Partial Evaluation Applied to Ray Tracing

Peter Holst Andersen

January 10, 1995

Contents

Abstract

The purpose of this paper is to collect the experiences made during a recent ray tracing project, and to document the results. The objective of the project is to redo Mogensen's experiment [Mogensen 86] with applying partial evaluation to ray tracing, but in another setting. We have implemented the standard recursive ray tracing algorithm [Foley 90], [Glassner 89], [Hall 89] in C and programmed it for speed. The focus in this project is on using partial evaluation to optimize an already efficient program. Also we want to investigate the problems that occur when partial evaluation is applied to production quality programs. The experiments show that this particular ray tracer can be optimized by partial evaluation to run nearly three times as fast.

Section 1 explains the motivation for this project and gives an introduction to key concepts and terminology. Section 2 describes the ray tracer used in this project. Section 3 describes the machines and compilers used. Section 4 presents a comparison of our implementation with Rayshade 4.0. Section 5 shows the results of specializing the ray tracer. Section 6 explains the binding-time improvements made. Section 7 concludes. The ray tracer program can be found in Appendix A.

This paper requires basic knowledge of partial evaluation corresponding to for example [Jones 93, Part II], but does require any previous knowlegde of ray tracing.

1   Introduction

Focus and Objective.   Our aim is to demonstrate that partial evaluation is useful as a general optimization tool. We do this by applying partial evaluation to a graphics application, more precisely a ray tracer. The ray tracer is a 2000 lines C program.

Partial evaluation is an automatic program specialization technique, which given partial knowledge `s` of a program's input, can

1. perform optimizations, that are too tedious to do by hand, and which ordinary compiler will not do (either because they are too complex, or because they depend on the input `s`, which is not available at compile-time), and

2. remove administrative overhead introduced by modular or general programming

Point 2 has been demonstrated in many applications of partial evaluation (e.g. [Mogensen 86], [Berlin 90b], [Berlin 90a], [Berlin 94], [Jorgensen 91], [Lisper 91], [Mossin 93], [Baier 94]), but little attention has been paid to point 1. In this project we will focus on point 1, by applying partial evaluation to a realistic application that is already programmed for speed. Point 2 will be of no concern.

Other goals for the project include to:

- investigate problems occurring when partial evaluation is applied to production quality programs

- develop a demonstration that can *visualize* the speedup

Partial Evaluation. A partial evaluator is a program which, when given a program and some of its input data (the *static* data), produces a so-called residual or specialized program. Running the residual program on the remaining input data (the *dynamic* data) yields the same results as running the original program on all of its input data. A partial evaluator unrolls loops, unfold function calls, pre-computes expressions that only depend on static data, and reduces expressions depending on dynamic data.

For this project we use an *off-line* partial evaluator for C (C-Mix [Andersen 92a], [Andersen 92b], [Andersen 93], [Andersen 94]). Off-line means that a pre-phase (the *binding-time analysis* or *BTA*) decides which statements to execute and which to suspend (i.e. generate code for; also called *residualize*). In most cases it is necessary to change the source program slightly in order get better results from specializing. Such transformations are called *binding-time improvements* [Jones 93, Chapter 12].

In this paper we use *speedup* as defined in [Jones 93, Chapter 6]: let $t_\mathtt{p}(\mathtt{s}, \mathtt{d})$ be the running time of the original program $\mathtt{p}$, and let $t_\mathtt{ps}(\mathtt{d})$ be the running time of the specialized program $\mathtt{p_s}$. The $speedup_\mathtt{s}(\mathtt{d})$ for that particular run is defined by the ratio between the two:

$$speedup_\mathtt{s}(\mathtt{d}) = \frac{t_\mathtt{p}(\mathtt{s}, \mathtt{d})}{t_\mathtt{ps}(\mathtt{d})}$$

Ray tracing. Ray tracing is a method to give good picture rendition of scene (i.e. a collection of 3-dimensional objects) on a screen.

The input to the ray tracer are a set of objects, a set of light sources, the viewer's position (the eye point) and a window. The window is thought of as divided into a regular grid, whose elements corresponds to pixels at the desired resolution. The colour of a pixel is then determined by the colour and amount of light that passes through the corresponding element in the grid from the scene to the eye point. The colour is found by tracing a ray[1] backwards from the eye point through the center of the pixel element into the scene. If the ray intersects the surface of an object, a number of rays originating at the intersection point might be spawned to determine the intensity at that point: a *shadow ray* for each light source in the scene, to determine if any objects are blocking the path between the light source and the intersection point. If the object is transparent, and if total internal reflection does not occur, then a *refraction ray* is sent into the object at an angle determined by Snell's law. If the object is specularly reflective, then a *reflection ray* is reflected about the surface normal. Each of these reflection and refraction rays may, in turn, recursively spawn shadow, reflection, and refraction rays. The rays thus form a *ray tree*. Recursion terminates when either the contribution is too small or a certain maximum depth is reached. The rays originating from the eye point are often called *primary rays*, while shadow rays, refraction rays, and reflection rays often are called *secondary rays*. Usually only a small fraction of the total number of rays are primary rays.

The usual implementation is by a general algorithm which, given a scene and a ray (defined by an origin point and a direction vector), performs computation to follow its

---

[1]a ray is defined by an origin point and a direction vector

path. The main loop of the ray tracing algorithm could look like this:

```
for each pixel (x,y) in the picture do
    ray = <compute the primary ray using (x,y)>;
    color = trace(scene, ray);
    plot(x, y, color);
```

Since `trace` calls itself recursively to model reflected and refracted light, the algorithm is rather time consuming.

Partial evaluation applied to ray tracing. In all calls to `trace` the scene is the same, which makes partial evaluation highly relevant. Thus we specify that `scene` is static and `ray` is dynamic. Given the program and the static scene data, the partial evaluator will produce a specialized program, where the main loop will look like this:

```
for each pixel (x,y) in the picture do
    ray = <compute primary ray using (x,y)>;
    color = trace_scene(ray);
    plot(x, y, color);
```

The function `trace_scene` is a version of `trace`, which is specialized with respect to one particular scene. It is often significantly faster than the general trace function.

Plan. We develop a program that implements this general algorithm. The goal is to make the ray tracer as efficient as possible, without implementing advanced ray tracing acceleration features. To assess the efficiency of the ray tracer, we compare it with Rayshade 4.0, which is one of the fastest public domain ray tracers [Haines 93].

Then we specialize the ray tracer, to see how well partial evaluation can optimize the program. Due to the large amount of computation, even a small speedup of 1.1-1.3 would be quite worthwhile. By comparison, ordinary register allocation typically gives speedups around 1.1.

2   The ray tracer

Our main focus in this project is automatic optimization of basic ray tracing techniques, and not advanced ray tracing features. Therefore we will implement a simple ray tracer. The following objects are handled: spheres, squares and discs. Only point light sources are implemented.

The intensity at the intersection point is calculated using a simple version of Whitted's shading model:

$$I_\lambda = K_{a\lambda}I_{a\lambda} + K_{d\lambda}\sum_{n=1}^{ls}(-L_n \cdot N)I_{n\lambda} + K_s\left[I_{r\lambda} + \sum_{n=1}^{ls}(-N \cdot H_n)^{Ns}I_{n\lambda}\right] + K_tI_{t\lambda}$$

$$\lambda \in \{red, green, blue\}$$

I is the resulting intensity. $K_a, K_d, K_s,$ and K_t are surface constants that specify ambient reflectance, diffuse reflectance, specular reflectance, and transmission, respectively. I_a is the amount of ambient light in the scene. The ordinary contribution from light source n is modelled by $K_{d\lambda}(-L_n \cdot N)I_{n\lambda}$, where L_n is the unit light vector (from the light source to the intersection point), N is the unit surface normal, and $I_{n\lambda}$ is the intensity of the light source. The factor $K_s(-N \cdot H_n)^{Ns}I_{n\lambda}$ models highlight produced by light source n. Here H_n is the average between the light vector reversed $(-L_n)$ and the incident ray reversed $(-R)$, where the incident ray is the ray from the eye point to the intersection point. Ns is a surface constant specifying how concentrated the highlight is. $I_{r\lambda}$ is the amount of reflected light, and $I_{t\lambda}$ is the amount of transmitted light.

The values of $I_{r\lambda}$ and $I_{t\lambda}$ are calculated by recursively evaluating the shading equation at the closest surface that the reflected and transmitted ray intersects.

The algorithm for the ray tracer is shown in Figure 1. The scene data is represented by a global data structure, which is not shown explicitly. The function `intersect_all` computes the closest intersection point (if any) between the ray and all the objects in the scene.

In two places an intersection test between a ray and all the objects in the scene is carried out; once in `trace`, and once in `shade`, to determine whether or not a given point is in shadow from given light source (the shadow ray test). Since these tests involve relatively much computation, and are carried out very often, the ray tracer will spend most of its time in the intersection code. Thus, we concentrate our effort on optimizing that code.

We refer to Appendix A for the program that implements the algorithm. The appendix also contains a description of the program. To avoid spending time coding a parser for scene description files, a number of scenes are hard-coded into the ray tracer, each identified by a number. When the ray tracer is executed, the desired scene is selected by supplying its number as an argument.

3   Experiment Platform

The ray tracer runs on a HP 9000/735 (frigg) running HP-UX version A.09.05, and the specializer runs on a Sparc 2 (fenris) running SunOS 4.1.3.

We have experimented with Gnu's C compiler `gcc` version 2.5.8 and the built-in C compiler `cc` supplied by HP. However it is not the case, that one compiler is always

```
  for(y = 0; y < screen.height; y++) {                         /* main loop */
    for(x = 0; x < screen.width; x++) {
      ray = <compute primary ray>;
      trace(0, ray, &col);
      plot(x, y, col);
    }
  }

/* Compute the contribution from ray and store it in col */
void trace(int level, rayType ray, colorType *col)
{
  object = intersect_all(ray, &distance);
  if (<object hit>)  shade(level, ray, object, distance, color);
  else               *col = background_color;
}


void shade(int level, int object, int distance, colorType *color)
{
  color = (0, 0, 0);
  sinfo = <surface information of the intersected object>;

  if (<leaving object?>) {
    if (level < maxlevel && <total internal reflection does not occur>) {
      tray = <calculate transmission direction>;   /* if we are leaving an    */
      trace(level + 1, tray, &tcol);               /* object, only transmitted */
      color = sinfo.transparency * tcol;           /* light contributes        */
    }
    return;
  }
  color = sinfo.ambient * scene.ambient;                    /* Ambient Light    */
  diffuse = (0, 0, 0);
  specular = (0, 0, 0);
  for (n = 0; n <= last_light; n++) {
    tray = <calculate the shadow ray>;
    intersect_all(tray, &tdistance);
    if (tdistance > <distance from the intersection point to the light source>) {
      diffuse = diffuse + light[n].c;                    /* Contribution from */
      specular = specular + <highlight contribution>;   /* the light source  */
    }
  }
  if (level < maxlevel) {
    tray = <calculate the reflection ray>;               /* Reflected light   */
    trace(level + 1, tray, &tcol);
    specular = specular + tcol;

    if (<total internal reflection does not occur>) {    /* Transmitted light */
      tray = <calculate transmission direction>;
      trace(level + 1, tray, &tcol);
      color = color + sinfo.transparency * tcol;
    }
  }
  color = color + diffuse * sinfo.diffuse + specular * sinfo.specular;
}
```

Figure 1: Ray tracing algorithm

better than other. As one would expect it depends on the program and the input to the program. To shed some light over the interactive between C-Mix and optimizing compilers, we report execution times for the programs compiled with both compilers.

The following parameters are used: For both compilers it is specified that all the intersection functions must be inlined. The optimizations option `-O2` is used for `gcc`, and `+O4 +Onolimit` is used for `cc`. The option `+Onolimit` means that the compiler may used as much memory and time as it finds necessary during compilation. To reduce the effect of caching, multiprogramming, virtual memory, etc. in the time measurements, we have execucted each program at least three times at a time no other CPU-intensive process were running. In case the times varied we have executed the program a few extra time. We report the fastest running time, since it represents the run where the process was least affected by external circumstances.

In the experiments we use the following scenes:

Scene	Description
1	1 light source, 2 spheres, 1 square
2	1 light source, 5 spheres
3	1 light source, 5 discs
4	1 light source, 5 squares
5	1 light source, 36 spheres, 1 square
6	5 light sources, 5 spheres
7	5 light sources, 5 discs
8	5 light sources, 5 squares
9	5 light sources, 36 spheres, 1 square

Scene 1 is just a simple scene. The purpose of specializing the ray tracer with respect to scene 2, 3 and 4 is to examine if one intersection routine is better suited for specialization than another. The reason for including scene 5 is to see what happens when the ray tracer is specialized with respect to a larger scene. And lastly the reason for having scenes 6 to 9 is to see how the number of light sources affect specialization.

4 Comparison with Rayshade 4.0

To show that our implementation is indeed an efficient one, we compare the unspecialized version of our implementation with Rayshade.

In order to make the comparison as fair as possible, we have tried to set the options to Rayshade, so the two ray tracers will perform the same computations. The options we have used can be found in the scene description files for Rayshade in Appendix B. It was not possible to configure Rayshade to do exactly the same as our implementation, because Rayshade always supersamples (i.e. traces more than one primary ray per pixel) the pixels on the edge of the picture, and because Rayshade uses bounding volumes. We thus compare the ray tracers on the raw speed of intersection tests.

The 'Tests' column shows the total number of intersection tests made. The 'Time' column shows the total running time in CPU user seconds. The 'Ratio' column shows the running time in microseconds divided by the total number of tests.

Scene	Rayshade 4.0			Our implementation		
	Tests	Time	Ratio	Tests	Time	Ratio
1	4561634	42.4	9.3	2743881	16.3	5.9
2	2290111	27.3	11.9	2100910	7.4	3.5
3	2021108	16.6	8.2	2307030	11.7	5.1
5	127751256	343.4	2.7	62082648	160.4	2.6
6	2979125	20.4	6.9	3581190	12.3	3.4
7	2864048	22.7	7.9	3394870	17.1	5.0
8	266095358	724.7	2.7	129378384	326.0	2.5

For all these scenes our implementation is faster than Rayshade measured on raw intersection speed. This is no big surprise, since the intersection code in Rayshade is very modular in order to make introduction of new object types painless. Also Rayshade has a lot more features than our implementation. Still, it is safe to conclude that our implementation is efficient.

5 Application of Partial Evaluation to Ray Tracing

A series of experiments was carried out in which we specialized our ray tracer to several scenes. This section describes the speedups obtained and the reasons for them.

To compare how different parts of the program benefits from specialization, we have performed four experiments, specializing a bit more of the program each time: 1. specialization of the intersection functions with respect to the scene. 2. specialization of the intersection functions and the shading function with respect to the scene. 3. specialization of the intersection functions and the shading function with respect to the scene and the light sources. 4. specialization of the intersection functions and the shading function with respect to the scene, the light sources, and the eye point. The table below shows which data is static (S) respectively dynamic (D) in the four experiments.

Experiment	Objects	Surface Data	Light sources	Eye point	The rest
1	S	D	D	D	D
2	S	S	D	D	D
3	S	S	S	D	D
4	S	S	S	S	D

'The rest' is the specification of the window and the resolution of the image. This data together with the eye point determines the primary rays.

