Visionaray: A Cross-Platform Ray Tracing Template Library

Stefan Zellmann
University of Cologne
zellmann@uni-koeln.de

Daniel Wickeroth
University of Cologne
wickeroth@uni-koeln.de

Ulrich Lang
University of Cologne
lang@uni-koeln.de

SEARIS 2017 10th Workshop, colocated with IEEE VR, 3/19/17, Los Angeles, CA
Motivation

• Why another ray tracing library?
  – Holistic approach, vendor libraries
    • implement only kernel functions (e.g. BVH traversal)
      (Intel Embree, AMD RadeonRays)
    • have fixed programmable pipeline (Nvidia OptiX)
  – Cross platform, vendor libraries
    • are simply not.. cross platform, they are vendor libraries (RadeonRays is an exception..)
Our Approach

• Loose set of ray tracing-related C++ templates, similar to Boost or STL
• Rely on modern C++1{1|y|z} features
• Dedicated abstractions for some omnipresent tasks (e.g. parallel ray traversal)
• Not just “kernels”, but templates for every task related to ray tracing
• Not limited to image generation
Cross-Platform

CPU

- GCC
- Intel
- MSVC

2007

GPU, FPGA

- Nvidia CUDA
- AMD ROCm
- Xilinx Vivado HLS
- CLANG

2017
void func() {
    // Let's start with a single ray
    ray r(vec3(0,0,0), normalize(vec3(0,0,1)));
}

An Example
void func() {
    // Let's start with a single ray
    ray r(vec3(0,0,0), normalize(vec3(0,0,1)));

    // And some triangles..
    aligned_vector<triangle> ts;
    ts.emplace_back(vec3(0,0,0), vec3(1,0,0), vec3(1,1,0));
    ts.emplace_back(vec3(0,0,0), vec3(2,0,0), vec3(3,4,0));
}

An Example

```cpp
void func() {
    // Let's start with a single ray
    ray r(vec3(0,0,0), normalize(vec3(0,0,1)));

    // And some triangles..
    aligned_vector<triangle> ts;
    ts.emplace_back(vec3(0,0,0), vec3(1,0,0), vec3(1,1,0));
    ts.emplace_back(vec3(0,0,0), vec3(2,0,0), vec3(3,4,0));

    // closest_hit(), a generic algorithm
    auto hit_rec = closest_hit(r, ts.begin(), ts.end());
}
```
void func() {
  // Let's start with a single ray
  ray r(vec3(0,0,0), normalize(vec3(0,0,1)));

  // And some triangles..
  aligned_vector<triangle> ts;
  ts.emplace_back(vec3(0,0,0), vec3(1,0,0), vec3(1,1,0));
  ts.emplace_back(vec3(0,0,0), vec3(2,0,0), vec3(3,4,0));

  // closest_hit(), a generic algorithm
  auto hit_rec = closest_hit(r, ts.begin(), ts.end());

  // Process the result
  if (hit_rec.hit) { /* ... */ }
}
void func() {
    // Let's start with a single ray
    ray r(vec3(0,0,0), normalize(vec3(0,0,1)));

    // Why not trace some spheres?
    aligned_vector<sphere> ts;
    ts.emplace_back(vec3(0,0,0), 3.14f);
    ts.emplace_back(vec3(0,0,0), 23.0f);

    // closest_hit(), a generic algorithm
    auto hit_rec = closest_hit(r, ts.begin(), ts.end());

    // Process the result
    if (hit_rec.hit) { /* ... */ }
}
void func() {
    // Let's start with a single ray
    ray r(vec3(0,0,0), normalize(vec3(0,0,1)));

    // Why not trace some spheres?
    aligned_vector<sphere> ts;
    ts.emplace_back(vec3(0,0,0),3.14f);
    ts.emplace_back(vec3(0,0,0),23.0f);

    // closest_hit(), a generic algorithm
    auto hit_rec = closest_hit(r, ts.begin(), ts.end());

