6-4-1996

Object Oriented Scenes for Virtual Light

Jonathan A. Moore
Dartmouth College

Object Oriented Scenes
for
Virtual Light

An object oriented structure for ray tracing software

Jonathan A. Moore
class of 1996
Honors Thesis
John M. Danskin,
Advisor
Computer Science
Dartmouth College
June, 4 1996
PCS-TR96-291

Object Oriented Scenes
for
Virtual Light

Introduction

Ray tracing is one of many way to use a computer to generate an image. Ray tracers produce images by simulating light. Normally, ray tracing refers to *backwards* ray tracing: rays that represent light are shot from an "eye" into a scene. Objects in the scene reflect and transmit the ray. Eventually, it reaches a light source. One ray is shot to determine the color of each pixel of the final image.

Writing a ray tracer can be a daunting endeavor. A great number of details might distract one from the interesting part of the algorithm that actually models light. Eliminating these details was the main goal of my thesis project. The software I have written can be divide into three parts: the virtual frame buffer, the support classes and the ray tracing abstract base classes.

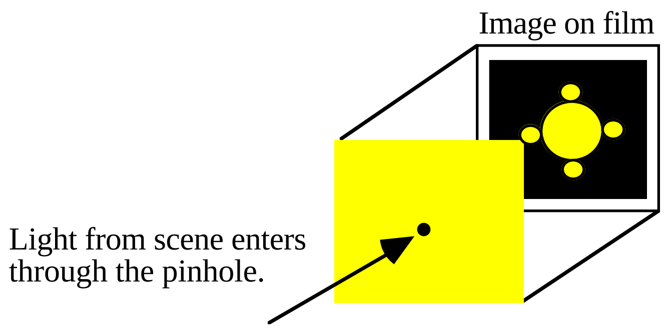
The virtual frame buffer class, `vfb`, provides a simple means of rendering and studying the final image produced by a graphical algorithm. A user may draw to the `vfb` with simple calls to `put_pixel()`, but the flexibility to create new schemes for representing pixels in memory is also built into the `vfb`. The `vfb` implements a Tk widget with many commands for accessing and resizing the pixels. With a little knowledge of Tk, a simple script can implement a graphical user interface with individual pixel access and features like load, save, pan, and zoom.

The support classes provide an elegant notation for the equations involved in ray tracing. A *ray* is a geometric object used to find a *color*. It is defined by a *vector* and *point*. Objects often must be transformed by *matrices*. None of these types are part of standard C++. In order to implement a ray tracer all of these classes and their operations must be represented. The `vector`, `point` and `matrix` classes support homogenous coordinates which allow one to perform linear transformations uniformly. The same transformation `matrix` may be used to modify a `point`, `vector`, another `matrix`, or any object defined by homogenous objects.

The ray tracing base classes and associated classes provide a object oriented structure for defining the objects that make up a scene. Two abstract base classes, `rtab_component` and `rtab_composite`, implement the *Composite* pattern as discussed in Design Patterns.[1] This structure was designed with some specific goals in mind: allow users to implement their own primitive objects, composite objects and ray tracing algorithms, but force all of these elements to be interchangeable. These goals imply that once an algorithm has been implemented any objects based the abstract classes, either primitive or composite, will automatically work with the algorithm and visa versa. This allows students to focus on the particular ray tracing algorithm they are studying, or to implement new objects without worrying about an algorithm. Because an object may actually be a group of primitive objects, another goal became necessary: Algorithms should interface with primitive and composite objects in exactly the same way.

Pinhole Camera Model[2] and Ray Tracing Concepts

Taking a photograph of a scene in the physical world is one way of creating a two dimensional image of a three dimensional world. The simplest way to make a photograph is using a pinhole camera. A pinhole camera is a light proof box with a piece of film attached to one of the inner sides of the box. A small hole is made in the center of the side opposite the film and covered. A picture is taken by allowing light to enter the box through the pinhole.



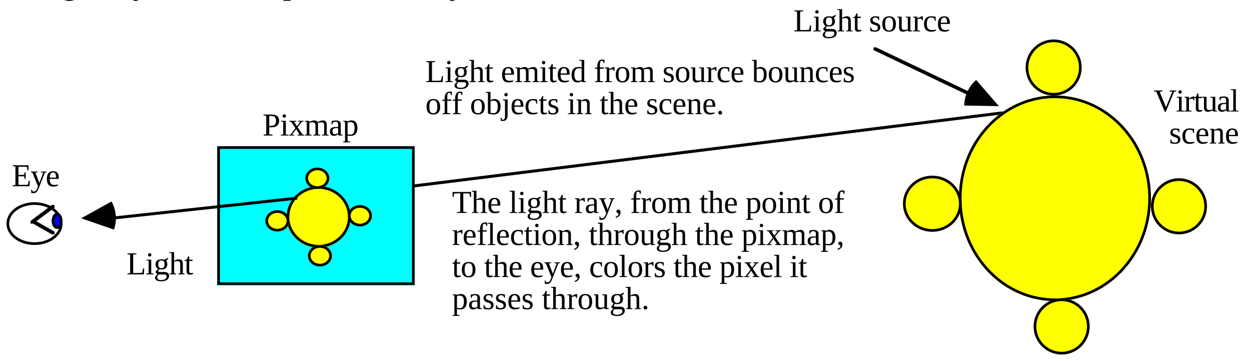
Light particles, photons, travel along straight lines called rays. Light sources continuously emit many photons of various colors in all directions. Photons that hit the film inside the camera cause a chemical reaction to color the film. The effect of the photons on the film is additive; the color of a particular point on the film is an average of the all the photons that hit that point. If a blue photon were to hit a point on the film and later a red photon hit the same point, it would appear purple, a combination of their colors.

The hole in the pinhole camera is very small; thus, it limits the number of photons that hit the film. For a specific point on the film only photons moving along a path through the pinhole to that point will reach the film. In theory, a pinhole large enough for one photon would produce the best image by allowing each point on the film to be colored by photons moving along just one path. The picture would be very sharp;

however, it would require a long time to produce given the extremely small amount of light allowed to enter the camera at one time.

A light source emits photons. The photons are reflected, refracted and absorbed by objects in the scene being photographed. A very small group of photons pass through the pinhole. The film records the outside scene as it is colored by photons moving along the rays from the pinhole to each specific point on the film.

Ray tracing produces two dimensional images of virtual three dimensional space by simulating the process of the pinhole camera. There are many variations on the general idea, but a few concepts in the model are usually changed. The pinhole is replaced by a conceptual point sized “eye”. The film is replaced by a viewing window or a pixmap and it is placed between the eye and the virtual scene. Instead of film recording rays that pass from the outside world into the camera along paths from the pinhole to each specific point on the film, the pixmap of the ray tracing algorithm records rays that originate in the virtual scene and pass through a specific pixel of the pixmap along a ray from that pixel to the eye.


Light source
Light emited from source bounces off objects in the scene.
Virtual scene
Pixmap
Eye
Light
The light ray, from the point of reflection, through the pixmap, to the eye, colors the pixel it passes through.

Forward ray tracing algorithms model light by following protons progress from the light source, off the objects, through the pixmap, to the eye. For real photons this happens almost instantaneously, but to simulate these photons a computer must perform many calculations whenever the photons encounter objects in the scene. Light sources emit photons continuously in all possible directions. Very few of these photons eventually enter the pinhole in the camera. A computer can only simulate this by tracing rays one at a time in random directions. Thus, forward ray tracing algorithms are very slow. If the algorithm were run for an infinite amount of time, a near perfect image would be produced; however, that much time is usually not available.

Although photons are emitted in all directions the vast majority of these never pass through the pinhole or reach the eye. A great deal of time is wasted simulating the photons that will not effect the image. *Backwards* ray tracing attempts to eliminate these rays. This algorithm reverses the simulation by tracing rays from the eye, through a specific point on the pixmap, to objects, and back to the light source. Thus, backwards ray tracing only studies the particular rays that would have reached the eye if traced forwards. Unless otherwise specified, *ray tracing* generally refers to the *backwards* ray tracing algorithms.

Photons move at the speed of light along straight paths in one direction. Ray tracing algorithms simulate photons motion with rays. Usually a ray does not model the actual photon, instead it represents the photon's entire path. A ray is defined by its origin, point and direction vector. Ray tracing algorithms begin their backwards simulation by defining a number of rays. Each ray's origin is the "eye". In general, a vector may be defined by subtracting one point from another. A ray's direction is the vector defined by subtracting the eye point from one point of the pixmap. One ray is studied for each point of the pixmap.

Each of theses rays is tested for an intersection with the objects in the scene. If the ray does not intersect any object it is disregarded. If it does intersect an object, the color observed at the point of intersection determines the color of the ray's pixel. The color observed at the point of intersection is found by recursively shooting new rays. The observed color is a combination of the object's color and the colors found with the new rays. Many of these rays' direction vectors are calculated using the surface normal of the object. One ray is directed at each of the lights in the scene. If the light rays do not intersect anything before hitting the light, the color of the light is returned. Other rays are directed according to the particular algorithm being used. In general they simulate reflection, refraction, and other properties of light. Each of these rays is tested for intersection with other objects. If an intersection is found the algorithm recurses on that point.

The general algorithm implies that every ray shot into a scene must be tested for intersection with every object in the scene. The total number of rays shot into the scene not only includes those originating at the eye but also those originating at each intersection point. A common approach for improving efficiency groups objects to eliminate intersection tests. For example, one hundred objects are grouped together into a group object. A ray is shot into the scene and tested for intersection with the group object. If the ray does not intersect the group object we know that it will not intersect any of the one hundred objects in the group, thus we save 100 intersection tests.

