

# AnySL: Efficient and Portable Shading for Ray Tracing

*Ralf Karrenberg, Dmitri Rubinstein, Philipp Slusallek,  
and Sebastian Hack*

# Motivation

- Setting
  - Realtime ray tracer (packet-based)
- Goal
  - Visually appealing effects in high-performance ray tracer



# Option 1: Native Shading

- Internal
- Renderer language
- **Pro:** Fast!
- **Con:** Flexibility? Portability?
- **Con:** Don't want to make artists write SIMD shaders!

# Option 2: Shading Languages

- External
- Domain-specific languages (e.g. RenderMan, OpenSL)
- **Pro:** Designed for this purpose
- **Pro:** Widely adopted
- **Pro:** Existing shaders can be reused
- **Con:** Have to interpret or write entire compiler toolchain

# Option 3: AnySL

- A Library for high-performance shading
- Interface for the integration of shading languages into rendering engines
- Embedded just-in-time compiler (LLVM)
- Customized linker
- Special code transformation passes

# Option 3: AnySL

- A library for high-performance shading
- An interface for the integration of shading languages into rendering engines
- An embedded just-in-time compiler (LLVM)
- A customized linker
- **Pro:** “Best of options 1 & 2”

# Option 3: AnySL

- A library for high-performance shading
  - An interface for the integration of shading languages into rendering engines
  - An embedded just-in-time compiler (LLVM)
  - A customized linker
- 
- **Pro:** “Best of options 1 & 2”
  - **Con:** Nothing, of course ;-)

# Option 3: AnySL

- A library for high-performance shading
- An interface for the integration of shading languages into rendering engines
- An embedded just-in-time compiler (LLVM)
- A customized linker
  
- **Pro:** “Best of options 1 & 2”
- **Con:** ~~Nothing, of course~~ → Later ;-)

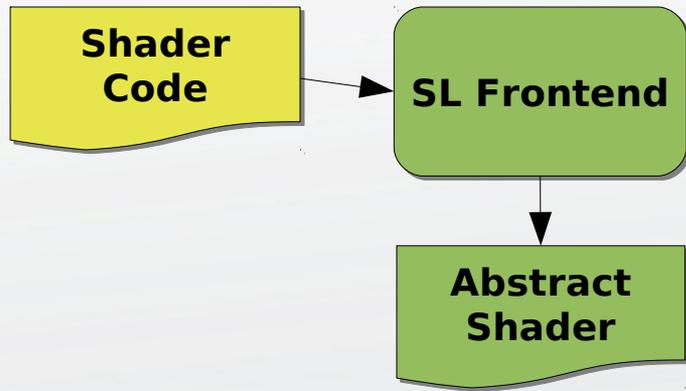
# AnySL: Benefits

- *Efficiency*  
Use existing shading languages in your renderer without sacrificing performance
- *Portability*  
Reuse the same shader in different renderers
- *Simplicity*  
Minimize the integration effort
- *Flexibility*  
Edit, recompile & specialize shaders at runtime

