3.2 Coding Exercise (25 + 25 Points)

In this exercise, we will deal with spacial acceleration structures. Implement everything within `src/exercise03.cpp`. As before, there should be no need to touch any of the other files.
Please make yourself familiar with the rest of the given code.

a) Implement the construction of the Bounding Volume Hierarchy (`BVH`):

```
void BVH::construct(const Mesh& mesh)
```

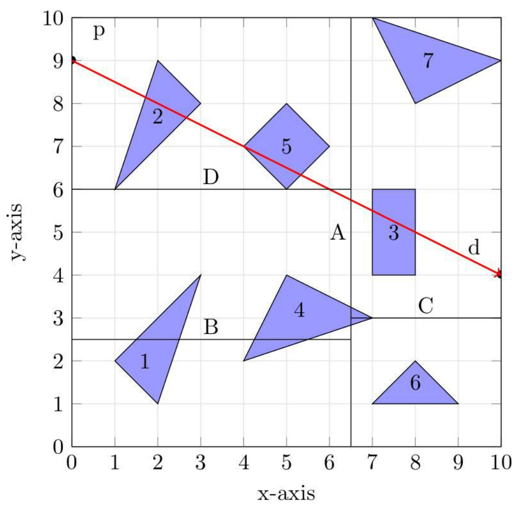


Figure 3: Multiple primitives in a 10×10 grid.

The root node at index `0` is already given and contains all `Triangles` in the mesh. It is your task to recursively (using a loop, not recursion) subdivide this node as long as it contains more than `numFacesPerLeaf = 4` triangles (faces). Stop creating new leaf nodes beyond `maxDepth = 20`.

To perform a subdivision, check your current bounding box of your `BVH::Node` and divide it along the axis with the largest extent. Split the contained `Triangles` into two equally sized halves, ordered by their `triangleCenter` in the chosen split dimension (median split).

Each `BVH::Node` contains the indices [`facesBegin`, `facesEnd`) containing the first and one past the last index into the `BVH`'s `faceIndices` vector. It contains the index of all triangles (faces). Initially, all these indices are simply `0..numFaces-1`. To sort them into the left and right `BVH::Nodes`, we don't change the given `Mesh`, but rather only permute this `faceIndices` vector.

The order of the faces within the child nodes is not important, so you can use `std::nth_element` to efficiently perform the median split. You will have to give it a comparison operator (e.g. as a lambda function) which compares two triangle centers in the given split dimension and returns true if the first one is smaller than the second one.

To get all face indices belonging to a node, you can use the helper function `std::span<uint32_t> getFaceIndices(const Node& node)`. (You can loop over that with a range-based for loop.)

To get a `Triangle` from its index, use `mesh.getTriangleFromFaceIndex(faceIndex)`.

After computing the split, create two new leaf nodes and assign each half to one of them. Compute their bounding boxes based on the assigned triangles and add them to the `nodes` vector.

You should end up with a full binary tree, where the left and right child nodes of every node `i` are at the indices `2*i+1` and `2*i+2`. See Figure 4 for an example.

b) Implement the accelerated intersection test for `Rays` and `Meshes`:

```
Intersection intersect(const Mesh& mesh, const IntersectionRay& ray)
```

Return the closest found intersection with a triangle in the `Mesh`. Traverse the `BVH`'s `Nodes` to quickly exclude the majority of triangles from being tested. With a good `BVH`, the **`dragon.obj`** model should render within just a few seconds instead of minutes.

In a bit more detail: Start at the root node (index `0`). Then, for all nodes you check: If the ray intersects its bounding box, also check both its child nodes. When there are no child nodes, instead test all its contained `Triangles` for intersection with the ray.

A node is a leaf node, if its child node indices would be outside of the `node` vector or if the child node's `facesEnd` is `0` (this can happen when `numFacesPerLeaf` has been reached before – but that should not happen when using the median split).

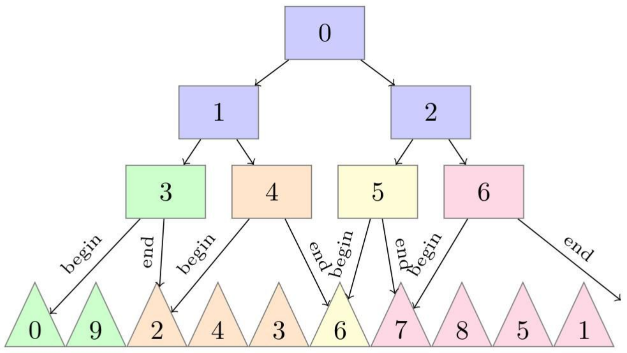


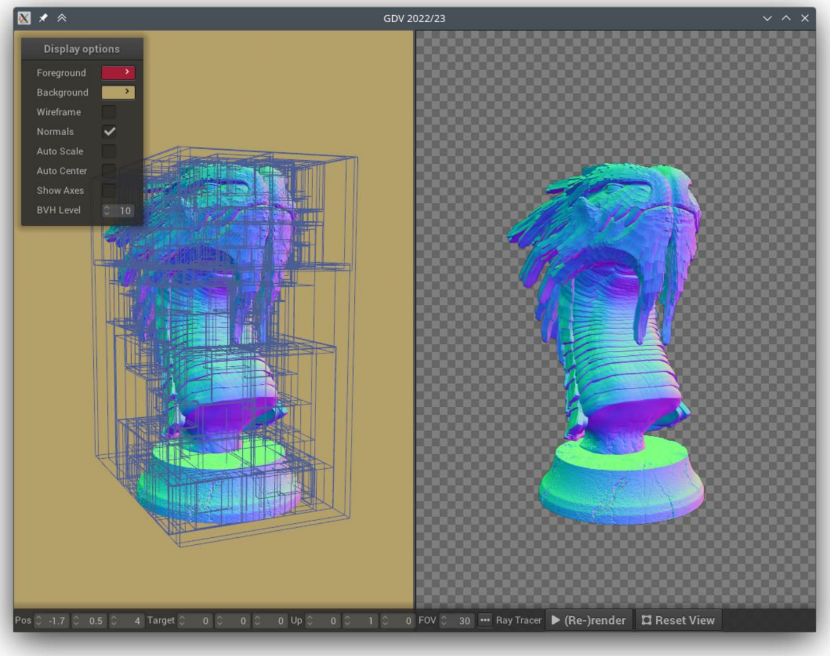
Figure 4: Example of a `BVH`'s `nodes` (on top) and the assigned `faceIndices` (on the bottom).

> If there is no intersection, the value `t` of the `Intersection` `struct` needs to remain at ∞.
>
> Optional: It is possible to traverse the entire `BVH` without even keeping a stack of nodes to visit. Since we're dealing with a full binary tree, you can easily figure out the indices of the left child, right sibling and parent to traverse the BVH in a fixed order (parent, left child, right child, ...).

Tracing rays in "Debug" mode is very slow. Consider using the "Release" or "RelWithDebInfo" mode for these tasks. To test your implementation, simply make sure that you render the same image on the right with your ray tracer as the OpenGL preview computes on the left.

A new feature this time is the visualization of your Bounding Volume Hierarchy: The "BVH Level" shows all bounding boxes in your `BVH` up to the given level.

You can see an example of the application in Figure 5.



Foreground
Background
Wireframe
Normals ✓
Auto Scale
Auto Center
Show Axes
BVH Level 10

Pos -1.7 0.5 4 Target 0 0 0 Up 0 1 0 FOV 30

Figure 5: The GUI of the demo application with the OpenGL 3D view on the left and the ray traced image on the right. Crystal Dragon CC BY-NC-ND 4.0 Vlado Turek.