VISIONARAY
A Modern C++ Framework For Prototyping Ray Tracing Kernels

Stefan Zellmann
University of Cologne
zellmann(at)uni-koeln.de
Motivation
Visualization as a Service

• University of Cologne, Chair of Computer Science with emphasis on Scientific Visualization

• Several Virtual Reality installations (Powerwall, VR-Bench, CAVE, several HMDs)

• Visualization as a service for other institutes:
  – Natural sciences
  – Engineering
  – Geo sciences
Motivation
Scientific Visualization

• Applications:
  – Archaeology
  – Medical imaging
  – Virtual reality
  – Physics
  – (…)

- Scientific Visualization
- Applications:
Motivation
Scientific Visualization

• Applications:
  – Archaeology
  – Medical imaging
  – Virtual reality
  – Physics
  – (…)

Photorealism
Motivation
Scientific Visualization

• Applications:
  – Archaeology
  – Medical imaging
  – Virtual reality
  – Physics
  – (...)

Quick insight, photorealism irrelevant
Motivation
Scientific Visualization

• Applications:
  - Archaeology
  - Medical imaging
  - **Virtual reality**
  - Physics
  - (...)

High realism in real-time
Motivation
Scientific Visualization

• Applications:
  – Archaeology
  – Medical imaging
  – Virtual reality
  – Physics
  – (…)

Higher-order interpolation
Implicit surfaces
Motivation
Scientific Visualization

• A “zoo” of 3D rendering algorithms
  – Direct volume rendering CT/MRI data
  – Path tracing for surfaces
  – Iso-surface rendering
  – Whitted-style ray tracing
  – ...

Motivation
Scientific Visualization

• A “zoo” of 3D rendering algorithms
  – Direct volume rendering CT/MRI data
  – Path tracing for surfaces
  – Iso-surface rendering
  – Whitted-style ray tracing
  – ...

Many are naturally implemented with ray tracing
Motivation
Direct Volume Rendering

Video not included in PDF slides
Motivation
Direct Volume Rendering

Algorithm Setting
Motivation

Direct Volume Rendering

Algorithm Setting

Viewing Position
Motivation
Direct Volume Rendering

Algorithm Setting

Image Plane
Motivation
Direct Volume Rendering

Algorithm Setting

Density Data
Motivation
Direct Volume Rendering

- Ray Marching
- Pixel Color
- Piecewise Integration
Motivation

Path Tracing

Video not included in PDF slides
Motivation
Path Tracing

Again shoot rays...

...to determine pixel color
Motivation

Path Tracing

But this time don’t integrate over density...

...but over reflection functions with Monte Carlo.
Motivation
Path Tracing

Or have some user-defined primitives...
Visionaray

Scope

- Visionaray is a “Ray Tracing Template Library” providing algorithms and data structures that are related to ray tracing
- Focus on real-time and VR, but not necessarily so
- In contrast to competitors (e.g. Intel Embree), we provide facilities for all kinds of ray tracing related tasks:
  - Custom geometric primitives (not just triangles)
  - SIMD packet traversal
  - Texture filtering (CPU and GPU, higher-order)
  - BVH traversal (not just triangles, but any user primitive)
Visionaray Basics
Schedulers and Kernels
Visionaray Basics
Schedulers and Kernels

Schedulers
• Primary ray generation in parallel
• Store final pixel color
• Independent of the algorithm!
• Run any algorithm on any hardware!

Kernels
• “Algorithmic phase”, most often described in terms of single rays
• Similar basic functions:
  – Texture access
  – Primitive traversal
  – …
Various Hardware Platforms

Schedulers and Kernels

Various Hardware Platforms

Scheduler HW1

Scheduler HW2

Scheduler HW3

Execution

scheduler sched;
sched.frame(
            kernel
);
rt = sched.render_target;
rt.display_color_buffer();

Path tracing kernel

Bidirectional path tracing kernel

Volume rendering kernel

More user-defined kernels
Visionaray Basics
Schedulers and Kernels

Various Hardware Platforms

- Scheduler HW1
- Scheduler HW2
- Scheduler HW3

x86 CPUs, clusters, GPGPUs, FPGAs

Execution

```c
scheduler sched;
sched.frame(3, kernel);
rt = sched.render_target;
rt.display_color_buffer();
```

Path tracing kernel
Bidirectional path tracing kernel
Volume rendering kernel
More user-defined kernels
Various Hardware Platforms

Schedulers and Kernels

Execution

```
scheduler sched;  
sched.frame(  
   kernel  
);  
rt = sched.render_target;  
rt.display_color_buffer();
```

