RenderMan Compiler for the RPU Ray Tracing Hardware Architecture

Tomasz Węgrzanowski

Eidesstattliche Erklärung

Hiermit erkare ich an Eides Statt, dass ich die vorliegende Arbeit selbständig verfasst und außer den angegebenen keine weiteren Hilfsmittel verwendet habe.

Saarbrücken den 30. September, 2006

Tomasz Węgrzanowski

Table of Contents

Abstract

RPU Ray Tracing Architecture promises radical improvement in photorealism and performance of real-time computer graphics by shifting paradigm from rasterization to recursive ray tracing. This requires shaders different and more complex than shaders used by current generation of GPUs, and effective means of building them. RenderMan Shading Language is a de-facto industry standards for programming shaders on the high end. A compiler which translates a subset of RenderMan Shading Language into RPU assembly was developed. Also developed were a virtual machine emulating RPU and a comprehensive test suite verifying correctness of the compiler.

1 Introduction

1.1 Real-time photorealistic computer graphics

Real-time photorealistic computer graphics is highly computationally intensive task, and general-purpose CPUs do not provide sufficient power to fulfill graphical needs of computer games and other applications. Special hardware is used to enhance graphical capabilities of computers since as early as the 1970s [WIKI]. The hardware was initially very primitive and accelerated only simple 2D operations. It gradually advanced and by the mid 1990s fixed-pipeline rasterization-based GPUs became the standard for majority of personal computers.

The rendering pipelines became more flexible with addition of vertex and pixel shaders (Figure 1.1) – simple programs that have limited control over the rendering process. Even relatively simple shaders can affect photorealism much more than increase in scene geometric complexity. Initially the shaders were only able to run identical computations for each pixel (or vertex). Limited support for conditional execution and control flow was added later, however advanced features like recursive function calls are still not supported.

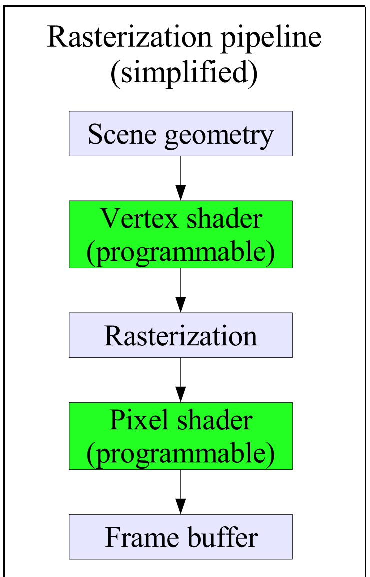

Rasterization pipeline (simplified)
Scene geometry
Vertex shader (programmable)
Rasterization
Pixel shader (programmable)
Frame buffer

Figure 1.1

As of 2006 the paradigm is still rasterization one triangle at a time. Rasterization is a purely local computation and does not require access to the complete scene information. This feature makes it possible to achieve high parallelism. On the other hand lack of access to the complete scene makes it very difficult and expensive to compute global effects like shadows and reflections. Another problem of rasterization is that as each triangle is processed separately, the computation time is linearly proportional to scene complexity.

Another common approach to photorealistic 3D graphics – one that dominates at the high end – is recursive ray tracing. In ray tracing instead of rendering single triangle against the complete screen, a single ray is rendered against whole scene. This approach has advantage of handling global effects much more easily and of average logarithmic complexity in number of scene elements. Ray tracing also provides greater opportunities for scalability, as rays are completely independent from each other. The main disadvantage of ray tracing is lack of efficient hardware support. Fully software ray tracing has insufficient performance for real time computer graphics.

1.2 RPU Ray Tracing Hardware Architecture

RPU [RPU05] is a project with goals of developing hardware for real-time 3D graphics based on ray tracing that can in the long term surpass the current design of GPUs in photorealism and performance. The main differences in architecture compared to mainstream GPUs are using hardware-accelerated ray tracing instead of rasterization, and providing support for significantly more advanced shaders.

Shaders in RPU have much greater control over the rendering process. Shaders can have arbitrary control flow, including recursive function calls. They can also shot additional rays to compute shadows, reflections, transparency, and other effects that are difficult to achieve using rasterization.

This new power requires adequate support for specifying shaders. Programing them in RPU-specific assembly language is difficult and very error-prone. As RPU is a novel architecture and does not use any standard instruction set, no existing compiler could be easily adapted to target RPU, and a new compiler had to be developed from scratch.

1.3 RenderMan Shading Language (RSL)

The issue of writing highly complex shaders has already been dealt with in high end rendering systems. The de-facto standard for programming high-end shaders is RenderMan Shading Language (RSL) developed by Pixar. One of design goals of RenderMan was making it "future-proof" and not tying it to any specific technology. Since its publication in 1989 it became supported by multiple renderers, and RSL become the most commonly used shading language.

The RenderMan Interface Specification [RISPEC] describes the rendering model in terms of ray-tracing, what makes sure that it will fit systems based on ray tracing well. It was originally aimed at software renderers using either ray-tracing or REYES algorithm.

Some alternative shading languages have been recently developed for programming GPU shaders, including OpenGL Shading Language, NVIDIA's Cg, and Microsoft's HLSL. They were not investigated, but it is possible that replacing the parser and minor changes to the compiler can make it support some useful subset of these languages.

1.4 RSL compiler for RPU

The shaders for RPU were initially coded by hand in assembly. Programming and especially debugging in low-level languages is very difficult. Additionally, the shaders are more complex than traditional GPU shaders and even more effort is required to obtain highly efficient code. Together these reasons make it necessary to use a high level language for writing shaders.

As a de-facto industry standard for high-level shader programming, RenderMan Shading Language is a natural choice for RPU. However, support for full RenderMan Shading Language was considered infeasible.

First, many features of RSL are very difficult to implement in hardware-based solution. Some of them are string processing, which includes POSIX-compatible regular expressions, string building by functions like `concat` and `format` (what would require memory management) and I/O using `printf`. Displacement shaders that can affect scene geometry would also be very difficult to support without modifying the hardware. The second reason for not supporting full RSL is its complexity. Reasonably complete RSL compiler would take too much work, especially since compiling to hardware is significantly more difficult than a software implementation.

Relatively small subset of RenderMan Shading Language features – functions, surface shaders, scalar and vector operations and basic standard functions – is sufficient to achieve the project's goals of providing simple to use way of programming complex shaders.

Extending range of supported features is potential scope of future work. It would make it possible to use existing RSL shaders with fewer modifications, and possibly even obtaining full compatibility, most likely by software fallbacks in case of features that cannot be efficiently implemented in hardware.

1.5 Overview

As compiler debugging without special hardware support can be very difficult, a virtual machine that emulated RPU was implemented before work on the actual compiler started.

The compiler takes RSL files that contain functions and shaders as input and generates assembly code as output. An assembler program then generates a form suitable for the virtual machine. An alternative assembler program can also target the actual hardware.

2 RPU Ray Tracing Hardware Architecture

RPU ray tracing hardware architecture can be conceptually decomposed into two parts. One part is responsible for computing ray/scene intersections, and for all our purposes it can be considered a black box. The other part is a highly parallel processing unit that executes shaders. It consists of large numbers of simple processors which concurrently run multiple threads.

Processors are not independent, but grouped in fours. Every such group executes a “chunk” of 4 threads. Processors in every group are synchronized and execute identical instructions. If threads in a chunk become desynchronized by taking different branches of a conditional jump, the chunk is divided. When processors execute incomplete chunk, only some of them are active, while others stay idle. Divided chunk is reassembled on function return instruction.

Many such chunks are available at the same time. When execution of current chuck cannot proceed, usually because its next instruction waits for earlier instructions, memory loads, or ray tracing to finish, group of processors switches to a different chunk. Switching does not cost even a single cycle. Switching guarantees very high utilization of the hardware. Dependencies between instructions are statically calculated by assembler, and there is no out-of-order execution, speculative execution, branch prediction etc.

Threads in a chunk typically represent rays for consecutive pixels. For good performance, they should avoid desynchronization and if it is necessary resynchronize early. For typical scenes synchronization rates are very high, however desynchronization can significantly harm performance if scene has very high geometric complexity or shaders have multiple branches.

Parallelism of RPU is highly transparent, and it does not need to be explicitly considered to efficiently program shaders. Even a very simple model of unpipelined sequential processor gives a reasonable measure of shader performance – as switches do not introduce pipeline stalls, number of instruction per processor per cycle executed does not significantly depend on code, assuming sufficient performance of memory and ray tracing units.

Certain performance issues are not predicted by such simple model, including expensiveness of branching and, in scenes with high geometric complexity, trace instruction due to possible desynchronization, and need to encapsulate code with multiple branches in functions to take advantage of resynchronization at function return.

2.1 RPU rendering model

In the RPU system a new thread is generated for each pixel. Groups of four neighboring pixels are assigned to the same chunk. Typically the main shader for a pixel will generate "primary" ray, and run the shader associated with the triangle hit by the ray. This shader can in turn generate "secondary" rays. Full control of the process is in the shaders, so a different path can be followed. For example pixels inside program's interface area could get their colors from a static texture instead of tracing rays.

2.2 Instruction set

The instruction set (Tables 2.1, 2.2) is modeled after current GPUs. All instructions can operate on 4-element vectors (xyzw). It is also possible to split an operation into two independent parts – doing either 3/1 (one operation on xyz, another on w) or 2/2 split (one operation on xy, another on zw). This is usually very efficient, as the most common data types in computer graphics are 1- (scalars), 3- (points, vectors, colors), and occasionally 2- (2D point and vectors, pixel and texture coordinates), or 4- (rows of transformation matrices) -element vectors of floating point numbers.

Basic operations like component-wise copying, addition and multiplication are available. Other operations include conversions to and from integers, component-wise extraction of fractional part, fused multiply-and-add, multiple kinds of dot product operations, memory load and store, texture load and store, conditional jump, recursive call, function return, and trace instruction that shots a ray and finds a point where it hits the scene.

All operands can be modified by multiplying by ±0.5, ±1, ±2 or ±4, and by arbitrary rearrangement (swizzling) of components. It is possible to limit the components modified in the target register by using a write-back mask.

Instructions can have modifiers `rcp`, `rsq` and `sat`. `rcp` and `rsq` compute a respectively reciprocal and reciprocal of square root of the result's w component, and save it to the special S register. They are commonly used in scalar division and square root computations, and together with dot product opcodes to compute vector length or normalize it. `sat` modifier clamps result to [0,1] range before writeback. It is not currently used by the compiler due to lack of a construction in the RenderMan Shading Language that can be easily compiled to it (`clamp` function can be made more efficient using `sat` sometimes, but it is difficult to do that automatically).

Integer instructions (addition, multiplication, bit shifts and logic instructions) are supported in the fourth (w) component. As RSL does not have integers at all, the compiler does not use such operations.

Floating-point Arithmetics Instructions

(.i notation means computing each component independently)

Opcode	**Meaning**
`mov` R_{dest} , R_{src1}	$R_{dest}.i \leftarrow R_{src1}.i$
`frac` R_{dest} , R_{src1}	$R_{dest}.i \leftarrow \mathrm{frac}(R_{src1}.i)$
`add` R_{dest} , R_{src1} , R_{src2}	$R_{dest}.i \leftarrow R_{src1}.i + R_{src2}.i$
`mul` R_{dest} , R_{src1} , R_{src2}	$R_{dest}.i \leftarrow R_{src1}.i \cdot R_{src2}.i$
`mad` R_{dest} , R_{src1} , R_{src2} , R_{src3}	$R_{dest}.i \leftarrow R_{src1}.i \cdot R_{src2}.i + R_{src3}.i$
`dp2h` R_{dest} , R_{src1} , R_{src2}	$\begin{aligned} v &\leftarrow R_{src1}.x \cdot R_{src2}.x + R_{src1}.y \cdot R_{src2}.y \\ &+ R_{src1}.z \\ R_{dest} &\leftarrow (v, v, v, v) \end{aligned}$
`dp3` R_{dest} , R_{src1} , R_{src2}	$\begin{aligned} v &\leftarrow R_{src1}.x \cdot R_{src2}.x + R_{src1}.y \cdot R_{src2}.y \\ &+ R_{src1}.z \cdot R_{src2}.z \\ R_{dest} &\leftarrow (v, v, v, v) \end{aligned}$
`dp3h` R_{dest} , R_{src1} , R_{src2}	$\begin{aligned} v &\leftarrow R_{src1}.x \cdot R_{src2}.x + R_{src1}.y \cdot R_{src2}.y \\ &+ R_{src1}.z \cdot R_{src2}.z + R_{src1}.w \\ R_{dest} &\leftarrow (v, v, v, v) \end{aligned}$
`dp4` R_{dest} , R_{src1} , R_{src2}	$\begin{aligned} v &\leftarrow R_{src1}.x \cdot R_{src2}.x + R_{src1}.y \cdot R_{src2}.y \\ &+ R_{src1}.z \cdot R_{src2}.z + R_{src1}.w \cdot R_{src2}.w \\ R_{dest} &\leftarrow (v, v, v, v) \end{aligned}$

Table 2.1

Other Instructions

(integers operations omitted)

Opcode	**Meaning**
`jmp label`	Jump to `label`
`call label push n`	Call a function, address of which is `label`
`call` R_{addr} `push n`	Call a function, address of which is in R_{addr}
`return`	Return from a function
`trace` R_{orig} , R_{dir} , R_{arg}	Shot a ray from R_{orig} in direction R_{dir} with parameters R_{arg}
`load` I_i , $A.j$, `n`	Load from memory address $A.j+n$ to I_i

`load4` $A.j$, `n`	Equivalent of:
	`load` I_0 , $A.j$, `n`
	`load` I_1 , $A.j$, `n+1`
	`load` I_2 , $A.j$, `n+2`
	`load` I_3 , $A.j$, `n+3`

`store` $A.j$ `,  n,` R_s	Store R_s under memory address $A.j+n$

Table 2.2

Control instructions (`jmp`, `call`, `return`) can be executed conditionally by pairing them with a regular instruction. Such pairing has form "`primary_instruction + control_instruction condition`":

```
add R15, R3, R5 + jmp label and xyzw (<0)
```

What means jump to `label` if all components of R3+R5 are negative.