Homogenous Coordinates

points and vectors

Normally, 3 coordinates (x, y, z) represent a point in three dimensional space. This presents a problem. In many graphic algorithms it is necessary to transform groups of points by rotating, scaling and translating the group. These operations are accomplished by applying a matrix to the group of points. Translation moves a point by adding a one column matrix (T) to it:

$P' = T + P$.

Other transformation are applied by multiplying the point by a matrix. A point is rotated around the origin by multiplying it by a 3 by 3 matrix (R).

$P' = R * P$.

Since the matrices are of different sizes and the transformations do not rely on the same operations, the transformations cannot be treated uniformly or composed into one convenient matrix.

Homogenous coordinates overcome this problem. Homogenous coordinates use 4 coordinates (x, y, z, W) to represent points in 3 dimensions. The point (0, 0, 0, 0) is not allowed. The points (x, y, z, W) and (x', y', z', W') are equal if they are multiples of each other. The 3 dimensional sub-space W = 1 of the homogenous coordinates 4 space represents Cartesian 3 space. These two ideas imply that all homogenous points (x, y, z, $W \neq 0$) give the same point in the 3 dimensional sub-space (x/W, y/W, z/W, 1). Dividing through by W is called homogenizing the point.

When W = 0 the point is said to be at infinity and only gives a direction. Points at infinity represent 3 dimensional vectors. In the homogenous coordinate system 3 dimensional points and vectors are represented the same way, as 4 dimensional points. However, making the distinction between finite points and points at infinity (vectors) can be helpful when using homogenous coordinates. A point is normally written as a matrix with only one column,

$$\begin{bmatrix} x \\ y \\ z \\ W \end{bmatrix},$$

and vector is expressed as a row, $\begin{bmatrix} x & y & z & 0 \end{bmatrix}$.

A point may be translated by adding a vector to it:

P(x, y, z, 1) + V(dx, dy, dz, 0) = P(x+dx, y+dy, z+dz, 1),

or by multiplying a matrix by it (as a column):

$$P' = M * P = \begin{bmatrix} 1 & 0 & 0 & dx \\ 0 & 1 & 0 & dy \\ 0 & 0 & 1 & dz \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x+dx \\ y+dy \\ z+dz \\ 1 \end{bmatrix}$$

A vector is multiplied by the matrix to transform it,

V' = V * M.

All other transformations are carried out using 4 by 4 matrices.

The distinction between points and vectors is made because some operations do not make sense for finite points. We might try to scale a finite point by multiplying it by a scalar. This gives the same point when the coordinates are homogenized,

s * P(x, y, z, W) = P'(sx, sy, sz, sW).

To scale a point like this we must add a vector or multiply by a matrix (see above).

We may also wish to find a point P' using two finite points P1 and P2. Adding two finite points also does not work:

P1(x, y, z, W) + P2(x', y', z', W') = P(x+x'/W+W', y+y'/W+W', z+z'/W+W', 1) not P'(x+x', y+y', z+z', 1).

To get P' we must use the vector ,

V = P2 - (0, 0, 0, 1),

in the transformation,

P' = P1 + V.

Design Patterns and The *Composite* Pattern

The *Composite* pattern as described in Design Patterns was used as a guide while developing the ray tracing abstract base classes. Design Patterns is a well known survey of object oriented design ideas. The introduction provides a general discussion of object-oriented design, while the bulk of the text catalogs a number of proven Design Patterns that make use of object oriented technology.

Upon a first reading, Design Patters provides so much information that it is confusing and seems of little use. Without the understanding of its organization and terminology that comes from several readings, determining which patterns apply to a problem is difficult. However, the general suggestions made in the introduction are immediately beneficial. After working on a problem and determining some design goals, a second look at Design Patterns becomes useful. The patterns bring clarity to a design, help answer key questions, and provide a language for documentation. In general, after a problem's specifications and the programmer's goals have been well thought out Design Patterns becomes most useful. The patterns provide ideas for solving a problem and a language for thinking, speaking and writing about it, but do not provide a finished solution.

Each pattern includes several sections to describe the pattern and its uses. The *intent*, *motivation*, *applicability* and *consequences* sections help in choosing a pattern to apply to a problem. Once a particular pattern has been chosen the *structure, participants*, *collaborations, implementation, sample code* and *related patterns* sections describe how to go about applying the design pattern.

The *Composite* pattern is broken up into the *participants*, *component*, *leaf*, *composite* and *client*. The *component* declares the interface for objects in the composition, implements default behavior and declares an interface for managing children. *Leaf participants* have no children and define the behavior for primitive objects. The *composite participant* defines the behavior for components having children, stores child components, and implements child related operations. The *client* manipulates the *leaf* and *composite participants* through the *component's* interface.

The *intent* and *consequences* of the *Composite* pattern provide the closest match to the design goals of the ray tracing base objects. The *intent* explains that "clients may treat individual objects and composite objects uniformly." The noteworthy *consequences* of the composite pattern are: primitive objects may be composed into more complex composite objects which may recursively be composed into still more complex composite objects; wherever a client expects a primitive, it can take a composite; the client code is simplified because it "avoids having to write tag-and-case-statement-style functions over the classes that define the composition"; and "newly defined Composite or Leaf subclasses work automatically with existing structures and client code."[3]

The *implementation* section discusses some key issues to consider while when using the *Composite* pattern. Of concern in working out the ray tracing base classes was the discussion about where to declare child operations, in the *component* or *composite* participants.[4] If they are declared in the *component*, a situation occurs where a child operation could be called on a *leaf*. But, if they are left to the *composite* class we lose the *composite-leaf* transparency.

The *Composite* Pattern in `rtab_component` and `rtab_composite`

In ray tracing, a *component* is any ray traceable object: a polygon, box, or scene. The *leaf* is a primitive object, such as a sphere or polygon. The *composite* is any combination of primitive objects and composite objects. For example, a box is a composite of six rectangular polygons, or a scene may be a composite of 2 boxes and another polygon. The *clients* are the ray tracing algorithms.

The consequences of using the *Composite* pattern are the key to how it meets the design goals. I wanted to make ray tracing algorithms easy to implement: "The Composite pattern makes the client simple."[5] I wanted to make it simple to add new types of objects and contain them in various types of composite objects: "The Composite pattern makes it easier to add new kinds of components. Newly defined Composite of Leaf subclasses work automatically with existing structures and client code."

The *component* and *leaf participants*

`rtab_component`

There are two general classes of ray traceable objects, primitive and composite. My goal of simplifying ray tracing algorithms depends on objects of the different classes being treated in exactly the same manner. The *Composite* pattern achieves this transparency in the *component* participant. The *component* declares the interface for objects in the composition, implements default behavior and declares an interface for managing children.

My *component participant* is the abstract base class `rtab_component`. It declares a common interface for primitive and composite objects. Primitive ray traceable objects must intersect themselves with rays, have a surface normal, know if they contain a point, know their size, and have a model for their surface physics. Composite objects must be able to intersect themselves with rays, know their size, insert new composite or primitive objects, and free memory used by their sub-objects when they go out of scope. `rtab_component` includes an interface for all of the above items.

Leaf participants have no children and define the behavior for primitive objects. They are the concrete primitive classes that inherit from `rtab_component` and define or redefine the surface physics, `normal()`, `intersect()`, `min_point()`, `max_point()` and `contains()` members appropriately.

When designing `rtab_component` I looked carefully at which functions should have default behavior. The surface physics model interface given in `rtab_component` is based on the Hall Shading Model[6]. Objects used by a particular algorithm may be defined by a different person than the algorithm itself. Since all algorithms do not include the same members of a surface physics model, there may be discrepancies between the model used by the primitive object and the model used by the algorithm. Default behavior is given for the entire surface physics model. The default behavior attempts to reflect what happens if a member is not defined. For example, the default `kdr`, 0, means that there is no diffuse reflection for the object. This ensures that primitive objects which do not implement a particular member of the model may be used with algorithms that use that member.

Since all algorithms will not rely on bounding volumes I provide default behavior for the functions concerning an objects size, `min_point()` and `max_point()`. By default they return `NaN_point`. If they are not redefined the object cannot be inserted into a bounding volume. If a ray that intersects an object originates inside the object the surface normal at that point must be negated. Given a point (like the ray's origin) the `contains` member is defined to return if that point is inside the object. The `contains()` member returns `false` by default because it only makes sense for objects that contain a volume inside a surface. For example, can fog contain a point within in its surface?

While the members of the surface physics model that are used by an algorithm may vary, there are some functions that always must be defined for primitive objects. These members, `normal()` and `intersect()`, have been declared pure virtual or abstract (`normal(point p) = 0`). Without a `normal` or `intersect` a object is totally undefined.

The child management members, `insert()` and `delete_all_children()`, have no meaning for primitive objects, but their interface is included in `rtab_component`. Maintaining the transparency between primitive and composite objects outweighs eliminating the possibility of an inappropriate call to one of a primitive's child management members. To make their declaration safer, these members print error messages by default in `rtab_component`.

The *composite participant*

`rtab_composite`

The *composite* participant defines the behavior for components having children, stores child components, and implements child related operations. It is important that a user be able to design new, interchangeable composite objects. Instead of implementing one concrete composite class I have defined a second abstract base class, `rtab_composite`, to serve as a template for concrete composite classes.

Many concrete composite objects will implement some type of tree structure. `child` has been declared for storing an array of pointers to sub-objects. The constructor initializes `child` to point to `NULL`. Assuming the user allocates space for `child` objects using `new`, the default behavior of `delete_all_children()`, called by the destructor, should take care of freeing memory when composite objects go out of scope.

