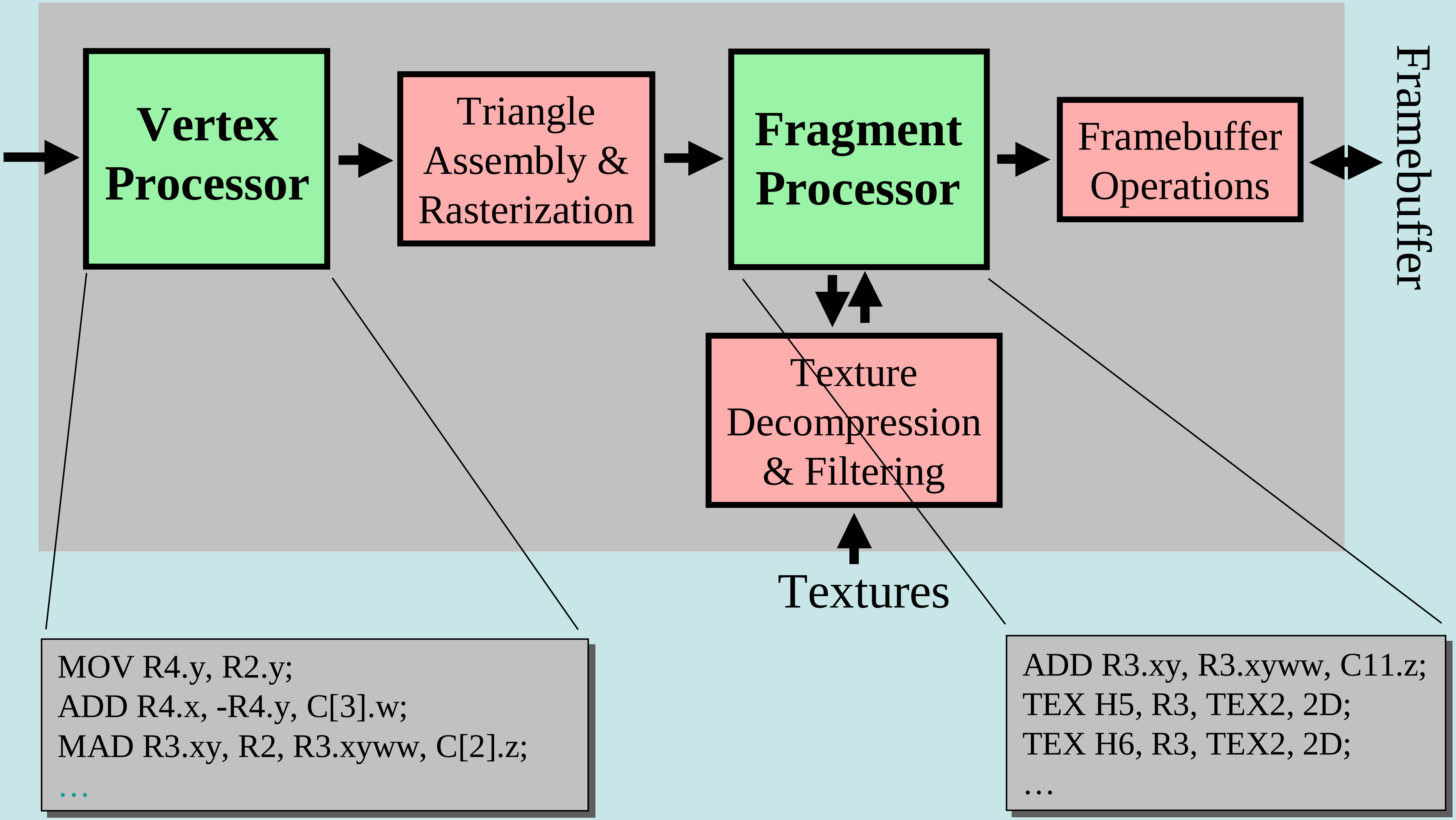
Cg: A system for programming graphics hardware in a C-like language

William R. Mark – *University of Texas at Austin*
R. Steven Glanville – *NVIDIA*
Kurt Akeley – *NVIDIA and Stanford University*
Mark Kilgard – *NVIDIA*