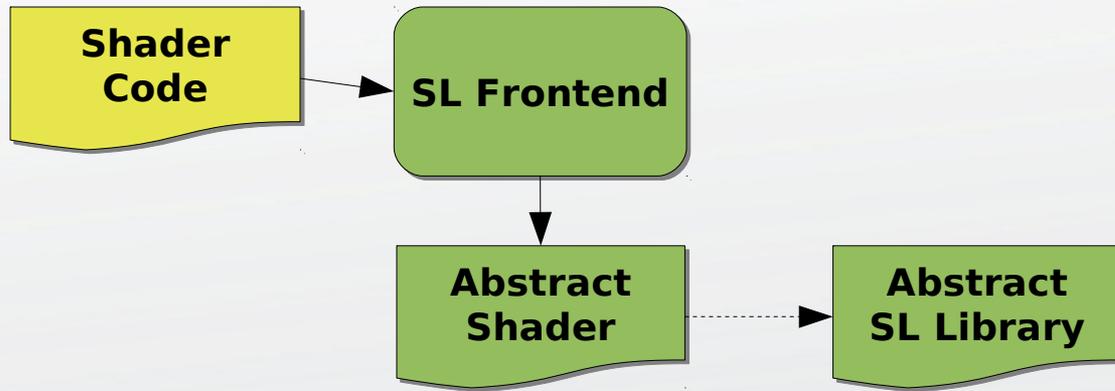
# AnySL: Shader Compilation

Shader  
Code

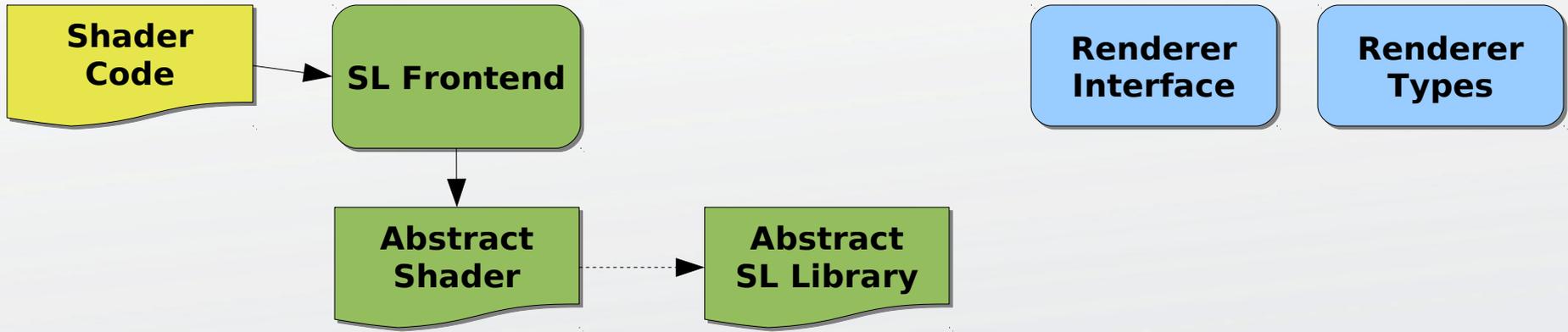
# AnySL: Shader Compilation



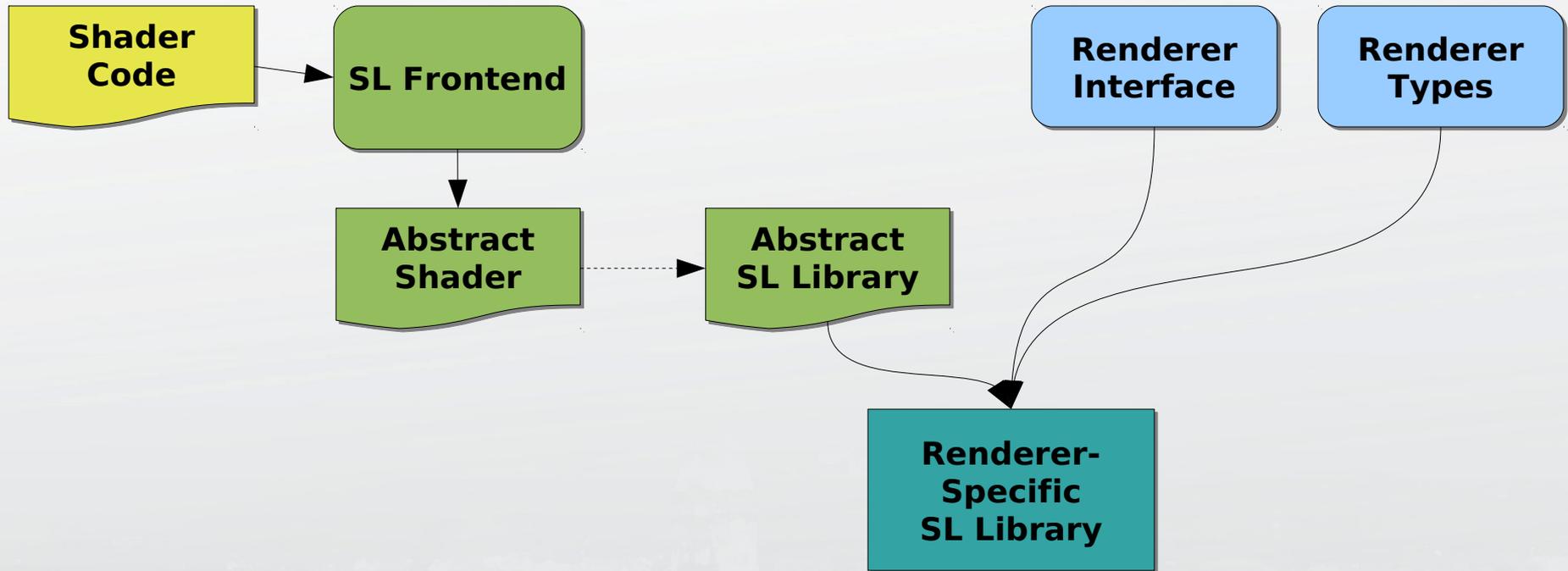
# AnySL: Shader Compilation



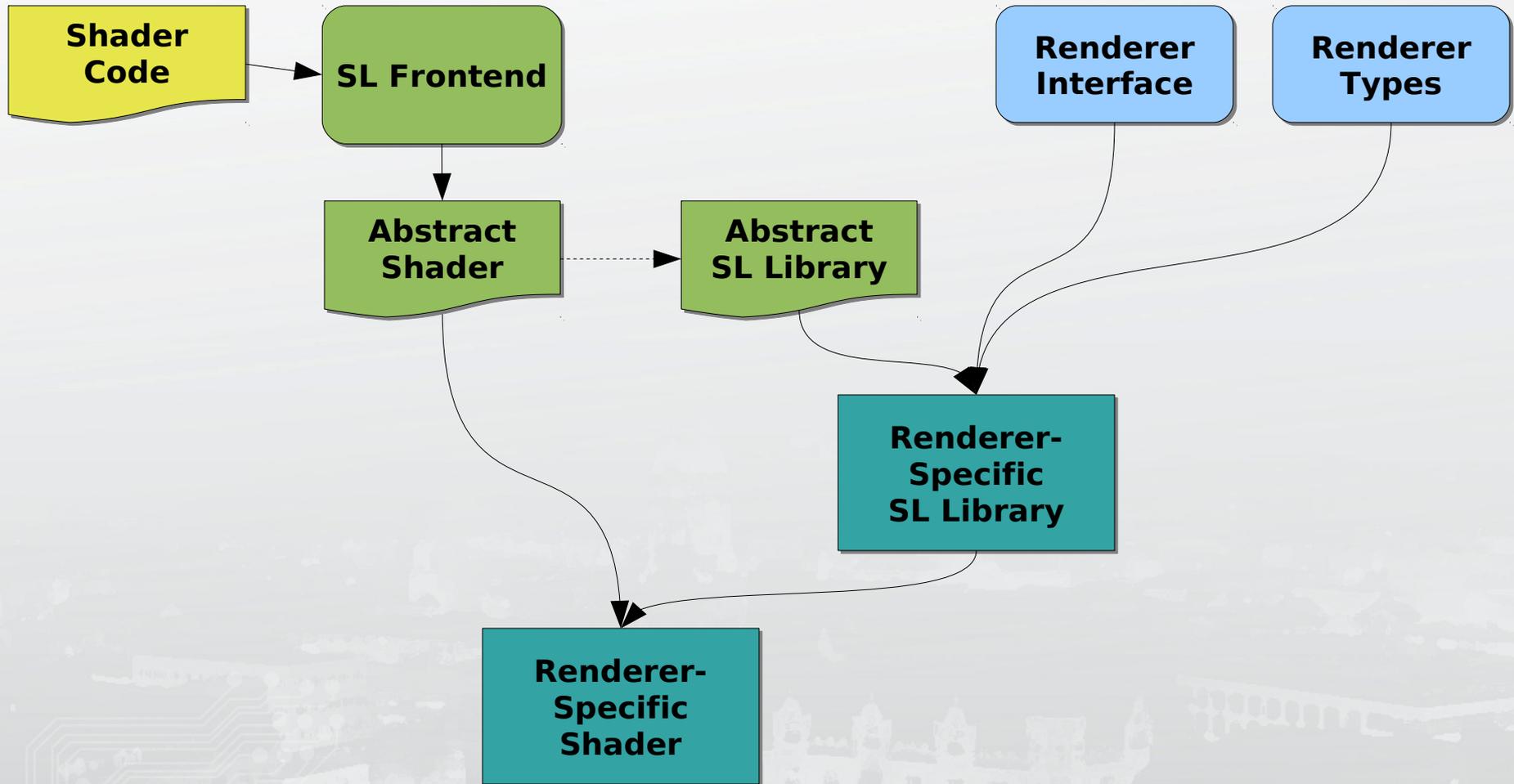
# AnySL: Shader Compilation



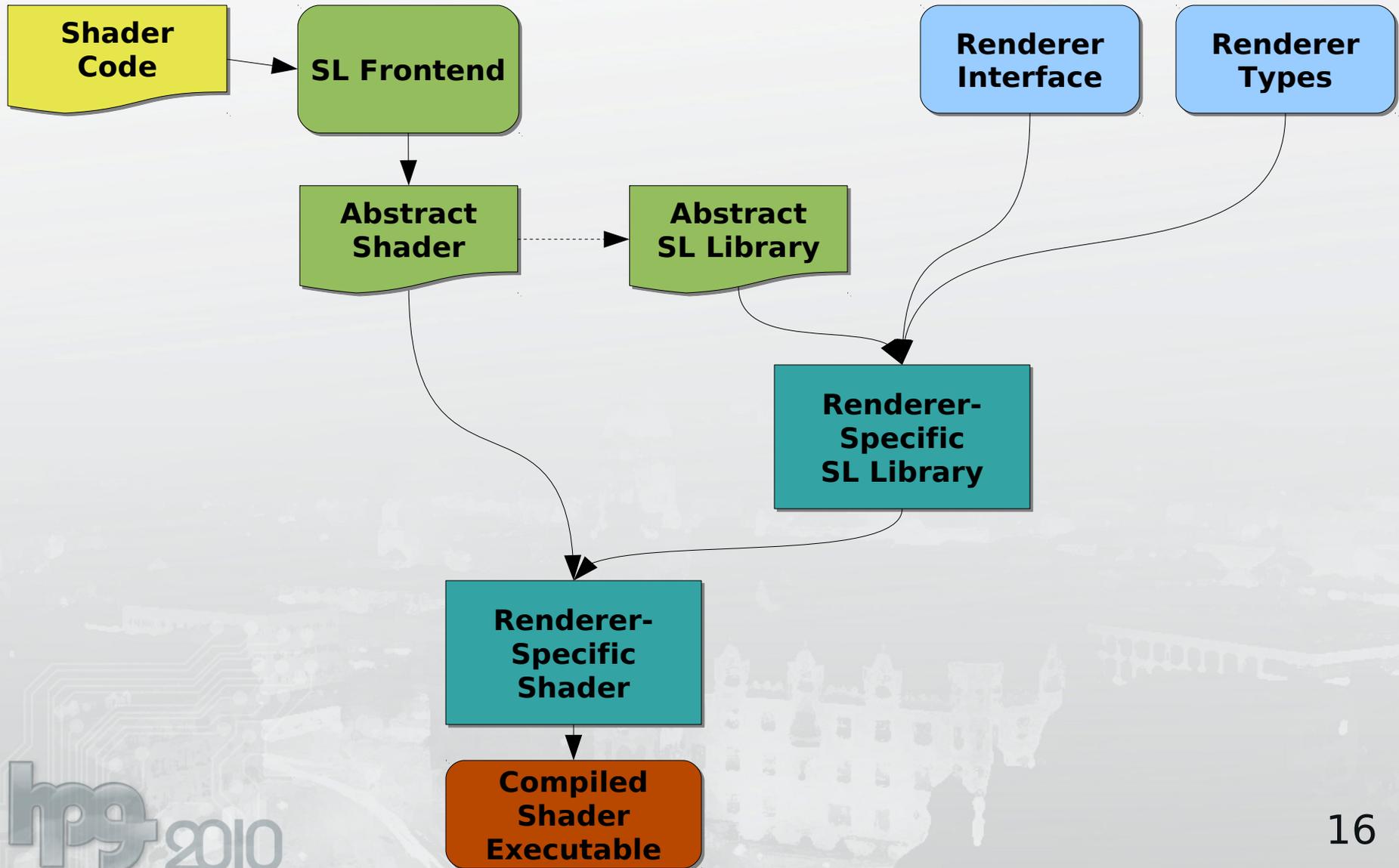
# AnySL: Shader Compilation



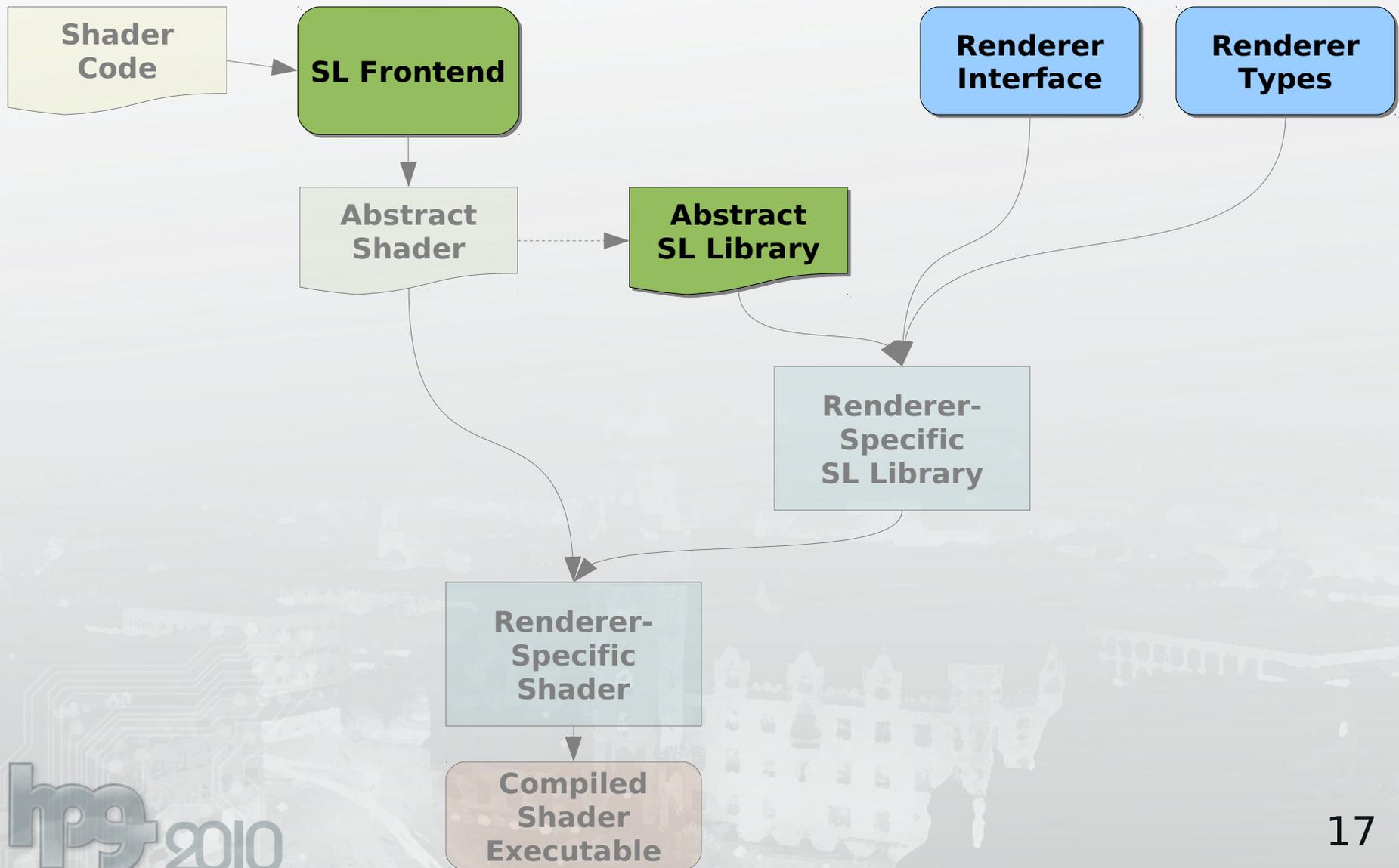
# AnySL: Shader Compilation



# AnySL: Shader Compilation

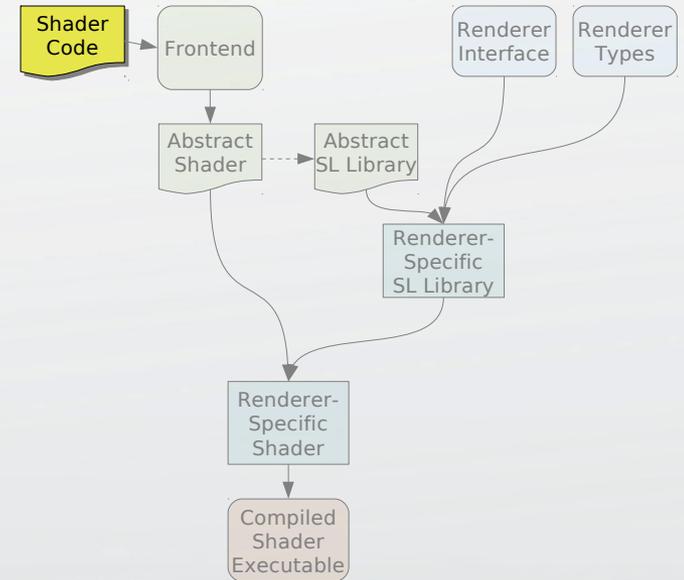


