

Real-Time Procedural Shading for Programmable Graphics Hardware

Bill Mark

Group members & collaborators:

**Pat Hanrahan, Kekoa Proudfoot, Svetoslav Tzvetkov,
Pradeep Sen, Ren Ng, Eric Chan, John Owens, David Ebert**

Three related topics

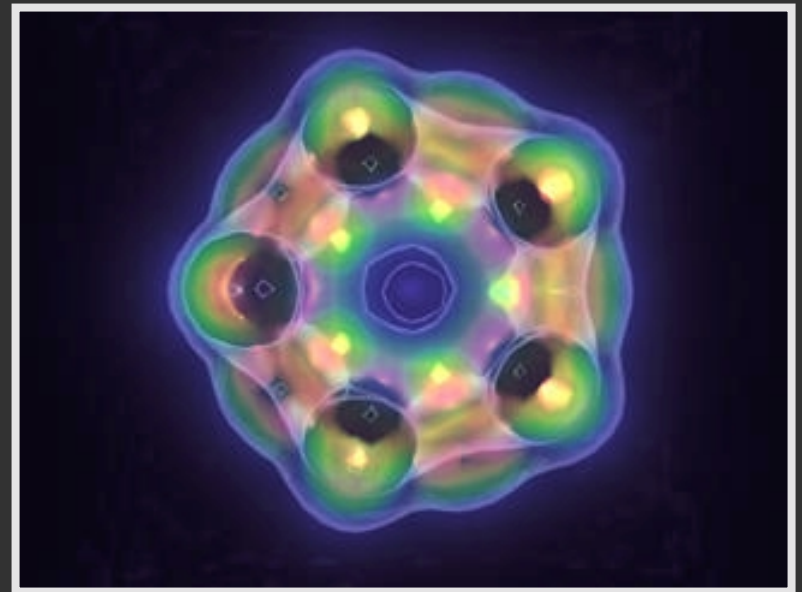
- Overview of real-time shading system
- Comments on domain-specific languages
- Graphics hardware as a stream processor

The goal: Movie quality in real time



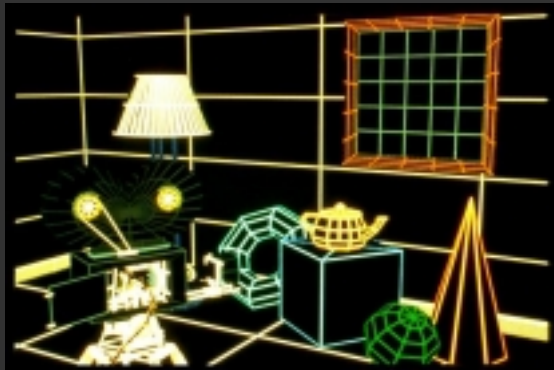
Toy Story
(RenderMan software)

David Bock's
Scientific Visualizations
(NCSA)



Realism comes from materials and lighting

Geometry



Shaders

```
surface shader clampfv heidrich1 (texref star, texref anisotex) {  
  //  
  // surface texture  
  //  
  vertex floatv tcoord = {Pobj[2]*0.5+0.5, Pobj[0]*0.5+0.5, 0, 1};  
  fragment clampfv surface = texture(star, tcoord);  
  //  
  // lighting model (per-light)  
  //  
  perlight vertex floatv uv = {dot(B,E)*0.5+0.5, dot(B,L)*0.5+0.5, 0, 1};  
  perlight fragment clampfv fr = max(dot(N,L),0) * texture(anisotex, uv);  
  //  
  // Combine everything, including ambient  
  //  
  constant clampfv amb = {0.1, 0.1, 0.1, 1.0};  
  return surface * (amb * Ca + integrate(Cl * fr));  
}
```

