

RTfact: Concepts for Generic and High Performance Ray Tracing

Iliyan Georgiev

University of Saarland

Philipp Slusallek

University of Saarland

DFKI Saarbrücken

- ◎ Ray tracers are used in RT08 papers
 - > Change packet size?
 - > Change data structures?
- ◎ No common software base
 - > No tools for writing composable software
 - > Cannot reuse and combine parts
 - > Comparison/benchmarking difficult
- ◎ RT08 – arguments for combining more different structures and algorithms

Trends

- ◎ Hardware trends
 - > Massive compute power
 - > Massive memory bandwidth
 - > Massive parallelism
 - Multi-core chips
 - Data-parallel computation
- ◎ Different architectures
- ◎ *Composable portability*
 - > Not 1:1, but reuse as much as possible

Trends

- More capable hardware
- Trend towards software-based graphics
 - > GPUs
 - > Larrabee
- Need fast and flexible software architectures

Existing Ray Tracing Systems

- ⦿ Performance gap between flexible and interactive systems
- ⦿ Extreme examples
 - > PBRT [Pharr et al. 2004]
 - Fine-grained abstractions
 - Highly extensible
 - Slow
 - > Arauna [Bikker 2007]
 - High performance
 - Fixed functionality
 - Hand-optimized
- ⦿ Ideally – both performance and flexibility

Possible Approaches

- ⦿ Manual SIMD coding
- ⦿ Automatic compiler techniques
- ⦿ Object-oriented programming
- ⦿ Domain-specific languages

Possible Approaches

On the Lower Level

- ⦿ Manual coding (e.g. SSE intrinsics)
 - > Best possible performance
 - > Full control of what gets executed
 - > Sensitive to small changes
 - > Unintuitive programming model
 - > Modern day assembly language
 - > Not portable

Possible Approaches

On the Lower Level

- ◎ Rely on compilers
 - > The ideal solution
 - > Do not have enough context
 - Application-specific optimizations?
 - Automatic parallelization?
 - > No signs for getting smart enough soon

Possible Approaches

Object-oriented programming

- ◎ Run-time flexibility
 - > Late binding - virtual functions
 - ◎ Good when types are unknown at compile time
 - ◎ But
 - > Coupling of algorithms and data structures
 - Different existing algorithms for the same structure (and vice versa)
 - > Large run-time overhead at a fine granularity
 - Special-case code
 - Disables optimizations
- ⇒ Used only at coarse level (e.g. Manta)

Possible Approaches

On Multiple Levels

- ◎ Domain-specific language (DSL)
 - > Simple application-suitable syntax
 - > Specialized compiler with enough context
 - Limited scope make optimizations easier
 - Data parallelism
- ◎ But
 - > Need to juggle two different abstractions
 - > Lots of infrastructure work
 - > Tends to converge to a general purpose programming language

Requirements

- ⦿ Composability
 - > At design/compile-time
- ⦿ Compatibility
 - > Use existing tools
 - > Integration with other software
- ⦿ Multi-level abstractions
 - > From basic arithmetic types to renderers
- ⦿ Portability
- ⦿ Performance
 - > Allow efficient code generation

Generic Programming

C++ Templates

- Compile-time...

- > Composability
- > Specialization
- > Optimizations
 - Inlining

- ⇒ *Enables*

- > Arbitrary-grained abstractions
- > Performance
- > Decoupling of algorithms and data structures