---

CUDA, AVX, MPI, Intel TBB

---

Path tracing kernel

Bidirectional path tracing kernel

Volume rendering kernel

More user-defined kernels
Various Hardware Platforms

Schedulers and Kernels

Execution

```
scheduler sched;
sched.frame();
rt = sched.render_target;
rt.display_color_buffer();
```

Path tracing kernel

Bidirectional path tracing kernel

Volume rendering kernel

More user-defined kernels

Now all algorithms can run on any type of hardware
#include <visionaray/camera.h>
#include <visionaray/scheduler.h>

using namespace visionaray;

typedef basic_ray<float> ray_type;
typedef simple_sched<ray_type> sched_type;
typedef vector<4, float> color_type;

sched_type sched;
auto sparams = make_sched_params(
  camera,
  render_target
);

sched.frame( [=](ray_type r) -> color_type {
  return color_type(1.0, 1.0, 1.0, 1.0);
},
sparams);
#include <visionaray/camera.h>
#include <visionaray/scheduler.h>

using namespace visionaray;

typedef basic_ray<float> ray_type;
typedef simple_sched<ray_type> sched_type;
typedef vector<4, float> color_type;

sched_type sched;
auto sparams = make_sched_params(
    camera,
    render_target
);

sched.frame( [=](ray_type r) -> color_type {
    return color_type(1.0, 1.0, 1.0, 1.0);
},
sparams);

All basic types are templates (e.g. for SIMD)
#include <visionaray/camera.h>
#include <visionaray/scheduler.h>

using namespace visionaray;

typedef basic_ray<float> ray_type;
typedef simple_sched<ray_type> sched_type;
typedef vector<4, float> color_type;

sched_type sched;
auto sparams = make_sched_params(
    camera,
    render_target);

sched.frame( [=] (ray_type r) -> color_type { return color_type(1.0, 1.0, 1.0, 1.0); },
    sparams);

“simple_sched” is the most basic scheduler
Visionary Example
Ray Tracing Hello World 😊

```cpp
#include <visionaray/camera.h>
#include <visionaray/scheduler.h>

using namespace visionaray;

typedef basic_ray<float> ray_type;
typedef simple_sched<ray_type> sched_type;
typedef vector<4, float> color_type;

sched_type sched;
auto sparams = make_sched_params(
    camera,
    render_target
);

sched.frame(=[ray_type r] -> color_type
{
    return color_type(1.0, 1.0, 1.0, 1.0);
},
sparams);
```

Create a scheduler instance
#include <visionaray/camera.h>
#include <visionaray/scheduler.h>

using namespace visionaray;

typedef basic_ray<float> ray_type;
typedef simple_sched<ray_type> sched_type;
typedef vector<4, float> color_type;

sched_type sched;
auto sparams = make_sched_params(
    camera,
    render_target
);

sched.frame([=](ray_type r) -> color_type {
    return color_type(1.0, 1.0, 1.0, 1.0);
},
    sparams);
#include <visionaray/camera.h>
#include <visionaray/scheduler.h>

using namespace visionaray;

typedef basic_ray<float> ray_type;
typedef simple_sched(ray_type) sched_type;
typedef vector<4, float> color_type;

sched_type sched;
auto sparams = make_sched_params(
    camera,
    render_target
);

sched.frame(=[ray_type r] -> color_type {
    return color_type(1.0, 1.0, 1.0, 1.0),
    sparams);

Create a scheduler instance

Rendertarget: framebuffer
Visionaray Example
Ray Tracing Hello World 😊

```cpp
#include <visionaray/camera.h>
#include <visionaray/scheduler.h>

using namespace visionaray;

typedef basic_ray<float> ray_type;
typedef simple_sched<ray_type> sched_type;
typedef vector<4, float> color_type;

sched_type sched;
auto sparams = make_sched_params(
    camera,
    render_target
);

sched.frame( [=](ray_type r) -> color_type
{
    return color_type(1.0, 1.0, 1.0, 1.0);
},
sparams);
```