Final Image

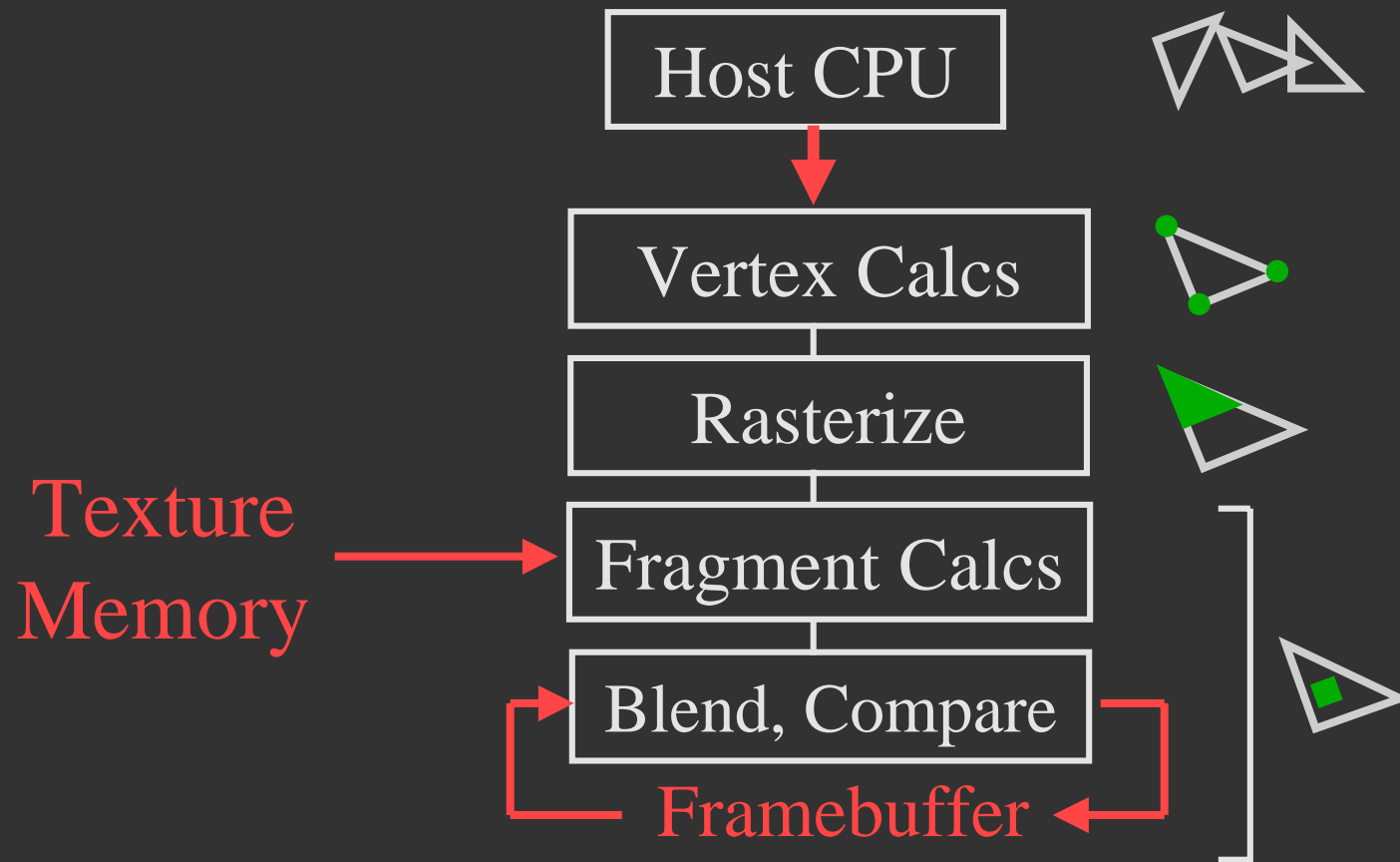


Shaders calculate surface color
(and sometimes opacity, displacement, ...)

Shading calcs may be broken into multiple parts

- e.g. “lights” and “surfaces”

Graphics hardware pipeline



The opportunity

Current generation of graphics hardware is very capable

- Programmable vertex hardware
- Programmable fragment hardware
- Performance much greater than CPU



