Compiling to a VLIW Fragment Pipeline

William R. Mark and Kekoa Proudfoot

Department of Computer Science*
Stanford University

Abstract

The latest generation of graphics hardware supports fully programmable vertex and pixel/fragment operations, but programming this hardware at a low level is difficult and time consuming. To address this problem, we have developed a complete real-time procedural shading system that compiles a high-level shading language to programmable vertex and fragment hardware, as described in a separate publication. In this paper, we describe in detail the algorithms used by this system to generate and optimize fragment code for NVIDIA's register combiner architecture and show that our compiler generates efficient code. The register combiner architecture has some similarities to VLIW CPU architectures, so we compare our compilation algorithms to those described in the literature for VLIW CPU architectures. We also discuss some of the lessons we learned from building and using this compiler that may be useful to the designers of future programmable graphics hardware.

CR Categories: I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Color, shading, shadowing, and texture; D.3.4 [Programming Languages]: Processors – Compilers and code generation I.3.1 [Computer Graphics]: Hardware Architecture—Graphics processors; I.3.6 [Computer Graphics]: Methodology and Techniques—Languages

1 Introduction

Real-time consumer-level graphics hardware is rapidly becoming application-programmable at both the vertex processing and pixel/fragment processing stages. NVIDIA's GeForce3 chip is the first example of such hardware, but we expect others to appear soon. Programmable graphics hardware enables applications to implement complex shading algorithms while maintaining high performance. These shading algorithms can more realistically model the enormous variety of materials and lighting effects that exist in the real world.

Applications specify vertex and fragment programs using either the DirectX8 API, or vendor-specific OpenGL extensions. In either case, the programs are specified at a level somewhere between assembly-language and microcode. As a result, programs are difficult to write, difficult to understand, and often not portable across graphics chips from different vendors.

One solution to this problem is to write shading programs (*shaders*) in a high-level shading language, and compile these programs to the graphics hardware. Procedural shading languages [14], such as RenderMan [1, 4], have been successfully used for off-line rendering for many years. More recently, procedural shading techniques have been adapted to real-time rendering, for both PixelFlow's SIMD pixel-processor arrays [12], and for conventional non-programmable graphics hardware [5, 13].

We have developed a real-time procedural shading system [15] that is designed to support graphics hardware with programmable vertex and fragment units. This system's compiler allows per-fragment, per-vertex, and per-primitive-group computations to be mixed within a single shader. The compiler's front end separates the three types of computations, and invokes separate compiler back ends to generate the code for each. The system can currently choose one of two different primitive-group back ends, one of three different vertex back ends, and one of two different fragment back ends.

Our system's first fragment back end generates code for standard OpenGL 1.2 hardware (plus a limited set of extensions), and the second fragment back end generates code specifically for NVIDIA's register-combiner hardware. Thus, this second fragment back end is the only one that targets what we consider to be programmable fragment hardware. This back end is the most complex one in our system, and is the subject of this paper.

In this paper, we

- Briefly describe the register combiner architecture.
- Describe in detail the algorithms used by our register-combiner compiler, and the reasons for choosing these algorithms.
- Compare our compilation strategy to that used by conventional VLIW compilers.
- Demonstrate that the efficiency of code generated by our compiler is comparable that of hand-written code.
- Discuss some of the lessons we learned by building and using the compiler, and the implications of these lessons for the design of future graphics hardware and compilers.

But first, we will describe the input to our compiler, and outline the basic problem that our compiler addresses.

The input to our register-combiner compiler back end is a directed acyclic graph (DAG) of operations such as ADD and MULTIPLY. This DAG represents a unification of a surface shader with zero or more co-compiled light shaders. The root node of this DAG returns a single RGBA color to be stored in the framebuffer. Most of the nodes that can appear in the DAG correspond directly to operators supported by our shading language.

The DAG does not contain any control constructs such as branches or loops – compiler writers would say that it represents a single *basic block*. Thus, the compilation problem that we address is that of generating and optimizing register-combiner code for a single basic block.

In developing this compiler back end, we have focused on the previously-unsolved problem of compiling as many operations as

*Gates Building; Stanford, CA 94305 USA.
email: {billmark | kekoa}@graphics.stanford.edu
www: http://graphics.stanford.edu/~{billmark | kekoa}

possible into a single programmable rendering pass. Other systems [5, 13] have already shown that multi-pass compilation is possible, and our system's other fragment backend uses these techniques. Currently, our register-combiner compiler will fail with an error message if a shader is too complex for a single rendering pass, although we have designed the compiler's infrastructure to facilitate the future addition of multi-pass compilation capability.

Our compiler is re-targetable to two different generations of the register combiner architecture. The first generation (GeForce1 and 2) supports two register combiners with two textures, while the second generation (GeForce3) supports eight register combiners, four textures, and a different mechanism for specifying constant values. Our compiler also makes partial use of the GeForce3's texture-shader hardware. Because our compiler's design is intimately tied to some of the properties of the hardware architecture, we will describe the architecture in detail before discussing the compiler further.

2 Register Combiner Architecture

Pipeline organization

Figure 1 depicts the complete architecture to which we are compiling [11]. The host CPU or vertex program supplies a set of interpolants (position, "colors", and texture coordinates) with each vertex. We use the more generic term *vertex interpolant* to refer to the interpolated "colors". The texture coordinates are used to index textures, or when the NV_texture_shader extension is available, to perform more complex texture-addressing operations. At the top of the register-combiner pipeline, the vertex interpolants and filtered texture colors are placed into specific registers.

