Symbolic Differentiation in GPU Shaders

Derivatives arise frequently in graphics and scientific computation applications. As GPU's become more widely used for scientific computation the need for derivatives can be expected to increase. To meet this need we have added symbolic differentiation as a built in language feature in the HLSL shading language. The symbolic derivative is computed at compile time so it is available in all types of shaders (geometry, pixel, vertex, etc.). The algorithm for computing the symbolic derivative is simple and has reasonable compilation and run time overhead.

1 Introduction

As scientific computations increasingly migrate to the GPU [Hanrahan 2009], GPU shading languages like CG and HLSL are changing from special purpose graphics shading languages to mainstream scientific/engineering languages. In reviewing the types of applications likely to be executed on the GPU now and in the near future we noticed a recurring pattern. Many scientific/engineering and graphics computations require derivatives to compute such functions as:

- line and surface tangents, surface normals, surface curvature
- Jacobians of $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$, such as texture to screen space mappings, robot manipulator Jacobians
- velocities, accelerations, Lagrangians

Computing derivatives for use in GPU shaders is cumbersome and inefficient. Derivatives can be computed with symbolic math programs such as Mathematica or Maple, but this requires two languages, programming environments, and code bases. It also causes versioning problems when changes to the symbolic function are not propagated to the corresponding shader function.

Derivatives can be computed by hand, which is tedious, error prone, and impractical for functions of the size commonly encountered in today's shader programs. Derivatives can be approximated numerically, but this introduces both approximation and roundoff errors which are difficult to predict and generally impossible to correct.

Motivated by the frequent need to compute accurate, efficient derivatives, and the present difficulty of doing so, we have added symbolic differentiation as a built in HLSL language feature[1]. It is available in all types of shaders (geometry, pixel, vertex, etc.) and any hardware that is compatible with DirectX 9 or higher. because it is a compiler feature; the symbolic derivative is computed at compile time and derivative code is generated along with the original function code.

[1]Released in the June 2010 DirectX SDK.

2 Previous Work

Symbolic differentiation has previously only been available as a built in language feature in special purpose mathematical programming languages such as Mathematica and Maple. The differentiation and symbolic simplification algorithms used in symbolic math programs are not suitable for a mainstream language like HLSL because they are complex and can have unacceptable memory and time requirements as expressions get large.

Derivatives could be computed with a source to source transformation automatic differentiation program such as ADIC[Bischof et al. 1997]. The C source for a function is transformed into a new source file which contains a function that computes the derivative. However ADIC takes C programs as input; GPU shader language syntax is not identical to that of C. This requires manual modification of both the input and output programs to make them compatible with the shader language syntax. In addition, because automatic differentiation does not create a symbolic derivative, it is difficult to perform simple expression graph transformations and algebraic simplifications that can sometimes significantly improve efficiency.

The D* algorithm [Guenter 2007] creates efficient derivatives by factoring the *derivative graph*, described in section 4. It is relatively fast and generates efficient derivatives but its implementation is complex and retrofitting it into an existing compiler requires a substantial engineering effort. In addition, the time to compute the derivative can be as high as $O(v^4)$ in the worst-case[2], where v is the size of the expression graph being differentiated. This could lead to unacceptably long compile times.

The RenderMan shading language [Upstill 1989] includes a differentiation feature that allows users to take derivatives with respect to parameterizing variables of surfaces. But it is not specified if the derivative is computed numerically or symbolically. The Renderman language is not a mainstream, widely used language; it is used primarily for high end film special effects production. It is not a GPU language

Motivated to reduce complexity and compilation time we developed a symbolic differentiation algorithm, onePass, which is much simpler than D* and takes $O(n_d v)$ time in the worst case to compute the derivative, where n_d is the domain dimension of the function being differentiated. Like D* onePass factors the derivative graph and applies common subexpression elimination and simple algebraic simplification. The price we pay for simplicity and quick compilation is potentially inefficient runtime peformance. In the worst case the derivatives generated by onePass may do n_d times as much computation as those generated by D*. The functions used in graphics applications typically have domain dimension not greater than 4 and our experience, so far, has been that the difference in runtime efficiency is generally acceptable. In the future, as general purpose processing becomes more prevalent and higher domain dimension functions become more common, we may revisit this decision and implement the full D* algorithm.

3 Contributions

To the best of our knowledge this is the first time that symbolic differentiation has been included as a language feature in a main-

[2]Typical case complexity varies between $O(v^2)$ and $O(v^3)$

stream conventional programming language, as opposed to a special purpose symbolic language like Maple or Mathematica. The contributions of this paper are:

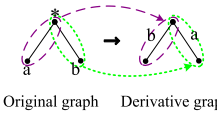
- a demonstration that symbolic differentiation can be integrated into existing "C" like programming langauges, with minimal compilation and run time overhead
- a simple, fast algorithm to compute efficient symbolic derivatives during compilation
- a description of the implementation compromises that inevitably arise when making such a radical change to a programming language

Because symbolic differentiation is such an unconventional feature for mainstream programming languages we have included two fully worked out example applications in section 6 to demonstrate how this feature can simplify and improve many common graphics computations.

4 The onePass Differentiation Algorithm

The onePass algorithm implicitly factors the derivative graph by performing a simple, one pass (hence the name) traversal of the expression graph. The derivative graph is the special graph structure which represents the derivative of an expression graph. Before describing the algorithm we will briefly review the derivative graph concept and notation; for more details see [Guenter 2007].

