Programming Graphics Processors Functionally

Conal Elliott*

Abstract

Graphics cards for personal computers have recently undergone a radical transformation from fixed-function graphics pipelines to multi-processor, programmable architectures. Multi-processor architectures are clearly advantageous for graphics for the simple reason that graphics computations are naturally concurrent, mapping well to stateless stream processing. They therefore parallelize easily and need no random access to memory with its problematic latencies.

This paper presents *Vertigo*, a purely functional, Haskell-embedded language for 3D graphics and an optimizing compiler that generates graphics processor code. The language integrates procedural surface modeling, shading, and texture generation, and the compiler exploits the unusual processor architecture. The shading sublanguage is based on a simple and precise semantic model, in contrast to previous shading languages. Geometry and textures are also defined via a very simple denotational semantics. The formal semantics yields not only programs that are easy to understand and reason about, but also very efficient implementation, thanks to a compiler based on partial evaluation and symbolic optimization, much in the style of Pan [2].

Haskell's overloading facility is extremely useful throughout Vertigo. For instance, math operators are used not just for floating point numbers, but also expressions (for differentiation and compilation), tuples, and functions. Typically, these overloadings cascade, as in the case of surfaces, which may be combined via math operators, though they are really functions over tuples of expressions on floating point numbers. Shaders may be composed with the same notational convenience. Functional dependencies are exploited for vector spaces, cross products, and derivatives.

*The work reported in this paper was done while the author was at Microsoft Research.

Haskell'04, September 22, 2004, Snowbird, Utah, USA.

Categories and Subject Descriptors

D.1.1 [**Programming Techniques**]: Applicative (Functional) Programming; D.1.3 [**Programming Techniques**]: Concurrent Programming; D.3.4 [**Programming Languages**]: Processors—*code generation*, *compilers*; I.3.3 [**Computer Graphics**]: Picture/Image Generation; I.3.5 [**Computer Graphics**]: Computational Geometry and Object Modeling; I.3.6 [**Computer Graphics**]: Methodology and Techniques—*Graphics data structures and data types*,*Languages*; I.3.7 [**Computer Graphics**]: Three-Dimensional Graphics and Realism

General Terms

Algorithms, Performance, Design, Languages

Keywords

Computer graphics, graphics processors, compilers, code generation, partial evaluation, computer algebra, domain-specific languages, functional programming, functional geometry, 3D modeling, graphics languages, shading languages, procedural geometry, procedural shading

1 Introduction

There has recently been a revolution in processor architecture for personal computers. High-performance, multi-processor, data-streaming computers are now found on consumer-level graphics cards. The performance of these cards is growing at a much faster rate than CPUs, at roughly Moore's law cubed [4]. Soon the computational power of these graphics processing units ("GPUs") will surpass that of the system CPU.

Some common applications of GPUs include geometric transformation, traditional and alternative lighting and shading models ("programmable shaders"), and procedural geometry, textures, and animation.

The accepted programming interfaces are assembler and C-like "shading languages", having roots in RenderMan's shading language [5, 14, 3, 10]. This is an unfortunate choice, because the computations performed are naturally functional. In fact, these C-like languages are only superfically imperative. This paper offers a functional alternative to existing shading languages that simplifies and generalizes them without sacrificing performance.

GPU architectures are naturally functional as well. The low-level

execution model is programs acting in parallel over input streams producing new output streams with no dependence between stream members, i.e., pure functions mapped over lists. Pipelining is used between the different processor types (vertex and pixel processors in the current architectures), much like compositions of *lazy* stream functions.

The main contributions reported in this paper are as follows:

- Optimized compilation of a functional language to modern graphics hardware.
- A simple and practical embedding of parametric surfaces definition and composition (generative modeling [12]) in a functional programming language. (See also [6].)
- A simple but powerful semantic model for shading languages, with direct implementation of that model.

2 Why Functional Graphics?

Functional programming is a natural fit for computer graphics simply because most of objects of interest *are* functions.

- Parametric surfaces are functions of type $\mathcal{R}^2 \to \mathcal{R}^3$, to be evaluated over a subregion of $\mathcal{R}^2$.
- Implicit surfaces and spatial regions are functions of type $\mathcal{R}^3 \to \mathcal{R}$ where surface, inside and outside are distinguished by the sign of the resulting real value. Planar regions are functions of type $\mathcal{R}^2 \to \mathcal{R}$.
- Height fields, as used to represent a class of geometry as well as bump mapping and displacement mapping, are functions of type $\mathcal{R}^2 \to \mathcal{R}$.
- Spatial transformations (e.g., affines and deformations) are functions of type $\mathcal{R}^3 \to \mathcal{R}^3$ for 3D or $\mathcal{R}^2 \to \mathcal{R}^2$ for 2D.
- Resolution-independent images are functions of type $\mathcal{R}^2 \to Color$.
- 2D & 3D animations and time-varying values of all types are functions from $\mathcal{R}$.
- Lights of all kinds are functions from points in $\mathcal{R}^3$ to the direction and color of the light delivered to that point.
- Shaders are functions from view information (ambient color, eye point and set of active lights) and surface point information (color, location and surface derivatives).

Computer graphics math makes extensive use linear algebra, and in particular matrices for representing linear, affine, or projective spatial transformations. There are actually competing conventions for transforming vectors with matrices using matrix multiplication. In one, the matrix is on the left and the vector is a column, while in the other, the vector is a row and the matrix is on the right. Transformations are composed by multiplying the matrices, taking care with the order, consistently with the pre-multiply or post-multiply convention. With a functional foundation, one can simply let the transformations be functions that happen to be linear, affine or projective, or might be arbitrary spatial deformations, such as bends, twists, or tapers.

3 Graphics processors

Vertigo targets the DirectX 8.1 vertex shader model shown in Figure 1, which is taken from [9]. This model and a multiprocessor implementation are described in [8]. This unit is replicated, typically with four or eight instances. Every register is a quadruple of 32-bit floating point numbers (a "quad-float"). Every "vertex" is represented by up to 16 registers, having user-specified semantics, e.g., coordinates of a 3D point, its normal vector, one or more sets of texture coordinates, etc. Vertex and constant registers are read-only, and the output registers are write-only. Temporary registers may be written and read during a vertex computation but are cleared before each new vertex. That property is important, because it means that (a) several vertex processors may run in parallel, and (b) vertex processing is simply mapping of a pure function over a vertex stream.

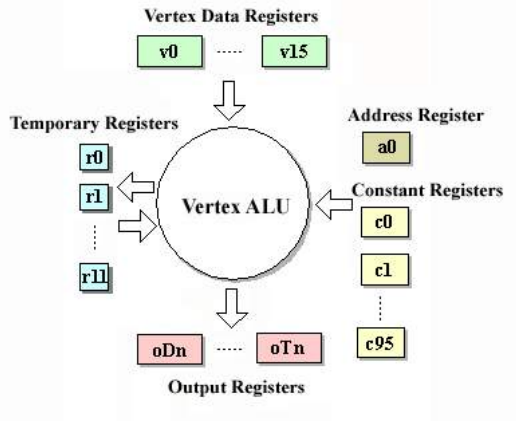

Figure 1. Vertex shader model

The input vertex stream is parceled out to the vertex processors, and the resulting output is reassembled and fed to the pool of pixel processors, which are not discussed in this article.

An important aspect of this model is that random memory access is *extremely* limited (to these registers). Large amounts of vertex data are accessed by streaming from video RAM rather than being accessed randomly system.

One reason GPUs and functional programming fit together is that GPUs inherently compute staged functions. Vertex computations depend on "constant" registers and on vertex registers. Values held in the constant registers may be set at most once per stream of vertices, being held constant among vertices in a stream. Typically these constant registers contain both actual constants and time-varying values. Thus any vertex computation may be cast as a curried function:

$$vc :: MeshData \to (VertexData \to Vout)$$

Given such a computation *vc*, mesh data *md*, and a stream *svd* of vertex data, the vertex processor hardware simply computes

$$map\ (vc\ md)\ svd$$

4 Geometry

3D graphics cards mainly render vertex meshes, with each containing information such as 3D location, normal vector, and texture coordinate vertices. The new breed of graphics processors, being programmable, are very flexible in the type of streams they can operate on and what computations they can perform. Vertigo concentrates on synthetic (or "procedural") geometry, from which vertex meshes

are extracted automatically and efficiently. The main type of interest is a (parametric) surface, which is simply a mapping from $\mathcal{R}^2$ to $\mathcal{R}^3$.

$$\textbf{type } \mathit{Surf} = \mathcal{R}^2 \to \mathcal{R}^3$$

$$\textbf{type } \mathcal{R}^2 = (\mathcal{R}, \mathcal{R})$$
$$\textbf{type } \mathcal{R}^3 = (\mathcal{R}, \mathcal{R}, \mathcal{R})$$

By convention, during display, surfaces will be sampled over the 2D interval $[-1/2, 1/2] \times [-1/2, 1/2]$.

