Fully parallel BVH construction on the GPU

Daniel Opitz

Abstract

In this assignment it will be discussed, how a bounding volume hierarchy (BVH) can be efficiently created. To build the BVH efficiently on a OpenCL capable GPU, a highly parallel algorithm for it's construction will be presented.

1. INTRODUCTION

A bounding volume hierarchy is a tree like data structure for geometric objects. Each of the geometric objects is represented as a leaf node in the tree. Each node in the BVH is a bonding volume, conservatively enclosing all the bounding volumes of it's child nodes, or the geometry stored in the leaf node.

A bounding volume hierarchy can be used in various multiple ways, reaching from acceleration of ray tracing, speeding up collision detection for physics simulation, to efficiently selecting light sources in rendering techniques like clustered and tiled deferred shading. This wide range of use cases however, does not come without any drawbacks. If a BVH, e.g. for a set of objects for fast collision detection, is computed it is very difficult to keep the BVH consistent, if these objects are not static, but move around freely. Tracking objects and applying transformations to an existing BVH is highly impractical and consecutive updates may lead to a very unbalanced tree.

A different approach is to create the BVH from scratch. In this assignment an efficient algorithm for a BVH construction, that runs solely on the GPU is implemented. In the following algorithm it is supposed, that the actual bounding volumes for each object already exist, and need not be computed. For the bounding volume type AABBs have been chosen. The algorithm can be split into 3 larger tasks that need to be completed to obtain a BVH:

1. Compute the Morton codes for each object
2. Sort the Morton codes
3. Find inner nodes of the BVH

In the following it will be detailed, why each step has to be performed and how it can be efficiently implemented.

2. Computing Morton Codes

In advance to step 4 (Computing Inner Nodes it is important to notice, that the inner nodes shall be represented a range of indices to leaf nodes. This is reason, why leaf nodes, that belong to the same parent node must reside consecutively in memory. Therefore we need to establish an order of those leaf nodes, so that bounding volumes of inner nodes cover as less free space as possible. One way to attain such an ordering are sorting the leaf nodes using Morton codes.

Morton codes are a way to obtain a z-order curve in 3 dimensional space. An example of the z-order curve is given in figure 2. The figure illustrates, that we can use this z-order curve to arrange the leaf node objects in a one dimensional array. As another beneficial property, we can accurately divide the space into equal portions only by observing the Morton codes. This is because on the first half of the z-order curve the MSG of the Morton codes there is 0, on the second half it is 1. Due to the recursive nature of the z-order curve the process is repeated for less significant bits, while alternating the axis, in which the space is split. This is nice, because we can find the split point of e.g. the root node just by inspecting the MSB if the Morton codes. In the following it will be explained how the Morton codes can be computed from the center points of the AABBs of out objects.

2.1. Finding the Root Node AABB

To calculate the Morton Code for each leaf node AABB, we have to get a representation of the AABB's center points where all coordinates are in range of `[0, 1]`. Similar to barycentric coordinates, we can use an AABB instead of the usual simplex. This AABB is exactly the AABB of the root node in the BVH. To compute this AABB a simple reduction as covered by Assignment 2 over all leaf node AABBs, that reduces the minimum and maximum value for each axis, is a

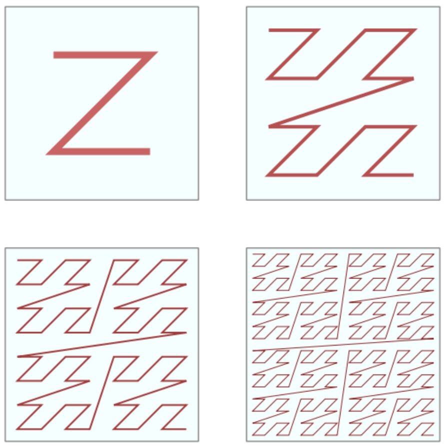

Figure 1. 4 Iterations of the z-curve in 2D (from wikipedia)

simple and efficient way to reach this goal.

2.2. Calculating the Morton code

At this point all leaf node AABB center points can be represented as 3D point located within the unit cube. To compute an actual single unsigned integral value from them, each coordinate will be scaled to range `[0, 1023]`, cast into an `unsigned int` and finally padded with zeroes in between consecutive bits. Figure 2.2 illustrates this process.

