Adding Custom Intersectors to the C++ Ray Tracing Template Library Visionaray

Stefan Zellmann[†]

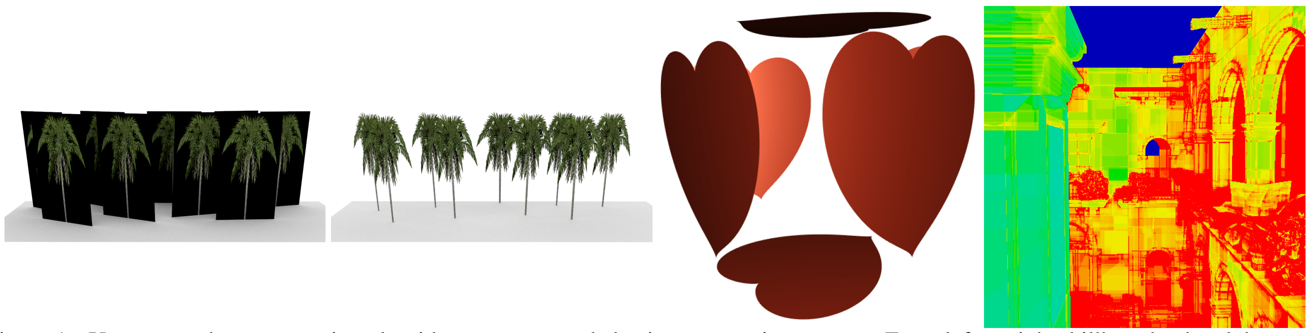

Figure 1: Use cases where ray tracing algorithms were extended using custom intersectors. From left to right: billboards, the alpha mask stored in the billboard images is ignored. Second from left: the alpha mask from the billboard images is used to conditionally continue BVH traversal when the surface has zero opacity near the hit point. Second from right: procedural alpha mask applied using custom intersectors. Right: debug image, the number of BVH nodes and the number of primitives inside the encountered leaf nodes is used to generate a heat map. This is done by intercepting the BVH traversal routine with a custom intersector that counts the number of ray object interactions.

Abstract

Most ray tracing libraries allow the user to provide custom functionality that is executed when a potential ray surface interaction was encountered to determine if the interaction was valid or traversal should be continued. This is e.g. useful for alpha mask validation and allows the user to reuse existing ray object intersection routines rather than reimplementing them. Augmenting ray traversal with custom intersection logic requires some kind of callback mechanism that injects user code into existing library routines. With template libraries, this injection can happen statically since the user compiles the binary code herself. We present an implementation of this "custom intersector" approach and its integration into the C++ ray tracing template library Visionaray.

1. Introduction

Typical ray tracing libraries provide a default implementation for some primitive type—e.g. for triangles—but allow the user to extend the library in several ways. One such way might be support for completely new primitive types that behave differently than the default primitive type. Often, the functionality that the user desires however generally maps to the default primitive type, but is instead an extension to the default behavior of that type. One such example is support for alpha masks to selectively carve out areas from planar surfaces based on a 2-d texture lookup. That functionality can be implemented by extending the ray primitive intersection routine; the traversal algorithm that tests the ray against a number of surfaces therefore first performs the default intersection test, but instead of immediately reporting an intersection first performs a lookup in the alpha texture and only reports the hit if the lookup indicated the primitive was fully opaque at the intersection position.

The functionality just described is usually implemented by extending the intersection algorithm which will call some type of callback mechanism whenever a potential hit was reported, and only report an actual hit if the callback mechanism also reports an intersection. A default implementation might just always report an intersection, no matter what the actual intersection position was. This type of interface to the library can for example be found in Embree [WWB*14] where it is called an *intersection filter* or in OptiX [PBD*10] where the functionality can be achieved by implementing a custom *any-hit program*.

In the context of libraries like Embree or OptiX that the program-

[†] zellmann@uni-koeln.de, Department of Computer Science, University of Cologne

mer integrates into her application by means of static or dynamic linking, this extension mechanism must be evaluated at runtime: the library will perform some type of runtime check if an intersection filter or any-hit program was registered and only execute some custom functionality if a function pointer or some other means to conditionally execute the user-supplied routine was properly initialized.