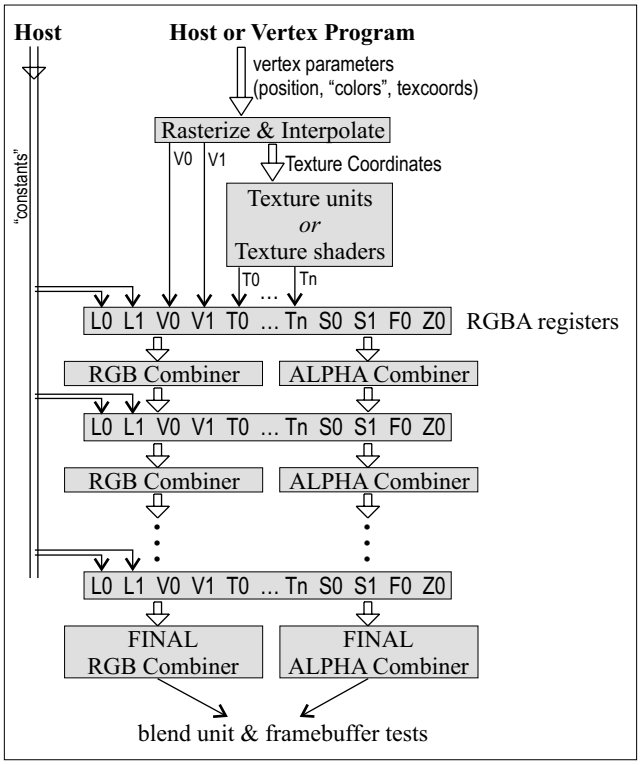

Figure 1: *The register combiner architecture.*

Each register combiner stage has an "RGB combiner" and "ALPHA combiner". These combiners can read values from several registers, operate on these values, and write the results to registers. The RGB and ALPHA combiners are controlled independently, and operate in parallel. For this reason, we consider each register-combiner stage to be similar to an instruction for a conventional VLIW CPU architecture. VLIW instructions allow several completely independent register-to-register operations to be performed concurrently.

Any register that is not written to by a particular register-combiner stage preserves its value from the previous stage. Again, this behavior is consistent with thinking of the register combiners as a series of register-to-register VLIW instructions. On a GeForce3, eight such VLIW instructions are available per rendering pass, and on a GeForce1, two such instructions are available per rendering pass.

The architecture also includes special "final" RGB and ALPHA combiners. These final combiners calculate the RGBA value to be placed in the framebuffer from values taken from the RGBA registers. The capabilities of these final combiners are different from those of the standard combiners. For example, the final combiners operate on unsigned values, while the standard combiners operate on signed values. For simplicity, our compiler only uses the final combiners to choose which RGB and A registers will be written to the framebuffer, and to perform simple input mappings such as $y = (1 - x)$.

The architecture has two read-only "constant" registers, as well as a read-only zero-valued register (Z0). The values of the constant registers are set by the host at pipeline-configuration time. Our system uses these constant registers to hold both true constant values and per-primitive-group values. On the GeForce1, the constant registers are *global*, in the sense that the first constant register has the same value at every combiner stage, as does the second constant register. We refer to these global constant registers as C0 and C1. On the GeForce3, which supports the NV_register_combiner2 extension, the constant registers are *per-stage*, meaning that they can hold a different value at every combiner stage. This difference in behavior has a surprisingly broad impact on the compiler, and to emphasize the difference from the GeForce1's constant registers we refer to the GeForce3's constant registers as L0 and L1.

Internal combiner structure

Figure 2 shows the internal structure of an RGB (3-vector) combiner. The ALPHA (scalar) combiner is very similar, but because it operates on scalar values it can not perform dot products. The complexity of this internal structure is the main feature that distinguishes the register combiner architecture from most VLIW architectures. In a typical VLIW architecture, the register-to-register operations are simple, but in the register combiner architecture they can be quite complex – e.g. the combination of two dot products, a multiplexing operation, and a scale/bias.

Table 1 lists the possible combinations of scale, bias, and clamping operations that can be performed by the input-mapping units and output-mapping units in a combiner. Note that while the input-mapping units can be controlled independently, the three output-mapping units in an RGB or ALPHA combiner share the same configuration. This coupling introduces added complexity to the compilation algorithms.

If an RGB or ALPHA combiner's ADD/MUX unit is not used, we can split the combiner into two independent pieces. We refer to each piece as a *half combiner*, as illustrated in Figure 3. In this case, we think of the VLIW instruction as controlling up to two half RGB combiners, and two half ALPHA combiners. Unfortunately, there is still some potential coupling between half combiners, because the OUTMAP must be the same in the two halves.

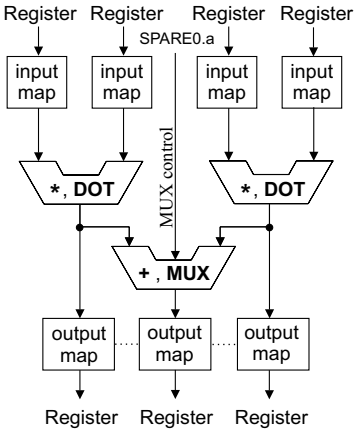

Figure 2: *Internal structure of a standard RGB combiner. All operations are performed on 3-vectors. The DOT/MULTIPLY unit can be configured to perform either a dot product or a vector multiply. The ADD/MUX unit can be configured to perform either a vector add or a vector multiplexing operation. A standard ALPHA combiner is similar to the standard RGB combiner, but operates on scalars.*

INMAP's	OUTMAP's
$-x$	$0.5x$
$\max(0, x)$	$2x$
$1 - x$	$4x$
$\pm 2(x - \frac{1}{2})^*$	$2(x - \frac{1}{2})^*$
$\pm(\max(0, x) - \frac{1}{2})^*$	$x - \frac{1}{2}^*$

Table 1: *Combiner input mappings (INMAP's) and output mappings (OUTMAP's) other than identity. All mappings clamp to [-1,1] after completion. The mappings with an asterisk can serve as either type of mapping (what we call a BOTHMAP) under some conditions.*

An RGB combiner always writes its outputs to the RGB portion of registers, and an ALPHA combiner always writes its outputs to the ALPHA portion of registers. The possibilities for combiner inputs are more complex. An RGB combiner may take an input from an RGB register, or from an ALPHA register by replicating the scalar value across all three components. Later, we will refer to this second possibility as a *TRIPLE* operation. An ALPHA combiner may take an input from an ALPHA register, or from the BLUE component of an RGB register (which we will refer to as a *BLUE* operation).

There are several other important properties of the register combiner architecture. First, if the RGB combiner performs an ADD in the ADD/MUX unit, it can not perform any dot products. This restriction allows sharing of the adder hardware. Second, the result of a DOT operation is replicated across all three RGB components. Third, the register combiner architecture uses a [-1,1] range for most values, but uses a [-2,2] range for values between the MUL/DOT unit and the scale/bias unit. Fourth, the control input for the MUX unit is always taken from the ALPHA portion of the SPARE0 register (S0.a). The control input is true if S0.a ≥ 0.5, and false otherwise.

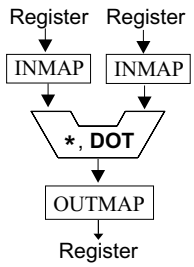

Figure 3: *Internal structure of a HALF combiner.*

DirectX 8

Microsoft's DirectX 8 pixel-shader instructions express a subset of the capabilities of the NVIDIA register-combiner and texture-shader OpenGL extensions. Although our compiler does not currently target DirectX8, we believe that it would be relatively easy to modify it to do so. The most important change would be the use of a simpler model for each register combiner.

