Compiler Technology in Open Shading Language

Larry Gritz
Sony Pictures Imageworks
Open Shading Language (OSL)

- Designed for physically-based GI
- Scales to production use
- A language spec that could be used by any renderer
- A library that can be embedded in CPU renderers
- Open source
- In production now!
Motivation

• (Alice in Wonderland images omitted)
What’s wrong with shaders

- Black boxes, can’t reason about them
- Can’t sample, defer, or reorder
- Suboptimal for a modern ray tracer
- Units are sloppy, hard to be physically correct
- If C/C++: difficult, versionitis, can crash, hard to globally optimize.
- Hardware dependence & limitations
Radiance closures

• OSL shaders don’t return colors
• Return a symbolic rep that can be “run” later
  – Act “as if” they are a radiance value
  – But aren’t evaluated until later
• View independent
• Consistent units (radiance)
• Can be sampled
• Unify reflection, transparency, emission
OSL goals

• Similar to RSL/GSL, but evolved & easier
• Separate description vs implementation
  • End versionitis nightmare
  • Late-stage optimization
  • Hide renderer internals
• No crashing, NaN, etc.
• Allow multiple back-ends
• Renderer control of rays / physical shading
  • no light loops or trace calls
• Lazy running of layers
• Closures describe materials/lights
• Automatic differentiation
System workflow

- Compiler (oslC) precompiles individual source modules (.osl) to bytecode (.oso)
- At render time, material networks are assembled
- JIT to x86 to execute
- OSL runtime execution is a library
- Renderer provides a callback interface
Compiling shaders

**gamma.osl**

```osl
shader gamma (color Cin = 1,
              float gam = 1,
              output color Cout = 1)
{
    Cout = pow (Cin, 1/gam);
}
```

**gamma.oso**

```osl
OpenShadingLanguage 1.00

# Compiled by oslc 0.6.0
shader gamma

param color Cin 1 1 1
param float gam 1
oparam color Cout 1 1 1

temp float $tmp1
const float $const2 1

code __main__

# gamma.osl:5
    div $tmp1 $const2 gam
    pow Cout Cin $tmp1
    end
```

Monday, August 15, 2011
First try: SIMD interpreter
- render batches of points at once
- interpret one instruction at a time, all points in lockstep
- analyze to find uniform values
- amortize overhead over the grid
Interpreter vs LLVM

• Works great if batches are big enough
• Easy for primary rays, secondary rays incoherent
• Batches small, too much overhead cohering
Interpreter vs LLVM

• Next try: translate oso into LLVM IR, JIT
  • no exploitation of ‘uniform’ values
  • but no interpreter overhead
  • no need to try to scrape together coherent rays
  • LLVM optimizer

• Generate full IR for some ops
• Others “call” functions, inlined by LLVM
• Generate enter/exit code
• Lazy evaluation of shader nodes
Interpreter vs LLVM

• LLVM vastly outperformed interpreter
• Greatly simplified the entire system
  – other than LLVM dependency
• Simplified renderer, no need for batches
C = texture ("foo.exr", s, t, ...) 

• To properly filter this texture lookup, you want to know how s & t vary over a pixel area.
• dsdx, dsdy, dtdx, dtdy
Most renderers calculate derivatives by:

- Ignoring the problem
- Having “special” texture coordinates
Derivatives

• Most renderers calculate derivatives by:
  • Ignoring the problem
  • Having “special” texture coordinates
  • Computing on grids (Reyes)
Derivatives

Most renderers calculate derivatives by:

- Ignoring the problem
- Having “special” texture coordinates
- Computing on grids (Reyes)
- Shade rays as 3 point grids (Gritz, JGT ‘96)
Derivatives

• Most renderers calculate derivatives by:
  • Ignoring the problem
  • Having “special” texture coordinates
  • Computing on grids (Reyes)
  • Shade rays as 3 point grids (Gritz, JGT ‘96)

• We don’t have grids
• We don’t want to compute extra points
• We want derivs of arbitrary expressions

Monday, August 15, 2011
Automatic differentiation

- Use dual arithmetic (Piponi, JGT 2004)
- Each variable can carry $d/dx$ and $d/dy$ differentials: $x = \{\text{val, dx, dy}\}$
- Define math ops on these dual variables
template<class T>

Dual2<T> operator* (const Dual2<T> &a, 
                    const Dual2<T> &b)
{
    return Dual2<T> (a.val()*b.val(),
                     a.val()*b.dx() + a.dx()*b.val(),
                     a.val()*b.dy() + a.dy()*b.val());
}
Only some symbols need derivs

• Find all data dependencies
  • add R, A, B  →  R depends on A and B
  • “w” args to an op depend on all the “r” args to that op

• Only some ops take derivs of their args
  aastep, area, displace, Dx, Dy, environment, texture

• Mark those symbols as “needing derivatives”

• And so on for their dependencies...

