Chapter 10

Multi-Pass RenderMan

Marc Olano

Multi-Pass RenderMan

Marc Olano
SGI

1. The RenderMan Shading Language

RenderMan is an interface for talking to Renderers [Upstill90]. There are several RenderMan-compatible software renderers, the most well known of which is Pixar's PhotoRealistic RenderMan. One of the most interesting features of the RenderMan Interface is its shading language [Hanrahan90]. The RenderMan shading language is not the only shading language for software rendering (see Chapter 2), but it is often used as a standard of comparison thanks to its power and its popularity.

So, if RenderMan is "the standard", why aren't there any real-time RenderMan implementations? It isn't that we wouldn't want one. First, graphics hardware simply isn't capable enough yet. Second, the running time for a RenderMan shader can be arbitrarily long. Finally, real-time rendering encourages a different style of shader writing than software rendering.

1.1. Necessary Hardware Features

Why isn't interactive graphics hardware capable of RenderMan, real-time or not? The principle feature lacking on current graphics hardware is floating point. Machines with floating-point have been designed (e.g. PixelFlow), but none have made it to commercial availability. All existing graphics hardware store results and do most computations using clamped fixed point representations. In contrast, the RenderMan shading language has only one numeric representation - floating point. This provides great freedom to the shader writer. Quantities can be expressed in physical units; overflow and loss of precision are occasional problems, not something you worry about on every line of source; you can even have an enormous range in scale and precision across a single surface.

At the time [Peercy00] was written, most hardware also lacked the ability to look up texture results based on previous computations. Many current graphics systems support either pixel texture (interpret per-pixel framebuffer color as texture coordinates) or dependent texture (interpret previous texturing results as new texture coordinates). Either of these extensions is sufficient to allow RenderMan's indirect lookups.

If the necessary hardware capabilities were present, it would be completely possible to have a hardware accelerated RenderMan. We have demonstrated this using a software implementation of OpenGL (a modified version of the SGI OpenGL sample implementation). This modified OpenGL uses floating point for all computations and storage, it supports pixel texture and color matrix OpenGL extensions, and base OpenGL 1.2. This demo system translates any RenderMan shader into multiple OpenGL rendering passes.

1.2. Arbitrary running time

The RenderMan shading language is similar in structure to C. Like C, it is easy to write programs

that never stop. A shader that never finishes isn't terribly interesting on any rendering system, software or real-time. However, shaders can (and sometimes are!) thousands of lines long or loop for thousands of iterations.

More concretely, even if we had the ability to run RenderMan on graphics hardware we couldn't guarantee that **all** RenderMan shaders would run in real-time. Some

1.3. Programming style

The final factor preventing real-time RenderMan is shader programming style. It isn't that real-time shaders can't be written in RenderMan, but even with all the necessary hardware features, even perfectly ordinary RenderMan shaders may not run in real-time. Shaders written for real-time rendering typically rely heavily on stuffing as much computation as possible into the values stored in each texture, leaving the shader to combine these textures in interesting ways. Shaders written for RenderMan do use texture, but they can also use arbitrary computation. That's part of what makes the shading language so powerful.

2. Sample RenderMan shader

An example is worth a thousand words. This simple RenderMan shader creates a simple beach ball. The ball appearance is entirely procedurally generated, there no textures are used. This example demonstrates both how, given the right extensions, translating a RenderMan shader to multi-pass OpenGL can by quite straightforward, and how this doesn't necessarily give real-time RenderMan.

