use the distributed version of setpixel
ConvergenceObject::SetPixel = DISTsetpixel;
cout << k << " processes spawned..." << endl;
}

// during this loop k will be the number of accumulated RICs. If the
// Q is empty we will place RayTrace::nProcess/2 new RICs in the Q
*

NumMiss = NumHits = 0;
tot = k = 0;
total = vpSize[0] * vpSize[1] * RayTrace::ssr;

// find (non—normalized) WS ray from eye to pixel; note that
// mapping takes lower—left corner of pixel (0,0), and upper—
// right corner of pixel (vp[0]—1, vp[1]—1), to frustum corners

// allocate jitter coefficients (no jittering if 1 sample/pixel)
Vector2D *jitter = new Vector2D[RayTrace::ssr]; assert ( jitter );
float gridspacing = 1.0 / (RayTrace::ssg + 1);
for (int j = 0; j < RayTrace::ssr; j++)
{
    jitter[j][0] = jitter[j][1] = 0.f;
    if (RayTrace::jitter)
    {
        // BBB -- jittered (randomized) sampling: generate random sample
        // hint: use RayTrace::ssr and randr () with suitable lo, hi
        jitter[j][0] = randr(0,1);
        jitter[j][1] = randr(0,1);
    }
    else
    {
        // BBB -- non—jittered (regular) sampling: NxN sample grid
        // hint: use ( j % RayTrace::ssg ) and ( j / RayTrace::ssg )
        jitter[j][0] = (j % RayTrace::ssg) * gridspacing + 0.5 * gridspacing;
        jitter[j][1] = (j / RayTrace::ssg) * gridspacing + 0.5 * gridspacing;
    }
}

for (int y = 0; y < vpSize[1]; y++) { // for each scanline y
    for (int x = 0; x < vpSize[0]; x++) { // for each pixel x
        // define index of refraction in vacuum for lambda = r,g,b
        Vector etavacuum(1,1,1);
        if (RayTrace::NewTrace) // are the procs delegating work?
        {
            // check if we should use a CTreeRec or a CTree
            if (curCTRec < RayTrace::CTreeRecSize) {
                // this pixel will be recorded
                if ((x == RayTrace::CTreeRecIndices[curCTRec * 2]) &&
                    (y == RayTrace::CTreeRecIndices[curCTRec * 2 + 1]))
                {
                    curCTRec++;
                    C = new CTreeRec(x, y, NextCTreeRecFileName());
                }}}}
} else // don't record this pixel
    if (RayTrace::ssr>1) C = new supCTree(x, y, RayTrace::ssr);
    else C = new CTree(x, y);
} else // we are not recording any pixels
    if (RayTrace::ssr>1) C = new supCTree(x, y, RayTrace::ssr);
    else C = new CTree(x, y);

} else // if I'm not delegating
    // I know this is a hack, but I'll allow myself this one
sC = new supConvergenceObject(x, y, RayTrace::ssr);

Radiance radiance(0,0,0);
// supersample the pixel with ssr rays
// CCC -- sample over time interval
// CCC -- sample in depth of field
for ( int s = 0; s < RayTrace::ssr; s++ ) {
    float jx = x + jitter[s][0];
    float jy = y + jitter[s][1];
    Vector epvec = zhat + ((2. * jx + 1.) / vpSize[0] - 1.) * xhat
                   + ((2. * jy + 1.) / vpSize[1] - 1.) * yhat;

    // normalize it
    epvec.normalize();

    // accumulate radiance coming to eye along this sample ray
    if (RayTrace::NewTrace) {
    Ray *eyeray = new Ray( vV->eye, epvec);
    RayPath *r = NULL;
    RIC *ric;
    if (!RayTrace::Incremental)
        ric = (RIC *)new RIC_CO(eyeray, sC, s);
    else { // I'm using a CTree
        r = C->Root(s);
        ric = (RIC *)new RIC_CT(eyeray, C, r);
    }

    // if k < RICBufSize then add ric to RICBuf
    RICBuf[k++] = ric;

    if (k == RICBufSize) {
        // wait until the Q is empty.
        GlobalCompQ.WaitOnSizeLessThan(k);

        // add the k new RICs to the Q
        GlobalCompQ.add_comp(RICBuf, k);

        tot += k;
        if (tot%1000 == 0)
            cout << tot << "/" << total << " pixels dispatched" << endl;
        // reset k
        k = 0;
    }
} else
radiance += ivTrace (Ray( vV->eye, epvec ), 0, 1., eta-vacuum,
&octree, lights, nLights);
}

if (!RayTrace::NewTrace) {
  radiance /= RayTrace::ssr;
  // since this is not a concurrent execution, use the standard
  // set pixel
  setPixel( x, y, radiance);
}

cout << "Cleaning Up...

// place the remaining RICs in the Q.
if (RayTrace::NewTrace) {
  /* if there are any remaining computations then add them
     to the queue */
  if (k > 0) {
    GlobalCompQ.add_comp(RICBuf, k);
    cout << "Last " << k << " pixels... ";
  }

  // wait until there are no ConvergenceObject instances
  ConvergenceObject::WaitOnNoInst();
  // kill the processes
  for (k = 0; k < RayTrace::nProcesses; k++)
    RICBuf[k] = NULL;
  GlobalCompQ.add_comp(RICBuf, k);
  // wait for them to die
  for (k = 0; k < RayTrace::nProcesses; k++)
    TProcSyncLockArray[k].lock_resource();
  RayTrace::shademode = oldMode;

  // delete the pid and comps arrays
  delete [] TProcSyncLockArray;
  delete [] TProcPidArray;
  delete [] TProcComps;
  delete [] RICBuf;
}

cout << "RayExt() finishes:" << endl
     << "Entries added to compQ " << GlobalCompQ.EntriesAdded() << endl;

cout << "A total of " << tot << " computations." << endl;

cout << "Number of Nonempty waits " << waitTOT << endl;
cout << "rays cast: " << Ray::NumRaysCast() << endl;
cout << "number of hits: " << NumHits << endl;
cout << "number of misses: " << NumMiss << endl;
float tmp = (float) (NumHits + NumMiss) / (float) (vpSize[0] * vpSize[1]);
float depth = logf(tmp) / logf(2.0);
printf("average depth: %f\n",depth);
// Trace.h
#ifndef RT_TRACE
#define _RTTRACE
#include "RayTraceTypes.h"
#include "Octree.h"

Radiance ivTrace ( Ray &ray,
    int depth, float weight, Vector eta,
    Octree *octree,
    LightData *lights, int nLights );
#endif /* ! _RT_TRACE */

// Trace.C

// system include files
#include <math.h>
#include <stdio.h>
#include <assert.h>

// local include files
#include "Octree.h"
#include "MathUtils.h"
#include "RayTraceTypes.h"
#include "Intersect.h"
#include "Shade.h"
#include "Trace.h"

/* ivTrace():
   // given a ray origin, a (possibly non-normalized) direction, and
   // a list of scene objects, return radiance along the given ray.
   // depth and weight are used for recursion termination; eta is
   // the index of refraction (>= 1.0) for this ray's travel medium
   // calls ivIntersect(), ivShade()
*/
Radiance ivTrace ( Ray &ray,
    int depth, float weight, Vector eta,
    Octree *octree,
    LightData *lights, int nLights ) {  
  // find first intersection of ray with scene object
  Hit hit;
  if ( ivIntersect ( ray, octree, hit ) ) {
    // ray hit some surface; compute radiance at hit
    // point as viewed along the ray direction
    return ivShade ( hit, ray.dirWS, ray.spd,
        depth, weight, eta,
        ...
else {
  // ray escaped scene; compute environmental radiance along exiting ray direction
  return ivShade(hit, ray.dirWS, ray.spd,
                  depth, weight, eta,
                  octree, NULL, 0);
}

// Shade.h
#ifndef _RT_SHADE_
#define _RT_SHADE_

#include "RayTraceTypes.h"
#include "Octree.h"

Radiance ivShade(Hit &hit,
                 Vector &viewdWS,
                 unsigned char spd[3],
                 int depth, float weight, Vector eta,
                 Octree *octree,
                 LightData *lights, int nLights);

int TransmissionDirection(Vector &I, Vector &N,
                          float &etal, float &etaT,
                          OUTPUT Vector &tray);

/* Background Map():
// returns a radiance value from the background environment map specified
// by the ray direction; in this dummy case, a simple RGB color space is used.
*/
Radiance BackgroundMap(Vector &D);

/* ReflectionDirection():
// compute reflection direction of ray at interface
// returns 0 if there is no reflected vector
*/
int ReflectionDirection(Vector &I, Vector &N,
                        OUTPUT Vector &ray);

#endif /* !_RT_SHADE_ */

// Shade.C

// system include files
#include <math.h>
```c
#include <stdio.h>
#include <iostream.h>
#include <assert.h>

// local include files
#include "MathUtils.h"
#include "RayTraceTypes.h"
#include "Intersect.h"
#include "Shade.h"
#include "Trace.h"

extern int NumHits, NumMiss; // global hit/miss counts

// Component-wise multiply.
static SbColor mult(const SbColor &a, const SbColor &b) {
    return SbColor(a[0] * b[0], a[1] * b[1], a[2] * b[2]);
}

/* Background Map():
   // returns a radiance value from the background
   // environment map specified by the ray direction;
   // in this dummy case, a simple RGB color space is used.
   */
Radiance BackgroundMap(Vector &D) {
    float ssum = sqrtf(D[0]*D[0]+D[1]*D[1]+D[2]*D[2]);
    float r = fabsf (D[0]/ssum);
    float g = fabsf (D[1]/ssum);
    float b = fabsf (D[2]/ssum);
    return (Radiance(r, g, b));
}

/* ReflectionDirection():
   // compute reflection direction of ray at interface
   // returns 0 if there is no reflected vector
   */
int ReflectionDirection ( Vector &I, Vector &N, OUTPUT Vector &ray ) {
    // BBB -- fill in this routine
    ray = I - 2.0 * N.dot(I) * N;
    ray.normalize();
    return(1);
}

/* TransmissionDirection():
   // compute transmission direction of ray at interface
   */
```

int TransmissionDirection ( Vector &I, Vector &N, float &etal, float &etaT, OUTPUT Vector &tray )
{
  // BBB -- fill in this routine
  // do NOT use any inverse trig!

  // For convenience.
  float ratio = etal / etaT;
  float cosine = N.dot(-I);
  float d = 1.0 - (ratio * ratio) * (1.0 - cosine * cosine);
  if (d <= 0) { // Total internal reflection.
    return(1);
  }
  tray = ratio * I + N * (ratio * cosine - sqrtf(d));
  return(0);
}

Radiance
ivShade ( Hit &hit, Vector &viewdWS, unsigned char spd[3], int depth, float weight, Vector eta, Octree *octree, LightData *lights, int nLights )
{
  ObjectData *obj = hit.object;

  if ( !obj ) {
    // ray escaped scene; return flat grey background radiance
    // CCC -- for extra credit, modify this code to compute the
    // radiance along the escaping ray; for example, by sampling
// an environment map which represents far-field objects
// return BackgroundMap (viewdWS);
NumMiss++; // update miss counts
return Radiance(0.0, 0.0, 0.0);
} else {
    NumHits++; // update hit counts
}

Radiance radiance = Radiance(0, 0, 0);
int castshadowrays = FALSE;

switch (RayTrace::shademode) {
    case SHADE_NORMAL:
        // AAA -- BBB -- this code is supplied for you
        // map normal [-1..+1] x [-1..+1] x [-1..+1] to radiance
        radiance[0] = 0.5 * (1. + hit.surfnWS[0]);
        radiance[1] = 0.5 * (1. + hit.surfnWS[1]);
        radiance[2] = 0.5 * (1. + hit.surfnWS[2]);
        break;
    case SHADE_FLAT:
        // AAA -- BBB -- this code is supplied for you
        // shade using object's diffuse color
        radiance = obj->diffuse;
        break;
    case SHADE_RECURSIVE:
        if (depth < RayTrace::maxdepth) {
            // BBB -- compute refl direction with this routine
            Vector rraydirWS;
            int ref = ReflectionDirection (viewdWS, hit.surfnWS, rraydirWS);

            // BBB -- collect radiance along reflection ray rraydir
            // BBB -- collect radiance along reflected ray,
            // multiplied by k_reflection
            if (ref & (obj->ks*weight >= RayTrace::minweight)) {
                rraydirWS.normalize();
                Ray ray(hit.surfWS+.001*rraydirWS, rraydirWS);
                radiance += obj->ks * ivTrace(ray, depth+1, obj->ks*weight, 
                    eta, octree, lights, nLights);
            }

            // transparency 0 => opaque;
            // transparency 1 => completely transparent
            float transp = obj->transparency;
            if (transp > 0.0) {
                // collect radiance along transmission ray traydir
                // here, we have encoded the index of refraction for
                // r, g, and b wavelengths in environment "fog" color
                // which has been placed in the object's "eta" field
            }
        }
}
Vector traydirWS;
Vector eta_vacuum(1.0,1.0,1.0);

// BBB -- compute the transmission direction(s) with this routine
// (you may wish to compute transm rays at each of R,G, and B)
// check the Ray.spd[] mask for radiance at R,G,B
// R: tir = TransmissionDirection (viewdWS, hit.surfnWS,
// eta[0], obj->eta[0], traydirWS );
// G: 
// B: etc.

int tir = 0;
for(int rgb = 0; rgb<3; rgb++ ) { // loop through R,G,B
    if (!spd[rgb]) continue;
    if (viewdWS.dot(hit.surfnWS) < 0.0) { // entering object
        tir = TransmissionDirection (viewdWS, hit.surfnWS,
            eta_vacuum[rgb], obj->eta[rgb], traydirWS);
    } else { // exiting object
        tir = TransmissionDirection (viewdWS, -hit.surfnWS,
            obj->eta[rgb], eta_vacuum[rgb], traydirWS);
    }

    // CCC -- generalize ivray's notion of spectral
    // distribution, and sample across continuous spectrum
    // to produce physically realistic dispersion effects

    // BBB -- if object is transparent, and total internal reflection
    // did not occur, spawn transmission ray and accumulate the result
    if (!tir && (transp*weight >= RayTrace::minweight)) {
        // BBB -- collect radiance along ray, multiplied by k_trans
        // transpar 0 => opaque; transpar 1 => completely transparent
        traydirWS.normalize();
        Ray tray(hit.surfpWS+.001*traydirWS, traydirWS);
        tray.spd[rgb] = 1;
        Color trad = ivTrace(tray, depth+1, transp*weight,
            eta, octree, lights, nLights);
        radiance[rgb] += transp * trad[rgb];
    }
}
if (viewdWS.dot(hit.surfnWS) >= 0.0) { // exiting object
    break;
}
}

// CCC -- cast "distribution rays" to handle area lights
// CCC -- cast "dispersion rays" to handle wavelength dependence
// note: no break; drop through to collect DIRECT illumination

case SHADE_DIRECT :
    // enable shadow rays
castshadowrays = TRUE;
// note: no break; drop through to collect LOCAL illumination

case SHADE_LOCAL :
// sum individual contributions of light sources
{
    int k;
    for (k = 0; k < nLights; k++) {
        LightData *light = &lights[k];
        // light was transformed into WS once at start of time;
        // Vector light->locationWS; // WS light location, if LOCAL/SPOT
        // Vector light->directionWS; // WS light direction, if INF/SPOT

        Vector LdirWS; // unit vector pointing from shade pt to light
        Vector ToLight; // full-length vector pointing from shade pt to light
        int Ltype = light->lighttype;

        if (Ltype == INFINITE_LIGHT) {
            LdirWS = -light->directionWS;
            ToLight = LdirWS * HUGE;
        } else {
            // if LOCAL or SPOT_LIGHT
            ToLight = light->locationWS - hit.surfpWS;
            LdirWS = ToLight;
        }
        LdirWS.normalize();

        // BBB -- if shadow ray testing is enabled, cast shadow ray

        float trans1 = 1.0; // medium transparency
        float trans1a = 1.0; // medium transparency
        float trans2 = obj->transparency; // object transparency
        int inters;

        if (castshadowrays) {
            Hit hit2;

            // displace slightly away from the object surface

            Ray ray(hit.surfpWS+.001*LdirWS, LdirWS);

            if (LdirWS.dot(hit.surfnWS) < 0) {
                continue;
            } else {

                float distToLight = ToLight.length();
                float distToHit = 0;
                Vector ToHit;

                do { // accumulate medium transparency
                    inters = ivIntersect(ray, octree, hit2);
                    if (inters) {
                        ray = Ray(hit2.surfpWS+.001*LdirWS, LdirWS);
                        trans1 = trans1a;
                } else {

                    float distToLight = ToLight.length();
                    float distToHit = 0;
                    Vector ToHit;

                    do { // accumulate medium transparency
                        inters = ivIntersect(ray, octree, hit2);
                        if (inters) {
                            ray = Ray(hit2.surfpWS+.001*LdirWS, LdirWS);
                            trans1 = trans1a;
        } else {

                    float distToLight = ToLight.length();
                    float distToHit = 0;
                    Vector ToHit;

                    do { // accumulate medium transparency
                        inters = ivIntersect(ray, octree, hit2);
                        if (inters) {
                            ray = Ray(hit2.surfpWS+.001*LdirWS, LdirWS);
                            trans1 = trans1a;

transla *= hit2.object->transparency;
ToHit = hit2.surfpWS - hit.surfpWS;
distToHit = ToHit.length();
}
} while (inters && (transl>0) && (distToHit < distToLight));

if (transl==0) {
    continue; // if shadowing, skip to next light
}
}
}

// BBB -- if spot light, and inside cone of influence,
// compute angular attenuation for SPOT_LIGHT
float cosine;
float tmp;
float atten = 1.0;

if (Ltype == SPOT_LIGHT) {
    light->directionWS.normalize();
    cosine = light->directionWS.dot(-LdirWS);
    tmp = ((SoSpotLight *)light->light)->cutOffAngle.getValue();
    if (cosine < cos(tmp)) {
        continue;
    }
    tmp = ((SoSpotLight *)light->light)->dropOffRate.getValue();
    atten *= powf (fmax (cosine, 0.0), 128.0 * tmp);
}

// compute distance attenuation for LOCAL and SPOT_LIGHT
if (light->lighttype != INFINITE_LIGHT) {
    Vector Att = obj->environment->attenuation.getValue();
    float dist = ToLight.length();
    tmp = fmin (1.0 / (Att[0]*dist*dist + Att[1]*dist + Att[2]), 1.0);
    atten *= tmp;
}

// BBB -- collect radiance due to diffuse reflection
// this should only be a function of N, L, material...
// Radiance dradiance = DiffuseRadiance(hit, light);
// radiance += ...

hit.surfnWS.normalize();
cosine = fmax (LdirWS.dot(hit.surfnWS), 0.0);
Color diffuse = mult (obj->diffuse * cosine,
    light->light->intensity.getValue() *
    light->light->color.getValue());
radiance += atten * transl * (1-trans2) * diffuse;

// BBB -- collect radiance due to specular reflection
// this should be a function of N, L, viewdir, material...
// for example, you might do
Radiance sradiance = SpecularRadiance(hit, viewdWS, light);
radiance += ...