Generic Programming

◉ Generic algorithm

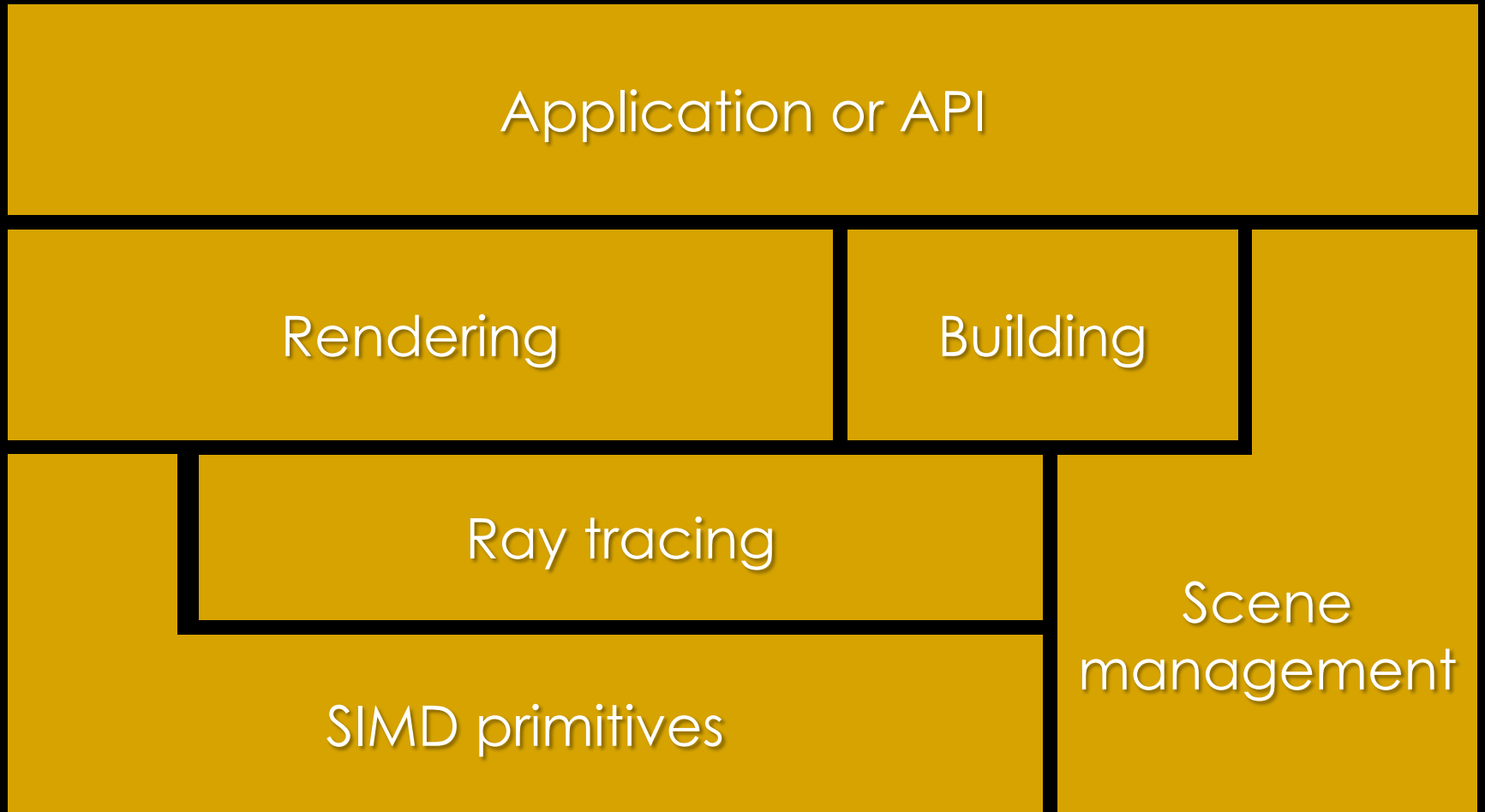
```
template<class Comparable>
void order(Comparable& element1, Comparable& element2) {
    if(element1 > element2) {
        Type tempElement = element2;
        element2 = element1;
        element1 = tempElement;
    }
}
```

◉ .Concepts

```
class Comparable
{
public:
    bool operator>(const Comparable& other);
}
```