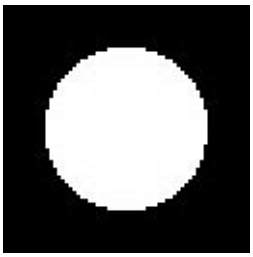
```
surface
beachball(
    uniform float Ka = 1, Kd = 1;
    uniform float Ks = .5, roughness = .1;
    uniform color starcolor = color (1,.5,0);
    uniform color bandcolor = color (1,.2,.2);
    uniform float rmin = .15, rmax = .4;
    uniform float npoints = 5;
     )
{
```

```
    color Ct;
    float angle, r, a, in_out;
    uniform float starangle = 2*PI/npoints;
    uniform point p0 = rmax*point(cos(0),sin(0),0);
    uniform point p1 = rmin*
        point(cos(starangle/2),sin(starangle/2),0);
    uniform vector d0 = p1 - p0;
    vector d1;

    angle = 2*PI * s;
    r = .5-abs(t-.5);
    a = mod(angle, starangle)/starangle;

    if (a >= 0.5)
        a = 1 - a;
    d1 = r*(cos(a), sin(a),0) - p0;
    in_out = step(0, zcomp(d0^d1));
    Ct = mix(mix(Cs, starcolor, in_out), bandcolor, step(rmax,r));

    /* specular shading model */
    normal Nf = normalize(faceforward(N,I));
    Oi = Os;
    Ci = Os * (Ct * (Ka * ambient() +
                     Kd * diffuse(Nf)) +
               Ks * specular(Nf,-normalize(I),roughness));
}
```

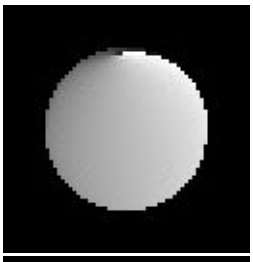
3. Passes for varying computation

Each line of text below is a pass used as part of the varying computation in the above shader. Corresponding images appear to the left, though no image appears for any pass that does not change the framebuffer. No optimizations are included in this example as they can make the correspondence between source code and passes harder to follow. In the thumbnails here, all values are clamped to [0,1] **for display purposes only**; the value stored in the floating point framebuffer or floating point textures still remain unclamped.

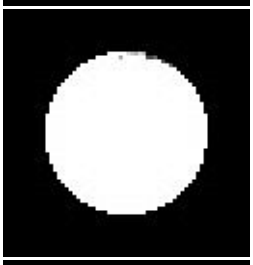
```
// set stencil for masking in later passes
```



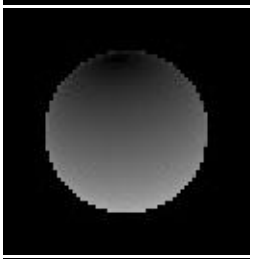
```
// draw geometry with 's' as color
angle = 2*PI * s;
```



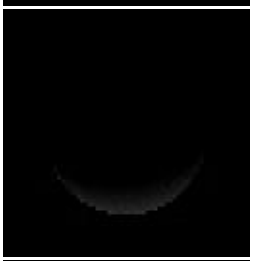
```
// use blend to multiply by 2*PI
angle = 2*PI * s;
// store in texture named "angle"
angle = 2*PI * s;
```



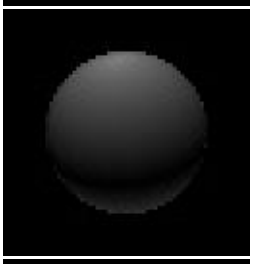
```
// draw geometry with 't' as color
r = .5-abs(t-.5);
```



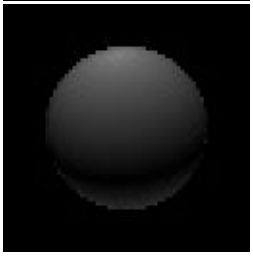
```
// use blend to subtract .5
r = .5-abs(t-.5);
```



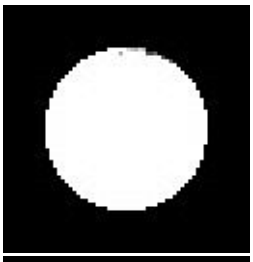
```
// copy through "abs" color table
r = .5-abs(t-.5);
```



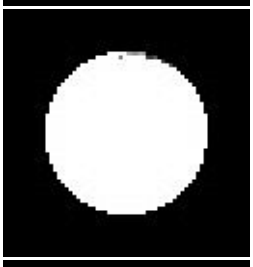
```
// blend: subtract from .5
r = .5-abs(t-.5);
// store in texture named "r"
r = .5-abs(t-.5);
```