• Careful about connected shader parameters
Derivative ops

• Now we know which symbols need derivs
  – Renderer supplies derivs of (P, u, v, interpolated vars)

• Ops involving them generate deriv IR
  – Shortcut: if the w args of an op don’t need derivs, just do the non-deriv computations

• In practice, ~5% of symbols need to carry derivs

• Total execution cost of arbitrary derivs is <10%
Runtime optimization

• At runtime, we know:
  – layout and connectivity of the shader network
  – parameter values

• So we optimize the shader oso right before LLVM IR
Runtime optimization

• Unconnected, uninterpolated params $\rightarrow$ constants
  – also connected if upstream layer knows output value
• Until A is reassigned, or control flow

• This lets us treat a lot of variables as if they were constant within a basic block.
assign A $constB  

(now we know A’s value)

• Until A is reassigned, or control flow

• This lets us treat a lot of variables as if they were constant within a basic block.
Track “aliasing” within blocks

assign A $constB  
(now we know A’s value)

assign A B  
(now we know A == B)

• Until A is reassigned, or control flow

• This lets us treat a lot of variables as if they were constant within a basic block.
Constant folding
Constant folding

add A $constB $constC
assign A $constD
Constant folding

add A $\text{constB} \: \text{constC}$

assign A $\text{constD}$

add A B $\text{const0}$

assign A B

Monday, August 15, 2011
Constant folding

add A $\text{constB}$ $\text{constC}$

add A B $\text{const0}$

div A A $\text{const1}$

assign A $\text{constD}$

assign A B

nop

Monday, August 15, 2011
Constant folding

add A $\text{constB} \quad$ $\text{constC}$

assign A $\text{constD}$

add A B $\text{const0}$

assign A B

div A A $\text{const1}$

nop

mul A B $\text{const0}$

assign A $\text{const0}$
Useless op elimination

add A A θ         nop
Useless op elimination

add A A 0
add A A C
sub A A C

nop
nop

Monday, August 15, 2011
Useless op elimination

add A A 0       nop
add A A C       nop
sub A A C
assign A B      nop (B is an alias of A)
Useless op elimination

add A A 0        nop

add A A C        nop

sub A A C

assign A B        nop  (B is an alias of A)

assign A B        nop  (A & B have the same value)
Runtime optimization

• Dead code elimination
  – entire conditionals, loops
  – assignments to variables that aren’t used again
Runtime optimization

• Dead code elimination
  – entire conditionals, loops
  – assignments to variables that aren’t used again

• Dead variable elimination
Runtime optimization

• Dead code elimination
  – entire conditionals, loops
  – assignments to variables that aren’t used again

• Dead variable elimination

• Dead shader parameter/output elimination
Runtime optimization

- Dead code elimination
  - entire conditionals, loops
  - assignments to variables that aren’t used again
- Dead variable elimination
- Dead shader parameter/output elimination
- Dead shader layer elimination
Runtime optimization

- Dead code elimination
  - entire conditionals, loops
  - assignments to variables that aren’t used again
- Dead variable elimination
- Dead shader parameter/output elimination
- Dead shader layer elimination
- Coalesce temporaries with nonoverlapping lifetimes
Runtime optimization results
Runtime optimization results

• Reduce code & symbols 95-98% before LLVM
  – IR gen, LLVM opt, JIT in seconds, not minutes
  – LLVM also optimizes its IR
Runtime optimization results

• Reduce code & symbols 95-98% before LLVM
  – IR gen, LLVM opt, JIT in seconds, not minutes
  – LLVM also optimizes its IR
Runtime optimization results

• Reduce code & symbols 95-98% before LLVM
  – IR gen, LLVM opt, JIT in seconds, not minutes
  – LLVM also optimizes its IR

• 20-25% faster execution than old C shaders
  – and safe! (no buffer overflows, crashes, etc.)
Putting it all together

• (“The Amazing Spider-Man” shot omitted, sorry.)
Some stats: frame 1350

- 43 different shader masters (distinct .osl/osd)
- 1885 shader groups (materials)
- 140,964 shader instances (master + params)
- Average 75 instances per group

Load, runtime opt, LLVM IR/opt/JIT:
- 5m22s across all threads (~26s per thread)
- Out of a 3h22:00 render with 12 threads
- Aside: more time assembling/loading than rendering
Some stats: frame 1350

- Typical shader group pre-optimized:
  - 50-100k ops
  - 20-40k symbols (including temporaries)

- After runtime optimization:
  - 1k-5k ops
  - 100-2k symbols
  - many shader groups eliminated entirely
Some stats: frame 1350

• Texture:
  – 497M texture queries (each of which is a bicubic mipmap lookup, more when anisotropic)
  – ~9500 textures (~6700 with unique texels)
  – 700 GB of texture referenced (not counting dupes)
  – Runtime memory: 500 MB cache
  – www.openimageio.org
This is already old

• I’ve seen shader groups with 1.5M ops
• Not uncommon for >> 1 TB texture referenced
Where are we?

- Our shader library is converted
- Our shader writers are exclusively writing OSL
- All new shows using OSL
  - The Amazing Spider-Man
  - Men in Black 3
  - Oz, the Great and Powerful
  - other things I can’t say
- We’ve ripped out support for C shaders
Open source

- opensource.imageworks.com
- github.com/imageworks/OpenShadingLanguage
- “New BSD” license
- This is really our live development tree!
Main takeaways

- Small domain-specific language
- Separate implementation from description
- LLVM to JIT native code at runtime
- Extensive runtime optimization when network and parameters are known
- Outperforms compiled C shaders
- Open source
Acknowledgements / Q&A

• OSL developers: Cliff Stein, Chris Kulla, Alejandro Conty, Solomon Boulos
• SPI renderer, shader teams, SW management
• Participants on the mail list

• Contact:
  opensource.imageworks.com
  lg@imageworks.com