With template libraries like Visionaray [ZWL17], this check can instead be performed at compile time and will incur zero cost if no custom code was supplied by the user. As ray surface intersections are evaluated in the innermost loop of the ray tracing algorithm, avoiding additional runtime checks at this phase might improve overall performance.

In contrast to typical ray tracing libraries like Embree or OptiX, Visionaray is a ray tracing template library where most of the functionality resides in C++ header files and is directly compiled into the user's application. This has the advantage that the code can be inlined and optimized by the compiler in the context of the application program. The approach also has certain disadvantages, such as increased compile time for the application programmer, or the fact that the application programer needs to make sure and rely on her compiler that the program is properly optimized. An advantage of the approach however is that code that is not needed is never actually compiled into the application and can thus not have a negative impact for example on instruction cache utilization.

In this paper we describe the integration of *custom intersectors* as an application programming interface for static routines that the user implements and passes to Visionaray at compile time.

2. Related Work

Visionaray provides support for several features that are required to develop ray tracing algorithms, such as a streamlined texture interface that can be leveraged on both CPUs with vector instructions as well as on NVIDIA GPUs [ZPL15]. Visionaray supports several types of intersection queries, including the multi-hit query type [ZHL17]. Support for multiple primitive, material or light types in the same ray tracing program is provided by means of compile type polymorphism. The effectiveness of that approach was thoroughly evaluated in [ZL17]. We used Visionaray and its various library subsystems such as the vector math system or SIMD library component to implement several algorithms, e.g. the ones from [ZSL18], [ZHL19], [ZSL19], and [ZML19].

3. Integration of Custom Intersectors into Visionaray

In this section we first describe the application programming interface (API) by which custom intersectors can be used by the application programmer, and then provide details about how that was internally implemented in the library.

3.1. Application Programming Interface

Visionaray has a *customization point* interface where the user can overwrite or augment behavior by implementing free functions for custom types. Custom geometric primitives e.g. can be added by implementing a set of free functions, one of them being the `intersect` function that tests if a ray intersects the primitive:

```
template <typename Ray>
hit_record<Ray, primitive<unsigned>> intersect(
        Ray const& ray,
        custom_primitive const& prim
        );
```

`custom_primitive` in this case is a user-defined type, and the `hit_record` template contains a member variable `hit` that indicates whether an intersection occurred or not.

Custom intersectors allow to augment the behavior of *existing* primitive types like triangles; the user may e.g. be perfectly fine with the triangle intersection routine as such (and may also wish to reuse the existing builtin triangle type instead of implementing a completely new one), but wants to add an alpha mask from an image texture.

The API for that consists of deriving from the `basic_intersector` template using the "curiously recurring template pattern" (CRTP):

```
struct custom_intersector
    : basic_intersector<custom_intersector> {
    using basic_intersector<
        custom_intersector
        >::operator();

    // Implementation goes here
    ...
```

The user then implements her own `operator()` as a member function, with the signature of the `intersect` function from before, with the ray as first and the custom primitive as second parameter:

```
    ...
    template <typename Ray>
    hit_record<Ray, primitive<unsigned>>
    operator()(
            Ray const& ray,
            custom_primitive const& prim
            )
    {
        auto hr = intersect(ray, prim);

        // use hit record, e.g. barycentrics
        // for texture lookups
        ...

        // manipulate the hit record
        hr.hit &= ...;

        // after manipulation, return
        return hr;
    }
};
```

Visionaray supports several visibility queries; usually, the ray is tested against a *bounding volume hierarchy* (BVH) built over some primitives, and either the closest hit point (closest-hit query), the first encountered hit point (any-hit query), or the first N hit points (multi-hit query) are returned. The interface for the various queries is similar and exemplarily is presented here for the closest-hit query:

```
// Default overload
template <
    typename Ray,
    typename PrimIterator
    >
auto closest_hit(
        Ray          ray,
        PrimIterator first_prim,
        PrimIterator last_prim
        )
    -> decltype(intersect(ray, *first_prim));

// Overload w/ custom intersector
template <
    typename Ray,
    typename PrimIterator,
    typename Intersector
    >
auto closest_hit(
        Ray          ray,
        PrimIterator first_prim,
        PrimIterator last_prim,
        Intersector  intersector
        )
    -> decltype(intersect(ray, *first_prim));
```

Visionaray comes with a set of default kernels that implement various algorithms like path tracing or plain primary visibility ray casting; those algorithms make use of the aforementioned visibility queries and can be passed a custom intersector using the scheduling parameters (for more details see [ZWL17]).