2.3 Registers

The register set (Table 2.3) consists of 16 general-purpose registers, named R0 to R15, 32 constant registers C0 to C31, hardware stack – 8 top elements of which are accessible (S0 to S7), and a few special registers – S for storing results of rcp/rsq operations, I0 to I3 for memory input, A for memory addresses, HIT, HIT_TRI and HIT_OBJ for storing results of trace operation.

All registers except for HIT_TRI and HIT_OBJ are 4-element floating point vectors. Address (A) and special (S) register are also vectors, however as operations they in which they are used are inherently scalar, they are usually treated instead as groups of 4 independent scalar registers.

Registers

General purpose	R0 – R15
Constant	C0 – C31
Stack frame	S0 – S7
Special	S.x – S.w
Memory Input	I0 – I3
Address	A.x – A.w
Hit information	HIT HIT_TRI HIT_OBJ
Program Counter	PC

Table 2.3

By common assembly convention address of currently executed instruction is treated as a special "PC" (Program Counter) register, even through it is not a real directly accessible register.

2.4 Instruction execution

Instructions set of RPU is modeled after GPU shader assembly languages. Compared with most modern general-purpose architectures, instructions are more complex and have more stages, like swizzling, premultiplication and optional clamping.

Disregarding 3/1 and 2/2 instruction pairings for simplicity, execution of a single instruction proceeds as follows:

- Contents of each requested register is read. Scalar constant can be specified as a source instead of a register, in which case it is loaded into all 4 components.
- Components are rearranged (swizzled). Any component can take any other component as input. There are 256 possible ways to swizzle a source. Components cannot be swizzled between different sources. For example yzxz swizzling mask means that:
    - x component gets value of y,
    - y component gets value of z,
    - z component gets value of x,
    - w component gets value of z.

As a notational convention, if fewer than four components are specified, the last one is duplicated to fill the remaining positions, so xy mask is equivalent to xyyy. Not giving a swizzling mask is equivalent to swizzling mask xyzw (no rearrangement).

- For every source, all four components are multiplied by ±0.5, ±1, ±2 or ±4 (by adding -1, 0, 1 or 2 to the floating point exponent and possibly changing the sign). All components must use the same multiplier.
- The actual computation is performed.
- If `_sat` modifier is specified, the result is clamped to [0,1] range.
- The components of the result specified in the writeback mask are saved to the target register.
- If `_rcp` or `_rsq` is specified, reciprocal or respectively reciprocal of square root of result's fourth (w) component is computed and saved to special S register using the same writeback mask. Typically the writeback mask has only one component on, so
- If secondary operations (most commonly conditional jump, but also call and return) was paired with the operation, individual components of the result are compared with 0 and 1. If all (or any – depending on a flag) components selected in a condition selection mask are in correct ranges, the secondary instruction is executed. Example of such condition is `or xy (>=0 and <1)`, which means "either x or y component of the result is within [0,1) range".

This instruction set is very powerful, however it is very challenging for compiler or programmer to use it efficiently. As an example, cross product can be encoded in just 2 instructions:

```
mul R0.xyz, R1.yzx, R2.zxy
mad R0.xyz, -R1.zxy, R2.yzx, R0
```

Let's demonstrate execution by computing cross product of R1=(1,2,3,0) with R2=(4,5,6,0).

- First operation:
  - Sources are read. Source 1 (R1) is $(1,2,3,0)$, source 2 (R2) is $(4,5,6,0)$
  - Sources are swizzled to $(2,3,1,1)$ and $(6,4,5,5)$.
  - There is no negation or multiplication.
  - Sources are multiplied component-wise giving $(12,12,5,5)$
  - First three components of the result $(12,12,5)$ are saved to R0
- Second operation:
  - Sources are read. Source 1 (R1) is $(1,2,3,0)$, source 2 (R2) is $(4,5,6,0)$, source 3 (R0) is $(12,12,5,0)$
  - Sources are swizzled to $(3,1,2,2)$, $(5,6,4,4)$, and $(12,12,5,0)$.
  - The first source is negated, the other two are not modified, giving $(-3,-1,-2,-2)$, $(5,6,4,4)$ and $(12,12,5,0)$
  - Sources are multiplied and added component-wise, giving $(-3,6,-3,-8)$.
  - First three components of the result $(-3,6,-3)$ are saved to R0. This is the computed cross product of $(1,2,3)$ and $(4,5,6)$.

The fourth component was not useful in the computations. This is a very common situation, as 3-element vectors are the most popular data type in computer graphics. For that reason those two instructions would most likely be paired with scalar computations using 3/1 split (first instruction deals with xyz, the second with w), resulting in even better performance. As neither the compiler nor the virtual machine supports such pairings, the subject will not be elaborated upon.

2.5 Call stack

As RPU supports recursive function calls, it needs some kind of a call stack. The stack design is a hybrid of SPARC-style register windows and regular memory stack with register-based argument passing.

The call stack is best presented as a pair of synchronized stacks – the control stack and the data stack. The control stack contains return addresses and shift sizes, and the data stack contains the actual data. Only an 8-element "register window" of the data stack is accessible for reading and writing (Figure 2.1). The register window affects only "stack registers" and does not affect the regular registers (R0–R15).

There are two operations that control the stack:

- Function call – `call f push n` – pushes address of the next instruction and the shift value `n` onto the control stack, shifts the stack register window by `n`, and jumps to the requested label `f`.
- Function return – `return` – based on the the top entry of the control stack, register window is restored, and program counter is set to return address. Then top entry of the control stack is popped.

Changes in the stack after two function calls, with shifts of 3 and 2.

Shaded cells are not accessible.

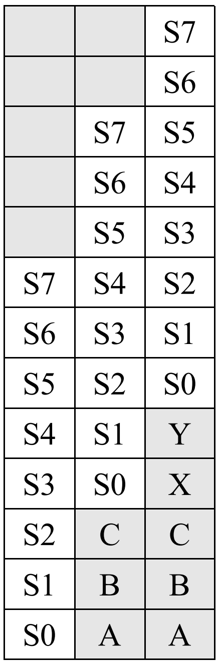


Figure 2.1

The reason SPARC-style register windows are not used is that register file needs 3 read ports (for `mad` instruction), and big register file for stack together with 3 read ports would be very expensive. The stack file has only one read port, that is only one value can be read from it in a single cycle – so it is impossible to add two values on a stack without copying one of them to a regular register first.

As there are no push/pop instructions, it is impossible to use the stack for register spilling if there are more variables than registers, like most other compilers do. Physical size of the stack is not known. In case of stack overflow, part of the stack is saved to the memory, and restored when it is needed. Therefore the stack behavior is identical to that of infinitely big stack, and RPU programs do no need to worry about stack size.

Such stack design forces function arguments and return values to use regular registers, as the visible window of only 8 registers can easily be too small for both arguments and saved values.

2.6 Limitations

The shaders can be rather complex, however the architecture still has many limitations. The most apparent is very low-precision (24 bit) and lack of IEEE 754 compatibility of floating point operations. While some parts of the IEEE 754 standard like alternative rounding modes, exceptions, and denormals are rarely needed by shaders, the hardware currently does not even have a fully reliable way to compare floating point numbers for equality, what can be a major problem.

The computations must be structured in a way that keeps shaders of neighboring pixels execute the same code most of the time. If the control flow of individual pixels is highly independent, performance will suffer. Ray tracing typically keeps the synchronization at high level.

3 RenderMan Interface

RenderMan Shading Language is part of RenderMan Interface Specification, current version of which (3.2) was published by Pixar in September 1989 (minimally revised version 3.2.1 was published in November 2005). One of the goals of the specification was being “future-proof”, and not being bound to any rendering program or algorithm. It fully achieved this goal, still being a de-facto standard in 2006, 17 years after publication, in spite of great changes in computing and computer graphics that took place during that time.

3.1 RenderMan rendering model

RenderMan rendering model (Figure 3.1) can be best described in terms of ray tracing.

A ray is shot into the scene, and where it intersects with element (triangle) of the scene, surface shader of this element is called. The surface shader can request information about local illumination, what can result in one or more light shaders being called. It can also shot additional rays that automatically call surface shaders when they hit another scene element. If space through which a ray travels has a volume shader associated with it, a volume shader will be called to modify the ray color. After the original ray returned with color information, imager shader can modify it before the pixel color is finally set.

Displacement shaders control shape of scene elements and do not have direct equivalent in the ray tracing model.

RenderMan rendering model

Data flow represented by arrows. Displacement shaders are separate from the main rendering pipeline.

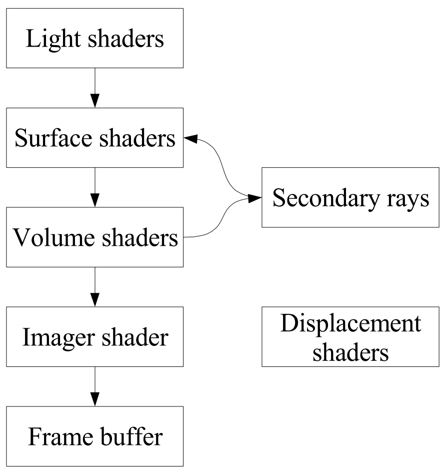


Figure 3.1

3.2 RenderMan Shading Language

RSL source files consist of six kinds of subroutines:

- functions – routines that compute and return values based on their arguments. They can be used by all types of shaders and by other functions.

- surface shaders – the most important type of shaders, they specify color of light that arrives from a point on a surface. Each surface has an associated surface shader. Surface shaders can also start new rays using RSL function `trace`.
- light shaders – the second most important type of shaders, they specify color of light coming from light sources. They are used by surface shaders to calculate incoming light.
- volume shaders – specify what happens to the light passing through a volume, be it atmosphere or inside of an object.
- imager shaders – convert incoming light information to final pixel values. They can be used for techniques like high dynamic range imaging.
- displacement shaders – can displace and change normals of surface. Normal modification is straightforward to implement in most rendering algorithms, and is used by bump mapping. Displacement support is more difficult to implement and it marked optional by the specification.

RSL syntax is similar to C. The supported types are floating point numbers, strings, colors, 3D vectors, points, and normals, and 4x4 matrices. Points, vectors and normals can be freely mixed. Colors have 3 components by default, but according to the specification different number can be used. This feature is not supported, as it would be difficult to implement it efficiently in hardware.

The supported operations include addition (`+`), subtraction (`-`), multiplication (`*`) and division (`/`) and also two graphics-specific operations – dot product (`.`) and cross product (`^`). Many other operations are implemented as standard library functions.

Flow can be controlled by conditional execution (`if/else`) and loops (`while/for`). RSL supports `break/continue` statements. `break/continue` can be followed by a number which indicates that they refer to a loop other than the innermost. `return` statement can be used to return values from functions, but cannot be used inside shaders. Shaders take arguments and return values using special variables.

There are also `illuminance/illuminate/solar` loop-like control structure that describe scene illumination.

A simple surface shader can be seen on Figure 3.2.

A matte surface shader in RSL

```
surface matte(
  float Ka = 1;
  float Kd = 1;
)
{
  normal Nf = faceforward(normalize(N), I);
  Oi = Os;
  Ci = Os * Cs * (Ka*ambient() +
                  Kd*diffuse(Nf));
}
```

Figure 3.2

3.3 Mapping between RenderMan shading model and the hardware

Rendering models of RenderMan and of the RPU hardware are significantly different. In RenderMan shaders are never called explicitly, both primary ray and secondary rays shot by `trace` instruction call appropriate shaders behind the scenes. RPU performs only explicit shader calls. In similar way, primary rays are shot implicitly in RenderMan, but are controlled by a shader in RPU.

RenderMan has six kinds of shaders that need to be mapped to hardware shaders. The most kind are displacement shaders. As scene geometry in RPU is not controlled by shaders, displacement shader would need to be run before scene is sent to the hardware.

Other shader types are more straightforward. Primary rays generation and imager shader together become “main” shader in RPU. Both primary rays and secondary rays shot by RenderMan `trace` instruction are converted to RPU `trace` followed by call to shader associated with the hit object. RenderMan surface shaders are compiled to RPU shaders associated with scene objects.

Volume shaders could be implemented by replacing volumes by their boundaries, and volume shaders by surface shader of the boundaries. This solution is not without problems. In the most common case, boundaries of volume are determined by scene polygons, which already have own surface shaders. When such boundary is hit, two shaders must be run – first shader associated with the volume, then surface shader of the polygon. The opposite side of the polygon can be associated with a different volume shader. In RPU every object can be associated with at most one shader. Therefore to support volume shaders we would need to compile a special shader that checks which side was hit, calls volume shader associated with the side if any, and then calls the proper surface shader.

Light shaders can be compiled to functions. Surface shaders requesting information on illumination would then call such functions. As number and parameters of light sources are not known, a global list of all light sources would be required, and functions associated with them would be called in a loop and their results added.

4 Virtual machine

Due to lack of access to the actual hardware, a virtual machine has been implemented. The virtual machine is an Objective Caml library. Hardware state is represented by:

- register file
- call stack
- code memory
- data memory

Separating code and data memory makes it possible to represent opcodes in convenient form instead of decoding them from binary format at each operation. The virtual machine contains procedures that execute shaders, and procedures that perform ray tracing.

4.1 Ray tracing in the virtual machine

Ray tracing in the virtual machine supports only scenes consisting of triangles organized into a hierarchy of axis-aligned bounding boxes. The memory layout of the scene is not compatible with the hardware memory layout, however this does not cause any compatibility problems, as the generated code treats ray tracing as a black box and does not need to know anything about how it is implemented.

Memory layout of data associated with scene elements is not used by either hardware ray tracing unit or virtual machine ray tracing procedures, so the compiler can use arbitrary layout as long as it is consistent with the application that generates the scenes.