3 Compilation Overview

We use an example to both illustrate the capabilities of our compiler and to provide a framework for understanding the compilation task. Figure 4 shows a shader written in our shading language. Most of the per-fragment operations expressed in the shading language

```
// Bumping-mapping function
surface float3
lightmodel_bumps(float3 a, float3 d, float3 s, texref bumps, floatv uv_bumps) {
    // Compute normalized tangent-space light vectors
    vertex perlight float3 Ltan = tangentspace(L);
    vertex perlight float3 Htan = tangentspace(H);
    // Lookup from bump map
    float4 Nlookup = texture(bumps, uv_bumps); // alpha has short len
    float3 Nbump = 2.0*(rgb(Nlookup)-triple(0.5));
    float N_avglen = Nlookup[3]; // Length of mipmap filtered N, before renorm
    // Diffuse
    perlight fragment float3 Lfrag = Ltan; // Interpolate vertex→ fragment
    perlight float NdotL = dot(Nbump, Lfrag);
    perlight float shadow = 4*(Lfrag[2] + Lfrag[2]); // Geometric shadow ramp
    perlight float3 diff = d * clamp01(NdotL) * clamp01(shadow) * N_avglen;
    // Specular
    perlight float3 Hlookup = cubenorm(Htan); // Interpolate and normalize
    perlight float3 Hnorm = 2.0*(Hlookup-.5,.5,.5);
    perlight float NdotH = clamp01(dot(Nbump, Hnorm));
    perlight float NdotHs = select(Hlookup[2] >= 0.5, NdotH, 0.0);
    perlight float NdotH2 = NdotHs * NdotHs;
    perlight float NdotH4 = NdotH2 * NdotH2;
    perlight float NdotH8 = NdotH4 * NdotH4;
    perlight float3 spec = NdotH8 * shadow * s;
    // Combine diffuse, specular, and ambient
    perlight float3 C = diff + spec;
    return integrate(rgb(Cl) * C) + a; // Sum over lights, and add ambient
}

// Surface shader for bowling pin
surface shader float4
bowling_pin (texref basemarks, texref decals, texref bumps, float4 uv) {
    // Per-vertex texture-coordinate computations omitted for brevity
    ...
    // Fragment computations
    float4 Decals = texture(decals, uv_decals);
    float4 BaseMarks = texture(basemarks, uv_basemarks);
    float Marks = alpha(BaseMarks);
    float3 Base = rgb(BaseMarks);
    float3 Ma = {.4,.4,.4}; float3 Md = {.5,.5,.5}; float3 Ms = {.3,.3,.3};
    float3 Kd = rgb((Decals over {Base, 1.0}) * Marks);
    float3 C = lightmodel_bumps(Kd * Ma, Kd * Md, Ms, bumps, uv_bumps);
    return {C, 1.0};
}
```

Figure 4: *Example surface shader (a bump-mapped bowling pin inspired by [17]). This shader compiles to eight register combiners, when it is used with a single light shader that returns a per-vertex light intensity. The* clamp01() *function clamps a value to the range [0,1], and the* tangentspace() *function transforms a vector into local tangent space.*

are operations such as adds, multiplies, dot products, and texture lookups. The shading language also allows the user to extract a scalar from a vector (using the [] *index* notation), and to join a three-vector and scalar to form an RGBA four-vector (using the {} *join* notation).

When we first decided to build a compiler for the register combiner architecture, we hoped to adapt dynamic-programming algorithms to the problem [3]. We had successfully used these algorithms in our fragment backend for non-programmable graphics hardware, as had Peercy *et al.* before us [13]. Both of these systems used these algorithms to find the lowest-cost set of rendering passes that could evaluate an expression, by matching a set of rules describing possible passes (i.e. instructions) to an expression tree.

However, this strategy proved to be unworkable for the register combiner architecture. With eight combiners, it isn't feasible to treat each rendering pass as a single instruction, because the instruction would be too complex. This problem can be partially circumvented by representing the structure of the fragment pipeline hierarchically, but these hierarchical representations don't allow dynamic-programming algorithms to properly track shared resources, such as interpolants and registers. Furthermore, with an eight-combiner pipeline it is crucial to be able to work with expression DAGs as well as expression trees, but the dynamic-programming algorithms do not support DAGs within a single instruction.

In general, optimal compilation is an NP hard problem [9]. Code generation for VLIW architectures is no exception [2]. We chose to adopt the usual approach to this challenge – we break the problem into a series of stages based on heuristic algorithms. While this strategy does not guarantee optimal code, it typically works well.

Our partitioning of the compilation problem roughly matches the structure of the register-combiner hardware. The five main stages of our compiler are the following:

1. **Extract texture-shader operations.** Extract any operations from the shader's fragment DAG that must be mapped to texture shaders rather than register combiners. We use simple pattern-matching techniques to attempt to compile these operations into a texture-shader configuration. The texture-shader hardware is too idiosyncratic to be considered to be truly programmable, so we will not discuss this part of the compiler any further.
2. **Rewrite DAG to use hardware operations (§4).** Rewrite the fragment DAG to use the basic operations and data types supported by the register-combiner hardware. The data types are three-vectors and scalars. The operations include ADD, MULTIPLY, MUX controlled by compare with 0.5, INMAP, OUTMAP, etc.
3. **Select instructions (§5).** Convert the DAG of basic operations into a DAG of register-to-register operations. That is, group basic operations together to form partial register combiners. Each partial combiner is either a half combiner (no ADD/MUX) or full combiner (uses ADD/MUX). Each partial combiner is either 3-vector (RGB) or scalar (ALPHA).
4. **Allocate pipeline-input registers (§6).** Allocate registers for the values that enter the top of the combiner pipeline. These values include vertex interpolants, results from the texture units (or texture shaders), and GeForce1 global constants. They do not include GeForce3 per-stage constants.
5. **Schedule instructions and allocate registers (§7).** The partial combiners from step #3 are placed into specific positions in the register-combiner pipeline. The compiler also allocates temporary registers and GeForce3 per-stage constant registers.

We will describe each of these stages in detail in a corresponding section of the paper. An important advantage of breaking the compilation process into multiple stages is the flexibility it allows in implementing each stage. For example, step #3 from the list above uses a top-down DAG traversal, but step #5 uses a bottom-up DAG traversal. If all of these stages were unified into a single compilation stage, the compiler could only use one traversal order.