The specialization in experiment 1 and 2 cannot eliminate any of the computation

(multiplications, additions, etc.), since nothing depends solely on object data, but still some speedup from (inter-procedural) constant propagation and loop unrolling can be expected. In experiment 3 and 4 additional speedup is possible, since part of the intersection computation depends solely on the object data and light source or the eye point data.

We have not included the time it takes to specialize in the measurements below. The reason is that the specialized ray tracer is expected to be run several times on different values of the dynamic data, e.g. surface data, light sources, eye point (for animation, e.g. a 'fly through'). However one can benefit from specializing even if the residual program is run only once for large images, since the time it takes to specialize does not depend on the image size. This is also the case for complex scenes, i.e. scenes with many reflecting or transparent objects.

5.1 Specialization of `intersect_all` with respect to the scene

The function `intersect_all` has been specialized with respect to scene 1, 2, 3, 4 and 5. The time is given in CPU user seconds, and the size gives the number of kilobytes of the objectfile (`.o`) as reported by `size`. 'Spd' is the speedup.

Scene	gcc					cc				
	Original		Specialized			Original		Specialized		
	Time	Size	Time	Size	Spd	Time	Size	Time	Size	Spd
1	16.3	25.4	12.3	20.5	1.3	16.1	23.8	7.9	21.9	2.0
2	7.4	25.4	5.7	21.0	1.3	6.9	23.8	3.9	22.6	1.8
3	11.7	25.4	8.7	21.7	1.3	14.9	23.8	6.1	23.9	2.4
4	18.3	25.4	12.1	21.5	1.5	22.4	23.8	8.6	23.7	2.6
5	160.4	25.4	116.1	31.7	1.4	154.6	23.8	66.6	39.5	2.3

What gives the speedup? The following transformation has been performed:

- Inter-procedural constant propagation

- Unrolling of the while loop in `intersect_all`

- Specializing the switch-case statement in `intersection` away

These transformations alone cannot account for the speedups in the case of the experiments with `cc`. It is probably the case that there is a positive interactive between C-Mix and `cc`, i.e. simplification of the program may allow the compiler to perform new optimizations: algebraic simplifications, better register allocation, better pipeline utilization, etc. However it is very hard to pinpoint exactly what the extra optimizations are without inspecting the generated assembler code, and even then it is a time consuming task.

Note that none of the intersection computation (multiplications, etc.) has been eliminated, since nothing depends solely on object data.

9

5.2 Specialization of `intersect_all` and `shade` with respect to the scene

Although only relatively little time is spent in the shading function, we try to specialize it anyway, to see if additional speedup can be gained. We specialize to the same five scenes as above.

Scene	gcc					cc				
	Original		Specialized			Original		Specialized		
	Time	Size	Time	Size	Spd	Time	Size	Time	Size	Spd
1	16.3	25.4	11.0	23.3	1.5	16.1	23.8	7.8	21.9	2.1
2	7.4	25.4	5.2	24.5	1.4	6.9	23.8	3.6	22.6	1.9
3	11.7	25.4	8.2	24.8	1.4	14.9	23.8	5.8	23.9	2.6
4	18.3	25.4	11.7	24.7	1.6	22.4	23.8	8.1	23.7	2.8
5	160.4	25.4	111.2	91.9	1.4	154.6	23.8	65.3	39.5	2.4

As expected a little extra speedup was achieved.

5.3 Specialization of `intersect_all` (and `shade`) with respect to the objects and the light sources

To achieve extra speedup, we have applied a simple but very important binding-time improvement. Normally, when we want to determine whether a point A is in shadow from a light source at point B, we perform an intersection test between the scene and the vector from A to B. In the ray tracer a ray is represented by an origin point and a vector, and since A is dynamic, both the point and the vector will be dynamic. The binding-time improvement is to do the intersection test the other way around: from B to A. This has no effect at all on the original program, but the binding-time separation is improved: now the origin point is equal to B, which is static, even though the vector is dynamic.

Since part of the intersection computation depends solely on the object data and the origin of the ray, additional speedup can be expected for shadow ray tests and primary ray intersection tests (i.e. specializing with respect to the eye point, see the next section).

We have specialized the ray tracer with respect to the same five scenes above and four new scenes. The new scenes consist of the same objects as scene 2, 3, 4, and 5, but are lighted by 5 light sources instead.

Scene	gcc					cc				
	Original		Specialized			Original		Specialized		
	Time	Size	Time	Size	Spd	Time	Size	Time	Size	Spd
1	16.3	25.4	10.1	29.3	1.6	16.1	23.8	7.4	32.3	2.2
2	7.4	25.4	5.0	31.1	1.5	6.9	23.8	3.5	35.8	1.9
3	11.7	25.4	7.9	31.9	1.5	14.9	23.8	5.6	38.1	2.7
4	18.3	25.4	10.9	31.8	1.7	22.4	23.8	7.6	37.7	2.9
5	160.4	25.4	100.7	108.9	1.6	154.6	23.8	59.6	139.4	2.6
6	12.3	25.4	7.6	47.0	1.6	11.5	23.8	5.4	57.5	2.1
7	17.1	25.4	11.0	50.1	1.6	22.1	23.8	8.6	61.4	2.6
8	35.9	25.4	18.7	49.7	1.9	43.4	23.8	14.2	61.3	3.1
9	326.0	25.4	175.6	219.5	1.9	315.4	23.8	105.6	295.2	3.0

The additional speedup is substantial for the scenes with more than one light source, which is no surprise, since some of the intersection computation has been eliminated.

Since other parts of the intersection computation depend solely on the object data and the direction of the ray, a similar specialization can be performed for directional light sources.

5.4 Specialization of `intersect_all` (and `shade`) with respect to the objects, the light sources, and the eye point

Since the shadow ray test and primary ray test have the same characteristics with respect to binding times, the above experiment can be extended to include specializing with respect to eye points by changing only a single line in the source code.

We have specialized the ray tracer with respect to the same scenes as above.

Scene	gcc					cc				
	Original		Specialized			Original		Specialized		
	Time	Size	Time	Size	Spd	Time	Size	Time	Size	Spd
1	16.3	25.4	9.4	31.5	1.7	16.1	23.8	7.0	36.1	2.3
2	7.4	25.4	3.9	34.2	1.9	6.9	23.8	3.0	40.0	2.3
3	11.7	25.4	6.4	35.8	1.8	14.9	23.8	4.8	43.8	3.1
4	18.3	25.4	9.3	35.4	2.0	22.4	23.8	6.6	41.9	3.4
5	160.4	25.4	94.3	131.3	1.7	154.6	23.8	56.6	175.2	2.7
6	12.3	25.4	6.5	50.1	1.9	11.5	23.8	5.0	61.5	2.3
7	17.1	25.4	9.4	53.9	1.8	22.1	23.8	7.9	66.0	2.8
8	35.9	25.4	17.2	53.3	2.1	43.4	23.8	13.2	65.7	3.3
9	326.0	25.4	171.1	241.9	1.9	315.4	23.8	105.0	324.4	3.0

There is a clear connection between the addition speedup gained and the ratio between primary rays and total number of rays. In scene 2, where the additional speedup is largest,

62 % of the rays are primary rays. In scene 9, where the additional speedup is insignificant, the primary rays only make up 7 % of the total number of rays.

6   Binding-time improvements

Two programs that are semantically, and even operationally, equivalent with respect to time or space usage may specialize very differently, giving residual programs with large differences in efficiency, size, or runtime memory usage. A program transformation that preserves semantics but makes the program more suited for partial evaluation is called a binding-time improvement. [Jones 93, Chapter 12].

One binding-time improvement has already been described in Section 5.3. In this section we will describe a binding-time improvement that will solve a problem with code size explosion, which sometimes occurs when unrolling loops. We have also applied some of the well known binding-time improvements, namely 'bounded static variation (The Trick)' [Jones 93, Chapter 12], splitting partially static data structures, and polyvariant binding times of functions by copying the functions. The C variants of these are described.

Reducing the size of the residual program.   This binding-time improvement has been applied in all the experiments. It reduces the size of the residual program from $O(m^2)$ to $O(m)$, where m is the number of objects in the scene. A similar problem occurs when binary search is specialized [Jones 93, Chapter 13].

The problem occurs in the while loop in `intersect_all`. The function computes the closest intersection point (if any) between the ray and all the objects in the scene by testing the objects one by one. If the ray intersects an object that is closer than a previous intersection point, then the distance is stored in `*isect_t` and the object's index is stored in `n`. The distance between the ray's origin point and the intersection must be greater than the constant `MIN_DISTANCE`, which is defined as 0.001. This is necessary when tracing secondary rays to avoid hitting the object from where the ray originates.

```
int intersect_all(rayType ray, double *isect_t)
{
    int i, n;
    double t1;

    n = -1;
    i = 0;
    while(scene[i].tag != NONE) {
        t1 = intersection(i, ray);
        if (MIN_DISTANCE < t1 && t1 < *isect_t - MIN_DISTANCE) {
            *isect_t = t1;
            n = i;
        }
        i += 1;
    }
    return n;
}
```

The array `scene` is static, and the variables `ray` and `isect_t` are dynamic. The binding-time analysis will classify `n` (and `i`) as static, since they do not depend on dynamic data. The while loop will then be specialized with respect to the different values of `n`, which will result in a big residual program. Forcing `n` to be dynamic will reduce the size of the residual program. The problem is illustrated by the following example.

Example 6.1 Consider the following piece of code containing a similar while loop:

```
int f(int *a)
{
    int i, n, m;

    i = 0;
    n = -1;
    m = 4;
    while (i < m) {
        if (a[i]) n = i;
        i += 1;
    }
    return n;
}
```

Suppose `a` is dynamic. The binding-time analysis will classify `i` and `n` as static, and the result will be the *annotated* program shown below. Statements, expressions and variables that depend on dynamic input is underlined. The `lift` marks a static expression that occurs in a dynamic context:

```
int f(_int _*a)
{
    int i, n, m;

    i = 0;
    n = -1;
    m = 4;
    while (i < m) {
        _if (_a[lift(i)]) n = i;
        i += 1;
    }
    _return(lift(n));
}
```

When `f` is specialized, the dynamic if-statement will be specialized with respect to all different possible combination of the values of `n` and `i`. This is perfectly correct since `n` and `i` are still live (`n` is used in the return statement, and `i` is used in `n = i` for instance). The residual program is shown below (to improve readability some post processing has been done: removing superfluous parentheses, etc.)

```
int f_1(int *a)
{
    if (a[0]) {
        if (a[1]) {
        lab_11:
            if (a[2]) {
            lab_15:
                if (a[3])
                    return 3;
                else
                    return 2;
            }
            else {
                if (a[3])
                    return 3;
                else
                    return 1;
            }
        }
        else {
            if (a[2]) {
                goto lab_15;
            }
            else {
                if (a[3])
                    return 3;
                else
                    return 0;
            }
        }
    }
    else {
        if (a[1]) {
            goto lab_11;
        }
        else {
            if (a[2]) {
                goto lab_15;
            }
            else {
                if (a[3])
                    return 3;
                else
                    return -1;
            }
        }
    }
}
```

When the function is specialized, the first time around in the loop, the if-statement will be specialized with respect to `i = 0` and `n = -1`, which yields the first if-statement in

the residual program. Following the then-branch in the original program, the if-statement will be specialized with respect to `i = 1` and `n = 0`. Following the else-branch, the if-statement will also be specialized with respect to `i = 1` and `n = -1`. This will produce the two conditionals `if (a[1]) ...`, etc. In general when the loop is iterated m times, the if statement will be specialized $\sum_{i=1}^{m} i$ times or $O(m^2)$.

When `n` is forced to be dynamic, the annotated program is as follows:

```
int f(_int _*a)
{
    int i;
    _int n;
    int m;

    _n = lift(-1);
    i = 0;
    m = 4;
    while i < m) {
        _if (_a[lift(i)]) _n = lift(i);
        i += 1;
    }
    _return(_n);
}
```

Now the dynamic if-statement will only be specialized with respect to the different values of `i`, so much more code can be shared. The residual program:

```
int f_1(int *a)
{
    int n;

    n = -1;
    if (a[0]) {
        n = 0;
    lab_11:
        if (a[1]) {
            n = 1;
        lab_18:
            if (a[2]) {
                n = 2;
            lab_25:
                if (a[3]) {
                    n = 3;
                lab_32:
                    return n;
                }
                else goto lab_32;
            }
            else goto lab_25;
        }
        else goto lab_18;
    }
    else goto lab_11;
}
```

The if-statement in the original program is specialized with respect to `i` equal to 0, 1, 2, and 3. Since the code is not specialized with respect to different `n`'s, the size of the residual program only grows linearly.

Even though the function in this example is not very realistic, we will report some measured speedups and increases in code size. In the first specialized version `n` is static and in the second version `n` is dynamic. We have specialized each version with respect to `m` equal to 50 and `m` equal to 100. Since the control flow of the programs depend on the values of elements in the array `a`, we have measured the running time for each program given two different arrays. In one run the array contains only zeros, and in the other it contains only ones as shown in the 'Values' column under `a`. In all runs the function `f` were called a million times. The columns labeled 'Spd' show the speedups, and the columns labeled 'Blw' show the blowups in size (measured as the size of the residual program divided by the size of the original program).

Compiler	Values		Original		Specialized version 1				Specialized version 2			
	m	a	Time	Size	Time	Size	Spd	Blw	Time	Size	Spd	Blw
gcc	50	0	3.2	0.04	1.8	11.0	1.8	280	1.6	0.6	2.0	15
gcc	50	1	3.2	0.04	1.7	11.0	1.9	280	1.6	0.6	2.0	15
gcc	100	0	6.2	0.04	4.3	53.6	1.4	1373	3.2	1.2	1.9	30
gcc	100	1	6.2	0.04	3.3	53.6	1.9	1373	3.2	3.5	1.9	30
cc	50	0	3.2	0.04	2.6	15.0	1.2	385	1.7	0.6	1.9	15
cc	50	1	3.2	0.04	1.7	15.0	1.9	385	1.6	0.6	2.0	15
cc	100	0	6.2	0.04	4.6	59.4	1.3	1521	3.2	1.2	1.9	30
cc	100	1	6.2	0.04	3.3	59.4	1.9	1521	3.2	1.2	1.9	30

The measured sizes of the residual programs confirm that the binding-time improvement reduces the size of the residual program from $O(m^2)$ to $O(m)$. It is noteworthy that doing the binding-time improvement also results in a bigger speedup. The reason could be that the smaller program has a higher cache hitrate.

It is perhaps worth mentioning that even though the object files in the worst cases are only around 59 Kb, the corresponding C program is 613 Kb. Compiling is therefore a rather time consuming task (several minutes), and may also use a substantial amount of memory (e.g. 20 Mb). This might be reason enough to apply the binding-time improvement; at least while developing the program. □

For larger scenes, the binding-time improvement is necessary. Without it the specializer will either run out of memory, or produce a huge residual program that will slow everything down. Applying the binding-time improvement does not affect the speedup as drastic as in the example above. The reason is that the loop in the ray tracer contains much more code making the impact on compiler optimizations much less – if any at all.

The explosion in code size occurs because a static loop contains a dynamic conditional statement that changes the value of a live static variable. Identification of loops and

live variable analysis are described in [Aho 86]. With this information the binding-time improvement can be applied automatically, at the risk of being too conservative. It might be the case that the loop is only iterated a few times (e.g. 10) in which case the user may want the program specialized with respect to the variable in question. Even if the loop is iterated say 50 times, the user may prefer a large residual program knowing it will pay off.

