4 Acceleration Data Structures

This exercise focuses on the construction and usage of acceleration data structures.

4.1 Theory (5 Points)

4.1.1 Separating Axis Theorem (2 Points)

Figure 1 shows a triangle and a quad defined by their vertices. Project these objects onto the given direction v in the spirit of the Separating Axis Theorem and determine if a collision can be ruled out using the 1D intervals.

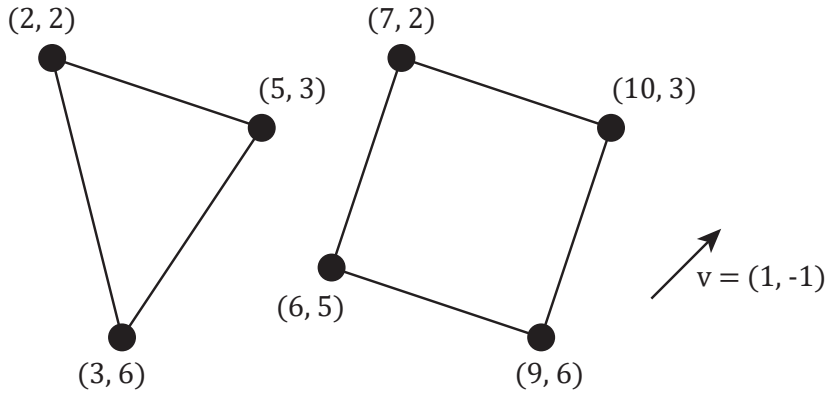

Figure 1: Given objects and projection direction v

4.1.2 Ray Traversal in a Uniform Grid (3 Points)

Given a 3D grid with cell size $w = 0,3$ and a ray with starting point $\underline{p} = (1, 0.2, 0.5)^T$ and direction $\hat{d} = \frac{1}{\sqrt{0.3^2+0.5^2+1}}(0.3, 0.5, 1)^T$. The grid lies in the positive octant and cell indices are represented in the format (x index, y index, z index), starting with zero at the origin.

- Calculate the value of the ray parameter t for the first two intersections of the ray with the grid. (**1 Point**).
- State the indices of the first four cells that the ray traverses (**1 Point**).
- Visually, the result is similar to a rasterized line produced by, e.g., Bresenham's algorithm. However, there are differences. Explain these differences using an example (**1 Point**).

4.2 Practical Part (10 Points)

The practical part focuses on two acceleration data structures: the AABB tree and the hash grid. Both data structures can be used to access the vertices, edges, or faces of a mesh. You will use the AABB tree to realize a nearest neighbor search. The grid will help you identify cells that are intersected by the primitives of a mesh. Furthermore, you will implement an incremental ray traversal.

4.2.1 AABB Tree (4 Points)

The AABB tree is a bounding volume tree that used axis-aligned bounding boxes (AABB) as bounding volumes. First, the method `Insert` adds all elements to an STL vector. Then, the method `Complete` starts the actual construction of the tree. There are two types of nodes: `AABBSplitNode`s (inner nodes) and `AABBLeafNode`s. Both nodes are derived from the super class `AABBNode`. Each node comprises a bounding volume `bounds` and implements the abstract method `bool IsLeaf()`. Split nodes additionally contain two pointers to the left and right child nodes. A leaf node contains two iterators `primitivesBegin` and `primitivesEnd` that mark the beginning and the end of a region in the entire primitive list. All primitives associated to a leaf node lie in the interval [begin,end). The construction is performed top-down, starting with the construction of a bounding volume that contains all primitives. The set of primitives is then sorted in the direction of largest extent and split into two equally sized subsets. Those subsets are processed recursively until a maximum depth `maxDepth` or number of primitives `minSize` is reached. In this task, you will use this tree to find the closest point on any of the inserted primitives.

The primitives `Point` and `Triangle` already contain an implementation of `ClosestPoint(p)`. When you start the program, you can set a query point via the GUI (Query X, Y, Z). The program will then draw a connection between query point and closest point on the primitive. The selection box *Use Vertices* lets you choose the primitives to use. In the current state of the program, queries for vertices and facets should work, whereas queries for edges are not implemented yet. It is recommended to compile this task in release mode since tree construction can take very long in debug builds.