RTfact

General Infrastructure



Generic SIMD Primitives

- Parallelism
 - > Threading – on the algorithmic/application level
 - > SIMD - impacts the whole infrastructure
- 4 basic SIMD primitives
 - > Parameterized by size (at compile time)
 - > Packet, Vec3f, PacketMask, BitMask
- Simultaneously model 3 basic concepts
 - > Value, ValueContainer, ContainerContainer
 - > Packets have 3-fold nature
- Portable
- Different sizes means different types
- Data is allocated on the stack

```
template<unsigned int size>
Vec3f<size> computeColor(Vec3f<size>& dir, Vec3f<size>& offset, Vec3f<size>& normal)
{
    Packet<size,float> dotN;
    PacketMask<size> mask;
    BitMask<size> bitMask;
    ...
    return color;
}
```

```
template<unsigned int size>
Vec3f<size> computeColor(Vec3f<size>& dir, Vec3f<size>& offset, Vec3f<size>& normal)
{
    Packet<size,float> dotN;
    PacketMask<size> mask;
    BitMask<size> bitMask;
    ...
    return color;
}

dotN = Dot(dir + offset, normal); //
mask = (dotNormal > Packet<size,float>::C_0()); // SCALAR-LIKE
color = mask.blend(Vec3f<size>::RED(), Vec3f<size>::BLACK()); // (Value concept)
bitMask = mask.getBitMask(); //
```

```

template<unsigned int size>
Vec3f<size> computeColor(Vec3f<size>& dir, Vec3f<size>& offset, Vec3f<size>& normal)
{
    Packet<size,float> dotN;
    PacketMask<size> mask;
    BitMask<size> bitMask;
    ...
    return color;
}

dotN = Dot(dir + offset, normal); //
mask = (dotNormal > Packet<size,float>::C_0()); // SCALAR-LIKE
color = mask.blend(Vec3f<size>::RED(), Vec3f<size>::BLACK()); // (Value concept)
bitMask = mask.getBitMask(); //

for(int i = 0; i < size; ++i) //
{ //
    dotN [i] = Dot(dir[i] + offset[i], normal[i]); // COMPONENT-WISE
    mask[i] = (dotN[i] > 0); // (ValueContainer
    color.set(i, mask[i] ? Vec3f<1>::RED(), Vec3f<1>::BLACK()); // concept)
    bitMask.set(i, mask[i]); //
} //

```

```

template<unsigned int size>
Vec3f<size> computeColor(Vec3f<size>& dir, Vec3f<size>& offset, Vec3f<size>& normal)
{
    Packet<size,float> dotN;
    PacketMask<size> mask;
    BitMask<size> bitMask;
    ...
    return color;
}

dotN = Dot(dir + offset, normal); //
mask = (dotNormal > Packet<size,float>::C_0()); // SCALAR-LIKE
color = mask.blend(Vec3f<size>::RED(), Vec3f<size>::BLACK()); // (Value concept)
bitMask = mask.getBitMask(); //

for(int i = 0; i < size; ++i) //
{ //
    dotN [i] = Dot(dir[i] + offset[i], normal[i]); // COMPONENT-WISE
    mask[i] = (dotN[i] > 0); // (ValueContainer
    color.set(i, mask[i] ? Vec3f<1>::RED(), Vec3f<1>::BLACK()); // concept)
    bitMask.set(i, mask[i]); //
} //

for(int i = 0; i < Packet<size, float>::CONTAINER_COUNT; ++i) //
{ //
    dotN (i) = Dot(dir(i) + offset(i), normal(i)); //
    mask(i) = (dotN(i) > Packet<size,float>::Container::C_0()); // CONTAINER-WISE
    color(i) = mask(i).blend(Vec3f<size>::Container::RED(), // (ContainerContainer
                            Vec3f<size>::Container::BLACK()); // concept)
    bitMask.setContainer(i, mask(i).getBitMask()); //
}

```

Rays and Ray Packets

- Ray packets – aggregation of other packets

```
template<unsigned int size>
struct RayPacket
{
    Vec3f<size> org;
    Vec3f<size> dir;
    Packet<size, float> tMin;
    Packet<size, float> tMax;
}
```