Bounded static variation (The Trick). The technique can be employed when a dynamic variable is known to assume one of a finite set of statically computable values. In the ray tracer this occurs in the `trace` function:

```
void trace(int level, double weight, rayType ray, colorType *color)
{
    double isect_t;
    int isect_object, n;

    isect_t = 20000000000.0; /* 2.0e+10; */
    isect_object = intersect_all(ray, &isect_t);
    if (isect_object == -1)
        background_color(color);
    else
        shade(level, weight, ray, isect_t, isect_object, color);
}
```

Here `isect_object` is dynamic, but since we know it lies in the range between -1 and `last_object` inclusive, we can replace the call to `shade` with a for loop:

```
void trace(int level, double weight, rayType ray, colorType *color)
{
    double isect_t;
    int isect_object, n;

    isect_t = 20000000000.0; /* 2.0e+10; */
    isect_object = intersect_all(ray, &isect_t);
    if (isect_object == -1)
        background_color(color);
    else {
      for(n = 0; n <= last_object; n += 1)
            if (isect_object == n) {
                shade(level, weight, ray, isect_t, n, color);
                break;
            }
    }
}
```

The variable `isect_object` is still dynamic, but `n` is static, which enables us to specialize `shade` with respect to each object (and its surface data) in the scene.

In this case it is possible to detect that `isect_object` is limited by two values, so the binding-time improvement can be applied automatically. However we might not want to apply 'The Trick' in case a variable is limited by, say -10000 and 10000, to avoid a huge residual program.

Splitting partially static data structures. Splitting of partially static data structures is not difficult to do automatically, but was not implemented in the specializer used in this project. Instead, the binding-time analysis makes a conservative choice by classifying arguments with mixed binding-times as dynamic. In this project the situation arises in two variants: 1) Where partially static data structures are used as arguments to a function. 2) Assignment of structs, where the left hand side is dynamic and the right hand side is static. In the example below `N` is a dynamic vector (a struct with three elements) and we want `scene` to be static, so

```
N = scene[isect_object].u2_disc.n;
```

is changed into

```
N.x = scene[isect_object].u2_disc.n.x;
N.y = scene[isect_object].u2_disc.n.y;
N.z = scene[isect_object].u2_disc.n.z;
```

The right hand expressions of the assignments can now be lifted, since they are of base types. Remark: these binding-time improvements are now performed automatically by the new version of C-Mix.

Polyvariant binding-times by copying. Some of the vector functions are sometimes called with static arguments and sometimes with dynamic arguments. Since the binding-time analysis requires that the binding-time of a function's argument must be the same in all calls to that function, we make two copies of each relevant function: one for the static calls and one for the dynamic calls. The intersection functions are copied for the same reason in the third and fourth experiment.

7 Conclusion

We have used partial evaluation to optimize an already efficient ray tracer, gaining speedups from 1.8 to 3.3 (using cc) and from 1.3 to 2.1 (using gcc) depending on the scene and the degree of specialization. Much of the speedup comes from inter-procedural constant propagation and unrolling of loops. Most optimizing compilers will perform these kinds of optimizations, but generally only based on intra-procedural information, whereas C-Mix is based on inter-procedural information and part of the input. However C-Mix is very aggressive, and will unroll a (static) loop regardless of the increase in code

size. This means the user must aid C-Mix in some cases by specifying that a particular loop should not be unrolled.

The experiments with the ray tracer revealed a shortcoming of the C-Mix implementation, namely that it used far too much memory. The reason was that all the global data (including the scene description) was copied for each specialized function. The memory usage can be reduced as follows. For each specialized function it is sufficient to store the data that is *in use*. In case two functions are specialized with respect to the same global data, it is only necessary to keep one copy in memory. An *in-use* analysis and a scheme for handling identical copies of global data is described in Section 6.3 and Section 3.10 of [Andersen 94] respectively.

Many C programs have some global data structures, which are initialized in the beginning of the run and do not change during the rest of the run. Specializing that kind of program will most likely pay off – how well depends on how heavily the global data is used.

7.1 Related Work

Mogensen specialized a very modular ray tracer written in a functional language [Mogensen 86], showing that the administrative overhead could be removed.

Hanrahan has a 'surface compiler' which accepts as input the equation of a surface and outputs the intersection code as a series of C statements [Hanrahan 83]. This is clearly a form of partial evaluation targeted for a specific application. His surface compiler also performs algebraic simplification, whereas C-Mix leaves this for the C compiler. He reports a speedup of 1.3.

7.2 Future Work

It would be interesting to see if it is possible to obtain similar results by applying C-Mix to a 'real' ray tracer, for example Rayshade. The major obstacle is of practical nature. Rayshade uses alot of function pointers and dynamic allocation, and the C-Mix implementation cannot handle these yet. The theory has already been developed in [Andersen 94].

It would also be interesting to see how the presence various ray tracing acceleration techniques would impact specialization of a ray tracer. There are two major ways to make a ray tracer run faster: one is to optimize the intersection computation, which has been done in this project, the other is to reduce the number of intersection tests. The latter can be realized in several ways: spatial subdivision of the scene (uniform or non-uniform), bounding volumes, clever representations (e.g. octtrees), ... Specializing the intersection code in the presence of a clever representation will unfold the structure of the scene data into structure in the residual program. This might give extra speedup since more computation can be done at specialization time.

Another subject that deserves attention is the interplay between the partial evaluator and the compiler (or interpreter) for the particular language. In the case of C-Mix it amounts to examine how specialization affects subsequent optimizations (and perhaps

also code generation) performed by the compiler. Section 9.3 of [Andersen 94] study the interference between partial evaluation and classical optimizations such as loop invariant motion and common subexpression elimination. However to get a full understanding of what goes on, one must examine the assembler code produced by the compiler.

References

[Aho 86]   Aho, A.V., Sethi, R., and Ullman, J.D. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[Andersen 92a] Andersen, L.O. Partial Evaluation of C and Automatic Compiler Generation (extended abstract). In Kastens, U. and Pfahler, P. (editors), *Proc. of Compiler Constructions—4th International Conference, CC'92 (LNCS 641)*, pages 251–257. Springer-Verlag, October 1992.

[Andersen 92b] Andersen, L.O. Self-Applicable C Program Specialization. In *Proceeding of PEPM'92: Partial Evaluation and Semantics-Based Program Manipulation*, pages 54–61. June 1992. Available as Technical Report from Yale University.

[Andersen 93]   Andersen, L.O. Binding-Time Analysis and the Taming of C Pointers. In Schmidt, David (editor), *Proc. of ACM Symposium on Partial Evaluation and Semantics-Based Program Manipulation, PEPM'93*, pages 47–58. 1993.

[Andersen 94]   Andersen, L.O. *Program Analysis and Specialization for the C Programming Language.* PhD thesis, DIKU, University of Copenhagen, May 1994. (DIKU report 94/19).

[Baier 94]   Baier, Romana, Glück, Robert, and Zöchling, Robert. Partial Evaluation of Numerical Programs in Fortran. In *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation.* 1994.

[Berlin 90a]   Berlin, A. and Weise, D. Compiling Scientific Code Using Partial Evaluation. *IEEE Computer* 23(12):25–37, December 1990.

[Berlin 90b]   Berlin, A.A. Partial Evaluation Applied to Numerical Computation. In *1990 ACM Conference on Lisp and Functional Programming, Nice, France*, pages 139–150. New York: ACM, 1990.

[Berlin 94]   Berlin, Andrew A. and Surati, Rajeev J. Partial Evaluation for Scientific Computing: The Supercomputer Toolkit Experience. In *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation.* 1994.

[Foley 90]   Foley, van Dam, Feiner, and Hughes. *Computer Graphics principles and practice.* Reading, MA: Addison-Wesley, 1990.

[Glassner 89]   Glassner, Andrew (editor). *An Introduction to Ray Tracing.* Academic Press, 1989.

[Haines 93]   Haines, Eric. Ray Tracer Races, Round 2. *Ray Tracing News* , july 1993.

[Hall 89]   Hall, Roy. *Illumination and Color in Computer Generated Imagery.* 1989.

[Hanrahan 83]   Hanrahan, Pat. Ray Tracing Algebraic Surfaces. *Computer Graphics* Volume 17, Number 3, july 1983.

[Jones 93]   Jones, N. D., Gomard, C. K., and Sestoft, P. *Partial Evaluation and Automatic Program Generation.* Prentice-Hall, 1993.

[Jorgensen 91]   Jørgensen, Jesper. Compiler Generation by Partial Evaluation. Master's thesis, DIKU, University of Copenhagen, Denmark, 1991.

[Lisper 91]   Lisper, B. Detecting Static Algorithms by Partial Evaluation. In *Partial Evaluation and Semantics-Based Program Manipulation, New Haven, Connecticut. (Sigplan Notices, vol. 26, no. 9, September 1991)*, pages 31–42. New York: ACM, 1991.

[Mogensen 86]   Mogensen, Torben. The application of Partial Evaluation to Ray-Tracing. Master's thesis, DIKU, University of Copenhagen, Denmark, 1986.

[Mossin 93]   Mossin, Christian. Partial Evaluation of General Parsers. In Schmidt, David (editor), *Proc. of ACM Symposium on Partial Evaluation and Semantics-Based Program Manipulation, PEPM'93.* 1993.

A   Program

A.1   Functionality

This section briefly describes the functionality of the central functions in the ray tracer. A description of how the highlight component is computed and how the image is displayed is also included.

The `main` function parses the command line arguments and sets the variables `to_file`, `to_screen`, `use_scene`, and `image_file` accordingly. Then `init_scene` and `init_srgp` is called. At last the main loop is entered:

```
rayType ray;
colorType col;

for(y = 0; y < screen.height; y++) {
    for(x = 0; x < screen.width; x++) {
        compute_primary_ray(x + 0.5, y + 0.5, &ray);
        trace(0, 1.0, ray, &col);
        if (to_screen)
            plot(x, y, col);
        if (to_file)
            ...
    }
}
```

For each pixel, the ray that runs from the eye point towards the center of the pixel is calculated and stored in `ray`. A trace is started with level `0` and weight `1.0`. The result is stored in `col`.

The function `init_scene` sets the variables `view`, `screen`, `background`, `maxlevel`, and `minweight` to their default values. Then the appropriate scene function is called, which creates the objects and light sources in the scene. The scene initialization function may also change some of the default values mentioned above. At last `view.dir`, `screen.firstv`, `screen.scrnx`, and `screen.scrny` are calculated on the basis of the view and screen specifications.

The `trace` function calls `intersect_all`. If the ray hits an object, `shade` is called to determine the contribution from that ray, otherwise the background color is returned.

The main intersection function `intersect_all` tests the all objects to find the closest intersection (if any). The distance from the ray's anchor point and the intersection point must be greater than `MIN_DISTANCE` (which is equal to 0.001), and less than `*isect_t - MIN_DISTANCE`. The last condition is required for shadow-ray tests. If the ray intersections an object, `intersect_all` returns the object's index and `*isect_t` is set to the distance from the ray's anchor point to the intersection point.

The `shade` function. Set `*color` to (0,0,0). Calculate the intersection point `isect.p` from `isect.t` and `ray`. Determine the surface type and store it in `sinfo`. Compute the surface normal and store it in `N`. Compute the dot product of the surface normal and the *view vector* and store it in `n_dot_v`. Note that the view vector is equal to `-ray.v`. Flip the normal if the object has no inside (if it is flat). This means that flat objects are always entered.

If we are leaving an object, only transmitted light contributes. If `level < maxlevel` and `sinfo.transparency * weight > minweight` the transmission direction is calculated, and `trace` is called recursively.

If we are entering an object, ambient light, diffuse light, specular reflectance, and transmitted light are computed.

Calculation of the highlight component: $(N \cdot H)^{Ns}$.

$$H = \frac{-R - L}{|-R - L|} = -\frac{R + L}{|R + L|}$$

$$N \cdot H = N \cdot \left(-\frac{R + L}{|R + L|}\right) = -\frac{N \cdot R + N \cdot L}{|R + L|}$$

To avoid a square-root operation we limit Ns to even values.

$$
\begin{aligned}
|R + L|^2 &= (R_x + L_x)^2 + (R_y + L_y)^2 + (R_z + L_z)^2 \\
&= R \cdot R + L \cdot L + 2(R \cdot L) \\
&= 2 + 2(R \cdot L) \\
&= 2(1 + R \cdot L)
\end{aligned}
$$

$$(N \cdot H)^{Ns} = \left(\frac{(N \cdot R + N \cdot L)^2}{2(1 + R \cdot L)}\right)^{Ns/2}$$

Displaying the image. The image must be displaying while generating it. As the quality is less important in this phase, the image is converted to grayscale and displayed using random-dithering. The following formula for converting to grayscale is used:

$$luminosity = \frac{0.299 \times red + 0.587 \times green + 0.114 \times blue}{3}$$

A 24-bit colour image is saved in a file, which can be viewed with `imxv`. Diku's image format is used.

A.2 Files

A.2.1 ray.h

```
/*
 * Author:         Peter Holst Andersen
 * Last change:    August, 1994
 * Contents:       Constants.
 *                 Declaration of types, external variables, and functions.
 */

#define TRUE   1
#define FALSE  0
                                                                              10
#define DEFAULT_HEIGHT     128          /* Default image height    */
#define DEFAULT_WIDTH      128          /* Default image width     */

#define MAXLEVEL           5            /* Default maximum depth of
                                         * the trace tree          */
#define MINWEIGHT          0.002        /* Default minimum weight of
                                         * the trace tree          */

#define MIN_DISTANCE       0.001        /* Minimum distance for
                                         * intersection tests      */       20

#define BACKGROUND_RED            0.0   /* Background color         */
#define BACKGROUND_GREEN          0.0
#define BACKGROUND_BLUE           0.0

#define MAX_ARITY_OBJECT_TREE     50
#define MAX_OBJECTS_SCENE         50
#define MAX_SURFACES_SCENE        50
#define MAX_LIGHTS_SCENE          50
                                                                              30
/* object tags */

#define NONE                 0
#define BOUNDING_SPHERE      1
#define SPHERE               2
#define DISC                 3
#define SQUARE               4


/* surface tags */
                                                                              40
#define SIMPLE          1
#define CHECKED         2

#define PI              3.14159265

#define deg2rad(x)      (x * PI / 180.0)

typedef double myfloat;
typedef unsigned short ush;
typedef struct { myfloat x, y, z; } pointType;                                50
typedef struct { myfloat red, green, blue; } colorType;
typedef pointType vectorType;
typedef struct { pointType p; vectorType v; } rayType;

typedef struct {          /* A sphere centered in (c.x, c.y, c.z)      */
    myfloat r, r2;        /* with radius r. r2 = r*r                   */
    pointType c;
} sphereType;

typedef struct {          /* The square lies in the plane described by */    60
    pointType c;          /* n.x * x + n.y * y + n.z * z = d            */
    vectorType n, v;      /* The square is centered in (c.x, c.y, c.z)  */
```

