Partial Evaluation Applied to Ray Tracing

Peter Holst Andersen

DIKU, Department of Computer Science, University of Copenhagen,
Universitetsparken 1, DK-2100 Copenhagen, Denmark,
e-mail: txix@diku.dk

Summary: We use partial evaluation to speed up an already efficient ray tracer by specialization with respect to part of the input. The ray tracer was implemented in C and specialized using C-Mix, a partial evaluator for the C programming language. The resulting programs run up to three times faster than the original program on the same input.

1 Introduction

Objective. Our aim is to evaluate the use of partial evaluation as a general optimization tool. We do this by applying partial evaluation to a ray tracer that has already been programmed for speed. The ray tracer is specialized with respect to the objects and light sources in the scene. The ray tracer is a 2000 lines C program.

Given partial knowledge `s` of the ray tracer's input, we will show how partial evaluation can perform optimizations, that are too tedious to do by hand, and which ordinary compilers will not do (either because they are too complex, or because they depend on the input `s`, which is not available at compile time). At the same time we demonstrate the underlying idea of partial evaluation, namely that a general program can automatically be transformed into a specialized one, which potentially is much faster.

We have used C-Mix [And94], a partial evaluator for the C programming language. More information can be found on the C-Mix home page:
`http://www.diku.dk/research-groups/topps/activities/cmix.html`.
This paper is a short version of the technical report [And95].

Overview. Section 2 gives a brief introduction to ray tracing, section 3 describes the results of specializing the ray tracer, and section 4 describes related work and concludes.

An introduction to partial evaluation may be found in this volume.

2 Ray Tracing

Ray tracing is a method used in computer graphics, which is known to give good picture rendition of a scene (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 correspond 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 (a ray is defined by an origin point and a direction vector) 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.
- Finally, a *reflection ray* is spawned to model reflections.

Each of these reflection and refraction rays may, in turn, recursively spawn shadow, reflection, and refraction rays. Recursion terminates when either the contribution is too small or a certain maximum depth is reached.

The usual implementation is by a general algorithm which, given a scene and a ray, performs computation to follow its path. The main loop of the ray tracing algorithm could look like this:

```
for each pixel (x,y) in the picture do
    ray = <the ray from the eye point through (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.

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 = <the ray from the eye point through (x,y)>
    color = trace_scene(ray);
    plot(x, y, color);
```

The function `trace_scene` is a version of `trace` specialized with respect to a particular scene. The specialized function is often significantly faster than the original trace function, because it can be optimized with respect to the scene.

The implementation. In this paper we focus on the automatic optimization of basic ray tracing techniques, and not of advanced ray tracing features. We implemented a simple, but yet efficient, ray tracer using Whitted's shading model [FvDFH90]. Our implementation supports point-formed light sources and the following object types: spheres, squares and discs.

The most time consuming computation of virtually all ray tracers are the *intersection tests*. For each object type the ray tracer has an intersection function that, given an object and a ray, determines whether or not they intersect. A ray tracer performs millions of intersection tests to generate an image, which makes optimization highly relevant. Below is the intersection function for spheres:

```
double intersection_sphere(sphereType sphere, rayType ray)
{
    double x, y, z, b, d, t;

    x = sphere.c.x - ray.p.x;
    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;              /* No intersection */
    d = sqrt(d);
    t = b - d;
    if (t <= 0.001) {                      /* Avoid choosing a point */
        t = b + d;                         /* behind or too close to */
        if (t <= 0.001) return 0.0;        /* the origin             */
    }
    return t;
}
```

`sphere.c` is the center of the sphere, and `sphere.r2` is the radius squared. The ray's origin point is `ray.p` and `ray.v` is its direction.

We have compared our implementation with Rayshade 4.0, a public domain raytracer, which has been reported to have fast intersection routines [Hai93]. Experiments show that our implementation of the intersection routines are more efficient than Rayshade's. This is not surprising, since the intersection code in Rayshade is very modular in order to make introduction of new object types painless. Also Rayshade offers more features than our implementation. Still, it is safe to conclude that our implementation is as least as efficient as Rayshade.

The details of the comparison runs can be found in the technical report.

3 Partial Evaluation of the Ray Tracer

A series of experiments was carried out in which we specialized our ray tracer with respect to different scenes. This section reports some of the speedups obtained and the reasons for them. We report the results of two experiments (more can be found in the technical report):

- Specializing the intersection functions (*e.g.* `intersection_sphere`) with respect to the objects in the scene.
- Specializing the intersection functions plus the shading function with respect to the objects and the light sources in the scene.

The specialization in the first experiment cannot eliminate the intersection 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 the second experiment additional speedup is possible, since part of the intersection computation depends solely on the object data and the light source 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 (*e.g.* a "fly through" for animation). 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 with many reflecting or transparent objects.