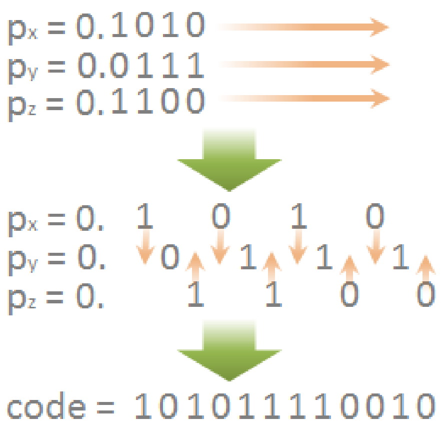

Figure 2. Scheme for swizzeling the position into a Morton code (from `https://devblogs.nvidia.com/parallelforall/thinking-parallel-part-iii-tree-construction-gpu/`)

3. Sorting Morton Codes

To utilize the Morton codes we have to sort them as well. There are several parallel sort algorithms for GPUs, the one implemented for this assignment is a flavor of radix sort. In this case we will iterate over 32 bit starting from the LSB and calculate prefix sums to reorder the sorting keys (Morton codes) ascending. One iteration for bit at position `k` can be outlined as:

1. Build two arrays of flags A and B, where A is 1 for each key that has it's bit at `k` set and else 0. B is the inversion of A.
2. Compute the prefix sums over A and B. Add the maximum value of A to each element in B.
3. Reorder the keys using A and B as new indices (depending if the key's `k` bit is set or not).

Sorting the Morton codes this way, is fast using only programming primitives already discussed in Assignment 2. Although there are more efficient algorithms to sort on GPUs, it is deemed that this is sufficient.

4. Computing Inner Nodes

As it was already teased in section Computing Morton Codes, the inner nodes can now be calculated by determining a splitting point in the array of Morton codes. In a naive algorithm this splitting point can be determined starting with the root. This will yield 2 ranges for the two child nodes of the root. The two child nodes now can process their ranges the same way resulting in 2 more split points and all in all 4 ranges of leaf nodes. This process can be continued until all ranges are one element long. This way the constructed BVH is correct, however the GPU utilization is at the very low end. This is because in the first round only one thread is needed to compute the split point and ranges. On a modern GPU with more than 1K possibly parallel executed threads the occupancy is at meagerly 0.001%. In the next round 2 threads are launched doubling the occupancy as well, but an acceptable occupancy is reached in round 15 with still very low 32%. However at this point the BVH creation is already completed, given that there are around 30K leaf node objects.

Another approach is to compute the split points of each inner node in parallel. This is possible, because in a binary tree there are exactly `N-1` inner nodes, if there are `N` leaf nodes and the split points can be determined without knowing the range before hand. Using those properties we enumerate nodes as shown in figure 4. From here for each node it's range has to be determined.

Therefore we have to find out if a node resides on the upper bound or the lower bound of it's range, by comparing neighboring Morton codes. When the direction is set, the other end of the range can be determined with two binary searches, searching for the farthest apart element, but where the length of Morton code's common prefix is only one. If the range is known, the split point is computed using another binary search as one would have implemented it in the naive version.

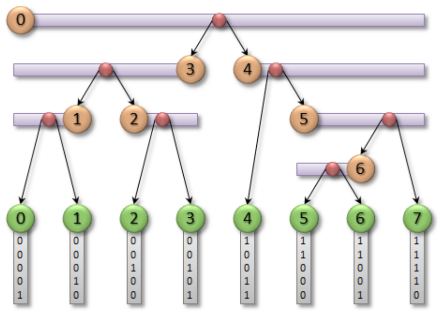

Figure 3. Enumeration of inner nodes and highlighting their range (from `https://devblogs.nvidia.com/parallelforall/thinking-parallel-part-iii-tree-construction-gpu/`)

Note, that in the parallel approach each thread has to do more work (3 binary searches instead of one) but the benefit is, we can compute every node's split point and range in parallel and do not have to wait for all nodes further up in the hierarchy to finish.