At this point, the reader may safely interpret $\mathcal{R}$ as synonymous with *Float*. The actual meaning of $\mathcal{R}$ is *expressions* over *Float*, so that the implementation can perform optimizing compilation (Section 6) and symbolic differentiation (Section 8).

Now one can start defining surfaces directly. For instance, here are a unit sphere and a cylinder with a given height and unit radius.

$$\begin{aligned}
&\mathit{sphere} :: \mathit{Surf}\\
&\mathit{sphere}\ (u, v) = (\mathit{cos}\ \theta \cdot \mathit{sin}\ \phi, \mathit{sin}\ \theta \cdot \mathit{sin}\ \phi, \mathit{cos}\ \phi)\\
&\quad \textbf{where } \theta = 2 \cdot \pi \cdot u\\
&\qquad\quad\ \ \phi = \pi \cdot v
\end{aligned}$$

$$\begin{aligned}
&\mathit{cylinder} :: \mathcal{R} \to \mathit{Surf}\\
&\mathit{cylinder}\ h\ (u, v) = (\mathit{cos}\ \theta, \mathit{sin}\ \theta, h \cdot v)\\
&\quad \textbf{where } \theta = 2 \cdot \pi \cdot u
\end{aligned}$$

Note that as u and v vary between $-1/2$ and $1/2$, θ varies between $-\pi$ and π, while ϕ varies between $-\pi/2$ and $\pi/2$ (south and north poles).

More powerfully, using higher-order functions, we can construct surfaces compositionally, as in the method of generative modeling [12, 11]. The next several examples introduce and demonstrate a collection of useful combinators for surface composition.

4.1 Height fields

"Height fields" are simply functions from $\mathcal{R}^2$ to $\mathcal{R}$, and may be visualized in 3D in the usual way:

$$\textbf{type } \mathit{HeightField} = \mathcal{R}^2 \to \mathcal{R}$$

$$\begin{aligned}
&\mathit{hfSurf} :: \mathit{HeightField} \to \mathit{Surf}\\
&\mathit{hfSurf}\ \mathit{field}\ (u, v) = (u, v, \mathit{field}\ (u, v))
\end{aligned}$$

A simple definition produces ripples:

$$\begin{aligned}
&\mathit{ripple} :: \mathit{HeightField}\\
&\mathit{ripple} = \mathit{sinU} \circ \mathit{magnitude}
\end{aligned}$$

Here *sinU* is a convenient variant of the *sin* function, normalized to have unit period. (The typeset code examples in this paper use an infix "$\cdot$" operator for regular multiplication and for scalar/vector multiplation introduced below.)

$$\begin{aligned}
&\mathit{cosU}, \mathit{sinU} :: \mathcal{R} \to \mathcal{R}\\
&\mathit{cosU}\ \theta = \mathit{cos}\ (2 \cdot \pi \cdot \theta)\\
&\mathit{sinU}\ \theta\ = \mathit{sin}\ (2 \cdot \pi \cdot \theta)
\end{aligned}$$

Now let's add the ability to alter the frequency and magnitude of the ripples. This ability is useful in many examples, so abstract it out:

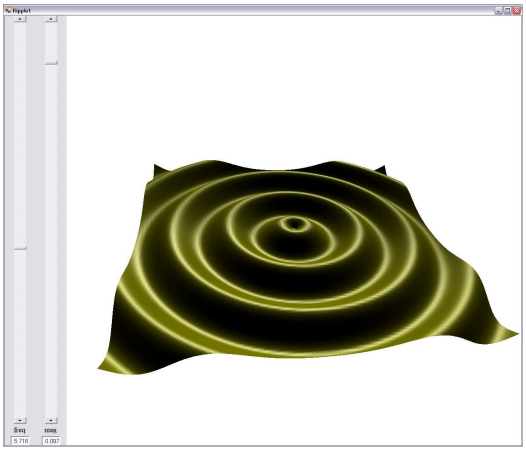

Figure 2. rippleS 5.7 0.1

$$\begin{aligned}
&\mathit{freqMag} :: \mathit{Surf} \to (\mathcal{R}, \mathcal{R}) \to \mathit{Surf}\\
&\mathit{freqMag}\ f\ (\mathit{freq}, \mathit{mag}) = (\mathit{mag}\cdot) \circ f \circ (\mathit{freq}\cdot)
\end{aligned}$$

Combining, we get the surface shown in Figure 2.[1]

$$\begin{aligned}
&\mathit{rippleS} :: \mathcal{R}^2 \to \mathit{Surf}\\
&\mathit{rippleS} = \mathit{hfSurf} \circ \mathit{freqMag}\ \mathit{ripple}
\end{aligned}$$

The definition of *freqMag* uses operators to scale the incoming $\mathcal{R}^2$ and outgoing $\mathcal{R}^3$ points. These operators belong to the vector space type class defined as follows, for a scalar type s and a vector space v over s. (The actual operator for scalar multiplication is "*^".)

$$\begin{aligned}
&\textbf{class } \mathit{Floating}\ s \Rightarrow \mathit{VectorOf}\ s\ v \mid v \to s \textbf{ where}\\
&\quad (\cdot) \qquad :: s \to v \to v\\
&\quad (<\cdot>) :: v \to v \to s \qquad -\!-\ \text{dot product}
\end{aligned}$$

The general type of *freqMag* then is as follows.

$$\begin{aligned}
&\mathit{freqMag} :: (\mathit{VectorOf}\ \mathit{si}\ \mathit{vi}, \mathit{VectorOf}\ \mathit{so}\ \mathit{vo})\\
&\qquad \Rightarrow (\mathit{vi} \to \mathit{vo}) \to (\mathit{si}, \mathit{so}) \to (\mathit{vi} \to \mathit{vo})
\end{aligned}$$

The constraints here say that the types *vi* and *vo* are vector spaces over the scalar field *si* and *so*, respectively.

As another surface example, here is a wavy "eggcrate" height field:

$$\begin{aligned}
&\mathit{eggcrate} :: \mathit{HeightField}\\
&\mathit{eggcrate}\ (u, v) = \mathit{cosU}\ u \cdot \mathit{sinU}\ v
\end{aligned}$$

The definition of $\mathit{eggcrate}\ (u, v)$ above fits a pattern: the result comes from sampling one function at u and another at v and combining the results. Since this pattern arises in other examples, we abstract it out.

$$\begin{aligned}
&\mathit{eggcrate} = \mathit{cartF}\ (\cdot)\ \mathit{cosU}\ \mathit{sinU}\\
&\\
&\mathit{cartF}\ \ :: \ (a \to b \to c) \to (u \to a) \to (v \to b)\\
&\qquad \to (u, v) \to c\\
&\mathit{cartF}\ \mathit{op}\ f\ g\ (u, v) = f\ u\ `\mathit{op}`\ g\ v
\end{aligned}$$

[1] The GUIs shown in this paper are automatically generated based on the type of a parameterized surface and a small specification of the labels and ranges for parameter sliders.

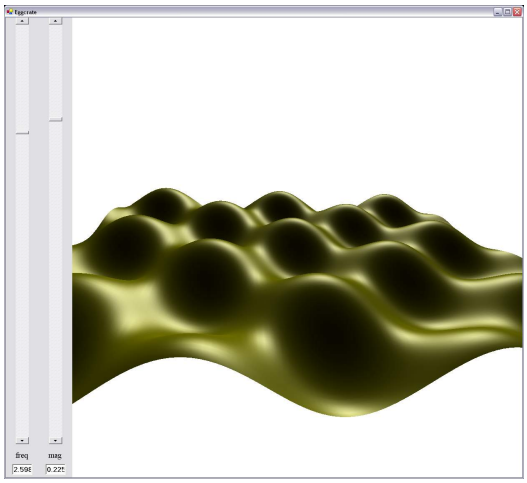

Figure 3. *eggcrateS* 2.6 0.23

Now add control for frequency and magnitude of the waves, to get the surface shown in Figure 3.

$$\begin{aligned}&\mathit{eggcrateS} :: \mathcal{R}^2 \to \mathit{Surf}\\&\mathit{eggcrateS} = \mathit{hfSurf} \circ \mathit{freqMag}\ \mathit{eggcrate}\end{aligned}$$

4.2 Sweeps

Another surface composition technique is using one curve to "sweep" another.

$$\begin{aligned}&\textbf{type}\ \mathit{Curve}_2 = \mathcal{R} \to \mathcal{R}^2\\&\textbf{type}\ \mathit{Curve}_3 = \mathcal{R} \to \mathcal{R}^3\end{aligned}$$

$$\begin{aligned}&\mathit{sweep} :: \mathit{Curve}_3 \to \mathit{Curve}_3 \to \mathit{Surf}\\&\mathit{sweep}\ \mathit{basis}\ \mathit{scurve}\ (u,v) = \mathit{basis}\ u + \mathit{scurve}\ v\end{aligned}$$