    // Process the result
    if (hit_rec.hit) { /* ... */ }
}
void func() {
    // Let's start with a single ray
    ray r(vec3(0,0,0), normalize(vec3(0,0,1)));
    // Why not trace some spheres?
    aligned_vector<sphere> ts;
    ts.emplace_back(vec3(0,0,0),3.14f);
    ts.emplace_back(vec3(0,0,0),23.0f);
    // closest_hit(), a generic algorithm
    auto hit_rec = closest_hit(r, ts.begin(), ts.end());
    // Process the result
    if (hit_rec.hit) { /* ... */ }
}
void func() {
    // Let's start with a single ray
    ray r(vec3(0,0,0), normalize(vec3(0,0,1)));

    // Why not trace some spheres?
    aligned_vector<
        sphere
    > ts;
    ts.emplace_back(vec3(0,0,0), 3.14f);
    ts.emplace_back(vec3(0,0,0), 23.0f);

    // closest_hit(), a generic algorithm
    auto hit_rec = closest_hit(r, ts.begin(), ts.end());

    // Process the result
    if (hit_rec.hit) { /* ... */ }
}
void func() {
    // Let's start with a single ray
    ray r(vec3(0,0,0), normalize(vec3(0,0,1)));

    // Why not trace some spheres?
    aligned_vector<sphere> ts;
    ts.emplace_back(vec3(0,0,0), 3.14f);
    ts.emplace_back(vec3(0,0,0), 23.0f);

    // closest_hit(), a generic algorithm
    auto hit_rec = closest_hit(r, ts.begin(), ts.end());

    // Process the result
    if (hit_rec.hit) { /* ... */ }
}
```cpp
void func() {
    // Let's start with a single ray
    ray r(vec3(0,0,0), normalize(vec3(0,0,1)));

    // Why not trace some spheres?
    aligned_vector<sphere> ts;
    ts.emplace_back(vec3(0,0,0), 3.14f);
    ts.emplace_back(vec3(0,0,0), 23.0f);

    // closest_hit(), a generic algorithm
    auto hit_rec = closest_hit(r, ts.begin(), ts.end());

    // Process the result
    if (hit_rec.hit) { /* ... */ }
}
```

**Ray**
- ori: Vec3
dir: Vec3

**Vec3**
- x: FloatT
- y: FloatT
- z: FloatT
Algorithms

```c
void func() {
    // Let's start with a single ray
    ray r(vec3(0,0,0), normalize(vec3(0,0,1)));

    // Why not trace some spheres?
    aligned_vector<sphere> ts;
    ts.emplace_back(vec3(0,0,0),3.14f);
    ts.emplace_back(vec3(0,0,0),23.0f);

    // closest_hit(), a generic algorithm
    auto hit_rec = closest_hit(r, ts.begin(), ts.end());

    // Process the result
    if (hit_rec.hit) { /* ... */ }
}
```
Design Principles

• Loose coupling, algorithm specifies what Concept it is compatible with, works with any type that implements that Concept

• Customization points: User provides drop-in replacement to customize behavior
Design Principles

• It’s all templates ➔ argument dependent lookup instead of runtime polymorphism
  – Must keep (custom) types POD!
    • Works well with templates, ADL, and STL containers
    • POD means you can memcpy() ... and cudaMemcpy()!
    • POD maps well to system memory!
List of most important concepts

- **Primitive**
  - Requirement: \textit{intersect}(Ray, Primitive)

- **Ray**
  - Requirement: Members [ori: Vec3;dir:Vec3]

- **Vec\{2|3|4|N\}**
  - Requirement: Members [x: FloatT;y: FloatT;z:FloatT;w:FloatT]

- **FloatT**
  - Semantics like builtin float, but can e.g. wrap SIMD types (SSE: \_\_m128, NEON: float32x4_t, ..)
List of most important concepts

• Material
  – POD type with functions $shade()$ and $sample()$

• BRDF
  – POD type with functions $f()$ and $sample_f()$

• Light
  – POD type with functions $intensity()$, $sample()$, ...

• RenderTarget
  – ...