Vector V = -viewdWS;
V.normalize();
Vector H = V + LdirWS;
H.normalize();
tmp = powf(fmax(H.dot(hit.surfnWS), 0.0), 256.0 * obj->shininess);
Color specular = mult(obj->specular * tmp,
    light->light->intensity.getValue() *
    light->light->color.getValue());
radiance += atten * trans1 * (1-trans2) * specular;

}  // note: no break; drop through to collect AMBIENT illumination

case SHADE_AMBIENT:
{
    // BBB -- collect radiance due to environmental ambient light
    Color ambient = mult(obj->environment->ambientIntensity.getValue() *
        obj->environment->ambientColor.getValue(),
        obj->ambient);
radiance += ambient;
    // BBB -- collect radiance due to object's "self-emission"
radiance += obj->emission;
}
break;

switch

// clamp RGB output at 1.0

if (radiance[0] > 1.0) radiance[0] = 1.0;
if (radiance[1] > 1.0) radiance[1] = 1.0;
return radiance;

Octree.h

#ifndef RT_OCTREE_
#define RT_OCTREE_
#include <Inventor/SbLinear.h>
#include <Inventor/SbBox.h>
typedef SbVec3f Point;
typedef SbBox3f Bbox;
typedef Bbox *BboxP;
typedef void *VoidP;

class Octree {
public:
   // constructor for empty root node
   Octree( void );

   // constructor for nonempty root node
   Octree( float xmin, float ymin, float zmin,
           float xmax, float ymax, float zmax );

   // destructor
   *Octree( void );

   // true iff no children
   int Octree::Leaf( void );

   // true iff no parent
   int Octree::Root( void );

   // expand to contain p
   int Octree::Grow ( Point &p );

   // expand to contain bbox
   int Octree::Grow ( Bbox &bbox );

   // true iff boxes intersect
   int Octree::Incident ( Bbox &bbox );

   // insert opaque item pointer
   int Octree::Insert ( void *item, Bbox *bbox );

   // split at point c
   int Octree::Split( Point &c );

   // return item pointer array
   void **ItemList( void );

   // return number of associated items
   int NumItems( void );

private:
   Bbox extent; // axial extent of this octree cell
   Octree *parent; // 0 iff this node is root of octree
   Octree **children; // 0, or array of 8 octree pointers
   int leaf; // true iff this node has no children
   int nitems; // number of items associated with node
   void **items; // array of n opaque pointers to item
   Bbox **bboxs; // array of n pointers to item bboxs
   int maxitems; // length of allocated item, bbox arrays

};

typedef Octree *OctreeP;

/* !RT_OCTREE */
// Octree.C

#include <assert.h>
#include <math.h>
#include <iostream.h>

// local includes
#include "Octree.h"

// constructor, produces empty extent
Octree::Octree( void)
{
    extent = Bbox( HUGE, HUGE, HUGE, -HUGE, -HUGE, -HUGE );
    parent = 0;
    children = 0;
    leaf = 1;
    nitems = maxitems = 0;
    items = 0;
    bboxes = 0;
}

// constructor, given min and max values
Octree::Octree( float xmin, float ymin, float zmin,
                float xmax, float ymax, float zmax )
{
    extent = Bbox( xmin, ymin, zmin, xmax, ymax, zmax );
    parent = 0;
    children = 0;
    leaf = 1;
    nitems = maxitems = 0;
    items = 0;
    bboxes = 0;
}

// destructor
Octree::~Octree( void )
{
    // delete children
    if ( !Leaf() ) {
        for ( int k = 0; k < 8; k++ ) {
            Octree *child = children[k];
            delete child;
        }
        delete [] children;
        leaf = 1;
    }

    // delete array of item pointers
    delete [] items;

    // delete array of item bboxes
    delete [] bboxes;
}
// true iff no children
int Octree::Leaf( void )
{
    return !children;
}

// true iff no parent
int Octree::Root( void )
{
    return !parent;
}

// expand to contain p
int Octree::Grow ( Point &p )
{
    // must be root and leaf
    if ( !(Root() && Leaf()) )
    {
        cout << "can only grow node that is root and leaf!" << endl;
        return 0;
    }
    else {
        extent.extendBy( p ); // grow to contain point
        return 1;
    }
}

// expand to contain bbox
int Octree::Grow ( Bbox &bbox )
{
    // must be root and leaf
    if ( !(Root() && Leaf()) )
    {
        cout << "can only grow node that is root and leaf!" << endl;
        return 0;
    }
    else {
        extent.extendBy( bbox ); // grow to contain bbox
        return 1;
    }
}

// true iff boxes intersect
int Octree::Incident ( Bbox &bbox )


definitionxit extent.intersect ( bbox );
}

// insert opaque item pointer
int
Octree::Insert ( void *item, Bbox *bbox )
{
    if ( nitems == maxitems ) {
        maxitems = (!maxitems) ? 8 : (maxitems << 1);

        // reallocate item, bbox pointer arrays
        void **newitems = new VoidP[maxitems];
        Bbox **newbboxes = new BboxP[maxitems];

        // copy existing item, bbox pointers
        for ( int k = 0; k < nitems; k++ ) {
            newitems[k] = items[k];
            newbboxes[k] = bboxes[k];
        }

        // replace item pointer array
        if ( items ) delete [] items;
        items = newitems;

        // replace bbox pointer array
        if ( bboxes ) delete [] bboxes;
        bboxes = newbboxes;
    }

    // copy in newest item, bbox
    items[nitems] = item;
    bboxes[nitems] = bbox;
    nitems++;

    // all ok
    return 1;
}

// split at some point
int
Octree::Split( Point &c )
{
    if ( !Leaf() ) {
        cout << "can't split an internal node!" << endl;
        return 0;
    }

    if ( !extent.intersect ( c ) ) {
        cout << "can't split at non-contained point!" << endl;
        return 0;
    }
}
allocate eight children
children = new OctreeP[8];
Point min = extent.getMin();
Point max = extent.getMax();
for (int x = 0; x < 2; x++)
   for (int y = 0; y < 2; y++)
      for (int z = 0; z < 2; z++) {
         int cid = (x << 2) + (y << 1) + z;
         children[cid] = new Octree(
            !x ? min[0] : c[0],
            !y ? min[1] : c[1],
            !x ? c[0] : max[0],
            !y ? c[1] : max[1],
         children[cid]->parent = this;
         children[cid]->leaf = 1;
      }
// CCC distribute items, bboxes into children

// this node is no longer a leaf
leaf = 0;

// all ok
return 1;
}

// return item pointer array
void ** Octree::ItemList( void )
{
   return items;
}

// return number of associated items
int Octree::NumItems( void )
{
   return nitems;
}

// Intersect.h
#ifdef RT_INTERSECT
#define RT_INTERSECT
#endif
#include "RayTraceTypes.h"
#include "Octree.h"
int ivIntersect ( Ray &ray, // R, D, etc.
    Octree *octree, // scene octree
    Hit &hit ); // hit structure

#endif /* _RT_INTERSECT */

// Intersect.C

// system include files
#include <math.h>
#include <stdlib.h> // for rand()
#include <assert.h>
#include <iostream.h>

// Inventor include files
#include <Inventor/SbBasic.h>
#include <Inventor/SbLinear.h>
#include <Inventor/SbBox.h>
#include <Inventor/SbViewportRegion.h>
#include <Inventor/nodes/SoSeparator.h>
#include <Inventor/nodes/SoCamera.h>
#include <Inventor/nodes/SoSelection.h>
#include <Inventor/nodes/SoShape.h>
#include <Inventor/nodes/SoCube.h>
#include <Inventor/nodes/SoSphere.h>
#include <Inventor/nodes/SoCone.h>
#include <Inventor/nodes/SoCylinder.h>
#include <Inventor/nodes/SoIndexedFaceSet.h>
#include <Inventor/nodes/SoIndexedNurbsSurface.h>

// local include files
#include "MathUtils.h"
#include "RayTraceTypes.h"
#include "IntersectNurb.h"
#include "Intersect.h"
#include "IntersectBbox.h"

int ivIntersectBboxn = 0;

/* ivIntersectBbox();
   //
   // intersect ray with axial bounding box
   //
   // returns object space t of intersection point, or HUGE if none
   // surfnOS is assigned iff returned intersection parameter t < HUGE */
float...
ivIntersectBbox ( const Bbox &box, Point &rayorigOS, Vector &raydirOS,
    OUTPUT Vector &surfnOS )
{
    ivIntersectBboxn++;  
    // AAA -- this code is supplied for you.
    // CCC -- for extra credit, optimize it.
    // (it's fast as is; can you do better?)

    // parallelepiped is centered at origin
    // get half-width, height, depth of box
    Vector min = box.getMin();
    Vector max = box.getMax();

    // ||epiped must have non-zero volume
    assert ( (max[0] > min[0])
        && (max[1] > min[1]) && (max[2] > min[2]) );

    // ray is of form rayorigOS + t * raydirOS
    // find smallest t such that ray intersects ||epiped.
    float tmin = -HUGE, tmax = HUGE;  // from <math.h>
    float t1, t2;

    static Vector axes[3] = { Vector ( 1, 0, 0 ),
        Vector ( 0, 1, 0 ), Vector ( 0, 0, 1 ) };

    char maxNormSet = 0, minNormSet = 0;
    Vector maxNorm, minNorm;

    // intersect ray with x,y,z slabs
    for ( int k = 0; k < 3; k++ ) {
        // check ray against planes xk = -dx_k, xk = +dx_k
        if ( raydirOS[k] != 0. ) {
            t1 = (min[k] - rayorigOS[k]) / raydirOS[k];
            t2 = (max[k] - rayorigOS[k]) / raydirOS[k];

            tmin = fmax ( tmin, fmin (t1, t2) );
            tmax = fmin ( tmax, fmax (t1, t2) );

            if ( t1 > 0 &\& tmin == t1 ) {  // neg face
                minNorm = -axes[k];
                minNormSet = 1;
            }
            else if ( t2 > 0 &\& tmin == t2 ) {  // pos face
                minNorm = axes[k];
                minNormSet = 1;
            }
            else if ( t1 > 0 &\& tmax == t1 ) {  // neg face
                maxNorm = -axes[k];
                maxNormSet = 1;
            }
            else if ( t2 > 0 &\& tmax == t2 ) {  // pos face
                maxNorm = axes[k];
                maxNormSet = 1;
            }
        }
    }
}
// if
} // // k

// tmin..tmax now define intersection with parallelepiped
if ( tmin >= tmax ) // intersection is empty or a point
    return HUGE;
else if ( tmin > 0 ) {
    assert(minNormSet);
    surfnOS = minNorm;
    return tmin;
}
else if ( tmax > 0 ) {
    assert(maxNormSet);
    surfnOS = maxNorm;
    return tmax;
}
else // no hit
    return HUGE;

/* ivIntersectBbox(), overloaded version discards computed normal */
*/
static float
ivIntersectBbox ( const Bbox &box, Point &rayorigOS, Vector &raydirOS )
{
    Vector dummy;
    return ivIntersectBbox ( box, rayorigOS, raydirOS, dummy );
}

/* ivIntersectCube():
//
// intersect ray with SoCube: actually an axial parallelepiped
// returns object space t of intersection point, or HUGE if none
// surfnOS is assigned iff returned intersection parameter t < HUGE
*/
static float
ivIntersectCube ( ObjectData *obj, Point &rayorigOS, Vector &raydirOS,
 OUTPUT Vector &surfnOS )
{
    SoCube *cube = (SoCube *)obj->object;
    // AAA -- ray/SoCube intersection code is supplied
    // note that one call to ivIntersectBbox suffices
    // cube is centered at the origin, with dimensions
    // cube->width, height, depth.getValue() 

    // call ivIntersectBbox() with the cube's half dimensions
    return ( ivIntersectBbox ( Bbox(
        -0.5 * cube->width.getValue(),
        -0.5 * cube->height.getValue(),
        -0.5 * cube->depth.getValue(),
        0.5 * cube->width.getValue(),
    ) //
0.5 * cube->height.getValue(),
0.5 * cube->depth.getValue(),
rayorigOS, raydirOS, surfnOS );

/* ivIntersectSphere():
   // intersect ray with SoSphere
   // returns object space t of intersection point, or HUGE if none
   // surfnOS is assigned iff returned intersection parameter t < HUGE */
static float
ivIntersectSphere ( ObjectData *obj, Point &rayorigOS,
   Vector &raydirOS,
   OUTPUT Vector &surfnOS )
{
SoSphere *sphere = (SoSphere *)obj->object;
   // AAA -- fill in code for ray/sphere intersection
   // sphere is centered at the origin in object
   // space, with radius sphere->radius.getValue()

   // Solve the quadratic equation a*t^2 + b*t + c = 0 where
   // a = 1
   // b = 2 (Rx*Dx + Ry*Dy + Rz*Dz)
   // c = RxRx + RyRy + RzRz - r^2

   float Rx = rayorigOS[0];
   float Ry = rayorigOS[1];
   float Rz = rayorigOS[2];
   float Dx = raydirOS[0];
   float Dy = raydirOS[1];
   float Dz = raydirOS[2];
   float r = sphere->radius.getValue();

   float a = 1.0;
   float b = 2.0*(Rx*Dx + Ry*Dy + Rz*Dz);
   float c = Rx*Rx + Ry*Ry + Rz*Rz - r*r;
   float d = b*b - 4*a*c;
   float t;  // local variables

   if (d < 0.0) { // no intersection
      return HUGE;
   }
   else if (d == 0.0) { // line tangent to sphere
      t = -b / (2.0*a);
      if (t < 0) return HUGE;
   }
   else { // line intersects sphere
      float t1 = (-b + fsqrt(d)) / (2.0*a);
      float t2 = (-b - fsqrt(d)) / (2.0*a);

      if (t2 <= 0) {
if (t1 <= 0) return HUGE;
else t = t1;
} else t = t2;
}

// construct surface normal //

surfnOS = rayorigOS + t * raydirOS;
surfnOS.normalize();
return t;

// Intersect ray with the body of a cylinder with radius r, height h,
// centered at origin, pointing toward +y.

static float
ivIntersectCylinderBody (float r, float h, Point &rayorigOS, Vector &raydirOS, OUTPUT Vector &surfnOS)
{

    // Solve the quadratic equation  a*t^2 + b*t + c = 0 where
    // a = Dx*Dx + Dz*Dz
    // b = 2 (Rx*Dx + Rz*Dz)
    // c = Rx*Rx + Rz*Rz - r^2

    float Rx = rayorigOS[0];
    float Ry = rayorigOS[1];
    float Rz = rayorigOS[2];
    float Dx = raydirOS[0];
    float Dy = raydirOS[1];
    float Dz = raydirOS[2];

    float a = Dx*Dx + Dz*Dz;
    float b = 2.0*(Rx*Dx + Rz*Dz);
    float c = Rx*Rx + Rz*Rz - r*r;
    float d = b*b - 4*a*c;
    float t, y; // local variables

    if (d < 0.0) { // no intersection
        return HUGE;
    } else if (d == 0.0) { // line tangent to cylinder
        t = -b / (2.0*a);
        if (t < 0) return HUGE;
    }

    // check to see if intersection is within cylinder body //

    y = Ry + t*Dy;
    if ((y < -h/2.0) || (y > h/2.0)) return HUGE;

    } else { // line intersects cylinder
        float t1 = (-b + fsqrt(d)) / (2.0*a);
        float t2 = (-b - fsqrt(d)) / (2.0*a);
    }
// check to see if intersection is within cylinder body //

\[ y = R_y + t_1 \cdot D_y; \]
if \((y < -h/2.0) \lor (y > h/2.0)\) \(t_1 = -1;\)
\[ y = R_y + t_2 \cdot D_y; \]
if \((y < -h/2.0) \lor (y > h/2.0)\) \(t_2 = -1;\)

if \((t_2 <= 0)\) {
    if \((t_1 <= 0)\) return HUGE;
    else \(t = t_1;\)
} else \(t = t_2;\)

// construct surface normal //

surfnOS = rayorigOS + t \cdot raydirOS;
surfnOS[1] = 0.0;
surfnOS.normalize();
return t;

// Intersect ray with a circular disk of radius \(r\) parallel to the \(y\) plane

static float
ivIntersectDisk (float \(r\), float \(y\), Point &rayorigOS, Vector &raydirOS) {
    // ray nearly parallel to plane
    if (fabsf(raydirOS[1]) < 1.0E-6F) return HUGE;

    float \(t\) = \((y - rayorigOS[1]) / raydirOS[1];\)
    if \((t < 0)\) return HUGE;

    // check to see if intersection is within the disk //

    Point \(p\) = rayorigOS + \(t \cdot raydirOS;\)
    if \((p[0]^2 + p[2]^2 > r^2)\) return HUGE;

    return \(t;\)
}

/* ivIntersectCylinder();: */
//
// intersect ray with SoCylinder
//
// returns object space \(t\) of intersection point, or HUGE if none
// surfnOS is assigned iff returned intersection parameter \(t < HUGE\)
*/

static float
ivIntersectCylinder (ObjectData *obj, Point &rayorigOS,
                      Vector &raydirOS,
// Output Vector &surfNorm
{
SoCylinder *cyl = (SoCylinder *)obj->object;
// AAA —— fill in code for ray/cylinder intersection
// the cylinder is centered at the origin, has
// its symmetry axis as the *Y* axis, with
// radius cyl->radius.getValue(), and
// height cyl->height.getValue(). assume
// that the "parts" field of the cylinder is ALL,
// that is, that the cylinder’s top and bottom
// circular “caps” are part of the cylinder object
// and can cause a ray/object intersection to occur
// CCC —— for extra credit, handle parts correctly.

Vector norm;
float t1 = HUGE;
float t2 = HUGE;
float t3 = HUGE;

if (cyl->hasPart(SoCylinder::SIDES)) {
    t1 = ivIntersectCylinderBody (cyl->radius.getValue(),
        cyl->height.getValue(), rayorigOS, raydirOS, norm);
}

if (cyl->hasPart(SoCylinder::TOP)) {
    t2 = ivIntersectDisk (cyl->radius.getValue(),
        .5*cyl->height.getValue(), rayorigOS, raydirOS);
}

if (cyl->hasPart(SoCylinder::BOTTOM)) {
    t3 = ivIntersectDisk (cyl->radius.getValue(),
        -.5*cyl->height.getValue(), rayorigOS, raydirOS);
}

float t;
    // return value
if (t1 <= t2 && t1 <= t3) {
    t = t1;
    surfNorm = norm;
}
else if (t2 < t1 && t2 <= t3) {
    t = t2;
    surfNorm = Vector(0,1,0);
}
else if (t3 < t1 && t3 < t2) {
    t = t3;
    surfNorm = Vector(0,-1,0);
}
else return HUGE;

return t;
}

// Intersect ray with the body of a cone with base radius r, height h,
// centered at origin, pointing toward +y.

static float
ivIntersectConeBody (float r, float h, Point &rayorigOS,
Vector &raydirOS, OUTPUT Vector &surfnOS )
{
    // Solve the quadratic equation  \( a \cdot t^2 + b \cdot t + c = 0 \) where
    // \( a = D_x^2 + D_z^2 - D_y \cdot r^2 / h^2 \)
    // \( b = 2 \cdot (R_x \cdot D_x + R_z \cdot D_z - R_y \cdot D_y \cdot r^2 / h^2) + D_y \cdot r^2 / h \)
    // \( c = R_x \cdot R_x + R_z \cdot R_z - R_y \cdot R_y \cdot r^2 / h^2 + R_y \cdot r^2 / h - r^2 / 4 \)

    float Rx = rayorigOS[0];
    float Ry = rayorigOS[1];
    float Rz = rayorigOS[2];
    float Dx = raydirOS[0];
    float Dy = raydirOS[1];
    float Dz = raydirOS[2];

    float a = Dx*Dx + Dz*Dz - Dy*r*r/(h*h);
    float b = 2.0*(Rx*Dx + Rz*Dz - Ry*Dy*r*r/(h*h)) + Dy*r*r/h;
    float c = Rx*Rx + Rz*Rz - Ry*Ry*r*r/(h*h) + Ry*r*r/h - r*r/4;
    float d = b*b - 4*a*c; // local variables

    if (d < 0.0) { // no intersection
        return HUGE;
    } else if (d == 0.0) { // line tangent to cone
        t = -b / (2.0*a);
        if (t < 0) return HUGE;

        // check to see if intersection is within cone body //
        y = Ry + t*Dy;
        if ((y < -h/2.0) || (y > h/2.0)) return HUGE;
    } else { // line intersects cone
        float t1 = (-b + fsqrt(d)) / (2.0*a);
        float t2 = (-b - fsqrt(d)) / (2.0*a);

        // check to see if intersection is within cone body //
        y = Ry + t1*Dy;
        if ((y < -h/2.0) || (y > h/2.0)) t1 = -1;
        y = Ry + t2*Dy;
        if ((y < -h/2.0) || (y > h/2.0)) t2 = -1;

        if (t2 <= 0) {
            if (t1 <= 0) return HUGE;
            else t = t1;
        } else t = t2;
    }
}
// construct surface normal //

surfnOS = rayorigOS + t * raydirOS;
surfnOS[1] = fsqrt(surfnOS[0] * surfnOS[0] + surfnOS[2] * surfnOS[2]) * r/h;
surfnOS.normalize();
return t;
}

/*
// ivlntersectCone():
// intersect ray with SoCone
// returns object space t of intersection point, or HUGE if none
// surfnOS is assigned iff returned intersection parameter t < HUGE
*/

static float
ivIntersectCone ( ObjectData *obj, Point &rayorigOS, Vector &raydirOS,
OUTPUT Vector &surfnOS )
{
    SoCone *cone = (SoCone *)obj->object;
    // AAA -- fill in code for ray/cone intersection
    // the cone is centered at the origin, has a
    // radius of cone->bottomRadius.getValue() at its
    // base, and a height of cone->height.getValue().
    // the cone's symmetry axis is the *Y* axis.
    // assume that the cone's "parts" field is ALL,
    // that is, that the cone has a bottom circular
    // "caps", and that this cap can cause a ray/object
    // intersection to occur
    // CCC -- for extra credit, handle parts correctly.

