

## A Real-Time Procedural Shading System for Programmable Graphics Hardware

Kekoa Proudfoot  
William R. Mark  
Pat Hanrahan  
Stanford University

Svetoslav Tzvetkov  
NVIDIA Corporation

<http://graphics.stanford.edu/projects/shading/>

## Programmable graphics hardware

### Register-machine based processing units



e.g. NV vertex programs  
e.g. DirectX8 vertex shaders  
e.g. ATI vertex shaders

100s of instructions  
floating-point  
complex math ops

e.g. NV register combiners  
e.g. DirectX8 pixel shaders  
e.g. ATI fragment shaders

10s of instructions  
fixed-point  
texture mapping

Many interesting effects possible

## Hardware is difficult to use

Two problems:

- Programming interfaces are low-level
- Functionality varies between chipsets

## Hardware is difficult to use

Two problems:

- Programming interfaces are low-level
- Functionality varies between chipsets

User must support:

NV vertex programs  
NV register combiners

```
glCombinerInputNV(GL_COMBINER0_N  
    V, GL_RGB, GL_VARIABLE_A_NV,  
    GL_TEXTURE0_ARB,  
    GL_EXPAND_NORMAL_NV, GL_RGB);  
glCombinerInputNV(...);  
glCombinerInputNV(...);  
glCombinerInputNV(...);  
glCombinerOutputNV(GL_COMBINER0_N  
    V, GL_RGB, GL_SPARE0_NV,  
    GL_DISCARD_NV, GL_DISCARD_NV,  
    GL_NONE, GL_NONE, GL_TRUE,  
    GL_FALSE, GL_FALSE);  
...
```

NV register combiners

## Hardware is difficult to use

Two problems:

- Programming interfaces are low-level
- Functionality varies between chipsets

User must support:  
NV vertex programs

```
DP4 o[HPOS].x, v[OPOS], c[0];  
DP4 o[HPOS].y, v[OPOS], c[1];  
DP4 o[HPOS].z, v[OPOS], c[2];  
DP4 o[HPOS].w, v[OPOS], c[3];  
DP3 R0.x, v[NRM], c[4];  
DP3 R0.y, v[NRM], c[5];  
DP3 R0.z, v[NRM], c[6];  
DP3 R0.w, R0, R0;  
RSQ R0.w, R0.w;  
MUL R0, R0, R0.w;  
DP3 R0.x, R0, c[7];  
LIT R0, R0;  
MUL o[COL0].xyz, R0.y, c[8];
```

NV vertex programs

## Hardware is difficult to use

Two problems:

- Programming interfaces are low-level
- Functionality varies between chipsets

User must support:

NV vertex programs  
NV register combiners  
ATI vertex shaders

```
v0 = glGenSymbolsATI(...);  
glBeginVertexShaderATI(...);  
c0 = glGenSymbolsATI(...);  
glSetLocalConstant(...);  
r0 = glGenSymbolsATI(...);  
r1 = glGenSymbolsATI(...);  
glShaderOp2ATI(...);  
glShaderOp1ATI(...);  
glExtractComponentATI(...);  
glShaderOp2ATI(...);  
glShaderOp2ATI(...);  
...  
glEndVertexShaderATI();
```

ATI vertex shaders

## Hardware is difficult to use

### Two problems:

- Programming interfaces are low-level
- Functionality varies between chipsets

#### User must support:

```
NV vertex programs
NV register combiners
ATI vertex shaders
ATI fragment shaders
...
glBeginFragmentShaderATI(...);
glPassTexCoordATIX(...);
glPassTexCoordATIX(...);
...
glColorFragmentOp2ATIX(GL_MUL_AT
    IX, GL_REG_0_ATIX, GL_RED,
    GL_NONE, GL_REG_4_ATIX,
    GL_NONE, GL_NONE,
    GL_REG_3_ATIX, GL_NONE,
    GL_NONE);
glAlphaFragmentOp2ATIX(...);
...
glEndFragmentShaderATI();
ATI fragment shaders
```