3.2. Implementation Notes

Visionaray's bounding volume implementation that is based on the `while-while` traversal scheme from [AL09] calls `intersect` twice. A high-level representation of the traversal scheme looks like this:

```
procedure INTERSECT(ray, BVH)
    while ray not terminated do
        while node is inner do
            INTERSECT(ray, node.bounds)
        end while
                                        ▷ Found a leaf
        while node contains untested primitives do
            HitRecord ← INTERSECT(ray, node.prims++)
        end while
    end while
end procedure
```

The traversal function, at compile time, is passed the custom intersector class, and the two calls to `intersect`—the one that tests the ray against the bounding box of the BVH nodes and the one that tests against the individual primitives—are statically replaced with the calls to `operator()` provided by the custom intersector.

Also note how `intersect` inside the BVH traversal routine is not only called for each geometric primitive, but also when the ray is tested against the BVH nodes' bounds (which have type `aabb` with Visionaray). Custom intersectors thus cannot only be used to intercept the behavior of ray vs. primitive intersection, but also that of testing rays against BVH nodes. This can be accomplished by adding an `operator()` overload to the custom intersector that takes an `aabb` as second parameter.

An implementation detail worth mentioning is that the ray vs. BVH traversal function in Visionaray is also called `inter-sect`. The reasoning behind this is that the visibility queries (`closest_hit`, `any_hit` and `multi_hit`) will iterate linearly over a list, where BVHs may themselves act as (compound) primitives. This allows us to easily implement object instancing, where the BVH will store BVHs as primitives, and where the object hierarchy may optionally have more than one root node. Conversely, in certain cases the user might decide that a BVH is not required and just pass iterators to a linear list of primitives to the query routines. Special care is necessary to support this behavior: when the visibility query is executed on a list of primitives that are not composed into a BVH, the custom intersector replaces the respective call to `intersect` inside the traveral loop. When the query is however executed on a list of BVHs, the custom intersector is passed on to the BVH intersection routine, which will replace its respective calls to `intersect`. Discerning the two implementations is done at compile time using the "substitution failure is not an error" (SFINAE) pattern.

4. Use Cases

Custom intersectors can e.g. be used to implement the use cases from Figure 1. 3-d models often come with separate alpha masks stored in texture images that are used to carve out details from the otherwise coarse geometry that serves as an impostor or a billboard. This can easily be implemented by providing a custom intersector which stores a pointer to the texture and texture coordinate lists as member variables. The custom intersector provides an `opera-tor()` that intercepts interactions with the surface geometry and performs a texture lookup using the barycentric coordinates at the hit point to determine if the surface was actually hit. It will then manipulate the hit point based on the alpha information from the mask texture and only then return the hit record:

```
template <typename Ray>
auto operator()(
        Ray const& ray,
        basic_triangle<3, float> const& tri
        )
{
    auto hr = intersect(ray, tri);

    auto const& tex = textures[hr.geom_id];
    vec2 coord = lerp(
            tex_coords[hr.prim_id * 3],
            tex_coords[hr.prim_id * 3 + 1],
            tex_coords[hr.prim_id * 3 + 2],
            hr.u, hr.v
            );
    vec4 color = tex2D(tex, coord);

    hr.hit &= color.w >= .01f;

    return hr;
}
```

Another use case that is depicted in the second from right image of Figure 1 is procedural alpha masking which can be implemented in a similar way that alpha masking with texture images is implemented, but uses a procedural to determine visibility when passed the uv coordinates.