Or more succinctly,

$$\mathit{sweep} = \mathit{cartF}\ (+)$$

For instance, a cylinder is a circle swept by a line.

$$\mathit{cylinder}\ h = \mathit{sweep}\ (\mathit{addZ}\ \mathit{circle})\ (\mathit{addXY}\ (h\cdot))$$

The helper functions *addXY* and *addZ* simply increase the dimensionality of a value in $\mathcal{R}$ or $\mathcal{R}^2$ respectively, inserting zeros. For convenience, they actually apply to functions that produce $\mathcal{R}$ or $\mathcal{R}^2$.

$$\begin{aligned}&\mathit{addX}, \mathit{addY}, \mathit{addZ} :: (a \to \mathcal{R}^2) \to (a \to \mathcal{R}^3)\\&\mathit{addX} = \mathit{lift}_1\ (\lambda(y,z) \to (0,y,z))\\&\mathit{addY} = \mathit{lift}_1\ (\lambda(x,z) \to (x,0,z))\\&\mathit{addZ} = \mathit{lift}_1\ (\lambda(x,y) \to (x,y,0))\end{aligned}$$

$$\begin{aligned}&\mathit{addYZ}, \mathit{addXZ}, \mathit{addXY} :: (a \to \mathcal{R}) \to (a \to \mathcal{R}^3)\\&\mathit{addYZ} = \mathit{lift}_1\ (\lambda x \to (x,0,0))\\&\mathit{addXZ} = \mathit{lift}_1\ (\lambda y \to (0,y,0))\\&\mathit{addXY} = \mathit{lift}_1\ (\lambda z \to (0,0,z))\end{aligned}$$

The handy "lifting" functionals are defined as follows:

$$\begin{aligned}\mathit{lift}_1\ h\ f_1\ x &= h\ (f_1\ x)\\ \mathit{lift}_2\ h\ f_1\ f_2\ x &= h\ (f_1\ x)\ (f_2\ x)\\ \mathit{lift}_3\ h\ f_1\ f_2\ f_3\ x &= h\ (f_1\ x)\ (f_2\ x)\ (f_3\ x)\\ &\ldots\end{aligned}$$

We can define the *circle* curve out of lower-dimensional functional pieces as well:[2]

$$\begin{aligned}&\mathit{circle} :: \mathit{Curve}_2\\&\mathit{circle} = \mathit{cosU}\ `\mathit{pairF}`\ \mathit{sinU}\end{aligned}$$

$$\begin{aligned}&\mathit{pairF} :: (c \to a) \to (c \to b) \to (c \to (a,b))\\&\mathit{pairF} = \mathit{lift}_2\ (,)\end{aligned}$$

4.3 Surfaces of revolution

Another commonly useful building block is revolution of a curve. To define revolution, simply lift the curve into $\mathcal{R}^3$ by adding a zero Z coordinate, and then rotate around the Y axis.

$$\begin{aligned}&\mathit{revolve} :: \mathit{Curve}_2 \to \mathit{Surf}\\&\mathit{revolve}\ \mathit{curve}\ (u,v) = \mathit{rotY}\ (2\cdot\pi\cdot u)\ (\mathit{addZ}\ \mathit{curve}\ v)\end{aligned}$$

The function *rotY* is an example of a 3D spatial "transform". Traditionally in computer graphics, transforms are restricted to linear, affine, or projective mappings and are represented by matrices. In a functional setting, they may more simply and more generally be functions:

$$\begin{aligned}&\textbf{type}\ \mathit{Transform}_1 = \mathcal{R} \to \mathcal{R}\\&\textbf{type}\ \mathit{Transform}_2 = \mathcal{R}^2 \to \mathcal{R}^2\\&\textbf{type}\ \mathit{Transform}_3 = \mathcal{R}^3 \to \mathcal{R}^3\end{aligned}$$

To rotate a 3D point about the Y axis, it suffices to rotate (x,z) in 2D and hold y constant:

$$\begin{aligned}&\mathit{rotY} :: \mathcal{R} \to \mathit{Transform}_3\\&\mathit{rotY}\ \theta = \mathit{onXZ}\ (\mathit{rotate}\ \theta)\end{aligned}$$

$$\begin{aligned}&\mathit{rotate} :: \mathcal{R} \to \mathit{Transform}_2\\&\mathit{rotate}\ \theta\ (x,y) = (x\cdot c - y\cdot s, y\cdot c + x\cdot s)\\&\quad\textbf{where}\ c = \mathit{cos}\ \theta\\&\qquad\quad s = \mathit{sin}\ \theta\end{aligned}$$

$$\begin{aligned}&\mathit{onXY}, \mathit{onYZ}, \mathit{onXZ} :: \mathit{Transform}_2 \to \mathit{Transform}_3\\&\mathit{onXY}\ f\ (x,y,z) = (x',y',z)\\&\quad\textbf{where}\ (x',y') = f\ (x,y)\\&\mathit{onXZ}\ f\ (x,y,z) = (x',y,z')\\&\quad\textbf{where}\ (x',z') = f\ (x,z)\\&\mathit{onYZ}\ f\ (x,y,z) = (x,y',z')\\&\quad\textbf{where}\ (y',z') = f\ (y,z)\end{aligned}$$

Spheres and cylinders are surfaces of revolution:

$$\begin{aligned}&\mathit{sphere} = \mathit{revolve}\ \mathit{semiCircle}\\&\mathit{cylinder}\ h = \mathit{onZ}\ (h\cdot) \circ \mathit{revolve}\ (\lambda y \to (1,y))\end{aligned}$$

A semi-circle is just a circle sampled over half of its usual domain ($[-1/4, 1/4]$ instead of $[-1/2, 1/2]$):

$$\mathit{semiCircle} = \mathit{circle} \circ (/2)$$

[2] Building higher-dimensional shapes out of lower ones is one of the themes of generative modeling [12, 11].

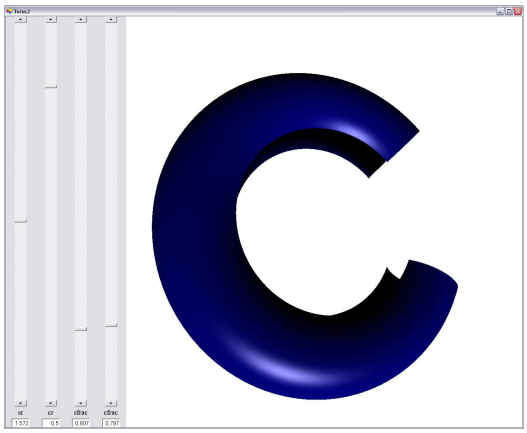

Figure 4. *torusFrac* 1.5 0.5 0.8 0.8

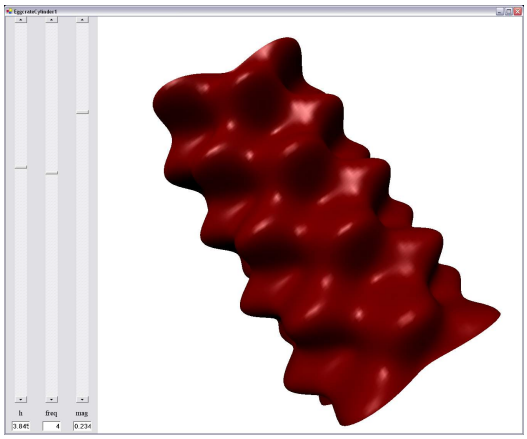

Figure 5. *eggcrateCylinder* 3.8 4.0 0.23

The torus is a more interesting example. It is the revolution of a scaled and offset circle.

$$\begin{aligned}&\mathit{torus} :: \mathcal{R} \to \mathcal{R} \to \mathit{Surf}\\&\mathit{torus}\ \mathit{sr}\ \mathit{cr} = \mathit{revolve}\ (\mathit{const}\ (\mathit{sr}, 0) + \mathit{const}\ \mathit{cr} \cdot \mathit{circle})\end{aligned}$$

Note that the addition and multiplication here are working directly on 2D curves, thanks to arithmetic overloading on functions and on tuples.

$$\begin{aligned}&\textbf{instance}\ \mathit{Num}\ b \Rightarrow \mathit{Num}\ (a \to b)\ \textbf{where}\\&\quad (+) \quad\quad\ = \mathit{lift}_2\ (+)\\&\quad (\cdot) \quad\quad\ = \mathit{lift}_2\ (\cdot)\\&\quad \mathit{negate} \quad = \mathit{lift}_1\ \mathit{negate}\\&\quad \mathit{fromInteger} = \mathit{const} \circ \mathit{fromInteger}\\&\qquad \text{—}\ \text{etc.}\end{aligned}$$

