Abstract Shade Trees

Morgan McGuire, Brown University
George Stathis, Harvard Extension School
Hanspeter Pfister, MERL
Shriram Krishnamurthi, Brown University

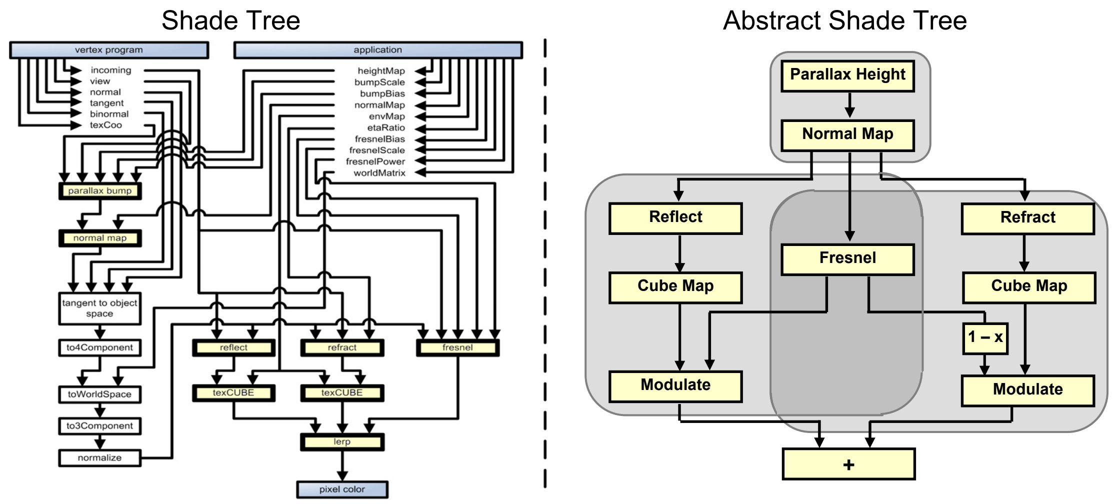

Figure 1: *A conventional Shade Tree (left) for a "Bumpy Glass" shader. The equivalent Abstract Shade Tree (right) is simpler; the compiler automatically handles vector basis conversion, normalization, and parameter linkage. Outline bubbles mark underlying features (clockwise from top): parallax mapping, refraction, and refraction. Note that the Fresnel term cross-cuts two features.*

Abstract.
As GPU-powered special effects become more sophisticated, it becomes harder to create and manage effect interaction using the fairly primitive shading languages. This difficulty also introduces a workflow problem: artists design effects but only programmers can implement them, making it impossible for them to work asynchronously.

To address these problems we present *abstract shade trees* and heuristic algorithms that operate over them. The trees allow designers to easily create effects by connecting primitives such as cube mapping and modulation. These primitives publish semantically rich types that encapsulate notions like vector basis and normalization. The algorithms employ these published types to automatically infer atomic and compound connectors between the primitives, and generate code for the tree. We also describe a visual editing environment for specifying the trees.

Our data structure and algorithms spare designers from having to specify low-level programming details, enabling them to experiment without depending on programmers. The algorithms ensure that the generated code will be free of type-mismatches, a problem in previous shade trees. The abstract shade tree can also naturally express high-level features like shadows and reflections whose implementations overlap; that cross-cutting has made them difficult to modularize in more traditional ways. In experiments, the generated shaders are as efficient as hand-written code.

1. Introduction

Modern GPUs manifest another turn of Ivan Sutherland's "Wheel of Reincarnation," where general-purpose and specialized hardware alternate as the best implementation technology. Unfortunately, 3D graphics and other media APIs have not kept pace with the move to general-purpose graphics processors. Current APIs for GPU programming avoid layering, moving the application logic very close to the hardware. The emphasis is on time-to-market, not on robustness, and exposing hardware peculiarities is seen as a competitive advantage. With the notable exception of Sh [McCool02], current shading languages for GPUs (e.g., GLSL) lack language mechanisms for encapsulation, modularity, and abstraction. This makes it very hard to create and maintain long GPU programs. Because GPUs are inherently digital signal processors with long pipelines, no stack, and unusual performance limitations on random access to memory, it is likely that they will remain difficult to program by hand.

The lack of abstraction also makes it difficult for artists to create new graphics effects without learning how to program. It leads to a close coupling between programmers and artists: The artist tells the programmer what the effect should be, the programmer writes the shader, the artist changes the requirements, the programmer revises the code, etc. Once a special effect is created, it is practically impossible to re-use it within the framework of another shader without substantial additional coding. This stifles productivity and creativity.

One way to address this problem is to look for other representations of shader programs. Cook [1984] introduced the notion of modular shading components with **shade trees**. He describes a system in which basic shading blocks (called **atoms)** are nodes linked by edges representing variables. Because shaders have one output (the pixel color) and many inputs, the root of the tree represents the output and the leaves are the input.

Shade trees weren't originally intended for visual programming. In fact, Cook's shaders were authored as code and converted to trees as a post-process for compilation. Abram and Whitted [1990] invert this paradigm with Building Block Shaders. Their visual programming tool represents shaders directly as directed acyclic graphs (DAGs), which are essentially Cook's trees extended with side effects and confluent paths. Although they are DAGs, we continue to refer to visual shader representations as *trees* in deference to the original work.

Implementations of this idea have since been created for today's GPUs and hardware programming languages [Borau04][Unreal06]. Figure 1 (left) shows an example shader for "bumpy glass" that combines the rendering effects of parallax mapping, Fresnel reflection, and Fresnel refraction. The immediate advantage over source code is that shade trees encourage experimentation and are more approachable by non-programmers [Abram90]. Programmers implement a library of carefully optimized atoms, which non-programmers combine to build shaders. A visual editor with preview capabilities offers the advantage that artists can experiment without understanding the inner workings of the atoms.

Figure 1 also demonstrates some of the drawbacks of shade trees. Although this example uses a simple shader, the tree appears complicated and visually cluttered. The tree also contains atoms that an artist might not understand, like *TangentToObjectSpace* and *ExtendToHomogeneousVector*. Most importantly, there is the potential for type mismatches between input and output arguments of atoms. When they introduced the first visual shade tree editor, Abram and Whitted [1990] noted:

"One problem with [the Building Block] graphical shading language is the potential for type mis-matching."