NVIDIA GeForce3 (NV20):
~80 GFlops/sec
~800 Gops/sec

The problem

Hardware is difficult to program

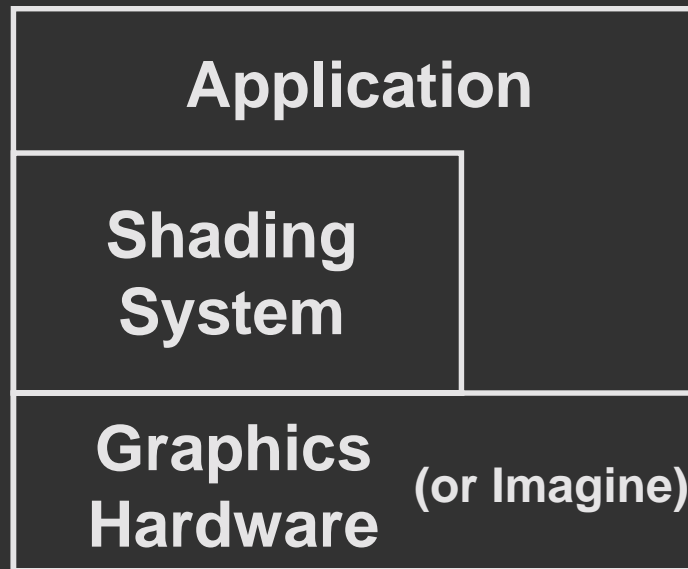
- Programming is like writing microcode
- Hard to coordinate host, vertex, and fragment code
- Must rewrite code for each HW platform

```
SGE R1.x, v[0].z, c[17].y;  
MAD R1.z, R1.x, -c[17].y, c[17].y;  
MAD R1.z, R1.x, c[18].w, R1.z ;  
SLT R1.y, c[19].w, v[4].x;  
MAD R1.x, R1.y, -c[18].w, c[18].w;
```