List of most important algorithms

- Ray / object intersection
  - intersect(Ray, Primitive)
- Ray / object traversal
  - closest_hit(), any_hit(), multi_hit()
- BVH construction
  - build<BvhT>()
- Texture filtering
  - tex{1|2|3}D()
Schedulers and Kernels

- Dedicated infrastructure for parallel ray tracing

- Schedulers
  - encapsulate hardware (CUDA-scheduler, TBB-scheduler, HCC-scheduler...)
  - generate primary rays in parallel

- Kernels
  - are function objects (lambda, free func, etc.) that process primary rays and perform traversal
  - together with schedulers allow for reusing complex ray traversal algorithms across platforms
Schedulers and Kernels

// frame() generates camera rays in parallel
sched.frame()

    // now follows a simple kernel (the lambda)
    [&](basic_ray<float> ray) {
        // spheres is a container (e.g. std::vector<>)
        // captured from parent scope!
        return closest_hit(ray,
            spheres.begin(),
            spheres.end());
    },

    // sched. params contain camera and render target
    sparams);
Many more abstraction layers

• C++ variants (tagged unions) to pack multiple POD types (primitives, materials, light sources) into a single type
• “Compile time polymorphism”
• Light weight referencing with interface injection to pass data handles between host and device
• Abstraction layer for texture access, similar to CUDA texture objects
Generic Design Patterns

• SFINAE (Substitution Failure Is Not An Error)
  – esp. since C++ has no real concepts yet..
• Type traits
  – Go hand in hand with SFINAE
• Curiously recurring template pattern
• Basically see Andrei Alexandrescu’s book “Modern C++ design” on generic patterns...
• No “overall pattern” like e.g. MVC, because Visionaray is a loose collection of templates and data structures
Case Study: AO Benchmark

- Scenes from Aila & Laine’s 2012 GPGPU ray traversal paper
- Mrays/s, $1024^2$ primary rays, 8 AO rays
Case Study: AO Benchmark

- Tests on Raspberry PI 3 ☺, Xeon 2x8-core, NVIDIA Quadro K6000, and Intel Knights Landing (64 core model, 4x SMT enabled)
## Case Study: AO Benchmark

<table>
<thead>
<tr>
<th></th>
<th>331K tris 253K BVH nodes</th>
<th>174K tris 132K BVH nodes</th>
<th>75K tris 74K BVH nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>RPi3</td>
<td>2.0 (2.3)</td>
<td>1.2 (1.3)</td>
<td>1.7 (1.9)</td>
</tr>
<tr>
<td>Xeon</td>
<td>102.0 (117.0)</td>
<td>55.0 (57.1)</td>
<td>94.2 (105.6)</td>
</tr>
<tr>
<td>K6000</td>
<td>142.4 (189.1)</td>
<td>63.1 (71.1)</td>
<td>113.2 (142.9)</td>
</tr>
<tr>
<td>KNL</td>
<td>254.0 (273.2)</td>
<td>130.0 (130.4)</td>
<td>247.7 (268.2)</td>
</tr>
</tbody>
</table>

Binned BVH (Binned BVH + spatial splits), units: Mrays/s
Case Study: Volume Rendering

- Direct volume rendering in VR
- CAVE, Display Wall, ART head tracking
- Applications in medical visualization and stereotactic operation planning
- Interactive clipping infrastructure
- GPGPU implementation
Case Study: Volume Rendering
Case Study: Scientific Visualization

- Integration with SciVis software COVISE
- Plugin that bypasses scenegraph rendering with custom image generation code
- High quality rendering through ray tracing / path tracing
- Ongoing work!
Case Study: Scientific Visualization
Case Study: Large Point Clouds

- LiDAR data: points + RGB
- Represent points by spheres
- \( O(\log n) \) complexity allows for large data sets w/o using LOD
- Video (next): point cloud with over 152 million points
Case Study: Large Point Clouds
Discussion

• Different approaches for cross platform ray tracing library:
  – Lowest common denominator approach: use ANSI-C, OpenCL 1.2, etc. (e.g. RadeonRays)
  – Dedicated DSL for ray tracing (e.g. AnyDSL)
  – SPMD (e.g. Intel ispc, shares commonalities with DSLs)
  – Generic approach (e.g. Visionaray)
Discussion

• We believe it’s undecided which approach is best
  – Need e.g. compiler construction knowledge to design a DSL
  – Lack of expressive language features in ANSI-C
  – Code complexity with C++ templates
Main reasons for us to use C++:

- Compile time meta programming suitable for good runtime performance
- C++ has evolved significantly over the last 7-8 years
- C++ compilers have evolved, too
- There’s Clang
- High availability recently: CUDA, AMD ROCm, even FPGAs (Xilinx Vivado HLS)
Questions?

https://vis.uni-koeln.de/visionaray.html
https://github.com/szellmann/visionaray