## Hardware is difficult to use

### Two problems:

- Programming interfaces are low-level
- Functionality varies between chipsets

#### User must support:

```
NV vertex programs
NV register combiners
ATI vertex shaders
ATI fragment shaders
Multipass OpenGL
...
glActiveTextureARB(GL_TEXTURE0_A
    RB);
glTexEnv( GL_TEXTURE_ENV,
    GL_TEXTURE_ENV_MODE, GL_ADD );
glActiveTextureARB(GL_TEXTURE1_A
    RB);
glTexEnv( GL_TEXTURE_ENV, ... );
glTexEnv( GL_TEXTURE_ENV, ... );
glTexEnv( GL_TEXTURE_ENV, ... );
glEnable(GL_BLEND);
glBlendFunc(GL_DST_COLOR,
    GL_ZERO);
...
Multipass OpenGL
```

## Hardware is difficult to use

### Two problems:

- Programming interfaces are low-level
- Functionality varies between chipsets

#### User must support:

```
NV vertex programs
NV register combiners
ATI vertex shaders
ATI fragment shaders
Multipass OpenGL
Host vertex code
...
for ( i = 0; i < n_verts; i++)
{
    // perform vertex processing
}
```

Host vertex code

## Hardware is difficult to use

### Two problems:

- Programming interfaces are low-level
- Functionality varies between chipsets

#### User must support:

```
NV vertex programs
NV register combiners
ATI vertex shaders
ATI fragment shaders
Multipass OpenGL
Host vertex code
...
Next-generation hardware
```

Next-generation hardware

## Shading languages

### Solution:

- Shading languages provide an easy-to-use abstraction of programmability
  - e.g. RenderMan, Hanrahan and Lawson SIGGRAPH 90
- Apply shading languages to real-time systems

## Related work

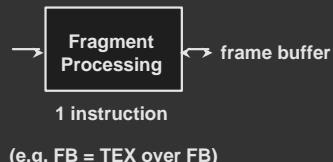
- Olano and Lastra, SIGGRAPH 98
  - Shading language specialized for PixelFlow
- id Software's Quake 3 shader scripts
  - Scripting language for shading computations
- Peercy et al., SIGGRAPH 00
  - SIMD processor abstraction for multipass rendering
- McCool's SMASH API
  - API for specifying shading computations