The derivative graph of an expression graph has the same structure as the expression graph but the meaning of nodes and edges is different. In a conventional expression graph nodes represent functions and edges represent function composition. In a derivative graph an edge represents the partial derivative of the parent node function with respect to the child node argument. Nodes do not represent operations; they serve only to connect edges. For example Fig.1 shows the graph representing the function $f = ab$ and its corresponding derivative graph.



Original graph Derivative graph

Figure 1: *The derivative graph of multiplication. The derivative graph has the same structure as its corresponding expression graph but the meaning of edges and nodes is different: edges represent partial derivatives and nodes have no operational function.*

The edge connecting the $*$ and a symbols in the original function graph corresponds to the edge representing the partial $\frac{\partial ab}{\partial a} = b$ in the derivative graph. Similarly, the $*, b$ edge in the original graph corresponds to the edge $\frac{\partial ab}{\partial b} = a$ in the derivative graph.

A path product is the product of all the partial terms on a single non-branching path from a root node to a leaf node. The sum of all path products from a root node to a leaf node is the derivative of the function. Factoring the path product expression is the key to generating efficient derivatives. As an illustrative example we will apply onePass to the derivative graph of Fig. 2. In the sum of all path products form[3] the derivative is[4]:

[3]The terms are ordered as they would be encountered in a traversal from a root to a leaf node.

[4]For $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$, f^i_j is the derivative of the ith range element with respect to the jth domain element.

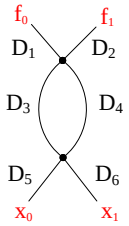


Figure 2: *Derivative graph for an arbitrary function. The D_i terms are partial derivatives.*

$$\begin{aligned} f^0_0 &= D_1D_3D_5 + D_1D_4D_5 \\ f^0_1 &= D_1D_3D_6 + D_1D_4D_6 \\ f^1_0 &= D_2D_3D_5 + D_2D_4D_5 \\ f^1_1 &= D_2D_3D_6 + D_2D_4D_6 \end{aligned}$$

By the nature of its graph traversal order the onePass algorithm implicitly factors out common *prefixes* of the sum of products expression. For example the derivative $[f^0_0, f^1_0]$, as computed by onePass, is:

$$f^0_0 = D_1(D_3D_5 + D_4D_5) \quad (1)$$
$$f^1_0 = D_2(D_3D_5 + D_4D_5) \quad (2)$$

The prefixes D_1 and D_2 are factored out[5]. If we run onePass again to compute $[f^0_1, f^1_1]$ we get:

$$f^0_1 = D_1(D_3D_6 + D_4D_6) \quad (3)$$
$$f^1_1 = D_2(D_3D_6 + D_4D_6) \quad (4)$$

Again, prefixes D_1 and D_2 are factored out, but the common term $D_3D_5 + D_4D_5$ in eq. 1 is not the same as the common term $D_3D_6 + D_4D_6$ in eq. 3 so we do redundant work[6].

The derivative $[f^0_0, f^0_1, f^1_0, f^1_1]$, as computed by onePass, requires 8 multiplies and 2 adds. By comparison, D* would factor out common prefixes *and* suffixes to compute this derivative

$$\begin{aligned} f^0_0 &= D_1(D_3 + D_4)D_5 \\ f^0_1 &= D_1(D_3 + D_4)D_6 \\ f^1_0 &= D_2(D_3 + D_4)D_5 \\ f^1_1 &= D_2(D_3 + D_4)D_6 \end{aligned}$$

which takes 6 multiplies and 1 add, only $\frac{7}{10}$ of the computation required by the onePass derivative.

The onePass algorithm, shown in pseudocode below, performs a single traversal up the expression graph, accumulating the derivative as it goes[7].

[5]Common subexpression elimination causes the common term $D_3D_5 + D_4D_5$ to be computed only once.

[6]The D* algorithm eliminates this redundancy, as well as others.

[7]For those familiar with the terminology of automatic differentiation, the onePass algorithm is the symbolic equivalent of forward automatic differentiation.

Each node in the expression graph is a symbolic expression, called `Symb` in the pseudocode. A node is either a function, such as cos, sin, *, +, etc., or a leaf, such as a variable or constant. Every function has an associated `partial` function which defines the partial derivative of the function with respect to an argument of the function. For example for the `sin()` function `partial(arg)= cos(arg)`.

```
Symb[] dval = new Symb[range.Length];

onePass(v) {//v is the variable differentiating wrt
   for (int i = 0; i < range.Length; i++) dval[i] = range[i].D(v);
}

Symb D(v) {
   if (v is a variable){
      if (v is the current node) { return 1.0; }
      else { return 0.0; }
   }
   if (v is a constant) return 0.0;
   if (this node has already been visited) { return cached derivative; }
   mark this node as visited;
   Symb sum = 0.0;
   for (int i = 0; i < args.Length; i++){
      Symb argi = args[i];
      sum = sum + partial(i) * argi.D(v);
   }
   return sum;
}
```

After executing `onePass` entry i in the returned array contains $\frac{\partial f^i}{\partial v}$, where v is the variable being differentiated with respect to. For functions of the form $f : \mathbb{R}^1 \rightarrow \mathbb{R}^m$; for $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$ onePass is applied to each of the n $\mathbb{R}^1 \rightarrow \mathbb{R}^m$ function subgraphs.