- Single rays simply have size 1

```
RayPacket<1> ray;
```

- Different packets are different types
 - > Can exist at the same time

Algorithms and Data Structures

- Data structures decoupled from algorithms
 - Algorithms are described in terms of **concepts**

```
template<class Element>
class KdTree : public Primitive {
public:
    class NodeIterator;
    class ElementIterator;
    // interface for structure building
    void createInnerNode(NodeIterator node, int axis, float splitVal);
    template<class Iterator>
    void createLeaf(NodeIterator leaf, const BBox& bounds,
                  Iterator begin, Iterator end);
    // interface for structure traversal
    NodeIterator getRoot() const;
    NodeIterator getLeftChild(NodeIterator node) const;
    NodeIterator getRightChild(NodeIterator node) const;
    int getSplitAxis(NodeIterator node) const;
    float getSplitValue(NodeIterator node) const;
    std::pair<ElementIterator, ElementIterator>
        getElements(NodeIterator leaf) const;
};
```

```

template<class ElemIsect>                // nested element intersector
template<int intersDataMask,           // intersection data needed
        bool commonOrg,                // common ray origin?
        unsigned int size,             // size of the ray packet
        class KdTree>                  // models the KdTree concept
void KdTreeIntersector<ElemIsect>::intersect(
    RayPacket<size>& rayPacket,         // the actual rays
    KdTree& tree,                       // the actual kd-tree
    ElemIsect::Intersection<size>& r)   // intersection defined by the nested intersector
{
    typedef BitMask<size>               t_BitMask;
    typedef Packet<size, float>        t_Packet;
    typedef typename t_Packet::Container t_PContainer;
    ...
    while((splitDim = tree.getSplitAxis(node)) != 3) {
        t_PContainer splitValue = t_PContainer::replicate(tree.getSplitValue(node));
        t_BitMask nearChildMask, farChildMask;
        t_PContainer tSplitFactor;

        if(commonOrg) // compile-time constant decision
            tSplitFactor = splitValue - rayPacket.org(0).get(splitDimension);

        for(int i = 0; i < RayPacket<size>::CONTAINER_COUNT; ++i)
        {
            if(!commonOrg) // compile-time constant decision
                tSplitFactor = splitValue - rayPacket.org(i).get(splitDimension);

            const t_PContainer tSplit = tSplitFactor * rayPacket.invDir(i).get(splitDimension);
            nearChildMask.setContainer(i, (tSplit(i) > currentTMax(i)).getIntMask());
            farChildMask.setContainer(i, (currentTMin(i) > tSplit(i)).getIntMask());
        }
        /* omitted: determine the nearest child from masks and descend */
    }
    /* omitted: leaf intersection and stack operations */
}

```

```

template<class ElemIsect>                // nested element intersector
template<int intersDataMask,           // intersection data needed
        bool commonOrg,                // common ray origin?
        unsigned int size,             // size of the ray packet
        class KdTree>                  // models the KdTree concept
void KdTreeIntersector<ElemIsect>::intersect(
    RayPacket<size>& rayPacket,         // the actual rays
    KdTree& tree,                       // the actual kd-tree
    ElemIsect::Intersection<size>& r)   // intersection defined by the nested intersector
{
    typedef BitMask<size>              t_BitMask;
    typedef Packet<size, float>        t_Packet;
    typedef typename t_Packet::Container t_PContainer;
    ...
    while((splitDim = tree.getSplitAxis(node)) != 3) {
        t_PContainer splitValue = t_PContainer::replicate(tree.getSplitValue(node));
        t_BitMask nearChildMask, farChildMask;
        t_PContainer tSplitFactor;

        if(commonOrg) // compile-time constant decision
            tSplitFactor = splitValue - rayPacket.org(0).get(splitDimension);

        for(int i = 0; i < RayPacket<size>::CONTAINER_COUNT; ++i)
        {
            if(!commonOrg) // compile-time constant decision
                tSplitFactor = splitValue - rayPacket.org(i).get(splitDimension);

            const t_PContainer tSplit = tSplitFactor * rayPacket.invDir(i).get(splitDimension);
            nearChildMask.setContainer(i, (tSplit(i) > currentTMax(i)).getIntMask());
            farChildMask.setContainer(i, (currentTMin(i) > tSplit(i)).getIntMask());
        }
        /* omitted: determine the nearest child from masks and descend */
    }
    /* omitted: leaf intersection and stack operations */
}