Our Approach: Domain-specific language

Shading language →

OpenGL →



Benefits of domain-specific languages

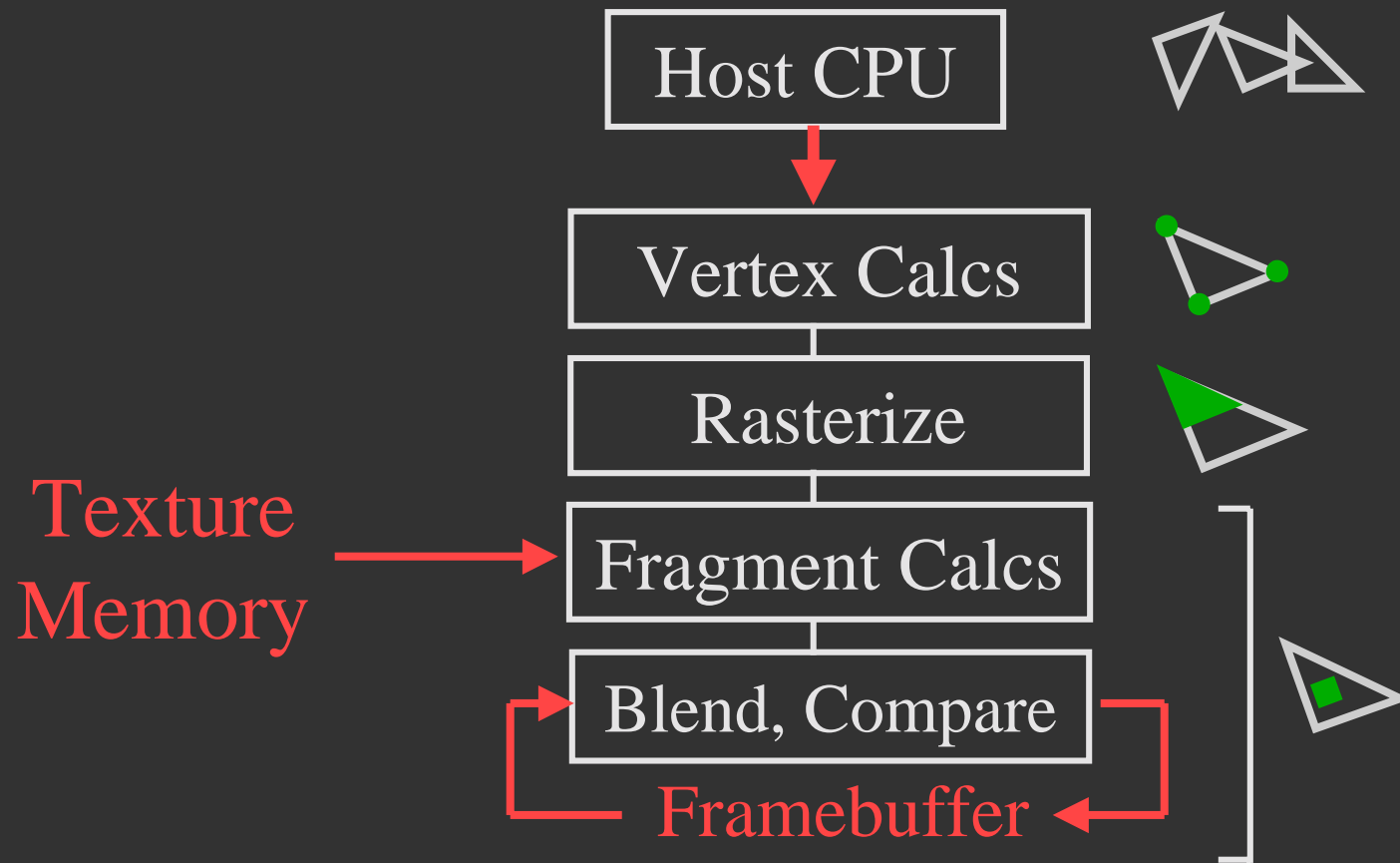
Constrain the user

- Easier for the user (hopefully)
- Facilitates optimization
 - Highly-structured dependency information
 - “Non-programmable” parts of system can use hardware-specific code

