Specializing Shaders

Brian Guenter
Todd B. Knoblock
Erik Ruf

August 2, 1995

Technical Report
MSR-TR-95-49

Microsoft Research
Advanced Technology Division
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052

This report is a preprint of the paper "Specializing Shaders," to appear in Proceedings of SIGGRAPH 95 (*Computer Graphics* Proceedings, Annual Conference Series, 1995).

Specializing Shaders

Brian Guenter, Todd B. Knoblock, Erik Ruf *

Microsoft Research

Abstract

We have developed a system for interactive manipulation of shading parameters for three dimensional rendering. The system takes as input user-defined shaders, written in a subset of C, which are then *specialized* for interactive use. Since users typically experiment with different values of a single shader parameter while leaving the others constant, we can benefit by automatically generating a specialized shader that performs only those computations depending on the parameter being varied; all other values needed by the shader can be precomputed and cached. The specialized shaders are as much as 95 times faster than the original user defined shader. This dramatic improvement in speed makes it possible to interactively view parameter changes for relatively complex shading models, such as procedural solid texturing.

1. Introduction

During the process of interactively producing a computer-rendered image, a user will typically experiment with various image parameters, such as the positions of objects and light sources, the optical characteristics of surfaces, and surface textures. We would like to display the results of such parameter changes as rapidly as possible. Some of these changes, such as moving an object, are likely to require significant recomputation, while others, such as changing the color of one object, should not.

This paper describes a mechanism for rapidly reacting to changes to shading parameters that control the *local* shading of objects in the scene. The local shading of an object is dependent only on object surface properties, such as the surface normal and surface position, and the geometric relationship of those properties with the illumination sources and the viewpoint. Global illumination effects, such as specular and diffuse interreflectance among objects, and shadows, are not part of the local shading model.

Consider the simple Phong-like shader in Figure 1. If the user changes control parameter `ks`, only the multiplication `ks*specular_component` and the enclosing addition and vector multiplication operations need be reperformed; all other computations are guaranteed to return the same values as before. If, on the other hand, the parameter `specular` is changed, we may either have to recompute `specular_component` (and all expressions referring to it), or nothing at all, depending on the value of `n_dot_h` for that particular pixel. In either case, changing a parameter value invalidates only a small portion of the work already performed.

We take advantage of this observation as follows. Given an arbitrary user shading routine written in a subset of C and an indication of which parameter is to be altered, we perform an analysis to determine which of the shader's computations are invariant across changing values of the distinguished parameter. We then automatically generate

1. code to perform the non-varying computations and save an appropriate subset of the results into a per-pixel cache area, and
2. code to efficiently perform shading given the new parameter value and the cached information.

After executing (1) once, we need only execute the more efficient code in (2) each time the user alters the parameter. When the user selects a different parameter, we simply install new versions of (1) and (2). Figures 2 and 3 show the code resulting from the specialization of `Phong` on the parameter `specular`.

The remainder of this paper is divided into five sections. Section 2 outlines the general architecture of our system, while Section 3 describes the specialization process in more detail. In Section 4, we analyze the costs and benefits of our approach on several shaders. We conclude, in Sections 5 and 6, with descriptions of related and future work.

2. System Architecture

Our system consists of three main components: the geometric renderer, the specializer, and the user interface driver loop.

2.1 Geometric Renderer

We render geometric image information offline before the interactive manipulation stage. Each pixel of the geometric image contains an object identifier associated with the object visible at that pixel, and the depth and surface normal of the point on the object surface which is at the center of the pixel. Our prototype uses a modified version of the Vivid ray tracer [17] to compute this information.

During the interactive stage, the user loads a precomputed geometric image and then adjusts shading parameters on objects in the scene. Because the geometric image is rendered prior to the interactive stage, some image properties cannot be varied interactively. For example, the eyepoint is fixed, as are the position and orientation of all the objects in the scene. However, many scene characteristics, such as the position of lights, and the color and reflectance characteristics of objects, can be varied interactively.

*Authors' address: Microsoft Research, One Microsoft Way, Redmond, WA 98052. Email: {briangu, toddk, erikruf}@microsoft.com.