24

```c
    myfloat r2, d;          /* v determines the orientation of the square      */
} squareType;               /* If s is the sidelength, then r2 is (s/2)^2 */

typedef struct {            /* The disc lies in the plane described by */
    pointType c;            /* n.x * x + n.y * y + n.z * z = d                  */
    vectorType n;           /* The disc is centered in (c.x, c.y, c.z)         */
    myfloat r, r2, d;       /* r is the radius of the disc, and r2 = r*r       */
} discType;                                                                        70

typedef struct {
    colorType ambient;      /* Ambient reflectance constant                    */
    colorType diffuse;      /* Diffuse reflectance constant                    */
    myfloat reflectivity;   /* Specular reflection constant                    */
    myfloat transparency;   /* Transmission constant                           */
    myfloat refrindex;      /* Refraction index                                */
    int specpow;            /* Specular exponent (= Ns/2)                      */
} surfaceInfoType;
                                                                                   80
typedef struct {            /* Positional light source in p with               */
    pointType p;            /* intensity c                                     */
    colorType c;
} lightType;

typedef struct {
    int tag;                /* SIMPLE or CHECKED                               */
    union {
        surfaceInfoType sinfo;      /* For simple surfaces                     */
        struct {                    /* For checked surfaces                    */   90
            int s1, s2;             /* Surface for the "squares"               */
            myfloat checksize;      /* Checksize                               */
            myfloat dbl_checksize;  /* Twice the checksize                     */
        } checked;
    } u;
} surfaceType;

typedef struct {
    int tag;                /* NONE, BOUNDING_SPHERE, SPHERE, DISC, or SQUARE */
    union {                                                                         100
        int skip_on_miss;   /* For BOUNDING_SPHERES                            */
        int surface;        /* For SPHERE, DISC, and SQAURE                    */
    } u1;
    union {
        sphereType sphere;
        discType disc;
        squareType square;
    } u2;
} objectType;
                                                                                   110
typedef struct {
    pointType pos;          /* Eyeposition                                     */
    pointType lookp;        /* The point we are looking at                     */
    vectorType up;          /* View up vector                                  */
    vectorType dir;         /* dir = (lookp-pos)/|lookp-pos|                   */
    myfloat lookdist;       /* lookdist = |lookp-pos|                          */
    myfloat hfov, vfov;     /* Horisontal and vectical field of view */
} viewType;

typedef struct {                                                                   120
    vectorType firstv;      /* Direction of the first primary ray              */
    vectorType scrnx, scrny;    /* Increment for the other primary rays */
    vectorType scrni;       /* scrni = norm(view.dir x view.up)                */
    vectorType scrnj;       /* scrnj = norm(scrni x view.dir)                  */
    int height, width;      /* Height and width of the screen                  */
} screenType;

typedef struct {
    myfloat t;              /* distance from the ray's anchor point            */
```

```
    int enter;               /* entering: enter = TRUE , exiting: enter = FALSE */        130
    int object;              /* Intersected object                          */
    pointType p;             /* Intersection point                          */
} intersectionType;

#ifndef INLINE_EXTERN
#define INLINE_EXTERN
#endif

int init_srgp(char *, int, int);
void plot(int, int, colorType);                                                           140
myfloat mypower(myfloat, int);
void init_scene();
int flat_object(ush);
int bounding_object(ush);
void compute_primary_ray(myfloat, myfloat, rayType *);
void trace(int, myfloat, rayType, colorType *);
void trace1(int, rayType, intersectionType *);
void shade(int, myfloat, rayType, intersectionType, colorType *);
void background_color(colorType *);
int shadow(rayType, myfloat);                                                             150
INLINE_EXTERN int intersect_all(rayType, double *);
INLINE_EXTERN myfloat intersection(int, rayType);
INLINE_EXTERN myfloat intersection_sphere(sphereType, rayType);
INLINE_EXTERN myfloat intersection_disc(discType, rayType);
INLINE_EXTERN myfloat intersection_square(squareType, rayType);
INLINE_EXTERN myfloat intersection_plane(vectorType, myfloat, rayType);
void compute_normal(intersectionType, vectorType *);
void compute_normal_sphere(intersectionType, vectorType *);
void specular_direction(myfloat, vectorType *, vectorType *, vectorType *);
int transmission_direction(myfloat, myfloat, vectorType *, vectorType *, vectorType *);    160
void create_disc(objectType *, myfloat, myfloat, myfloat, myfloat, myfloat, myfloat, myfloat, int);
void create_square(objectType *, myfloat, myfloat, myfloat, myfloat, myfloat, myfloat, myfloat, myfloat, myfloat, myfloat, int);
void create_sphere(objectType *, myfloat, myfloat, myfloat, myfloat, int);
void create_light(lightType *, myfloat, myfloat, myfloat, myfloat, myfloat, myfloat);
void calculate_uv(intersectionType, myfloat *, myfloat *);
void calculate_uv_square(intersectionType, myfloat *, myfloat *);

long random();
int srandom(int);
/* int fprintf(); */                                                                      170

void vector_sub(vectorType *, vectorType *, vectorType *);
void vector_copy(vectorType *, vectorType *);
myfloat  vector_norm(vectorType *);
myfloat  vector_dot(vectorType *, vectorType *);
void vector_cross(vectorType *, vectorType *, vectorType *);
myfloat vector_norm_cross(vectorType *, vectorType *, vectorType *);
void vector_scale(myfloat, vectorType *, vectorType *);

extern int root_object;          /* root_object is always 0                 */            180
extern objectType scene[];       /* objects in the scene                    */
extern surfaceType surface[];    /* surface descriptions                    */
extern lightType light[];        /* light sources                           */
extern int last_light;           /* last light source                       */
extern viewType view;            /* view specification                      */
extern screenType screen;        /* screen specification                    */
extern char prg_name[];          /* prg_name = argv[0]                      */
extern int number_of_colors;     /* number of colors on the Xterminal       */
extern colorType background;     /* background color                        */
extern int to_screen;            /* write to screen if to_screen = TRUE     */            190
extern int to_file;              /* write to file if to_file = TRUE         */
extern int maxlevel;             /* Maximum depth of the trace tree         */
extern myfloat minweight;        /* Minimum weight in the trace tree        */
extern int use_scene;            /* scene number                            */
extern int stat_intersections;   /* Total number of intersection tests */
extern int stat_eye;             /* Total number of eye rays                */
```

26

```
extern int stat_shadow;          /* Total number of shadow rays          */
extern int stat_reflected;       /* Total number of reflected rays       */
extern int stat_refracted;       /* Total number of refracted rays       */
```
200

```
#define srandom srand48
#define random lrand48
```

A.2.2 `main.c`

```
/*
 * Author:        Peter Holst Andersen
 * Last change:   August, 1994
 * Contents:      The main function and declaration of global variables.
 */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#ifdef USE_SRGP                                                              10
#include <srgp.h>
#endif
#include <string.h>
#ifdef USE_IMAGE
#include <image.h>
#endif
#include "ray.h"

int to_screen, to_file;
int maxlevel;                                                                20
myfloat minweight;
int number_of_colors;
char prg_name[256];
int root_object;
colorType background;
int last_light;
lightType light[MAX_LIGHTS_SCENE];
objectType scene[MAX_OBJECTS_SCENE];
surfaceType surface[MAX_SURFACES_SCENE];
viewType view;                                                               30
screenType screen;
int use_scene;
int stat_intersections;
int stat_eye;
int stat_shadow;
int stat_reflected;
int stat_refracted;

void usage()
{                                                                            40
    fprintf(stderr, "usage %s [-n scene#] [-o filename] [-s]\n", prg_name);
    exit(-1);
}

void main(int argc, char **argv)
{
    int x, y, n;
    rayType ray;
    colorType col;
#ifdef USE_IMAGE                                                             50
    color image image_out;
#endif
    char image_file[256];
    char title[256];

    strcpy(prg_name, argv[0]);
```

27

```c
            to_screen = TRUE;
            to_file = FALSE;
            use_scene = 1;
            srandom(getpid());                                                          60

            for(n = 1; n < argc; n++) {
                fprintf(stderr, "parsing arg: %s\n", argv[n]);
                if (argv[n][0] == '-') {
                    switch (argv[n][1]) {
                    case 's':
                        to_screen = FALSE;
                        break;
                    case 'o':
                        to_file = TRUE;                                                 70
                        if (++n < argc)
                            strcpy(image_file, argv[n]);
                        else
                            usage();
                        break;
                    case 'n':
                        if (++n < argc)
                            use_scene = atoi(argv[n]);
                        else
                            usage();                                                    80
                        break;
                    default:
                        usage();
                    }
                }
                else
                    usage();
            }

#ifdef COLLECT_STATISTICS                                                               90
            stat_intersections = 0;
            stat_eye = 0;
            stat_shadow = 0;
            stat_reflected = 0;
            stat_refracted = 0;
#endif

            init_scene();
            fprintf(stderr, "Init scene done\n");
                                                                                        100
#ifdef USE_SRGP
            if (to_screen) {
                sprintf(title, "%s: scene %d", prg_name, use_scene);
                number_of_colors = init_srgp(title, screen.height, screen.width);
            }
            fprintf(stderr, "init_srgp done\n");
#endif

#ifdef USE_IMAGE
            if (to_file) {                                                              110
                fprintf(stderr, "Trying to allocate image (%d, %d)\n",
                        screen.width, screen.height);
                if (static_image(image_out, screen.width, screen.height, 0) !=
                    COMPLETE) {
                    fprintf(stderr, "Could not allocate image\n");
                    to_file = FALSE;
                }
            }
            fprintf(stderr, "image allocation done\n");
#endif                                                                                  120

#ifdef DEBUG
            fprintf(stderr, "%s: scene = %d\n", prg_name, use_scene);
```

28

```c
#endif

    for(y = 0; y < screen.height; y++) {
        for(x = 0; x < screen.width; x++) {
            compute_primary_ray((double) x + 0.5, (double) y + 0.5, &ray);
#ifdef COLLECT_STATISTICS
            stat_eye++;                                                         130
#endif
            trace(0, 1.0, ray, &col);
#ifdef DEBUG
            if (col.red > 1.0 || col.red < 0.0 ||
                col.green > 1.0 || col.green < 0.0 ||
                col.blue > 1.0 || col.blue < 0.0)
                fprintf(stderr, "Color out of range (%f,%f,%f)\n",
                        col.red, col.green, col.blue);
#endif
            if (col.red > 0.9999)                                               140
                col.red = 0.9999;
            if (col.green > 0.9999)
                col.green = 0.9999;
            if (col.blue > 0.9999)
                col.blue = 0.9999;

#ifdef USE_SRGP
            if (to_screen)
                plot(x, y, col);
#endif                                                                          150

#ifdef USE_IMAGE
            if (to_file) {
                image_out[screen.height − y − 1][x].red = col.red * 256;
                image_out[screen.height − y − 1][x].green = col.green * 256;
                image_out[screen.height − y − 1][x].blue = col.blue * 256;
            }
#endif
        }
#ifdef USE_SRGP                                                                 160
        if (to_screen)
            SRGP_refresh();
#endif
        if (y % 10 == 0)
            fprintf(stderr, "main: completed line %d\n", y);
    }

#ifdef USE_IMAGE
    if (to_file) {
        put_image_class(image_out, IM_CLASS_COLOR);                            170
        if (write_image(image_file, image_out) != COMPLETE)
            fprintf(stderr, "output image cannot be written\n");
    }
#endif

#ifdef COLLECT_STATISTICS
    fprintf(stderr, "Intersections test = %d\n", stat_intersections);
    fprintf(stderr, "Eye rays            = %d\n", stat_eye);
    fprintf(stderr, "Shadow rays         = %d\n", stat_shadow);
    fprintf(stderr, "Reflected rays      = %d\n", stat_reflected);             180
    fprintf(stderr, "Refracted rays      = %d\n", stat_refracted);
#endif

    fprintf(stderr, "%s: completed\n", prg_name);
    if (to_screen)
        for(;;) sleep(10000);

}
```

A.2.3 `mypower.c`

```c
/*
 * Author:        Peter Holst Andersen
 * Last change:   August, 1994
 * Contents:      The function mypower, which computes the nth power of x,
 *                where n is an integer and x is a float, using the russian
 *                pesant algorithm.
 */

#include <stdio.h>
#include "ray.h"                                                          10

/*
 * e.g. mypower(x, 27) = x^16 * x^8 * x^2 * x^1 = x^(16+8+2+1) = x^27
 */
myfloat mypower(myfloat x, int n)
{
    myfloat a = 1.0;

    do {
        if (1 & n)                                                        20
            a *= x;
        x *= x;
    } while (n >>= 1);

    return a;
}
```