    Vector norm;
    float t1 = HUGE;
    float t2 = HUGE;

    if (cone->hasPart(SoCone::SIDES)) {
        t1 = ivIntersectConeBody (cone->bottomRadius.getValue(),
        cone->height.getValue(), rayorigOS, raydirOS, norm);
    }

    if (cone->hasPart(SoCone::BOTTOM)) {
        t2 = ivIntersectDisk (cone->bottomRadius.getValue(),
        -.5*cone->height.getValue(), rayorigOS, raydirOS);
    }

    float t;  // return value
    if (t1 <= t2) {
        t = t1;
        surfnOS = norm;
    } else if (t2 < t1) {
        t = t2;
        surfnOS = Vector(0,-1,0);
else return HUGE;
return t;
}

/* ivIntersectPlane():
//
// intersect ray with plane (Hx,Hy,Hz guaranteed to be normalized)
//
// returns object space t of intersection point, or HUGE if none
// surfnOS is assigned iff returned intersection parameter t < HUGE
*/
static float ivIntersectPlane ( Plane &planeOS, Point &rayorigOS, Vector &raydirOS,
OUTPUT Vector &surfnOS )
{
  // AAA -- compute ray/plane intersection
  // hint 1: start with the next few lines.
  // hint 2: use the given function distance()
  Vector Hnormal(planeOS[0], planeOS[1], planeOS[2]);
  float HdotD = raydirOS.dot (Hnormal);

  // ray nearly parallel to plane
  if ( fabsf (HdotD) < 1.0E-6F )
    return HUGE;

  float t = -distance(planeOS,rayorigOS) / HdotD;
  if (t < 0) return HUGE; // surface pointing wrong way

  // construct surface normal //
  surfnOS = Hnormal;
  surfnOS.normalize();
  return t;
}

/* ivIntersectFace():
//
// intersect ray with single face, assumed to be planar
//
// returns object space t of intersection point, or HUGE if none
// surfnOS is assigned iff returned intersection parameter t < HUGE
*/
static float ivIntersectFace ( int nVertices, Point *,
Plane &plane, Plane *fedgeplanes,
Point &rayorigOS, Vector &raydirOS,
OUTPUT Vector &surfnOS )
{
  // AAA -- you need not write ray/face intersection;
  // it is supplied for you. however, you may wish to
// CCC --- optimize the face intersection code below
// you may need to alter the defs in RayTraceTypes.h
assert ( nVertices > 2 );

// extract face normal from face plane
Vector fnormal(fplane[0], fplane[1], fplane[2]);

// intersect ray with face plane to find intersection point
Vector hitn;
float thit = ivIntersectPlane ( fplane, rayorigOS,
                                 raydirOS, hitn );
if( thit == HUGE )
  return HUGE;

// check point for inclusion in each edge bounding plane
int j = nVertices - 1;
Point hitpt = rayorigOS + thit * raydirOS;
for ( int k = 0; k < nVertices; k++ ) {
  // hit point must be above edge plane
  if( distance ( fedgeplanes[k], hitpt ) <= 0 )
    return HUGE;
  // advance
  j = k;
}

// inside all edges
surfnOS = hitn;
return thit;

/* ivIntersectFaceSet():
   // intersect ray with indexed face set
   // returns object space t of intersection point, or HUGE if none
   // surfnOS is assigned iff returned intersection parameter t < HUGE
   */
static float
ivIntersectFaceSet ( ObjectData *obj, Point &rayorigOS,
                     Vector &raydirOS,
                     OUTPUT Vector &surfnOS )
{
  // AAA --- this code is supplied for you.
  float thit, tmin = HUGE;

  Vector hitn;
  // check each face for intersection
  for ( int f = 0; f < obj->nfaces; f++ ) {
    thit = ivIntersectFace ( obj->flengths[f], obj->fvertices[f],
                            obj->fplanes[f], obj->fedgeplanes[f],
                            rayorigOS, raydirOS, hitn );
    if ( thit < tmin ) {
      surfnOS = hitn;
  }
```c
float tminWS = HUGE; // from <math.h>
Vector hitnOS, surfnOS;
hit.object = NULL; // assume no intersection

// CCC -- this loop checks all N scene objects at octree root
// for extra credit, implement a more sophisticated recursive
// search along the lines of the Traverse() routine in lecture
ObjectData **objects = (ObjectData **)octree->ItemList0;
int nObjects = octree->NumItems0;

// loop over each object, checking for ray-object intersection.
for (int k = 0; k < nObjects; k++) { // for each object
    ObjectData *obj = objects[k];
    Point rayorigOS; Vector raydirOS;
    float thitOS = HUGE;

    // AAA -- check WS ray for intersection with object's WS bbox
    // hint: a call to ivIntersectBbox(), and a continue, suffices
    if (ivIntersectBbox(obj->bboxWS, ray.origWS, ray.dirWS) == HUGE)
        continue;

    // transform world-space origin into object coordinates
    obj->modelMatInv.multVecMatrix( ray.origWS, rayorigOS );

    // transform world-space ray into object coordinates
    // it is a direction vector, so suppress translations
    obj->modelMatInv.multDirMatrix( ray.dirWS, raydirOS );

    // A call to ivIntersectO()
    // this should be the only externally visible routine
    // given a ray origin, a (possibly non-normalized) direction, and
    // a list of scene objects, return 1 iff ray intersects scene
    // calls ivIntersect*, where * is the set of supported primitives
    // returns hit information in Hit structure
    int ivIntersect ( Ray &ray, // R, D, etc.
                     Octree *octree, // scene octree
                     OUTPUT Hit &hit ) // hit structure
    {
        // ...code...
    }

    // face
    // hit, miss
    return tmin;
}
```
// keep track of ratio between WS and OS distance metrics. note
// that we can't do this by checking matrix determinant, since
// aggregate scaling might be anisotropic (direction dependent)
float nlengthOS = raydirOS.length();

// normalize to account for absolute or anisotropic scaling
raydirOS.normalize();

// AAA -- check OS ray for intersection with object's OS bbox
// hint: a call to ivIntersectBbox(), and a continue, suffices
if (ivIntersectBbox(obj->bboxOS, rayorigOS, raydirOS) == HUGE) continue;

// use a sequential "if" dispatches to intersection routine,
// which intersects ray/ primitive to find tOS (object space)
if (obj->object->isOfType (SoCube::getClassTypeId())) {
    // compute intersection with cube
    thitOS = ivIntersectCube ( obj, rayorigOS, raydirOS, hitnOS );
}
else if (obj->object->isOfType (SoSphere::getClassTypeId())) {
    // compute intersection with sphere
    thitOS = ivIntersectSphere ( obj, rayorigOS, raydirOS, hitnOS );
}
else if (obj->object->isOfType (SoCylinder::getClassTypeId())) {
    // compute intersection with cylinder
    thitOS = ivIntersectCylinder ( obj, rayorigOS, raydirOS, hitnOS );
}
else if (obj->object->isOfType (SoCone::getClassTypeId())) {
    // compute intersection with cone
    thitOS = ivIntersectCone ( obj, rayorigOS, raydirOS, hitnOS );
}
else if (obj->object->isOfType (SoIndexedFaceSet::getClassTypeId())) {
    // compute intersection with indexed face set
    thitOS = ivIntersectFaceSet ( obj, rayorigOS, raydirOS, hitnOS );
}
else if (obj->object->isOfType (SoIndexedNurbsSurface::getClassTypeId())) {
    // compute intersection with nurb patch
    thitOS = ivIntersectNurb ( obj, rayorigOS, raydirOS, hitnOS );
}
else {
    if (RayTrace::debug > 1)
cout << "ivTrace(): can't intersect with primitive "
<< obj->object->getTypeId().getName().getString() 
<< endl;
}

// check for intersection; update if closer than closest
if ( thitOS < HUGE ) {
  // compensate for OS/WS scaling
  float thitWS = thitOS / nlengthOS;
  if ( thitWS < tminWS ) {
    tminWS = thitWS;  // retain closest
    hit.object = obj;  // retain object id
    surfnOS = hitnOS;  // retain OS normal
  } // thitWS < tminWS
  } // thitOS < HUGE
}

// for each object
if ( hit.object ) {
  // construct world-space ray-surface intersection point
  hit.surfpWS = ray.origWS + tminWS * ray.dirWS;

  // transform normal vector from object to world coords
  // it transforms as a normal -- i.e., by the inverse
  // transpose of the matrix modelMat that takes OS to WS
  // note that we use the method multDirMatrix(), rather
  // than multVecMatrix(), because the normal [a b c]
  // is equivalent to the homogenous point [a b c 0]
  // (see 8 september and 18 october lecture notes)
  hit.object->modelMatInvTrans.multDirMatrix( surfnOS, hit.surfnWS );

  // normalize to adjust for any anisotropic scaling
  hit.surfnWS.normalize();

  // decide whether ray is enterong or leaving object
  // by convention, object surface normals point "out",
  // so if we are leaving the object, negate our normal
  // so that shading, reflections and refractions work
  if ( !(hit.entering = ( ray.dirWS.dot(hit.surfnWS) < 0 )))
    hit.surfnWS = -hit.surfnWS;
}

// return 1 iff hit
return (hit.object) ? 1 : 0;
}

// IntersectBbox.h
#ifndef _RT_INTERSECT_BBOX_
#define _RT_INTERSECT_BBOX_
#include "RayTraceTypes.h"

#include "RayTraceTypes.h"

149
float
divIntersectBbox ( const Bbox &box, Point &rayorigOS, Vector &raydirOS,
OUTPUT Vector &surfnOS );

#endif /* _RT_INTERSECT_BBOX */

// IntersectBbox.C

// system include files
#include <math.h>
#include <stdlib.h>    // for rand()
#include <assert.h>
#include <iostream.h>

// Inventor include files
#include <Inventor/SbBasic.h>
#include <Inventor/SbLinear.h>
#include <Inventor/SbBox.h>

// local include files
#include "MathUtils.h"
#include "RayTraceTypes.h"
#include "IntersectBbox.h"

/* divIntersectBbox():
   
   intersect ray with axial bounding box
   
   returns object space t of intersection point, or HUGE if none
   surfnOS is assigned iff returned intersection parameter t < HUGE
*/
float
divIntersectBbox ( const Bbox &box, Point &rayorigOS, Vector &raydirOS,
OUTPUT Vector &surfnOS )
{
   // AAA -- this code is supplied for you.
   // CCC -- for extra credit, optimize it.
   // (it's fast as is; can you do better?)

   // parallelepiped is centered at origin
   // get half--width, height, depth of box
   Vector min = box.getMin();
   Vector max = box.getMax();

   // ||epiped must have non--zero volume
   assert ( ( max[0] > min[0] )
            && ( max[2] > min[2] ) );
// ray is of form rayorigOS + t * raydirOS
// find smallest t such that ray intersects |epiped.
float tmin = -HUGE, tmax = HUGE;  // from <math.h>
float t1, t2;

static Vector axes[3] = { Vector (1, 0, 0),
                        Vector (0, 1, 0), Vector (0, 0, 1) };

char maxNormSet = 0, minNormSet = 0;
Vector maxNorm, minNorm;

// intersect ray with x,y,z slabs
for (int k = 0; k < 3; k++) {
    // check ray against planes xk = -dx_k, xk = +dx_k
    if (raydirOS[k] != 0.) {
        t1 = (min[k] - rayorigOS[k]) / raydirOS[k];
        t2 = (max[k] - rayorigOS[k]) / raydirOS[k];

        tmin = fmax(tmin, fmin(t1, t2));
        tmax = fmin(tmax, fmax(t1, t2));

        if (t1 > 0 && tmin == t1) {  // neg face
            minNorm = -axes[k];
            minNormSet = 1;
        }
        else if (t2 > 0 && tmin == t2) {  // pos face
            minNorm = axes[k];
            minNormSet = 1;
        }
        else if (t1 > 0 && tmax == t1) {  // neg face
            maxNorm = -axes[k];
            maxNormSet = 1;
        }
        else if (t2 > 0 && tmax == t2) {  // pos face
            maxNorm = axes[k];
            maxNormSet = 1;
        }
    }

}  // if
}  // k

// tmin..tmax now define intersection with parallelepiped
if (tmin >= tmax)  // intersection is empty or a point
    return HUGE;
else if (tmin > 0) {
    assert(minNormSet);
    surfnOS = minNorm;
    return tmin;
}
else if (tmax > 0) {
    assert(maxNormSet);
    surfnOS = maxNorm;
    return tmax;
}
else  // no hit
return HUGE;
}

// IntersectNurb.h

#ifndef _RT_INTERSECT_NURB
#define _RT_INTERSECT_NURB

#include "RayTraceTypes.h"

/* ivIntersectNurb():
// intersect ray with indexed Nurbs surface
// returns object space t of intersection point, or HUGE if none
// surfnOS is assigned iff returned intersection parameter t < HUGE */
float ivIntersectNurb ( ObjectData *obj, Point &rayorigOS, Vector &raydirOS,
OUTPUT Vector &surfnOS );

#endif /* !_RT_INTERSECT_NURB_ */

// Intersect.C

// system include files
#include <math.h>
#include <stdlib.h>  // for rand()
#include <assert.h>
#include <iostream.h>

// Inventor include files
#include <Inventor/SbBasic.h>
#include <Inventor/SbColor.h>
#include <Inventor/SbLinear.h>
#include <Inventor/SbBox.h>
#include <Inventor/SbViewportRegion.h>
#include <Inventor/SbPickedPoint.h>
#include <Inventor/nodes/SoSelection.h>
#include <Inventor/actions/SoRayPickAction.h>
#include <Inventor/nodes/SoSeparator.h>
#include <Inventor/nodes/SoShape.h>
#include <Inventor/nodes/SoIndexedNurbsSurface.h>

// local include files
#include "MathUtils.h"
#include "RayTraceTypes.h"
#include "IntersectNurb.h"
/* ivIntersectNurb():

// intersect ray with indexed Nurbs surface

// returns object space t of intersection point, or HUGE if none
// surfnOS is assigned iff returned intersection parameter t < HUGE

*/

float ivIntersectNurb ( ObjectData *obj, Point &rayorigOS, Vector &raydirOS,
OUTPUT Vector &surfnOS )
{
    #if 0
    SoIndexedNurbsSurface *nurb = (SoIndexedNurbsSurface *)obj->object;

    // this code performs ray--NURB intersection using Inventor.
    // however, Inventor's nurb implementation seems broken.
    // so, until SGI sends us a fix, we can't ray trace NURBs.

    // construct inventor ray picking object for dummy viewport
    SoRayPickAction pick = SoRayPickAction( SbViewportRegion ( 100, 100 ) );

    // instruct pick object to select the closest scene
    // object along the ray, if any intersection exists
    pick.setPickAll( FALSE );

    // set the pick action's OS ray
    pick.setRay( rayorigOS, raydirOS, -1.0, -1.0 );

    // search flattened scene graph for intersection
    // using Inventor's pre-fabricated pick method
    static SoSelection *sroot = 0;
    if ( !sroot ) {
        sroot = new SoSelection;
        sroot->ref();
    }

    // add coordinate node
    if ( obj->coordinate3 )
        sroot->addChild( obj->coordinate3 );
    else if ( obj->coordinate4 )
        sroot->addChild( obj->coordinate4 );

    // add the nurb itself
    sroot->addChild( (SoNode *)nurb );

    // apply the pick action
    pick.apply( nurb );

    // remove the nurb
    sroot->removeChild( (SoNode *)nurb );

    // remove coordinate node
    if ( obj->coordinate3 )
        sroot->removeChild( obj->coordinate3 );
else if ( obj->coordinate4 )
    sroot->removeChild( obj->coordinate4 );

    // get intersection results into pp
    SoPickedPoint *pp = pick.getPickedPoint();
    if ( pp ) { /* found intersection */
        // extract intersection point from pp
        Point surfpOS = pp->getPoint();

        // recover thitOS from surface point, ray origin
        float thitOS = (surfOS - rayorigOS).length();

        // extract surface normal from pp
        Vector surfnOS = pp->getNormal();

        return thitOS;
    }
}
#endif

    // no hit
    return HUGE;
}
// adjust H.d to contain p
int plane_from_three_points ( Point &a, Point &b, Point &c, Plane &H );

// construct plane from point p and normal n
void plane_from_point_normal ( Point &p, Vector &n, Plane &H );

// construct plane containing p, parallel to par, perpendicular to perp
int plane_from_point_perp_par ( Point &p, Vector &par,
                                Vector &perp, OUTPUT Plane &H );

#endif /* !RT_MATHUTILS */

// MathUtils.C: math utility functions

#include "MathUtils.h"

// min two floats
float fmin ( float a, float b )
{
    return a < b ? a : b;
}

// max two floats
float fmax ( float a, float b )
{
    return a > b ? a : b;
}

// return random float in [low..high]
float randr ( float low, float high )
{
    float value;

    value = low + (rand() / ((1<<15)−1.0)) * (high−low);
    return (value);
}

// max component of reflectance
float vmax ( Color color )
{
    return fmax (color[0], fmax (color[1], color[2]));
}

// multiply two vectors, componentwise, to produce vector
Vector vmult ( Vector &a, Vector &b )
{
    return Vector ( a[0] * b[0], a[1] * b[1], a[2] * b[2] );
}

// perpendicular point/plane distance
float distance ( Plane &H, Point &p )
{
}
int plane_from_three_points ( Point &a, Point &b, Point &c, Plane &H )
{
    Vector n = ( b - a ).cross( c - a );
    if ( n.length() < 1.0E-6F )
        return 0;
    n.normalize();
    H = Plane(n[0], n[1], n[2], -n.dot(a));
    return 1;
}

void plane_from_point_normal ( Point &p, Vector &n, Plane &H )
{
    H = Plane(n[0], n[1], n[2], -n.dot(p));
}

int plane_from_point_perp_par ( Point &p, Vector &par, Vector &perp, OUTPUT Plane &H )
{
    // H's normal is perp
    Vector normal = perp.cross(par);
    if ( normal.length() < 1.0E-6F )
        return 0;

    // adjust H.d to contain p
    normal.normalize();
    plane_from_point_normal ( p, normal, H );
    return 1;
}
Appendix E

Concurrent Ray Trace Functions and Data Objects

/***** TraceProc.h *****/

#ifndef _TraceProc_
#define _TraceProc_ 

#include "iv/Octree.h"
#include "compQ.h"
#include "lock.h"

class TraceArg {
   public:
   TraceArg(compQ *c, OctreeP o, LightData *l, int n, int i, lock *lk):
      _compQ(c), _Octree(o), Index(i), syncLock(lk), _LightData(l),
      nLights(n) {};

   compQ *compQ;
   OctreeP _Octree;
   LightData *_LightData;
   int nLights;
   int Index;
   lock *syncLock;
};

/* TraceProcIncremental will take RIC's off the Q and will subsequently 
delegate the responsibility of computing the calculations in the 
convergence by placing new RIC's back in the Q for other TraceProcs to 
compute. */

void TraceProcIncremental(void *);
void TraceProcRecurse(void *);

#include <sys/types.h>
#include <iostream.h>
#include "iv/RayTraceTypes.h"
#include "iv/Intersect.h"
#include "iv/Trace.h"
#include "iv/Shade.h"
#include "ConvObj.h"
#include "CTree.h"
#include "TraceProc.h"
#include "supCTree.h"

extern pid_t *TProcPidArray;
extern int *TProcComps;
extern int NumHits, NumMiss; // global hit/miss counts

void TraceProcIncremental(void *args)
{
    cerr << "TraceProcIncrement Spawned!" << endl;

    compQ *_compQ = ((TraceArg *) args)->_compQ;
    OctreeP _Octree = ((TraceArg *) args)->_Octree;
    LightData *lights = ((TraceArg *) args)->_LightData;
    int nLights = ((TraceArg *) args)->nLights;
    int index = ((TraceArg *) args)->Index;
    lock *syncLock = ((TraceArg *) args)->syncLock;
    syncLock->lock_resource();
    delete (TraceArg *)args;
    pid_t MyPid = getpid();

    RIC_CT * RICp = NULL;
    CTree *ctree = NULL;
    RayPath *rp = NULL;
    Ray *ray = NULL;
    Hit _hit;
    Radiance radiance, NOCONTRIBUTION(0,0,0);
    Vector eta_vacuum(1.0,1.0,1.0);
    BOOLEAN converged = FALSE;
/* keep looping until externally terminated */
for (TProcComps[index] = 0; TProcComps[index]++){
    /* if we don’t already have a computation then get the next one */
    if (RICp == NULL) {
        while (!(_compQ->get_comp((RIC **)&RICp))){
            // wait until the Q is non empty
            _compQ->WaitOnNonEmpty(MyPid);
            /* try to get the next computation
            if we didn’t get one then try again */
        }
    }

    if (RICp == NULL) break;

    _ctree = RICp->_c;

    _ctree->IncRef();
    _rp = RICp->_rp;
    _ray = RICp->_ray;
}

/* if (TProcComps[index]%1000 == 0)
    cout << TProcComps[index] << " comps executed by # " << index
    << " # " << MyPid << " TraceProcIncrement()..." << endl; */

/* determine closest intersection point on RICp’s ray */
if (IvIntersect(*_ray, _Octree, _hit)) {
    ObjectData *obj = _hit.object;
    float weight = _ctree->GetWeight(_rp);

    /* get the local contribution and place it in the CTree
    if I am exiting an object, then there is no local contribution
    at the intersection point of the exit */
    if (_ray->dirWS.dot(_hit.surfnWS) >= 0.0) // exiting object
        _ctree->SetLocal(_rp, NOCONTRIBUTION);
    else {
        // weight spd and depth don’t really matter
        radiance = ivShade(_hit, _ray->dirWS, 0, 1.0, 0, etavacuum,
           _Octree, lights, nLights);
        _ctree->SetLocal(_rp, radiance);
    } // end if exiting object
}