Paper to appear in Proceedings of SIGGRAPH 95 (*Computer Graphics* Proceedings, Annual Conference Series, 1995).

```
void Phong(int i, int j, Vec eye_position, Vec surface_normal, Flt depth,
             Matrix view_inverse, Matrix model_inverse, Matrix shade_inverse,
             PImageStruct dib,
             /* the following are the control parameters for this shader */
             double red, double green, double blue, double specular,
             double ks, double kd, double ambient, double lx, double ly, double lz)
{
       Vec light = {lx, ly, lz};
       Vec world_position, light_vector, h_vector, eye_vector;
       double specular_component, diffuse_component, n_dot_h;
       Vec rgb = {red, green, blue};

       world_position = trans_vector(view_inverse, vec(i, j, depth));
       eye_vector = VecNormalize(VecSub(eye_position, world_position));
       light_vector = VecNormalize(VecSub(light, world_position));
       h_vector = VecNormalize(VecAdd(light_vector, eye_vector));
       n_dot_h = VecDot(surface_normal, h_vector);

       if (n_dot_h > 0)
              specular_component = pow(n_dot_h, specular);
       else
              specular_component = 0;

       diffuse_component = VecDot(surface_normal, light_vector);

       if (diffuse_component < 0)
              diffuse_component = 0;

       rgb = VecScalarMul(rgb,
              (kd*diffuse_component + ambient) + ks*specular_component);

       SetPixelColor(j, i, rgb, dib);
}
```

Figure 1: Source for a simple Phong shader. The parts of the shader that are dependent upon the parameter specular are underlined.

2.2 Specializer

The specializer takes a user shader function and a designation of the control parameter to be varied and produces shading code specialized for that parameter.

The user writes the shaders in a subset of C augmented with a library of useful graphics and mathematical functions. The framework is similar to that provided for writing shaders in *RenderMan* [7, 16]. The restricted subset of C excludes the following C features:

- struct, union, pointer, and array types other than limited support for primitive data types such as vectors and points,
- continue, break, and goto statements, and
- recursion in the shader functions (although subroutines may employ recursion).

In some cases (e.g., struct types and structured loop exits), these restrictions simplify the implementation; in other cases (e.g., pointers, goto), they serve to make the necessary analyses tractable.

The shader is responsible for calculating rgb values for each pixel based on image model information common to all shaders, and on shader-specific *control parameters*. Consider the sample shader, `Phong`, from Figure 1. The first 9 parameters of `Phong`, up through `dib`, are standard; they provide rendering data about each point and an output context. The remainder are the control parameters, and are associated with sliders in the user interface.

Given a shader and a designated parameter, the specializer produces C++ code for two new procedures: a *cache loader* that precomputes and caches intermediate values that are independent of the designated parameter, and a *cache reader* that performs shading using the cached information. Figures 2 and 3 show the generated loader and reader functions, respectively. We rely upon the compiler's optimizer to further improve the code by, e.g., eliminating dead code and hoisting loop-invariant computations.

We install a new shader into the system by calling the specializer on the shader and each of the shader's control parameters, generating a loader/reader pair for each parameter. These procedures are then compiled and dynamically linked into the system.

2.3 Driver Loop

The user interface driver loop allows the user to perform housekeeping tasks such as loading images, selecting shaders, and designating particular objects within the image for shading. It also allows the user to interactively modify the current shader's control parameters, and performs shading and image redisplay after each modification. Pseudocode for this portion of the driver is shown in Figure 4.

After the user has selected a parameter to modify, the driver executes the corresponding cache loader to precompute the shader

```
typedef struct {
  int c0;
  double c1;
  double c2;
} Cache;

world_position = trans_vector(view_inverse, vec(i, j, depth));
eye_vector = VecNormalize(VecSub(eye_position, world_position));
light_vector = VecNormalize(VecSub(light, world_position));
h_vector = VecNormalize(VecAdd(light_vector, eye_vector));
n_dot_h = VecDot(surface_normal, h_vector);
if (pcache->c0 = n_dot_h>0)
  pcache->c1 = n_dot_h;
else
  specular_component = 0;
diffuse_component = VecDot(surface_normal, light_vector);
if (diffuse_component<0)
  diffuse_component = 0;
pcache->c2 = kd*diffuse_component+ambient;
```

Figure 2: Body of the cache loader for the specialization with respect to specular.

```
if (pcache->c0)
  specular_component = pow(pcache->c1, specular);
else
  specular_component = 0;

rgb = VecScalarMul(rgb, pcache->c2+ks*specular_component);
SetPixelColor(j, i, rgb, dib);
```

Figure 3: Body of the cache reader for the specialization with respect to specular.

```
repeat until new image/object/shader
  user selects a parameter P
  for each pixel in current object
    invoke loader[P]
  repeat until new parameter P          (*)
    user supplies a value V for P
    for each pixel in current object
      invoke reader[P](V)
    redisplay all pixels in object
```

Figure 4: User interface driver loop.

values that will be needed by the cache reader. Then, each time the user supplies a value for the chosen parameter, the cache reader is executed to perform the actual shading and the image is redisplayed. This runs more quickly because we have hoisted the parameter-independent computations out of the inner `repeat` loop (marked with *) in Figure 4.

