SMASH

A Next-Generation API for Programmable Graphics Accelerators

Michael D. McCool

Computer Graphics Lab

Department of Computer Science
University of Waterloo
200 University Avenue West
Waterloo, Ontario, Canada N2L 3G1

mmccool@cgl.uwaterloo.ca
http://www.cgl.uwaterloo.ca/

Abstract

SMASH is a testbed for low-level graphics API concepts and is meant to act as concrete design target for the development of extensions to OpenGL. Specifically, SMASH is syntactically and conceptually similar to OpenGL but, along with other experimental features, supports a programmable shader sub-API that is compatible with both multi-pass and single-pass implementations of shaders.

Arbitrary numbers of shader parameters of various types can be bound to vertices of geometric primitives using a simple immediate-mode mechanism. Run-time specification, manipulation, and compilation of shaders at various levels of resolution is supported, including integrated support for per-primitive, per-vertex and per-fragment shaders.

Implementation of rendering effects using SMASH can be enhanced with metaprogramming toolkits and techniques, up to and including RenderMan-like shading languages. We give examples of a two-term separable BRDF approximation.

SMASH

Simple **M**odelling **A**nd **SH**ading API:

What: API design for next-generation graphics systems.

Motivation:

- Develop algorithms for *future* hardware.
- See the development of powerful, flexible, and appropriate accelerator capabilities.
- Have a simple, elegant, powerful, portable, easy-to-use API.

Approach

- Conservatively radical.
- Hardware/software codesign:
  evaluate limitations and explore opportunities.
- Intentionally "academic":
  not *too* constrained by backward compatibility.

Focus and Status

- Programmable shaders and geometry.
- Conceptual model of programmable pipeline.
- Metaprogramming API.
- Work in progress…

Outline

That was the motivation.

Now, will cover the design itself:

- Conceptual architecture.
- Implementation options.
- API and Examples.

Programmable Shading

Multiple ways to implement programmable shading:

- Pure multipass.
- Multipass with multitexturing.
- Multipass with register combiners.
- Multipass with vertex shaders and dependent register combiners.

.
.
.
.
.

- Single-pass programmable shading (with fallback to multipass).

Single-Pass Conceptual Model

Advantages of single-pass conceptual model:

- Single-pass shading programs can be unrolled to multipass.
- Inferring single-pass shader from multipass much harder.
- All relevant information in one place.