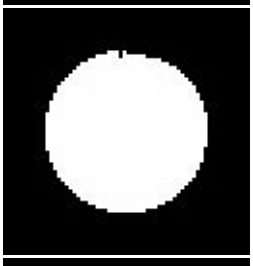
```
// load "angle" from texture
a = mod(angle, starangle)/starangle;
```



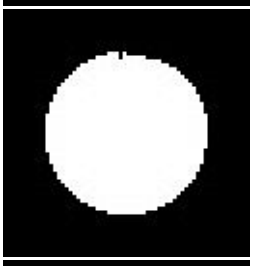
```
// blend: multiply by 1/starangle
a = mod(angle, starangle)/starangle;
```



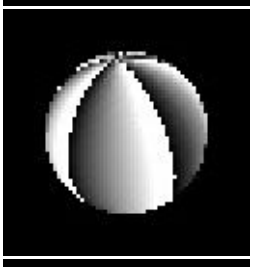
```
// copy through "floor" color table
a = mod(angle, starangle)/starangle;
```



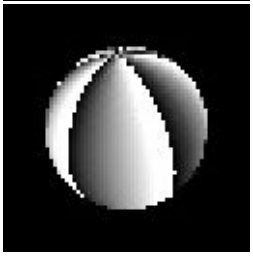
```
// blend: multiply by starangle
a = mod(angle, starangle)/starangle;
```



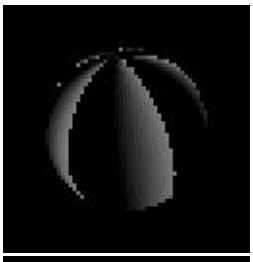
```
// blend: subtract from "angle"
a = mod(angle, starangle)/starangle;
```



```
// blend: multiply by 1/starangle
a = mod(angle, starangle)/starangle;
// store in texture named "a"
a = mod(angle, starangle)/starangle;
// load "a" from texture
if (a >= 0.5)
```

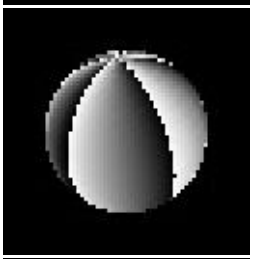


```
// blend: subtract .5
if (a >= 0.5)
```

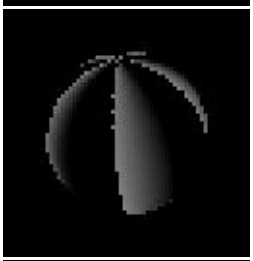


```
// alpha test: set stencil mask
if (a >= 0.5)
```

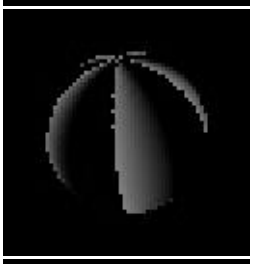
```
// load "a" from texture
    a = 1 - a;
```



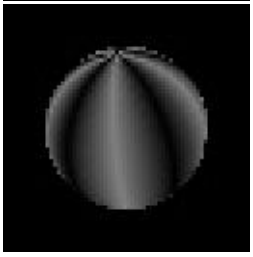
```
// blend: subtract from 1
    a = 1 - a;
```



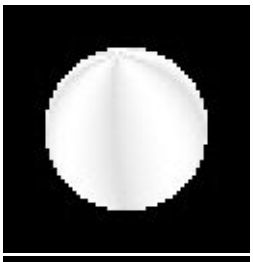
```
// load "a" & combine with stencil
    a = 1 - a;
// store in texture named "a"
    a = 1 - a;
```



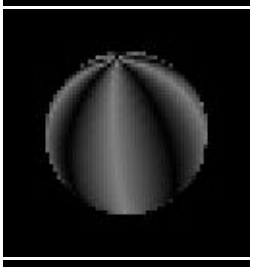
```
// load "a" from texture
d1 = r*(cos(a), sin(a),0) - p0;
```



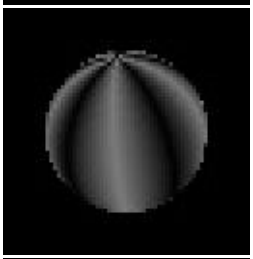
```
// copy through "cos" color table
d1 = r*(cos(a), sin(a),0) - p0;
// store in texture named "ftemp0"
```