# AnySL: External “Plugins”



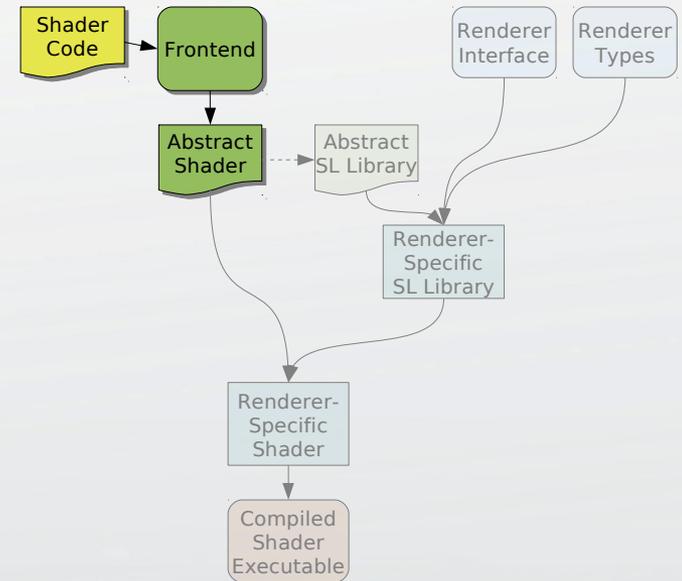
# Example: RenderMan Mirror Shader

```
surface mirror() {  
    vector dir = reflect(I, N);  
    Ci = trace(P, dir);  
}
```



# Subroutine-Threaded Code (STC)

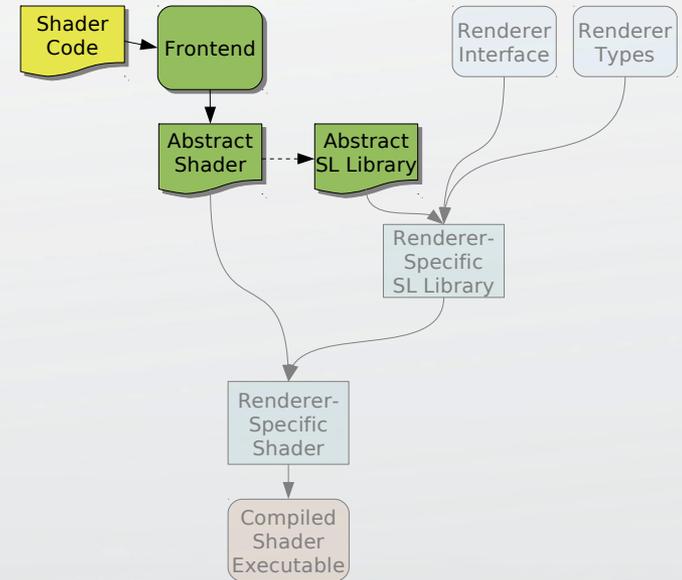
```
void shade(void* context) {  
    void* dir = anysl_rsl_alloc("vector");  
  
    anysl_rsl_reflect(dir, I, N);  
    anysl_rsl_trace(Ci, context, P, reflDir);  
}
```