A.2.4 `ray.c`

```c
/*
 * Author:        Peter Holst Andersen
 * Last change:   August, 1994
 * Contents:      The core of the ray tracer.
 *                Functions for creating the scene: create_*
 *                Scene descriptions: scene*
 *                Intersection routines: intersection_*, intersect_all
 *                The shading function: shade
 */
                                                                          10
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#ifdef USE_SRGP
#include <srgp.h>
#endif
#include <math.h>
#include "ray.h"
                                                                          20
int last_object;

void light_falloff(colorType *col, myfloat d)
{
    if (d < 1.0)
        return;
    col->red = col->red / d;
    col->green = col->green / d;
    col->blue = col->blue / d;
}                                                                         30

void create_simple_surface(int i, myfloat am, myfloat di,
                           myfloat re, myfloat tr, myfloat ref, int specpow)
```

```
{
    surface[i].tag = SIMPLE;
    surface[i].u.sinfo.ambient.red = am;
    surface[i].u.sinfo.ambient.green = am;
    surface[i].u.sinfo.ambient.blue = am;
    surface[i].u.sinfo.diffuse.red = di;
    surface[i].u.sinfo.diffuse.green = di;                                      40
    surface[i].u.sinfo.diffuse.blue = di;
    surface[i].u.sinfo.reflectivity = re;
    surface[i].u.sinfo.transparency = tr;
    surface[i].u.sinfo.refrindex = ref;
    surface[i].u.sinfo.specpow = specpow;
}

void create_checked_surface(int i, int s1, int s2, myfloat checksize)
{
    surface[i].tag = CHECKED;                                                   50
    surface[i].u.checked.s1 = s1;
    surface[i].u.checked.s2 = s2;
    surface[i].u.checked.checksize = checksize;
    surface[i].u.checked.dbl_checksize = checksize * 2.0;
}

void create_disc(objectType *obj, myfloat cx, myfloat cy, myfloat cz,
                 myfloat nx, myfloat ny, myfloat nz,
                 myfloat r, int s)
{                                                                              60
    obj->tag = DISC;
    obj->u1.surface = s;
    obj->u2.disc.c.x = cx;
    obj->u2.disc.c.y = cy;
    obj->u2.disc.c.z = cz;
    obj->u2.disc.n.x = nx;
    obj->u2.disc.n.y = ny;
    obj->u2.disc.n.z = nz;
    obj->u2.disc.r = r;
    obj->u2.disc.r2 = r*r;                                                      70
    vector_norm(&obj->u2.disc.n);
    obj->u2.disc.d = obj->u2.disc.n.x*cx + obj->u2.disc.n.y*cy +
                     obj->u2.disc.n.z*cz;
}

void create_square(objectType *obj, myfloat cx, myfloat cy, myfloat cz,
                   myfloat nx, myfloat ny, myfloat nz,
                   myfloat vx, myfloat vy, myfloat vz,
                   myfloat r, int s)
{                                                                              80
    obj->tag = SQUARE;
    obj->u1.surface = s;
    obj->u2.square.c.x = cx;
    obj->u2.square.c.y = cy;
    obj->u2.square.c.z = cz;
    obj->u2.square.n.x = nx;
    obj->u2.square.n.y = ny;
    obj->u2.square.n.z = nz;
    obj->u2.square.v.x = vx;
    obj->u2.square.v.y = vy;                                                    90
    obj->u2.square.v.z = vz;
    obj->u2.square.r2 = r*r;
    vector_norm(&obj->u2.square.n);
    vector_norm(&obj->u2.square.v);
    obj->u2.square.d = obj->u2.square.n.x*cx + obj->u2.square.n.y*cy +
                       obj->u2.square.n.z*cz;
}

#include "../cogen/scenes.c"

                                                                             100
```

```
void create_sphere(objectType *obj,
                   myfloat x, myfloat y, myfloat z, myfloat r, int s)
{
    obj->tag = SPHERE;
    obj->u1.surface = s;
    obj->u2.sphere.r = r;
    obj->u2.sphere.r2 = r*r;
    obj->u2.sphere.c.x = x;
    obj->u2.sphere.c.y = y;
    obj->u2.sphere.c.z = z;                                                    110
    fprintf(stderr, "sphere s%d %lf %lf %lf %lf\n", s, r, x, y, z);
}

void create_light(lightType *li, myfloat r, myfloat g, myfloat b,
                  myfloat x, myfloat y, myfloat z)
{
    li->p.x = x;
    li->p.y = y;
    li->p.z = z;
    li->c.red = r;                                                            120
    li->c.green = g;
    li->c.blue = b;
}

void init_scene()
{
    myfloat magnitude;

    view.lookp.x = view.lookp.y = view.lookp.z = 0.0;
    view.pos.x = 0.0;                                                         130
    view.pos.y = -8.0;
    view.pos.z = 0.0;
    view.up.x = 0.0;
    view.up.y = 0.0;
    view.up.z = 1.0;
    view.hfov = view.vfov = 45.0;

    screen.height = DEFAULT_HEIGHT;
    screen.width = DEFAULT_WIDTH;
                                                                             140
    root_object = 0;
    background.red = BACKGROUND_RED;
    background.green = BACKGROUND_GREEN;
    background.blue = BACKGROUND_BLUE;
    maxlevel = MAXLEVEL;
    minweight = MINWEIGHT;

    switch (use_scene) {
    case 1:
        scene1();                                                            150
        break;
    case 17:
        scene17();
        break;
    case 21:
        scene21();
        break;
    case 23:
        scene23();
        break;                                                               160
    case 24:
        scene24();
        break;
    case 25:
        scene25();
        break;
    case 26:
```

```
                scene26();
                break;
            case 27:                                                                                          170
                scene27();
                break;
            case 28:
                scene28();
                break;
            case 29:
                scene29();
                break;
            case 32:
                scene32();                                                                                    180
                break;
            case 33:
                scene33();
                break;
            default:
                fprintf(stderr, "Unknown scene: %d\n", use_scene);
                exit(−1);
        }

        vector_sub(&view.lookp, &view.pos, &view.dir);                                                        190
        screen.firstv = view.dir;
        view.lookdist = vector_norm(&view.dir);
        if (vector_norm_cross(&view.dir, &view.up, &screen.scrni) == 0.0)
            fprintf(stderr, "%s: The view and up directions are identical\n",
                    prg_name);
        vector_norm_cross(&screen.scrni, &view.dir, &screen.scrnj);
        magnitude = 2.0 ∗ view.lookdist ∗ tan((double) 0.5∗deg2rad(view.hfov)) /
                    screen.width;
        vector_scale(magnitude, &screen.scrni, &screen.scrnx);
        magnitude = 2.0 ∗ view.lookdist ∗ tan((double) 0.5∗deg2rad(view.vfov)) /                              200
                    screen.height;
        vector_scale(magnitude, &screen.scrnj, &screen.scrny);
        screen.firstv.x −= 0.5∗screen.height∗screen.scrny.x +
                            0.5∗screen.width∗screen.scrnx.x;
        screen.firstv.y −= 0.5∗screen.height∗screen.scrny.y +
                            0.5∗screen.width∗screen.scrnx.y;
        screen.firstv.z −= 0.5∗screen.height∗screen.scrny.z +
                            0.5∗screen.width∗screen.scrnx.z;
}
                                                                                                              210
int bounding_object(ush i)
{
    return (scene[i].tag == BOUNDING_SPHERE);
}


int flat_object(ush i)
{
    return (scene[i].tag == DISC || scene[i].tag == SQUARE);
}
                                                                                                              220
void compute_primary_ray(myfloat x, myfloat y, rayType ∗ray)
{
    ray−>p = view.pos;
    ray−>v.x = screen.firstv.x + x∗screen.scrnx.x + y∗screen.scrny.x;
    ray−>v.y = screen.firstv.y + x∗screen.scrnx.y + y∗screen.scrny.y;
    ray−>v.z = screen.firstv.z + x∗screen.scrnx.z + y∗screen.scrny.z;
    vector_norm(&ray−>v);
}


INLINE_EXTERN myfloat intersection_sphere(sphereType sphere, rayType ray)                                    230
{
    static myfloat x, y, z, b, d, t;

    x = sphere.c.x − ray.p.x;
```

```c
        y = sphere.c.y − ray.p.y;
        z = sphere.c.z − ray.p.z;
        b = x * ray.v.x + y * ray.v.y + z * ray.v.z;
        d = b*b − x*x − y*y − z*z + sphere.r2;
        if (d <= 0.0) return 0.0;
        d = sqrt((double) d);                                                          240
        t = b − d;
        if (t <= MIN_DISTANCE) {
            t = b + d;
            if (t <= MIN_DISTANCE)
                return 0.0;
        }
        return t;
}


INLINE_EXTERN myfloat intersection_plane(vectorType n, myfloat d, rayType ray)        250
{
        static myfloat t;

        t = n.x * ray.v.x + n.y * ray.v.y + n.z * ray.v.z;
        if (t == 0.0) return 0.0;
        t = −(n.x * ray.p.x + n.y * ray.p.y + n.z * ray.p.z − d) / t;
        if (t <= MIN_DISTANCE) return 0.0;
        return t;
}
                                                                                       260
INLINE_EXTERN myfloat intersection_disc(discType disc, rayType ray)
{
        static myfloat a, d, t;

        t = intersection_plane(disc.n, disc.d, ray);
        if (t > MIN_DISTANCE) {
            a = disc.c.x − ray.p.x − t * ray.v.x;
            d = a*a;
            if (d >= disc.r2)
                return 0.0;                                                            270
            a = disc.c.y − ray.p.y − t * ray.v.y;
            d += a*a;
            if (d >= disc.r2)
                return 0.0;
            a = disc.c.z − ray.p.z − t * ray.v.z;
            if (d + a*a >= disc.r2)
                return 0.0;
            else
                return t;
        }                                                                              280
        return 0.0;
}

INLINE_EXTERN myfloat intersection_square(squareType square, rayType ray)
{
        static myfloat x, y, z, a, b, t;

        t = intersection_plane(square.n, square.d, ray);
        if (t > MIN_DISTANCE) {
            x = square.c.x − (ray.p.x + t * ray.v.x);                                  290
            y = square.c.y − (ray.p.y + t * ray.v.y);
            z = square.c.z − (ray.p.z + t * ray.v.z);
            a = x * square.v.x + y * square.v.y + z * square.v.z;
            a = a*a;
            b = x*x + y*y + z*z;
            if ((a < square.r2) && (b − a < square.r2))
                return t;
        }
        return 0.0;
}                                                                                      300
```

34

```
INLINE_EXTERN myfloat intersection(int i, rayType ray)
{
#ifdef COLLECT_STATISTICS
    stat_intersections++;
#endif
    switch(scene[i].tag) {
    case BOUNDING_SPHERE:
    case SPHERE:
        return intersection_sphere(scene[i].u2.sphere, ray);                              310
    case DISC:
        return intersection_disc(scene[i].u2.disc, ray);
    case SQUARE:
        return intersection_square(scene[i].u2.square, ray);
    default:
        fprintf(stderr, "intersection: invalid object tag: %d, object = %d\n",
                scene[i].tag, i);
    }
    return 0.0;
}                                                                                          320

INLINE_EXTERN int intersect_all(rayType ray, double *isect_t)
{
    int n, i;
    double t1;

    n = −1;
    i = 0;
    while(scene[i].tag != NONE) {
        t1 = intersection(i, ray);                                                         330
        if (MIN_DISTANCE < t1 && t1 < *isect_t − MIN_DISTANCE) {
            *isect_t = t1;
            n = i;
        }
        i += 1;
    }
    return n;
}

void trace(int level, myfloat weight, rayType ray, colorType *color)                       340
{
    intersectionType isect;

    isect.t = 1.0e+20;
    isect.object = intersect_all(ray, &isect.t);
#ifdef DEBUG_INTERSECT
    fprintf(stderr, "intersect all returns (%d,%f)\n", isect.object,
            isect.t);
#endif
    if (isect.object >= 0)                                                                 350
        shade(level, weight, ray, isect, color);
    else
        background_color(color);
}

void shade(int level,
           myfloat weight,
           rayType ray,
           intersectionType isect,
           colorType *color)                                                               360
{
    int n;
    surfaceInfoType sinfo;
    vectorType N, L;
    colorType diffuse, specular;
    rayType tray;
    myfloat n_dot_l, n_dot_v, distance, a, u, v;
    colorType tcol;
```

35

```
#ifdef DEBUG1
    fprintf(stderr, "shade: level = %d, weight = %f, object = %d, dir = (%f, %f, %f)\n",
            level, weight, isect.object, ray.v.x, ray.v.y, ray.v.z);
#endif

    color->red = 0.0;
    color->green = 0.0;
    color->blue = 0.0;

    isect.p.x = ray.p.x + isect.t * ray.v.x;
    isect.p.y = ray.p.y + isect.t * ray.v.y;
    isect.p.z = ray.p.z + isect.t * ray.v.z;

    n = scene[isect.object].u1.surface;
    while (surface[n].tag != SIMPLE) {
        switch (surface[n].tag) {
        case CHECKED:
            calculate_uv(isect, &u, &v);
            while (u < 0)
                u += 100.0*surface[n].u.checked.checksize;
            while (v < 0)
                v += 100.0*surface[n].u.checked.checksize;
            if (fmod(u, surface[n].u.checked.dbl_checksize) <
                surface[n].u.checked.checksize) {
                if (fmod(v, surface[n].u.checked.dbl_checksize) <
                    surface[n].u.checked.checksize)
                    n = surface[n].u.checked.s1;
                else
                    n = surface[n].u.checked.s2;
            }
            else {
                if (fmod(v, surface[n].u.checked.dbl_checksize) <
                    surface[n].u.checked.checksize)
                    n = surface[n].u.checked.s2;
                else
                    n = surface[n].u.checked.s1;
            }
            break;
        default:
            fprintf(stderr, "%s: no such surface %d\n", prg_name, n);
            exit(-1);
        }
    }

    sinfo = surface[n].u.sinfo;

    /* isect.p is needed to compute normal for spheres */

    compute_normal(isect, &N);

    n_dot_v = - vector_dot(&N, &ray.v);

    /* Flip normal if the object has no inside (if it's flat) */

    if (flat_object(isect.object)) {
        if (n_dot_v < 0.0) {
            N.x = -N.x;
            N.y = -N.y;
            N.z = -N.z;
            n_dot_v = - n_dot_v;
        }
        isect.enter = TRUE;
    }
    else {
        if (n_dot_v < 0.0)
            isect.enter = FALSE;
```

36

```
            else
                isect.enter = TRUE;
        }

#ifdef DEBUG1                                                                                          440
        fprintf(stderr, "ray = (%f, %f, %f)\nnml = (%f, %f, %f)\n",
                ray.v.x, ray.v.y, ray.v.z, N.x, N.y, N.z);
#endif

        /* if we are leaving an object, only transmitted light contributes */

        if (!isect.enter) {
            if (level < maxlevel && sinfo.transparency * weight > minweight) {
                tray.p = isect.p;
                if (transmission_direction(n_dot_v, sinfo.refrindex, &ray.v,       450
                                            &N, &tray.v)) {
                    trace(level + 1, sinfo.transparency * weight, tray, &tcol);
                    color->red   = sinfo.transparency * tcol.red;
                    color->green = sinfo.transparency * tcol.green;
                    color->blue  = sinfo.transparency * tcol.blue;
#ifdef DEBUG_TRANS
                    fprintf(stderr, "shade: Leaving object: transmitted light = (%f, %f, %f)\n", color->red, color->green, color->blu
#endif
                }
            }                                                                                          460
#ifdef LIGHT_FALLOFF
            light_falloff(color, isect.t);
#endif
#ifdef DEBUG
            fprintf(stderr, "shade returns (%f, %f, %f)\n",
                    color->red, color->green, color->blue);
#endif
            return;
        }
                                                                                                       470
        /* Ambient Light */

        color->red   = sinfo.ambient.red;
        color->green = sinfo.ambient.green;
        color->blue  = sinfo.ambient.blue;

        /* Diffuse light and specular reflectance */

        diffuse.red = diffuse.green = diffuse.blue = 0.0;
        specular.red = specular.green = specular.blue = 0.0;                                            480
        for(n = 0; n <= last_light; n++) {
            vector_sub(&isect.p, &light[n].p, &L);
            distance = vector_norm(&L);
            n_dot_l = vector_dot(&N, &L);
            if (n_dot_l < 0.0) {
                tray.p = light[n].p;
                tray.v = L;
#ifdef COLLECT_STATISTICS
                stat_shadow++;
#endif                                                                                                 490
                if (intersect_all(tray, &distance) == −1) {
                    diffuse.red   += light[n].c.red * sinfo.diffuse.red;
                    diffuse.green += light[n].c.green * sinfo.diffuse.green;
                    diffuse.blue  += light[n].c.blue * sinfo.diffuse.blue;
                    if (sinfo.specpow != 0.0) {
                        a = − n_dot_l + n_dot_v;
                        a = a*a / (2.0 * (1.0 + ray.v.x * L.x + ray.v.y * L.y +
                                            ray.v.z * L.z));
#ifdef DEBUG_HIGHLIGHT
                        fprintf(stderr, "shade: a = %f, specpow = %d\n",             500
                                a, sinfo.specpow);
#endif
```

37

```c
                    a = mypower(a, sinfo.specpow);
#ifdef DEBUG_HIGHLIGHT
                    fprintf(stderr, "shade: a = %f\n", a);
#endif
                    specular.red   += a * light[n].c.red;
                    specular.green += a * light[n].c.green;
                    specular.blue  += a * light[n].c.blue;
                }                                                                          510
            }
        }
    }

#ifdef DEBUG
    fprintf(stderr, "shade: diffuse: (%f, %f, %f)\n",
            diffuse.red, diffuse.green, diffuse.blue);
    fprintf(stderr, "shade: shadow rays: (%f, %f, %f)\n",
            specular.red, specular.green, specular.blue);
#endif                                                                                     520

    color->red   += diffuse.red + sinfo.reflectivity * specular.red;
    color->green += diffuse.green + sinfo.reflectivity * specular.green;
    color->blue  += diffuse.blue + sinfo.reflectivity * specular.blue;

    if (level < maxlevel) {
        tray.p = isect.p;

        /* Specular reflection */
        if (sinfo.reflectivity * weight > minweight) {                                     530
            specular_direction(n_dot_v, &ray.v, &N, &tray.v);
            trace(level + 1, sinfo.reflectivity * weight, tray, &tcol);
            color->red   += sinfo.reflectivity * tcol.red;
            color->green += sinfo.reflectivity * tcol.green;
            color->blue  += sinfo.reflectivity * tcol.blue;
#ifdef DEBUG_SPECULAR
            fprintf(stderr, "shade: reflected light: (%f, %f, %f)\n",
                    tcol.red, tcol.green, tcol.blue);
#endif
        }                                                                                  540

        /* Transmission ray */
        if (sinfo.transparency * weight > minweight) {
            ray.p = isect.p;
            if (flat_object(isect.object)) {
                trace(level + 1, sinfo.transparency * weight, ray, &tcol);
                color->red   += sinfo.transparency * tcol.red;
                color->green += sinfo.transparency * tcol.green;
                color->blue  += sinfo.transparency * tcol.blue;
#ifdef DEBUG_TRANS                                                                         550
                fprintf(stderr, "shade: transmitted light: (%f, %f, %f)\n",
                        tcol.red, tcol.green, tcol.blue);
#endif
            }
            else {
                if (transmission_direction(n_dot_v, sinfo.refrindex,
                                           &ray.v, &N, &tray.v)) {
                    trace(level + 1, sinfo.transparency * weight, tray, &tcol);
                    color->red   += sinfo.transparency * tcol.red;
                    color->green += sinfo.transparency * tcol.green;                       560
                    color->blue  += sinfo.transparency * tcol.blue;
#ifdef DEBUG_TRANS
                    fprintf(stderr, "shade: transmitted light: (%f, %f, %f)\n",
                            tcol.red, tcol.green, tcol.blue);
#endif
                }
            }
        }
    }
```