Our compiler takes less than one second to compile a shader, because it uses greedy algorithms rather than exponential-time algorithms. We have not performed in-depth studies of the compiler's running time because the compiler is already fast enough that it could be invoked every time that a graphics application starts up.

4 Rewriting the DAG to use Hardware Operations

The DAG generated by our system's compiler front end uses data types and operations that correspond to those supported in our shading language. However, these data types and operations do not necessarily have a one-to-one correspondence with basic capabilities of the register-combiner hardware. For example, our language can express the addition of two four-vectors as one operation, but to implement this operation in the combiner hardware requires that both an RGB and ALPHA combiner be configured to perform ADD operations. Therefore, the first step in the code-generation process is to re-write the DAG to use fundamental combiner operations and data types.

The compiler performs the following transformations:

- The compiler replaces four-vector operations with corresponding combinations of three-vector and scalar operations. As a side effect, the root node of the DAG is split into a paired RGB root node and ALPHA root node.
- The compiler recognizes groups of operations that correspond to combiner input mappings and/or output mappings, and rewrites them as a single INMAP, OUTMAP, or BOTHMAP node. A BOTHMAP node indicates an operation that can be implemented using either an input mapping or an output mapping. The compiler uses a top-down DAG traversal to greedily perform this re-write. The compiler gives higher priority to matches with more complex mappings.
- The compiler recognizes all conversions between 3-vector and scalar data, and rewrites them to use either a TRIPLE operation or BLUE operation. Given a scalar `x` and a three-vector `y`, a TRIPLE operation represents `y[0]=x; y[1]=x; y[2]=x`, and a BLUE operation represents `x=y[2]`. These operations correspond to capabilities of the register-combiner hardware. An example transformation is the conversion of `x * y` into `TRIPLE(x) * y`. Another example is the conversion of `y[0]` into `BLUE(DOT(y, {1,0,0})`.
- The mathematical dot product operation (MATHDOT) is rewritten to use the combiner dot product operation (DOT), which produces a three-vector result. So, `MATHDOT(vec3,vec3)` becomes `BLUE(DOT(vec3,vec3))`.
- The language's `select()` operation (which is similar to the `a ? b : c` operator in C) is rewritten to use the MUX operation. The control input to the MUX operation is implicitly tested against the value 0.5.

When these transformations are applied to the example shader in Figure 5, the compiler produces the DAG shown in Figure 6.

The strategy for grouping language operations into INMAP, OUTMAP, and BOTHMAP operations is complex. Some of the INMAP operations include a clamp to the range [0,1]. If the compiler can guarantee that the value being operated on is already in the range [0,1], it is acceptable to use such an INMAP even though the shader did not explicitly request the clamp. The

```
surface shader float4
example(texref tex1, texref tex2,              // Specifies textures #1 and #2
        vertex float4 uv, vertex float4 c,     // Texture coords. and interp. color
        primitive group float g1,              // Slowly-changing value #1
        primitive group float g2) {            // Slowly-changing value #2
    float4 t1 = texture(tex1, uv);
    float4 t2 = 2*(texture(tex2, uv) - {0.5,0.5,0.5,0.5});
    float x1 = g1 * t2[2] + g2;
    float x2 = dot(rgb(t1), N);
    float3 x3 = rgb(t1)*x1 + x2*rgb(t2);
    float x4 = c[3]*t1[3];
    return {x3, x4};
}
```

Figure 5: *An example shader that we trace through the compilation process. Since we designed this shader for pedagogical purposes, it doesn't make much sense as a shader.*

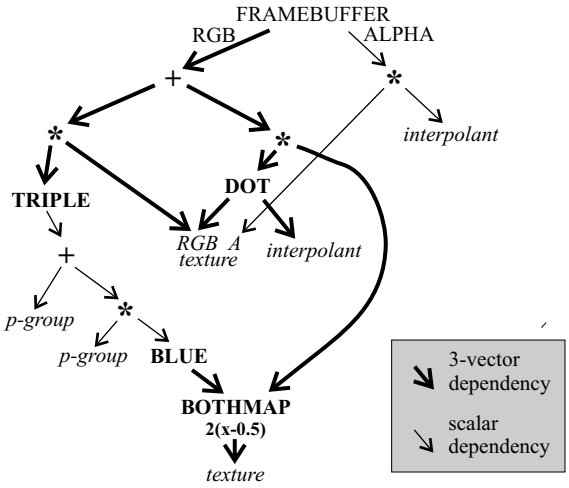

Figure 6: *The DAG for the example shader after it has been rewritten to use combiner operations. Arrows indicate dependencies (as is customary in DAGs), so the direction of data flow is against the arrows.*

compiler performs an interval-arithmetic analysis on the entire fragment DAG to detect these cases.

OUTMAP operations include multiply-by-2 and multiply-by-4. Therefore, a shader may specify a per-fragment multiply by 2.0 or 4.0. The compiler will issue an error for any other per-fragment use of constants outside the range [-1,1]. It would be possible to support multiplies by any constant in the range [-4,4] by automatically factoring the constant (e.g. $3.5x \rightarrow 4.0 \cdot 0.875x$), but the compiler doesn't currently perform this transformation.

5 Instruction Selection
(Creating a DAG of register-to-register operations)

Roughly speaking, the next stage of the compilation recognizes groups of DAG operations that can be mapped to a single register combiner. For example, the compiler maps a three-vector sum of products to an RGB register combiner. Thus, this stage of the compiler decides where to place operations within the structure of a register combiner.

More precisely, this stage recognizes groups of operations that can be mapped to a *partial register combiner*. We define a partial register combiner as a subset of a combiner that reads one or more inputs from register(s), operates on them, and writes the result(s) to registers. Figure 7 shows the six types of partial combiners, and Figure 8 shows the output of this compilation stage for our example shader.