The expression graph contained in the array returned by onePass is a purely symbolic representation of the derivative expression and can be used as an argument to other functions. Higher order derivatives are computed by repeated application of onePass.

5 Implementation

In section 5.1 we describe the changes to the language syntax necessary for writing derivatives, as well as limitations of the current implementation. In section 5.2 we explain an extensibility mechanism called derivative assignment, which allows the user to add derivative rules for functions not currently supported by the compiler. In the code examples in this section we generally elide variable definitions, which are always scalar floating point.

5.1 Syntax and Limitations

The syntatical changes to the language are minimal. Derivatives are specified with the ` operator, which was previously unused:

```
Float times = a*b;
Float Dtimes = times`(a); \\partial with respect to a
Float DDtimes = times`(a,a); \\second partial
Float DaDbtimes = times`(a,b); \\partial a, partial b
```

Derivatives can only be taken with respect to leaf nodes, not internal graph nodes:

```
x = 2*p*p;
float Dx = x`(2*p*p); \\can't take derivative wrt expressions
```

Conceptually[8] the compiler tracks variable definitions by inlining all function calls, which allows derivatives through function calls[9].

[8] The actual implementation is quite different but this is a good mental model of how the compiler processes functions.

[9] Although see section 5.2 for an exception to this rule.

For example this code

```
Float timesFunc(a){
return a*a;
}
Float DtimesFuncDp = timesFunc(p)`p; \\this compiles
```

after inlining becomes

```
Float temp{
return p*p;
}
Float DtimesFuncDp = temp`p;
```

There are several limitations in the current implementation. Some of these exist to keep the implementation as simple as possible, and some of them arise because of deeper issues with efficiently evaluating derivatives in the GPU environment, which doesn't have a system stack.

The current implementation will not differentiate through loop bodies, unless those bodies are annotated with `unroll`. Derivatives of `if` statements will not compile unless the compiler can transform the `if` statement into the ?: form. For example, this code

```
if(y<0){x = z*y;}
else{x = sqrt(z*y);}
Float Dx = x`(z); \\doesn't compile
```

will be transformed into

```
x=(y<0):y*y ? sqrt(y);
Float Dx = x`(z); \\this compiles
```

Derivatives cannot be taken with respect to array elements, nor can array elements be differentiated with respect to variables.

5.2 Derivative Assignment

The current implementation has derivative rules for many common functions but, to make the derivative feature extensible, we added a general mechanism, derivative assignment, that allows the user to define derivative rules for arbitrary functions. For example, discontinuous functions, such as `ceil`, `frac`[10], and `floor` do not have compiler differentiation rules. But they can be used if the derivatives are defined by assignment. For example, this code

```
Float fr = frac(a0);
Float Dfr = fr`(a0);
```

will not compile, but does compile after using derivative assignment

```
Float fr = frac(a0);
Float fr`(a0) = 1;
```

Derivative assignment adds great generality to the system but occassionally interacts counterintuitively with variable tracking by unexpectedly causing derivatives to be taken with respect to internal graph nodes, which is not allowed. For example, derivative assignment in this function

```
Noise(s) {
q = s + 1;
x = ceil(q);
x`(s) = 0; \\have to define derivative of ceil()
return x*x; }
```

won't work as expected when the function is called this way

```
b = Noise(2*p*p)`p
```

[10] `frac(x) = x-floor(x)`

because after inlining the original `Noise()` function is transformed to

```
Float temp; {
q = (2*p*p) + 1;
x = ceil(q);
x‘(2*p*p) = 0; \\can’t take derivative wrt expressions
temp = x*x; }
Float b = temp‘(p);
```

The problem is the line

```
x‘(2*p*p) = 0; \\can’t take derivative wrt expressions
```

The compiler doesn’t allow derivatives to be taken with respect to the expression 2*p*p that has been substituted for s. But the derivative assignment can’t be removed because the compiler doesn’t have a derivative rule for `ceil()`. In addition, derivative assignment follows conventional scoping rules, so the derivative assignment can’t be moved outside of the `Noise()` function . The solution is to pass in an auxiliary variable that will take the place of the variable we will eventually be differentiating with respect to:

```
Noise(s, t) {
q = s + 1;
x = ceil(Q);
x‘(t) = 0;
Return x*x; }
```

Then when `Noise(s,t)` is called like this

```
b = Noise(2*p*p,p)‘p
```

this expands into:

```
Float temp; {
q = (2*p*p) + 1;
x = ceil(q);
x‘(p) = 0; \\this works
temp = x*x; }
Float b = temp‘(p); \\this compiles
```

6 Examples

The range of calculations involving derivatives is vast so we cannot possibly show a representative sampling here. Instead, we use two graphics examples to show the benefits of the new symbolic differentiation feature. These examples are simple enough to be easily understood and implemented; full source code is included in the supplemental materials.

The procedural surface example uses an analytic description, $\boldsymbol{s} : \mathbb{R}^2 \rightarrow \mathbb{R}^3$, of a parametric surface encoded as a function in a pixel shader. Symbolic differentiation is used to create a general procedure to automatically compute surface normals, relieving the user of this difficult and error prone task. Any differentiable parametric surface can be rendered this way, including B-spline, NURBS, and Bezier patch surfaces.

The procedural texture example uses directional derivatives to reparameterize a volume texture function, $t : \mathbb{R}^3 \rightarrow \mathbb{R}^1$, in the u, v parametric space of a procedural surface.