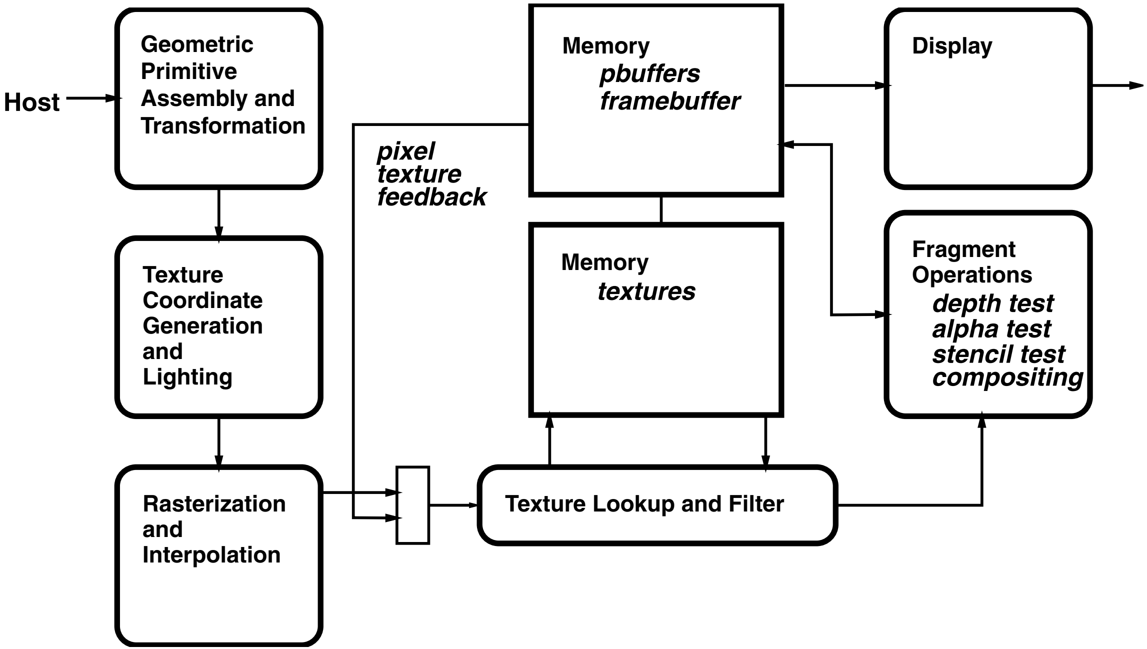
Single-Pass Implementation

Advantages of single-pass implementation:

- Potentially more efficient for bandwidth-limited graphics accelerators.
- Lower memory requirements.
- Can more easily handle high-precision computation.
- Can trade off hardware utilization requirements.

