Python on the GPU: Implications for Many-core

Nicholas C. Bray (nbray1@uiuc.edu)

Acknowledgments

- Ralph Johnson
- UPCRC

What's the Problem?

- Hardware is becoming harder to program
  - CPU programming model unsustainable
  - Parallelism is **not** the problem
- Software is becoming larger
  - We need less code and clearer code
- How can we resolve this tension?

Real-time Rendering Challenges

- Coprocessor programming
  - Limited language
    - Few abstractions
  - Two code bases
    - Functions
    - Data Structures
    - Semantics
  - Glue Code
    - Serialization
    - Relationships between shaders
  - Heterogeneous multi-core? Stream programming?
- Inherently Complex
  - Many algorithms working together
  - Algorithms span CPU and GPU

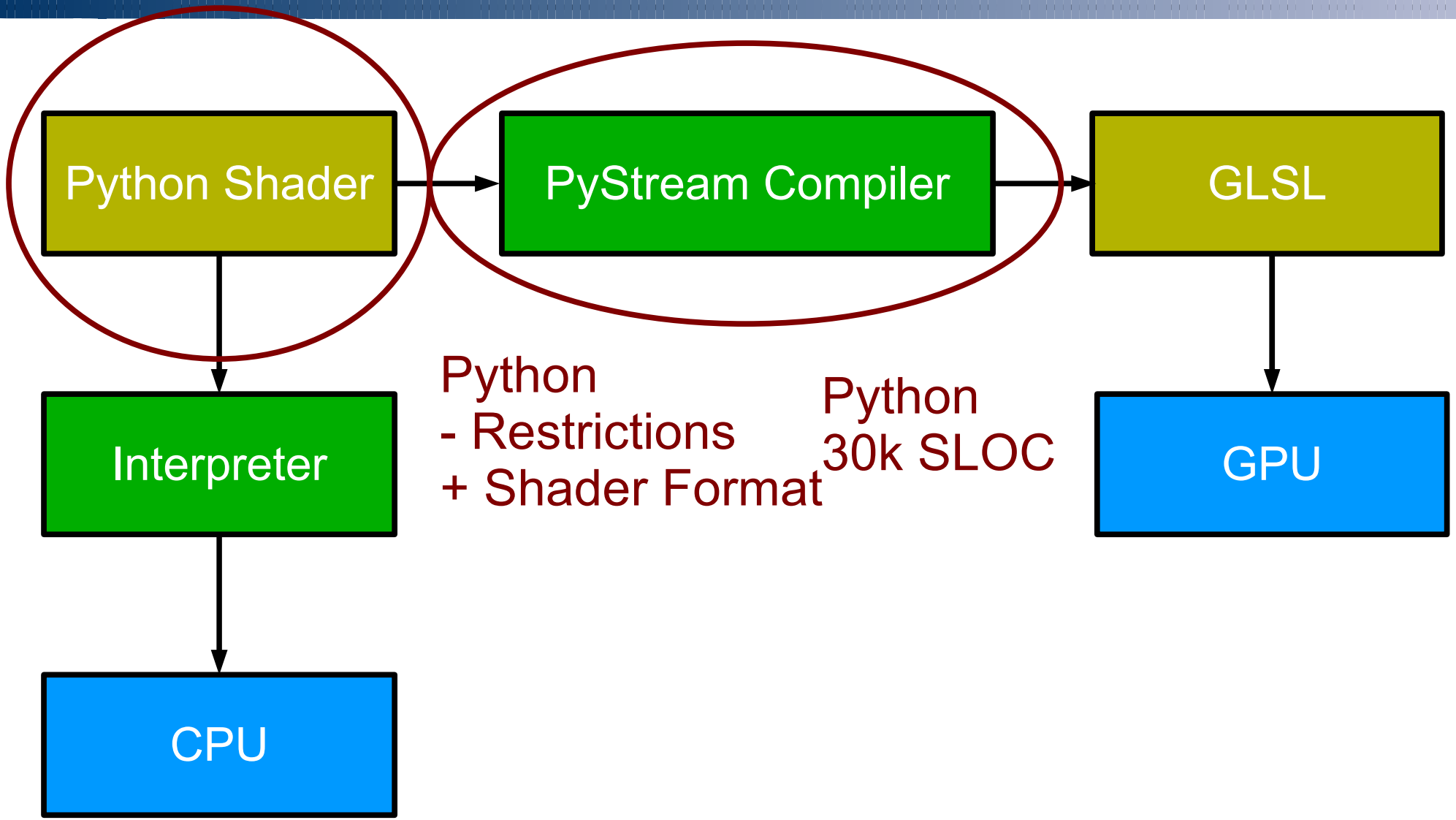
High-level Programming

- Programmers will create abstractions as needed
  - Code generation for shaders
  - Hard to optimize
- Python
  - 7th most popular language
  - Designed for rapid development
  - 100x-1000x slower, yet widely used!
  - Static compilation is an open problem
    - Starkiller, PyPy, Shed Skin

The Big Idea

- Python shaders on the GPU
  - High-level shader programming
  - Unify CPU + GPU code base
  - No glue code
- Python + GPU: a good idea
  - PyGPU
  - Copperhead

What is PyStream?