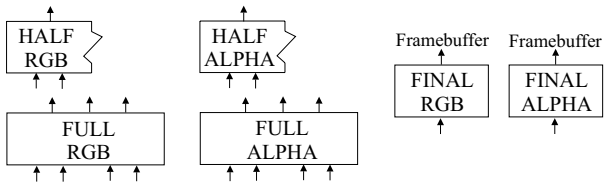

Figure 7: *The six types of partial combiners that we use to represent register-to-register operations. In this figure, the outputs from each combiner are on the top, and the inputs are on the bottom. Note that the internal structure of a "HALF" combiner is described earlier, in Figure 3.*

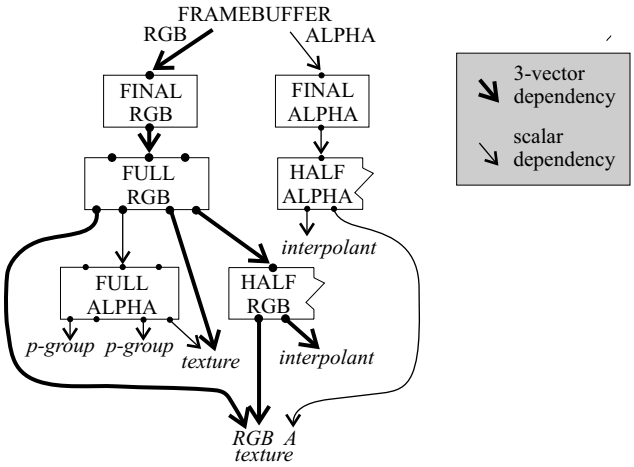

Figure 8: *The DAG of partial register combiners for the example shader. Again, arrows indicate dependencies.*

The output of this stage of the compilation is a new DAG whose nodes consist of partial combiners (i.e. register-to-register operations). For a conventional VLIW architecture, this compilation step can be greatly simplified or eliminated, because language operations such as ADD and MULTIPLY already directly correspond to the architecture's register-to-register operations. This direct correspondence is due to the fact that a conventional VLIW architecture's functional units always take their inputs from a register and write their outputs to a register. In contrast, the register combiner architecture directly connects some functional-unit outputs to other functional-unit inputs. Some application-specific signal processor architectures also have this property [16].

When deciding what algorithm to use for this stage of our compiler, we considered both dynamic programming algorithms and greedy algorithms. We decided against using a BURG-style dynamic programming algorithm [3] because this class of algorithms does not readily support some optimizations we wanted to perform. More specifically, the BURG-style algorithms have two shortcomings. First, they work best when an "instruction" produces only one result, but our FULL combiners produce up to three results. Second, these algorithms are fundamentally designed for expression trees rather than expression DAGs, but we need to support DAG-like behavior (and associated optimizations) even within a partial combiner. Despite these drawbacks, it would be possible to design this stage of a register-combiner compiler using a BURG-style algorithm, and such an approach might be attractive as part of a retargetable compiler.

We considered several classes of greedy algorithms for this stage of the compiler. Broadly speaking, these algorithms can work either from the root of the DAG towards the leaves (top-down) or from the leaves towards the root (bottom-up). Once these algorithms

have begun filling a particular partial combiner, they can work either from the outputs towards the inputs, or from the inputs to the outputs.

The structure of a partial register combiner is mostly tree-like; the only exception is the DOT/MUL outputs from a FULL combiner. Given this tree-like structure, it is natural to work from the output(s) of a combiner towards the inputs, and this is the strategy that we chose to use. This traversal direction corresponds to a top-down traversal of the DAG, which we also chose to use. It would be possible to combine an outputs-to-inputs combiner-filling order with a bottom-up DAG traversal, but it would be more awkward to do so.

Algorithm Details

Our algorithm fills one partial combiner at a time. When it begins working on a new partial combiner, it starts with one particular DAG node that it will place in that combiner. The algorithm chooses this node by taking it from the head of a priority queue. This queue contains all DAG nodes whose parents have all been placed into partial combiners. At the start of this compilation stage, the queue is initialized with the RGB and ALPHA root nodes from the DAG.

The priority of nodes in the queue is their weighted distance from the furthest leaf node beneath them. Thus, nodes closer to the root of the DAG have greater priority. The weight of an ADD or MUX node is slightly lower than that of other nodes (by epsilon), which improves compilation quality for some combinations of ADD and MULTIPLY operations.

Once the compilation algorithm has chosen a particular DAG node to place into a partial combiner, it continues working on that partial combiner until it is as full as possible. If the original DAG node is the root RGB or root ALPHA node, then the partial combiner is marked as a FINAL RGB or FINAL ALPHA combiner. Otherwise, the type of combiner is determined by whether or not an ADD or MUX node is encountered before a MUL or DOT node. In the first case, a FULL combiner is created and filled, and in the second case, a HALF combiner is created and filled.

Our algorithm begins at the output of the combiner and works towards the inputs. For a FULL combiner, the algorithm first fills the OUTMAP unit; then the ADD/MUX unit; then the left DOT/MUL unit; then the two INMAP units for this DOT/MUL. Next, it fills the right DOT/MUL unit and its two INMAP units. One of the possible choices when "filling" a unit is to configure it for pass-through.

For a FULL combiner, the two "side" DOT/MUL outputs are handled specially, because the algorithm never uses them to start working on a combiner. Instead, these outputs are handled when the DOT/MUL unit is filled, by checking to see if the relevant DOT/MUL DAG node has more than one parent (i.e. its result needs to be stored in a register). There are some additional complications in this procedure when the combiner's OUTMAP is set to something other than pass-through.

As a combiner is filled, the algorithm can dynamically transform the original DAG in one of two ways to better match the combiner structure. First, it can move a TRIPLE or BLUE operation leafward to allow additional operations to be packed into the combiner. For example the expression `y * TRIPLE(INMAP(x))` will be rewritten as `y * INMAP(TRIPLE(x))`. The second expression is equivalent to the first, and matches the combiner structure better.

The second type of DAG transformation is the replication of a TRIPLE, BLUE, or INMAP node when the node has more than one parent. This transformation provides each such parent with its own copy of the node. The compiler performs this transformation when it is necessary to allow the TRIPLE, BLUE, or INMAP node to be incorporated into the current partial combiner. In compiler terminology, this transformation is referred to as *forward substitution* [9].

In general, the need to efficiently handle cases where a DAG node has more than one parent (i.e. its result is used more than once) adds significant complexity to the compiler. To handle these cases, the compiler must keep track of which intermediate results can be stored into a register, and which are blocked from being stored in a register by other operations that have already been placed into the combiner.

6 Register allocation for pipeline inputs

The compiler performs most register allocation work at the same time that it schedules partial combiners into a pipeline (as we will discuss in section 7). For example, if a variable holds the output of a partial combiner, the compiler will perform register allocation for that variable during scheduling.

However, the compiler uses a separate, earlier step to allocate registers to values that enter the top of the combiner pipeline. We refer to these values as *pass inputs*. The pass inputs include vertex-to-fragment interpolants, results from the texture units (or texture shaders), and GeForce1 global constants.