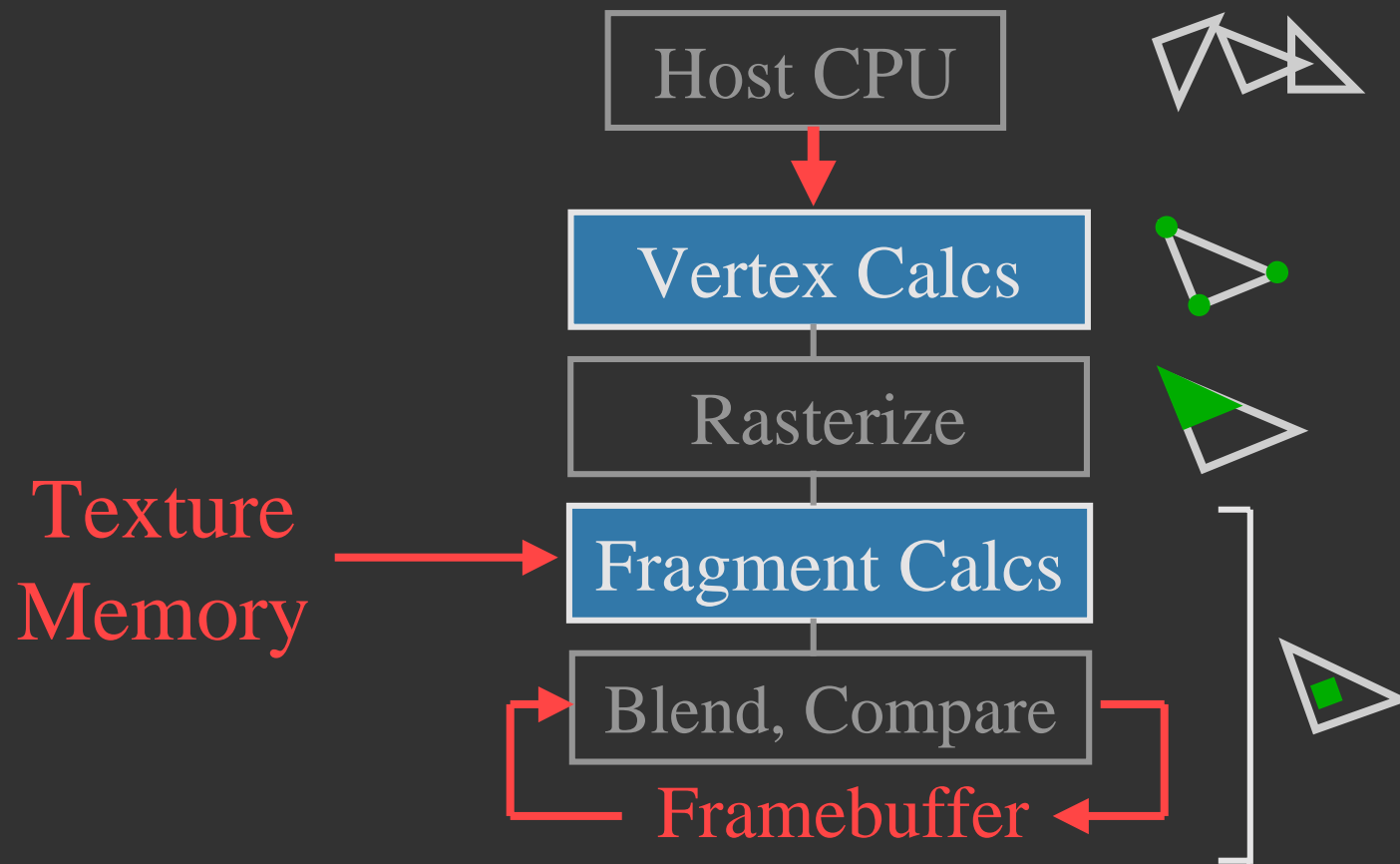
Provide leverage for the user

- Domain-specific abstractions, data types, operators, ...

Our system constrains the user



Our system constrains the user



Our system constrains the user

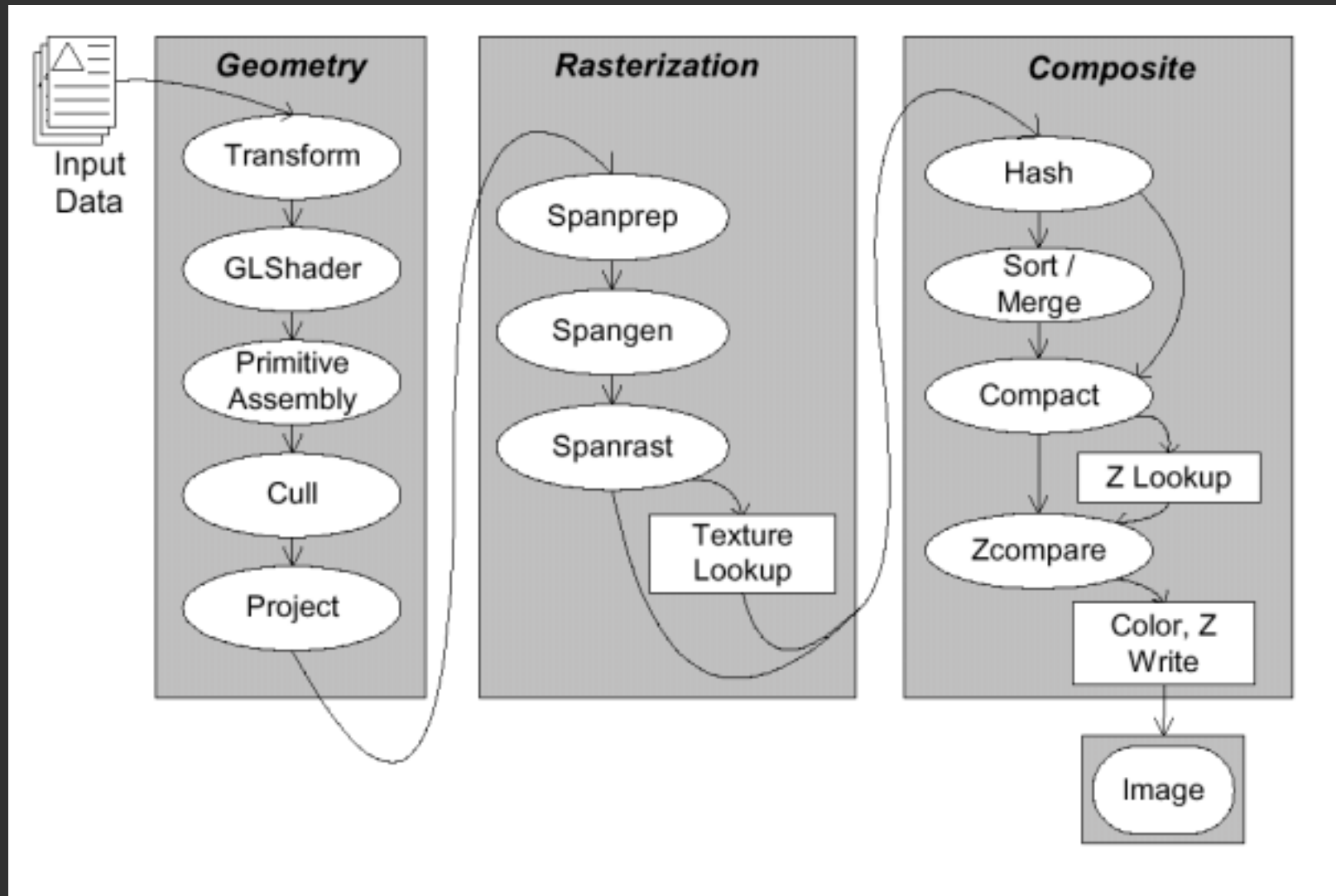
No user-specified communication between vertices