The algorithms used by virtual machine ray tracing are:

- Intersection tests between rays and axis-aligned bounding boxes
- Intersection tests between rays and triangles
- Algorithm that computes intersections between rays and the scene using the two aforementioned tests
- Algorithm that preprocesses the scene by generating a hierarchy of axis-aligned bounding boxes (it does not run inside the virtual machine, the scene needs to be preprocessed before)

4.2 Virtual machine drivers

As the virtual machine itself is a library, driver programs are needed to actually use it. There are multiple driver programs for the virtual machine. In addition to sharing the VM library they also share code that loads programs and scenes to memory, saves PNM images as output etc.

The drivers are:

- 9 test drivers for virtual machine testing. They generate 22 test images to verify that virtual machine works. The VM is further verified by compiler tests.
- A test driver for testing individual functions. Initial register state is given on command line, and the final register state is printed out when the virtual machine exits.
- A test driver for testing shaders. Image size and initial register state is given on command line, and the image (in PNM format) is printed to the standard output.

5 Compiler

The compiler consists of two executables (Figure 5.1). The auxiliary executable is an Objective Caml program that parses the source and generates internal representation, but without converting it to SSA form.

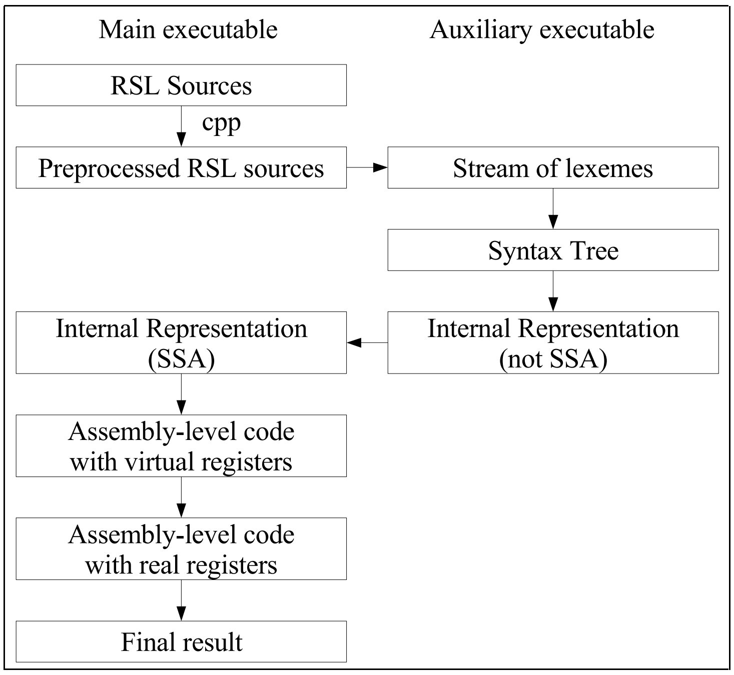
The main executable is a Ruby program that calls cpp preprocessor (optionally), calls the auxiliary executable, and does the actual compilation.

5.1 Compilation process

Main executable of the compiler reads its output, converts it to SSA form, optimizes it, converts it to low-level instruction graph with virtual registers, runs further optimizations, allocates registers, reschedules instructions, converts the code to the final assembly form and prints it out.

5.2 Parsing

The parsing is performed using lexer and parser automatically generated from grammar descriptions. The lexer is generated by ocamllex, and the parser by ocamlyacc. The programs are Objective Caml equivalents of the popular C lex/yacc tools.


Main executable
Auxiliary executable
RSL Sources
cpp
Preprocessed RSL sources
Stream of lexemes
Syntax Tree
Internal Representation (SSA)
Internal Representation (not SSA)
Assembly-level code with virtual registers
Assembly-level code with real registers
Final result
Figure 5.1

The abstract syntax tree generated by parsing is then expanded. The expansion algorithm is very straightforward, using depth-first traversal of expression trees, starting from the left child. The most difficult part is assigning types to nodes of function arguments and return values. Full compatibility with RenderMan would require significant complexity (user functions can be polymorphic in both arguments and return values), and the exact algorithm is not described in the specification. Therefore an ad-hoc solution is implemented with special cases for each polymorphic standard library function, and user functions cannot be polymorphic.

5.3 Function inlining

In all modern architectures function calls incur significant overhead. Sources of the overhead include:

- Pipeline stalls due to the call and again due to return.
- Moving function arguments to designated registers and fetching the return value. Stack is often in memory and significantly slower than registers.
- Saving and restoring registers across the function call.
- Inability to optimize across function calls. Even if the computations in the called function could be optimized based on the arguments, it will not be possible without complex inter-procedural optimizations.

The overhead is especially significant for very short functions, where it can be as much as an order of magnitude higher than cost of the actual computations. Overhead in RPU is relatively low, as there are no pipeline stalls (hardware switches to a different thread instead of stalling), arguments and return value are in fast registers, and computations can write directly to argument registers and read from return value register instead of using additional move instruction. Overhead of saving and restoring registers and inability to optimize across function calls are main sources of overhead.

Important part of the overhead is currently suboptimal code generation for function calls. Moving computations across the call could reduce number of registers that need saving, some variables could be assigned to stack registers instead of general registers, and if multiple functions are called, variables that are not used between calls should stay on stack instead of being copied to general registers and back to the stack. Certain ABI changes could also reduce overhead (more in section 7.3 ABI and performance).

The most common strategy for dealing with this overhead is inlining short functions. In inlining, instead of calling a function, all computations it would perform are copied directly into the caller. Inlining introduces a space-time tradeoff:

- Inlined functions save constant time per call, due to reduced overhead.
- Inlined functions save constant space per call, saving instructions to manage arguments, return values, save and restore variables, and perform the actual call.
- Inlined functions cost space proportional to function size, to copy the actual computations everywhere it is used.

Results of inlining can only be roughly estimated, as it is impossible to know in advance what optimizations will be made possible by inlining. However, it is possible to talk about two thresholds:

- First threshold – where inlining does not expand code size, as duplicating the function body is simpler than generating function call code. For a function that takes 2 arguments, returns 1 value, and requires 2 variables to be preserved, it can be estimated as 9 assembly operations:
    - 2 `mov`s to save variables
    - 2 `mov`s to to set function argument
    - `call` to call the function
    - `return` to return from the function
    - `mov` to use the return value
    - 2 `mov`s to restore variable

  This estimation is a very imprecise – it is possible that the return value can be used directly without moving it to another variable, or that arguments can be computed directly into the argument registers. Or the other way – function can take more than 9 operations, but it might be possible to optimize some of them away using information available inside the caller.
- Second threshold – where inlining costs some space, but saves enough time to offset it. It is very difficult to estimate this point, as we do not know how many cycles is one byte worth. Different compilers use different thresholds for inlining, often considering criteria different than just code size. For example IBM's AIX C++ compiler 7.0 uses threshold of 20 C++ statements by default. [AIX]

The RSL compiler inlines all standard library functions independent of their size. Most standard library functions are below the first threshold, and even the most complex implemented standard library function (component-wise vector clamp) expands to only 15 opcodes, what is still below the second threshold.

On the other hand on RPU there is one important reason for not inlining. Shaders are executed in chunks of four that should stay synchronized for performance. If synchronization is lost, RPU tries to restore it on function return. Even short functions can have complex control flow and be likely to get desynchronized. If we call such function, synchronization will be restored when it returns. On the other hand if we inline it, synchronization will be lost for much longer (until the caller returns), potentially resulting in performance hit much greater than the savings due to inlining. This consideration does not affect functions like `normalize` that have no branches.

However functions like vector component-wise `clamp` with 6 branches are possibly better called than inlined.

There are two possible ways to inline functions in the compiler:

- *early inlining* – inlining at intermediate representation level
- *late inlining* – inlining at assembly code generation

Early inlining is preferable, as is makes more optimizations possible, in particular common subexpression elimination. The compiler currently does a combination of early and late inlining. Functions that expand to simple operations without side effects are inlined early, and functions with side effects (like `trace`) are expanded late. Inlining of user function is not supported yet, and many functions that should be inlined early are inlined late.

Early inlining together with common subexpression elimination removes many redundant computations – for example calling `length` function of the same vector many times is compiled to just single length computation.

5.4 Compilation of multiple functions

More than one function can be present in a source file (Figure 5.2). In fact separate compilation is currently not supported as the virtual machine does not know how to use more than one assembly file (the hardware does not have to keep this limitation), so the only way to include multiple functions and shaders in a program is by having them in a single input file, or in multiple files that are included in a single compilation using a preprocessor.

Example of multiple functions

The following code:

```
float add(float a,b) { return a+b; }
float f() { return add(5,2) * 3; }
```

Gets compiled to:

```
SUBROUTINE_ENTRY_add:
  add R0.w, R0.w, R1.w
  return
SUBROUTINE_ENTRY_f:
  mov R0.w, 5.0
  mov R1.w, 2.0
  call SUBROUTINE_ENTRY_add push 0
  mul R0.w, R0.w, 3.0
  return
```

Figure 5.2

The files are assembled in order, so the default entry point (one that starts at PC=0) is the first function. All entry points of function and shaders are marked by label SUBROUTINE_ENTRY_(function name), and call instructions jump to labels of such form. However if we want to start computation from a function or shader different than the first one specified in the source, we need to know their addresses. The assembler program saves addresses of all entry point labels to source_file_name.epts file. The test driver or other application then reads contents of this file and can use the addresses to set an alternative entry point of the computations (by setting PC register), or by setting the shader address in the scene to the actual address.

Compilation makes only single pass over the sources. This means that functions should be in such order that they are always defined before they are called, or the compiler will not know their types and can compile wrong code. As an example – if 0 is passed as an argument to a function that expects a vector, proper compiler behavior is casting 0 to a vector, and allocating a vector register for this argument. On the other hand if this function was not seen by the compiler, it would incorrectly guess that a scalar argument is expected, and pass the argument in a scalar register. To deal with this problem (and with typos in function names), the compiler generates warnings if an user functions is called before it is defined.

In the long term it would be better to modify compiler architecture to perform multiple passes over the sources. However that would imply complex information flow between two parts of the compiler, what would greatly complicate the interface and make bugs more likely. For that reason it may be more practical to recode the Objective Caml part of the compiler and merge it with Ruby part before switching to multi-pass compilation.

6 Intermediate representation

The intermediate representation of a program in the compiler is based on "three-address code". The program is represented by a graph of instructions. Most of the opcodes at intermediate representation level (Table 6.1) are responsible for computations and they have form:

- operand target, source1, ..., sourceN (1 to 3 sources)

These operations do not have side effects, compute single value, and can be optimized by by common subexpression elimination.

There are also a few other instructions like `nop` (do nothing), function call, function return, and conditional jump.

Conditional jump is the only instruction that has two successors instead of one. There are three supported conditions – greater-equal, greater-equal-0, and equal.

Instructions are not grouped into basic blocks and there are no jump instructions, as each instruction can have any successor.

All variables are typed as float, string, color, vector or matrix. Vectors variables are tagged as point, vector or normal, however this information is currently unused, and is is not obvious what types should results of various vector operations have.

Currently matrices are not supported, and the code generator rejects any input that contains matrix variables.

Opcode	***Semantics***
`nop`	Do nothing
`mov A, B`	$A \leftarrow B$
`neg A, B`	$A \leftarrow -B$

Opcode	***Semantics***
`rcp A, B`	$A \leftarrow 1/B$
`rsq A, B`	$A \leftarrow 1/\sqrt{B}$
`frac A, B, C`	$A \leftarrow \mathrm{frac}(B)$
`add A, B, C`	$A \leftarrow B + C$
`sub A, B, C`	$A \leftarrow B - C$
`mul A, B, C`	$A \leftarrow B \cdot C$
`dp3 A, B, C`	$A \leftarrow \mathrm{dotproduct}(B, C)$
`xpd A, B, C`	$A \leftarrow \mathrm{crossproduct}(B, C)$
`tuple A, B, C, D`	$A \leftarrow (B, C, D)$
`call T, f, A, B, ..., Z`	Call function and assign $T \leftarrow \mathrm{f}(A, B, \ldots, Z)$
`return A`	Exit function or shader returning value of A
`cjmp eq, A, B`	If $A = B$ jump to second successor
`cjmp ge, A, B`	If $A \geqslant B$ jump to second successor
`cjmp ge0, A`	If $A \geqslant 0$ jump to second successor

Table 6.1

6.1 Single Static Assignment form

To simplify optimizations, internal representation is converted to Single Static Assignment (SSA) form [SSA]. In SSA form every variable has exactly one definition.

SSA also requires that if some variable is used it must have been defined. All variables are therefore initialized to 0 or (0,0,0) at beginning of the function (like in Java). This initialization is almost always removed, but it can be a small performance hit in rare cases. It would be possible to avoid this small hit by using special “X = undefined” definitions which are compiled into empty code instead of “X = 0”, so that X would get unspecified value if it was not initialized (like in most C/C++ compilers). This was not done, as performance gain is negligible and it can make debugging significantly more difficult.

It is possible that multiple definitions reach a certain point. In such case special φ-definition are inserted at nodes different predecessors of which are reached by different definitions. A φ-definition at a node defines a variable depending on an edge through which the node was entered.

SSA is used by most modern compilers, as it makes many optimizations simpler, and some advanced optimizations possible. The currently performed intermediate representation level optimizations are copy propagation, opcode simplification and common subexpression elimination.

Algorithm that converts to SSA computes set of definitions of a variable that reach given point. If multiple definitions reach inside a node, but one of its predecessors has only one outgoing definition, then such node gets a new φ-definition. If there is no such node, terminate and replace all references to the variable by references to its only definition that reaches given node.

It is easy to prove correctness of this algorithm. Assume that multiple definitions of a variable reach some node. Let's follow some path through which one of the definitions reached the node. As there is only one definition on the edge going out of the definition's node, and the path ends with multiple definitions, definition count must have increased somewhere. It could not have increased inside a node, as a node can only pass the same definitions from its input to its output, or replaces them all with its own definition. Therefore the definition count must have increased on a node's entry point. As its predecessors on the path had only one definition going out, the algorithm could not have terminated, what breaks the assumption.