Hardware Architectures

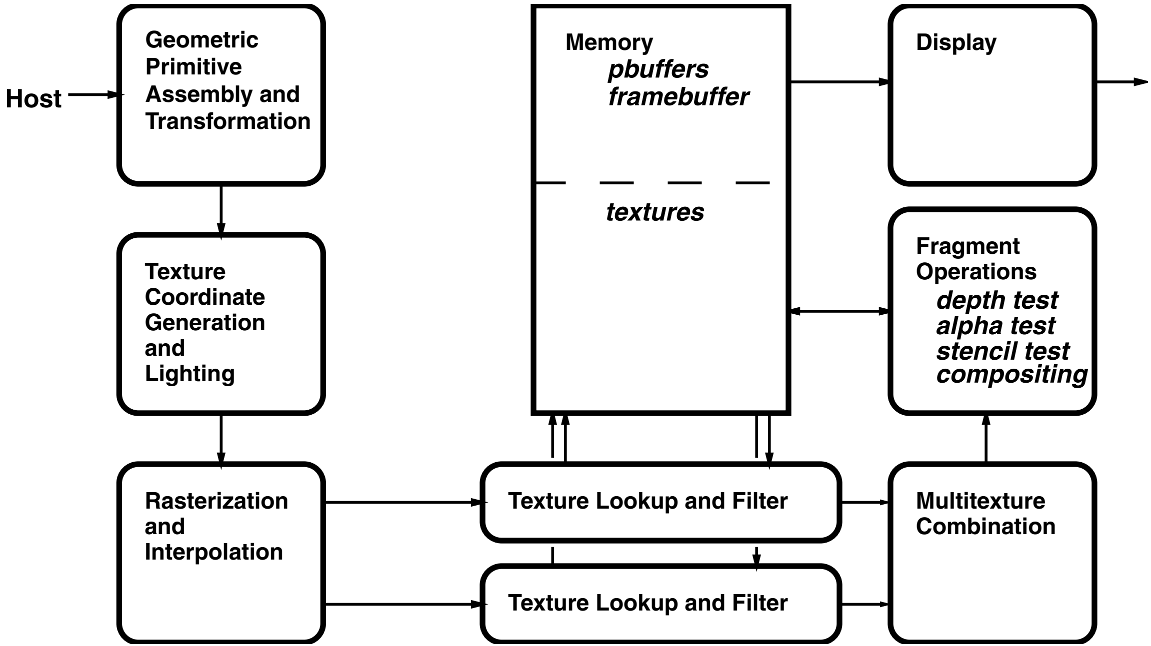
OpenGL 1.2.1 . . . On SGIs



Back
Close

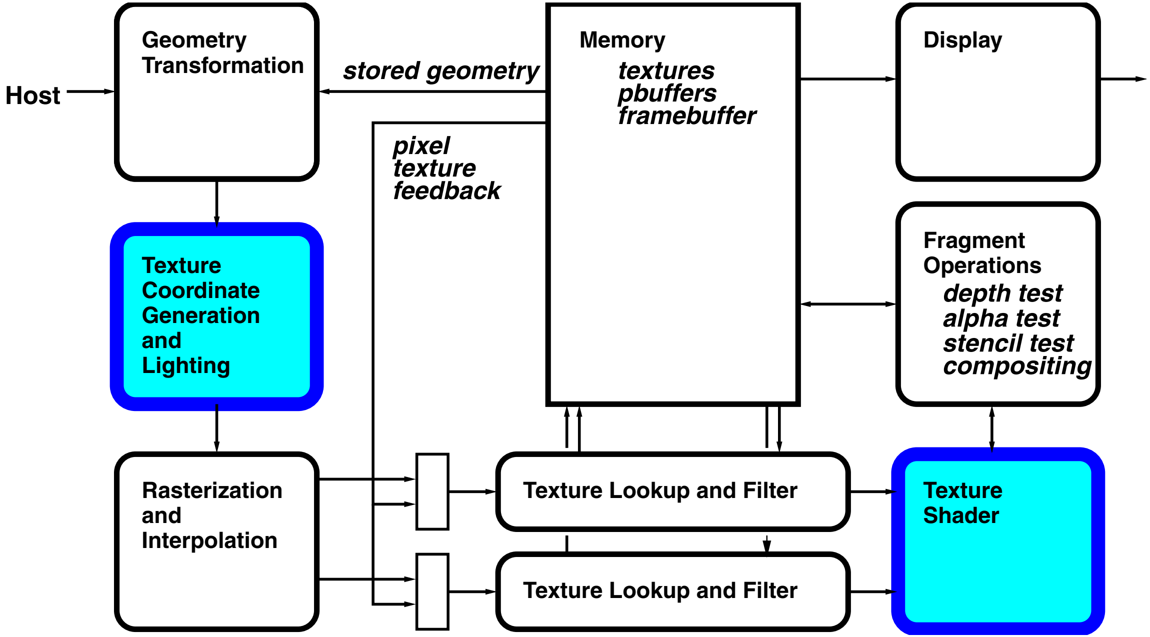
Hardware Architectures

OpenGL 1.2.1 . . . On PCs



Hardware Architectures

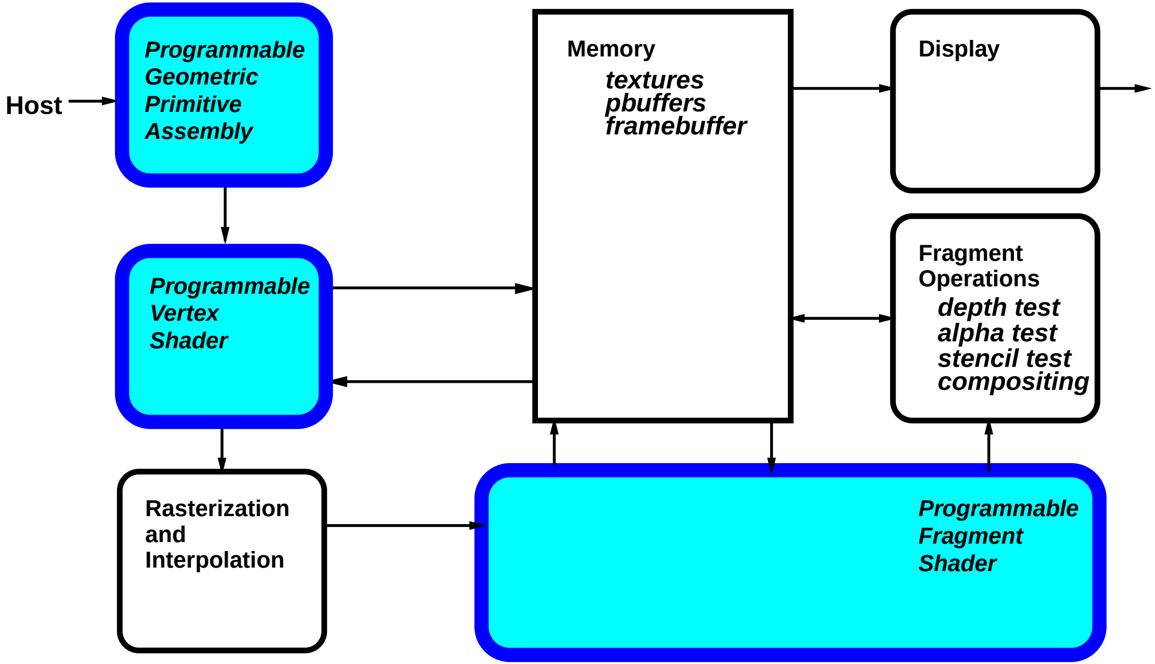
Proposed Architecture: Texture Shaders



Back
Close

Hardware Architectures

Proposed Architecture: SMASH



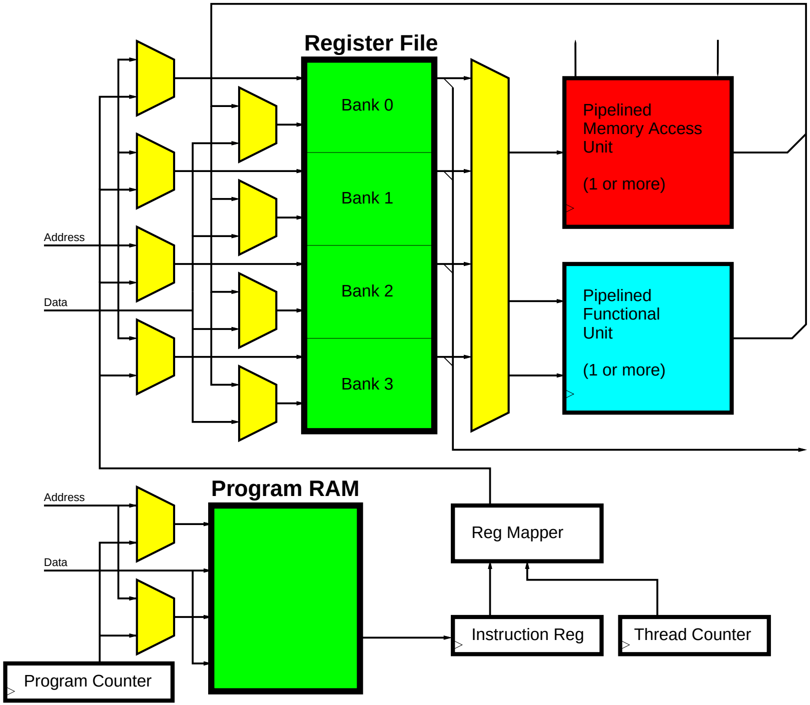
SMASH Architecture

- Programmable vertex and fragment shaders.
- *Same operations* can be applied in both shaders:
  - Arithmetic operations
  - Texture lookup
- Outputs colour and alpha (but *not* depth).
- Can discard fragments.
- Can use framebuffer as input.
- Rasterizer: hyperbolic interpolation of variable-length sequence of numbers: *shader parameters*
- Variety of texture formats (including pyramidal and sparse)

Implementation Options

- Optimized software on standard processors:
  - SIMD parallelism.
  - MIMD parallelism.
  - On-the-fly machine language generation.
- Specialized hardware:
  - Parallel multithreaded processors.
  - Reconfigurable pipelined array processor.
  - Stream processor.
  - Others . . .

Multithreaded Processor



Back
Close

Multithreading: Coherence

- Depends on temporal coherence of shading programs.
- To maximize efficiency, coherent “clusters” of shading evaluations need to be assembled.