PyStream Language Restrictions

- At least as expressive as GLSL
- No recursion
  - Function calls
  - Data structures
- Closed world
- Numeric types
- Engineering restrictions
  - Exceptions, keyword args, closures, generators
- Some data structures cannot be emulated
- Most restrictions are applied *after* compilation

Ambient Lighting Shader

```
class FullScreenEffect(object):

    def shadeVertex(self, context, pos, texCoord):
        context.position = pos
        return texCoord,

    def shadeFragment(self, context, texCoord):
        context.colors = (self.process(texCoord),)
```

```
class AmbientPass(FullScreenEffect):

    def process(self, texCoord):
        # Sample the underlying geometry
        g = self.gbuffer.sample(texCoord)

        # Sample the ambient occlusion
        ao = self.ao.texture(texCoord).xyz

        # Calculate the lighting
        ambientLight = self.env.ambientColor(g.normal)*ao

        # Modulate the output
        return vec4(g.diffuse*ambientLight, 1.0)
```

Using a PyStream Shader

```
# Creating the shader
shader = AmbientPass()
shader.gbuffer = gbuffer
shader.env = env
shader.ao = ao

# Using the shader to draw data
draw(shader, posBuffer, texBuffer)
```

PyStream vs. GLSL

Glue Code

```
class CompiledAmbientPass(pystreamruntime.BaseCompiledShader):

        def _bindUniforms(self, shader):
                bogus = tests.full.physics.AmbientPass.ao.__get__(shader)
                self.bind_uniform_sampler2D('samplerGroup2', bogus)
                bogus_0 = tests.full.physics.AmbientPass.gbuffer.__get__(shader)
                bogus_1 = tests.full.physics.GBuffer.surface.__get__(bogus_0)
                self.bind_uniform_sampler2D('samplerGroup0', bogus_1)
                bogus_2 = tests.full.physics.GBuffer.position.__get__(bogus_0)
                self.bind_uniform_sampler2D('samplerGroup1', bogus_2)
                bogus_3 = tests.full.physics.GBuffer.normal.__get__(bogus_0)
                self.bind_uniform_sampler2D('samplerGroup3', bogus_3)
                bogus_4 = tests.full.physics.AmbientPass.env.__get__(shader)
                bogus_5 = tests.full.physics.Environment.cameraToEnvironment.__get__(bogus_4)
                self.bind_uniform_mat4('uni_cameraToEnvironment_mat4', bogus_5)
                bogus_6 = tests.full.physics.Environment.ambientMap.__get__(bogus_4)
                self.bind_uniform_samplerCube('samplerGroup4', bogus_6)

        def bindStreams(self, pos, texCoord):
                self.bind_stream_vec4('inp_io_vec4', pos)
                self.bind_stream_vec2('inp_io_vec2', texCoord)
```

Python Shader

```
class AmbientPass(FullScreenEffect):

        def process(self, texCoord):
                g = self.gbuffer.sample(texCoord)
                ao = self.ao.texture(texCoord).xyz
                ambientLight = self.env.ambientColor(g.normal)*ao
                return vec4(g.diffuse*ambientLight, 1.0)
```

GLSL Vertex Shader

```
out vec2 inp_io_vec2_1;

in vec4 inp_io_vec4;
in vec2 inp_io_vec2;

void main()
{

        gl_Position = inp_io_vec4;
        inp_io_vec2_1 = inp_io_vec2;
}
```

GLSL Fragment Shader

```
uniform sampler2D samplerGroup0;
uniform sampler2D samplerGroup1;
uniform sampler2D samplerGroup2;
uniform sampler2D samplerGroup3;
uniform samplerCube samplerGroup4;
layout(shared) uniform uni
{
        uniform mat4 uni_cameraToEnvironment_mat4;
};

in vec2 inp_io_vec2_1;
out vec4 out_io_vec4;

void main()
{
        vec3 albedo, normal, cameraNormal, ambient, ao;

        albedo = texture(samplerGroup0, inp_io_vec2_1)).xyz;

        normal =  normalize(texture(samplerGroup3, inp_io_vec2_1).xyz);
        cameraNormal = (uni_cameraToEnvironment_mat4*vec4(normal, 0.0)).xyz;
        ambient = texture(samplerGroup4, cameraNormal);

        ao = texture(samplerGroup2, inp_io_vec2_1).xyz;

        out_io_vec4 = vec4(diffuse*ambient*ao, 1.0);
}
```

Example Rendering System

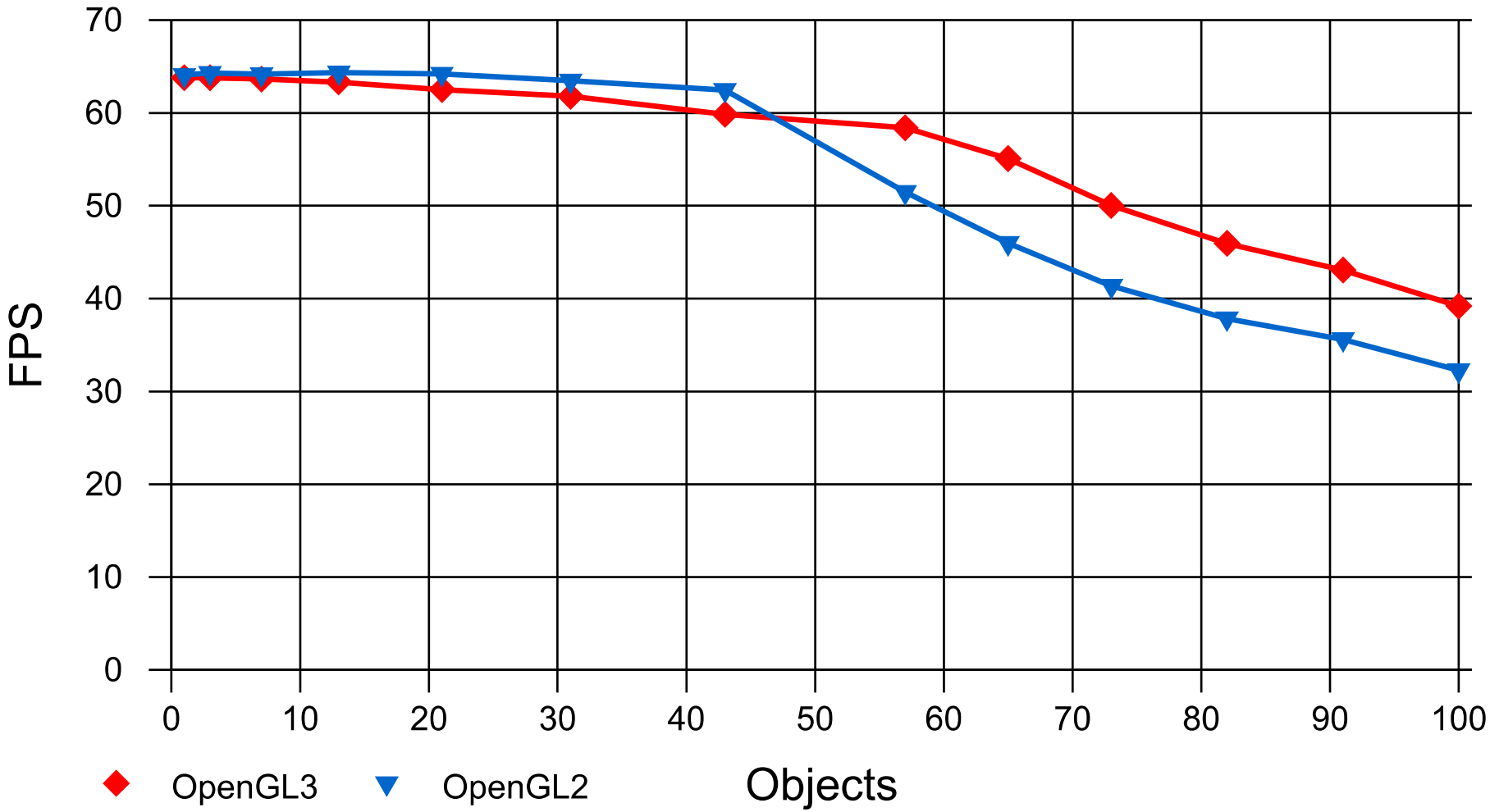
- 100% Python
  - No glue code
  - 147,000x faster
  - No overhead in generated shaders
- Deferred Rendering
- Image Quality
  - Gamma-correct HDR
  - SSAO
  - Parallax Occlusion Mapping
- ~60 fps