38

```c
#ifdef LIGHT_FALLOFF
    light_falloff(color, isect.t);
#endif
#ifdef DEBUG
    fprintf(stderr, "shade returns (%f, %f, %f)\n",
            color->red, color->green, color->blue);
#endif
}


void background_color(colorType *color)
{
    color->red = background.red;
    color->green = background.green;
    color->blue = background.blue;
}


void compute_normal(intersectionType isect, vectorType *N)
{
    switch (scene[isect.object].tag) {
    case SPHERE:
        compute_normal_sphere(isect, N);
        return;
    case DISC:
        vector_copy(&scene[isect.object].u2.disc.n, N);
        return;
    case SQUARE:
        vector_copy(&scene[isect.object].u2.square.n, N);
        return;
    default:
        fprintf(stderr, "compute_normal: invalid object tag: %d, object=%d\n",
                scene[isect.object].tag, isect.object);
    }
}


void compute_normal_sphere(intersectionType isect, vectorType *N)
{
    sphereType *sphere;

    sphere = &scene[isect.object].u2.sphere;
    N->x = (isect.p.x - sphere->c.x) / sphere->r;
    N->y = (isect.p.y - sphere->c.y) / sphere->r;
    N->z = (isect.p.z - sphere->c.z) / sphere->r;
}


/*
 * specular_direction: compute specular vector in specular direction
 * n_dot_v: dot product of N and V
 * V: view vector
 * N: normal vector
 * S: specular vector (result)
 * All vectors are normalized
 */
void specular_direction(myfloat n_dot_v,
                        vectorType *V,
                        vectorType *N,
                        vectorType *S)
{
#ifdef COLLECT_STATISTICS
    stat_reflected++;
#endif
    S->x = 2.0 * n_dot_v * N->x + V->x;
    S->y = 2.0 * n_dot_v * N->y + V->y;
    S->z = 2.0 * n_dot_v * N->z + V->z;
#ifdef DEBUG_SPECULAR
    fprintf(stderr, "specular direction = (%f,%f,%f)\n", S->x, S->y, S->z);
#endif
}
```

```c
/*
 * transmission_direction: compute transmission vector
 * n_dot_v: dot product of N and V
 * refrindex: refraction index
 * V: view vector
 * N: normal vector
 * T: transmission vector (result)
 * returns: FALSE if no transmission occurs, TRUE otherwise
 *
 * If the ray is entering an object (n_dot_v > 0.0), it's supposed that the
 * ray is leaving air. Otherwise it is supposed that the ray is entering air.
 * All vectors are normalized.
 */
int transmission_direction(myfloat n_dot_v,
                           myfloat r,
                           vectorType *V,
                           vectorType *N,
                           vectorType *T)
{
    myfloat c2, c3;

#ifdef COLLECT_STATISTICS
    stat_refracted++;
#endif
    if (n_dot_v > 0.0)
        r = 1.0/r;

    c2 = r*r*(1.0 - n_dot_v * n_dot_v);
#ifdef DEBUG_TRANS
    if (c2 > 1.0)
        fprintf(stderr, "transmission direction: c2 = %f > 1.0\n", c2);
#endif
    if (c2 > 1.0)
        return FALSE;
    c2 = sqrt(1.0 - c2);
    if (n_dot_v < 0.0)
        c2 = -c2;
    c3 = r * n_dot_v - c2;
    T->x = c3 * N->x + r * V->x;
    T->y = c3 * N->y + r * V->y;
    T->z = c3 * N->z + r * V->z;
#ifdef DEBUG_TRANS
    fprintf(stderr, "TRANS: r = %f, c1 = %f, c2 = %f, r*c1 - c2 = %f\n",
            r, n_dot_v, c2, c3);
    fprintf(stderr, "trn = (%f, %f, %f)\n", T->x, T->y, T->z);
#endif
    return TRUE;
}

/*
 * shadow: returns true if the ray doesn't hit any objects for t < tmax
 * false othwerwise
 */
/*
int shadow(rayType ray, myfloat tmax)
{
    intersectionType isect;

    isect.t = tmax;
    intersect_all(root_object, ray, &isect);
    return (isect.object == -1);
}
*/

void calculate_uv(intersectionType isect, myfloat *u, myfloat *v)
{
```

```
        switch (scene[isect.object].tag) {
        case SQUARE:
        case DISC:
            calculate_uv_square(isect, u, v);
            break;
        default:
            fprintf(stderr, "%s: unknow object tag %d\n",                                              710
                    prg_name, scene[isect.object].tag);
            exit(−1);
        }
}

void calculate_uv_square(intersectionType isect, myfloat ∗u, myfloat ∗v)
{
        squareType ∗square;

        square = &scene[isect.object].u2.square;                                                      720

        ∗u = isect.p.x∗square−>v.x + isect.p.y∗square−>v.y + isect.p.z∗square−>v.z;
        ∗v = sqrt(isect.p.x ∗ isect.p.x + isect.p.y ∗ isect.p.y +
                  isect.p.z ∗ isect.p.z − (∗u) ∗ (∗u));
}
```

A.2.5 `scenes.c`

```
/*
 ∗ Author:        Peter Holst Andersen
 ∗ Last change:   December, 1994
 ∗ Contents:      Scenes
 */

#ifndef _CMIX
#define return_void return
#endif
                                                                                                      10
void scene1();
void scene17();
void scene21();
void scene22();
void scene23();
void scene24();
void scene25();
void scene26();
void scene27();
void scene28();                                                                                       20
void scene29();
void scene32();

void scene1()
{
        create_simple_surface(0, 0.2, 0.2, 0.5, 0.0, 0.0, 5);
        create_sphere(&scene[0], 0.0, 0.0, 0.0, 2.0, 0);
        create_disc(&scene[1], 0.0, 0.0, −2.5, 0.0, 0.0, 1.0, 1.5, 0);
        create_square(&scene[2], 0.0, 0.0, 2.5, 0.0, 0.0, 1.0,
                      1.0, 0.0, 0.0, 1.5, 0);                                                          30
        scene[3].tag = NONE;
        last_object = 2;
        create_light(&light[0], 1.0, 1.0, 1.0, 8.0, −8.0, 8.0);
        last_light = 0;
        return_void;
}

/* Scene 1 in the report */
void scene17()
{                                                                                                     40
```

```c
        create_simple_surface(0, 0.2, 0.2, 0.5, 0.0, 0.0, 5);
        create_simple_surface(1, 0.0, 0.0, 0.0, 0.0, 0.0, 1);
        create_checked_surface(2, 0, 1, 1.5);
        create_simple_surface(3, 0.1, 0.1, 0.7, 0.0, 0.0, 15);
        create_simple_surface(4, 0.1, 0.1, 0.1, 0.8, 1.3, 15);
        create_sphere(&scene[0], 0.0, 4.0, 5.0, 2.5, 3);
        create_sphere(&scene[1], 0.7, −1.0, 4.0, 1.6, 4);
        create_square(&scene[2], 0.0, 0.0, 0.0, 0.0, 0.0, 1.0,
                    1.0, 0.0, 0.0, 20.0, 2);
        scene[3].tag = NONE;
        last_object = 2;
        create_light(&light[0], 1.0, 1.0, 1.0, 8.0, −8.0, 24.0);
        last_light = 0;
        view.pos.x = 0.0;
        view.pos.y = −8.0;
        view.pos.z = 6.0;
        view.lookp.x = 0.0;
        view.lookp.y = 0.0;
        view.lookp.z = 3.0;
        screen.width = 512;
        screen.height = 512;
        return_void;
}

/* Scene 5 in the report */
void scene21()
{
        double sqrt3, x, y, z;
        int i;

        sqrt3 = sqrt(3.0);
        i = 0;

        create_simple_surface(0, 0.01, 0.2, 0.1, 0.9, 1.3, 6);
        create_simple_surface(1, 0.0, 0.0, 0.0, 0.0, 0.0, 1);
        create_simple_surface(2, 0.2, 0.2, 0.5, 0.0, 0.0, 6);
        create_checked_surface(3, 2, 1, 0.8);

        create_sphere(&scene[i++], 0.0, 0.0, 0.0, 1.0, 0);
        create_sphere(&scene[i++], −1.0, sqrt3, 0.0, 1.0, 0);
        create_sphere(&scene[i++],  1.0, sqrt3, 0.0, 1.0, 0);
        create_sphere(&scene[i++], −2.0, 2.0*sqrt3, 0.0, 1.0, 0);
        create_sphere(&scene[i++],  0.0, 2.0*sqrt3, 0.0, 1.0, 0);
        create_sphere(&scene[i++],  2.0, 2.0*sqrt3, 0.0, 1.0, 0);
        create_sphere(&scene[i++], −3.0, 3.0*sqrt3, 0.0, 1.0, 0);
        create_sphere(&scene[i++], −1.0, 3.0*sqrt3, 0.0, 1.0, 0);
        create_sphere(&scene[i++],  1.0, 3.0*sqrt3, 0.0, 1.0, 0);
        create_sphere(&scene[i++],  3.0, 3.0*sqrt3, 0.0, 1.0, 0);
        create_sphere(&scene[i++], −4.0, 4.0*sqrt3, 0.0, 1.0, 0);
        create_sphere(&scene[i++], −2.0, 4.0*sqrt3, 0.0, 1.0, 0);
        create_sphere(&scene[i++],  0.0, 4.0*sqrt3, 0.0, 1.0, 0);
        create_sphere(&scene[i++],  2.0, 4.0*sqrt3, 0.0, 1.0, 0);
        create_sphere(&scene[i++],  4.0, 4.0*sqrt3, 0.0, 1.0, 0);
        z = sqrt(8.0/3.0);
        y = 2.0 / sqrt3;
        create_sphere(&scene[i++],  0.0, y, z, 1.0, 0);
        create_sphere(&scene[i++], −1.0, y+sqrt3, z, 1.0, 0);
        create_sphere(&scene[i++],  1.0, y+sqrt3, z, 1.0, 0);
        create_sphere(&scene[i++], −2.0, y+2.0*sqrt3, z, 1.0, 0);
        create_sphere(&scene[i++],  0.0, y+2.0*sqrt3, z, 1.0, 0);
        create_sphere(&scene[i++],  2.0, y+2.0*sqrt3, z, 1.0, 0);
        create_sphere(&scene[i++], −3.0, y+3.0*sqrt3, z, 1.0, 0);
        create_sphere(&scene[i++], −1.0, y+3.0*sqrt3, z, 1.0, 0);
        create_sphere(&scene[i++],  1.0, y+3.0*sqrt3, z, 1.0, 0);
        create_sphere(&scene[i++],  3.0, y+3.0*sqrt3, z, 1.0, 0);
        z = 2.0 * sqrt(8.0/3.0);
        y = 4.0 / sqrt3;
```