- All operations encoded in function calls
- Only abstract types
- Represented in LLVM bytecode

# Generic Shading Library

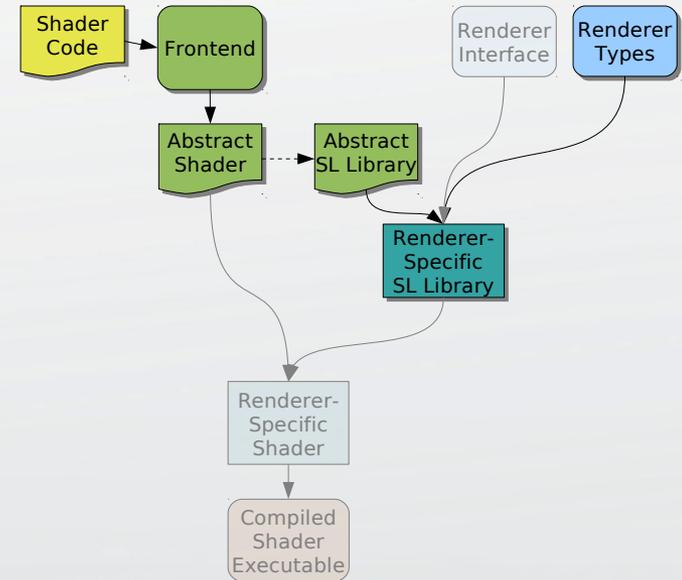
```
void anys1_rsl_reflect(void* RES, void* I, void* N) {  
    void* tmpv = anys1_rsl_alloca("vector");  
    void* tmpf = anys1_rsl_alloca("float");  
    void* c = anys1_rsl_float(2.f);  
  
    anys1_rsl_dot(tmpf, i, n);  
    anys1_rsl_mulff(tmpf, tmpf, c);  
    anys1_rsl_mulvf(tmpv, n, tmpf);  
    anys1_rsl_subvv(res, i, tmpv);  
}
```