Example Shader Programs

Shader	Pixels	GPU	GPU/Pixels	CPU	CPU/Pixels	Improvement
material	0.77	4.33 ms	5.62 ms	169.8 s	220.5 s	39,211x
skybox	0.27	0.22 ms	0.81 ms	9.6 s	35.5 s	43,568x
ssao	1.00	1.44 ms	1.44 ms	444.9 s	444.9 s	308,958x
bilateral	2.00	2.97 ms	1.49 ms	858.2 s	429.1 s	288,956x
ambient	1.00	0.84 ms	0.84 ms	64.1 s	64.1 s	76,310x
light	5.00	4.75 ms	0.95 ms	635.5 s	127.1 s	133,789x
blur	4.00	2.14 ms	0.54 ms	296.8 s	74.2 s	138,692x
post	0.23	2.20 ms	9.57 ms	101.8 s	442.6 s	46,272x
total	14.27	17.59 ms	1.23 ms	2580.7 s	180.8 s	146,712x

AMD Athlon 64 X2 3800+ CPU - NVidia 9800 GT GPU – Windows XP - CPython 2.6.4 – 1024x1024 resolution

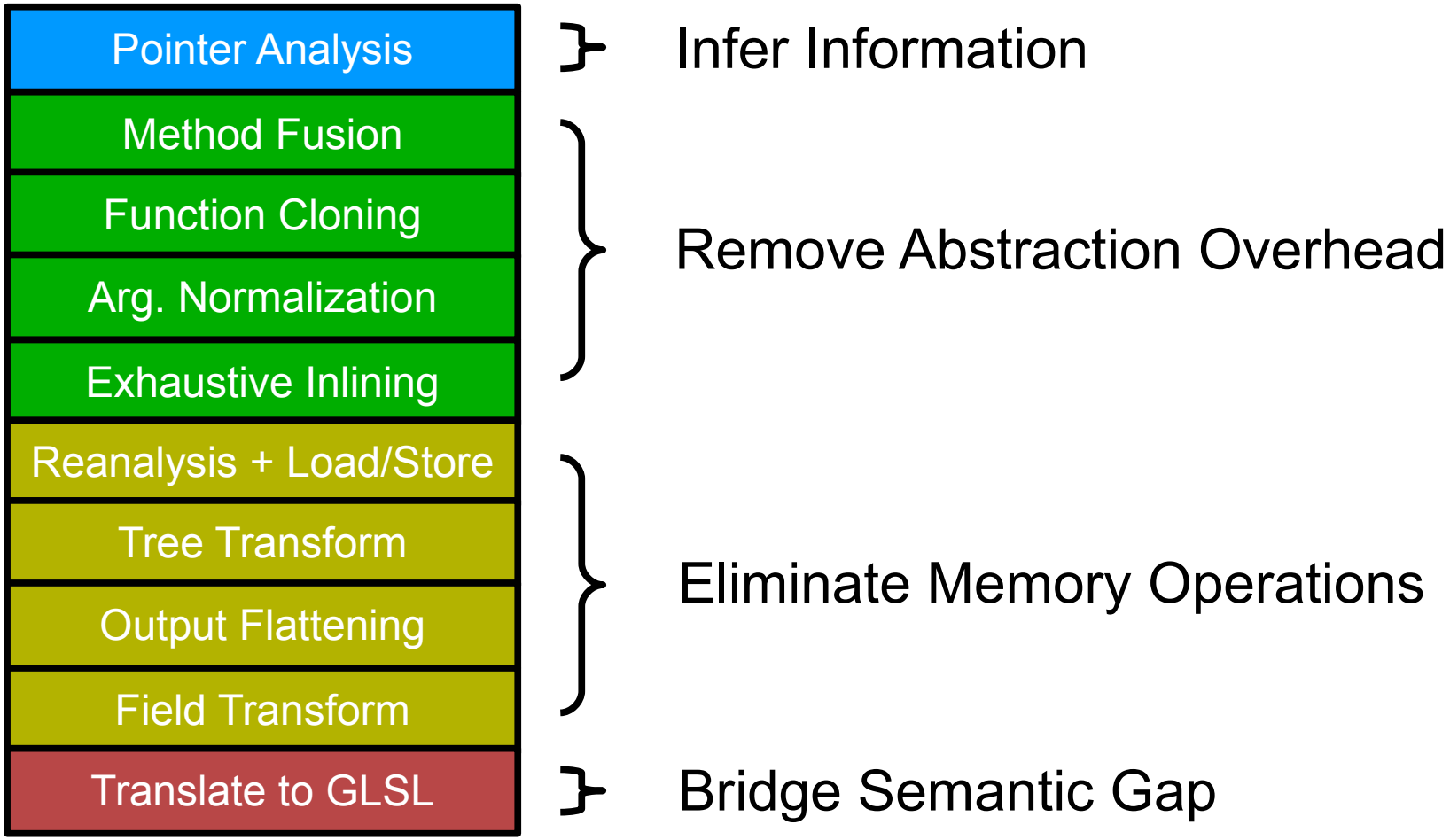
High-level Optimization



Challenges Compiling Python

- Kitchen sink design
- Complicated semantics
  - What is an attribute? (field, ad hoc, method, etc.)
  - What is addition?
  - Operations can be customized
- No initial information
  - No type information or call graph
  - **Highly** polymorphic
- “Declarations” are mutable structures
  - Must disallow dynamic code generation
- Critical information passes through the heap

PyStream Compiler Pipeline



Engineering PyStream

- What was used
  - Pointer analysis
  - Direct references
  - Constant folding
  - Dead code elimination
  - Function cloning
  - Function inlining
  - Load / store elimination
  - Python-specific transformations
    - Method fusion and argument normalization
  - Shader-specific transformations
    - Tree transform, output flattening and field transform

Engineering PyStream

- What did not work
  - Type inference
  - Classic heap sensitivity
  - BDD analysis algorithms
  - Tracked cell analysis
  - Dataflow IR
- What was not even tried
  - Loop analysis

High-level Analysis

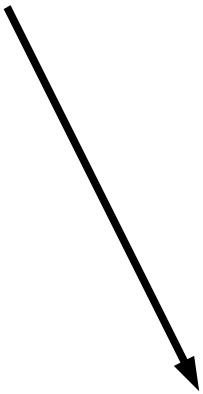
- Used by Starkiller and PyPy
- Attributes are fields
  - … except if they're methods
  - … except if they're overridden
  - Ignore the other cases
- Operations are resolved by the analysis

Low-level Analysis

- Interpreter and run time → analyzed code
  - Interpreter becomes part of the program
  - Add direct calls, memory operations
  - Keep “loops” in the analysis
- Operations → calls
- Model memory at the interpreter level
  - Field vs. method?