In fact, this problem extends beyond storage types to interface mismatches between atoms, e.g., assumptions like "the light vector has unit length" or "RGB values are pre-multiplied by the alpha channel." It is almost impossible for the user to ensure that the types are correct when the programming tool conceals those types.

In this paper we introduce **abstract shade trees** for pixel shaders. Of course, many of the same ideas can be applied to vertex shaders (and other programmable units, e.g., the geometry shaders in DirectX 10).

Figure 1(right) shows the abstract shade tree for our previous example. This tree is visually uncluttered and only contains atoms that are meaningful to the user. To avoid type and interface mismatches we implement parameter matching with automatic type coercions. This allows us to abstract all parameters between two atoms into a single data connection. The bubbles surrounding sub-graphs indicate the boundaries of the features that were combined to create the graph.

We build a system for abstract shade trees that consists of a visual programming tool and a **weaver** program that translates the abstract shade tree into OpenGL Shading Language (GLSL) code. The weaver determines how to connect parameters between atoms. It automatically introduces new atoms into the graph in cases where there is no output that exactly matches each input.

Automatic parameter matching by the weaver is our primary contribution. Because of it, abstract shade trees not only simplify shader authoring but also allow a programmer to change an atom interface without affecting the artist using that atom. This in turn allows complete separation between the roles of programmers and artists, thus enabling asynchronous workflow. Automatic matching also guarantees that the output is legal and correctly typed code, which solves the type mismatch problem noted by Abram and Whitted.

We also introduce the notion of feature-based programming to GPU shader development. Artists can easily extend and combine previously completed effects (features) whose boundaries are displayed in the editor.

2. Related Work

2.1. Shade Trees

Cook's [1984] Shade Trees first introduced the notions of shading languages, uniform shading parameters, and modular shading components. Abram and Whitted's [1990] Building Block Shaders (BBS) was the first visual programming tool for shaders. Others have since created implementations similar to BBS for today's GPUs and hardware programming languages [Borau04][Unreal06]. These all map **one-to-one** traditional programming elements like variables and functions to visual elements. Thus, while making programming more approachable, they still retain the complexity of source code. We extend the previous work by abstracting programming elements, solving the type mismatch problem, and introducing feature abstractions.

2.2. Shader Compilers

Efficient compilation of shading language code to GPU assembler is an active area of research and beyond the scope of this paper. We instead focus on producing reasonable high-level code from still higher-level abstractions. Nonetheless, we briefly review compiler work as it is the natural compilation target for our weaver.

McCool et. al's Shader Algebra [2004] extends their Sh language [2002] with **connect** and **combine** operations on primitives. These allow shaders to be optimized by a compiler and manipulated by a programmer without knowledge of the primitives. The connect operator requires the number, type, and storage classifier (and implicitly, the semantics) of arguments to agree. Therefore the output of the weaver provides ideal input for McCool et al.'s optimizing compilers.

Many hand-written shaders are short in part because it is impractical to write large shaders by hand in today's shading languages. By simplifying the process of creating complex shaders, abstract shade trees naturally raise the problem of creating shaders too large for resource-limited GPUs. The solution is to follow our tree compilation with a partitioning compiler. Chan et. al's [2002] compiler naturally fits within our framework—their system partitions trees into subtrees that execute in a single rendering pass.

The Brook language [Buck04] extends the C language so that it can be efficiently compiled for streaming processors like GPUs for non-graphics tasks. This is not directly related to our work; however, we note that the purely functional style of programming that is enforced by our system has been long noted to be ideal for compilation on parallel processors.

Like our work, Pellacini's [2005] recent shader simplification system manipulates the structure of shaders in semi-blind manner. Neither compiler is fully aware of the intent of the manipulated code and *could* introduce a transformation that destroys the underlying rendering effect. Yet in both cases one can perform useful work despite the potential pitfall. Our transformations go beyond single expressions and must synthesize the glue code between them. To reduce errors in this synthesis, we extend the type system with stronger semantics and require that the weaver preserve this semantic type safety.

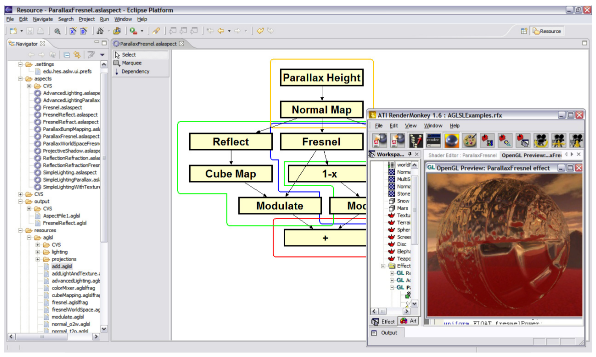
Parallax Height
Normal Map
Reflect
Fresnel
Cube Map
1-x
Modulate
Moc
+
ATI RenderMonkey 1.6 : AGLSLExamples.rfx
OpenGL Preview: ParallaxFresnel effect

Figure 2: *Creating an Abstract Shade Tree in our GUI.*

2.3. Feature-Based Programming

In the software engineering literature, the term **feature** refers to a user-identifiable attribute of a system, which a client might be willing to pay for [Turner99]. It is therefore natural to consider a rendering effect like shadows to be a feature. Thinking of programs as collections of features is not new: the idea is inherent in Parnas' [1972] seminal paper on modularity, and in Dijkstra's [1976] book on programming, where the latter discusses the "separation of concerns."

More recently, there has been significant activity on building programming languages—particularly module systems—that enable programmers to explicitly represent a system's features [Batory92, Kiczales97, Batory04]. In these languages, each module describes some feature, and module composition corresponds to building a system that consists of these features. (We adopt the term 'weaver' from one of these languages, known as aspect-oriented programming [Kiczales97].) Because client requirements tend to be in terms of features, these systems can more easily be reconfigured to accommodate evolving requirements. Indeed, it is now routine to talk about a **product line** of programs that can be built from a collection of features, by analogy to manufacturing. Software product lines have long been popular in the telecommunications industry, and are now increasingly popular in application software [Clements02]. Our tool offers a pre-created library of effects as well as the ability to create new ones from atoms, so a library of effects defines a product line of shaders.