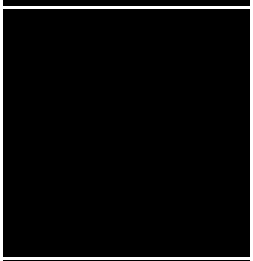
```
// load "a" from texture
d1 = r*(cos(a), sin(a),0) - p0;
```



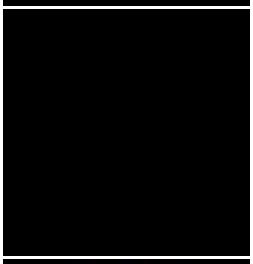
```
// copy through "cos" color table
d1 = r*(cos(a), sin(a),0) - p0;
// store in texture named "ftemp1"
```



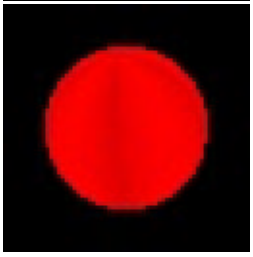
```
// load constant value of 0
d1 = r*(cos(a), sin(a),0) - p0;
```



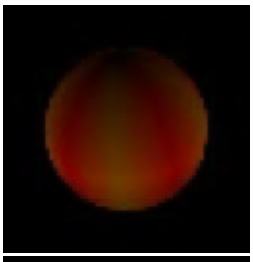
```
// load "ftemp0" into red
d1 = r*(cos(a), sin(a),0) - p0;
```



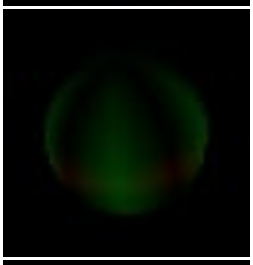
```
// load "ftemp1" into green
d1 = r*(cos(a), sin(a),0) - p0;
```



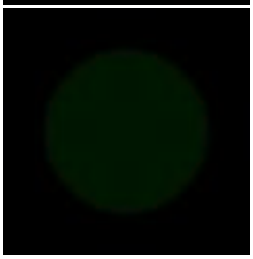
```
// blend: multiply by texture "r"
d1 = r*(cos(a), sin(a),0) - p0;
```



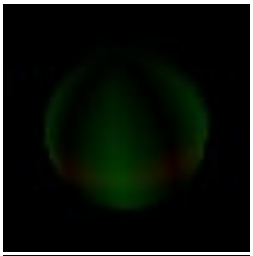
```
// blend: subtract uniform p0
d1 = r*(cos(a), sin(a),0) - p0
// store in texture named "d1"
d1 = r*(cos(a), sin(a),0) - p0
```



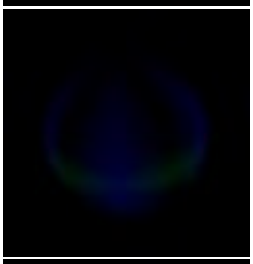
```
// load uniform d0
in_out = step(0, zcomp(d0^d1));
// color matrix: store in yzx order in "ctemp0"

// color matrix: store in zxy order in "ctemp1"
```



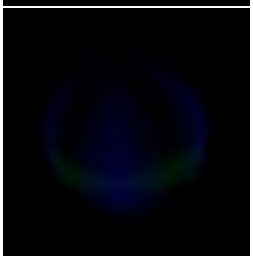
```
// load "d1" from texture
in_out = step(0, zcomp(d0^d1));
// color matrix: store in yzx order in "ctemp2"
```



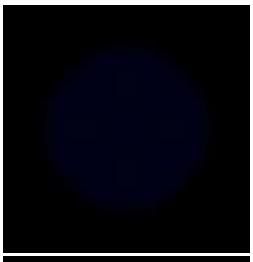
```
// color matrix: shuffle to zxy order
in_out = step(0, zcomp(d0^d1));
```



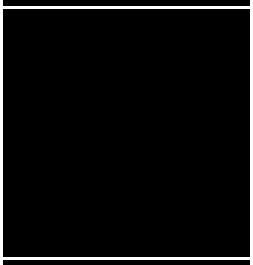
```
// blend: multiply by "ctemp0"
in_out = step(0, zcomp(d0^d1));
// store back into "ctemp0"
```