Both examples demonstrate how the ability to compute accurate derivatives can dramatically reduce memory footprint and memory IO.

Once the surface and/or texture function is defined the normal calculation, rendering, and triangulation is handled simply and automatically by the shader; this makes it possible to use many different surface/texture descriptions without having to code special purpose tessellation routines for each type.

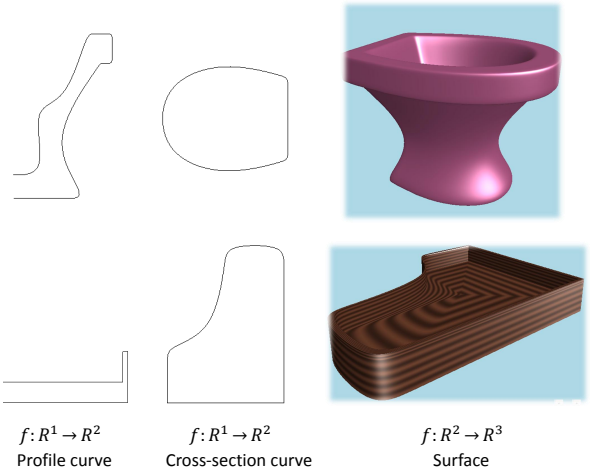


Figure 3: *The profile product of two 2D curves creates a 3D surface.*

6.1 Procedural Surfaces

Many objects can be modeled by combining 2D curves in simple ways [Snyder 1992]. Our example uses a profile product surface (fig. 3) which is formed by scaling and translating a cross section curve, $\boldsymbol{c}(v) : \mathbb{R}^1 \rightarrow \mathbb{R}^2$, by a profile curve, $\boldsymbol{p}(u) : \mathbb{R}^1 \rightarrow \mathbb{R}^2$:

$$\boldsymbol{s}(u,v) = \begin{bmatrix} p_1(u)c_1(v) \\ p_2(u)c_1(v) \\ c_2(v) \end{bmatrix}$$

To shade these objects we need the surface normal

$$\boldsymbol{n}(u,v) = \frac{\partial \boldsymbol{s}(u,v)}{\partial u} \times \frac{\partial \boldsymbol{s}(u,v)}{\partial v}$$

which requires computing partial derivatives. Using the symbolic differentiation feature this is now straightforward.

The following snippet of code implements a generic function defining a 2D B-spline curve, $\boldsymbol{f}(t) : \mathbb{R}^1 \rightarrow \mathbb{R}^2$:

```
struct PARAMETRICCURVE2D_OUTPUT {
float2 Position : POSITION; // 2D position on the curve
};

PARAMETRICCURVE2D_OUTPUT FUNC_LABEL ( const float input ) {
float scaledT = float( CONTROL_POINT_COUNT - 3 ) * input;
uint currentSegment = uint(floor(scaledT));

float2 c0 = CONTROL_POINTS[currentSegment + 0];
float2 c1 = CONTROL_POINTS[currentSegment + 1];
float2 c2 = CONTROL_POINTS[currentSegment + 2];
float2 c3 = CONTROL_POINTS[currentSegment + 3];
scaledT = frac(scaledT); // Param is in [0..1] range
scaledT‘(input) = 1.0f;

float relativeT = clamp(scaledT, 0.f, 1.f);

float relativeT2 = relativeT * relativeT;
float relativeT3 = relativeT2 * relativeT;
float4x4 controlMatrix = float4x4(c0.xy, 0, 0,
c1.xy, 0, 0,
c2.xy, 0, 0,
c3.xy, 0, 0);

float4x4 combinedMatrix = mul(BSpline_BaseMatrix, controlMatrix);
float4 tVector0 = float4(relativeT3, relativeT2, relativeT, 1.0f);
```

```
PARAMETRICCURVE2D_OUTPUT output;
output.Position = mul(tVector0, combinedMatrix).xy;
return output;
}
```

This is the function that computes the profile product surface:

```
struct PARAMETRICSURFACE3D_OUTPUT {
float3 Position : POSITION; // 3D position on the plane
};

PARAMETRICSURFACE3D_OUTPUT ProfileProduct_EvaluateInterp(const float2 input ) {

PARAMETRICCURVE2D_OUTPUT profileCurve = BSpline_Evaluate_FuncU( input.x );
profileCurve.Position *= .004;
PARAMETRICCURVE2D_OUTPUT crossSectionCurve = BSpline_Evaluate_FuncV( input.y );
crossSectionCurve.Position *= .01;

/// Explicitly tell the compiler there is no interdependence of parameters.
profileCurve.Position`(input.y) = 0;
crossSectionCurve.Position`(input.x) = 0;

PARAMETRICSURFACE3D_OUTPUT output;
// Compute position
output.Position.x = crossSectionCurve.Position.x * profileCurve.Position.x;
output.Position.y = crossSectionCurve.Position.y * profileCurve.Position.x;
output.Position.z = .7 - profileCurve.Position.y;
return output; }
```

The two derivative assignments

```
/// Explicitly tell the compiler there is no interdependence on parameters.
profileCurve.Position`(crossSectionCurve.Param) = 0;
crossSectionCurve.Position`(profileCurve.Param) = 0;
```

are necessary because establishing the independence of the two parameterizing variables requires dependency analysis across function boundaries, which the current implementation does not do.