- Analysis and optimization are *much* simpler
  - Looks like C + indirections
  - Should scale well
- Interpreter vs. JIT vs. pointer analysis

Low-level Example

```
def simple(a, b):
    return a+b
```



```
def interpreter_add(self, other):
    result = NotImplemented

    # Forward add
    selfcls <load>(self, 'LowLevel', 'type')
    clsDict = <load>(selfcls, 'LowLevel', 'dictionary')
    if <check>(clsDict, 'Dictionary', '__add__'):
        meth = <load>(clsDict, 'Dictionary', '__add__')
        result = meth(self, other)

    if result is NotImplemented:
        # Reverse add
        othercls = <load>(other, 'LowLevel', 'type')
        clsDict = <load>(othercls, 'LowLevel', 'dictionary')
        if <check>(clsDict, 'Dictionary', '__radd__'):
            meth = <load>(clsDict, 'Dictionary', '__radd__')
            result = meth(other, self)

    if result is NotImplemented:
        # Invalid operation
        raise ...

    return result

def simple(a, b):
    interpreter_add(a, b)
```

Ambient Shader - Low Level Analysis

```
def process(self, texCoord):
    g = self.gbuffer.sample(texCoord)
    ao = self.ao.texture(texCoord).xyz
    ambientLight = self.env.ambientColor(g.normal)*ao
    return vec4(g.diffuse*ambientLight, 1.0)
```

```
def process(self, texCoord):
    _0 = <interpreter_getattribute>(self, 'gbuffer')
    _1 = <interpreter_getattribute>(_0, 'sample')
    g = _1(texCoord)
    _2 = <interpreter_getattribute>(self, 'ao')
    _3 = <interpreter_getattribute>(_2, 'texture')
    _4 = _3(texCoord)
    ao = <interpreter_getattribute>(_4, 'xyz')
    _5 = <interpreter_getattribute>(self, 'env')
    _6 = <interpreter_getattribute>(_5, 'ambientColor')
    _7 = <interpreter_getattribute>(g, 'normal')
    _8 = _6(_7)
    ambientLight = <interpreter__mul__>(_8, ao)
    _9 = <interpreterLoadGlobal>(internal_self, 'vec4')
    _10 = <interpreter_getattribute>(g, 'diffuse')
    _11 = <interpreter__mul__>(_10, ambientLight)
    _12 = _9(_11, 1.0)
    return _12
```

Method Fusion

- Fuse getattr + call → method call
  - Requires interprocedural analysis
- High-level information is great!
- High-level information is awful!
  - Compiler vs. programmer
  - Reconstruct when useful

```
m = o.f
m(a, b)
```