- Implement `ClosestPoint` for the `LineSegment`. If your implementation is correct, the closest point query should also work in the *Use Edges* mode. (**1 Point**).
- Implement an efficient closest primitive search that uses the AABB tree in `ClosestPrimitive` in class `AABBTree`. (**3 Points**). Hint: This method is already implemented with a slower linear search to allow testing of the previous task. Hint 2: Many subproblems are already solved in the C++ Standard Template Library. E.g., you can use the `std::priority_queue`. The console prints how much time the individual queries took. Compare the times between the linear search and the AABB tree-based query.

4.2.2 Hash Grid (6 Points)

A hash grid is essentially a hash map that maps a 3D integer grid coordinate (key) to the corresponding content of the grid cell (value). In our case, the content of a cell is a STL vector of all primitives that overlap the cell. A hash grid stores only cells that are occupied (in contrast to a dense grid). We use the STL hash map implementation `std::unordered_map`. The hash function is $h((i,j,k)^T) = p_1 * i + p_2 * j + p_3 * k$ with the Mersenne primes $p_1 = 131071$, $p_2 = 524287$ und $p_3 = 8191$. To add a primitive to the grid, we

use the method `Insert{Primitive& p}`, which uses two utility function `ComputeBounds()` and `Overlaps(const Box& b)`. The former calculates an axis-oriented bounding box around the primitive, which is used to enumerate all grid cells that overlap the boundig volume. Then, the latter utility function is used to test of the primitive actually overlaps the cell. The primitive is then added to the respective cells. Although there are more efficient ways to do this, this simple approach is usually fast enough. Make yourself familiar with the interface of `Box`. It contains a couple of helpful methods that can be used for your tasks.

As soon as you load a model, a hash grid will be constructed from the three primitive types. The console outputs how many cells are occupied. In the initial state, the hash grid for triangles should not contain any cells since the required functions are not implemented.

- Implement `ComputeBounds()` for a triangle in class `Triangle` (**1 Point**). If you are successful, the hash grid for triangles should now also contain cells. It should have at least as many cells as the other two grids. You can visualize the cells with the checkbox *Render Non-Empty Grid Cells*.
- Implement the overlap test for a triangle and an axis-aligned box in `Overlaps(const Box& b)` of class `Triangle` (**3 Points**). The number of cells in the triangle grid should then decrease slightly. Use the Separating Axis Theorem. You can find more information in the following paper: `http://fileadmin.cs.lth.se/cs/Personal/Tomas_Akenine-Moller/pubs/tribox.pdf`).
- Implement the incremental ray traversal algorithm through a grid in class `GridTraverser`. This class is essentially an interator. The dereferencing operator `operator*()` returns the 3D index of the current cell. The post-increment operator `operator++(int)` continues to the next on the ray. Additionally, the `Init()` method is necessary to initialize the internal state of the iterator, including setting the first cell. Implement the `Init` method and the increment operator. The class stores the origin and the direction of the ray in its attributes `orig` and `dir` as well as the extent of a grid cell in x, y, and z direction in its attribute `cellExtents`. You can add more attributes as you see fit. You can test your implementation via the GUI. For this, acrivate the option *Render Ray* and set ray origin, direction, and the number of visualized steps. (**2 Points**).

4.2.3 Bonus Tasks (max. +5 Points)

- Replace the grid insert method for lines and triangles with a more efficient scan line algorithm. (**3 Points**)
- Add support for convex polygons for the AABB tree and the hash grid to avoid triangulation of the mesh (**3 Points**).
- Implement a k-nearest neighbor query for the AABB tree (**2 Points**).
- Implement a frustum query for the AABB tree that returns primitives that overlap a given frustum. Represent the frustum as an OpenGL ViewProjection matrix. (**3 Points**)
- Implement a bounding volume tree with spheres instead of AABBs (**5 Points**).