Shader Metaprogramming

Michael D. McCool
Zheng Qin
Tiberiu S. Popa

Computer Graphics Lab
University of Waterloo

Outline

- Goals and motivation
- Related work
- Testbed architecture
- Expression parsing
- Modularity, types, specialization
- Control constructs
- Conclusions

Goals and Motivation

- Graphics hardware has programmable features
- Assembly-language interface too low-level
- String-based interface inconvenient for “coprocessor” applications
- In C++, can use operator overloading to help build ***inline*** shading language

Related Work

- Renderman shading language
- NVIDIA and ATI vertex and fragment shader extensions
- DX9 shading language
- OpenGL 2.0 proposal
- NVIDIA's Cg language
- Stanford's shading language compiler
- SGI's ISL compiler

Testbed Architecture

- Used SMASH (actually, Sm) as initial compiler target
- Basically DX9 assembly language plus
  - noise functions
  - jumps
  - conditional branches
- Function-call based API so machine code can be generated on the fly
- Infinite register model (virtual machine)
- Sm as intermediate language?

Virtual Machine API

- Explicit allocation (and deallocation) of registers
- Function call per instruction
- Swizzling, negation using function calls
- Labels declared with calls

```
smBeginShader(0);
  SMreg a = smAllocInputReg(3);
  SMreg b = smAllocInputReg(3);
  Smreg c = smAllocOutputReg(3);
  smBLT(0,a,b);
  smSUB(c,a,b);
  smJ(1);
  smLBL(0);
  smSUB(c,b,a);
  smLBL(1);
smEndShader();
```

Example 1: Wood

```
ShMatrix3x4f modelview;
ShMatrix4x4f perspective;
ShPoint3f light_position;
ShColor3f light_color;
ShAttrib1f phong_exp;
ShMatrix4x4f quadric_coefficients;
ShAttrib4f pnm_alpha;
ShTexture1DColor3f pnm_cd, pnm_cs;

ShShader wood0 = SH_BEGIN_SHADER(0) {
  ShInputNormal3f nm;
  ShInputPoint3f pm;
  ShOutputPoint4f ax, x(pm);
  ShOutputVector3f hv;
  ShOutputNormal3f nv;
  ShOutputColor3f ec;
  ShOutputPoint4f pd;
  ShPoint3f pv = modelview | pm;
  pd = perspective | pv;
  nv = normalize(nm | adj(modelview));
  ShVector3f lvv = light_position - pv;
  ShAttrib1f rsq = 1.0/(lvv|lvv);
  lvv *= sqrt(rsq);
  ShAttrib1f ct = max(0,(nv|lvv));
  ec = light_color * rsq * ct;
  ShVector3f vvv =
    -normalize(ShVector3f(pv));
  hv = normalize(lvv + vvv);
  ax = quadric_coefficients | x;
} SH_END_SHADER

ShShader wood1 = SH_BEGIN_SHADER(1) {
  ShInputPoint4f ax, x;
  ShInputVector3f hv;
  ShInputNormal3f nv;
  ShInputColor3f ec;
  ShInputAttrib1f pdz;
  ShInputAttrib2us pdxy;
  ShOutputColor3f fc;
  ShOutputAttrib1f fpdz(pdz);
  ShOutputAttrib2us fpdxy(pdxy);
  ShTexCoord1f u = (x|ax) +
    noise(pnm_alpha,x);
  fc = pnm_cd[u] + pnm_cs[u] *
    pow((normalize(hv)|normalize(nv)),
    phong_exp);
  fc *= ec;
} SH_END_SHADER
```

Global (Uniform) Parameters

```
ShMatrix3x4f modelview;
ShMatrix4x4f perspective;

ShPoint3f light_position;
ShColor3f light_color;
ShAttrib1f phong_exp;

ShMatrix4x4f quadric_coefficients;
ShAttrib4f pnm_alpha;
ShTexture1DColor3f pnm_cd, pnm_cs;
```

Vertex Shader I/O Attributes

```
ShInputNormal3f nm;
ShInputPoint3f pm;

ShOutputPoint4f ax, x(pm);
ShOutputVector3f hv;
ShOutputNormal3f nv;
ShOutputColor3f ec;
ShOutputPoint4f pd;
```