→

```
<method o f>(a, b)
```

Python to GLSL

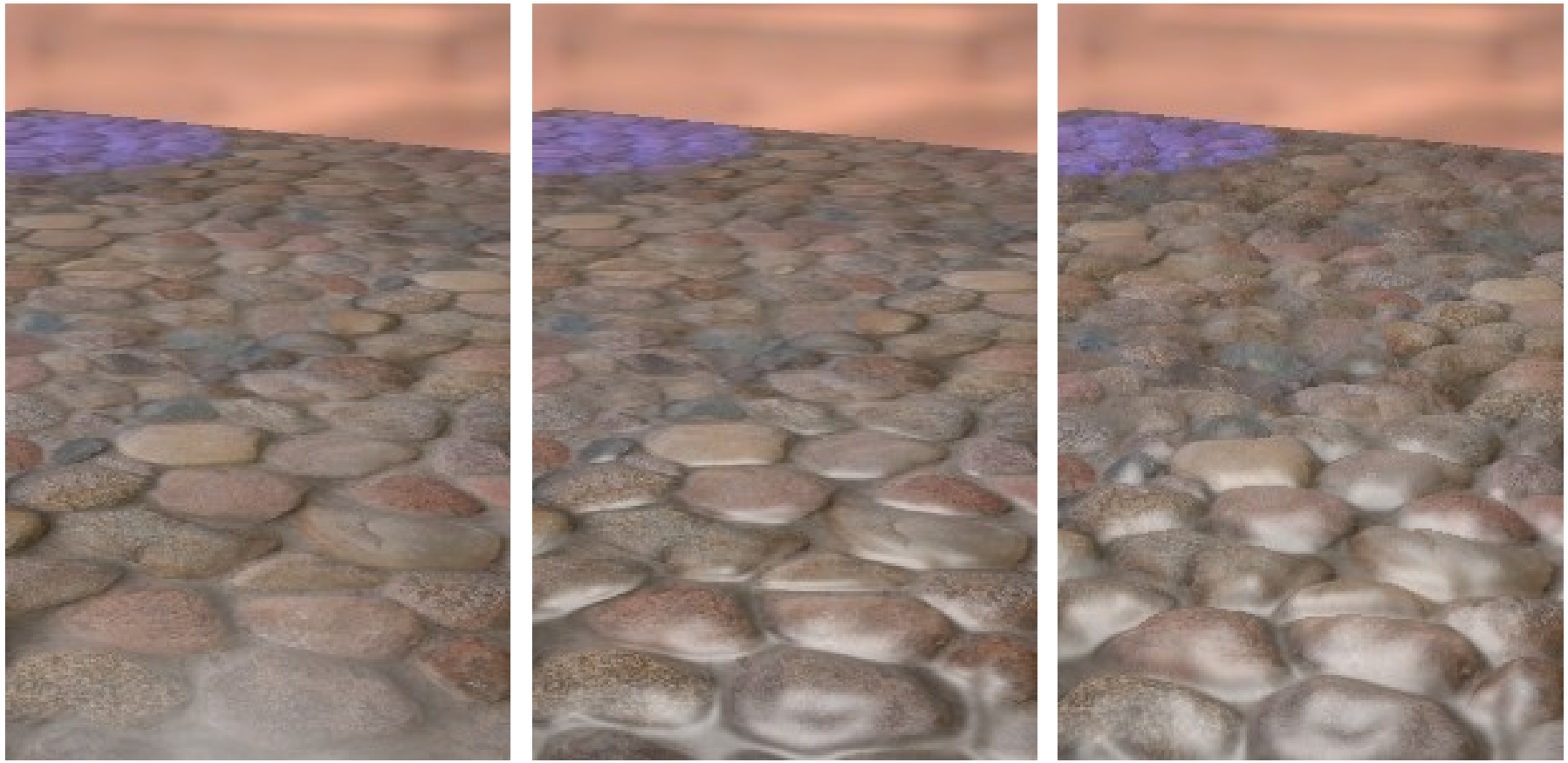
- Simplified by prior optimization
- Copy values wherever possible
- No memory allocation: bound structure size
  - No recursive data structures
  - No variable-size container objects
  - Dictionary lookups are difficult
- Volatility
  - Object modified while held by 2+ references
  - Cannot be soundly copied
  - Does not occur in the example shaders

Bottom Line

- Programmability and performance can co-exist
  - Existing techniques + R&D
  - Work at a lower level when necessary
  - No cookbooks, yet
- Overcoming language/architecture mismatch
  - Easy, 90% of the time?
  - New techniques needed for the rest
    - Change our expectations?
  - Program at an even higher level

Backup sides start here

Object-oriented Polymorphism



PyStream's Pointer Analysis

- What objects can a variable point to?
  - What is the type of a variable?
  - What is the call graph?
  - What memory operations are dependent? *
- Low-level analysis simplifies the problem
  - Almost C, but not quite...
  - Large amounts of polymorphism

Analyizing Polymorphism

- Parametric polymorphism: CPA contexts
  - Implicit “template” functions
- Data polymorphism: extended types
  - Bound method objects
- Ad hoc polymorphism: control sensitivity
  - No types, no function overloading
  - Python idiom: explicit decoding of types

```
def add(a, b):
    return a+b

print add(1, 2)
print add('hello', '!')
```

```
class vec2(object):
    def __init__(self, x, y=None):
        if isinstance(x, vec2):
            assert y is None
            self.x = x.x
            self.y = x.y
        else:
            self.x = x
            self.y = y
```

Novel Transform: Direct References

```
temp = <interpreterLoadGlobal>(internal_self, 'vec4')
return temp(vec3data, 1.0)
```

```
temp = <class 'shader.vec.vec4'>
return temp(vec3data, 1.0)
```

```
return <type_call, <class 'shader.vec.vec4'>>(vec3data, 1.0)
```