# Low-Level Operation Implementation

```
#include "Renderer/Types.h"
```

```
void anys_l_rsl_mulvf(void* RES, void* V, void* F) {  
    Renderer::Vector* res = (Renderer::Vector*)RES;  
    Renderer::Vector v = *(Renderer::Vector*)V;  
    Renderer::Float f = *(Renderer::Float*)F;  
  
    Renderer::mulvf(res, v, f);  
}
```

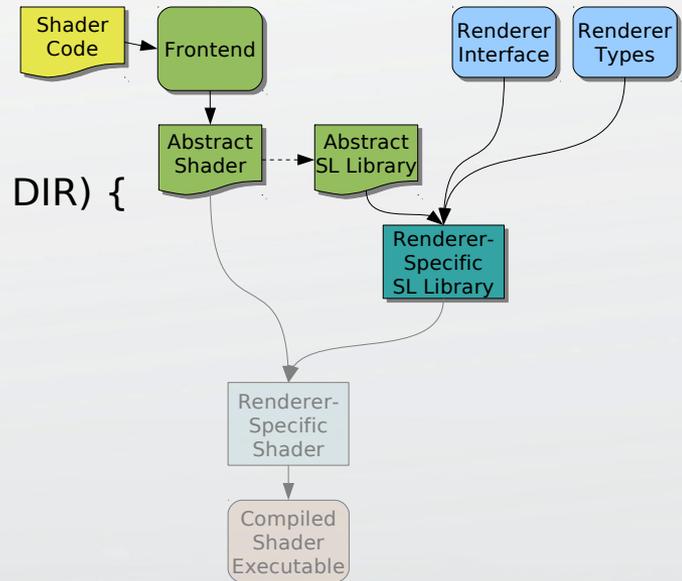


- Low-level operations have to be implemented with renderer's types

# Renderer-Dependent Language Features

```
#include "Renderer/Interface.h"
```

```
void anysI_rsl_trace(void* RES, void* context, void* P, void* DIR) {  
    Renderer::Color* res = (Renderer::Color*)RES;  
    Renderer::Point p = *(Renderer::Point*)P;  
    Renderer::Vector dir = *(Renderer::Vector*)DIR;  
  
    Renderer::traceRay(context, res, p, dir);  
}
```



- Functionality of renderer required for *trace()*, *illuminate()*, etc.

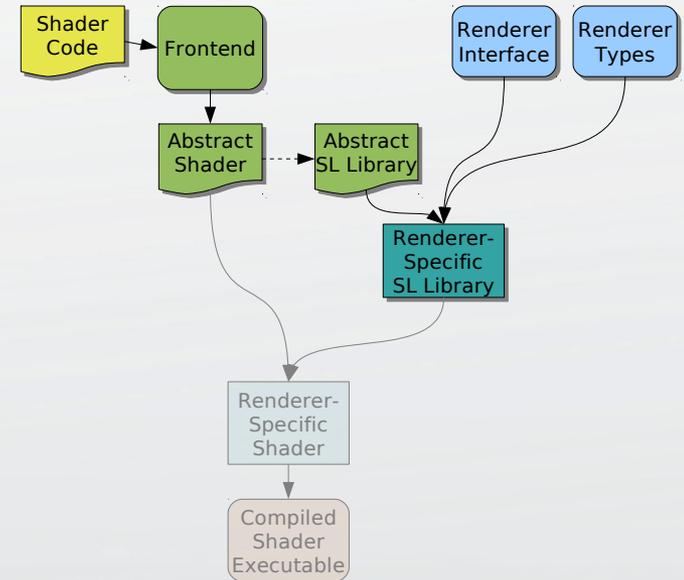
# Inlining 1: Concrete Types → Low-Level

```
typedef float Renderer::Float;  
typedef struct { float x; float y; float z; } Renderer::Vector;
```

```
void anys1_rsl_mulvf(void* RES, void* V, void* F) {  
    Renderer::Vector* res = (Renderer::Vector*)RES;  
    Renderer::Vector v = *(Renderer::Vector*)V;  
    Renderer::Float f = *(Renderer::Float*)F;
```

```
    res->x = v.x * f;  
    res->y = v.y * f;  
    res->z = v.z * f;
```

```
}
```

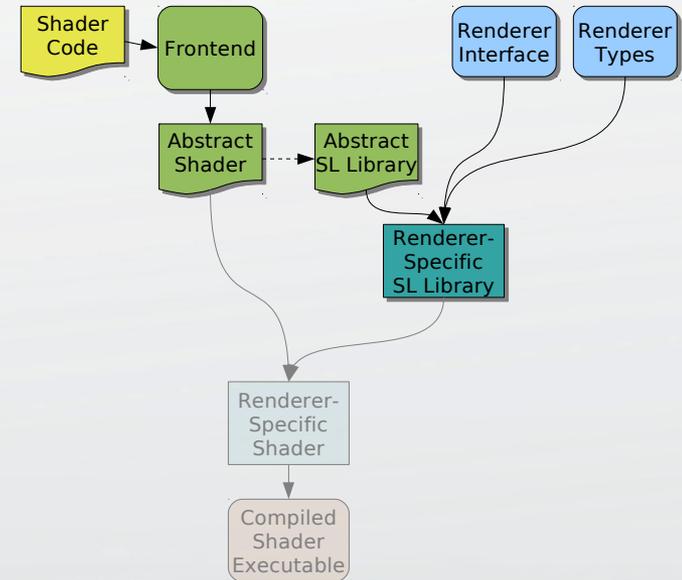


# Inlining 2: Low-Level → Generic Library

```
typedef float Renderer::Float;  
typedef struct { float x; float y; float z; } Renderer::Vector;
```

```
void anys1_rsl_reflect(void* RES, void* I, void* N) {  
    Renderer::Vector* res = (Renderer::Vector*)RES;  
    Renderer::Vector i = *(Renderer::Vector*)I;  
    Renderer::Normal n = *(Renderer::Normal*)N;  
    Renderer::Float c = Renderer::Float(2.f);  
    Renderer::Vector v;  
    Renderer::Float f;
```

```
    f = i.x * n.x + i.y * n.y + i.z * n.z; // dot  
    f *= c; // mulff  
    v.x = n.x * f; // mulvf  
    v.y = n.y * f;  
    v.z = n.z * f;  
    res->x = i.x - v.x; // subvv  
    res->y = i.y - v.y;  
    res->z = i.z - v.z;  
}
```