To make the example more interesting, add parameters to scale down the surface parameters *u* and *v*. The result is an incomplete torus, as in Figure 4.

$$\begin{aligned}&\mathit{torusFrac}\ \mathit{sr}\ \mathit{cr}\ \mathit{cfrac}\ \mathit{sfrac} =\\&\quad \mathit{torus}\ \mathit{sr}\ \mathit{cr} \circ (\cdot(\mathit{cfrac}, \mathit{sfrac}))\end{aligned}$$

4.4 Displacement surfaces

As a final example of surface construction, Figure 5 results from "displacing" a cylinder using the eggcrate height field.

$$\begin{aligned}&\mathit{eggcrateCylinder}\ h\ \mathit{fm} =\\&\quad \mathit{displace}\ (\mathit{cylinder}\ h)\ (\mathit{freqMag}\ \mathit{eggcrate}\ \mathit{fm})\end{aligned}$$

The definition of displacement is direct:

$$\begin{aligned}&\mathit{displace} :: \mathit{Surf} \to \mathit{HeightField} \to \mathit{Surf}\\&\mathit{displace}\ \mathit{surf}\ \mathit{field} = \mathit{surf} + \mathit{field} \cdot \mathit{normal}\ \mathit{surf}\end{aligned}$$

Note that the surface, its normal, and the height field are all sampled at the same point in $\mathcal{R}^2$. The displacement vector gets its direction from the surface normal and its distance from the height field.

Normals are computed by taking the cross products of the partial derivatives.

$$\begin{aligned}&\mathit{normal} :: \mathit{Surf} \to \mathit{Surf}\\&\mathit{normal} = \mathit{normalize} \circ \mathit{cross} \circ \mathit{derivative}\end{aligned}$$

As described in Section 8, Vertigo computes derivatives exactly, not through numeric approximation.

Vector normalization scales to unit length, and is defined independently of any particular vector space.

$$\begin{aligned}&\mathit{normalize} :: \mathit{VectorOf}\ s\ v \Rightarrow v \to v\\&\mathit{normalize}\ v = v / \mathit{magnitude}\ v\end{aligned}$$

$$\begin{aligned}&\mathit{magnitude} :: \mathit{VectorOf}\ s\ v \Rightarrow v \to s\\&\mathit{magnitude}\ v = \mathit{sqrt}\ (v{<}\cdot{>}v)\end{aligned}$$

The type of *normal* is actually more general:

$$\begin{aligned}&\mathit{normal} :: (\mathit{Derivative}\ c\ \mathit{vec}\ \mathit{vecs}\\&\qquad\quad , \mathit{Cross}\ \mathit{vecs}\ \mathit{vec}\\&\qquad\quad , \mathit{VectorOf}\ s\ \mathit{vec})\\&\qquad \Rightarrow (c \to \mathit{vec}) \to (c \to \mathit{vec})\end{aligned}$$

The constraints mean that (a) the derivative of a $c \to \mathit{vec}$ function has type $c \to \mathit{vecs}$, (b) the cross product of a *vecs* value has type *vec*, and (c) the type *vec* is a vector space over the scalar field *s*. In the *Surf* case, $s = \mathcal{R}$, $c = \mathcal{R}^2$, $\mathit{vec} = \mathcal{R}^3$, and $\mathit{vecs} = (\mathcal{R}^3, \mathcal{R}^3)$.

The inferred type of *displace* is also more general than given above.

$$\begin{aligned}&\mathit{displace} :: (\mathit{Num}\ (c \to \mathit{vec})\\&\qquad\quad , \mathit{Cross}\ \mathit{vecs}\ \mathit{vec}\\&\qquad\quad , \mathit{Derivative}\ c\ \mathit{vec}\ \mathit{vecs}\\&\qquad\quad , \mathit{VectorOf}\ s\ \mathit{vec}\\&\qquad\quad , \mathit{VectorOf}\ (c \to s)\ (c \to \mathit{vec}))\\&\qquad \Rightarrow (c \to \mathit{vec}) \to (c \to s) \to (c \to \mathit{vec})\end{aligned}$$

For instance, the cross product of a single 2D vector (x, y) is the 2D vector $(y, -x)$, and the *displace* function may be used to displace one 2D curve with a "2D height field" (of type $\mathcal{R} \to \mathcal{R}$). In this case, $s = \mathcal{R}$, $c = \mathcal{R}$, $\mathit{vec} = \mathcal{R}^2$, and $\mathit{vecs} = \mathcal{R}^2$.

5 Shading

Shading languages began with Cook's "shade trees", which were expression trees used to represent shading calculations. The most successful shading language has been RenderMan's [5, 14].

One interesting aspect of RenderMan's shading language is that the data it uses comes in at different frequencies (surfaces patches, points on surfaces, and light sources) . As an example, here is a def-

inition of a diffusely reflecting surface [14, page 335] (simplified).

```
surface
matte(float Ka, Kd)
{
    Ci = Cs * (Ka*ambient() + Kd*diffuse(N));
}
```

In explanations of this shading language, invocations of a parameterized shader like `matte` are referred to as "instances", and the parameters like `Ka` and `Kd` are referred to as "instance variables". A given instance instance is "called" perhaps thousands or millions of times for different sample points on a surface. These "calls" to a shader instance supply information specific to surface points, such as surface normal (`N`) and surface color (`Cs`). "It may be useful to think of a shader instance as an object bundling the functionality of the shading procedure with values for the instance variables used by the procedure" [14, Chapter 16]. Shader calls read from and write to special global variables.

There is a third frequency of evaluation as well, namely the contribution of several light sources per surface point. Here is a definition of a diffuse lighting function, commonly used in shader definitions [14, Chapter 16].

```
color
diffuse(point norm)
{
    color C = 0;
    unitnorm = normalize(norm);
    illuminance( P, unitnorm, PI/2 )
            C += Cl * normalize(L).unitnorm;
    return C;
}
```

The `illuminance` construct iterates over light sources, combining the effects of its body statement, using light-source-specific values for light color (`Cl`) and direction (`L`).

5.1 The essence of shading languages

To create a semantic basis for shaders, consider the information that a shader has access to and what it can produce. Some information comes from the viewing environment, some comes from a point on the surface, and some from a light source relative to that point.

A viewing environment consists of an ambient light color, an 3D eye position, and a collection of light sources:

$$\textbf{type } \mathit{ViewEnv} = (\mathit{Color}, \mathcal{R}^3, [\mathit{Light}])$$

Information about a surface at a point includes the point's position, a pair of partial derivatives (each tangent to the surface at that point), and an intrinsic color:

$$\textbf{type } \mathit{SurfPt} = (\mathcal{R}^3, (\mathcal{R}^3, \mathcal{R}^3), \mathit{Color})$$

For our purposes, a light source is something that provides light information to every point in space (though to some points it provides blackness), independent of obstructions.[3]

$$\textbf{type } \mathit{Light} = \mathcal{R}^3 \to \mathit{LightInfo}$$

Light information delivered to a point consists simply of color and direction. Any given shader will decide what to do with this information. Attenuation and relation of light position (if finitely distant) to surface position are already accounted for.

$$\textbf{type } \mathit{LightInfo} = (\mathit{Color}, N_3)$$

For example, here are definitions for simple directional and point lights (without distance-based attenuation):

$$\begin{aligned}&\mathit{dirLight} :: \mathit{Color} \to N_3 \to \mathit{Light}\\ &\mathit{dirLight\ col\ dir} = \mathit{const}\ (\mathit{col}, \mathit{dir})\end{aligned}$$

$$\begin{aligned}&\mathit{pointLight} :: \mathit{Color} \to \mathcal{R}^3 \to \mathit{Light}\\ &\mathit{pointLight\ col\ lightPos\ p} =\\ &\quad (\mathit{col}, \mathit{normalize}\ (\mathit{lightPos} - p))\end{aligned}$$

There are three different kinds of shaders, corresponding to the three stages of information used in the shading process. "View shaders" depend only on viewing environment; "surface shaders" depend additionally on surface point info; and "light shaders" depend additionally on a single light info. View shaders are not particularly useful, but are included for completeness.