`insert()` may still be undefined (thus returns the error message from `rtab_component`) for composite objects. We may have a composite object like box, which is always made up of 6 polygons. We would never want to insert objects into such an object.

The `intersect()` member has been declared pure virtual or abstract to force its redefinition. To complete concrete classes the user will have to implement `intersect` (and `insert()` if appropriate). This will most likely recursively call itself on one of the children until some primitive object is reached. Determining the relationship between children and the method for choosing a child to recurse on will be the bulk of a concrete class' s definition. A private `find_child()` function will usually implement this; however, since the function would be private, I did not declare it as part of the interface.

Finally, the *client* participants manipulate the objects through the *component* ‘s interface. I have defined a `scene` class that implements a ray tracing algorithm. It also includes functions for inputting objects from a file. It is mainly meant to be an example; however, by inheriting `scene` and redefining the `ray_trace()` member it could be used as the framework for other ray tracers.

Support Classes

There are a few classes that do not fit into the *Composite* pattern, but are essential to making ray tracing and other graphics algorithms easier to write: `color`, `vector`, `point`, `color_point`, `matrix`, `ray` and `intersection`. These classes are provided to simplify the code in ray tracing algorithms.

`color`, `vector` and `point` are all based on an abstract class `four_tuple`. `four_tuple` defines most of the operations that are carried out on `colors`, `vectors` and `points`. The `color`, `vector` and `point` classes determine which operations are defined between the various classes. All arguments to the operations are passed as constant references so that copies of the objects are not created whenever the operations are called. The transformations from `color`, `point` and `vector` to `four_tuple` are all done `inline` to avoid the overhead associated with function calls.

When ray tracing a scene, a large percentage of the resources go into finding the intersection of a ray and an object. Container objects divide space to improve the efficiency of finding ray-object intersections. If a ray does not hit a container object then it will not intersects the objects within the container. The `intersection` class provides class variables for keeping track of how many intersection tests were performed, how many tests resulted with no intersection found, and how many intersections were found with container objects but no primitive objects. It does this by incrementing one class variable each time a new `intersection` object is created. Two more class variables are incremented in the destructor depending on what type of intersection was found. This code is all enclosed in `#ifdef's`. When the intersection performance flag is not set, the code is not compiled and it does not slow down the intersection tests at all.

The `intersection` class has two members for returning a ray object intersection, `t` and `object_handle`. Given a `ray` and object, `t` is the value such that the point of ray-object intersection is given by the equation:

intersection point = `ray_origin + t * ray_direction`.

The `ray` class includes an inline member function, `t_point`, for converting `t` to a `point`, and `operator*` has been overloaded to perform the conversion. `operator*` has also been overloaded to define `ray * matrix`.

The `object_handle` is a pointer to the object that the ray intersected. It should be noted that in a case where a ray intersects a container object but no primitive object, `t` should equal zero and `object_handle` should point to the container object. This allows for testing the effectiveness of the container objects by comparing the `intersection` class variables (see above).

The Virtual Frame Buffer

and

`vfb_pixel`

The virtual frame buffer provides a simple means for rendering images generated with graphics algorithms including ray tracing. Users do not have to worry about color maps, graphics contexts, lighting models, windows or any of the other headaches usually involved with displaying graphics on UNIX systems. The only way to write to the buffer is one pixel at a time with `put_pixel()`. The second feature of the virtual frame buffer is the extreme flexibility of how pixels are represented. Users of the virtual frame buffer have control over how pixels are represented via the `STORAGE_CLASS` `typedef` in `pixel.hh`. The `vfb_pixel`'s constructor has been overload so pixels may be thought of as black or white; shades of gray; rgb colors; z-buffered rgb colors; rgb alpha colors; or antialiased, z-buffered, rgb colors. If `STORAGE_CLASS` is not set to support the type of pixel you want to use, the constructor will automatically make the correct adjustments to approximate that pixel type.

No matter how pixels are stored, they always have the same interface which is declared in the abstract base class `pixel_interface`. The `vfb_pixel` class inherits from the `STORAGE_CLASS` which inherits from `pixel_interface` and defines its members. Class `vfb_pixel` does nothing more than implement a group of constructors. Each of these constructors allows a different concept of a pixel (black/white, gray, rgb, etc.). All of the constructors call a protected function, `store_pixel()`, inherited from `STORAGE_CLASS`. `store_pixel()` does nothing different for the various concepts of a pixel; however, the way it is called defines what type of pixel is created. For example, calling `store_pixel()` with its red, green and blue arguments all set to the same `float` variable implements a shade of gray pixel. While calling it with the red, green and blue arguments set to 3 different `float` variables implements one of the rgb pixel types.

The vfb widget

Originally the virtual frame buffer used Open GL functions to create a window and render virtual pixels. The virtual pixels needed to be drawn only as simple geometric objects like points, filled circles or rectangles, but developing a user interface demanded a high level of functionality. Open GL provided a wealth of support for rendering images with lighting models, clipping, and lots of other complicate graphics functionality that I did not need. GL's `AUX` library provided minimal support for creating a user interface. Writing the virtual frame buffer using GL also bound it to the SGI workstations. The SGI's most students have access to are known to be rather slow. The `vfb` was intended to be used for ray tracing which is a slow process to begin with, so it was rather important to port the `vfb` away from the SGI's.

These problems inspired me to re-implement the virtual frame buffer's user interface as a Tk widget. Tk is an extension of the Tcl scripting language for creating user interfaces; thus, It provides plenty of support for designing the virtual frame buffer's interface. Tk provides a canvas widget that I could have used to implement the `vfb`; however, all of the `vfb'`s functionality would have been carried out by Tk's interpreter. A ray tracer putting hundreds of thousands of pixels in the buffer would be extremely slow. I also wanted the `vfb` to be accessible within my other C++ classes.

I decided to implement a Tk widget and include it as part of the `vfb` class with three ideas in mind: make redraws as fast as possible, make it simple to use, and provide commands for all necessary functionality. Tk essentially takes care of making it simple to use. The `vfb` widget is no more difficult to use than any of Tk's built in widgets. I created a command for each member of the class that might need accessing at runtime. This is accomplished in the member function, `vfbWidgetCmd()`. Each command makes up one section of a large `if`, `else if`, statement. (See the user manual for a complete listing of commands and configuration options.)

The most difficult part of designing the `vfb` widget was making the redraws fast. Redrawing involves accessing the `vfb`'s array of pixels hundreds of thousands of times. Anytime the virtual pixel size changes the entire section of the `vfb` displayed by the widget must be redrawn. This is carried out in the `vfbDisplay()` member function. The initial redraw algorithm called `get_pixel()` for each of the pixels visible in the widget and drew a virtual pixel to a pixmap. After all the virtual pixels were drawn on the pixmap, the pixmap was copied to the widget.

Each virtual pixel was drawn as a filled circle, a rather slow operation. I assumed that drawing one filled circle was slower than copying a small pixmap to a larger pixmap. Based on this assumption, the next idea was to draw just one virtual pixel to a small pixmap. Inside the loop I change the foreground color and copied the small pixmap to the locations where virtual pixels should be drawn. This idea did not seem to work. In fact it apparently is slower than the previous method. I have left the type of redraw as a configuration option.

The final version of the `vfb` maintains a large pixmap. When the widget needs to be redrawn the appropriate area of this pixmap is copied to the widget. Consequently, most redraws are extremely fast. However, anytime the virtual pixel size changes the redraws are extremely slow, because the entire `vfb` must be redrawn. The size of the pixmap has been left as a configuration option, with the default set to twice the width and height of the screen When the widget needs to display a portion of the `vfb` that did not fit on the large pixmap, that portion must be redrawn. The remainder of the pixmap is copied from its existing location. A larger pixmap requires fewer redraws (except for pixel size changes), but takes more time to redraw and uses more memory. If the pixmap is smaller than the widget, the `vfb` will not display correctly.

Ray Tracing Examples

This section includes several code fragments to show how to use the ray tracing abstract base classes and support classes. First, `class sphere` illustrates how to derive a primitive concrete class from `rtab_component`. It is an instance of the *leaf participant* of the *Composite* pattern. `sphere` is a complete primitive object.

```
class sphere: virtual public rtab_component {
private:
  double sphere_ksr;
  double sphere_kdr;
  double sphere_n;
  double sphere_zero_t;

public:
  color_point sphere_center;
  double      sphere_radius;

  sphere() {};

  // Surface physics
  color  clr()    {return sphere_center.clr();}
  double ksr()    {return sphere_ksr;}
  double kdr()    {return sphere_kdr;}
  double n()      {return sphere_n;}
  double zero_t() {return sphere_zero_t;}

  istream& read(istream& s)
       {s >> sphere_center >> sphere_radius >> sphere_ksr >> sphere_kdr >> sphere_n >> sphere_zero_t;
        return s;}

  ostream& write(ostream& s);
       {s << sphere_center << sphere_radius << sphere_ksr << sphere_kdr << sphere_n << sphere_zero_t;
        return s;}

  bool contains(point p)
       {return (sphere_center - p).length() <= sphere_radius - sphere_zero_t;}

  vector normal(point p);  {return (p - sphere_center.pnt()).unit();}

  intersection intersect(ray r);
 };
```