```

```

template<class ElemIsect>                // nested element intersector
template<int intersDataMask,            // intersection data needed
        bool commonOrg,                 // common ray origin?
        unsigned int size,             // size of the ray packet
        class KdTree>                  // models the KdTree concept
void KdTreeIntersector<ElemIsect>::intersect(
    RayPacket<size>& rayPacket,         // the actual rays
    KdTree& tree,                       // the actual kd-tree
    ElemIsect::Intersection<size>& r)   // intersection defined by the nested intersector
{
    typedef BitMask<size>              t_BitMask;
    typedef Packet<size, float>        t_Packet;
    typedef typename t_Packet::Container t_PContainer;
    ...
    while((splitDim = tree.getSplitAxis(node)) != 3) {
        t_PContainer splitValue = t_PContainer::replicate(tree.getSplitValue(node));
        t_BitMask nearChildMask, farChildMask;
        t_PContainer tSplitFactor;

        if(commonOrg) // compile-time constant decision
            tSplitFactor = splitValue - rayPacket.org(0).get(splitDimension);

        for(int i = 0; i < RayPacket<size>::CONTAINER_COUNT; ++i)
        {
            if(!commonOrg) // compile-time constant decision
                tSplitFactor = splitValue - rayPacket.org(i).get(splitDimension);

            const t_PContainer tSplit = tSplitFactor * rayPacket.invDir(i).get(splitDimension);
            nearChildMask.setContainer(i, (tSplit(i) > currentTMax(i)).getIntMask());
            farChildMask.setContainer(i, (currentTMin(i) > tSplit(i)).getIntMask());
        }
        /* omitted: determine the nearest child from masks and descend */
    }
    /* omitted: leaf intersection and stack operations */
}

```

```

template<class ElemIsect>                // nested element intersector
template<int intersDataMask,            // intersection data needed
        bool commonOrg,                 // common ray origin?
        unsigned int size,             // size of the ray packet
        class KdTree>                  // models the KdTree concept
void KdTreeIntersector<ElemIsect>::intersect(
    RayPacket<size>& rayPacket,         // the actual rays
    KdTree& tree,                       // the actual kd-tree
    ElemIsect::Intersection<size>& r)   // intersection defined by the nested intersector
{
    typedef BitMask<size>               t_BitMask;
    typedef Packet<size, float>        t_Packet;
    typedef typename t_Packet::Container t_PContainer;
    ...
    while((splitDim = tree.getSplitAxis(node)) != 3) {
        t_PContainer splitValue = t_PContainer::replicate(tree.getSplitValue(node));
        t_BitMask nearChildMask, farChildMask;
        t_PContainer tSplitFactor;

        if(commonOrg) // compile-time constant decision
            tSplitFactor = splitValue - rayPacket.org(0).get(splitDimension);

        for(int i = 0; i < RayPacket<size>::CONTAINER_COUNT; ++i)
        {
            if(!commonOrg) // compile-time constant decision
                tSplitFactor = splitValue - rayPacket.org(i).get(splitDimension);

            const t_PContainer tSplit = tSplitFactor * rayPacket.invDir(i).get(splitDimension);
            nearChildMask.setContainer(i, (tSplit(i) > currentTMax(i)).getIntMask());
            farChildMask.setContainer(i, (currentTMin(i) > tSplit(i)).getIntMask());
        }
        /* omitted: determine the nearest child from masks and descend */
    }
    /* omitted: leaf intersection and stack operations */
}

```

Algorithms and Data Structures

Compositions

- Acceleration structures can store objects of any type

```
// data structure  
BVH<KdTree<Triangle>> tree;
```

- Unified intersector interfaces

```
// corresponding intersector  
BVHIntersector<KdTreeIntersector<  
    SimpleTriangleIntersector>> intersector;
```

Rendering Components

Shading

- ⦿ No shading in basic ray tracing components
- ⦿ Integrator – the shading “entry point”
 - > Uses ray tracing components for visibility queries
- ⦿ Two shading models supported
 - > Classic “imperative” (OpenRT- and Manta-like)
 - Convenient and powerful
 - Ray coherence problems
 - > Decoupled “declarative” (PBRT-like)
 - Can preserve ray coherence
 - Limited to physically-based shading

```

struct Integrator {
    template<unsigned int size> struct Result;    // the result type
    template<unsigned int size, class Sample, class Scene,
            class Primitive, class Intersector>
    Color<size> eval(Sample<size>& sample,        // image sample
                    RayPacket<size>& rayPacket, // initial ray packet
                    Primitive& primitive,       // top-level primitive
                    Intersector& intersector,   // top-level intersector
                    Scene& scene);             // shading scene data
};

```

```

struct Material {
    template<unsigned int size,                // packet size
            unsigned int bsdfType>           // BSDF parts to evaluate
    void evaluate(Vec3f<size>& w_o,           // outgoing direction
                 Vec3f<size>& w_i,           // incoming direction
                 ShadingData<size>& sh,      // hit point, normal, etc.
                 Vec3f<size>& result);       // returns computed radiance
    ...
};