Rather than restricting to a single resulting value type like *Color*, it will be useful to generalize to arbitrary result types:[4]

$$\begin{aligned}&\textbf{type } \mathit{VShader}\ a = \mathit{ViewEnv} \to a\\ &\textbf{type } \mathit{SShader}\ a = \mathit{VShader}\ (\mathit{SurfPt} \to a)\\ &\textbf{type } \mathit{LShader}\ a = \mathit{SShader}\ (\mathit{LightInfo} \to a)\end{aligned}$$

5.2 A "shading language"

Given the model above, one could simply start writing shaders as functions. Doing so leads to awkward-looking code, however, due to the explicit passing around and extraction of view, surface point, and light information. This explicit passing is not necessary in the RenderMan shading language thanks to the use of global variables. Fortunately, we can keep our function-based semantic model and remove the notational clutter. The trick is to build shaders using higher-order building blocks, and define overloadings.[5]

First define extractors that access information from the view environment:

$$\begin{aligned}&\mathit{ca} :: \mathit{VShader\ Color}\ ;\ \mathit{ca}\ (c, \_, \_) = c\\ &\mathit{eye} :: \mathit{VShader}\ \mathcal{N}_3\ ;\ \mathit{eye}\ (\_, e, \_) = e\\ &\mathit{lights} :: \mathit{VShader}\ [\mathit{Light}];\ \mathit{lights}\ (\_, \_, l) = l\end{aligned}$$

Similarly for surface point info:

$$\begin{aligned}&\mathit{pobj} :: \mathit{SShader}\ \mathcal{R}^3\ ;\ \mathit{pobj}\ \_\ (p, \_, \_) = p\\ &\mathit{dp} :: \mathit{SShader}\ (\mathcal{R}^3, \mathcal{R}^3);\ \mathit{dp}\ \_\ (\_, d, \_) = d\\ &\mathit{cs} :: \mathit{SShader\ Color}\ ;\ \mathit{cs}\ \_\ (\_, \_, c) = c\end{aligned}$$

Using the full derivative (Jacobian matrix) *dp*, we can easily define the two partial derivatives by selection and surface normal vector

[3] In a more sophisticated model, a light source would probably also take into consideration atmosphere and solid obstructions.

[4] In the Renderman shading language, shaders do not have return values at all, but rather assign to globals, and shaders are not allowed to call other shaders. There are also "functions", which return values and can be called by shaders and other functions.

[5] As discussed in Section 5.3, one could instead use implicit parameters.

by cross product.

```
dpdu, dpdv :: SShader R^3
dpdu e s = fst (dp e s)
dpdv e s = snd (dp e s)

n :: SShader N_3
n = normalize (cross dp)
```

Light shaders need extractors as well:

```
cl :: LShader Color ; cl _ _ (c, _) = c
l  :: LShader Dir3E; l  _ _ (_, d) = d
```

It is easy to precisely define a counterpart to RenderMan's `illuminance` construct. To turn a light shader into a surface shader, simply iterate over the light sources in the viewing environment, apply to the surface point to get the required light information, and sum the results.[6]

```
illuminance :: Num a ⇒ LShader a → SShader a
illuminance lshader v@(_, _, ls) s@(p, _, _) =
  sum [lshader v s (light p) | light ← ls]
```

Sometimes we need to mix light and surface shaders, which we do by lifting a surface shader into a light shader. For instance, the dot product between normal vector and light direction is commonly used in shaders.

```
ndotL :: LShader R
ndotL = toLS n<·>l
```

The dot product here is on functions.

The *toLS* function simply adds an ignored argument:

```
toLS ss v s _ = ss v s
```

This function is actually overloaded to work on view shaders and non-shaders as well, adding one or two ignored arguments, respectively. Similarly, there are overloaded *toES* and *toSS* functions.

5.3 Implicit parameters

We also implemented the shading language using implicit parameters [7]. The following definitions describe dependencies on view, surface point, and light information, abstracting out the details:

```
type ViewDep a =
  (?ca :: Color, ?eye :: R^3, ?lights :: [Light]) ⇒ a
type SurfDep a =
  (?cs :: Color, ?pobj :: R^3, ?d :: (R^3, R^3)) ⇒ a
type LightDep a = (?cl :: Color, ?l :: R^3) ⇒ a

type VShader a = ViewDep a
type SShader a = VShader (SurfDep a)
type LShader a = SShader (LightDep a)
```

This formulation eliminates the need for *toLS* and the *lift_i* functions used in the explicit function formulation. It is, however, rather demanding of the type system. The original implementations of implicit parameters in GHC did not support type definitions like *ViewDep*, *SurfDep*, and *LightDep*, requiring instead that all of the implicit parameters be mentioned explicitly at every use. For example, instead of the simple types for *n* and *ndotL* above, we would have something like the following.

```
n :: (?d :: (R^3, R^3)) ⇒ N_3
n = normalize (cross?d)

ndotL :: (?d :: (R^3, R^3), ?l :: R^3) ⇒ R^3
ndotL = n<·>?l
```

Note how these *implementations* of *n* and *ndotL* show through in their types. It gets worse from there: as more and more pieces of the view, surface point, and light contexts are used, the explicit lists of implicit parameters grow. Fortunately, GHC's type checker was improved to handle definitions like *ViewDep* and the others, so we were able to hide all of the implicit parameters. The actual definitions look like the following.

```
dp :: SShader (R^3, R^3)
dp = ?dp

n :: SShader R^3
n = normalize (cross dp)

ndotL :: LShader R^3
ndotL = n<·>l
```

The improvements made to GHC for supporting such convenient definitions are not present in Hugs, which we also wanted to use, so for now, Vertigo has both the explicit and implicit parameter approaches. Since the latter is more convenient, we will use it for the examples in the next section.

5.4 Sample shading specifications

Given this simple shading language, we can define some common shaders. The simplest (other than pure ambient or pure intrinsic) is pure diffuse. It uses $n<\cdot>l$ to scale the light color, and sums over all light directions *l*.

```
diffuse :: SShader Color
diffuse = illuminance (ndotL · cl)
```

We then make a weighted combination of pure ambient (*ca*) and diffuse:

```
ambDiff :: R^2 → SShader Color
ambDiff (ka, kd) = cs · (ka · ca + kd · diffuse)
```

To make surfaces look shiny, we turn to specular shading, which is independent of intrinsic color.

```
specular :: R → SShader Color
specular sh = illuminance ((vdotR**sh) · cl)

vdotR :: LShader R
vdotR = eyeDir<·>reflect l n

eyeDir :: SShader N_3
eyeDir = normalize (eye − pobj)
```

[6] A more sophisticated renderer might use a different set of light sources, synthesized from the environment's lights, simulate area light sources and inter-object reflection and occlusion.

The pictures in Section 4 are made using a weighted combination of ambient, diffuse, and specular shading.

$$\begin{aligned}&\mathit{basic} :: \mathcal{R}^4 \to \mathit{Shader\ Color}\\&\mathit{basic}\ (\mathit{ka}, \mathit{kd}, \mathit{ks}, \mathit{sh}) =\\&\quad \mathit{ambDiff}\ (\mathit{ka}, \mathit{kd}) + \mathit{ks} \cdot \mathit{specular\ sh}\end{aligned}$$

Many other shaders may be defined, e.g., brushed metal.

6 The GPU compiler

Vertigo is implemented as an optimizing compiler, in the style of Pan [2]. The main difference is that Vertigo targets a modern graphics processor architecture, rather than a general purpose CPU instruction set.

The target GPU architecture and instruction set have some unusual traits that make it challenging and interesting to compile into correct and efficient code.

- Most operations work on quad-floats.
- Operand registers may be negated and/or "swizzled" for free. Swizzling is extraction and rearrangement of scalar components to form a new vector, possibly omitting or replicating components. The same component may be used more than once to form an operand.
- There are no literals in the assembly code. All literals must be loaded into constant registers (also quad-floats).
- At most one constant register and one vertex register can be accessed per instruction.
- There is no conditional instruction.
- There is a multiply-add instruction ($a \cdot b + c$).
- There are no trig functions, so they must be approximated.

6.1 Front end

The front end of the Vertigo compiler is similar to that of Pan [2], with the following main differences:

- The data types supported are 1- to 4-tuples of 32 bit floats.
- The primitive operations are altered to target GPUs.
- Many of the algebraic rewrites use associative-commutative matching.

The programming interface is a set of statically typed definitions that make calls to a layer of dynamically typed "smart constructors", as in Pan [2]. The type $\mathcal{R}$ used above refers to statically typed, float-valued expressions.