42

```
        create_sphere(&scene[i++],   0.0, y, z, 1.0, 0);
        create_sphere(&scene[i++],  −1.0, y+sqrt3, z, 1.0, 0);
        create_sphere(&scene[i++],   1.0, y+sqrt3, z, 1.0, 0);                                    110
        create_sphere(&scene[i++],  −2.0, y+2.0*sqrt3, z, 1.0, 0);
        create_sphere(&scene[i++],   0.0, y+2.0*sqrt3, z, 1.0, 0);
        create_sphere(&scene[i++],   2.0, y+2.0*sqrt3, z, 1.0, 0);
        z = 3.0 * sqrt(8.0/3.0);
        y = 6.0 / sqrt3;
        create_sphere(&scene[i++],   0.0, y, z, 1.0, 0);
        create_sphere(&scene[i++],  −1.0, y+sqrt3, z, 1.0, 0);
        create_sphere(&scene[i++],   1.0, y+sqrt3, z, 1.0, 0);

        create_sphere(&scene[i++],   0.0, 8.0 / sqrt3, 4.0 * sqrt(8.0/3.0), 1.0, 0);               120

        create_square(&scene[i++], 0.0, 0.0, −1.0, 0.0, 0.0, 1.0,
                                   1.0, 0.0, 0.0, 40.0, 3);

        scene[i].tag = NONE;
        last_object = i−1;

        create_light(&light[0], 1.0, 1.0, 1.0, 5.0, −20.0, 20.0);
        last_light = 0;
                                                                                                  130
        view.lookp.x = 0.0;
        view.lookp.y = 2.0 * sqrt3;
        view.lookp.z = 2.75;

        view.pos.x = 3.0;
        view.pos.y = −10.0;
        view.pos.z = 6.0;

        screen.width = 512;
        screen.height = 512;                                                                      140
        return_void;
}

/* Scene 6 in the report */
void scene23()
{
        create_simple_surface(0, 0.2, 0.2, 0.5, 0.0, 0.0, 5);
        create_sphere(&scene[0], 0.0, 0.0, 0.0, 1.2, 0);
        create_sphere(&scene[1], 0.0, −2.0, −0.5, 0.5, 0);
        create_sphere(&scene[2], 3.0, 0.0, 0.0, 1.2, 0);                                          150
        create_sphere(&scene[3], −3.0, 0.0, 0.0, 1.2, 0);
        create_sphere(&scene[4], 0.0, 0.0, −8.0, 5.0, 0);
        scene[5].tag = NONE;
        last_object = 4;
        create_light(&light[0], 0.0, 0.4, 0.4, −40.0, −20.0, 20.0);
        create_light(&light[1], 0.6, 0.0, 0.0, −20.0, −20.0, 20.0);
        create_light(&light[2], 0.0, 0.6, 0.0, 0.0, −20.0, 20.0);
        create_light(&light[3], 0.0, 0.0, 0.6, 20.0, −20.0, 20.0);
        create_light(&light[4], 0.4, 0.4, 0.0, 40.0, −20.0, 20.0);
        last_light = 4;                                                                           160
        screen.height = 512;
        screen.width = 512;
        return_void;
}

/* Scene 7 in the report */
void scene24()
{
        create_simple_surface(0, 0.2, 0.2, 0.5, 0.0, 0.0, 5);
        create_disc(&scene[0], 0.0, 0.0, −3.0, 0.0, 0.0, 1.0, 2.0, 0);                             170
        create_disc(&scene[1], 0.0, 0.0, −2.0, 0.0, 0.2, 0.7, 2.0, 0);
        create_disc(&scene[2], 0.0, 0.0, −1.0, 0.0, 0.2, 0.6, 2.0, 0);
        create_disc(&scene[3], 0.0, 0.0,  1.0, 0.0, 0.2, 0.5, 2.0, 0);
        create_disc(&scene[4], 0.0, 0.0,  3.0, 0.0, 0.1, 0.2, 2.0, 0);
```

```
    scene[5].tag = NONE;
    last_object = 4;
    create_light(&light[0], 0.0, 0.4, 0.4, −40.0, −20.0, 20.0);
    create_light(&light[1], 0.6, 0.0, 0.0, −20.0, −20.0, 20.0);
    create_light(&light[2], 0.0, 0.6, 0.0, 0.0, −20.0, 20.0);
    create_light(&light[3], 0.0, 0.0, 0.6, 20.0, −20.0, 20.0);                          180
    create_light(&light[4], 0.4, 0.4, 0.0, 40.0, −20.0, 20.0);
    last_light = 4;
    screen.width = 512;
    screen.height = 512;
    return_void;
}

/∗ Scene 8 in the report ∗/
void scene25()
{                                                                                     190
    create_simple_surface(0, 0.2, 0.2, 0.5, 0.0, 0.0, 5);
    create_square(&scene[0],0.0,0.0,−4.0, 0.0,0.0,1.0, 1.0,0.0,0.0, 2.0, 0);
    create_square(&scene[1],0.0,0.0,−2.0, 0.0,0.2,0.8, 1.0,0.0,0.0, 2.0, 0);
    create_square(&scene[2],0.0,0.0, 0.0, 0.0,0.4,0.6, 1.0,0.0,0.0, 2.0, 0);
    create_square(&scene[3],0.0,0.0, 2.0, 0.0,0.6,0.4, 1.0,0.0,0.0, 2.0, 0);
    create_square(&scene[4],0.0,0.0, 4.0, 0.0,0.8,0.2, 1.0,0.0,0.0, 2.0, 0);
    scene[5].tag = NONE;
    last_object = 4;
    create_light(&light[0], 0.0, 0.4, 0.4, −40.0, −20.0, 20.0);
    create_light(&light[1], 0.6, 0.0, 0.0, −20.0, −20.0, 20.0);                          200
    create_light(&light[2], 0.0, 0.6, 0.0, 0.0, −20.0, 20.0);
    create_light(&light[3], 0.0, 0.0, 0.6, 20.0, −20.0, 20.0);
    create_light(&light[4], 0.4, 0.4, 0.0, 40.0, −20.0, 20.0);
    last_light = 4;
    screen.width = 512;
    screen.height = 512;
    return_void;
}

/∗ Scene 2 in the report ∗/                                                            210
void scene26()
{
    /∗ Scene with 5 spheres and 1 light source ∗/

    create_simple_surface(0, 0.2, 0.2, 0.5, 0.0, 0.0, 5);
    create_sphere(&scene[0], 0.0, 0.0, 0.0, 1.2, 0);
    create_sphere(&scene[1], 0.0, −2.0, −0.5, 0.5, 0);
    create_sphere(&scene[2], 3.0, 0.0, 0.0, 1.2, 0);
    create_sphere(&scene[3], −3.0, 0.0, 0.0, 1.2, 0);
    create_sphere(&scene[4], 0.0, 0.0, −8.0, 5.0, 0);                                    220
    scene[5].tag = NONE;
    last_object = 4;
    create_light(&light[0], 1.0, 1.0, 1.0, −40.0, −20.0, 20.0);
    last_light = 0;
    screen.height = 512;
    screen.width = 512;
    return_void;
}

/∗ Scene 3 in the report ∗/                                                            230
void scene27()
{
    /∗ Scene with 5 discs and 1 lightsource ∗/

    create_simple_surface(0, 0.2, 0.2, 0.5, 0.0, 0.0, 5);
    create_disc(&scene[0], 0.0, 0.0, −3.0, 0.0, 0.0, 1.0, 2.0, 0);
    create_disc(&scene[1], 0.0, 0.0, −2.0, 0.0, 0.2, 0.7, 2.0, 0);
    create_disc(&scene[2], 0.0, 0.0, −1.0, 0.0, 0.2, 0.6, 2.0, 0);
    create_disc(&scene[3], 0.0, 0.0,  1.0, 0.0, 0.2, 0.5, 2.0, 0);
    create_disc(&scene[4], 0.0, 0.0,  3.0, 0.0, 0.1, 0.2, 2.0, 0);                       240
    scene[5].tag = NONE;
```

```
    last_object = 4;
    create_light(&light[0], 1.0, 1.0, 1.0, −40.0, −20.0, 20.0);
    last_light = 0;
    screen.height = 512;
    screen.width = 512;
    return_void;
}

/* Scene 4 in the report */                                                                        250
void scene28()
{
    /* Scene with 5 squares and 1 lightsource */

    create_simple_surface(0, 0.2, 0.2, 0.5, 0.0, 0.0, 5);
    create_square(&scene[0],0.0,0.0,−4.0, 0.0,0.0,1.0, 1.0,0.0,0.0, 2.0, 0);
    create_square(&scene[1],0.0,0.0,−2.0, 0.0,0.2,0.8, 1.0,0.0,0.0, 2.0, 0);
    create_square(&scene[2],0.0,0.0, 0.0, 0.0,0.4,0.6, 1.0,0.0,0.0, 2.0, 0);
    create_square(&scene[3],0.0,0.0, 2.0, 0.0,0.6,0.4, 1.0,0.0,0.0, 2.0, 0);
    create_square(&scene[4],0.0,0.0, 4.0, 0.0,0.8,0.2, 1.0,0.0,0.0, 2.0, 0);        260
    scene[5].tag = NONE;
    last_object = 4;
    create_light(&light[0], 1.0, 1.0, 1.0, −40.0, −20.0, 20.0);
    last_light = 0;
    screen.height = 512;
    screen.width = 512;
    return_void;
}

void scene29()                                                                                     270
{
    /* An open box (5 squares) lighted by 5 lightsources */

    screen.height = 512;
    screen.width = 512;
    create_simple_surface(0, 0.1, 0.1, 0.5, 0.0, 0.0, 6);
    create_simple_surface(1, 0.0, 0.0, 0.0, 0.0, 0.0, 1);
    create_simple_surface(2, 0.1, 0.1, 0.5, 0.0, 0.0, 6);
    create_checked_surface(3, 2, 1, 0.5);
    create_square(&scene[0], 0.0, 0.0, 0.0, 0.0,−1.0,0.0, 1.0,0.0,0.0, 2.0, 3);     280
    create_square(&scene[1], 0.0,−2.0, 2.0, 0.0, 0.0,1.0, 1.0,0.0,0.0, 2.0, 0);
    create_square(&scene[2], 0.0,−2.0,−2.0, 0.0, 0.0,1.0, 1.0,0.0,0.0, 2.0, 0);
    create_square(&scene[3], 2.0,−2.0, 0.0, 1.0, 0.0,0.0, 0.0,1.0,0.0, 2.0, 0);
    create_square(&scene[4],−2.0,−2.0, 0.0, 1.0, 0.0,0.0, 0.0,1.0,0.0, 2.0, 0);
    scene[5].tag = NONE;
    last_object = 4;
    create_light(&light[0], 1.0, 1.0, 1.0, 0.0, −20.0, 0.0);
    create_light(&light[1], 0.0, 0.4, 0.4, 0.0, −20.0, 20.0);
    create_light(&light[2], 0.6, 0.0, 0.0, 0.0, −20.0, −20.0);
    create_light(&light[3], 0.0, 0.6, 0.0, 20.0, −20.0, 0.0);                        290
    create_light(&light[4], 0.0, 0.0, 0.6, −20.0, −20.0, 0.0);

    last_light = 4;

    view.pos.x = 6.0;
    view.pos.y = −13.0;
    view.pos.z = 6.0;
    return_void;
}
                                                                                                   300
void scene32()
{
    screen.height = 512;
    screen.width = 512;
    create_simple_surface(0, 0.1, 0.1, 0.6, 0.0, 0.0, 1000);
    create_simple_surface(1, 0.0, 0.0, 0.1, 0.0, 0.0, 1000);
    create_checked_surface(2, 0, 1, 1.5);
    create_simple_surface(3, 0.01, 0.1, 0.1, 0.9, 1.3, 500);
```

45

```
    create_simple_surface(4,   0.1, 0.05, 0.8, 0.0, 0.0, 500);
    create_square(&scene[0],   0.0,0.0,0.0, 0.0,1.0,0.0, 1.0,0.0,0.0, 20.0, 2);          310
    create_sphere(&scene[1],  −3.0, 2.0, 10.0, 1.2, 4);
    create_sphere(&scene[2],   0.0, 2.0, 10.0, 1.2, 3);
    create_sphere(&scene[3],   3.0, 2.0, 10.0, 1.2, 4);
    scene[4].tag = NONE;
    last_object = 3;

    create_light(&light[0], 0.4, 0.4, 0.0,    0.0, 30.0, −10.0);
    create_light(&light[1], 0.0, 0.4, 0.4, −20.0, 30.0,  10.0);
    create_light(&light[2], 0.6, 0.0, 0.0, −14.0, 30.0,  −4.0);
    create_light(&light[3], 0.0, 0.6, 0.0,  14.0, 30.0,  −4.0);          320
    create_light(&light[4], 0.0, 0.0, 0.6,  20.0, 30.0,  10.0);

    last_light = 4;

    view.lookp.x = 0.0;
    view.lookp.y = 2.0;
    view.lookp.z = 10.0;

    view.pos.x = 4.0;
    view.pos.y = 5.0;          330
    view.pos.z = 3.0;

    view.up.x = 0.0;
    view.up.y = 1.0;
    view.up.z = 0.0;

    return_void;
}

/∗ Scene 9 in the report ∗/                                             340
void scene33()
{
    double sqrt3, x, y, z;
    int i;

    sqrt3 = sqrt(3.0);
    i = 0;

    create_simple_surface(0, 0.01, 0.2, 0.1, 0.9, 1.3, 6);
    create_simple_surface(1, 0.0, 0.0, 0.0, 0.0, 0.0, 1);          350
    create_simple_surface(2, 0.2, 0.2, 0.5, 0.0, 0.0, 6);
    create_checked_surface(3, 2, 1, 0.8);

    create_sphere(&scene[i++],  0.0, 0.0, 0.0, 1.0, 0);
    create_sphere(&scene[i++], −1.0, sqrt3, 0.0, 1.0, 0);
    create_sphere(&scene[i++],  1.0, sqrt3, 0.0, 1.0, 0);
    create_sphere(&scene[i++], −2.0, 2.0∗sqrt3, 0.0, 1.0, 0);
    create_sphere(&scene[i++],  0.0, 2.0∗sqrt3, 0.0, 1.0, 0);
    create_sphere(&scene[i++],  2.0, 2.0∗sqrt3, 0.0, 1.0, 0);
    create_sphere(&scene[i++], −3.0, 3.0∗sqrt3, 0.0, 1.0, 0);          360
    create_sphere(&scene[i++], −1.0, 3.0∗sqrt3, 0.0, 1.0, 0);
    create_sphere(&scene[i++],  1.0, 3.0∗sqrt3, 0.0, 1.0, 0);
    create_sphere(&scene[i++],  3.0, 3.0∗sqrt3, 0.0, 1.0, 0);
    create_sphere(&scene[i++], −4.0, 4.0∗sqrt3, 0.0, 1.0, 0);
    create_sphere(&scene[i++], −2.0, 4.0∗sqrt3, 0.0, 1.0, 0);
    create_sphere(&scene[i++],  0.0, 4.0∗sqrt3, 0.0, 1.0, 0);
    create_sphere(&scene[i++],  2.0, 4.0∗sqrt3, 0.0, 1.0, 0);
    create_sphere(&scene[i++],  4.0, 4.0∗sqrt3, 0.0, 1.0, 0);
    z = sqrt(8.0/3.0);
    y = 2.0 / sqrt3;          370
    create_sphere(&scene[i++],  0.0, y, z, 1.0, 0);
    create_sphere(&scene[i++], −1.0, y+sqrt3, z, 1.0, 0);
    create_sphere(&scene[i++],  1.0, y+sqrt3, z, 1.0, 0);
    create_sphere(&scene[i++], −2.0, y+2.0∗sqrt3, z, 1.0, 0);
    create_sphere(&scene[i++],  0.0, y+2.0∗sqrt3, z, 1.0, 0);
```

```
        create_sphere(&scene[i++],   2.0, y+2.0*sqrt3, z, 1.0, 0);
        create_sphere(&scene[i++],  −3.0, y+3.0*sqrt3, z, 1.0, 0);
        create_sphere(&scene[i++],  −1.0, y+3.0*sqrt3, z, 1.0, 0);
        create_sphere(&scene[i++],   1.0, y+3.0*sqrt3, z, 1.0, 0);
        create_sphere(&scene[i++],   3.0, y+3.0*sqrt3, z, 1.0, 0);         380
        z = 2.0 * sqrt(8.0/3.0);
        y = 4.0 / sqrt3;
        create_sphere(&scene[i++],   0.0, y, z, 1.0, 0);
        create_sphere(&scene[i++],  −1.0, y+sqrt3, z, 1.0, 0);
        create_sphere(&scene[i++],   1.0, y+sqrt3, z, 1.0, 0);
        create_sphere(&scene[i++],  −2.0, y+2.0*sqrt3, z, 1.0, 0);
        create_sphere(&scene[i++],   0.0, y+2.0*sqrt3, z, 1.0, 0);
        create_sphere(&scene[i++],   2.0, y+2.0*sqrt3, z, 1.0, 0);
        z = 3.0 * sqrt(8.0/3.0);
        y = 6.0 / sqrt3;                                                    390
        create_sphere(&scene[i++],   0.0, y, z, 1.0, 0);
        create_sphere(&scene[i++],  −1.0, y+sqrt3, z, 1.0, 0);
        create_sphere(&scene[i++],   1.0, y+sqrt3, z, 1.0, 0);

        create_sphere(&scene[i++],   0.0, 8.0 / sqrt3, 4.0 * sqrt(8.0/3.0), 1.0, 0);

        create_square(&scene[i++], 0.0, 0.0, −1.0, 0.0, 0.0, 1.0,
                                   1.0, 0.0, 0.0, 40.0, 3);

        scene[i].tag = NONE;                                               400
        last_object = i−1;

        create_light(&light[0], 0.0, 0.4, 0.4, −40.0, −20.0, 20.0);
        create_light(&light[1], 0.6, 0.0, 0.0, −20.0, −20.0, 20.0);
        create_light(&light[2], 0.0, 0.6, 0.0, 0.0, −20.0, 20.0);
        create_light(&light[3], 0.0, 0.0, 0.6, 20.0, −20.0, 20.0);
        create_light(&light[4], 0.4, 0.4, 0.0, 40.0, −20.0, 20.0);
        last_light = 4;

        view.lookp.x = 0.0;                                               410
        view.lookp.y = 2.0 * sqrt3;
        view.lookp.z = 2.75;

        view.pos.x = 3.0;
        view.pos.y = −10.0;
        view.pos.z = 6.0;

        screen.width = 512;
        screen.height = 512;
        return_void;                                                     420
}
```