The surface physics member functions override the default behavior of `rtab_component` and return the corresponding private variables. `read()` uses the overloaded `operator >>` to set the values of sphere's public variables and private surface physics variables. `write()` outputs the same information using `<<`. `normal()` and `contains()` also rely on operators overloaded for the support classes in their calculations. The vector normal to a sphere at a point on its surface is found by subtracting the center from the point. If the distance from a point to the center of a sphere is less than the radius of the sphere the point lies within the sphere. `intersect()` is the only non inline member function:

```
intersection traceable_sphere::intersect(ray r) {

  // These variables are used to solve the quadratic equation that results from
  // combining the ray's and sphere's equations.
  vector g = r.origin() - sphere_center;
  double b = 2 * (r.direction() * g);
  double c = (g * g) - (sphere_radius * sphere_radius);
  double d = (b * b) - 4 * c;

  double t0, t1;
  intersection i;

  // A root is real iff d >= 0.
  if (d >= 0) {
    t0 = (-b - sqrt(d)) / 2;
    t1 = (-b + sqrt(d)) / 2;
  }

  // Rays only extend in one direction so t must be positive
  if ((t0 > zero_t()) || (t1 > zero_t())) {

       // intersect should return only the first ray-sphere intersection so
       // we return the smaller of the positive t's.
       if (((zero_t() < t0) && (t0 < t1)) || ((t1 < zero_t())))
         i.t = t0;
       else
         i.t = t1;

       i.object_handle = this;
   }

  //The ray does not intersect the sphere
  else {
       i.t = 0;
       i.object_handle = 0;
   }
  return i;
}
```

The intersection is found by plugging the equation of the ray,

```
p = ray.origin() + t * ray.direction(),
```

into the equation of the sphere,

```
(p - sphere_center) * (p - sphere_center) = sphere_radius,
```

and using the quadratic equation to solve for `t`. Since rays only extend in one direction we only consider positive values of `t`. `intersect()` should indicate the first ray-sphere intersection, so it returns the smaller of the positive `t`'s. If the ray intersected the object at its origin, `t` would equal zero. As an algorithm recursively shoots rays from a ray-object intersection the original intersection point should not be considered. For these two reason a ray's origin is not considered part of the ray.

After implementing a new ray traceable class we must also update the `get_type()` function (see the User's Manual Appendix for details on doing this). For `sphere`, the following line is inserted into `get_type()`'s `if then-else-statement`:

```
else if (strcmp(name, "SPHERE") == 0) return new sphere;
```

Now that the new concrete class has been defined, the *client* code to read object definitions from a file and insert the objects into a container, `objects`, is simple:

```
object_list* objects;
rtab_component* object;
do {
     object = get_type(f);
     if (object) {
        object->read(f);
        objects->insert(object);
     }
  while (object);
```

`get_type()` works by reading a string from `f`. It compares the string to a number of class names. If the string matches a class name an instance of that class is created with `new`, and a pointer to the variable is returned. If the string does not match any of the class names `NULL` is returned. Since `get_type()` allocates a new variable of a class derived from `rtab_component`, the virtual nature of the call `object->read(f)` will invoke the appropriate version of `read()`.

`object_list` is a linked list of component objects derived from `rtab_composite`. `object_list` is a concrete example of the *composite participant.* It holds all of the objects placed in the scene, some of which may be composite objects or other `object_lists`, themselves. `rtab_component`, the *component participant,* allows `object_list::intersect()`, or any other code, to treat all objects exactly the same way: In the call `cur->object->intersect(r)`, `object` could point to primitive or composite objects.

```
intersection
object_list::intersect(ray r) {

   while (cur) {
    tmp_i = cur->object->intersect(r);
    if ((tmp_i.t > 0) &&
         ((tmp_i.t < min_i.t) || (min_i.t == 0))) {
        min_i.t = tmp_i.t;
        min_i.object_handle = tmp_i.object_handle;
    }
    cur = cur->next;
 }
     return min_i;
}
```

Finally, we take a look at an algorithm for ray tracing the `spheres` and other objects in the `object_list`. This last example is the *client participant* of the *Composite* pattern. Again, note we have inserted many types of objects into the scene, but in the client code no statement takes different action based on what `object` points to.

```
color
scene::trace(ray r, int depth) {

  intersection i = objects->intersect(r);
  point intersection_point = r.t_point(i.t);
  rtab_component* object = i.object_handle;

  if (t > 0) {
       color dif_ref_light(0,0,0);
       color spec_ref_light(0,0,0);
       color spec_ref_objects(0,0,0);

       vector V = -r.direction();
       vector N = object->normal(intersection_point);

       if (object->contains(intersection_point))
         N = -N;

       vector I = -V;
       vector R = I - 2 * (N * I) * N;
       ray r2(intersection_point, R);

       if (depth > 0)
         spec_ref_objects = trace(r2, depth - 1);

         for (int light_index = 0; light_index < num_lights; ligth_index++) {
           color_point light = lights[light_index];
           vector L = (light - intersection_point);
           intersection i2 = objects->intersect(ray(intersection_point, L));
           if ((i2.t == 0) || (i2.t >= L.length())) {
               L = L.unit();
               vector H = ((L + V) / (L + V).length()).unit();
               float spec_factor = pow((N * H), object->n());
               dif_ref_light += (N * L) * light.clr();
               spec_ref_light += spec_factor * light.clr();
           }
         }
         return  ambient_light * object->clr() * object->kdr() +
                 dif_ref_light    * object->kdr() +
                 spec_ref_light   * object->ksr() +
                 spec_ref_objects * object->ksr();
        }
        else
          return color(0, 0, 0);
}
```

Also notice the code relies heavily on the geometric classes, the support classes and their operators. If the statements were written without the classes and their operators the components would have to be dealt with separately, making the code cumbersome and far more difficult to read. For example, if the classes `vector` and `color` were defined without their overloaded operators, the relatively succinct line:

```
dif_ref_light += (N * L) * light.clr();
```

would become nearly incomprehensible:

```
dif_ref_light = dif_ref_light.add(light.clr().multiply(N.dot_product(L)));
```

Virtual Frame Buffer Examples

This example helps explain creating a new STORAGE_CLASS to change the way vfb_pixel's are stored in memory. Our new STORAGE_CLASS will represent pixels in memory as simply a gray value. When a new STORAGE_CLASS is to be used we first update the STORAGE_CLASS typedef:

```
typedef class gray_pixel_storage STORAGE_CLASS;
```

Then, the new class must inherit pixel_interface and be defined. gray_pixel_storage represents a pixel with one float, value. The definition is rather simple:

```
class gray_pixel_storage: public pixel_interface {
private:
  float value;

protected:
  float_storage() : pixel_interface() {}
  int store_pixel(float az, float r, float g, float b, int sub_pixel)
       {value = ((r + g + b) / 3);  return 1;}

public:
  float az() const {return -INFINITY;}
  float r()  const {return value;}
  float g()  const {return value;}
  float b()  const {return value;}

  void read(ifstream& f)  { f >> value;}
  void write(ofstream& f) { f << value << " ";}
};
```

If the client code using vfb_pixel treats pixels as black/white or gray the above class will completely handle the code. However, if the client expects rgb pixels some information will be lost. The client code might include the following line

```
vfb_pixel rgb_pix(0.4, 0.1, 0.1);
```

gray_pixel_storage::store_pixel() would translate the three floats to a single float, value. rgb_pix.r() would return 0.2 instead of the expected 0.4. Although some of the information would be lost, the client code would still work, because vfb_pixel only has one interface regardless of how the pixels are being represented in memory.

Discussion

The examples included above give a glimpse at how my code can be used to facilitate ray tracing. Far more complex examples are included in the source code and give an idea of the flexibility built into the software. The first examples show how easily one can implement simple objects. To some extent the complexity of an object's implementation depends the object, but because of the transparency between object types, complex objects can be built up in simple steps. The fractal spheres that appear in the included images are an example of a complex object developed in simple steps. `bounding_sphere` inherits `rtab_composite` and `traceable_sphere` to complete a concrete container class. Then, `frac_sphere` inherits `bounding_sphere`. To complete `frac_sphere`'s definition only one new variable is declared, and the `read() and write()` functions are redefined. Given the steps that have already been taken to implement `bounding_sphere`, frac_sphere makes only the next simple step.

The support classes have been shown to compress cumbersome functionality into manageable notation. Along with the object type transparency, this notation is essential to making ray tracing algorithms comprehensible. The example above shows how difficult writing the tedious ray tracing calculations would be without the support classes. Similarly, the support classes make ray traceable objects' definitions more clear. The code for members like `read()` and `write()` would become cluttered without overloaded operators like `>>` and `<<.` Members like `intersect()` often rely on notation made possible by the overloaded arithmetic operators.

The virtual frame buffer example continues the emphasis on ease of use and flexibility. `gray_pixel_storage` is a very simple example, but it does show an important concept: a vfb_pixel may not fully represent a particular concept of a pixel but it will still interface with the client code correctly. This becomes important when more complex pixels, like antialiased pixels, are implemented. If antialiasing is achieved by dividing a pixels into sub-pixels the amount of memory needed to store one pixel increases by a factor equal to the number of sub-pixels. If a system is low on memory, recording all of the sub-pixels may be too expensive. In such an instance switching to a non antailiased `STORAGE_CLASS` conveniently does not require any changes to the code using the pixels.

This document was intended to show how my code meets the original goals of my thesis project: to facilitate the development of ray tracing software by providing an object oriented structure for the software and a number of support classes. The examples above are not meant to be a complete guide for using the software in new ray tracing or graphical pursuits. Reading the code in `rt_base_objects.hh` and `rt_base_objects.cc` along with the appendix will provide a more complete understanding of `rtab_component`, `rtab_composite` and `intersection`. Along with the appendix, the code in `vfb.hh` and `vfb.cc` shows all of the `vfb's` functionality, while `pixel.hh` and `pixel.cc` will help in creating new pixel types. Finally, `scene.hh`, `scene.cc` and `render.cc` (in `src/render`) may be studied as a full example of how all of the parts fit together.