The compiler performs this register allocation separately because there are so many constraints on the allocation, and because good allocation is often critical to packing as much computation as possible into a single rendering pass. For example, on a GeForce1, vertex-to-fragment interpolants must be placed in either the primary color register (V0) or the secondary color register (V1). But, these two registers are not interchangeable – V0 supports RGBA (4-vector) interpolants, but V1 only supports RGB (3-vector) interpolants. On a GeForce3, the texture registers T0-T3 can also be used to hold interpolants, by using the texture-shader pass-through mechanism.

The pass-input register allocations are further constrained because the RGB and ALPHA portions of a register are often tightly coupled. For example, allocating a texture register to a texture lookup is an all-or-nothing proposition. In contrast, it is possible to allocate the RGB portion of a constant register to a 3-vector, and allocate the ALPHA portion of that same constant register to a completely unrelated scalar. Our compiler correctly handles both of these cases.

We use a greedy algorithm to perform pass-input register allocation. To allow our algorithms to adapt to different architectures (GeForce1 vs. GeForce3), we use a fairly general mechanism to describe the capabilities of each register for each class of pass inputs.

There are four classes of pass inputs: True constants; primitive-group values; vertex-to-fragment interpolants; and texture-unit outputs. For each register and each class of pass inputs, we indicate which part(s) of the register are available. The two parts are RGB and ALPHA. So, secondary-color register V1 is marked as having its RGB part available for vertex-to-fragment interpolants, but not its (non-existent) ALPHA part. Global constant register C0 is marked as having RGB and ALPHA available for true constants and for primitive-group values, but not for vertex-to-fragment interpolants.

For each class of pass inputs, we also specify whether or not each of five possible sets of possible manipulations is allowed during register allocation:

- **SPLIT**: A single RGBA value can be split up into an RGB part and an ALPHA part, with each part placed in a different register. This manipulation is allowed on constants, for example.
- **JOIN**: An unrelated RGB and ALPHA value can be allocated to the same RGBA register. This manipulation is not legal for

textures, and is only legal for vertex-to-fragment interpolants when other parts of the compiler support it.

- **PRETRIPLE**: A scalar value can be replicated across an RGB register, rather than being provided in an ALPHA register as is typical.
- **SCALAR_AS_BLUE**: A scalar value can be provided in the BLUE portion of a register.
- **BLUE_AS_ALPHA**: The blue part of an RGB value can be provided via the ALPHA portion of a register, rather than via the RGB portion as is typical.

The complete details of the greedy pass-input register allocation algorithm are too complex to provide here, so we will just summarize it. The algorithm classifies variables into one of several categories of scheduling difficulty. For example, a four-vector variable that is not allowed to be **SPLIT** is put into a more difficult scheduling category than a scalar variable that is allowed to be **JOIN**'d. The compiler first allocates registers to variables in the hardest category, then proceeds to variables in successively easier categories.

When it is time for the compiler to assign a particular variable to a register, the compiler must choose one register from the pool of not-yet-allocated registers. In general, the greedy algorithm chooses the least-flexible register that can accommodate the variable. For example, the compiler prefers to place a three-vector in an RGB register rather than an RGB+ALPHA register. In some cases, the compiler bases its choice on comparisons between several counters that it maintains. These counters provide information about the number of remaining unscheduled variables and registers of certain types.

To allow unrelated three-vector and scalar interpolants to be **JOIN**'d (that is, to share a register such as V0), the compiler must modify the per-vertex code so that it combines the unrelated values into one RGBA value. Our compiler doesn't yet make this modification, so **JOIN**'s are currently forbidden for interpolants.

7 Instruction scheduling and register allocation

The final major stage in the compilation process is to schedule the DAG of partial combiners into an ordered pipeline of complete register combiners. During this stage, the compiler also allocates temporary registers and GeForce3 per-stage constant registers. The output of this stage is a complete configuration of the register-combiner pipeline. Figure 9 illustrates the output of this stage for our example shader.

The task performed by this stage is very similar to classical VLIW instruction scheduling and register allocation. Chapter 7 of Ellis's book [2] discusses some of these algorithms. The task is somewhat simpler for the register combiner architecture than it is for some VLIW architectures because register-combiner operations always complete in one "cycle". However, our algorithm must handle both three-vector and scalar operations, and must properly cope with various quirks of the register combiner architecture.

Our compiler uses a variant of the *operation scheduling* approach to instruction scheduling [2]. Our algorithm repeatedly picks the deepest (furthest from root) node in the partial-combiner DAG, and places this node as close as possible to the top of the register-combiner pipeline. This approach gives the highest scheduling priority to operations that are part of a long dependency chain. Since these chains often determine the total number of combiners required by a shader, it is critical to give them high priority during scheduling.

Register allocation is performed on the fly, using a simple greedy algorithm. After scheduling an operation that writes to a register, the compiler chooses a free register and tentatively reserves it

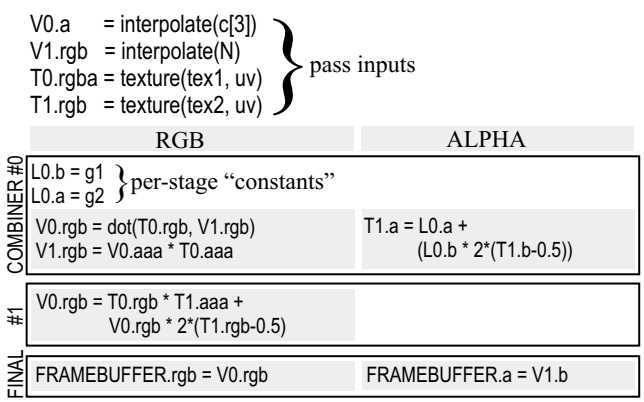

Figure 9: *The complete pipeline configuration for our example shader. The compiler has placed both of the HALF combiners shown in Figure 8 (one RGB and one scalar) into the RGB portion of combiner #0.*

through the end of the combiner pipeline. After all operations that use this register have been scheduled, the compiler releases the tail portion of the tentative reservation. We treat RGB registers and ALPHA registers as completely independent entities. The initial allocation status of registers is determined by the results from the pass-input register-scheduling stage described earlier.