Termination proof is also easy. As a node with φ-definition for the variable cannot have multiple definitions reach inside it, we can only put a φ-definition for the variable on a node that does not have one yet, and there is finite number of nodes, so the algorithm finishes in finite number of iterations.

6.2 Copy propagation

Copy propagation is an optimization that removes unnecessary copy operations. It also propagates constants. In addition to making code simpler, it makes further optimizations possible – propagating constants enables the compiler to perform constant folding, and can make new common subexpressions available (for example if copy propagation replaces B by C, A + B and A + C become equivalent).

For sources in SSA form copy propagation is particularly simple:

- For every $\boxed{B \leftarrow A}$ operation in the code:
    - Replace all B by A and drop the operation.

The compiler also treats floating point loads and casts from float to vectors and colors as copy operations.

Correctness proof for this optimization is very straightforward, and it can serve as an example as most SSA-based optimizations can be proven in similar way. The proof only deals with copy propagation, but it can easily be extended to float load and cast propagation.

Let's assume that a function is in SSA form, and there is a node $\boxed{B \leftarrow A}$. If B always equals A at every point where it is used, then the optimization is trivially correct, so let's only consider the case where there is a reachable point where B is used, but where B has different value than A .

As B is by SSA assumption defined at each point where it is used, $\boxed{B \leftarrow A}$ must have been visited at least once. As we assumed that $B \neq A$ it also means that node $\boxed{A \leftarrow \cdots}$ was visited between the last visit to $\boxed{B \leftarrow A}$ and visit to the node that uses B . So it must be possible to reach a node that uses B from $\boxed{A \leftarrow \cdots}$ without going through node $\boxed{B \leftarrow A}$.

As each variable must be defined before it is used, and it is possible to reach $\boxed{B \leftarrow A}$, then it must be possible to reach $\boxed{A \leftarrow \cdots}$ from the entry without going through $\boxed{B \leftarrow A}$.

Taking the two together, as it is possible to reach $\boxed{A \leftarrow \cdots}$ from the entry without visiting $\boxed{B \leftarrow A}$, and it is possible to reach a node that uses B from $\boxed{A \leftarrow \cdots}$ without visiting $\boxed{B \leftarrow A}$, then it is possible to reach a node that uses B from the start node without visiting $\boxed{B \leftarrow A}$. As we assumed that each variable is defined before it is used, we get a contradiction. Therefore, the copy propagation algorithm must be correct.

Analogously to copy propagation algorithm, most analyses and optimizations based on SSA are inherently global and do not consider the control and data flow explicitly. All relevant flow information was already extracted during conversion to SSA form.

One kind of propagation that is not performed yet is propagating φ-definitions all arguments of which are identical. Newly generated φ-definitions always have different arguments, however copy propagations or eliminating unreachable branches can make them identical.

A somewhat contrived example of such situation would be:

```
if (x > 0)
  y = 2;
else
  y = 2;
z = 1/y;
```

which gets compiled to equivalent of:

```
if (x > 0)
  y0 = 2;
else
  y1 = 2;
y2 = φ(y0, y1);
z = 1/y2;
```

What after propagation becomes:

```
if (x > 0)
  {}
else
  {}
y2 = φ(2, 2);
z = 1/y2;
```

And as the conditional has two identical outgoing links it becomes:

```
y2 = φ(2);
z = 1/y2;
```

Simplifying φ-definitions would let the compiler evaluate z as 0.5 at compile time, what currently needs to be done at run time. Forwarding and constant folding at assembly level are not sufficient in this case, as assembly `mov_rcp` instruction has side effects and cannot be easily optimized.

6.3 Opcode simplification and constant folding

Opcode simplification replaces complex opcodes by simpler ones. Constant folding performs computations that can be completed at compile time. The primary reason for these optimizations is not directly improving performance, as in current architecture most opcodes have the same cost, but enabling further optimizations. If a complex opcode can be replaced by a constant load or a variable copy, it can be forwarded and the opcode is then eliminated completely, what reduces computations and decreases register pressure. Branch instructions can be simplified if the condition is always true or always false, what often makes some code unreachable. Such code is then eliminated, and with fewer possible execution paths the compiler is often able to find more invariants and use them in further optimizations.

Opcode simplification and constant folding are run together. Each opcode is optimized individually. It is matched against a set of known patterns. In case of a match, it is replaced by a simpler opcode. Most commonly the replacement is a `mov` operation (constant load or variable copy), which is immediately propagated and dropped.

Among the matched pattens are addition and subtraction of 0, multiplication by 1, 0, or -1, and any computation all arguments of which are constant. Opcode simplification can also remove branches if either condition is known in advance, or both branches point to the same code (with the same φ-arguments).

Example of opcode simplification in action:

```
  y = 1;
  z = x / y;
```

What gets compiled to equivalent of:

```
  y = 1;
  a = rcp(y); /* rcp(y) means 1/y */
  z = x * a;
```

Due to constant propagation is becomes:

```
  a = rcp(1);
  z = x * a;
```

Then by opcode simplification:

```
  a = 1;
  z = x * a;
```

What automatically triggers propagation of `a`:

```
  z = x * 1;
```

And the final opcode simplification:

```
  z = x;
```

And then `x` is propagated everywhere where `z` was, and the whole fragment code is removed.

As a purely local optimization, it does not depend on the code being in SSA form. Similar optimization is performed also on the assembly-level code, however with different rules as the opcodes are different.

Opcode simplification does not preserve behavior if non-finite numbers are present. For example multiplying a variable containing infinity by a constant 0 returns not-a-number without opcode simplification, but 0 with it. Such IEEE 754 incompatible behavior is common in other compilers, like gcc with -ffast-math command line option.

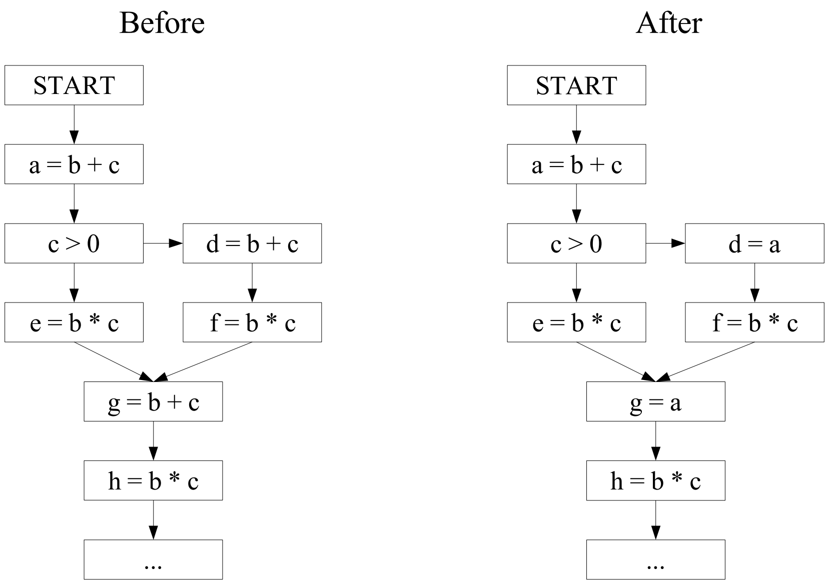
6.4 Common Subexpression Elimination

Common Subexpression Elimination avoids some duplicated computations, by replacing some of the repeated computations by copy operations.

If the same computation is performed multiple times, and one of the points at which the computation is performed is always reached before another point performing the same computation, then the result is forwarded instead of being recomputed. After forwarding the code is simplified and Common Subexpression Elimination is repeated.

Only weak form of Common Subexpression Elimination is implemented (Figure 6.1). Computations are not forwarded unless their results are guaranteed to be in the same register. The algorithm could be modified to handle such cases.

Example of Common Subexpression Elimination



In the example, `b+c` and `b*c` are computed multiple times. Recomputation at nodes `d=b+c` and `g=b+c` can be eliminated as `b+c` is guaranteed to be in variable `a` when they are executed. On the other hand `h=b*c` cannot be simplified, because while `b*c` is always computed before the node is executed, results are not guaranteed to be in single register, and depending on the path they are in either `e` or `f`.

Figure 6.1

6.5 Other optimizations

SSA form makes many advanced optimizations possible. Most of them perform analysis of whole procedure (or whole program), and optimize using information obtained in the analysis. Such optimizations can efficiently deal even with big programs with complicated control flows. The compiler currently implements only a limited number of them, focusing instead on low-level optimizations. This is adequate for small programs with simple control flow, but for long-term viability of the project, it is important to keep an option of easily implementing more advanced optimizations.

Possible SSA-based optimizations include:

- Sparse Conditional Constant Propagation [SCCP] – simultaneously performs Constant Propagation and Dead Code Elimination. In some cases it can be significantly stronger than performing these two optimizations separately.
- Global Value Numbering [GVN] – reduces redundancy by analysis of equivalences

between variables. It removes more redundancy than Copy Propagation and Common Subexpression Elimination.

- Partial Redundancy Elimination [PRESSA] – eliminates expressions that are redundant on some but not all code paths and performs limited code motion. Partial Redundancy Elimination is also possible without SSA [PRE], but is less powerful.

7 Code generation

The assembly-level code is generated in the following way:

- Each node of the program graph generates a small graph with appropriate assembly instructions as nodes.
    - Any edges that lead to φ-definitions are converted into `mov` instructions that perform necessary assignments. This step can introduce new temporary variables in some cases.
    - The links out of the small graphs are represented by placeholder values “next” or “then”/”else” in case of cjmp nodes.
- After every node generated its own graph, theses graphs are connected. Placeholder links are replaced by links to entry instructions of appropriate graphs. This two-phase generation is necessary, as the code can contain cycles in which we must generate nodes first and connect them later.
- Entry instruction of the entry node becomes subroutine's entry instruction.

Late inlining is performed during this phase. A call instruction is replaced by a graph that implements appropriate computations.

Instructions at this level are very close to instructions of the final assembly. The main differences:

- Virtual registers are used instead of physical registers.
- The program is represented as a graph, which will be later converted to linear output.
- `call` opcode at this level handles variable saving/restoring automatically.

7.1 Application Binary Interface

Programs for RPU can consist of multiple parts (shaders, functions) that need to communicate with each other. This requires a well-defined binary interface, which is not fully determined by the hardware.

The most important parts of binary interface are register use patterns, calling convention, and memory layouts of various objects (which are not defined yet).

Registers R0-R14 are divided into vector part (xyz) and scalar part (w). From compiler point of view there are 15 scalar registers (R0.w–R14.w) and 15 vector registers (R0.xyz–R14.xyz), and there is no relation between the two sets.

R15 register is used globally as a “junk” register. All operations that want to discard computed value assign it to one of R15's components. In particular this refers to:

- conditional jumps – comparison can only be implemented as a subtraction, and R15 becomes a target.
- reciprocal and reciprocal of square root computations – opcodes that perform such

computations produce two values. The final value is saved in S register, and the intermediate value is saved in a normal register. Usually only the value saved in S register is relevant, so R15 is used as a target to ignore the intermediate value.

Four components of S register are used as separate special scalar registers. They are used as temporary storage of results of rcp/rsq operations.

HIT, HIT_TRI and HIT_OBJ registers are treated more like global variables than like other registers.

7.2 Calling Convention

Functions take arbitrary number of arguments and return at most one value.

The arguments are moved to predetermined registers before the call. Vector/color arguments are put into registers R0.xyz, R1.xyz and so on up to R14.xyz. Float/string arguments are put into registers R0.w, R1.w and so on up to R14.w. (Figure 7.1)

Vectors have x component in x part, y component in y part and z component in z part. Colors have red component in x part, green component in y part and blue component in y part. Strings are represented by hashes – currently first 24 bits of MD5 hash, however this should be adjusted to match precision of the hardware registers.

Register allocation for:

```
color f(
  vector a;
  color b;
  float c
) { ... }
```

is:

- a – R0.xyz
- b – R1.xyz
- c – R0.w
- return value – R0.xyz

Figure 7.1

Caller is fully responsible for preserving any registers it wants preserved – the called function is free to modify all of them.

Surface shaders take hit information from HIT and HIT_TRI registers, and return the color information (Ci) in R0.xyz and opacity (Oi) in R1.xyz. Details of object memory layout are not specified.

Light shaders are not supported yet. A single point light source has location C0.xyz, and color C1.xyz. This definition is sufficient for testing surface shaders.

Interface to other types of shaders is not defined, as they are not implemented.

Complete ABI specification must wait until light and imager shaders are implemented, with support for multiple light sources, and applications that generate scene data are developed.

7.3 ABI and performance

Some aspects of ABI can significantly affect performance. One of them is responsibility for saving registers across function calls. Registers can be:

- "callee-saved" – called function is not allowed to change them unless it restores the original values before return
- "caller-saver" – called function can overwrite them at will, if caller wants their values to be preserved, it must save them and restore after function call
- Registers used for passing arguments or returning values are implicitly caller-saved.

Saving is necessary if and only if both caller and callee use the same register. As compiler does not know it compiling either caller or callee, unnecessary saving can be introduced in both cases. The issue is most pronounced in architectures with many registers (like RPU), as ABI can force saving even when there are enough registers for both functions.

Most architectures use both caller-saved and callee-saved general registers. For example in DEC Alpha [ALPHA] there are 7 callee-saved, 12 caller-saved, 6 argument passing and 1 return value registers.

If ABI contains both kinds of registers, compiler can reduce number of register saving by using registers from either set. If function never calls other functions, it can freely overwrite caller-saved registers. On the other hand if function calls many other functions, it can place long-lived variables in callee-saved registers, saving only once instead of once for function call.

Such mixed ABI are typically better than purely caller-saved ABI, but make register allocation more complex. Some aggressively optimizing compilers like Intel C++ Compiler [INTEL] use interprocedural analysis to make decisions on saving registers instead of relying on fixed ABI, what improves efficiency at cost of significantly higher compiler complexity.

In case of RPU it might also be possible to have different ABI for different kinds of shaders – most calls are between subroutines of different kind (main shader calls surface shaders, surface shaders call mostly light shaders and normal functions, small functions are inlined so main shader, light shaders and normal functions would rarely call further functions), and it should be possible to make surface shaders use one kind of registers, and other shaders and normal functions use a different kind, with saving necessary only for calls to functions of the same kind, and when number of registers needed is very high.