The smart constructors perform bottom-up algebraic simplifications and build expressions, which may be literals, variables, applications of primitive operators, or let-bindings:

$$\begin{aligned}&\textbf{data}\ \mathit{Exp} = \mathit{LitVec\ Vector}\\&\qquad\quad \mid \mathit{Var\ Id\ Type}\\&\qquad\quad \mid \mathit{Apply\ Op}\ [\mathit{Exp}]\\&\qquad\quad \mid \mathit{Let}\ [(\mathit{Id}, \mathit{Exp})]\ \mathit{Exp}\end{aligned}$$

$$\textbf{type}\ \mathit{Vector} = [\mathit{Float}]$$

$$\textbf{type}\ \mathit{Id} = \mathit{String} \qquad \text{— variable name}$$

The set of primitive operators reflect the GPU instruction set:

$$\begin{aligned}&\textbf{data}\ \mathit{Op} = \mathit{Add} \mid \mathit{Mul} \mid \mathit{Mad} \mid \mathit{Max} \mid \mathit{Min} \mid \mathit{Sge} \mid \mathit{Slt}\\&\qquad\quad \mid \mathit{Mov}\\&\qquad\quad \mid \mathit{Rcp} \mid \mathit{Rsq} \mid \mathit{Log} \mid \mathit{Exp}\\&\qquad\quad \mid \mathit{Dp3} \mid \mathit{Dp4}\\&\qquad\quad \mid \mathit{Expp} \mid \mathit{Logp} \mid \mathit{Frc}\\&\qquad\quad \mid \mathit{Negate} \mid \mathit{Swizzle}\ [\mathit{Int}] \mid \mathit{MkVec}\\&\qquad\quad \mid \mathit{Frac}\\&\qquad\quad \mid \mathit{Cos} \mid \mathit{Sin}\end{aligned}$$

Notes:

- The first line (add, multiply, multiply-add, max, min, $\geq$, and $<$) contains SIMD operations: The last two return a vector containing floats that represent booleans, using zero for false and one for true. All are binary except *Mad*, which is ternary ($a \cdot b + c$).
- *Mov* is the unary identity operator.
- The third line ($1/x$, $1/\sqrt{x}$, $log_2 x$, and 2^x) contains operations that work only on scalar values (presumably because SIMD execution would use too much time or silicon).
- The fourth line contains 3D and 4D dot product operations, computing scalar results.
- Negation and swizzling are pseudo-operations. They are integrated into each generated instruction but are logically separate at this level. Vector construction is also a pseudo-op.
- The *Sin* and *Cos* operators are introduced but replaced later by approximations. The main reason is to allow computation of derivatives before approximation rather than after, resulting in a more precise approximation of the derivative.

6.2 Smart constructors

The smart constructors invoked by the statically typed interface differ from those in Pan because of the target architecture.

For instance, as the only comparators are $\geq$ and $<$, other boolean operators must be synthesized. For clarity, we state the translations in concrete syntax, though the actual implementation does pattern matching on the *Exp*.

$$\begin{aligned}e_1 = e_2 &= e_1 \geq e_2 \wedge e_2 \geq e_1\\e_1 \neq e_2 &= e_1 < e_2 \vee e_2 < e_1\end{aligned}$$

$$\begin{aligned}a > b &= b < a\\a \leq b &= b \geq a\end{aligned}$$

$$\begin{aligned}\mathit{not}\ (e_1 < e_2) &= e_1 \geq e_2\\\mathit{not}\ (e_1 \geq e_2) &= e_1 < e_2\end{aligned}$$

Although the statically typed layer has a *Bool* type, the GPU architecture simulates booleans via floating point numbers, using 1.0 for *True* and 0.0 for *False*. Thus,

$$\mathit{not}\ c = 1 - c$$

$$\begin{aligned}(\wedge) &= \mathit{min}\\(\vee) &= \mathit{max}\end{aligned}$$

$$\textbf{if}\ c\ \textbf{then}\ a\ \textbf{else}\ b = c \cdot a + \mathit{not}\ c \cdot b$$

Note in this last definition that **if-then-else** is strict.[7]

6.3 Literal extraction

Because the target instruction set does not support literals, the compiler must extract literals and allocate them into the constant register set. Extraction proceeds in three phases: *discover* the literals, *pack* efficiently into a constant register file, and *replace* the literals with variables (possibly swizzled and negated).

```
extractLiterals :: Int → Exp → (Exp, RegFile)
extractLiterals numRegs exp =
  (replace regs exp, regs)
  where
    regs = pack numRegs (discover exp)

discover :: Exp → [Vector]
pack     :: Int → [Vector] → RegFile
replace  :: RegFile → Exp → Exp

type RegFile = [Vector]
```

6.4 Codegen normal form

In preparation for code generation, the Vertigo compiler rewrites expressions into "codegen normal form" (CNF) designed to reflect what the processor can do.

CNF is a subset of the *Exp* type such that:

- There are no literals.
- Operators other than *MkVec* may only be applied to only to "operands", which are swizzled and possibly negated variables.
- Swizzling, negation, and variables show up *only* in these operands. (If necessary, a *Mov* (identity) operator application is inserted.)

Variables will correspond to readable registers, possibly swizzled for layout. Swizzling and negation get rewritten away whenever possible, by using distributive properties and pushing them into operand position where they cost nothing.

For negation, the following distributive properties are used:[8]

$$
\begin{array}{ll}
-(-a) & = a \\
-(a+b) & = (-a)+(-b) \\
-(max\ a\ b) & = min\ (-a)\ (-b) \\
-(min\ a\ b) & = max\ (-a)\ (-b) \\
-(a \cdot b + c) & = a \cdot (-b) + (-c) \\
-(a \cdot b) & = (-a) \cdot b \\
-(1/a) & = 1/(-a) \\
-(a <\cdot> b) & = (-a) <\cdot> b
\end{array}
$$

$$
\begin{array}{ll}
-(e_1, \ldots, e_n) & = (-e_1, \ldots, -e_n) \\
-(e.swiz) & = (-e).swiz
\end{array}
$$

The last rule refers to negations of swizzled expressions. Here *swiz* refers a sequence of of *x*, *y*, *z*, and *w* components (with *n* components if $e :: \mathcal{R}^n$).

Similarly, there are helpful properties for rewriting swizzlings. For all SIMD operations *op*,

$$(op\ e_1\ \ldots\ e_n).swiz = op\ (e_1.swiz)\ \ldots\ (e_n.swiz)$$

Swizzlings of explicit vector constructions get swizzled syntactically, e.g.,

$$(a, b, c).xzyz = (a, c, b, c)$$

Composed swizzles are composed syntactically, e.g.,

$$(e.yzw).yx = e.zy$$

When a negation or swizzling cannot be pushed into an existing operator, we simply introduce a new identity operator (*Mov*) to push it into, which will cost an additional instruction.

CNF conversion also turns combinations of multiply and add into single *Mad* applications.

6.5 Assembly language modeling

An assembly program is simply a list of instructions. All instructions are operator applications (even *Mov*) and contain a comment, in which the compiler inserts a binding in CNF.

```
type Asm  = [Instr]
data Instr = PrimOp Op Dest [Source] String
```

A register has a register class and index and a friendly name

```
data RegClass = RegIn | RegConst | RegTemp
              | RegAddr | RegOut

data Reg = Reg RegClass Int String
```

Source registers may be swizzled and negated. The register may not be an output.

```
data Source = Source NegSwiz Reg

data NegSwiz = NegSwiz Bool Swizzle
type Index   = Int
type Swizzle = [Index]
```

Each destination has a register and a layout saying which floats within the register get used. The register may not be an input.

```
data Dest   = Dest Reg Layout
type Layout = [Index]      -- distinct
```

6.6 Code generation

Given an expression in CNF, code generation is fairly straightforward. Because GPUs have no random memory access, optimized register allocation is particularly important. The Vertigo compiler uses a simple functional implementation of the traditional dynamic programming technique [1].

A "code generator" tells how much free register space is needed (in floats) and how to generate code. The free space requirement will

[7] More modern GPU architectures do support booleans and non-strict conditionals.

[8] These rewrites do not need to be applied recursively. One application suffices to move the negation to operand position. Recall that $<\cdot>$ is dot product.

be used for argument reordering.

$$\textbf{type } \mathit{CodeGen} = (\mathit{Int}, \mathit{Gen})$$

A *Gen* generates code for a given destination, an extra swizzle required to accommodate the destination layout, a mapping from variables to sources, and a pool of free temporary registers.

$$\textbf{type } \mathit{Gen} = \mathit{Dest} \to \mathit{Swizzle} \to \mathit{SourceEnv} \to \mathit{Pool} \to \mathit{Asm}$$

$$\textbf{type } \mathit{SourceEnv} = [(\mathit{Id}, \mathit{Source})] \qquad \text{-- assoc list}$$

Code generation then maps an expression in CNF into a *CodeGen*:

$$\mathit{codegen} :: \mathit{CNF} \to \mathit{CodeGen}$$

Thanks to CNF, there are only two cases: (a) applications of operators to optionally negated and swizzled variables, and (b) **let** expressions.

The application case is simple: for each argument, get the source bound to the variable in the environment, and compose the contextual negation and swizzle with the source's to form the instruction operand. Then use the destination layout as a mask for the result register.

There is one tricky point arising from layout. Variables smaller than $\mathcal{R}^4$ may require swizzling on write, which is not supported in general by the processor architecture. However, for *almost all* operations, a write swizzle can be correctly simulated by a combination of write masking and argument swizzling. For SIMD ops, it suffices to swizzle each argument correspondingly. For scalar-producing ops, the same scalar result is written to all components of the output, so write swizzling is just write masking. The remaining instructions write to all four components, and so do not pose a problem, unless a non-obvious layout were used. To handle this concern, all four-float allocations are given the identity layout, so that unpredictable layout swizzling cannot happen. If we were not so lucky with the instruction set, we could insert a *Mov* instruction that swizzled its argument as necessary.