The code for computing the surface normal is completely independent of the surface definition code:

```
PS_INPUT_HARDWARE VS_Hardware(const PARAMETRICSURFACE3D_OUTPUT surfacePoint,
      float2 domainPoint) {
float4 positionWorld = mul( float4(surfacePoint.Position, 1), LocalToWorld );
float4 positionView = mul( positionWorld, WorldToCamera );
float4 positionScreen= mul( positionView, CameraToNDC );

PS_INPUT_HARDWARE output;
output.Position = positionScreen; //
output.Position = float4(domainPoint.x * 2.0f - 1.0f, -(domainPoint.y * 2.0f -
      1.0f), 0.5f, 1.0f);
output.DomainPos = domainPoint;
output.LocalPos = surfacePoint.Position;
float3 dF_du = surfacePoint.Position`(domainPoint.x);
float3 dF_dv = surfacePoint.Position`(domainPoint.y);
float3 norm = cross(dF_du, dF_dv);
output.LocalNormal = normalize(norm);
return output;
}
```

As a consequence many other types of parametric surfaces, such as B-spline, NURBS, and Bezier patch surfaces, can easily be incorporated into this framework.

Surfaces defined using this framework are entirely represented by a shader which contains the surface coefficients as constants. Once the shader is loaded into the GPU no futher GPU memory IO is necessary to render the surface[11].

6.1.1 Real Time Triangulation

Procedural surfaces can be rendered at any desired triangulation level without having to generate vertices or index buffers on the CPU and then pass them to the GPU. The basic idea is to use the "system value semantic" SV_VertexId (available in DX10 and later). This causes the GPU to generate vertex id's and pass them to the vertex shader, without transferring any triangle data from the CPU to the GPU. From the vertex id, we can calculate the location of the vertex being processed within the grid of our virtual tessellation, and from that generate the domain coordinates.

First we compute the number of triangles, n_t, to be used

$$n_t = (n_u\text{–}1) * (n_v\text{–}1)$$

where n_u, n_v are the number of sample points along the u, v axes, respectively. Then we set the vertex layout to NULL (signifying there is no vertex buffer) and issue a draw call in the form:

```
Device->Draw(NumTriangles * 3, 0);
```

In the vertex shader, the vertex input is declared as:

```
struct VSHARDWARE_INPUT { uint Index : SV_VertexID; };
```

where the only input data is the system generated vertex id. The GPU hardware automatically makes NumTriangles*3 calls to the vertex shader, each time incrementing the value of Index by 1. The variable Index is converted to UV coordinates as shown in this shader code snippet

```
const uint tri = input.Index / 3;
const uint quad = tri / 2;
const bool odd = (tri % 2) != 0;
const float U = quad % int(NumU 1);
const float V = quad / int(NumU - 1);
const uint triVert = input.Index % 3;
const float triU = U + ( odd ? (triVert + 1) / 2 : triVert % 2 );
const float triV = V + ( odd ? (triVert + 1) % 2 : triVert / 2 );
domainPoint = float2(triU, triV) / float2(NumU 1, NumV 1);
```

where NumU and NumV are the number of virtual vertices in the U and V directions.

We can also use the GPU tessellator to generate the vertices[12]. In that case, the domain coordinates are computed automatically in the fixed function tessellator. When using multiple patches to cover the domain [0..1][0..1], we do a little more arithmetic to combine the [0..1][0..1] coordinates over the patch and the patch ID to cover the entire surface:

```
float2 domainPoint;
domainPoint.x = (UV.x + (input.PatchID % (int)NumU)) / NumU;
domainPoint.y = (UV.y + (input.PatchID / (int)NumU)) / NumV;
return domainPoint;
```

6.1.2 Automatic LOD

One of the nice features of procedurally defined geometry is the ease of implementing automatic level of detail, because the models are inherently resolution independent. A still from a real time rendering of 2500 procedural objects, $\frac{1}{3}$ of which have procedural textures, is show in fig. 4. Because it is easy to compute derivatives one could use a curvature based subdivision for level of detail. However this would significantly increase the complexity of the examples. Instead we implemented a simple distance based LOD algorithm:

$$\begin{aligned} p &= 20,000/(d_{co}^2) \\ n_t &= \begin{cases} 500 & p < 500 \\ 10,000 & p > 10,000 \\ p & otherwise \end{cases} \end{aligned}$$

[11] Except for writes to the frame buffer.

[12] On the ATI Radeon 5870 the tessellator version is slightly slower than using vertex id's.

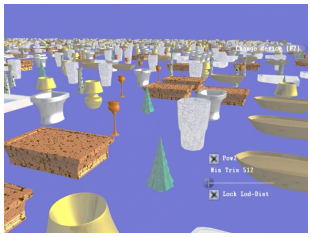


Figure 4: *Still from multi-object real time rendering. There are 2500 objects, each randomly selected from 9 different models, on a 50 x 50 grid.* $\frac{1}{3}$ *of the models have procedural textures. There is no bounding volume culling so all 2500 objects are drawn every frame. Update speed is 39 frames per second with* $1.9 * 10^6$ *triangles/frame.*

where d_{co} is the distance from the camera to the object. The number of triangles used to represent the object is $2^{round(log_2(n_t))}$.