Vertex Computation

```
ShPoint3f pv = modelview | pm;
pd = perspective | pv;
nv = normalize(nm | adj(modelview));

ShVector3f lvv = light_position - pv;
ShAttrib1f rsq = 1.0/(lvv|lvv);
lvv *= sqrt(rsq);
ShAttrib1f ct = max(0,(nv|lvv));
ec = light_color * rsq * ct;
ShVector3f vvv = -normalize(ShVector3f(pv));
hv = normalize(lvv + vvv);

ax = quadric_coefficients | x;
```

Vertex Computation (alt)

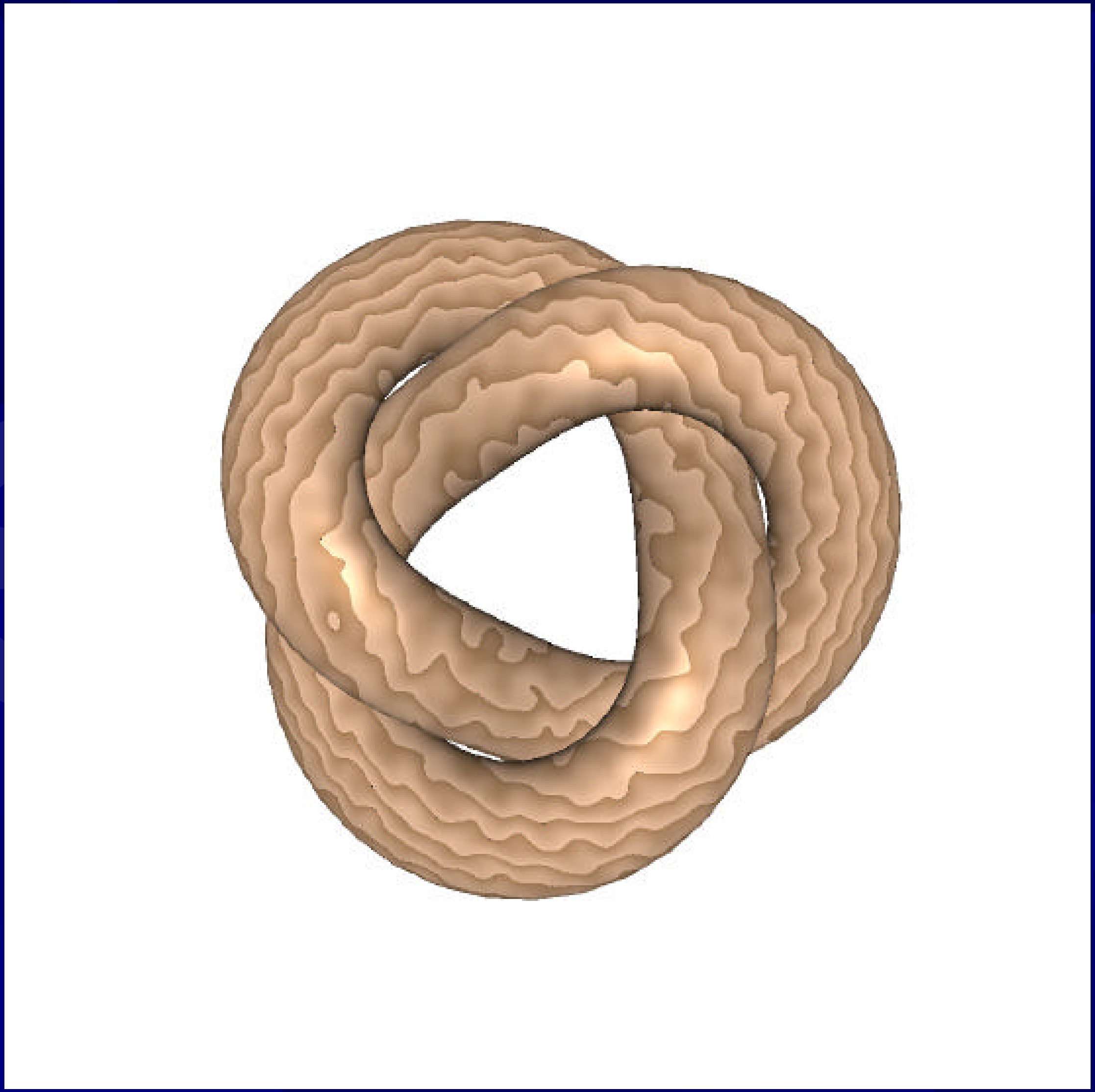
```
ShPoint3f pv;
transform(
   pd, pv,
   pm
);

blinn_phong0(
   ec, hv,
   nv, pv, light_position, light_color
);

ax = quadric_coefficients | x;
```