3. Specialization

The specializer begins by parsing and typechecking the shader function. The result is an abstract syntax tree (AST) annotated with type information. The core of the specialization process then proceeds in three steps.

- First, the dependence analysis determines which parts of the shader function are independent of the value of the designated control parameter.
- Second, the caching analysis determines which of the independent computations should be cached.
- Third, the splitting transformation separates the original shader function into cache loader and cache reader functions.

Space limitations prevent us from presenting the specializer in detail. The next three sections present a simplified description of the three specialization steps.

3.1 Dependence Analysis

Dependence analysis determines which parts of the computation are dependent or independent of the designated control parameter. The basic rule for dependence is that an expression is dependent on the designated parameter if one or more of its operands is. Although this basic rule would handle pure functional programs, the imperative assignment of variables in C requires a slightly more complicated analysis.

A reference to a variable is dependent (on the designated control parameter) if the value assigned to the variable is dependent. It is also dependent if the choice of which (independent) value to assign is dependent. For example, if an assignment statement is guarded by an `if` statement whose predicate is dependent, then a downstream reference to the variable is dependent.

In Figure 1, the parts of the function that are dependent upon the parameter `specular` are underlined. In this function, the first assignment to `specular_component` is dependent because it is assigned a value computed from the designated parameter, `specular.` This in turn forces the part of the calculation of `rgb` involving `specular_component` to be dependent. Finally, the call to `SetPixelColor` is dependent because it contains a reference to `rgb,` which is dependent because of the previous assignment.

Our solution to dependence analysis is a straightforward application of standard dataflow analysis/abstract interpretation techniques. It works by flowing a model of the dependence of every variable though the shader function along the control paths. The final result of the dependence analysis is an annotation on every node of the AST as to whether or not its value may be dependent upon the designated control parameter. These annotations are necessarily conservative; assuming that a computation is dependent upon the control parameter when in fact it is not may make the specializer miss an optimization opportunity, but will not make it produce the wrong result.

3.2 Caching Analysis

The caching analysis determines which of the independent values in the shader should be cached and which parts of the shader only need to occur in the cache reader. The results of the dependence analysis help to provide an upper bound on what might be cached. A priori, any independent computation is a candidate to be cached. However, independence is not sufficient to determine what *should* be cached. Consider the cache for the specialization of `Phong` on `specular`, represented in Figures 2 and 3. Since the value for `kd*diffuse_component+ambient` is cached, caching the either subexpression `kd*diffuse_component`, or the individual values of `kd`, `diffuse_component`, or `ambient`, would be inefficient.

The caching analysis classifies each shader expression as either *static*, *cache*, or *dynamic*. *Static* expressions are those that need only be evaluated at load time. For example, in the assignment

```
h_vector =
  VecNormalize(VecAdd(lightVector,
                      eye_vector));
```

even though the right hand side is independent of the designated parameter, it does not need to be cached because the only reference to `h_vector` is itself cached. *Cache* expressions are also evaluated at load time, but are consumed by dependent computations, so their results must be stored into the cache for later use by the cache reader. *Dynamic* expressions need to be computed during cache reading.

This classification is performed using simple heuristics. An expression is marked as cache if it is independent, is the maximal such expression, is governed only by independent control predicates, and is not within a dependent `while` statement. Independent, non-cached expressions that do not directly contribute to dependent computations are marked as static. The remaining expressions are marked as dynamic.

In the `Phong` shader, three expressions are selected for caching:[1]

- `n_dot_h>0`,
- `n_dot_h`, and
- `kd*diffuse_component+ambient`.