Novel Transform: Methods & Args

```
m = o.f
m(a, b)
```

→

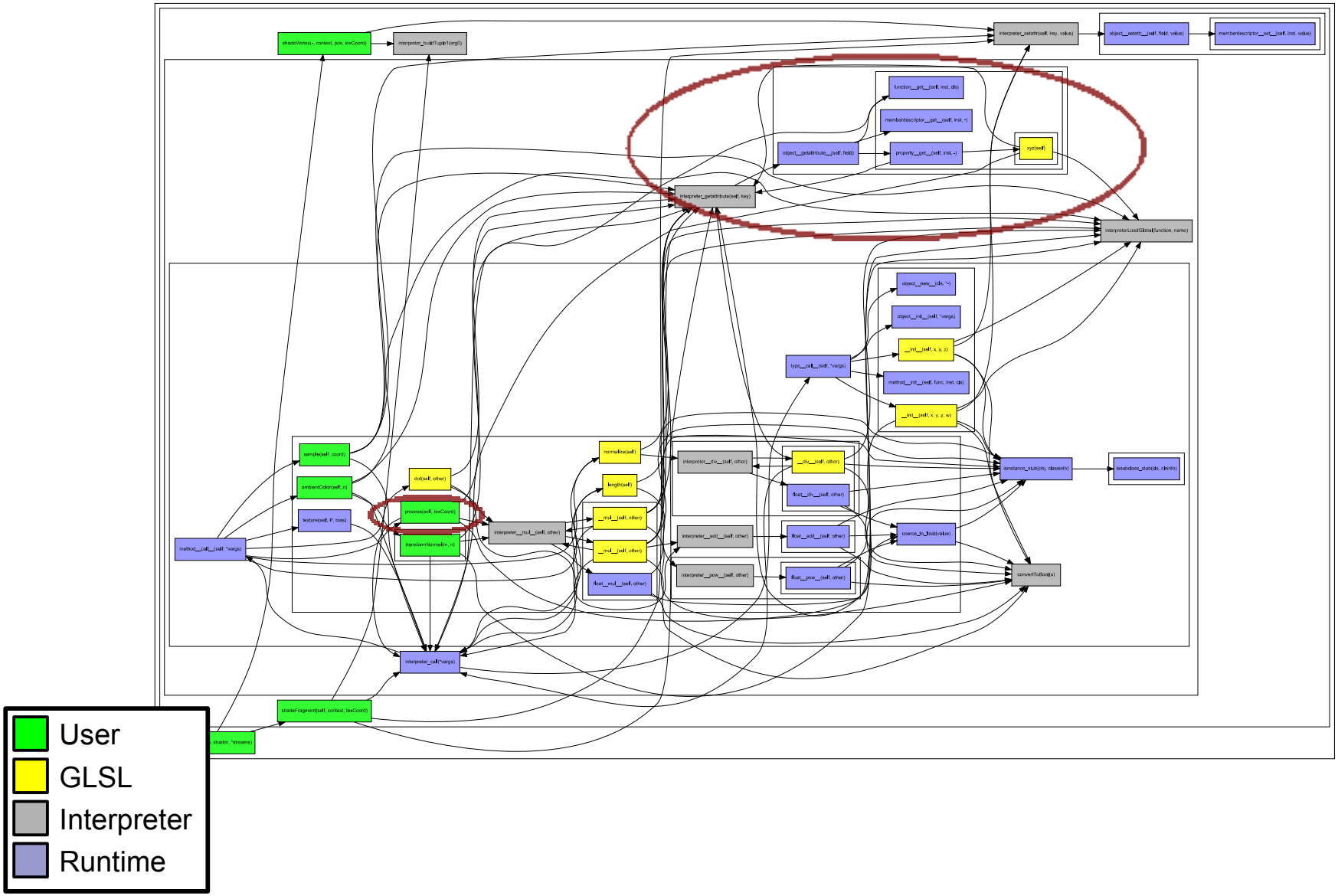
```
<method o f>(a, b)
```

```
def type_call(cls, *vparam):
    obj = cls.__new__(*vparam)
    obj.__init__(*vparam)
    return obj
```

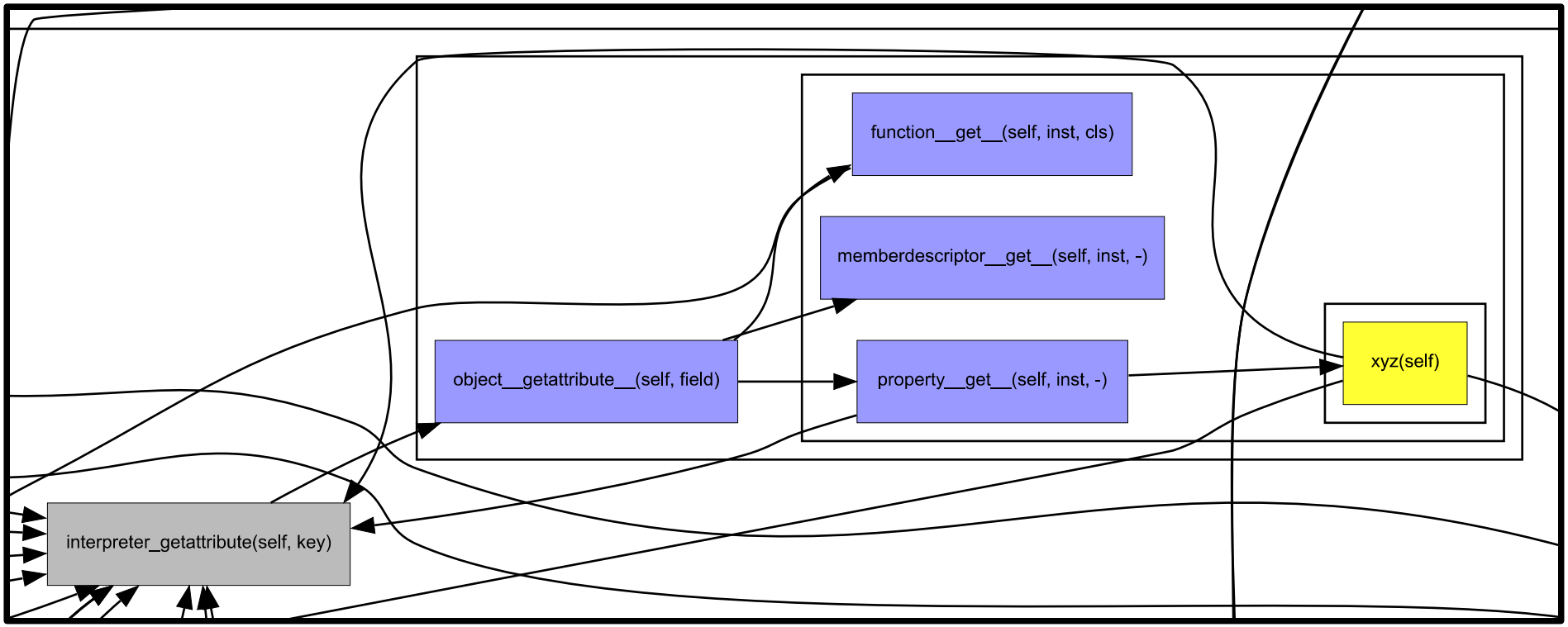
→

```
def type_call(cls, p0, p1):
    obj = cls.__new__(p0, p1)
    obj.__init__(p0, p1)
    return obj
```