3. System and Workflow Overview

Our system comprises a GUI tree-editor on the front-end (Figure 2) and a back-end that compiles trees to GLSL shaders. It leverages existing tools (e.g., ATI RenderMonkey) to provide real-time execution and preview of the shaders. We assume a library of hand-optimized primitives and pre-created effects is available.

To create a shader, an artist uses a drag-and-drop interface to place multiple existing effects in a common workspace. These effects appear as sub-graphs of named atoms connected by arrows. Each effect is surrounded by a colored boundary. The artist then interconnects the effects by adding additional arrows to form a single abstract shade tree. It is also possible for the artist to insert and remove individual atoms.

The tree is *abstract* because arrows represent a data dependency, not individual parameter mappings between atoms. As shown in Figure 1(right), when instances of a node common to two features are combined into a single node, the rendered feature boundaries correctly overlap.

Pressing the "preview" button executes the weaver, which follows the algorithm described in Section 5. This algorithm works backwards through the tree from the shader output (a pixel color) to the inputs, producing GLSL code. Its primary task is replacing each abstract arrow with pairs of input and output parameters for the atoms it connects. In many cases, those parameters do not naturally correspond and the weaver must inject substantial code to correct the problem. The output code resembles that produced when implementing a shader by hand, which shows that we have removed the tedium of shader production while preserving the creative aspects.

Our system's workflow consists of two asynchronous editing cycles. A programmer continually optimizes the atoms and introduces new atoms and sample effects into the system. Meanwhile, the artist edits abstract shade trees. Because the connections in the tree are abstract, the programmer may frequently change not only the implementation of atoms but also the API, i.e., the number and type of arguments, without requiring the artist to update the abstract shade tree.

4. Atom Definitions

Atoms are defined by a declaration, a set of struct/global function definitions, and a body. They are hand-coded and optimized in an extension to GLSL that includes atom declarations and semantic types. Atom declarations describe the number, name, and type of inputs and outputs of a block of code. They differ from traditional shading function declarations in two ways. First, there may be multiple output arguments. Second, no lexical scope is applied to the definition. Instead, free variables must be explicitly declared as global parameters. These globals also serve as hints to the weaver during parameter matching. The declaration syntax is a structured comment so atoms are backwards compatible with GLSL.

It is common practice in the games industry to squeeze every possible cycle from graphics routines. Programmers commonly examine the assembly produced by both shading and C++ compilers. To support this scrutiny, the weaver preserves whitespace, variable names, and documentation comments from atom bodies. This helps the programmer trace the effect of a code change on the abstract shade tree, the weaver's GLSL output, and the GLSL compiler's assembly output.

Many atoms, like the one in Listing 1, are simply GLSL standard library routines wrapped by a declaration. Often those standard library routines are intrinsics that map directly to a hardware feature.

```
//! START CubeMapping
//! @uniform environmentMap:TEX3D
//! @param cubeTexCoord:VEC3__W_VEC__
//! @param cubeMap:TEX3D = environmentMap
//! @return outColor:VEC3      RGB
outColor = textureCube(cubeMap, cubeTexCoord);
//! END CubeMapping
```

Listing 1: *Sample atom code for cube mapping.*

In the atom syntax[1], *param* declares an input parameter and *return* declares an output parameter. Atom declarations may also include two kinds of immutable **global parameters**. A *uniform* parameter is passed from the application to the entire shader. It is uniform over a series of rendering calls. A *varying* parameter is passed from the vertex shader to the fragment shader. It is interpolated between vertices by the hardware. Global parameters have two roles. In addition to declaring inputs passed outside the call chain, they may also appear as default values to satisfy a specific input parameter if the weaver is unable to find an appropriate output parameter from a connected node. For

[1] The actual atom syntax in our implementation is more verbose, containing documentation and other non-semantic fields in the comments.

example, in Listing 1, the *environmentMap* is not explicitly used by the atom body. However, it is declared as a global and listed as the default for match for the *cubeMap* input parameter. It will be used only if no other node producing a *TEX3D* is connected to the node with the *CubeMapping* atom.

We require all global names to be unique across the set of atoms. That is, if two atoms declare the *environmentMap* global parameter, it must have precisely identical semantic types in each. In this example, in every case where an environment map is provided as a parameter, it too must be named *environmentMap*. This is not an unreasonable requirement—after all, these are *global* variables. Since the shader APIs already dictate that globals must be synchronized with hand-written vertex shader and application code, it is not especially burdensome to require programmers to also synchronize globals between atoms. GLSL supports limited records called structs and global functions, which can be declared in the same manner as globals and have the same uniqueness constraint.

Semantic Types

We introduce **semantic types**, which are so specific that two variables with precisely the same type are likely semantically interchangeable. Some examples appear to the right of the colons in the annotations of Listing 1.

Regular GLSL types are merely C-style storage specifiers with little value as abstractions. For example, a color, a 3D location, and a row of a 3×3 matrix have the same type, which is also indistinguishable from an array of three floating-point numbers. Another extreme possibility is where types are so specific that each value has its own type: e.g, the integers '7' and '8' have separate types. This latter extreme, of each value being its own singleton type, is impractical. However, we find it advantageous to extend GLSL towards this extreme in order to encapsulate graphics concepts directly into the type system. Just as many languages assign different types to natural (unsigned) numbers and integers, we type vectors differently based on several mathematically meaningful properties. For example, in the case of vector *length*, we recognize two important values: unit and arbitrary. This allows the system to distinguish normalized vectors within the type system.

We use a convention where the name of a vector is the concatenation of a series of short codes for each semantic property. The properties and codes for vectors are:

Dimension: {2, 3, 4, _ }

Underbar is Wildcard

Length: {U: Unit, _ }

Basis: {T: tangent, O: object, W: world, S: screen space, _ }

Interpretation:
{RGB: color, TEX: texture coordinate,
NOR: surface normal (covector), VEC: direction,
PNT: point, _ }

Precision: {F: float32, I: int32, B: Boolean, _ }

The underbar is a wildcard for supporting polymorphic types, for example, *vec4__* is a four-component vector in any basis. These can also be viewed as type unions: e.g., "*vec4__* = *vec4_O* ∪ *vec4_W* ∪ *vec4_T* ∪ *vec4_S*." We created the whole list of properties based on distinctions we found meaningful and expect that more properties will be added in the future to help further distinguish semantics.