Images, Demo's and Directories

The photographs included in this document were all generated using the ray tracing software in this package. They are all 875 x 700 pixels, traced with 4 rays per pixel. The color printouts are graphical displays of the information the intersection class variables provide. Each pixel of these images is colored according to how many intersection objects were created for the pixel to be ray traced. Along one edge you will notice a color scale indicating the relative number of intersections. Black is at the low end of the scale and white indicates the highest number of intersection tests.

To start the virtual frame buffer use the Tk script, `vfb,` and provide the maximum virtual coordinates. For example, to open a 301 by 301 pixel (remember (`0, 0`) is included in the `vfb`) `vfb`, type:

```
vfb 300 300
```

The Tk script provided only implements a minimal user interface, but it does take advantage of most of the `vfb's` widget commands. To load, save or ray trace (render) an image click on the load/save button. Then, enter a filename and click on the appropriate button. Do not include the `.vfb` or `.ray_trace` extension. The script adds it for you. All of the files in the image directory (with the exception of the files in `rt_data`) may also be viewed using `xv`.

If you want to create your own scene to ray trace with the programs provided, enter data for the objects in it into a `.ray_trace` file. Look at the files in the `rt_data` directory and at the `read()` function in `my_object` for the files format. The program `rt` will render the file and save the image to disk without opening a widget. `rt` may be started with a command like:

```
rt my_scene.ray_trace
```

In the `scr/tuples` directory there is a demo program that reads in a few `tuples` and `tupels_based` variables and then applies a number of operators to them.

On the next page there is a complete list of all the directories and files included with this project. **Directories** are in **bold** face type.

```
usr/desolation/morals
        documentation
                Thesis
                Users_Manual
        Makefile
        objs
        bin
                vfbsh
                rt
                vfb    (the Tk script)
        src
                ray_trace
                        Makefile
                        rt_base_objects.cc
                        my_objects.cc
                        scene.cc
                        inlude -> ../../include
                render
                        Makefile
                        time.c
                        performance.cc
                        render.cc
                        include -> ../../include
                tuples
                        Makefile
                        tuples.cc
                        tuples_based.cc
                        tuples_demo.cc
                        include -> ../../include
                vfb
                        Makefile
                        pixel.cc
                        vfb.cc
                        vfbsh.cc
                        render.cc
                        include -> ../../include
        include
                tuples.hh
                tuples_based.hh
                pixel.hh
                vfb.hh
                rt_base_objects.hh
                my_objects.hh
                scene.hh
                time.h
                performance.hh
                render.hh
        images
                vfb
                intersection_vfb
                TIFF
                rt_data
```

User's Manual

Support and Ray Tracing Classes

`tuples` and `tuples_based`

The `color`, `vector`, `point`, `color_point`, and `matrix` classes are defined by a number of overloaded operators and a few other member functions. In pratical use the classes successfully implement homogeneous coordinates, but the implementation may not completely satisfy the theory behind homogenous coordinates.

The basic tuples classes may all be constructed with only three values passed. For each constructor a default value is given for the fourth argument:

```
color (r, g, b, a = 0)
vector (x, y, z, w = 0)
point (x, y, z, w = 1)
```

The operators `<<` and `>>` have been overloaded for the `tuples` and `tuples_based` classes. The operators for `vector` and `color` are defined similar to `point` (below). `color_point` first reads calls `>>` or `<<` for `color`, then for `point`.

```
friend istream&  operator>>(istream& s, point& p);
friend ostream&  operator<<(ostream& s, point p);
```

First `>>` reads a string of 0 or more alphabetical characters ('a' .. 'z', 'A' .. 'Z') from `s`, then four `float`s. `s` may ONLY have white space immediately before the string and each of the four `float`s. The string allows an optional label to precede the data defining the variable. It is ignored. The `<<` operator writes the `point` to `s` in the format below.

```
point (x, y, z, w)
```

The lists on the following pages define the remainder of the `color`, `vector` and `point` classes. They specifie the operand types and return types for all of the operations. A few associated functions and constants are also listed. All operation symbols have their normal meaning except where noted.

i => int
d => double
c => color
v => vector
p => point
cp => color_point
m => matrix
r => ray

class color

(tuples)

```
c + c = c
c - c = c
c * d = c
d * c = c
c / d = c
c * c = c   NOT a dot product:  c(r1 * r2, g1 * g2, b1 * b2, a1 * a2)

c += c
c -= c
c *= d
c /= d

(c == c) = i
(c != c) = i

Constant colors
BLACK(0, 0, 0, 0)
WHITE(1, 1, 1, 0)
NaN_color(NaN, NaN, NaN, NaN)
```

class vector

(tuples)

```
v + v = v
v - v = v
v * d = v
d * v  = v
v / d = v
v * v = d     Dot product
v % v = v     Cross product

v += v
v -= v
v *= d
v /= d
v %= v Cross product

v.unit() = v       Returns a vector of length 1 parallel to v
v.length() = d           Returns the length of v

(v == v) = i
(v != v) = i
```

Constant vectors

```
NaN_vector (NaN, NaN, NaN, NaN)
VECTOR_DIRECTION[NEG_Z] (0, 0, -1)
VECTOR_DIRECTION[POS_Z] (0, 0, 1)
VECTOR_DIRECTION[NEG_Y] (0, -1, 0)
VECTOR_DIRECTION[POS_Y] (0, 1, 0)
VECTOR_DIRECTION[NEG_X] (-1, 0, 0)
VECTOR_DIRECTION[POS_X] (1 0, 0)
VECTOR_DIRECTION[NO_DIRECTION] (0, 0, 0)
```

`class point`

`(tuples)`

`p + v = p`	
`p - p = v`	
`p - v = p`	
`p * d = p`	Protected scaler multiplication*
`d * p = p`	Protected scaler multiplication*
`p / d = p`	Protected scaler division*
`p * v = d`	Protected dot product*
`v * p = d`	Protected dot product*
`p * p = d`	Protected dot product*
`p % p = p`	Protected cross product*
`v % p = p`	Protected cross product*

`p += v`	
`p -= v`	
`p *= d`	Protected scaler multiplication*
`p /= d`	Protected scaler division*
`p %= v`	Protected cross product*

`(p == p) = i` NOTE: `point` relational operations compare the HOMOGENIZED
`(p != p) = i` coordinates of their operands.

`p.homogenized() = p` `p.homogenized() = point(x/w, y/w, z/w, 1)`

Constant `points`

`NaN_point(NaN, NaN, NaN, NaN)`

*The multiplication, division, dot product and cross product operations for `points` implicitly convert their `point` operands to `vectors` as if `point(0, 0, 0, 1)` had been subtracted from the `point` operand. The multiplication, division and cross product operators convert the `vector` back to a `point` by adding the `vector` operand returned to the `point(0, 0, 0, 1)`. All of these operations are protected and only included so `class matrix` may treat points as column vectors.

class color_point

(tuples_based)

The constructor for `color_points` has been heavily overloaded so that almost any combination of `color`, `point` and `float` will result in an appropriate `color_point`. For the full list see the `tuples_based.hh` file.

All of the operation that apply to `point` are defined in exactly the same way (including if they are `protected`) for `color_point`, except that they have a `color` associated with them. With the exception of the relational operators, all `color_point` operations only effect the `point`, not the `color`. The relational operations, `==` and `!=`, compare both the `colors` and `points` of their operands.

class matrix

(tuples_based)

Matrices may be constructed from 16 `doubles`, 4 row `vector`s or as 4 column `point`s. Once a `matrix` is constructed it may also be accessed as `vector`s, `points` or `float`s:

```
m.v1() = vector (m.m11, m.m21, m.m31, m.m41)
m.p1() = point  (m.m11, m.m12, m.m13, m.m14)
```

These operations are defined for `class matrix`:

`m * m = m`	
`m * v = v`	
`p * m = p`	
`r * m = r`	Multiplies both the `origin` and `direction` by `matrix m`.
`m * d = m`	
`d * m = m`	
`m + m = m`	
`m - m = m`	
`(m == m) = i`	Relational operators are applied to the row `vector`s NOT the
`(m != m) = i`	column `points`, so that the `point`s are NOT homogenized.
`cp  * m = cp`	

`m.inverse() = m`

Returns `matrix Im` such that `m * Im = I`. This function uses row reduction, and will not work if any of the column `points` = `point (0, 0, 0, 0)` at the time their column is being reduced. This has not been found to cause problems.

These functions may be helpful for creating transformation matrices.

`matrix translate_matrix(vector v)`

Returns `matrix m` such that:

```
point * m = point + v
```

and

```
m * vector = vector + v
```

`matrix shear_matrix(double shx, double shy)`

Returns a `matrix m` suitable for shearing the x coordinate of anything that is multiplied by `m` by `shx`, and y coordinate by `shy`.

rt_base_objects

class ray

(rt_base_objects)

Given an point origin and a vector direction a ray is defined as all points p such that:

p = origin + t * direction, t > 0

This idea is implemented in the `class ray` with members and operators:

```
ray(point origin, vector direction)
```

The `direction vector` is always stored as a unit `vector`.

```
point origin()
vector direction()
point t_point(double t)
```

Returns the `origin + t * direction`

```
d * r = p
r * d = p
```

Equivalent to `p = t_point(r)`

```
r * m = r
```

Allows rays to be transformed. Both the `origin` and `direction` are multiplied by the `matrix m`.

class intersection

(rt_base_objects)

Any class that is derived from the abstract base class `rtab_component` will eventually have to define the `intersect()` member. `intersect()` has been declared to return an `intersection` class object. There are a few rules that should be followed when returning `intersection`s.