## SIMD processor model

Single instruction per pass

Many passes

Fragments only

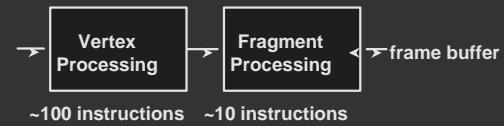


## Programmable processor model

Many instructions per pass

Fewer passes – often just one

Vertices and fragments



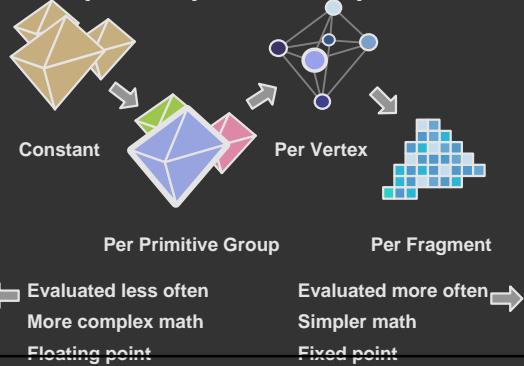
## Programmable model advantages

1. Support for vertex operations
2. Reduced off-chip bandwidth
3. Better match to current hardware

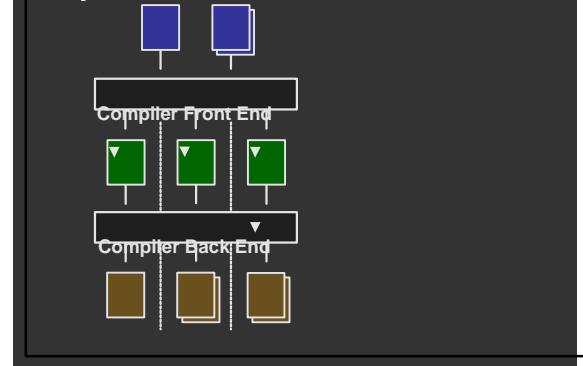
## System design goals

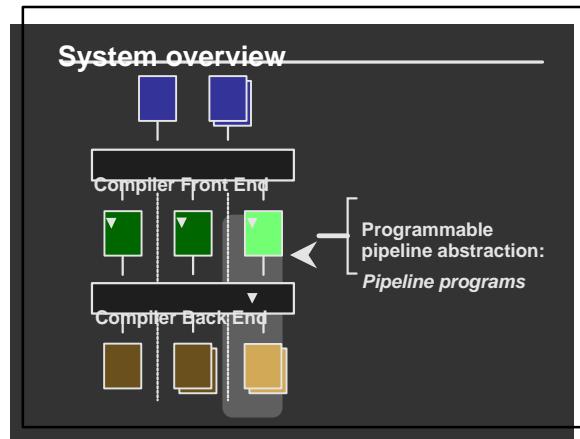
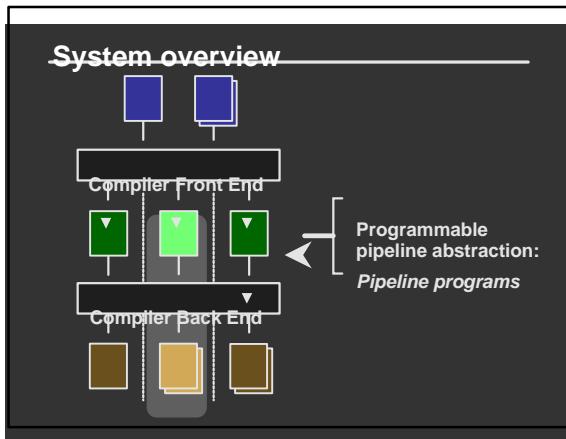
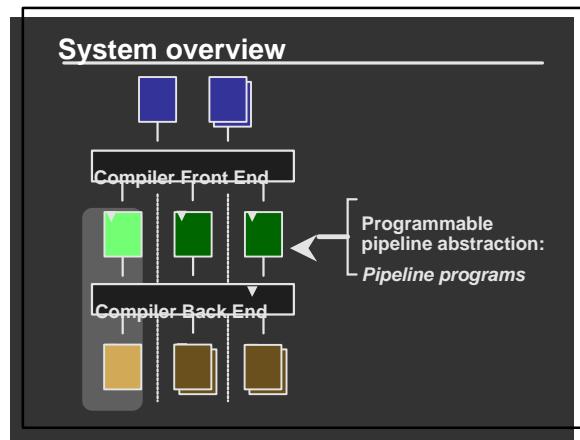
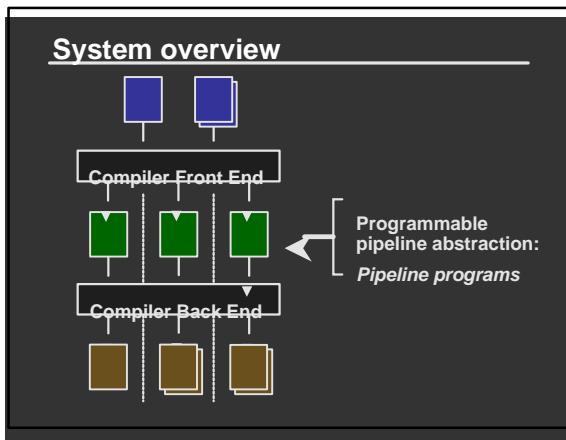
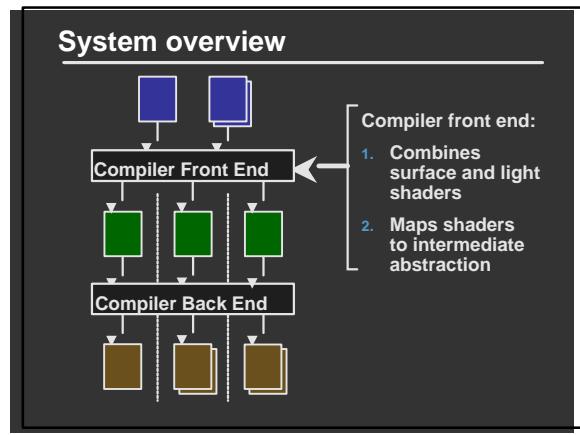
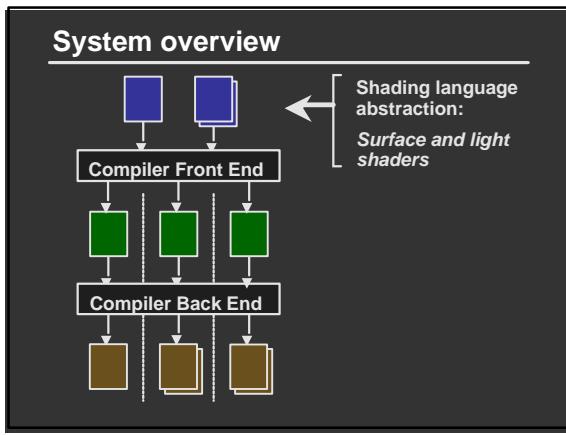
1. Implement a shading language
2. Support a variety of hardware
3. Generate fast and efficient code
4. Create a framework for future hardware design