// if we are not too deep
if (_rp->depth() < RayTrace::maxdepth) {
    // create a refraction ray and put it in the compQ
    Vector traydirWS;
    float transp = obj->transparency;

    if (transp > 0.0) {
        // collect radiance along transmission ray traydir
        // here, we have encoded the index of refraction for
        // r, g, and b wavelengths in environment “fog” color
        // which has been placed in the object’s “eta” field
        Vector traydirWS;
        Vector etavacuum(1.0, 1.0, 1.0);
// BBB -- compute the transmission direction(s) with this routine
// (you may wish to compute transm rays at each of R, G, and B)
// check the Ray.spd[] mask for radiance at R, G, B
// R: tir = TransmissionDirection (viewdWS, hit.surfnWS,
//     eta[0], obj->eta[0], traydirWS);
// G:
// B: etc.

int tir = 0;
for (int rgb = 0; rgb < 3; rgb++) {  // loop through R, G, B
  if (!ray->spd[rgb]) {
    ctree->SetRefracted(rgb, _rp, NOCONTRIBUTION);
    continue;
  }
  if (ray->dirWS.dot(hit.surfnWS) < 0.0) {  // entering object
    tir = TransmissionDirection (_ray->dirWS, _hit.surfnWS,
                                eta_vacuum[rgb], obj->eta[rgb],
                                traydirWS);
  } else {
    // exiting object
    tir = TransmissionDirection (_ray->dirWS, -_hit.surfnWS,
                                 obj->eta[rgb], eta_vacuum[rgb],
                                 traydirWS);
  }
  if (!tir && (transp*weight >= RayTrace::minweight)) {
    // BBB -- collect radiance along tray, multiplied by k_trans
    // transpar 0 => opaque; transpar 1 => completely transparent
    traydirWS.normalize();
    Ray *tray = new Ray(_hit.surfpWS+.001*traydirWS, traydirWS);
    tray->spd[rgb] = 1;
    RayPath *new_rp = _rp->duplicate();
    if (!new_rp->down_refraction(rgb))
      cerr << "Error in down_refraction: " << rgb << endl;
    ctree->SetWeightScale(new_rp, transp*weight, transp);
    ctree->SetSPD(new_rp, tray->spd[0], tray->spd[1], tray->spd[2]);
    RIC_CT *ReFrRIC = new RIC_CT(tray, _ctree, new_rp);
    _compQ->add_comp(ReFrRIC);
  } else
    ctree->SetRefracted(rgb, _rp, NOCONTRIBUTION);
}
} else {  // if there is no refraction
  ctree->SetRefracted(rbiRedRefrBranch, _rp, NOCONTRIBUTION);
  ctree->SetRefracted(rbiGreenRefrBranch, _rp, NOCONTRIBUTION);
  ctree->SetRefracted(rbiBlueRefrBranch, _rp, NOCONTRIBUTION);
}

Vector rraydirWS;
int ref = ReflectionDirection (_ray->dirWS, _hit.surfnWS, rraydirWS);
if (ref && (obj->ks*weight >= RayTrace::minweight)) {
/ * if this happens then on the next iteration I will be calculating the reflection. So set up RIC then continue */
raydirWS.normalize();
Ray *ray = new Ray(_hit.surfWS+.001*raydirWS, raydirWS);
_rp->down_reflection();  // no need to create a new ray path
_ctree->SetWeightScale(_rp, obj->ks*weight, obj->ks);

// I can delete the old ray
delete _ray;
// RICp->_ctree and _ctree are already set
// RICp->_rp and _rp are is already set
RICp->_ray = ray;
_ray = ray;

} else { // if there is no reflection
converged = _ctree->SetReflected(_rp, NOCONTRIBUTION, TRUE);
// I can delete the old _rp, RICp and old ray
delete _ray;
delete _rp;
delete RICp;
RICp = NULL;
} // end if there is reflection

} else { // we are too deep
// put no contributions in for the reflected and refracted rays
_ctree->SetRefracted(rbiRedRefrBranch, _rp, NOCONTRIBUTION);
_ctree->SetRefracted(rbiGreenRefrBranch, _rp, NOCONTRIBUTION);
_ctree->SetRefracted(rbiBlueRefrBranch, _rp, NOCONTRIBUTION);
converged = _ctree->SetReflected(_rp, NOCONTRIBUTION, TRUE);
// delete the ray and rp from the RIC
delete _ray;
delete _rp;
delete RICp;
RICp = NULL;

} // end if intersection

else { // if no object was hit
NumMiss++;
_ctree->SetLocal(_rp, NOCONTRIBUTION);
_ctree->SetRefracted(rbiRedRefrBranch, _rp, NOCONTRIBUTION);
_ctree->SetRefracted(rbiGreenRefrBranch, _rp, NOCONTRIBUTION);
_ctree->SetRefracted(rbiBlueRefrBranch, _rp, NOCONTRIBUTION);
converged = _ctree->SetReflected(_rp, NOCONTRIBUTION, TRUE);
delete _ray;
delete _rp;
delete RICp;
RICp = NULL;
} // end if intersection

if (converged) {
// if by setting a value we were able to converge the CTree
if (!_ctree->WriteOut())
    perror("TraceProc had a problem calling _ctree->WriteOut()");
converged = FALSE;
_ctree->DecRef();

}
delete _ctree;
} else if (RICp == NULL)
    /* if RICp == NULL and we didn't converge then we
       are no longer referring to _ctree
    */
    _ctree->DecRef();
    // end if converged
}
} // for loop

cerr << MyPid << " exiting!
";
syncLock->unlock_resource();
exit(0);
} // traceprocincrement

/* TraceProcRecurse calls ivTrace, so the output is the same as if we
ran the old code. */

void TraceProcRecurse(void *args)
{
    cerr << "TraceProcRecurse Spawned!" << endl;

    compQ * _compQ = ((TraceArg *) args)->compQ;
    OctreeP _Octree = ((TraceArg *) args)->_Octree;
    LightData *lights = ((TraceArg *) args)->_LightData;
    int nLights = ((TraceArg *) args)->nLights;
    int index = ((TraceArg *) args)->Index;
    lock *syncLock = ((TraceArg *) args)->syncLock;
    syncLock->lock_resource();
    delete (TraceArg *)args;
    pid_t MyPid = getpid();

    RIC_CO * RICp = NULL;
    int sample_num;
    Radiance radiance(0,0,0);
    Vector eta_vacuum(1.0,1.0,1.0);
    ConvergenceObject *CO;
    BOOLEAN converged;

    for (TProcComps[index] = 0; TProcComps[index]++) {
        converged = FALSE;
        while (!(_compQ->get_comp((RIC **) & RICp))) {
            // wait until the Q is non empty
            _compQ->WaitOnNonEmpty(MyPid);
            // try to get the next computation
        }
        if (RICp == NULL) break;

        CO = RICp->_c;
        sample_num = RICp->Sample;
        radiance = ivTrace(*(RICp->_ray), 0, 1.0, eta_vacuum,
            _Octree, lights, nLights);
        converged = CO->SetRadiance(radiance, sample_num);
    }
if (converged)
    RICp->c->WriteOut();
}
delete RICp->ray;
delete RICp->c;
delete RICp;
RICp = NULL;

}  
cerr << MyPid << " exiting!\n";
syncLock->unlock_resource();
exit(0);

/***** lock.h *****/

#ifndef lock
#define lock

#define _SGI_MPSOURCE
#include <abi_mutex.h>
#include "boolean.h"

class lock {
public:
    lock();

    // islocked returns TRUE if the resource is locked.
    BOOLEAN islocked(){return ((stat_lock(&lock) == LOCKED) ? TRUE : FALSE);} 
    BOOLEAN lock_resource(BOOLEAN blocking = TRUE);
    void unlock_resource();

private:
    abilock_t_lock;
};

#define _SGI_MPSOURCE
#include <stdio.h>
#include "lock.h"

lock::lock()
{
    if (init_lock(&lock))
        perror("init_lock failed in lock::lock");
}

/* lock.C */

#define _SGI_MPSOURCE
#include <stdio.h>
#include "lock.h"

lock::lock()
{
    if (init_lock(&lock))
        perror("init_lock failed in lock::lock");
}
BOOLEAN lock::lock_resource(BOOLEAN blocking) 
{
    if (blocking) {
        spin_lock(&_lock);
        return TRUE;
    } else
        return (acquire_lock(&_lock) == 0);
}

void lock::unlock_resource() 
{
    if (release_lock(&_lock))
        perror("release_lock error in lock::unlock_resource");
}

/***** llmutex.h *****/

#ifndef _llist_h
#define _llist_h

#include "boolean.h"
#include "lock.h"

// This is a base definition for a simple singly linked list
class llEntry {
public:
    llEntry():next(0) {}
    llEntry *next;
};

// this is a linked list which uses mutex locks to synchronize access
class llmutex {
public:
    llmutex(BOOLEAN owned = TRUE): DeleteOnCompletion(owned), head(0),
        tail(0), _size(0), _EntriesAdded(0) {}

    /* the next function makes a copy of the llEntry that is
    pointed to by the argument. Therefore do not delete the llEntries until
    they are removed from the list and used or copied.
    The last argument in the next function indicates whether the
    lock has been acquired prior to the member function call (or if
    locking is not necessary). If so then the member function will
    not try to acquire any of its locks. Since the locks are
    protected access, this is useful for derived types that want to
    do some special functions on the list and don't want to acquire
    and release locks every time they call these functions */
```c
void add_entry(llEntry *, BOOLEAN AlreadyLocked= FALSE);

llEntry *get_entry();
int size() {return _size;}
int EntriesAdded() { return _EntriesAdded;}

protected:
lock headLock, tailLock, sizeLock;
BOOLEAN DeleteOnCompletion;
llEntry *head, *tail;
int _size;

/* notes: if locks are acquired externally, callers should adhere to
the following locking order size tail head */

private:
int _EntriesAdded;
};
#endif

//-- llmutex.C

#include "llmutex.h"
#include "lock.h"
#include "boolean.h"

void llmutex::add_entry(llEntry *e, BOOLEAN AlreadyLocked)
{
    if (AlreadyLocked) {
        if (_size++ == 0)
            head = tail = e;
        else {
            tail->next = e;
            tail = e;
        }
        _EntriesAdded++;
    } else {
        // if I need to lock resources
        sizeLock.lock_resource();
        tailLock.lock_resource();
        if (_size++ == 0) {
            // if the list was empty
            headLock.lock_resource();
            head = e;
            headLock.unlock_resource();
            tail = e;
        } else { // if it wasn't empty
            tail->next = e;
            tail = e;
        }
    }
```
EntriesAdded++;
tailLock.unlock_resource();
sizeLock.unlock_resource();
}

llEntry *llmutex::get_entry()
{
    // check if the Q is nonempty
    if (_size == 0) return 0;

    // lock the_size
    sizeLock.lock_resource(TRUE);

    // recheck_size
    if (_size == 0) {
        sizeLock.unlock_resource();
        return 0;
    }

    // manipulate_size
    _size--;

    // lock, manipulate and unlock top
    headLock.lock_resource();
    llEntry *e = head;
    head = head->next;
    headLock.unlock_resource();
    sizeLock.unlock_resource();

    return e;
}

// compQ.h
#ifndef compQ_
#define _compQ
#include <sys/types.h>
#include "iv/RayTraceTypes.h"
#include "llmutex.h"
#include "lock.h"
#include "CTree.h"
#include "boolean.h"
#include "ConvObj.h"

/*************** Ray Interception Computation (RIC) ********************/

class RIC {
public:
    Ray *ray;

    RIC(Ray *r = NULL): _ray(r){}
};
typedef RIC *RICp;

// an ric that has a ctree
class RIC_CT : public RIC {
    public:
        CTree * _c;
        RayPath * _rp;

        RIC_CT(Ray * r= NULL, CTree * c= NULL, RayPath * rp = NULL):
                    RIC(r), _c(c), _rp(rp) {}  
};

// an ric that has a convergence object
class RIC_CO : public RIC {
    public:
        ConvergenceObject * _c;
        int Sample;

        RIC_CO(Ray * r = NULL, ConvergenceObject * c = NULL, int s = 0):
                    RIC(r), _c(c), Sample(s) {}  
};

class compQEntry: public llEntry {
    public:
        compQEntry(RIC * r = NULL): llEntry(), _ric(r) {};
        RIC * _ric;
    };

class pidEntry: public llEntry {
    public:
        pidEntry(pid_t p): llEntry(), PID(p), wait_lock(0) {};
        lock * wait_lock;
        pid_t PID;
    };

class PidAwakenList: public llmutex {
    public:
        PidAwakenList(): llmutex() {};
        ~PidAwakenList();

        void AwakenNext(int n = 1);
        lock * add_entry(pidEntry *);
        pidEntry * get_entry() {return (pidEntry *)llmutex::get_entry();}  
    };

class compQ: public llmutex {
    public:
        compQ(): llmutex(), _WaitOnNonEmptyList(), SizeLessThan(0) {};
        ~compQ();
/* both of the following will awaken processes that are waiting for
computations to execute */
void add_comp(RIC *
);
void add_comp(RIC **, int);

BOOLEAN get_comp(RIC **);

// WaitOnEmpty is only called by RayExt.
void WaitOnSizeLessThan(int = 0);
void WaitOnEmpty() {WaitOnSizeLessThan();}
// WaitOnNonEmpty is called by everyone else, so send a pid
void WaitOnNonEmpty(pid_t);

private:
  PidAwakenList _WaitOnNonEmptyList;
  lock SizeLessThanLock;
  int SizeLessThan;

  /* Empty is called when the compQ becomes empty as a result of a
get_comp call */
  void Empty()
  {
    if (SizeLessThanLock.islocked())
        SizeLessThanLock.unlock_resource();
  }

#endif

// compQ.C

#define _SGI_MP_SOURCE
#include <iostream.h>
#include "compQ.h"
#include "boolean.h"

extern pid_t RayExtPid;
extern int waitTOT;

compQ::~compQ()
{
  if (!DeleteOnCompletion || !_size) return;

  if (_size == 1) {
    delete head;
    return;
  }
llEntry *n = head->next;
delete head;
while (n) {
    head = n;
    n = n->next;
    delete head;
}
}

void compQ::add_comp(RIC **rs, int n) 30
{
    // acquire all locks
    sizeLock.lock_resource();
    tailLock.lock_resource();
    headLock.lock_resource();

    for (int k = 0; k < n; k++)
        add_entry((llEntry *) new compQEntry(rs[k]), TRUE);

    headLock.unlock_resource();
    tailLock.unlock_resource();
    sizeLock.unlock_resource();

    // wake the next n processes
    WaitOnNonEmptyList.AwakenNext(n);
}

void compQ::add_comp(RIC *r) 50
{
    add_entry((llEntry *) new compQEntry(r));

    // wake the next waiting processes
    WaitOnNonEmptyList.AwakenNext();
}

BOOLEAN compQ::get_comp(RIC **r) 60
{
    compQEntry *e = (compQEntry *) get_entry();
    if (!e) {
        *r = NULL;
        return FALSE;
    }
    *r = e->ric;
    delete e;

    sizeLock.lock_resource();
    if (size() <= SizeLessThan) {
        sizeLock.unlock_resource();
        // if the Q is now empty then call the empty function
        Empty();
    } else sizeLock.unlock_resource();

    return TRUE;
```cpp
void compQ::WaitOnSizeLessThan(int ws)
{
    // lock, get and unlock the size
    sizeLock.lock_resource();
    int s = size();

    if (s <= ws) {
        sizeLock.unlock_resource();
        return;
    }

    SizeLessThan = ws;
    // lock the waitonempty resource
    SizeLessThanLock.lock_resource();
    // unlock the size so that add_comp is enabled
    sizeLock.unlock_resource();
    // * then try and lock it again. this will work when the resource is
    // unlocked by add_comp */
    SizeLessThanLock.lock_resource();

    // then unlock the resource and go.
    SizeLessThanLock.unlock_resource();
}

void compQ::WaitOnNonEmpty(pid_t pid)
{
    // lock get and unlock the size
    sizeLock.lock_resource();
    int s = size();

    // if the Q is already nonempty then unlock and return
    if (s > 0) {
        sizeLock.unlock_resource();
        return;
    }

    // otherwise add myself to the wait list
    lock * wait_lock = _WaitOnNonEmptyList.add_entry(new pidEntry(pid));
    // unlock the size so that add_comp is enabled
    sizeLock.unlock_resource();
    // try to acquire the wait_lock resource
    wait_lock->lock_resource();
    // * This will return when the Q is nonempty at that time deallocate
    // the lock */
    delete wait_lock;
    waitTOT++;
}

PidAwakenList::~PidAwakenList()
{
    pidEntry *old;
}
```
if (_size > 0) && (DeleteOnCompletion)) {
    while (head->next) {
        old = (pidEntry *) head;
        head = head->next;
        delete old;
    }
    delete head;
}

void PidAwakenList::AwakenNext(int n) {
    pidEntry *pidE = (pidEntry *) get_entry();

    while ((n > 0) && pidE) {
        // unlock the wait_lock
        pidE->wait_lock->unlock_resource();
        delete pidE;
        if (--n)
            pidE = (pidEntry *)get_entry();
    }
}

lock * PidAwakenList::add_entry(pidEntry *e) {
    if (!e->wait_lock) e->waitlock = new lock;
    llmutex::addentry((llEntry *)e);
    e->wait_lock->lock_resource();
    return e->wait_lock;
}

/***** ConvObj.h *****/

#ifndef_ConvObj_h_
#define_ConvObj_h_

#include "boolean.h"
#include "lock.h"
#include "iv/RayTraceTypes.h"

class ConvergenceObject {
public:
    ConvergenceObject():x(0),y(0),FinalResult(NULL),CONVERGE(1) {} 
    ConvergenceObject(int _x, int _y);
    virtual ~ConvergenceObject();

    virtual BOOLEAN SetRadiance(Radiance r, int)
    {(*(FinalResult[0]) = r;return TRUE;}
    virtual BOOLEAN WriteOut();

    // this will block the caller until Instances == 0
    static void (* SetPixel)(int, int, SbColor);
    static int Instances;
}
static void WaitOnNoInst();
static lock InstanceLock;

protected:
int x, y;
Radiance **FinalResult;
static lock ZeroInstWaitLock;
BOOLEAN CONVERGE;
};

#endif

// ConvObj.C

#include <iostream.h>
#include <unistd.h>
#include "Conv0bj.h"
#include "lock.h"

int ConvergenceObject::Instances = 0;
lock ConvergenceObject::InstanceLock;
lock ConvergenceObject::ZeroInstWaitLock;
void NoInstSigHandler(...);
extern pid_t RayExtPid;

void (* ConvergenceObject::SetPixel)(int, int, SbColor) = 0;

ConvergenceObject::ConvergenceObject(int _x, int _y): x(x), y(_y), CONVERGE(1)
{
    Instances++;

    FinalResult = new Radiance * [1];
    FinalResult[0] = new Radiance(0, 0, 0);
}

ConvergenceObject::~ConvergenceObject()
{
    Instances--;

    /* if there are no more instances, and the zeroinstwaitlock is locked
     * then unlock it */
    if ((Instances == 0) && (ZeroInstWaitLock.islocked()))
        ZeroInstWaitLock.unlock_resource();

    delete FinalResult[0];
    delete [] FinalResult;
}

BOOLEAN ConvergenceObject::WriteOut()
{
    SetPixel(x, y, *(FinalResult[0]));
    return TRUE;
void ConvergenceObject::WaitOnNoInst()
{
    if (Instances==0) { // if the size is empty
        return; // and return
    }

    ZeroInstWaitLock.lock_resource();
    cerr << "Waiting for no convergence instances..." << endl;
    /* when I can lock the resource for the second time that means that
        it has been unlocked in the convobj deallocation routine */
    ZeroInstWaitLock.lock_resource();
    ZeroInstWaitLock.unlock_resource();
}

/***** Boolean.h *****/

#ifndef _boolean_
#define _boolean_

#define BOOLEAN int
#define TRUE 1
#define FALSE 0

#endif

/***** CTree *****/

#ifndef _CTree_
#define _CTree_

#define _SGI_MP_SOURCE
#include <sys/types.h>
// #include <unistd.h>
#include <fstream.h>
#include "iv/RayTraceTypes.h"
#include "lock.h"
#include "boolean.h"
#include "ConvObj.h"

#define MaxDepth 10

// ray path branch types
#define rpRAYSOURCE 1
#define rpRAYREFLECTION 2
#define rpREDRAYREFRACTION 3
#define rpGREENRAYREFRACTION 4
#define rpBLUERAYREFRACTION 5