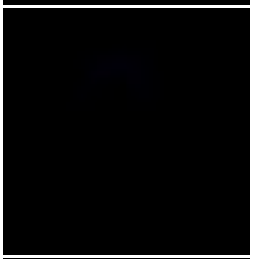
```
// load "ctemp1" from texture
in_out = step(0, zcomp(d0^d1));
```



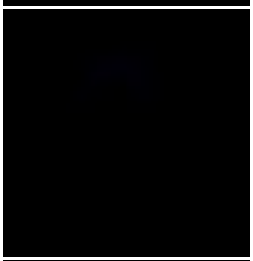
```
// blend: multiply by "ctemp2"
in_out = step(0, zcomp(d0^d1));
```



```
// blend: subtract "ctemp0"
in_out = step(0, zcomp(d0^d1));
```



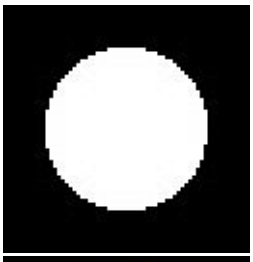
```
// blend: subtract "ctemp0"
in_out = step(0, zcomp(d0^d1));
```



```
// color matrix: copy z to all channels
in_out = step(0, zcomp(d0^d1));
// blend: subtract 0 (to shift step)
in_out = step(0, zcomp(d0^d1));
```

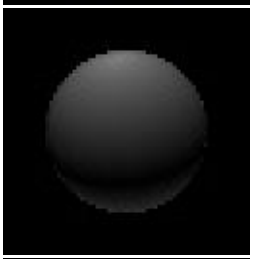
```
// copy through "step" color table
in_out = step(0, zcomp(d0^d1));
// store in texture named "in_out"
in_out = step(0, zcomp(d0^d1));
```

```
// load uniform Cs
...mix(Cs, starcolor, in_out)...
// load "in_out" into alpha
...mix(Cs, starcolor, in_out)...
```

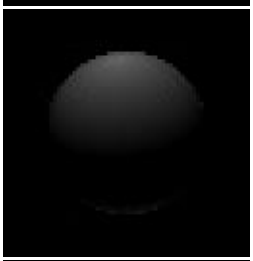


```
// blend: mix Cs and starcolor
...mix(Cs, starcolor, in_out)...
// store in texture named "ctemp0"
```

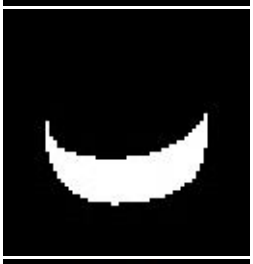
```
// load "r" from texture
...mix(..., bandcolor, step(rmax,r));
```



```
// blend: subtract from rmax
...mix(..., bandcolor, step(rmax,r));
```



```
// copy through "step" color table
...mix(..., bandcolor, step(rmax,r));
// store in texture named "ftemp0"
```

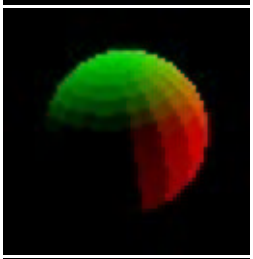


```
// load "ctemp0"
...mix(..., bandcolor, step(rmax,r));
// load "ftemp0" into alpha
...mix(..., bandcolor, step(rmax,r));
```

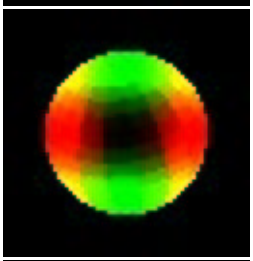
```
// blend: mix with bandcolor
...mix(..., bandcolor, step(rmax,r))
// store in texture named "Ct"
Ct = mix(...);
```

```
// draw geometry, with 'I' as color
...normalize(faceforward(N,I));
// store in texture named "ctemp0"
```