We could have used a different VLIW instruction-scheduling algorithm called *list scheduling* [9]. For our purpose, the most important difference between operation scheduling and list scheduling is that list scheduling finishes filling an entire instruction (combiner) before moving to the next one. The bookkeeping required for list scheduling algorithms is simpler than that required for operation scheduling algorithms. If we were to re-implement this stage of our compiler, we would consider switching to a list-scheduling algorithm. However, we do obtain one advantage from the operation-scheduling algorithm: It more efficiently supports the lazy insertion of new operations in the DAG to spill the SPARE0.ALPHA register to other registers. The compiler must move values in and out of the SPARE0.ALPHA register when there is contention for it due to its exclusive ability to control the combiner's MUX operation.

If the partial-combiner DAG is well balanced (tree-like rather than chain-like), then any depth-first scheduling algorithm will use a large number of live temporary registers. We were initially concerned that this behavior would result in overflow of the register file for some shaders, but we have not found this problem to occur in practice. If this behavior had been a problem, we could have used one of several known strategies for handling it [7, 10].

When our scheduling algorithm attempts to place a partial combiner as close as possible to the top of the pipeline, it must honor several constraints:

- The placement must honor the dependency constraints expressed in the partial-combiner DAG. That is, all inputs to the combiner must be computed in earlier combiner stages.
- There must be enough free registers for the partial combiner's outputs.
- If the partial combiner is a HALF combiner and it is being paired with another HALF combiner, the OUTMAP's of the two partial combiners must be the same.
- If the partial combiner includes a MUX operation, the control input must be in the SPARE0.ALPHA register. The register allocator preferentially places control variables in this register, but there may be contention for the register.
- On GeForce3, the set of partial combiners (both RGB and ALPHA) that are scheduled into the same complete combiner

must not require a total of more than two RGBA per-stage constants. The compiler uses the algorithm described in §6 to perform the optimized allocation of each stage's constant registers.

The compiler typically schedules scalar operations into ALPHA combiners, but under some circumstances the compiler will schedule a scalar operation into an RGB combiner. When checking to see if a scalar operation can be placed into a particular pipeline stage, the compiler first tries the ALPHA combiner, then the RGB combiner. The compiler will only use the RGB combiner if the combiner can be configured to produce the desired scalar result in all three RGB components. The first version of our compiler did not have this capability, but we found it to be important for shaders with a high ratio of scalar operations to vector operations.

8 Efficiency of Generated Code

The code generated by our compiler is efficient, as compared with manually-generated code for the same algorithm. The bowling-pin shader in Figure 4 compiles to eight register combiners. By using the compiler's output to guide source-code-level optimizations in the shader, it is possible to reduce it to seven combiners, with the side effect that the optimized source code is messier. We have not been able to do any better by hand coding the shader, although a hand-coded shader could use the final combiner in place of one of the standard combiners.

We have found this result to be typical. Usually the number of combiners required by a complex shader is determined by the longest dependency chain in the computation. Since the compiler schedules operations in this chain first, it usually schedules them well.

The user must modify the shader source code to perform optimizations that the compiler can not perform. In particular:

- The compiler can not design algorithms that map well to register combiners. The algorithm that we use in our bump-mapping routine was designed for register combiners [6], and thus shaders that call this routine usually compile well. Note that using a high-level language for hand-crafting of shader routines allows the compiler to efficiently combine these routines with other code, which would not be possible if the hand-crafted routines were written in assembly language.
- We forbid the compiler from reordering computations, just as ANSI C forbids compilers from doing so. Given the limited precision and range of register-combiner data types, we believe it would be unwise to allow the compiler to reorder operations that are mathematically commutative or associative, but not computationally so. When we manually optimize at the source-code level, we typically reorder computations to better match the sum-of-products structure of the register combiners, and to reduce the length of dependency chains.
- Given the constraints of our overall system design, the compiler can not reorganize texture data. A common optimization that we perform manually is to combine an RGB texture with a scalar texture to make a single RGBA texture.

9 Discussion

Vectors and Scalars

We were surprised to find that it is common for shaders to use more scalar operations than three-vector operations. Most shaders that use the register-combiner bump-mapping algorithm fall into this category, as do some shaders that we have written for volume rendering. By enhancing our compiler's scheduler to allow it to place a scalar computation into an RGB combiner, we significantly reduced the total number of combiner stages required by these shaders. In retrospect, perhaps we should have anticipated this situation, since scalar operations are common in RenderMan shaders.

This common use of scalar operations has implications for any programmable fragment architecture that supports fine-grained SIMD computations (i.e. operations on 3-vectors or 4-vectors). With a naive compiler and hardware, the utilization of the hardware's functional units will be poor for scalar computations. To improve utilization, either the compiler must group unrelated scalar computations together into vectors, or the hardware must perform some run-time scheduling of functional units. For hardware that supports general dependent texture reads, this same potential for under-utilization occurs with two-vector texture-coordinate computations (which we plan to support in our shading language by adding a `float2` data type).

Compiler Complexity and Hardware

Our compiler is complex, and it is worthwhile to examine some of the forces that drove this complexity. Broadly speaking, these forces fall into three categories: Those that are specific to the register combiner architecture, those that are specific to VLIW fragment architectures, and those that will be common to almost any fragment-processing architecture.

Much of the complexity in our compiler is driven by the large variety of data types it supports. Almost all graphics hardware has some support for data-parallel computations on short vectors. To compile efficiently to such hardware, code generation and register allocation algorithms must support scalars, three-vectors and four-vectors. These algorithms are more complex than equivalent scalar-only algorithms.

This inherent complexity is aggravated by the strange swizzling capabilities of the register combiner architecture. For example, the architecture can perform a scalar computation directly on the BLUE component of an RGB value, but not on the RED or GREEN components. The DirectX8/NVIDIA vertex program architecture [8] provides a more orthogonal set of swizzling capabilities.

The register combiner architecture uses an enormous variety of precisions and ranges for data at different points in the pipeline. Unfortunately, these data types are not orthogonal to operations (e.g. add, multiply, interpolate), so it is impossible to cleanly and efficiently expose the data types at the language level. Table 2 shows the scope of this problem. In some cases, we have been able to partially work around the limitations of the architecture. For example, our compiler automatically collapses constants and vertex interpolants into a [0,1] range then uses combiner input mappings to expand them back to [-1,1]. However, we have not been able to find a reasonable strategy for abstracting the [-1,1] and [-2,2] ranges that are supported within a combiner, so we just directly expose the combiner behavior to the user. We hope that future architectures will eliminate this problem by orthogonally supporting a small number of data types. Furthermore, we encourage all hardware vendors to agree on a single set of these data types, so that they can be exposed in portable shading languages and API's.