// ctree node flag masks
```cpp
#define ctnNOTHINGSET 0
#define ctnREFLECTSET 1
#define ctnGREFRACTSET 2
#define ctnRREFRACTSET 4
#define ctnBREFRACTSET 8
#define ctnLOCALSET 16
#define ctnCONVERGED 32

// Refr branch indices (types)
#define rbiRedRefrBranch 0
#define rbiGreenRefrBranch 1
#define rbiBlueRefrBranch 2

class CTreeNode {
public:
    CTreeNode();
    Radiance composite();

    // the 3 refraction branches correspond to r g b refraction rays
    CTreeNode *ReflBranch, *RefrBranch[3], *parent;
    Radiance ReflContrib, RefrContrib[3], LocalContrib;
    short int NodeFlag;
    unsigned char spd[3];
    float weight, ScalingCoeff;
    lock _lock;
};

class RayPath {
public:
    RayPath(RayPath *source);

    // if recNodes is TRUE then down_re.. will create CTreeNodeRec's
    RayPath(CTreeNode *RootNode, ofstream *f = NULL, lock *fl = NULL)
        : cur_depth(0), TreePointer(RootNode), fout(f), foutLock(fl){}
    int depth() {return cur_depth;}

    virtual RayPath * duplicate() {
        return new RayPath(this);
    }
    virtual CTreeNode * down_reflection(char * = "\0");
    virtual CTreeNode * down_refraction(int, char * = "\0");
    virtual CTreeNode * up_to_parent();
    int CurrentRayType() {return ((cur_depth == 0) ? rpRAYSOURCE:
        path[cur_depth - 1]);}
    CTreeNode * Pointer() { return TreePointer; }
    int * Path() { return path;}

protected:
    int cur_depth, path[MaxDepth];
    CTreeNode *TreePointer;
    ofstream *fout;
    lock *foutLock;
};

class CTree : public ConvergenceObject {
```
public:
CTree(int _x, int _y) : ConvergenceObject(_x, _y),
    _root(new CTreeNode*[1]), ReferenceCounter(0)
{ _root[0] = new CTreeNode; }
CTree() {}
virtual ~CTree();

virtual RayPath *Root(int = 0) {return new RayPath(_root[0]); }

/* if the raypath passed into one of these member functions is not a
raypath that was either returned by CTree::Root() (for this
instance) or by creating a new raypath from a raypath passed back
by CTree::Root(), then the executions of these functions are
unspecified. */

/* if a set member function returns 0 then convergence was
unsuccessful, if a non zero value is returned then convergence was
successful. if the last argument in the set function call is FALSE
then convergence is not attempted */

virtual BOOLEAN SetLocal(RayPath *, Radiance &, BOOLEAN = FALSE);
virtual BOOLEAN SetRefracted(int, RayPath *, Radiance &, BOOLEAN = FALSE);
virtual BOOLEAN SetReflected(RayPath *, Radiance &, BOOLEAN = FALSE);
virtual void SetWeightScale(RayPath *RP, float W, float C) {
    RP->Pointer()->weight = W;
    RP->Pointer()->ScalingCoeff = C;
}
virtual void SetSPD(RayPath *RP, unsigned char r, unsigned char g, unsigned char b) {
    { RP->Pointer()->spd[0] = r;
      RP->Pointer()->spd[1] = g;
      RP->Pointer()->spd[2] = b; }
}
virtual void SetSPD(RayPath *RP, unsigned char *spd) {
    for (int k = 0; k < 3; k++) RP->Pointer()->spd[k] = _spd[k];}

virtual void GetLocal(RayPath * RP, Radiance * R) {
    { (RP->Pointer())->LocalContrib = *R; }
    virtual void GetRefracted(int refindex, RayPath * RP, Radiance * R) {
        { (RP->Pointer())->RefrContrib[refindex] = *R; }
    virtual void GetReflected(RayPath * RP, Radiance * R) {
        { (RP->Pointer())->ReflContrib = *R; }
    virtual float GetWeight(RayPath * RP) {return (RP->Pointer())->weight; }
    virtual float GetScale(RayPath * RP) {
        return (RP->Pointer())->ScalingCoeff; }
    virtual unsigned char *GetSPD(RayPath * RP) {return RP->Pointer()->spd;}

virtual BOOLEAN Converged() {
    { return (_root[0]->NodeFlag == ctnCONVERGED); }
    virtual BOOLEAN WriteOut(); /* this will use SetPixel to write
the converged radiance */

    // these will update the reference counter
    void IncRef();
    void DecRef();
protected:
    CTreeNode **_root;
    lock _Reflock;
    int ReferenceCounter;
    /* ReferenceCounter is used for deallocation. When a processes
    starts to access a CTree in any way, ReferenceCounter
    is incremented. When the process is finished with the tree it
    will decrement the counter. When a processes is able to converge
    a tree all the way up to the root, then that process will wait
    until ReferenceCounter = 1 and will then be able to delete the
    CTree object. This method is not to be used
    indiscriminately. Instead it will be used exclusively in the
    context of setting contributions and converging the
    tree. Otherwise we will depend on the supreme skill of the
    programmer to ensure that destruction of a tree will not result
    in any unspecified behavior.
    */

    /* Converge returns TRUE if total convergence is achieved
    if the second argument is !=NULL(and convergence takes place)
    the resulting radiance will be copied into there. Otherwise...
    the result (upon convergence)will be placed in FinalResult[0]
    */
    virtual BOOLEAN Converge(RayPath *, Radiance * = NULL);
    void DestroyDepthFirst(CTreeNode *);
};

/* This is a CTree that records the pid of the process that wrote
the data in the tree */
class CTreeRec : public CTree {
public:
    CTreeRec(int, int, char *);
    CTreeRec(): CTree(0, 0){}

    virtual RayPath *Root(int = 0)
    {return new RayPath(_root[0], &fout, &foutLock); }

    virtual BOOLEAN SetLocal(RayPath *, Radiance &, BOOLEAN = FALSE);
    virtual BOOLEAN SetReflected(int, RayPath *, Radiance &, BOOLEAN = FALSE);
    virtual BOOLEAN SetRefracted(RayPath *, Radiance &, BOOLEAN = FALSE);
    virtual void SetWeightScale(RayPath *, float, float);
    virtual void SetSPD(RayPath *, unsigned char, unsigned char, unsigned char);

    virtual BOOLEAN WriteOutO;
};
void OutPutPath(ofstream &fout, int *path, int size)
{
    fout << " II";
    for (int k = 0; k < size; k++)
        fout << path[k] << " ";
    fout << endl;
}

// ************* CTreeNode *************
CTreeNode::CTreeNode():
    ReflBranch(NULL), parent(NULL), weight(1.0), ScalingCoeff(1.0),
    NodeFlag(ctnNOTHINGSET)
{
    RefrBranch[rbiRedRefrBranch] = RefrBranch[rbiGreenRefrBranch] =
        RefrBranch[rbiBlueRefrBranch] = NULL;
}

Radiance CTreeNode::composite()
{
    Radiance temp(LocalContrib + ReflContrib);

    if (spd[0])
        temp[0] += RefrContrib[rbiRedRefrBranch][0];
    if (spd[1])
        temp[1] += RefrContrib[rbiGreenRefrBranch][1];
    if (spd[2])
        temp[2] += RefrContrib[rbiBlueRefrBranch][2];

    if (temp[0] > 1.0) temp[0] = 1.0;
    if (temp[1] > 1.0) temp[1] = 1.0;
    return temp * ScalingCoeff;
}

// *************** RayPath definitions ***********************
RayPath::RayPath(RayPath *source)
: cur_depth(source->cur_depth), TreePointer(source->TreePointer),
fout(source->fout), foutLock(source->foutLock)
/ we only need to copy what’s there
{
for (int k = 0; k < source->cur_depth; k++)
  path[k] = source->path[k];
}

CTreeNode * RayPath::down_reflection(char *prefix)
{
  // check if we are at the maximum depth
  if (cur_depth == MaxDepth) return NULL;

  /* if not then add a reflection to the path
     and a new node to the tree. Then set the
     pointer to the new location in the tree */
  path[cur_depth++] = rpRAYREFLECTION;
  if (fout) {
    // place the new path in the output file
    foutLock->lock_resource();
    *fout << getpid() << prefix << " ";
    OutPutPath(*fout, path, curdepth);
    foutLock->unlock_resource();
  }

  /* since this method is used to expand the tree
     we can assume that TreePointer is a leaf in
     the tree. Therefore we can forego checking
     to see if it is a leaf. */
  TreePointer->ReflBranch = new CTreeNode;
  TreePointer->ReflBranch->parent = TreePointer;
  TreePointer = TreePointer->ReflBranch;

  return TreePointer;
}

CTreeNode * RayPath::down_refraction(int RefrInd, char *prefix)
{
  // check if we are at the maximum depth
  if (cur_depth == (MaxDepth - 1)) return NULL;
  if ((RefrInd < 0) || (RefrInd > 2)) return NULL;

  CTreeNode * n = new CTreeNode;
  TreePointer->RefrBranch[RefrInd] = n;
  TreePointer->RefrBranch[RefrInd]->parent = TreePointer;
  TreePointer = TreePointer->RefrBranch[RefrInd];

  switch (RefrInd)
  {
  case rbiRedRefrBranch:
    path[cur_depth] = rpREDRAYREFRACTION;
    break;
  case rbiGreenRefrBranch:
    path[cur_depth] = rpGREENRAYREFRACTION;
    break;
  }
case rbiBlueRefrBranch:
    path[cur_depth] = rpBLUERAYREFRACTION;
    break;
}

cur_depth++;
if (fout) {
    // place the new path in the output file
    foutLock->lock_resource();
    *fout << getpid() << prefix << " ";
    OutPutPath(*fout, path, cur_depth);
    foutLock->unlock_resource();
}
return TreePointer;
}

CTreeNode *RayPath::up_to_parent()
{
    // check to see if TreePointer is at the root of the tree
    if (! (TreePointer->parent)) return NULL;

    // if not set TreePointer
    TreePointer = TreePointer->parent;
    cur_depth--;
    return TreePointer;
}

// ********************** CTree definitions *************************

CTree::CTree()
{
    // wait until no process is refering to this tree
    while (ReferenceCounter > 0);
    DestroyDepthFirst(_root[0]);
    delete _root[0];
    delete [] _root;
}

BOOLEAN CTree::SetLocal(RayPath *RP, Radiance &R, BOOLEAN conv)
{
    CTreeNode * node = RP->PointerO;
    node->LocalContrib = R;
    node->lock.lock_resource();
    node->NodeFlag |= ctnLOCALSET;
    node->lock.unlock_resource();
    // check for convergence
    return (conv)?Converge(RP):FALSE;
}

BOOLEAN CTree::SetReflected(RayPath *RP, Radiance &R, BOOLEAN conv)
{
    CTreeNode * node = RP->PointerO;
    node->ReflContrib = R;
    node->lock.lock_resource();
    node->NodeFlag |= ctnLOCALSET;
    node->lock.unlock_resource();
    // check for convergence
    return (conv)?Converge(RP):FALSE;
}
node->lock.lock_resource();
node->NodeFlag |= ctnREFLECTSET;
node->lock.unlock_resource();
// check for convergence
return (conv)?Converge(RP):FALSE;
}

BOOLEAN CTree::SetRefracted(int refindex, RayPath *RP,
                         Radiance &R, BOOLEAN conv)
{
    if ((refindex < 0) || (refindex > 2)) {
        perror("invalid index given to SetRefraction");
        return FALSE;
    }
    CTreeNode * node = RP->PointerO;
    node->RefrContrib[refindex] = R;
    node->lock.lock_resource();
    switch (refindex)
    {
    case rbiRedRefrBranch:
        node->NodeFlag |= ctnRREFRACTSET;
        break;
    case rbiGreenRefrBranch:
        node->NodeFlag |= ctnGREFRACTSET;
        break;
    case rbiBlueRefrBranch:
        node->NodeFlag |= ctnBREFRACTSET;
        break;
    }
    node->lock.unlock_resource();
    // check for convergence
    return (conv)?Converge(RP):FALSE;
}

void CTree::DestroyDepthFirst(CTreeNode *node)
{
    // visit then destroy the children
    if (node->ReflBranch) {
        DestroyDepthFirst(node->ReflBranch);
        delete node->ReflBranch;
    }
    for (int k = 0; k < 3; k++)
    if (node->RefrBranch[k]) {
        DestroyDepthFirst(node->RefrBranch[k]);
        delete node->RefrBranch[k];
    }
}

BOOLEAN CTree::Converge(RayPath *RP, Radiance *R)
{
    // this is used by CTPlayer
    if (!CONVERGE) return FALSE;
}
CTreeNode *CurrentNode(RP->Pointer());
int ContribType;

do {
   // don't waste time if I can't converge
   if (CurrentNode->NodeFlag != (ctnREFLECTSET | ctnGREFRACTSET |
      ctnRREFRACTSET | ctnBREFRACTSET | ctnLOCALSET)) return FALSE;
   
   while (!((CurrentNode->lock.lock_resource(FALSE))));
   
   /* check to see if convergence is possible
      if convergence is not possible then return */
   if (CurrentNode->NodeFlag != (ctnREFLECTSET | ctnGREFRACTSET |
      ctnRREFRACTSET | ctnBREFRACTSET | ctnLOCALSET)) {
      CurrentNode->lock.unlock_resource();
      return FALSE;
   }
   
   // set the current flag to say that the data has been converged
   CurrentNode->NodeFlag = ctnCONVERGED;
   
   // see if we are already at the source (no intersection occurred)
   if (RP->depth() == 0) {
      CurrentNode->lock.unlock_resource();
      if (R) *R = CurrentNode->composite();
      else *(FinalResult[0]) = CurrentNode->composite();
      return TRUE;
   }
   
   // This loop will continue until a break is encountered from within.
   for (Radiance composite;;) {
      // accumulate the contributions
      composite = CurrentNode->composite();
      
      /* Figure out what type of contribution ray we are currently at.
         i.e. did the current ray result from a reflection, refraction
         or source */
      ContribType = RP->CurrentRayType();
      
      if (ContribType == rpRAYSOURCE) {
         CurrentNode->lock.unlock_resource();
         if (R) *R = CurrentNode->composite();
         else *(FinalResult[0]) = CurrentNode->composite();
         
         CurrentNode->lock.unlock_resource();
         return TRUE;
      }
      
      // unlock the current node
      CurrentNode->lock.unlock_resource();
   }

   } while (!((CurrentNode->lock.lock_resource(FALSE))));

   return FALSE;
}
CurrentNode = RP->up_to_parent();
CurrentNode->lock.lock_resource();

switch (ContributionType) {
    case rpRAYREFLECTION:
        CurrentNode->ReflContrib = composite;
        CurrentNode->NodeFlag |= ctnREFLECTSET;
        break;
    case rpREDRAYREFRACTION:
        CurrentNode->RefrContrib[rbiRedRefrBranch] = composite;
        CurrentNode->NodeFlag |= ctnRREFRACTSET;
        break;
    case rpBLUERAYREFRACTION:
        CurrentNode->RefrContrib[rbiBlueRefrBranch] = composite;
        CurrentNode->NodeFlag |= ctnBREFRACTSET;
        break;
    case rpGREENRAYREFRACTION:
        CurrentNode->RefrContrib[rbiGreenRefrBranch] = composite;
        CurrentNode->NodeFlag |= ctnGREFRACTSET;
        break;
    default:
        cerr << "Unknown Contrib type found in CTree converge!" << endl;
        break;
}

// check to see if convergence at RP is possible
if (CurrentNode->NodeFlag != (ctnREFLECTSET | ctnRREFRACTSET | ctnGREFRACTSET | ctnBREFRACTSET | ctnLOCALSET)) {
    CurrentNode->lock.unlock_resource();
    return FALSE;
} else CurrentNode->NodeFlag = ctnCONVERGED;

// repeat this until convergence at RP is not possible or we are at root

BOOLEAN CTree::WriteOut()
{
    if (! Converged()) return FALSE;
    SetPixel(x, y, *(FinalResult[0]));
    return TRUE;
}

void CTree::IncRef()
{
    // lock, increment and unlock the refLock
    _Reflock.lock_resource();
    ReferenceCounter++;
    _Reflock.unlock_resource();
}

void CTree::DecRef()
{
    // lock, decrement and unlock the refLock
Reflock.lock_resource();
ReferenceCounter--;
Reflock.unlock_resource();

/************CTreeRec definitions ************/

CTreeRec::CTreeRec(int _x, int _y, char * fn): CTree(_x, _y), fout(fn)
{
    if (!fn) perror("Error opening output file for CTreeRec!");
    fout << x << " " << y << endl;
}

BOOLEAN CTreeRec::SetLocal(RayPath *RP, Radiance &R, BOOLEAN conv)
{
    foutLock.lock_resource();
    fout << getpid() << " SetLocal " << R[0] << " " << R[1]
    << " " << R[2];
    OutPutPath(fout, RP->Path(), RP->depth());
    foutLock.unlock_resource();

    return CTree::SetLocal(RP, R, conv);
}

BOOLEAN CTreeRec::SetRefracted(int t, RayPath *RP, Radiance &R, BOOLEAN conv)
{
    foutLock.lock_resource();
    fout << getpid() << " SetRefracted " << t << " " << R[0]
    << " " << R[1] << " " << R[2];
    OutPutPath(fout, RP->Path(), RP->depth());
    foutLock.unlock_resource();

    return CTree::SetRefracted(t, RP, R, conv);
}

BOOLEAN CTreeRec::SetReflected(RayPath *RP, Radiance &R, BOOLEAN conv)
{
    foutLock.lock_resource();
    fout << getpid() << " SetReflected " << R[0] << " "
    << R[1] << " " << R[2];
    OutPutPath(fout, RP->Path(), RP->depth());
    foutLock.unlock_resource();

    return CTree::SetReflected(RP, R, conv);
}

void CTreeRec::SetWeightScale(RayPath *RP, float W, float S)
{
    foutLock.lock_resource();
    fout << getpid() << " SetWeightScale " << W << " " << S << " ";
    OutPutPath(fout, RP->Path(), RP->depth());
    foutLock.unlock_resource();
}
CTree::SetWeightScale(RP, W, S);
}

void CTreeRec::SetSPD(RayPath *RP, unsigned char r, unsigned char g,
unsigned char b)
{
    foutLock.lock_resource();
    fout << getpid() << " SetSPD " << (int)r << " " << (int)g << " "
    << (int)b << " ";
    OutPutPath(fout, RP->Path(), RP->depth());
    foutLock.unlock_resource();
}

BOOLEAN CTreeRec::WriteOut()
{
    foutLock.lock_resource();
    fout << "Final Result: " << (*FinalResult[0])[0] << " "
    << (*FinalResult[0])[1] << " " << (*FinalResult[0])[2] << endl;
    fout.close();
    foutLock.unlock_resource();
    cout << "CTreeRec " << x << " " << y << " saved and closed." << endl;
    return CTree::WriteOut();
}

BOOLEAN CTreeRec::Converge(RayPath *RP, Radiance *R)
{
    if (!CONVERGE) return FALSE;
    CTreeNode *CurrentNode(RP->Pointer());
    int ContribType;
    do {
        // don't waste time if I can't converge
        if (CurrentNode->_lock.lock_resource(FALSE))
            return FALSE;
        // check to see if convergence is possible
        if (CurrentNode->NodeFlag != (ctnREFLECTSET | ctnGREFRACTSET |
            ctnRREFRACTSET | ctnBREFRACTSET | ctnLOCALSET))
            return FALSE;
    } while (!((CurrentNode->_lock.lock_resource(FALSE))));

    // set the current flag to say that the data has been converged
    CurrentNode->NodeFlag = ctnCONVERGED;

    // see if we are already at the source (no intersection occurred)
if (RP->depth() == 0) {
    currentNode->NodeFlag = ctnCONVERGED;
    currentNode->lock.unlock_resource();
    if (R) *R = currentNode->composite();
    else *(FinalResult[0]) = currentNode->composite();
    return TRUE;
}

pid_t PID = getpid();

// This loop will continue until a break is encountered from within.
for (Radiance composite;;) {
    // accumulate the contributions
    composite = currentNode->composite();

    /* Figure out what type of contribution ray we are currently at.
    i.e. did the current ray result from a reflection, refraction
    or source */
    ContribType = RP->CurrentRayType();

    if (ContribType == rpRAYSOURCE) {
        currentNode->lock.unlock_resource();
        if (R) *R = currentNode->composite();
        else *(FinalResult[0]) = currentNode->composite();
        currentNode->lock.unlock_resource();
        return TRUE;
    }

    // unlock the current node
    currentNode->lock.unlock_resource();

    currentNode = RP->up_to_parent();
    currentNode->lock.lock_resource();
    switch (ContribType)
    {
    case rpRAYREFLECTION:
        foutLock.lock_resource();
        fout << PID << " CSetReflected " << composite[0] << " 
        " << composite[1] << " " << composite[2];
        OutPutPath(fout, RP->Path(), RP->depth());
        foutLock.unlock_resource();
        currentNode->ReflContrib = composite;
        currentNode->NodeFlag |= ctnREFLECTSET;
        break;
    case rpREDRAYREFRACTION:
        foutLock.lock_resource();
        fout << PID << " CSetRefracted " << rbiRedRefrBranch << " 
        " << composite[0] << " " << composite[1] << " " 
        " << composite[2];
        OutPutPath(fout, RP->Path(), RP->depth());
        foutLock.unlock_resource();
        currentNode->RefrContrib[rbiRedRefrBranch] = composite;
        currentNode->NodeFlag |= ctnRREFRACTSET;
        break;
    }
break;

case rpBLUERAYREFRACTION:
    foutLock.lock_resource();
    fout << PID << " CSetRefracted " << rbiBlueRefrBranch << " "
    << composite[0] << " " << composite[1] << " "
    << composite[2];
    OutPutPath(fout, RP->Path(), RP->depth());
    foutLock.unlock_resource();
    CurrentNode->RefrContrib[rbiBlueRefrBranch] = composite;
    CurrentNode->NodeFlag |= ctnBREFRACTSET;
    break;

case rpGREENRAYREFRACTION:
    foutLock.lock_resource();
    fout << PID << " CSetRefracted " << rbiGreenRefrBranch << " "
    << composite[0] << " " << composite[1] << " "
    << composite[2];
    OutPutPath(fout, RP->Path(), RP->depth());
    foutLock.unlock_resource();
    CurrentNode->RefrContrib[rbiGreenRefrBranch] = composite;
    CurrentNode->NodeFlag |= ctnGREFRACTSET;
    break;

default:
    cerr << "Unknown Contrib type found in CTree converge!" << endl;
    }