The type and naming scheme extends naturally to matrices, scalars, and textures. Semantic types can be made legal GLSL code by inserting a series of macros mapping them to storage classes, e.g., `#define VEC3_U_W_NOR_F vec3`.

A compiler typically uses a type system to validate programs. The weaver instead applies the type system as a set of rules for steering code generation creation—that is, generation is governed by the constraint that the output *must* be correctly typed. We define and use traditional typing rules on our semantic types, e.g.,

if v ***has type*** VEC3___T____
then (ObjectToTangentSpace * v)
    has type VEC3___O____

except that we apply these rules backward when seeking to coerce expression types. Thus the above rule would not be applied to type-check the product expression but instead to find a coercion of *v* from *VEC3___T____* to *VEC3___O____*. Section 5.4 describes how this coercion search occurs.

The *extremely* narrow application domain of shading languages is what makes this type system reasonable; these special-case typing rules and highly specified types probably cannot be generalized to other domains or general purpose languages.

5. Weaving Algorithm

We chose to implement the weaver as a pre-processor, without a full parser. This allows the weaver to preserve whitespace and comments within the atom bodies and allows atom bodies with partial statements, e.g.," *if (dot(N,V) > 0) {* ". Because the weaver doesn't parse the atom code, it can operate on a variety of shading language syntaxes (GLSL, HLSL, Cg), provided all atoms are implemented in the same language. This design decision also leads to a straightforward implementation in Java, which provides regular expressions and many other string manipulation routines.

We now detail the four steps of the weaving algorithm.

5.1. α-Rename Variables

Because the atoms are implemented individually, it is likely that some variable names are shared between them. In some cases this is because an output of one atom becomes the input of another and it really is the same variable. In other cases the same name is used for distinct variables that cannot be combined.

The weaver first creates a unique code body for each node. From this point forward there are no atoms bodies, only node bodies. Where two nodes use the same atom, two copies of that atom body are created. The weaver then assigns each node in the tree a unique ID. It iterates over all input and output parameter declarations (but not global declarations) of all nodes, seeking variable name conflicts where the same name is used in two different node bodies. Once all conflicts have been detected, the weaver renames all variables within node bodies that conflict by appending the unique node ID to the original name, e.g. *surfaceNormal* → *surfaceNormal_0001*. We include the original variable name to preserve readability of the output. This process is a common semantics preserving transformation called α-renaming.

Employing α-renaming is overly conservative because it destroys parameter linkage between nodes. However, this is not a problem because subsequent weaver steps ensure correct linkage, independent of parameter names.

Renaming only affects parameters that appear in the atom declaration. We avoid atom-local variable name conflicts by the convention of wrapping atom bodies with a local scope "*{...}*".

5.2. Topologically Sort Nodes

The weaver creates a new terminal node accepting a single input for the pixel color and a directed edge into this node from any node with no output (there is typically only one such node).

In the abstract tree DAG, edges represent the data dependencies between atoms. Without destroying the tree structure, the weaver assigns a topological ordering to the graph nodes based on these dependencies. The new node appears last in the topological ordering.

5.3. Match Inputs to Outputs

We now come to the core of the algorithm. The weaver begins with the terminal node at the bottom and works up the shade tree in reverse topological order to the inputs at the top.

For each node, the weaver matches each input parameter to an output parameter from a parent node. Two parameters match only if both have precisely the same semantic type. Our semantic types are specific enough that there is rarely a perfect match. The weaver therefore seeks an output and a **coercion** that will transform the output type to the input type. The coercion search proceeds as follows. Consider the implicit **coercion tree** in Figure 3 where the root is the type of the input parameter for which a corresponding output is being sought (note that this is unrelated to the shade tree). The leaves are the types of the available outputs from nodes higher up the abstract shade tree. The edges are coercions (i.e., typing rules run backwards) and the internal nodes are the types of intermediate expressions produced during a series of coercion operations. The tree is infinite because of cycles: one may reach world space from tangent space by the two-step coercion *tangent → object → world*, but also by *any* coercion of the form *tangent→object→tangent→object→ …→ world*.

Of course, we never want to apply such a complicated coercion path when a better alternative exists. Our notion of 'better' includes both the length of a coercion path and the time cost of traversing each edge. The cost of each coercion edge is based on the anticipated cycle count for executing that operation at run-time. For example, transforming a pre-normalized world-space light vector to object space by a matrix multiplication may be faster than normalizing an existing but non-unit object-space light vector. The children of a coercion tree node are arranged from left to right according to the increasing cost of each rule. The tree is explored on the fly and never fully constructed.

The search for a viable coercion begins at the root of the coercion tree and proceeds downwards breadth-first, left-to-right. It terminates when the first type node is encountered that matches one of the available output types, or when depth seven is reached. Any value around seven is a reasonable cutoff; the key idea is to allow enough coercions for the anticipated worst case, which is from an arbitrary tangent-space 3-vector to a normalized, swizzled, screen-space 4-vector. Because we have ordered the rules based on cost, the first condition indicates that we have found the best coercion to some output. Regardless of total performance cost, we consider a short coercion path better than a long one because it is likely semantically closer and therefore probably what the user intended.

The second termination condition indicates that there is likely no meaningful coercion available from an output parameter, possibly because we have reached a root of the abstract shade tree. In this case, the weaver then searches the original atom declaration for a default global to link that parameter against. If that search fails to find a match, the weaver introduces a new global uniform parameter of the matching type. The matching process is guaranteed to succeed.

When a match has been made for an input, any needed coercions are inserted back into the shade tree. Edges of the coercion tree become new nodes in the (now slightly less abstract) shade tree. The weaver then proceeds to the next input variable. Because the shade tree has been modified, any intermediate coercion product becomes available to match future inputs, as does any newly introduced uniform. This is necessary to avoid creating redundant coercions and globals. When all inputs of one node have been matched, the weaver proceeds to the next-higher node in the topological ordering.

5.4 Concatenate Node Bodies

To form the shader code, the weaver concatenates all global parameter declarations, struct declarations, and the node bodies in topological order wrapped by "*void main(void) {…}*".

Since we preserved variable names, the uniform parameters will have meaningful names. This makes it possible to map them to GUI elements in IDEs such as RenderMonkey or FX Composer for interactive adjustment.