`float t`

Should always equal 0 if no primitive object was intersected. Otherwise it should be assigned so that given a `ray`, with `direction d` and `origin o`, the intersection `point p` is given by the equation: `p = o + t * d`.

`rab_component* object_handle`

This should only equal `NULL` if no primitive objected was intersected AND no composite object was intersected. Otherwise it should point at the deepest node in the component tree that the ray hits.

A few class variables and functions are provided with `intersection` to indicate the number of intersection tests performed. (Since it is usually necessary to use more than one `intersection` in intersect functions, these variables will not correspond to the exact number of intersection tests performed. However, they are adequate for examining the relative number of intersection tests.) These are only declared if the `_INTRSCT_PRFM` flag is set. To set the flag add `-D _INTRSCT_PRFM` to the `CPPFLAGS` in your `Makefile`. The class variables are:

`int Num_Intersections`

If flag is set this is incremented by the constructor.

`int Num_NULL_Intersections`

If flag is set this is incremented by the destructor only if `object_handle = NULL.`

`int Num_t0_Intersections`

If flag is set this is incremented by the destructor if `t = 0.`

Their values can be accessed with the following class functions:

`int count()`

`int NULL_count()`

`int t0_count()`

`void clear_count()`

Clears all of the class variables. If the flags are set be sure to call this function often. Remember that during ray tracing hundreds of thousands of intersection tests are carried out. If you don't clear the counters they will probably cause the program to crash.

NOTE: IT IS VERY IMPORTANT TO CALL `clear_count()` IF THE FLAG IS SET (See above.)

`class rtab_component`

`(rt_base_objects)`

This is the abstract base class for all ray traceable objects, both primitive and composite. Some of the member functions have been given default behavior. The following is a list of all the member functions and how they should be defined in the concrete primitive and concrete composite classes.

`intersection intersect(ray r)`

Do not declare extraneous `intersection` objects. Doing so will throw off the intersection performance tests. (See `intersection` class above) The `origin` point of a ray should not be considered part of the ray, so if a ray intersects an object at only its origin `t = 0` and there is not an intersection. `intersect` should return `t = 0` and `object_handle = NULL` if `r` does not intersect the primitive. Rays only extend forward. Follow the convention that `t <= 0` indicates the primitive exists in a direction opposite that of the ray's direction relative to the ray's origin; therefore, there is no intersection. Always use `zero_t()` instead of `0` when comparing `t` to `0`. After many calculations a number that would be `0` if we had infinite precision will often be a very little bit more or less than `0`. `zero_t()` is meant to solve this problem (see `zero_t()` below).

Primitive: Primitive objects must define the `intersect()` member.

Composite: Composite objects will usually call `intersect()` for some or all of their children. They should always return the intersection with the smallest positive `t`. If `r` intersects the composite but none of its children return `t = 0` and set `object_handle =` the composite. This allows the `intersection` class to keep track of the number of rays that hit composites but not a primitive.

`vector normal(point p)`

Primitive: This returns the vector perpendicular to the surface of the object at `point p`. The direction of a normal usually depends on which side of the surface is the front; however, the front of a surface is relative. If we are inside a sphere the front faces inside, but if we are outside the sphere the front of the surface faces outside. By convention `normal()` should assume we are outside of the object and return a `vector` pointing towards the outside of an object. We can then safely assume that negating the value returned by `normal()` will give us the correct vector if we are inside the object. The `contains()` member is intended determine if we are inside an object. (See `contains()` below.)

Composite: Composite objects should not have to define `normal()`.

`point min_point()`

This should return the left, bottom, back corner of a bounding box just large enough to enclose the object. This point can be found by taking the x coordinate of the part of your object with the smallest x value, the y coordinate of the part of the object with the smallest y, and the z coordinate of the part of your object that has the least z value.

`point max_point()`

This should return the right, top, front corner of a bounding box just big enough to contain the object. Finding this point can be done in a manner similar to the one described for `min_point()`.

`min_point()` and `max_point()` members aid bounding volumes in determining the size and location of component objects.

Primitive: Primitive objects should, but do not have to define these functions.

Composite: Composite objects should, but do not have to define these functions.

`bool contains(point p)`

Primitive: This member is used to test if a ray originates inside an object. It should return true if `p` lies inside the object. For objects that do not have an interior, assume that the point always lies outside of the object and return false.

Composite: Composite objects should define this member if they have space between parts of their surface. For example a box made up of 6 polygons contains space. Container objects may want to define this member to test if they contain an objects `min_point()` or `max_point()`.

```
istream& read(istream& s)
ostream& write(ostream& s)
```

These functions, respectivly, should read data from `s` to define an object, and write the data that describes the object to `s`. Write may include a label telling what the object is. They should always return `s`. Since these functions are virtual and the `>>` and `<<` operators have been overloaded for `rtab_composite*`, `>>` and `<<` will work for any concrete classes that implement `read()` and `write()`. Also see `get_type()` at the end of this section.

Primitive: Primitive objects should redefine these members.

Composite: Composite objects should redefine these members.

`int insert(rtab_component object)`

This defines the interface for inserting components into a composite object.

Primitive: Primitives should not change the default behavior unless they redefine it to omit the error message and just return `0`.

Composite: All composite objects should redefine this member. For composite objects that insert does not make sense, like boxes made of polygons, new error behavior should be defined and the function should still return `0`.

`void delete_all_children()`

Primitive: Since primitive objects have no children the default behavior should not be changed unless it is redefine to omit the error message and just return `0`;

Composite: Default behavior is given for this member in the `rtab_composite` class. (See `rtab_composite` below.)

`zero_t()`

After many calculation a number that would be 0 if we had infinite precision may be slightly more or less than 0. This little bit of error can cause huge problems when we are testing values of `t` against `0` in the `intersect()` member and in the ray tracing algorithm. It is much safer to test against a value just a bit larger than `0`. If you intend to view the inside of an object, that particular object's `zero_t` should be negative and its class's `intersect()` should test against the absolute value of `zert_t()`.

Primitive: The default value will work in most cases, but it may be necessary to redefine it.

Composite: The default value will work in most cases, but it may be necessary to redefine it.

Surface physics members

The remainder of the members of the component class make up the surface physics model. While the user is free to chose whatever physics model best fits the ray tracing algorithm used the members have been named for the Hall Shading Model[7] If the functions and variables defined for the surface physics are not appropriate for the shading model being used, new variables may be added to the class. This is the only part of the class `rtab_component()` that should ever be changed.

Primitive: Default behavior is intended to mimic the effect of not including the member in the surface physics model. Redefine all members you want to use in your algorithm.

Composite: Composite objects should not have to define the surface physics model.

`rtab_component* get_type(istream& s)`

This function is not part of the `rtab_component` class, but whenever you create a new concrete primative or composite object you should add it to the `get_type()` function. `get_type()` reads the next string in `s` and compares it against a number of object names. Then it creates an object of the corresponding type using `new` and returns a pointer to it. When you have a new class of object register it in `get_type()` by adding line to the `if-then-else` statement like the following:

```
else if (strcmp(name, "YOUROBJ") == 0) return new your_obj;
```

`name` cannot be more than 10 characters long and should be all upper case letters ('A' .. 'Z'). Currently this function is in the file `my_objects.cc`. When you create new object classes it should be moved to your `.cc` file.

class rtab_composite

(rt_base_objects)

`rtab_composite` is a abstract class that inherits from `rtab_component`. It defines default behavior for freeing space allocated for it sub-objects, the `child` array. See `rtab_component` above for more details of how to implement concrete composite objects. The following are member and member functions that differ in definition from the `rtab_component` class.

`int num_children`

This variable must be maintained so that passes through the `child` array are possible.

`rtab_component** child`

This declares an array of pointers to `rtab_component`s. It is up to the concrete composite classes to allocate memory for their `child` array. This should be done with `new` so that the default destructor and `delete_all_children()` members will work correctly.

`rtab_composite()`

The constructor should be redefined in concrete classes to set the `num_children` and allocate space for the array of pointers, `child`.

`delete_all_children`()

By default `delete_all_children()` calls `delete_all_children()` on all objects in the `child` array then calls `delete` on each pointer in the array. Calling `delete` on a memory that was not allocated by `new` causes problems, so take care to use `new` when inserting new objects and allocating space for the `child` array.

`~rtab_composite()`

By default this calls `delete_all_children()`. If you do not allocate memory for all of your objects using `new` you must redefine this or the program will most likely crash.

The Virtual Frame Buffer

`pixel`

`class pixel_interface`

`(pixel)`

`pixel_interface` declares the interface for all pixel types that can be used in the virtual frame buffer. It is an abstract base class. It does not define most of the members that it declares. Storage classes (described below) are responsible for implementing the members of `pixel_interface`. Below is a brief description of what each member declared should do. For a full description see the section on `STORAGE_CLASS`.

`pixel_interface()`

This member is protected so that only classes that inherit `pixel_interface` may create `pixel_interface` objects.

`int store_pixel(float az, float r, float g, float b, int sub_pixel)`

This function converts all pixel types to a common type of storage pixel. It must be implemented by `STORAGE_CLASS`.

`float az();`

Converts back from the storage pixel to `float` for the alpha or z value. Pixels cannot be both alpha and z-buffered. It should be implemented by `STORAGE_CLASS`. By convention this should return `-INFINITY` if it is undefined by the storage pixel.

```
float r()
float g()
float b()
```

Functions for converting the storage pixels back to `float` values for red, green and blue, respectively. They should be implemented by `STORAGE_CLASS`.

`operator== (pixel p2)`

By default returns `1` if the values returned for `az()`, `r()`, `g()` and `b()` are equal.

`operator!= (pixel p2)`

By default returns `1` if the values returned for `az()`, `r()`, `g()` and `b()` are not equal.