All of the assignment statements other than the assignments to `specular_component` and `rgb` are marked as static. The

[1]The reason that `n_dot_h>0` and `n_dot_h` are both cached is that there are distinct (dependent) consumers of these values in the cache reader. One could choose to cache just `n_dot_h` and recalculate `n_dot_h>0` in the cache reader.

```
AST MakeLoader(AST e) {
  switch on mark(e)
    case Static:
      return e

    case Cache:
      allocate a cache slot s for e
      return MakeAssignment(s,e)

    case Dynamic:
      switch on form of e
        case e is an expression e0⊕e1:
          return MakeExp(⊕, MakeLoader(e0),
                            MakeLoader(e1))

        case e is if (e0) e1 else e2:
          if (e0 is independent)
            return MakeIf(MakeLoader(e0),
                          MakeLoader(e1),
                          MakeLoader(e2))
          else
            return MakeLoader(e0)

        case e is while(e0) e1:
          if (e0 and e1 are independent)
           return MakeWhile(MakeLoader(e0),
                            MakeLoader(e1))
          else
            return ∅

        case ...
 }
```

Figure 5: The cache loader transformation.

```
AST MakeReader(AST e) {
  switch mark(e)
    case Static:
      return ∅

    case Cache:
      find cache slot s allocated for e
      return s

    case Dynamic:
      switch on form of e
        case e is an expression e0⊕e1:
          return MakeExp(⊕, MakeReader(e0),
                            MakeReader(e1))

        case e is if (e0) e1 else e2:
          return MakeIf(MakeReader(e0),
                        MakeReader(e1),
                        MakeReader(e2))

        case e is while(e0) e1:
          return MakeWhile(MakeReader(e0),
                           MakeReader(e1))
        case ...
  }
```

Figure 6: The cache reader transformation.

first `if` statement is dynamic because its then-part is dependent, but the second `if` is static.

3.3 Splitting Transformation

In its third and final phase, the specializer derives cache loader and cache reader functions from the AST, which has been annotated with the results of the dependence and cache analyses. Figures 5 and 6 present partial pseudo-code for the transformations that produce the cache loader and cache reader. The transformations are based solely upon the annotations and a case analysis on the form of the AST node *e*:

- If the node *e* is marked as static, then by definition it only needs to be included in the cache loader.
- If the node *e* is marked as cache, then the loader allocates a cache slot for it and generates an assignment statement in the cache loader function that assigns the slot the value of the expression. The corresponding piece of the reader simply reads the cache slot.
- If the node *e* is a dynamic `if` statement and it has a test that is independent of the designated parameters, then the loader may load cache values in the branches of the `if`. Such loading cannot be performed if the test is dependent, since the cached expressions will no longer be guarded by the test, possibly inducing a run-time exception that would not have occurred in the original program.
- If the node *e* is a dynamic `while` statement, we conservatively choose not to cache anything within it. This implementation misses some cases where an independent value could be cached: where (1) the independent value is loop invariant or (2) where the only dependent consumers are outside the loop.

Figures 2 and 3 contain the results of running the splitting transformation on the specialization of `Phong` for `specular`. The first five assignments of the shader are static, and are reproduced only in the loader. The first `if` statement is dynamic with an independent test; thus a version of the `if` appears in both the loader and the reader. Both the predicate and the reference to `n_dot_h` are cached, and are thus assigned in the loader and refernced in the cache reader. The assignment to identifier `specular_component` is actually dead in the loader--we would expect the compiler optimizer to remove it. The second `if` is static, and so is included only in the loader. The third and final cache element represents the maximally independent expression `kd*diffuse_component+ambient`.

3.4 Pragmatics

Cache space is a valuable resource. The pseudo-code in figure 5 implies that new space is allocated for each value that is cached, which would be correct but inefficient. For example, there are situations where two cached values may be correctly allocated to overlapping space in the cache, e.g., when only one of the values will be live on any given invocation of the cache loader. Our specializer employs some simple rules to more efficiently allocate cache slots.

Cache space is also a finite resource, as performance will suffer if the total space allocated for the cache exceeds the available physical memory. For the moderate-complexity shaders we have specialized thus far, cache size has not been an issue. In the specialization of more complex shaders, it will be necessary to explicitly limit the amount of cache space that is allocated. To do this effectively, we plan to introduce a cost metric that approximates the cost of each computation that is a candidate for caching. This would then be used to guide what should be cached and what should be recomputed in the cache reader by caching only the top-scoring candidates.