In order to virtualize hardware resources using multi-pass rendering, the hardware must be able to store data in the framebuffer using any of the hardware's data types. The GeForce3 lacks this capability for its high-precision texture-shader intermediate results, and thus it is impossible to properly virtualize the texture-shader hardware. This inability to store certain intermediate results in the framebuffer also makes it difficult to debug shaders.

VLIW architectures are inherently more difficult to target than RISC-like architectures, because the compiler must extract

Range	Where it occurs
$[-1,1]$	registers, most computations, GeForce3 textures
$[0,1]$	interpolants, constants, GeForce1 textures, final combiner inputs
$[-2,2]$	output of ADD and DOT units
$[0,4]$	intermediate result in final RGB combiner

Table 2: *Numeric ranges in the register combiner architecture and where they occur.*

instruction-level parallelism from the user's code. It is not clear that VLIW instruction sets are the best choice for graphics architectures, because concurrent processing of multiple fragments or vertices can provide enormous parallelism without relying on instruction-level parallelism [8].

The register combiner architecture is a particularly complex VLIW architecture because the inputs and outputs of some functional units are not directly connected to a register file. Although this property of the architecture makes compilation more difficult, it has the obvious advantage that it allows a greater fraction of the hardware's gates to be used for functional units, rather than register files.

Hardware Resource Constraints

The key resource constraints in a register-combiner pipeline are combiners, registers, and pass inputs (interpolants and textures). The two standard combiners in the GeForce1 pipeline are not sufficient to write really interesting shaders, but the eight combiners in the GeForce3 pipeline allow enormous creativity in shader design. On a GeForce3, we have found that pass inputs are generally the most critical resource, but that in some cases we run out of combiners. We have not found any shader that runs out of temporary registers before other resources.

10 Conclusion

We have demonstrated that it is possible to efficiently compile shaders written in a high-level shading language to a programmable fragment pipeline. The user can perform additional hand optimization at the source-code level, avoiding the need to abandon the high level language to achieve maximum performance. Our programmable-shading system is available for download on the Internet at http://graphics.stanford.edu/projects/shading.

We hope that the task of building a compiler for future fragment architectures will be easier, because Moore's law provides hardware vendors with the opportunity to design more powerful hardware with cleaner programming interfaces. As real-time shading languages become more widely used, we also expect that compiler issues will be considered during the hardware design process. The coupling of hardware and compiler design has strongly influenced CPU architectures over the past fifteen years, and we believe that compiler technology will similarly influence the design of graphics hardware.

11 Acknowledgments

Pradeep Sen and Ren Ng helped to debug this compiler, sometimes involuntarily. Pat Hanrahan and the members of the Stanford graphics hardware group made numerous useful suggestions as we built this compiler. An early discussion with Reinhard Wilhelm and Philipp Slusallek helped us decide how to partition the compilation problem.

This research was performed as part of the Stanford real-time programmable shading project, which is sponsored by ATI, NVIDIA, SONY, and Sun.

We thank Matt Papakipos, Mark Kilgard, Svetoslav Tzvetkov, and Nick Triantos for numerous valuable discussions, and for providing us with extraordinary access to hardware, drivers, and bug fixes.

References

[1] Anthony A. Apodaca and Larry Gritz. *Advanced RenderMan: Creating CGI for motion pictures*. Morgan Kaufmann, 2000.

[2] John R. Ellis. *Bulldog: A compiler for VLIW architectures*. MIT Press, 1986.

[3] Christopher Fraser and David Hanson. *A Retargetable C Compiler: Design and Implementation*. Addison-Wesley, 1995.

[4] Pat Hanrahan and Jim Lawson. A language for shading and lighting calculations. *Computer Graphics (Proceedings of SIGGRAPH 90)*, 24(4):289–298, August 1990.

[5] Paul Jaquays and Brian Hook. *Quake 3: Arena Shader Manual, Revision 10*, September 1999.

[6] Mark J. Kilgard. A practical and robust bump-mapping technique for today's GPU's. Technical report, NVIDIA Corporation, February 2000. Available at http://www.nvidia.com/.

[7] T. Kiyohara and J. Gyllenhaal. Code scheduling for VLIW/superscalar processors with limited register files. In *Proceedings of the 25th Annual Workshop on Microarchitecture (MICRO 92)*, pages 197–201, Portland, OR, December 1992.

[8] Erik Lindholm, Mark J. Kilgard, and Henry Moreton. A user-programmable vertex engine. In *Computer Graphics Annual Conference Series (Proceedings of SIGGRAPH 2001)*, Los Angeles, CA, August 2001.

[9] Steven S. Muchnick. *Advanced compiler design and implementation*. Morgan Kaufmann, 1997.

[10] Cindy Norris and Lori L. Pollock. An experimental study of several cooperative register allocation and instruction scheduling strategies. In *Proceedings of the 28th Annual Workshop on Microarchitecture (MICRO 95)*, pages 169–179, Ann Arbor, MI, December 1995.

[11] NVIDIA Corporation. *NVIDIA OpenGL Extension Specifications*, March 2001.

[12] Marc Olano and Anselmo Lastra. A shading language on graphics hardware: The PixelFlow shading system. In *Computer Graphics Annual Conference Series (Proceedings of SIGGRAPH 98)*, pages 159–168, Orlando, FL, July 1998.

[13] Mark Peercy, Marc Olano, John Airey, and Jeff Ungar. Interactive multi-pass programmable shading. In *Computer Graphics Annual Conference Series (Proceedings of SIGGRAPH 2000)*, pages 425–432, New Orleans, LA, July 2000.

[14] Ken Perlin. An image synthesizer. *Computer Graphics (Proceedings of SIGGRAPH 85)*, 19(3):287–296, July 1985.

[15] Kekoa Proudfoot, William R. Mark, Svetoslav Tzvetkov, and Pat Hanrahan. A real-time procedural shading system for programmable graphics hardware. In *Computer Graphics Annual Conference Series (Proceedings of SIGGRAPH 2001)*, Los Angeles, CA, August 2001.

[16] Ken Rimey and Paul N. Hilfinger. Lazy data routing and greedy scheduling for application-specific signal processors. In *Proceedings of the 21st Annual Workshop on Microprogramming and Microarchitecture (MICRO 88)*, pages 111–115, San Diego, CA, November 1988.

[17] Steve Upstill. *The RenderMan companion: A Programmer's Guide to Realistic Computer Graphics*. Addison-Wesley, 1990.