// check to see if convergence at RP is possible
if (CurrentNode->NodeFlag != (ctnREFLECTSET | ctnRREFRACTSET |
    ctnGREFRACTSET | ctnBREFRACTSET |
    ctnLOCALSET)) {
    CurrentNode->lock.unlock_resource();
    return FALSE;
} else CurrentNode->NodeFlag = ctnCONVERGED;
// repeat this until convergence at RP is not possible or we are at root
}
}

/***** supCTree.h *****/

#ifndef _supCTree_h_
#define _supCTree_h_
#include "ConvObj.h"
#include "CTree.h"

class supConvergenceObject : public virtual ConvergenceObject {
public:
    supConvergenceObject(){CONVERGE=TRUE;}
supConvergenceObject(int, int, int);
    virtual "supConvergenceObject();
    virtual BOOLEAN SetRadiance(Radiance, int = 0);
    virtual Converged();
};
virtual WriteOut();

protected:
int samples;
int *supConverged;
lock supConvLock;
};

class supRayPath : public RayPath {
public:
supRayPath(supRayPath *sRP): sample(sRP->sample), RayPath(sRP) {}
supRayPath(CTreeNode *RootNode, int _sample = 0,
    ofstream *f = NULL, lock *fl = NULL)
    : RayPath(RootNode, f, fl), sample(_sample) {}

virtual RayPath * duplicate()
    { return (RayPath *) new supRayPath(this);}
int Sample() { return sample;}

private:
int sample;
};

class supCTree : public CTree {
public:
supCTree(int x, int y, int _sample = 1); // x, y, number of samples
virtual ~supCTree();

virtual RayPath *Root(int s = 0) // return a path to the nth root
    { if ((s < 0) || (s >= samples)) return NULL;
    else return (new supRayPath(_root[s], s));}

virtual BOOLEAN SetRadiance(Radiance, int _sample = 0);
virtual BOOLEAN WriteOutO;
virtual BOOLEAN Converged(); /* returns true if all supersample trees
    are set */

protected:
int samples;
int *supConverged;
lock supConvLock;

/* now we will use _root and _FinalResult as an array of CTreeNodes and
radiances */
virtual BOOLEAN Converge(RayPath *, Radiance * = NULL);
};

#endif

/***** supCTree ****/
#include <stdlib.h>
#include <iostream.h>
#include "supCTree.h"
#include "boolean.h"

supConvergenceObject::supConvergenceObject(int x, int y, int samples):
    samples(_samples), ConvergenceObject()
{
    int k;
    Radiance **t = new Radiance *[samples];
    FinalResult = t;
    for (k = 0; k < _samples; k++)
        FinalResult[k] = new Radiance;
    supConverged = new BOOLEAN[_samples];
    for (k = 0; k < _samples; k++)
        supConverged[k] = FALSE;
    x = _x;
    y = _y;
    Instances++;
}

supConvergenceObject::~supConvergenceObject()
{
    delete [] supConverged;
    for (int k = 1; k < samples; k++)
        delete FinalResult[k];
}

BOOLEAN supConvergenceObject::SetRadiance(Radiance R, int s)
{
    if ((s < 0) || (s >= samples)) {
        cerr << "SupConvergenceObject received invalid sample value!" << endl;
        return FALSE;
    }
    *(FinalResult[s]) = R / samples;
    supConvLock.lock_resource();
    supConverged[s] = TRUE;
    BOOLEAN c = Converged();
    supConvLock.unlock_resource();
    return c;
}

BOOLEAN supConvergenceObject::Converged()
{
    for (int k = 0; k < samples; k++)
        if (!supConverged[k]) return FALSE;
    return TRUE;
}
BOOLEAN supConvergenceObject::WriteOut()
{
    if (!Converged()) return FALSE;
    Radiance temp(0, 0, 0);
    for (int k = 0; k < samples; k++)
        temp+= *(FinalResult[k]);
    SetPixel(x, y, temp);
    return TRUE;
}

/************ supCTree **********/

supCTree::supCTree(int _x, int _y, int _samples):
samples(_samples), supConverged(new BOOLEAN[_samples])
{
    int k;
    ReferenceCounter = 0;
    _root = new CTreeNode * [_samples];
    FinalResult = new Radiance * [_samples];
    for (k = 0; k < _samples; k++)
    {
        _root[k] = new CTreeNode;
        FinalResult[k] = new Radiance;
        supConverged[k] = FALSE;
    }
    x = _x;
    y = _y;
    Instances++;
    CONVERGE = TRUE;
}

supCTree::~supCTree()
{
    // wait until no process is referring to this tree
    while (ReferenceCounter > 0);

    for (int k = 1; k < samples; k++)
        DestroyDepthFirst(_root[k]);
    for (k = 1; k < samples; k++)
        delete FinalResult[k];
    // _root and _root[0] are deleted in ~Ctree
    // FinalResult is deleted in ConvergenceObject
    delete [] supConverged;
}

BOOLEAN supCTree::Converged()
{
    for (int k = 0; k < samples; k++)
        if (!supConverged[k]) return FALSE;
    return TRUE;
}
BOOLEAN supCTree::SetRadiance(Radiance R, int s)
{
    if ((s < 0) || (s >= samples)) {
        cerr << "supConvergenceObject received invalid sample value!" << endl;
        return FALSE;
    }

    *(FinalResult[s]) = R / samples;
    supConvLock.lock_resource();
    supConverged[s] = TRUE;
    BOOLEAN c = Converged();
    supConvLock.unlock_resource();
    return c;
}

BOOLEAN supCTree::Converge(RayPath *RP, Radiance *)
{
    Radiance R;
    // if RP does not converged then return FALSE
    if (!CTree::Converge(RP, &R))
        return FALSE;
    else
        return SetRadiance(R, ((supRayPath*)RP)->Sample());
}

BOOLEAN supCTree::WriteOut()
{
    if (!Converged()) return FALSE;
    Radiance temp(0, 0, 0);
    for (int k = 0; k < samples; k++)
        temp += *(FinalResult[k]);
    SetPixel(x, y, temp);
    return TRUE;
}
Appendix F

CTPlayer Code

/***** CTPlayerMain.C *****/

#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include "boolean.h"
#include "Renderer.h"
#include "CTreePlayer.h"

char * filename = NULL;

BOOLEAN getArgs(int argc, char **argv)
{
    FILE *f;
    int k = 1;

    while (k < argc) {
        if (filename == NULL)
            f = fopen(argv[k], "r");

        if ((filename==NULL) && f) { // if this is a filename
            filename = argv[k++];
            fclose(f);
        }

        else // don't know what it is
            k++;
    }

    return TRUE;
}
```c
void usage()
{
    fprintf(stderr, "Usage: CTplayer\n\n\tfilename\n\n\n");
}

void main(int argc, char** argv)
{
    if (!getArgs(argc, argv)) {
        usage();
        exit(0);
    }

    graphicRenderer gui(argc, argv);

    assert(filename);
    fprintf(stderr, "Loading %s ...\n", filename);
    CTreePlayer *player = NULL;
    player = new CTreePlayer(filename);
    gui.SetPlayer(player);

    gui.UILoop();
}

/***** CTreePlayer.h *****/

#ifndef _CTreePlayer_h_
#define _CTreePlayer_h_

#include "CTree.h"
#define STRLen 128

class CTreeNodePlayer : public CTreeNode {
public:
    CTreeNodePlayer(int x = 0, int _y = 0):
        CTreeNode(), mynum(num++), x(x), y(_y), maxy(_y-1),
        REDRAW(1), HIGH(0) {}

friend void ShiftDown1(CTreeNodePlayer *c)
    {ShiftDownN(c, 1);}  
friend void ShiftUp1(CTreeNodePlayer *c)
    {ShiftDownN(c, -1);}  
friend void ShiftDownN(CTreeNodePlayer *, int);

/* x, y specify where on the screen the node will
be drawn. maxy is the maximum y used by a child
of this node. if maxy is < y then that signifies that
this node has no children. mynum is the number of
the node */
```
int x, y, maxy;
int mynum;
static int num;
unsigned int REDRAW, HIGH;
);

class strllNode {
public:
strllNode(char *s = NULL, strllNode *n = NULL):
str(s), next(n) {}
char *str;
strllNode *next;
};

class strll { /* a linked list of strings */
public:
strll(); head(NULL), tail(NULL), cur(NULL), size(0) {}
~strll(){clear();}
void clear();
char * first();
char * next();
void append(char *);
// convert the strll to an array of char *
char ** Arrayize();
// convert the strll to a char * buffer
char *Compactize(int *);
int Size() {return size;}

private:
strllNode *head;
strllNode *cur;
strllNode *tail;
int size;
};

class RayPathPlayer : public RayPath {
public:
RayPathPlayer(CTreeNode *ct):
RayPath(ct){}

/* the path is set to the int array of arg1. The size of the array is arg2
 (create new nodes if necessary *)
void SetPath(int *pi, int p);
void UndoDown();
CTreeNode * down_reflection(char * = "\0");
CTreeNode * down_refraction(int, char * = "\0");
private:
/* make space for a new node that is a descendant of TreePointer. */
void MakeSpace();
void CompactSpace();
/* add 1 to all y locations rooted at arg1 */
};

class CTreePlayer : public CTree {
public:
CTreePlayer(char *);
~CTreePlayer();
int position(){return current_position;}
unsigned int done(){return (current_position == size-1);}
char * step(int s = 1) /* step arg1 times forward. */
{ return ((s>=0) ?
applyTo(current_position+s):
unapplyTo(current_position+s));}
char * skipTo(int g) /* skipTo absolute location arg1 */
{ return (g<current_position)?unapplyTo(g):applyTo(g);}
char * start() {
Restarto; return EventBuffer[0];
}
char * end() {
return skipTo(size);
}
char * String(int k)
{ return
(((k<0) || (k>=size))?NULL:EventBuffer[k]);
}
virtual RayPath *Root(int = 0);
char * ClearTexto{return ClearTextBuffer;}
int ClearTextSize() {return ClearTextBufferSize;}
/* immediately after a skipTo, this will
this will be set to 0 if no redraw is necessary
otherwise nonzero */
unsigned int REDRAW;

private:
/* update the mazy, x and y positions of all nodes under arg1 */
void UpdatePositions(CTreeNodePlayer *);
void Restart();
/* these functions apply (or unapply) operations up to their arg. In the
 case of applyto, arg is not applied. In the case of unapplyto arg is
 unapplied */
char * applyTo(int);
char * unapplyTo(int);
void unSetLocal(RayPath *RP)
    {RP->Pointer() ->NodeFlag &= ~ctnLOCALSET;}
void unSetRefracted(int c, RayPath *RP)
    {switch (c)
    {case rbiRedRefrBranch:
        RP->Pointer() ->NodeFlag &= ~ctnRREFRACTIONSET;
        break;
    case rbiGreenRefrBranch:
        RP->Pointer() ->NodeFlag &= ~ctnGREFRACTIONSET;
        break;
    case rbiBlueRefrBranch:
        RP->Pointer() ->NodeFlag &= ~ctnBREFRACTIONSET;
        break;
    }
}
void unSetReflected(RayPath *RP)
    {RP->Pointer() ->NodeFlag &= ~ctnREFLECTSET;}
void unSetWeightScale(RayPath *RP) {
    RP->Pointer() ->weight = 1.0;
    RP->Pointer() ->ScalingCoeff = 1.0;
}
void unSetSPD(RayPath *RP)
{RP->Pointer()->spd[0] = 1;
RP->Pointer()->spd[1] = 1;
RP->Pointer()->spd[2] = 1;}

/* This function will take a string from a ct file
and create a new string that in plain text describes
what the ct string described */
char * Clarify(int, char *);

char **EventBuffer;
char *ClearTextBuffer;
int ClearTextBufferSize;
int size;
int current_position;
};

/* give this func a string that contains ||, and it will set the values
of arg2 to the path that follows (the int list of the string). Then it
will return the number of int in arg2 */
int ParsePath(char *, int *);

/* give this guy a string (arg1) and it will return an int representing
the operation to be executed. It will also set arg2 to the char after
the command */
int ParseCommand(char *, char **);

/* give this guy a string (arg1) and it will take the first 3 token as
R G B values of a radiance. It will then set the values of arg2 to those
RGB values. */
void ParseRadiance(char *, Radiance &);

// same as above for weight / scale lines */
void ParseWeightScale(char *, double &, double &);

// same as above for SPD */
void ParseSPD(char *, unsigned char *);

#define NumberOfCommands 8
#define _SetLocal 0
#define _SetWeightScale 1
#define _SetSPD 2
#define _SetRefractedR 3
#define _SetRefractedG 4
#define _SetRefractedB 5
#define _SetReflected 6
#define _Down 7

// CTreePlayer REDRAW values
#define DRAW_NONE 0
#define DRAW_ALL 1
#define DRAW_ONE 2
#define DRAW_WINDOW 3

// CTreeNodePlayer REDRAW values
#define NODRAW 0
#define DRAWHIGH 1
#define DRAWLOW 2


/** / **** CTreePlayer.C *****/

#include <fstream.h>
#include <iostream.h>
#include <string.h>
#include <assert.h>
#include "CTreePlayer.h"

// the first command is actually the
const char Commands[7][20] = {
  "SetLocal",
  "SetWeightScale",
  "SetSPD",
  "SetRefracted 0",
  "SetRefracted 1",
  "SetRefracted 2",
  "SetReflected"};

#define max(a, b) (((a) < (b))?(b):(a))
#define ROOTX 0
#define ROOTY 0

// ************* RayPathPlayer definitions *************/

/* set the path to pi . this function assumes that the path is currently 
at the root of the path */
void RayPathPlayer::SetPath(int *pi, int p)
{
  for (int k = 0; k < p; k++) {
    switch (pi[k]) {
    case rpRAYSOURCE:
      break;
    case rpRAYREFLECTION:
      if (!((TreePointer->ReflBranch)) down_reflection();
    else {
      TreePointer = TreePointer->ReflBranch;
      path[k] = pi[k];
      cur_depth++;
    }
    break;
    case rpREDRAYREFRACTION:
      if (!((TreePointer->RefrBranch[rbiRedRefrBranch]))
    down_refraction(rbiRedRefrBranch);
    else {
      TreePointer = TreePointer->RefrBranch[rbiRedRefrBranch];
      path[k] = pi[k];
      cur_depth++;
    }
    break;
    case rpGREENRAYREFRACTION:

if (!(TreePointer->RefrBranch[rbiGreenRefrBranch]))
    down_refraction(rbiGreenRefrBranch);
else {
    TreePointer = TreePointer->RefrBranch[rbiGreenRefrBranch];
    path[k] = pi[k];
    cur_depth++;}
break;
case rpBLUERAYREFRACTION:
    if (!(TreePointer->RefrBranch[rbiBlueRefrBranch]))
        down_refraction(rbiBlueRefrBranch);
    else {
        TreePointer = TreePointer->RefrBranch[rbiBlueRefrBranch];
        path[k] = pi[k];
        cur_depth++;}
break;
}
}

CTreeNode * RayPathPlayer::down_reflection(char *)
{
    // check if we are at the maximum depth
    if (cur_depth == MaxDepth) return NULL;

    /* if not then add a reflection to the path and a new node to the tree. Then set the pointer to the new location in the tree */
    path[cur_depth++] = rpRAYREFLECTION;
    /* since this method is used to expand the tree we can assume that TreePointer is a leaf in the tree. Therefore we can forego checking to see if it is a leaf. */

    // get and set the y and x
    int x, y;
    y = ++(((CTreeNodePlayer *)TreePointer)->maxy;
    if (y > ((CTreeNodePlayer *)TreePointer)->y)
        MakeSpace();
    x = ((CTreeNodePlayer *)TreePointer)->x + 1;
    TreePointer->ReflBranch = new CTreeNodePlayer(x, y);
    TreePointer->ReflBranch->parent = TreePointer;
    TreePointer = TreePointer->ReflBranch;

    return TreePointer;
}

CTreeNode * RayPathPlayer::down_refraction(int RefrInd, char *)
{
    // check if we are at the maximum depth
    if (cur_depth == (MaxDepth - 1)) return NULL;
if ((RefrInd < 0) || (RefrInd > 2)) return NULL;

CTreeNode * n;
// get and set the y and x
int x, y;
y = ++((CTreeNodePlayer *)TreePointer)->maxy;

if (y > ((CTreeNodePlayer *)TreePointer)->y)  
    MakeSpace();
    x = ((CTreeNodePlayer *)TreePointer)->x + 1;
n = new CTreeNodePlayer(x, y);

TreePointer->RefrBranch[RefrInd] = n;
TreePointer->RefrBranch[RefrInd]->parent = TreePointer;
TreePointer = TreePointer->RefrBranch[RefrInd];

switch (RefrInd)
{
    case rbiRedRefrBranch:
        path[cur_depth] = rpREDRAYREFRACTION;
        break;
    case rbiGreenRefrBranch:
        path[cur_depth] = rpGREENRAYREFRACTION;
        break;
    case rbiBlueRefrBranch:
        path[cur_depth] = rpBLUERAYREFRACTION;
        break;
}

cur_depth++;  
return TreePointer;
}

void RayPathPlayer::UndoDown()
{
    CompactSpace();
    int todel = CurrentRayType();
    CTreeNode * todelP = Pointer();
    CTreeNode * npointer = up_to_parent();
    assert(npointer);

    switch (todel)
    {
    case rpRAYREFLECTION:
        delete npointer->ReflBranch;
        npointer->ReflBranch = NULL;
        break;
    case rpREDRAYREFRACTION:
        delete npointer->RefrBranch[rbiRedRefrBranch];
        npointer->RefrBranch[rbiRedRefrBranch] = NULL;
        break;
    case rpGREENRAYREFRACTION:
        delete npointer->RefrBranch[rbiGreenRefrBranch];
        break;
npointer->RefrBranch[rbiGreenRefrBranch] = NULL;
break;
case rpBLUERAYREFRACTION:
delete npointer->RefrBranch[rbiBlueRefrBranch];
npointer->RefrBranch[rbiBlueRefrBranch] = NULL;
break;
}
}

void RayPathPlayer::MakeSpace()
{
  CTreeNodePlayer *p = (CTreeNodePlayer *)TreePointer->parent;
  CTreeNodePlayer *t = (CTreeNodePlayer *)TreePointer;

  // while a parent exists
  while (p) {
    // check all the other children of p to see if there is an overlap

    /* if t is not the reflection branch, and the reflection branch
       is below (on the screen) t */
    if (p->ReflBranch) {
      if ((p->ReflBranch != t) &&
          (((CTreeNodePlayer *)p->ReflBranch)->y >= t->y)) {
        ShiftDown1((CTreeNodePlayer *)p->ReflBranch);
        /* take the max of the child’s mazy and y (since if the child has
           no children the child’s y will be greater than its mazy */
        p->maxy = max(p->maxy, ((CTreeNodePlayer *)p->ReflBranch)->maxy);
        p->maxy = max(p->maxy, ((CTreeNodePlayer *)p->ReflBranch)->y);
      }
    }

    /* check each Refraction branch in the same manner */
    for (int k = 0; k < 3; k++)
      if (!p->RefrBranch[k]) continue;
      else {
        if ((p->RefrBranch[k] != t) &&
            (((CTreeNodePlayer *)p->RefrBranch[k])->y >= t->y)) {
          ShiftDown1((CTreeNodePlayer *)p->RefrBranch[k]);
          p->maxy = max(p->maxy, ((CTreeNodePlayer *)p->RefrBranch[k])->maxy);
          p->maxy = max(p->maxy, ((CTreeNodePlayer *)p->RefrBranch[k])->y);
        }
      }

    // move up a level
    t = p;
    p = (CTreeNodePlayer *)t->parent;
  }
}

void RayPathPlayer::CompactSpace()
{
  CTreeNodePlayer *p = (CTreeNodePlayer *)TreePointer->parent;
  CTreeNodePlayer *t = (CTreeNodePlayer *)TreePointer;

  // while a parent exists
  while (p) {
int curmax = p->y - 1;
// check all the other children of p to see if there is an overlap