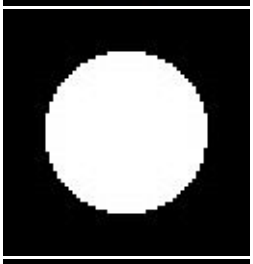
```
// draw geometry, with 'Ng' as color
...normalize(faceforward(N,I));
```



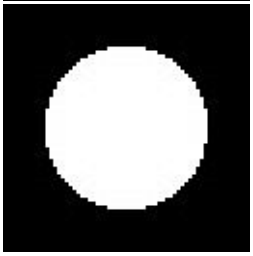
```
// blend: multiply by texture "ctemp0"
...normalize(faceforward(N,I));
```



```
// color matrix: add x+y+z
...normalize(faceforward(N,I));
```

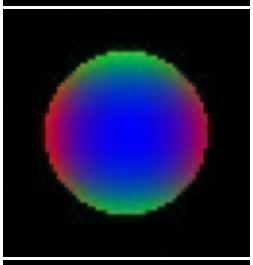


```
// copy through "flip" color table
...normalize(faceforward(N,I));
```

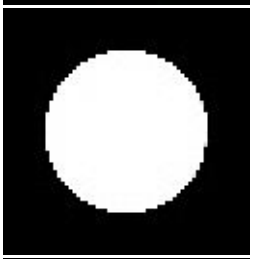


```
// blend: draw 'N' & multiply
...normalize(faceforward(N,I));
// store in texture named "ctemp0"
```

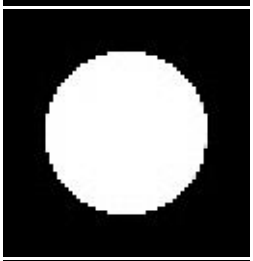
```
// blend: multiply (to square)
normal Nf = normalize(...);
```



```
// color matrix: sum channels
normal Nf = normalize(...);
```

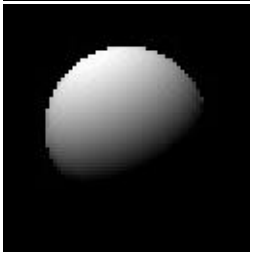


```
// copy through "invsqrt" color table
normal Nf = normalize(...);
```

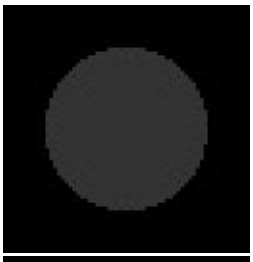


```
// blend: multiply by texture "ctemp0"
normal Nf = normalize(...);
// store in texture named "Nf"
normal Nf = normalize(...);
```

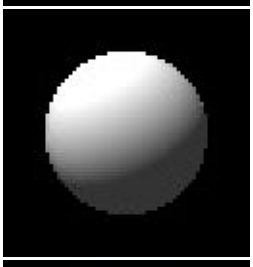
```
// Lighting passes omitted
Ci = ...(... + Kd * diffuse(Nf))...
// store in texture named "ctemp0"
```



```
// Lighting passes omitted
Ci = ...(Ka * ambient() + ...)...
```

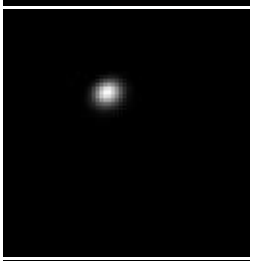


```
// blend: add "ctemp0"
Ci = ...(Ka * ambient() + ...)...
```

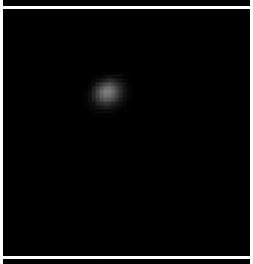


```
// blend: multiply by "Ct"
Ci = ...Ct * (...)...
// store in texture named "ctemp0"
```

```
// Lighting passes omitted
Ci = ...Ks * specular(...));
```



```
// blend: multiply by uniform 'Ks'
Ci = ...Ks * specular(...));
```



```
// blend: add "ctemp0" (diffuse & ambient)
Ci = Os * (...);
// blend: multiply by Os
Ci = Os * (...);
// load "Ci" outside object using stencil
Ci = Os * (...);
// store combined Ci into texture named "Ci"
Ci = Os * (...);
```