`void clear()`

By default clears to `color(0, 0, 0)` and `az` value `-INFINITY` by calling `store_pixel`.

STORAGE_CLASS

(pixel)

The `STORAGE_CLASS` `typedef` allows a user to switch between methods for storing pixels by changing only one line of code (once a storage class has been defined). A storage class should inherit from `pixel_interface` and is intended to be inherited by `vfb_pixel`. The constructor should always be `protected` so objects can only be created by `vfb_pixel`. The best way to describe a storage class is by example. I will work through an example implementation of a storage class that stores pixels as a `float` and 3 `char`s. Below is a general description of how each of the members declared, but not defined, in `pixel_interface` should be defined. Before using the new storage type `STORAGE_TYPE`, `MAX_PIX` and `MAGIC_NUMBER` must be set at the begining of `pixel.hh`. `STORAGE_TYPE` is the class name. `MAX_PIX` is the maximum value for one component of a pixel's color. Depending on how pixels are saved, `MAGIC_NUMBER` along with `MAX_PIX` enable `vfb`s to be `ppm` or `pbm` compatible. (These may not apply for all pixel types.) See the man pages on `ppm` for details on setting `MAGIC_NUMBER`.

```
class float_3char_storage {
  float alpha_z;
  char red;
  char green;
  char blue;
```

Storage classes must declare and define the variables they use to store the pixels. These specific variables are just the ones I choose for the example. Notice the variables do not allow for antialiasing.

```
protected :
  float_3char_storage();
```

The constructor is protected so only `vfb_pixel` can call it.

```
int store_pixel(float az, float r, float g, float b, int sub_pixel);
```

This member must convert the 4 `floats` `r`, `g`, `b` and `az`, and a `int`, `sub_pixel`, to the variables chosen to store the pixel. I have held the convention that the red green and blue values should be between 0 and 1. A check for this may be appropriate here. Arguments passed to `store_pixel` that are not used may simply be ignored. For example `float_3char_storage` does not use `sub_pixel` because it does not implement antialiasing. Code for my example would look something like:

```
int store_pixel(float az, float r, float g, float b, int sub_pixel) {

        if (r < 0) r = 0;
        if (g < 0) g = 0;
        if (b < 0) b = 0;
        if (r > 1) r = 1;
        if (g > 1) g = 1;
        if (b > 1) b = 1;

        alpha_z = az;

        // each part of the pixel is stored as a fraction of MAX_PIX
        red =  (char) (r * MAX_PIX);
        green = (char) (g * MAX_PIX);
        blue =  (char) (b * MAX_PIX);
}
```

In a case where `r`, `g` or `b` is not between 0 and 1 we still store the pixel. These values may not fit in the range, but we can just round them to 0 or 1. `MAX_PIX` is a constant `float` value set in `pixel.hh`.

```
float az()   {return alpha_z);}
```

If `az` was converted before storing it must be changed back to a `float`. Antialaising presents a problem here since we can only return one value for all of the sub-pixels. Should we return the average of the sub-pixels, the greatest, or the least? (In my code I choose to return the average.) For the example, this is not a problem since we do not have sub-pixels to.

```
float r();
float g();
float b();
```

Theses function must convert the stored type back to a `float` between 0 and 1. If we are implementing antialiased pixels these return the average of the sub-pixels. Code for `r()` in my example would look something like:

```
        return float(red / MAX_PIX);
```

```
operator=(const float_3char_storage& p2);
```

The `operator=` should simply copy the right values to the left pixel, unless we are implementing z-buffering. In that case the new values should only be assigned if the new pixel's `az` value is greater than the current `az`. Antialiased pixel types should compare the individual sub-pixels `az` values. Here is the code for my example:

```
        if (p2.alpha_z >= alpha_z) {
              red = p2.red;
              green = p2.green;
              blue = p2.blue;
        }
```

Notice that we use the actual variables not calls to `az()`, `r()`, `g()` or `b()`. This eliminates any conversions between the `float` and `char`.

```
void read(ifstream& f);
void write(ofstream& f);
```

These functions require that stream `f` be open at the correct location to read or write the particular pixel. Since the values have already been converted to `char` we can just write them to a file as binary data:

```
        f.write(&red, 1);
```

or read them into the variables:

```
        f.read(&red, 1);
```

Writing pixels to disk as `char` will make the files `ppm` compatible if `MAGIC_NUMBER` has appropriately been set to "`P6`".

class vfb_pixel

(pixel)

Class `vfb_pixel` inherits from whichever type of storage pixel `STORAGE_CLASS` is set to in the

```
    typedef class ???? STORAGE_CLASS
```

line of `pixel.hh`. This class is used to construct a storage pixel given a set of arguments. The types and number of arguments given allow the caller to choose how he wants to think of pixels. For example if he wants to think of pixels as either black or white a call to `vfb_pixel (int c)` is appropriate. `STORAGE_CLASS` restricts which types of pixels are actually created. Calls to construct pixel of a type more complex than `STORAGE_CLASS` can handle are legal, but the extra information will be lost. Take the example of a program that uses antialiased pixels, but is taking up too much memory. The `STORAGE_CLASS` can be changed so that antialiasing is not implemented. The original code for the program will still work, but the pixels will not be antialiased. Nothing in this class should be changed. NOTE: if you want to force a pixel to be non-antialiased either omit the `x, y` arguments to the constructor or make sure that they are `int`s. (See the second to last constructor below.)

```
vfb_pixel(int c)
```

Construct a pixel that is either black (0) or white (1).

`vfb_pixel(float c)`

Construct a pixel that is a shade of gray between black (0) and white (1).

`vfb_pixel(float r, float g, float b)`

Construct a pixel that has 3 color components, red `(r)` green (`g`) , and blue `(b)`, between 0 and 1.

`vfb_pixel(float az, float r, float g, float b)`

Construct a pixel that has 3 color components, red (`r`), green (`g`) and blue (`b`), between 0 and 1 and either a alpha value or a z value (`az`). There is no condition for restricting `az` other than is should be a `float`.

`vfb_pixel(int x, int y, float zx, float r, float g, float b)`

Construct a non-antialiased pixel for a buffer of antialiased pixels by setting all sub-pixels to the given red (`r`), green (`g`), blue (`b`) and `az`.

`vfb_pixel(float x, float y, float az, float r, float g, float b)`

Construct an antialiased pixel by setting the appropriate sub-pixel (based on the `x` and `y` arguments) to the given red (`r`), green (`g`), blue (`b`) and `az` values.

`vfb_pixel(point p, color c)`

Construct an antialiased pixel by setting the appropriate sub-pixel (based on the `p.x()` and `p.y()`) to the given red (`c.r()`), green (`c.g()`), blue (`c.b()`) and `p.z()` values.

`operator=(const vfb_pixel& p2)`

Calls the appropriate version of the overloaded `operator`().

vfb

class vfb

(vfb)

The `vfb` class implements virtual frame buffer as Tk widget. It has a number of operations that can be called from a Tk script to modify and view the buffer. `vfb` objects may also be use in C++ code. The only operations that should be called directly from C++ code on the frame buffer are listed below. Note: type `vfb_int` is the same as type `int`. I used a `typedef` so that all arguments and return values that should be in virtual pixels, not actual pixels, are `vfb_int`s.

`vfb(vfb_int maxx, vfb_int maxy)`

> The constructor allocates memory for the array of `vfb_pixel`s that make up the `vfb`. The size of the buffer is specified in the call to the constructor by `maxx` and `maxy`. The coordinate system has the (0, 0) in the lower left corner and (`maxx`, `maxy` in the upper right corner.

`int put_pixel(vfb_int x, vfb_int y, vfb_pixel pix)`

> This puts a pix in the frame buffer. The location is specified in virtual coordinates. The overloaded `operator=(vfb_pixel)` is used to make the assignment. See the `vfb_pixel` class for details.

`vfb_pixel get_pixel(vfb_int x, vfb_int y)`

> Returns the `vfb_pixel` at a given location in the frame buffer. The location is specified in virtual coordinates.

`clear()`

> Calls `vfb_pixel::clear()` on each pixel in the frame buffer.

`load(char* filename, vfb_int x1, vfb_int y1)`

> Loads a `vfb` from a file. The saved `vfb` is transposed so that its `(0, 0)` will be written to (x1, y1). First, the file is opened and the header information, including the virtual size of the saved buffer, is read in. Then `load()` uses `vfb_pixel::read()` to read each pixel from the file.

KNOW PROBLEM: Reading pixels one at a time causes this function to be very slow. The load function should be updated to read a large chuck of the file into a temporary stream. That stream could be interpreted with calls to `vfb_pixel::read()`.

`save(char* filename)`

Writes the `vfb` to a file. Some header information is written to the file before the vfb. This information is written in the following format:

```
MAGIC_NUMBER
virtual_width virtual_height
MAX_PIX
```

`MAGIC_NUMBER` and `MAX_PIX` allow the `vfb` to be interpreted as a `pbm` or `ppm` file. They are defined in `pixel.hh`. `MAGIC_NUMBER` indicates how the pixels are stored and `MAX_PIX` indicates the maximum intensity of a pixels component. This information does not guarantee that the file will be compatible. `vfb_pixel` must also read and write individual pixels in the corresponding format. (See `man` pages on `ppm` and `pbm`.)

NOTE: `virtual_width` and `virtual_height` indicate the size of the frame buffer. Since `vfb`s include `(0, 0)` these numbers are respectively each one greater than `max_virt_x` and `max_virt_y`.

KNOW PROBLEM: This function should also avoid accessing the file for pixels one at a time (see `load()` above.)