/* if t is not the reflection branch, and the reflection branch
is below (on the screen) t */
if (p->ReflBranch)
if (p->ReflBranch != t) {
    if (((CTreeNodePlayer *)p->ReflBranch)->y >= t->y)
        ShiftUpl((CTreeNodePlayer *)p->ReflBranch);
    // make sure that even if there are no children this will work
    curmax = max(curmax,
                 max(((CTreeNodePlayer *)p->ReflBranch)->maxy,
                     ((CTreeNodePlayer *)p->ReflBranch)->y));
}

/* check each Refraction branch in the same manner */
for (int k = 0; k < 3; k++)
    if (!p->RefrBranch[k]) continue;
    else {
        if (p->RefrBranch[k] != t) {
            if (((CTreeNodePlayer *)p->RefrBranch[k])->y >= t->y)
                ShiftUpl((CTreeNodePlayer *)p->RefrBranch[k]);
            // make sure that even if there are no children this will work
            curmax = max(curmax,
                         max(((CTreeNodePlayer *)p->RefrBranch[k])->y,
                             ((CTreeNodePlayer *)p->RefrBranch[k])->maxy));
        }
    }
    // move up a level
    p->maxy = curmax;
    /* if I eliminated the only child of a node, then
    there's no need to go up a level */
    if (p->maxy < p->y) break;
    t = p;
    p = (CTreeNodePlayer *)t->parent;
}

/******* CTreePlayer **********/

CTreePlayer::CTreePlayer(char *filename)
:current_position(0), REDRAW(0)
{
    ifstream fin(filename);
    if (!fin) {
        cerr << "Error opening " << filename << "!" << endl;
        exit(0);
    }

    char * inString = NULL;
    stril StringList;
    strll ClearStringList;
    // get the position of the pixel
    fin >> x >> y;
ConvergenceObject(x, y);
CONVERGE = FALSE;

while(fin.get() != ' \n');
int k, NumberOfProcesses = 0;
unsigned short FOUND;
 /* I think its safe to assume that we run with less than
100 processes */
int Parray[100];

while (fin) {
    if (finString)
        inString = new char[STRLen];
    fin.getline(inString, STRLen);
    if (strlen(inString) > 0) {
        // check for the current pid
        sscanf(inString, "%d", Parray+NumberOfProcesses);
        for (k = 0, FOUND = 0; (k < NumberOfProcesses) && (!FOUND); k++)
            if (Parray[k] == Parray[NumberOfProcesses]) FOUND = 1;
        if (!FOUND) NumberOfProcesses++;
        StringList.append(inString);
        int n = ClearStringList.Size();
        ClearStringList.append(Clarify(n, inString));
        inString = NULL;
    }
}
printf("%d Processes found in the CT file\n", NumberOfProcesses);
for (k=1, printf("%d", Parray[0]); k<NumberOfProcesses; k++)
    printf(", %d", Parray[k]);
printf("\n\n");
EventBuffer = StringList.Arrayize();
size = StringList.Size();
ClearTextBuffer = ClearStringList.Compactize(&ClearTextBufferSize);

_root = new CTreeNode *
_root = new CTreeNodePlayer(ROOTX, ROOTY);
FinalResult = new Radiance *[1];
FinalResult[0] = new Radiance(0, 0, 0);
}

CTreePlayer::~CTreePlayer()
{
    for (int k = 0; k < size; k++)
        delete EventBuffer[k];
    delete EventBuffer;
    delete ClearTextBuffer;
}

RayPath *CTreePlayer::Root(int)
{
return (RayPath *) new RayPathPlayer(_root[0]);
}

void CTreePlayer::Restart()
{
    DestroyDepthFirst(_root[0]);
    delete _root[0];
    _root[0] = new CTreeNode;
}

char * CTreePlayer::applyTo(int n)
{
    if ((n < current_position) || (n >= size)) {
        cerr << "applyTo given invalid arg!\n";
        return NULL;
    }

    char * args;
    int comm, pathl[MaxDepth], sizel;

    while (current_position < n) {
        sizel = ParsePath(EventBuffer[current_position], pathl);
        RayPathPlayer *rp = (RayPathPlayer *)Root();
        rp->SetPath(pathl, sizel);
        comm = ParseCommand(EventBuffer[current_position],&args);
        current_position++;
        Radiance R;

switch (comm)
{
    case _Down:
        REDRAW = DRAW_ALL;
        break;
    case _SetLocal:
        REDRAW = DRAW_ONE;
        ((CTreeNodePlayer *)rp->Pointer())->REDRAW = 1;
        ParseRadiance(args, R);
        SetLocal(rp, R);
        break;
    case _SetWeightScale:
        double w, s;
        ParseWeightScale(args, w, s);
        SetWeightScale(rp, w, s);
        break;
    case _SetSPD:
        unsigned char spd[3];
        ParseSPD(args, spd);
        SetSPD(rp, spd);
        break;
    case _SetRefractedR:
        REDRAW = DRAW_ONE;
        ((CTreeNodePlayer *)rp->Pointer())->REDRAW = 1;
        ParseRadiance(args, R);
        SetRefracted(rbiRedRefrBranch, rp, R);

202
break;
  case _SetRefractedG:
    REDRAW = DRAW_ONE;
    ((CTreeNodePlayer *)rp->Pointer())->REDRAW = 1;
    ParseRadiance(args, R);
    SetRefracted(rbiGreenRefrBranch, rp, R);
    break;
  case _SetRefractedB:
    REDRAW = DRAW_ONE;
    ((CTreeNodePlayer *)rp->Pointer())->REDRAW = 1;
    ParseRadiance(args, R);
    SetRefracted(rbiBlueRefrBranch, rp, R);
    break;
  case _SetReflected:
    REDRAW = DRAW_ONE;
    ParseRadiance(args, R);
    SetReflected(rp, R);
    break;
  }
  delete rp;
}
return EventBuffer[current_position];

char * CTreePlayer::unapplyTo(int n)
{
  if (n < 0) n = 0;
  if (n >= current-position) {
    cerr << "unapplyTo given invalid arg!\n";
    return NULL;
  }
  char * args;
  RayPathPlayer *rp = (RayPathPlayer *)Root();
  int comm, pathl[MaxDepth], sizel;
  while (current_position > n) {
    current_position--;
    sizel = ParsePath(EventBuffer[current_position], pathl);
    rp->SetPath(pathl, sizel);
    comm = ParseCommand(EventBuffer[current_position],&args);
    if (rp->Pointer()->NodeFlag == ctnCONVERGED)
      rp->Pointer()->NodeFlag = (ctnREFLECTSET |
      ctnLOCALSET |
      ctnRREFRACTSET |
      ctnGREFRACTSET |
      ctnBREFRACTSET);
    switch (comm)
      {
    case _SetLocal:
      REDRAW = DRAW_ONE;
((CTreeNodePlayer *)rp->Pointer())->REDRAW = 1;
unSetLocal(rp);
break;
case _SetWeightScale:
unSetWeightScale(rp);
break;
case _SetSPD:
unSetSPD(rp);
break;
case _SetRefractedR:
    REDRAW = DRAW_ONE;
    ((CTreeNodePlayer *)rp->Pointer())->REDRAW = 1;
    unSetRefracted(rbiRedRefrBranch, rp);
break;
case _SetRefractedG:
    REDRAW = DRAW_ONE;
    ((CTreeNodePlayer *)rp->Pointer())->REDRAW = 1;
    unSetRefracted(rbiGreenRefrBranch, rp);
break;
case _SetRefractedB:
    REDRAW = DRAW_ONE;
    ((CTreeNodePlayer *)rp->Pointer())->REDRAW = 1;
    unSetRefracted(rbiBlueRefrBranch, rp);
break;
case _SetReflected:
    REDRAW = DRAW_ONE;
    ((CTreeNodePlayer *)rp->Pointer())->REDRAW = 1;
    unSetReflected(rp);
break;
case _Down:
    if (size1 > 0) {
        rp->UndoDown();
        REDRAW = DRAW_ALL;
    } break;
} delete rp;
return EventBuffer[current_position];
}

char * CTreePlayer::Clarify(int NUM, char *ins)
{
    int UID;
    char *NewString = new char[STRLen];
    unsigned int RefrColor, ConvergeFlag = 0;

    // if this is the final result
    if (strstr(ins, "Final Result")) {
        sprintf(NewString, "%d. \t%s\n", NUM, ins);
        return NewString;
    }

    // otherwise there will be a uid

    return NewString;
}
sscanf(ins, "%d", &UID);

// skip to the start of the Command.
ins = strstr(ins, " ") + 1;

// If this character is 'C' then we are converging
if (*ins == 'C') {
  ConvergeFlag = 1;
  ins++;
}

// get the command and set ins to the path
if (strstr(ins, "SetLocal")) {
  ins = strstr(ins, " ");
  sprintf(NewString, "%d. %d Set a Local radiance to %s ",
          NUM, UID, ins);
} else if (strstr(ins, "SetWeightScale")) {
  ins = strstr(ins, " ");
  sprintf(NewString, "%d. %d Set a Weight and Scale to %s ",
          NUM, UID, ins);
} else if (strstr(ins, "SetRefracted")) {
  ins = strstr(ins, " ");
  sscanf(ins, "%d", &RefrColor);
  ins = strpbrk(ins, "012");
  ins++;
  sprintf(NewString, "%d. %d %s a Refraction radiance to %s ",
          NUM, UID,
          ((ConvergeFlag)? "Converged":"Set"),
          ((RefrColor==rbiRedRefrBranch)? "Red":
            ((RefrColor==rbiBlueRefrBranch)? "Blue":"Green"),
          ins);
} else if (strstr(ins, "SetReflected")) {
  // place ins at the radiance
  ins = strstr(ins, " ");
  sprintf(NewString, "%d. %d %s a Reflected radiance to %s ",
          NUM, UID,
          ((ConvergeFlag)? "Converged":"Set"), ins);
} else if (strstr(ins, "SetSPD")) {
  ins = strstr(ins, " ");
  sprintf(NewString, "%d. %d set an SPD to %s ", NUM, UID, ins);
} else {
  // it must be a down
  sprintf(NewString, "%d. %d creates a new ", NUM, UID);
  // get past the || divider
  ins = strstr(ins, "||") + 2;
  // loop through the path until I reach the last
  char *nptr = strpbrk(ins, "2345");
  while (nptr) {
    ins = nptr;
  }
}
nptr = strpbrk(ins+1, "2345");
}
switch (*ins - '0')
{
    case rpRAYREFLECTION:
        strcat(NewString, "Reflection Ray.");
        break;
    case rpREDRAYREFRACTION:
        strcat(NewString, "Red Refraction Ray.");
        break;
    case rpGREENRAYREFRACTION:
        strcat(NewString, "Green Refraction Ray.");
        break;
    case rpBLUERAYREFRACTION:
        strcat(NewString, "Blue Refraction Ray.");
        break;
}

// get rid of the path trailer
if ((ins = strstr(NewString, "|")))
    *ins = '\0';
// add a new line
strcat(NewString, "\n");
return NewString;

/**************************** MISC **************************/

int ParseCommand(char *1, char **p)
{
    int k = 0;
    char *temp = NULL;
    while (k < NumberOfCommands-1) {
        if ((temp = strstr(1, Commands[k])) != NULL) {
            *p = strstr(temp, " ");
            switch (k)
            {
                case _SetRefractedR:
                case _SetRefractedG:
                case _SetRefractedB:
                    *p += 2;
                    default: *p++;
            }
            return k;
        } else
            k++;
    }
    return _Down;
}

int ParsePath(char *1, int *i)
{
if (!1) return 0;

char buf[STRLen];
char * path = strstr(l, "\"\");  
if (!path) return 0;

int r = 0;
path +=2;
strcpy(buf, path);
char *n = strtok(buf, " \t\n");

while (n) {
    i[r++] = atoi(n);
    n = strtok(NULL, " \t\n");
}

if (r == 10)
    cerr << "Parse Path had problems with an invalid path!\n";
return r;

void ParseRadiance(char *s, Radiance &R) {
    R[0] = strtod(s, &s);
    R[1] = strtod(s, &s);
    R[2] = strtod(s, &s);
}

void ParseWeightScale(char *s, double &w, double &sc) {
    w = strtod(s, &s);
    s = strstr(s, " ");
    sc = strtod(s, &s);
}

void ParseSPD(char *s, unsigned char *spd) {
    spd[0] = (unsigned char)strtol(s, &s, 0);
    spd[1] = (unsigned char)strtol(s, &s, 0);
    spd[2] = (unsigned char)strtol(s, &s, 0);
}

void ShiftDownN(CTreeNodePlayer *n, int N) {
    if (n) {
        n->y +=N;
        n->maxy +=N;
        ShiftDownN((CTreeNodePlayer *)n->ReflBranch, N);
        for (int k = 0; k < 3; k++)
            ShiftDownN((CTreeNodePlayer *)n->RefrBranch[k], N);
    }
}

void strll::clear()
{  
cur = head;
while (cur->next){
   cur = cur->next;
   delete head;
   head = cur;
}

delte cur;
cur = head = tail = NULL;
size = 0;
}

char * strll::first(){
  
cur = head;
  return cur->str;
}

char * strll::next(){
  
if (!(cur->next)) return NULL;
cur = cur->next;
return cur->str;
}

void strll::append(char *s){
  
if (!tail) cur = head = tail = new strllNode(s);
else {
   tail->next = new strllNode(s);
tail = tail->next;
}
size++;
}

char ** strll::Arrayize(){
  
char ** ar = new char * [size];
char *s, int k;
for (k = 0, s = first(); s; k++, s = next())
ar[k] = s;
return ar;
}

char * strll::Compactize(int *csize){
  
char *s, *buf, *buftp;
for (*csize = 0, s = first();
s;
s = next())
csize+=strlen(s);
// make space for the trailing null
(*csize)++; 700
bufp = buf = new char[*csize];
bufp[0] = 0;

for (s=first(); s; s = next()) {
    strncat(bufp, s, strlen(s));
    // move bufp to the end of the current string
    bufp += strlen(s);
}
return buf;
}

/***** Renderer.h *****/

#ifndef _Renderer_h_
#define _Renderer_h_

#include <Xm/Xm.h>
#include "CTreePlayer.h"

class Renderer {
public:
    Renderer(int xd = 0, int yd = 0):
        _YDist(yd), _XDist(xd), player(NULL){}
    int YDist(){return _YDist;}
    int XDist(){return _XDist;}
    virtual void SetPlayer(CTreePlayer *p) {player = p;}

protected:
    /* distance between one node and another */
    int _YDist, _XDist;
    CTreePlayer *player;
};

class graphicRenderer : public Renderer {
public:
graphicRenderer(int argc = 0, char **argv = NULL);
    virtual ~graphicRenderer(){}
    void UILoop() { XtAppMainLoop(app_context);}
    virtual void SetPlayer(CTreePlayer *p);

private:
    Widget MainWin, buttonbox, topLevel;
    XtApplicationContext app_context;
    Widget PlayButton, StopButton, QuitButton, SkipButton;
    Widget FStepButton, BStepButton, Canvas, CanvasScrollBar;
    Widget TextWindow, TextWinHeader, TextWinForm, CanvasWindow;
    Widget skipShell, skipDialog, skipApplyButton, skipCancelButton;
};

209
// CallBacks
void QuitCB(Widget, XtPointer, XtPointer);
void PlayCB(Widget, XtPointer, XtPointer);
void StopCB(Widget, XtPointer, XtPointer);
void FStepCB(Widget, XtPointer, XtPointer);
void BStepCB(Widget, XtPointer, XtPointer);
void SkipCB(Widget, XtPointer, XtPointer);
void SkipApplyCB(Widget, XtPointer, XtPointer);
void SkipCancelCB(Widget, XtPointer, XtPointer);
void ScrollProc(Widget, XtPointer, XtPointer);
void JumpProc(Widget, XtPointer, XtPointer);
void TimerPlayBackCB(XtPointer, XtIntervalId *);

// Event Handelers
void DrawTreeEH(Widget, XtPointer, XEvent *, Boolean *
);

// data structs
struct DrawEHStruct {
  GC gc;
  CTreePlayer **cpP;
  // this is used to track what node should be highlighted
  CTreeNodePlayer *cpn;
  XColor colors[5];
  colormap cm;
  Widget scroller;
};

#define RED 0
#define _GREEN_ 1
#define BLUE 2
#define HIWHITE 3
#define LOWWHITE 4

struct StepStruct {
  Widget Canvas, TextWindow;
  // this is a double pointer because the player isn't known
  until later in the code, so to make the player visible
  I send a pointer to the pointer that will point to the
  player... pretty tricky */
  Display *disp;
  Window w;
  DrawEHStruct *des;
};

struct DrawInfo {
  Display *disp;
  Drawable drable;
  GC gc;
  XColor *colors;
  colormap cm;
  unsigned int REDRAW;
};
struct PlayStruct {
    StepStruct *ss;
    XtAppContext app;
};

struct SkipInfo {
    StepStruct *ss;
    Widget TopLevel;
    Widget SkipButton;
    Widget SkipShell;
    Widget SkipDialog;
};

struct QuitInfo {
    DrawEHStruct *des;
    StepStruct *ss;
    PlayStruct *ps;
    SkipInfo *si;
};

// TreeDrawing subroutines
void DrawTreeNode(CTreeNode *, DrawInfo *);
void DrawNode(CTreeNode *, DrawInfo *);
void ClearNode(CTreeNode *, DrawInfo *);

// this function will move the text location in the text window n times
void MoveTextLocation(int, CTreePlayer *);

#endif

/***** Render.C *****/

// #include <X11/Intrinsic.h>

/*
   * Standard Toolkit include files:
   */
#include <X11/Intrinsic.h>
#include <X11/StringDefs.h>

/*
   * Public include files for widgets used in this file.
   */
#include <X11/Xaw/Command.h>
#include <X11/Xaw/Box.h>
#include <X11/Xaw/Form.h>
#include <X11/Xaw/Label.h>
#include <X11/Xaw/AsciiText.h>
#include <X11/Xaw/Dialog.h>
#include <X11/Xaw/Scrollbar.h>
#include <assert.h>
#include <ctype.h>
#include <math.h>
#include "Renderer.h"

int CTreeNodePlayer::num = 0;
#define NODEWIDTH 50
#define NODEHEIGHT 15
#define VSPACE 10
#define HSPACE 40
#define NORMALLINEWIDTH 1
#define THICKLINEWIDTH 3
#define SCROLLLEN 500
#define CANVASHEIGHT 500

int TRANSLATION = 0;
float ThumbSize = CANVASHEIGHT;
float ThumbLocation = 0;
#define DeviceX(x) (HSPACE + (HSPACE + NODEWIDTH) * (x))
#define DeviceY(y) (VSPACE + (VSPACE + NODEHEIGHT) * (y) + TRANSLATION)
#define AbsoluteY(y) (VSPACE + (VSPACE + NODEHEIGHT) * (y))
#define min(a, b) (((a)<(b))?(a):(b))

// half second update interval
#define INTERVAL 500
XtIntervalId PlayTimer = 0;
XawTextPosition StartTextLocation;
XawTextPosition EndTextLocation;

graphicRenderer::graphicRenderer(int argc, char **argv) {
    /* create GUI */
    XtSetLanguageProc(NULL, (XtLanguageProc)NULL, NULL);

topLevel = XtVaAppInitialize (&app_context,
        "CTPlayer",
        NULL, 0,
        &argc, argv,
        NULL, NULL);

MainWin = XtCreateManagedWidget("form", formWidgetClass,
        topLevel, NULL, 0);

buttonbox = XtVaCreateManagedWidget("box",
        boxWidgetClass,
        MainWin,
        NULL);

PlayButton = XtVaCreateManagedWidget("Play",
        commandWidgetClass,
        buttonbox,
        NULL);

212
StopButton = XtVaCreateManagedWidget("Stop",
commandWidgetClass,
buttonbox,
NULL);

FStepButton = XtVaCreateManagedWidget("FStep",
commandWidgetClass,
buttonbox,
NULL);

BStepButton = XtVaCreateManagedWidget("BStep",
commandWidgetClass,
buttonbox,
NULL);

SkipButton = XtVaCreateManagedWidget("Skip",
commandWidgetClass,
buttonbox,
NULL);

QuitButton = XtVaCreateManagedWidget("Quit",
commandWidgetClass,
buttonbox,
NULL);

CanvasWindow = XtCreateManagedWidget ("CanvasWindow",
boxWidgetClass,
MainWin, NULL, 0);

XtVaSetValues(CanvasWindow, XtNwidth, 610,
XtNheight, CANVASHEIGHT, NULL);

Canvas = XtCreateManagedWidget("Canvas",
coreWidgetClass,
CanvasWindow, NULL, 0);

XtVaSetValues(Canvas, XtNwidth, 580,
XtNheight, CANVASHEIGHT, NULL);

CanvasScrollBar = XtVaCreateManagedWidget("CanvasScrollBar",
scrollbarWidgetClass,
CanvasWindow,
XtNlength, SCROLLLEN, NULL);

XawScrollbarSetThumb(CanvasScrollBar, 0, CANVASHEIGHT);