Currently the compiler uses the simplest possible ABI, with only caller-saved registers. Aggressive inlining reduces effects of function call inefficiency.

7.4 Assembly-level optimizations

The following optimizations are performed at assembly level:

- Opcode simplification and constant folding – corresponding to the same operation performed at SSA level. At assembly level it is possible to take advantage of features like source modifiers, that were not represented at SSA level.
- Forwarding – a limited version of copy propagation. Most copy operations are already removed at SSA level, however new copy operations are introduced during conversion to assembly or by opcode simplification. One important case is forwarding of S subregisters.
- Liveness analysis and dead computation elimination.
- Register allocation tries to allocate registers in a way that results in better code, so it can be considered an optimization.
- Useless copy elimination (copying from register to the same register) and removing nop opcodes.
- Instruction scheduling.

7.5 Opcode simplification and constant folding

Opcode simplification and constant folding is very similar to such optimization on intermediate representation level. Operations are replaced by simpler equivalent operations, constants are folded, conditional branches are simplified etc.

Two most important groups of operations that are simplified at this step are optimizations of functions that were inlined late (what can trigger further optimizations), and simplification by introduction of source modifiers.

For example the following computation in intermediate representation:

```
mul X, A, 2
mul Y, B, -4
sub Z, X, Y
```

cannot be optimized on SSA level. It is converted to assembly-level code:

```
mul X, A, 2
mul Y, B, -4
add Z, X, -Y
```

which can be simplified to:

```
mov X, 2*A
mov Y, -4*B
add Z, X, -Y
```

and then by forwarding converted to:

```
mov X, 2*A
mov Y, -4*B
add Z, 2*A, 4*B
```

What (assuming X and Y are not used anywhere else) gets optimized by dead code elimination to:

```
add Z, 2*A, 4*B
```

7.6 Forwarding

Forwarding is a limited version of copy propagation. Instead of global “find a copy, replace everywhere”, a more complicated and less powerful algorithm is used, which however does not depend on the code being in SSA form.

As more powerful copy propagation was performed at SSA level, forwarding is used mostly to optimize code generated by late function inlining, in particular to forward subregisters of S, and to assemble `mad` and `dp3_rsq` instructions from simpler instructions. Assembling of `mad` and `dp3_rsq` can be considered a separate optimization, but it is run together with normal forwarding.

The algorithm iterates two steps:

- Propagate assertions about values of variables at edges in the program
- For each node, use assertions for that node to simplify its opcode if possible. If any opcode was simplified, start another iteration.

Three kinds of assertion are currently used. The most important is “A is B” assertion, which means “A equals B at this point and it uses of A should be replaced by uses of B if possible”. Other assertions are “A is B multiplied by C” and “A is dot product of B and C”, which are used only to merge `mul` followed by `add` into `mad`, and `dp3` followed by `mov_rsq` into `dp3_rsq`.

If copy operation from B to A is found in the code, “A is B” is added at the outgoing edge of this operation. Such assertions are not generated if B has a register allocated to it, as forwarding such variables can increase their live ranges and easily introduce new conflicts. No such precautions are necessary in case of A, as its live range can only be reduced by replacing its uses with uses of B.

Assertions that are present on each incoming edge of the node and are not broken inside the node (if either A or B is modified, “A is B” assertion must be dropped), are propagated to all outgoing edges of the node.

We can follow the algorithm on the following code fragment that multiplies vector B by length of vector `A`:

```
dp3 C, A, A           ; C = Length of A squared
mov_rsq R15.x, C      ; S.x = 1 / Length of A
mov D, S.x
mov_rcp R15.y, D      ; S.y = Length of A
mov E, S.y
mul F, B, E           ; F = B * Length of A
```

There are two `mov` instructions and one `dp3` that generate assertions:

```
dp3 C, A, A
mov_rsq R15.x, C      ; (C is dot product of A and A)
mov D, S.x            ; (D is S.x)
mov_rcp R15.y, D
mov E, S.y            ; (E is S.y)
mul F, B, E
```

Propagation is very straightforward as there are no branches, and none of A, C, D, E, S.x, S.y is modified on path from their origin to end of the fragment:

```
dp3 C, A, A
mov_rsq R15.x, C      ; (C is dot product of A and A)
mov D, S.x            ; (C is dot product of A and A; D is S.x)
mov_rcp R15.y, D      ; (C is dot product of A and A; D is S.x)
mov E, S.y            ; (C is dot product of A and A; D is S.x; E is S.y)
mul F, B, E           ; (C is dot product of A and A; D is S.x; E is S.y)
```

Having finished assertion propagation, we can now use the assertions for code simplification:

```
dp3 C, A, A
dp3_rsq R15.x, A, A ; dp3 merged with mov
mov D, S.x
mov_rcp R15.y, S.x    ; D replaced by S.x
mov E, S.y
mul F, B, S.y         ; E replaced by S.y
```

This optimization by itself does not simplify the code, but it allows dead code elimination remove computation of `C` and assignments to `D` and `E`.

```
dp3_rsq R15.x, A, A
mov_rcp R15.y, S.x
mul F, B, S.y
```

Even if these instructions were not eliminated, the code already became faster, as computation of `F` now requires only 3 steps instead of 6, and computation of `C`, `D` and `E` can be done in parallel, without stalling the pipeline.

Assertions also support source modifiers, so “A is -2*B.yzx” is a valid assertion.

There is one simple kind of assertions that could be used but is currectly not – “A is-x/y/z-field-of B”. They would make it possible to simplify code like:

```
  mov B, A.x
  mul D, C, B
```

to:

```
  mul D, C, A.x
```

7.7 Dead code elimination

Dead code elimination removes code which does not contribute to the results. There are two kinds of dead code – unreachable code, which due to graph-based representation of program is automatically removed when it becomes unreachable, and code which only computes “dead” variables and has no other effects, which is removed by this optimization.

Code liveness and variable liveness are computed together. Code which returns values from function or has side effects (like control flow, `call`, and `trace`) is unconditionally live. Code is also live if it modifies a live variable.

If code is dead, set of variables that are live at its entry is equal to set of variables that are live at its exit (it is considered equivalent to `nop`). If code is live, set of variables that are live at its entry is equal to set of variables that are live at its exit minus variables defined by the node, plus variables used by the node.

Computations start with all code and all variables considered dead, and increase sets of live code and variable until reaching fixed point. Code that is marked dead at that point is eliminated. Information on live ranges of variables is saved, as it is also used by register allocation and code output. In current ordering of optimizations it does not need to be recomputed.

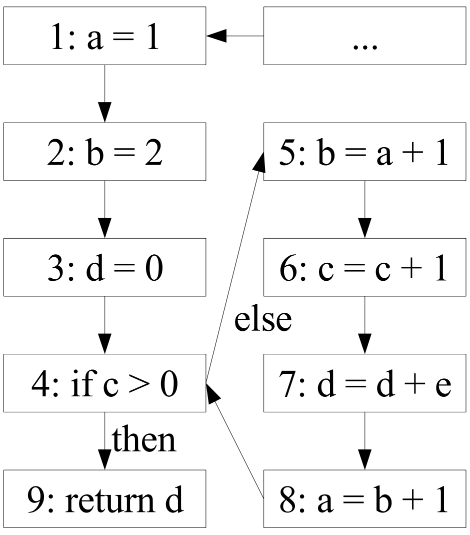


Figure 7.2

An example (Figure 7.2) is fragment of a function, including a simple loop (... denotes rest of the function).

Five variables are used – a, b, c, d, e. At first two nodes are live and two variables are live at their entries. Node 4 affects control flow and uses c, so c must be live at its entry. Node 9 returns d from function, so d must be live at its entry. (Figure 7.3).

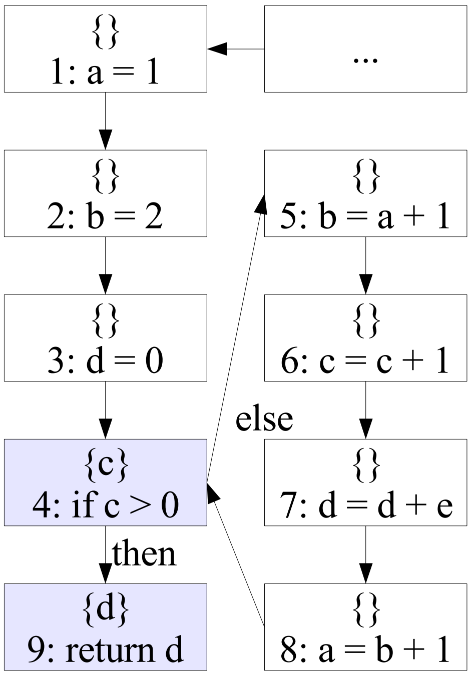


Figure 7.3

Liveness is propagated in direction opposite to control flow. When fixed point is reached (Figure 7.4), nodes 3, 6, 7 are live, while nodes 1, 2, 5, 8 are dead, as are variables a and b.

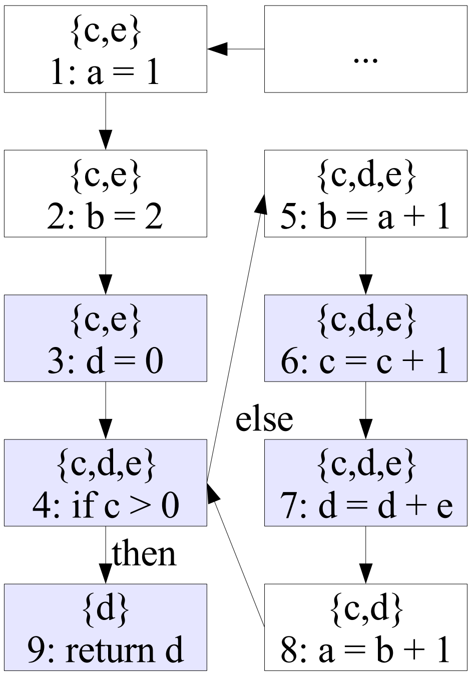


Figure 7.4

Dead instructions 1, 2, 5, 8 are then be dropped (Figure 7.5).

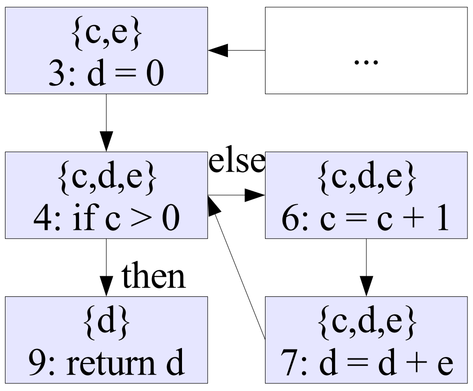


Figure 7.5

8 Register allocation

Two important parts of an optimizing compiler are register allocation and instruction scheduling. It is possible to run either of them first, and a few experimental algorithms even run them together, however the last option leads to significant added complexity and is rarely used.

There is a tradeoff between the orders:

- If register allocation is run first, it can introduce unnecessary instruction dependencies that make instruction scheduling more difficult.
- If instruction scheduling is run first, it is likely to significantly increase register pressure, what can lead to register spills (copying contents of some of the variable into memory instead of registers and reloading it later), or even make the code uncompilable.

As spills are considered much more expensive than pipeline stalls, most compilers run register allocation first. In case of the RSL compiler it is significantly better to run register allocation first, as:

- It is not possible to spill, so increased register pressure can make the code uncompilable.
- Stalls do not actually stop computations, but only force a switch to another thread. Excessive thread switching should be avoided, but cost of suboptimal scheduling is very small compared to most other modern architectures.
- If register pressure (number of variables that are alive at any given time) is low, register allocation will efficiently balance register usage, and introduce few additional instruction dependencies.
- If register pressure is high, register allocation is likely to introduce many additional dependencies. However in such case increasing the register pressure any further is very likely to make the code uncompilable.

Register allocation algorithm is based on graph-coloring [COLOR], which is used by most modern compilers. The problem of allocating a large number of variables to small number of registers can be reformulated as follows:

- Compute information on variables' live ranges (locations between opcodes at which their values must be remembered). If variables are live at the same type they cannot use the same register.
- Compute enemies and friends of each variable.
    - “Enemies” are variables which must not use the same registers (or overlapping registers if architecture supported them). Variables become enemies if they can use registers of the same type and at any point are simultaneously alive.

      Correct allocation must fulfill all “enemy” constraints.
    - “Friend” are variables which should share the same register if possible. In current implementation a friend is a variable which uses registers of the same type, is not an enemy, and there exists a “mov A, B” instruction somewhere in the shader, such that one of the variables in a target, and the other is a source.

      “Friends” constraints do not affect correctness, and only affect performance of generated code.

      As register sets for scalars and vectors are disjoint, scalars can be neither enemies nor friends of vectors.
- Create a graph in which:
    - Every variable gets a node. The variables which are already allocated (like function parameters etc.) are not represented.
    - There is an edge between nodes corresponding to every pair of enemies.
- Color the graph using no more colors than the number of available registers such that:
    - Every variable gets a register of the correct type
    - If there is an edge between two variables, they get different registers

Graph coloring using limited number of colors is NP-Hard [NP], however a very efficient algorithm exists that works correctly in most real world situations.

The algorithm is based on the following observation:

- If a node X has fewer edges than there are available colors, then if we can color the graph at all, then we can do it by coloring the graph without node X, and afterwards selecting a color for X, what is always possible as its neighbors must have left at least one color unused.

The version used in the RSL compiler generates the order of coloring in the following way:

- Find a node with fewest edges (variable with fewest conflicts), remove if from the graph and add to allocation list
    - If there are multiple variables with the same number of conflicts, select one with fewest “friends”

  Selecting nodes with fewest edges first is simply part of the graph coloring algorithm.

  Selecting nodes with fewest friends first (so that nodes with most friends are allocated earlier) is a heuristic that sometimes improves allocation. If variables with multiple friends are allocated first, their friends try to use the same register if possible. On the other hand if variables with few friends were allocated first, they would try to use different registers to avoid register overcrowding, even if they do not conflict. Variables with multiple friends would then usually be unable to share allocation with more than one of their friends.