- To ensure output completes in order:
  - No data-dependent loops.
  - No conditional execution.

Multithreading: Memory

Shader with small memory requirements/computational cost:

- many threads (64 to 256)
- small part of register file allocated to each
- number of threads greater than texture unit pipeline delay
- no scheduling required

Multithreading: Memory

Shader with large memory requirements/computational cost:

- fewer threads
- large part of register file allocated to each
- number of threads less than pipeline delay
- explicit scheduling among many operations

Multithreading: Utilization

Performance:

- multiple parallel shaders
- possibly running different shader programs
- timing/resynchronization issues

Storage and reloading of shader program:

- processor only fetches instruction every n clocks
- can share instruction memory among cluster

Register file:

- limits complexity of shader
- make large: doubles as fragment packet FIFO.

API: Parameter Binding

- Parameters transformed and normalized by type: `smParam*`, `smColor*`, `smVector*`, `smTangent*`, `smPoint*`, `smCovector*`, `smNormal*`, `smPlane*`, `smTexCoord*`.
- Pushed onto *parameter stack*.
- Outside `smBegin`/`smEnd`: per-primitive constant.
- Inside `smBegin`/`smEnd`: per-vertex interpolated.
- `smVertex*` call makes copy of stack, resets stack to state at prior `smBegin`.