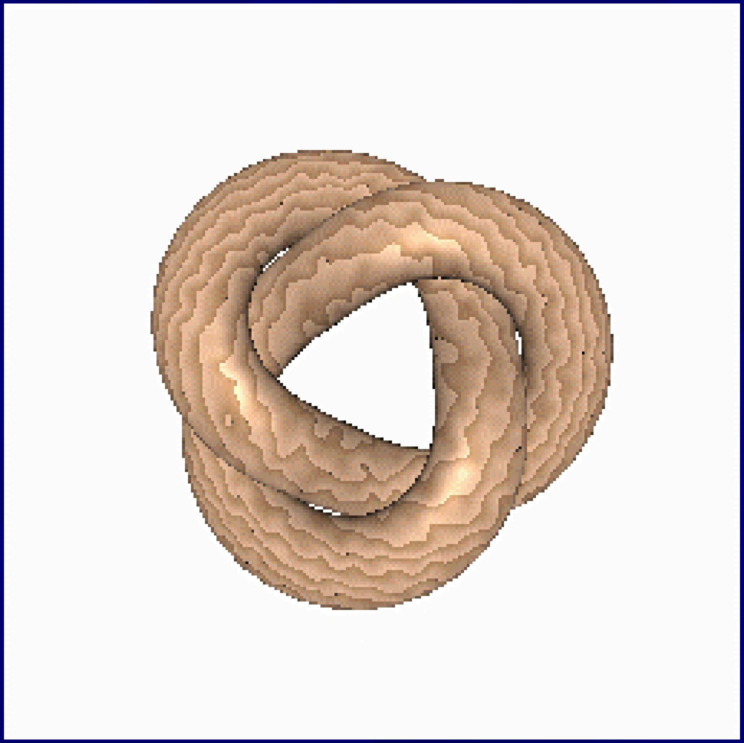
Fragment I/O Attributes

```
ShInputPoint4f ax, x;
ShInputVector3f hv;
ShInputNormal3f nv;
ShInputColor3f ec;
ShInputAttrib1f pdz;
ShInputAttrib2us pdxy;

ShOutputColor3f fc;
ShOutputAttrib1f fpdz(pdz);
ShOutputAttrib2us fpdxy(pdxy);
```

Fragment Computation

```
ShTexCoord1f u = (x|ax)
               + noise(pnm_alpha,x);

fc = pnm_cd[u] + pnm_cs[u] *
    pow((normalize(hv)|normalize(nv)),
        phong_exp);

fc *= ec;
```





Parsing

- Expressions
  - Use operator overloading to build parse trees for expressions
- Control constructs
  - Use calls to insert control keywords into token stream
  - Recursive descent parser parses token stream when shader complete

Expressions

- ***Shader variables***: reference-counting ``smart pointers'' to expression parse tree nodes
- ***Operators on variables***: generate new nodes that point to nodes of inputs, return smart pointers to new nodes
- ***Assignment statement***: adds assignment statement token to shader which refers to expression parse trees

Types

- ShAttrib[1234]f

- ShVector[1234]f
- ShNormal[1234]f
- ShPoint[1234]f
- ShPlane[1234]f
- ShColor[1234]f
- ShTexCoord[1234]f

- ShTexture[123]D*
- ShTextureCube*

- ShMatrix[1234]x[1234]f

- ShInput*
- ShOutput*

Arithmetic Operators

- **+**, **-**, *****, **/**: act on all values componentwise
- **|** is the matrix multiplication operator
  - *tuple***|***tuple*: dot product
  - *matrix***|***tuple*: tuple is column vector
  - *tuple***|***matrix*: tuple is row vector
  - *matrix***|***matrix*: matrix multiplication
  - Special rules for size promotion to handle homogenous coordinates, affine xforms
- **&** is cross product operator