We can simultaneously improve cache size and reader efficiency by rewriting computations in the shader. One such opportunity is the associative rearrangement of arithmetic expressions.[2] For example, the expression `(x+y)+z` contains two computations that are independent of `x`, namely `y` and `z`. But if we reassociate it as `x+(y+z)`, this contains just one independent computation, `y+z`. The transformed code requires one less cache slot, and eliminates one "+" operation in the cache reader (Hanrahan and Lawson [7] describe a similar optimization).

Thus far, we have described specializing shaders on a single control parameter. The specializer, in fact, supports specializations on multiple parameters. So, for example, one could produce a specialization that would allow the fast manipulation of any of the `red`, `green`, or `blue` parameters without having to call the loader again. For combinations of control parameters, there are too many possible specializations for them all to be generated in advance. We plan to extend the system to dynamically call the specializer at run-time when a subset of the parameters is designated by the user.

4. Results

The specialization algorithm described above has been implemented in a prototype system consisting of approximately 20K lines of C++, Flex, and Bison code. The user interface driver loop is implemented in a small amount of Visual Basic. This section presents timing results for specialized shaders.

4.1 Timings

We show results of the specializations of two shaders: the `Phong` shader of Figure 1, which has 10 control parameters, and a procedural solid texture shader, `Texture`, having 18 control parameters. These shaders were tested on the 320 by 243 pixel image shown in Plate 1. This image consists of five objects, including the background plane. We chose to time our shaders on the background plane because it is the largest object in the image. The timings were performed on a Gateway 2000 P5-90 Pentium processor with 64 megabytes of physical memory. The performance of the specializer is not an issue; specialization is performed offline and typically requires less than one second per specialization generated. Thus, we present timings only for the shaders themselves.

Four statistics are of interest:

1. the time to execute the original shader,
2. the time to precompute the cached values,

[2]While floating point arithmetic does not obey the mathematical associativity laws, as long as the rewritten expression does not overflow, the effects are unlikely to be observable in a shading function.

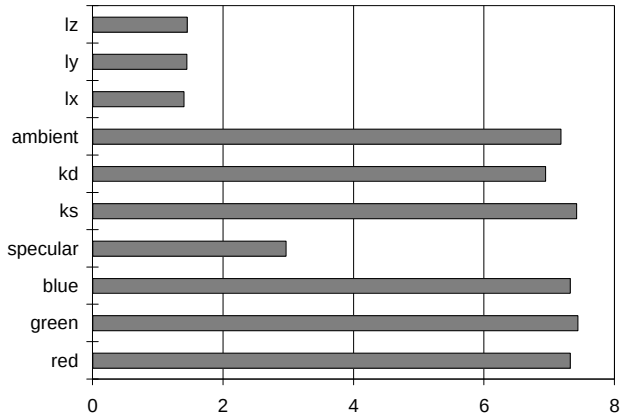

Figure 7: Relative speedup due to specialization for Phong for each control parameter.

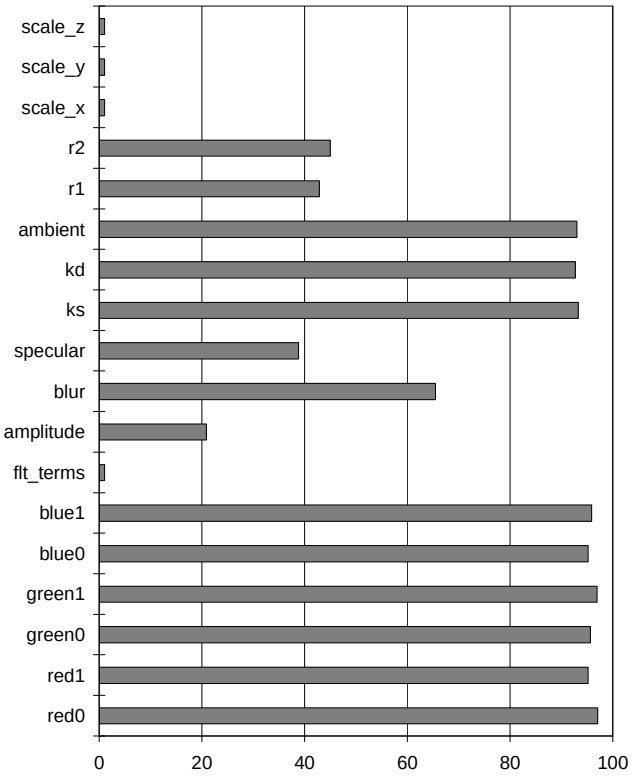

Figure 8: Relative speedup due to specialization for Texture for each control parameter

3. the time to perform shading given the cached and the control parameter values, and
4. the ratio of (1) and (3).