The blending between LODs is then based on the fraction

log2(targetNumTris) – log2(powerOfTwoNumTris)

6.2 Procedural Textures

The procedural textures we will use are functions of the form $t(\boldsymbol{s}(u,v)) : \mathbb{R}^3 \rightarrow \mathbb{R}^1$ which are applied to a surface defined by $\boldsymbol{s}(u,v) : \mathbb{R}^2 \rightarrow \mathbb{R}^3$. A new displaced surface $\boldsymbol{g}(u,v) : \mathbb{R}^2 \rightarrow \mathbb{R}^3$ is generated by displacing the surface along the surface normal, $\boldsymbol{n}_s(u,v)$:

$$\boldsymbol{g}(u,v) = \boldsymbol{s}(u,v) + \boldsymbol{n}_s(u,v)t(\boldsymbol{s}(u,v))$$

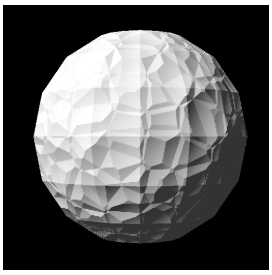
The textured surface $\boldsymbol{g}(u,v)$ can be rendered by creating a true offset surface, as in fig. 7a, or by shading with the surface normal, $\boldsymbol{n}_g$, of $\boldsymbol{g}(u,v)$, as in figs. 7b and 7c (we have elided the arguments to the functions $\boldsymbol{n}_s$ and t to reduce clutter in the equations)

$$\begin{aligned} \boldsymbol{n}_g &= \frac{\partial \boldsymbol{g}}{\partial u} \times \frac{\partial \boldsymbol{g}}{\partial v} \\ &= \frac{\partial \boldsymbol{s}}{\partial u} \times \frac{\partial \boldsymbol{s}}{\partial v} + \frac{\partial \boldsymbol{s}}{\partial u} \times \frac{\partial (t\boldsymbol{n}_s)}{\partial v} + \frac{\partial (t\boldsymbol{n}_s)}{\partial u} \times \frac{\partial \boldsymbol{s}}{\partial v} + \frac{\partial (t\boldsymbol{n}_s)}{\partial u} \times \frac{\partial (t\boldsymbol{n}_s)}{\partial v} \end{aligned} \tag{5}$$

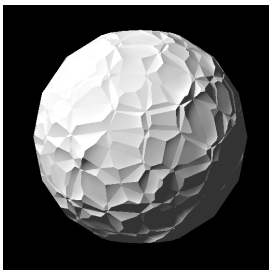
For analytically defined surfaces this completely defines the normal. But, for polygonal surfaces, whose normals are interpolated from vertex values, computing the terms $\frac{\partial \boldsymbol{n}_s}{\partial u}$, $\frac{\partial \boldsymbol{n}_s}{\partial v}$ is problematic because $\boldsymbol{n}_s$ is defined by the vertex normals, the barycentric coordinates, and the screen space partials of these terms, none of which are generally available to the pixel shader. For polygonal surfaces we can set the partials, $\frac{\partial \boldsymbol{n}_s}{\partial u}$, $\frac{\partial \boldsymbol{n}_s}{\partial v}$, to zero. After simplification (5) becomes :

$$\boldsymbol{n}_g \approx \boldsymbol{n}_s + \frac{\partial \boldsymbol{s}}{\partial u} \times \frac{\partial t}{\partial v}\boldsymbol{n}_s - \frac{\partial \boldsymbol{s}}{\partial v} \times \frac{\partial t}{\partial u}\boldsymbol{n}_s$$

where $\boldsymbol{n}_s$ is the interpolated normal value, rather than an analytic surface normal. All of these terms are easily computed in the pixel shader. This approximation of $\boldsymbol{n}_g$ gives a smooth variation in the offset surface normal, which hides the underlying polygonal geometry.



(a) *hardware finite difference*



(b) *symbolic derivative*

Figure 5: *Voronoi texture on a polygonal surface. Notice the shading discontinuities at polygon boundaries in (a) which uses a finite difference approximation to the derivative.*

Since the texture function, $t(\boldsymbol{s}(u,v))$, is not explicity defined in terms of the parameterizing variables we can compute $\frac{\partial t}{\partial u}$, $\frac{\partial t}{\partial v}$ by taking directional derivatives

$$\begin{aligned} \frac{\partial t}{\partial u} &= \bigtriangledown t \cdot \hat{u} \\ \frac{\partial t}{\partial v} &= \bigtriangledown t \cdot \hat{v} \end{aligned}$$

where $\bigtriangledown t$ is the gradient of the texture function and $\hat{u}, \hat{v}$ are axes of an orthonormal frame, $\{\hat{u}, \boldsymbol{n}_s, \hat{v}\}$, based on the surface normal $\boldsymbol{n}_s$. The surface is parameterized using the $\hat{u}, \hat{v}$ axes (chosen in any convenient manner) as parametric directions.