API: Parameter Binding

```
// per-primitive parameters
smVector3d(100.0, 10.5, 15.6);
smTangent3d(0.0, -0.5, 0.5);
smBegin(SM_TRIANGLES);
    // per-vertex parameters
    smColor3d(0.3, 0.5, 0.2);
    smNormal3d(0.7, 0.5, 0.5);
    smVertex3d(0.0, 0.0, 0.0);

    // per-vertex parameters
    smColor3d(0.4, 0.6, 0.1);
    smNormal3d(0.5, 0.7, 0.5);
    smVertex3d(0.0, 1.0, 0.5);

    // per-vertex parameters
    smColor3d(0.3, 0.4, 0.1);
    smNormal3d(0.5, 0.5, 0.7);
    smVertex3d(1.0, 1.0, 1.5);
smEnd();
```

API: Parameter Binding

```
// per-primitive parameters
smVector3d(100.0, 10.5, 15.6);
smPushMatrix();
    smRotated(30.0, 0.0, 1.0, 0.0);
    smTangent3d(1.0, 0.0, 0.0);
smPopMatrix();
smBegin(SM_TRIANGLES);
    // per-vertex parameters
    smColor3d(0.3, 0.5, 0.2);
    smNormal3d(0.7, 0.5, 0.5);
    smVertex3d(0.0, 0.0, 0.0);

    // per-vertex parameters
    smColor3d(0.4, 0.6, 0.1);
    smNormal3d(0.5, 0.7, 0.5);
    smVertex3d(0.0, 1.0, 0.5);

    ⋮
```

API: Shader Specification

- Shader definition opened with `smBeginShader`.
- Shader API calls add declarations and instructions to open shader definition.
- Shader expressed using stack/register virtual machine (FORTH-like).
- Stack and registers hold n-tuples.
- Shader definition closed and compiled with `smEndShader`.
- Shader activated with `smShader`.
- Shader prefetch hint with `smNextShader(s)`.

Back
Close

API: Shader Specification

```
SMshader sepbrdf = smBeginShader();
{
    // declare parameters
    SMparam L = smShaderDeclareVector(3);
    SMparam T = smShaderDeclareTangent(3);
    SMparam C = smShaderDeclareColor(3);
    SMparam N = smShaderDeclareNormal(3);

    // Allocate and name registers
    SMreg p = smShaderAllocReg(3);
    SMreg q = smShaderAllocReg(3);

    ⋮
```

API: Shader Specification

```
// Compute tangent-space coordinates
// Specify may be computed at vertices
smBeginSubShader(SM_VERTEX);
  smShaderGetNormal(N);       // $\hat{n}$
  smShaderGetTangent(T);      // $\hat{n}, \vec{t}$
  smShaderGetVector(L);       // $\hat{n}, \vec{t}, \hat{l}$
  HalfDiffSurfaceCoords();    // $\hat{p}, \vec{q}$
smEndSubShader();

// Put interpolated surface coordinates in registers
smShaderStore(q);                 // $\hat{p}$
smShaderStoreCopy(p);             // $\hat{p}$
⋮
```

API: Shader Specification

```
// Compute BRDF (2-term separable approx)
smShaderLookup(a);           // a[p̂]
smShaderLoad(q);             // a[p̂], q⃗
smShaderLookup(b);           // a[p̂], b[q⃗]
smShaderMult();              // ab = a[p̂] * b[q⃗]
smShaderColor3dv(aAB);       // ab, α
smShaderMult();              // AB = ab * α
smShaderLoad(p);             // AB, p̂
smShaderLookup(c);           // AB, c[p̂]
smShaderColor3dv(bC);        // AB, c[p̂], β1
smShaderAdd();               // AB, bc = c[p̂] + β1
smShaderLoad(q);             // AB, bc, q⃗
smShaderLookup(d);           // AB, bc, d[q⃗]
smShaderColor3dv(bD);        // AB, bc, d[q⃗], β2
smShaderAdd();               // AB, bc, bd = d[q⃗] + β2
smShaderMult();              // AB, bcd = bc * bd
smShaderColor3dv(aCD);       // AB, bcd, γ
smShaderMult();              // AB, CD = bcd * γ
smShaderAdd();               // f = AB + CD
⋮
```