Figures 7 and 8 show the speedup ratio (4) for both shaders. These ratios vary from 1.4 through 7.4 for `Phong`, and 1.0 through 97.0 for `Texture`. Table 1 gives the absolute data for `Texture`; the table of absolute data for `Phong` is omitted to save space.

4.2 Discussion

The overhead of our two-phase shading approach is the time taken to run the cache loader. In our test cases, the sum of the loader and reader times is approximately equal to the time taken for a single invocation of the unspecialized shader. The cost of the additional memory operations to load the cache is balanced by the removal of statements that do not contribute to the cached values.

When the user first selects a new control parameter, the system must execute both the cache loader and reader; thus, the first shading may be slower than the original, unspecialized shader. Subsequent parameter changes only execute the cache reader; thus, our interactive speedup is the ratio of the original and reader times. The user will experience this speedup most of the time when interacting with the system.

The speedup ratio is highly dependent upon the particular control parameter chosen because we may only cache the results of computations that are independent of this parameter. In some cases, virtually all of the computations depend upon the chosen parameter, e.g., `lx` in `Phong` and `flt_terms` in `Texture`. Speedups for such cases are low, approaching 1, but never less than 1. In the majority of cases, however, the cache reader is much faster than the unspecialized shader, meaning that we achieve a net gain in processing speed when the user changes a parameter twice or more successively.

Figures 7 and 8 demonstrate that the majority of parameters are computed significantly faster in specialized form. For example, 7 of the 10 parameters in `Phong` were at least 3 times faster, and 6 parameters were at least 7 times faster, in specialized form. The speedup ratios are even more dramatic for the procedural solid texture shader. Of 18 parameters, 14 were at least 25 times faster and 9 were at least 90 times faster. The `Texture` shader contains several expensive computations that depend on only a few control parameters, allowing significant speedups in specializations on other parameters.

Parameter	User Shader	Cache Loader	Cache Reader	Ratio
scale_z	31.14	2.24	30.54	1.02
scale_y	31.15	2.43	30.46	1.02
scale_x	31.14	2.24	30.22	1.03
r2	31.27	32.25	0.70	44.95
r1	30.95	32.11	0.72	42.82
ambient	31.15	31.33	0.33	93.03
kd	31.15	31.22	0.34	92.69
ks	31.15	31.02	0.33	93.26
specular	31.11	30.70	0.80	38.80
blur	31.15	31.32	0.48	65.45
amplitude	31.15	32.43	1.50	20.83
flt_terms	44.81	2.23	43.08	1.04
blue1	31.15	31.25	0.32	95.87
blue0	31.15	31.47	0.33	95.17
green1	31.15	31.06	0.32	96.94
green0	31.15	31.03	0.33	95.64
red1	31.15	31.07	0.33	95.20
red0	31.15	31.24	0.32	97.05

Table 1: Execution time in seconds for the user shader, cache loader, and cache reader, and ratio of user shader time to reader time.

Color Plates 1 and 2 show a typical incremental control parameter change. In Plate 1, the amplitude parameter of the procedural texture shader for the tiled floor object is set to 12. In plate 2, it has been changed to 6. The time to reshade the image following this change was approximately 12 seconds for the original (unspecialized) shader and 0.5 seconds for the specialized shader.

5. Related Work

Relevant work includes the specialization of programs on partial input and the automatic construction of incremental versions of programs.

5.1 Partial Evaluation

The specialization of programs on partial input specifications is known as *partial evaluation* [9,13]. A partial evaluator takes a program and values for some of its inputs, and produces a new, specialized, program parameterized by the remaining inputs, while a *partial evaluator generator* takes a program and produces a transformer which, given some input values, generates the appropriate specialized program, eliminating the need to analyze the original program each time a specialization is needed. General-purpose partial evaluators for imperative languages include Andersen's C-Mix [1] and the system of Baier et al. [3], which process programs written in subsets of C and Fortran, respectively. Osgood [11] addresses the full C language, but the large time and space costs of his algorithm limit its applicability.