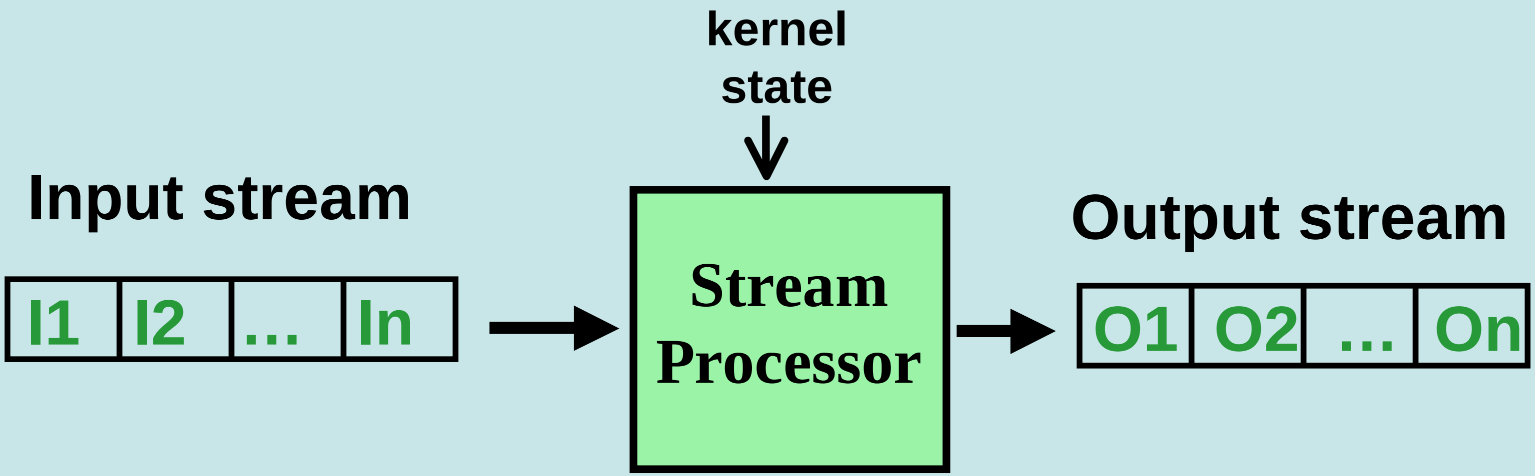
Background

- Cg is both a language and a system
  - Cg language is for writing stream kernels
  - Cg system targets graphics hardware
- Developed by NVIDIA
  - In collaboration with Microsoft
  - Runs on lots of hardware (not just NVIDIA's)
- Widely deployed
  - Shipping for over a year
  - Anyone can download it
- Lots of information available:
  - Cg language specification – via download
  - Cg tutorial – buy on amazon.com
  - Paper in SIGGRAPH 2003

GPUs are now programmable

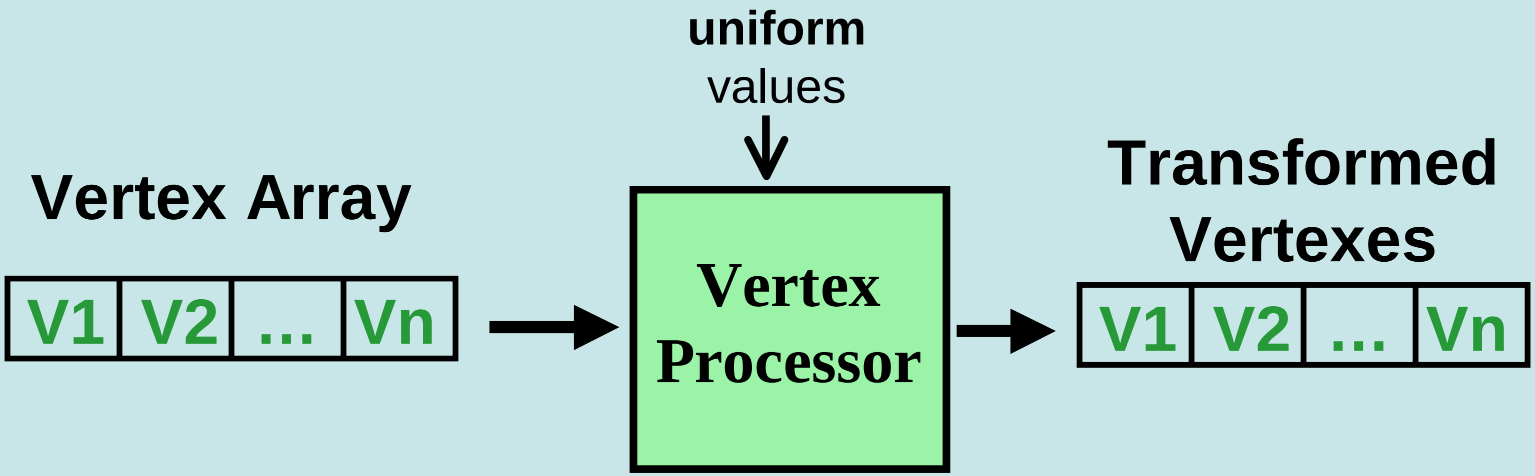


Programmable units in GPUs are stream processors



- The programmable unit executes a computational kernel for each input element
- Streams consist of ordered elements