- If we were able to remove all nodes from the graph, and number of conflicts of each removed variable was always lower than number of the registers of the appropriate type (that is less than 15 conflicts), then allocation is always possible. Otherwise it might be impossible to allocate correctly, but we do not know it at this point.
- Go over the list of variables in reverse order (the last added – that is the most difficult – first), and allocate each variable to a register that is not used by any variable that conflicts with it. If there are multiple such registers use the following heuristics:
    - Use allocation that is shared with most "friend variables".
    - If the first criterion did not select a single register, use one with the fewest variables allocated to it. Balancing register usage makes instruction scheduling easier.

      Each friend with which allocation is shared saves at least one "mov" instruction, while balancing register usage has unknown and usually smaller effect on instruction scheduling. As the first effect is typically much stronger, it is considered first.
    - If there are multiple candidates with the same score, take one with smallest register number. This step makes the allocator deterministic and simplifies compiler debugging.

9 Instruction scheduling

Like register allocation, optimal instruction scheduling is also an NP-Hard problem even under significantly simplified assumptions [SCHED]. The algorithm used is "forward list scheduling" of basic blocks. Most compilers use similar algorithm for scheduling, differing only in the scheduling direction (forward or backward), use of various methods to merge basic block into larger blocks, and different criteria for selecting scheduled instruction from list of candidates.

A "basic block" is a sequence of instruction that are either executed all in specified order, or none of them is executed. This means no instruction except for the first can have predecessors outside the block (or more than one predecessor), and no instruction except for the last can have successors outside the block (or more than one successor).

As the compiler does not use basic blocks, but the program graph nodes are individual instructions, the first stage of scheduling algorithm must be basic block consolidation. Instructions that are not "reschedulable" are forced to have own basic blocks. Non-reschedulable instructions are conditional jump, function call, return, and trace instructions. Links to the first and from the last instruction of each block are saved at this point.

As the next step each basic block that contains more than one instruction is rescheduled independently. After rescheduling, basic blocks are reconnected into a complete program, at which point information on basic blocks is thrown away.

The core of instruction scheduling is scheduling of a single basic block. The algorithm works as follows:

- Directed graph showing dependencies between instructions is built. Each instruction in the graph has a node, and an arrow exists from A to B if B cannot start before A finishes. There are three kinds of dependencies:

- Real dependencies ("Read After Write") – A writes to register and then B reads from that register.
- "Write After Read" dependencies – A reads from a register and then B writes to this register. A and B cannot be reordered or the value read by A would be incorrect.
- "Write After Write" dependencies – A writes to the register and then B writes to the same register A and B cannot be reordered or the register would have different value in the end. Writes to junk register (R15) are not considered, as its final value is irrelevant.
- Every variable without other links gets a link to the "sink node".

These dependencies contain complete information which instructions can and which cannot be reordered.

The second and the third kind of dependencies do not represent real data flow dependencies and could be avoided by different register allocation. As the register allocation tries to balance register use, it can be expected that few such unnecessary conflicts will be introduced unless register pressure is very high.

- Start the scheduling algorithm

The algorithm tries to track which instructions can be scheduled on which processor cycle, starting from the cycle in which the scheduled block was entered, and uses a greedy algorithm for scheduling. Some simplifying assumptions are used. Memory and trace instructions are assumed to have constant latency. It is assumed that when block is entered all prior computations are finished.

A few concepts help understanding the applied heuristics. An "optimal scheduling" is any scheduling of so far unscheduled instructions which takes the fewest cycles in our model of processor. Instructions are "critical" if one of them must be scheduled immediately in any optimal scheduling. Every time a non-critical instruction is selected when critical instruction was available, at least 1 cycle is lost. In current model the loss is always exactly 1 cycle.

It is possible that only non-critical instructions are available, in which case selection of any instruction (and even stalling a cycle) is optimal. It typically happens when all instructions from long computations are waiting for dependencies, and one or more instructions from shorter computations are available.

If some instruction is always issued when available, the "worst scheduling" can be no worse than (number of instructions – 1) longer than an optimal scheduling (-1 because the last instruction issued must be critical). This happens only in an unlikely case when at every step a critical instruction was available but non-critical one was issued. In other words, the worst scheduling is less than 2x longer than an optimal scheduling.

The "worst scheduling" always issues some instructions when available, so it is still significantly better than not scheduling. In non-scheduled code it is very likely that processor must wait for the next instruction when it could execute some other instruction. In the most pathological case almost every instruction must wait number of cycles equal to instruction latency, while a different instruction is always available (then, if instruction latency is 10, the worst case of non-scheduling is close to 10x longer than optimal).

The heuristics try to avoid taking non-critical instructions when critical instructions are available.

- Initialize cycle counter to 0
- As long as there are any instructions that are not scheduled:
    - Check if there is any instruction, all dependencies of which have finished executing.
        - If not, increase cycle counter by 1 and restart. As the dependency graph is acyclic, we can always schedule all instructions if we wait long enough.

          This case corresponds to situation where processor cannot execute any instructions and must switch to a different thread.
    - Some instructions can be scheduled. This case corresponds to situation where some instructions can be executed by processor.
    - If there are multiple possible instruction to schedule in current cycle, compute for each instruction length of the longest latency-weighted path in the dependency graph from the instruction to the sink node ("path"), and keep only these instructions for which path is not shorter than path of some other candidate (it is possible that a blocked instruction has even longer path, but they are not considered).

      Length of a path is number of cycles during which computations of this path are performed.

      Thus difference between optimal scheduling length and path of a node is number of computation cycles during which, under optimal scheduling, computations of the path are not performed.

      If nodes are critical, one of them must start computations immediately. Situations where computations must begin immediately but are later allowed to wait many cycles are unusual.

      By this reasoning, instructions with the longest paths are significantly more likely to be critical than instructions with even slightly shorter paths.

      In practice, always issuing instruction when possible and the longest path heuristics together are responsible for almost all optimization. Further heuristics typically have only a minor effect.
    - If there are still multiple candidates, keep only those with most instructions directly depending on them.

      Instruction on which multiple computations depend is more likely to be critical than instruction on which fewer computations depend, when their longest paths are the same.
    - If there are still multiple candidates, keep only those with biggest latency.

      Often computation is tree-like, and there are more instructions available earlier in the block than later. This is a problem, as at first there are available instructions but no free slots, while later there are free slots but no available instructions. Scheduling fewer longer instructions early, and more shorter instructions late sometimes improves performance.
    - Select first of the candidates (to make the algorithm deterministic), remove it from the list of instructions to be scheduled and record the cycle when it finishes execution.

- Increase cycle counter by 1.

Scheduled instructions are connected linearly inside each basic block.

The basic blocks remember successors of the original last instruction. They are converted to links to the first instruction of each successor's basic block, and attached to the last instruction. Reverse links are recreated in analogous way.

9.1 Instruction scheduling example

Figure 9.1 shows an unscheduled block of 6 instructions and dependency graph for it. The example assumes that `add` takes 3 cycles, while `mul` and `dp3` take 5 cycles.

Without scheduling, the block takes 16 cycles to complete.

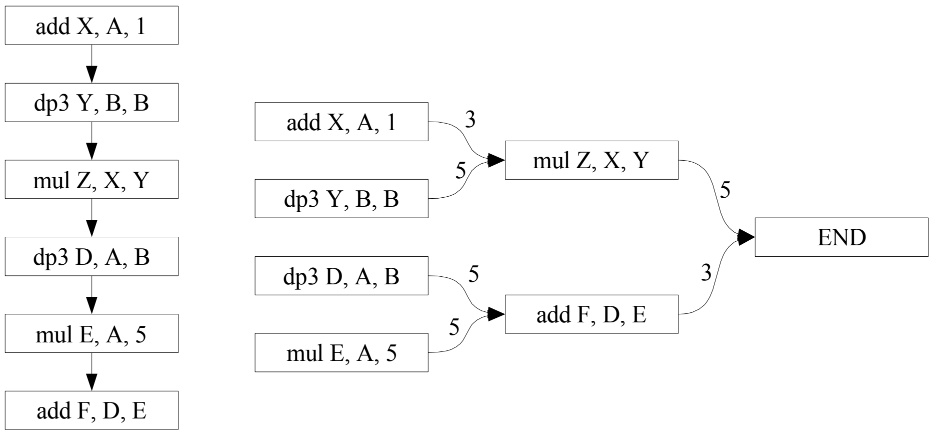


Figure 9.1

Instructions have following weights:

Instruction	Latency of longest path	Direct successors	Latency
1: `add X, A, 1`	8	1	3
2: `dp3 Y, B, B`	10	1	5
3: `mul Z, X, Y`	5	0	5
4: `dp3 D, A, B`	8	1	5
5: `mul E, A, 5`	8	1	5
6: `add F, D, E`	3	0	3

Allocation proceeds as follows:

Cycle	Newly available	Available variables	Candidates	Selected instruction
1		A, B	1, 2, 4, 5	2: `dp3 Y, B, B`
2		A, B	1, 4, 5	4: `dp3 D, A, B`
3		A, B	1, 5	5: `mul E, A, 5`

4		A, B	1	1: `add X, A, 1`
5		A, B	None	Stall
6	Y	A, B, Y	None	Stall
7	D, X	A, B, D, X, Y	3	3: `mul Z, X, Y`
8	E	A, B, D, E, X, Y	6	6: `add F, D, E`
9		A, B, D, E, X, Y	None	Stall
10		A, B, D, E, X, Y	None	Stall
11	F	A, B, D, E, F, X, Y	None	Stall
12	Z	A, B, D, E, F, X, Y, Z	None	End

Execution time was reduced from 16 to 11 cycles.

9.2 Special operations in scheduling

There is also limited support for rescheduling instructions that modify S registers. During conversion from intermediate representation to assembly, generated `mov_rcp/mov_rsq` instructions are assigned to four different S subregisters in a round-robin fashion. This means that a result of one special operation can be forwarded across another special operation as long as it uses the a different S register, and it also makes it possible to reschedule `mov_rcp/mov_rsq` operations.

Naive round-robin allocation during code generation can lead to less than optimal code if there are many rsq/rcp operations in the shader, and possibly generating virtual S registers and allocating them by the graph-coloring based register allocator would lead to better results. As there are only 4 available registers, spilling to standard scalar registers should also be implemented. If such register allocation was implemented, it should probably be run before the main register allocation phase, as it can increase register pressure of standard scalar registers.

This is only a problem in code that contains more than four rcp/rsq operations close to each other. Common subexpression elimination can often eliminate some of them – for example if `length` and `normalize` are called on the same vector, rsq will only be called once.

9.3 Instruction pairing

Due to lack of virtual machine support for instruction pairing, the scheduler issues only one instruction per cycle. The algorithm can be straightforwardly modified to support instruction pairs. The necessary modifications are:

- When finding candidates for scheduling verify that they can be scheduled together with the instructions have already been scheduled in current cycle.
- Mark the instructions scheduled in a single cycle as pairs.
- Do not automatically increase cycle number after successfully scheduling an instruction.

9.4 Other possible improvements of the scheduling algorithm

A simple modification of the scheduling algorithm would be including conditional jump instruction directly after the next block in the dependency graph. After conditional jump the thread switch is unavoidable, so our priority should be making all registers used in the condition check available when cjmp is executed to avoid second thread switch just before the condition check. Simply marking cjmp reschedulable, but not scheduling it until everything else was scheduled would be most likely sufficient.

Another simple improvement would be passing information between blocks. After a block is scheduled, information is available on how many cycles it takes before each register can be used. This information could be used to schedule successor blocks better. If the code does not contain loops, it is easy to select order of block scheduling in which such information is always available. As shaders typically contain no or very few loops, scheduling quality could be greatly improved.

10 Code output

After the code has been generated, registers allocated, and the instructions scheduled, the only step left is printing the final assembly code. The printing is not completely trivial, as the program representation in memory is different from the final representation. Two main differences are:

- The program is represented as a graph, while the generated assembly code must be linear.
- `call` opcode at this level handles variable saving/restoring automatically.

Expanding calls uses information on live variables before and after the call. If a variable is alive before and after the call (that is – it should be preserved), then it is copied to the stack before the call, and restored after the call. The push size is the smallest size required to push all saved variables out of the called function's visible stack space.

Saving and restoring is done so late because various optimizations can introduce and remove variables, and change variables' live ranges, and it would be difficult to modify the saving/restoring code accordingly to keep it correct and efficient. On the other hand performing saving and restoring so late makes some optimizations impossible. One such optimization would be forwarding of variables on the stack, so copies from the stack into registers can be avoided. Another optimization would be computing directly onto the stack, instead of computing into registers and then copying them onto the stack. Third optimizations missed is avoiding unnecessary restores followed by saves if multiple function calls are performed in sequence without branches between them.

The algorithm used to convert program graph into linear assembly performs a depth first search, by printing opcode label followed by opcode and jump to the next label. If jump to a label is followed by the label, it is removed from the generated text. Also, if a label is never referred, it is removed from the text before printing. This does not change the semantics however it can make assembly more readable.

11 Implementation issues

11.1 Testing

Optimizing compilers are very complex programs, and subtle interactions between optimizations can easily lead to bugs that occur under rare conditions, and can easily stay undetected for a long time. As the RSL compiler for RPU does not have a benefit of large user base of testers, the only way of assuring low bug rate is extensive automatic testing.

Majority of test are verifying the whole setup, starting from RenderMan Shading Language files as input, compiling them, running on a virtual machine, and finally verifying the output. Bugs at any phase – parsing, code generation, optimizations, or virtual machine execution – can make the whole process fail. Bugs in the tests are also possible.