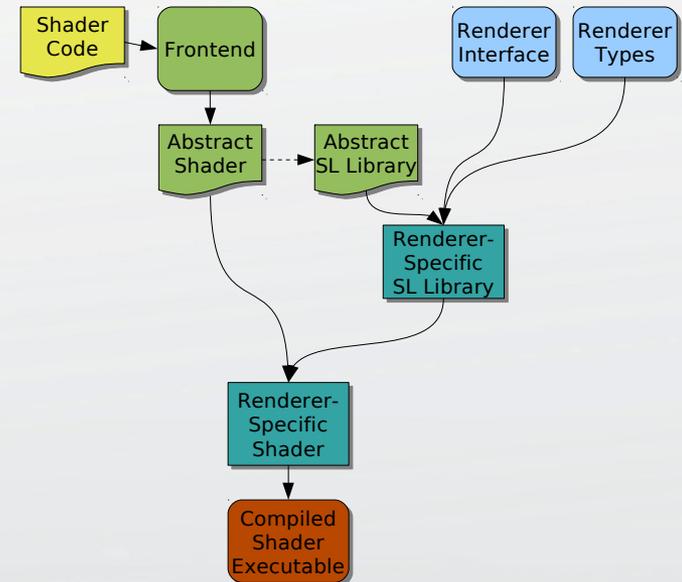
# Inlining 3: Generic Library → Shader

```
typedef float Renderer::Float;  
typedef struct { float x; float y; float z; } Renderer::Vector;
```

```
void shade(void* context) {  
    Renderer::Color* ci = (Renderer::Color*)Ci;  
    Renderer::Vector i = *(Renderer::Vector*)I;  
    Renderer::Normal n = *(Renderer::Normal*)N;  
    Renderer::Point p = *(Renderer::Point*)P;  
    Renderer::Vector dir;  
    Renderer::Vector v;  
    Renderer::Float f;
```

```
    f = i.x * n.x + i.y * n.y + i.z * n.z;  
    f *= 2.f;  
    v.x = n.x * f;  
    v.y = n.y * f;  
    v.z = n.z * f;  
    dir.x = i.x - v.x;  
    dir.y = i.y - v.y;  
    dir.z = i.z - v.z;
```

```
    Renderer::traceRay(context, ci, p, dir);
```



Not distinguishable from  
native code anymore!

# STC: Flexible Type Replacement

```
void anys1_rsl_mulff(void* RES, void* A, void* B) {  
    Renderer::Float* res = (Renderer::Float*)RES;  
    Renderer::Float a = *(Renderer::Float*)A;  
    Renderer::Float b = *(Renderer::Float*)B;  
  
    Renderer::mulff(res, a, b);  
}
```

# STC: Flexible Type Replacement

```
void anys1_rsl_mulff(void* RES, void* A, void* B) {  
    Renderer::Float* res = (Renderer::Float*)RES;  
    Renderer::Float a = *(Renderer::Float*)A;  
    Renderer::Float b = *(Renderer::Float*)B;  
  
    Renderer::mulff(res, a, b);  
}
```

```
typedef float  
    Renderer::Float;
```

# STC: Flexible Type Replacement

```
void anys1_rsl_mulff(void* RES, void* A, void* B) {  
    Renderer::Float* res = (Renderer::Float*)RES;  
    Renderer::Float a = *(Renderer::Float*)A;  
    Renderer::Float b = *(Renderer::Float*)B;  
  
    Renderer::mulff(res, a, b);  
}
```

**Standard Arithmetic**

```
typedef float  
    Renderer::Float;  
  
void anys1_rsl_mulff(void* RES, void* A, void* B) {  
    Renderer::Float* res = (Renderer::Float*)RES;  
    Renderer::Float a = *(Renderer::Float*)A;  
    Renderer::Float b = *(Renderer::Float*)B;  
  
    *res = a * b;  
}
```

# STC: Flexible Type Replacement

```
void anys1_rsl_mulff(void* RES, void* A, void* B) {  
    Renderer::Float* res = (Renderer::Float*)RES;  
    Renderer::Float a = *(Renderer::Float*)A;  
    Renderer::Float b = *(Renderer::Float*)B;  
  
    Renderer::mulff(res, a, b);  
}
```

## Standard Arithmetic

```
typedef float  
    Renderer::Float;  
  
void anys1_rsl_mulff(void* RES, void* A, void* B) {  
    Renderer::Float* res = (Renderer::Float*)RES;  
    Renderer::Float a = *(Renderer::Float*)A;  
    Renderer::Float b = *(Renderer::Float*)B;  
  
    *res = a * b;  
}
```

```
typedef struct { float f; float u; float v; }  
    Renderer::Float;
```

# STC: Flexible Type Replacement

```
void anys1_rsl_mulff(void* RES, void* A, void* B) {  
    Renderer::Float* res = (Renderer::Float*)RES;  
    Renderer::Float a = *(Renderer::Float*)A;  
    Renderer::Float b = *(Renderer::Float*)B;  
  
    Renderer::mulff(res, a, b);  
}
```

**Standard Arithmetic**

```
typedef float  
    Renderer::Float;  
  
void anys1_rsl_mulff(void* RES, void* A, void* B) {  
    Renderer::Float* res = (Renderer::Float*)RES;  
    Renderer::Float a = *(Renderer::Float*)A;  
    Renderer::Float b = *(Renderer::Float*)B;  
  
    *res = a * b;  
}
```

**Automatic Differentiation**

```
typedef struct { float f; float u; float v; }  
    Renderer::Float;  
  
void anys1_rsl_mulff(void* RES, void* A, void* B) {  
    Renderer::Float* res = (Renderer::Float*)RES;  
    Renderer::Float a = *(Renderer::Float*)A;  
    Renderer::Float b = *(Renderer::Float*)B;  
  
    res->f = a.f * b.f;  
    res->u = a.u * b.f + a.f * b.u;  
    res->v = a.v * b.f + a.f * b.v;  
}
```

# AnySL: Benefits

- *Efficiency*
- *Portability*
- *Simplicity*
- *Flexibility*

# AnySL: Benefits

- *Efficiency*
- *Portability* ✓  
STC is renderer- and platform-independent
- *Simplicity* ✓  
Implement frontend + interpreter instead of compiler toolchain
- *Flexibility* ✓  
STC is resolved at runtime → edit, recompile, AD, ...