Example:
Vertex processor



Fragment processor can do more:

It can read from texture memory

Previous work

- Earlier programmable graphics HW
  - Ikonas [England 86]
  - Pixar FLAP [Levinthal 87]
  - UNC PixelFlow [Olano 98]
  - Multipass SIMD: SGI ISL [Peercy00]
- RenderMan [Hanrahan90]
- Stanford RTSL [Proudfoot01]

Design goals

- Easier programming of GPUs
- Ease of adoption

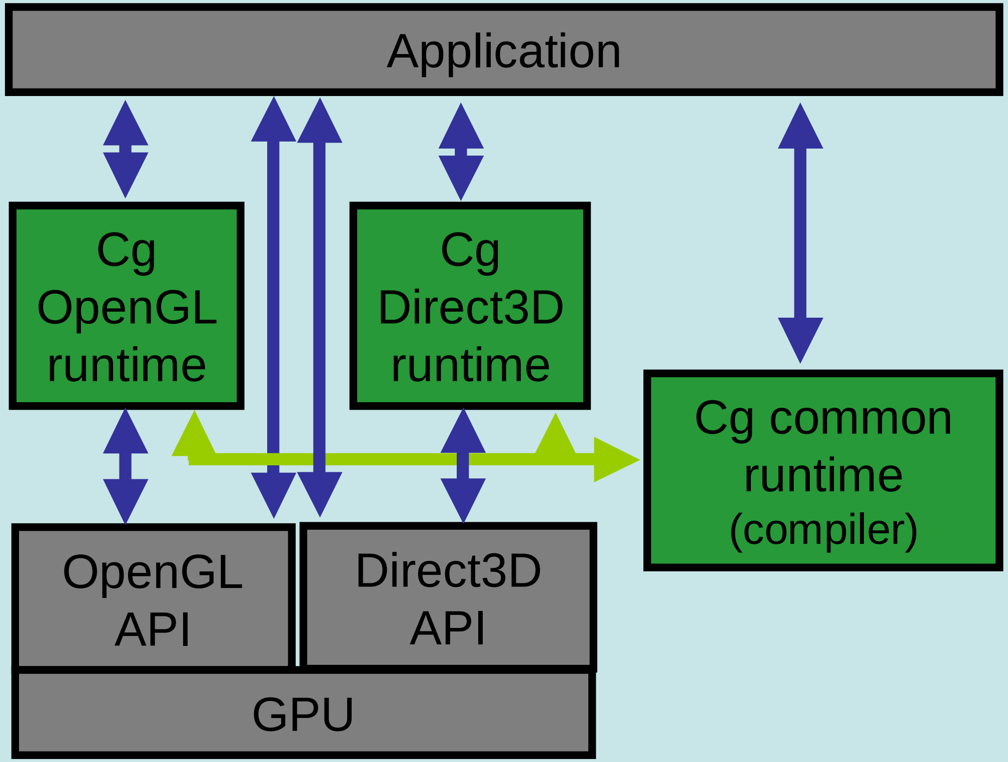
- Portability – HW, API, OS
- Complete support for HW capabilities
- Performance - similar to assembly
- Minimal interference with application data
- Longevity -- useful for future hardware
- Support for non-shading computations

Non-goal: Backward compatibility

Major Design Decisions

Modular or monolithic architecture?