No user-specified communication between fragments

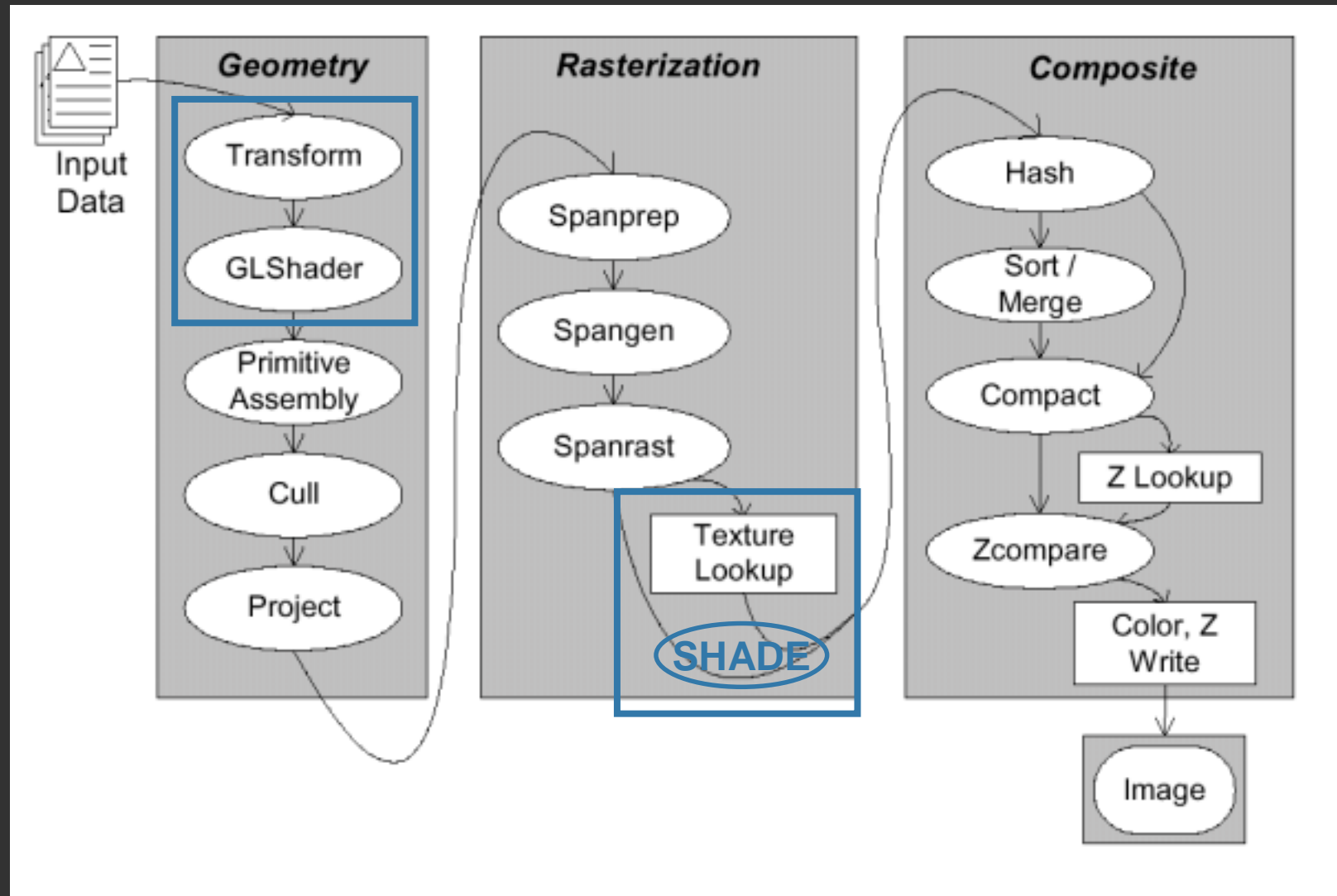
Memory access is either read-only (textures) or write-only (framebuffer)