All register allocation is handled by the **let** case. For an *n*-ary **let**, there are $n+1$ stages of evaluation: one for each right hand side and one for the body. The register use will be the maximum of the register uses over the n+1 stages. At each stage, we have to preserve the registers used to hold the results of previous stages. Since later stages have the added burden of preserving earlier results, we rearrange the bindings to put the less register-intensive bindings later, thus minimizing the maximum register usage over the stages. We cannot move the body, since it depends on all the bindings.

$$\begin{array}{l}
\mathit{codegen}\ (\mathit{Let\ bindings\ body}) = (\mathit{nr}, \mathit{gen}) \\
\quad \textbf{where} \\
\quad\quad (\mathit{vars}, \mathit{types}, \mathit{cgs}) = \\
\quad\quad\quad \mathit{unzip3}\ (\mathit{reorder}\ (\mathit{zip3}\ \mathit{vars}_0 \\
\quad\quad\quad\quad\quad\quad\quad\quad (\mathit{map\ typeOf\ exps}_0) \\
\quad\quad\quad\quad\quad\quad\quad\quad (\mathit{map\ codegen\ exps}_0))) \\
\quad\quad (\mathit{vars}_0, \mathit{exps}_0) \quad = \mathit{unzip\ bindings}
\end{array}$$

Reordering just sorts by decreasing register use:

$$\mathit{reorder} = \mathit{sortF}\ (\lambda(\_, \_, (\mathit{nr}, \_)) \to -\mathit{nr})$$

where *sortF* sorts based on a given key extractor function:

$$\begin{array}{l}
\mathit{sortF} :: \mathit{Ord}\ k \Rightarrow (a \to k) \to [a] \to [a] \\
\mathit{sortF}\ \mathit{key} = \\
\quad \mathit{sortBy}\ (\lambda a\ b \to \mathit{key}\ a\ \text{`}\mathit{compare}\text{`}\ \mathit{key}\ b)
\end{array}$$

To determine the number of free registers needed for the **let** expression, we need to know (a) the space tied up at each stage (sum of the sizes of values saved so far), and (b) the amount of free space needed for each binding.

$$\begin{array}{ll}
\mathit{savedRs} & = \mathit{scanl1}\ (+)\ (\\
 & \quad\quad 0 : \mathit{map\ type2size\ types}) \\
\mathit{nr} & = \mathit{maximum}\ (\\
 & \quad\quad \mathit{zipWith}\ (+)\ (\mathit{nrs} +\!\!+ [\mathit{nrb}])\ \mathit{savedRs}) \\
(\mathit{nrs}, \mathit{gens}) & = \mathit{unzip\ cgs} \\
(\mathit{nrb}, \mathit{genb}) & = \mathit{codegen\ body}
\end{array}$$

The code generated for the **let** expression comes from code generated for the bindings followed by code for the body.

$$\begin{array}{l}
\mathit{gen\ dest\ swiz\ env\ pool} = \\
\quad \mathit{asm} +\!\!+ \mathit{genb\ dest\ swiz\ env}'\ \mathit{pool}' \\
\quad \textbf{where} \\
\quad\quad (\mathit{asm}, \mathit{env}', \mathit{pool}') = \\
\quad\quad\quad \mathit{genBindings\ vars\ types\ gens\ env\ pool}
\end{array}$$

Code generation for bindings (*genBindings*) works simply by looping through the (now reordered) bindings, allocating space from the temporary registers, and recursively generating code for the right hand sides.

7 Sample optimizations

In this section, we show examples to give a flavor of the kinds of optimizations that Vertigo performs in practice.

7.1 Vector normalization

It is common to need to normalize vectors (i.e., scale them to unit length). One use is the construction of normals for shading (Section 5.2) and for displacement surfaces (Section 4.4). A painful tradeoff in graphics programming is whether utility functions like normal computation should normalize their vector arguments or assume them to have been normalized. Since execution speed is so important, the choice is often made to assume pre-normalization, so that the normalization can be avoided in a few cases. Unfortunately, this choice encourages one of the classic computer graphics programming bugs, which is failure to normalize before calling, either due to forgetting requirement or falsely assuming it to hold.

With a sufficiently aggressive optimizer, one might hope to eliminate the pre-normalization requirement and still get efficient code when the actually argument has been normalized. That is, the compiler should perform the following optimization (interprocedurally).[9]

$$\mathit{normalize}\ (\mathit{normalize}\ v) = \mathit{normalize}\ v$$

Rather than wire this domain-specific optimization into an otherwise domain-independent compiler, Vertigo performs simpler and more general rewrites. Given the definition of *normalize* from Sec-

[9] There may be other, subtler, sources of redundant normalization.

tion 4.4), *normalize* (*normalize v*) expands to

$$\mathit{normalize}\ v/\mathit{sqrt}\ (\mathit{normalize}\ v{<}{\cdot}{>}\mathit{normalize}\ v)$$

The sub-expression *normalize v*<·>*normalize v* expands to

$$(v/\mathit{sqrt}\ (v{<}{\cdot}{>}v)){<}{\cdot}{>}(v/\mathit{sqrt}\ (v{<}{\cdot}{>}v))$$

The following rewrites apply, with r and s ranging over scalars and u and v over vectors:[10]

$$\begin{array}{ll} v/s & = (1/s)\cdot v \\ (s\cdot u){<}{\cdot}{>}v & = s\cdot(u{<}{\cdot}{>}v) \\ (1/r)\cdot(1/s) & = 1/(r\cdot s) \\ \mathit{sqrt}\ r\cdot\mathit{sqrt}\ s & = \mathit{sqrt}\ (r\cdot s) \\ \mathit{sqrt}\ (s\cdot s) & = s \end{array}$$

The result is

$$(v{<}{\cdot}{>}v)/(v{<}{\cdot}{>}v)$$

which simplifies to 1. The overall expression then becomes

$$\mathit{normalize}\ v/\mathit{sqrt}\ 1$$

which simplifies to *normalize v*.

As a particularly fortuitous example of this optimization and others, consider *normal sphere*. Without optimization there are 28 additions, 50 multiplications, and four trigonometry operations. With optimization there are two additions, four multiplications, and four trigonometry operations. In fact, the result is identical to *sphere* itself, so the savings are compounded when rendering a sphere, which requires the surface and its normal.

7.2 Cross products

The previous example is architecture-independent. The definition of 3D cross products, used also in normal computation, gives rise to an example of architecture-specific optimization.

$$\begin{array}{l} (\times) :: (\mathit{Num}\ s, \mathit{VectorOf}\ s\ (s,s,s)) \\ \quad \Rightarrow (s,s,s) \to (s,s,s) \to (s,s,s) \\ (a_1,b_1,c_1)\ \times\ (a_2,b_2,c_2) = \\ \quad (b_1\cdot c_2 - b_2\cdot c_1,\ c_1\cdot a_2 - c_2\cdot a_1,\ a_1\cdot b_2 - a_2\cdot b_1) \end{array}$$

Automatic vectorization performs the following transformation for all SIMD operations *op*:

$$\begin{array}{l} (\mathit{op}\ a_1\ b_1\ \ldots, \mathit{op}\ a_2\ b_2\ \ldots, \ldots, \mathit{op}\ a_n\ b_n\ \ldots) = \\ \quad \mathit{op}\ (a_1,a_2,\ldots,a_n)\ (b_1,b_2,\ldots,b_n)\ \ldots \end{array}$$

This rule applies four times in the definition of $\times$ (since subtraction is represented by addition and unary negation), yielding

$$\begin{array}{l} (a_1,b_1,c_1)\ \times\ (a_2,b_2,c_2) = \\ \quad (b_1,c_1,a_1)\cdot(c_2,a_2,b_2) - (b_2,c_2,a_2)\cdot(c_1,a_1,b_1) \end{array}$$

Note that each constructed vector is a rearrangment of one of the vector arguments to $\times$. That fact means that the vectors are just swizzlings:

$$u\ \times\ v = u.yzx\cdot v.zxy - v.yzx\cdot u.zxy$$

[10]The Vertigo compiler matches for these rules modulo associativity and commutativity of multiplication and dot product.