Finally, the weaver inserts a series of *#define* macros that map all semantic types used in the shader to legal GLSL storage classes. The output shader can be run from any OpenGL program or shader preview tool.

6. GUI Implementation

Our GUI tree editor, shown in Figure 2 with a 3D scene in the RenderMonkey, is implemented as a plug-in to the Eclipse IDE. This allows programmers to easily move between atom editing in a traditional code editor and experimentation with those atoms in the abstract shade tree editor. Eclipse provides automatic layout and rendering of graphs, simplifying the implementation.

We render the feature outlines to off-screen bitmaps and then composite them over the tree. Each feature outline is rendered with a variation on [Raskar99] as follows: Clear the off-screen bitmap to transparent. For each node in the feature (note that a node may belong to multiple features, like *Fresnel* in Figure 1), render a colored, solid, rounded rectangle *f* pixels larger than the node itself, where *f* is a unique small integer for each feature. The varying radii keep adjacent features outlines from overlapping. Likewise, render a thickened line segment for each arrow between two nodes in the feature. Finally, clear the interior by rendering the same shape with an *f* – 2 radius and a transparent fill color, and composite the resulting outline over the graph.

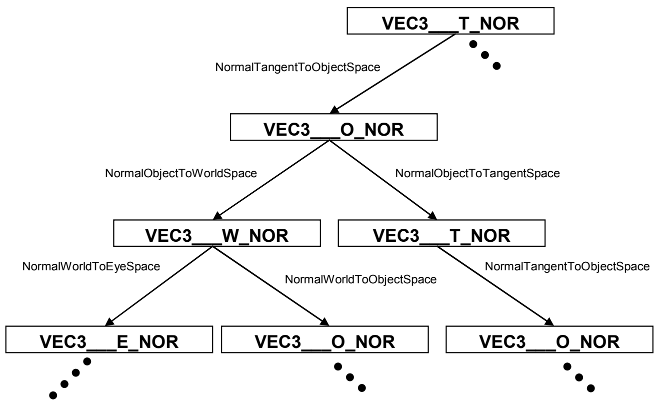

Figure 3: *Sample coercion tree mapping the possible paths through the type rules for coercing a tangent space normal to an eye-space normal. Many possible paths are not explored because the all-left branch leads to a successful coercion.*

7. Results

Figures 4-7 show abstract shade trees created with our system. Each figure displays the actual tree as it appears in the authoring system (left) and the GLSL shader produced by the weaver for that tree (right), superimposed over an image of an object rendered with that shader. The GLSL output in the result figures is color-coded. Light lines of code correspond to atoms that have been inlined. Dim lines correspond to parameter linkage, coercions, and type macros inferred by the weaver.

Figure 4 is the bumpy glass shader we considered in Figure 1. The full code is given in the Appendix to give a sense of the weaver's output. In allowing non-programmers to effectively create shaders, we have not diminished the importance of programmers on a team but instead focused their role. Note that most of the output code in the Appendix is necessary but uninteresting because it is boilerplate and linkage between atoms. This is also true in the other examples. Because the weaver assumes the duty of generating the necessary "glue" code, programmers concentrate on creating and optimizing atom bodies, which is the interesting part of their role that requires graphics, programming, and mathematical expertise.

The underlying features/effects described in the captions are clearly visible in the tree diagrams. Even a non-programmer can see the interaction between features and manipulate them easily. In Figures 4 and 7 the features share central nodes where they overlap. To create these, the user dropped the separate features into the workspace, which created duplicate nodes. The user then explicitly combined those common nodes.

7.1. Performance

The result figures demonstrate that shader creation is easy in our system, that the weaver can produce correct GLSL code, and that the abstract shade tree is both more compact and easier to understand than a traditional tree or code. The generated shaders are efficient. All examples run at hundreds of frames per second on a laptop with a Radeon 9700 Mobile GPU. The weaver itself is efficient; each example took less than one second to generate.

To compare the performance of the generated code to hand written code, we hand-wrote an optimized GLSL shader for the effects in the bumpy glass shader. The manual implementation contained only 58 lines of GLSL code compared to the weaver's 188 lines, which are shown in the appendix. However, the weaver generates a lot of comment and variable name linkage overhead.

When both shaders are compiled to hardware assembly with NVIDIA's Cg compiler, the weaver's implementation contains 51 instructions and the manual implementation contains 46 instructions. Shading every pixel at 512×512, the weaver's implementation achieves 240 fps and the manual implementation achieves 245 fps. At 1024×768, both render at 50 fps.

7.2. Limitations

Our system always produces a legal, type-safe program. However, there are three ways that program can still fail to meet expectations. The first is that it allows creation of shaders that exceed the instruction and register count limits of today's hardware. See Chan et. al [2002] for a multi-pass solution.

Second, the weaver can produce less efficient code than a programmer in cases where a whole-program optimization is appropriate, *e.g.*, moving all lighting from world space to object space to avoid repeated per-pixel transformations. To perform such an optimization, the compiler would have to both understand

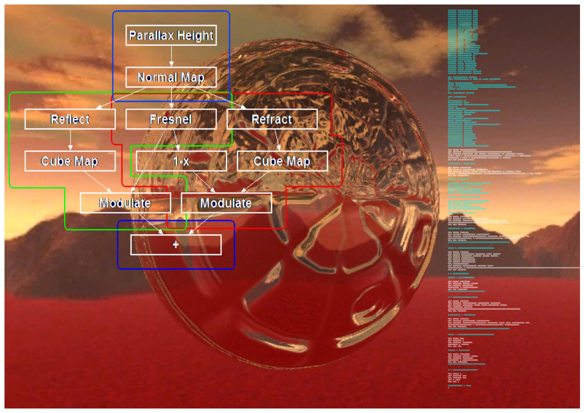

Figure 4: *Bumpy glass.*

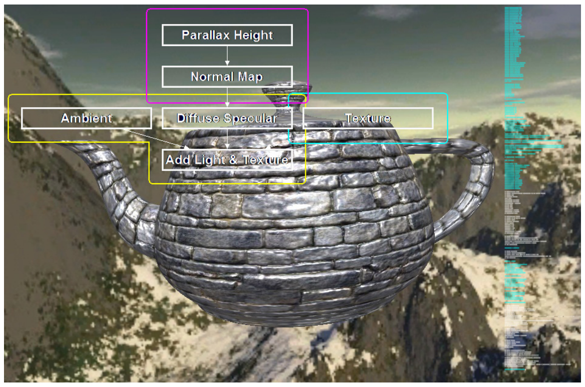

Figure 5: *Parallax mapping, texture mapping, and Phong illumination on a teapot.*