# AnySL: Benefits

- *Efficiency?*  
Inlining & optimizations remove all overhead
- *Portability* ✓  
STC is renderer- and platform-independent
- *Simplicity* ✓  
Implement frontend + interpreter instead of compiler toolchain
- *Flexibility* ✓  
STC is resolved at runtime → edit, recompile, AD, ...

# Even Better Performance?

- Fastest ray tracers exploit low-level SIMD primitives

# Even Better Performance?

- Fastest ray tracers exploit low-level SIMD primitives
- Shading languages operate on single rays (for a reason!)
- Worst case: 4x slower than native SIMD shader

# Even Better Performance?

- Fastest ray tracers exploit low-level SIMD primitives
- Shading languages operate on single rays (for a reason!)
- Worst case: 4x slower than native SIMD shader
- Solution: AnySL transforms shader to execute  $k$  instances of original code in parallel

# Automatic SIMD Code Generation

- Where is the problem?

## scalar

```
float scalarFn(float a, float b) {  
    for (int i=0; i < b; ++i) {  
        if (a > b) ++b;  
        else ++a;  
    }  
    return a * b;  
}
```

# Automatic SIMD Code Generation

- Where is the problem?
  - Control-flow of scalar instances can diverge (requires blending)

## scalar

```
float scalarFn(float a, float b) {  
    for (int i=0; i < b; ++i) {  
        if (a > b) ++b;  
        else ++a;  
    }  
    return a * b;  
}
```

# Automatic SIMD Code Generation

- Where is the problem?
  - Control-flow of scalar instances can diverge (requires blending)
  - Don't want to write that code by hand...

## scalar

```
float scalarFn(float a, float b) {  
    for (int i=0; i < b; ++i) {  
        if (a > b) ++b;  
        else ++a;  
    }  
    return a * b;  
}
```

## packetized

```
__m128 simdFn(__m128 a, __m128 b) {  
    __m128 _mm_one = _mm_set_ps1(1.f);  
    __m128 i = _mm_set_ps1(0.f);  
    do {  
        __m128 loopMask = _mm_cmpgt_ps(i, b);  
        i = _mm_add_ps(i, _mm_one);  
        __m128 ifcmp = _mm_cmpgt_ps(a, b);  
        __m128 maskT = _mm_and_ps(loopMask, ifcmp);  
        __m128 maskF = _mm_and_ps(loopMask, _mm_neg_ps(ifcmp));  
        __m128 b' = _mm_add_ps(b, _mm_one);  
        __m128 a' = _mm_add_ps(a, _mm_one);  
        b = _mm_blendv_ps(maskT, b', b);  
        a = _mm_blendv_ps(maskF, a', a);  
    } while (!_mm_movemask_ps(loopMask));  
    return _mm_mul_ps(a, b);  
}
```

# RenderMan Glass Shader



# RenderMan Glass Shader: Code

```
surface
glass ( float Ka = 0.2, Kd = 0, Ks = 0.5;
      float Kr = 1, Kt = 1, roughness = 0.05, blur = 0, eta = 1.5;
      color specularcolor = 1, transmitcolor = 1;
      float samples = 1; )
{
    normal Nf;          /* Forward facing normal vector */
    vector IN;          /* normalized incident vector */
    vector Rfdir, Rrdir; /* Smooth reflection/refraction directions */
    vector uoffset, voffset; /* Offsets for blur */
    color ev = 0;       /* Color of the environment reflections */
    color cr = 0;       /* Color of the refractions */
    vector R, Rdir;     /* Direction to cast the ray */
    uniform float i, j;
    float kr, kt;

    /* Construct a normalized incident vector */
    IN = normalize (I);

    /* Construct a forward facing surface normal */
    Nf = faceforward (normalize(N), I);

    /* Compute the reflection & refraction directions and amounts */
    fresnel (IN, Nf, (I.N < 0) ? 1.0/eta : eta, kr, kt, Rfdir, Rrdir);
    kr *= Kr;
    kt *= Kt;

    /* Calculate the reflection color */
    if (kr > 0.001) {
        /* Rdir gets the perfect reflection direction */
        Rdir = normalize (Rfdir);
        if (blur > 0) {
            /* Construct orthogonal components to Rdir */
            uoffset = blur * normalize (vector (zcomp(Rdir) - ycomp(Rdir),
            xcomp(Rdir) - zcomp(Rdir),
            ycomp(Rdir) - xcomp(Rdir)));
            voffset = Rdir ^ uoffset;
            for (i = 0; i < samples; i += 1) {
                for (j = 0; j < samples; j += 1) {
                    /* Add a random offset to the smooth reflection vector */
                    R = Rdir +
                    ((i + float random())/samples - 0.5) * uoffset +
                    ((j + float random())/samples - 0.5) * voffset;
                }
            }
        }
    }
}
```

```
        ((i + float random())/samples - 0.5) * uoffset +
        ((j + float random())/samples - 0.5) * voffset;
        ev += trace (P, normalize(R));
    }
}
ev *= kr / (samples*samples);
} else {
    /* No blur, just do a simple trace */
    ev = kr * trace (P, Rdir);
}
}

/* Calculate the refraction color */
if (kt > 0.001) {
    /* Rdir gets the perfect refraction direction */
    Rdir = normalize (Rrdir);
    if (blur > 0) {
        /* Construct orthogonal components to Rdir */
        uoffset = blur * normalize (vector(zcomp(Rrdir) - ycomp(Rrdir),
        xcomp(Rrdir) - zcomp(Rrdir),
        ycomp(Rrdir) - xcomp(Rrdir)));
        voffset = Rrdir ^ uoffset;
        for (i = 0; i < samples; i += 1) {
            for (j = 0; j < samples; j += 1) {
                /* Add a random offset to the smooth reflection vector */
                R = Rdir +
                ((i + float random())/samples - 0.5) * uoffset +
                ((j + float random())/samples - 0.5) * voffset;
                cr += trace (P, R);
            }
        }
        cr *= kt / (samples*samples);
    } else {
        /* No blur, just do a simple trace */
        cr = kt * trace (P, Rdir);
    }
}

Oi = 1;
Ci = ( Cs * (Ka*ambient() + Kd*diffuse(Nf)) +
specularcolor * (ev + Ks*specular(Nf,-IN,roughness)) +
transmitcolor * cr );
}
```