```

```

template<class Context>                        // for materials, lights, and rays
struct SurfaceShader {
    template<bool commonOrg,                  // common ray origin?
            unsigned int size>              // packet size
    void shade(const RayPacket<size>& rayPacket, // the actual rays
               ShadingData<size>& intersection, // ray intersection structure
               Context& context,              // rendering context
               Vec3f<size>& result);          // returns computed radiance
};

```

```

struct Integrator {
    template<unsigned int size> struct Result;    // the result type
    template<unsigned int size, class Sample, class Scene,
            class Primitive, class Intersector>
    Color<size> eval(Sample<size>& sample,        // image sample
                    RayPacket<size>& rayPacket, // initial ray packet
                    Primitive& primitive,       // top-level primitive
                    Intersector& intersector,   // top-level intersector
                    Scene& scene);              // shading scene data
};

```

```

struct Material {
    template<unsigned int size,                // packet size
            unsigned int bsdfType>           // BSDF parts to evaluate
    void evaluate(Vec3f<size>& w_o,           // outgoing direction
                 Vec3f<size>& w_i,           // incoming direction
                 ShadingData<size>& sh,      // hit point, normal, etc.
                 Vec3f<size>& result);       // returns computed radiance
    ...
};

```

```

template<class Context>                        // for materials, lights, and rays
struct SurfaceShader {
    template<bool commonOrig,                 // common ray origin?
            unsigned int size>              // packet size
    void shade(const RayPacket<size>& rayPacket, // the actual rays
               ShadingData<size>& intersection, // ray intersection structure
               Context& context,              // rendering context
               Vec3f<size>& result);          // returns computed radiance
};

```

```

struct Integrator {
    template<unsigned int size> struct Result;    // the result type
    template<unsigned int size, class Sample, class Scene,
            class Primitive, class Intersector>
    Color<size> eval(Sample<size>& sample,        // image sample
                    RayPacket<size>& rayPacket, // initial ray packet
                    Primitive& primitive,       // top-level primitive
                    Intersector& intersector,   // top-level intersector
                    Scene& scene);              // shading scene data
};

```

```

struct Material {
    template<unsigned int size,                // packet size
            unsigned int bsdfType>           // BSDF parts to evaluate
    void evaluate(Vec3f<size>& w_o,           // outgoing direction
                 Vec3f<size>& w_i,           // incoming direction
                 ShadingData<size>& sh,      // hit point, normal, etc.
                 Vec3f<size>& result);       // returns computed radiance
    ...
};

```

```

template<class Context>                        // for materials, lights, and rays
struct SurfaceShader {
    template<bool commonOrg,                  // common ray origin?
            unsigned int size>              // packet size
    void shade(const RayPacket<size>& rayPacket, // the actual rays
               ShadingData<size>& intersection, // ray intersection structure
               Context& context,              // rendering context
               Vec3f<size>& result);          // returns computed radiance
};

```

Rendering Pipelines

- Renderer – a top-level concept for processing image tiles

```

template<class PixelSampler, class Integrator>
class RayTracingRenderer : public Renderer {
    PixelSampler m_sampler;
    Integrator    m_integrator;
public:
    template<unsigned int size,                // the size of primary ray packets
            class Camera, class Scene, class Primitive,
            class Intersector, class Framebuffer>
    void render(
        Scene& scene, Camera& camera,
        Framebuffer& framebuffer, ImageClipRegion& clip,
        Primitive& primitive, Intersector& intersector)
    {
        PixelSampler::Sample<size> sample;
        PixelSampler::Iterator<size> it =
            m_sampler.getIterator<size>(clip);
        while(it.getNextSample(sample))
        {
            RayPacket<size> rays = camera.genRay(sample);
            Integrator::Color<size> color = m_integrator.eval(
                sample, rays, primitive, intersector, scene);
            m_sampler.writeColor(sample, color, framebuffer);
        }
    }
    /* omitted: preprocess() function for pre-integration*/
};