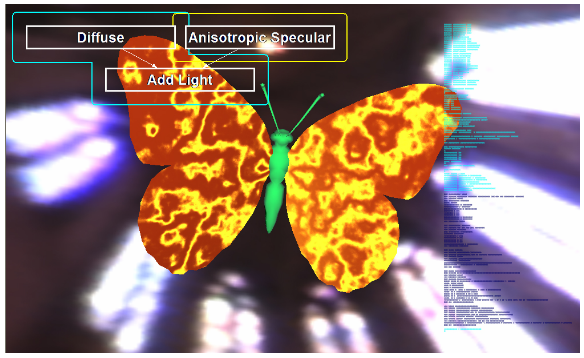

Figure 6: *Anisotropic specular reflection with isotropic diffuse reflection on the wings of a butterfly.*

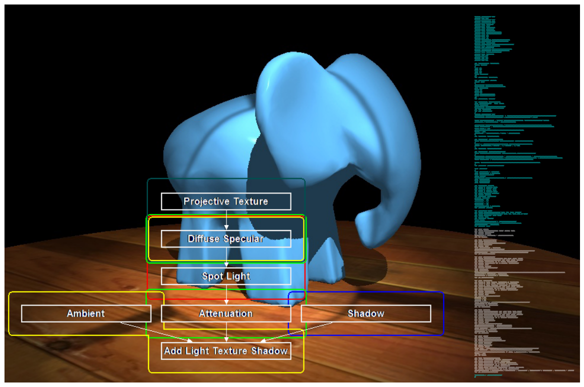

Figure 7: *Projective light, shadow map, and Phong illumination. Note the interlacing of features.*

spatial transformations at not just a semantic but an operational level, and have control over not only the shading algorithm but also the host C++ program into which it is integrated. This is interesting future work but significantly beyond the scope of our semantic type approach.

The third case, semantic errors, is the most interesting. We invited laypeople (non-programmers, non-artists) to experiment with effect creation. After we explained the UI and primitives they were generally able to produce shaders, which we consider a great success compared to current tools. However, they had difficulty choosing between similar primitives with different types, which often led to inefficient semantic type coercions for the desired effects. When the tool failed to produce the effect that the user expected, it was usually because two nodes received input from the same output when they should have been distinct (e.g., imagine both *Modulate* nodes linked to input *x* in Figure 1). This occurs when the user fails to add sufficient dependency arrows and when the dependency is implicit in a global variable. Like the excessive transformations, missing dependencies are a user error that would likely not occur with real artists; a follow-up user study will measure the experiences of trained artists.

Incorrect linkage of two primitives to the same global variable only occurs when the common semantic type of the primitives is too general. Here, fault lies with the programmer who created the primitives and not the artist or the weaver. The programmer must, however, tread a delicate line: types that are *too* specific will never match exactly and require more coercions, but types that are too general produce incorrect semantics. On one hand, it is a drawback of our system that programmers must expend much effort tailoring the interface types. On the other hand, we argue that interface semantics are exactly where programmers should think hard! We automate the linkage, boilerplate, and coercion precisely so that programmers can focus on design, which requires human intelligence.

8. Other Future Work

Our cost ranking of coercions is based on instruction count and intuition. A natural step is to use an actual cycle count. The challenge here is that the true time cost of a GPU operation depends on the instructions surrounding it, cache state, and the instruction scheduler.

On the programming language side, we envision an extension to parameterized types like C++ templates, which would enable more specific function types than our polymorphism. For example, we assign the addition operator the type *VEC___* × *VEC___* → *VEC___*, yet *VEC_<T>* × *VEC_<T>* → *VEC_<T>* is more specific.

A formal semantics, type system, and proofs for our system will serve as a good case study for the literature on domain-specific languages and feature-based programming.

9. Conclusions

We addressed the problem originally noted by Abram and Whitted by making type mismatches in shaders impossible. We also enable shader creation by users who do not even have knowledge of types, programming concepts like variables, or the vector math used to implement algorithms inside the atoms.

We presented a new system for authoring complex GPU programs through automatic combination of primitive shading functions. In doing so, we extended GLSL with semantics types specific to computer graphics in a backwards-compatible manner. We anticipate that GLSL (like assembly language and C before it) will increasingly be produced as the output of a higher order tool. We will propose to the OpenGL architectural review board that semantic types be considered for the language standard, for use by both programmers and other tools like ours.

Our system uses many heuristics to infer parameter linkage. Even with our strong semantic types, it is a natural concern that the heuristics can produce a legal program with semantic errors. Fortunately, in graphics a semantic error is easily diagnosed because the image produced is incorrect. The interative editing context of our Eclipse plug-in and RenderMonkey allows the shader author to correct the shade tree until the desired result is achieved. We speculate that our methodology is appropriate for similar domains like image- and audio-filter design, but inappropriate for general purpose programming that lacks easy feedback and a small set of semantic types.

Our system extends previous work, allowing non-programmers to more easily create more complex shaders. It also visualizes shaders with easy-to-understand block diagrams. The advantage of abstract shade trees over hand-coded shaders or the one-to-one visual editors will only increase as hardware becomes ever more capable and the desired shaders increase commensurately in complexity. It literally took only seconds to create each of the abstract shade trees for our result figures; implementing similar shaders by hand in GLSL took us hours of coding and debugging for each shader. These shader examples contain about four effects each. Now consider GPUs of the future that are able to render ten or twenty interacting effects in real-time. Under today's workflow, an artist might ask a programmer to hand code a different effect combination for every object in a scene. That will not be feasible when each shader contains thousands of lines of code. Our tool addresses the authoring problem by making it possible for artists to create these shaders themselves and by making the process easy and enjoyable.

This work is supported in part by the National Science Foundation.