GRAPHICS HARDWARE 2002

Access Operators

- [ ] is texture and array access operator
  - c = t[u]
- ( ) is swizzling and writemask operator
  - c(0,1,2) = c(2,1,0)
- [ ] on one component is equivalent to ( ) on one component
  - m01 = m[0][1] = m[0](1)

Attributes

- Attached to vertices and fragments
- Ex: vertex normals, fragment (interpolated) texture coordinates
- Declared as inputs and outputs in each shader program
- Binding given by order and type, not name

Parameters

- Use *same* types for declaration as attributes
- Considered "uniform" if declared *outside* shader definition
- May only be modified outside shader
- Loaded into constant registers when:
  - Shader that uses them is loaded, *and*
  - When they are modified by host program
- Simulate semantics of "global variables"

Modularity

- Classes and functions can be used to organize (parts of) shaders
- Functions in the host language can be used as “macros” for the shading language
- Classes that create shaders when instantiated can be used to construct specialized shader instances

Types

- Types declared in C++ act as types in shading language
- Type checking within a shader happens at compile time of application program
- Library supports types to abstract textures, matrices, points, vectors, etc.
- User can subclass these, or put in classes or structs as members

GRAPHICS HARDWARE 2002

Control Constructs

- Calls to add keywords to token stream of open shader definition:

  shIF(*expr***)**;

  shWHILE(*expr***)**;

  shELSE();

  shENDWHILE();

  shENDIF();

Control Constructs

- Use macros to hide extra punctuation:

```
#define SH_IF(expr)        shIF(expr);
#define SH_WHILE(expr)  shWHILE(expr);
#define SH_ELSE            shELSE();
#define SH_ENDWHILE        shENDWHILE();
#define SH_ENDIF           shENDIF();
```

- When shader complete, use recursive-descent parser to complete generation of parse tree