This code snippet shows how the gradient of a texture function is computed:

```
#define EXTRACT_DERIV3(a, b) float4(a`(b.x), a`(b.y), a`(b.z))

float3 NoiseIn = In.WorldPos; // position the noise is sampled at
float VNoise = VoronoiNoise( NoiseIn); //particular noise type
float3 deriv = EXTRACT_DERIV3(VNoise, NoiseIn).xyz // compute the gradient
```

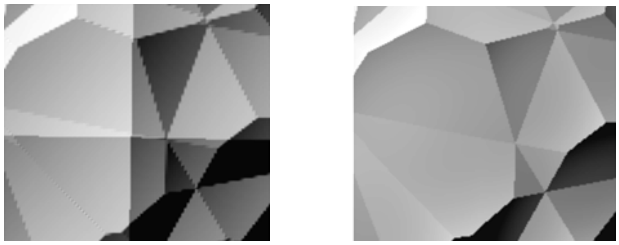
Instead of computing symbolic derivatives we could use the GPU hardware DDX fine/DDY fine[13] finite difference instructions (introduced in Shader Model 5) to compute an approximation to the displaced surface normal:

$$\boldsymbol{n}_g \approx \frac{\boldsymbol{g}(u + \triangle u, v) - \boldsymbol{g}(u,v)}{\triangle u} \times \frac{\boldsymbol{g}(u, v + \triangle v) - \boldsymbol{g}(u,v)}{\triangle v}$$

A comparison of the two methods is shown in figs. 5 and 6. The underlying geometry is a coarse polygonal approximation to a sphere with a Voronoi cell procedural displacement applied to it. Both methods run at the same speed but the finite difference approximation has two types of objectionable artifacts. Because the hardware finite difference is computed on pixel quads there are much more noticeable artifacts along the boundaries of the Voronoi cells. Also, the finite difference approximation shows discontinuities along polygon boundaries. The symbolic and hardware DDX methods have similar performance (Ddx/ddy 540 FPS, symbolic 290 FPS) but the quality of the procedural texture is much better using the symbolic derivative.

Several other textures created using symbolic derivatives are shown in fig. 7. It is straightforward to compose a procedural surface function with a procedural texture function to make a textured procedural surface, fig. 8, that exists entirely in the source code of a shader.

[13]The DDX hardware instruction is less accurate and produces much more noticeable image artifacts.



(a) *hardware finite difference* **(b)** *symbolic derivative*

Figure 6: *Close up of Voronoi texture. The finite difference approximation introduces jagged edges along Voronoi cell boundaries*

For simplicity we did not implement antialiasing for the procedural textures, but it would be straightforward to compute the Jacobian relating texture and screen space and use this to control the number of octaves of procedural noise used to generate the texture.

Antialiasing procedural textures applied to procedural surfaces requires computing the mapping from texture space to screen space. These volume textures are not parameterized by the procedural surface parameterizing variables u, v so we must reparameterize the surface to be consistent with the procedural textures. Because the texture functions are roughly isotropic any orthonormal parametrizing basis for the surface will do. We have already derived such a basis, $\hat{u}, \hat{v}$. The differential mapping from the procedural surface to screen space is then given by

$$\Delta \boldsymbol{x} = \mathbf{D}_{proj}\mathbf{D}_{\hat{s}}\Delta\hat{\boldsymbol{u}}$$

where

$$\mathbf{D}_{proj} = \begin{bmatrix} \frac{\partial x}{\partial x_w} & \frac{\partial x}{\partial y_w} & \frac{\partial x}{\partial z_w} \\ \frac{\partial y}{\partial x_w} & \frac{\partial y}{\partial y_w} & \frac{\partial y}{\partial z_w} \end{bmatrix}$$

is the derivative of the world to screen space projection matrix, easily computed using symbolic differentiation.

The vector $[x_w, y_w, z_w]^T$ is the position of a point on the procedural surface, in world space. For polygonal surfaces the matrix $\mathbf{D}_{\hat{s}}$ is formed from the orthornormal surface basis:

$$\mathbf{D}_{\hat{s}} = [\hat{u}, \hat{v}]$$

The surface to screen mapping matrix

$$\mathbf{D}_{scr} = \mathbf{D}_{proj}\mathbf{D}_{\hat{s}}$$

is 2 by 2. The stretching, or shrinking, of the texture on the screen can be approximated by computing the SVD of this matrix. We use Jacobi iteration, which for 2 by 2 matrices is guaranteed to converge to a solution in a single iteration. It is also numerically well conditioned and relatively efficient[14]. One could also directly solve the characteristic equation

$$\begin{aligned} \text{Det}\left[\mathbf{D}_{scr}^T\mathbf{D}_{scr} - \boldsymbol{\lambda}\mathbf{I}\right] &= 0 \\ \text{Det}\begin{bmatrix} a_{11}-\lambda_i & a_{12} \\ a_{21} & a_{22}-\lambda_i \end{bmatrix} &= 0 \end{aligned}$$

which gives a quadratic equation in λ_i

[14]The code for Jacobi iteration is included in the supplementary materials.

(a) *Displacement surface* **(b)** *Fractal Voronoi* **(c)** *Lava Planet*

Figure 7: *Other procedural textures generated using symbolic derivatives. (a) is a true displacement surface. (b), (c) are both normal mapped.*

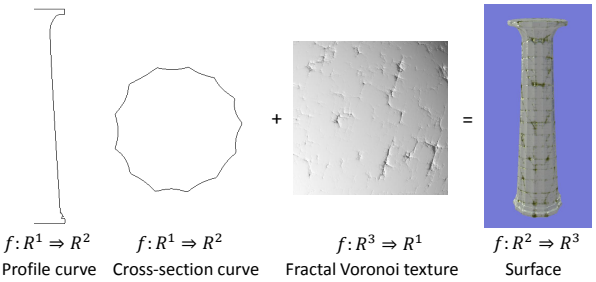


Figure 8: *Fractal Voronoi texture applied to procedurally generated surface*

$$(a_{11}-\lambda_i)(a_{22}-\lambda_i) - a_{21}a_{12} = 0$$

This will take slightly less computation but squares the condition number of $\mathbf{D}_{scr}$. For typical graphics purposes this probably has acceptable numerical properties.