```

```

template<class PixelSampler, class Integrator>
class RayTracingRenderer : public Renderer {
    PixelSampler m_sampler;
    Integrator    m_integrator;
public:
    template<unsigned int size,                // the size of primary ray packets
            class Camera, class Scene, class Primitive,
            class Intersector, class Framebuffer>
    void render(
        Scene& scene, Camera& camera,
        Framebuffer& framebuffer, ImageClipRegion& clip,
        Primitive& primitive, Intersector& intersector)
    {
        PixelSampler::Sample<size> sample;
        PixelSampler::Iterator<size> it =
            m_sampler.getIterator<size>(clip);
        while(it.getNextSample(sample))
        {
            RayPacket<size> rays = camera.genRay(sample);
            Integrator::Color<size> color = m_integrator.eval(
                sample, rays, primitive, intersector, scene);
            m_sampler.writeColor(sample, color, framebuffer);
        }
    }
    /* omitted: preprocess() function for pre-integration*/
};

```

```

template<class PixelSampler, class Integrator>
class RayTracingRenderer : public Renderer {
    PixelSampler m_sampler;
    Integrator    m_integrator;
public:
    template<unsigned int size,                // the size of primary ray packets
            class Camera, class Scene, class Primitive,
            class Intersector, class Framebuffer>
    void render(
        Scene& scene, Camera& camera,
        Framebuffer& framebuffer, ImageClipRegion& clip,
        Primitive& primitive, Intersector& intersector)
    {
        PixelSampler::Sample<size> sample;
        PixelSampler::Iterator<size> it =
            m_sampler.getIterator<size>(clip);
        while(it.getNextSample(sample))
        {
            RayPacket<size> rays = camera.genRay(sample);
            Integrator::Color<size> color = m_integrator.eval(
                sample, rays, primitive, intersector, scene);
            m_sampler.writeColor(sample, color, framebuffer);
        }
    }
    /* omitted: preprocess() function for pre-integration*/
};

```

```
template<class PixelSampler, class Integrator>
class RayTracingRenderer : public Renderer {
    PixelSampler m_sampler;
    Integrator m_integrator;
public:
    template<unsigned int size, // the size of primary ray packets
            class Camera, class Scene, class Primitive,
            class Intersector, class Framebuffer>
    void render(
        Scene& scene, Camera& camera,
        Framebuffer& framebuffer, ImageClipRegion& clip,
        Primitive& primitive, Intersector& intersector)
    {
        PixelSampler::Sample<size> sample;
        PixelSampler::Iterator<size> it =
            m_sampler.getIterator<size>(clip);
        while(it.getNextSample(sample))
        {
            RayPacket<size> rays = camera.genRay(sample);
            Integrator::Color<size> color = m_integrator.eval(
                sample, rays, primitive, intersector, scene);
            m_sampler.writeColor(sample, color, framebuffer);
        }
    }
    /* omitted: preprocess() function for pre-integration*/
};
```

```

template<class PixelSampler, class Integrator>
class RayTracingRenderer : public Renderer {
    PixelSampler m_sampler;
    Integrator    m_integrator;
public:
    template<unsigned int size,                // the size of primary ray packets
             class Camera, class Scene, class Primitive,
             class Intersector, class Framebuffer>
    void render(
        Scene& scene, Camera& camera,
        Framebuffer& framebuffer, ImageClipRegion& clip,
        Primitive& primitive, Intersector& intersector)
    {
        PixelSampler::Sample<size> sample;
        PixelSampler::Iterator<size> it =
            m_sampler.getIterator<size>(clip);
        while(it.getNextSample(sample))
        {
            RayPacket<size> rays = camera.genRay(sample);
            Integrator::Color<size> color = m_integrator.eval(
                sample, rays, primitive, intersector, scene);
            m_sampler.writeColor(sample, color, framebuffer);
        }
    }
    /* omitted: preprocess() function for pre-integration*/
};