Our system can be viewed as a partial evaluator generator because it processes the original program only with respect to particular argument *positions*, generating a program (the cache loader) that later makes use of particular argument *values*. However, our approach differs from traditional partial evaluation strategies in that we are not limited to constructing specialized program text; we also construct specialized data values (such strategies are referred to as *mixed computation* in the literature). Applying a partial evaluator to our driver loop would yield a specialized shading procedure for each pixel in the image, whereas we limit ourselves to constructing specialized intermediate data values for each pixel, which are then processed by a single specialized shading procedure common to all pixels. The traditional approach might achieve better performance because of per-pixel code customization but is impractical due to the huge amount of specialized code that would be generated.

Several researchers have applied partial evaluation to ray-tracing by using a general-purpose partial evaluator to specialize a general intersection routine with respect to a scene, producing a specialized ray-tracer parameterized only by the viewpoint. Mogensen [10] partially evaluated a simple ray-tracer coded in Lisp and got a two-fold speedup; when shading was added to the ray-tracer, the speedup increased to 8 times. Andersen [2] performed a similar experiment in C, and achieved speedups of 1.8 with a code size blowup of 15-90 times. Other researchers used somewhat more application-specific specialization techniques. Hanrahan's "surface compiler" [6] symbolically simplifies input equations defining a general algebraic surface, then outputs a C program that uses numerical methods to find roots of the resulting polynomial. He reported a speedup of 1.3. Goad [5] proposed specializing a hidden surface elimination routine with respect to a fixed scene and variable viewpoint. In the area of mixed computation, Sequin [14] used ray tree caching to significantly speed up shading computations for a ray traced image.

5.2 Incremental program execution

We know of two main approaches to efficiently handling incremental changes. One strategy encodes the program as a set of constraints, and explicitly reestablishes the constraints after each input change. Attribute grammars [4] are one example of a constraint system with an efficient incremental solver. Constraint systems are common in graphics; however, they are primarily used for their convenience and generality rather than as a foundation for efficient incremental execution.

Most other approaches are similar to ours in that they cache intermediate results of an existing batch program. Recent research on memoization for functional programs includes improved caching strategies [12] and methods for memoizing specialized functions instead of data values [15]. There is also research on adding dependence information to existing imperative code [8]. Incremental execution techniques are general, but suffer in performance due to the need to dynamically process arbitrary input changes. Our restricted application domain, in which the user performs a series of repeated modifications to individual scalar inputs, makes it practical for us to perform the dependence analysis and caching transformations statically, and to rebuild the intermediate value cache using batch rather than incremental methods.

6. Conclusion

We have described an automatic method for improving the performance of user-written shading routines by specializing them with respect to individual control parameters. By precomputing, caching, and reusing intermediate results that will remain constant as the user experiments with a single control parameter, the specialized shaders achieve up to a factor of 95 improvement in performance, allowing even complex shaders to be used in an interactive environment.

We plan to extend this work in several ways. The analyses and transformations will be extended to handle a larger subset of the C language, and to perform additional optimizations such as associative rearrangement of expressions and cost-based allocation of cache storage. Specializing the mathematical library routines (as opposed to only their user-level clients) may also achieve a significant gain in performance.

7. Acknowledgements

The authors would like to thank Stephen Coy for the use of his ray tracer and Greg Kusnick, Daniel Ling, Ellen Spertus, and the reviewers for helpful comments on previous drafts of this paper.

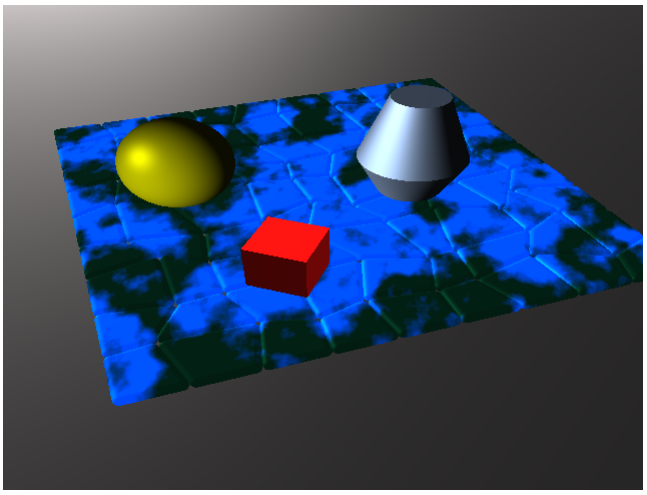

Plate 1: Test image.