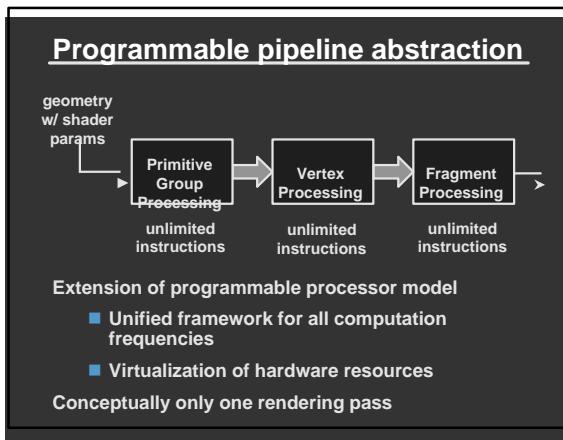
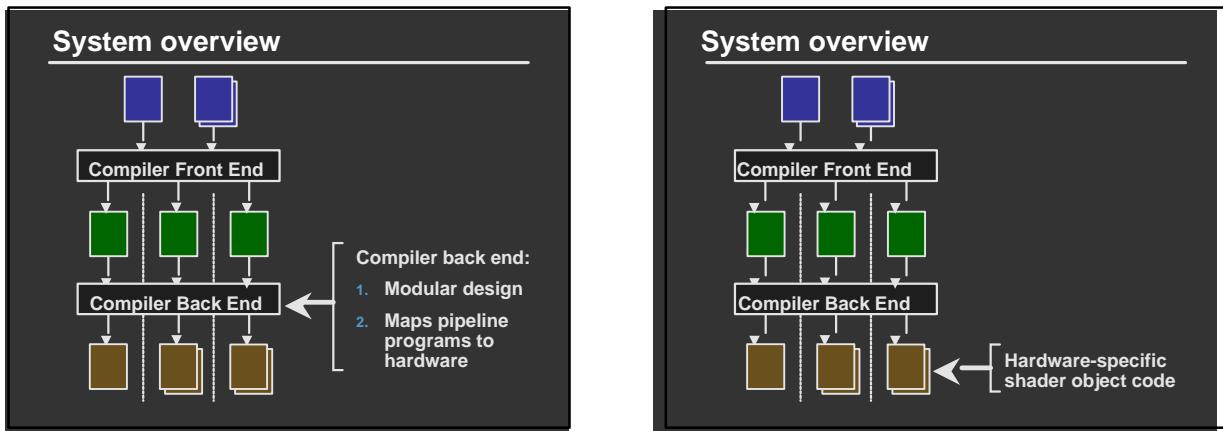
## Multiple computation frequencies



## System overview







- ### Accommodating today's hardware
- No true fragment floating-point type
    - Implement with fixed-point instead
  - Not every operator is available at every computation frequency
    - e.g. no vertex textures
    - e.g. no complex per-fragment ops
  - Not every operator is available on all hardware
    - e.g. cubemaps and 3D textures

### Shading language example

```
surface shader float4
anisotropic_ball (texref anisotex, texref star)
{
    // generate texture coordinates
    perlight float4 uv = { center(dot(B, E)),
                           center(dot(B, L)),
                           0, 1 };

    // compute reflection coefficient
    perlight float4 fd = max(dot(N, L), 0);
    perlight float4 fr = fd * texture(anisotex, uv);

    // compute amount of reflected light
    float4 lightcolor = 0.2 * Ca + integrate(Cl * fr);

    // modulate reflected light color
    float4 uv_base = { center(Pobj[2]), center(Pobj[0]),
                       0, 1 };
    return lightcolor * texture(star, uv_base);
}
```

### Shading language example

```
surface shader float4
anisotropic_ball (texref anisotex, texref star)
{
    // generate texture coordinates
    perlight float4 uv = { center(dot(B, E)),
                           center(dot(B, L)),
                           0, 1 };

    // compute reflection coefficient
    perlight float4 fd = max(dot(N, L), 0);
    perlight float4 fr = fd * texture(anisotex, uv);

    // compute amount of reflected light
    float4 lightcolor = 0.2 * Ca + integrate(Cl * fr);

    // modulate reflected light color
    float4 uv_base = { center(Pobj[2]), center(Pobj[0]),
                       0, 1 };
    return lightcolor * texture(star, uv_base);
}
```