Ambient Lighting Shader



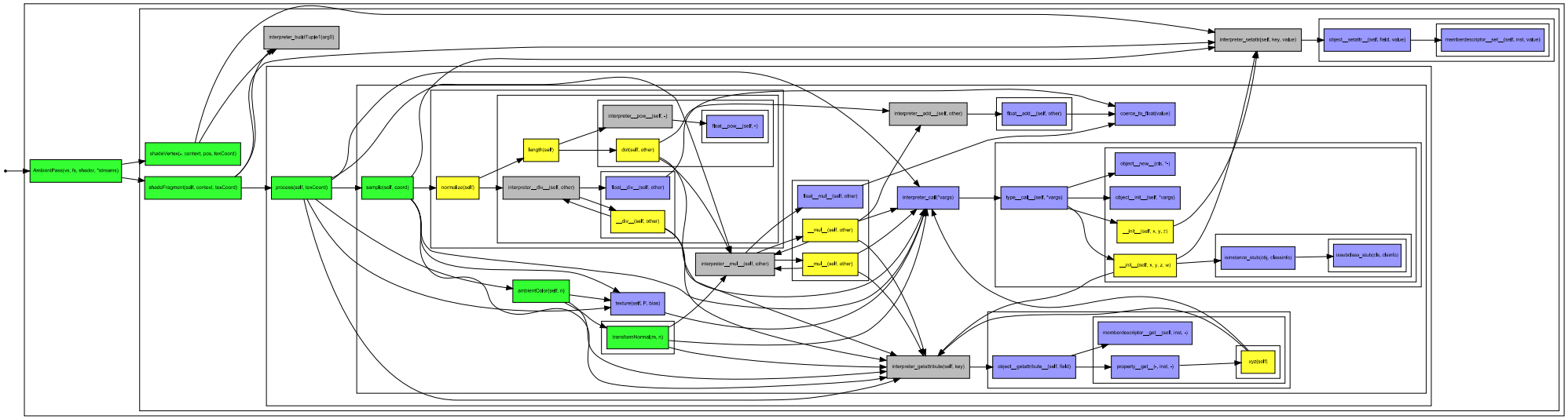
Get Attribute




```
@property
def xyz(self):
    return vec3(self.x, self.y, self.z)
```