GRAPHICS HARDWARE 2002

Example 2: Julia Set

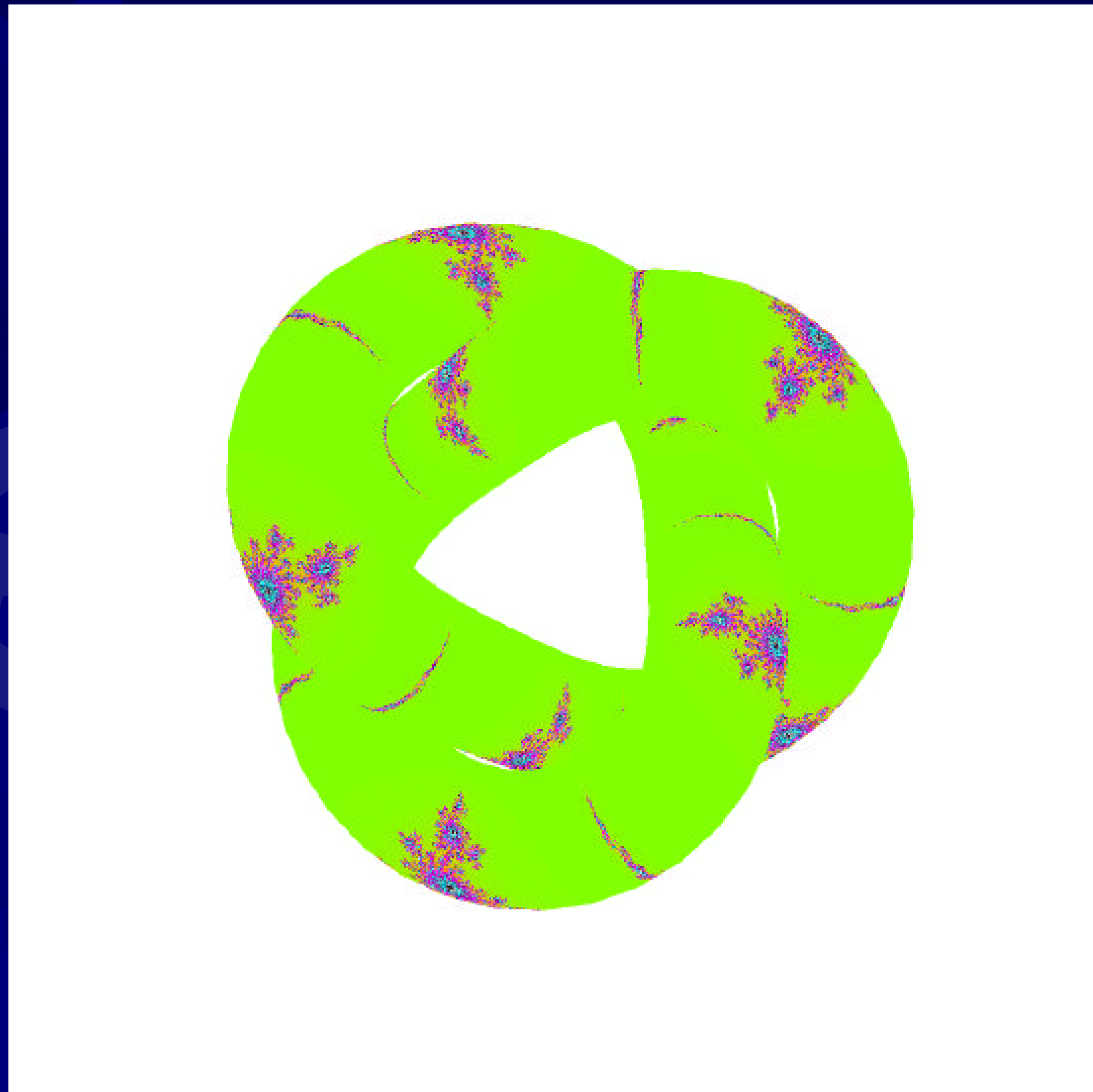
```
ShMatrix3x4f modelview;
ShMatrix4x4f perspective;
ShAttrib1f julia_max_iter;
ShAttrib2f julia_c;
ShAttrib1f julia_scale;
ShTexture1DColor3f julia_map;

ShShader julia0 = SH_BEGIN_SHADER(0) {
  ShInputAttrib2f ui;
  ShInputPoint3f pm;
  ShOutputAttrib2f uo(ui);
  ShOutputPoint4f pd;
  pd = (perspective | modelview) | pm;
} SH_END_SHADER
```

```
ShShader julia1 = SH_BEGIN_SHADER(1) {
  ShInputAttrib2f u;
  ShInputAttrib1f pdz;
  ShInputAttrib2us pdxy;
  ShOutputColor3f fc;
  ShOutputAttrib1f fpdz(pdz);
  ShOutputAttrib2us fpdxy(pdxy);
  ShAttrib1f i = 0.0;
  ShAttrib2f v = u;
  SH_WHILE((v|v) < 2.0 &&
           i < julia_max_iter) {
     v(0) = u(0)*u(0) - u(1)*u(1);
     v(1) = 2*u(0)*u(1);
     u = v + julia_c;
     i++;
  } SH_ENDWHILE
  fc = julia_map[julia_scale*i];
} SH_END_SHADER
```

GRAPHICS HARDWARE 2002

Fragment Computation

```
ShAttrib1f i = 0.0;
ShAttrib2f v = u;
SH_WHILE((v|v) < 2.0 && i < julia_max_iter) {
      v(0) = u(0)*u(0) - u(1)*u(1);
      v(1) = 2*u(0)*u(1);
      u = v + julia_c;
      i++;
} SH_ENDWHILE
fc = julia_map[julia_scale*i];
```



GRAPHICS HARDWARE 2002

Future Work

- Target real hardware
- Arrays
- Subroutines
- Procedural textures
- Standard library
- Asset management
- Introspection

GRAPHICS HARDWARE 2002

Conclusions

- High-level shading language can be embedded in C++ API
- Just a different way to implement a parser
- Benefits:
  - Tighter binding between specification of parameters and use
  - Can “lift” type and modularity constructs from C++ into shading language
  - Simpler implementation of advanced programming features