## Shading language example

```
surface shader float4
anisotropic_ball (texref anisotex, texref star)
{
    // generate texture coordinates
    perlight float4 uv = { center(dot(B, E)),
                           center(dot(B, L)),
                           0, 1 };

    // compute reflection coefficient
    perlight float4 fd = max(dot(N, L), 0);
    perlight float4 fr = fd * texture(anisotex, uv);

    // compute amount of reflected light
    float4 lightcolor = 0.2 * Ca + integrate(Cl * fr);

    // modulate reflected light color
    float4 uv_base = { center(Pobj[2]), center(Pobj[0]),
                       0, 1 };
    return lightcolor * texture(star, uv_base);
}
```



## Shading language example

```
surface shader float4
anisotropic_ball (texref anisotex, texref star)
{
    // generate texture coordinates
    perlight float4 uv = { center(dot(B, E)),
                           center(dot(B, L)),
                           0, 1 };

    // compute reflection coefficient
    perlight float4 fd = max(dot(N, L), 0);
    perlight float4 fr = fd * texture(anisotex, uv);

    // compute amount of reflected light
    float4 lightcolor = 0.2 * Ca + integrate(Cl * fr);

    // modulate reflected light color
    float4 uv_base = { center(Pobj[2]), center(Pobj[0]),
                       0, 1 };
    return lightcolor * texture(star, uv_base);
}
```



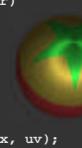
## Shading language example

```
surface shader float4
anisotropic_ball (texref anisotex, texref star)
{
    // generate texture coordinates
    perlight float4 uv = { center(dot(B, E)),
                           center(dot(B, L)),
                           0, 1 };

    // compute reflection coefficient
    perlight float4 fd = max(dot(N, L), 0);
    perlight float4 fr = fd * texture(anisotex, uv);

    // compute amount of reflected light
    float4 lightcolor = 0.2 * Ca + integrate(Cl * fr);

    // modulate reflected light color
    float4 uv_base = { center(Pobj[2]), center(Pobj[0]),
                       0, 1 };
    return lightcolor * texture(star, uv_base);
}
```



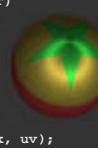
## Computation frequency analysis

```
surface shader float4
anisotropic_ball (texref anisotex, texref star)
{
    // generate texture coordinates
    perlight float4 uv = { center(dot(B, E)),
                           center(dot(B, L)),
                           0, 1 };

    // compute reflection coefficient
    perlight float4 fd = max(dot(N, L), 0);
    perlight float4 fr = fd * texture(anisotex, uv);

    // compute amount of reflected light
    float4 lightcolor = 0.2 * Ca + integrate(Cl * fr);

    // modulate reflected light color
    float4 uv_base = { center(Pobj[2]), center(Pobj[0]),
                       0, 1 };
    return lightcolor * texture(star, uv_base);
}
```



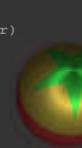
## Computation frequency analysis

```
surface shader float4
anisotropic_ball (texref anisotex, texref star)
{
    // generate texture coordinates
    perlight float4 uv = { center(dot(B, E)),
                           center(dot(B, L)),
                           0, 1 };

    // compute reflection coefficient
    perlight float4 fd = max(dot(N, L), 0);
    perlight float4 fr = fd * texture(anisotex, uv);

    // compute amount of reflected light
    float4 lightcolor = 0.2 * Ca + integrate(Cl * fr);

    // modulate reflected light color
    float4 uv_base = { center(Pobj[2]), center(Pobj[0]),
                       0, 1 };
    return lightcolor * texture(star, uv_base);
}
```



## Computation frequency analysis

```
surface shader float4
anisotropic_ball (texref anisotex, texref star)
{
    // generate texture coordinates
    perlight float4 uv = { center(dot(B, E)),
                           center(dot(B, L)),
                           0, 1 };

    // compute reflection coefficient
    perlight float4 fd = max(dot(N, L), 0);
    perlight float4 fr = fd * texture(anisotex, uv);

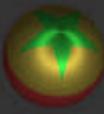
    // compute amount of reflected light
    float4 lightcolor = 0.2 * Ca + integrate(Cl * fr);

    // modulate reflected light color
    float4 uv_base = { center(Pobj[2]), center(Pobj[0]),
                       0, 1 };
    return lightcolor * texture(star, uv_base);
}
```