- Cg system is modular
- Provides access to assembly-level interfaces
- Other systems chose differently
- Compile off-line, or at run time



The Cg system manages Cg programs AND the flow of data on which they operate

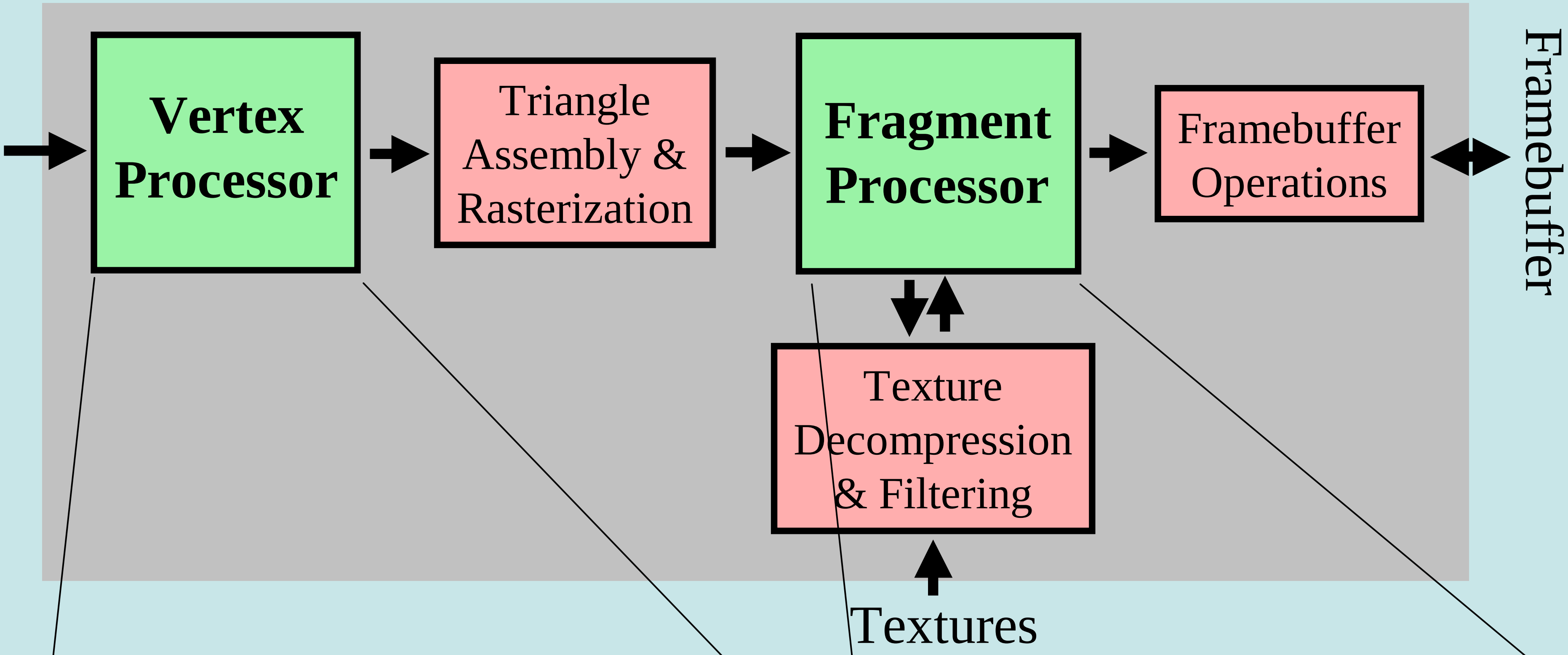
Specialize for shading?

- RenderMan language is domain-specific
  - Domain-specific types (e.g. “color”)
  - Domain-specific constructs (e.g. “illuminance”)
  - Imposes a particular model on the user
- C is general purpose
  - A HW oriented kernel language
  - Avoids assumptions about problem domain
- Cg follows C’s philosophy
  - Language follows syntax and philosophy of C
  - Targets GPU HW – performance transparency
  - Some exceptions – we were not dogmatic

Cg language example

```
void simpleTransform(float4   objectPosition : POSITION,
                     float4   color          : COLOR,
                     float4   decalCoord     : TEXCOORD0,
                 out float4   clipPosition   : POSITION,
                 out float4   Color          : COLOR,
                 out float4   oDecalCoord    : TEXCOORD0,
             uniform float    brightness,
             uniform float4x4 modelViewProjection)
{
  clipPosition = mul(modelViewProjection, objectPosition);
  oColor = brightness * color;
  oDecalCoord = decalCoord;
}
```

One program or two?




```
void vprog(float4 objP  : POSITION,
           float4 color : COLOR,
...
```

Vertex program

```
void fprog(float4 t0  : TEXCOORD0,
           float4 t1 : TEXCOORD1,
...
```

Fragment program

How should stream HW be exposed in the language?

- Basic organization:
  - Separate program for each kernel
  - Kernels may include control flow (SPMD)
- Inputs & outputs:
  - One input stream, one output stream
  - Multiple variables in each record
  - No conditional inputs or outputs
  - Multiple non-stream inputs (“uniform” vars)
- Memory:
  - Memory-read operations OK within kernel

How should system support different levels of HW?

