Faster Ray Casting

In this assignment we will first implement a ray traversal algorithm that has sub-linear cost (in contrast to linear search over all triangles as we did in the last exercise). We will then speed that algorithm up considerably by implementing a number of optimizations.

Setting Up

The remarks on the last sheet still apply, the tarball is called `rtgi-2021-a02.tar.bz2`. Make sure to copy over both of the additional models supplied on the GRIPS page (as subdirectories in `render-data/`).

Context

The distribution tarball contains three scenes, the first, `a2-tris` is very simple and only holds 2 triangles. `a2-sibenik` is the scene that you know from the last exercise, with a low number of samples per pixel and limited resolution you can render it with the sequential triangle intersection method. `a2-sponza` is a larger scene that will take a long time to render with the sequential approach.

The code in `rt/bbvh/bvh.h` holds two variants of a BVH that can be selected via commands as defined in `interaction.cpp` and exemplified in the distributed script files.

The next assignment sheet will be a bit shorter, while this current one is a bit longer, so never mind taking some of the assignments over to the next week. ;)

Assignment 1

Implement the ray/box intersection method described in the lecture in `libgi/intersect.h` function `intersect`. Test it directly in main for a few rays to be sure it works independently from anything else.

Assignment 2

As with the sequential triangle intersection method, the new ray tracers have a `build`, `closest_hit` and `any_hit` methods.

Implement `naive_bvh::subdivide` in `rt/bbvh/bvh.cpp` to compute an object-median BVH from the input primitives, as described in the lecture. Then, implement `naive_bvh::closest_hit`. Note that we do not use pointers to connect the node to its children but indices into a vector (i.e. an array) of nodes – this is more compact and usually sufficient. When building the BVH add new nodes to the `nodes` vector.

You can verify your implementation with the simple two-triangles scene, it should be possible to follow the algorithm by print statements or in the debugger for this simple case.

Note: The nodes are stored in a vector that grows dynamically in each call to `subdivide`. Thus, a reference to a node from that vector may be invalidated after a recursive call to `subdivide`. This can be a subtle bug!

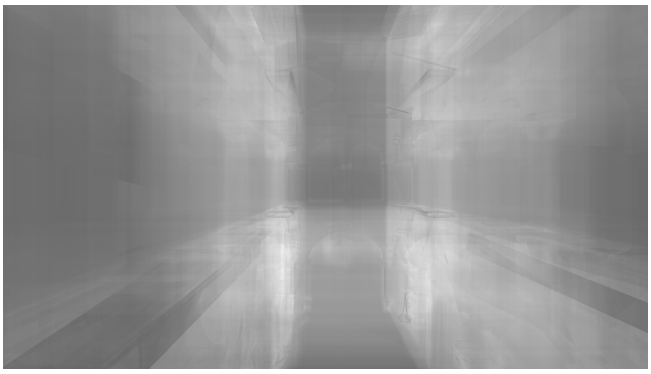
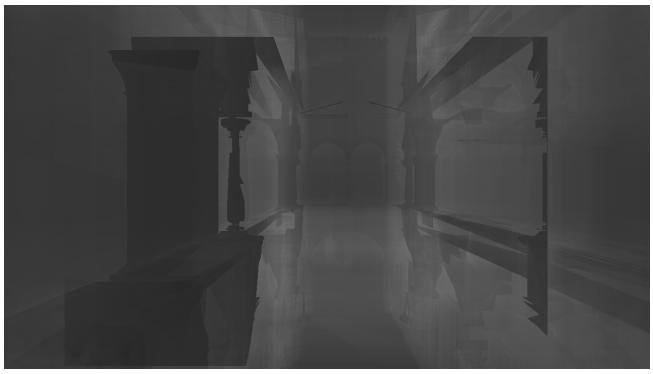
Note: Implement the traversal with an explicit stack of node indices, not recursively, e.g. with `int stack[24], sp=0;`. For each node popped from the stack, load the node, intersect the bounding geometry with the ray and if the intersection is valid, push the node's children on the stack.

Assignment 3

Now we turn from `naive_bvh` to `binary_bvh`. Actually, as you have seen, both are binary, but the former is naive as well. The naive part ist how nodes are stored and traversed: imagine a node with two children, such that one child is closer to the ray's origin and one is further away. If there is an intersection in the closer child, then it is very likely that any intersection in ther farther child will be farther from the origin. Therefore, it makes sense to always descend into the closer child node first and not descend into a node at all if the ray/box intersection point is behind the closest intersection found so far.

To allow this scheme, `binary_bvh_tracer::node` stores the boxes of both child nodes instead of its own box. Implement a variant of your `naive_bvh::subdivide` method from the previous assignment to support this different node layout in `binary_bvh_tracer::subdivide_om`. Then, implement the traversal code in `binary_bvh_tracer::closest_hit` such that it follows the scheme described above.

Include a version of this code (as well as in the previous traversal method in `naive_bvh`) (e.g. via a preprocessor conditional) that counts, per pixel, how many nodes were visited during traversal and return that value as a grey-scale color code. The results (example below) should indicate quite clearly why this new version is more efficient. Verify this also by looking at the render times printed to the terminal.

Assignment 4

As described in the lecture, a better BVH can be obtained when using a spatial median, and even constructed faster than by object median. Implement `binary_bvh_tracer::subdivide_sm` to construct such a BVH (traversal code does not have to be touched for this). Be sure to avoid the potential infinite recursion (it should be triggered by the two-triangle scene).

Assignment 5

When you look at your ray/box intersection code (cf. the reference solution if this does not match your implementation) you will notice that a lot of the code manages the case where any of the ray direction's components can be 0. Since we rely on full IEEE floating point numbers we can make full use of this and exploit that $\frac{+x}{0} \mapsto +\infty$ and that $\frac{-x}{0} \mapsto -\infty$. We can use this to transparently manage those cases where the direction is 0 along an axis.

This is described in more detail in Shirley's book (an excerpt is available on the course website).

Implement this scheme in the function `intersect2`. Pick a viewpoint in Sibenik and Sponza and compare the rendering times for the naive BVH, the new BVH with `intersect` and the new BVH with `intersect2`. Save the images and record the render times.

Assignment 6

A very subtle optimization is by the fact that the C-library (also the C++-library) minimum operations are quite expensive as they take care of proper handling of all possible IEEE cases, including equality.

It is often more efficient to find the minimum of two values by code such as this:

```
float min = a<b ? a : b;
```

For this assignment it suffices to implement the compares as just described, convince yourself by the result and just keep in mind that there are cases where this can be problematic. But if you want to know more, then this stackoverflow-answer is a good starting point to understand it better:

https://stackoverflow.com/questions/40196817/what-is-the-instruction-that-gives-branchless-fp-min-and-max-on-x86/40199125#40199125

Assignment 7

Furthermore, division is usually a very expensive operation, even with floating point numbers. Copy your `intersect2` code over to `intersect3`, initialize a variable at the top of the function to hold the inverse directions and replace all divisions by the ray direction with multiplications of this factor.

Assignment 8

The framework code actually already provides rays with an `id` component that holds $\frac{1}{d}$. Copy over your `intersect3` to `intersect4` using this factor instead of a factor that is computed for each ray/box intersection.

Take the times of these two further optimizations and add them to your performance results.

Happy Hacking :)