10. References

ABRAM G. D., WHITTED T. Building block shaders. Computer Graphics, 24:4, 1990, pp 283—288

BATORY D., Feature-Oriented Programming and the AHEAD Tool Suite, ICSE, 2004

BORAU R., DOMIK G., GOETZ F. An XML-based visual shading language for vertex and fragment shaders, 3D technologies for the World Wide Web, in Proc. 3D Web Tech., 2004, pp 87—97

BATORY D. AND O'MALLEY D., The Design and Implementation of Hierarchical Software Systems with Reusable Components, TOSEM, 1992, 1:4, pp 355—398

BUCK I., FOLEY T., HORN D., SUGERMAN J., FATAHALIAN K., HOUSTON M., AND HANRAHAN P., Brook for GPUs: Stream Computing on Graphics Hardware, ACM Trans. on Graph., 23:3, 2004, pp 777—784

CLEMENTS P. AND NORTHROP L., Software Product Lines: Practices and Patterns, Addison-Wesley, 2002

COOK R. L. Shade Trees. *In Proc*. Computer Graphics and Interactive Techniques, July 1984, pp 223—231

CHAN E., NG R., SEN, P., PROUDFOOT K., HANRAHAN, P. Efficient Partitioning of Fragment Shaders for Multipass Rendering on Programmable Graphics Hardware, Graphics Hardware, 2002, pp 69—78

DIJKSTRA E. W., A Discipline of Programming, Prentice-Hall, 1976

KICZALES G., LAMPING J., MENDHEKAR A., MAEDA C., LOPES C. V., LOINGTIER J., AND IRWIN J., Aspect-Oriented Programming, ECOOP, June 1997

MCCOOL M. D., QIN Z., AND POPA T. S. Shader Metaprogramming. Graphics Hardware, 2002. pp 57—68

MCCOOL M. D., DU TOIT S., POPA T., CHAN B., MOULE K., Shader Algebra. ACM Trans. on Grap., 23:3, 2004, pp. 787—795

PARNAS D. L., On the Criteria To Be Used in Decomposing Systems Into Modules, CACM, vol 15, no 12, Dec 1972, pp. 1053—1058

PELLACINI F., User-configurable automatic shader simplification, ACM Trans. Graph., 24:3, 2005, pp 445—452

RASKAR R., COHEN M., Image Precision Silhouette Edges, I3D, 1999, pp 135—140

TURNER C. R., FUGGETTA A., LAVAZZA L., WOLF A. L., A Conceptual Basis for Feature Engineering, Journal of Systems and Software, vol 49, no 1, Dec 1999, pp 3—15

Visual Material Editor, Unreal Engine 3 Whitepaper, Epic Corporation, 2006
http://www.unrealtechnology.com/html/technology/ue30.shtml

Appendix

GLSL code produced by the weaver from the abstract shade tree in Figure 1. **Boxed lines** are hand written node bodies. Gray lines are comments carried through from atom declarations. The remaining lines are parameter linkage and coercions generated by the weaver.

```
#define VEC3___T_NOR vec3
#define VEC3___W_NOR vec3
#define VEC4___W_NOR vec4
#define VEC4_____NOR vec4
#define VEC3_____NOR vec3
#define VEC3_____TEX vec3
#define VEC4_____RGB vec4
#define VEC4________ vec4
#define VEC2_____TEX vec2
#define VEC3___T____ vec3
#define TEX2D sampler2D
#define MAT4 mat4
#define VEC3___O____ vec3
#define VEC3___O_NOR vec3
#define VEC3___W____ vec3
#define TEX4D samplerCube
#define FLOAT float
uniform FLOAT etaRatio;
uniform FLOAT fresPower;
uniform FLOAT fresScale;
uniform FLOAT fresBias;
uniform TEX4D evntCubeMap;
varying VEC3___W____ vIncoming_w;
varying VEC3___O_NOR nor;
varying VEC3___O____ bin;
varying VEC3___O____ tan;
uniform MAT4 matNorO2W;
uniform TEX2D normalMap;
uniform FLOAT bumpScale;
uniform FLOAT bumpBias;
uniform TEX2D heightTex;
varying VEC3___T____ vVew_t;
varying VEC2_____TEX texCoo;

//! STARTROUTINE refract
vec3 refract(vec3 I, vec3 N, float etaRatio) {
float cosI=dot(-I,N);
float cosT2=1.0-etaRatio*etaRatio*(1.0-cosI*cosI);
vec3 T = etaRatio*I+((etaRatio*
  cosI-sqrt(abs(cosT2)))*N);
return T*vec3(cosT2>0.0);
}
//! ENDROUTINE refract

void main(void) {
VEC4________ a;
VEC4________ b;
VEC4________ sum;
VEC4________ modOutput_1010696501;
FLOAT x;
FLOAT oneMinus;
VEC4_____RGB colorFromEvt_314286916;
VEC3_____TEX cooRefr_w;
VEC4________ vTmp4;
FLOAT factor;
VEC4________ modOutput_1345567180;
FLOAT fresnelRatio;
VEC3_____TEX cubetexCoo;
VEC4_____RGB colorFromEvt_444161459;
VEC3_____NOR nor3Tmp;
VEC3___O_NOR nor3_o;
VEC4_____NOR nor4Tmp;
VEC4___W_NOR nor4_w;
VEC3___W_NOR vNormal_w;
VEC3_____TEX cooRefl_w;
VEC2_____TEX bumpCoords;
VEC3___T_NOR vNor_t;
VEC2_____TEX offsetCoo;
VEC4________ modOutput;
VEC4_____RGB colorFromEvt;

//! START ParallaxHeight
//! @params texCoo,vVew_t,heightTex,bumpBias,bumpScale
//! @return offsetCoo parallax cords
FLOAT bump=((texture2D(heightTex,texCoo).a)+ bumpBias)*bumpScale;
offsetCoo = bump * vec2(vVew_t.x, vVew_t.y) + texCoo;
//! END ParallaxHeight
bumpCoords = offsetCoo;

//! START NormalMap
//! @params bumpCoords, normalMap
//! @return vNor_t tangent space normal
vNor_t = texture2D(normalMap,vec2(bumpCoords.x,
        bumpCoords.y)).xyz* 2.0-1.0;
//! END NormalMap

nor3Tmp = vNor_t;

//! START NormalTangentToObjectSpace
//! @params tan, bin, nor, nor3Tmp
//! @return nor3_o
mat3 matNorT2O = mat3(tan, bin, nor);
nor3_o = matNorT2O*nor3Tmp;
//! END NormalTangentToObjectSpace
nor4Tmp = vec4(nor3_o,0.0);

//! START NormalObjectToWorldSpace
//! @params matNorO2W, nor4Tmp
//! @return nor4_w
nor4_w = matNorO2W*nor4Tmp;
//! END NormalObjectToWorldSpace
vNormal_w = nor4_w.xyz;

//! START Reflect
//! @params vIncoming_w, vNormal_w
//! @return cooRefl_w World space reflected vector
cooRefl_w=normalize(reflect(vIncoming_w,vNormal_w));
//! END Reflect

cubetexCoo = cooRefl_w;

//! START CubeMap
//! @params cubetexCoo, evntCubeMap
//! @return colorFromEvt color from environment map
colorFromEvt = textureCube(evntCubeMap, cubetexCoo);
//! END CubeMap

colorFromEvt_444161459=colorFromEvt;
vTmp4 = colorFromEvt_444161459;

//! START Fresnel
//! @params vIncoming_w,vNormal_w,fresBias,
//! @params fresScale,fresPower
//! @multi-aspect
//! @return fresnelRatio Fresnel ratio
fresnelRatio = max(0.0,min(1.0,fresBias+fresScale*
           pow(1.0+ dot(normalize(vIncoming_w),
           normalize(vNormal_w)), resPower)));
//! END Fresnel

x = fresnelRatio;
factor = fresnelRatio;

//! START Modulate
//! @params vTmp4, factor
//! @return modOutput
modOutput = vTmp4*factor;
//! END Modulate
modOutput_1345567180=modOutput;

a = modOutput_1345567180;

//! START Refract
//! @params vIncoming_w, vNormal_w, etaRatio
//! @return cooRefr_w World space refracted vector
//! @routine refract
cooRefr_w = normalize(refract(vIncoming_w, vNormal_w, etaRatio));
//! END Refract

cubetexCoo = cooRefr_w;

//! START CubeMap
//! @params cubetexCoo, evntCubeMap
//! @return colorFromEvt color from environment map
colorFromEvt = textureCube(evntCubeMap, cubetexCoo);
//! END CubeMap
colorFromEvt_314286916=colorFromEvt;
vTmp4 = colorFromEvt_314286916;

//! START 1-x
//! @params x
//! @return oneMinus
oneMinus = 1.0-x;
//! END 1-x
factor = oneMinus;

//! START Modulate
//! @params vTmp4, factor
//! @return modOutput
modOutput = vTmp4*factor;
//! END Modulate
modOutput_1010696501=modOutput;
b = modOutput_1010696501;

//! START +
//! @params a,b
//! @return sum
sum = a+b;
//! END +

gl_FragColor = sum;
 }
```