→ **User-specified computations are “embarrassingly parallel”.**

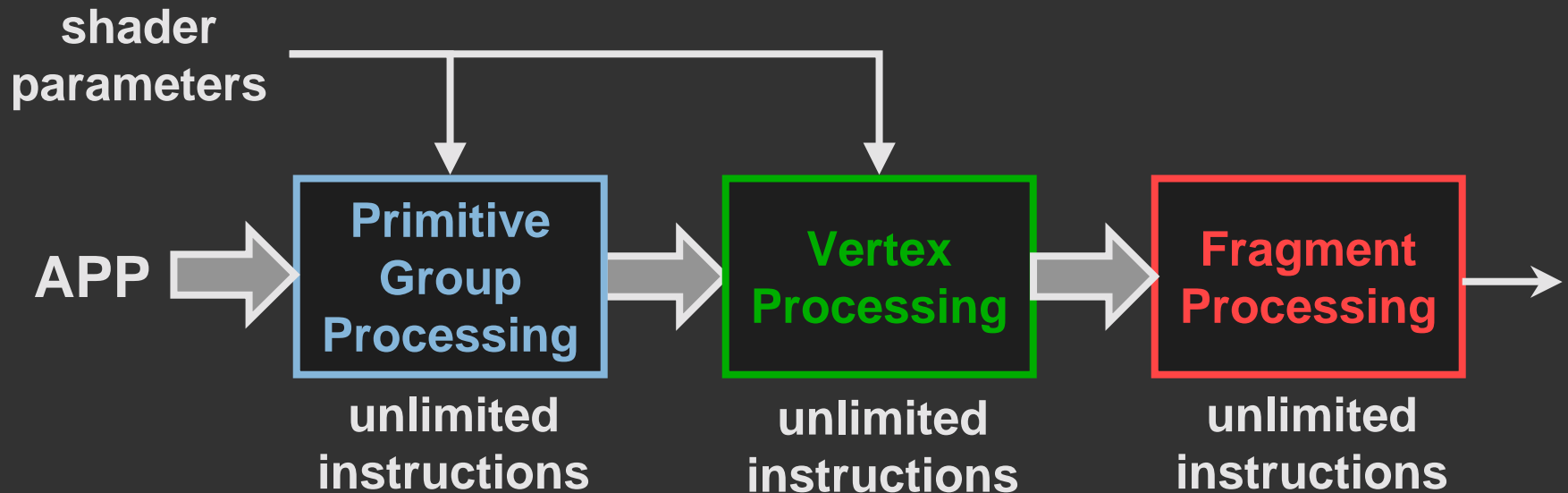
Polygon rendering on Imagine



Polygon rendering on Imagine



Our system's abstraction



Extends current hardware models

- Unified framework for all computation frequencies
- Virtualization of hardware resources

Conceptually only one rendering pass

Language Example

constant
primitive-group
vertex
fragment

```
surface float3
lightmodel_bumps(float3 a, float3 d, float3 s,
                 texref bumps, floatv uv_bumps) {
    // Compute normalized tangent-space light vectors
    vertex perlight float3 Ltan = tangentspace(L);
    vertex perlight float3 Htan = tangentspace(H);

    // Lookup from normal map
    float4 Nlookup = texture(bumps, uv_bumps);
    float3 Nbump   = 2.0*(rgb(Nlookup)-triple(0.5));
    float  N_avglen = Nlookup[3];

    // Diffuse
    perlight fragment float3 Lfrag = Ltan; // interpolate
    perlight float  NdotL = dot(Nbump, Lfrag);
    perlight float  shadow = 4*(Lfrag[2] + Lfrag[2]); // Geometric shadow
    perlight float3 diff  = d * clamp01(NdotL) * clamp01(shadow) * N_avglen;
    :
    :
```

Some other features of our language

Data types

- e.g. clamp to $[0,1]$ range
- e.g. four-vectors

Built-in functions – e.g. matrix operations

Predefined variables – e.g. eye-space normal vector

Allow “surfaces” and “lights” to be defined separately

- combine them with special “integrate()” operator.