`render_to_vfb(char* filename, vfb& v)`

This function is not part of `class vfb` nor is it a friend function. It is only declared with `class vfb`, not defined. Writing C++ code that uses the `vfb` member functions is easy. Using a Tk script and the vfb widget to create a user interface is also easy. However, linking the vfb object in C++ code with the particular `vfb` object used in the script is difficult. This function declaration provides the link. It is intended to let users invoke C++ code on the particular `vfb` associated with the widget. This function is called by the `render` widget command described below. It is up to the users of this software to define the function. `render_to_vfb()` is expected to return 1 if the C++ code completes normally or 0 otherwise. Selecting a rendering algorithm based on `filename`, then passing `v` and `filename` to the function that implements the algorithm is the best way to use `render_to_vfb()`. See `src/vfb/render.cc` for a simple example of this with stubs for the rendering functions. A full ray tracing algorithm is implemented in `src/render/render.cc`

There are a few friend functions associates with the `vfb` class. You should not need to call these. They only serve as an interface for Tk.

`vfb` Widget Commands

In order to make use of the `vfb` widget one must know a little Tcl and Tk. If you are not familiar with the Tk toolkit consult Tcl and the Tk Toolkit[8]. Even if you have never heard of Tcl just a few hours of reading will give you enough background to begin using the `vfb` widget. The `vfbsh` application is the version of the `wish` shell that incorporates the `vfb` widget. The following commands may be used in a `vfbsh` script with the `vfb` widget. Arguments with the prefix `vfb_` expect virtual coordinates. Arguments with the prefix `tk_` expect values given in Tk's coordinate system.

`set_min_x vfb_x`

Sets the leftmost column of virtual pixels displayed by the widget to `vfb_x`. Use this to pan horizontally.

`set_min_y vfb_x`

Sets the lowest row of virtual pixels displayed by the widget to `vfb_y`. Use this to pan vertically.

`virtual_x tk_x`

Returns the virtual coordinate corresponding to a Tk's `tk_y` value. Use this to convert Tk's coordinate into virtual coordinates.

`virtual_y tk_y`

Returns the virtual coordinate corresponding to a Tk's `tk_y` value. Use this to convert Tk's coordinate into virtual coordinates.

`get_pixel vfb_x vfb_y`

Returns the red, green and blue `float` values for the pixel at (`vfb_x`, `vfb_y`)

`put_pixel vfb_x vfb_y z r g b`

Colors the pixel at (`vfb_x, vfb_y)` according to the floats `z r g` and `b`, by calling `put_pixel()`. Pixels may be z-buffered. See `put_pixel()` and `vfb_pixel` for more information.

`pixel_size size`

Sets the virtual pixel's size to `size`. `size` is a screen distance that Tk converts into an `int`[9].

`clear`

Calls `vfb_pixel::clear()` on each pixel in the vfb..

`load filename vfb_x1 vfb_y1`

Loads a `vfb` saved in `filename`. The `vfb` being loaded is transposed so that its (0,0) will correspond to `(vfb_x1, vfb_y1)`. These arguments are optional; the `vfb` is loaded to (0, 0) if they are omitted. If the `vfb` in the file is larger than the active `vfb`, only the part that fits will be loaded.

`save filename`

Saves a `vfb` to `filename`.

`render filename`

This command provides a simple interface for invoking C++ code on a vfb widget. The command calls the function `render_to_vfb()` and passes it a reference to the active vfb (`*this`) and the string filename (`argv[2]`):

```
    render_to_vfb(argv[2], *this)
```

`render_to_vfb()` is declared, but not fully defined in the provided software. It is intended to be used to select between rendering algorithms based on filename. For example if filename ends in `.ray_trace`, `render_to_vfb()` could pass the vfb to `ray_trace()` which would call `put_pixel()` on the vfb. (For more information see `render()` in the `vfb` member functions section.)

`vfb` Widget Configuration Options

The vfb widget has a few configuration options that determine how redraws are accomplished. The defaults are given in ().

`-grid_on int (1)`

A value of 0 indicates that the grid will never be drawn. 1 indicates that a grid will be drawn when `pixel_size` is large enough.

`-grid_on_at int (7)`

The grid is drawn when `pixel_size` is greater than this value.

`-disk_on int (1)`

A value of 0 indicates that the pixels will never be drawn as disks. 1 indicates that pixels will be drawn as disks when `pixel_size` is large enough.

`-disk_on_at int (7)`

Pixels are drawn as disks when `pixel_size` is greater than this value.

`-pixmap_drawing_on int (1)`

This determines how virtual pixels will be drawn. If its value is 1, a rectangle or disk will be drawn in a `pixel_size`, square pixmap. This pixmap will then be copied to draw virtual pixels. If it is 0, each virtual pixel will be drawn as a rectangle or disk. If `pixel_size` = 1 pixels are drawn as points and this has no effect.

`-pm_width int (2560)`

This value specifies the width (in actual pixels) of a large pixmap. When the widget is re displayed a section of this pixmap is copied to the widget's window. When `pixel_size` changes the entire pixmap must be redrawn. Larger pixmaps take longer to redraw. If the widget needs to display a section of the `vfb` that is not drawn on the pixmap, part of the pixmap must be redrawn. Smaller pixmaps require more partial redraws. The pixmap must be at least the size of the widget's window.

`-pm_height int (2048)`

Specifies the height (in actual pixels) of the pixmap described above.

C++ code and the `vfb` widget

Using `render_to_vfb()` is the easiest way to invoke C++ code on a `vfb` widget. If you have not read about the `render vfb` widget command and `render_to_vfb()` do so before reading this section. It is also possible to create a `vfb` widget from within a C++ program. A warning, this is far more complicated. What you will actually be doing is creating a new Tcl command. This command is written in C++. You will have to create a new Tcl interpreter by registering your command. The command will carry out the operation you need to perform on the `vfb` (or call functions to do this), then return `vfb::makewidget()`. Here is a more detailed description of what to do[10]:

1) Write the body of functions that use the `vfb`.

This is your Tcl command. It must have the following declaration:

```
int MyCmd(ClientData clientdata,
     Tcl_Interp* interp,
     int argc,
     char* argv[])
```

The first step in your command procedure will probably be to use some of the command line arguments. Any arguments that are specific to your command must be removed so that `argv` looks something like

```
"vfb" ".my_vfb" "\0"
```

The command can use the `vfb` itself or call other functions to use the `vfb`, but be sure to declare the `vfb` as `static`. If you do not it will go out of scope when you return from your Tcl command and will not exist for Tk to use. When you declare it you will also determine its size (see the constructor above). Your Tcl command must return with the following call to `make_widget`:

```
return (my_vfb.make_widget(clientdata, interp, argc, argv));
```

2) Register the command. For instructions on doing this consult chapter 31 and section 39.7 of Tcl and the Tk Toolkit.

3) Write a script that uses you new Tcl command to create a `vfb` widget

Known Problems

1) `vfb::load()` and `vfb::save()` are very slow, because pixels are read and written one at a time. See `vfb` member functions for more information about this problem.

2) Signaling NaN's cause floating point exceptions on DEC's Alphas. A function is used to generate a silent NaN, which defines a constant `float` and the constants `NaN_point`, `NaN_vector` and `NaN_color`. When operations are carried out on silent NaN's the operations must be returning signaling NaNs, because a floating point exception occurs.

3) Exiting the `vfb` widget causes a Segmentation Fault and core dump on the Alphas. This problem was only recently discovered and I have not been able to determine the cause.

4) The `Makefiles` that have been provided work correctly; however, they are probably not the best way to compile my code.

End Notes

[1] Gamma, Helm, Johnson, Vlissides, Design Patterns

[2] This discussion of ray tracing is partially based on Glassner's An Introduction to Ray Tracing, pp 2 - 16.

[3] Gamma, Helm, Johnson, Vlissides,Design Patterns, pg 166

[4] Gamma, Helm, Johnson, Vlissides,Design Patterns, pg 167

[5] Gamma, Helm, Johnson, Vlissides,Design Patterns, pg 166

[6] For a description of the Hall Shading Model see, Glassner's An Introduction to Ray Tracing, pg 152.

[7] Glassner, An in Introdution to Ray Tracing, pg 152.

[8] Ousterhout, Tcl and the Tk Toolkit

[9] Ousterhout Tcl and the Tk Toolkit, section 16.1.2.

[10] Ousterhout, Tcl and the TK Toolkit, Part IV .

Bibliography

Foley, James A., van Dam, Andries, Fiener, Steven K., Hughes, John F., Computer Graphics Principles and Practice, Addison and Wesley Publishing Company, Reading, MA, 1990.

Gamma, Erich, Helm, Richard, Johnson, Ralph, Vlissides, John, Design Patterns, Addision Wesley Publishing Company, Reading MA, 1995.

Glassner, Andrew S., Graphics Gems, Academic Press, New York, 1990.

Glassner, Andrew S., An Introduction to Ray Tracing, Academic Press, New York, 1989.

Hanrahan, Pat, "Lecture 9: 2D Projective Geometry", 1996

Hanrahan, Pat, "Lecture 10: 3D Projective Geometry", 1996

Ousterhout, John K., Tcl and The Tk Toolkit, Addison-Wesley Publishing Company, Reading MA, 1994.

Kaplan Michael R., “Space-Tracing, a Constant Time Ray Tracer”, SIGGRAPH85 Conference - “Tutorial on the Uses of Spatial Coherence in Ray-Tracing” Course Notes, San Francisco, 1985.

Neider, Jackie, Davis, Tom, Woo, Mason, Open GL Programming Guide, Addison-Wesley Publishing Company, Reading, MA, 1993 by Silcon Graphics, Inc.