A third use case that is depicted in the right image of Figure 1 is spotting and visualizing performance issues with the intersection routine or the BVH traversal. Therefore, a custom intersector is implemented that just counts the number of interactions. A full implementation might look like this:

```
struct bvh_costs : basic_intersector<bvh_costs> {
    using basic_intersector<
        bvh_costs
        >::operator();

    // Intercept and count ray/aabb tests
    template <typename Ray, typename ...Args>
    auto operator()(
            Ray const& ray,
            aabb const& box,
            Args&&... args
            )
    {
        ++num_boxes;
        return intersect(
                ray,
                box,
                std::forward<Args>(args)...
                );
    }

    // Intercept and count ray/triangle tests
    template <typename Ray>
    auto operator()(
            Ray const& ray,
            basic_triangle<3, float> const& tri
            )
    {
        ++num_tris;
        return intersect(ray, tri);
    }

    unsigned num_boxes = 0;
    unsigned num_tris  = 0;
};
```

After performing ray vs. BVH traversal with the bvh_costs intersector, the number of interactions with bounding boxes and with triangles will be stored in the variables num_boxes and num_tris respectively and can be used for further analysis or visualization. Note that the aabb intersection function has a special interface as it also takes the inverse ray direction as a parameter. This is opaquely handled by the custom intersector, which just passes any additional parameters on to the intersection function using variadic templates. The implementation provides an example of how to support intersection functions with a non-standard interface.

5. Conclusion

We presented custom intersectors as a way to augment the ray tracing template library Visionaray. Custom intersectors or similar concepts are supported by many ray tracing libraries, but due to the static and compile time nature of Visionaray, the feature is supported in a unique way that is different from the usual function pointer approach that other libraries implement. We presented the API to make use of custom intersectors, some implementation notes, and also some sample implementations that exemplarily present how to use that feature with Visionaray.

References

[AL09] AILA T., LAINE S.: Understanding the efficiency of ray traversal on GPUs. In *Proceedings of the Conference on High Performance Graphics 2009* (New York, NY, USA, 2009), HPG '09, ACM, pp. 145–149. 3

[PBD*10] PARKER S. G., BIGLER J., DIETRICH A., FRIEDRICH H., HOBEROCK J., LUEBKE D., MCALLISTER D., MCGUIRE M., MORLEY K., ROBISON A., STICH M.: OptiX: A general purpose ray tracing engine. *ACM Trans. Graph. 29*, 4 (July 2010), 66:1–66:13. 1

[WWB*14] WALD I., WOOP S., BENTHIN C., JOHNSON G. S., ERNST M.: Embree: A kernel framework for efficient CPU ray tracing. *ACM Trans. Graph. 33*, 4 (July 2014), 143:1–143:8. 1

[ZHL17] ZELLMANN S., HOEVELS M., LANG U.: Ray traced volume clipping using multi-hit BVH traversal. In *Proceedings of Visualization and Data Analysis (VDA)* (2017), IS&T. 2

[ZHL19] ZELLMANN S., HELLMANN M., LANG U.: A linear time BVH construction algorithm for sparse volumes. In *Proceedings of the 12th IEEE Pacific Visualization Symposium* (2019), IEEE. 2

[ZL17] ZELLMANN S., LANG U.: C++ compile time polymorphism for ray tracing. In *Proceedings of the Conference on Vision, Modeling and Visualization* (Goslar Germany, Germany, 2017), VMV '17, Eurographics Association, pp. 129–136. 2

[ZML19] ZELLMANN S., MEURER D., LANG U.: Hybrid grids for sparse volume rendering. In *IEEE VIS 2019 - Short Papers* (2019). 2

[ZPL15] ZELLMANN S., PERCAN Y., LANG U.: Advanced texture filtering: a versatile framework for reconstructing multi-dimensional image data on heterogeneous architectures. In *Visualization and Data Analysis 2015* (2015), Kao D. L., Hao M. C., Livingston M. A., Wischgoll T., (Eds.), vol. 9397, International Society for Optics and Photonics, SPIE, pp. 110 – 120. 2

[ZSL18] ZELLMANN S., SCHULZE J. P., LANG U.: Rapid k-d tree construction for sparse volume data. In *Eurographics Symposium on Parallel Graphics and Visualization* (2018), Childs H., Cucchietti F., (Eds.), The Eurographics Association. 2

[ZSL19] ZELLMANN S., SCHULZE J. P., LANG U.: Binned k-d tree construction for sparse volume data on multi-core and GPU systems. *IEEE Transactions on Visualization and Computer Graphics* (2019), 1–1. 2

[ZWL17] ZELLMANN S., WICKEROTH D., LANG U.: Visionaray: A cross-platform ray tracing template library. In *Proceedings of the 10th Workshop on Software Engineering and Architectures for Realtime Interactive Systems (IEEE SEARIS 2017)* (in press, 2017), IEEE. 2, 3