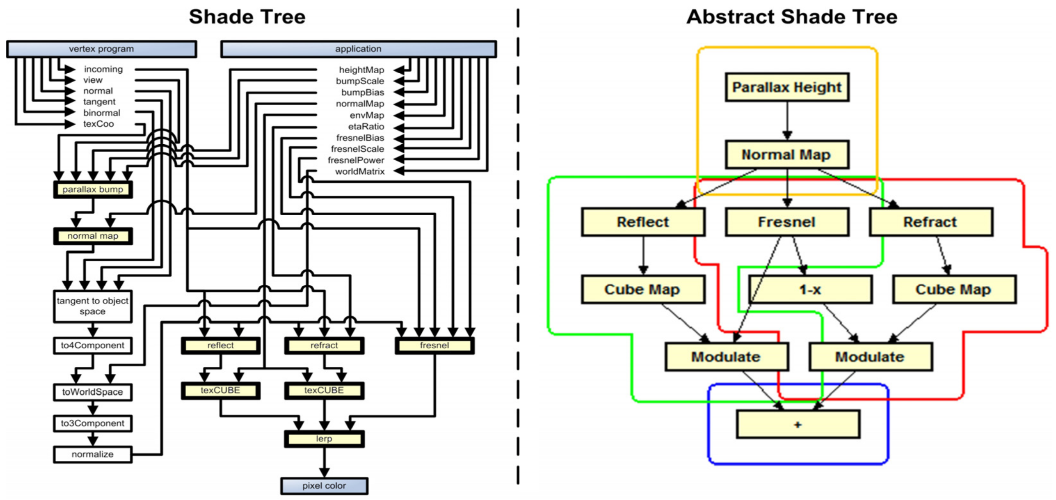

Figure 1 Revisited: *A conventional Shade Tree (left) for a "Bumpy Glass" shader, mocked up in the Visio drawing program. The equivalent Abstract Shade Tree on the right is an actual screenshot from our shader authoring plugin to the Eclipse IDE.*

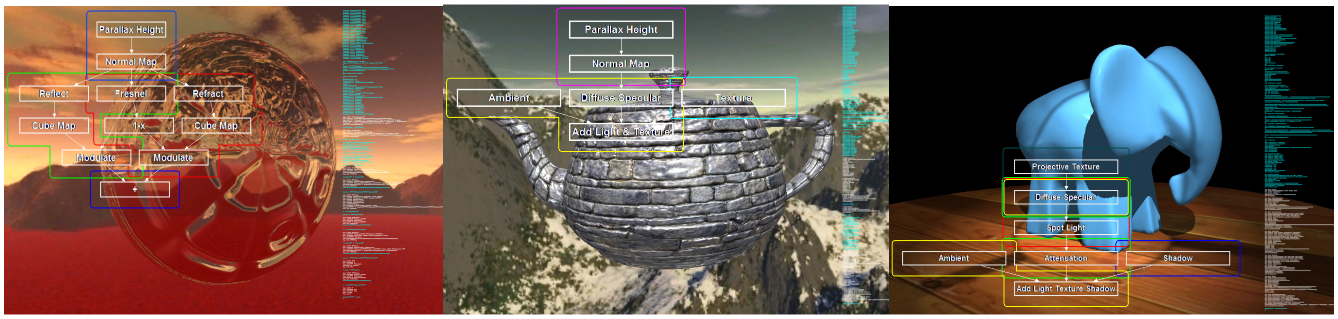

Figure 8. *Shade trees and the equivalent code composited over the effects they produce. (Color reproductions of Figures 4-7).*