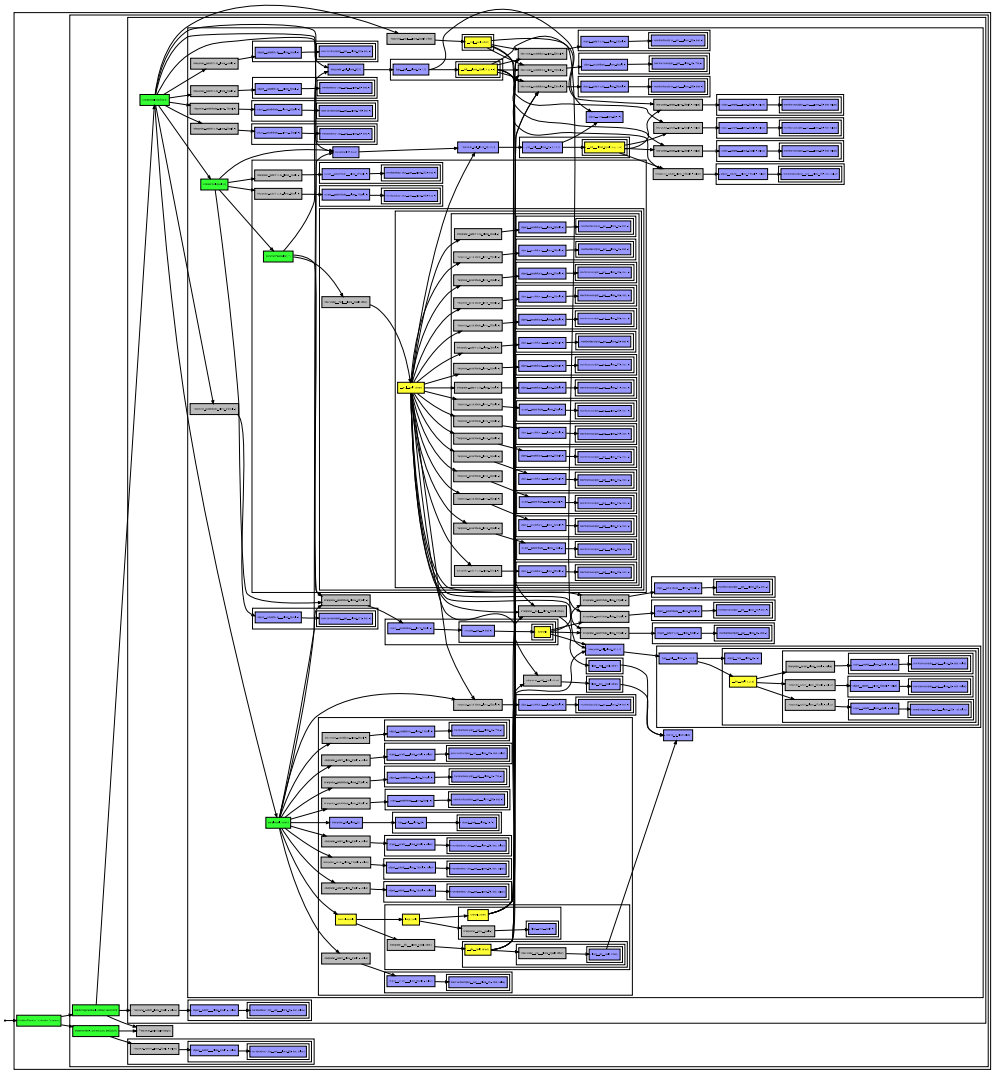
- User
- GLSL
- Interpreter
- Runtime

Ambient Lighting - Optimized



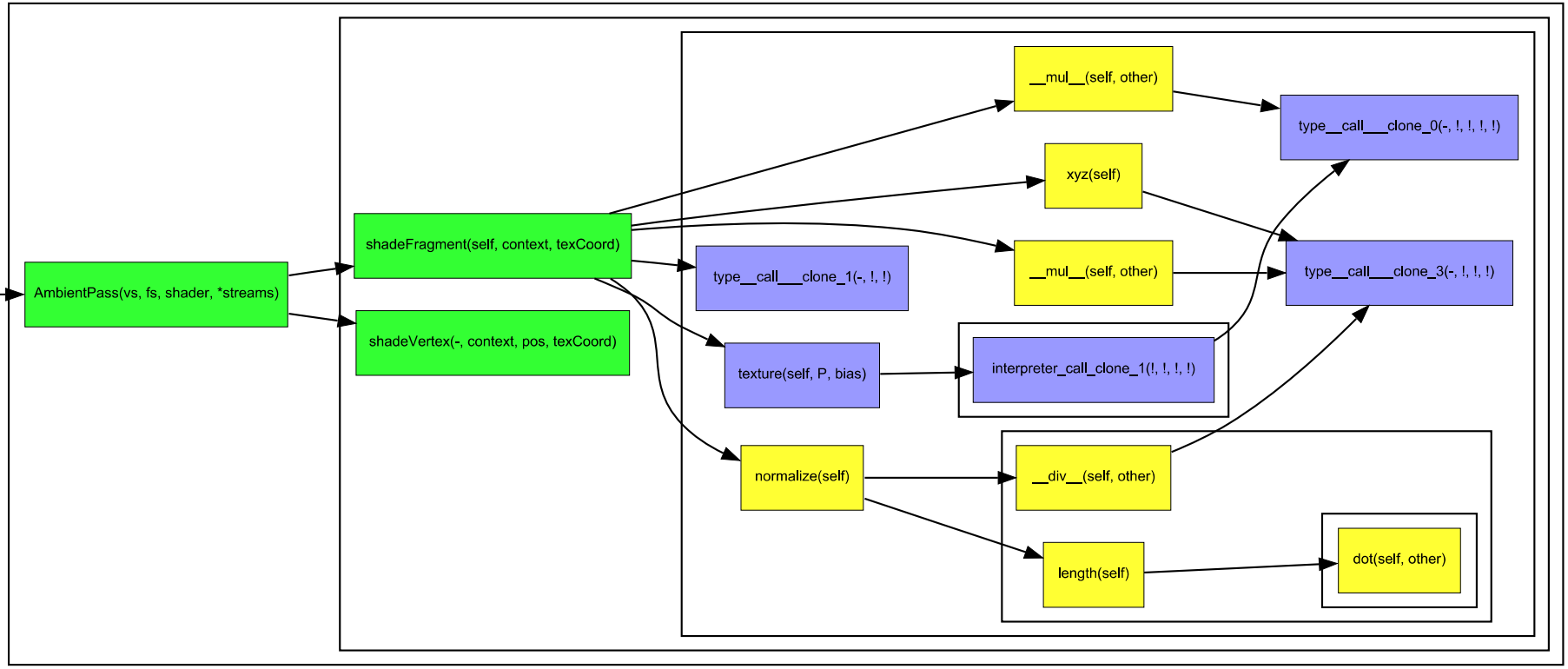
User
GLSL
Interpreter
Runtime

Ambient Lighting - Cloned



User
GLSL
Interpreter
Runtime

Ambient Lighting - Inlined



User
GLSL
Interpreter
Runtime

Eliminating Memory Operations

- Memory operations may remain
  - Reading input structures
  - Building output structures
  - Ambiguous memory operations
  - Allocated in loops, carried through merges
- Immutability → aliasing does not matter
  - Assume inputs are tree shaped
  - Coerce outputs into tree shape
- Transform fields into local variables
  - Unique and mutually exclusive
- Some memory operations can survive

Python to GLSL

- Must emulate references in GLSL
- Emulate reference semantics
  - Inline fields into references
    - No recursive data structures!
  - Inline intrinsic objects into references
  - Copy values between references when possible
- Ambiguous memory operations → array

Volatility

- Values can be indirectly modified
  - Held by more than one reference and modified
- Store in “pools” and hold by index
- Rare in practice?
  - Can ignore side effects of GLSL functions
  - Few memory operations → few side effects
  - SSA + load elimination → uniquely held
  - Coding style: allocate rather than modify

	Multiple Refs	Uniquely Held
Mutable	0	0
Immutable	45	307
Samplers	0	22

What Can Not Be Emulated?

- Data structures of unbounded size
  - Recursive data structures
  - Container objects, most of the time
  - Long integers and strings
  - Resource limits
- Unbounded numbers of volatile values
- Dictionary lookups
  - … unless keys can be explicitly enumerated
- These cases can be compiled away