TextWinForm = XtVaCreateManagedWidget("TextWinForm",
formWidgetClass,
MainWin, NULL);

TextWinHeader = XtVaCreateManagedWidget("TextWinHeader",
labelWidgetClass,
TextWinForm,
XtNlabel,
" Op#    PID Operation",
XtNwidth, 600, NULL);
TextWindow = XtVaCreateManagedWidget("TextWindow",
asciiTextWidgetClass,
TextWinForm,
XtNscrollVertical,
XawtextScrollWhenNeeded,
NULL);

skipShell = XtVaCreateManagedWidget("skipShell",
 transientShellWidgetClass,
topLevel,
NULL);

skipDialog = XtVaCreateManagedWidget("skipDialog",
 dialogWidgetClass,
skipShell,
NULL);

skipApplyButton = XtVaCreateManagedWidget("skipApplyButton",
 commandWidgetClass,
skipDialog, NULL);

skipCancelButton = XtVaCreateManagedWidget("skipCancelButton",
 commandWidgetClass,
skipDialog, NULL);

Display *disp = XtDisplay(topLevel);

// ***** Create the DrawEHStruct *****
DrawEHStruct *t = new DrawEHStruct;
XGCValues values;
values.foreground = 1;
values.background = 0;
values.line_width = NORMALLINEWIDTH;
t->gc = XCreateGC(XtDisplay(topLevel),
RootWindowOfScreen(XtScreen(topLevel)),
GCForeground | GCBACKGROUND | GCLINEWIDTH,
&values);
t->cpP = &player;
XColor exact, closest;
Colormap cm = DefaultColormap(disp, DefaultScreen(disp));
t->cm = cm;

// get RED
Status s = XLookupColor(XtDisplay(topLevel), cm, "red",
&exact, &closest);
t->colors[RED] = closest;

// get GREEN
s = XLookupColor(XtDisplay(topLevel), cm, "green",
&exact, &closest);
t->colors[GREEN] = closest;

// get BLUE
s = XLookupColor(XtDisplay(topLevel), cm, "DeepSkyBlue",
&exact, &closest);
t->colors[_BLUE_] = closest;
// get LOWWHITE
s = XLookupColor(XtDisplay(topLevel), cm, "gray",
&exact, &closest);
t->colors[_LOWWHITE_] = closest;
// get HIWHITE
s = XLookupColor(XtDisplay(topLevel), cm, "white",
&exact, &closest);
t->colors[_HIWHITE_] = closest;

t->scroller = CanvasScrollBar;

// ***** Create the StepStruct *****
StepStruct *ss = new StepStruct;
ss->Canvas = Canvas;
/* I need to send a reference to the player pointer,
because the player isn't set yet. */

ss->disp = disp;
ss->w = RootWindowOfScreen(XtScreen(topLevel));
ss->des = t;
ss->TextWindow = TextWindow;

// ***** Create PlayStruct *****
PlayStruct *ps = new PlayStruct;
ps->ss = ss;
ps->app = app_context;

// ***** create skip fino *****
SkipInfo *si = new SkipInfo;
si->TopLevel = topLevel;
si->SkipButton = SkipButton;
si->SkipShell = skipShell;
si->SkipDialog = skipDialog;
si->ss = ss;

// ***** create quit struct *****
QuitInfo *qs = new QuitInfo;
qs->des = t;
qs->ss = ss;
qs->ps = ps;
qs->si = si;

XtAddCallback(QuitButton, XtNcallback, QuitCB, qs);
XtAddCallback(PlayButton, XtNcallback, PlayCB, ps);
XtAddCallback(StopButton, XtNcallback, StopCB, 0);
XtAddCallback(FStepButton, XtNcallback, FStepCB, ss);
XtAddCallback(BStepButton, XtNcallback, BStepCB, ss);
XtAddCallback(SkipButton, XtNcallback, SkipCB, ss);
XtAddCallback(skipApplyButton, XtNcallback, SkipApplyCB, si);
XtAddCallback(skipCancelButton, XtNcallback, SkipCancelCB, si);
XtAddCallback(CanvasScrollBar, XtNjumpProc, JumpProc, ss);
XtAddCallback(CanvasScrollBar, XtNscrollProc, ScrollProc, ss);
void graphicRenderer::SetPlayer(CTreePlayer *p)  
{  
player = p;  
char *t = p->ClearText();  

XtVaSetValues(TextWindow,  
    XtNlength, p->ClearTextSize(),  
    XtNstring, p->ClearText(),  
    NULL);  
StartTextLocation = 0;  
EndTextLocation = -1;  
}  

void QuitCB(Widget, XtPointer Data, XtPointer)  
{  
    printf("QuitCB\n");  
    /* since I sent in a reference to the pointer to player  
       do a dereference and then deref the deref */  
    QuitInfo *qs = (QuitInfo *)Data;  
    delete (*qs->des->cpP));  
    delete qs->des;  
    delete qs->ps;  
    delete qs->si;  
    delete qs;  
    exit(0);  
}  

void FStepCB(Widget, XtPointer Data, XtPointer)  
{  
    /* since I sent in a reference to the pointer to player  
       do a dereference and then deref the deref */  
    StepStruct *ss = (StepStruct *)Data;  
    CTreePlayer *p = *(ss->des->cpP);  

    unsigned int temp = p->REDRAW;  
    char *s = p->stepO;  
    if (temp != DRAWNONE) p->REDRAW = temp;  
    // if s = NULL then we are at the end of the buffer  
    if (!s) return;  
    printf("%d: %s\n", p->positionO, s);  

    if (p->REDRAW != DRAW_NONE) {  
        RayPathPlayer *rp = (RayPathPlayer *)p->Root();  
        if (p->REDRAW != DRAW_ONE) {  
            int maxy = ((p->REDRAW == DRAW_WINDOW)?0:  
                DeviceY(((CTreeNodePlayer *)rp->Pointer())->maxy + 1));  
            XClearArea(ss->disp, XtWindow(ss->Canvas), 0, 0, maxy,  
                False);  
        }  
    }
DrawTreeEH(ss->Canvas, ss->des, NULL, NULL);

delete rp;
p->REDRAW = DRAW_NONE;
}

XawTextUnsetSelection(ss->TextWindow);
MoveTextLocation(1, p);
XawTextSetInsertionPoint(ss->TextWindow, StartTextLocation);
XawTextSetSelection(ss->TextWindow, StartTextLocation,

EndTextLocation);
}

void BStepCB(Widget, XtPointer Data, XtPointer)
{
    /* since I sent in a reference to the pointer to player
     do a dereference and then deref the deref */
    StepStruct *ss = (StepStruct *)Data;
    CTreePlayer *p = *(ss->des->cpP);

    RayPathPlayer *rp = (RayPathPlayer *)p->RootO;
    int maxy = DeviceY(((CTreeNodePlayer *)rp->Pointer())->maxy + 1);

    unsigned int temp = p->REDRAW;
    char *s = p->step(-1);
    if (temp != DRAW_NONE) p->REDRAW = temp;
    if (!s) return;
    printf("%d: %s
", p->positiono, s);

    if (p->REDRAW != DRAW_NONE) {
        if (p->REDRAW != DRAW_ONE) {
            if (p->REDRAW == DRAW_WINDOW) maxy = 0;
            XClearArea(ss->disp, XtWindow(ss->Canvas), 0, 0, 0,

            maxy, False);
        }
        DrawTreeEH(ss->Canvas, ss->des, NULL, NULL);
        p->REDRAW = DRAW_NONE;
    }

delete rp;

XawTextUnsetSelection(ss->TextWindow);
/* the end is 1 before the old end.
   this will skip the \n */
if (StartTextLocation > 0)
    MoveTextLocation(-1, p);
else
    EndTextLocation = StartTextLocation - 1;

XawTextSetInsertionPoint(ss->TextWindow, StartTextLocation);
void DrawTreeEH(Widget w, XtPointer xt, XEvent *Over, Boolean *)
{
    DrawEHStruct *des = (DrawEHStruct *)xt;
    DrawInfo *di = new DrawInfo;
    CTreePlayer *p = *(des->cpP);

    assert(des->gc);
    assert(p);

    int *Pathi[MaxDepth];
    char *s = p->String(p->position()-1);
    int PathSize = ParsePath(s, RayPathi);
    RayPathPlayer *rp = (RayPathPlayer *)(*(des->cpP))->Root();
    rp->SetPath(RayPathi, PathSize);
    ((CTreeNodePlayer *)rp->Pointer())->HIGH = DRAWHIGH;
    delete rp;

    di->disp = XDisplay(w);
    di->drable = XWindow(w);
    di->gc = des->gc;
    di->colors = des->colors;
    di->cm = des->cm;
    di->REDRAW = (Over != NULL)? DRAW_WINDOW : p->REDRAW;
    rp = (RayPathPlayer *)p->Root();
    DrawTreePreOrder(rp->Pointer(), di);
    delete di;

    Int Absolute = AbsoluteY(((CTreeNodePlayer *)rp->Pointer())->maxy + 1);
    ThumbSize = ((float)CANVASHEIGHT/(float)Absolute);
    XawScrollbarSetThumb(des->scroller, ThumbLocation, ThumbSize);
    delete rp;
}

void DrawTreePreOrder(CTreeNode *root, DrawInfo *di)
{
    DrawNode(root, di);
    if (root->ReflBranch) DrawTreePreOrder(root->ReflBranch, di);
    if (root->RefrBranch[rbiRedRefrBranch])
        DrawTreePreOrder(root->RefrBranch[rbiRedRefrBranch], di);
    if (root->RefrBranch[rbiGreenRefrBranch])
        DrawTreePreOrder(root->RefrBranch[rbiGreenRefrBranch], di);
    if (root->RefrBranch[rbiBlueRefrBranch])
        DrawTreePreOrder(root->RefrBranch[rbiBlueRefrBranch], di);
}

void ClearNode(CTreeNode *node, DrawInfo *di)
{
    CTreeNodePlayer *n = (CTreeNodePlayer *)node;
    XClearArea(di->disp, di->drable, DeviceX(n->x), DeviceY(n->y),
    
    350

    360

    370

    380

    390
NODEWIDTH, NODEHEIGHT,
False);

void DrawNode(CTreeNode *node, DrawInfo *di)
{
    CTreeNodePlayer *n = (CTreeNodePlayer *)node;
    if (di->REDRAW == DRAW_ONE)
        if (((n->REDRAW) && (n->HIGH == NODRAW)) return;

    n->REDRAW = 0;
    XColor *colors = di->colors;
    XColor temp;
    Status s;

    if (n->HIGH == DRAWLOW)
        n->HIGH = NODRAW;

    ClearNode(node, di);

    // is this the hilight node?
    if (n->HIGH == DRAWHIGH){
        // thicker border all else to default
        XSetLineAttributes(di->disp, di->gc, THICKLINEWIDTH,
            LineSolid, CapButt, JoinMiter);
        temp = colors[HIIWHITE_];
        s = XAllocColor(di->disp, di->cm, &temp);
        XSetForeground(di->disp, di->gc, temp.pixel);
    }

    // Find out how much to fill
    float fill;
    if (n->NodeFlag == ctnNOTHINGSET) fill = 0;
    else if (n->NodeFlag == ctnCONVERGED) fill = 1;
    else {
        fill = 0;
        if (n->NodeFlag & ctnREFLECTSET) fill += 1.0/5.0;
        if (n->NodeFlag & ctnGREFRACSET) fill += 1.0/5.0;
        if (n->NodeFlag & ctnRREFRACSET) fill += 1.0/5.0;
        if (n->NodeFlag & ctnBREFRACTSET) fill += 1.0/5.0;
        if (n->NodeFlag & ctnLOCALSET) fill += 1.0/5.0;
    }

    // Draw this node
    XDrawRectangle(di->disp, di->drable, di->gc,
        DeviceX(n->x), DeviceY(n->y),
        NODEWIDTH, NODEHEIGHT);
    if (fill>0) {
        int startx = DeviceX(n->x) + ((1-fill)*NODEWIDTH);
        XFillRectangle(di->disp, di->drable, di->gc,
            startx, DeviceY(n->y),
            (int)ceil(NODEWIDTH*fill), NODEHEIGHT);
    }
}
// if we highlighted then reset
if (n->HIGH == DRAWHIGH) {
    n->HIGH = DRAWLOW;
}

// thicker border all else to default
XSetLineAttributes(di->disp, di->gc, NORMALLINEWIDTH,
                    LineSolid, CapButt, JoinMiter);
    temp = colors[LOWWHITE];
    s = XAllocColor(di->disp, di->cm, &temp);
    XSetForeground(di->disp, di->gc, temp.pixel);
}

// if this node has a parent
if (n->parent) {
    CTreeNodePlayer *p = (CTreeNodePlayer *)n->parent;
    /* draw a line from the center of the left side of this node
to the center of the right side of the parent node */
    if (p->ReflBranch != n) {
        if (p->ReflBranch[0] == n) temp = colors[RED];
        else if (p->ReflBranch[1] == n) temp = colors[GREEN];
        else temp = colors[BLUE];
        s = XAllocColor(di->disp, di->cm, &temp);
        XSetForeground(di->disp, di->gc, temp.pixel);
    }

    int nx = DeviceX(n->x);
    int ny = DeviceY(n->y)+NODEHEIGHT/2;
    int px = DeviceX(p->x)+NODEWIDTH;
    int py = DeviceY(p->y)+NODEHEIGHT/2;
    XSetLineAttributes(di->disp, di->gc, THICKLINEWIDTH,
                        LineSolid, CapButt, JoinMiter);
    XDrawLine(di->disp, di->drable, di->gc,
               px, py, nx, ny);
    XSetLineAttributes(di->disp, di->gc, NORMALLINEWIDTH,
                        LineSolid, CapButt, JoinMiter);

    // assume that the old foreground was white and reset
    if (p->ReflBranch != n) {
        temp = colors[LOWWHITE];
        s = XAllocColor(di->disp, di->cm, &temp);
        XSetForeground(di->disp, di->gc, temp.pixel);
    }
}

void PlayCB(Widget, XtPointer Data, XtPointer)
{
    printf("PlayCB\n");
    if (PlayTimer) return;

    PlayStruct *ps = (PlayStruct *)Data;
}
PlayTimer = XtAppAddTimeOut(ps->app, INTERVAL, TimerPlayBackCB, Data);

void StopCB(Widget, XtPointer, XtPointer)
{
    printf("StopCB\n");
    if (!PlayTimer) return;
    XtRemoveTimeOut(PlayTimer);
    PlayTimer = 0;
}

void TimerPlayBackCB(XtPointer Data, XtIntervalId *)
{
    printf("TimerPlayBackCB\n");
    PlayStruct *ps = (PlayStruct *)Data;
    FStepCB(NULL, ps->ss, NULL);
    // if we're not at the end then restart the timer
    if (!(*(ps->ss->des->cpP))->done())
        PlayTimer = XtAppAddTimeOut(ps->app, INTERVAL, TimerPlayBackCB, Data);
    else {
        PlayTimer = 0;
        printf("End Of Playback !\n");
    }
}

void SkipCB(Widget, XtPointer Data, XtPointer)
{
    SkipInfo *si= (SkipInfo *)Data;
    if (PlayTimer) StopCB(NULL, NULL, NULL);
    XtSetSensitive(si->SkipButton, FALSE);
    XtVaSetValues(si->SkipDialog, XtNvalue, "", NULL);
    XtPopup(si->SkipShell, XtGrabNonexclusive);
}

void SkipApplyCB(Widget, XtPointer Data, XtPointer)
{
    SkipInfo *si= (SkipInfo *)Data;
    String string;
    XtPopdown(si->SkipShell);
    XtSetSensitive(si->SkipButton, TRUE);
    string = XawDialogGetString(si->SkipDialog);
    printf("applying: %s\n", string);
    int k;
    for (k = 0; ((string[k] != '\0') && (isalnum(string[k]))); k++);
if (isdigit(string[k])){
    CTreePlayer *p = *(si->ss->des->cpP);
    k = atoi(string);
    int pos = p->position() - 1;
    if (k != pos) {
        if (k < pos) {
            MoveTextLocation(k - pos + 1, p);
            while ((p->position() - 1) > (k+1))
                p->step(-1);
            p->REDRAW = DRAW_WINDOW;
            BStepCB(NULL, si->ss, NULL);
        } else {
            MoveTextLocation(k - pos - 1, p);
            while ((p->position() - 1) < (k-1))
                p->step();
            p->REDRAW = DRAW_WINDOW;
            FStepCB(NULL, si->ss, NULL);
        }
    }
}
}

void SkipCancelCB(Widget, XtPointer Data, XtPointer)
{
    SkipInfo *si= (SkipInfo *)Data;
    XtPopdown(si->SkipShell);
    XtSetSensitive(si->SkipButton, TRUE);

    printf("cancelling\n");
}

void MoveTextLocation(Int n, CTreePlayer *p) {
    if (n == 0) return;
    int k = 0;
    char *t;
    if (n > 0) {
        for (; k < n; k++) {
            /* the new start is 1 after the old end.
               this will skip the \n */
            StartTextLocation = EndTextLocation + 1;
            long size;
            for (size = p->ClearTextSize(), t = p->ClearText(),
                EndTextLocation = StartTextLocation;
                (EndTextLocation < size-1) && (t[EndTextLocation] != '\n');
                EndTextLocation++);
        }
    } else {
        for (; k > n; k--) {
            EndTextLocation = StartTextLocation - 1;
            for (t = p->ClearText(), StartTextLocation = EndTextLocation - 1;
                ((StartTextLocation > 0) && (t[StartTextLocation] != '\n'));
                StartTextLocation--);
            if (t[StartTextLocation]== '\n') StartTextLocation++;
void ScrollProc(Widget, XtPointer Data, XtPointer Data2)
{
    StepStruct *ss = (StepStruct *) Data;
    float percent = ((float) abs((int)Data2)) / ((float)SCROLLLEN);
    CTreePlayer *player = *(ss->des->cpP);
    RayPath *rp = player->Root();
    int devmaxy = min(DeviceY(((CTreeNodePlayer *)rp->Pointer())->maxy + 1),
                     CANVASHEIGHT);
    int Absolute = AbsoluteY(((CTreeNodePlayer *)rp->Pointer())->maxy + 1);
    delete rp;

    if (Absolute < CANVASHEIGHT) {
        if (TRANSLATION != 0) {
            TRANSLATION = 0;
            XClearArea(ss->disp, ss->w, 0, 0, 0, devmaxy, False);
            ThumbLocation = 0;
            DrawTreeEH(ss->Canvas, ss->des, NULL, NULL);
        }
    } else if ((Absolute >= CANVASHEIGHT) || (TRANSLATION != 0)) {
        XClearArea(ss->disp, XtWindow(ss->Canvas), 0, 0, 0, devmaxy, False);

        TRANSLATION = -(int) (percent * Absolute);
        ThumbLocation = percent;
        DrawTreeEH(ss->Canvas, ss->des, NULL, NULL);
    }
}

void JumpProc(Widget, XtPointer Data, XtPointer Data2)
{
    StepStruct *ss = (StepStruct *) Data;
    float percent = *((float) Data2);
    CTreePlayer *player = *(ss->des->cpP);
    RayPath *rp = player->Root();
    int devmaxy = min(DeviceY(((CTreeNodePlayer *)rp->Pointer())->maxy + 1),
                     CANVASHEIGHT);
    int Absolute = AbsoluteY(((CTreeNodePlayer *)rp->Pointer())->maxy + 1);
    delete rp;

    if (Absolute < CANVASHEIGHT) {
        if (TRANSLATION != 0) {
            TRANSLATION = 0;
            XClearArea(ss->disp, ss->w, 0, 0, 0, devmaxy, False);
            ThumbLocation = 0;
            DrawTreeEH(ss->Canvas, ss->des, NULL, NULL);
        }
    } else if ((Absolute >= CANVASHEIGHT) || (TRANSLATION != 0)) {

    223
XClearArea(ss->disp, XtWindow(ss->Canvas), 0, 0, 0, devmaxy, False);

TRANSLATION = -(int) (percent * Absolute);
ThumbLocation = percent;
DrawTreeEH(ss->Canvas, ss->des, NULL, NULL);
}
}

! CTPlayer.Xdefaults
!
! This file contains some default X resource values for the
! CTPlayer application
!

CTPlayer*CanvasWindow.fromVert: box
CTPlayer*Canvas.borderColor: black

CTPlayer*CanvasScrollBar.fromHoriz: Canvas
CTPlayer*CanvasScrollBar.length: 500

CTPlayer*reverseVideo: on

CTPlayer*box.width: 400

CTPlayer*TextWinForm.fromVert: CanvasWindow
CTPlayer*TextWinHeader.width: 600
CTPlayer*TextWinHeader.justify: left
CTPlayer*TextWindow.fromVert: TextWinHeader
CTPlayer*TextWindow.width: 600
CTPlayer*TextWindow.height: 100
CTPlayer*TextWindow.displayCaret: False

CTPlayer*skipDialog.label: Enter The Location To Skip To:
CTPlayer*skipApplyButton.label: Apply
CTPlayer*skipCancelButton.label: Cancel
CTPlayer*skipDialog.value:
Bibliography