# Evaluation

- Automatic SIMD Code Generation



**sequential**

# Evaluation

- Automatic SIMD Code Generation



**sequential**



**packetized**

# Evaluation

- Automatic SIMD Code Generation



sequential

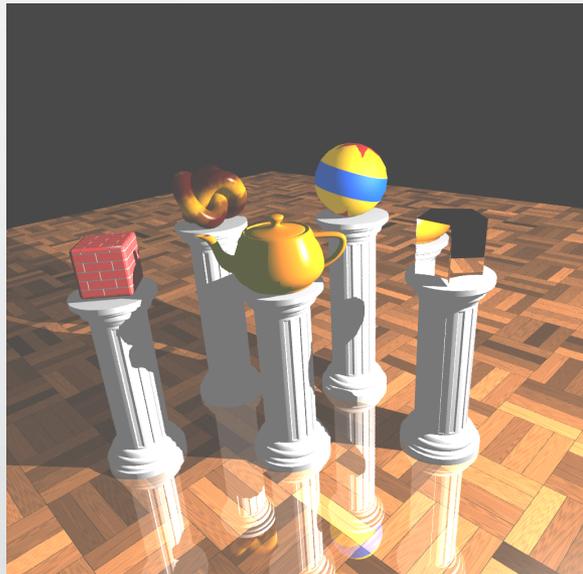


packetized

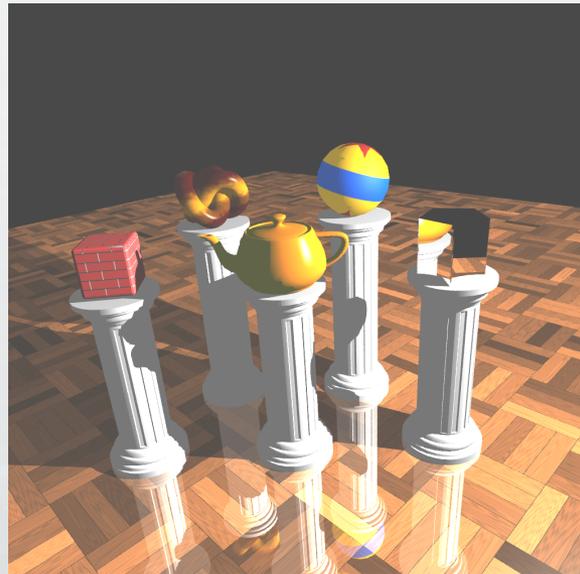
- RTfact: average speedup vs. sequential shading: **3.9x !**
  - SIMD width 4 (SSE4.1)
  - Within 10% of performance of hand-optimized shaders

# Evaluation

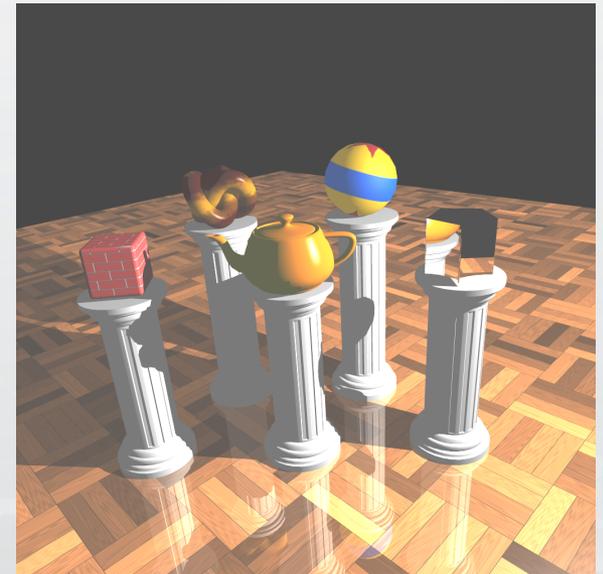
- Equal visual appearance across renderers



**Manta**



**PBRT**



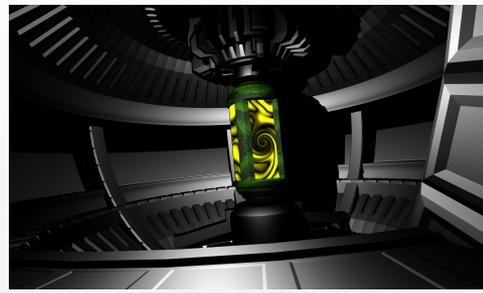
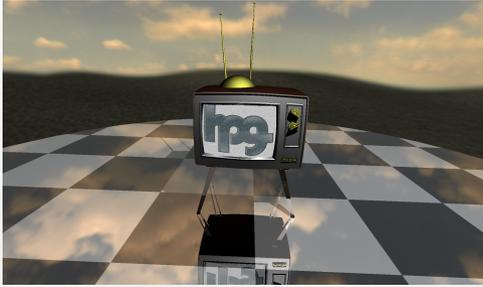
**RTfact**

# Limitations

- Performance vs. hand-tuned native shading
- Currently only surface shaders
- No unified API for shading languages

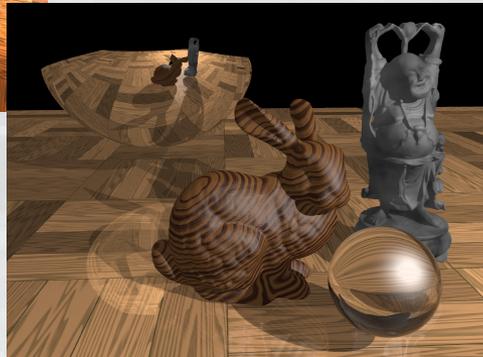
# Future Work

- Rasterization support
  - PTX Back-end for LLVM  
<http://sourceforge.net/projects/llvmptxbackend/>
  - First results with deferred shading in OGRE
  - Native shading via GLSL/HLSL in progress
- GPU Ray Tracer (OptiX?)
- Front-ends: OpenSL, GLSL/HLSL



# Thank you!

## Questions?



# Summary

- Efficient Integration of shading languages into a renderer made easy:
  - Use/Write front-end that produces threaded code
  - Integrate AnySL into renderer via slim API
  - Implement renderer-dependent language features (e.g. illuminance)
  - No more implementation of full-fledged compiler toolchain!
- Subroutine-Threaded Code & Type Replacement
  - Portable shader format
  - Automatic differentiation
- Efficiency & Flexibility:
  - Inlining & Optimization of STC → no performance sacrifice
  - Runtime recompilation & specialization
- Automatic SIMD Code Generation → 3.9x speedup