Scenes. Scene 1 consists of two spheres and one square. Scene 2, 3, and 4 consists of five objects each; spheres, discs, and squares respectively. Scene 5 consists of 36 spheres and one square. Scene 1 to 5 all have one light source. Scenes 6 to 9 are made up of the same objects as scenes 2 to 5, but they have five light sources instead of one.

Platform. The experiments were performed on a HP 9000/735 running HP-UX version A.09.05. We compiled the programs with two different compilers: Gnu's C compiler version 2.5.8 (`gcc`) and the compiler supplied by HP (`cc`). The following parameters were used: for both compilers it was specified that all the intersection functions should be inlined. The optimizations option `-O2` was used for `gcc`, and `+O4 +Onolimit` was used for `cc`. The option `+Onolimit` means that the compiler may use 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 executed each program at least three times when no other CPU-intensive processes were running. In case the times

varied we 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.

3.1 Specializing with respect to objects

The intersection functions have been specialized with respect to the first five scenes. The time is given in CPU user seconds, and the size gives the number of kilobytes of the object file (`.o`) as reported by `size`. "Spd" is the speedup.

	Gnu C compiler (gcc)					HP C compiler (cc)				
Scene	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**

Below is the result of specializing the function `intersection_sphere` from above with respect to a sphere with center in (1.0, 2.0, 3.0) and with a radius of 2.0:

```
double intersection_sphere_42(rayType ray)
{
    double x, y, z, b, d, t;

    x = 1.0 - ray.p.x;
    y = 2.0 - ray.p.y;
    z = 3.0 - 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 + 4.0;
    if (d <= 0.0) return 0.0;
    d = sqrt(d);
    t = b - d;
    if (t <= 0.001) {
        t = b + d;
        if (t <= 0.001) return 0.0;
    }
    return t;
}
```

Note that the intersection computations have not been eliminated, since they depend on the dynamic ray data.

What gives the speedup? The following transformations have been performed: inter-procedural constant propagation, unrolling of a loop that iterates over the objects, and specializing away a switch-case statement, that branches on object type. 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 the optimizations performed by 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.

3.2 Specialization with respect to objects and light sources

To achieve extra speedup, we have applied a simple but 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.

	Gnu C compiler (gcc)					HP C compiler (cc)				
Scene	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**

Below is the sphere intersection function specialized with respect to same sphere as before, but here it has also been specialized with respect to a ray with origin point in (10.0, 10.0, 10.0):

```
double intersection_sphere_42(vetorType ray_v)
{
    double b, d, t;

    b = - 9.0*ray.v.x - 8.0*ray.v.y - 7.0*ray.v.z;
    d = b*b - 81.0 - 64.0 - 49.0 + 4.0;
    if (d <= 0.0) return 0.0;
    d = sqrt(d);
    t = b - d;
    if (t <= 0.001) {
        t = b + d;
        if (t <= 0.001) return 0.0;
    }
    return t;
}
```

Note that the computations of `x`, `y`, and `z` have been eliminated as well the computations of their squares (line 2 of the function above). We leave the reduction of the constant subexpression (`- 81.0 - 64.0 - 49.0 + 4.0`) to the compiler. Since the function has been specialized with respect to the sphere's geometry and the origin point of the ray, the specialized function only takes the direction vector of the ray as parameter.

4 Related Work, Conclusion, and Future work

Related work. Mogensen specialized a very modular ray tracer written in a functional language [Mog86], 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 [Han83]. 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.

Conclusion. We have used partial evaluation to optimize an already efficient ray tracer, gaining speedups from 1.8 to 3.0 (using cc) and from 1.3 to 1.9 (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 the static 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.

Many C programs have some global data structures, that 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.

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 obstacles are of practical nature. Rayshade uses function pointers and dynamic allocation, which the present C-Mix implementation does not support (the algorithms are described in [And94]).

It would also be interesting to see how the presence of various ray tracing acceleration techniques would impact the 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 we have done here, 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), etc. 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. In general, we expect that the speedup from most ray tracing acceleration techniques are orthogonal to those achieved by partial evaluation, since we optimize the intersection routines whereas most acceleration techniques reduce the number of calls to the routines.

Bibliography

[And94] Lars Ole Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, May 1994. DIKU report 94/19.

[And95] Peter Holst Andersen. Partial evaluation applied to ray tracing. 1995. DIKU report 95/2.

[FvDFH90] Foley, van Dam, Feiner, and Hughes. *Computer Graphics Principles and Practice*. Reading, MA: Addison-Wesley, 1990.

[Hai93] Eric Haines. Ray tracer races, round 2. *Ray Tracing News*, july 1993.

[Han83] Pat Hanrahan. Ray tracing algebraic surfaces. *Computer Graphics*, Volume 17, Number 3, July 1983.

[Mog86] Torben Mogensen. *The Application of Partial Evaluation to Ray-Tracing*. Master's thesis, DIKU, University of Copenhagen, Denmark, 1986.