Tests (example in Figure 11.1) are implemented in Ruby using unit-testing framework Test::Unit, which is very similar to the well-known JUnit testing framework for Java. The typical testing scenario is:

Test example – test case that verifies `normalize` function

```
def test_normalize
  f = "vector f(vector x) { return normalize(x); }"
  res = Driver.compile_and_run(__LINE__, f,
                               [3.0, 4.0, 0.0])
  assert_equal([0.6, 0.8, 0.0],
               res[:vector],
               "normalize(x) should work")
end
```

Figure 11.1

- A single small snippet of RenderMan shading language code containing 1 or 2 functions is saved.
- The code is preprocessed (optionally) and compiled to assembly.
- The code is assembled.
- The virtual machine runs the code, using arguments specified in the test case.
- The output (either state of the registers at exit or generated file) is verified.

All intermediate files are saved to make it easier to manually locate source of the problem in case the test fails.

About 300 tests are performed, testing:

- All primitive operations.
- All standard library functions. If the function's behavior depends on the argument value, multiple tests are performed to test all possibilities. For example `abs` function has two tests – one for positive and one for negative values.
- Optimizations and instruction scheduling. In these tests in addition to verifying the output, the generated assembly code is checked against a regular expression that represents well-optimized code. The tested code fragments are small enough for this approach to be viable. Verifying more complex functions would probably require modifying the virtual machine to provide information on number of operations and clock cycles used.
- Generation of actual pictures. The pictures are verified against known-good pictures using MD5. These tests can check that changes in the compiler did to introduce bugs

in the rendering. However floating point inaccuracies can make the images be slightly different, so in case of a MD5 mismatch a human must verify whether there it is an actual bug or just a minor mismatch.

11.2 Automatic code coverage checking

Test coverage is automatically checked using rcov program. Data from all tests is aggregated and statistics are generated in form of HTML files. Currently 91% lines of the Ruby part of the compiler is executed in at least one test. Most of the code that is not executed consists of:

- Various sanity checks, like code that raises an exception when a function was called with wrong type of arguments.
- Code related to debugging, that is inactive during normal compiler execution.
- Code that is impossible to trigger due to interactions between optimizations, like code that prints out nop instructions, which is never run as all nops are eliminated by optimizations. It would be possible to test such code by making it possible to selectively turn off individual optimizations.
- Code that is inactive due to having incomplete support further in the pipeline, like texture-related code, when textures are not supported by testing driver of the virtual machine yet.
- Very small amount of code that can be realistically executed during compiler execution, and which should be covered by tests.

Objective Caml part is not currently tested due to lack of adequate testing tools for Objective Caml. This makes it more likely that some bugs have been left in the code. The simplest solution would be recoding the rest of the compiler to Ruby.

Very high coverage makes it less likely that bugs were missed by testing, but of course it does not provide absolute guarantees (it is commonly stated that good coverage level is 80-90%). The measured coverage has line resolution, so it is possible that some code is incorrectly reported as executed.

11.3 Build system

As the system consists of multiple executables written in different languages, the build system can be quite complex. Compilation of code written in Objective Caml is controlled by GNU make files using OCamlMakefile library to control dependencies between various Objective Caml files automatically. Without OCamlMakefile controlling build process would be more difficult.

The test framework uses Ruby-native rake as its build system, however the only Rakefile contains a few simple testing and cleaning related actions, so it might be better to port it to GNU make for consistency. Alternatively OCamlMakefile could be ported to rake, what would enhance portability and reliability of the build system, however it would require significantly more work.

11.4 Strings support

Full RenderMan-compatible string support was considered infeasible as features like string-generation at runtime require memory management and significant architectural changes would be necessary to integrate it. On the other hand full strings support is needed by few shaders, as the strings are most commonly used as identifiers, and the

the only operation needed for that is equality testing. There are multiple possible ways to implement such limited strings. Some of them are:

- **Representing strings as memory pointers**, and performing equality testing by iteratively comparing the referenced memory locations. The memory representation could be 0-terminated or specify explicit length. This implementation would be extremely slow as a loop and multiple memory loads would be required. The main benefit of this implementation is its easy extension to more advanced string support in the future, unconditional correctness, and relative architectural simplicity.
- **Mapping strings to unique numbers using a string registry**, and performing equality testing by comparing the numbers. Such implementation would have very good performance and would be unconditionally correct. Inside a single compilation unit it would be a preferred choice. The major problem of this implementation is maintaining the registry across different compiled units. Separately compiled shaders need to use the same identifiers for the strings. Also if string identifiers are used in the scene data, they need to be kept synchronized with registries used by the shaders. This greatly complicates the architecture.
- **Mapping strings to unique numbers using a hashing function**, and performing equality testing by comparing the numbers. This implementation has great performance characteristics and does not require any architectural support. Its major downside is small possibility of a random collision. According to the birthday paradox, the chance of collision is roughly proportional to square of number of strings. For 24-bit hashes, 600 strings have approximately 1% chance of collision, an 4800 strings have approximately 50% chance. It is considered unlikely for shaders to use that many strings, and even in case where a total number of strings in all shaders is high, they are most likely to be compared only with a few other strings, thus most collisions will not lead to any errors.

The compiler uses hashing solution. As an additional protection, a warning is issued by the compiler if multiple strings in a single compilation hash to the same value. The compiler uses 24 first bits of MD5 hash, converted to a floating point number. Number of bits may need to be adapted to precision of the hardware floating point format.

Other solutions are also possible, like mapping at run time by a special linker, or using a different methods inside a compilation unit and across compilation units. Such methods invariably lead to greater architectural complexity, and have not been further investigated.

11.5 Version control

All source files of the project are managed using Subversion revision control system. Subversion has many advantages over the older CVS system, including fully atomic commits and ability to copy, move and delete files inside the repository without losing revision history. It is used by many large project including Apache, KDE, GCC, and Python.

Automatic revision control with very frequent commits enabled development without having to worry about accidentally breaking the code. A few times it also made it significantly easier to trace back the change that was responsible for a newly found bug.

Commits had been made after any complete change, sometimes as often as every 5 minutes. After the test suite had been developed, it had been run before any commit to make sure that the changes did not accidentally break unrelated code. As a general rule a version that passed fewer tests than the last one was not committed.

12 Results

Two complete examples from the test set are presented. They both apply different shaders to the Stanford Bunny model [BUNNY].

12.1 Depth shaded bunny

The first example is shader which smoothly changes color from green to magenta depending on distance from the camera.

Code consists of two parts – a "main" shader which shots primary rays, in form of a RenderMan function which takes point argument (pixel position) and returns color, and a RenderMan surface shader.

```
/* Default surface shader */
surface s() {
    float dist = distance(P, E);
    float g = 1.8 - dist;
    float rb = 1.0 - g;
    Ci = (rb, g, rb);
}

/* "Main" shader */
color m(point P) {
    point  orig   = (0.0, 0.0, 0.0);
    vector dir0   = (0.0, 0.0, 1.0);
    vector diri   = (2.0,-2.0, 0.0);
    vector s      = (0.5, 0.5, 0.0);
    vector dir    = normalize(dir0 + (P - s) * diri);

    return trace(orig, dir);
}
```

The surface shader is compiled to the following assembly code:

```
SUBROUTINE_ENTRY_s:
   ; R3.xyz = ray direction * distance traveled by ray
   mul R3.xyz, R1.xyz, HIT.z
   ; R2.xyz (P) = ray origin + (ray direction * distance traveled by ray)
   ; E is currently always <0, 0, 0>,
   ; so compiler knows R2 is also (P – E)
   add R2.xyz, R0.xyz, R3.xyz
   ; R15.y = 1/sqrt((P – E) . (P – E))  = 1 / distance(P, E)
   ; Uses merged instruction dp3_rsq
   dp3_rsq R15.y, R2.xyz, R2.xyz
   ; S.z = distance(P, E)
   ; S.y was forwarded instead of using a register
```

```
    mov_rcp R15.z, S.y
    ; R1.w (g) = 1.8 – dist
    ; S.z was forwarded like S.y
    add R1.w, 1.8, -S.z
    ; R0.w (rb) = 1.0 – g
    add R0.w, 1.0, -R1.w
    ; R0 (Ci) = (rb, g, rb)
    ; Compiler sets two components with single mov
    ; It is still unable to compute directly into
    ; component or components of a vector
    mov R0.xz, R0.w
    mov R0.y, R1.w
    mov R0, R0 + return or xyzw (>=0 or <1)
```

The main shader:

```
SUBROUTINE_ENTRY_m:
    ; Initialization of vectors and computation of ray parameters
    mov R2.xy, 0.5
    mov R3.x, 2.0
    mov R2.z, 0.0
    mov R4.xy, 0.0
    mov R3.y, -2.0
    add R5.xyz, R0.xyz, -R2.xyz
    mov R4.z, 1.0
    mov R3.z, 0.0
    mad R6.xyz, R5.xyz, R3.xyz, R4.xyz
    dp3_rsq R15.w, R6.xyz, R6.xyz
    ; Ray direction is computed directly into R1.xyz (see more explanations below)
    mul R1.xyz, R6.xyz, S.w
    ; The ray is shot
    ; 0.0 is point of origin. Constant 0.0 means all components are 0.0
    ; R1.xyz is computed direction
    ; C31 contains default parameters for trace (culling depth etc.)
    trace 0.0, R1.xyz, C31
    ; R0.xyz contains barycentric coordinates and distance traveled by ray
    ; This mov is not eliminated as in current representation trace+mov
    ; is a single instruction, and only get separated on code output.
    mov R0.xyz, HIT.xyz
    ; If hit, go to L23
    mov R15.w, HIT.z + jmp L23, w (>=0)
    ; If miss, set color to <0.0, 0.0, 0.0> and return
    mov R0.xyz, 0.0
L26:
    mov R0, R0 + return or xyzw (>=0 or <1)
L23:
    ; R0.xyz (ray origin) = <0.0, 0.0, 0.0>
    ; R1.xyz should be ray direction. Instead of setting it here,
    ; the compiler computes it directly to R1.xyz earlier.
    ; This was achieved by "friend" allocation. As ray direction was moved
    ; to R1.xyz, register R1.xyz was the most preferred register for it.
    mov R0.xyz, 0.0
    ; Call the surface shader (address in HIT.w) and return.
    ; Both the surface shader and this function return color in R0.xyz,
    ; so nothing needs to be copied.
    call HIT.w push 0
    jmp L26
```

The rendered result can be seen on Image 12.1.

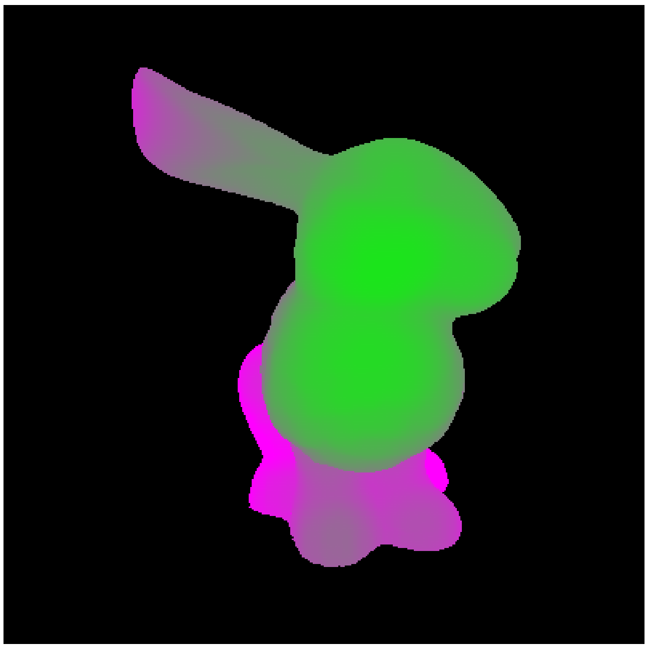


Image 12.1

12.2 More realistic bunny

A second example is the same bunny in very simple but more realistic shading model. The compiler does not yet perform normal interpolation, so all polygons are single-colored, and the result does not look very well. The compiler can easily be modified to support normal interpolation by changing surface shader initialization assembly code, however this would also require time-consuming changes in programs used to import scenes.

The code consists of two parts, "main" shader identical to one in the previous example, and the following surface shader:

```
surface s() {
    vector norm = N;
    if (norm . I > 0)
        norm = -norm;

    vector light_source = vector(-0.2, 0.5, 0.0);
    vector hit_to_light = normalize(light_source - P);
    float uf = clamp(norm . hit_to_light, 0, 1);
    float i  = clamp(0.6 * uf + 0.4, 0, 1);
    Ci = Cs * i;
}
```

The shader calculates amount of light coming from a single point light source by computing clamped dot product of surface normal and normalized vector pointing from the current surface point (P) towards the light source. As all polygons have two sides, normal (N) can flipped depending on direction of incoming camera ray (I). Light intensity is sum of ambient light (0.4) and light coming from the light source (0.0 to 0.6). Light intensity is multiplied by surface color (Cs) and returned (Ci).