A.2.6 `srgp.c`

```
/*
 * Author:        Peter Holst Andersen
 * Last change:   August, 1994
 * Contents:      Functions for showing the image using SRGP
 *                (Simple Raster Graphics Package).
 */

#ifdef USE_SRGP

#include <stdio.h>                                                        10
#include <stdlib.h>
#include <srgp.h>
#include "ray.h"

int init_srgp(char *title, int height, int width)
{
```

47

```c
    int i, j, m;
    unsigned short k;

    SRGP_begin(title, width, height, 0, FALSE);                                    20
    SRGP_refresh();
    i = SRGP_inquireCanvasDepth();
    j = (1 << i);
#ifdef DEBUG
    fprintf(stderr, "init_srgp: canvas depth: %d\n", j);
#endif
    for(m = 0; m < j - 2; m++) {
        k = 256*256*m/j;
        SRGP_loadColorTable(m + 2, 1, &k, &k, &k);
    }                                                                              30
    return j;
}


void plot(int x, int y, colorType color)
{
    int i;

    i = (color.red * 0.299 + color.green * 0.587 + color.blue * 0.114) *
number_of_colors + (random()%100000)/100000.0;
                                                                                   40
#ifdef DEBUG
    if (i < 0 || i >= number_of_colors)
        fprintf(stderr, "plot: color of out range (0-%d): %d\n",
                number_of_colors-1, i);
#endif

    if (i <= 0)
        i = SRGP_BLACK;
    else if (i >= number_of_colors - 1)
        i = SRGP_WHITE;                                                            50
    else
        i++;

    SRGP_setColor(i);
    SRGP_pointCoord(x, y);
}

#endif /* USE_SRGP */
```

A.2.7 `vector.c`

```c
/*
 * Author:        Peter Holst Andersen
 * Last change:   August, 1994
 * Contents:      Functions to perform vector operations.
 */

#include <stdio.h>
#include <math.h>
#include "ray.h"
                                                                                   10
void vector_sub(vectorType *A, vectorType *B, vectorType *C)
{
    C->x = A->x - B->x;
    C->y = A->y - B->y;
    C->z = A->z - B->z;
}


void vector_copy(vectorType *A, vectorType *B)
{
    B->x = A->x;                                                                   20
```

```
        B−>y  =  A−>y;
        B−>z  =  A−>z;
}

/*
 * vector_norm(A) = |A|, A = A/|A|
 */
myfloat vector_norm(vectorType *A)
{
        myfloat a, b;                                                                                          30

        if ((a = vector_dot(A, A)) <= 0.0) return 0.0;
        b = sqrt((double) a);
        A−>x  /=  b;
        A−>y  /=  b;
        A−>z  /=  b;
        return b;
}

myfloat vector_dot(vectorType *A, vectorType *B)                                                                40
{
        return (A−>x * B−>x + A−>y * B−>y + A−>z * B−>z);
}

void vector_cross(vectorType *A, vectorType *B, vectorType *C)
{
        C−>x  =  (A−>y * B−>z) − (A−>z * B−>y);
        C−>y  =  (A−>z * B−>x) − (A−>x * B−>z);
        C−>z  =  (A−>x * B−>y) − (A−>y * B−>x);
}                                                                                                              50

myfloat vector_norm_cross(vectorType *A, vectorType *B, vectorType *C)
{
        vector_cross(A, B, C);
        return vector_norm(C);
}

void vector_scale(myfloat s, vectorType *A, vectorType *B)
{
        B−>x  =  A−>x * s;                                                                                      60
        B−>y  =  A−>y * s;
        B−>z  =  A−>z * s;
}
```

B   Scene Description Files for Rayshade

```
/* Options */

screen 512 512
sample 1
contrast 1.0 1.0 1.0
maxdepth 5
cutoff 0.002
eyep 0.0 -8.0 0.0
lookp 0.0 0.0 0.0
up 0.0 0.0 1.0
fov 45 45
light 1.0 1.0 1.0 ambient
shadowtransp
```

49

```
/* Scene 1 */

#include "options"

surface s0
  ambient 0.2 0.2 0.2
  diffuse 0.2 0.2 0.2
  specular 1.0 1.0 1.0
  reflect 0.5
  specpow 10

surface black
  ambient 0.0 0.0 0.0
  diffuse 0.0 0.0 0.0
  specular 0.0 0.0 0.0
  reflect 0.0

surface s3
  ambient 0.1 0.1 0.1
  diffuse 0.1 0.1 0.1
  specular 1.0 1.0 1.0
  reflect 0.7
  specpow 30

surface s4
  ambient 0.1 0.1 0.1
  diffuse 0.1 0.1 0.1
  specular 1.0 1.0 1.0
  reflect 0.1
  transp 0.8
  body 1.0 1.0 1.0
  index 1.3
  specpow 30

eyep 0.0 -8.0 6.0
lookp 0.0 0.0 3.0
screen 512 512

light 1.0 1.0 1.0 point 8.0 -8.0 24.0

sphere s3 2.5 0.0  4.0 5.0
sphere s4 1.6 0.7 -1.0 4.0
poly s0 20.0 20.0 0.0 -20.0 20.0 0.0 -20.0 -20.0 0.0 20.0 -20.0 0.0
  texture checker black

/* Scene 2 */

screen 512 512
sample 1
contrast 1.0 1.0 1.0
maxdepth 5
cutoff 0.002
eyep 0.0 -8.0 0.0
```

```
lookp 0.0 0.0 0.0
up 0.0 0.0 1.0
fov 45 45
light 1.0 1.0 1.0 ambient
light 1.0 1.0 1.0 point -40.0 -20.0 20.0
shadowtransp
applysurf
  ambient 0.2 0.2 0.2
  diffuse 0.2 0.2 0.2
  specular 1.0 1.0 1.0
  reflect 0.5
  specpow 10
sphere 1.2 0.0 0.0 0.0
sphere 0.5 0.0 -2.0 -0.5
sphere 1.2 3.0 0.0 0.0
sphere 1.2 -3.0 0.0 0.0
sphere 5.0 0.0 0.0 -8.0

/* Scene 3 */

#include "options"

surface s0
  ambient 0.2 0.2 0.2
  diffuse 0.2 0.2 0.2
  specular 1.0 1.0 1.0
  reflect 0.5
  specpow 10

disc s0 2.0 0.0 0.0 -3.0 0.0 0.0 1.0
disc s0 2.0 0.0 0.0 -2.0 0.0 0.2 0.7
disc s0 2.0 0.0 0.0 -1.0 0.0 0.2 0.6
disc s0 2.0 0.0 0.0  1.0 0.0 0.2 0.5
disc s0 2.0 0.0 0.0  3.0 0.0 0.1 0.2

light 1.0 1.0 1.0 point -40.0 -20.0 20.0

/* Scene 5 */

#include "options"

surface s0
  ambient 0.01 0.01 0.01
  diffuse 0.2 0.2 0.2
  specular 1.0 1.0 1.0
  reflect 0.1
  transp 0.9
  body 1.0 1.0 1.0
  index 1.3
  specpow 12

surface black
  ambient 0.0 0.0 0.0
```

```
   diffuse 0.0 0.0 0.0
   specular 0.0 0.0 0.0
   reflect 0.0

surface s2
   ambient 0.2 0.2 0.2
   diffuse 0.2 0.2 0.2
   specular 1.0 1.0 1.0
   reflect 0.5
   specpow 12

surface s3
   ambient 0.1 0.1 0.1
   diffuse 0.1 0.1 0.1
   specular 1.0 1.0 1.0
   reflect 0.25
   specpow 12

polygon s3 -40.0 40.0 -1.0 40.0 40.0 -1.0 40.0 -40.0 -1.0 -40.0 -40 -1.0

sphere s0 1.000000 0.000000 0.000000 0.000000
sphere s0 1.000000 -1.000000 1.732051 0.000000
sphere s0 1.000000 1.000000 1.732051 0.000000
sphere s0 1.000000 -2.000000 3.464102 0.000000
sphere s0 1.000000 0.000000 3.464102 0.000000
sphere s0 1.000000 2.000000 3.464102 0.000000
sphere s0 1.000000 -3.000000 5.196152 0.000000
sphere s0 1.000000 -1.000000 5.196152 0.000000
sphere s0 1.000000 1.000000 5.196152 0.000000
sphere s0 1.000000 3.000000 5.196152 0.000000
sphere s0 1.000000 -4.000000 6.928203 0.000000
sphere s0 1.000000 -2.000000 6.928203 0.000000
sphere s0 1.000000 0.000000 6.928203 0.000000
sphere s0 1.000000 2.000000 6.928203 0.000000
sphere s0 1.000000 4.000000 6.928203 0.000000
sphere s0 1.000000 0.000000 1.154701 1.632993
sphere s0 1.000000 -1.000000 2.886751 1.632993
sphere s0 1.000000 1.000000 2.886751 1.632993
sphere s0 1.000000 -2.000000 4.618802 1.632993
sphere s0 1.000000 0.000000 4.618802 1.632993
sphere s0 1.000000 2.000000 4.618802 1.632993
sphere s0 1.000000 -3.000000 6.350853 1.632993
sphere s0 1.000000 -1.000000 6.350853 1.632993
sphere s0 1.000000 1.000000 6.350853 1.632993
sphere s0 1.000000 3.000000 6.350853 1.632993
sphere s0 1.000000 0.000000 2.309401 3.265986
sphere s0 1.000000 -1.000000 4.041452 3.265986
sphere s0 1.000000 1.000000 4.041452 3.265986
sphere s0 1.000000 -2.000000 5.773503 3.265986
sphere s0 1.000000 0.000000 5.773503 3.265986
sphere s0 1.000000 2.000000 5.773503 3.265986
sphere s0 1.000000 0.000000 3.464102 4.898979
sphere s0 1.000000 -1.000000 5.196152 4.898979
```

```
sphere s0 1.000000 1.000000 5.196152 4.898979
sphere s0 1.000000 0.000000 4.618802 6.531973


light 1.0 1.0 1.0 point 5.0 -20.0 20.0


eyep 3.0 -10.0 6.0
lookp 0.0 3.464102 2.75



/* Scene 6 */

#include "options"

surface s0
  ambient 0.2 0.2 0.2
  diffuse 0.2 0.2 0.2
  specular 1.0 1.0 1.0
  reflect 0.5
  specpow 10

sphere 1.2  0.0  0.0  0.0
sphere 0.5  0.0 -2.0 -0.5
sphere 1.2  3.0  0.0  0.0
sphere 1.2 -3.0  0.0  0.0
sphere 5.0  0.0  0.0 -8.0

light 0.0 0.4 0.4 point -40.0 -20.0 20.0
light 0.6 0.0 0.0 point -20.0 -20.0 20.0
light 0.0 0.6 0.0 point   0.0 -20.0 20.0
light 0.0 0.0 0.6 point  20.0 -20.0 20.0
light 0.4 0.4 0.0 point  40.0 -20.0 20.0

/* Scene 7 */

#include "options"

surface s0
  ambient 0.2 0.2 0.2
  diffuse 0.2 0.2 0.2
  specular 1.0 1.0 1.0
  reflect 0.5
  specpow 10

disc s0 2.0 0.0 0.0 -3.0 0.0 0.0 1.0
disc s0 2.0 0.0 0.0 -2.0 0.0 0.2 0.7
disc s0 2.0 0.0 0.0 -1.0 0.0 0.2 0.6
disc s0 2.0 0.0 0.0  1.0 0.0 0.2 0.5
disc s0 2.0 0.0 0.0  3.0 0.0 0.1 0.2


light 0.0 0.4 0.4 point -40.0 -20.0 20.0
light 0.6 0.0 0.0 point -20.0 -20.0 20.0
light 0.0 0.6 0.0 point   0.0 -20.0 20.0
light 0.0 0.0 0.6 point  20.0 -20.0 20.0
```

```
light 0.4 0.4 0.0 point  40.0 -20.0 20.0


/* Scene 9 */

#include "options"

surface s0
  ambient 0.01 0.01 0.01
  diffuse 0.2 0.2 0.2
  specular 1.0 1.0 1.0
  reflect 0.1
  transp 0.9
  body 1.0 1.0 1.0
  index 1.3
  specpow 12

surface black
  ambient 0.0 0.0 0.0
  diffuse 0.0 0.0 0.0
  specular 0.0 0.0 0.0
  reflect 0.0

surface s2
  ambient 0.2 0.2 0.2
  diffuse 0.2 0.2 0.2
  specular 1.0 1.0 1.0
  reflect 0.5
  specpow 12

surface s3
  ambient 0.1 0.1 0.1
  diffuse 0.1 0.1 0.1
  specular 1.0 1.0 1.0
  reflect 0.25
  specpow 12

polygon s3 -40.0 40.0 -1.0 40.0 40.0 -1.0 40.0 -40.0 -1.0 -40.0 -40 -1.0

sphere s0 1.000000 0.000000 0.000000 0.000000
sphere s0 1.000000 -1.000000 1.732051 0.000000
sphere s0 1.000000 1.000000 1.732051 0.000000
sphere s0 1.000000 -2.000000 3.464102 0.000000
sphere s0 1.000000 0.000000 3.464102 0.000000
sphere s0 1.000000 2.000000 3.464102 0.000000
sphere s0 1.000000 -3.000000 5.196152 0.000000
sphere s0 1.000000 -1.000000 5.196152 0.000000
sphere s0 1.000000 1.000000 5.196152 0.000000
sphere s0 1.000000 3.000000 5.196152 0.000000
sphere s0 1.000000 -4.000000 6.928203 0.000000
sphere s0 1.000000 -2.000000 6.928203 0.000000
sphere s0 1.000000 0.000000 6.928203 0.000000
sphere s0 1.000000 2.000000 6.928203 0.000000
```

```
sphere s0 1.000000 4.000000 6.928203 0.000000
sphere s0 1.000000 0.000000 1.154701 1.632993
sphere s0 1.000000 -1.000000 2.886751 1.632993
sphere s0 1.000000 1.000000 2.886751 1.632993
sphere s0 1.000000 -2.000000 4.618802 1.632993
sphere s0 1.000000 0.000000 4.618802 1.632993
sphere s0 1.000000 2.000000 4.618802 1.632993
sphere s0 1.000000 -3.000000 6.350853 1.632993
sphere s0 1.000000 -1.000000 6.350853 1.632993
sphere s0 1.000000 1.000000 6.350853 1.632993
sphere s0 1.000000 3.000000 6.350853 1.632993
sphere s0 1.000000 0.000000 2.309401 3.265986
sphere s0 1.000000 -1.000000 4.041452 3.265986
sphere s0 1.000000 1.000000 4.041452 3.265986
sphere s0 1.000000 -2.000000 5.773503 3.265986
sphere s0 1.000000 0.000000 5.773503 3.265986
sphere s0 1.000000 2.000000 5.773503 3.265986
sphere s0 1.000000 0.000000 3.464102 4.898979
sphere s0 1.000000 -1.000000 5.196152 4.898979
sphere s0 1.000000 1.000000 5.196152 4.898979
sphere s0 1.000000 0.000000 4.618802 6.531973

light 0.0 0.4 0.4 point -40.0 -20.0 20.0
light 0.6 0.0 0.0 point -20.0 -20.0 20.0
light 0.0 0.6 0.0 point   0.0 -20.0 20.0
light 0.0 0.0 0.6 point  20.0 -20.0 20.0
light 0.4 0.4 0.0 point  40.0 -20.0 20.0

eyep 3.0 -10.0 6.0
lookp 0.0 3.464102 2.75
```