Most simple Visionaray kernel
Visionary Example
Ray Tracing Hello World 😊

#include <visionaray/camera.h>
#include <visionaray/scheduler.h>

using namespace visionaray;

typedef basic_ray<float> ray_type;
typedef simple_sched<ray_type> sched_type;
typedef vector<4, float> color_type;

sched_type sched;
auto sparams = make_sched_params(
    camera,
    render_target
);

sched.frame( [=](ray_type r) -> color_type
{
    return color_type(1.0, 1.0, 1.0, 1.0);
},
sparams);
#include <visionaray/camera.h>
#include <visionaray/scheduler.h>

using namespace visionaray;

typedef basic_ray<float> ray_type;
typedef simple_sched<ray_type> sched_type;
typedef vector<4, float> color_type;

sched_type sched;
auto sparams = make_sched_params(
    camera,
    render_target
);

sched.frame(=[] (ray_type r) -> color_type
{
    return color_type(1.0, 1.0, 1.0, 1.0);
},
sparams);
Visionaray Example
Ray Tracing Hello World 🎉

```cpp
#include <visionaray/camera.h>
#include <visionaray/scheduler.h>

using namespace visionaray;

typedef basic_ray<float> ray_type;
typedef simple_sched<ray_type> sched_type;
typedef vector<4, float> color_type;

sched_type sched;
auto sparams = make_sched_params(
    camera,
    render_target
);

sched.frame( [=](ray_type r) -> color_type
{
    return color_type(1.0, 1.0, 1.0, 1.0);
},
    sparams);
```

Returns a color
Visionaray built-in types
Schedulers and Kernels

• Built-in schedulers at your convenience:
  – simple_sched (pure scanlines, no multi-threading)
  – tiled_sched (task queue-based with atomics)
  – cuda_sched
  – sycl_sched (w.i.p.)
  – mpi_sched (w.i.p.)

• Built-in kernels
  – “Simple”: primary visibility only
  – “Whitted”: perfect reflections and hard shadows
  – “Pathtracing”: basic path tracer
Visionaray built-in types
Schedulers and Kernels

• We expect 3 types of framework users:
  – Reuse existing algorithms but implement e.g. custom primitives (e.g. Bezier Patches, Nurbs, ...) or custom texturing algorithms
  – Reuse existing schedulers, write completely new kernels (immediately available on a variety of platforms)
  – Port existing algorithm (i.e. kernel) to new platform: write a new scheduler (hard task, might involve adaptation of intrinsic functions (which are all templates, anyway 😊))
Visionaray built-in types
Geometric Primitives

- Built-in kernels deal with surfaces
  - (Your own kernels need not to!)
- Everything is a template, so:
  - Triangles and spheres are implemented
  - But just implement your own ones. Built-in kernels (basically) cope with any primitive that implements $\text{intersect}(\text{ray}, \text{primitive})$
  - No virtual inheritance w/ templates $\Rightarrow$ for multiple primitives use variants instead ("tagged unions")
    - This is highly efficient because for pure triangle ray tracing no switch-case/vtable lookup necessary in the inner loop
### Visionary built-in types

**Single Primitives Example**

```cpp
typedef basic_triangle<3, float> triangle_t;

std::vector<triangle_t> triangles;
for (int i=0; i<count; ++i) triangles.emplace_back(v1, e1, e2);

auto kparams = make_kernel_params(
    triangles.data(), // begin iterator
    triangles.data() + triangles.size(), // end iterator
    /* more parameters here, like normals or textures */
);

whitted::kernel<decltype(kparams)> kernel;
kernel.params = kparams;

sched.frame(kernel, sparams);
```
typedef basic_triangle<3, float> triangle_t;
typedef basic_sphere<float> sphere_t;
typedef basic_myown<float> myown_t;

typedef generic_primitive<
    triangle_t,
sphere_t,
myown_t
>          primitive_t;
Visionary built-in types
Generic Primitives Example

typedef basic_triangle<3, float> triangle_t;
typedef basic_sphere<float> sphere_t;
typedef basic_myown<float> myown_t;

typedef generic_primitive<
    triangle_t,
    sphere_t,
    myown_t
> primitive_t;

Now use primitive_t just like in the example before!
typedef ... primitive_t;

std::vector<primitive_t> primitives;
for (int i=0; i<count; ++i) primitives.emplace_back(...);

auto kparams = make_kernel_params(
    primitives.data(), // begin iterator
    primitives.data() + primitives.size(), // end iterator
    /* more parameters here, like normals or textures */
);

whitted::kernel<decltype(kparams)> kernel;
kernel.params = kparams;

sched.frame(kernel, sparams);
Example Programs on Github
Demonstrate How to Use All This!

https://github.com/szellmann/visionaray
Virtual Reality
Plugin in for OpenCOVER

- Virtual environments
- Head tracking
- Scene graph rendering
- Walk-in immersive environments ("CAVE")
Questions?

https://github.com/szellmann/visionaray
https://github.com/hlrs-vis/covise

Stefan Zellmann
University of Cologne
zellmann(at)uni-koeln.de