Some things we've done right

We built something quickly and then iterated

We found users for our system

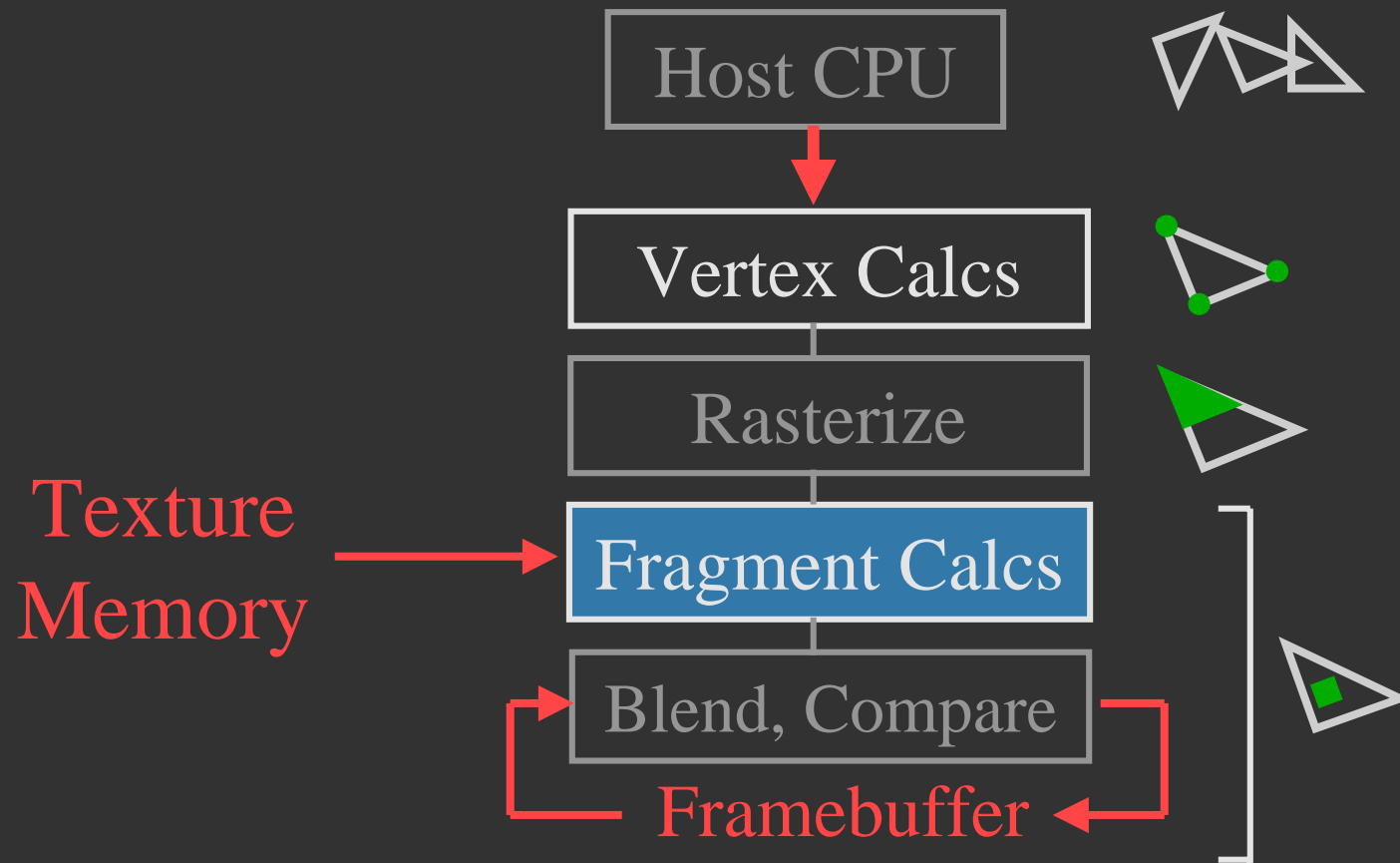
- **Mostly in-house**
- **They had other goals they were trying to reach**
 - **Real-time programmable volume rendering**
 - **Implementing new shading algorithms**
- **They complained!**

We focused on barriers to acceptance

- **E.g. performance of compiled code**
- **Got people to want our system, now!**

We had a well-defined hardware model from the start

Graphics HW → stream processor



What's needed for stream processing

Full set of floating-point fragment operations

Generalized texture reads

FIFO “framebuffer” (a.k.a Stream buffer)

All of these are needed already, for programmable shading!

Other algorithms (e.g. raytracing) *will* be efficiently implementable within two HW generations.

One of our goals: Push this evolution

→ massive parallelism on the desktop.