```

```

template<class PixelSampler, class Integrator>
class RayTracingRenderer : public Renderer {
    PixelSampler m_sampler;
    Integrator    m_integrator;
public:
    template<unsigned int size,                // the size of primary ray packets
            class Camera, class Scene, class Primitive,
            class Intersector, class Framebuffer>
    void render(
        Scene& scene, Camera& camera,
        Framebuffer& framebuffer, ImageClipRegion& clip,
        Primitive& primitive, Intersector& intersector)
    {
        PixelSampler::Sample<size> sample;
        PixelSampler::Iterator<size> it =
            m_sampler.getIterator<size>(clip);
        while(it.getNextSample(sample))
        {
            RayPacket<size> rays = camera.genRay(sample);
            Integrator::Color<size> color = m_integrator.eval(
                sample, rays, primitive, intersector, scene);
            m_sampler.writeColor(sample, color, framebuffer);
        }
    }
    /* omitted: preprocess() function for pre-integration*/
};

```

Putting It All Together

◉ Example application

```
PinholeCamera camera;
OpenGLFramebuffer fb;

BasicScene<Triangle> scene; //initialization omitted
BVH<Triangle> tree;
BVHBuilder builder;
BVHIntersector<PlueckerTriangleIntersector> intersector;
RayTracingRenderer<PixelCenterSampler,
                  DirectIlluminationIntegrator> renderer;

builder.build(tree, scene.prim.begin(), scene.prim.end());

renderer.render<64>(scene, camera, fb, fb.getClipRegion(),
                  tree, intersector);
```

Putting It All Together

◉ Example application

```
PinholeCamera camera;
OpenGLFramebuffer fb;

BasicScene<Triangle> scene; //initialization omitted
BVH<Triangle> tree;
BVHBuilder builder;
BVHIntersector<PlueckerTriangleIntersector> intersector;
RayTracingRenderer<PixelCenterSampler,
                  DirectIlluminationIntegrator> renderer;

builder.build(tree, scene.prim.begin(), scene.prim.end());

renderer.render<64>(scene, camera, fb, fb.getClipRegion(),
                  tree, intersector);
```

Putting It All Together

◉ Example application

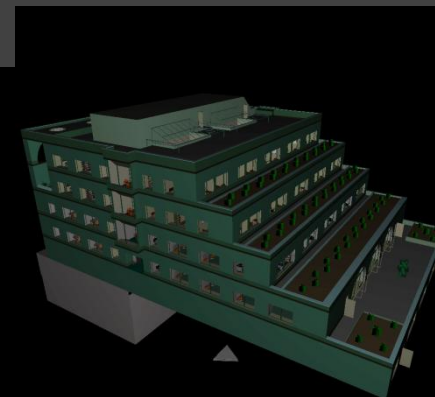
```
PinholeCamera camera;
OpenGLFramebuffer fb;

BasicScene<Point> scene; //initialization omitted
LoDKdTree<Point> tree;
LoDKdTreeBuilder builder;
LoDKdTreeIntersector<PointIntersector> intersector;
RayTracingRenderer<PixelCenterSampler,
                  LoDIntegrator> renderer;

builder.build(tree, scene.prim.begin(), scene.prim.end());

renderer.render<16>(scene, camera, fb, fb.getClipRegion(),
                  tree, intersector);
```

Performance

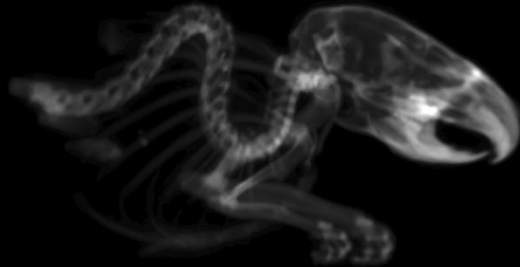


packet kd-tree	Sponza	Conference	Soda Hall
OpenRT	4.5	4.2	5.1
Manta	4.7	4.2	5.4
RTfact	6.8	6.4	6.5
frustum BVH	Sponza	Conference	Soda Hall
DynBVH	N/A	9.3	11.1
Manta	4.5	4.8	5.6
Arauna	13.2	11.3	N/A
RTfact	13.1	11.6	11.4

Applications



David (3.7M points)



Mouse (CT scan)



Dragon (3.6M points)



Engine (CT scan)

Conclusions and Future Work

◎ RTfact

- > Generic ray tracing framework
- > Composability and extensibility
- > High performance

◎ Not a perfect solution

- > Many language limitations

◎ Future

- > Release as open source
- > Integrate into a scene graph library
- > To be used in DFKI visualization center
- > Explore other language approaches

Questions?