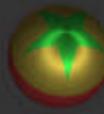
## Computation frequency analysis

```
surface shader float4  
anisotropic_ball (texref anisotex, texref star)  
{  
    // generate texture coordinates  
    perlight float4 uv = { center(dot(B, E)),  
                           center(dot(B, L)),  
                           0, 1 };  
  
    // compute reflection coefficient  
    perlight float4 fd = max(dot(N, L), 0);  
    perlight float4 fr = fd * texture(anisotex, uv);  
  
    // compute amount of reflected light  
    float4 lightcolor = 0.2 * Ca + integrate(Cl * fr);  
  
    // modulate reflected light color  
    float4 uv_base = { center(Pobj[2]), center(Pobj[0]),  
                       0, 1 };  
    return lightcolor * texture(star, uv_base);  
}
```



## Computation frequency analysis

```
surface shader float4  
anisotropic_ball (texref anisotex, texref star)  
{  
    // generate texture coordinates  
    perlight float4 uv = { center(dot(B, E)),  
                           center(dot(B, L)),  
                           0, 1 };  
  
    // compute reflection coefficient  
    perlight float4 fd = max(dot(N, L), 0);  
    perlight float4 fr = fd * texture(anisotex, uv);  
  
    // compute amount of reflected light  
    float4 lightcolor = 0.2 * Ca + integrate(Cl * fr);  
  
    // modulate reflected light color  
    float4 uv_base = { center(Pobj[2]), center(Pobj[0]),  
                       0, 1 };  
    return lightcolor * texture(star, uv_base);  
}
```



## Analysis and optimization

### Global optimization

- All functions are inlined
- Surfaces and lights are compiled together

### Single basic block

- No data-dependent loops or conditionals

*All shading computations are reduced to a pair of expressions*

## Retargetable compiler back end

### Two goals:

- Provide support for many hardware platforms
- Virtualize hardware resources

## Virtualizing hardware resources

### Multipass rendering

- Arbitrarily-complex fragment computations
- Reduces vertex resource requirements

### Host processing

- Useful when vertex operations are very complex

## Back end module interactions

1. Pass-related interactions
  - e.g. partition vertex code by pass
2. Resource constraint interactions
  - e.g. fall back to host if vertex-processing resources insufficient
3. Data flow interactions
  - e.g. define data formats for computed values

## Current back end modules

### Host:

- C code with external compiler
- Internal x86 assembler

### Hardware:

- Multipass OpenGL with extensions
- NVIDIA vertex programs
- NVIDIA register combiners
- ATI vertex and fragment shaders\*
- Stanford Imagine processor\*

\* recent additions

## Vertex back end experiences

### Current vertex processing architectures:

- Clean design
- Easy to target

True for both ATI and NVIDIA

- Different APIs
- Equivalent functionality
- ATI hides register allocation

See paper for compilation details

## Fragment back end experiences

### Multipass OpenGL with extensions

- Relatively easy to target using tree-matching

### NVIDIA register combiners

- Tree matching does not work!
- Use VLIW compilation techniques instead
- Hardware is not orthogonal
- Compilation quality is still very good
- Current back end: single pass only
- See also: HWWS 01 paper

## Demo

### Demo machine info:

- 866 MHz Pentium III
- NVIDIA GeForce3

All shaders compiled using NVIDIA vertex program and NVIDIA register combiner back ends

## Summary

### Hardware

- Very capable, but difficult to use
- Provide a shading language interface

### Stanford shading system

- Shading language designed for hardware
- Programmable pipeline abstraction
- Retargetable compiler back end
- System runs in real time on today's hardware

## Final thoughts

### Past

- Fixed-function pipelines
- Feature-based interfaces

### Present

- Limited programmability
- Assembly-language interfaces

### Future

- Generalized programmability
- Shading language interfaces
- Hardware designed for compiled code

## Acknowledgements

---

### Stanford

Pradeep Sen, Ren Ng, Eric Chan, John Owens,  
Bill Dally, Philipp Slussalek

### Bowling pin data

Anselmo Lastra, Lawrence Kesteloot,  
Frederik Fatemi

### Fish data

Xiaoyuan Tu, Homan Igehy

### Sponsors

ATI, SUN, SGI, SONY, NVIDIA  
DARPA

### Web site and downloads

<http://graphics.stanford.edu/projects/shading/>