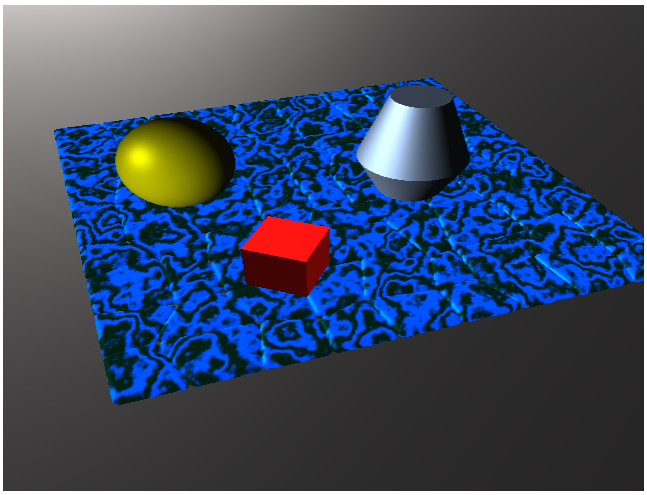

Plate 2: Image obtained by reshading image of Plate 1. This took 12 seconds unspecialized, 0.5 seconds specialized.

Bibliography

[1] Andersen, Lars Ole. Self-applicable C Program Specialization. Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (San Francisco, California, June 12-20, 1992). Yale University technical report YALEU/DCS/RR-909, 1992, 54-61.

[2] Andersen, Peter Holst. Partial Evaluation Applied to Ray Tracing. Unpublished manuscript, October 1994.

[3] Baier, Romana, Robert Glück, and Robert Zöchling. Partial Evaluation of Numerical Programs in Fortran. Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (Orlando, Florida, June 25, 1994). University of Melbourne technical report 94/9, 1994, 119-132

[4] Demers, Alan J., Thomas Reps, and Tim Teitelbaum. Incremental Evaluation for Attribute Grammars with Application to Syntax-directed Editors. Proceedings of the Eighth Annual ACM Symposium on Principles of Programming Languages (Williamsburg, Virginia, January 1981), 105-116.

[5] Goad, Chris. Special Purpose Automatic Programming for Hidden Surface Elimination. Proceedings of SIGGRAPH 82 (Boston, Massachusetts, July 26-30, 1982). In *Computer Graphics* 16, 3 (July 1982), 167-178.

[6] Hanrahan, Pat. Ray Tracing Algebraic Surfaces. Proceedings of SIGGRAPH 83 (Detroit, Michigan, July 25-29, 1983). In *Computer Graphics* 17, 3 (July 1983), 83-90.

[7] Hanrahan, Pat and Jim Lawson. A Language for Shading and Lighting Calculations. *Computer Graphics* 24, 4 (August 1990), 289-298.

[8] Hoover, Roger. Alphonse: Incremental Computation as a Programming Abstraction. Proceedings of the SIGPLAN '92 Conference on Programming Language Design and Implementation (San Francisco, California, June 1992), 261-272.

[9] Jones, Neil D., Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice-Hall, 1993.

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

[11] Osgood, Nathaniel David. PARTICLE: an Automatic Program Specialization System for Imperative and Low-level Languages. Master's thesis, MIT, September 1993.

[12] Pugh, William and Tim Teitelbaum. Incremental Computation Via Function Caching. Proceedings of the Sixteenth Annual ACM Symposium on Principles of Programming Languages (Austin, Texas, January 1989), 315-328.

[13] Ruf, Erik. Topics in Online Partial Evaluation. Ph.D. thesis, Stanford University, April 1993. Published as Stanford Computer Systems Laboratory technical report CSL-TR-93-563, March 1993.

[14] Sequin, Carlo H. and Eliot K. Smyrl. Parameterized Ray Tracing. Proceedings of SIGGRAPH 89 (Boston, Massachusetts, July 31-August 4, 1989). In *Computer Graphics* 23, 3 (July 1989), 307-314.

[15] Sundaresh, R.S. and Paul Hudak. A Theory of Incremental Computation and Its Application. Proceedings of the Eighteenth Annual ACM Symposium on Principles of Programming Languages (Orlando, Florida, January 1991), 1-13.

[16] Upstill, Steve. *The RenderMan Companion.* Addison-Wesley, 1989.

[17] Watkins, Christopher D., Stephen B. Coy, and Mark Finlay. *Photorealism and Ray Tracing in C.* M&T Books, 1992

[18] Watt, Alan and Mark Watt. *Advanced Animation and Rendering Techniques*. ACM Press, 1992.