In CNF, the body of this definition is

$$\begin{array}{l} \textbf{let}\ q = v.yzx\cdot u.zxy\ \textbf{in} \\ \quad \mathit{mad}\ (u.yzx)\ (v.zxy)\ (-q) \end{array}$$

Assuming u and v are allocated in the first three floats of registers `r0` and `r1` respectively, and the result should go into the first float of `r2`, Vertigo produces the following two instructions:

```
mul r2.x, r1.yzx, r0.zxy
mad r2.x, r0.yzx, r1.zxy, -r2.x
```

8 Derivatives

The *derivative* operator maps functions to functions. In general, the derivative of a function of type $\alpha \to \beta$ is a function of type $\alpha \to L(\alpha;\beta)$, where "$L(\alpha;\beta)$" means the *linear* subset of $\alpha \to \beta$ [13]. These linear maps are typically represented by real numbers, vectors, matrices, etc, depending on α and β. Because Vertigo uses these data representations rather than functions for derivative values, *derivative* belongs to a multi-parameter type class:

$$\begin{array}{l} \textbf{class}\ \mathit{Derivative}\ \alpha\ \beta\ \mathit{lmap} \mid \alpha\ \beta \to \mathit{lmap}\ \textbf{where} \\ \quad \mathit{derivative} :: (\alpha \to \beta) \to (\alpha \to \mathit{lmap}) \end{array}$$

When $\alpha = \mathcal{R}$, $L(\alpha;\beta)$ can be represented by β for vector space types β.

$$\begin{array}{l} \textbf{instance}\ (\mathit{DDeriv}\ b, \mathit{Substable}\ b) \Rightarrow \\ \quad \mathit{Derivative}\ \mathcal{R}\ b\ b\ \textbf{where}\quad \ldots \end{array}$$

If, for instance, $\beta = \mathcal{R}^3$, then the derivative values are represented as vectors of three scalar-valued "partial derivatives".

When $\alpha = \alpha_1 \times \cdots \times \alpha_n$, $L(\alpha;\beta)$ can be represented by $\gamma_1 \times \cdots \times \gamma_n$, where γ_i represents $L(\alpha_i;\beta)$.

$$\begin{array}{l} \textbf{instance}\ (\mathit{Derivative}\ \alpha_1\ \beta\ \gamma_1 \\ \qquad , \mathit{Derivative}\ \alpha_2\ \beta\ \gamma_2) \Rightarrow \\ \quad \mathit{Derivative}\ (\alpha_1,\alpha_2)\ \beta\ (\gamma_1,\gamma_2)\ \textbf{where}\quad \ldots \end{array}$$

Similarly for triples, etc. When β happens to be a tuple type, the resulting derivative value representation is a tuple of tuples and coincides with what is known as a "Jacobian matrix".

The *Substable* type class contains types that support "substitution" of an expression for a variable. In the Vertigo implementation, the *Substable* instances are $\mathcal{R}$ and tuples of *Substable* types. Differentiation of functions works by applying the function to one or more variable expressions, symbolically differentiating the resulting expression(s), and turning the result back into a function that substitutes for the introduced variables.

The *DDeriv* class supports differentiation with respect to *variables*. It includes $\mathcal{R}$ (expressions over *Float*), and tuples of *DDeriv* types. The $\mathcal{R}$ case simply removes the statically typed wrapper, revealing an underlying *Exp* (Section 6.1), where the actual, recursive symbolic differentiation algorithm takes place. It is critical for efficiency to memoize that algorithm, in order to avoid the usual problem of time and space blow-up for symbolic differentiation. This differentiation algorithm is very simple (Figure 8).

9 Further work

While the Vertigo compiler does a good job of algebraic simplification, reducing instructions generated and registers used, there is

```
ederiv :: Id → Exp → Exp
ederiv v exp = d exp
  where
    d                          = memo nd
      —— nd is the non-memoized d
    nd e@(LitVec _)            = zero (typeOf e)
    nd (Var v' ty) | v == v'   = one ty
    nd (Var _ ty)              = one ty
    nd (Apply Add [u, v])      = d u + d v
    nd (Apply Mul [u, v])      = u · d v + v · d u
    nd (Apply Rcp [v])         = −d v/(v^2)
    nd (Apply Sin [u])         = cos u · d u
    nd (Apply Cos [u])         = −sin u · d u
    nd (Apply Rsq [u])         = −d u · rsqrt u/(twoF · u)
    nd e@(Apply Exp [u])       = e · d u · logTwoF
    nd (Apply Log [u])         = recip u · logTwoF · d u
    nd (Apply Negate [u])      = −(d u)
    nd (Apply MkVec es)        = vecL (map d es)
    nd (Apply (Swizzle s) [u]) = swizzle s (d u)
    nd (Apply Frac [u])        = d u
    nd (Apply Dp3 [u, v])      = dp3 u (d v) + dp3 v (d u)
    nd (Apply Dp4 [u, v])      = dp4 u (d v) + dp4 v (d u)
    nd (Apply Slt [u, v])      = zero (typeOf u)
    nd (Apply Sge [u, v])      = zero (typeOf u)
    nd (Apply Max [u, v])      = ifE (u > v) (d u) (d v)
    nd (Apply Min [u, v])      = ifE (u < v) (d u) (d v)
```

Figure 6. Symbolic differentiation

much room for improvement.

One improvement would be connecting algebraic simplification with register allocation. For instance, the automatic vectorization transformation mentioned in Section 7.2 replaces one vector construction with n of them, and is only beneficial when the vector constructions become register swizzles, which are free. More generally, it is important to coalesce scalar operations into vectors operations where possible, but not at the cost of moving scalars into vectors at run-time. More sophisticated analysis could allocate scalars in the same vector at compile time, when doing so would allow replacing several scalar operations with vector operations. Since the same scalar may be used in more than once, there may be competition among different potentially vectorizable uses of a given scalar.

A related issue is the tension between optimization and sharing. Consider the definition of **if-then-else** in Section 6.2. Optimizing *not c* could easily break the sharing of part of the computation of *c*, which may more than defeat the optimization.

Newer generations of graphics vertex processors have more powerful instruction sets, including looping, predicated instructions, conditional branching, boolean and integer registers. They also have larger register sets and program length bounds. These advancements introduce opportunities and challenges for compilation. New pixel processors are also much more general and now worth targeting. Another general challenge is partitioning computation between vertex and pixel processors.

10 Conclusions

Programmable multiprocessor architectures have finally reached the masses in the form of modern graphics cards. This is a great opportunity for functional programming, because statelessness naturally fits the hardware, and because the objects of interest in computer graphics tend to be functions. This paper describes Vertigo, a functional language for 3D shape and shading and an optimizing compiler that targets graphics processors. The language has simple, transparent semantics in terms of first-class functions. Higher-order programming provides powerful abstractions that allow surfaces to be composed from simpler components, often of lower dimension.

Shading languages since Renderman's have had a rather peculiar execution model, explained as "instancing", "calling", and iteration over light sources. As we have shown, execution can be explained simply as curried functions having natural staging: shader-specific parameters (instancing), view and surface point information (calling), and per-light information.

The Vertigo system runs on Windows, with DirectX 9 and the .NET framework. It may be downloaded from `http://conal.net/Vertigo`.

11 References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, Tools*. Addison-Wesley, 1986.

[2] C. Elliott, S. Finne, and O. de Moor. Compiling embedded languages. *Journal of Functional Programming*, 13(2), 2003. `http://conal.net/papers/jfp-saig`.

[3] R. Fernando and M. J. Kilgard. *The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics*. Addison Wesley, 2003.

[4] P. Hanrahan. Why is graphics hardware so fast? Unpublished talk, 2002. `http://graphics.stanford.edu/~hanrahan/talks/why`.

[5] P. Hanrahan and J. Lawson. A language for shading and lighting calculations. In *SIGGRAPH Proceedings*, 1990.

[6] J. Karczmarczuk. Geometric modelling in functional style. In *International Latino-American Conference on Functional Programming*, 1999.

[7] J. R. Lewis, M. Shields, J. Launchbury, and E. Meijer. Implicit parameters: Dynamic scoping with static types. In *Symposium on Principles of Programming Languages*, 2000.

[8] E. Lindholm, M. J. Kilgard, and H. Moreton. A user-programmable vertex engine. In *SIGGRAPH Proceedings*, 2001.

[9] Microsoft. Microsoft DirectX 8.1 programmable vertex shader architecture. `http://msdn.microsoft.com`.

[10] K. Proudfoot, W. R. Mark, S. Tzvetkov, and P. Hanrahan. A real-time procedural shading system for programmable graphics. In *SIGGRAPH Proceedings*. ACM Press, 2001.

[11] J. M. Snyder. *Generative modeling for computer graphics and CAD: symbolic shape design using interval analysis*. Academic Press Professional, Inc., 1992.

[12] J. M. Snyder and J. T. Kajiya. Generative modeling: a symbolic system for geometric modeling. In *SIGGRAPH Proceedings*. ACM Press, 1992.

[13] M. Spivak. *Calculus on Manifolds*. Westview Press, 1965.

[14] S. Upstill. *The RenderMan Companion*. Addison-Wesley, Reading, MA, 1989.