NV20 R300 NV40 NV50

R200 NV30 R400 … ?

- HW capabilities change each generation
  - Data types
  - Support for branch instructions, …
- We expect this problem to persist
  - Future GPUs will have new features
- Mandate exactly one feature set?
  - Must strand older HW or limit newer HW

Two options for handling HW differences

- Emulate missing features?
  - Too slow on GPU
  - Too slow on CPU, especially for fragment HW
- Expose differences to programmer?
  - We chose this option
  - Differences exposed via **subsetting**
  - A **profile** is a named subset
  - Cg supports function overloading by profile

Cg is closely related to other recent languages

- Microsoft HLSL
  - Largely compatible with Cg
  - NVIDIA and Microsoft collaborated
- OpenGL ARB shading language
- All three languages are similar
  - Overlapping development
  - Extensive cross-pollination of ideas
  - Designers mostly agreed on right approach
- Systems are different

Summary

- Cg system:
  - A system for programming GPUs
- Cg language:
  - Extends and restricts C as needed for GPU’s
  - Expresses stream kernels
  - HW oriented language
- Designed to age well
  - By reintroducing missing C features

Acknowledgements

- Language co-designers at Microsoft:
  Craig Peeper and Loren McQuade
- Interface functionality: Craig Kolb, Matt Pharr
- Initial language direction: Cass Everitt
- Standard library: Chris Wynn
- Design and implementation team:
  Geoff Berry, Mike Bunnell, Chris Dodd, Wes Hunt, Jayant Kolhe, Rev Lebaredian, Nathan Paymer, Matt Pharr, Doug Rogers
- Director: Nick Triantos
- Many others at NVIDIA and Microsoft
- Code and experience from Stanford RTSL: Kekoa Proudfoot, Pat Hanrahan, and others

Questions

Backup slides

Two detailed design decisions

How to specify a returnable function parameter?

- C uses pointers
  - But no pointers on current GPUs

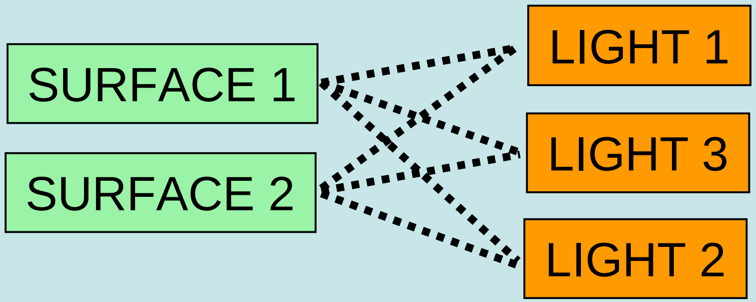
```
void foobar(float *y) {
    *y = 2*x;
    ...
```

- C++ uses pass-by-reference: float &y
  - Preferred implementation still uses pointers
- Cg uses pass-by-value-result
  - Implementation doesn’t need pointers

```
void foobar(out float y) {
    y = 2*x;
    ....
```

- **By using new syntax, we preserve C and C++ syntax for the future**

General mechanism for surface/light separability



- Convenient to put **surface** and **light** code in different modules…
  - Decouples surface selection from light selection
  - Proven usefulness: Classic OpenGL; RenderMan; etc.
  - We lost it for a while!
- RenderMan uses a specialized mechanism
- Cg uses a general-purpose mechanism
  - More flexible
  - Follows C-like philosophy

Light/surface example…
First: Declare an interface

```
// Declare interface to lights
interface Light {
    float3 direction(float3 from);
    float4 illuminate(float3 p, out float3 lv);
};
```

Mechanism adapted from Java, C#

Second: Declare a light that implements interface

```
// Declare object type for point lights
struct PointLight : Light {
    float3 pos, color;
    float3 direction(float3 p) { return pos - p; }
    float3 illuminate(float3 p, out float3 lv) {
        lv = normalize(direction(p));
        return color;
    }
};
```

Declare an object that implements the interface

Third: Define surface that calls interface

```
// Main program (surface shader)
float4 main(appin IN, out float4 COUT,
            uniform Light lights[ ]) {
  ...
  for (int i=0; i < lights.Length; i++) { // for each light
    Cl = lights[i].illuminate(IN.pos, L); // get dir/color
    color += Cl * Plastic(texcolor, L, Nn, In, 30); // apply
    }
  COUT = color;
}
```

Call object(s) via the interface type