API: Shader Specification

```
    // Compute irradiance and multiply by BRDF
    smBeginSubShader(SM_VERTEX);
      smShaderGetVector(L);         // f, l̂
      smShaderGetNormal(N);         // f, l̂, n̂
      smShaderDot(N);               // f, (l̂·n̂)
      smShaderParam1d(0);           // f, (l̂·n̂), 0
      smShaderMax();                // f, s = max((l̂·n̂), 0)
      smShaderGetColor(C);          // f, s, c
      smShaderMult();               // f, e = s * c
    smEndSubShader();

    smShaderMult();                 // f * e

    // Set output fragment color
    smSetColor();                   // <empty>
} smEndShader();
```

API: Activating Shaders

```
// Storage
//    allocate storage for program
SMsizei s = smShaderSize(leaves);
SMubyte* leaves_prog = (SMubyte*)malloc(s);
smGetShaderProg(leaves_prog,leaves);
//    now store program in file...

...

// Retreival
//     get program from file, then...
smSetShaderProg(leaves,leaves_prog);

...

// Activation
smShader(bark);           // Make active
smNextShader(leaves);     // Preload hint
```

API: Metaprogramming

- Shader specialization.
- Adapt program complexity.
- Higher-level toolkits:
  - Macros (can hide platform dependencies/provide hooks)
  - Partial or full textual shading languages (i.e. RenderMan SL)
  - C++ operator overloading

API: Macros

```
void
HalfDiffSurfaceCoords () // n̂, u⃗, l̂
{
  // Save register allocation state
  smShaderBeginBlock(); {

    // Allocate and name registers
    SMreg h = smShaderAllocReg(3);
    SMreg t = smShaderAllocReg(3);
    SMreg s = smShaderAllocReg(3);
    SMreg n = smShaderAllocReg(3);
    SMreg tp = smShaderAllocReg(3);
    ⋮
```

API: Macros

// Compute normalized half vector $\hat{h}$

`smShaderGetViewVec();` // $\hat{n},\ \vec{u},\ \hat{l},\ \vec{v}$
`smShaderNorm();` // $\hat{n},\ \vec{u},\ \hat{l},\ \hat{v}$
`smShaderAdd();` // $\hat{n},\ \vec{u},\ \vec{h} = \hat{l} + \hat{v}$
`smShaderNorm();` // $\hat{n},\ \vec{u},\ \hat{h}$
`smShaderStore(h);` // $\hat{n},\ \vec{u}$

// Generate full surface frame from $\hat{n}$ *and* $\vec{u}$

`smShaderSwap();` // $\vec{u},\ \hat{n}$
`smShaderStoreCopy(n);` // $\vec{u},\ \hat{n}$
`Orthonormalize();` // $\hat{t}$
`smShaderStoreCopy(t);` // $\hat{t}$
`smShaderLoad(n);` // $\hat{t},\ \hat{n}$
`smShaderSwap();` // $\hat{n},\ \hat{t}$
`smShaderCross();` // $\hat{s} = (\hat{n} \times \hat{t})$
`smShaderStore(s);` // <empty>
⋮

API: Macros

```
// Orthonormalize $\hat{t}$ against $\hat{h}$
smShaderLoad(t);        // $\hat{t}$
smShaderLoad(h);        // $\hat{t}$, $\hat{h}$
Orthonormalize();       // $\hat{t}'$
smShaderStore(tp);      // <empty>

// Coordinates of $\hat{h}$ relative to $(\hat{t}, \hat{s}, \hat{n})$
smShaderLoad(t);        // $\hat{t}$
smShaderLoad(s);        // $\hat{t}$, $\hat{s}$
smShaderLoad(n);        // $\hat{t}$, $\hat{s}$, $\hat{n}$
smShaderLoad(h);        // $\hat{t}$, $\hat{s}$, $\hat{n}$, $\hat{h}$
FrameVector();          // $\hat{p} = ((\hat{t} \cdot \hat{h}), (\hat{s} \cdot \hat{h}), (\hat{n} \cdot \hat{h}))$
⋮
```