Singular values less than 1 represent a shrinking of the texture, and consequently an increase in spatial frequencies present on the screen. Values greater than 1 represent a stretching of the texture. We find the minimum singular value and use this to compute a scale factor α which is applied to the texture normal offset equation

$$\boldsymbol{g}(u,v) = \boldsymbol{s}(u,v) + \alpha\boldsymbol{n}_s(u,v)t(\boldsymbol{s}(u,v))$$

for true displacement surfaces or

$$\boldsymbol{n}_g = \boldsymbol{n}_s + \alpha\left(t\frac{\partial \boldsymbol{n}_s}{\partial u} + \boldsymbol{n}_s\frac{\partial t}{\partial u}\right) \times \left(t\frac{\partial \boldsymbol{n}_s}{\partial v} + \boldsymbol{n}_s\frac{\partial t}{\partial v}\right)$$

for normal mapped surfaces.

7 Results and Discussion

All timings reported in this section were performed on an Intel(R) Xeon(R) CPU E5640 @ 2.67GHz, 4 Cores with an ATI Radeon HD 5870 GPU, 1024MB GDDR5, core clock 850MHz, memory clock 1200MHz .

The procedural surfaces render at high rates, measured in triangles/sec., especially at high triangulation counts. Table 1 shows the triangle rendering rates for a procedural lamp object. The rendering computes diffuse lighting, with the lamp object filling a 640x480

Triangles	Tri/sec.
$200*10^3$	$540*10^6$
$20*10^3$	$333*10^6$
$2*10^3$	$50*10^6$
576	$19.2*10^6$

Table 1: *Rendering speed of procedurally defined lamp object at various triangulation levels. Triangle rendering speeds do not include the time required to run the render loop setup which consists of: screen clear, control and text display.*

Object	Only surface computations	Surface + normal computations	increase
sphere	36	47	17%
lamp	79	96	30%
doric column	1454	2221	54%

Table 2: *Computation required by the symbolic differentiation to compute the exact surface normal. The doric column has the largest increase because of the large number of computations required to compute the procedural texture offset.*

window, using the symbolic derivative to compute the normal at every pixel.

The overhead associated with calling the shaders for the procedural surfaces (each object is one shader call) doesn't become extreme until the number of objects is quite large. Fig. 4 shows a still from an animation with 2500 procedural objects, $\frac{1}{3}$ of which have procedural textures. We did not implement bounding volume culling, an obvious efficiency improvement, so all 2500 objects are drawn each frame. Even with this large number of objects the update rate was 39 frames per second with $1.9 * 10^6$ triangles per frame, or $73 * 10^6$ triangles/sec. For a scene with a 100 object grid (not illustrated) the update rate at 3,300,000 tris/frame was 43Hz, or 142 million tris/second.

In general the cost of computing derivatives is quite small. In Table 2 we show the incremental cost of computing the exact symbolic surface normal for three different procedural surfaces. The largest increase is only about 50%, which occurs only for the doric column object. This object has a very complex procedural texture, which is counted in the normal computations but not in the surface computations.

Overall the system performs as we had hoped: derivatives are easy to specify and the resulting derivative code is very efficient. However, there is one thing we hope to change in future versions. When we designed the system the assumption was that derivative assignment would be uncommon, so the interaction between derivative assignment and taking derivatives through functions would not be burdensome to the programmer. In real world shaders we discovered that the functions `floor`, `ceil`, `frac`, which do not have default derivative rules, occurred much more frequently than we supposed. Derivative assignment had to be used in many shaders, which required all those shaders to have an additional derivative variable argument. In future versions we hope to provide default derivative definitions for `floor`, `ceil`, `frac` which will eliminate this problem if no other derivative assignments are used. We also may ease the restriction on taking derivatives with respect to interior nodes, which would allow derivative assignment to be used inside functions without requiring an extra derivative variable.

8 Conclusion and Future Work

The new symbolic differentiation language feature now makes it possible to easily and efficiently compute derivatives of complex functions in GPU shaders. The derivatives are generally quite efficient; in the example applications the symbolic derivative capability improved the quality of texture rendering, with no loss in speed relative to hardware finite differencing, and allowed for the rendering of procedural surfaces at very high speed. Many other applications will also benefit from this new feature.

In the future we may add variable dependency tracking, which should eliminate the need to pass differentiation variables between functions. We may also implement the full D* algorithm [Guenter 2007] if the domain dimension of typical functions increases substantially.

References

BISCHOF, C., ROH, L., AND MAUER-OATS, A. 1997. Adic: An extensible automatic differentiation tool for ansi-c. *Tech report 27*, 1427–1456.

GUENTER, B. 2007. Efficient symbolic differentiation algorithm for graphics applications. *ACM Computer Graphics 27*, 3, 108.

HANRAHAN, P. 2009. Domain-specific languages for heterogenous gpu computing. *NVIDIA Technology Conference*.

SNYDER, J. 1992. *Generative Modeling for Computer Graphics and CAD*. Academic Press.

UPSTILL, S. 1989. *The Renderman Companion*. Addison Wesley.