Generated code is:

```
SUBROUTINE_ENTRY_s:
   ; HIT_TRI contains address of triangle data
   ; Load the triangle data
   load I0, HIT_TRI, 0
   ; Data consists of 4 pointers:
   ; I0 – [Data for vertex A]
   ; I1 – [Data for vertex B]
   ; I2 – [Data for vertex C]
   ; I3 – [Data for the triangle itself]
   ; Load first 4 words of each of the four
   mov A, I0
   load I0, A0, 0
   load I1, A1, 0
   load I2, A2, 0
   load I3, A3, 0
   ; Code may seem chaotic, as scheduling separates related instruction.
   ; R9.xyz = Vector from B to A
   add R9.xyz, I1.xyz, -I0.xyz
   ; R8.xyz = Vector from C to A
   add R8.xyz, I2.xyz, -I0.xyz
   ; R11.xyz = Vector traveled by ray
   mul R11.xyz, R1.xyz, HIT.z
   ; R5.xyz (Cs) = triangle color
   mov R5.xyz, I3.xyz
   ; Normal computations, part 1
   mul R10.xyz, R9.zxy, R8.yzx
   ; R4.xyz (P) = hit position
   add R4.xyz, R0.xyz, R11.xyz
   ; Normal computations, part 2
   ; By using mad instruction, cross product can be performed in just 2 operations
   ; R12.xyz = geometric normal before normalization
   mad R12.xyz, R9.yzx, R8.zxy, -R10.xyz
   ; Normalize  R12.xyz
   ; R2.xyz is Ng (geometric normal)
   ; By default N (normal) equals Ng (geometric normal)
   ; As the compiler does not perform normal interpolation, R2.xyz is also N.
   dp3_rsq R15.x, R12.xyz, R12.xyz
   mul R2.xyz, R12.xyz, S.x
   ; Check whether normal needs flipping by computing dot product of
   ; R2.xyz (N) and R1.xyz (I – incoming ray direction)
   dp3 R2.w, R2.xyz, R1.xyz
   ; If not, skip the next instruction
   add R15.w, 0.0, -R2.w + jmp L17, w (>=0)
   ; If yes, flip the normal, which is still in R2.xyz.
   ; In SSA form, "normal before flipping" and "normal after possible flipping"
   ; are unrelated variables.
   ; Because not-flipping branch had a copy from the former to the latter,
   ; register allocation allocated both to the same register R2.xyz
   ; and the original variable was reassembled.
   mov R2.xyz, -R2.xyz
L17:
   ; R3.xyz is location of the light source
   mov R3.x, -0.2
   mov R3.y, 0.5
   mov R3.z, 0.0
   ; Compute and normalize hit_to_light vector
   add R7.xyz, R3.xyz, -R4.xyz
   dp3_rsq R15.y, R7.xyz, R7.xyz
   mul R6.xyz, R7.xyz, S.y
   ; Compute light coming from the light source
   ; Unfortunately compiler does not know how to use _sat modifier, and instead
clamps by two branches.
   dp3 R1.w, R2.xyz, R6.xyz
   add R15.w, 0.0, -R1.w + jmp L26, w (>=0)
   add R15.w, 1.0, -R1.w + jmp L28, w (>=0)
   mov R1.w, 1.0
L28:
   ; Compute total light intensity.. Again, _sat should be used.
   ; This clamp is not strictly necessary, as with these constants
   ; insensity can never go out of range.
   ; This mad instruction contains two constants and most likely will
   ; fail to compile on the real hardware.
   ; The compiler does not currently know about such limits,
   ; as the virtual machine does not have them.
   mad R0.w, 0.6, R1.w, 0.4
   add R15.w, 0.0, -R0.w + jmp L31, w (>=0)
   add R15.w, 1.0, -R0.w + jmp L33, w (>=0)
   mov R0.w, 1.0
L33:
   ; Color (Ci – R0.xyz) = surface color (Cs – R5.xyz) * light intensity (R0.w)
   mul R0.xyz, R5.xyz, R0.w
   mov R0, R0 + return or xyzw (>=0 or <1)
```

```
L31:
   mov R0.w, 0.0
   jmp L33
L26:
   mov R1.w, 0.0
   jmp L28
```

The rendered result can be seen on Image 12.2.

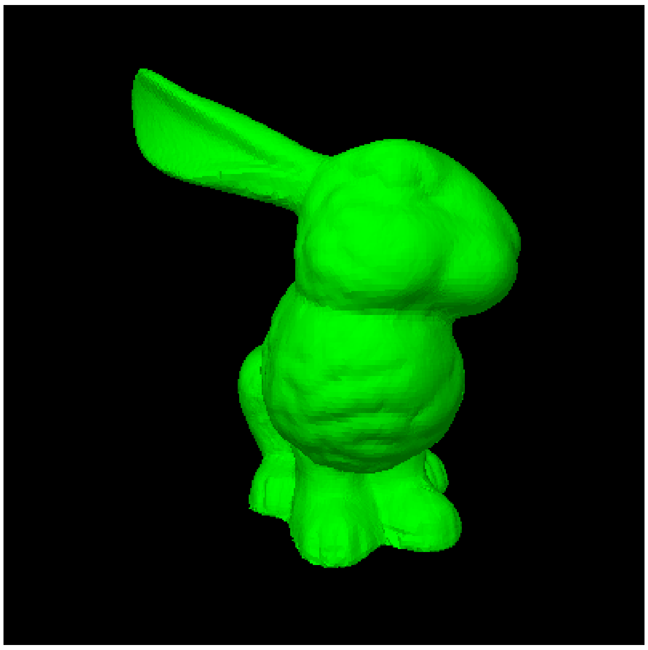


Image 12.2

13 Limitations and future work

The project reached its goals of enabling shader programming for RPU in a high level language. However there is a big scope for possible future enhancements. Multiple limitations and potential enhancements have been mentioned in sections about individual components.

13.1 Hardware compatibility

The compiler is targeting the virtual machine, not the actual hardware. So it is likely that some modifications are necessary to make the generated code work on the RPU hardware.

One identified area where virtual machine and hardware mismatch are equality tests. The compiler assumes that equality tests and comparisons are always correct, and that subtracting a number from itself always yields 0.

The only tests generated by the compiler are (>=0) and (=0). Other tests provided by the instruction set, like (>=1) and various combinations of tests are not used. All inequality operators in RenderMan source (>=, >, <=, <) are converted to a subtraction and a (>=0) test. Equality test operation (==, !=) is converted to a subtraction and a (=0) test.

Because of hardware's inconsistent behavior, it was decided that `sign` function returns 1 for non-negative numbers and -1 for negative numbers. Returning 0 for 0 is not implemented, as it would be impossible to do it consistently on the current hardware. This incompatibility should be corrected when the hardware support for 0 is considered adequate.

Another area where compatibility needs some work is memory interface. As memory and texture format of the hardware are not documented, it is very likely that compiler modifications will be necessary to support memory access and texture access correctly.

13.2 RenderMan compatibility

The implemented subset of RenderMan Shading Language is sufficient for coding simple shaders, however it would be beneficial to support larger subset of RSL and reduce incompatibilities.

In particular the following parts of RSL are not supported:

- Matrices and multiple coordinate systems.
- Many standard library functions like noise functions. Most of them can be implemented as user functions without changes in the compiler.
- Functions with complex interfaces, like those that use extern variables, modify their arguments, nested functions etc.
- Volume shaders, imager shaders, better support for complex light shaders, displacement shaders (limited to modifying normals).
- There is no support for polymorphic functions except for a few functions in the standard library like mix and clamp. The precise algorithm is not described in the RenderMan specification, so some reverse engineering would be necessary to support RenderMan-compatible polymorphism.

13.3 Streamlining

The current design fulfills its role well, however there are a few areas where significant simplification could be performed without loss of functionality. The Objective Caml part of the compiler could be merged with the Ruby part. It would simplify the design, and make automatic test coverage measurements possible.

Using two separate build systems can be problematic, and it would be preferable to merge them. The build system is not able to build standard installable packages (like RPM or DEB) and the only way to use the compiler is to install it from source.

13.4 Testability

Compiler design already has high consideration for testability. Many great improvements can be still possible. Special virtual machine driver could be built that would report execution statistics like number of operations executed, code cycles, and stalls, for easier testing of optimizations. It should even be possible to extract information like coherence between neighboring threads.

A benchmark suite using such driver could be prepared for testing effects of optimizations, and identifying their weak and strong points. Together with an option of selective disabling of individual optimizations it could make it possible to identify redundant optimizations to improve code quality.

It should also be possible to compare per-pixel rendering of scenes (within tolerance range due to imprecision) with a software implementation of RenderMan instead of relying on human inspection and hashing against well-know rendering.

13.5 Optimizations

The generated code is often of good quality, however there is always space for improvement. Particular areas where improvement is possible include:

- User functions should be inlinable, and inlining should often be performed earlier.
- Four S subregisters should be used more efficiently. The current scheme is inelegant and can lead to somewhat suboptimal code.
- More global optimizations. Among possible optimizations are: loop unrolling, code motion, and automatic vectorization. Many advanced SSA-based algorithms can easily be fitted into the current framework.
- Optimizations of single-component access are particularly lacking – for example the compiler does not know that setting x, y and z components of a vector can be done in parallel. Scalar variables cannot be allocated to components of vector variables. A fully generic scheme where vectors are split into 3 independent scalar variables and are only reassembled after the optimization
- Instruction scheduling that uses information outside of a single basic block. There are multiple algorithms to do so.

13.6 Taking advantage of full power of the hardware

Power of the hardware is far from being fully utilized by the compiler. Some of the suboptimalities are:

- Vector and scalar instructions are never paired with each other.
- Pipeline model is very simplistic:
    - Instruction latencies are completely arbitrary, what can result in suboptimal instruction scheduling.
    - Compiler does not generate flags that control instruction dependencies, leaving it to assembler. The compiler knows that two writes to junk register R15 do not block each other, on the other hand if the dependencies are handled inside assembler, this information is not available any more, and less efficient code is going to be generated.
    - "Read after Write" (actual data flow) dependencies require the second instruction to wait for the first to finish. With other dependencies it might be enough not to reorder instructions. For example if we have instructions:

      ```
      add R0, R1, R2
      mul R2, R3, R4
      ```

      Then they cannot be reordered or value of R2 would be wrong. On the other hand the second instruction has no reason to wait for the first, as the write to R2 will not be performed before read of R2 in any realistic scenario. Misestimating latency in such cases can lead to selection of suboptimal scheduling.
- Only two pairs of instructions are merged – `dp3` and `mov_rsq` into `dp3_rsq`, and `mul` and `add` into `mad`. Other pairs could be merged, for example `mov_rsq` and `mov_rcp` can be merged with most other instructions.

13.7 Portability

The project uses mostly standard tools available on most distributions of Linux and other Unix-like systems. It has only been tested on Ubuntu Linux 6.06, however it is likely that ports to other Unix-like systems like BSD, Solaris or Mac OS X will be very simple.

The only project's dependency that is not prepackaged for Ubuntu 6.06 is rcov 0.7 for automatic test coverage reports. The available version 0.6 lacks essential feature of aggregating coverage data over multiple runs. It can be installed as a Ruby gem. rcov is only used for generating coverage reports and the project is fully functional without it.

Porting to non-Unix systems like Microsoft Windows can be more difficult, especially considering the fragile build system. However nothing in the project is fundamentally non-portable.

The project uses two pieces of foreign code – unmodified OCamlMakefile is included as it is required to build the Objective Caml part of the compiler, and DynArray extracted from OCaml ExtLib to avoid unnecessary compile-time dependencies. They are included in the repository for convenience and system versions can be used instead if available.

13.8 Other improvements

One possible improvement would be replacing the current parser by one that has better error reporting. It can be done by modifying current parser, using alternative parser generator (like ANTLR), or coding a recursive-descent parser by hand. In my opinion the preferred solution would be using ANTLR, as it gives acceptable error reporting out of the box and is easier to write and debug than hand-written parsers.

14 Conclusions

The project was able to reach its goals. The supported subset of RenderMan Shading Language is sufficient for writing realistic shaders, and generated code is reasonably efficient. In addition to the main compiler, two other important elements developed for the project were a virtual machine and a comprehensive test suite of about 300 tests that cover most of the code.

While the project can be considered successful, and is definitely useful, it needs a lot of further work to reach production quality.

Only an actual field test can tell for sure how "future-proof" is the compiler, however I was able to do extensive modifications (like addition of instruction scheduling, early inlining, and common subexpression elimination) in matter of hours, so in my expectations it should be able to accommodate significant modifications without any problems. Particularly valuable is the test suite that assures that modifications did not break unrelated code – an event very common due to unpredicted interactions between different optimizations.

One aspect that was not realized was testing on testing with the real hardware. This final phase of compiler development is still to be done.

Bibliography

- [WIKI] – "Graphics processing unit" article on Wikipedia http://en.wikipedia.org/wiki/Graphics_processing_unit

- [RISPEC] – The RenderMan Interface, Version 3.2.1, http://renderman.pixar.com/products/rispec/rispec_pdf/RISpec3_2.pdf
- [RPU05] - Sven Woop, Jörg Schmittler, and Philipp Slusallek, RPU: A Programmable Ray Processing Unit for Realtime Ray Tracing. Proceedings of ACM SIGGRAPH 2005.
- [SSA] - Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451-490, Oct 1991.
- [AIX] – C for AIX User's Guide, http://www.ibm.com/software/awdtools/caix/downloads/caix50.pdf
- [COLOR] - G.J. Chaitin, M.A. Auslander, A.K. Chandra, J. Cocke, M.E. Hopkins, and P.W. Markstein. *Register Allocation via Coloring*. Computer Languages, 6:45--57, January 1981.
- [NP] - Richard M. Karp. "Reducibility Among Combinatorial Problems." In *Complexity of Computer Computations*, Proc. Sympos. IBM Thomas J. Watson Res. Center, Yorktown Heights, N.Y.. New York: Plenum, p.85-103. 1972.
- [SCCP] - Wegman, Mark N. and Zadeck, F. Kenneth. "Constant Propagation with Conditional Branches." *ACM Transactions on Programming Languages and Systems*, 13(2), April 1991, pages 181-210.
- [GVN] - Alpern, Bowen, Wegman, Mark N., and Zadeck, F. Kenneth. "Detecting Equality of Variables in Programs." *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages* (POPL), ACM Press, San Diego, CA, USA, January 1988, pages 1-11.
- [PRESSA] - Kennedy, R., Chan, S., Liu, S.M., Lo, R., Peng, T., and Chow, F. *Partial Redundancy Elimination in SSA Form*. ACM Transactions on Programming Languages Vol. 21, Num. 3, pp. 627-676, 1999.
- [PRE] - Morel, E., and Renvoise, C. *Global Optimization by Suppression of Partial Redundancies*. Communications of the acm, Vol. 22, Num. 2, Feb. 1979.
- [ALPHA] – Tru64 Calling Standard for Alpha Systems, http://h30097.www3.hp.com/docs/base_doc/DOCUMENTATION/V51_HTML/ARH9MBTE/NCH0001X.HTM
- [SCHED] – D. Bernstein, M. Rodeh, and I. Gertner. On the Complexity of Scheduling Problems for Parallel/Pipelined Machines. IEEE Transactions on Computers, 38(9):1308–13, September 1989
- [BUNNY] – The Stanford Bunny home page, http://www.gvu.gatech.edu/people/faculty/greg.turk/bunny/bunny.html