Back
Close

API: Macros

```
    // Coordinates of v⃗ relative to (t̂′, ŝ′, ĥ)
    smShaderLoad(tp);       // p̂, t̂′
    smShaderLoad(h);        // p̂, t̂′, ĥ
    smShaderDup(1);         // p̂, t̂′, ĥ, t̂′
    smShaderCross();        // p̂, t̂′, ŝ′ = (ĥ×t̂′))
    smShaderLoad(h);        // p̂, t̂′, ŝ′, ĥ
    smShaderGetViewVec();   // p̂, t̂′, ŝ′, ĥ, v⃗
    FrameVector();          // p̂, q⃗ = ((v⃗·t̂′), (v⃗·ŝ′), (v⃗·ĥ))

  // Release registers
  } smShaderEndBlock();
} // p̂, q⃗
```

Back
Close

API: Textual Shading Language

```
SMshader sepbrdf = smBeginShader();
{
    // allocate and name parameters
    SMparam C = smuShaderDeclareColor(3,"C");
    SMparam L = smuShaderDeclareVector(3,"L");
    SMparam T = smuShaderDeclareTangent(3,"T");
    SMparam N = smuShaderDeclareNormal(3,"N");

    // Allocate and name registers
    SMreg p = smuShaderAllocReg(3,"p");
    SMreg q = smuShaderAllocReg(3,"q");
    ⋮
```

API: Textual Shading Language

```
// Compute surface coordinates using macro
smBeginSubShader(SM_VERTEX);
    smShaderGetNormal(N);
    smShaderGetTangent(T);
    smShaderGetVector(L);
    HalfDiffSurfaceCoords();
smEndSubShader();

// Put interpolated surface coordinates in registers
smShaderStore(q);

// Another (silly) way of doing the above
smuShaderExpr("p = %0");
⋮
```

Back
Close

API: Textual Shading Language

```
    // Compute BRDF
    smShaderColor3dv(aAB);
    smuShaderExpr("%0*a[p]*b[q]");
    smShaderColor3dv(bD);
    smShaderColor3dv(bC);
    smShaderColor3dv(aCD);
    smuShaderExpr("%0*(c[p]+%1)*(d[q]+%2)");

    // Compute irradiance and multiply by BRDF
    smBeginSubShader(SM_VERTEX);
        smuShaderExpr("C*max((L|N),0)");
    smEndSubShader();

    smShaderMult();

    // Set output fragment color
    smSetColor();
} smEndShader();
```

API: C++ Toolkit

```
Shader sepbrdf = beginshader();
{
    // allocate and name parameters
    Color C(3);
    Vector L;
    Tangent T;
    Normal N;

    // wrap constants and textures
    // (normally would do at point of definition)
    ConstColor alpha(3,aBC);
    ConstColor beta1(3,bC);
    ConstColor beta2(3,bD);
    ConstColor gamma(3,aCD);
    Texture A(a);
    Texture B(b);
    Texture C(c);
    Texture D(d);
    ⋮
```

API: C++ Toolkit

```
    // Compute surface coordinates using macro
    Expr<VERTEX> p, q;
    HalfDiffSurfaceCoords(p,q,N,L,T);

    // Compute BRDF
    Expr f = alpha*A[p]*B[q] + gamma*(C[p]+beta1)*(D[q]+beta2);

    // Compute irradiance and multiply by BRDF
    Expr<VERTEX> e = C*max((L|N),0.0);
    Expr fe = f*e;

    // Emit shader instructions
    fe->compile();

    // Set output fragment color
    setcolor();
} endshader();
```

Prototype

- In *process* of building prototype implementations:
  - Software implementation.
  - Mapping onto standard graphics accelerators (multipass).
  - Hardware simulation model (Handel-C/VHDL/SystemC)
  - Xilinx FPGA hardware implementation (Handel-C/VHDL/SystemC).

Future Work

- Displacement mapping with adaptive tesselation.
- Subdivision surfaces.
- Geometry decompression.
- Programmable geometric primitive assembly.
- Programmable image processing.
- Display list management and occlusion culling.
- Test cases and architectural analysis.

Conclusions

- Programmability a fundamental change.
- Internal vs. external programmability.
- Many "legacy" features in current graphics systems.
- Need for scalability and portability.
- Need for goal-driven as well as evolutionary change.
- Need for standards.
- **SMASH**:
  Hardware/software codesign of programmable graphics system. Testbed for ideas, *not* a production system.
- Research:
  Design algorithms for systems available in *future*. . .